DIGISTRAT
CONSULTING

C# - SQL training

Oct 2021

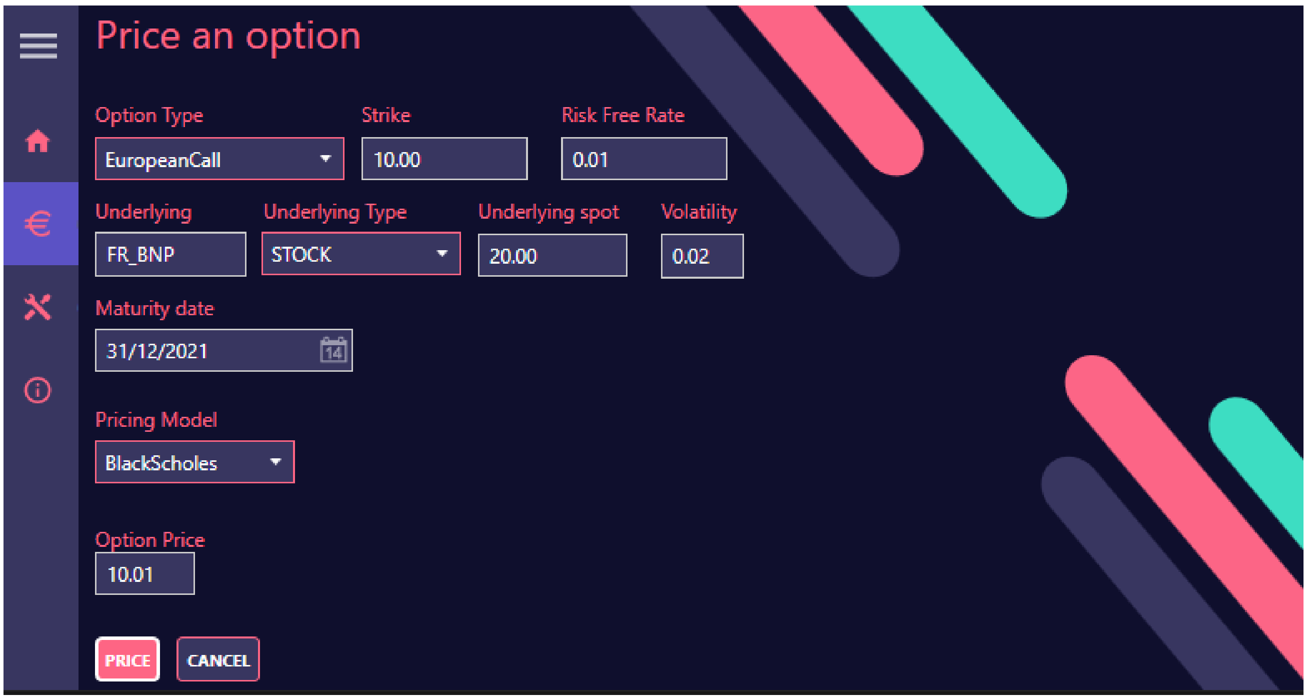
Summary

sami.benabidallah@digistratconsulting.com

DIGISTRAT
CONSULTING

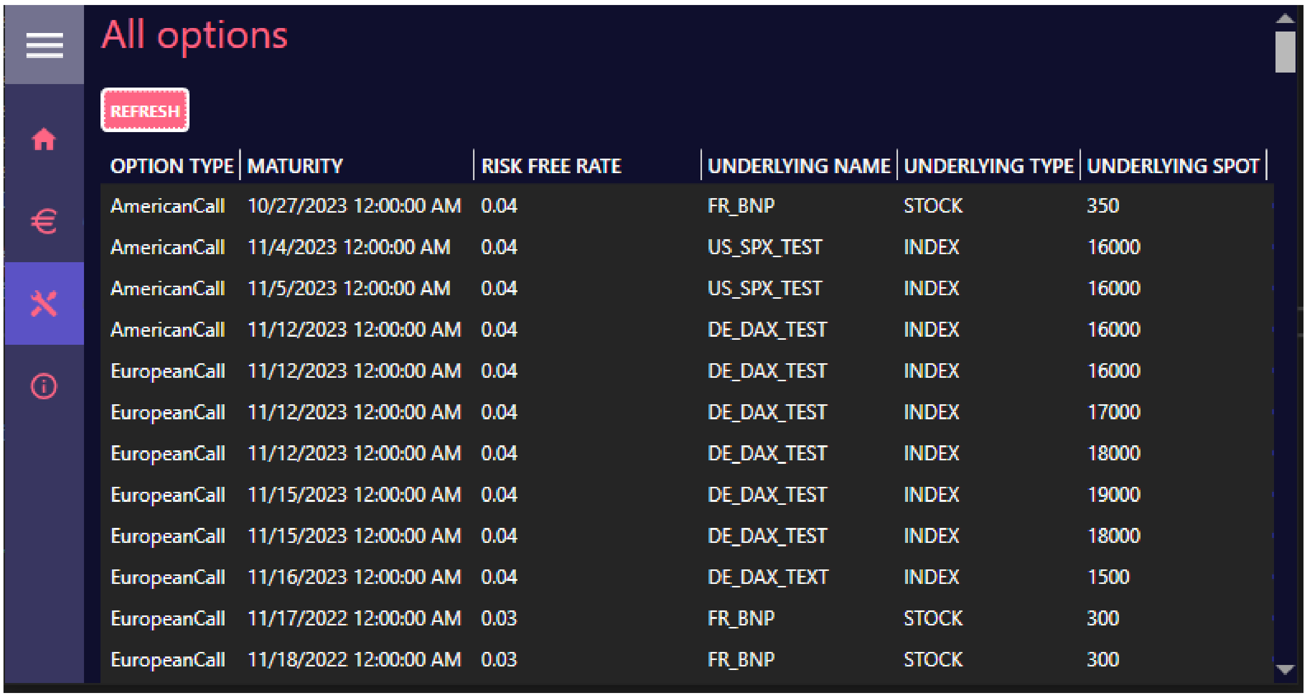
1- Introduction

During this training, we will be covering several topics. First, by going through .NET memory management, C# language, transport layer (NetMQ) then SQL and finally graphic interface (WPF).
We will illustrate these topics by fully implementing an option pricing application (from backend to front end, going through database management)



Price an option

Option Type	EuropeanCall
Strike	10.00
Risk Free Rate	0.01
Underlying	FR_BNP
Underlying Type	STOCK
Underlying spot	20.00
Volatility	0.02
Maturity date	31/12/2021
Pricing Model	BlackScholes
Option Price	10.01



All options

OPTION TYPE	MATURITY	RISK FREE RATE	UNDERLYING NAME	UNDERLYING TYPE	UNDERLYING SPOT
AmericanCall	10/27/2023 12:00:00 AM	0.04	FR_BNP	STOCK	350
AmericanCall	11/4/2023 12:00:00 AM	0.04	US_SPX_TEST	INDEX	16000
AmericanCall	11/5/2023 12:00:00 AM	0.04	US_SPX_TEST	INDEX	16000
AmericanCall	11/12/2023 12:00:00 AM	0.04	DE_DAX_TEST	INDEX	16000
EuropeanCall	11/12/2023 12:00:00 AM	0.04	DE_DAX_TEST	INDEX	16000
EuropeanCall	11/12/2023 12:00:00 AM	0.04	DE_DAX_TEST	INDEX	17000
EuropeanCall	11/12/2023 12:00:00 AM	0.04	DE_DAX_TEST	INDEX	18000
EuropeanCall	11/15/2023 12:00:00 AM	0.04	DE_DAX_TEST	INDEX	19000
EuropeanCall	11/15/2023 12:00:00 AM	0.04	DE_DAX_TEST	INDEX	18000
EuropeanCall	11/16/2023 12:00:00 AM	0.04	DE_DAX_TEXT	INDEX	1500
EuropeanCall	11/17/2022 12:00:00 AM	0.03	FR_BNP	STOCK	300
EuropeanCall	11/18/2022 12:00:00 AM	0.03	FR_BNP	STOCK	300

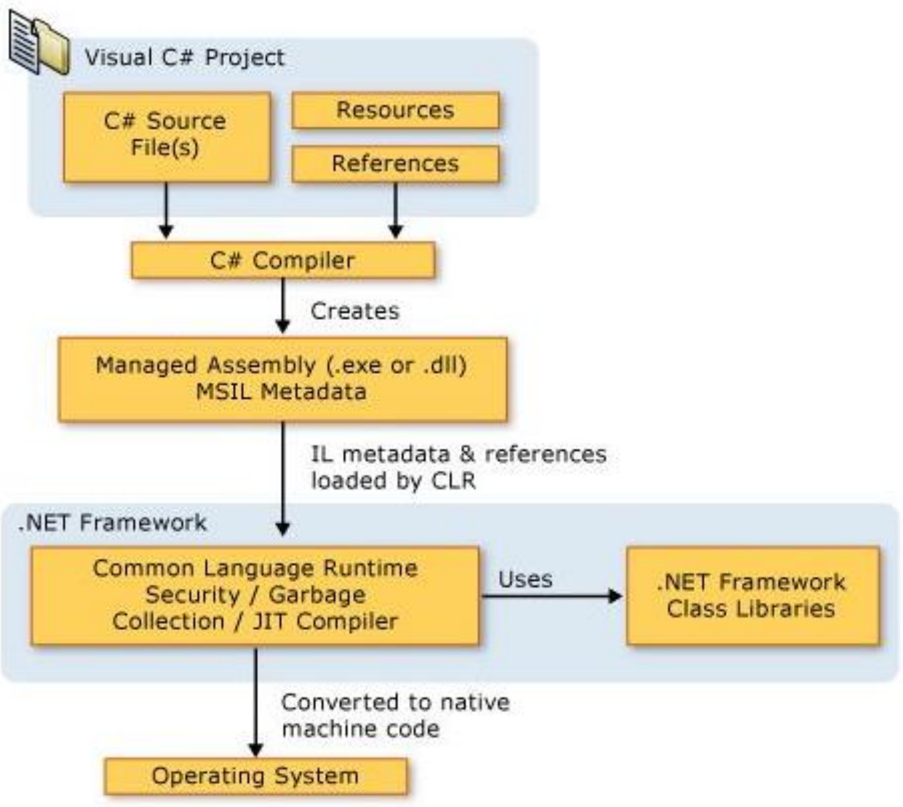
2- First interview – Skills assessment

The first session goal is to understand the skills of individuals as well as identify areas where training is needed.
The topics covered are : CLR, memory management, managed/unmanaged code, object oriented programming concepts, multithreading, WPF, SQL

I- CLR:

https://www.c-sharpcorner.com/UploadFile/9582c9/what-is-common-language-runtime-in-C-Sharp/
The heart of .NET Framework is a run-time environment called the common language runtime (CLR). The CLR manages the life cycle and executes .NET applications (code). It also provides services that make the development process easier.

- As part of the Microsoft .NET Framework, the Common Language Runtime (CLR) is the programming (Virtual Machine component) that manages the execution of programs written in any language that uses the .NET Framework, for example C#, VB.Net, F# and so on.
- Those programs are compiled into an intermediate form of code called CLI in a portable execution file (PE) that can be managed and used by the CLR and then the CLR converts it into machine code to be will executed by the processor.
- The information about the environment, programming language, its version and what class libraries will be used for this code are stored in the form of metadata with the compiler that tells the CLR how to handle this code.


Visual C# Project
C# Source File(s)
Resources
References
C# Compiler
Creates
Managed Assembly (.exe or .dll) MSIL Metadata
IL metadata & references loaded by CLR
.NET Framework
Common Language Runtime Security / Garbage Collection / JIT Compiler
Uses
.NET Framework Class Libraries
Converted to native machine code
Operating System

The CLR provides several functions :

- It converts code into CLI
- Type safety
- Exception handling

- Platform independency
- Memory management (using the Garbage Collector)

CLR components :

- Exception handler – handles exceptions at run time
- Thread support – supports multithreading
- Garbage Collector – manages the memory. Collect all unused objects and deallocate them to reduce memory
- Code Manager – manages the code at run time
- JIT – Just In Time compiler converts MSIL code to native code
- Class Loader – Used to load all classes at run-time

II- Managed Unmanaged code

.NET supports two kind of coding

1. Managed Code
2. Unmanaged Code

Managed code :

The code, which is developed in .NET framework, is known as managed code. This code is directly executed by CLR with help of managed code execution.

Unmanaged code :

Pointer, data base connections, socket, files …

The code, which is developed outside .NET framework is known as unmanaged code.

Applications that do not run under the control of the CLR are said to be unmanaged

Unmanaged code is executed with help of wrapper classes.

Wrapper classes are of two types : **CCW (COM Callable Wrapper)** and **RCW (Runtime Callable Wrapper).**

III- Memory leak

In general, a memory leak is a process in which a program or application persistently retains a computer’s memory. It occurs when the program doesn’t release allocated memory space, even after exection.

In unmanaged code, a memory leak is a failutre to release unreachable memory, which can no longer be allocated again by any process. It can be addressed by using Garbage Collector techniques.

In managed code, a memory leak is a failure to release reachable memory which is no longer needed for the program to run correctly.

Memory leak causes :

- Unmanaged leaks (code that allocates unmanaged code)
- Resource leaks (code that allocates and uses unmanaged resources, like files, sockets …)
- Extended lifetime of objects
- Bugs in the .NET runtime
- Incorrect understanding of how GC and .NET memory management works

The first two are usually handled by two different pieces of code:

- Implementing IDisposable on the object and disposing of the unmanaged memory/resource in the Dispose method
- Implementing a finalizer, to make sure unmanaged resources are deallocated when GC has found the object to be eligible for collection

The Dispose/Finalize Pattern

It is recommended to implement both Dispose and Finalize when working with unmanaged resources.
The correct sequence is for a developper to call Dispose.
The Finalize implementation would run and the resources would still be released when the object is garbage collected even if the developper neglected to call the Dispose method explicitly.
After the Dispose method has been called on an objectn we should suppress calls to the Finalize method by invoking GC.SuppressFinalize method as a measure of performance optimization.

```
public class Base : IDisposable
    {
        private bool isDisposed = false;
        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
        protected virtual void Dispose(bool disposing)
        {
            if (!isDisposed)
            {
                if (disposing)
                {
                    // Code to dispose the managed resources
                    // held by the class
                }
            }
            // Code to dispose the unmanaged resources
            // held by the class
            isDisposed = true;
            base.Dispose(disposing);
        }

        ~Base()
        {
            Dispose(false);
        }
    }
```

IV- .NET memory

Stack and heap are portions of the memory. The CLR allocates memory for objects in these parts.
Heap:
Allows objects to be allocated or deallocated in a random order.
Variables allocated on heap have their memory allocated at run time.
Access slower.
Holds reference type.

Stack:
Stack is a LIFO structure.
Variables allocated on the stack are stored directly to the memory.
Access very fast.
Allocation done when program is compiled.
Holds value types.

Struct is a value type. It's fast because stored in the Stack.

V- Inheritance

When one object acquires all the properties and behaviors of a parent object, it is know as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

VI- Polymorphism

If one task is performed in different ways, it is known as polymorphism. For instance : to draw something shape, triangle, rectangle …
It is achieved by using overloading and method overriding.

VII- Abstraction

Hiding internal details and showing functionality is know as abstraction.
Abstract class and interface are used to achieve abstraction.
An abstract class allows to create functionality that subclasses can implement or override.
An interface only allows to define functionality, not implement it.
Whereas a class can extend only one abstract class, it can take advantage of multiple interfaces.

VIII- Encapsulation

Wrapping code and data together into a sigle unit are known as encapsulation.

IX- Coupling

It references to the dependency of another class. When classes are aware of each others.
If a class has the details information of another class, there is strong coupling.
Interface are used for weaker coupling because there is no concrete implementation.

X- Multithreading

For long time, most applications were single-threaded. You could never do computation A until completing computation B.
Microprocessors speed are rising up, technology needs to find new ways to be able to optimize speed and performance in computer technology. Thus the need to understand multithreading.

Problems with threading :

- Thread safety
- Race conditions
- Dead locks :
  Deadlock is a situation where two or more threads are frozen in their execution because they are waiting for each other to finish.

One of the tools in .NET to handle multithreading is the Task Parallel Library (TPL). It provides a higher level of abstraction. It provides Task which is an easier way to execute something asynchronously and in parallel compare to a thread.

XI- Garbage Collector

It is an automatic memory manager. Garbage collection is a process of realsing the memory used by the objects that are no longer referenced.

XII- SQL

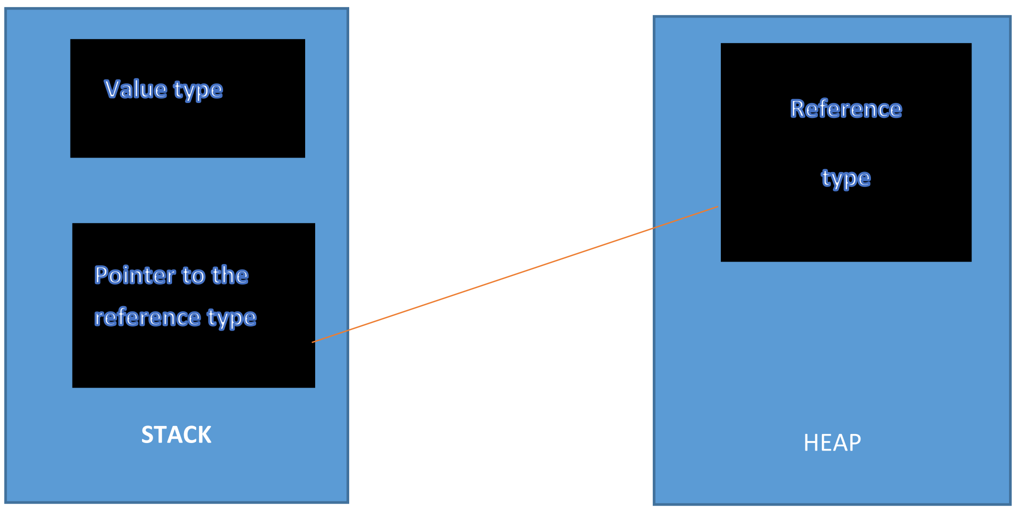
- SQL stored procedure :
  - It is a collection of SQL statements and sql command logic, which is compiled and stored on the database.
  - It allows to create SQL queries to be stored and executed on the server.
- Index :
  - Are used to speed up searching in the database.
  - It can be used to efficiently find all rows matching some column in the query and then walk through only the subset of the table to find exact matches.
  - It we don't have indexes on any column in the WHERE clause, the SQL server has to walk through the whole table and check every row to see if it matches.
  - Clustered index :
    - It is related to how the data is stored (physical ordered storage of all of the data for the table)
    - There is only one of them possible per table
  - Non-clustered index :
    - We can have many of them
    - They then point at the data in the clustered index

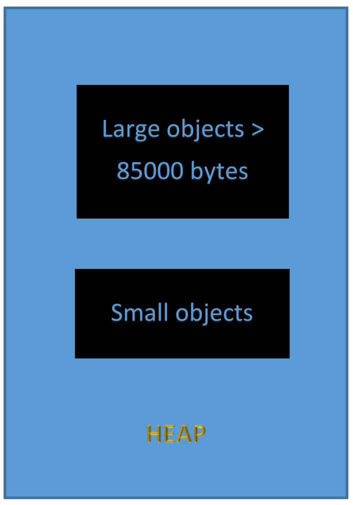
3-C# language and memory management in .NET (part 1)

Stack and Heap
Stack and heap are portions of the memory. The CLR allocates memory for objects in these parts.

Stack is a simple LIFO (last-in-first-out) structure. Variables allocated on the stack are stored directly to the memory and access to this memory is very fast. Its allocation is done when the program is compiled.

Heap allows objects to be allocated or deallocated in a random order. Variables allocated on the heap have their memory allocated at run time and accessing this memory is a bit slower. Its size is only limited by the size of virtual memory. The heap requires a garbage collector.





Value Type and Reference Type

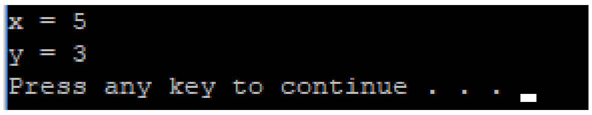
A value type holds the data within its own memory location.
Value types are : bool, byte, char, decimal, double, float, int, long, uint, ulong, ushort, enum, struct

A reference type contains a pointer to another memory location that holds the real data.
Reference types are : class, internface, delegate, string, object, dynamic, arrays

If we assign a value type variable to another variable, the value is copied directly.

```
static void Main(string[] args)
        {
            int x = 3;
            int y = x;
            x = 5;
            Console.WriteLine($"x = {x}");
            Console.WriteLine($"y = {y}");
        }
```

Output :

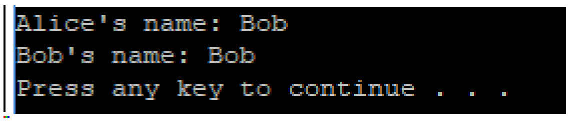


```
x = 5
y = 3
Press any key to continue . . .
```

If we assign a reference type variable to another, as reference type variables represent the address of the variable, the reference is copied and both variables point to the same location of the heap.

```
class Person
        {
            public string Name { get; set; }
            public int Age { get; set; }
        }

        static void Main(string[] args)
        {
            Person alice = new Person { Name = "Alice", Age = 36 };
            Person bob = alice;
            bob.Name = "Bob";

            Console.WriteLine($"Alice's name: {alice.Name} \nBob's name:
{bob.Name}");
        }
```

Output :



```
Alice's name: Bob
Bob's name: Bob
Press any key to continue . . .
```

Finalizer & Dispose

Why these are needed and used ?

Garbage Collector (we will talk about the details in the next part) handles memory management in .NET Framework, it's responsible for the allocation and reclaiming of memory.

However, when we use unmanaged resources such as windows, files, network and database connections, we need to release those resources explicitly after using them.

