Lab: Mass Effect

This document defines a lab assignment from the "OOP" Course @ Software University.

Mass Effect is a video game where **starships** can **travel from one star system to another** and **attack other ships**. You are given a skeleton (partially written code) for the game as well as a problem description. Your task is to finish the game by applying the best practices of OOP. You can test your solution in the automated Judge system here.

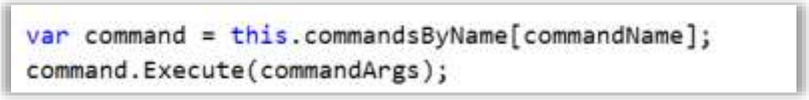
Step 1 - Read the Problem Description

Read the given problem description to better understand the problem.

Step 2 - Study the Code

Being able to find your way around someone else's code is a very important skill. Let's study the provided classes in the skeleton one by one:

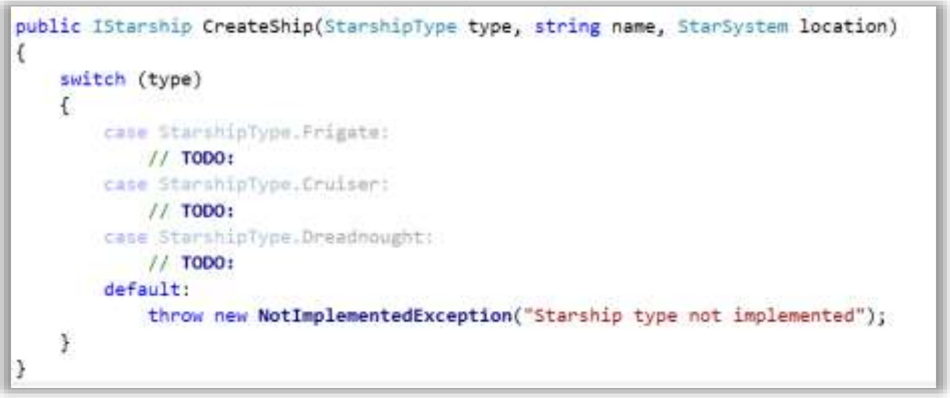
- Engine
  - **GameEngine** - implements the IGameEngine interface. The **`Run()`** method starts a while loop until its **IsRunning** property yields false. It reads a **string command** from the console and passes it to the **`ProcessCommand()`** method of its CommandManager. A **`try-catch`** block ensures that any **ShipException** that might occurr in **`ProcessCommand()`** will be properly handled.
  - **CommandManager** - keeps all commands in a dictionary where the key is the command as a string (e.g. "attack") and the value is the instance of the corresponding command (e.g. **`new AttackCommand()`**);
    `ProcessCommand()` retrieves a command from the dictionary and calls its **`Execute()`** method:



```
var command = this.commandsByName[commandName];
command.Execute(commandArgs);
```

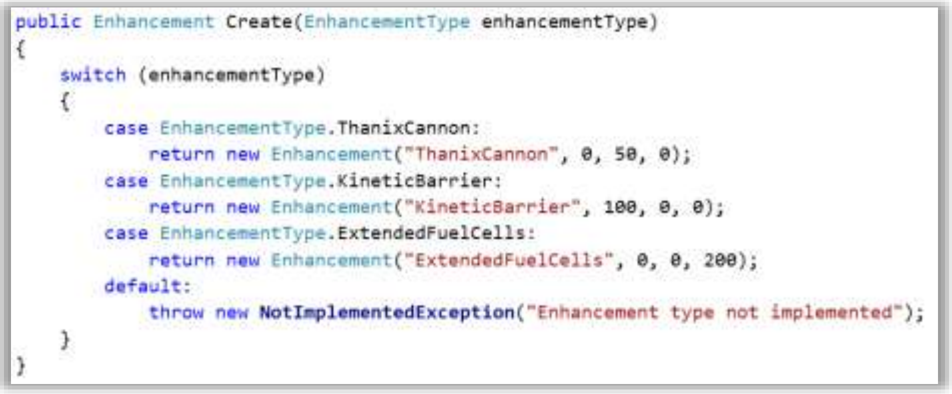
  - Commands - **none** of the commands are implemented
    - **Command** - base class for any command; holds a reference to an IGameEngine and an empty **`Execute()`** method.
    - **AttackCommand** - encapsulates logic for executing an attack
    - **CreateCommand** - encapsulates logic for creating a ship
    - **OverCommand** - encapsulates logic for stopping the game engine
    - **PlotJumpCommand** - encapsulates logic for changing the location of a ship
    - **StatusReportCommand** - encapsulates logic for displaying info about a ship
  - Factories

        - **ShipFactory -** holds method **`CreateShip()`** that creates a ship, given a type, name and location



```
public IStarship CreateShip(StarshipType type, string name, StarSystem location)
{
    switch (type)
    {
        case StarshipType.Frigate:
            // TODO:
        case StarshipType.Cruiser:
            // TODO:
        case StarshipType.Dreadnought:
            // TODO:
        default:
            throw new NotImplementedException("Starship type not implemented");
    }
}
```

        - **EnhancementFactory -** holds method **`Create()`** that creates an enhancement, given a type



```
public Enhancement Create(EnhancementType enhancementType)
{
    switch (enhancementType)
    {
        case EnhancementType.ThanixCannon:
            return new Enhancement("ThanixCannon", 0, 50, 0);
        case EnhancementType.KineticBarrier:
            return new Enhancement("KineticBarrier", 100, 0, 0);
        case EnhancementType.ExtendedFuelCells:
            return new Enhancement("ExtendedFuelCells", 0, 0, 200);
        default:
            throw new NotImplementedException("Enhancement type not implemented");
    }
}
```

    - **Messages** - a static class that holds **messages** as public **constants**, available for use in the entire application
- Exceptions - the namespace holds **custom exception classes**
    - **ShipException**
    - **InsufficientFuelException**
    - **LocationOutOfRangeException**
- GameObjects
    - Enhancements
        - **Enhancement** - holds **ShieldBonus**, **DamageBounus** and **FuelBonus**
        - **EnhancementType** - enumeration that holds the 3 possible enhancements - **ThanixCannon**, **KineticBarrier**, **ExtendedFuelCells**
    - Locations
        - **StarSystem** - holds name and a dictionary with all neighboring star systems, where the key is a **reference to another star system** and value is the **fuel required** to travel there (e.g. ArtemisTau -> 120).
    - Ships
        - **StarshipType** - enumeration that holds the 3 possible starship types - **Frigate**, **Cruiser** and **Dreadnought**
    - Projectiles - empty namespace left for future projectile implementations
    - **Galaxy** - class that holds a **set of all star systems** and 2 methods: **`GetStarSystemByName()`** and **`TravelTo()`**
- Interfaces

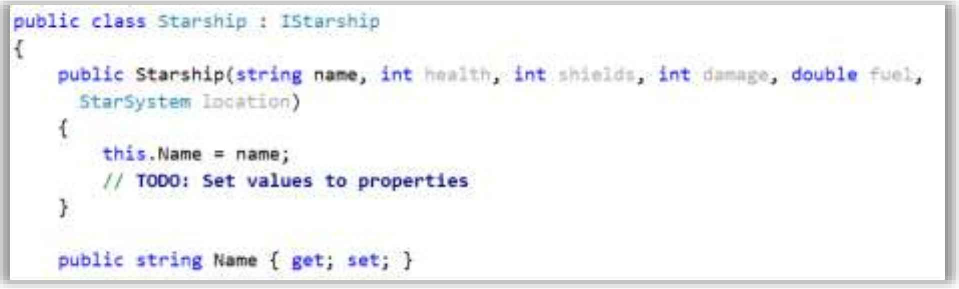
- ICommandManager - defines what a CommandManager should have - hold a reference to a **`GameEngine`**, **`ProcessCommand()`** method for processing individual commands and **`SeedCommands()`** method for initializing the dictionary with commands.
- IEnhanceable - defines something that can be enhanced (meaning upgraded) - holds **`IEnumerable<Enhancements>`** (a read-only collection of enhancements) and **`AddEnhancement()`** method
- IGameEngine - defines a GameEngine (see the interface for more info)
- IProjectile - defines a Projectile - holds **Damage** and **`Hit()`** method
- IStarship - defines a Starship (see the interface for more info). Extends **IEnhanceable**.

Take your time and study the provided code well before proceeding with the next tasks.

Step 3 - Implement Ships

The whole game depends on the 3 ship types. Let's implement classes for them! Create 3 classes in the Ships namespace - **Frigate**, **Cruiser** and **Dreadnought**. All three ships have Name, Health, Shields, Damage, Fuel, Location, enhancements and methods: **`ProduceAttack()`**, **`RespondToAttack()`**, **`AddEnhancement()`**.

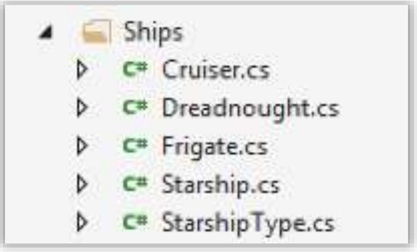
In other words, we have **common properties** and **methods**. Let's extract those common members in a **base** class - **Starship**. It should implement the **IStarship** interface (since it defines the behavior of a starship, and our class is one).

