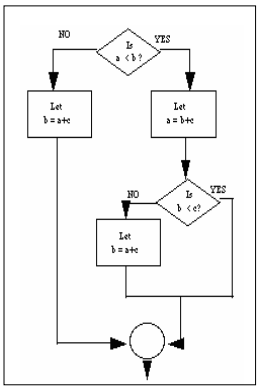
PROGRAMACIÓN ORIENTADA A OBJETOS

Un poco de Historia

- La programación orientada a objetos es una de las formas más populares de programar actualmente con gran acogida por parte de los programadores.
- Esta gran aceptación es debido a sus grandes capacidades y ventajas frente a las antiguas formas de programas.
- La POO ofrece amplias mejoras en el diseño, desarrollo y mantenimiento de Software ofreciendo soluciones de portabilidad y reusabilidad.

Programación Lineal

- Tradicionalmente, la programación fue hecha en una manera secuencial o lineal, es decir, una serie de pasos consecutivos con estructuras consecutivas y bifurcaciones.
- Los lenguajes basado en esta forma de programación estaban muy bien, hasta que se volvían complejos.
- El mayor problema de la programación “espagueti” es que no ofrecían flexibilidad ni mantenimiento.

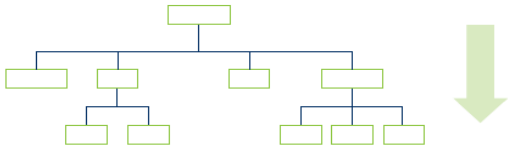


Programación Estructurada

- Frente a esta dificultad, surgieron los lenguajes basados en la programación estructurada. (C, Pascal, Modula,…)
- Su idea principal es la de hacer una programación separada en partes (módulos) se sean ejecutados conforme se requieran.
- Aparece un diseño modular, compuesto por módulos independientes.
- Esta programación se orienta a intentar descomponer el programa mas y mas.

Programación Estructurada

- Según indicó Edsger Dijkstra , todo programa se puede escribir usando:
  - 3 estructuras básicas de control:
    - Secuenciales, Condicionales, Iteraciones
  - No utilización de la sentencia GOTO
  - Definición de Módulos
- De esta forma nos encontramos un diseño TOP-DOWN (esta división toma la forma de un árbol cuya raíz es el programa principal)



Programación Estructurada

- El problema que tuvo esta programación, es la creación de programas mas y mas grandes y complejos.
- El mantenimiento de todos los módulos, junto con la imposibilidad de su reutilización y efectos colaterales, fue la consecuencia de la pérdida de fuerza en el horizonte de las técnicas de programación.
- De esta forma, aparece la Programación Orientada a Objetos como una evolución de la Programación Estructurada.

Programación O.O

- El primer enfoque hacia la Orientación a Objetos, se produjo a finales de los años 70. Sin embargo hasta los años 90 no tuvieron una evolución final.
- La necesidad de programas cada vez mas grandes y complejos, así como la necesidad de reutilización, ayudo a ello.
- POO surge como una evolución de la Programación Estructurada, añadiendo nuevos conceptos y formas de hacer las cosas.
- POO se basa en dividir el programa en pequeñas unidades lógicas de código (objetos)

Programación O.O

- Los objetos son unidades independientes que se comunican mediante mensajes.
- Las características de POO los podríamos resumir en:

  La Programación Orientada a Objetos ayuda a los desarrolladores a reflejar en papel el mundo real.

Programación O.O

- CARACTERISTICAS
  - Relaciona el sistema con la Vida Real.
  - Permite su aplicación a Sistemas Complejos.
  - Reutilización y Extensión del Código
  - Facilita creación de programas Visuales.
  - Agiliza el desarrollo
  - Facilita trabajo en equipo
  - Facilita Mantenimiento de Sotfware.
- Un punto a tener muy en cuenta es que POO proporciona conceptos claros y Herramientas con las cuales se modela y representa el mundo real.

Fundamentos

- La base fundamental es intentar llevar el problema real al papel mediante objetos.
- Para entender la POO necesitamos conocer una serie de conceptos claves y como interactúan entre ello.
- Conceptos como: objetos, clases, abstracción, etc, forman parte del mundo de la programación Orienta a Objetos.

