GDAPS1 – Practice Exercise

Abstract Classes

NOTE: Tutorial-like PE

This PE is tutorial-like. It contains extensive explanation and code snippets to help students understand what is happening, why the program behaves as such, and how to prevent errors from occurring.

Good luck! ☺

Overview

Practice with abstract classes, exception handling, returning object instances, and file IO.

Details

This practice exercise is broken into 3 sections: one for generating the base classes, another for file reading, and a third for exception handling.

Part 1:

- Create base class
- Create 2 derived classes
- Test them in Main

Part 2:

- Create a manager class to handle file reading and hold a List of dragon instances
- Create methods to:
  - read from a file and instantiate dragons
  - return an instance of a dragon
  - print all instances of a Dragon

Part 3:

- Exception handling with file reading

Part 1: Inheritance

Part 1: Base Class

Start with an **abstract** base class called `Dragon`.

> *Why an abstract class instead of a non-abstract ("regular") class?*
> *In determining if you need an abstract class or not, answer these questions: What does a "dragon" look like? How does a "dragon" act? Are there enough characteristics amongst dragons that one would ever need a generic "dragon"? Or will your program only ever instantiate specific instances of a "dragon"?*

Fields:

- **health**, which can go increase or decrease, but never below zero
- **name**, which will never change once set
- a **`Random`** object
- A randomly chosen attack message is printed when a `Dragon` attacks, livening up the experience for a player. Use a **string List** for these messages.

Create appropriate properties which account for the aforementioned behavior.

Constructor:

Use a parameterized constructor to initialize three of the fields: health, name, and `Random` should be passed into the constructor. Initialize the string list inside the constructor; the `Dragon` class does not need a passed-in list.

> *Passing a reference of an existing random number generator (created in Main) into this Dragon class constructor ensures the program will not generate the same numbers.*

Methods:

The `Dragon` class needs an **abstract** method called **`Attack`**, which receives no parameters and returns an integer representing attack damage.

> *Why an abstract method instead of a regular virtual method?*
> *We do not know how a generic "dragon" would attack, making it useless to write a method body that would never be utilized. All child classes attack in their own special way. Neither child would use the parent's Attack method. So we make it abstract!*

Override **`ToString`** to return the dragon's name and health, like so: **"**`Jimmy has 30 health`**".** `Dragon` subclasses will add on to this string in their own `ToString()` overrides.

> *Why is this method not abstract?*
> *It's already declared in the Object class. The Object class is not abstract. Therefore, it cannot contain an abstract method.*

Part 1: FireDragon

The `FireDragon` class should inherit from `Dragon`.

The `FireDragon` class requires no additional fields. It is a child class of `Dragon` not because it has additional attributes, but because it attacks differently than the other `Dragon` subclass.

Constructor:
Receive a health, name, and reference to a Random object as parameters. Pass all required data to the parent class's constructor.

> *Remember that this class's constructor does not need to handle value-setting of health, name and Random – their values are set in the parent Dragon class.*

The list of strings was initialized inside the parent class but it's empty. Add 3 to 5 different attack messages. For example, "Spark burns their opponent to a crisp!" or "Smoke rises as Spark unleashes flames." They don't all need to contain the dragon's name. Let loose your creativity!

Methods:
The `FireDragon` implementation of the `Attack` method must return a random integer between 10 and 21 (inclusive), representing how much damage this fire dragon can do. Before the return, randomly choose one of the attack messages and print it to the console window.

Override `ToString` by calling the parent's version and adding more specific information. For example, it might return: "*The Fire dragon* ***Spark has 30 health*****.**"
(In the above statement, blue text represents specific `FireDragon` info and orange text represents parent class's ToString().

Part 1: FrostDragon

The `FrostDragon` class is similar to `FireDragon`.

- inherits from `Dragon`
- Add 3 to 5 attack messages to the list
- implements `Attack`
- overrides `ToString`

Attacks:
One of the randomly-chosen attack messages is printed to the console when a Frost dragon attacks.

Damage: Frost dragons attack differently than fire dragons. Frost dragons have a base attack value between 10 and 18 (inclusive) but they have a 35% chance of 5 additional bonus attack damage. The bonus damage means that Frost dragons can potentially attack for 22. When a bonus attack occurs, print a message to the console like "Icicle hits for bonus ice damage!" (Feel free to be creative here.)

ToString:
`ToString` for `FrostDragon` should also call the parent version and add on extra information, like so: "*The Frost dragon* ***Spark has 30 health*****.**"

Part 1: Main Method

Let's test our classes!

Create a **`Dragon`** with an initial health of 200.
Since each dragon requires a reference to a `Random` object, create an instance of **`Random`** here in `Main` and pass it to the `Dragon`.

You can't actually make a `Dragon`? Excellent! Your parent class is working the way it should. ☺

Wait… it's all underlined in red. Why can't I make a Dragon?
Because it's an abstract class!

Create a **`FireDragon`** with an initial health of 100 and a **`FrostDragon`** with an initial health of 105. Before they battle, print their stats to the console window using ToString().

Loop while both dragons are still alive. Simulate each dragon attacking the other, using their methods and properties to generate attack values and adjust each dragon's health appropriately. Capture and print the results of each attack so you can "watch" the fight as it occurs. Your output should detail the attack damage dealt to each dragon, like so:

```
Spark received 10 points of damage and Icicle received 17 points
of damage.
```

After each "round" of battle, print the stats for each dragon using their `ToString` methods.

Once the fight is over, display the winner of the battle. Be sure to account for potential ties where both dragons deal final blows to each other.

Is the battle balanced? Not really. Fire dragons and frost dragons "should" win about equally, while a tie occurs more often than a single victor.

Part 1: Sample Run

```
The Fire dragon Inferno has 100 health.
The Frost dragon Wynter has 105 health.

Inferno burns their opponent to a crisp!
Wynter freezes their opponent!
Inferno received 15 points of damage and Wynter received 14 points of damage.
The Fire dragon Inferno has 85 health.
The Frost dragon Wynter has 91 health.

Smoke rises as Inferno unleashes flames!
An icy blast spirals through the air!
Wynter hits for bonus ice damage!
Inferno received 16 points of damage and Wynter received 16 points of damage.
The Fire dragon Inferno has 69 health.
The Frost dragon Wynter has 75 health.

Inferno burns their opponent to a crisp!
Wynter freezes their opponent!
Inferno received 16 points of damage and Wynter received 21 points of damage.
The Fire dragon Inferno has 53 health.
The Frost dragon Wynter has 54 health.

Smoke rises as Inferno unleashes flames!
An icy blast spirals through the air!
Wynter hits for bonus ice damage!
Inferno received 18 points of damage and Wynter received 17 points of damage.
The Fire dragon Inferno has 35 health.
The Frost dragon Wynter has 37 health.

A burst of fire explodes through the air!
Wynter freezes their opponent!
Inferno received 15 points of damage and Wynter received 10 points of damage.
The Fire dragon Inferno has 20 health.
The Frost dragon Wynter has 27 health.

A burst of fire explodes through the air!
An icy blast spirals through the air!
Inferno received 15 points of damage and Wynter received 10 points of damage.
The Fire dragon Inferno has 5 health.
The Frost dragon Wynter has 17 health.

Inferno burns their opponent to a crisp!
Wynter freezes their opponent!
Inferno received 11 points of damage and Wynter received 15 points of damage.
The Fire dragon Inferno has 0 health.
The Frost dragon Wynter has 2 health.

The air freezes as Wynter emerges victorious!
```

Final statements could also be:
  Both dragons fall to the ground, exhausted from their battle!
OR
  Flames rise from the ground as Inferno emerges victorious!
Depending on who won.

