# Conceptos de POO

*Programación Orientada a Objetos*
*Facultad de Informática*

**Juan Pavón Mestras**
**Dep. Sistemas Informáticos y Programación**
**Universidad Complutense Madrid**

## Tecnología de objetos

- Construcción de software utilizando ***componentes reutilizables*** con ***interfaces*** bien definidas

Las aplicaciones no se construyen desde cero

- Se pueden considerar varias capas:
  - La capa inferior son ***objetos*** (software chips)
  - La capa intermedia son colecciones de objetos relacionados (***patrones de diseño OO***)
  - La capa superior son aplicaciones que resuelven problemas específicos (armazones o ***frameworks***)
- Promueve el diseño basado en interfaces y arquitecturas estándares, con una organización y un proceso

## Tecnología de objetos

- Los objetos permiten representar los conceptos esenciales de una entidad ignorando sus propiedades accidentales

Lo principal es la visión externa

- Durante el desarrollo del sistema lo importante es "qué es" y "qué hace" un objeto antes de decidir "cómo" se implementará
  - Comportamiento: operaciones que los clientes realizarán en el objeto, y operaciones que realizará sobre otros objetos
    - Un cliente es un objeto que usa los recursos/servicios de otros objetos (servidores)



## Tecnología de objetos

- Motivos que han conducido al éxito la tecnología de objetos
  - Avances en arquitectura de computadores
  - Avances en lenguajes de programación (C++, Smalltalk, Ada, Java, ...).
  - Ingeniería del software (modularidad, encapsulado de la información, proceso de desarrollo incremental)
  - Los límites de la capacidad de gestionar la complejidad de los sistemas simplemente con técnicas de descomposición algorítmica

Más centrado en el diseño y aplicación de técnicas de ingeniería de software

## Tecnología de objetos

- Ventajas de la tecnología de objetos
  - Mejoras significativas de la productividad y calidad del código
  - Estabilidad de los modelos respecto a entidades del mundo real
  - Construcción iterativa
  - Promueve la reutilización de software y de diseños (componentes, frameworks)
  - Los sistemas OO son generalmente más pequeños que su equivalente no OO: menos código y más reutilización
  - Permite desarrollar sistemas más preparados para el cambio
  - Vále para aplicaciones de pequeño y gran tamaño

Más centrado en el diseño y aplicación de técnicas de ingeniería de software



## Qué es la Programación Orientada a Objetos

- Organización de los programas de manera que representan la interacción de las cosas en el mundo real
  - Un programa consta de un conjunto de objetos
  - Los objetos son abstracciones de cosas del mundo real
  - Nos interesa qué se puede hacer con los objetos más que cómo se hace
  - Cada objeto es responsable de unas tareas
  - Los objetos interactúan entre sí por medio de mensajes
  - Cada objeto es un ejemplar de una clase
  - Las clases se pueden organizar en una jerarquía de herencia

*La programación OO es una simulación de un modelo del universo*

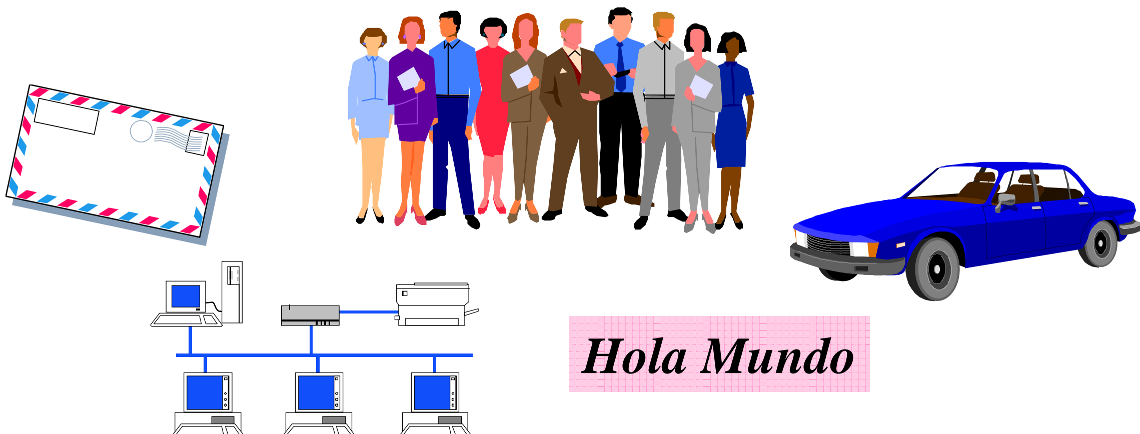
# Conceptos generales

- Objetos y clases
- Atributos
- Mensajes y métodos
- Encapsulado y ocultación
- Interfaces
- Herencia de clases
- Polimorfismo
- Vinculación dinámica
- Composición de objetos
- Modelado con objetos
- Bibliotecas



# Objetos

- Los objetos son cosas
- Los objetos pueden ser simples o complejos
- Los objetos pueden ser reales o imaginarios



## Atributos

- Valores o características de los objetos
- Permiten definir el estado del objeto u otras cualidades

- Velocidad
- Aceleración
- Capacidad de combustible

variables

- Marca
- Color
- Potencia
- Velocidad máxima
- Carburante

constantes



## Mensajes

- Los objetos se comunican e interaccionen entre sí por medio de mensajes
- Si un objeto desea que otro objeto ejecute un método le envía un mensaje que puede tener información adicional en forma de parámetros
- Componentes de un mensaje
  - Objeto destinatario del mensaje (miCoche)
  - Método que se debe ejecutar como respuesta (cambiar marcha)
  - Parámetros necesarios del método (segunda)

## Métodos (u operaciones)

- Un objeto puede realizar una serie de acciones

- Arrancar motor
- Parar motor
- Acelerar
- Frenar
- Girar a la derecha (grados)
- Girar a la izquierda (grados)
- Cambiar marcha (nueva marcha)

método argumentos



## Clases

- Los objetos con estados similares y mismo comportamiento se agrupan en clases
- La definición de la clase especifica el comportamiento y los atributos de los ejemplares (objetos) de la clase

## Definición de clases en C++

```
enum Carburante {
      diesel, super, sinplomo
};

class Coche {
  char* marca;
  double vel_max;
  int potencia;
  Carburante tipo_carburante;

  double velocidad;
  double aceleracion;

public:
  void arrancar() {
    // instrucciones para arrancar el coche
  };
```

```
void frenar() {
    // instrucciones para frenar el coche
  };

  void acelerar() {
    // instrucciones para acelerar el coche
  };

  void girar_derecha(short grados) {
    // instrucciones para girar a la derecha
  };


  // etc.
}; // fin de definición de la clase Coche
```



## Definición de clases en Java

```
class Coche {
// atributos:
  String marca;
  double vel_max;
  int potencia;
  String tipo_carburante;

  double velocidad;
  double aceleracion;

// métodos:
  void arrancar() {
    // instrucciones para arrancar el coche
  };
```

```
void frenar() {
    // instrucciones para frenar el coche
  };

  void acelerar() {
    // instrucciones para acelerar el coche
  };

  void girar_derecha(short grados) {
    // instrucciones para girar a la derecha
  };


  // etc.

}; // fin de definición de la clase Coche
```

# Un programa en POO

- Un programa consta de un conjunto de instancias o ejemplares de objetos (*object instances*) y un flujo de control principal (*main*)
- Durante la ejecución del programa:
  - Los objetos se crean y se destruyen
    - Gestión dinámica de la memoria
  - Se les solicita a los objetos que ejecuten métodos (operaciones)



# Un programa en C++

```
main() {
   Coche *c=new Coche();  // crea un objeto Coche
   c-> arrancar           // utiliza el objeto
   // ...
   dispose(c);            // elimina el objeto
}
```

## Un programa en Java

```
class Programa {
   public static void main(String args[]) {
     Coche c=new Coche();   // crea un objeto Coche
     c.arrancar();               // utiliza el objeto
     // ...
   } // se elimina el objeto cuando nadie lo puede utilizar
     // ¡ automáticamente !
}
```



## Un programa en POO

Ejercicio:

¿Cuál es la representación en memoria de un POO?

## Técnicas de la POO

- Proporciona los siguientes mecanismos
  - Clasificación
  - Encapsulado y ocultación
    - Abstracción
  - Herencia de clases
    - Polimorfismo
  - Composición de objetos
  - Asociaciones
- Y adicionalmente
  - Concurrencia
  - Persistencia
  - Genericidad
  - Excepciones



## Clasificación

- Clasificación es un mecanismo para ordenar el conocimiento
  - Es fundamentalmente un problema de búsqueda de similitudes
  - Al clasificar buscamos grupos de cosas que tienen la misma estructura o muestran un comportamiento similar
- Clasificación y desarrollo orientado a objetos
  - Mediante la clasificación, los objetos con la misma estructura de datos y comportamiento se agrupan para formar una clase
  - La clasificación permite identificar clases y objetos en el desarrollo
    - Esta es una de las tareas más difíciles del A&D OO
- Después de clasificar podemos
  - Establecer asociaciones
  - Agrupar
  - Definir jerarquías

## Encapsulado y ocultación

- Los objetos encapsulan (agrupan) sus operaciones y su estado:
  - el comportamiento del objeto está definido por los métodos
  - el estado está definido por los datos o atributos del objeto
- Ocultación de información
  - Las partes necesarias para utilizar un objeto son visibles (interfaz *pública*): métodos
  - Las demás partes son ocultas (*privadas*)
    - El volante, el cuentakilómetros, los pedales y la palanca de cambios representan una interfaz pública hacia los mecanismos de funcionamiento del automóvil
    - Para conducir no es necesario conocer la mecánica del automóvil (su implementación)



## Encapsulación

- Consiste en separar los aspectos externos del objeto (las partes a las que pueden acceder otros objetos) de los detalles de implementación internos (ocultos a otros objetos)
  - Los objetos agrupan su estado (los atributos o datos) y su comportamiento (los métodos)
    - Relación clara entre el código y los datos
- Modularidad
  - El código de un objeto puede desarrollarse y mantenerse independientemente del código de los otros objetos
- Ocultación de información
  - Sólo son visibles las partes necesarias para utilizar un objeto (interfaz pública): métodos y atributos
  - Las demás partes son ocultas (privadas). No es necesario saber como esta hecho para utilizarlo

## Encapsulación

- Toda clase debe tener dos secciones:
  - Interfaz: parte visible
  - Implementación: representación de la abstracción
- Ejemplo
  - El volante, el cuentakilómetros, los pedales y la palanca de cambios representan una interfaz pública hacia los mecanismos de funcionamiento del automóvil
  - Para conducir no es necesario conocer la mecánica del automóvil (su implementación)



## Abstracción

- Capacidad de especificar las características comunes a un conjunto de clases
  - Definición parcial del estado y del comportamiento
  - Declaración del comportamiento (interfaz)
- Clases abstractas
  - Especificación de datos y comportamiento común a un conjunto de clases
  - Forzar a que las subclases proporcionen un comportamiento específico
- Interfaces
  - Declaración de métodos a incorporar en las clases que implementen la interfaz
  - Definición de constantes

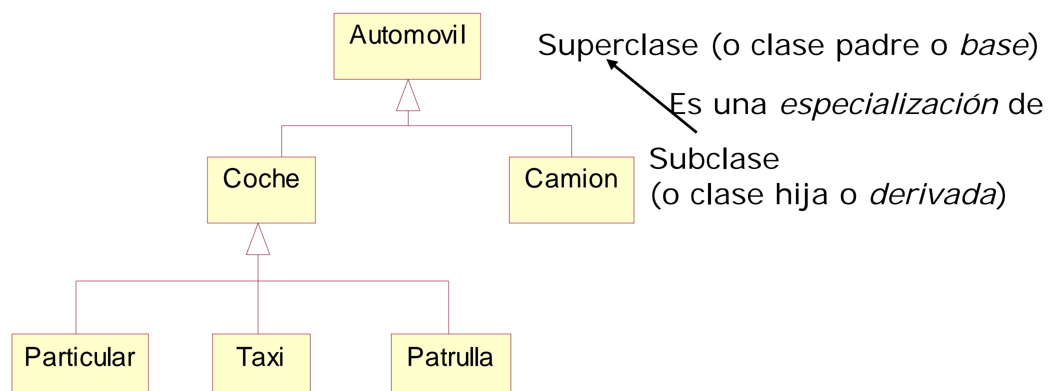
## Interfaces

- La interfaz define un conjunto de métodos
- Una clase de objetos puede implementar una o varias interfaces
  - La interfaz *Volante* ofrece los métodos:
    - Girar a la izquierda
    - Girar a la derecha
  - La interfaz *PedalAcelerador* ofrece los métodos:
    - Pisar acelerador
    - Levantar acelerador
  - La clase *Coche* implementa las interfaces *Volante* y *PedalAcelerador*,



## Herencia de clases

- Permite definir una clase especializando otra ya existente
  - Se extiende un tipo de datos, heredando las características comunes y especificando las diferencias
  - Permite la reutilización de código
- Implementa la relación *es un* o *es una clase de*
- Define una jerarquía de clases:



## Herencia de clases

- Los métodos y atributos de la superclase son heredados por las subclases,

- pero, en la especialización:
  - Se pueden añadir nuevos métodos y atributos
    - La clase Taxi tiene taxímetro y las operaciones poner en marcha taxímetro y para taxímetro
  - Se pueden redefinir los métodos
    - La clase Taxi redefine el método arrancar: si es recién subido un nuevo cliente, poner en marcha taxímetro
  - Se pueden ocultar métodos
    - La subclase CocheAutomático oculta el método cambiar marcha



## Polimorfismo

- Propiedad de una entidad de adquirir formas distintas
  - Polimorfismo inclusivo
    - Un objeto de un tipo dado (tipo estático) puede ser sustituido por otro del mismo tipo, pero de distinta clase
      - Las interfaces definen tipos
      - Las clases definen implementaciones
  - Polimorfismo paramétrico
    - Un método puede actuar sobre diversos tipos
    - Ejemplo: el método *insertar()* de una lista ordenada funciona sobre cualquier clase que tenga implementado el método *menor()*
  - Sobrecarga de método
    - un mismo nombre de método y varias implementaciones del mismo (en la misma o distintas clases)
    - Ejemplo: el método *insertar()* implementado en una lista, en una lista ordenada, en un conjunto y en una tabla.

## Vinculación

- Asociación de un atributo con su valor
  - Variable con un valor
  - Llamada a función con el cuerpo de la función
- Puede ser
  - Estático (*early binding*): en tiempo de compilación
    - C++, Pascal, COBOL, FORTRAN
  - Dinámico (*late binding*): en tiempo de ejecución
    - C++, Smalltalk, Objetive-C, Java



## Vinculación dinámica

- Concepto relacionado con el polimorfismo inclusivo
  - Cuando una variable de un tipo Coche posee una referencia a un objeto de clase Taxi (siendo Taxi descendiente de Coche) y se invoca a un método por medio de la variable,

    ¿cuál se ejecuta si hay una implementación en Coche y otra en Taxi (p.ej. Arranca)?
  - Los lenguajes con vinculación dinámica tratan de asociar la ejecución al tipo más específico, por tanto a Taxi
  - El mecanismo general de búsqueda es empezar por el tipo dinámico de la variable receptora y remontarse por la jerarquía de herencia hasta encontrar una implementación válida

## Clases abstractas

- Descripción incompleta de algo
  - Describen partes de objetos
  - Se usan en la jerarquía de herencia de clases
  - Utilizadas en polimorfismo para definir operaciones comunes
  - No pueden crearse objetos directamente de una clase abstracta
    - la clase Automóvil permite definir qué se puede hacer con objetos de las subclases
    - pero los objetos son ejemplares de clases concretas: Taxi, Camión, etc.



## Implementación de interfaces

- Las interfaces sólo definen la signatura de un conjunto de métodos
- Las clases implementan interfaces
- Facilitan la definición de bibliotecas de componentes reutilizables y armazones software (*frameworks*)

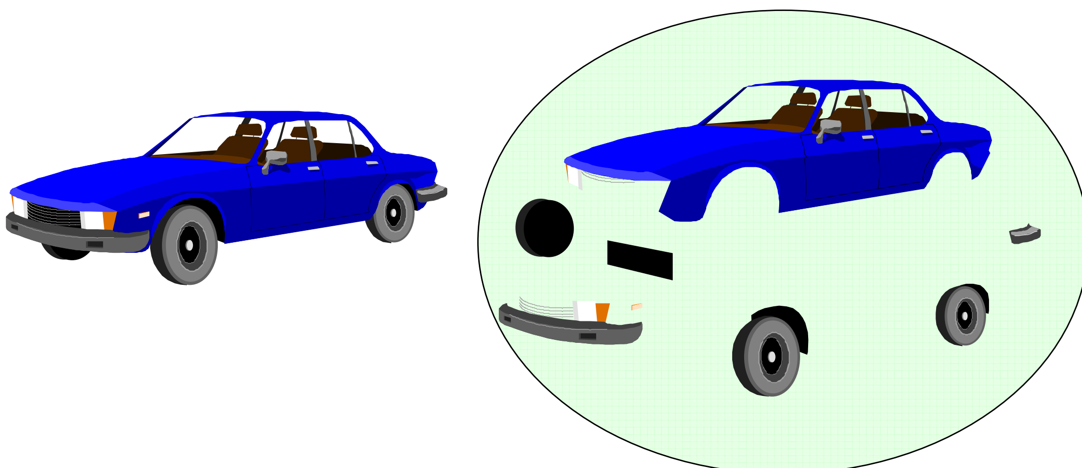
## Diseñar con interfaces

- Todas las clases que implementan una interfaz pueden responder a un mismo conjunto de peticiones
- Los clientes no tienen que preocuparse de las clases específicas de los objetos que utilizan
  - La implementación del cliente está menos atada a la evolución de dichas clases
  - Es posible incluso cambiar unas clases por otras (siempre que implementen las operaciones de la interfaz)
- Recomendación:
  *Programar con interfaces y no con implementaciones*
  - Declarar variables del tipo de interfaces, no de clases
  - Utilizar patrones de creación que abstraigan el proceso de creación de objetos concretos



## Composición de objetos

- Agregar o componer varios objetos para obtener una mayor funcionalidad
- Un objeto se puede construir a partir de otros objetos



## Herencia vs. Composición

- Herencia: permite definir una clase a partir de otra
  - Reutilización de *caja blanca*: los aspectos internos de la superclase son visibles a las subclases
  - Soportada por el lenguaje de programación
  - Estática: se define en tiempo de compilación

- Composición: nueva funcionalidad mediante composición de objetos
  - Reutilización de *caja negra*: no hay visibilidad de los aspectos internos de los objetos (objetos como cajas negras)
  - Requiere interfaces bien definidas
  - Dinámica: se define en tiempo de ejecución



## Herencia vs. Composición

- La herencia no permite cambios en tiempo de ejecución
- La herencia rompe la encapsulación
  - La herencia impone al menos una parte de la representación física a las subclases
  - Cambios en la superclase pueden afectar a las subclases
  - Las implementaciones de superclase y subclases están ligadas
  - Si hacen falta cambios para reutilizar una clase en nuevos dominios de aplicación habrá que cambiarla
  - Limita la flexibilidad y al final la reutilización
- Por ello es más práctico utilizar interfaces y clases abstractas
  - Ya que proporcionan menos o ninguna implementación

## Herencia vs. Composición

- La composición es dinámica, en tiempo de ejecución
  - Los objetos adquieren referencias de otros objetos
  - Los objetos tienen que respetar las interfaces de los otros objetos
    - Exige un diseño cuidadoso de las interfaces
  - Hay menos dependencias de implementación
  - Habrá más objetos en el sistema y por tanto el comportamiento del sistema dependerá de las interacciones entre objetos en vez de estar definido en una clase



## Herencia vs. Composición

- Por tanto,

  *es preferible la composición de objetos a la herencia de clases*

- Sin embargo,
  - El conjunto de componentes (objetos) no suele ser lo suficientemente rico en la práctica
  - Mediante herencia se pueden crear nuevos componentes que componer con los existentes

