# Overview:

* The decorator pattern will allow you to attach additional responsibilities to an object dynamically:
  + Allows a user to add new functionality to an existing object without altering its structure.
* Decorators provide a flexible alternative to sub-classing for extending functionality.
* The main principle of this pattern says that we cannot modify existing functionalities but we can extend them:
  + Open for extension but closed for modification.
* The core concept applies when we want to add some specific functionalities to some specific object instead of the whole class.
* The decorator is used to modify the functionality of an object at runtime.
  + Other instances of the same class will not be affected by this, so individual object gets the modified behavior.

# Why not use inheritance?

* With the decorator, we want to add responsibilities to individual objects, not an entire class.
* One way to add responsibilities is with inheritance:
  + We need to create a new class for new responsibilities.
  + There will be many classes inside the system, which in turn can make the system complex.
* Inheriting a border from another class puts a border around every subclass instance.
  + Inflexible because the choice of border is made statically.
  + A client cannot control how and when to decorate the component with a border.
* A more flexible approach is to enclose the component in another object that adds the border:
  + The enclosing object is called the decorator.
  + The decorator conforms to the interface of the component it decorates.
* We can add or remove responsibilities by simply attaching or detaching decorators.

# Examples:

* Suppose you already own a house:
  + You want to add an additional floor.
  + You do not want to change the architecture of ground floor (or existing floors).
* You want to change the design of the architecture for the newly added floor without affecting the existing architecture for existing floors.
  + Use a Decorator.
* Many object-oriented user interface toolkits use decorators to add graphical additions to widgets:
  + You want to add properties like borders or behaviors like scrolling to any user interface component
  + Do not want to change the core functionality of the graphical user interface.
* Suppose we have a TextView object that displays text in a window:
  + TextView has no scroll bars by default because we might not always need them.
  + When we need scroll bars, we can use a ScrollDecorator to add them.
* Streams are a fundamental abstraction in most I/O facilities.
  + Can provide an interface for converting objects into a sequence of bytes of characters.
  + Let us transcribe an object to a file or to a string in memory for retrieval later.
* The large number of classes in the java.io package is largely based on using a decorator:
  + Use decorators to add functionality for reading/writing data from/to a file.
* For example, BufferedInputStream and LineNumberInputStream both extend FilterInputStream.
  + FilterInputStream acts as an abstract decorator class.

# Advantages and drawbacks:

* The biggest advantage of using this pattern is that we can add new functionality to a particular object without disturbing existing objects in the system.
* More flexibility than static inheritance
  + Responsibilities can be added and removed at run-time simply by attaching and detaching them.
* Decorators also make it easy to add a property twice:
  + To give a TextView a double border, simply attach two BorderDecorators.
* We can code incrementally:
  + We make a simple class first and then one by one we can add decorator objects to them as needed.
  + We do not need to take care of each possible scenario in the beginning.
* One drawback may be that designs using this pattern often result in a large number of small classes that can be overwhelming:
  + Many little objects can be hard to learn and debug.

# Decorator as a structural pattern?

* Many developers believe that the decorator pattern should be classified as a behavioral pattern.
* Structural patterns describe how classes and objects are composed to create new structures or new functionality.
* The Decorator pattern allows you to compose objects by wrapping one object with another to provide new functionality.
* The focus is on how you compose the objects dynamically to gain functionality.
* The focus is not on the communication and interconnection between objects (behavioral patterns).

# When to use a Decorator:

* When you want to add responsibilities to individual objects dynamically and transparently without affecting other objects.
* When extension by sub-classing is impractical.
* Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination.
* Alternatively, a class definition may be hidden or otherwise unavailable for sub-classing.

# Summary

* Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in a design.
* We should allow behavior to be extended without the need to modify existing code.
* The Decorator pattern provides an alternative to sub-classing for extending behavior.
* A way to add additional behavior to an existing class dynamically.
* Decorators can result in many small objects in a design, and overuse can make your design more complex.

# Implementation (Overview):

* The Decorator pattern will create a set of decorator classes that are used to wrap concrete components:
  + Provides additional functionality keeping class methods signature intact.
* Decorator classes mirror the type of the components they decorate.
  + They are the same type as the components they decorate.
* Decorators change the behavior of their components by adding new functionality before and/or after method calls to the component.
* You can wrap a component with any number of decorators.
* Decorators are typically transparent to the client of the component:
  + Unless the client is relying on the component’s concrete type.

# Participants:

* **Component:**
  + Defines the interface for objects.
  + Can have responsibilities added to them dynamical.
* **ConcreteComponent**:
  + Defines an object to which additional responsibilities can be attached.
* **Decorator:**
  + Maintains a reference to a Component object.
* **ConcreteDecorator**:
  + Adds responsibilities to the component.
* The decorator forwards requests to its Component object
  + May optionally perform additional operations before and after forwarding the request.