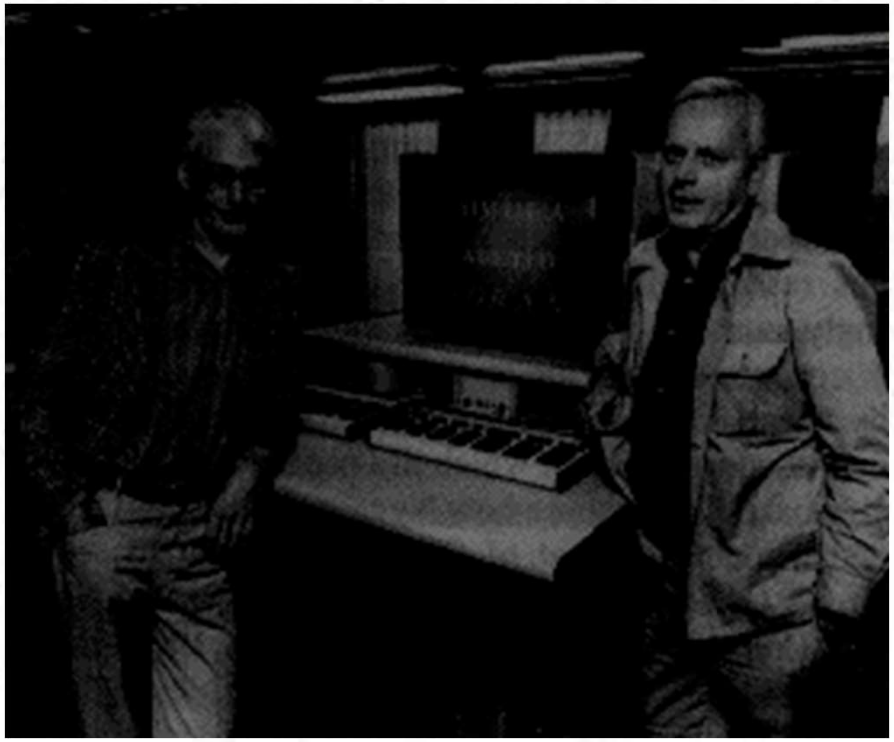
Projecte de Programació

Conceptos de Orientación a Objetos en Java

Conceptos Orientación a Objetos

Historia: El año 1967 los noruegos Ole-Johan Dahl y Kristen Nygaard inventan el lenguaje **Simula-67**, especialmente diseñado para hacer simulaciones de sistemas físicos complejos. Por primera vez aparecen los conceptos de *clase*, *subclase*, *instancias de clases*



Conceptos Orientación a Objetos

Historia: Durante los años 70, en el centro de investigación Xerox PARC se desarrolla un proyecto de entorno amigable que aportó, entre otros:

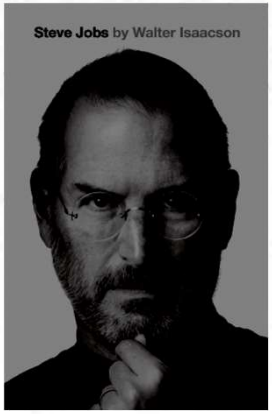
- El ratón
- El primer entorno gráfico con iconos
- El concepto moderno de lenguaje orientado a objetos nace con **Smalltalk.** Smalltalk es el primer LP orientado a objetos como tal y es la influencia principal en Objective-C, Ruby, Java, etc.

Hoy en día cualquier lenguaje de programación incorpora orientación a objetos, pero la mayoría mantienen lo que ya había -> LPs OO puros: Smalltalk, Java

Metodología OO extendida a todos los ámbitos de la informática

Conceptos Orientación a Objetos

The Smalltalk demonstration showed three amazing features. One was how computers could be networked; the second was how object-oriented programming worked. But Jobs and his team paid little attention to these attributes because they were so amazed by the third feature, the graphical interface that was made possible by a bitmapped screen. "It was like a veil being lifted from my eyes," Jobs recalled. "I could see what the future of computing was destined to be."



Steve Jobs (p. 137)
Walter Isaacson
Simon & Schuster, 2011

Conceptos Orientación a Objetos

La Orientación a Objetos entiende la computación como *resultado de la interacción entre objetos que son capaces de comunicarse*

Ventajas:

- Reutilización y reciclaje de código: Gran número de librerías, facilidad para construirlas
- Encapsulamiento: En diversos sentidos, no hace falta conocer la implementación para utilizar objetos y tampoco se puede acceder al estado de los objetos
- Escala muy bien
- Facilidad de mantenimiento

La OO no es algo específico de un LP, sino una forma de entender el desarrollo del software

Conceptos Orientación a Objetos

Ejemplo*

Un individuo A quiere enviar flores a otro individuo B, que vive en otra ciudad. Debido a la distancia, A no puede coger las flores y llevárselas a B.
Para conseguirlo, A se acerca a la floristería de C, le dice cuántas y qué tipo de flores quiere, y la dirección a la que hay que enviarlas. Si todo funciona bien, A puede estar seguro de que las flores llegarán a B.
Hay 2 tipos de floristerías: las que cultivan sus propias flores y las que no, en cuyo caso tienen que contactar con los que las cultivan y pedírselas.

* **Del libro "An Introduction to Object-Oriented Programming", Budd & Pearson, 2008**

Conceptos Orientación a Objetos

Ejemplo

Implementación version TADs

```
main () {
  A, B: Individuo;
  x: Flores;
  C: Floristeria:
  . . . obtener/informar A,B,x,C
  enviarFlores (A,B,x,C);
}

void enviarFlores (A,B: Individuo, x: Flores, C:Floristeria){
  d: Direccion; n: Nombre;
  d := obtenerDireccion(B);
  n := obtenerNombre(A);
  if C.tipo == "cultivadora"
    ordenarAFloristeriaCultivadoraEnviarFlores(C,n,x,d);
  else
    ordenarAFloristeriaNOCultivadoraEnviarFlores(C,n,x,d);
}
```

Conceptos Orientación a Objetos

Ejemplo

Solución "humana":

C llamará a una floristería D de la ciudad donde vive B para encargarle el pedido.
Y D probablemente pedirá a un empleado E que las lleve físicamente.
Los que cultivan las flores, a su vez, tendrán un equipo de jardineros.
En cualquier caso, a A le trae sin cuidado lo que haga C.

Abstrayendo:

1. El mecanismo para resolver el problema ha sido encontrar un agente/objeto apropiado (C) y pasarle un **mensaje** conteniendo la petición. Es responsabilidad de C satisfacerla. Para ello, C ha seguido algún **método**. A puede no saber los detalles de este método: están **ocultos**

2. La solución al problema ha requerido de la participación de muchos individuos. En general, un programa OO se estructura como una comunidad de agentes (los **objetos**), que interactúan entre ellos. Cada objeto tiene un **rol**, **proporciona** un **servicio** o **ejecuta** una **acción** que es usada por otros miembros de la comunidad

Conceptos Orientación a Objetos

Ejemplo

Implementación version Orientación a Objetos (1/2)

```
main () {
  A, B: Individuo;
  x: Flores;
  C: Floristeria:
  . . . obtener/informar A,B,x,C
  A.enviarFlores (B,x,C);
}

clase Individuo {
 void enviarFlores (B: Individuo, x: Flores, C:Floristeria){
   d: Direccion; n: Nombre;
   d := B.obtenerDireccion();
   n := YOMISMO.obtenerNombre();
   C.enviarFlores(n,x,d);
 }
 …
}
```

Conceptos Orientación a Objetos

Ejemplo

Implementación version Orientación a Objetos (2/2)

```
clase Floristeria {
 … características generales de TODAS las floristerías
}

clase FloristeriaCultivadora es una subclase de Floristeria {
 void enviarFlores(n: Nombre, x: Flores, d: Direccion) {
  …el equivalente a ordenarAFloristeriaCultivadoraEnviarFlores
 }
}

clase FloristeriaNOCultivadora es una subclase de Floristeria {
 void enviarFlores(n: Nombre, x: Flores, d: Direccion) {
  …el equivalente a ordenarAFloristeriaNOCultivadoraEnviarFlores
 }
}
```

Conceptos Orientación a Objetos

Mensajes i métodos:

En un programa OO la acción comienza con el *envío de un mensaje* (invocación de método) de un objeto a otro objeto

Diferencias entre paso de mensajes y llamadas a procedimientos:

- Siempre hay un *receptor* del mensaje

- Los mensajes pueden tener diferentes interpretaciones en función del receptor... → *Polimorfismo*

...y a veces éste no se conoce hasta que se ejecuta el programa → *Dynamic binding/dispatch*

Conceptos Orientación a Objetos

Resumiendo:

- Todo es un objeto

- Cada objeto es instancia de una clase

- Una clase define un modelo para sus instancias. Define los atributos (el estado) y el comportamiento y responsabilidad de los objetos (operaciones sobre los datos)

- La computación se realiza mediante objetos comunicándose entre sí. Se envían mensajes que el receptor se encargará de interpretar al recibirlos, en el momento de la ejecución

"Ask not what you can do **to** your data structures, ask what your data structures can do **for** you"

"Programar en OO es programar ensamblando componentes"

Conceptos Orientación a Objetos

Y qué sacamos de usar objetos?