Part 2: File IO and a Manager class

Part 2: Manage those dragons!

At this point in the program, `Main` is not especially huge.

- ➪ Creating both dragon instances in `Main` uses only 2 lines of code.
- ➪ Printing their stats before battle is another 2 lines.
- ➪ The battle loop and winner declaration could be maybe 25 – 40 lines.

However...
We hard-coded stats for the `FireDragon` and `FrostDragon` instances. The data for each of them is coming from us, the programmer.

Programs that are data-driven are more robust.
We will use a text file that contains data for each dragon's name and health values. Then create `Dragon` instances based on the read-in information.

Which means...
That file reading must occur before we instantiate the dragons.
That's going to add more overhead to `Main`.
And it means that `Main` is handling everything:
File reading. Creation of dragons. The battle. Winner declaration.

Main is now cluttered with extra "stuff".

Let's not clutter Main.
A better approach would be to allow `Main` to just "do stuff" with the dragons after they are properly instantiated.

How do we get the dragons "out" of `Main` and "into" code elsewhere? A manager class!
A new class called **`DragonManager`** will do the following:

- "Manage" the dragons by:
  - Containing a field for each dragon
  - Instantiate those dragons via reading from a file
  - Returning a dragon reference, so `Main` can use that reference

Then `Main` could contain an instance of `DragonManager`, and the `DragonManager` class will handle making the dragons and "giving" them to `Main` when needed!

Get data.
Download the file dragonData.txt from myCourses. Place it in the project folder of this program. That's the directory where Program.cs and Dragon.cs and FireDragon.cs and FrostDragon.cs are.

Part 2: DragonManager class

Create a new class called **`DragonManager`**.

Fields:

- A **list of `Dragons`**
    - This list will contain both `FireDragons` and `FrostDragons`. Remember polymorphism?
- A **filename**
- **Random** object

Constructor:
Pass in the filename and reference to `Random` as parameters. Instantiate the list.

Methods:

1. `void InstantiateDragonsFromFile()`

Create a **`StreamReader`** and establish a connection to the **`filename`**.

Read the first line of the file outside of a loop. This line says: "Fire|4|Frost|3"

This means the file contains data for 4 FireDragons and 3 FrostDragons

Split the line using the appropriate delimiter.

We have used a while loop to read all data from a file before. However... that's when all data was of the same type and we could safely read all data and add it to a list of strings.

This file contains data for 2 different types of objects: FireDragons and FrostDragons. We need to add 7 overall Dragons to the list of dragons, but you need to know which child type to create. That first line of the file tells you how much FireDragon data there is, followed by a number of FrostDragon data.

Think of how you can use the data in `Split`'s returned string array to set up 1 or more loops so that your program properly reads the data for all dragons.

You could use a while loop to handle all 7 dragons, or 2 for loops – the first having 4 iterations, and the second having 3 iterations.

In this/these loop(s):

- Read a line of data from the file. Each line will look like this: `Inferno:100`
  The ***file format*** is that a dragon's name comes first and that dragon's health is second, separated by a colon.
- `Split` it using an appropriate delimiter.
- Use that split data to properly instantiate a `FireDragon` or a `FrostDragon` with the read-in name and health. Add the dragon to the `Dragon` list.

Close the reader.

DragonManager class, continued

Methods:

`2. void PrintDragons()`

Print every Dragon in the list to the console window.

`3. Dragon ChooseDefender()`

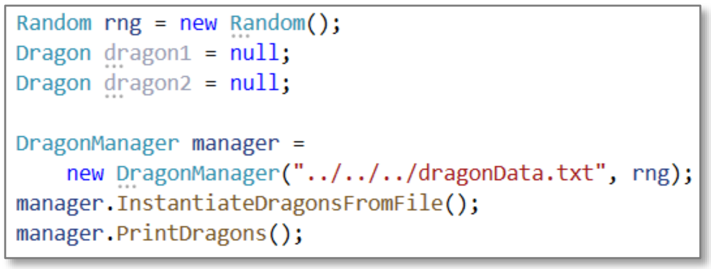
Return a randomly-chosen Dragon from the list.

Part 2: Using the DragonManager class in Main

Comment old code:

Comment out ALL of the code in Main, except the Random instantiation.

Create and call methods on DragonManager:



```
Random rng = new Random();
Dragon dragon1 = null;
Dragon dragon2 = null;

DragonManager manager =
    new DragonManager("../../../dragonData.txt", rng);
manager.InstantiateDragonsFromFile();
manager.PrintDragons();
```

- Create an instance of `DragonManager`. Pass in the dragon data file path and a reference to the instance of Random.
- Call the `InstantiateDragonsFromFile()` method.
- Call the `PrintDragons()` method.
- Create 2 variables for dragons and assign the value null.
  These variables ensure that the Dragons have the scope of ALL of Main, but we aren't sure which Dragon references they will hold yet. The DragonManager will give us a randomly chosen Dragon reference from its list.

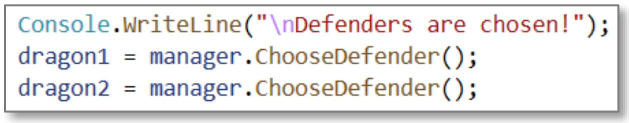
If properly done, your output should appear like so:

```
The Fire dragon Inferno has 100 health.
The Fire dragon Blaze has 100 health.
The Fire dragon Ember has 100 health.
The Fire dragon Scorch has 100 health.
The Frost dragon Wynter has 105 health.
The Frost dragon Glacier has 105 health.
The Frost dragon Snowy has 105 health.
```

Part 2: Using the DragonManager class, continued

Assign values to the Dragon variables

Now, let's assign those 2 `Dragon` variables a reference to one of the dragons from the list! Call `ChooseDefender()` to assign a value to the two `Dragon` variables.



```
Console.WriteLine("\nDefenders are chosen!");
dragon1 = manager.ChooseDefender();
dragon2 = manager.ChooseDefender();
```

Print each `Dragon's` stats before battle, just like we did when we tested the `Dragons` in part 1.

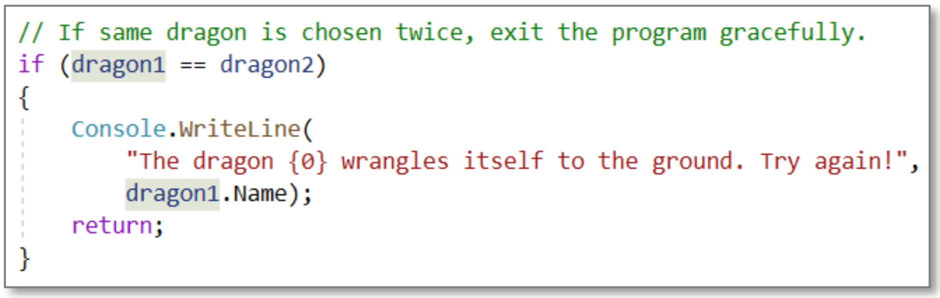
Run the program several times to ensure your methods are all working.

Wait a second...

You may notice that there's a chance that the 2 chosen `Dragons` are actually the same reference! In that case, we can "pretend" that the `Dragon` is a little crazy and fights itself, then stop the program.

That can be done with a simple comparison of the `Dragons` variables and a `return;` statement.

Remember that each Dragon variable is not a brand new Dragon... they are each holding a reference to a Dragon in the heap. The if statement checks if each variable is "pointing" to the same Dragon in memory.



```
// If same dragon is chosen twice, exit the program gracefully.
if (dragon1 == dragon2)
{
    Console.WriteLine(
        "The dragon {0} wrangles itself to the ground. Try again!",
        dragon1.Name);
    return;
}
```

The battle returns!

At this point, your program contains a reference to 2 randomly chosen dragons. You may have 2 `FireDragons` battling each other, or 2 `FrostDragons` duking it out, or you could have one of each dragon subtype. It's random!

Uncomment your battle code. Alter it to work with these 2 new `Dragon` variables instead of a hard-coded `FireDragon` and `FrostDragon`.

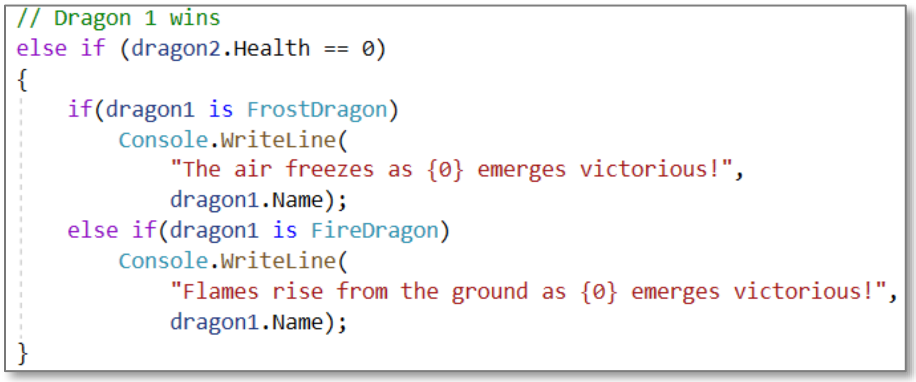
Part 2: Winners and Polymorphism

Remember how the sample run output was different depending on which type of `Dragon` won?
If the `FireDragon` won, the output would say "Flames rise from the ground as Inferno emerges victorious!" and "The air freezes as Wynter emerges victorious!" for the `FrostDragon`.

How do we do that now that the competing dragons are stored in variables of type `Dragon`?

The is keyword
One keyword is super-useful for polymorphic objects: **`is`**. Use the **`is`** keyword to determine which type of dragon the winner is.

There are multiple ways to handle printing the winner out. The code below shows just one way to do it. It's not the most efficient or clean way of doing it – but it's quite explicit.



```
// Dragon 1 wins
else if (dragon2.Health == 0)
{
    if(dragon1 is FrostDragon)
        Console.WriteLine(
            "The air freezes as {0} emerges victorious!",
            dragon1.Name);
    else if(dragon1 is FireDragon)
        Console.WriteLine(
            "Flames rise from the ground as {0} emerges victorious!",
            dragon1.Name);
}
```

Sample Runs

```
The Fire dragon Inferno has 100 health.
The Fire dragon Blaze has 100 health.
The Fire dragon Ember has 100 health.
The Fire dragon Scorch has 100 health.
The Frost dragon Wynter has 105 health.
The Frost dragon Glacier has 105 health.
The Frost dragon Snowy has 105 health.
```

And then either:

```
Defenders are chosen!
The dragon Glacier wrangles itself to the ground. Try again!
```

