# Object Oriented Programming

* It is a methodology we use to design our programs just using classes and objects.
* It makes it easier to develop and maintain your project as it gets bigger.

## Object

* It is any entity that has a state and behavior

## Class

* Template that are used to create objects and define the object data types and methods
* They are a blueprint from which you can create multiple objects from

# Inheritance

* It is just a mechanism in which a class can acquire all the properties and methods of another class.
* It allows us to have code reusability and organization
* It allows us to use one of the tools in polymorphism (Method Overriding)

## Different types of inheritance

* Single inheritance
  + Where the child class inherits one parent class
* **Multilevel** inheritance
  + Like the single inheritance but the child class inherits from another child class that inherits from the parent class
* Hierarchical inheritance
  + Where the parent class has multiple child classes.

# Polymorphism

* The ability of an object to take on many forms
* It allows you to substitute different implementation/behaviors for different needs
* It can also be used to add more functionality to pre-existing code

## Method Overriding

* When a derived/child class changes the implementation details of a method from the base/parent class

## Method Overloading

* When there is multiple method but with different parameters and most of the time, different implementation/behavior details

# Encapsulation

* The process of wrapping code and data together into a single unit.
* So essentially, the validation/how you set a value and any processing of your data in your class will be handle by the class itself.
* They prevent unauthorized access to your object’s properties and setting values that shouldn’t be there.

# Abstraction

* The process of hiding the implementation/how something is done and only showing the functionality to the user.
* Simplify complex written code and just worrying about how it will function in your application
* Ex: You know how to send a text message in your phone, but do you really know how it is able to do that?

## Interface

* It contains nothing but **abstract** methods and properties
* You can add a method and not give it any code/implementation/function/behavior
* You can implement multiple interfaces (THIS IS NOT MULTIPLE INHERITANCE)
* You cannot create a constructor in an interface

## Abstract class

* May contain some methods and properties with implementation details
* You can create a constructor in an abstract class

# Access Modifiers

* They are a way to restrict access.
  + If you need a way to not have a method, be inherited from a parent class
  + If you need a property to only be accessible within the class

## Public

* Everything has access to it

## Internal

* Access within the class
* Access within the child class
* Access within the same project/assembly
* Default access modifiers for classes

## Protected

* Access within the class
* Access within the child class

## Private

* Access within the class
* This is the default access modifiers for class members (fields, methods, constructor, properties)

# Non-access Modifiers

## Abstract

* Enables you to create incomplete implementation of whatever you applied to, and it must be implemented by the child class.

## Static

* The class members belong to the class itself rather than a specific object
  + If one object changes the value of that static field, every object will also change that value.
* Static classes cannot be instantiated or inherited; its members must also be static.

## Const

* Constant fields may not be modified.
* Constants can be numbers, Boolean, values string, null reference.
* Don’t create a constant to represent something that will always change in the future.
* Must be initialized at declaration.
* They are implicitly static.

## Readonly

* That field can only be instantiated/set a value once
  + Can be instantiated/set a value later (mostly inside of a constructor)

## Sealed

* Sealed classes cannot be inherited by other classes
* Sealed methods, properties cannot be inherited by other classes

## Virtual

* Allows for a class member to be overridden in the child class

## Override

* It will look for the class member in the parent class and checks if the child class overrides that method.
* If you put override non-access modifier then that method must override something or else, it will give you an exception.

## Partial

* When you split a class into two or more files.
* When you want two or more developers work on the same class in C#.
* At compile time, it will “merge” those partial classes into one.

# S.O.L.I.D Principles

* There are five design patterns which are intended to make software design more flexible and understandable and maintainable.
  + They are kinda like OOP, but it is a just a bunch of rules you should follow to make your life easier.

## Single Responsibility Principle

* A class should have one and only one reason to change
* If one class has more responsibility, then just segregate them into many classes instead
* Ex: Software engineer shouldn’t be responsible for also managing the financial forms of the company, it will be handled instead by an accountant.