1.- **Encapsulamiento**:


énfasis en los procesos


énfasis en crear módulos (objetos) con estado y comportamiento, con una interficie clara y definida y ocultación de la implement.

Ejemplo: Utilizar objetos *iteradores* para controlar bucles (en lugar de hacerlo con enteros)

Ejemplo: No sólo utilizar objetos como estructuras de datos, también como *generadores* de estas estructuras o como *vistas* de estas estructuras

Conceptos Orientación a Objetos

Y qué sacamos de usar objetos?

2.- **Simulación de la realidad**:

Todo programa es una formalización de un fragmento del mundo interpretado de acuerdo a las herramientas (p.ej. lenguaje de programación) que tiene el programador. En el caso del OOP los modelos que hacemos del mundo (los programas) són más similares, o se ajustan mejor, a cómo pensamos el mundo (al menos en occidente)*

** **Classes vs. Prototypes, Some Philosophical and Historical Observations,** A.Taivalsaari, cap. 2 en **Prototype-Based Programming** J.Noble, A.Taivalsaari & I.Moore (eds.) Springer 1999*

Conceptos Orientación a Objetos

Y qué sacamos de usar objetos?

3.- **Reutilización**:

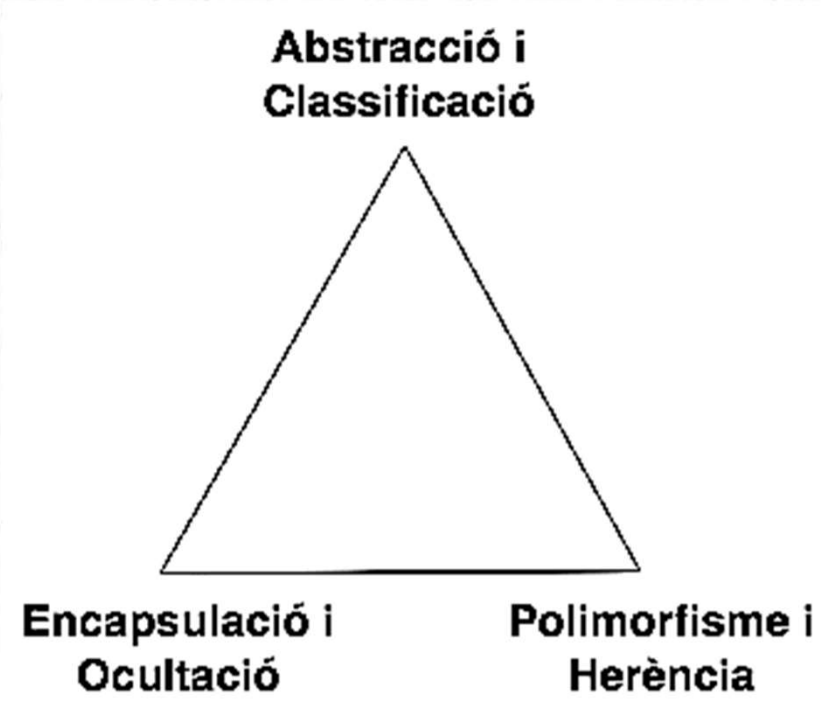
1) Una aplicación OO bien diseñada ha de evitar al máximo "reinventar la rueda", es decir, debería utilizar tanto como sea posible las librerías que tenga disponibles

2) A la hora de diseñar nuestras clases deberíamos hacerlas tan generales como sea posible para poder reutilizarlas más adelante -> nuestras clases deberían contener funciones generales de la clase, NO de nuestra aplicación

Conceptos Orientación a Objetos

Pilares básicos de la OOP

Los mecanismos que los lenguajes orientados a objetos ponen a nuestra disposición para explotar las ventajas de la OOP se pueden representar en un triángulo



Java

Java in a Nutshell, 8th Edition by David Flanagan
Released February 2023 Publisher(s): O'Reilly Media, Inc.
https://www.oreilly.com/library/view/java-in-a/9781098130992/

Java

Historia:
Surge en 1991 como proyecto de Sun Microsystems para diseñar un lenguaje para la programación de electrodomésticos:
"Write Once, Run Everywhere"

Se desarrolló un código "neutro" que no dependía del tipo de electrodoméstico y se ejecutaba sobre una máquina virtual, la Java Virtual Machine (JVM) -> la JVM interpretaba el código neutro convirtiéndolo a código particular de la CPU usada
…

En 1995 aparece como lenguaje de programación para ordenadores: se incorpora un intérprete de Java en la versión 2.0 del Netscape Navigator, que produce un revolución en Internet

Java

Entorno Básico

En PROP: Java SE 21 (jdk-21.*)

1) Compilación: *.java (código fuente)

*.class (*bytecodes* o código neutro)

```
javac nombreFichero.java -> nombreFichero.class
```

La compilación es incremental (basta con compilar la clase que contiene el programa principal). No obstante, siempre se puede hacer:

```
javac *.java
```

Java

Entorno Básico

2) Ejecución: el código neutro se ejecuta en un intérprete, la JVM

```
java [-cp ClassPath] nombreFichero [parámetros]
```

Flag -cp: dónde buscar los .class:
- Si la var de entorno CLASSPATH está asignada, las clases han de estar en el CLASSPATH o declaradas con el –cp
- Si la var de entorno CLASSPATH no está asignada, las clases han de estar en el directorio actual de trabajo o en el -cp

Se pueden montar bibliotecas de clases en ficheros .jar:

```
java -jar nombreFicheroJar.jar
java -cp .;C:\junit-4.5.jar nombreFichero
```

Java

Particularidades PROP

- Java es portable: se puede trabajar en Windows, Linux o Mac
- Se puede usar cualquier IDE, pero en las entregas el código se ha de poder recompilar y ejecutar desde terminal
- De cara a las entregas, los .java son el código fuente y los .class son los ejecutables. Han de estar todos presentes aunque también se entreguen uno o varios .jar ejecutables directamente
- De cara a las entregas, se tiene que replicar la estructura de directorios y packages para que se pueda recompilar/ejecutar directamente (sin tener que reorganizar los .java/.class respectivamente)
- De cara a las entregas, si se usa algún CLASSPATH/opciones de compilación o ejecución, se han de incluir en un fichero readme o en un .bat, .sh o makefile
  Ejemplo: para evitar java heap space error -> `java -Xmx4G …`
  (ver `java -help`)

Java

Estructura General de una Clase

```
import…
[public] class MiClase {

  atributos (formato: tipo nombre)

  métodos

  [opcionalmente]
  private class MiClasePrivada {
    …
  }
}
```

- Al compilar, se generaría:
  MiClase.class
  MiClase$MiClasePrivada.class

Java

Estructura General de una Clase

```
import…
[public] class Persona {

  int DNI;
  String nombre;
  Date fechaNacimiento;

  public int edad() {
  …
  }

}
```

* Sugerencia: ver guía de estilo de programación (*Java Programming Style Guidelines)* en la web de PROP

Java

Estructura General de una Clase

Para poder ejecutar directamente un .class, éste ha de contener el método:

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

```
[public] class HelloWorld {
  public static void main (String[] args) {
    System.out.println("Hello world, my name is " + args[0]);
  }
}
```

- `javac HelloWorld.java`

- `java HelloWorld Alicia`

```
Hello World, my name is Alicia
```

Java

Estructura General de una clase

- En un fichero se pueden definir varias clases, pero como máximo una pública

- Si un fichero contiene una clase pública, se ha de llamar como esta clase (con extension .java)