or:

```
Defenders are chosen!
The Fire dragon Inferno has 100 health.
The Fire dragon Scorch has 100 health.

A burst of fire explodes through the air!
Smoke rises as Scorch unleashes flames!
Inferno received 21 points of damage and Scorch received 11 points of damage.
The Fire dragon Inferno has 79 health.
The Fire dragon Scorch has 89 health.

A burst of fire explodes through the air!
Smoke rises as Scorch unleashes flames!
Inferno received 21 points of damage and Scorch received 20 points of damage.
The Fire dragon Inferno has 58 health.
The Fire dragon Scorch has 69 health.

Smoke rises as Inferno unleashes flames!
A burst of fire explodes through the air!
Inferno received 10 points of damage and Scorch received 21 points of damage.
The Fire dragon Inferno has 48 health.
The Fire dragon Scorch has 48 health.

A burst of fire explodes through the air!
Scorch burns their opponent to a crisp!
Inferno received 17 points of damage and Scorch received 12 points of damage.
The Fire dragon Inferno has 31 health.
The Fire dragon Scorch has 36 health.

Smoke rises as Inferno unleashes flames!
Smoke rises as Scorch unleashes flames!
Inferno received 10 points of damage and Scorch received 14 points of damage.
The Fire dragon Inferno has 21 health.
The Fire dragon Scorch has 22 health.

Smoke rises as Inferno unleashes flames!
Scorch burns their opponent to a crisp!
Inferno received 13 points of damage and Scorch received 20 points of damage.
The Fire dragon Inferno has 8 health.
The Fire dragon Scorch has 2 health.

Inferno burns their opponent to a crisp!
Scorch burns their opponent to a crisp!
Inferno received 15 points of damage and Scorch received 11 points of damage.
The Fire dragon Inferno has 0 health.
The Fire dragon Scorch has 0 health.

Both dragons fall to the ground, exhausted from their battle.
```

Part 3: Exception Handling

Part 3: Exception Handling

The last part of this exercise uses exception handling.
Exception handling allows a program to continue – gracefully – without crashing.

When reading a file, there is a possibility that the filename does not exist or is corrupted in some way. If that occurs, the program crashes.
To handle the exception, use a **`try/catch`** when reading from the dragonData.txt file.

Part 3: Change a DragonManager method

The `InstatiateDragonsFromFile()` method connects to the text file, reads data from it, and closes that file. If there is a problem finding the file, the program will crash.

> *It's never a good idea to just let a program crash! Imagine if your favorite games just... stopped and crashed... every single time something possibly went wrong.*

Use try & catch

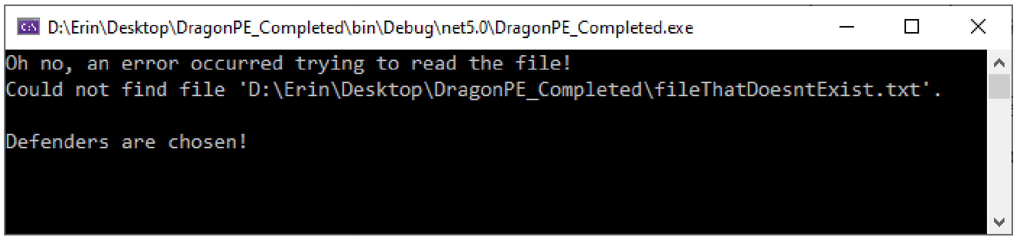
In order to keep this from occurring, place the file reading in a **`try/catch`** block.

- Declare the `StreamReader` object outside of the try block.
  Assign it a value of null.
- Inside the **`try`** block:
  Instantiate the `StreamReader` using the filename.
  Perform all file reading & closing here.

If there is a problem with the file and the `StreamReader` throws a `FileNotFoundException`, your program will immediately skip all remaining code in the try block and run the catch block.

A catch block offers the chance to run alternate code instead of crashing. We are going to catch the thrown `FileNotFoundException` and print a message to the Console window instead of crashing.

- Inside the **`catch`** block:
  Print the `Exception` object's **`Message`** property to the console window with some additional text.


D:\Erin\Desktop\DragonPE_Completed\bin\Debug\net5.0\DragonPE_Completed.exe

```
Oh no, an error occurred trying to read the file!
Could not find file 'D:\Erin\Desktop\DragonPE_Completed\fileThatDoesntExist.txt'.

Defenders are chosen!
```

BUT WAIT! The program still crashes, just in a different spot! Is that right?