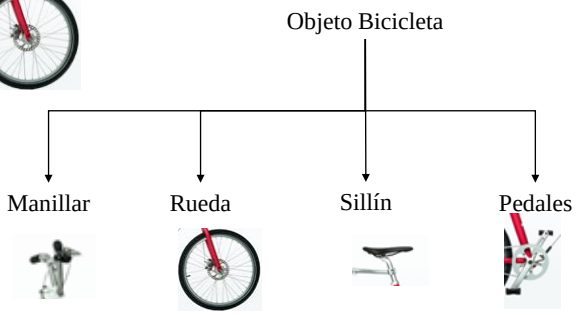
Objetos

- Son la clave de los lenguajes Orientados a Objeto.
- Su mejor definición, es definir un OBJETO en el mundo real.
- Definición Objeto Real:
  - Es cualquier cosa que vemos a nuestro alrededor (teléfono, árbol, coche, ordenador, etc)
  - En los objetos del mundo real, no necesitamos ser un experto para usarlo, ni como funciona al 100%. Por ejemplo:
    - ordenador (placa, procesador, HD, etc..)
    - árbol (tipo fruta, dame fruta, regarlo, etc…)

Objetos

- Cada objeto en el mundo real, puede estar compuesto por componentes internos (objetos) pero trabajan en conjunto y son autónomos.
- La Programación Orientada a Objetos trabaja de esa misma forma.
- Los objetos no son componentes aislados, sino que forman parte de una jerarquía o de otros objetos

Objetos



Objetos

- Todos los objetos reales tiene una serie de características y elementos que los diferencia de los demás, esto es, tienen:

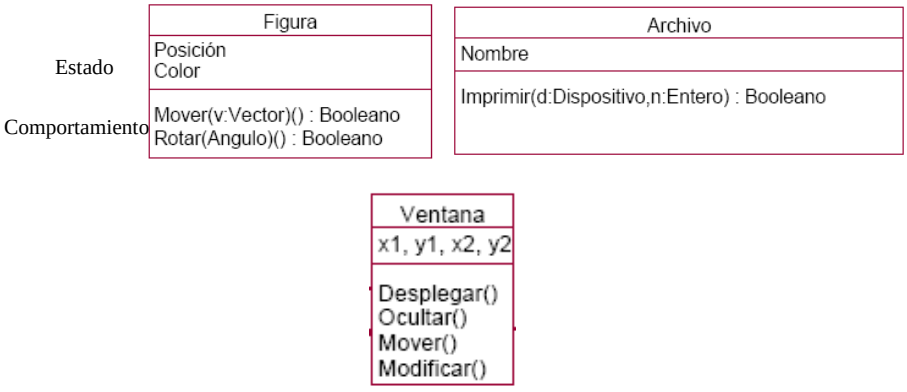
	Reales	POO
De un Coche podemos decir:	marca, modelo, color	atributos
Podemos hacer estas cosas:	frenar, adelantar	métodos

- En Programación, también se define un objeto como una instancia de una clase.

Componentes

- Los objetos están compuestos de :
  - Características (Propiedades/Estado)
    - Son los diferentes atributos (variables) que diferencian un objeto de otro
  - Comportamiento (implementación).
    - Es el código que logra el funcionamiento del objeto, métodos (también llamados funciones, subrutinas, etc)
  - Interface
    - Conjunto de métodos, propiedades, eventos y atributos que permiten la interconexión con el exterior
- La representación de los objetos se puede hacer usando diferentes METODOLOGÍAS, normalmente UML o también MLG.

Componentes



Atributos

- También conocidos como campos, son la memoria del objetos.
- Todos los objetos definen su estado con sus atributos.
- Se corresponden con las clásicas variables de cualquier programa, pero con diferencias: los atributos se están asociados al objeto (son parte de el), se pueden heredar, las variables no.
- Se definen dentro del propio objeto.
- Estos atributos pueden estar disponibles por definición propia o por herencia.

Métodos

- Los métodos son el comportamiento del objeto: fragmentos procedurales, asociados al objeto, que acceden a los datos del objeto y sólo se pueden usar a través de mensajes (llamadas).
- Los métodos en POO son sinónimos de otros elementos de otras metodologías de programación como: procedimientos, funciones, rutinas, etc.
- Los métodos pueden tener argumentos y pueden estar disponibles por definición propia o por herencia.

Clases

Al igual que ocurre en el mundo real, existen multitud de objetos del mismo tipo (coches, frigoríficos, cajeros, etc) todos ellos independientes, distintos pero con las misma base de creación.




Clases

- Una clase es un prototipo, modelo o plantilla que define los atributos y métodos comunes para poder crear objetos del mismo tipo: las instancias.
- Las clases definen las características y el comportamiento que luego las instancias utilizarán.
  - CLASES → Modelo/plantilla (diseño)
  - INTANCIAS → Casos concretos (ejecución)
- Tanto las clases como las instancias son objetos.