- La herencia y la composición trabajan juntas

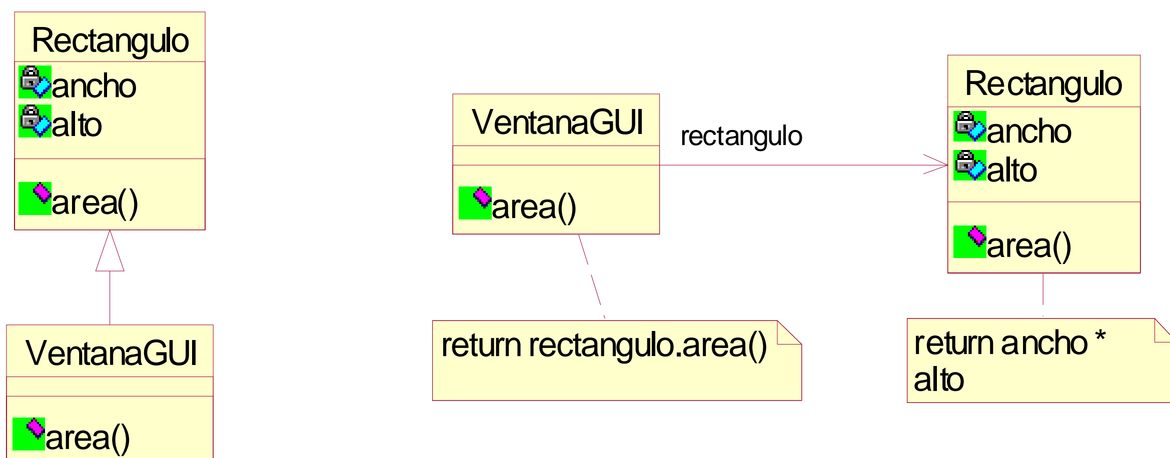
# Herencia vs. Composición

- Delegación de objetos como alternativa a la herencia
  - En la delegación un objeto que recibe una petición delega la ejecución del método a otro objeto (el *delegado*)
    - En el fondo esto es lo que hace una subclase respecto a la superclase (al objeto *this*)
  - Ejemplo: la clase *Ventana*, en vez de heredar de *Rectangulo* (aunque las ventanas son rectangulares) tiene una referencia a un objeto asociado de esa clase y delega algunas operaciones en ella
    - Si hiciera falta cambiar la forma de la ventana en tiempo de ejecución, por ejemplo a *Circulo*, bastaría con cambiar la referencia del objeto correspondiente



# Herencia vs. Composición

- Ejemplo de delegación de objetos como alternativa a la herencia



## Agregación vs. Asociación

- Agregación: un objeto es propietario o responsable de otro objeto
  - Relación *es parte de*
  - Implica que ambos objetos tienen el mismo tiempo de vida
- Asociación: un objeto conoce otro (tiene una referencia)
  - Un objeto puede solicitar una operación en otro objeto pero no es responsable de él
  - La relación de asociación es más débil que la de agregación
- La diferencia es más de intención que de implementación
- Normalmente hay menos agregaciones que asociaciones pero son más duraderas
  - Hay asociaciones que sólo existen durante la ejecución de un método



## ¿Cuándo usar...?

- Herencia, composición, asociación, agregación, ....

  => Patrones de diseño

## Bibliotecas

- Un conjunto de clases e interfaces se pueden agrupar en bibliotecas
  - Así pueden ser reutilizadas en programas distintos, mediante:
    - Herencia de clases
    - Composición de objetos
- Bibliotecas estándar:
  - Están probadas, son robustas y eficientes
  - Suelen adaptarse bien a múltiples dominios
- *Recomendación*: Evitar reinventar la rueda
  - Antes de implementar nada, ver si ya existe



## Modelado con objetos

- Dependiendo del sistema que pretendemos implementar realizaremos unas u otras abstracciones del mundo real
  - El constructor de automóviles definirá como objetos las partes del motor, las ruedas, el chásis, la transmisión, etc.
  - El conductor verá el volante, el panel de mandos, los pedales, etc.
  - El guardia civil considerará la velocidad y la matrícula
- La POO permite representar el problema en términos específicos del dominio
  - Fáciles de comprender
  - Verificables por el usuario que define los requisitos del sistema

## Características adicionales de los lenguajes OO

- Concurrencia
- Persistencia
- Genericidad
- Excepciones



## Concurrencia

- Procesos vs. Threads

- En lenguajes de programación
  - Ada: package
  - Smalltalk: clase Process
  - C++: utilización de librerías que llaman al sistema
  - Java: clase Thread e interfaz Runnable

- Objetos activos vs. Objetos pasivos
  - Objeto activo: tiene su propio hilo de ejecución y contorla su flujo de aplicación
  - Objeto pasivo (modelo clásico): sólo está activo cuando recibe un mensaje de otro objeto (que le presta el flujo de ejecución)

