Object Oriented Programming with C#

Beginner to Advanced

by Tod Vachev

Bestselling Udemy Instructor

“Magic Destroyers” Project

Introduction

This is the project that will guide you throughout the course and help you practice everything that you are learning. Do not hesitate to ask if you have any questions.

Follow the parts of the project as you are going throughout the course. Feel free to add/remove/modify any parts of the project as you see fit, as long as you are still able to practice the given concept. You are free to add your own creative elements to the project, I am encouraging you to do so!

I am also encouraging you to try and implement the parts of the project on your own before(!) you watch the project videos in the course. That way you will be able to practice unbiased, you will be able to detect your mistakes (if there are any) and fix them when you watch the project videos.

Part 1

Creating the classes

Allright, now that we know all of the basic building blocks of an Object (Class), its time to put them in practice.

This is the first part of our Project, the project is called "Magic Destroyers" (feel free to rename it and share the name if you come up with something better! ☺).

It's rather longer than the other parts of this project, but that's because here we will be setuping our "playground". As you know, preparation is key in anything you do. This is what we will be doing in this part. It's long, but not hard and you will practice creating classes.

We will be making a game, but the focus is not going to be entirely on the game logic, but rather on the architecture of the game.

Preparation and planning is extremely important. Correct and somewhat detailed plan is 50% of your work completed.

As you keep going through the different parts of the project throughout the course, I will give you chunks from my plan that you will use to continue the project.

Feel free to add/modify anything in the project as you see fit. If you don't break stuff and then spend hours trying to find out how to fix it, or make it better, you will not learn. Its also fun sometimes.

Before you start creating the project, create an entirely new solution within Visual Studio, you will use one solution while going through the course, and another solution specificly for the project, so that we can keep things nice and clean.

Now, as I said, we will start with creating our classes.

Your tasks for this exercise are:

1. Create appropriate namespaces and folders to properly structure your classes and files, according to the given information about the project
2. Create all classes along with their corresponding fields and methods
3. Create a constructor for each of these classes, it can be just one default empty constructor, we will be adding more later in the course

In the game that we are going to create we will have 6 different types of Characters, split between Melees and Spellcasters:

- Characters
    - Melee
        - *Warrior*
        - *Knight*
        - *Assassin*
    - Spellcasters
        - *Mage*
        - *Necromancer*
        - *Druid*

That's why I called the game "Magic Destroyers", there will be a 3v3 between the Melees and the Spellcasters, the Melees are trying to destroy the Spellcasters.

All of these characters should have the following fields:

- abilityPoints (int)
- faction (string for now, will be changed to something else later)
- healthPoints (int)
- level (int)
- name (string)
- bodyArmor (Specific Armor type (new class needs to be created, keep reading))
- weapon (Specific Weapon type, keep reading)

Hint: *You can hold on creating Properties until you learn more about them in the Properties section of the course, so that you don't have to redo something.*

All of the characters will have 3 abilities, 2 offensive and 1 defensive:

- *Warrior*
    - **Strike**
    - **Execute**
    - **SkinHarden**
- *Knight*
    - **HolyBlow**
    - **PurifySoul**
    - **RighteousWings**
- *Assassin*
    - **Raze**
    - **BleedToDeath**
    - **Survival**
- *Mage*
    - **ArcaneWrath**
    - **Firewall**
    - **Meditation**

- *Necromancer*
    - **ShadowRage**
    - **VampireTouch**
    - **BoneShield**
- *Druid*
    - **Moonfire**
    - **Starburst**
    - **OneWithTheNature**

Feel free to change these abilities to your preferences, I am just giving you examples, you don't need to stick with my implementation of the characters and their abilities. The important thing here is to simply practice creating classes and their members.

- Equipment
    - Armors
        - *Chainlink*
            - **armorPoints (field)**
        - *LightLeatherVest*
            - **armorPoints (field)**
        - *ClothRobe*
            - **armorPoints (field)**
    - Weapons
        - *Sword*
            - **damage (field)**
            - **Bloodthirst (ability)**
        - *Axe*
            - **damage (field)**
            - **HackNSlash (ability)**
        - *Hammer*
            - **damage (field)**
            - **Stun (ability)**
        - *Staff*
            - **damage (field)**
            - **Empower (ability)**

The armors will have a *armorPoints* field, and the weapons will have *damage* field along with the item abilities listed bellow each of the weapons.

You can use these new equipment classes for the armor/weapon types of the characters. For example, the weapon field of a warrior could become *Sword weapon,* Sword type, weapon field name.

Feel free to spread the equipment between the classes as you see fit, I will be doing it like this:

- *Warrior*
    - **Chainlink** *bodyArmor;*
    - **Axe** *weapon;*
- *Knight*
    - **Chainlink** *bodyArmor;*
    - **Hammer** *weapon;*
- *Assassin*
    - **LightLeatherVest** *bodyArmor;*
    - **Sword** *weapon;*
- *Mage*
    - **ClothRobe** *bodyArmor;*
    - **Staff** *weapon;*
- *Necromancer*
    - **LightLeatherVest** *bodyArmor;*
    - **Sword** *weapon;*
- *Druid*
    - **LightLeatherVest** *bodyArmor;*
    - **Staff** *weapon;*

Part 2

Add & Validate Properties

The second part of the project is an easier one. All you have to do is, wrap all your fields with properties and implement validation where you think it would be necessary to do so.

Tip: *You can do it once for a single character and then just copy/paste it for the other characters, at least for the fields/properties that are shared between them, still be very careful and avoid copy/paste as much as possible, and whenever you do it, triple-check that everything is correct.*

I will let you decide how to implement your own validations, but I will just give you a couple of directions:

- **Damage** for Weapons should be greater or equal to 1, it makes no sense to have a weapon with damage 0, or with negative damage;
- **ArmorPoints** for the Armors, should also be greater or equal to 1, same reason like with the weapons;
- **AbilityPoints**, Level, HealthPoints should also be positive at all times, some of them can be 0, but not negative, some of them can also vary between the different characters, e.g. Warrior may have 10 abilityPoints, Mage may have up to 20;
- There are only two **Factions** in our game, come up with your own names for the factions, I will use “Melee” and “SpellCasters”

Part 3

Creating constructors and using the this. keyword

Now that you've seen the ***this.*** keyword and if you have decided to use it, you need to implement it in all of your current properties.

I am currently not aware of any easy way to do it, so you would have to manually write it everywhere where you need to, I know, I know, its probably a lot of writing and you may be turned of by this fact, but don't be! Using the **this.** keyword, as we discussed is very helpful.

With this out of the way its time to create our constructors, which of course, will still be pretty similar for all of our characters.

I know that this seems like a lot of useless writing, and it actually is! But I am making you do it, so that you can really appreciate something that we will learn pretty soon in the course.

Create 2 additional constructors (3 total) for your classes (for the characters, theres no point in making more constructors for the weapons/armors).

These constructors should take the following initial arguments:

- **Constructor 1**, default constructor
- **Constructor 2**
    - *Name*
    - *Level*
- **Constructor 3**
    - *Name*
    - *Level*
    - *HealthPoints*

Do this for all characters.

After the constructors are in place, practice chaining them, so you should have no code within the body of Constructor 1 and Constructor 2, only in Constructor 3.

One last thing that you need to do here is to set default initial values inside Constructor 3 for all of the remaining Properties.

For example:

- *Warrior*
    - **this.Faction** = *“Melee”*
    - **this.AbilityPoints** = *100*
    - **this.Weapon** = *new Axe()*
    - **this.BodyArmor** = *new Chainlink()*

Use appropriate **Faction** for the given character, **abilityPoints** can be the same for all characters.

Part 4

Static Classes, their members and Enums

We will start this part of the project by adding default values as constants in our classes. So all of the default values that you have in your constructors, the "magical numbers", you should extract in constants using appropriate naming convention and appropriate values.

Since these are your initial default values, they should be private. Some of them can/should be different depending on the Character (just for diversity).

Create an Enumeration for the Faction, with only two options, so now the faction is no longer going to be a string, it will be of type Faction and will have two specific values, so that no one can ever input an incorrect faction.

We will have one major static class that is going to handle the information about our characters:

- *PlayersInfo*
  - *Fields:*
    - ***playersInfoDirectory***
    - ***fullInfo***
    - ***scores***
    - ***levels***
  - **Initialize()**
  - **Save()**
  - **UpdateFullInfo()**
  - **RetrieveFullInfo()**
  - **PrintFullInfo()**
  - **EraseFullInfo()**
  - **UpdateScores()**
  - **RetrieveScores()**
  - **PrintScores()**
  - **EraseScores()**
  - **UpdateLevels()**
  - **RetrieveLevels()**
  - **PrintLevels()**
  - **EraseLevels()**

So as you can see, we will have several methods in that static class, that are just going to handle the players info, we don't need a second instance of that class, that's why its static.

Implementing the logic within these methods is not the most important thing here, but we will do it.

In order to save the information from one game to the next, we need to use a text file. We need to use the IO classes that C# offers, Directory and File, which are also static, they just allow you to work with files. There are also the classes DirectoryInfo and FileInfo which are not static but allow you to work individually with different files and save them in instances so that you can work with them more efficiently, but that is not required in our case. We need to use them only in our Initialize() method so that we can initialize our fullinfo, scores and levels fields which will be used in the other methods.