Clases

Clase Teléfono
Marca Memoria
Llamar() Colgar ()

Objeto 1
Nokia 1 Gb
Llamar() Colgar()

Clase Bicicleta
Material Modelo
Pedalear() Freanar()

Objeto 1
Madera Futuro
Pedalear() Frenar()

Clases

- Para poder crear un objeto, es necesario disponer de la clase de la que vamos a instanciar el objeto.
- Cuando definimos un objeto de una clase determinada, los objetos comparten su COMPORTAMIENTO (métodos) pero tendrán valores diferentes en sus PROPIEDADES (Atributos).
- En programación, crear una clase dentro de una aplicación, implica crear un nuevo tipo de dato en la aplicación.

Tipos de Operaciones

- Dentro de lo POO disponemos de una serie de operaciones que nos permiten gestionar el objeto.
- Estás operaciones serán implementadas de alguna forma dependiendo del lenguaje de programación.
- La forma de implementación es indiferente, sólo nos preocupa su funcionalidad.
  - Constructor — Crear una nueva instancia de la Clase (objeto)
  - Destructor — Destruye una instancia concreta.
  - Selector — Selecciona una parte del Estado o atributo de un objeto.
  - Modificador — Modifica una parte del Estado o atributo.
  - Clonador — Copia una instancia en otra nueva.
  - Iterador — Permite recorrer una estructura de datos (objetos).
  - Visualizador — Muestra todo el estado de un objeto de forma elaborada.

Constructores y destructores

- Los constructores dirigen la creación del objeto, inicializan o asignan valor a los atributos y realizan las operaciones necesarias para crear una instancia plenamente funcional con un estado controlado.
- Un error en el constructor cancela la creación de la instancia.
- Los destructores dirigen la desaparición del objeto, permiten controlar como se liberan los recursos del objeto cuando el sistema destruye el objeto.
- Los destructores son invocados automáticamente por el sistema y son únicos por clase.

Propiedades

- Las propiedades pertenecen a la orientación a componentes pero muchos lenguajes orientados a objetos permiten su implementación.
- Las propiedades se comportan como atributos cuando se obtiene acceso a ellas. Pero, a diferencia de los atributos, las propiedades se implementan con métodos de acceso que definen las instrucciones que se ejecutan cuando se recupera o se asigna una propiedad. Las propiedades permiten la comodidad de los atributos pero con la robustez de los métodos.
- No existe necesariamente una correspondencia uno a uno entre atributos y propiedades, aunque en la mayoría de los casos se produce, una propiedad puede ser la síntesis de varios atributos o una parte de un atributo.

Miembros de clase

- Dado que la clases son objetos pueden tener atributos y métodos de la propia clase independientemente de las instancias de la clase. Por lo tanto los miembros se clasifican en miembros de clase (o estáticos) y miembros de instancia.
- Los atributos de clase son únicos por clase, se comparten para todas las instancias de la clase y son accesibles desde cualquier método de clase o de instancia.
- Los métodos de clase solo pueden acceder a atributos y métodos de clase, dado que las instancias son múltiples para una clase o puede no existir ninguna. Los métodos de clase son accesibles a través del nombre de la clase.
- Los métodos de instancia tienen acceso a todos los miembros de clase.

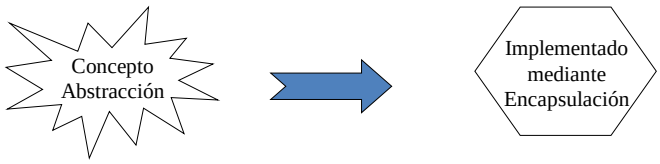
Principios de P.O.O

- Además de todo lo visto anteriormente, y según Booch: "*si alguno de los siguientes elementos no existen, se dice que el modelo no es orientado a objetos*"

- Estos elementos son:
  - Abstracción / Ocultación
  - Encapsulación
  - Modularidad
  - Jerarquía
  - Polimorfismo

Abstracción

- Consiste en captar las características esenciales de un objeto, así como su comportamiento.
- Una buena abstracción se centra en la vista externa del objeto, eliminando todos los detalles pocos importantes.
- La Abstracción es un concepto teórico, y los lenguajes Orientados a Objeto utilizan la encapsulación para aplicar este concepto.



Abstracción

- Una clase bien diseñada expone un número mínimo de métodos cuidadosamente elegidos para proporcionar el comportamiento esencial de la clase.
- Hacer buenas abstracciones no es una tarea fácil.

