**Design patterns and SOLID principles**

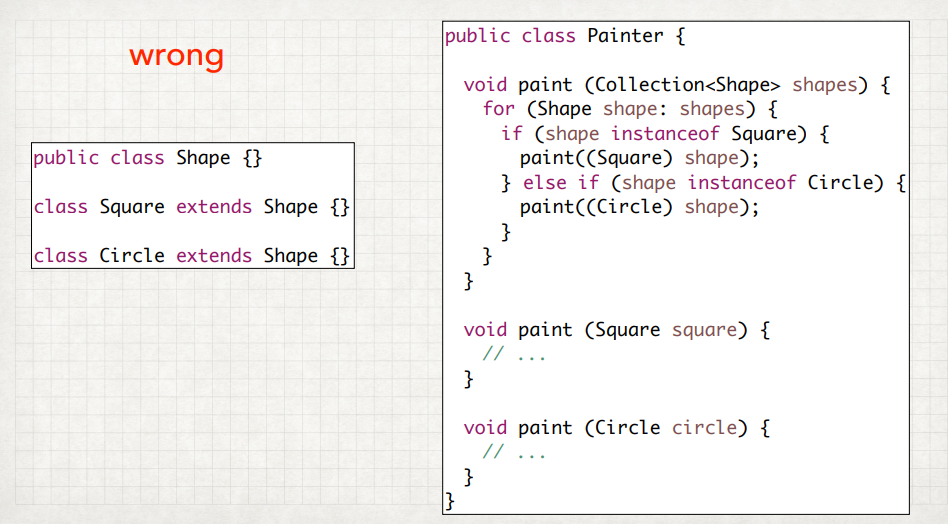
# SOLID principles

## Single Responsibility Principle

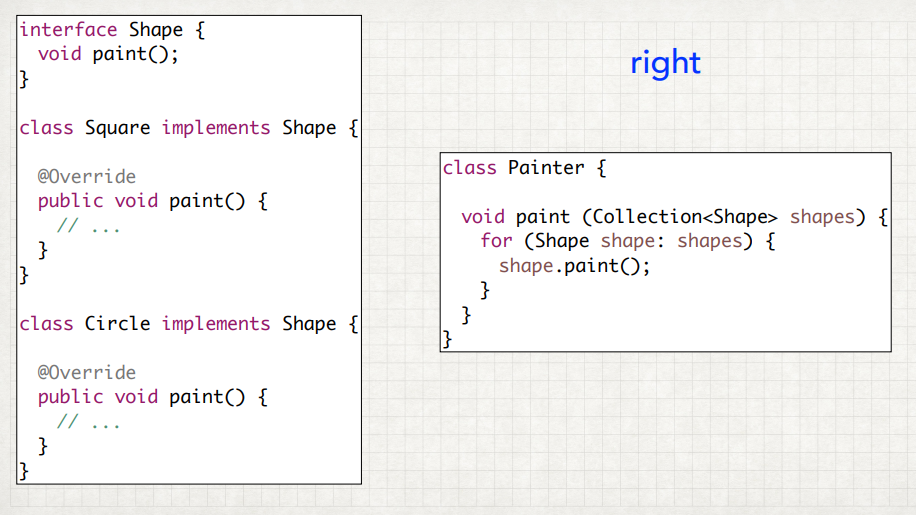
* Cohesion
  + Degree to which the elements inside a module belong together
  + Low cohesion: if a module groups various functionalities
  + High cohesion:
    - if a module has a single responsibility
    - is desirable to improve features of our code such as maintainability and reusability.
  + A class should only have one reason to change.
    - A frequent case of violation of this principle is when we mix business logic with presentation logic in the same method. In this scenario the method may change when a rule associated with the business changes or when we want to present the data to a customer in a different way.

## Open Closed Principle

* Is our design ready to be easily extended ?
* Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.
  + Our system must be extensible however we need it to be, but that extension must be done without modifying the existing code. Changing the existing code would mean we would introduce errors and we need to repeat all the tests we’ve already carried out.
  + If our system can be extended , but at the cost of having to modify the original code, that does not comply the Open/Closed Principle.



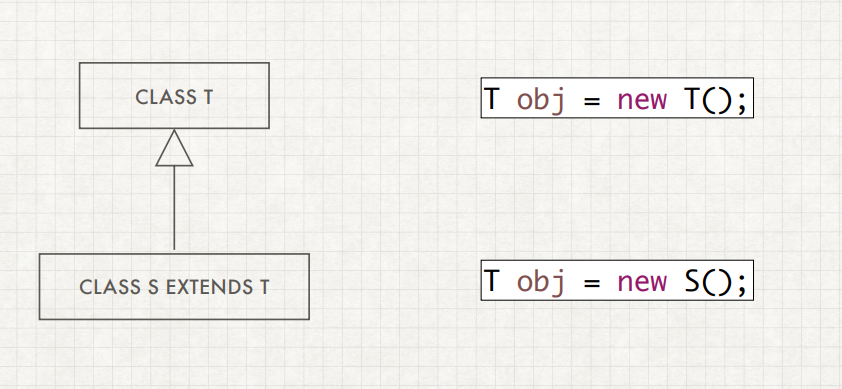
* If we want to add a new shape we need to create the type as a class that extends the shape and must do that without modifying the current code. This isn’t the problem, but we also have to modify the paint method from the Painter class. This ex exactly what the Open/Closed principle warns us not to do.

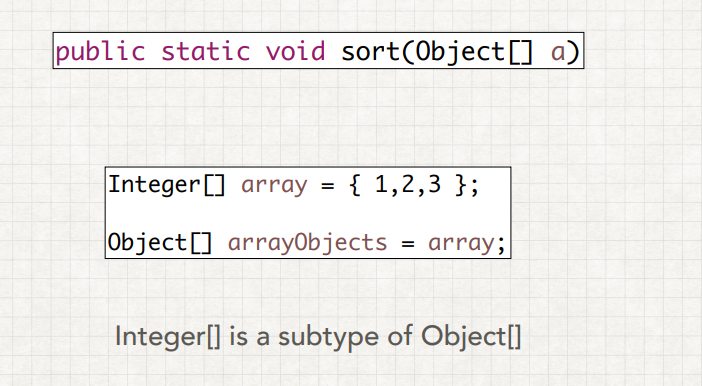


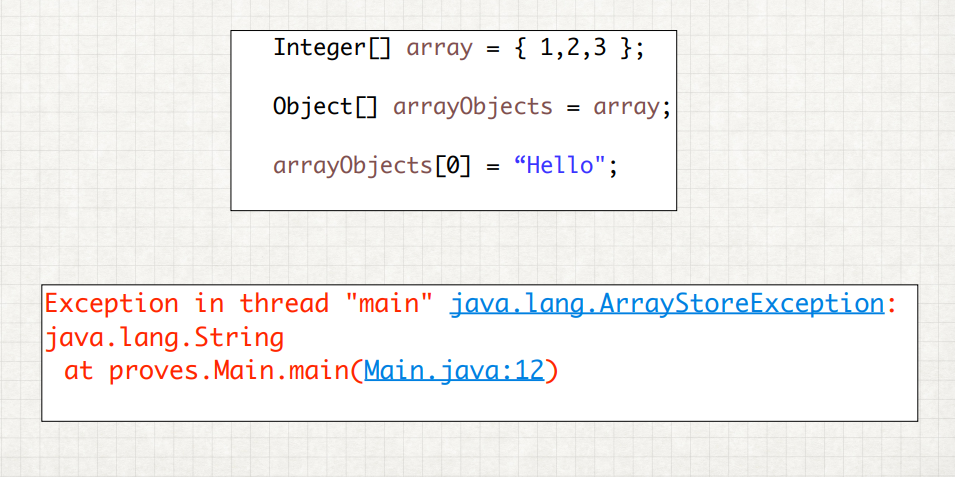
* The implementation of each shape should not be within the Painter class, this is a mistake. It should be in each of the shapes. Converting the shape to an interface and declaring the abstract method Paint there. Therefore each class implements it and we have the ability to add new shapes without having to modify the code we’ve already written.
* Keeping the Open/Closed principle in mind during our design forces us to think about which part of out code is extensible and explicitly represented in our code. This principle is an excellent tool to produce more solid and maintainable designs.