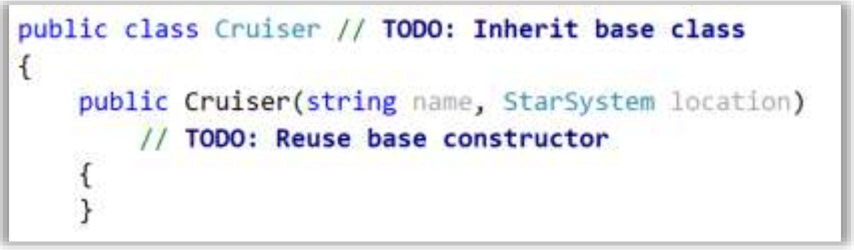


```
public class Starship : IStarship
{
    public Starship(string name, int health, int shields, int damage, double fuel,
      StarSystem location)
    {
        this.Name = name;
        // TODO: Set values to properties
    }

    public string Name { get; set; }
```

Implement the missing members of that interface (properties, methods). Think about **if the class should be allowed to be instantiated** (tip: **abstraction**). In that sense, should the constructor of Starship be **public**?

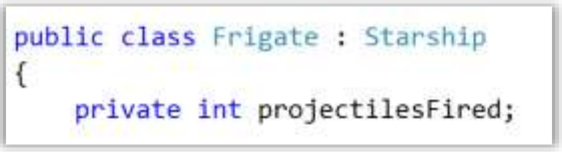
Implement concrete classes - **Frigate**, **Cruiser** and **Dreadnought**.





```
public class Cruiser // TODO: Inherit base class
{
    public Cruiser(string name, StarSystem location)
        // TODO: Reuse base constructor
    {
    }
```

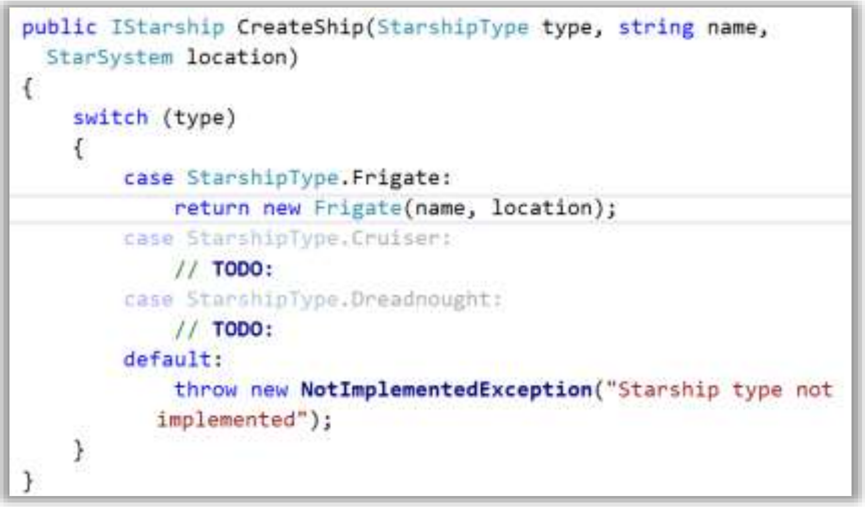
Notice the problem description states that the **Frigate** should also keep **count of all projectiles fired** (we'll talk more about later). For now, just create a field **projectilesFired** in the frigate class for keeping track of all fired porjectiles.



```
public class Frigate : Starship
{
    private int projectilesFired;
```

Step 4 - Implement TODOs in ShipFactory

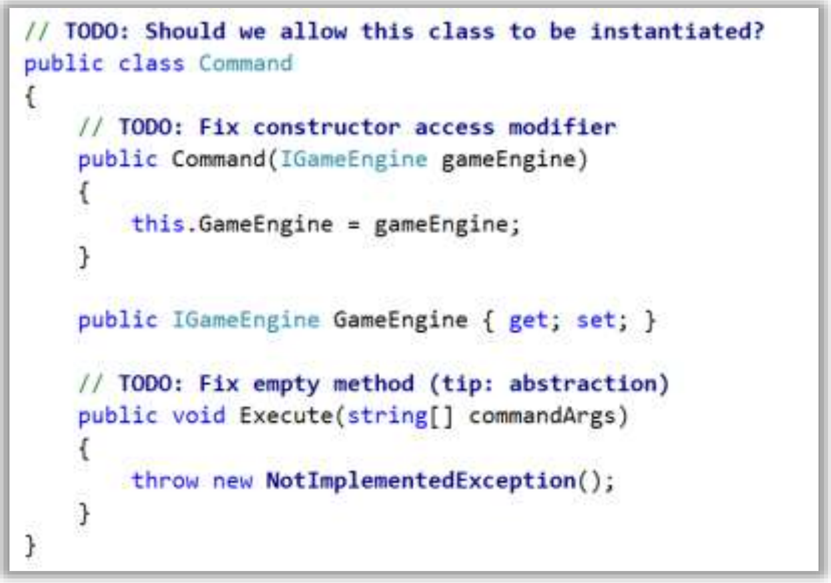
Now that we have classes for each ship in our game, let's implement the missing parts of the **ShipFactory**.



```
public IStarship CreateShip(StarshipType type, string name,
  StarSystem location)
{
    switch (type)
    {
        case StarshipType.Frigate:
            return new Frigate(name, location);
        case StarshipType.Cruiser:
            // TODO:
        case StarshipType.Dreadnought:
            // TODO:
        default:
            throw new NotImplementedException("Starship type not
          implemented");
    }
}
```

Step 5 - Create Command

It's time we implemented our first command - **create**. The Commands namespace contains several commands, all of which inherit the base **Command** class. However, there are 3 things wrong with it.

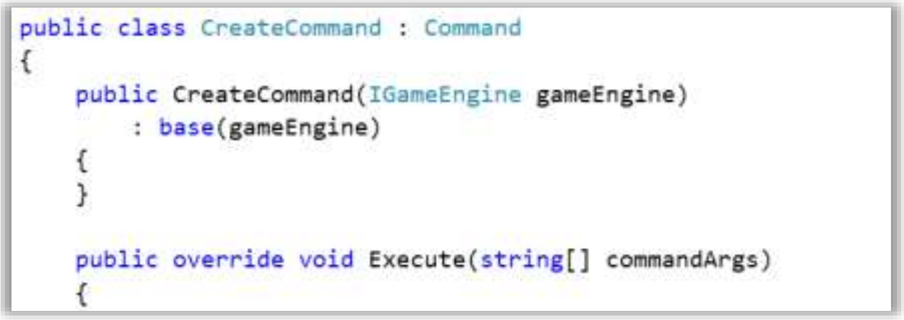


```
// TODO: Should we allow this class to be instantiated?
public class Command
{
    // TODO: Fix constructor access modifier
    public Command(IGameEngine gameEngine)
    {
        this.GameEngine = gameEngine;
    }

    public IGameEngine GameEngine { get; set; }

    // TODO: Fix empty method (tip: abstraction)
    public void Execute(string[] commandArgs)
    {
        throw new NotImplementedException();
    }
}
```

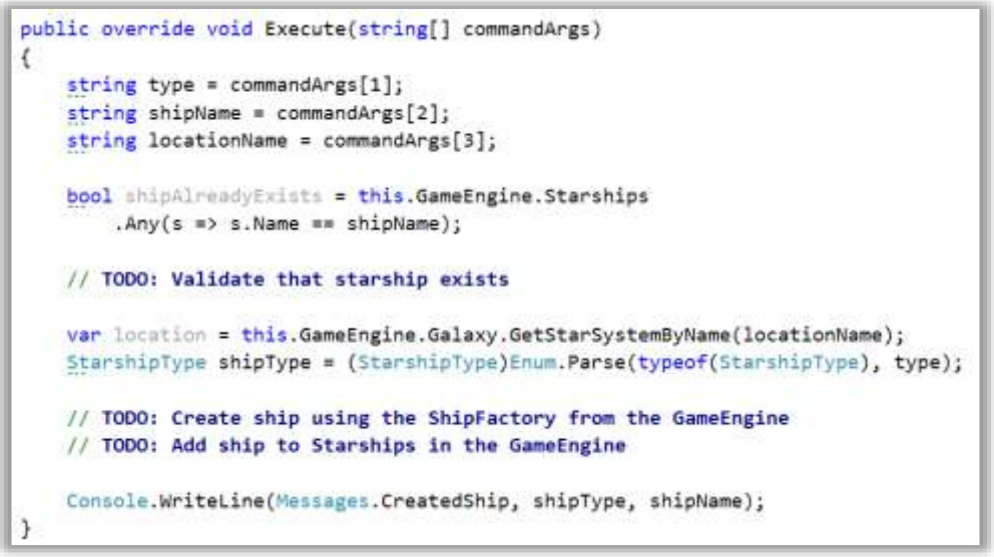
Now that we've fixed the base Command class, it's time we started implementing our concrete commands.

As we already saw, each command's **`Execute()`** method is called whenever a command string is entered by the user. Obviously, each command class will implement that method differently.

In order for a descendant class to change a method, it needs to **override** it.



```
public class CreateCommand : Command
{
    public CreateCommand(IGameEngine gameEngine)
        : base(gameEngine)
    {
    }

    public override void Execute(string[] commandArgs)
    {
```

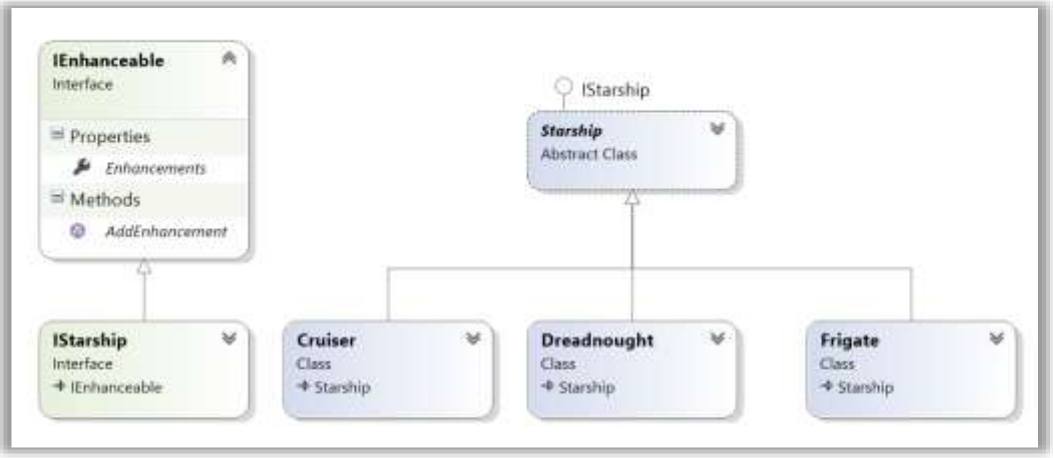


```
public override void Execute(string[] commandArgs)
{
    string type = commandArgs[1];
    string shipName = commandArgs[2];
    string locationName = commandArgs[3];

    bool shipAlreadyExists = this.GameEngine.Starships
        .Any(s => s.Name == shipName);

    // TODO: Validate that starship exists

    var location = this.GameEngine.Galaxy.GetStarSystemByName(locationName);
    StarshipType shipType = (StarshipType)Enum.Parse(typeof(StarshipType), type);

    // TODO: Create ship using the ShipFactory from the GameEngine
    // TODO: Add ship to Starships in the GameEngine

    Console.WriteLine(Messages.CreatedShip, shipType, shipName);
}
```

If everything is correct, the **Create** command should successfully create a ship and add it to the Starships collection in the engine.

Step 6 - Implement AddEnhancement()

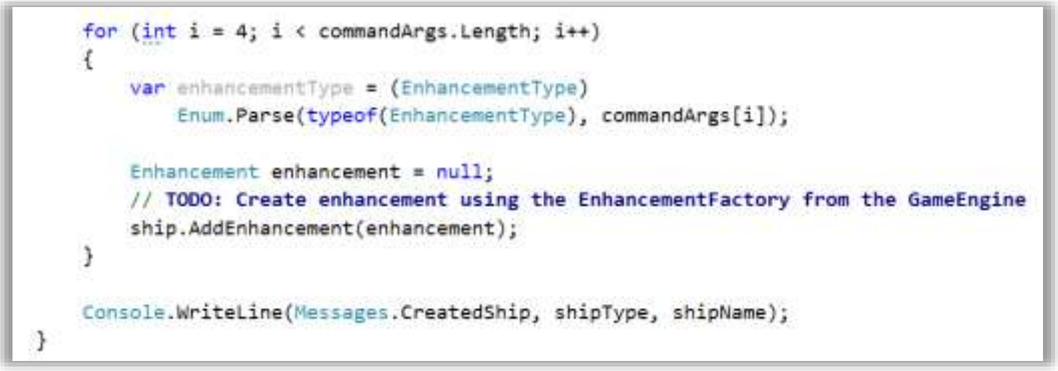
The starships we made (**Frigate**, **Cruiser** and **Dreadnought**) inherit **Starship**. Starship implements the **IStarship** interface. **IStarship** extends the **IEnhanceable** (which defines behavior for holding enhancements and adding new ones). Therefore, all ships are **enhanceable**.



Since all ships are **IEnhanceable**, they have **AddEnhancement()** method.

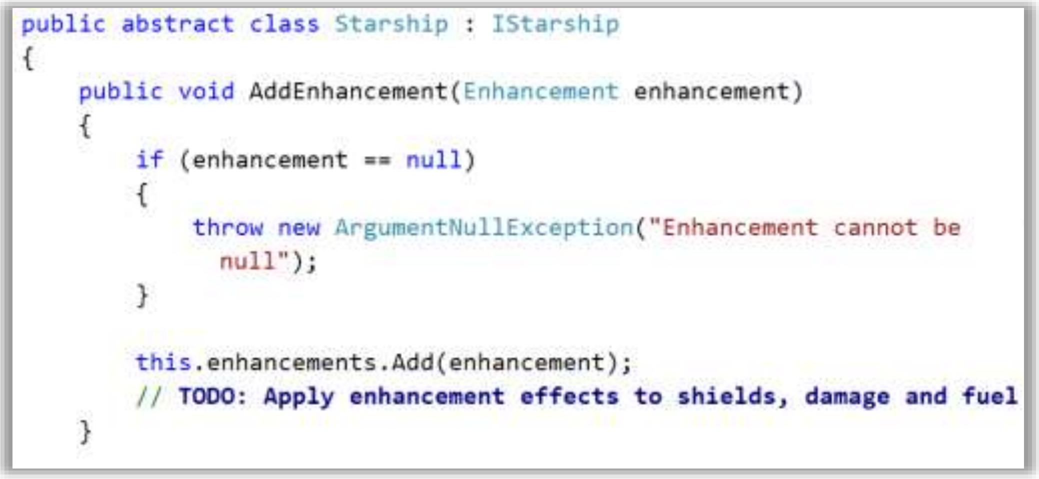
Judging from the **Create** command, all arguments after the 4th will be enhancements (e.g. **`create {shipType} {shipName} {starSystem} {enhancement1 enhancements2 ...}`**).

Let's go back to the **CreateCommand** class and add enhancements after we've created a ship.



```
        for (int i = 4; i < commandArgs.Length; i++)
        {
            var enhancementType = (EnhancementType)
                Enum.Parse(typeof(EnhancementType), commandArgs[i]);

            Enhancement enhancement = null;
            // TODO: Create enhancement using the EnhancementFactory from the GameEngine
            ship.AddEnhancement(enhancement);
        }

        Console.WriteLine(Messages.CreatedShip, shipType, shipName);
    }
```

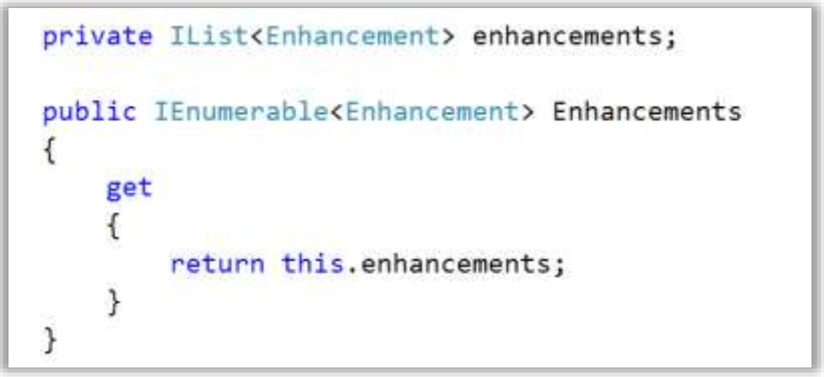
So far so good, but we need to implement the **`AddEnhancement()`** method. The question is - where? In Frigate? In Dreadnought? No, in **Spaceship** - it is the base class for all ships so each ship will reuse it.



```
public abstract class Starship : IStarship
{
    public void AddEnhancement(Enhancement enhancement)
    {
        if (enhancement == null)
        {
            throw new ArgumentNullException("Enhancement cannot be
              null");
        }

        this.enhancements.Add(enhancement);
        // TODO: Apply enhancement effects to shields, damage and fuel
    }
```

But, **enhancements** is **`IEnumerable<Enhancement>`**. How do we add an element to an **IEnumerable** collection? We don't.

Internally we keep the enhancements as **`private IList<Enhancement>`** (a collection that allows adding elements) and add to that collection. But we reveal it as **`IEnumerable<Enhancement>`** (a collection that can only be iterated).



```
private IList<Enhancement> enhancements;

public IEnumerable<Enhancement> Enhancements
{
    get
    {
        return this.enhancements;
    }
}
```

Why is this allowed? Because **IList<T>** extends **IEnumerable<T>** and thanks to **polymorphism** we can reveal a more concrete object as a more abstract one.

Note: That way nobody from outside the class can add/remove enhancements, because the collection is revealed as **IEnumerable**. Inside the class, however, we can work with **IList** and we can add/remove elements.

Step 7 - Status Report Command

Just like **`CreateCommand`**, **`StatusReportCommand`** should inherit the base Command class and override its **`Execute()`** method.