Encapsulación

- Es la propiedad que permite asegurar que el contenido de la información de su objeto está oculta al mundo exterior.
- La encapsulación también es conocido como OCULTACION DE INFORMACION.
- Gracias a ella, se nos permite la división de un programa en módulos.
- El encapsulamiento proporciona la abstracción y se implementa mediante clases.

Encapsulación

- Si todo lo relacionado con el objeto estuviese oculto, no se podría acceder a él; en la práctica, existen 2 partes bien diferenciadas:
  - Interface
    - Es la parte externa del Objeto
  - Implementación
    - Es el código interno.
- Gracias a la encapsulación se controla el acceso a los datos y métodos, mediante modificadores de acceso en los lenguajes O.O
  - Private, Protected, Public, etc.

Encapsulación

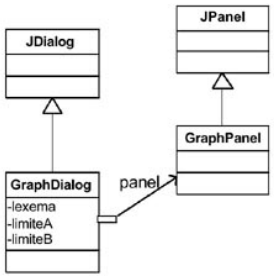
- ¿Por qué Encapsulación?
  - Es muy sencillo y responde a 2 importantes razones:
    - Al ocultar los detalles de la implementación se evita que se usen de manera no deseada y que se accedan de forma incontrolada.
    - El hecho de que cada objeto sea una “capsula” facilita que sea transportado entre diferentes Sistemas o parte de Sistemas (Reutilización)

Modificadores de acceso

- Privada:
  - Solo accesible desde los métodos de la propia clase, estrictamente.
- Protegida:
  - Accesible desde los métodos de la propia clase o de uno de sus herederos.
- Publica:
  - Accesible sin restricción desde cualquier método, propio o ajenos
- Interna, compartida, amigos, defecto:
  - Solo accesible desde los métodos de las clases del mismo paquete, ensamblado, módulo, biblioteca, …

Modularidad

- Es la propiedad que permite subdividir una aplicación en partes mas pequeñas (módulos). (*.java)
- Estos módulos deben de ser tan independientes como sea posible.
- Los módulos se pueden compilar por separado, pero normalmente tienen conexiones con otros módulos.



Modularidad

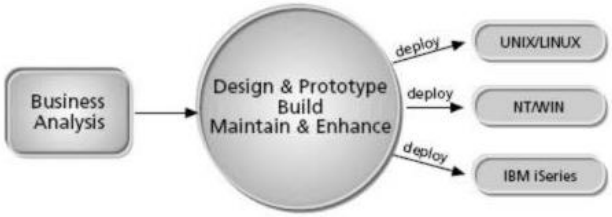
- Siempre que se habla de modularidad, se tiene asociados los conceptos de Dependencia, Acoplamiento y Coherencia.
- Dependencia:
  - Se dicen que el módulo A depende del Módulo B, si al cambiar el modulo B tenemos que cambiar el modulo A.
- Acoplamiento:
  - Define el tipo de dependencia entre módulos
    - FUERTE : Módulos con muchas dependencias
    - DEBIL: Módulos con pocas dependencias
- Coherencia:
  - Determina el tipo de Abstracción conseguida
    - FUERTE: Abstracción Buena
    - DEBIL: Abstracción Mala

Jerarquía

- Es la propiedad que permite la ordenación de las abstracciones.
- Existen 2 grandes jerarquías en P.O.O
  - Generalización/Especificación (“es una”)
    - Afecta a las estructuras de clases y también es conocido como HERENCIA
    - Relación entre CLASES
  - Agregaciones (“parte de”)
    - Afecta a la estructuras de Objetos
    - Permite el agrupamiento físico de objetos relacionados. (camión, ruedas, motor, ….)

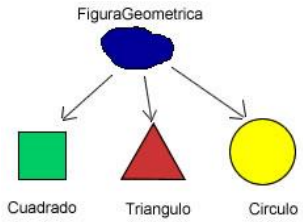
Polimorfismo

- Es la propiedad que indica la posibilidad de referirse a objetos de clases diferentes mediante el mismo elemento de programa (método), haciendo implementaciones diferentes.
- Por ejemplo:
  - Los mamíferos comen pero no todos de la misma forma.
  - Las piezas del ajedrez se mueven pero no igual.