## Liskov Substitution Principle

* Liskov questions under what conditions the substitution of one type for another can alter the correct execution of a program.
* If we have class T and we create a subclass S which extends T by virtue of polymorphism, objects of type S can be seen as object of the more general type T. Thererfore a program initially written assigning the variable of type T, could be substituted without changing the object declaration by assigning it type S which is more specific.
* Liskov questions: what requirements does type S have to meet for this change to work without any errors occurring ?
  + Taking into account the importance of inheritance and polymorphism it is clearly a very important consideration.
  + We could summarize that a necessary requirement of subclass S, so that it genuinely is a subtype of T and then can be used in place of T, is that all properties that the program required of T are also required of S.



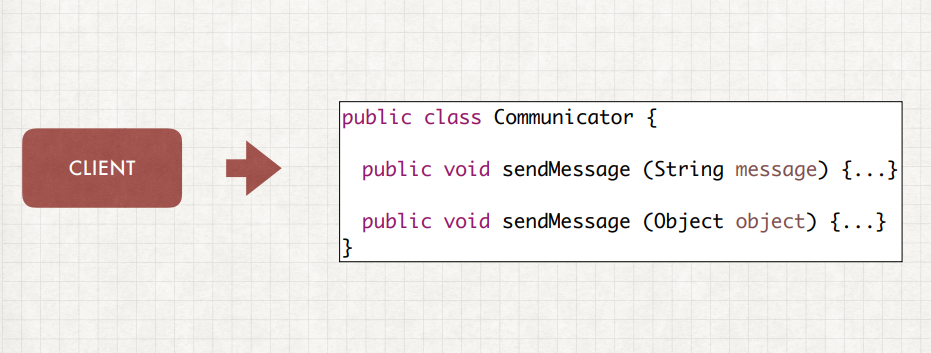


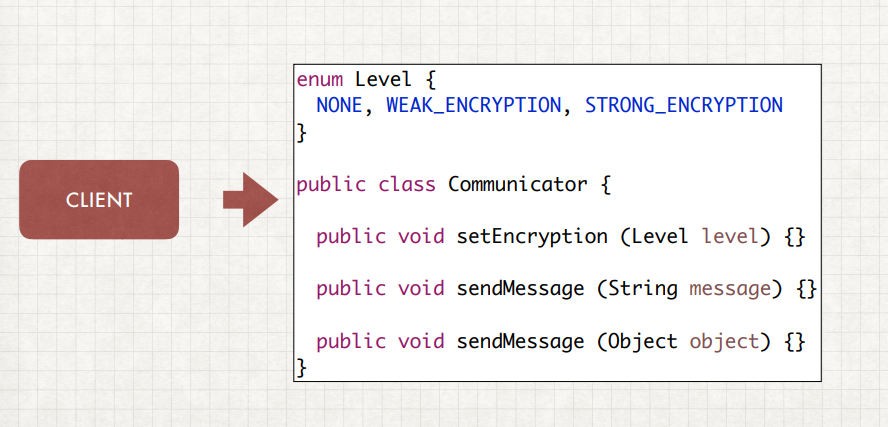


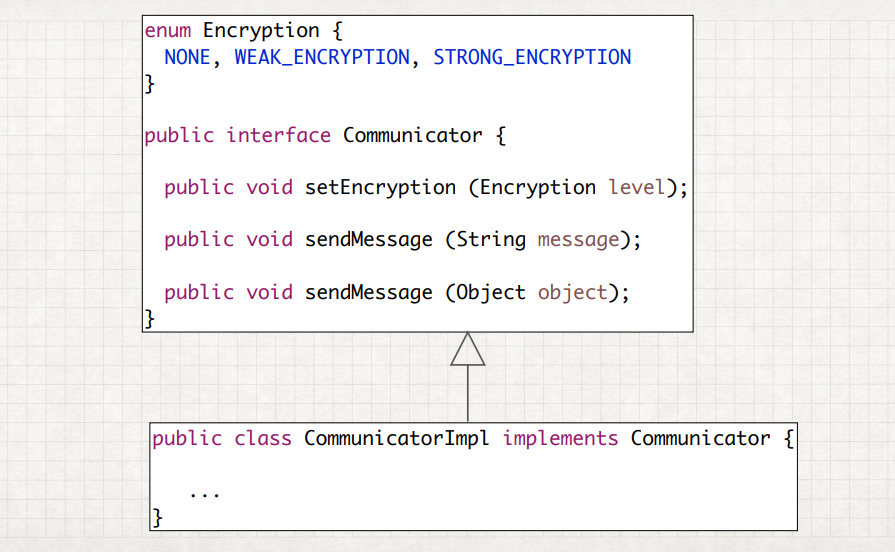
* The principle defines that object of a superclass shall be replaceable with object of its subclasses without breaking the application. That requires the objects of you subclasses to behave in the same way as the objects of your superclass.
* An overridden methods of a subclass needs to accepts the same input parameter values as the method of the superclass. That means you can implement less restrictive validation rules, but you are not allowed to enforce stricter ones in your subclass
* The return value if a method of the subclass needs to comply with the same rules as the return value of the method of the superclass.

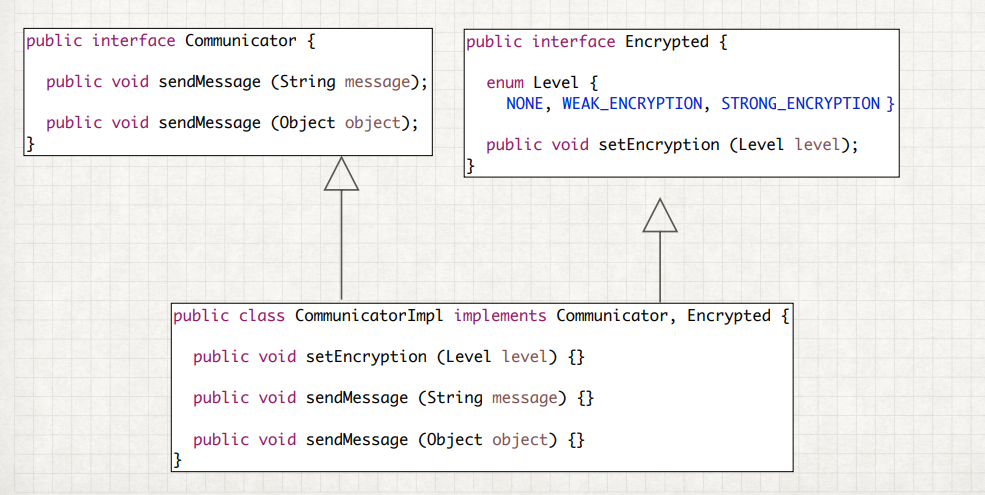
## Interface Segregation Principle

* To avoid forcing client to deal with methods not if interest to them.
* Client should not be forces to depend on methods they do not use
* A tool to obtain ‘high cohesion’ in the interface that our clients use.