## Open/Closed Principle

* A class should be open for extensions but closed for modifications
* It just means you can add new functionality without changing existing code
* Use interfaces so if you need to add a new functionality, adding a method in an interface will automatically tell you all the other files that implement this interface and must have some sort of implementation to them

## Liskov Substitution Principle

* The child class should be substitutable for their parent class.
* It just means the child class should not behave in such a way that it will cause problems when you replace it with its parent class.
* This just means it is good practice to use base keyword
  + It is used to call upon the parent’s implementation details

## Interface Segregation Principle

* You should not be forced to implement methods that you don’t need in an interface
* Just segregate the interface into multiple interfaces

## Dependency Inversion Principle

* High-level and low-level modules should depend on abstractions and not on each other
* If a class uses the design and implementation of another class, it raises the risk that changing one class could potentially break the other class
* So, we use interface to essentially guarantee that it will always have those methods that will return the same thing and have the same parameters

# Introduction to Exceptions

* An exception is an event that occurs during the execution of a program
* It will disrupt the flow of your instructions/code
* Good when trying to find bugs, very bad if you deployed your code and expect everything to work just fine
* An error is a serious problem that you as a developer cannot fix.
  + Fatal to the program at runtime
  + Cannot be handled with exception handling
* There are 3 types of error
  + Usage Error – error in your program logic and can be solved by modifying your code
    - Ex: while loop that doesn’t end
  + Program Error – It’s the C# developers’ fault (ex: sdk is corrupt or new version of it doesn’t work on your old code)
  + System Error – run-time error that cannot be handled programmatically in a meaningful (ex: your ram hardware is faulty)

## Exception Handling

* Using a try-catch block
* If you know the block of code you will run will have a risk of throwing an exception, you can put a try catch block to handle it
* You shouldn’t handle every exception that your program throws at you
  + Some exceptions are useful for debugging/fixing logic in your code.
* Finally block
  + It runs the block of code no matter what the result of your try block
  + Useful for cleaning up used resources

## Throw Exception

* This allows us to give an exception if needed
* Useful for enforcing rules/logic in your program

## Exception Hierarchy

* All exceptions inherit from Exception class.
* Essentially SystemException is the most generic exception that will catch any exception.

# Design Patterns

* They are reusable solution that will solve problems that occurs pretty frequently while coding.
* Some people saw that this problem keeps happening across multiple people and decided to standardize a solution every time you come across that problem to make your life easier.
* They are the best practice a programmer can do to solve common problems while designing your application

## Singleton Pattern

* This pattern revolves around creating only one concurrent instance of a class
  + So, you can only have one object created from that class
  + This class provides global access for certain value to let other classes have direct access to it

### Advantage

* Provides global point of access for multiple classes
* It is easy to maintain since there is only one instance of that class

### Disadvantage

* Cannot perform multi-threading that well since creating one object will put a halt on your program until it is done
  + This will only happen if you apply singleton pattern right
* Very difficult to unit test since it has global access

## Factory Pattern

* Allows the creation of objects without exposing the creation logic
* We will use an interface to abstract the implementation from the user
* It allows the object to be create at run-time when it is required
* It simplifies the process in that the end-user just needs to call on the method from the factory to create an object

# Variance

* They enable implicit/automatic conversion of classes/types
* This is what makes my menu in RRUI have many different functionalities when pointing to another menu object

## Covariance

* It is when a more generic and the parent class/interface points to its child class
  + This allows the child class’s methods to be used instead of the parent class’s method.

## Boxing

* It is when a value type gets casted into an object
* It is implicit conversion meaning it will automatically happen without you doing anything

## Unboxing

* When you extract the value from an object and convert it into a value type instead
* It is explicit conversion meaning it needs you to write a syntax to tell the compiler to convert it.

## Casting

* When you try convert a value type into another value type
* Ex: convert int into a double or double into an int

# Test Driven Development

* TDD is a software development process that creates test cases for the software requirement first before developing the software.

## The general flow of TDD

1. Create a test case – what you expect the feature/function is supposed to do
2. Run the test case and will expect it to fail – since you haven’t implemented the function
3. Write the code so the new test case will pass
4. Make sure your old test cases won’t fail because of the new feature
5. Clean up the code and have proper documentation for other developers

## Arrange, Act, and Assert

* Arrange – this is where you initialize the objects or some values you will need for the test
* Act – This is where you use the method/function to essentially check if it is working
* Assert – Verifies that the action of the method under the test behaves as expected