INHERITANCE – Exercises

Read this first I've designed these exercise to help you apply what you have learned in this section. Make sure to do these exercises before moving on to the next chapter if you want to get the most out of this course. Where are the answers? I have deliberately decided not to provide the answers. Why? Because I want to make a great programmer out of you, not a lazy programmer who is used to copying/pasting code from various sources, without knowing what is happening! In the real world, when you are at work, your job is to solve a problem. No one (including myself) knows the answers to real world problems initially. So we need to research, think, try different ideas, and see what works best. And that's exactly the kind of attitude I would like to see in you. If I give you the answers, you're going to get lazy and quickly look at the solution. This way, your programming brain will not be trained and you will always be dependent on other people's answers in the real-world. So, do your best to solve these problems. They are not excessively hard, and any student who has taken all the lectures in this section should be able to solve these problems with a little effort. If you get stuck along the way, post your question in the discussion area. I'm happy to help you out. But please check the discussion area first to make sure no one has asked the same question before. This way, you'll save both your and my time. So, let's get started

Exercise: Design a Stack

A Stack is a data structure for storing a list of elements in a LIFO (last in, first out) fashion.

Design a class called Stack with three methods.

void Push(object obj)

object Pop()

void Clear()

The Push() method stores the given object on top of the stack. We use the "object" type here so we can store any objects inside the stack. Remember the "object" class is the base of all classes in the .NET Framework. So any types can be automatically upcast to the object. Make sure to take into account the scenario that null is passed to this object. We should not store null references in the stack. So if null is passed to this method, you should throw an InvalidOperationException. Remember, when coding every method, you should think of all possibilities and make sure the method behaves properly in all these edge cases. That's what distinguishes you from an "average" programmer.

The Pop() method removes the object on top of the stack and returns it. Make sure to take into account the scenario that we call the Pop() method on an empty stack. In this case, this method

should throw an InvalidOperationException. Remember, your classes should always be in a valid state and used properly. When they are misused, they should throw exceptions. Again, thinking of all these edge cases, separates you from an average programmer. The code written this way will be more robust and with less bugs.

The Clear() method removes all objects from the stack.

We should be able to use this stack class as follows:

```
var stack = new Stack();
stack.Push(1);
stack.Push(2);
stack.Push(3);
Console.WriteLine(stack.Pop());
Console.WriteLine(stack.Pop());
Console.WriteLine(stack.Pop());
```

The output of this program will be

```
3
2
1
```

Note: The downside of using the object class here is that if we store value types (eg int, char, bool, DateTime) in our Stack, boxing and unboxing occurs, which comes with a small performance penalty. In my C# Advanced course, I'll teach you how to resolve this by using generics, but for now don't worry about it.

2

Real-world use case: Stacks are very popular in real-world applications. Think of your browser. As you navigate the web, the address of each page you visit is stored in a stack. As you click the Back button, the most recent address is popped. This is because of the LIFO behaviour of stacks.

POLYMORPHISM – Exercises

Read this first I've designed these exercise to help you apply what you have learned in this section. Make sure to do these exercises before moving on to the next chapter if you want to get the most out of this course. Where are the answers? I have deliberately decided not to provide the answers. Why? Because I want to make a great programmer out of you, not a lazy programmer who is used to copying/pasting code from various sources, without knowing what is happening! In the real world, when you are at work, your job is to solve a problem. No one (including myself) knows the answers to real world problems initially. So we need to research, think, try different ideas, and see what works best. And that's exactly the kind of attitude I would like to see in you. If I give you the answers, you're going to get lazy and quickly look at the solution. This way, your programming brain will not be trained and you will always be dependent on other people's answers in the real-world. So, do your best to solve these problems. They are not excessively hard, and any student who has taken all the lectures in this section should be able to solve these problems with a little effort. If you get stuck along the way, post your question in the discussion area. I'm happy to help you out. But please check the discussion area first to make sure no one has asked the same question before. This way, you'll save both your and my time. So, let's get started

Exercise 1: Design a database connection

To access a database, we need to open a connection to it first and close it once our job is done. Connecting to a database depends on the type of the target database and the database management system (DBMS). For example, connecting to a SQL Server database is different from connecting to an Oracle database. But both these connections have a few things in common: • They have a connection string • They can be opened • They can be closed • They may have a timeout attribute (so if the connection could not be opened within the timeout, an exception will be thrown). Your job is to represent these commonalities in a base class called DbConnection. This class should have two properties: ConnectionString : string Timeout : TimeSpan A DbConnection will not be in a valid state if it doesn't have a connection string. So you need to pass a connection string in the constructor of this class. Also, take into account the scenarios where null or an empty string is sent as the connection string. Make sure to throw an exception to guarantee that your class will always be in a valid state. Our DbConnection should also have two methods for opening and closing a connection. We don't know how to open or close a connection in a DbConnection and this should be left to the classes that derive from DbConnection. These

classes (eg SqlConnection or OracleConnection) will provide the actual implementation. So you need to declare these methods as abstract. Derive two classes SqlConnection and OracleConnection from DbConnection and provide a simple implementation of opening and closing connections using Console.WriteLine(). In the real-world, SQL Server provides an API for opening or closing a connection to a database. But for this exercise, we don't need to worry about it.

Exercise 2: Design a database command

Now that we have the concept of a DbConnection, let's work out how to represent a DbCommand. Design a class called DbCommand for executing an instruction against the database. A DbCommand cannot be in a valid state without having a connection. So in the constructor of this class, pass a DbConnection. Don't forget to cater for the null. Each DbCommand should also have the instruction to be sent to the database. In case of SQL Server, this instruction is expressed in T-SQL language. Use a string to represent this instruction. Again, a command cannot be in a valid state without this instruction. So make sure to receive it in the constructor and cater for the null reference or an empty string. Each command should be executable. So we need to create a method called Execute(). In this method, we need a simple implementation as follows: Open the connection Run the instruction Close the connection Note that here, inside the DbCommand, we have a reference to DbConnection. Depending on the type of DbConnection sent at runtime, opening and closing a connection will be different. For example, if we initialize this DbCommand with a SqlConnection, we will open and close a connection to a Sql Server database. This is polymorphism. Interestingly, DbCommand doesn't care about how a connection is opened or closed. It's not the responsibility of the DbCommand. All it cares about is to send an instruction to a database. For running the instruction, simply output it to the Console. In the real-world, SQL Server (or any other DBMS) provides an API for running an instruction against the database. We don't need to worry about it for this exercise. In the main method, initialize a DbCommand with some string as the instruction and a SqlConnection. Execute the command and see the result on the console. Then, swap the SqlConnection with an OracleConnection and see polymorphism in action