Exercise: Inheritance and Abstraction

This document defines an in-class exercise from the "OOP" Course @ Software University.

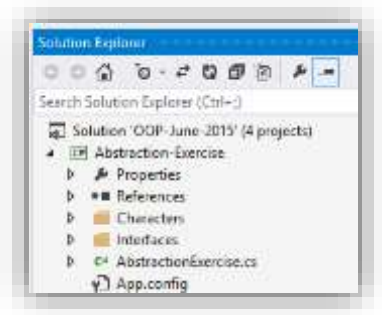
Problem 2. Working with Abstraction

Abstraction allows us to define properties and behavior in a **general** way and leave the implementation details to descendent classes. In your teamwork project, you'll need to use abstraction, so let's practice it.

In an RPG game there are different types of characters – mages, warriors, priests, etc. They have some common characteristics, so it's a good idea to have a **base** class which describes these common characteristics and let the descendants reuse them through **inheritance**. There are also a set of actions the characters can perform; actions are usually defined in **interfaces**.

Step 1. Create Basic Structure

For this simple task, add two folders (these will be the namespaces in the application) – Interfaces and Characters.



Step 2. Define Interfaces

To keep things simple, let's assume that characters can attack each other and not much else.

Define an interface **IAttack** which holds one method – Attack. The Priest is a special character, he can attack, but also **heal** other characters. This calls for another interface **IHeal**, which should contain a method Heal. Both the Heal and Attack methods should accept a target as a parameter; the target should be of type Character (which we'll create in the next step). Example (IAttack interface):

```
using Characters;

public interface IAttack
{
    void Attack(Character target);
}
```

Step 3. Create Abstract Class Character

We'll have three types of characters which have some commonalities, but also differences (the priest can heal, other characters cannot). Raising the level of abstraction means we can **extract all common features** of the three specific classes into a **base** class called **Character**. It's not logical for the user to directly instanciate something non-specific like a Character, therefore the class should be declared **abstract**.

All characters should be able to attack, so the Character class **implements** the **IAttack** interface. Character should either implement the Attack method or leave it abstract for its descendants to implement. In this case, leave the Attack method **abstract**.

A character should have **properties** Health, Mana and Damage. Add a **constructor** which takes the initial values for health, mana and damage and sets them. Since the constructor will be used only by the child classes, its access modifier should be **protected** as a rule.

Example:

```
using Interfaces;

public abstract class Character : IAttack
{
    protected Character(int health, int mana, int damage)
    {
        this.Health = health;
        this.Mana = mana;
        this.Damage = damage;
    }

    public int Health { get; set; }

    public int Mana { get; set; }

    public int Damage { get; set; }

    public abstract void Attack(Character target);
}
```

Step 4. Create Concrete Classes

Now that we have an abstract class, we can create the child classes. Add classes Mage, Warrior and Priest.

The Mage has initial health of 100, initial mana of 300 and damage of 75. The Warrior has initial health of 300, initial mana of 0 and damage of 120. The Priest has initial health of 125, initial mana of 200 and damage of 100.

The child classes can reuse the parent class's properties and constructors. In this case, all we need to do is call the **base constructor** by providing the initial values for health, mana and damage for each class like this:

```
public class Mage : Character
{
    public Mage()
        : base(100, 300, 75)
    {
    }
```

The Priest is special. Besides attacking, he can also heal. Therefore, the Priest should implement the **IHeal** interface.

Step 5. Implement Interface Methods

In the Character class we left the Attack method unimplemented (**abstract**). This allows us to define (**override**) the method in the child classes so that each character attacks in a different way.

Let's say that when the Warrior attacks, he deals damage to the opponent. But when the Mage attacks, he uses 100 mana and deals twice his default damage. When the Priest attacks, he deals damage to the opponent, but also heals himself (life steal) – e.g. if he dealt 100 damage to the opponent, he restores 10% of that, or 10 health points.

Example (Mage attack method):

```
public class Mage : Character
{
    public Mage()
        : base(100, 300, 75)
    {
    }

    public override void Attack(Character target)
    {
        this.Mana -= 100;
        target.Health -= 2 * this.Damage;
    }
}
```

When the healer Heals, he reduces his mana by 100 and heals the target character by increasing his health with 150 health points. Example:

```
public class Priest : Character, IHeal
{
    public Priest()
        : base(125, 200, 100)
    {
    }

    public override void Attack(Character target)
    {
        this.Mana -= 100;
        target.Health -= this.Damage;
        this.Health += this.Damage / 10;
    }

    public void Heal(Character target)
    {
        this.Mana -= 100;
        target.Health += 150;
    }
}
```

Step 6. View Class Hierarchy

Create a **class diagram** by right-clicking the project and selecting **View -> View Class Diagram**. Check out the relationships between the classes, see if you made a mistake (especially useful in larger projects like your teamwork). Example: