**WEEK 02** - What is version control and why is it important?

A **version control system** is a method for developing projects, programs, etc., which allows saving the information provided by a team member, shows that information to the rest, and allows coordination and work in real time.

One of the best tools is its version saving, as if it were a video game that saves the game in a certain place. When a member uploads his version of the project, everyone can view it, download it, work on it, or even create their own branched versions. They can go back in time and evaluate previous versions of the project, to avoid bugs and errors, which can be recovered at any time.

All team members can see who made changes, when they were made, what changes were posted, and even comments explaining why they were made.

**Git** is a popular example of a version control system. One of its commands is "git branch 'myBranch' ", which allows you to create a branch of the project with the name 'myBranch' and work on that branch without modifying the main project. Once a member wants to merge that branch to the project trunk, he/she can use the "git merge" command.

Version control can be used, for example, to develop a mobile app among a team. Members work on a repository (where the files for application development are stored), create local copies on their computers, and work from home, proposing improvements, new code, and more. Thus, they can work on the same project, physically separated, but together in terms of vision and objective.

**WEEK 04** - What is abstraction and why is it important?

**Abstraction** is a way of breaking down a complex idea, staying with the essential and simple parts, and building the project with a good correlation of those simple parts taken.

Some advantages of applying the abstraction when programming are security, since only the important details are provided to the user, also the possibility of internally modifying classes, independent of the main program handled by a user, and it improves reusability and avoids duplicating lines of code.

An application of abstraction is the implementation of classes. Just as objects have states and behaviors that allow them to fulfill their responsibility, they are similar to classes, which have attributes (states, variables) and methods (behaviors, functions).

For example, if we wanted to display a resume, showing each of the jobs and their specific characteristics (such as company, title, years, etc), instead of keeping everything in lists of complex "strings", we could define a class called Job that can store the attributes of Title, Company, Years, etc, and then create in another program a list with "Job" as a type of variables, which is actually a class. This way we store the information more comfortably and can access it in order.

//The development of the Job Class

*public class Job {*

*// Its attributes*

*public string \_company;*

*public string \_jobTitle;*

*public int \_startYear;*

*public int \_endYear;*

*// Its methods*

*public void DisplayJob() {*

*Console.WriteLine($"{\_jobTitle} ({\_company}) {\_startYear}-{\_endYear}"); }}*

// The creation of the list of Job types of variables in other program by using the Job class.

*public List<Job> \_jobs = new List<Job>();*

// Calling the DisplayJob method of the Job Class to use it (as an example).

*\_jobs[0].DisplayJob();*

**WEEK 06** - What is encapsulation and why is it important?

**Encapsulation** is a fundamental principle in programming that involves bundling data and methods within a single unit, known as a class, and restricting access to internal components from the outside world. It enables the creation of a modular and organized code structure while ensuring data integrity and security.

One of the significant benefits of encapsulation is data hiding. By encapsulating data within a class, we can control its accessibility and prevent direct manipulation from external sources. This promotes data integrity, as it becomes easier to enforce validation rules and maintain consistency throughout the program. Encapsulation also enhances code readability and maintainability, as changes made within the class do not affect the code outside the class.

An application of encapsulation can be seen in a banking system. The account class may encapsulate private data such as the account holder's name, account number, and balance. It provides methods like deposit, withdraw, and getBalance, which allow controlled access to the encapsulated data. This encapsulation ensures the privacy and security of sensitive information, preventing unauthorized access and manipulation.

Here is an example of encapsulation:

public class Car

{

private string model;

private int year;

public string GetModel()

{

return model;

}

public void SetModel(string newModel)

{

model = newModel;

}

public int GetYear()

{

return year;

}

public void SetYear(int newYear)

{

if (newYear >= 1900 && newYear <= DateTime.Now.Year)

{

year = newYear;

}

else

{

Console.WriteLine("Invalid year!");

}

}

}

public class Program

{

public static void Main()

{

Car myCar = new Car();

myCar.SetModel("Toyota Camry");

myCar.SetYear(2018);

Console.WriteLine("Car Model: " + myCar.GetModel());

Console.WriteLine("Car Year: " + myCar.GetYear());

}

}

In this example, the Car class encapsulates private variables model and year within the class. The private variables are not directly accessible from outside the class. Instead, public getter and setter methods (GetModel(), SetModel(), GetYear(), SetYear()) are used to control access to these variables.