## Persistencia

- Un objeto es persistente cuando su existencia no está ligada a la de su creador
  - En algún proceso se crea el objeto
  - Se guarda su estado en algún repositorio (fichero, BD)
  - Otros procesos pueden recrearlo y volver a guardarlo



## Genericidad

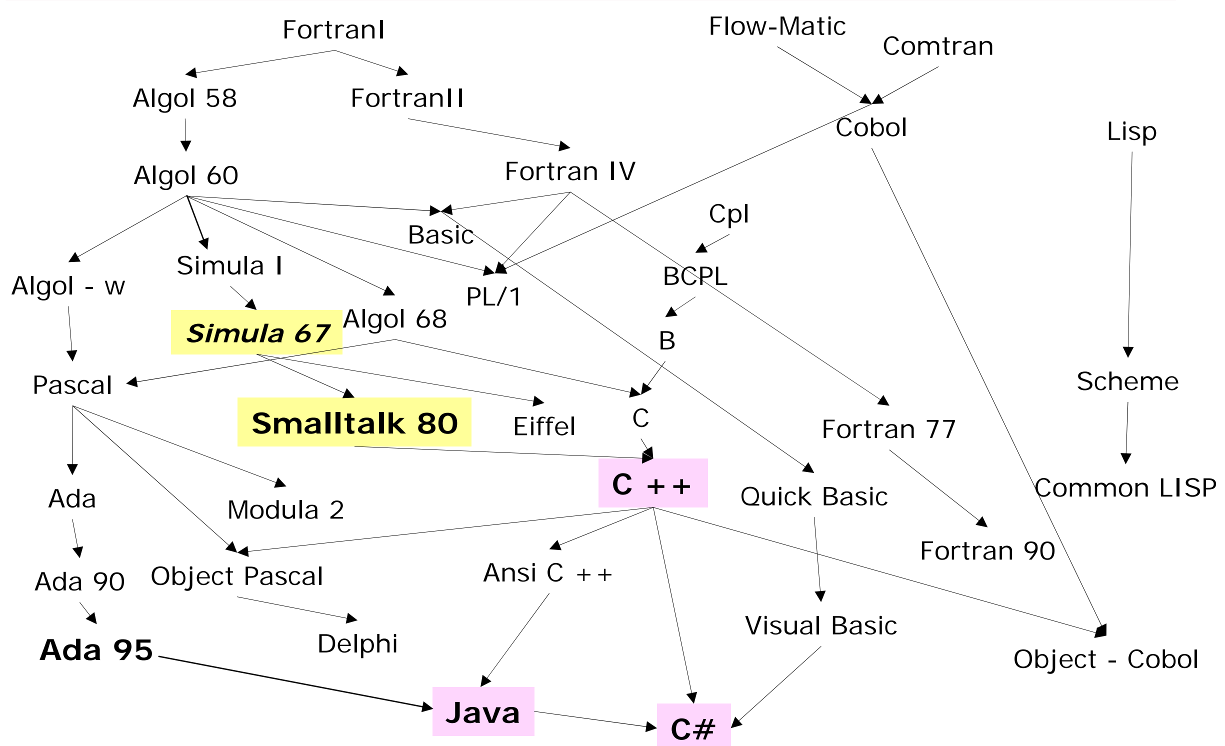
- Una clase genérica o parametrizada es una plantilla para otras clases
  - Ejemplo: con Lista<item> se puede crear Lista<int>, Lista<persona>, Lista<Lista<persona>>
  - La plantilla se puede parametrizar por otras clases, tipos y operaciones
- C++ y Ada ofrecen la posibilidad de definir tipos genéricos