It can be done by ;

- Using finalizers
- Implementing Dispose method from IDisposable interface

Finalizer :

They are also called destructors are used to perform final clean-uup when a class instance is collected by the Garbage Collector :

- Only one finalizer per class
- It doesn't have any modifiers or parameters
- They cannot be called explicitly, instead, they are called by the Garbage Collector when it considers the object eligible for finalization. They are called also when the program finishes.

Example :

```
public class MyClass
    {
        // properties
        public string UnderlyingName { get; set; }
        public string UnderlyingType { get; set; }

        // constructor
        public MyClass()
        {
            // init
        }
        // finalizer
        ~MyClass() {
            // cleanup
        }
    }
```

Finalizer overrides the Finalize method of the base class.

The finalizer is called recursively from most deriverd to least derived.

Dispose method

We should release explicitly external resources. We can do this by implementing Dispose method from *IDisposable* internface.

```
public class DatabaseConnection : IDisposable
    {
        private bool disposedValue = false; // to detect redundant calls

        protected virtual void Dispose(bool disposing)
        {
            if (!disposedValue)
            {
                Console.WriteLine("This is the first call to Dispose");
                if (disposing)
                {
                    // dispose managed objects
                    Console.WriteLine("Dispose is called by the user");
                }
                else
                {
                    Console.WriteLine("Dispose is called through finalizer");
                }
                // free unamanaged resources
                Console.WriteLine("Unmanaged rsources are cleaned up here");
                disposedValue = true;
            }
            else
            {
                Console.WriteLine("Disposed is called more than one time, no
clean needed")
            }
        }

        ~DatabaseConnection()
        {
            Dispose(false);
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
    }
```

DisposedValue boolean variable allows that clients can call the method multiple times without getting an exception.
The dispose override is either called by the client or the finalizer.
Explicit Garbage Collector to suppress the finalization of this object is done because the client explicitly forces the release of resources.

```
public class Program
  {
      public static void Main(string[] args)
      {
          var connection = new DatabaseConnection();
          try
          {
              // some process
          }
          finally
          {
              connection.Dispose();
          }
      }
  }
```

Another commonly used method to call Dispose is using *using* statement :

```
public class Program
  {
      public static void Main(string[] args)
      {
         using (var connection = new DatabaseConnection())
          {
              // some code
          }
      }
  }
```

Garbage Collector
.NET's GC manages the allocation and release of memory.
It enables to develop without having to free memory.
It reclaims objects that are no longer used, clears their memory.

It provides memory safety by making sure an object cannot use the content of another object.

The Garbage Collector is triggered if one of these conditions are reached :

- The system has low physical memory
- Memory used by allocated objects on the managed heap is greater than threshold
- GC.Collect method called

When the garbage collection starts, all managed threads are suspendend except for the thread that launched the garbage collection.

Generations

The heap is organized into generations.

Generational GC : Young objects die young, reclamation algorithm should not waste time on old objects. Copying survivors is cheaper than scanning corpses.

There are 3 generations :

- Generation 0 :
  - Youngest generation
  - Contains short lived objects (temp variables)
  - Garbage Collection more frequent
- Generation 1 :
  - Acts like a buffer between short lived objects and long lived objects
  - Contains objects that survive to generation 0.
- Generation 2 :
  - Contains long lived objects (like static variables that live for the duration of the process)
  - Objects surviving to Gen2 remain Gen2

Garbage Collector work :

Marking: Finds and creates a list of alive objects

Relocating : Updates the references to the objects that will be compacted

Compacting : Reclaims memory occupied by dead objects and compacts surviving objects

Are objects still alive ?

The GC uses the following information :

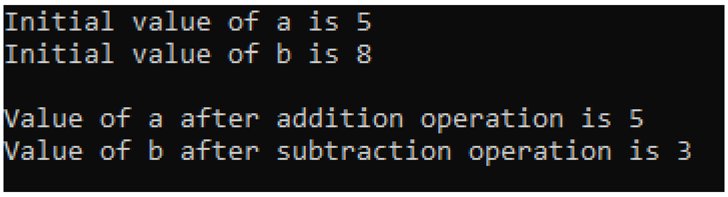
- Stack roots : stack variables provided by the JIT compiler
- Garbage collection handles : Handles that point to managed objects, that be allocated by user code of by the CLR
- Static data : static objects in application domains that could be referencing other objects

Ref keyword :

The ref keyword in C# is used for passing or returing references of values to or from methods.

It means that any change made to a value that is passed by reference will reflect this change since we are modifying the value at the address and not just the value.

```
static void Main(string[] args)
        {
            // Initialize a and b
            int a = 5, b = 8;
            // Display initial values
            Console.WriteLine("Initial value of a is {0}", a);
            Console.WriteLine("Initial value of b is {0}", b);
            Console.WriteLine();
            // Call addValue method by value
            addValue(a);
            // Display modified value of a
            Console.WriteLine($"Value of a after addition operation is {a}");
            // Call subtractValue method by ref
            subtractValue(ref b);
            // Display modified value of b
            Console.WriteLine($"Value of b after subtraction operation is {b}");
            Console.ReadLine();
        }
        public static void addValue(int a)
        {
            a += 10;
        }
        public static void subtractValue(ref int b)
        {
            b -= 5;
        }
```



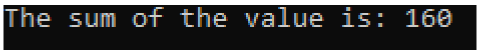
```
Initial value of a is 5
Initial value of b is 8

Value of a after addition operation is 5
Value of b after subtraction operation is 3
```

Out keyword:

The out keyword in C# is used for passing arguments to methods as a reference type. It is generally used when a method returns multiple values.

```
static public void Main()
        {

            // Declaring variable
            // without assigning value
            int G;

            // Pass variable G to the method
            // using out keyword
            Sum(out G);

            // Display the value G
            Console.WriteLine($"The sum of the value is: {G}");
        }
        public static void Sum(out int s)
        {
            s = 80;
            s += s;
        }
```

Output:



```
The sum of the value is: 160
```

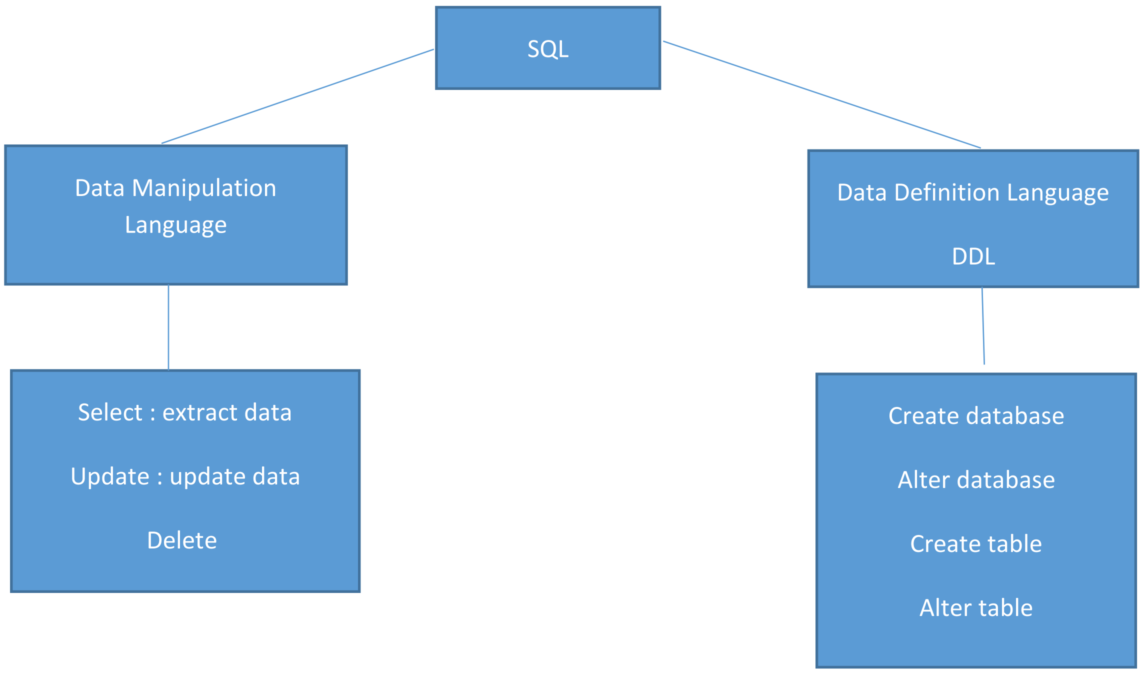
Option pricing application requirements :

- SQL Server Express
- Visual Studio
- SQL Server Management Studio

3- SQL basics, data model creation

SQL stands for Structured Query Language.
It is an ANSI standard (American National Standards Institute)

RDBMS : Relational Database Management System
Table : Collection of related data entries and it consists of columns and rows.
SQL is not case sensitive



SQL statements return resultSet.

Result stored in a result table.

Select data :

SELECT col_names FROM table_name

Select distinct :

Some columns may contain duplicate values

SELECT DISTINCT col_names FROM table_name

Returns distinct values

Text fields :

Declared around ‘ ‘

Numeric field :

No ‘ ‘

Where :

SELECT col_names FROM table_name WHERE colName1 OPERATOR value

Operators for WHERE clause :

= equal

<> NOT EQUAL

> greater than

>= greater or equal

< less than

<= less or equal

Between : range (name Between ‘Bernard’ AND ‘David’)

Like : pattern

IN : result in a defined list : WHERE column_name IN (value1, value2, value3, ...)

AND : WHERE condition1 AND condition2

OR : WHERE condition1 OR condition2

Order by :

Used to sort the resultset by a specified column

By default : Ascending order

If descending order than use DESC

Insert into :

Used to insert new row in a table

a- INSERT INTO table_name VALUES(value1, value2,….)
b- INSERT INTO table_name (colName1, colName2,…) VALUES(value1, value2,…)
   Enables to add data only in specific columns

Update :

UPDATE tableName SET col1=value1, col2=value2, … WHERE someColumn=someValue

Delete:

DELETE FROM tableName WHERE someCo=someValue

DELETE * FROM tableName

TOP

Specify number of records to return

SELECT TOP nbr|percent column_nmaes FROM tableName

Like

Used as Pattern

SELECT * FROM tableName WHERE colName LIKE ‘s%’

Means where value starts with s

% means * char

_ means 1 char

‘[bsp]% means char list specified wildcard

IN

Allows to specify multiple value in WHERE clause

SELECT column_names FROM tableName WHERE colName IN (value1, value2, …)

Between

Select range of data between 2 values (numbers, text, dates)

SELECT column_Names FROM tableName WHERE colName BETWEEN value1 AND value2

Alias

Alias name can be given to a table or column

SELECT column_Names FROM tableName AS aliasName

SELECT colName AS aliasName FROM tableName

Union

Combines resultSet of 2 or more SELECT statements

- Each SELECT must have the same number of columns
- Columns must have similar data types
- Columns in each SELECT must be in the same order

SELECT column_Names FROM table1

UNION

SELECT column_Nmaes FROM table2

Select Into

Select data fom one table and inserts it into a different table

Example : to create backup copies

SELECT * INTO newTableName FROM oldTableName

Create database

CREATE DATABASE databaseName

Create table

CREATE TABLE tableName

(colName1 dataType1 constraint1,

….

colNameN datatypeN constraintN)