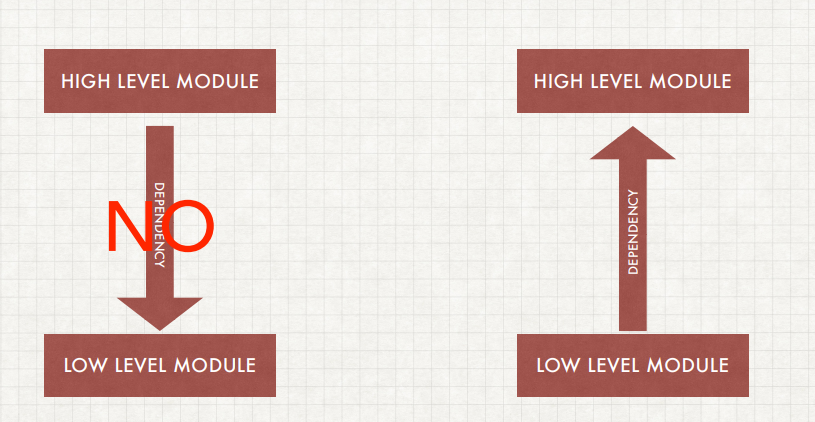


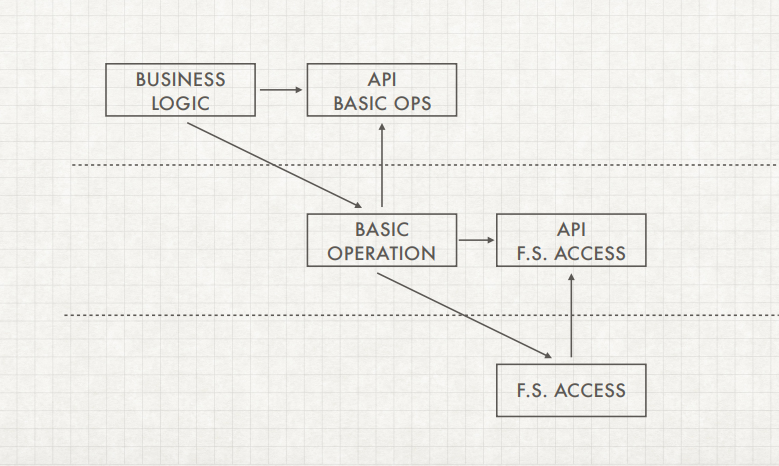




## Dependency Inversion Principle

* The use of the term ‘inversion’ is used because the principle aim is to have an inverse relationship to what we traditionally occur. In a traditional analysis the normal method is to break down tasks into a series of much simpler subtasks and so forth, until each level of detail becomes manageable.
* When this form of development is used, this ‘subtask’ modules that are created have a dependency on the higher-level modules, as opposed to the lover-level, more detailed modules. As if it’s written backwards.
  + High-level modules should not depend on low-level modules. Both should depend on abstractions.
  + Abstraction should not depend upon details. details should depend upon abstractions.





Dependecy injection: is a technique whereby one object (or static method) supplies the dependencies of another object. A dependency is an object that can be used (a [service](https://en.wikipedia.org/wiki/Service_(systems_architecture))).

one class depends on another 🡪 constructor

When class A uses some functionality of class B, then its said that class A has a dependency of class B.

Three types of Dependency injection:

1. constructor injection: the dependencies are provided through a class constructor.
2. setter injection: the client exposes a setter method that the injector uses to inject the dependency.
3. interface injection: the dependency provides an injector method that will inject the dependency into any client passed to it. Clients must implement an interface that exposes a setter method that accepts the dependency.

Abstracte class 🡪 kiezen welke methodes kunnen overschreven worden

Abstract: no body 🡪 must be overridden

Virtual: needs body and can be overridden 🡪 generic crud operations with include

Encapsulation: Is one of the fundamentals of OOP (object-oriented programming). It refers to the bundling of data with the methods that operate on that data. Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties’ direct access to them. Publicly accessible methods are generally provided in the class (so-called getters and setters) to access the values, and other client classes call these methods to retrieve and modify the values within the object.