- Lo habitual es escribir una sola clase por fichero (excepción: clases privadas)

- Las clases que no sean públicas ni privadas tendrán visibilidad package

Java

Cláusula *package*

- Permite crear paquetes (agrupaciones de clases)
- Cuando una clase pertenece a un package nomP1, su fichero java empieza:

```
package nomP1;
```

- Si hubiera subpaquetes, se concatenan los nombres:

```
package nomP1.nomP2;
```

- La estructura de paquetes se ha de corresponder con la de directorios y subdirectorios donde se almacenan los .java y .class de las clases
- Para usar las clases de un package se usa el *import*, que es siempre **relativo al directorio de compilación**

Java

Cláusula *package*

Ejemplo:

```
import Package1.*;
import Package1.Package2.*;

public class Main {
  public static void main (String[] args) {
    System.out.println("Hello, world");
    ClassPackage1 cp1 = new ClassPackage1();
    ClassPackage1Package2 cp1cp2 = new ClassPackage1Package2();
    …
  }
}
```

Ojo! Se ha de compilar/ejecutar desde el directorio de trabajo:

```
> javac Package1/Package2/ClassPackage1Package2.java
> java Package1.ClassPackage1
  (suponiendo que fuera ejecutable)
```

Java

Estructura típica para PROP (1/4):

Un *package* por capa (y *subpackages* en cada capa si se quiere)

```
main.java
 import Presentación.*;
 public static void main (String[] args) {
   CtrlPresentacion = new CtrlPresentacion();
   …
 }
```

Java

Estructura típica para PROP(2/4):

```
Presentacion/

 CtrlPresentacion.java
    package Presentacion;
    import Dominio.*;
    public class CtrlPresentacion {
      public CtrlPresentacion() {
        CtrlDominio CD = new CtrlDominio();
        …
      }
     …
    }

  VistaPresentacion.java
    package Presentacion;
    public class VistaPresentacion {
       …
    }
```

Java

Estructura típica para PROP (3/4):

```
Dominio/

  CtrlDominio.java
    package Dominio;
    import GestionDatos.*;
    public class CtrlDominio {
      …
    }

  Persona.java
    package Dominio;
    public class Persona {
      …
    }
```

Java

Estructura típica para PROP (4/4):

```
GestionDatos/

 CtrlGD.java
    package GestionDatos;
    import …   // librerías de implementación de persistencia
    public class CtrlGD {
      …
    }
```

Java

Tipos

- Simples o primitivos: *boolean*, *char*, enteros (*byte*, *short*, *int*, *long*), reales (*float*, *double*)
  - No se tienen que instanciar
  - Se pasan por valor
- Complejos (objetos):
  - Se implementan como una referencia al lugar de memoria donde está su información
  - Se tienen que instanciar antes de usarlos:

```
Persona p;
p = new Persona();
```

  - Se pasan por valor – Ojo! Lo que se pasa es el valor de la referencia:

`Persona p1 = p` -> *aliasing* (un mismo objeto puede tener más de un nombre), fenómeno a evitar (o vigilar!)

Java

Persona
DNI: int
nombre: String
fechaNac: Date

```
Persona p;                  p ---> referencia vacía (null)

p = new Persona();          p --->  DNI       --> 0
                                    nombre    --> referencia vacía
                                    fechaNac  --> referencia vacía

Persona p1 = p;             p1--->
```

Este objeto tiene dos nombres -> cualquier cambio que afecte a uno afectará también al otro:

```
p.setNombre("pepe");
System.out.println(p1.getNombre); -> "pepe"
```

Java

Tipos

- Paso de parámetros:
  - Los tipos simples ya son valores
  - Los tipos complejos a efectos prácticos es como si se pasasen por referencia -> las modificaciones que se le hagan dentro del método tienen efecto al volver:

```
void metodo1 (Persona p) {
        …
        p.setNombre("marc");
        …
}
```

```
Persona p1 = new Persona("pepe");
metodo1(p1);
System.out.println(p1.getNombre); -> "marc"
```

Java

Tipos

- Inicialización implícita:
  - Numéricos: 0/0.0
  - boolean: false
  - char: carácter nulo ('\u0000')
  - Objetos: null (referencia vacía)

No obstante, metodológicamente y por claridad del código, es mejor inicializar explícitamente…

Java

Tipos

- Clases *wrapper*: versiones complejas de los tipos simples (Boolean, Character, Byte, Short, Integer, Long, Float, Double)

```
Integer i1 = Integer.valueOf(5);
int i2 = i1.intValue();
```

  - Necesarias p.ej. para usar estos tipos como parámetros en clases genéricas
  - Se han de instanciar y se pasan "por referencia"
  - El compilador hace conversiones implícitas entre los tipos simples y sus wrappers:

```
i1++;
Integer i3 = i1 - 3;
void metodo1(Integer param1) -> metodo1(4);
```

Java

Tipos

- Clases *wrapper*:
  - Son **inmutables**: si se pasan como parámetro a un método, no cambian su valor aunque se hayan modificado dentro del método

```
void metodo1 (Integer i) {
        …
        i = 111;
        …
 }
```

```
Integer i1 = Integer.valueOf(3);
metodo1(i1);
System.out.println(i1); -> 3
```

Java

Constructoras y Destructoras

- Constructoras:
  - Unifican el nombre del método y el tipo de retorno
  - Tantas como se quiera (con distintos parámetros)

```
Persona () {
  nombre = null;
  dni = 0;
  fechaNacimiento = null;
}

Persona (String nom) {
  nombre = nom;
  dni = 0;
  fechaNacimiento = null;
}

Persona (String nom, Date fec, int dni1) {
  nombre = nom;
  dni = dni1;
  fechaNacimiento = fec;
}
```

Java

Constructoras y Destructoras

- Constructoras:
  - Se instancia con la cláusula "new"

```
Persona p1 = new Persona();
Persona p2 = new Persona("nombre");
```

- Destructoras:
  - En Java NO hay: el manejo de memoria es automático. Cada cierto tiempo se ejecuta el **Garbage Collector**, que detecta los objetos hacia los que no hay referencias y los elimina

Java

Jerarquía de Clases de Java

- Todas las clases complejas heredan de otra:
  - Explícitamente (herencia, cláusula *extends*)
  - Por defecto, heredando de la clase *Object* (clase top de la jerarquía de clases de Java)

  En definitiva, TODA clase hereda de *Object* (directa o indirectamente)

Conviene conocer los métodos de la clase *Object* (aunque algunas de las clases ya tienen estos métodos **redefinidos**)

Java

Clase *Object* (1/3)

protected Object clone()
creates and returns a copy of this object

- La implementación de *Object* hace una *shallow copy:*

Crea una instancia (referencia) diferente, pero con el mismo contenido -> puede crear referencias compartidas (*aliasing*)

```
Persona[] p1 = new Persona[100]();
Persona p2[] = p1.clone();  -- cada elemento de p1 apunta
                               a la misma referencia de
                               persona que cada elemento
                               de p2!
```

Si no queremos introducir referencias compartidas, habría que redefinir el método para implementar una *deep copy* (que copie cada contenido del objeto recursivamente de forma "*deep*")

Java

Clase *Object* (2/3)

3 niveles de comparación, de más a menos estricto:
- Misma referencia (operador ==)
- *Shallow*: distinta referencia pero los contenidos son referencias compartidas
- *Deep*: no existen referencias compartidas, igualdad de valores

boolean equals(Object)
indicates whether some other object is equal to this one

- La implementación de *Object* es a nivel de referencia (==)

```
Persona[] p1 = new Persona[100]();
Persona p2[] = p1.clone();

p1.equals(p2) -> false
```

Java

Clase *Object* (3/3)

Class getClass()
returns the runtime class of this object

- Útil en casos de polimorfismo
- "Equivalente" al operador booleano *instanceof*:

```
p1 instanceof Persona  <->  p1.getClass() == Persona
```

Otros métodos de *Object*: *toString(), hashCode(), …*

Conceptos Orientación a Objetos en Java

Encapsulamiento y Ocultación de la Información

Encapsulamiento: Las clases consisten, a grandes rasgos, en la definición de los datos + los métodos (funciones) que manipulan estos datos. Cada objeto instancia de esta clase contendrá su copia particular de estos datos, que sólo se podrán manipular a partir de les funciones diseñadas a tal efecto (la *interfície*).