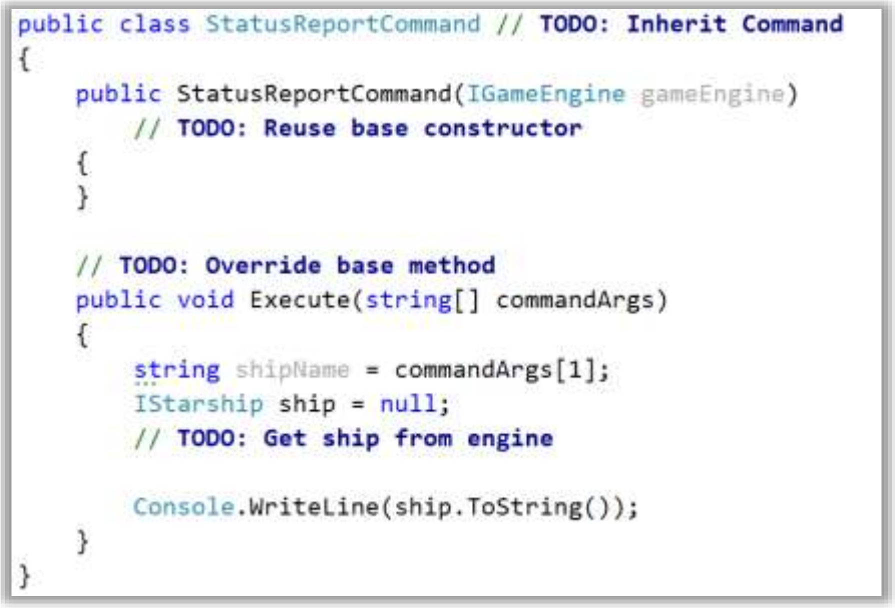
It should print information about a given ship in the following format:

If `health > 0`	If `health <= 0`

`--{shipName} - {shipType}`
`-Location: {locationName}`
`-Health: {health}`
`-Shields: {shields}`
`-Damage: {damage}`
`-Fuel: {fuel}`
`-Enhancements: {enh1, enh2, ...}`
	`--{shipName} - {shipType}`
	`(Destroyed)`

The format varies depending on the ship's health. Let's implement the command:

1. Inherit the **Command** class
2. Reuse the base constructor to avoid code repetition
3. Override the abstract **`Execute()`** method:
    - Get the ship from the engine by name
    - Print the ship to the console by calling its **`ToString()`** method

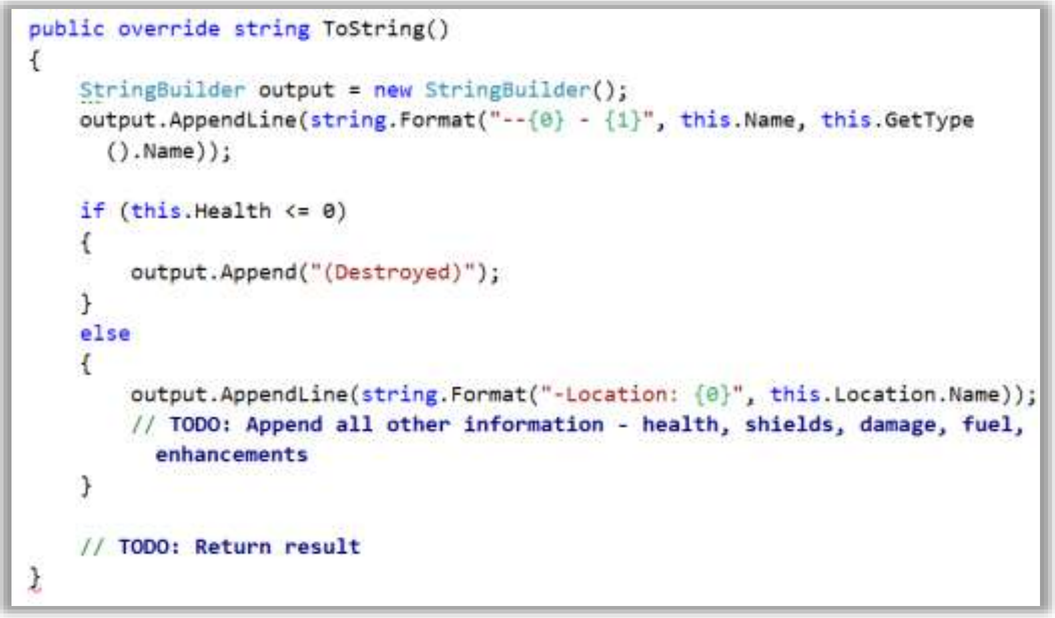


```
public class StatusReportCommand // TODO: Inherit Command
{
    public StatusReportCommand(IGameEngine gameEngine)
        // TODO: Reuse base constructor
    {
    }

    // TODO: Override base method
    public void Execute(string[] commandArgs)
    {
        string shipName = commandArgs[1];
        IStarship ship = null;
        // TODO: Get ship from engine

        Console.WriteLine(ship.ToString());
    }
}
```

Obviously the ship's **`ToString()`** method must return information about the ship.

However, by default **`ToString()`** returns the class' type. We want it to return ship info. Fortunately for us, **`ToString()`** is a virtual method - therefore we can override it and change its behavior for our needs.

Let's go to the base **Starship** class and override its `ToString()` method. By overriding a method in a parent class, all child classes also inherit the overriden method.



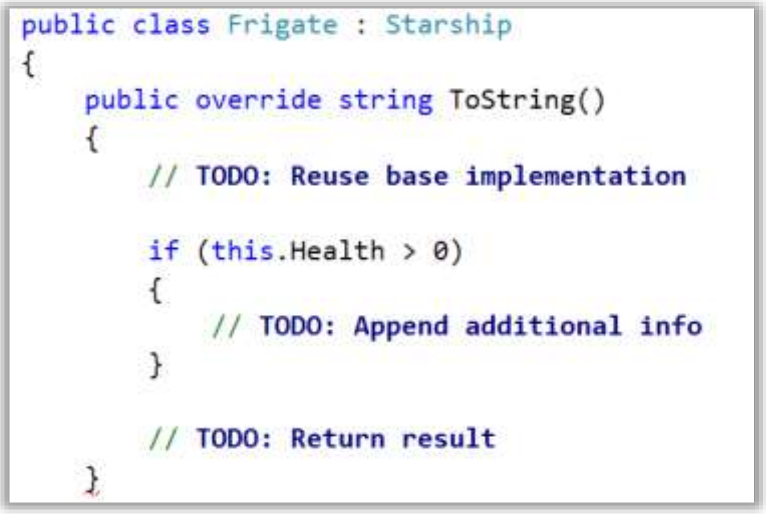
```
public override string ToString()
{
    StringBuilder output = new StringBuilder();
    output.AppendLine(string.Format("--{0} - {1}", this.Name, this.GetType
      ().Name));

    if (this.Health <= 0)
    {
        output.Append("(Destroyed)");
    }
    else
    {
        output.AppendLine(string.Format("-Location: {0}", this.Location.Name));
        // TODO: Append all other information - health, shields, damage, fuel,
          enhancements
    }

    // TODO: Return result
}
```

Thanks to **inheritance** the **Frigate**, **Cruiser** and **Dreadnought** classes inherit the above method implementation and we **avoid code repetition**!

However, there is one more thing: If the ship is a **frigate** (and not destroyed), it should also display the number of projectiles fired so far in the format:

```
-Projectiles fired: {count}
```

We need to override the **Frigate's ToString()** method too and add that additional line. Make sure you do not repeat any code by reusing the base (Starship) implementation!



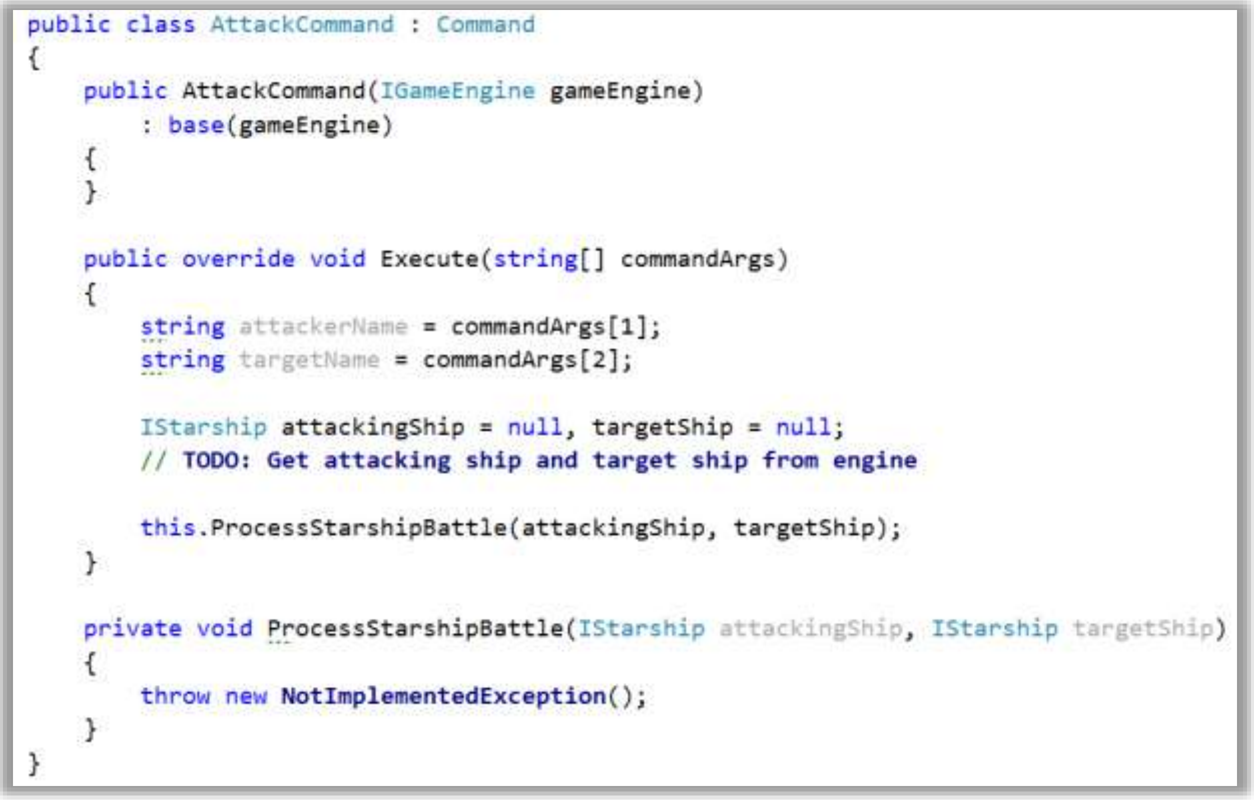
```
public class Frigate : Starship
{
    public override string ToString()
    {
        // TODO: Reuse base implementation

        if (this.Health > 0)
        {
            // TODO: Append additional info
        }

        // TODO: Return result
    }
```