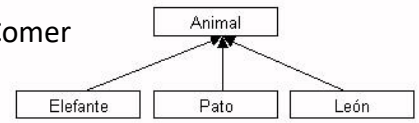


Polimorfismo

- Figuras geométricas. Área

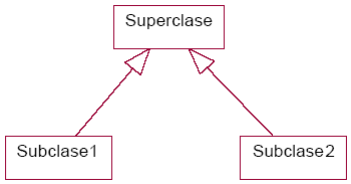


- Animales. Formas de Comer



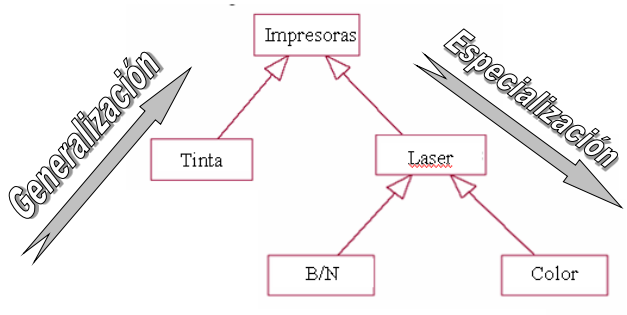
Concepto de Herencia

- La Herencia es un método para compartir similitudes entre clases donde los atributos y métodos de varias clases son comunes. Permite la reutilización de la definición de las clases.
- La clase superior es denominada, CLASE BASE o SUPERCLASE y de ella heredarán los atributos y métodos.
- La clase inferior es denominado CLASE DERIVADA o HIJA y ésta heredará atributos o métodos de la Clase BASE.



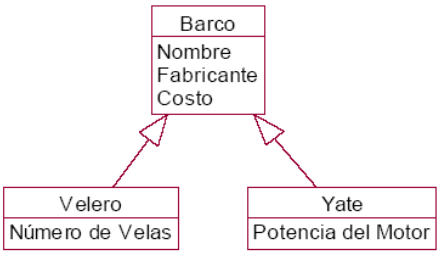
Concepto de Herencia

- La Herencia se utiliza para hacer una Jerarquía de clases.
- Según se va bajando en la jerarquía, se va Especializando, mientras que si subimos, vamos generalizando



Concepto de Herencia

- Cuando creamos un objeto de una clase Derivada, el objeto contiene toda la información de ese clase y también de la clase BASE.



Si creamos un objeto de Yate,
en él dispondremos de los atributos
de BARCO
Nombre, Fabricante, Costo

+ los atributos de Yate

Potencia del Motor

Concepto de Herencia

- CARACTERISTICAS de la HERENCIA
  - Los valores de una instancia derivada incluyen los valores de la/s clase/s base.
  - Cualquier operación de la clase base se puede aplicar a la clase derivada.
  - Las clases derivadas pueden redefinir las clases bases añadiendo atributos o métodos (extensión de la clase).
  - Las clases derivadas no pueden restringir los atributos y métodos heredados.

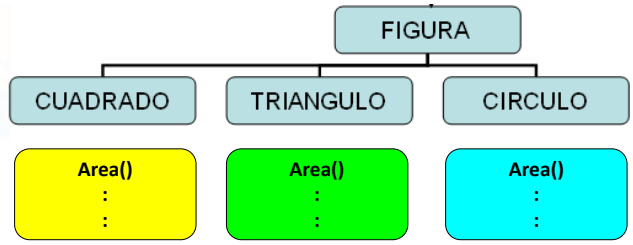
Sobrescritura de Métodos

- Sobrescritura: Es la posibilidad de que una clase derivada pueda sobrescribir valores de los atributos heredados y la implementación de los métodos.
- La sobrescritura de métodos permite la personalización del heredero respetando el interfaz heredado.
- Gracias a esta sobrescritura de métodos, podemos implementar el concepto de POLIMORFISMO.

Sobrescritura de Métodos

- Esta sobrescritura puede producirse por varias razones:
  - EXTENSION
    - Una nueva operación es igual a la heredada, excepto que agrega algún comportamiento nuevo.
  - RESTRICCION
    - Restringe el funcionamiento de lo heredado o el tipo de datos heredado para la operación.
  - OPTIMIZACION
    - Para aprovechar atributos y funcionalidades nuevas creadas con ese fin

Sobrescritura de Métodos



Podríamos disponer de un método a nivel de la Clase BASE llamado área y en cada uno de las subclases lo deberíamos sobrescribir para devolver el área correcta del objeto creado

Clases Abstractas

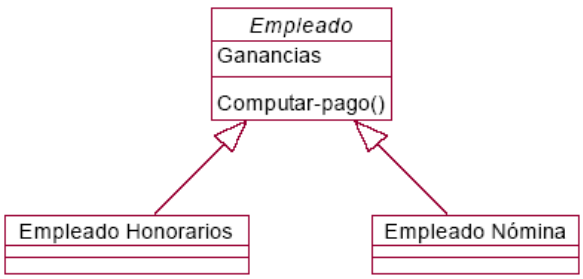
- Hasta ahora todas las clases descritas se conocen como Clases CONCRETAS.
- Cualquiera de estas clases son instanciables, es decir, se pueden crear objetos de ellas en cualquier momento.
- Por el contrario, una CLASE ABSTRACTA, es una clase que no tiene directamente instanciación y por lo tanto no puede tener objetos.
- Las clases Abstractas son un recurso para la HERENCIA.