Ocultación: La implementación particular de estas funciones queda oculta, ya que no es necesario conocerla para usarlas (es decir, para enviar mensajes al objeto)

Conceptos Orientación a Objetos en Java

Encapsulamiento y Ocultación de la Información

Ejemplo (1/2): Podemos definir la clase **`Ejemplo`**

```
public class Ejemplo {
        public int estado;
        // queremos que estado siempre sea positivo

        public Ejemplo (int estado) {
          this.estado = (estado < 0) ? 0 : estado;
        }
        . . .
}
```

si instanciamos esta clase:

```
Ejemplo e = new Ejemplo(5);
```

podemos hacer **`e.estado = -42`** y dejar **`estado`** en un valor no deseado

Conceptos Orientación a Objetos en Java

Encapsulamiento y Ocultación de la Información

Ejemplo (2/2): Para evitarlo, redefinimos **`Ejemplo`**

```
public class Ejemplo {
        private int estado;
        // queremos que estado siempre sea positivo

        public Ejemplo (int estado) {
          this.estado = (estado < 0) ? 0 : estado;
        }

        public int getEstado() { return this.estado };
        public boolean setEstado (int x) {
               if (x >= 0){
                this.estado = x;
                return true;
               }
               return false;
        }
        . . .
}
```

Y así controlamos el acceso al estado

Conceptos Orientación a Objetos en Java

Encapsulamiento y Ocultación de la Información

Ejemplo (1/3): La clase **`Fecha`** definiría un modelo con 3 atributos (los datos **`dia`**, **`mes`** y **`año`**) y diversos métodos, por ejemplo:

```
incrementar (int dias)
boolean comparar (Fecha f)*
            ...
```

Dados los objetos **`f1`** y **`f2`** de clase **`Fecha`** podríamos enviarles los mensajes:

```
f1.incrementar(28)
if (f1.comparar(f2)) { . . . }
```

*Alternativamente, se podría redefinir la operación *equals* en la clase **`Fecha`**

Conceptos Orientación a Objetos en Java

Encapsulamiento y Ocultación de la Información

Ejemplo (2/3): La clase **`Fecha`** también podría definir métodos para acceder a los atributos:

```
int getDia(), int getMes(), int getAño()
void/boolean setDia(int d),...
```

los atributos no son accesibles *directamente* desde el exterior del objeto, sólo mediante métodos definidos en la interfície, de una manera controlada (no se podrá actualizar la fecha si su valor es incorrecto)

La clase ofrece una ESPECIFICACIÓN CLARA Y PRECISA de sus funcionalidades (métodos), que será lo que se ofrezca a cualquier objeto "cliente" de esa clase

Conceptos Orientación a Objetos en Java

Encapsulamiento y Ocultación de la Información

Ejemplo (3/3): Ahora podríamos cambiar la implementación de la clase **`Fecha`** y, si conservamos la interfície, ningún usuario de los objetos instancia de esta clase tendría que ser capaz de diferenciar esta clase de la anterior. Por ej, podríamos tener sólo un atributo, **`diaJuliano`**, y mantener los métodos:

```
incrementar (int dias)
boolean comparar (Fecha d)
                ...
int getDia(),int getMes(),int getAño()
void/boolean setDia(int d),...
```

Eventualmente, podría pasar incluso que aún no tuviéramos una implementación concreta de la clase cuando estamos diseñando las clases que la usan

Java

Visibilidad

- La clase actúa como "unidad de visibilidad":
  - Por defecto es visible en su *package*
  - *private:* visible solo dentro de la clase en que se define
  - *public*: visible desde cualquier otra clase (usando *import* si es necesario)
- Un atributo o método de una clase:
  - Es visible en toda la clase a la que pertenece
  - Desde fuera de la clase:
    - Por defecto visible en todas las clases de su *package*
    - *private*: visible SOLO en la clase
    - *public*: acceso sin restricciones
    - *protected*: visible desde clases del mismo package y desde sus descendientes

Java

Visibilidad

- En forma de tabla:

Visibilidad desde...	*public*	*protected*	*private*	default
La propia clase	Sí	Sí	Sí	Sí
Otra clase del propio *package*	Sí	Sí	No	Sí
Otra clase fuera del *package*	Sí	No	No	No
Una subclase del propio *package*	Sí	Sí	No	Sí
Una subclase fuera del propio *package*	Sí	Sí	No	No

Java

Encapsulamiento y Ocultación de la Información

- Por defecto, todos los atributos (los datos, el estado) son *private* (o *protected* si es necesario en caso de herencia)
- Los métodos de la interficie son *public* (eventualmente se podría tener métodos *private*, auxiliares de los *public*)
- Para cada atributo (consultable), tendremos 2 métodos *public*:
  - Consultora

```
public tipoAtributo getAtributo() {
  return Atributo;
}
```

  - Modificadora

```
public boolean setAtributo (TipoAtributo valor) {
   if (filtroAtributo (valor)) {
     Atributo = valor;
     return true;
   }
   return false;
}
```

Conceptos Orientación a Objetos

Herencia

- El origen del concepto de herencia está en la IA (taxonomías: bases de conocimiento estructuradas en *frames*, creando relaciones *is-a* para poder hacer inferencia)

- Una clase *hijo* H hereda de la clase *padre* P sus atributos y métodos. Diremos que H es *subclase* de P, y que P es *superclase* de H. Se hereda TODO, no existe la herencia "selectiva"

- La clase H puede redefinir lo que ha heredado: tipos de los atributos, implementación de los métodos. En Java no se pueden renombrar atributos/métodos

Conceptos Orientación a Objetos

Herencia

Diremos que la clase C_2 **es una subclase de** la clase C_1 si:

-Las propiedades de C_2 son una *extensión* de las de C_1

-En cualquier contexto donde aparecen objetos instancia de la clase C_1 se puede utilizar un objeto instancia de C_2 (el objeto de C_1 es *sustituible* por el objeto de C_2)

La herencia es **transitiva**: si C_3 hereda de C_2 y C_2 hereda de C_1

C_3 hereda de C_1

Conceptos Orientación a Objetos en Java

Herencia

Ejemplo (1/4)

```
public class Point {
  protected int x,y;

  public Point(int xval, int yval) {
    x = xval;   // this.x = xval
    y = yval;   // this.y = yval
  }

  public int getX() {
    return x;  // return this.x
  }

  public boolean setX(int xval) {
    x = xval;
    return true;
  }

  public void dibujar() { … }
    . . .
}
```

this hace referencia al objeto "actual"

Conceptos Orientación a Objetos en Java

Herencia

Ejemplo (2/4)

```
import java.awt.Color;
public class ColorPoint extends Point {
  private Color c;

  public ColorPoint(int xval, int yval, Color cval) {
    super(xval, yval);
    c = cval;   // this.c = cval
  }

  public Color getC() {
    return c;  //return this.c
  }

  public boolean setC(Color cval) {
    c = cval;
    return true;
  }
        . . .
}
```

super hace referencia al objeto del cual *this* es hijo

Conceptos Orientación a Objetos en Java

Herencia

Ejemplo (3/4) : en Java se puede redefinir operaciones (*override*). Basta con definirla de nuevo con el mismo número y tipo de parámetros, y tipo de retorno (si lo hay) *compatible*

```
public class ColorPoint extends Point {
    . . .

    // queremos que sólo se puedan mover los puntos
    // de ciertos colores
    public boolean setX(int xval) {
      if (condFiltro(c)) {
        return super.setX(xval);
      } else {
        return false;
      }

   private boolean condFiltro(Color c) {…}
   . . .
}
```

Conceptos Orientación a Objetos en Java

Herencia

Ejemplo (4/4) : en Java se puede redefinir los tipos de los atributos
super también se puede usar para acceder a atributos o métodos del padre (si no son privados) cuando hay ambigüedad

```
public class OtherPoint extends Point {
  protected float x,y;      // se redefine el tipo

  public float getXNew() {
    return x;  // return this.x
  }

  . . .

  public void Metodo1() {  //necesita acceder al x entero
    . . .
    int posX = super.x; // getX();
                        // super.getX();
  }
}
```

(NO recomendable…)

Java

Cláusulas *static* y *final*

- *Static*: variable o método DE CLASE
  - Existe una sola copia para todos los objetos de la clase
  - Existen antes de instanciar el primer objeto
  - Se pueden acceder con el nombre de la clase:

```
nombreClase.atributoStatic / nombreClase.metodoStatic
```

