Lesson 22 - Classes & Objects III

Properties

It is a good idea to encapsulate members of a class and provide access through public methods. A **property** is a member that provides a flexible mechanism to read, write, or compute the value of a private field, but they actually include special methods called **accessors**.

Accessors contain executable statements that help in to `get` or `set` the value of the corresponding field. Accessor declarations can include `get`, `set`, or both.

```
public class BankAccount
{
        private int balance;

        public int Balance
        {
                get { return balance; }
                set { balance = value; }
        }

        public BankAccount(int openingBalance)
        {
                balance = openingBalance;
        }
}
```

Properties can be used using the dot operator just like any other public members of a class.

```
BankAccount acc = new(100);
Console.WriteLine(acc.Balance);
acc.Balance = 200;
Console.WriteLine(acc.Balance);
```

```
OUTPUT:
100
200
```

Properties can also be private, so they can only be called from withing the class. And you can also make a property read-only by omitting the `set` accessor.

Why use properties?

You can create logic for accessing variables using properties. Such as checking whether a variable is greater than zero before assigning. It is always recommended to use properties even when you don't need any custom logic.

Why write all that code when there is no need for custom logic?

You can actually use shorthand syntax to create properties when you don't need any custom logic. These are called autoimplemented properties. The following updated version of `BankAccount` functions the same way as the previous one.

```
public class BankAccount
{
        public int Balance { get; set; }

        public BankAccount(int openingBalance)
        {
                Balance = openingBalance;
        }
}
```

There is a special shorthand syntax to write single line methods, using which we can futher shorten our code:

```
public class BankAccount
{
    public int Balance { get; set; }

    public BankAccount(int openingBalance)
        => Balance = openingBalance;
}
```

Destructors/Finalizers

As constructors are used when a class is instantiated, destructors are automatically invoked when an object is destroyed or deleted.

Destructors have the following attributes:

- A class can only have one destructor.
- Destructors cannot be called. They are invoked automatically.
- A destructor does not take modifiers or have parameters.

- The name of a destructor is exactly the same as the class prefixed with a tilde `~`.
- They help in releasing resources before coming out of a program.

```
public class BankAccount
{
    public int Balance { get; set; }

    public BankAccount(int openingBalance)
    {
            Console.WriteLine("BankAccount Constructor.");
            Balance = openingBalance;
        }

        ~BankAccount()
        => Console.WriteLine("BankAccount Destructor.");
}
```

`this` keyword

It is used inside the class and refers to the current instance of the class, meaning the current object. Common uses are to disinguish class members of same names from other data such as parameters, or to pass current instance to a method.

```
public class BankAccount
{
    public int Balance { get; set; }

    public BankAccount(int Balance)
    {
            Console.WriteLine("BankAccount Constructor.");
            this.Balance = Balance;
        }
}
```

`readonly` modifier

This prevents the member of a class from being modified after construction. Fields declared as readonly can only be modified when you declare it or from within constructor.

```
BankAccount acc = new(100);
acc.ShowDetails();

BankAccount newAcc = new("Talha", 200);
newAcc.ShowDetails();

public class BankAccount
{
```

```
    public int Balance { get; set; }
    private readonly string accountTitle = "default";

    public BankAccount(int Balance)
        => this.Balance = Balance;

    public BankAccount(string accountTitle, int balance)
    {
        this.Balance = balance;
        this.accountTitle = accountTitle;
    }

    public void ShowDetails()
        => Console.WriteLine($"Account Title: {accountTitle}\nBalance:
{Balance}");
}
```

Extra Example

```
BankAccount blankAccount = new(100);
blankAccount.ShowDetails();

Console.WriteLine();

BankAccount titleAccount = new("Talha", 150);
titleAccount.ShowDetails();

Console.WriteLine();

BankAccount deadAccount = new();
deadAccount.ShowDetails();

public class BankAccount
{
    public int Balance { get; set; } = 0;
    private readonly string accountTitle = "default";

    public BankAccount() { }

    public BankAccount(int balance)
        => Balance = balance;

    public BankAccount(string accountTitle, int balance)
    {
        Balance = balance;
        this.accountTitle = accountTitle;
    }

    public void ShowDetails()
```

```
        => Console.WriteLine($"Title: {accountTitle}\nBalance: {Balance}");
}
```