

Programación Orientada a Objetos

Escuela Colombiana de Ingeniería Julio Garavito

2020 –I

Palabras reservadas

Palabra	Nivel	Objetivo
Extends	Clase	Herencia
Final	Clase	Nivel máximo de especialización
Final	Método	No se permite sobreescritura
Abstract	Clase	No se puede instanciar
Abstract	Método	Las subclases tienen que tener implementado el método
Interface	Clase	Es abstracta y establecen la forma que debe tener una clase. Define qué y no el cómo *.
Implements	Clase	Una clase implementa una interfaz
Throw	Método	Lanzar excepción
Throws	Método	Propagar excepción
Try / Catch	Método	Controlar excepción

Clase Abstracta vs. Interfaz

Clase Abstracta

- Deben tener constructor.
- Puede extender de cualquier clase (así la superclase no sea abstracta)
- Puede tener métodos abstractos o no.
- Los métodos pueden ser *public*, *protected* o *private*.
- Pueden existir variables y constantes. Estáticas o no.
- **Palabra reservada:** *Abstract*

- Notación UML:

ClassA

Interfaz

- No tienen constructor.
- No puede extender de cualquier clase: La superclase debe ser una interfaz también.
- Todos sus métodos son abstractos.
- Los métodos pueden ser *public*.
- Existen constantes (*final*) y son estáticas (*static*).
- **Palabra reservada:** *Interface*

- Notación UML:

**<<interface>>
Interface**

Herencia vs. Interfaz

Herencia

- Mecanismo para extender funcionalidades y atributos de una clase
- La superclase **puede** ser abstracta.
- **Palabra reservada subclases:** *Extends*
- En JAVA no se permite herencia múltiple, es decir, una clase herede de varias superclases

 ClaseC *extends* ClaseA, ClaseB

Interfaz

- Lista de funcionalidades que pueden implementar uno o varios objetos.
- Describe el qué (métodos / funciones). No describe el cómo (los métodos/funciones son vacías). Como una plantilla.
- **Palabra reservada en la interfaz:** *Interface*
- **Palabra reservada en las clases que implementan:** *Implements*
- Una clase puede implementar más de una interfaz

 ClaseC *implements* InterfaceA, InterfaceB

Herencia vs. Interfaz


Herencia

- Tanto las superclases como subclases debe tener constructor.
- Las subclases debe tener constructor.
- La superclase puede tener métodos abstractos.

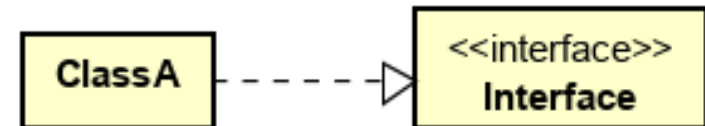
- Notación UML:



Interfaz

- Una interfaz puede heredar de otra.
 InterfaceD **extends** InterfaceB
- No tiene constructor.
- Todos sus métodos son abstractos.

- Notación UML:



Principios SOLID

PRINCIPIOS DE DISEÑO DE SOFTWARE

S Single Responsibility Principle

O Open / Closed Principle

L Liskov Substitution Principle

I Interface Segregation Principle

D Dependency Inversion Principle

Ventajas

- Permite tener un software estable y robusto
- Permite tener un código limpio
- Facilita gestionar los cambios modificando el código (Mantenibilidad)
- Código reutilizable
- Escalabilidad: Ampliar funcionalidades de manera más ágil

S

Single Responsibility Principle >> Principio de Responsabilidad Única

- Cada clase debe tener una única responsabilidad.
- Destinar a cada clase una finalidad (responsabilidad) sencilla y concreta

*" A class should have **one, and only one, reason to change.**" /
"Una clase debería tener **una, y solo una, razón para cambiar**"*

O

Open / Closed Principle >> Principio Abierto / Cerrado

- Extender el comportamiento de una clase sin modificarla.
- **Abierta / Cerrada:** Una clase debe estar abierta para poder extenderse y cerrada para modificarse.
- **Ejemplo:** Herencia, Interfaz o sobreescritura de métodos.
- Evita los IF / ELSE

*" You should be able to **extend a classes behavior, without modifying it.**" /
"Deberías estar en capacidad de **extender el comportamiento de una clase, sin modificarla**"*



Liskov Substitution Principle >> Principio de Sustitución de Liskov

- Una clase que hereda de otra puede usarse como su superclase (padre) sin modificar la funcionalidad del programa.
- En algunos casos, usar métodos como `InstanceOf` para definir cómo debe comportarse un objeto, violan este principio.

*" Derived classes **must be substitutable for their base classes.**" /
"Las subclases **deben ser sustituidas por sus clases bases**"*



Interface Segregation Principle >> Principio Segregación de la Interfaz

- Crear interfaces para una finalidad concreta.
- Es una mejor práctica tener varias interfaces con pocos métodos que mantener una interfaz con diferentes funcionalidades que no aportan valor a todas las clases que la implementan.
- Las clases deben implementar interfaces que contenga métodos que realmente va a utilizar.

*" Make fine grained **interfaces that are client specific.**" /
"Realizar **interfaces que sean específicas para cada tipo de cliente**"*

D Dependency Inversion Principle >> Principio de Inversión de dependencias

- Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones
- Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.
- **Ejemplo:** Relacionar las clases con la superclase y no con cada subclase

*"Depend on **abstractions**, not on **concretions**." /
"Depende de **abstracciones**, no de **clases concretas**"*

Referencias

- Los 5 principios SOLID. Ver: <https://medium.com/@ger86/los-5-principios-solid-68d697984abd>
- S.O.L.I.D: Object Oriented Desing. Ver: <https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design>
- SOLID. Ver: <https://es.wikipedia.org/wiki/SOLID>
- Solid, cinco principios básicos de diseño de clases. Ver: <https://www.genbeta.com/desarrollo/solid-cinco-principios-basicos-de-diseno-de-clases>
- SOLID: los 5 principios que te ayudarán a desarrollar software de calidad. Ver: <https://profile.es/blog/principios-solid-desarrollo-software-calidad/>