Step 8 - Attack Command

Time to implement the attack command. Let's go to the **AttackCommand** class and override the `Execute()` method.

1. We get the **attacker ship name** and **target ship name** from the command arguments

2. Then we get the **ships** with those **names** from the engine
3. Finally, we pass the 2 ships to our **`ProcessStarshipBattle()`** method (notice how it's **`private`** because there is **no need** for the method to be visible to the outside world)

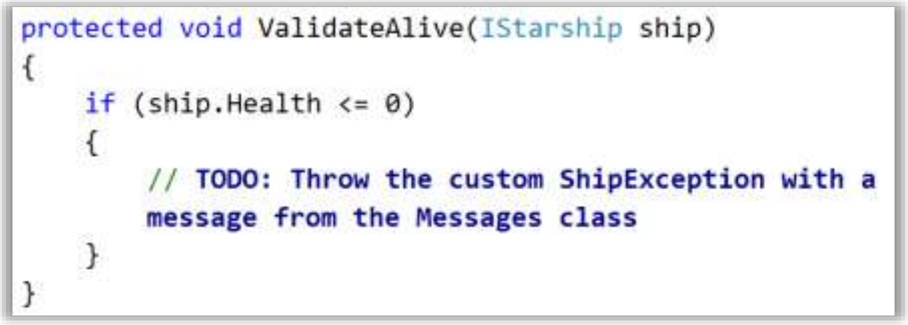


```
public class AttackCommand : Command
{
    public AttackCommand(IGameEngine gameEngine)
        : base(gameEngine)
    {
    }

    public override void Execute(string[] commandArgs)
    {
        string attackerName = commandArgs[1];
        string targetName = commandArgs[2];

        IStarship attackingShip = null, targetShip = null;
        // TODO: Get attacking ship and target ship from engine

        this.ProcessStarshipBattle(attackingShip, targetShip);
    }

    private void ProcessStarshipBattle(IStarship attackingShip, IStarship targetShip)
    {
        throw new NotImplementedException();
    }
}
```

The **`ProcessStarshipBattle()`** method should do the following things:

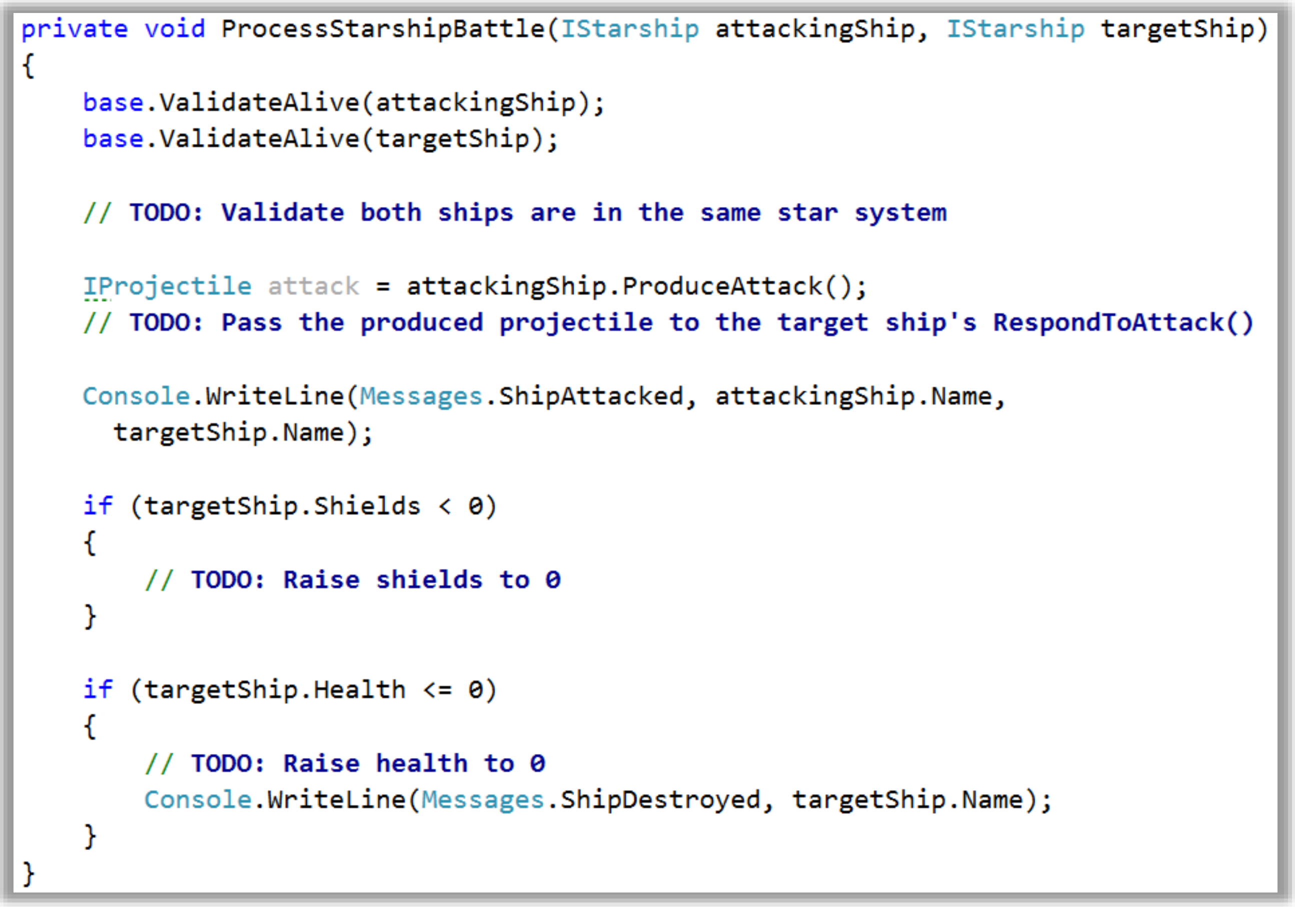
1. Validate that the 2 ships have **not been destroyed** (are still alive)
2. Validate the two ships are in the **same star system** (by rules, a ship cannot ships in other star systems)
3. The attacking ship's **`ProduceAttack()`** method should produce a projectile
4. The target ship should take the projectile using its **`RespondToAttack()`** method
5. Finally, check if the target ship's health or shields has fallen **below 0** and **raise them back to 0**

First, let's create method that validates whether a ship is alive (not destroyed).



```
protected void ValidateAlive(IStarship ship)
{
    if (ship.Health <= 0)
    {
        // TODO: Throw the custom ShipException with a
        message from the Messages class
    }
}
```

Think about where you should place this method - it will be used by several commands later (not only the **`AttackCommand`** class).



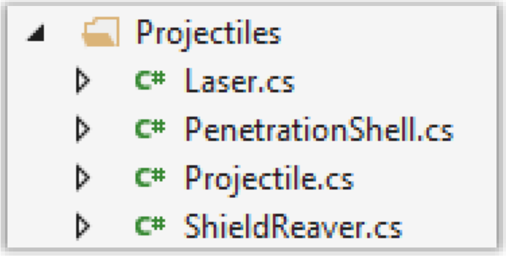
```
private void ProcessStarshipBattle(IStarship attackingShip, IStarship targetShip)
{
    base.ValidateAlive(attackingShip);
    base.ValidateAlive(targetShip);

    // TODO: Validate both ships are in the same star system

    IProjectile attack = attackingShip.ProduceAttack();
    // TODO: Pass the produced projectile to the target ship's RespondToAttack()

    Console.WriteLine(Messages.ShipAttacked, attackingShip.Name,
      targetShip.Name);

    if (targetShip.Shields < 0)
    {
        // TODO: Raise shields to 0
    }

    if (targetShip.Health <= 0)
    {
        // TODO: Raise health to 0
        Console.WriteLine(Messages.ShipDestroyed, targetShip.Name);
    }
}
```

If all is right, the attack command should be ready. But it cannot work correctly until we implement each ship's (Frigate, Cruiser and Dreadnought) **`ProduceAttack()`** and **`RespondToAttack()`** methods.

Step 9 - Implement `ProduceAttack()`

Every ship has a **`ProduceAttack()`** method (as defined by the **`IStarship`** interface). The method's return type is **`IProjectile`** - this suggests it must return the projectile of the attacking ship. The problem description states that every ship fires a different projectile - **Penetration Shell**, **Shield Reaver** or **Laser**. Let's implement the projectiles.

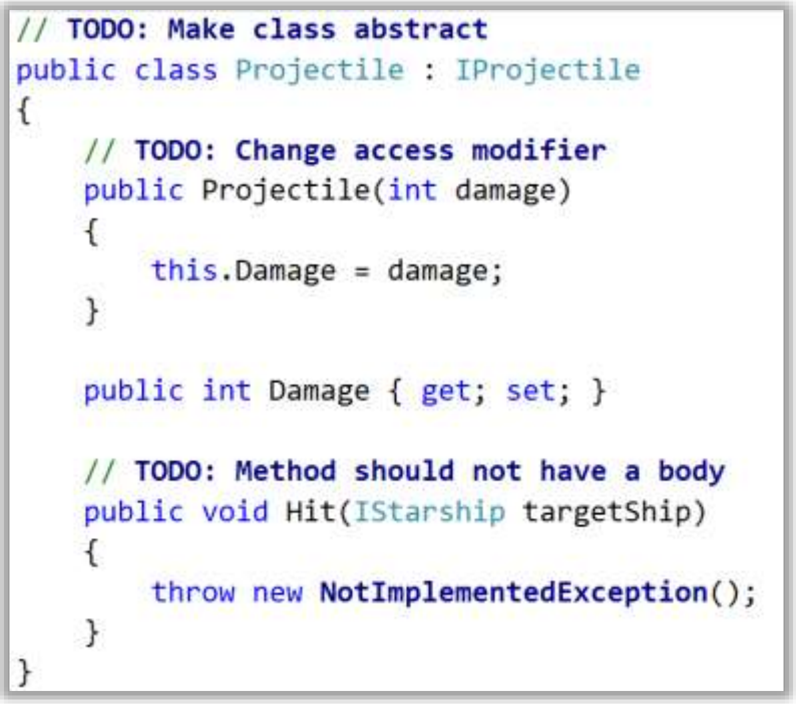
Create a base class **Projectile** and a class for each of the actual projectiles - **Laser**, **PenetrationShell** and **ShieldReaver**.



```
◢ Projectiles
    ▷ C# Laser.cs
    ▷ C# PenetrationShell.cs
    ▷ C# Projectile.cs
    ▷ C# ShieldReaver.cs
```

Projectile should serve as a **base class** for all projectiles and should **not allow to be instantiated**. It should also implement the **`IProjectile`** interface.

The interface defines that all projectile should have **damage** (passed by the firing ship) and **`Hit()`** method.



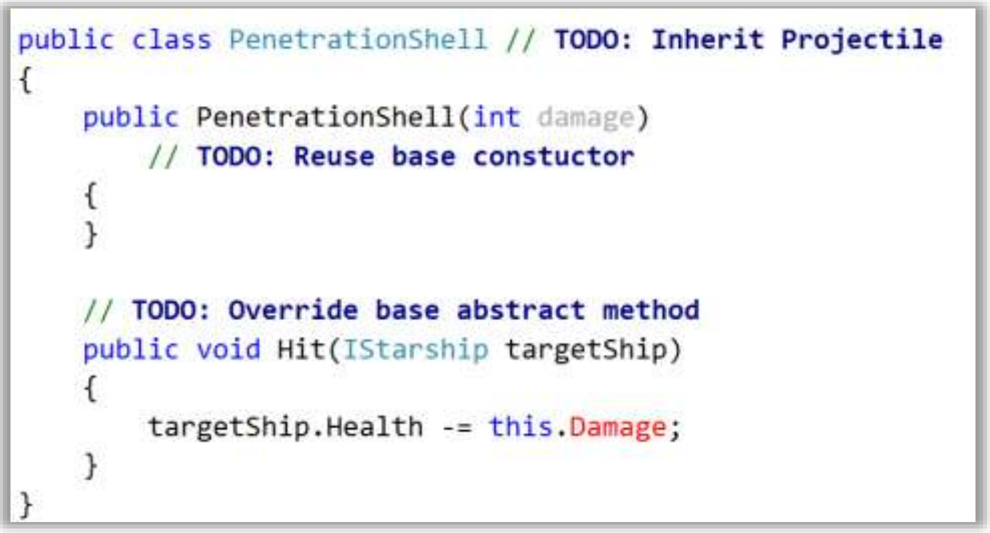
```
// TODO: Make class abstract
public class Projectile : IProjectile
{
    // TODO: Change access modifier
    public Projectile(int damage)
    {
        this.Damage = damage;
    }

    public int Damage { get; set; }

    // TODO: Method should not have a body
    public void Hit(IStarship targetShip)
    {
        throw new NotImplementedException();
    }
}
```

Let's take a look at what each projectile's **`Hit()`** method should do:

- **`PenetrationShell`** - removes **health** from the ship equal to the projectile's **damage**.
- **`ShieldReaver`** - removes **health** from the ship equal to **the projectile's damage**. It also removes **shields** from the ship equal to **2x the projectile's damage**.
- **`Laser`** - removes **shields** from the ship equal to the projectile's **damage**. If the damage is more than the ship's shields, it also takes health equal to the damage left. (e.g. **50 shields** and **100 health** - a laser of **80 damage** would remove **50** shields and **30** health, resulting in **0 shields** and **70 health** for the ship).

Let's start with the **`PenetrationShell`**:

1. Create a **PenetrationShell** class and **inherit** the **base Projectile** class
    a. Reuse the base constructor
2. Override the abstract **`Hit()`** method. It should **subtract health** from the **hit ship** equal to the **projectile's own damage**



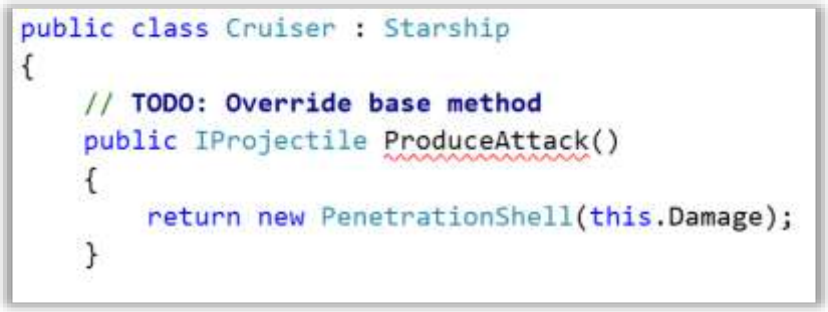
```
public class PenetrationShell // TODO: Inherit Projectile
{
    public PenetrationShell(int damage)
        // TODO: Reuse base constuctor
    {
    }

    // TODO: Override base abstract method
    public void Hit(IStarship targetShip)
    {
        targetShip.Health -= this.Damage;
    }
}
```

Do the same for each of the other **Projectile** classes - **ShieldReaver** and **Laser**.

Now that we have the projectile's available, it's time we implemented the **`ProduceAttack()`** method for each of our ships.

Ship	Projectile	Description
Frigate	PenetrationShell	Shoots a **`ShieldReaver`** with damage equal to its **own damage**.
Cruiser	ShieldReaver	Shoots a **`PenetrationShell`** with damage equal to its **own damage**.
Dreadnought	Laser	Shoots a **`Laser`** with damage equal to **half its shields + own damage**.

For example, a **`Cruiser`** should produce a **`PenetrationShell`** with damage equal to its own damage.



```
public class Cruiser : Starship
{
    // TODO: Override base method
    public IProjectile ProduceAttack()
    {
        return new PenetrationShell(this.Damage);
    }
```

Follow the table above and do the same for other ships as well.

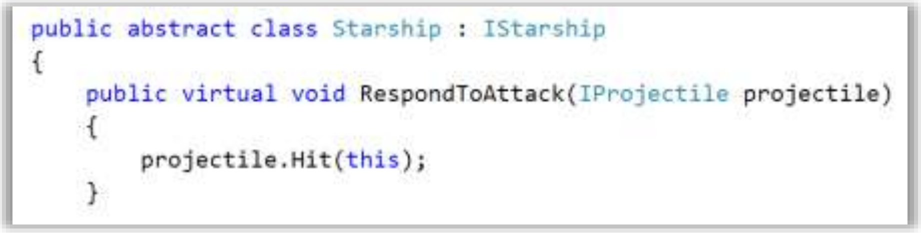
Hint: Increase the **`projectilesFired`** field of Frigates before producing a projectile.

Step 10 - Implement RespondToAttack()

Let's take a look at how different ships respond to attacks.