The SetModel() and SetYear() methods allow controlled modification of the private variables by performing necessary validations. In the SetYear() method, for instance, it checks if the provided year is within a valid range (1900 to the current year) before updating the year variable.

In the Main() method, we create an instance of the Car class (myCar) and use the public setter methods to set the car's model and year. Then, we use the public getter methods to retrieve and display the car's model and year.

In summary, encapsulation plays a crucial role in programming by providing a means to bundle data and methods, safeguarding data integrity, promoting code organization, and controlling access to sensitive information. It allows for the creation of robust and maintainable software systems, ensuring efficient development and facilitating collaborative programming.

**WEEK 08** - What is inheritance and why is it important?

**Inheritance** is a concept that allows a class to inherit the properties, methods, and characteristics of another “super” or “main” class. There is a hierarchical relationship between this “base class” and the “subclasses”.

One of the principal benefits is making reusable code. The derived classes inherit all their base class members, such as fields, properties and methods. This prevents rewriting common code, and changes made in the base class will automatically impact the derived classes. It helps with the readability and maintainability of the code.

A book management system can be an example. The base class "Book" contains properties like "Title" and "Author," and methods like "DisplayInfo()" and other getters and setters for the properties. Derived classes like "Novel" and "HorrorBook" inherit from "Book" and add specific properties and methods.

public class **Book**

{

private string \_title;

private string \_author;

public void DisplayInfo()

{

Console.WriteLine("Title: " + \_title);

Console.WriteLine("Author: " + \_author);

}

// Getters and setters for the title and author

// … (more code)

}

public class **Novel** : *Book*

{

private string \_genre;

public void Read()

{

Console.WriteLine("Enjoy reading the novel!");

}

// Getter and setter for the genre

// … (more code)

}

public class **HorrorBook** : *Book*

{

public void Scare()

{

Console.WriteLine("Prepare to be scared!");

}

}

public class **Program**

{

public static void Main()

{

Novel myNovel = new **Novel**();

myNovel.SetTitle("Pride and Prejudice");

myNovel.SetAuthor("Jane Austen");

myNovel.SetGenre("Romantic Fiction");

Console.WriteLine("Novel:");

myNovel.DisplayInfo();

Console.WriteLine("Genre: " + myNovel.DisplayGenre());

myNovel.Read();

Console.WriteLine();

HorrorBook myHorrorBook = new **HorrorBook**();

myHorrorBook.SetTitle("It");

myHorrorBook.SetAuthor("Stephen King");

Console.WriteLine("Horror Book:");

myHorrorBook.DisplayInfo();

myHorrorBook.Scare();

}

}

In this example, the Book base class contains common properties, such as \_title and \_author, and methods like DisplayInfo() and other getters and setters. The Novel and HorrorBook derived classes inherit from the Book class those properties and methods. When we create a Novel derived class, for example, we can set his \_author and \_title with the respectives getters and setters.

Also, the Novel class has a specific property and method (\_genre and Read()) which can’t be used in the HorrorBook class. The same for the Scare() method in the HorrorBook class.

In the Main() method, we create a Novel instance called myNovel and set its properties. Next, we display the information for the book using the DisplayInfo() method inherited from the base class and display the specific genre of the novel. Finally, we call the Read() method to enjoy reading the novel. Next, we create an instance of HorrorBook called myHorrorBook and set its properties. We display the book information and call the Scare() method to prepare for the book scare.

**WEEK 08** - What is polymorphism and why is it important?

**Polymorphism** is a fundamental concept; it refers to the ability of objects to take on different forms or behaviors based on their underlying types or classes. In other words, objects of different classes can be treated as objects of a common base class, allowing for flexibility and code reusability.