- *Final*
  - Variable *final*: no se puede modificar (sólo se puede asignar un valor en la constructora, y una única vez)
  - Método *final*: no se puede redefinir en descendientes de la clase (*frozen* de UML)
  - Clase *final*: no se puede heredar de ella

- Para declarar constantes:

```
public static final double PI = 3.14159265358979323846;
```

Conceptos Orientación a Objetos en Java

Herencia

2 tipos:

- **Simple**: Toda clase tiene una y solo una superclase, excepto la classe raíz (*Object* en Java) -> La estructura formada por las clases y sus relaciones de clase es un *árbol*
- **Múltiple**: Cada clase puede tener más de una superclase -> La estructura de herencia de les clases puede ser un *grafo*

La herencia múltiple puede ser fuente de conflictos (*colisiones*)

Java NO permite herencia multiple -> si en nuestro diseño aparece, hay 2 opciones para simularla (de forma parcial):

- Mecanismo de **delegación**
- *Interfaces*

Conceptos Orientación a Objetos en Java

Herencia: mecanismo de Delegación

Si pretendemos simular que una clase hijo hereda de 2 o más clases padre:

1. Se elige uno de los padres como "principal" (la clase más relevante o compleja) y se hereda explícitamente de ella
2. Para cada una del resto de las clases padre, se agrega un atributo privado de ese tipo en la clase hijo, que se instancia en la constructora
3. Se replican todos los métodos accesibles de las clases padre del punto anterior en la clase hijo (de forma que su interface sea la misma que si hubiera heredado directamente de todas las clases padre)

Conceptos Orientación a Objetos en Java

Herencia: mecanismo de Delegación

Ejemplo (1/2): queremos que una clase Profesor herede sus datos personales de la clase Persona y sus datos fiscales de otra clase PersonaFiscal

```
public class Persona {
   private int dni;
   private String nombre;
   . . .
}

public class PersonaFiscal {
   private String nif;
   …
   public String getNif() {
     return this.nif;
   }
   public boolean setNif (String x) { . . . }
   public void calcularTramoFiscal(float param1) {. . .}
}
```

Conceptos Orientación a Objetos en Java

Herencia: mecanismo de Delegación

Ejemplo (2/2): escogemos (p.ej.) Persona como clase más significativa y agregamos PersonaFiscal

```
public class Profesor extends Persona {
   private PersonaFiscal pf;
   public Profesor(…) {
     pf = new PersonaFiscal();
     . . .
   }
   public String getNif() {
     return pf.getNif();
   }
   public boolean setNif (String x) {
     return pf.setNif(x);
   }
   public void calcularTramoFiscal(float param1) {
      pf.calcularTramoFiscal(param1);
   }
          . . .
}
```

Conceptos Orientación a Objetos en Java

Herencia

La herencia ofrece:

- Reusabilidad
- Extensibilidad
- Bajo coste de mantenimiento
- Un método para relacionar las clases de forma semánticamente sensata
- Una forma de compartir código

Programar en OO es "**programar por diferencias**":
un método hijo normalmente tendrá pocas líneas de código (lo específico), mientras el método del padre implementa todo el núcleo de código común y probablemente sea llamado por el método del hijo (*super.metodo()*)

Conceptos Orientación a Objetos

Polimorfismo

- Un nombre puede denotar entidades (objetos o métodos) de tipos/clases diferentes: es posible usar valores de diferentes tipos con una interficie uniforme
- La misma entidad (objeto o método) puede tener más de un tipo/clase

La ambigüedad que es consecuencia del polimorfismo se resuelve en tiempo de compilación (*static binding*) o en tiempo de ejecución (*dynamic binding*), según el LP

3 tipos:

1. Paramétrico: polimorfismo de clases/objetos -> genericidad
2. Ad-Hoc: polimorfismo de métodos -> sobrecarga/*override*
3. De Subtipo o Inclusión: polimorfismo de objetos

Conceptos Orientación a Objetos en Java

Polimorfismo Ad-Hoc

Un mismo nombre denota métodos diferentes

3 variantes:

1. Redefinir ciertos operadores del lenguaje -> *overload*
   Java: sólo operador "+" para concatenar Strings
2. Sobrecargar métodos de una clase -> *overload*
   un mismo nombre de método se aplica a distinto número y/o tipo de argumentos y/o tipo de retorno
3. Redefinir la implementación del método de una clase dentro de una subclase -> *override*

Conceptos Orientación a Objetos en Java

Polimorfismo Ad-Hoc

Binding:

- La sobrecarga de operadores/métodos de la misma clase no ofrece ambigüedad, pues los parámetros son diferentes -> se resuelve en *tiempo de compilación*

  Java: no se permite que dos métodos se diferencien sólo en el tipo de lo que retornan (podría generar ambigüedades)

```
tipo1 g(t1 x, t2 y) {…}
tipo2 g(t1 x, t2 y) {…} //Error

tipo1 z = g(x,y);   //Error incluso si no ambiguo
```

- La redefinición dentro de una subclase se tiene que resolver en *tiempo de ejecución*

Conceptos Orientación a Objetos en Java

Polimorfismo de Subtipo

Es aquel en que con el mismo nombre de variable se pueden denotar instancias de clases diferentes

Binding:

- La ambigüedad sobre la adecuación del uso de estas variables se resuelve en *tiempo de compilación* (comprobación estática de tipos)

- La ambigüedad sobre qué objeto es el receptor del mensaje y qué método hay que ejecutar se resuelve en *tiempo de ejecución* -- *dynamic binding*

Conceptos Orientación a Objetos en Java

Polimorfismo de Subtipo

Ejemplo (1/ 3):

```
public class Animal {
   . . .
  public String talk() { return ""; }
}


public class Gat extends Animal {
   . . .
  public String talk() { return "Miau"; }
}


public class Gos extends Animal {
   . . .
  public String talk() { return "Guau"; }
}
```

Conceptos Orientación a Objetos en Java

Polimorfismo de Subtipo

Ejemplo (2/3):

```
public static void main(String args []) {
  Animal a;
  Gat ga = new Gat();
  Gos go = new Gos();
  a = ga;
  System.out.println(a.talk());
  a = go;
  System.out.println(a.talk());
}
```

Qué se escribirá?

Conceptos Orientación a Objetos en Java

Polimorfismo de Subtipo

Ejemplo (3/3) :

```
public static void main(String args []) {
  Random rnd = new Random();
  int i;
  Animal[] a = new Animal(100);
  for (i=0; i < 100; ++i) {
     a[i] = (rnd.nextInt(2) == 0) ?     new Gat()
                                        new Gos()
  }
  for (i=0; i < 100; ++i) {
     System.out.println(a[i].talk());
  }
}
```

Qué se escribirá?

Conceptos Orientación a Objetos en Java

Polimorfismo de Subtipo

Regla de Comprobación de Tipos en el Polimorfismo de Subtipo

Si un objeto x es de tipo T_2 y T_2 es subtipo de T_1 , entonces x es compatible con el tipo T_1 (pero NO al revés)

Eso implica que (suponiendo que T_2 es subtipo de T_1):

- Si x_2 es de tipo T_2 y x_1 es de tipo T_1 podemos asignar $x_1 = x_2$ (pero NO al revés)
- Si tenemos un método con cabecera 'm (T_1 x_1)' podemos llamarlo con 'm(x_2)' (pero NO al revés)
- En general, se aplica en cualquier sitio donde haga falta una comprobación de tipos (por ej., parámetros de clases genéricas)

Conceptos Orientación a Objetos en Java

Polimorfismo de Subtipo

Qué consecuencias tiene el hecho de que (suponiendo T_2 subtipo de T_1) si x_2 es de tipo T_2 y x_1 es de tipo T_1 podamos asignar $x_1 = x_2$?