# Excepciones

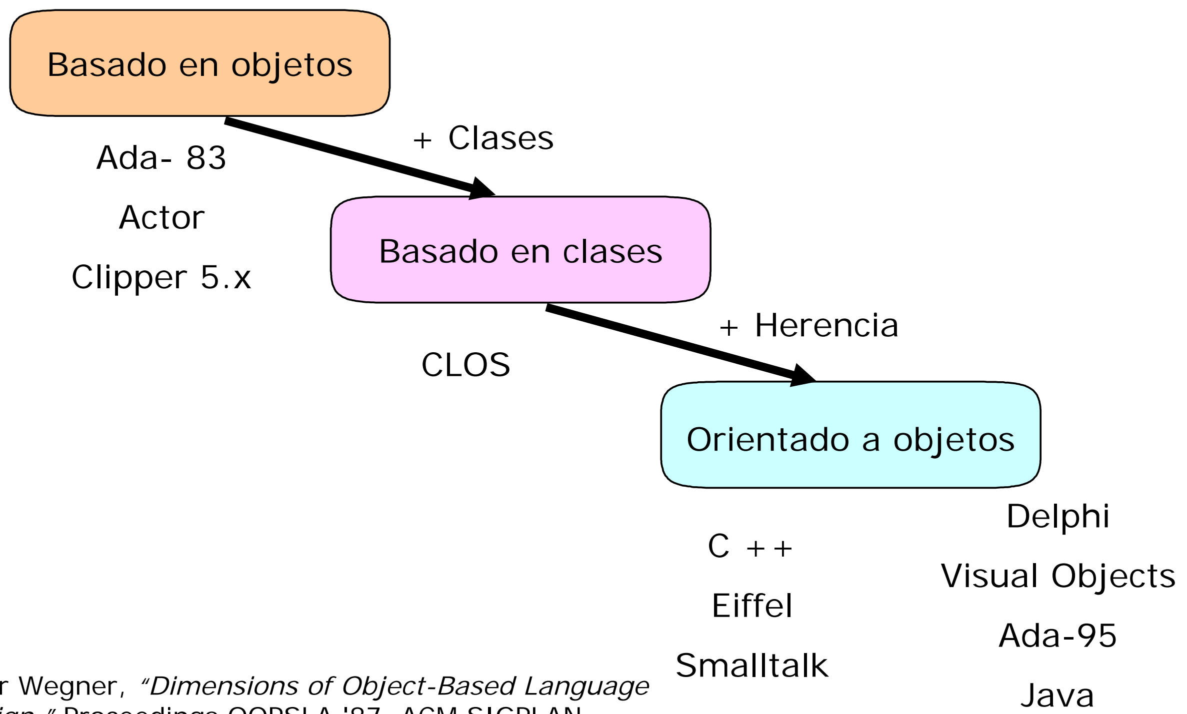
- Situaciones anormales o excepcionales que tienen lugar durante la invocación de un método
- Existen en C++, Ada y Java



# Evolución de lenguajes de programación



# Características de lenguajes de programación




Peter Wegner, *"Dimensions of Object-Based Language Design,"* Proceedings OOPSLA '87, ACM SIGPLAN



# Características de lenguajes de programación

| | Ada 95 | C++ | Java | Smalltalk |
|---|---|---|---|---|
| Paquetes | Si | Si | Si | No |
| Herencia | Simple | Multiple | Simple | Simple |
| Genericidad | Si | Si | No | No |
| Control de tipos | Fuerte | Fuerte | Fuerte | Sin tipos |
| Vinculación | Dinámica | Dinámica y estática | Dinámica | Dinámica |
| Concurrencia | Si | No | Si | No |
| Recolección de basura | No | No | Si | Si |
| Excepciones | Si | Si | Si | No |
| Persistencia | No | No | No | No |

## Resumen de conceptos de POO

- Un programa orientado a objetos es un conjunto de clases que describen el comportamiento de los objetos del sistema
- La computación se realiza por objetos que se comunican entre sí, solicitando que otros objetos realicen determinadas acciones
- Los objetos se comunican mediante mensajes
- Cada objeto tiene su propio estado, que consta de otros objetos
- Cada objeto es un ejemplar de una clase (agrupación de objetos)
- Todos los objetos que son ejemplares de una misma clase pueden realizar las mismas acciones
- Las clases están organizadas en una jerarquía de herencia



## Bibliografía


- T. Budd, *An introduction to Object-Oriented Programming (Third Edition).* Pearson Education, 2001
- E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Paterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994
- B. Stroustrup, *The C++ Programming Language (Third Edition).* Addison-Wesley, 1997
- Agustín Froufe. *Java 2. Manual de usuario y tutorial.* Ed. Ra-Ma
- J. Sánchez, G. Huecas, B. Fernández y P. Moreno, *Iniciación y referencia: Java 2*. Osborne McGraw-Hill, 2001.
- B. Meyer, *Object-Oriented Software Construction (Second Edition).* Prentice Hall, 1997