Clases Abstractas

- Las clases Derivadas de ellas, heredarán, como otras cualquiera, y podrán instanciarse.
- Las clases Abstractas pueden:
  - Definir atributos normales
  - Definir e implementar métodos normales
  - Definir métodos ABSTRACTOS
- Para que la clase Derivada sea instanciable, no abstracta, es necesario que en ella se sobrescriban los métodos declarados como Abstractos.

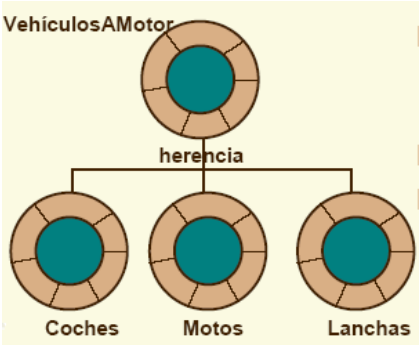
Clases Abstractas

- EJEMPLOS:



Clases Abstractas

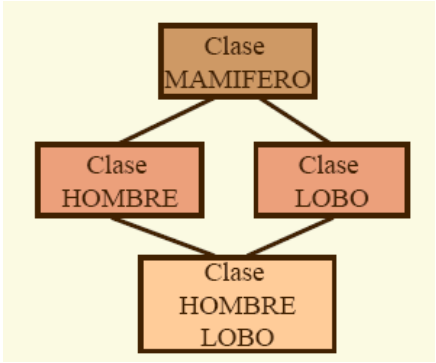
- EJEMPLOS:



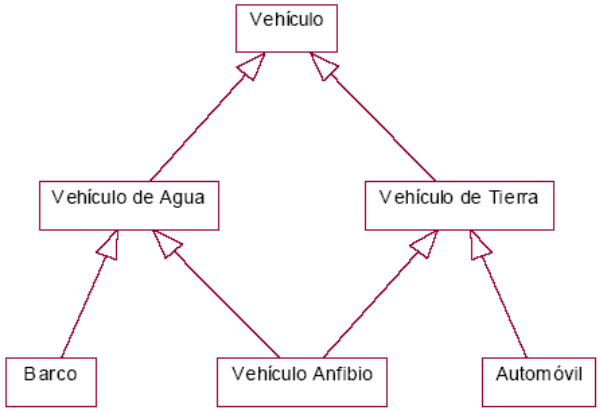
Herencia múltiple

- La Herencia múltiple permite a una clase Derivada tener más de una clase BASE y heredar de ambas.
- Es una característica que no todos los lenguajes la implementan (C++ si, Java no)
- Este tipo de herencia puede tener efectos colaterales, por ejemplo cuando se heredan métodos llamados de forma idéntica. Para evitar conflictos, deberíamos de sobrescribirlo.

Herencia múltiple



Herencia múltiple



Herencia múltiple

- La ventaja que proporciona la Herencia Múltiple son:
  - Incorporación e información en una sola operación de Herencia.
  - Oportunidad de reutilizar clases.
- Antes de intentar utilizar esta características debemos estar seguro que es soportada.
  - Lenguajes que la soportan:
    - C++, Centura SQL Windows, CLOS, Eiffel, Object REXX, Perl y Python
  - Lenguajes que NO la soportan:
    - Java, Nemerle, C# y Objective-C

Sobrecarga

- La Firma del método está formada por el número, orden y tipos de los parámetros.
- La sobrecarga permite que una clase tenga varios métodos con el mismo nombre pero con firmas diferentes.
- Los tipos de los argumentos pasados en la invocación del método permite resolver que método se quiere ejecutar.
- El tipo de retorno del método no forma parte de su firma y, por lo tanto, no sirve para diferenciar métodos.
- La sobrecarga permite tener varias versiones del mismo método facilitando su reutilización.