- Cuando después se usen los métodos de x_1, realmente se estarán ejecutando los que hay implementados en T2

```
GosPetit gp; //subclase de Gos
…
a = gp;
a.talk();
```

- Debido a la comprobación estática, x_1 no podrá usar los métodos de T2 que no estén en T1, pues el compilador comprueba que el método pertenezca a la clase (o alguna de sus superclases)

```
a = ga;        // asumiendo que solo los gatos arañan
a.scratch(); //Error de compilación
```

Conceptos Orientación a Objetos en Java

Compatibilidad y Comprobación de tipos

Cast: Mecanismo que permite la compatibilidad entre tipos en Java

- El cast es automático:
  - Entre clases numéricas simples
  - En la herencia de hijos a padres (padre = hijo)
- Resto de casos, se ha de hacer explícito: *objeto1 = (clase) objeto2*

Java hace un híbrido de comprobación estática y asignación dinámica de tipos. Si hay una asignación entre clases diferentes (A = B):

- En tiempo de compilación se comprueba que B sea del mismo tipo o hijo de A. Cada objeto solo puede acceder a los métodos de la clase en la que es declarado (o de sus ascendientes)
- En tiempo de ejecución se comprueba que B sea del mismo tipo o hijo de A -> si el tipo ha cambiado, se accede a los métodos de su nueva clase (visibilidad de ámbito dinámico: se accede siempre a la entidad más específica)

Conceptos Orientación a Objetos en Java

Compatibilidad y Comprobación de tipos

Ejemplo (1/3) :

```
public class Padre {
   public void metodoComun() {
     System.out.println("Soy metodoComun en Padre");
   }
}

public class Hijo extends Padre {
   public void metodoComun() {
     System.out.println("Soy metodoComun en Hijo");
   }

   public void metodoHijo() {
     System.out.println("Soy metodoHijo en Hijo");
   }
}

public class Nieto extends Hijo {
   ...
}
```

Conceptos Orientación a Objetos en Java

Compatibilidad y Comprobación de tipos

Ejemplo (2/3) :

- ```
  Padre p = new Padre();
  Hijo h = new Hijo();
  p = h;              // Compilación correcta
  p.metodoHijo(); // Error de compilación
  ```

- ```
  Padre p = new Padre();
  Hijo h1 = new Hijo();
  Hijo h2 = new Hijo();
  h1 = p;           // Error de compilación
  h2 = (Hijo) p; // Compilación correcta – Error de ejecución
  // Exception in thread "main" java.lang.ClassCastException:
  // Padre cannot be cast to Hijo
  ```

Conceptos Orientación a Objetos en Java

Compatibilidad y Comprobación de tipos

Ejemplo (3/3) :

- ```
  Padre p = new Padre();
  Hijo h1 = new Hijo();
  Hijo h2 = new Hijo();
  p = h1;           // Compilación correcta
  h2 = (Hijo) p;    // Compilación correcta
  h2.metodoHijo(); // Ejecución correcta
  ```

- ```
  Padre p = new Padre();
  Hijo h = new Hijo();
  Nieto n = new Nieto();
  p.metodoComun();  // Ejecuta el método de la clase Padre
  p = h;
  p.metodoComun();  // Ejecuta el método de la clase Hijo
  p = n;
  p.metodoComun();  // Ejecuta el método de la clase Hijo
  ```

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación

Abstracción es un proceso mediante el que:

- Destacamos detalles relevantes asociados al estudio o descripción de situaciones complejas
- Ignoramos los detalles irrelevantes para la comprensión de estas situaciones
- Aislamos un elemento de su contexto o del resto de elementos que lo acompañan
- Nos preocupamos más del "¿Qué hace?" que del "¿Cómo lo hace?"

Clasificación es un proceso mediante el que agrupamos los elementos de un conjunto según lo que tienen en común

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación

Ejemplo 1: Queremos guardar datos de tipo numérico (naturales, enteros, reales, etc) donde tendremos una serie de operaciones comunes (suma, resta, producto, etc) pero que se implementan de manera diferente en cada tipo numérico concreto

Solución: Definir una clase **`Numeric`** donde las operaciones *se definen, pero **NO** se implementan*. Será OBLIGATORIO implementarlas en las subclases -> Diremos que **`Numeric`** es una clase **abstracta**, y no se ha de instanciar

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación

Ejemplo 2: Queremos implementar el concepto de *conjunto de cosas* (p.ej. una lista o una cola) donde tengo claro cuáles son las operaciones del conjunto, pero me importa poco cuál es su contenido (las "cosas"). No queremos, de entrada, restringir la posibilidad del tipo que pueden tener los elementos del conjunto

Solución: **Parametrizo** la clase conjunto de cosas, haciéndola independiente de las cosas concretas y concentrando la implementación en las operaciones

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Clases Abstractas

Una clase **abstracta** (o de *implementación diferida*) es una clase a la que le falta algún método por definir*. Éstos se denominan *métodos abstractos*

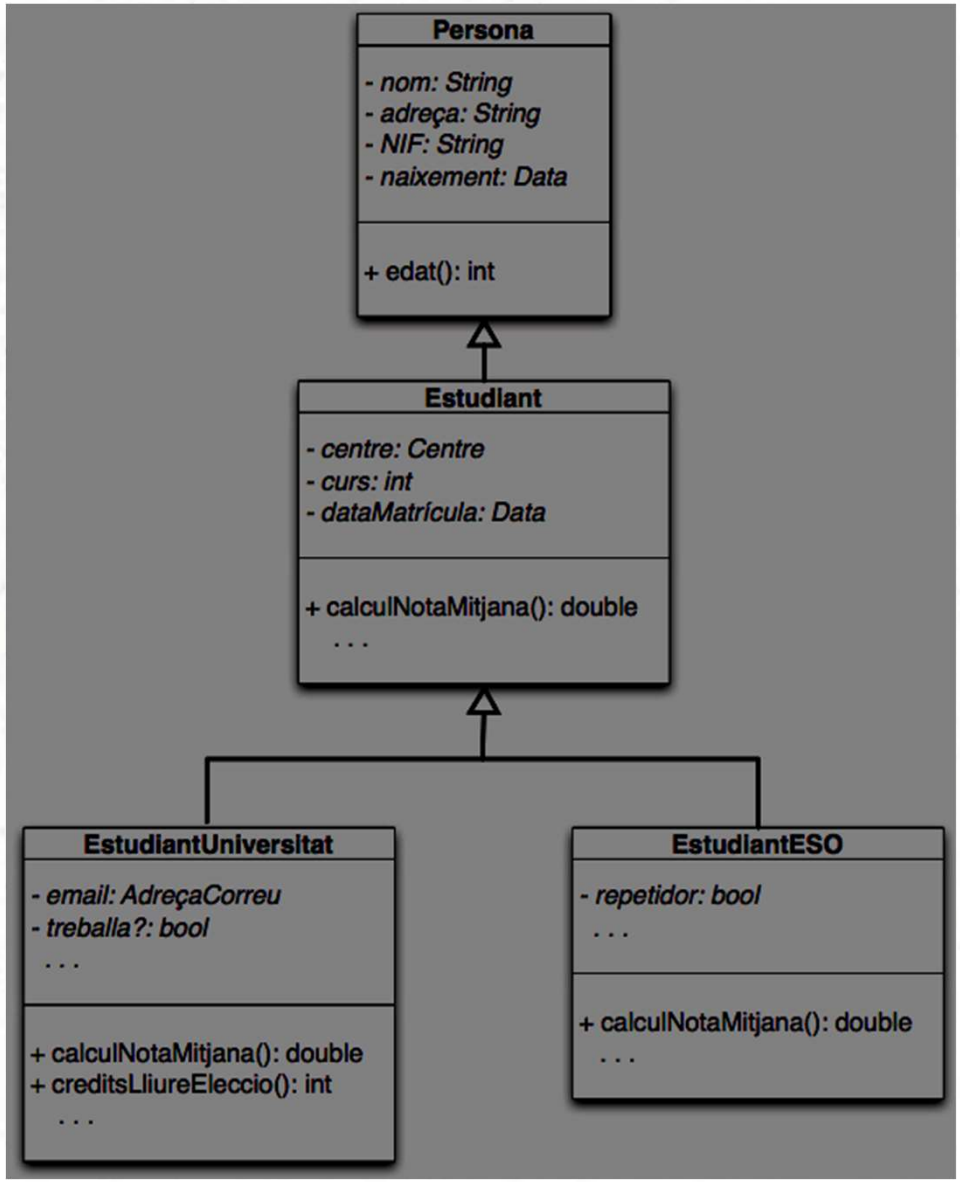
Cuándo usarlas?