Ship	Response
Frigate	None (i.e. they just get hit)
Cruiser	None (i.e. they just get hit)
Dreadnought	**Raises** its **shields by 50** before getting hit (and removes them after that)

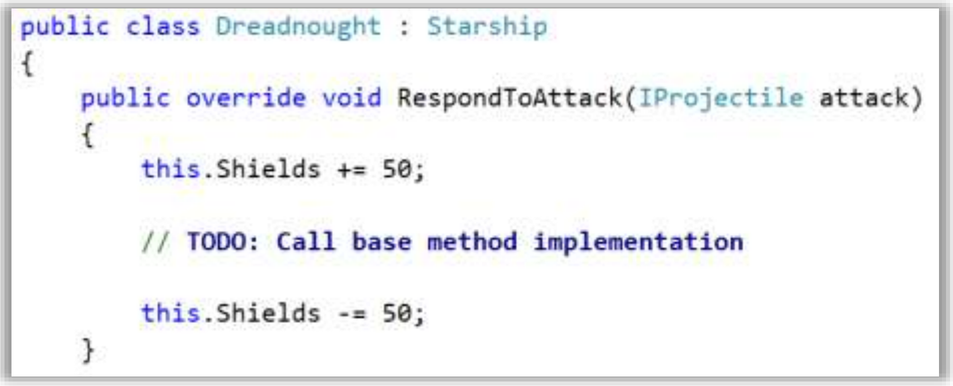
In other words, the **`RespondToAttack()`** method of **Frigates** and **Cruisers** should only call the **`Hit()`** method of the projectile.



```
public abstract class Starship : IStarship
{
    public virtual void RespondToAttack(IProjectile projectile)
    {
        projectile.Hit(this);
    }
```

Since this **behavior is common** for the majority of ships, it's safe to extract it to the base **Starship** class. We declare it virtual, so any descendants who wish to change the method should be free to do so.

The only descendant class which responds differently to attacks is the **Dreadnought**.

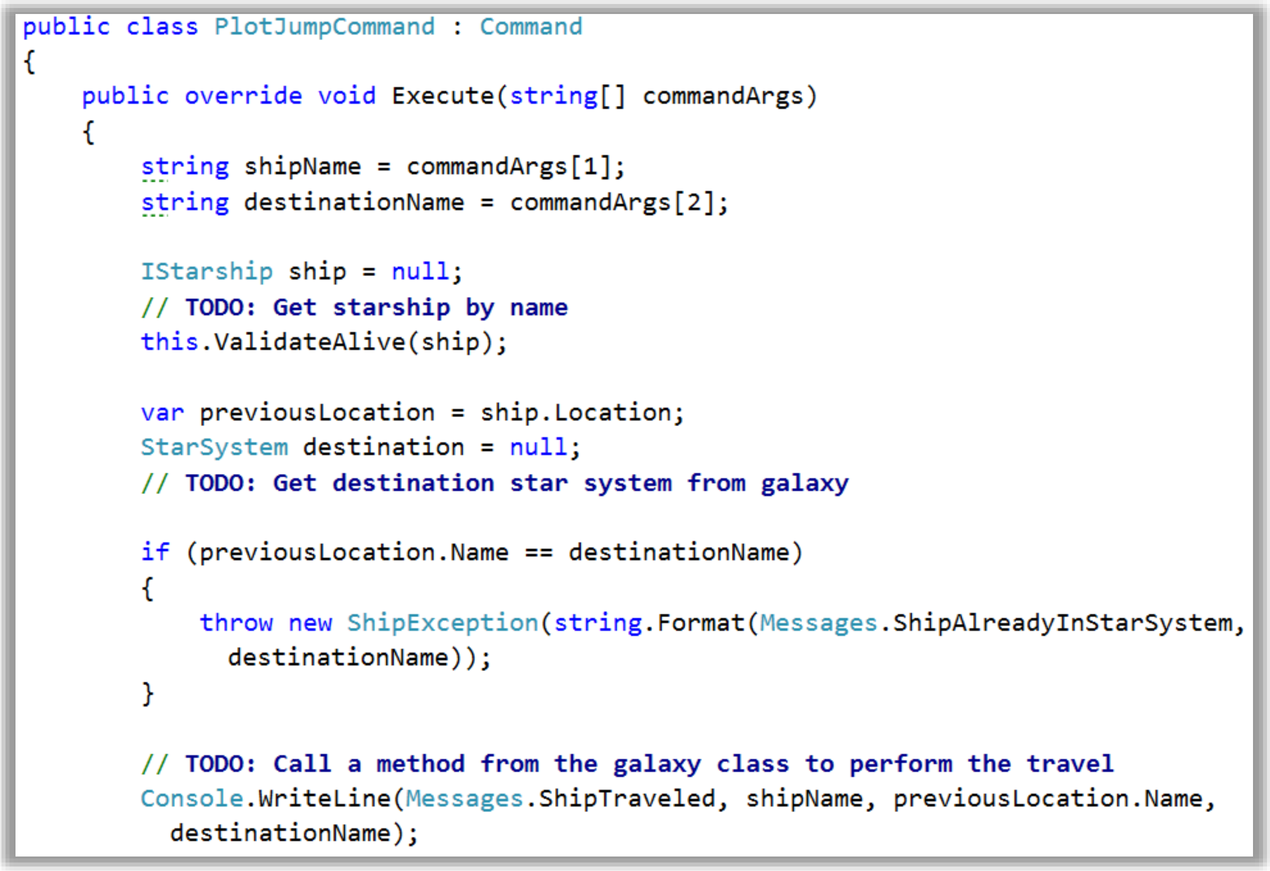


```
public class Dreadnought : Starship
{
    public override void RespondToAttack(IProjectile attack)
    {
        this.Shields += 50;

        // TODO: Call base method implementation

        this.Shields -= 50;
    }
```

Step 11 - Plot Jump Command

The command should change the location of the given starship to another star system. The following steps should be taken:

1. Get the **ship** from the engine by name
2. Validate it is not destroyed
3. Get the **destination star system** from the **Galaxy**
4. Validate that the ship is not already in the given destination
5. Call the appropriate method from the galaxy class to perform the travel for you
   Hint: Look through the **Galaxy** class



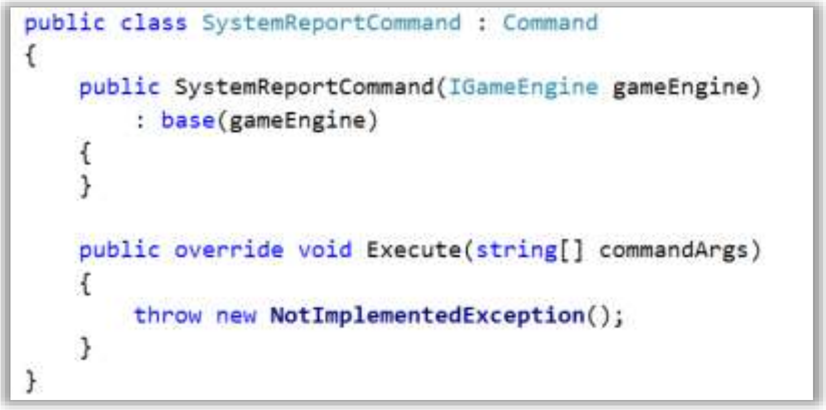
```
public class PlotJumpCommand : Command
{
    public override void Execute(string[] commandArgs)
    {
        string shipName = commandArgs[1];
        string destinationName = commandArgs[2];

        IStarship ship = null;
        // TODO: Get starship by name
        this.ValidateAlive(ship);

        var previousLocation = ship.Location;
        StarSystem destination = null;
        // TODO: Get destination star system from galaxy

        if (previousLocation.Name == destinationName)
        {
            throw new ShipException(string.Format(Messages.ShipAlreadyInStarSystem,
              destinationName));
        }

        // TODO: Call a method from the galaxy class to perform the travel
        Console.WriteLine(Messages.ShipTraveled, shipName, previousLocation.Name,
          destinationName);
```

Step 12 - Over Command

Implementing the **over** command is done like just any other command - we override the **Execute()** method in the **OverCommand** class. This one is up to you - look up the **GameEngine** class and see if there's any property you can change to stop the engine.