Interfaces

- Un interfaz define explícitamente un grupo de funcionalidades relacionadas, un mecanismo o base de conocimiento común de comunicaciones con los objetos.
- Los interface permiten crear nuevos tipos de datos, pero son tipos abstractos (no instanciables), solo declaran los métodos sin implementar.
- La clase que implementa un interface debe definir todos los métodos declarados en el interface, salvo que sean abstractas en cullo caso pueden delegar la implementación en sus herederos.
- Una clase puede implementar más de un interface.
- Un interface puede reutilizar la definición de uno o varios interfaces: heredar o, mas apropiadamente, sintetizar.

Referencias

- Las referencias son variables o atributos que pueden contener instancias.
- Las referencias se declaran de tipo clase o interfaz.
  - Una referencia de tipo clase puede contener instancias de la clase del tipo o de cualquiera de sus herederos.
  - Una referencia de tipo interfaz puede contener instancias de cualquier clase o heredero que implemente el interfaz.
- La referencia solo conoce los métodos del tipo definido, aunque la instancia tenga funcionalidad adicional no es accesible directamente.
- La instancia de una referencia puede recuperar su tipo original pero hay que indicarlo de forma explicita y requiere comprobación previa.

Vinculación dinámica

- En la orientación a objetos se pueden tener métodos diferentes con el mismo nombre pero no se debe confundir:
  - Sobrecargar: en la misma clase, significa definir nuevos métodos con firmas diferentes.
  - Sobrescribir: implica herencia, significa ocultar un método heredado con una nueva definición del método con la misma firma.
  - Polimorfismo: los métodos con la misma firma pertenecen a clases diferentes.
- La Vinculación dinámica es el mecanismo utilizado para averiguar, en tiempo de ejecución, a qué método debe llamar, a partir del tipo del objeto asignado a una referencia y los argumentos suministrados.

Genéricos

- Los lenguajes orientados a objetos son fuertemente tipados, requieren un tratamiento estricto de tipos para poder resolver la sobrecarga, el polimorfismo, … Esto provoca una inflexibilidad que debe resolverse con herencia, interfaces, …
- Los genéricos introducen el concepto de parámetros de tipo, lo que le permite diseñar clases y métodos que aplazan la especificación de uno o varios tipos hasta que el código de cliente declare y cree una instancia de la clase o el método.
- Los genéricos implementan el concepto de plantillas de clases, igual que una clase permite implementar múltiples instancias, un genérico permite definir múltiples clase.
- Las clases y métodos genéricos combinan reusabilidad, seguridad de tipos y eficacia de una manera en que sus homólogos no genéricos no pueden. Los genéricos se usan frecuentemente con colecciones y los métodos que funcionan en ellas.

Genéricos

```
public class MiLista< T > {
   private List< T > lista;
   private T elementoActual;
   public void add(T item) {
  …
}

MiLista<String> cadenas;
MiLista<Integer> enteros;
```

Excepciones

- Las excepciones son situaciones anómalas que pueden ocurrir durante la ejecución de las aplicaciones que impiden el correcto funcionamiento de las mismas.
- La mayoría de los lenguajes orientados a objetos gestionan los errores a través del tratamiento de excepciones.
- El tratamiento de las excepciones es obligatorio: cuando se produce una excepción se “rompe el código”, el flujo de control salta al primer controlador de excepciones asociado. En caso de no encontrarlo, la excepción se propaga por la pila de llamadas rompiendo el código hasta encontrar el controlador de excepciones apropiado. Si no hay ningún controlador de excepciones para una excepción determinada, el programa deja de ejecutarse con un mensaje de error.
- Trabajar con excepciones implica no solo su tratamiento, también es necesario crearlas y lanzarlas. Las excepciones son instancias de las clases apropiadas que representan la situación anómala. La definición de excepciones propias se hace a través de la herencia de clases.
- En lenguajes como el Java las excepciones son declarativas.

S.O.L.I.D.

- SOLID es el acrónimo que acuñó Michael Feathers, basándose en los 5 principios de la programación orientada a objetos que Robert C. Martin había recopilado en el año 2000 en su articulo “Design Principles and Design Patterns”.
- Los objetivos de estos 5 principios a la hora de escribir código son:
  - Crear un software eficaz: que cumpla con su cometido y que sea robusto y estable.
  - Escribir un código limpio y flexible ante los cambios: que se pueda modificar fácilmente según necesidad, que sea reutilizable y mantenible.
  - Permitir la escalabilidad: que acepte ser ampliado con nuevas funcionalidades de manera ágil.
- La aplicación de los principios SOLID está muy relacionada con la comprensión y el uso de patrones de diseño, que permitirán minimizar el acoplamiento (grado de interdependencia que tienen dos unidades de software entre sí) y maximizar la cohesión (grado en que elementos diferentes de un sistema permanecen unidos para alcanzar un mejor resultado que si trabajaran por separado).