- Para facilitar la definición de nuevas subclases: se crea una superclase para reflejar aquello que las subclases tienen en común y se definen métodos abstractos para que cada subclase implemente las diferencias
- Para obligar a definir un cierto método en las subclases

*Esta definición es general y en el caso de Java, hay que matizarla debido a que:
-- Java permite definir clases abstractas sin métodos abstractos
-- En Java existen las *interfaces*, que veremos más adelante

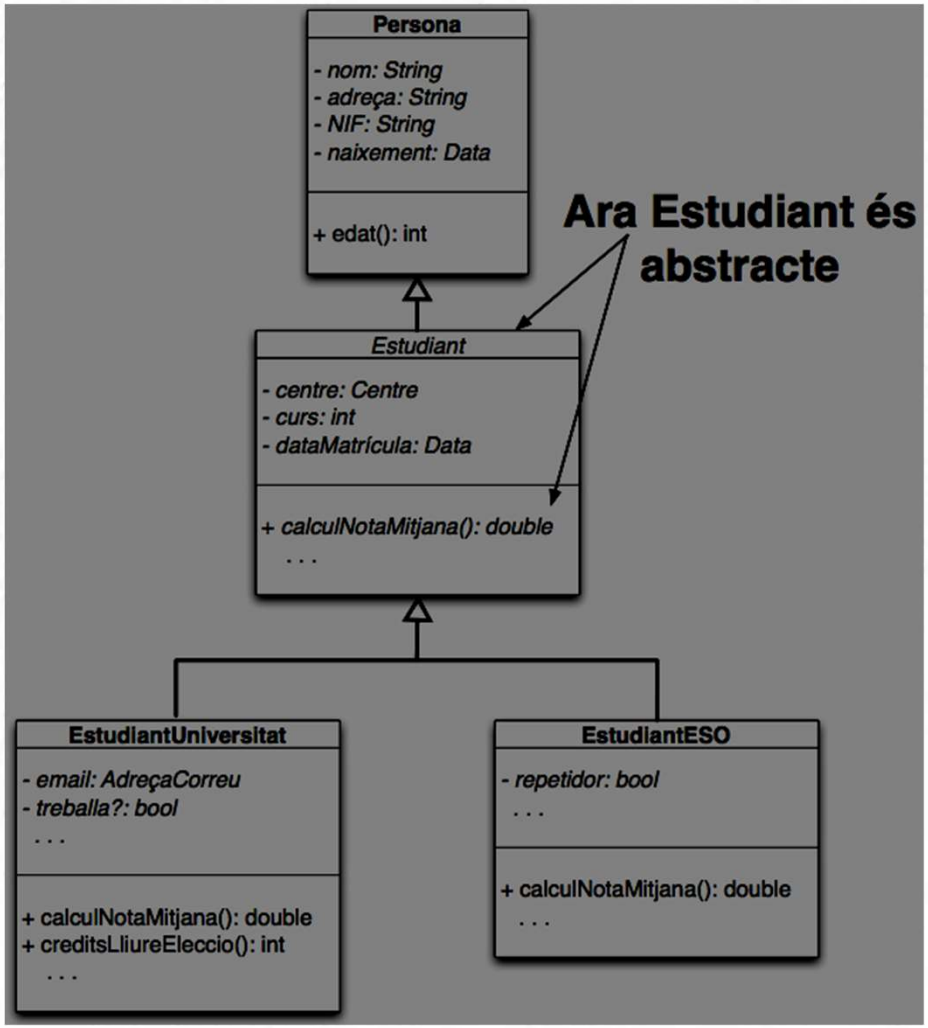
Conceptos Orientación a Objetos en Java

1) Herencia sin Abstracción



Conceptos Orientación a Objetos en Java

2) Herencia con Abstracción



Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Clases Abstractas

Las clases abstractas (*abstract*)

- no se pueden instanciar
- obligan a sus subclases a implementar el(los) método(s) abstracto(s) o a continuar siendo abstractas
- además del(los) método(s) abstracto(s), pueden tener métodos implementados
- pueden tener atributos concretos, pero no abstractos
- pueden ser hijas y padres de otra clase abstracta
- pueden ser hijas y padres de una clase concreta

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Clases Abstractas

En java se usa la cláusula *abstract* para prefijar clases y métodos
Ejemplo:

```
public abstract class Poligono {
  abstract Point centro();
  abstract void girar(int grados);
  void dibujar() { . . . }
  …
}

public class Cuadrado extends Poligono {
  Point centro() { . . . }
  void girar (int grados) { . . . }
  …
}

public class Triangulo extends Poligono {
  . . .
}
```

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Interfaces

Clases particulares de Java que son completamente abstractas: sólo contienen métodos de implementación diferida* y no pueden tener atributos

Se definen con la cláusula *interface* en lugar de *class*:

```
public interface Girable {
   public Point centro();
   public void girar(int grados);
}

public interface Pintable {
   public void setColor(Color c);
   public void pintar();
}
```

Sólo admiten visibilidad *public* y *package* (el default)
Pueden heredar (*extend*) de una o varias interfaces

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Interfaces

Diferencias con las herencia/clases abstractas:

- No es realmente una herencia (no hay nada implementado que heredar)
- Se usan con la cláusula *implements* (y la clase debe proporcionar implementación para todos los métodos declarados en la interface)

```
public class Cuadrado implements Girable {. . . }
```

- Una clase puede "implementar" más de una interface:

```
public class Cuadrado implements Girable, Pintable {...}
```

Se puede combinar con la herencia:

```
public class Cuadrado extends Poligono
                      implements Girable, Pintable {...}
```

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Interfaces

Se puede considerar que la herencia múltiple se cubre de forma bastante limitada con las interfaces:

- No presentan los problemas de falta de seguridad de la herencia múltiple porque no se pueden definir objetos de las interfaces
- No hay reglas sobre qué método se implementa si una clase implementa múltiples interficies con el mismo método (y signatura), pero como los métodos no están implementados no hay problemas de ambigüedad (colisiones)
- Pero: no se aprovecha nada de la herencia al no haber nada implementado *

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Interfaces

*Desde Java 8, existe el concepto de *interface default method*: los métodos pueden contener una implementación, que se usa **si y sólo** si la clase que implementa la interface no proporciona implementación para este método

```
public interface Girable {
  public Point centro();
  default Point centro() {
    ...
  }
  . . .
}
```

Si una clase implementa múltiples interfaces que contienen el mismo método (+ signatura), y al menos una de ellas declara el método como *default method*, el compilador exige que la clase implemente explícitamente el método

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Clases Genéricas

Las clases genéricas o parametrizadas usan una o más clases de manera indefinida (los *parámetros*), de tal forma que la clase genérica tiene pleno significado independientemente de cuál sea la clase particular que está indefinida: son clases que dependen de otras clases que no tienen por qué estar definidas *a priori*.

Ejemplo: las estructuras de datos que sirven de *contenedores* (pilas, árboles, colas, listas, etc). Se caracterizan por su estructura y sus operaciones, independientes de los elementos que contienen

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Clases Genéricas

La clase se define con uno (o más) parámetros *genéricos*:

```
Lista<T>  Hashtable<K,V>  ControladorMantenimiento<E>
```

Y cuando se instancia se ha de especificar quién es el(los) parámetro(s) *real(es)*:

```
Lista<Persona> list = new Lista<Persona>()
Hashtable<String,Integer> h=new Hashtable<String,Integer>()
```

Los parámetros genéricos han de ser de tipos complejos que hereden de *Object*

Ejemplo (uso de clase genérica de Java):

```
Vector<Integer> miVector = new Vector<Integer>();
for (int i=0;i<10;i++) miVector.add(Integer.valueOf(i*i));
Integer x = miVector.get(3);
```

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Clases Genéricas

Ejemplo : Las clases contenedoras en Java son genéricas (interface Collection<E>)

```
ArrayList<Fruit> f;

LinkedList<Integer> ll;

Set<String> set;

Queue<Character> q;

Map<String, String> myPets;
```

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Clases Genéricas

Ejemplo (creación de una nueva clase genérica):

```
public class Box<T> {
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}


Box<Integer> integerBox1;

Box<Integer> integerBox2 = new Box<Integer>();
```

```
integerBox2.set(Integer.valueOf(12));
Integer i1 = integerBox2.get();
String i2 = (String) integerBox2.get(); //ERROR compil
```

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Clases Genéricas

Genericidad Restringida:

Consiste en imponer restricciones sobre el parámetro (que sea *subclase de o implemente* alguna otra clase). Es útil para asegurarnos de que los parámetros reales dispondrán de ciertas operaciones que nos pueden hacer falta

```
public class ClaseGenerica<T extends C1,I1, I2, …>
```