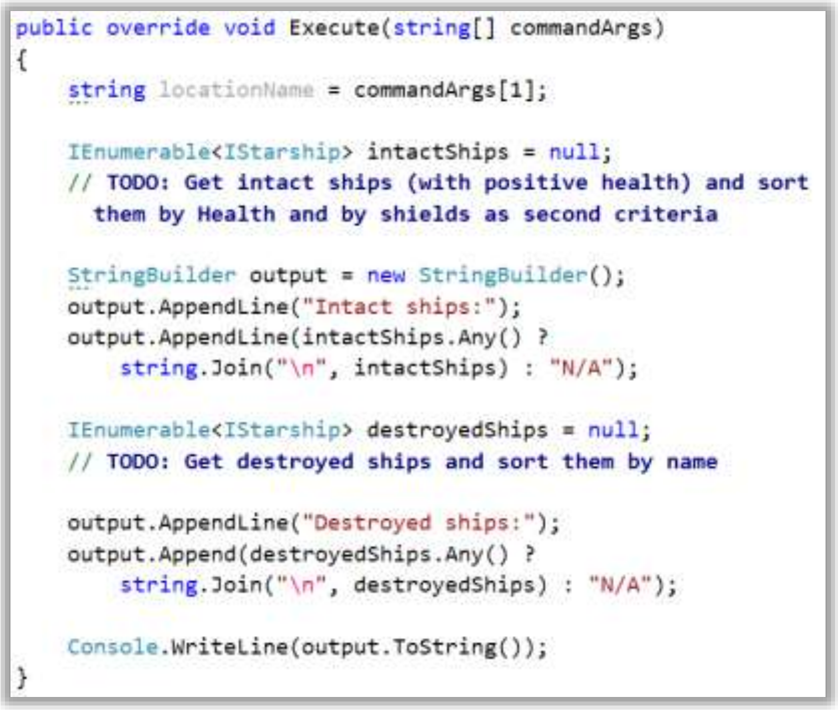
Step 13 - System Report Command

The **system-report** command should print all ships in the given star system. Let's create a new **SystemReportCommand** class (following the naming convention of the other command classes - "**CommandName** + **Command**"). It should inherit the base **Command** class and reuse its constructor.



```
public class SystemReportCommand : Command
{
    public SystemReportCommand(IGameEngine gameEngine)
        : base(gameEngine)
    {
    }

    public override void Execute(string[] commandArgs)
    {
        throw new NotImplementedException();
    }
}
```

Its **Execute()** method should print information about all ships in the given star system as defined in the description.



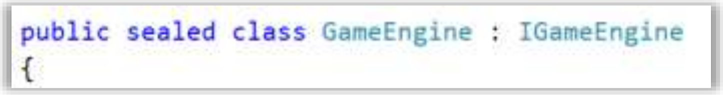
```
public override void Execute(string[] commandArgs)
{
    string locationName = commandArgs[1];

    IEnumerable<IStarship> intactShips = null;
    // TODO: Get intact ships (with positive health) and sort
      them by Health and by shields as second criteria

    StringBuilder output = new StringBuilder();
    output.AppendLine("Intact ships:");
    output.AppendLine(intactShips.Any() ?
        string.Join("\n", intactShips) : "N/A");

    IEnumerable<IStarship> destroyedShips = null;
    // TODO: Get destroyed ships and sort them by name

    output.AppendLine("Destroyed ships:");
    output.Append(destroyedShips.Any() ?
        string.Join("\n", destroyedShips) : "N/A");

    Console.WriteLine(output.ToString());
}
```

Step 14 - Extend the Engine

We have our new command class. However, the problem description explicitly tells us we have to extend the game engine **without editing its source code** (following the so-called **Open/Closed Principle** - open for extension, closed for modification).

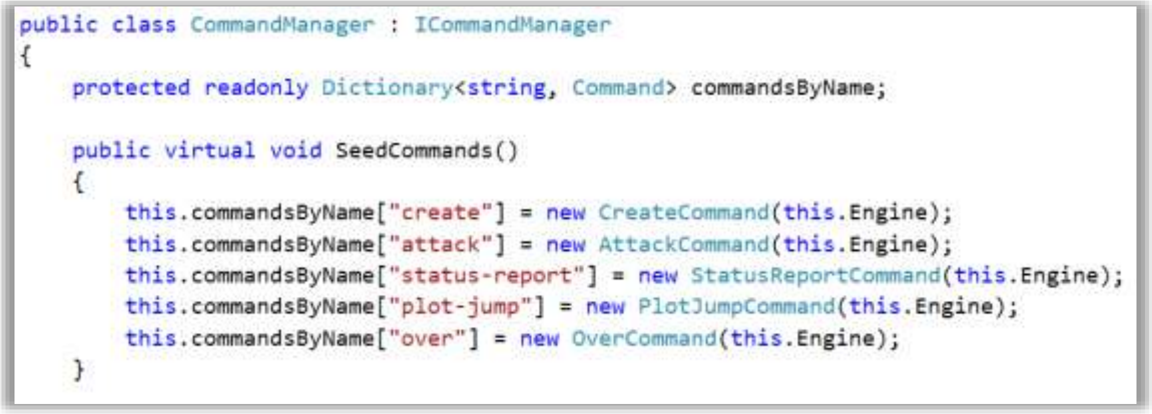
This is often the case with external libraries - we wish to extend a library's functionality by adding our own code, but the library is already compiled (thus we do not have access to the source code).

One possible way to **extend a class' functionality** is to **inherit the class** and **override the methods** we wish to change. Let's take a look at the **GameEngine** class.



```
public sealed class GameEngine : IGameEngine
{
```

It is declared sealed - therefore it cannot be inherited. But we need to add a new command to the engine - commands are stored in the **CommandManager** class.

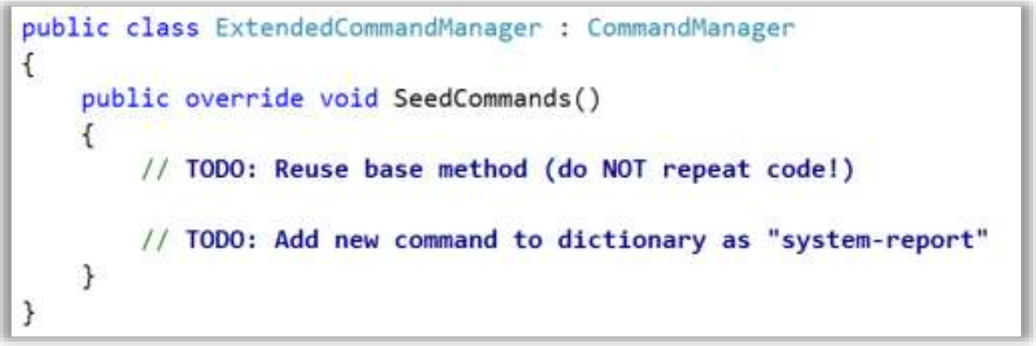


```
public class CommandManager : ICommandManager
{
    protected readonly Dictionary<string, Command> commandsByName;

    public virtual void SeedCommands()
    {
        this.commandsByName["create"] = new CreateCommand(this.Engine);
        this.commandsByName["attack"] = new AttackCommand(this.Engine);
        this.commandsByName["status-report"] = new StatusReportCommand(this.Engine);
        this.commandsByName["plot-jump"] = new PlotJumpCommand(this.Engine);
        this.commandsByName["over"] = new OverCommand(this.Engine);
    }
```

Again, we are not allowed to edit this class (if we were, we would simply add the **system-report** command and be done with it). But if we look closely:

- **`SeedCommands()`** is left **`virtual`** (i.e. can be overridden by descending classes)
- **`commandsByName`** has access modifier **`protected`** (i.e. can be accessed by descending classes)

Let's create a **ExtendedCommandManager** class that inherits the existing **CommandManager**. The new class should override the **`SeedCommands()`** method and add the newly created command to the dictionary.

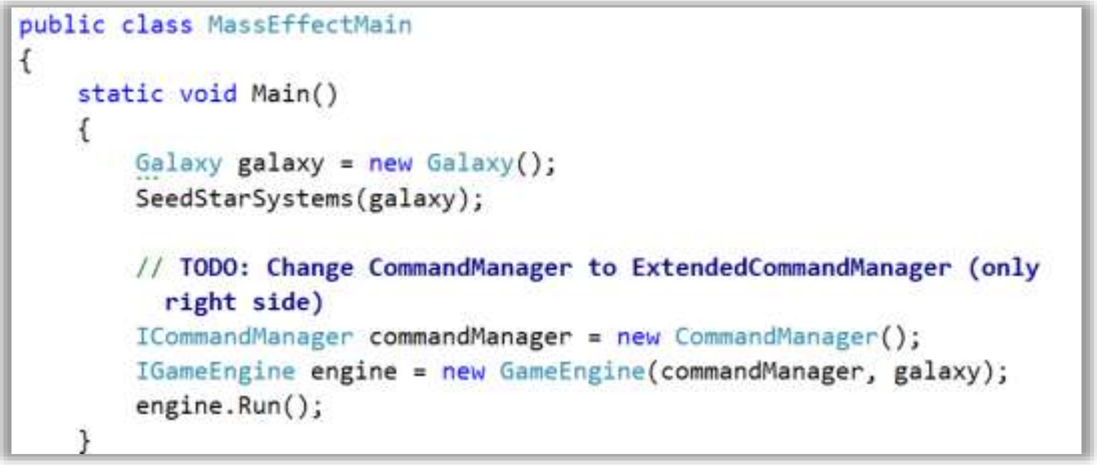


```
public class ExtendedCommandManager : CommandManager
{
    public override void SeedCommands()
    {
        // TODO: Reuse base method (do NOT repeat code!)

        // TODO: Add new command to dictionary as "system-report"
    }
}
```

Note: Reuse the base method implementation (just like you reuse a base constructor) - do NOT repeat code.

If all is well, the **ExtendedCommandManager** should support all old commands, as well as the newly created SystemReportCommand.

One last thing - we need to change the commandManager instance we pass to the GameEngine in the **`Main()`** method.



```
public class MassEffectMain
{
    static void Main()
    {
        Galaxy galaxy = new Galaxy();
        SeedStarSystems(galaxy);

        // TODO: Change CommandManager to ExtendedCommandManager (only
          right side)
        ICommandManager commandManager = new CommandManager();
        IGameEngine engine = new GameEngine(commandManager, galaxy);
        engine.Run();
    }
```