**Abstraction**

**What it means:** Abstraction means hiding the complex details of how things work and only showing what is necessary. It’s like driving a car—you don’t need to know how the engine works, you just use the steering wheel and pedals.

**How I used it:** In my goal tracking program, I created a Goal base class that had basic methods like RecordEvent() and GetDetails(). The user just had to choose a goal and record progress without knowing how each goal worked behind the scenes.

**Why it helps:** Abstraction made it easier to work with different types of goals (SimpleGoal, EternalGoal, ChecklistGoal) using one interface. This means if I ever change how a goal works internally, I don’t have to rewrite the main program or menu code

What is Encapsulation?

Encapsulation is the 2nd principle of OOP (Object Oriented) Programming, OOP seeks to write cleaner, less error-prone and more maintainable code where the objects do the work instead of procedural where the code is more of a process instead of a declaration. The role of encapsulation is to isolate the details of the object that’s doing the work, by applying modifiers that control how the various attributes and methods of the class are accessed, by other classes that invoke or instantiate them. This may seem needless, but once we can externally control data types of these attributes, we can find ourselves with a horde of runtime errors, that can be caused by any of the programs that instantiated the affected class.

Benefits of Encapsulation:

Encapsulation allows programs to be less error prone and to be more declarative and strictly-typed, what this means is that, since we can make our attributes private, when maintaining our code for example changes to internal implementation do not affect other parts of the code, the program becomes highly maintainable, additionally it helps with making the program modular, this allows us to extract one part of the code, analyze and improve and design its output data, note that modularity and maintainability are different, modularity can help accelerate production by having teams focus on different “modular” parts of the program.

Application of Encapsulation

Reusability, the different teams can still share common libraries of OOP, which were in part built using encapsulation.

Code Examples:

using System.Text.RegularExpressions;  
  
namespace ScriptureMemorizer;  
  
public class Word  
{  
 private string \_text;  
 private string \_wordPart;  
 private string \_punctuation;  
 private bool \_isHidden;  
  
 public Word(string text)  
 {  
 \_text = text;  
   
 Match match = Regex.Match(text, @"^(\w+)(\W\*)$");  
   
 if (match.Success)  
 {  
 \_wordPart = match.Groups[1].Value;   
 \_punctuation = match.Groups[2].Value;  
 }  
 else  
 {  
 \_wordPart = text;   
 \_punctuation = "";  
 }  
 }  
  
 public void Hide()  
 {  
 \_isHidden = true;  
 }  
  
 public void Show()  
 {  
 \_isHidden = false;  
 }  
  
 public bool IsHidden()  
 {  
 return \_isHidden;  
 }  
  
 public string GetDisplayText()  
 {  
 return \_isHidden ? new string('\_', \_wordPart.Length) + \_punctuation : \_text;  
 }  
}

Private Attributes (\_text, \_wordPart, \_punctuation, \_isHidden)

These fields ae hidden outside classes, they ensure data integrity, not external class can modify \_isHidden directly, preventing accidental changes.

Public Methods (Hide(), Show(), IsHidden(), GetDisplayText())

These methods provide controlled access to private data.

Hide() and Show() change \_isHidden safely

GetDisplayText() controls how words are displayed without exposing \_isHidden directly.

This demonstrates Encapsulation by restricting direct access to the internal details of the Word class while allowing interaction through provided methods.

**What is Inheritance?**

Inheritance is one of the core principles of Object Oriented Programming (OOP). It allows one class (called the *child* or *subclass*) to inherit attributes and behaviours (fields and methods) from another class (called the *parent* or *superclass*). This helps avoid repetition and keeps code organized. Instead of writing the same code in multiple places, we can just write it once in a base class and let other classes use it.

**Benefits of Inheritance:**

One major benefit of inheritance is that it promotes code reusability. If multiple classes share common behaviour, you can define that behaviour in a base class. This makes the code easier to maintain because changes to shared behaviour only need to be made in one place. Also, it helps with readability, developers can look at a class and understand what it does by seeing what it inherits from.

**Application of Inheritance:**

In my recent assignment, I built a simple program where I had a base class called Activity, and then three child classes: BreathingActivity, ReflectionActivity, and ListingActivity. Each of the child classes inherited shared methods like DisplayStartMessage() and DisplayEndMessage() from the base Activity class, while also having their own unique methods.

**Code Example:**



Both ShowSpinner() and GetEndTime() are methods inherited from the base class: Activity.

In this example, I’m also calling the base class constructor with two arguments—(name, description)

**What is Polymorphism**?

means "many forms." In programming, this lets us use the same method name, like RecordEvent, on different kinds of objects and get different results depending on the object’s type. This is helpful because we can write code that works with a general type (like a Goal), but the program will still know which specific version of the method to run at runtime.

A big benefit of polymorphism is that it makes our code **more flexible and easier to maintain**. For example, if I later add a new type of goal, I don’t have to change the rest of my program. I just need to make sure the new goal type has its own version of the method.

An example from my own program is this line from GoalsManager.cs:

private void RecordEvent()  
{  
 ListGoalNames();  
 Console.Write("Which goal did you accomplish? ");  
 int target = Convert.ToInt32(Console.ReadLine()) - 1;  
  
 var goal = \_goals[target];  
  
 goal.RecordEvent();  
  
 \_score += goal.GetPoints();  
}

goal.RecordEvent();

This works whether goal is a SimpleGoal, EternalGoal, or ChecklistGoal. Each of those classes has its own RecordEvent() method, and the program automatically uses the right one. I don’t have to write separate code for each goal type when recording an event.

Polymorphism is powerful because it lets me **write cleaner code**, reuse logic, and work at a higher level without worrying about the details of each goal type. It really helped me stay organized while building my goal tracker.