One of the key advantages is its ability to simplify code and improve maintainability. By using polymorphism, you can write code that operates on a generic object type, and it can automatically handle different specific object types without the need for explicit type checking and casting. This reduces code complexity and makes it easier to extend and modify the code in the future.

An application of polymorphism can be seen in a scenario involving a base class, such as Shape, and derived classes, such as Circle and Rectangle. By defining a common method, like CalculateArea, in the base class and implementing it differently in the derived classes, you can leverage polymorphism to call the method on objects of any derived class without knowing the specific type at compile-time.

public class Shape

{

public virtual double CalculateArea()

{

return 0;

}

}

public class Circle : Shape

{

public double Radius { get; set; }

public override double CalculateArea()

{

return Math.PI \* Radius \* Radius;

}

}

public class Rectangle : Shape

{

public double Width { get; set; }

public double Height { get; set; }

public override double CalculateArea()

{

return Width \* Height;

}

}

public class Program

{

public static void Main()

{

Shape circle = new Circle { Radius = 5 };

Shape rectangle = new Rectangle { Width = 4, Height = 6 };

Console.WriteLine("Circle Area: " + circle.CalculateArea());

Console.WriteLine("Rectangle Area: " + rectangle.CalculateArea());

}

}

In this example, the Shape class serves as the base class, and the Circle and Rectangle classes are derived from it. Each derived class overrides the CalculateArea method according to its specific shape formula. By declaring objects as the base class type (Shape), you can call the CalculateArea method on these objects, and polymorphism will ensure that the appropriate implementation is invoked based on the actual object type.

**W14 Prove: Articulate:** [Final Project Articulate Activity](https://byui-cse.github.io/cse210-course-2023/unit07/articulate.html)  
  
**Abstraction:**

Abstraction refers to the concept of simplifying complex systems by representing only the essential details and hiding the unnecessary complexity. It allows us to focus on the relevant aspects of an object or system while abstracting away the implementation details.

For example, in my final project, I used abstraction to create an abstract class called Video that defined the common properties and behaviors of a video. This abstraction allowed me to create different types of videos by extending this base class and implementing the specific details for each type.

**Encapsulation:**

Encapsulation is the process of bundling data and the methods that operate on that data into a single unit, known as a class. It helps in hiding the internal details of an object and exposing only the necessary methods and properties for interacting with it.

In my final project, I achieved encapsulation by defining private variables and using public methods to access and modify them. For example, in the Product class of my ordering system project, I encapsulated the price and quantity variables and provided public methods to retrieve and update these values.

**Inheritance:**

Inheritance is a fundamental principle of object-oriented programming that allows a class to inherit properties and behaviors from a parent class. It enables code reuse, extensibility, and the creation of hierarchical relationships between classes.

In my event planning project, I used inheritance to create different types of events (lectures, receptions, outdoor gatherings) by deriving them from a base Event class. The base class contained common attributes like event title, description, date, time, and address, while the derived classes added specific properties and behaviors unique to each event type.

**Polymorphism:**

Polymorphism refers to the ability of an object to take on different forms or behaviors depending on its context. It allows objects of different classes that are related through inheritance to be treated as objects of their common base class.

In my fitness tracking app project, I implemented polymorphism by defining a base Activity class and derived classes for different activities (running, cycling, swimming). Each derived class provided its own implementation of methods such as calculating distance, speed, and pace, allowing objects of different activity types to be treated uniformly when accessing these common behaviors.

Overall, by utilizing these principles of Programming with Classes, my final project became more flexible for future changes. For example, if I wanted to add a new type of activity in my fitness tracking app, I could easily create a new derived class that inherits from the base Activity class and override the necessary methods without modifying the existing code. Similarly, in the event planning system, I could introduce new event types by creating derived classes that inherit common attributes and behaviors from the base Event class. This flexibility allows for scalability and easier maintenance of the codebase.