Part 3: Yup. ;)

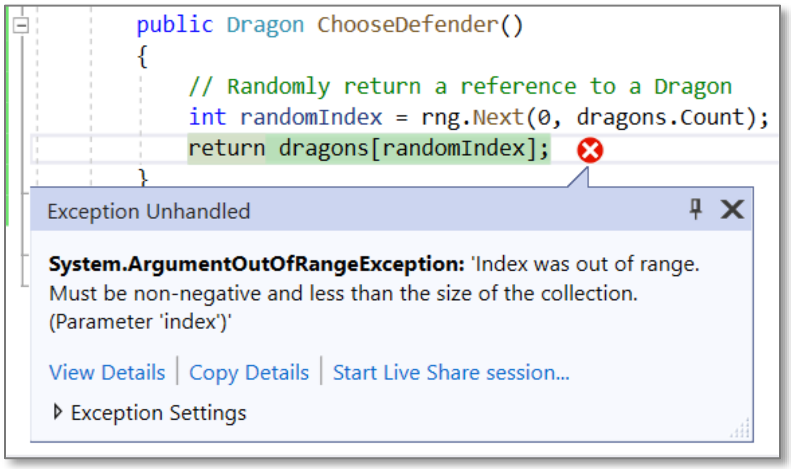
The program still crashes but in a different spot.
The program will crash in different places if the file reading is not completed.

Let's see what happens after the catch block executes!

1) The `DragonManager` class attempts to print all dragons in `Main`. This shouldn't crash as long as printing loop isn't accessing an invalid index. It shouldn't be since the `Dragon` list is empty and the loop should realistically not run at all.

2) A line is printed to the console. That happens just fine.

3) `ChooseDefender()` is called.
It attempts to return a randomly chosen `Dragon` from the list.
But it can't.
Because there are no `Dragons` in the list.
It's empty.
And if you coded something like this:



```
public Dragon ChooseDefender()
{
    // Randomly return a reference to a Dragon
    int randomIndex = rng.Next(0, dragons.Count);
    return dragons[randomIndex];
}
```

Exception Unhandled

System.ArgumentOutOfRangeException: 'Index was out of range. Must be non-negative and less than the size of the collection. (Parameter 'index')'

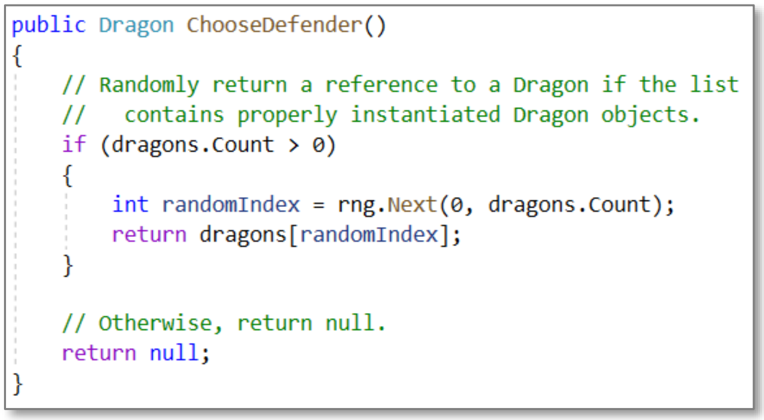
The `Next()` method is going to return 0.
And it attempts to return the first `Dragon` in the list.
Which there are none.
Get my point?

How do you handle that? Try/Catches all over the place?
Absolutely not. Try/Catch blocks should only be used when other error-checking methods cannot prevent an issue from occurring.

Error-Proofing

There are many different ways of error-proofing the program. You could...

Return a dragon instance from the list ONLY if the list is not empty, and return null otherwise.