Type : int, varchar(255), …

Constraints

Allow to limit type of data that can go into a table

It is specified when the table is created or when the table is already created by and ALTER TABLE statement

- Not null : By default columns can hold null values
- Unique : Uniquely identifies record in a database table
  At table creation :

  Unique(ColName)

  After table creation

  ALTER TABLE tableName ADD UNIQUE(colName)

  DROP

  Drop a unique constraint

  ALTER TABLE tableName DROP CONSTRAINT columnName
- Primary key :
  - automatically has Unique constraints
  - must contain unique values
  - cannot contain Null values
  - Each table should have only 1 Primary key

  At table creation:

  CREATE TABLE tableName

  (colName1 INT NOT NULL,

  colName2 VARCHAR(255),

  PRIMARY KEY(colName1))

  After table creation:

  ALTER TABLE tableName

  ADD PRIMARY KEY(colName)
- Foreign key:
  - In one table points to a primary key in another table
  - Foreign key constraint are used to prevent actions that would destroy links between tables
  - It prevents invalid data as value needs to be one of the values in the table it points to

  At table creation:

  CREATE TABLE tableName

(colName1 INT NOT NULL,

colName2 VARCHAR(255),

FOREIGN KEY (col) REFERENCES targetable(targetCol))

After table creation:

ALTER TABLE tableName

ADD FOREIGN KEY(col) REFERENCES targetable(targetCol)

- Check : used to limit value range that can be placed in a column
  Example : CHECK (columnName > 0)
- Default : to insert a default value into columns
  At table creation :
  DEFAULT ‘0’
  After table creation:
  ALTER TABLE tableName
  ALTER colName SET DEFAULT ‘value’

Indexes :

Indexes are used to find data fast

Truncate :

Delete data inside table and not the table

TRUNCATE TABLE tableName

Alter table : delete, add, modify, colName

Example :

ALTER TABLE tableName

ADD colName datatype

DROP COLUMN colName

ALTER COLUMN colName DATATYPE

Auto increment

Used to increment a column value whenever a new record is added

Example:

Auto increment primary key (used as in identity)

CREATE TABLE tableName

(col1 int NOT NULL AUTO_INCREMENT,

…

PRIMARY KEY(col1))

By default the increment starts at 1. It can be changes as follow

ALTER TABLE tableName AUTO_INCREMENT = 100

Col1 will be auto filled.

SQL functions:

Aggregate functions

- AVG() : average values :: SELECT AVG(colNames) FROM tableName
- COUNT() : counts number of rows :: SELECT COUNT(colName) FROM tableName or COUNT(*)
- First() : first value
- Last() : last value
- MAX() : maximum value
- MIN() : minimum value
- SUM() : sums values :: SELECT SUM(colName) FROM tableName

Scalar functions:

UCASE : upper case :: SELECT UCASE(colName) FROM tableName

LCASE : to lower case

MID() : middle

LEN(): length

ROUND() : rounding

NOW() : date

FORMAT() : format value :: SELECT FORMAT(colName, format) FROM tableName

GROUP BY:

It allows to group result sets by one or more columns.

SELECT colName, aggregate_function(colName)

FROM tableName WHERE colName OPERATOR value

GROUP BY colName

Having clause:

WHERE clause cannot be used with aggregate functions. HAVING was introduced

Stored procedures	Functions
- Can have both inputs and outputs - Can use DML - Cannot be used in a SELECT - Can use TRY CATCH - Can use TRANSACTION procedures - Cannot join Procedures - Can return n values (max(1024)) - Cannot be called from Functions	- Can have only inputs - A - Can be used in a SELECT - A - Cannot use TRANSACTION procedures - Can join functions - Can return only one value (mandatory) - Can be called from procedure

DELETE	TRUNCATE
- Removes rows from table based on WHERE - Uses lock on the row to delete	- Removes all rows from a table - Uses table lock

TRANSACTION

A transaction is a set of DB operations that are all executed on none of them

Example:

Bank -A -------------- +A Client

Both credit and debit operations are done or none of them.

ACID

Atomicity, Consistency, Isolation, Durability

Set of properties that guarantee DB transactions are reliable.

SQL Join

Used to query data from 2 or more tables based on a relationship between certain columns in these tables.

JOIN : returns rows when there is at least one match in both tables

LEFT JOIN : All rows from left table even if no match in right table

RIGHT JOIN : All rows from right table even if no match in left table

FULL JOIN : returns rows if match in one of the tables

INNER JOIN : returns rows when at least one match in both tables

Views

A view is a virtual table based on a result set of an SQL statement

CREATE VIEW viewName AS

SELECT colNames FROM tableNAme WHERE condition

To use above view :

SELECT * FROM viewName

CREATE OR REPLACE VIEW viewName AS …

DROP VIEW viewName

5- Introduction to an option pricing app, SQL Server data model, contrainsts, foreign/primary keys

Our Option pricing application will provide 3 different option pricing models.

The goal is to offer a pricing service for clients/traders and allow them to insert/delete options and price/reprice options.

First, before starting to create tables we need to define what are the entities involved in our application.

We would have the following entities:

- Option
- Underlying
- Product
- Price
- Pricing model
- Option type

What are option's parameters ?

- Strike: float
- Risk free rate: float
- Maturity: Date
- Volatility: float
- Underlying: varchar(max)

What are underlying's parameters?

- Underlying Name: varchar(max)
- Underlying product type: varchar(max)
- Underlying spot: float

What are the product's parameters?

- Product type: varchar(max)

What are the price's parameters?

- Price: float
- Pricing model: varchar(max)

- Option priced

What are the pricing model's parameters?

- Pricing model name: varchar(max)

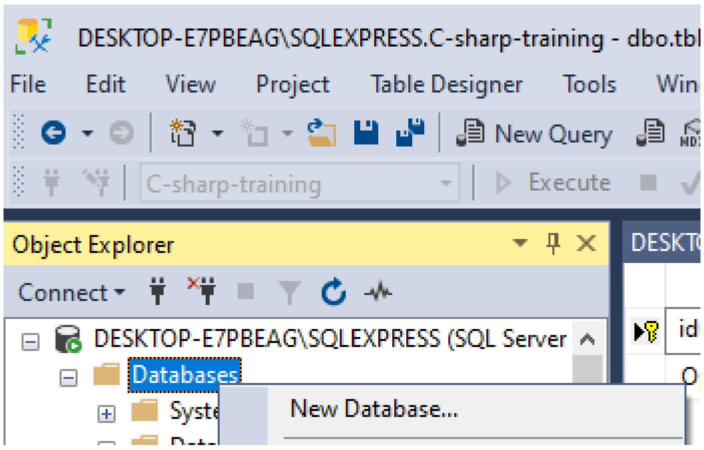
What are the option type parameters?

- Option type name: varchar(max)

Data base creation:

Open Microsoft SQL Server Management Studio

Right click on Databases node and hit new database



Name your database (here we named it C-sharp-training)

Table creation:

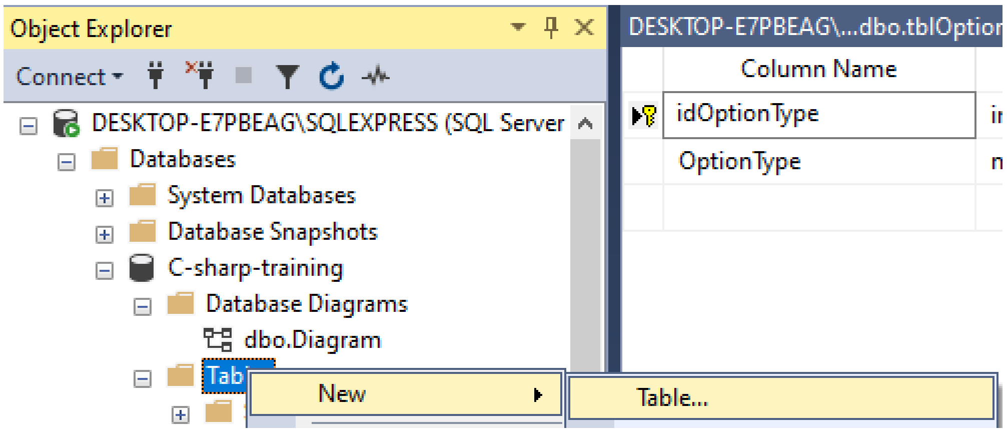
We should begin with the tables that have no link with other tables and create the tables that have links to them at the end.

Here we should begin with the following tables:

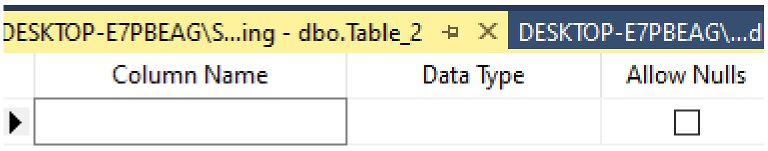
- Option type table
- Product table (underlying type)
- Pricing model table

Convention: table names should be tbl<name>

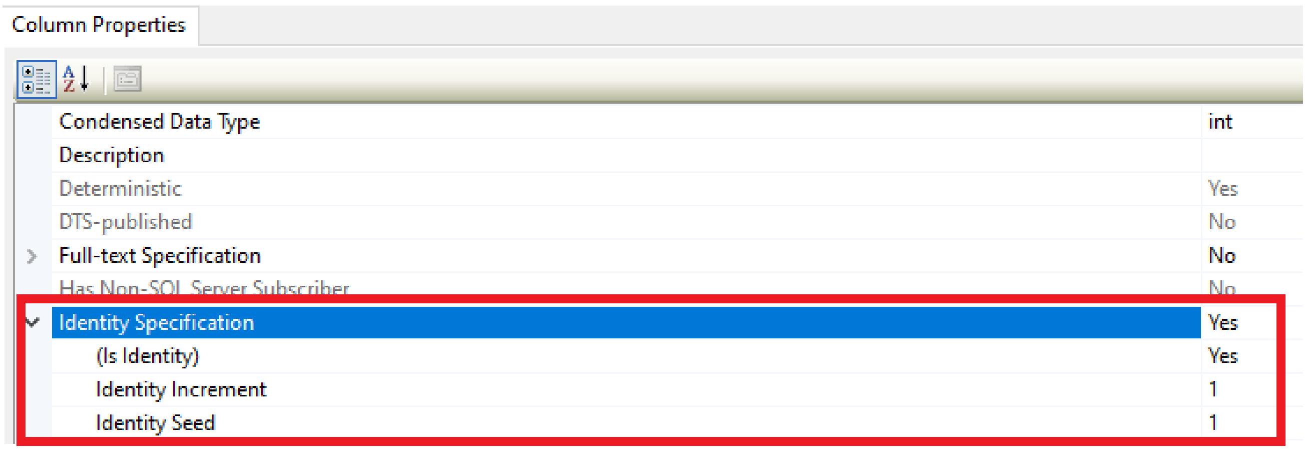
1- tblOptionType:



This will open a blank design view:



- Column 1
  We need to define a primary key for the table
  We will use an identity and make it auto increment
  We will name its column idOptionType, its data type is int and it shouldn't take null values
  To make it auto increment, we should enable the future under column properties:

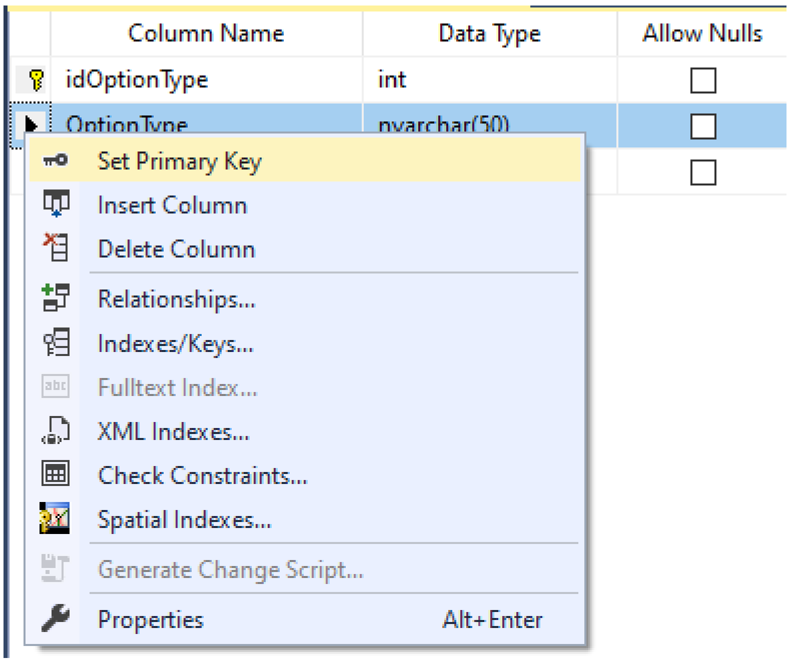

Column Properties

Condensed Data Type	int
Description	
Deterministic	Yes
DTS-published	No
Full-text Specification	No
Has Non-SQL Server Subscriber	No
Identity Specification	Yes
(Is Identity)	Yes
Identity Increment	1
Identity Seed	1

Identity increment: is the increment value

Identity Seed: is the beginning value

We should make it primary key as well.
To do so, right click and set it as primary key



- Column 2
  OptionType of type nvarchar(50) which will contain the option type (European call, American call …)

2- tblProduct

- Column1 : idProduct which will be an auto increment and set as primary key
- Column2 : productType which will contain the underlying type, its type is nvarchar(max) and can't contain null values

3- tblPricingModel

- Column 1 : idPricingModel which will be an auto increment and set as primary key
- Column 2 : pricingModel which will contain the pricing model used to price an option, its type is nvarchar(max) and can't contain null values

SQL request:

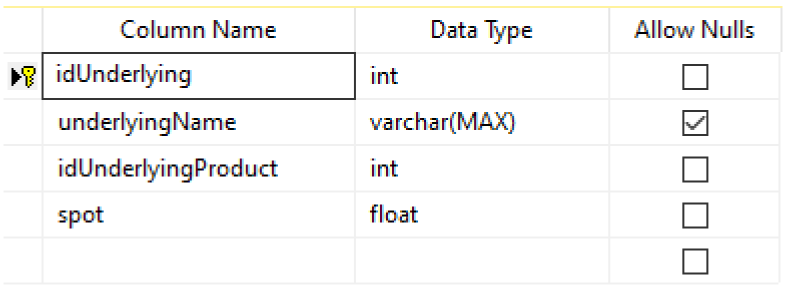
```
CREATE TABLE [dbo].[tblPricingModel](
	[idPricingModel] [int] IDENTITY(1,1) NOT NULL,
	[pricingModel] [nvarchar](max) NULL,
 CONSTRAINT [PK_tblPricingModel] PRIMARY KEY CLUSTERED
(
	[idPricingModel] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS
= ON, ALLOW_PAGE_LOCKS = ON, OPTIMIZE_FOR_SEQUENTIAL_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
```

```
GO
```

Now, we shall begin creating the tables that have relations with above base tables.

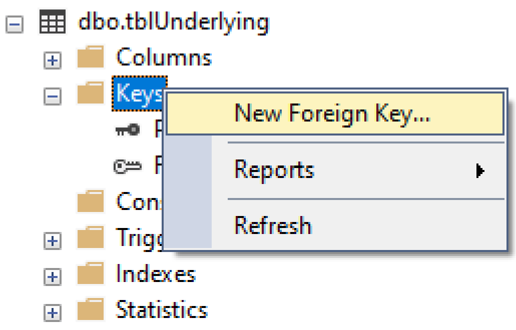
4- **tblUnderlying**

Underlying table should have an entry that specifies the underlying type. To do that we should reference the id of our underlying type in the **tblProduct**

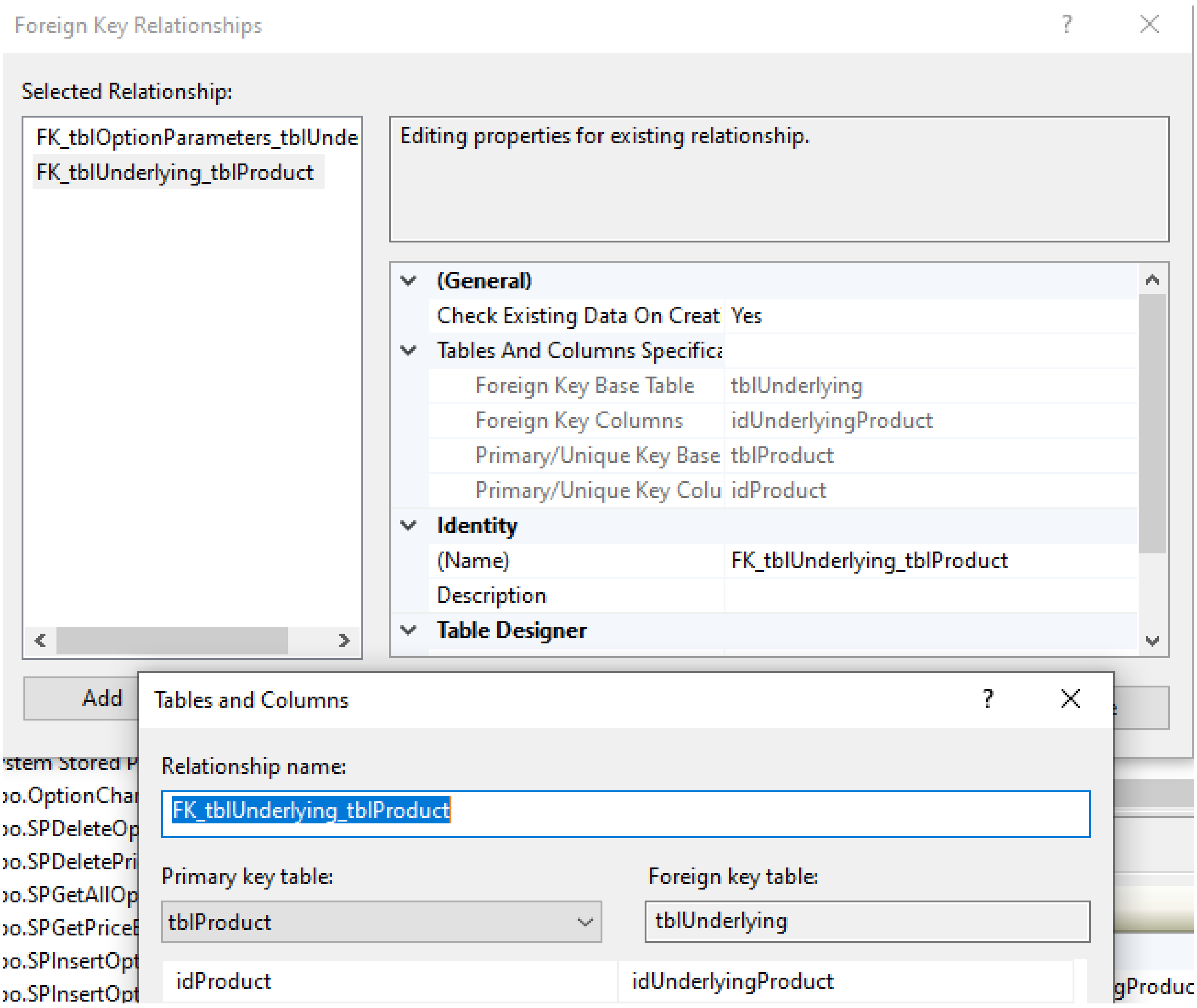


Column Name	Data Type	Allow Nulls
idUnderlying	int	☐
underlyingName	varchar(MAX)	☑
idUnderlyingProduct	int	☐
spot	float	☐
		☐

To make idUnderlyingProduct references the primary key of **tblProduct**, we need to add a foreign key. To do so, unfold tblUnderlying node and right click on Keys and hit New Foreign Key ...



Give it a name and add the reference:



Foreign Key Relationships

Selected Relationship:

FK_tblOptionParameters_tblUnde
FK_tblUnderlying_tblProduct

Editing properties for existing relationship.

(General)	
Check Existing Data On Creat	Yes
Tables And Columns Specifica	
Foreign Key Base Table	tblUnderlying
Foreign Key Columns	idUnderlyingProduct
Primary/Unique Key Base	tblProduct
Primary/Unique Key Colu	idProduct
Identity	
(Name)	FK_tblUnderlying_tblProduct
Description	
Table Designer	

Tables and Columns

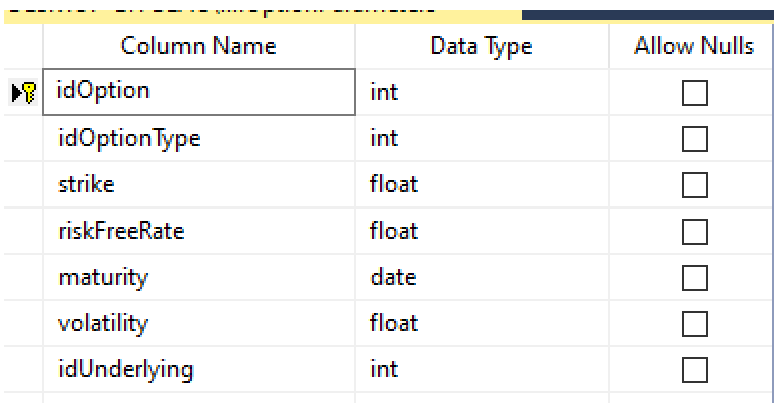
Relationship name:

FK_tblUnderlying_tblProduct

Primary key table:	Foreign key table:
tblProduct	tblUnderlying
idProduct	idUnderlyingProduct

5- tblOptionParameters

Now that we have the tables for underlying, option type. We can create **tblOptionParameters**



Column Name	Data Type	Allow Nulls
idOption	int	☐
idOptionType	int	☐
strike	float	☐
riskFreeRate	float	☐
maturity	date	☐
volatility	float	☐
idUnderlying	int	☐

