Design patterns are elegant solutions to repeating problems in software design

Gang of Four patterns include 23 design patterns in three categories:

**Creational patterns** are all about different ways to create objects

**Structural patterns** are about the relationships between these objects

**Behavioural patterns** are about interaction / communication between these objects

The advantages of Design Patterns is that they help us communicate with other developers at a more abstract level, for example telling a co-worker to use the command pattern to improve this code, the co-worker will understand without expressing or writing the code to explain what needs to be changed. Another benefit is learning to build reusable, extensible and maintainable software, independent of the programming language you use.

*The art of designing object-oriented software*

Started off by creating a command line application in Java in IntelliJ

Static means we can call this method directly without making an instance of the Main class

Coupling:

* Main class is coupled or dependant on the user class
* If we change the user class, the Man class can be affected
* When the constructor for user now requires age, the Main class is broken.
* Car is an example of a loosely coupled system, when a wheel breaks, you only have to replace it, not the steering wheel or engine.

Interfaces:

* A contract that specifies the capabilities that a class should provide.
* If you want to open a restaurant, you need a chef. It does not matter who the chef is, so you are not dependant on a specific chef, but only on someone who can fulfil the role of a chef with certain capabilities.
* This is an example of a loosely coupled system

Chef would be an interface, and the humans would be classes which implement chef.

With interfaces we can build loosely coupled applications.

TaxCalculator interface which specifies which methods a TaxCalculator class should implement. Example would be the calculateTax method of type float.

Two TaxCalculator classes were set up with two different implementations of the TaxCalculator interface , this is called programming to an interface.

If another method is added into the 2019 calculator class, it will not impact the Main method because we are programming to an interface not a concrete implementation.

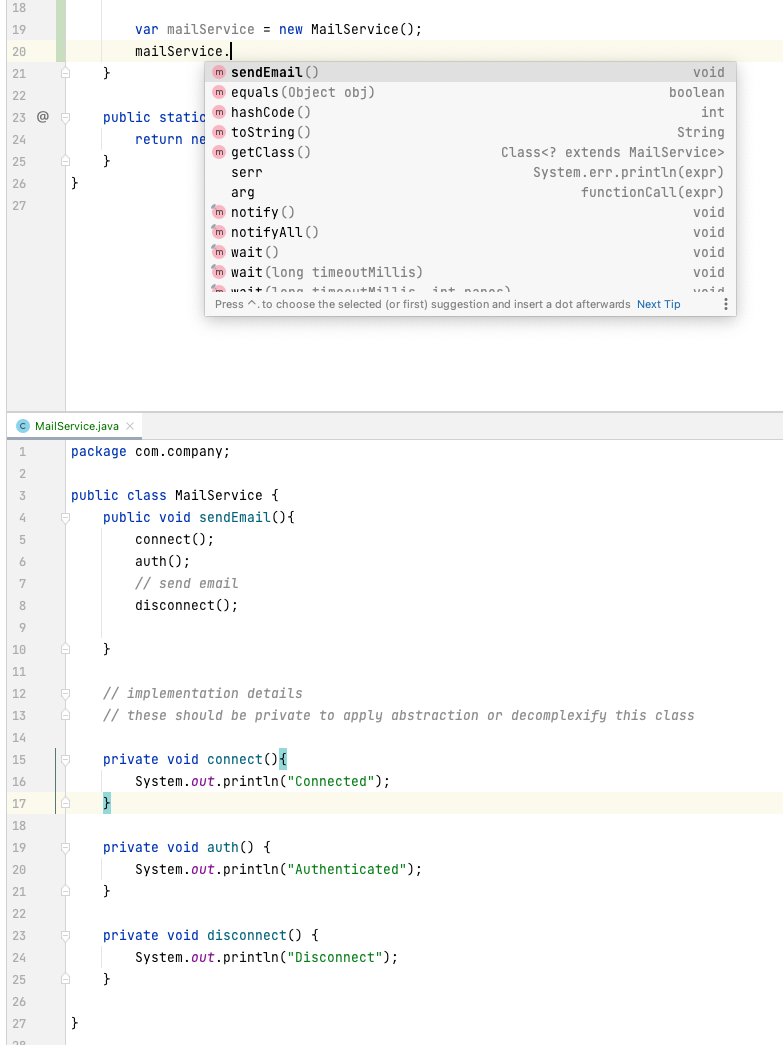
In a real application, we would use dependency injection that would give us real implementations of our interface.

Abstraction:

Reduce complexity by hiding unnecessary details in our classes.

Remote control of a TV has controls of the TV. The remote has a chip. We do not work with the remote’s chip, we work through the buttons on the remote.

A new class was added, MailService which has 3 methods for authenticating, connecting and disconnecting from a service. They just print out a string. These methods are called in the first method of the class called sendEmail. These methods are making it harder for the consumer to use this class as they clutter the interface or the complexity of the class. We should make them private, so to only have the user be able to use the sendEmail method.



Inheritance:

Mechanism for re-using code across our application. For example, a text box, button and a check box and subsequent methods like enable, focus and setPosition for all of them. These methods should be inherited as to not repeat them in every one of those classes.

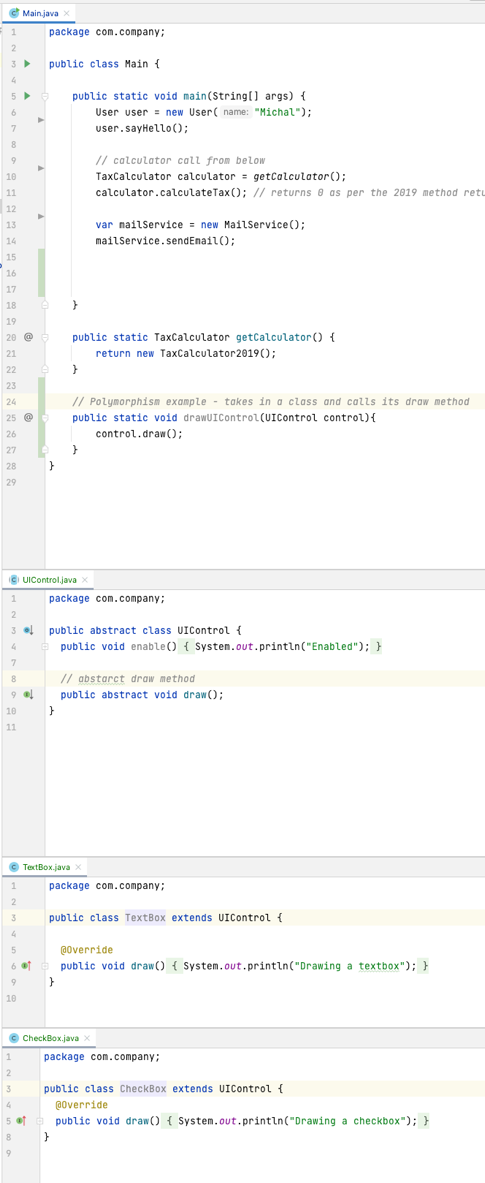
UIControl has a method called enable()

TextBox class extends UIControl and overrides enable() and prints something

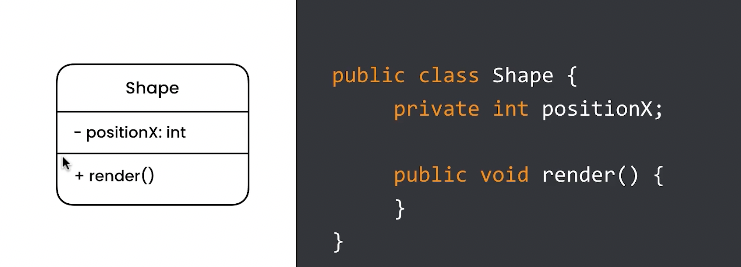
New instance of TextBox has access to the enable method inherited from UIControl.

Polymorphism:

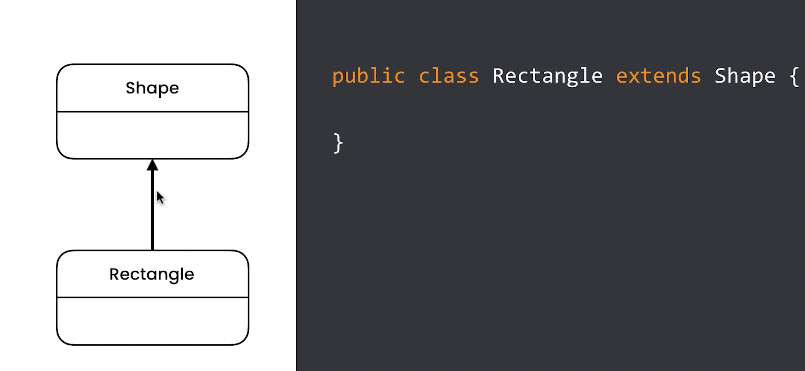
* The ability for an object to take on many forms.
* UIControl can have an abstract void draw() method.
* TextBox and CheckBox both extend UIControl and need the draw method.
* A public static void method which takes in a UIControl type object called dawUIControl is created under main, then we can pass in either the textbox or checkbox class into this method and it will run their draw method.



UML:



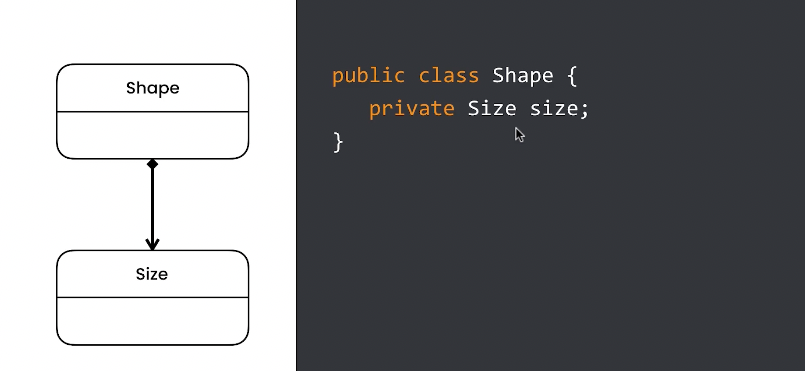
Inheritance:



Composition:

*Shape class is composed of the size class*

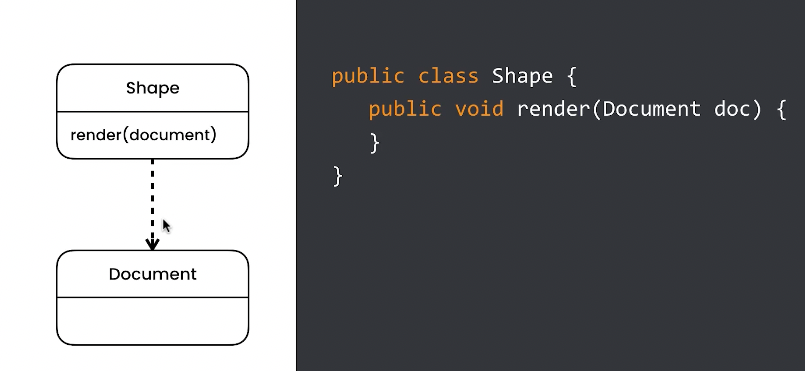
*Car class is composed of the wheel class*



Dependency relationship:

Somewhere in the shape class we have a reference of the Document class

The document is not a field but is used somewhere as a parameter



Relationships:

