**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**

Sol:

using System.Collections;

using System.Runtime.CompilerServices;

class StackByChida

{

private Object[] st = new Object[1];

private int top = 0;

public void push(Object element)

{

if (element == null)

{

throw new InvalidOperationException("Null cant be passed here.");

}

if (top == this.st.Length)

{

Object[] st2 = new Object[this.st.Length \* 2];

for (int i = 0; i < this.st.Length; i++)

{

st2[i] = this.st[i];

}

this.st = st2;

}

this.st[top] = element;

this.top++;

}

public Object pop()

{

if (top == 0)

{

throw new InvalidOperationException("Stack is empty!!!");

}

Object element = st[this.top - 1];

this.st = this.RemoveElement(this.st, this.top - 1);

this.top--;

return element;

}

private Object[] RemoveElement(Object[] arr, int RemoveAt)

{

Object[] arr2 = new Object[arr.Length - 1];

int i = 0;

int j = 0;

while (i < arr.Length)

{

if (i != RemoveAt)

{

arr2[j] = arr[i];

j++;

}

i++;

}

return arr2;

}

public void clear()

{

this.st = new Object[1];

this.top = 0;

}

public void display()

{

if (this.top == 0)

{

Console.WriteLine("Stack is Empty!!!");

}

else

{

for (int i = 0; i < this.top; i++)

{

Console.WriteLine(st[i]);

}

}

}

}

class Program

{

public static void Main(String[] args)

{

var s2 = new StackByChida();

StackByChida s1 = new StackByChida();

s1.push("Chida");

s1.push(true);

s1.push(2);

s1.push(4.5);

s1.push('c');

//s1.push(null);

s1.display();

Console.WriteLine(s1.pop());

Console.WriteLine(s1.pop());

Console.WriteLine(s1.pop());

Console.WriteLine(s1.pop());

Console.WriteLine(s1.pop());

//Console.WriteLine(s1.pop());

s1.display();

s1.push("vichu");

s1.push(true);

s1.push(2);

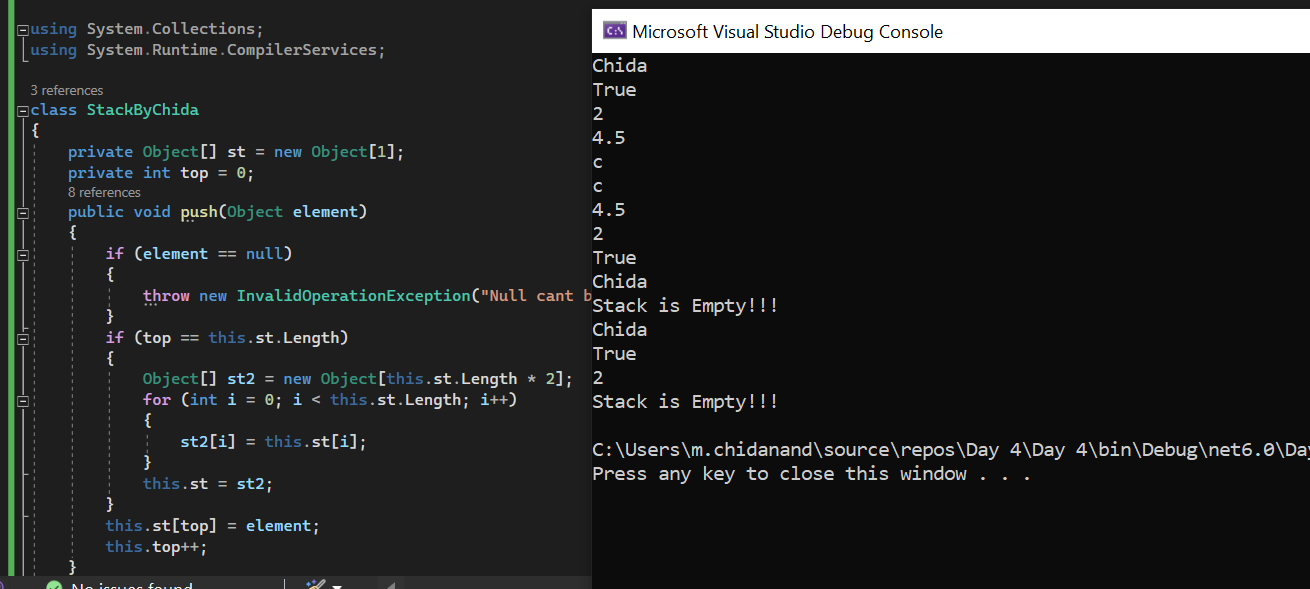
s1.display();

s1.clear();

s1.display();

}

}



**Exercise 2: 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.**

sol:

using System;

namespace program

{

public abstract class DBConnection

{

public abstract void open();

public abstract void close();

}

class SQLConnection : DBConnection

{

public string ConnectionString { get; set; }

public SQLConnection(string Connectionstring)

{

try

{

if (Connectionstring == null)

{

throw new ArgumentNullException("--> Null Instruction");

}

else

{

ConnectionString = Connectionstring;

Console.WriteLine("Connection string: " + ConnectionString);

}

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

public override void open()

{

Console.WriteLine("SQL connection opened");

}

public override void close()

{

Console.WriteLine("SQL connection closed");