Having this class in working order is not a top priority at the moment, since we still need to learn more things about OOP. This is why at this moment I will only give you general guidelines about its implementation if you want to do it now, and if you want you can wait until we finish with the 4 Pillars of Object Oriented Programming, at which point, we will have all of the tools that we need and we will start the overall game logic implementation of all classes and methods. We will be primarily dealing with the architecture of the project until that moment.

So, the first field is the directory to the file, and the other 3 fields should be arrays. Fullinfo will be 2x6 array, holding scores and levels in it, each has its own rows, this is the "raw" information that we will be reading from the file.

Scores and Levels fields will be 1x6 arrays, which will be extracting their information from the FullInfo array. All of that is done in the Initialize() method, which we will be running at the beginning of our game. That method will also be responsible for the creation (if not available), and initial reading of the text file that will hold all of that information.

The rest of the methods are pretty much self explanatory about what they are doing.

Part 5

Inheritance – Getting rid of the redundant code

Allright, if you got to this part, you probably already know what is coming.

As we have already discussed in the Inheritance part of the course, its main purpose is to eliminate redundant code, and we have a looot of it!

Just take a look at your character classes, how many fields/properties are shared between them, we wrote the same thing 6 times! Now, I am sorry that I had to put you through this, but I had to torture you a little, so that you can really appreciate the importance of the 4 Pillars of Object Oriented Programming.

So the first question we need to ask ourselves is what classed do we need to create in order to extract our redundant code and eliminate it.

If we look at our characters, we can see that we have 5 common fields for all characters. Lets drop this number down to 4, we will leave **abilityPoints** for something else, don't worry about it for now.

So, **faction**, **name**, **level**, and **healthPoints** need to be extracted as redundant code.

The first base class that we are going to create will be *Character*. *Character* is more abstract than *Warrior* or *Mage* and it can have all of the different types of characters that we have under it.

So, simply create a "Character" class, and extract the fields/properties like so:

- *Character*
    - **faction**
    - **name**
    - **level**
    - **healthPoints**

One thing that you need to be cautious about here is your properties. Remember when we did the validation of our properties in part 2 of the project? Some characters have different validation for the same property, so extracting these properties will be a bit different.

The way to solve this problem would be to make these properties Virtual, if you remember, having a virtual property/method allows you to override it and give it different implementation in a derived class.

So, lets take the **HealthPoints** property for example, its validation may state that a *Mage* can have a maximum of 100 health, while a *Warrior* can have a maximum of 150 health.

We will create a "default" virtual **HealthPoints** property in the base class (*Character*), and override it in any class which has a different validation from the validation in the base class.

Now, I said that we will leave **abilityPoints** for later, because we will change that property a bit.

We will have **AbilityPoints** for *Melee* characters, and **ManaPoints** for *Spellcasters*. Which is nothing more than a reason to create two more base classes, *Melee* and *Spellcaster*. When we do this, we will have the following architecture:

- *Character*
    - *Melee*
        - **Warrior**
        - **Knight**
        - **Assassin**
    - *Spellcaster*
        - **Mage**
        - **Necromancer**
        - **Druid**

Character will be the "most base" class, and the characters themselves will be last on the line, so *Warrior* will inherit from *Melee* and *Melee* will inherit from *Character*. *Necromancer* will inherit from *Spellcaster* and *Spellcaster* will inherit from *Character*.

All that *Melee* and *Spellcaster* classes will have is a field/property called **abilityPoints/manaPoints** respectively. Feel free to add any other fields that you want to populate the *Melee* and *Spellcaster* classes.

These new Properties can and should be also virtual if you have different validation in each of the derived classes.

Tip: *In order to avoid rewriting the* ***abilityPoints*** *fields/properties in Mage, Necromancer and Druid classes, what you can do is use the CTRL+R+R shortcut to quickly rename them, if you have the same validation for all of these Characters, you can simply delete the* ***abilityPoints*** *and create* ***manaPoints*** *in the Spellcaster class, which will now be available to all derived classes.*

Part 6

Abstraction & Encapsulation

Allrighty, in this part of the project we are going to transform all of our base classes into Abstract.

This includes:

- Character
- Melee
- Spellcaster
- Weapon
- Sharp
- Blunt
- Armor
- Light
- Leather
- Heavy

They all are base classes and are not describing a specific object, but rather a "category of objects", which is also the reason why we don't need to create an instance of either of these classes, as you know abstract classes can't have instances.

As you know, this also allows us to have abstract methods, which do not need any implementation, all that they require is a signature – return type and input arguments.

The derived classes have the job of implementing these methods.

In our game example each character has two ways to attack and one way to defend. We can then abstract out these methods much like the way we are using the character class as an abstract way to classify each type of character in our game.

Whenever we want to enforce a given concept in more than one classes, in our case the 3 abilities, we have to create an Interface.