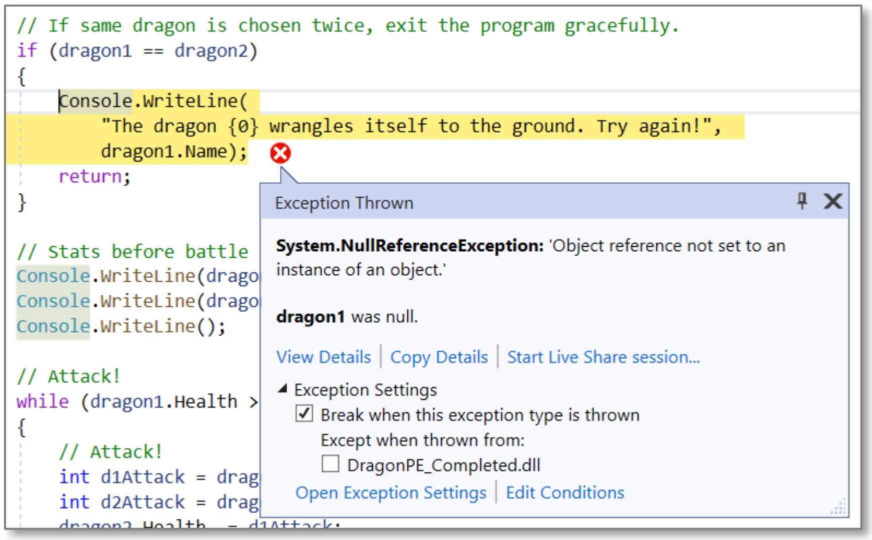


```
public Dragon ChooseDefender()
{
    // Randomly return a reference to a Dragon if the list
    //   contains properly instantiated Dragon objects.
    if (dragons.Count > 0)
    {
        int randomIndex = rng.Next(0, dragons.Count);
        return dragons[randomIndex];
    }

    // Otherwise, return null.
    return null;
}
```

That's a good idea, and I'm going to keep it in.

However, it leads to a crash here:



```
// If same dragon is chosen twice, exit the program gracefully.
if (dragon1 == dragon2)
{
    Console.WriteLine(
        "The dragon {0} wrangles itself to the ground. Try again!",
        dragon1.Name);
    return;
}

// Stats before battle
Console.WriteLine(drago
Console.WriteLine(drago
Console.WriteLine();

// Attack!
while (dragon1.Health >
{
    // Attack!
    int d1Attack = drag
    int d2Attack = drag
```

Exception Thrown

System.NullReferenceException: 'Object reference not set to an instance of an object.'

dragon1 was null.

View Details | Copy Details | Start Live Share session...

◢ Exception Settings
- ☑ Break when this exception type is thrown
- Except when thrown from:
- ☐ DragonPE_Completed.dll

Open Exception Settings | Edit Conditions

Because null == null, but you cannot retrieve a `Name` from null...
Or access the `Health` of null...
Or call `Attack()` on null...

What now? Try Catches?
Still no! These are all issues that can be handled with conditionals. A try/catch is NOT needed for any of these potential errors.

You could:

- Change `InstantiateDragonsFromFile()` to return a boolean value when file reading is successful, and only continue with the rest of code inside `Main` if it was, or
- Place all of the battle code in `Main` inside a conditional checking whether `dragon1` or `dragon2` are null, or
- Something else you want the program to do, or
- Close the application in the catch block.
  (If there really is nothing else to do, or closing is the best choice for user experience)
  See https://docs.microsoft.com/en-us/dotnet/api/system.environment.exit?view=net-6.0#remarks for information on Environment.Exit()

But why did we need a try/catch with file reading?
Couldn't we prevent that crash with a conditional?

Nope.

You, the programmer, won't always know if there are errors with files. And since you don't know if there are errors, you don't know if an exception will occur. There is no way to check if the file is available without using a Stream, and the Stream will crash if that file is not accessible.

<u>This is NOT a flaw. It's by design.</u>

Exceptions are ways for a program to communicate with you, the programmer.

An exception communicates from one class to another that something couldn't (or shouldn't) happen.

An exception represents something going wrong in a program.

The exception being thrown is a red flag that communicates "Hey programmer! The file wasn't there!"

Part 3: "Fix" the program

Choose your favorite solution from the list above, or create your own solution.
When implemented, your program should not crash when an invalid filename is given.

Part 4: Submission

You know what to do! ☺