Ejemplo: si añadimos a la clase *Box<T>* la operación

```
public void tratarElem() {
   …
   t.girar(grados);
   …
}
```

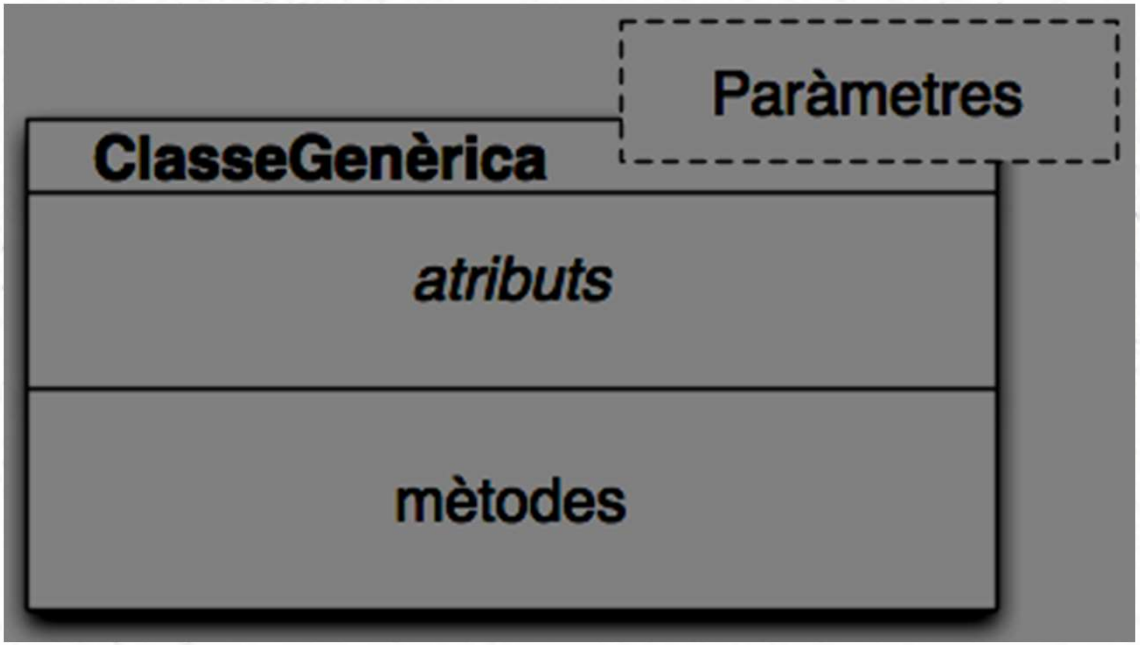
habría que definir la clase como:

```
public class Box<T extends Poligono>
public class Box<T extends Girable>
```

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Clases Genéricas

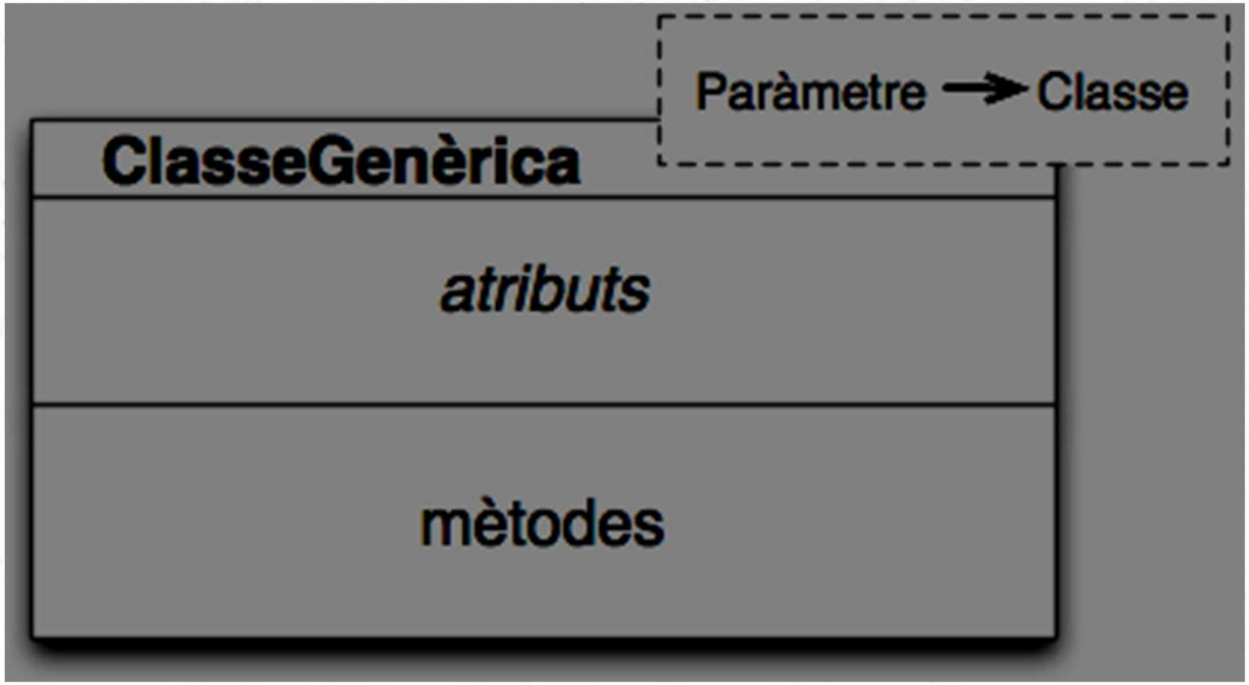
Notación UML (genericidad no restringida):



Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Clases Genéricas

Notación UML (genericidad restringida): Se indica que **`Paràmetre`** ha de ser subclase de **`Classe`**



Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Clases Genéricas

- La genericidad se puede usar de manera recursiva:

```
Lista<Lista<Persona>>
Map<ArrayList<String>,Map<Queue<Float>,Integer>>
HashTable<Key,Vector<E>>
```

- La genericidad se puede combinar con herencia:

```
class MyStringList extends ArrayList<String> {…}

class A<X,Y,Z> {…}
class B<N> extends A<N,String,Integer> {…}
```

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Clases Genéricas

- En Java, la genericidad se comprueba *en tiempo de compilación* (*static binding*) -> restricción importante: NO se puede hacer un *new* de la clase del parámetro dentro de la clase genérica:

```
public class ClaseGenerica<T> {
        …
    T instanciarParametro() {
      return new T();  //ERROR de compilación
    }
}
```

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación

Los conceptos de *clase abstracta* y *genérica* son diferentes:

- A diferencia de la abstracta, la clase genérica está completamente implementada, pero usa una o varias clases que se definirán en el momento de la instanciación
- Las clases abstractas necesitan de la herencia para ser instanciadas, las genéricas no

Una clase puede ser abstracta y genérica. Por ejemplo, en Java **`Iterator<E>`** es *como* una clase abstracta (*interface* en realidad) que obliga a implementar **`hasNext()`** y **`next()`**

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Iteradores

Para definir un iterador:

```
public class MiClaseConIterador<E> {
      …
  // método que devuelve el iterador
  public Iterator<E> iterator() {
    return new MiIterador();
  }
      …
  private class MiIterador implements Iterator<E> {
    public MiIterador() { … }
    public Boolean hasNext() { … }
    public E next() { … }
    public void remove() { … }
  }
}
```

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Iteradores

Para usar el iterador:

```
MiClaseConIterador<MiClase> claseIterador1 =
                          new MiClaseConIterador<MiClase>();
…
Iterator<MiClase> iter = claseIterador1.iterator();
while (iter.hasNext()) {
  MiClase elem = iter.next();
  …   //usar elem
}
…
```

Conceptos Orientación a Objetos en Java

Abstracción y Clasificación: Iteradores

Muchas de las clases de la librerías estandard de Java de contenedores tienen iteradores ya predefinidos:

```
Vector<Integer> list = new Vector<Integer>();
…
Iterator<Integer> iter = list.iterator();
while (iter.hasNext()) {
  Integer elem = iter.next();
  …  // usar elem
}
```

Aunque en estos casos se puede usar la version abreviada:

```
Vector<Integer> list = new Vector<Integer>();
…
for(Integer elem:list) {
  … // usar elem
}
```