So lets create two interfaces **IAttacking** and **IDefending**, with the signature of these 3 methods within them, for **IAttacking:**

- **void** Attack();
- **void** SpecialAttack();

And for **IDefending:**

- **void** Defend();

After we implement these interfaces within our Character class, we will get these methods, but in order to leave their implementation to each specific character, they need to be abstracted. So within our Character class they will become:

- **public abstract void** Attack();
- **public abstract void** SpecialAttack();
- **public abstract void** Defend();

Unfortunately, we can't finish our job with these methods until we reach the next part of the project – Polymorphism, because this is what we need in order to fully abstract these methods and make them available in a proper way to all classes. This is also why we will keep this part very short, a lot of the things that we have to do won't make sense until you are familiar with Polymorphism.

Part 7

Polymorphism

Polymorphism is another concept that is important in object-oriented programming (OOP) which allows classes to have many forms

At this point, you should already expect that all characters that we are going to initialize in our game will be of base type Character:

```
Character warrior = new Warrior();
Character knight = new Knight();
Character assassin = new Assassin();
Character mage = new Mage();
Character necromancer = new Necromancer();
Character druid = new Druid();
```

You know that Polymorphism allows a derived type to behave as a base type.

We will have all of these characters distributed into two lists:

- **List<Character>** meleeTeam = new List<Character>(); that will hold all Melee characters
- **List<Character>** spellTeam = new List<Character(); for the spellcasters

Tip: *If you want to be a little more specific, instead of using Character type, you could use Melee and Spellcaster for the different lists of teams. Keep in mind that if you decide to do this, you would also need to change the base type of all characters in the initialization process before the Lists – Melee warrior = new Warrior(); - and so on for the rest of them.*

If we want one of the characters from the melee team to attack one of the characters from the spellteam, we would need to have a lot of "if"s to find out what type of character that is so that we can use the proper ability, we would have a code looking a lot like this:

```
if (meleeTeam[1] is Warrior)
{
        tempCharacter = meleeTeam[1] as Warrior;
        tempCharacter.Execute(spellTeam[0]);
}
else if (meleeTeam[1] is Knight)
{
        tempCharacter = meleeTeam[1] as Knight;
        ……..
}
else if (meleeTeam[1] is Assassin) ……
```

So we would have three ifs, to determine exactly what Character we have at hand in order to pick the correct ability, and to make matters worse, we would need to have another set of ifs after that to determine of what type the character from spellTeam is so that we can use the proper defensive ability. Additional we even need a temporary character variable so that we can typecast the character in order to use the specific methods.

To solve this problem, we would have to use the Attack, SpecialAttack and Defend abstract methods that we created in the previous part of the project. We will implement them in a very simple manner within our individual characters.

We are going to make all our specific abilities methods private (Strike, Execute, Fireball, HolyBlow etc)

And within each of the implementations of the 3 abstract methods, we are simply going to call the inner methods, it will look like this:

```
public override void Attack()
{
        this.Strike();
}
public override void SpecialAttack()
{
        this.Execute();
}
public override void Defend()
{
        this.SkinHarden();
}
```

Tip: *Don't do this for all of your characters yet, because we have yet to determine exactly what the logic will be behind these methods, so we may need to add some arguments, change return types and if that happens you will need to rewrite code (unless you go with your own implementations, which is perfectly ok).*

It would be the same for all the other characters, so now all of the characters still have their individual abilities, but they are under the exact same name AND are shared by the base Character class.

We no longer have to guess of what type is our current character, all we have to do now is say:

```
meleeTeam[1].Attack();
spellTeam[0].Defend();
```

Since these abilities originate from the Character class, they will be usable, but each of them has its own implementation within each of the specific characters.

Mind you, its not necessary to have both Attack and Strike methods and call the Strike methods in the Attack method, the only reason I am doing this is for readability, you could have just as easily copied the code from the Strike method inside the implementation of the Attack method and deleted the Strike method, it would have worked the exact same way.

There is one additional way in which we can use Polymorphism to eliminate more redundant code.

Polymorphism can help us extract the Weapon and Armor properties out of the specific characters and move them too in the Character base class.

We would simply create these fields/properties with their base type Armor/Weapon, and within the constructors of our classes all that we would need to do is:

- For a Warrior: **base.**BodyArmor = **new** Chainlink();
- For a Mage: **base.**BodyArmor = **new** ClothRobe();

Both Chainlink and ClothRobe are of base type Armor, and since the properties are declared as Armor types in our Character class that is perfectly fine, essentially we are saying:

Armor bodyArmor = **new** Chainlink();

More redundant code eliminated.

From here on, the OOP part of our project is pretty much completed, now we have to populated our methods with logic and make the teams fight with each other.