}

}

class OracleConnection : DBConnection

{

public string ConnectionString { get; set; }

public OracleConnection(string Connectionstring)

{

try

{

if (Connectionstring == null)

{

throw new ArgumentNullException("--> Null Instruction");

}

else

{

ConnectionString = Connectionstring;

Console.WriteLine("Connection string: " + ConnectionString);

}

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

public override void open()

{

Console.WriteLine("Oracle connection opened");

}

public override void close()

{

Console.WriteLine("Oracle connection closed");

}

}

class MainClass

{

public static void Main(String[] args)

{

//BaseClassname objectName = new DerivedClassName();

Console.WriteLine("Enter the Connection string:");

string connString = Console.ReadLine();

Console.WriteLine("Enter 1 to connect to SQL database\nEnter 2 to connect to Oracle database\n");

int choice = Convert.ToInt32(Console.ReadLine());

if (choice == 1)

{

DBConnection connection1 = new SQLConnection(connString);

connection1.open();

connection1.close();

}

else if (choice == 2)

{

DBConnection connection2 = new OracleConnection(connString);

connection2.open();

connection2.close();

}

else

{

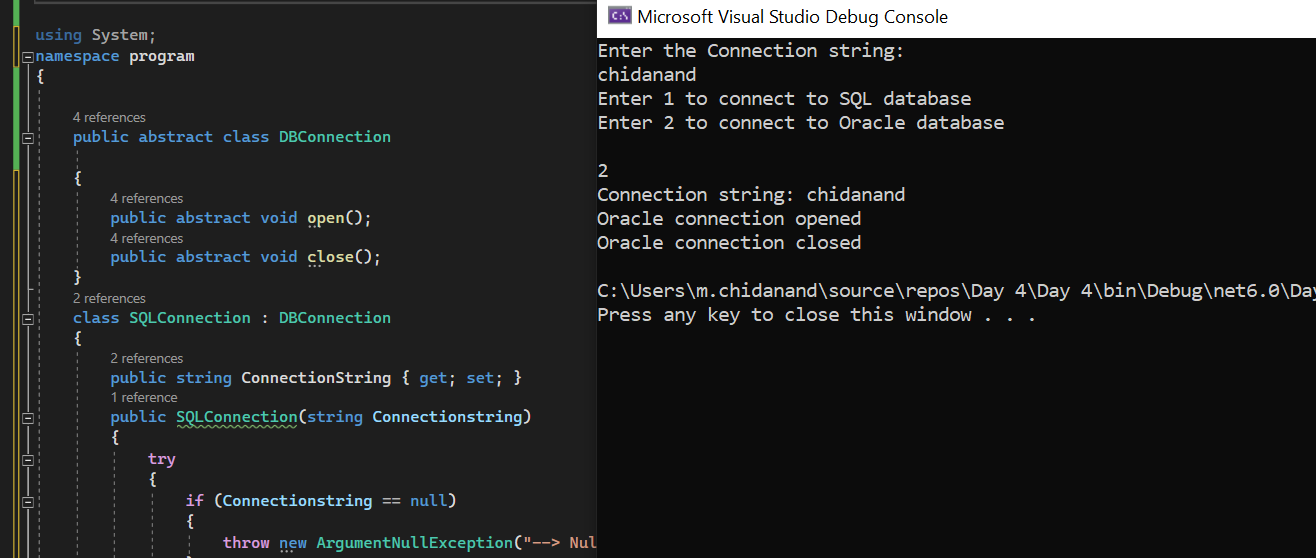
Console.WriteLine("-Exits-");

}

}

}

}



**Exercise 3: 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.**

Sol

using System;

namespace DbCommandPolymorphism

{

    public abstract class DbConnection

    {

        public string ConnectionString { get; set; }

        public TimeSpan Timeout { get; set; }

        public abstract void Open();

        public abstract void Close();

    }

    public class SqlConnection : DbConnection

    {

        public override void Open()

        {

            Console.WriteLine("Opening SQL connection...");

        }

        public override void Close()

        {

            Console.WriteLine("Closing SQL connection...");

        }

    }

    public class OracleConnection : DbConnection

    {

        public override void Open()

        {

            Console.WriteLine("Opening Oracle connection...");

        }

        public override void Close()

        {

            Console.WriteLine("Closing Oracle connection...");

        }

    }

    public class DbCommand

    {

        public string Instruction { get; set; }

        public DbConnection Connection { get; set; }

        public DbCommand(string instruction, DbConnection connection)

        {

            if (String.IsNullOrWhiteSpace(instruction))

                throw new ArgumentException("Instruction cannot be null or empty.");

            if (connection == null)

                throw new ArgumentNullException("connection", "Connection cannot be null.");

            Instruction = instruction;

            Connection = connection;

        }

        public void Execute()

        {

            Connection.Open();

            Console.WriteLine("Executing instruction: " + Instruction);

            Connection.Close();

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            DbConnection sqlConnection = new SqlConnection();

            DbCommand sqlCommand = new DbCommand("SELECT \* FROM Customers", sqlConnection);

            sqlCommand.Execute();

            Console.WriteLine();

            DbConnection oracleConnection = new OracleConnection();

            DbCommand oracleCommand = new DbCommand("SELECT \* FROM Customers", oracleConnection);

            oracleCommand.Execute();

            Console.ReadLine();

        }

    }

}