S.O.L.I.D.

S	• Single Responsibility Principle (SRP) • Principio de responsabilidad única
O	• Open/Closed Principle (OCP) • Principio de abierto-cerrado
L	• Liskov Substitution Principle (LSP) • Principio de sustitución de Liskov
I	• Interface Segregation Principle (ISP) • Principio de segregación de interfaces
D	• Dependency Inversion Principle (DIP) • Principio de inversión de dependencias

S.O.L.I.D.

- Principio de Responsabilidad Única

  "A class should have one, and only one, reason to change."

  – La S del acrónimo del que hablamos hoy se refiere a Single Responsibility Principle (SRP). Según este principio "una clase debería tener una, y solo una, razón para cambiar". Es esto, precisamente, "razón para cambiar", lo que Robert C. Martin identifica como "responsabilidad".

  – El principio de Responsabilidad Única es el más importante y fundamental de SOLID, muy sencillo de explicar, pero el más difícil de seguir en la práctica.

  – El propio Bob resume cómo hacerlo: "Reúne las cosas que cambian por las mismas razones. Separa aquellas que cambian por razones diferentes".

S.O.L.I.D.

- Principio de Abierto/Cerrado

  "You should be able to extend a classes behavior, without modifying it."

  – El segundo principio de SOLID lo formuló Bertrand Meyer en 1988 en su libro "Object Oriented Software Construction" y dice: "Deberías ser capaz de extender el comportamiento de una clase, sin modificarla". En otras palabras: las clases que usas deberían estar abiertas para poder extenderse y cerradas para modificarse.

  – El principio Open/Closed se suele resolver utilizando polimorfismo.

  – Es importante tener en cuenta el Open/Closed Principle (OCP) a la hora de desarrollar clases, librerías, frameworks, microservicios.

S.O.L.I.D.

- Principio de Sustitución de Liskov

  "Base classes must be substitutable for their derived classes."

  – La L de SOLID alude al apellido de quien lo creó, Barbara Liskov, y dice que "las clases base deben poder sustituirse por sus clases derivadas".

  – Esto significa que los objetos deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del sistema o lo que es lo mismo: si en un programa utilizamos cierta clase, deberíamos poder usar cualquiera de sus subclases sin interferir en la funcionalidad del programa.

  – Según Robert C. Martin incumplir el Liskov Substitution Principle (LSP) implica violar también el principio de Abierto/Cerrado.

S.O.L.I.D.

- Principio de Segregación de la Interfaz

  "Make fine grained interfaces that are client specific."

  – En el cuarto principio de SOLID, el tío Bob sugiere: "Haz interfaces que sean específicas para un tipo de cliente", es decir, para una finalidad concreta.

  – En este sentido, según el Interface Segregation Principle (ISP), es preferible contar con muchas interfaces especializadas que definan unos pocos métodos que tener una interface generalista que fuerce a implementar muchos métodos a los que no se dará uso.

S.O.L.I.D.

- Principio de Inversión de Dependencias

  "Depend on abstractions, not on concretions."
  - Llegamos al último principio: "Depende de abstracciones, no de clases concretas".
  - Así, Robert C. Martin recomienda:
    - Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
    - Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.
  - El objetivo del Dependency Inversion Principle (DIP) consiste en reducir las dependencias entre los módulos del código, es decir, alcanzar un bajo acoplamiento de las clases. Las abstracciones se logran mediante los interfaces.

Cómo encontrar las clases

- Uno de los sistemas más sencillos de los que dispones para identificar las clases y sus componentes es el análisis gramatical de las especificaciones.
- Según vas leyendo las especificaciones puedes marcar los sustantivos, los verbos y los adjetivos:
  - Los **sustantivos** indican la existencia de un objeto. Dicho objeto es susceptible de convertirse en una clase. Tienes que tener cuidado con los sinónimos, dado que hacen referencia al mismo objeto, si los representas te encontrarás con objetos duplicados, y la polisemia, con el mismo término hacen referencia diferentes conceptos, en cuyo caso faltaran objetos.
  - Los **verbos** denotan acciones. Las acciones son los métodos. Por lo tanto, cuando encuentras verbos, estás identificando los métodos de las clases.
  - Los **adjetivos** y, en muchos casos los **adverbios**, muestran características propias de los objetos, lo que hemos denominado atributos.
- Mediante este sistema tan simple, puedes realizar la primera aproximación al modelo de datos. En refinamientos sucesivos, eliminas los objetos que no son significativos, quitas los elementos duplicados, incorporas los elementos que faltan, ...