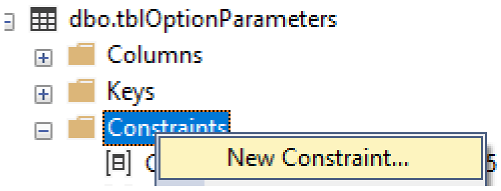
idOption is an auto increment and is a primary key.
idOptionType has a foreign key that references the id of **tblOptionType**
idUnderlying has a foreign key that references the id of **tblUnderlying**

Constraints:

To make our data clean, we need to define constraints to prevent inserting corrupted data

- maturity: an option maturity cannot be in the past because it would be priceless

to do so: right click on Constraints and add a new Constraint



Name the constraint

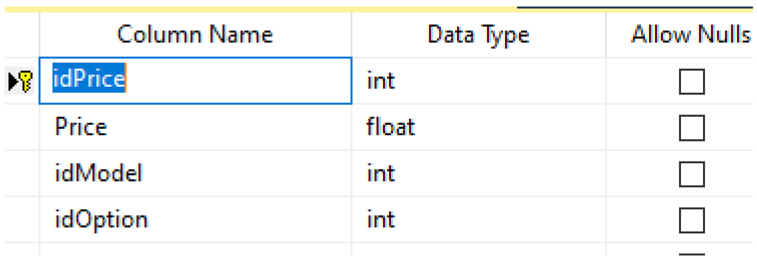
Add the expression:

([maturity]>getdate())

We should do the same for strike, risk free rate and volatility as they cannot be negative

6-tblPrice

Finally we can create the Price table that would store the price for a given option with a given pricing model

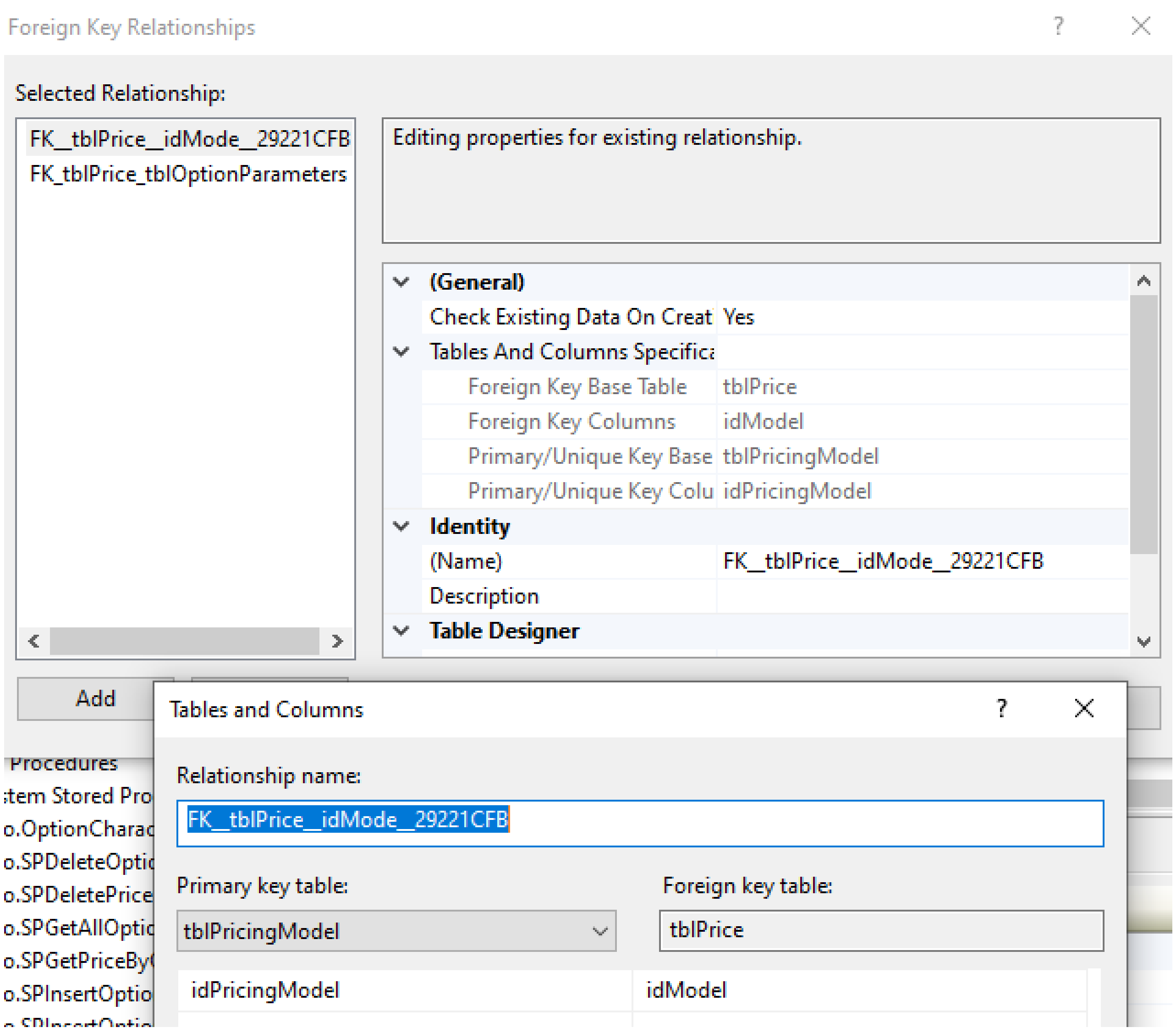


Column Name	Data Type	Allow Nulls
idPrice	int	☐
Price	float	☐
idModel	int	☐
idOption	int	☐

idPrice is an auto increment and is also a primary key

Price is the option price

idModel references the primary key of **tblPricingModel**


Foreign Key Relationships

Selected Relationship:

FK__tblPrice__idMode__29221CFB
FK_tblPrice_tblOptionParameters

Editing properties for existing relationship.

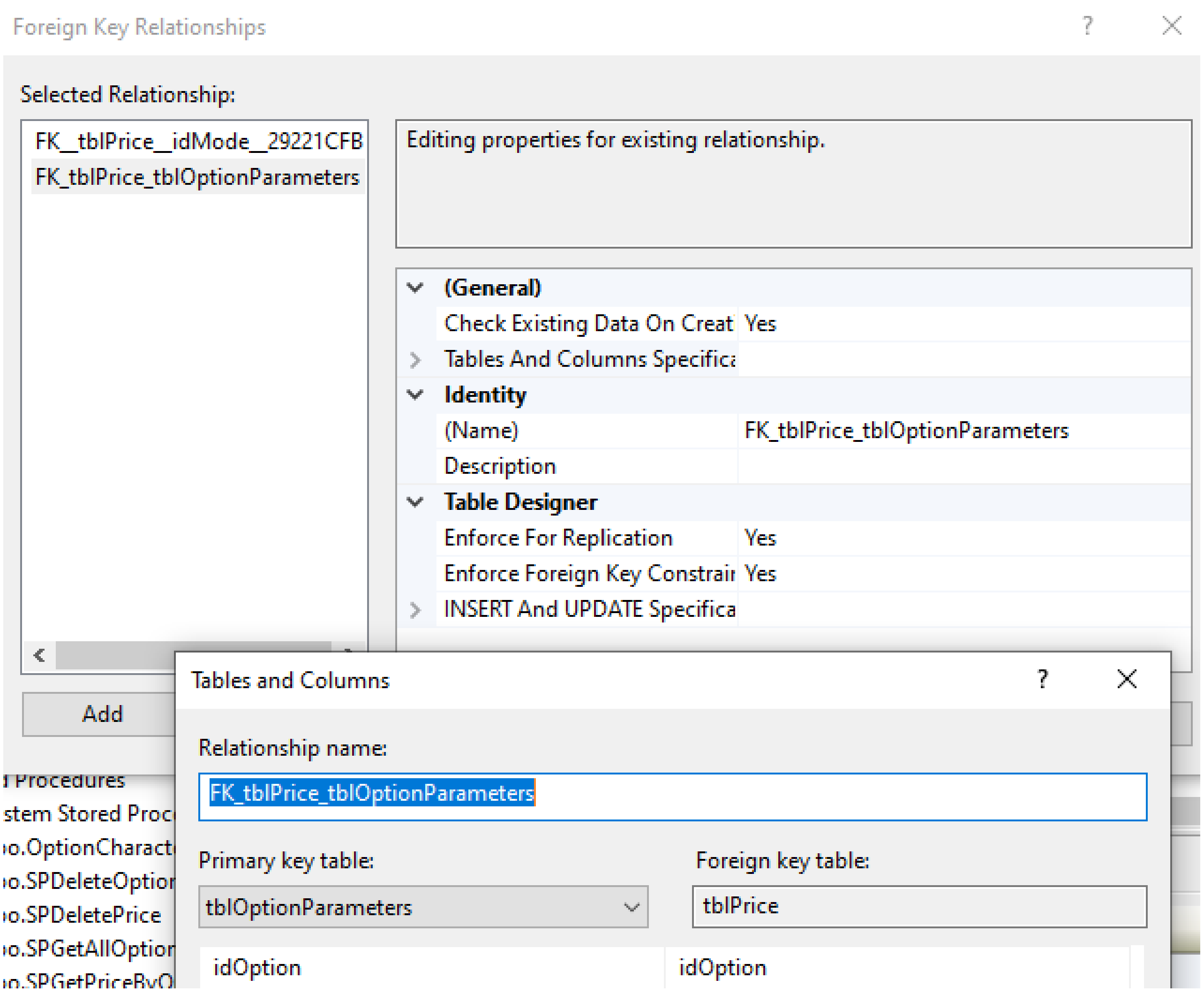
(General)	
Check Existing Data On Creat	Yes
Tables And Columns Specifica	
Foreign Key Base Table	tblPrice
Foreign Key Columns	idModel
Primary/Unique Key Base	tblPricingModel
Primary/Unique Key Colu	idPricingModel
Identity	
(Name)	FK__tblPrice__idMode__29221CFB
Description	
Table Designer	

Tables and Columns

Relationship name: FK__tblPrice__idMode__29221CFB

Primary key table:	Foreign key table:
tblPricingModel	tblPrice
idPricingModel	idModel

idOption references the primary key of **tblOption**



Foreign Key Relationships

Selected Relationship:

FK__tblPrice__idMode__29221CFB
FK_tblPrice_tblOptionParameters

Editing properties for existing relationship.

(General)	
Check Existing Data On Creat	Yes
Tables And Columns Specifica	
Identity	
(Name)	FK_tblPrice_tblOptionParameters
Description	
Table Designer	
Enforce For Replication	Yes
Enforce Foreign Key Constrair	Yes
INSERT And UPDATE Specifica	

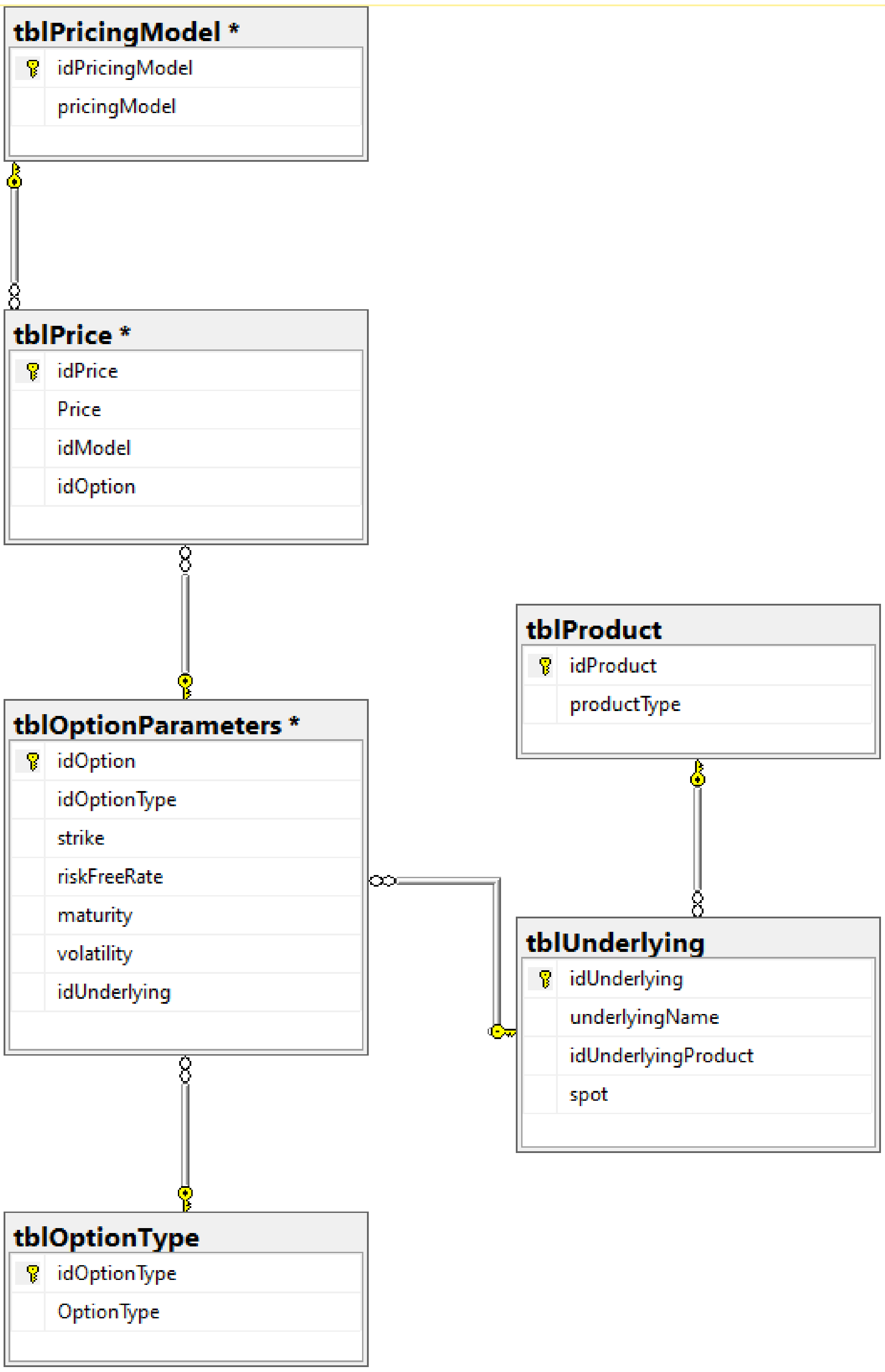
Tables and Columns

Relationship name:

FK_tblPrice_tblOptionParameters

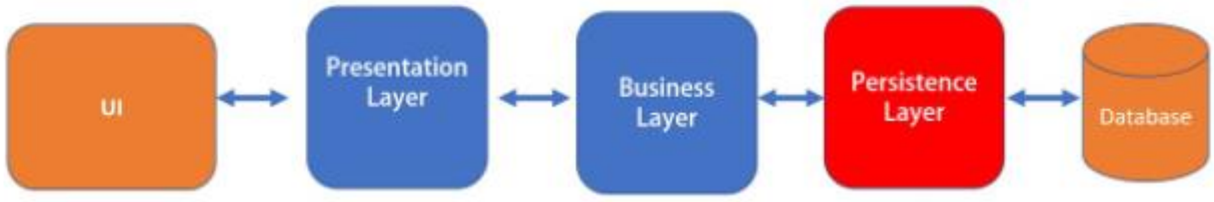
Primary key table:	Foreign key table:
tblOptionParameters	tblPrice
idOption	idOption

The final database diagram is:


tblPricingModel *
idPricingModel
pricingModel
tblPrice *
idPrice
Price
idModel
idOption
tblProduct
idProduct
productType
tblOptionParameters *
idOption
idOptionType
strike
riskFreeRate
maturity
volatility
idUnderlying
tblUnderlying
idUnderlying
underlyingName
idUnderlyingProduct
spot
tblOptionType
idOptionType
OptionType

6-SQL server stored procedures, DAO layer (part 1)

Our application's architecture is as follow:



In this chapter, we will be focusing on the Persistence Layer or DAO (Data Access Object) and how to achieve performance with inserting/updating the database.

Data Access Object Pattern or DAO pattern is used to separate low level data accessing API or operations from high level business services. Following are the participants in Data Access Object Pattern.

- Data Access Object Interface - This interface defines the standard operations to be performed on a model object(s).
- Data Access Object concrete class - This class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.
- Model Object or Value Object - This object is simple POJO containing get/set methods to store data retrieved using DAO class.

But first, we are going to provide database tools to have effective database interactions: stored procedures

A Stored Procedure is a type of code in SQL that can be stored for later use and can be used many times. So, whenever you need to execute the query, instead of calling it you can just call the stored procedure. You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter values that is passed.

The main advantages of stored procedure are given below:

1. Better Performance –
   The procedure calls are quick and efficient as stored procedures are compiled once and stored in executable form. Hence the response is quick. The executable code is automatically cached, hence lowers the memory requirements.

2. Higher Productivity –
   Since the same piece of code is used again and again so, it results in higher productivity.

3. Scalability –
   Stored procedures increase scalability by isolating application processing on the server.

4. Maintainability –
   Maintaining a procedure on a server is much easier then maintaining copies on various client machines, this is because scripts are in one location.

5. Security –
   Access to the data can be restricted by allowing users to manipulate the data only through stored procedures that execute with their definer's privileges.

Convention: stored procedure names being with SP

1- Insert Product

```
CREATE PROCEDURE [dbo].[SPInsertProduct]
	-- Add the parameters for the stored procedure here
	@ProductType varchar(32)
AS
BEGIN
	-- SET NOCOUNT ON added to prevent extra result sets from
	-- interfering with SELECT statements.
	SET NOCOUNT ON;
	Declare @output int;
	Select @output = idProduct from tblProduct where productType = @ProductType;
	If (@@ROWCOUNT <> 0)
		Begin
			return @output;
		END
	Insert into tblProduct (productType) Values(@ProductType);
	Select @output = SCOPE_IDENTITY();
	return @output;
END
```

Here, we check first if the product we are about to insert is not already in the table.
If it doesn't exist, then we insert it
Else, we return the id of the existing product.

Key elements:
@@ROWCOUNT is a global variable that gives the row count of last executed statement
SCOPE_IDENTITY() returns the id of the last record inserted
The stored procedure returns the id (newly inserting or the one of the existing)

2- Insert Underlying

To insert a new underlying, we need its name, its spot and its type

Here, first we will call the stored procedure to insert a product, (it will either returns the id of the row if the type already exists, or insert a new record and returns the id of this new record)

Then, we need to check if there is no identical underlying in the table, if it doesn't exist => insert type, else return the id of the already existing underlying record.

```
CREATE PROCEDURE [dbo].[SPInsertUnderlying]
	@underlyingName varchar(max),
	@spot float,
	@product varchar(32)
AS
BEGIN
	SET NOCOUNT ON;

	Declare @output int;
	Declare @idProduct int;
	Exec @idProduct = SPInsertProduct @product;
	Select @output = idUnderlying from tblUnderlying where idUnderlyingProduct =
@idProduct AND spot = @spot AND underlyingName = @underlyingName;
	If (@@ROWCOUNT <> 0)
		Begin
			return @output;
		End
	Insert into tblUnderlying (underlyingName, idUnderlyingProduct, spot)
Values(@underlyingName, @idProduct, @spot);
	Select @output = SCOPE_IDENTITY();
	return @output;
END
```

3- Insert pricing model

```
CREATE PROCEDURE [dbo].[SPInsertPricingModel]
	@pricingModel varchar(max)
AS
BEGIN
	-- SET NOCOUNT ON added to prevent extra result sets from
	-- interfering with SELECT statements.
	SET NOCOUNT ON;
	Declare @output int;
	Select @output = idPricingModel from tblPricingModel where pricingModel =
@pricingModel;
```

```
	If (@output is not null)
		Begin
			return @output;
		END
	Insert into tblPricingModel (pricingModel) Values(@pricingModel);
	Select @output = SCOPE_IDENTITY();
	return @output;
END
```

4- Insert option type

```
CREATE PROCEDURE [dbo].[SPInsertOptionType]
	-- Add the parameters for the stored procedure here
	@optionType varchar(32)
AS
BEGIN
	-- SET NOCOUNT ON added to prevent extra result sets from
	-- interfering with SELECT statements.
	SET NOCOUNT ON;
	Declare @output int;
	Select @output = idOptionType from tblOptionType where OptionType =
@optionType;
	If (@@ROWCOUNT <> 0)
		Begin
			return @output;
		END
	Insert into tblOptionType (OptionType) Values(@optionType);
	Select @output = SCOPE_IDENTITY();
	return @output;
END
```

5- Insert option parameters

Following the same logic, we create the stored procedure to insert options

```
ALTER PROCEDURE [dbo].[SPInsertOptionParameters]
	@optionType varchar(max),
	@strike float,
	@riskFreeRate float,
	@maturity Date,
	@volatility float,
	@underlying varchar(max),
	@spot float,
	@product varchar(max)
AS
BEGIN
	SET NOCOUNT ON;

	Declare @output int;
	Declare @idOptionType int;
	Declare @idUnderlying int;
	Exec @idOptionType = SPInsertOptionType @optionType;
	Exec @idUnderlying = SPInsertUnderlying @underlying, @spot, @product;

	Select @output = idOption from tblOptionParameters
	where
	idOptionType = @idOptionType
	AND strike = @strike
	AND riskFreeRate = @riskFreeRate
	AND maturity = @maturity
	AND volatility = @volatility
	AND idUnderlying = @idUnderlying;

	If (@@ROWCOUNT <> 0)
		Begin
			return @output;
		End
	Insert into tblOptionParameters Values(@idOptionType, @strike, @riskFreeRate, 
@maturity, @volatility, @idUnderlying);
	Select @output = SCOPE_IDENTITY();
	return @output;
END
```

6- Insert price

Finally, we can create the stored procedure to insert prices

```
ALTER PROCEDURE [dbo].[SPInsertPrice]
	@price float,
	@model varchar(max),
	@optionType varchar(max),
	@strike float,
	@riskFreeRate float,
	@maturity Date,
	@volatility float,
	@underlying varchar(max),
	@spot float,
	@product varchar(max)
AS
BEGIN
	SET NOCOUNT ON;

	Declare @output int;
	Declare @idModel int;
	Declare @idOption int;
	Exec @idModel = SPInsertPricingModel @model;
	Exec @idOption = SPInsertOptionParameters @optionType,@strike,
	@riskFreeRate, @maturity, @volatility, @underlying,   @spot, @product;
	Select @output = idPrice from tblPrice
	where Price = @price
	AND idModel = @idModel
	AND idOption = @idOption
	If (@output is not null)
		Begin
			return @output;
		End
	Insert into tblPrice (Price, idModel, idOption) Values (@price, @idModel,
@idOption);
	Select @output = SCOPE_IDENTITY();
	return @output;
END
```

The main needs we are going to have from the C# clients is to be able to manipulate Options and Prices. Thus, we create stored procedures to get, delete options and prices.

7- Delete Price

C# clients have no idea on the ids, they only manipulate DAO objects
So with the use of these parameters the stored procedure needs to find out the ids from the table.
Example:
From product type => find product type id
This will allow us to find the underlying record from (underlying name, product, spot)

```
ALTER PROCEDURE [dbo].[SPDeletePrice]
	@price float,
	@model varchar(max),
	@optionType varchar(max),
	@strike float,
	@riskFreeRate float,
	@maturity Date,
        @volatility float,
        @underlying varchar(max),
        @spot float,
        @product varchar(max)
AS
BEGIN
        SET NOCOUNT ON;

        Declare @idPrice int;
        Declare @idModel int;
        Declare @idOption int;
        Declare @idUnderlying int;
        Declare @idUnderlyingProduct int;
        Declare @idOptionType int;

        Select @idOptionType = idOptionType from tblOptionType where OptionType =
@optionType;

        Select @idModel = idPricingModel from tblPricingModel where pricingModel =
@model;

        Select @idUnderlyingProduct = idProduct from tblProduct where productType =
@product;

        Select @idUnderlying = idUnderlying from tblUnderlying
        where underlyingName = @underlying
        and idUnderlyingProduct = @idUnderlyingProduct
        and spot = @spot;

        Select @idOption = idOption from tblOptionParameters
        where idOptionType = @idOptionType
        and strike = @strike
        and riskFreeRate = @riskFreeRate
        and maturity = @maturity
        and volatility = @volatility
        and idUnderlying = @idUnderlying;

        Select @idPrice = idPrice from tblPrice
        where Price = @price
        AND idModel = @idModel
        AND idOption = @idOption;

        if (@idPrice is not null)
                BEGIN
```

```
                        delete from tblPrice where idPrice = @idPrice;
                END


END
```

8- Delete option

Options reference other data: option type, price, underlying

We need to delete the data recursively in order to prevent leaving prices or underlyings related to no option.

```
CREATE PROCEDURE [dbo].[SPDeleteOption]
              @optionType varchar(max),
              @strike float,
              @riskFreeRate float,
              @maturity Date,
              @volatility float,
              @underlying varchar(max),
              @spot float,
              @underlyingType varchar(max),
              @pricingModel varchar(max)
AS
BEGIN
       -- SET NOCOUNT ON added to prevent extra result sets from
       -- interfering with SELECT statements.
       SET NOCOUNT ON;

       Declare @idPricingModel int;
       select @idPricingModel = idPricingModel from tblPricingModel where
pricingModel = @pricingModel;

       Declare @idOptionType int;
       select @idOptionType = idOptionType from tblOptionType where OptionType =
@optionType;

       Declare @idProduct int;
       select @idProduct = idProduct from tblProduct where productType =
@underlyingType;

       Declare @idUnderlying int;
       select @idUnderlying = idUnderlying from tblUnderlying
       where
       underlyingName = @underlying
       and spot = @spot
       and idUnderlyingProduct = @idProduct;
```

```
	Declare @idOption int;
	select @idOption = idOption from tblOptionParameters
	where
	idOptionType = @idOptionType
	and strike = @strike
	and riskFreeRate = @riskFreeRate
	and maturity = @maturity
	and volatility = @volatility
	and idUnderlying = @idUnderlying;

	delete from tblPrice where idOption = @idOption;
	if (@pricingModel is not null)
	begin
		delete from tblPricingModel where idPricingModel = @idPricingModel;
	end
	Declare @optionTypeStillUsed int;
	select @optionTypeStillUsed = count(idOption) from tblOptionParameters where
idOptionType = @idOptionType
	delete from tblOptionParameters where idOption = @idOption;

	if (@optionTypeStillUsed = 0)
		Begin
			delete from tblOptionType where idOptionType = @idOptionType;
		End
	Declare @underlyingTypeStillUsed int;
	select @underlyingTypeStillUsed = count(idUnderlying) from tblUnderlying
where idUnderlying = @idUnderlying

	delete from tblUnderlying where idUnderlying = @idUnderlying;
	if (@underlyingTypeStillUsed = 0)
		BEGIN
		delete from tblProduct where idProduct = @idProduct;
		END
END
```

SQL Views:

In SQL, a view is a virtual table based on the result-set of an SQL statement.
A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.
A view is created with the CREATE VIEW statement.

The view we need to create should be a view business oriented, it means it should not contain any id and should be straight forward, meaning option ⇔ price

```
CREATE VIEW [dbo].[VWOptionPrice] AS
	select optionParam.idOption, optionParam.strike, optionParam.riskFreeRate, 
optionParam.maturity, optionParam.volatility, udl.underlyingName, optionTbl.OptionType, 
udl.spot, udlType.productType, price.Price,
	model.pricingModel
	from tblOptionParameters optionParam
	INNER JOIN tblOptionType optionTbl on optionParam.idOptionType = 
optionTbl.idOptionType
	INNER JOIN tblUnderlying udl on optionParam.idUnderlying = udl.idUnderlying
	INNER JOIN tblProduct udlType on udl.idUnderlyingProduct = udlType.idProduct
	LEFT JOIN tblPrice price on price.idOption = optionParam.idOption
	LEFT JOIN tblPricingModel model on model.idPricingModel = price.idModel
GO
```

9- **Get all options**
Now that we have our view, we can use it through SELECT statements in stored procedures

```
CREATE PROCEDURE [dbo].[SPGetAllOptions]

AS
BEGIN
	-- SET NOCOUNT ON added to prevent extra result sets from
	-- interfering with SELECT statements.
	SET NOCOUNT ON;

	select strike, riskFreeRate, maturity, volatility, underlyingName, 
OptionType, spot, productType from VWOptionPrice order by idOption;
	RETURN
END
```

10- **Get price by option**

```
CREATE PROCEDURE [dbo].[SPGetPriceByOption]
```

```
              @optionType varchar(max),
              @strike float,
              @riskFreeRate float,
              @maturity Date,
              @volatility float,
              @underlying varchar(max),
              @spot float,
              @underlyingType varchar(max),
              @pricingModel varchar(max)
AS
BEGIN
       -- SET NOCOUNT ON added to prevent extra result sets from
       -- interfering with SELECT statements.
       SET NOCOUNT ON;

       select Price from VWOptionPrice
       where strike = @strike
       and riskFreeRate = @riskFreeRate
       and maturity = @maturity
       and volatility = @volatility
       and underlyingName = @underlying
       and optionType = @optionType
       and spot = @spot
       and productType = @underlyingType
       and pricingModel = @pricingModel;

END
```

DAO layer:

Now that we have everything set up on the database side, we can go ahead and start implementing the Data Access Object layer in C#

Implementation

Create a .NET FRAMEWORK solution of Class Library output named **OptionPricingDAO**

First, since we have several types involved in the application. We are going to define them as ENUM.

- ContractEnum : which will act as value for option types
- PricingModelEnum : which will act as value for pricing models

- UnderlyingTypeEnum : which will act as underlying product

C# ENUM can't have underlying type string, so the idea is to create a class with static members that will act as an ENUM.

We need to define ourselves a static method to convert a string to an instance of this class as well as overriding Equals and Hashcode methods

```
namespace OptionPricingDAO.Common
{
       public class ContractEnum
       {
              private static string EUROPEANCALLvalue = "EuropeanCall";
              private static string EUROPEANPUTvalue = "EuropeanPut";
              private static string AMERICANCALLvalue = "AmericanCall";
              private static string AMERICANPUTvalue = "AmericanPut";


        private ContractEnum(string value) { Value = value; }
              public string Value { get; private set; }
              public static ContractEnum EUROPEANCALL { get { return new
ContractEnum(EUROPEANCALLvalue); } }
              public static ContractEnum EUROPEANPUT { get { return new
ContractEnum(EUROPEANPUTvalue); } }
              public static ContractEnum AMERICANCALL { get { return new
ContractEnum(AMERICANCALLvalue); } }
              public static ContractEnum AMERICANPUT { get { return new
ContractEnum(AMERICANPUTvalue); } }

              public static ContractEnum FromString(string str)
        {
                     if (EUROPEANCALLvalue.Equals(str,
StringComparison.InvariantCultureIgnoreCase))
            {
                            return EUROPEANCALL;
            }
                     if (EUROPEANPUTvalue.Equals(str,
StringComparison.InvariantCultureIgnoreCase))
                     {
                            return EUROPEANPUT;
                     }
                     if (AMERICANCALLvalue.Equals(str,
StringComparison.InvariantCultureIgnoreCase))
                     {
                            return AMERICANCALL;
                     }
```

```
                    if (AMERICANPUTvalue.Equals(str,
StringComparison.InvariantCultureIgnoreCase))
                    {
                        return AMERICANPUT;
                    }
                    return null;
                }

        public override bool Equals(object obj)
        {
            return obj is ContractEnum @enum &&
                   Value.Equals(@enum.Value);
        }

        public override int GetHashCode()
        {
            return -1937169414 + Value.GetHashCode();
        }
    }
}
```

The same goes for the 2 remaining enums.

Now, we are going to create a PriceDTO and OptionParametersDTO.

A Data Transfer Object is an object that is used to encapsulate data and send it from one subsystem of an application to another.

```
namespace OptionPricingDAO.DTOs
{
    public class PriceDTO
    {
        private double price;
              private PricingModelEnum model;
              private ContractEnum optionType;
              private double strike;
              private double riskFreeRate;
              private DateTime maturity;
              private double volatility;
              private string underlying;
              private double spot;
              private UnderlyingTypeEnum underlyingType;
        public double Price {
                get { return this.price; }
                set { this.price = value; }
              }
              public PricingModelEnum Model {
                get { return this.model; }
                set { this.model = value; }
              }
              public ContractEnum ContractType {
                get { return this.optionType; }
                set { this.optionType = value; }
              }
              public double Strike {
                get { return this.strike; }
                set { this.strike = value; }
              }
              public double RiskFreeRate {
                get { return this.riskFreeRate; }
                set { this.riskFreeRate = value; }
              }
              public DateTime Maturity {
                get { return this.maturity; }
                set { this.maturity = value; }
              }
              public double Volatility {
                get { return this.volatility; }
                set { this.volatility = value; }
              }
              public string Underlying {
                get { return this.underlying; }
                set { this.underlying = value; }
              }
              public double Spot {
                get { return this.spot; }
                set { this.spot = value; }
              }
              public UnderlyingTypeEnum UnderlyingType
              {
                get { return this.underlyingType; }
                set { this.underlyingType = value; }
              }
```

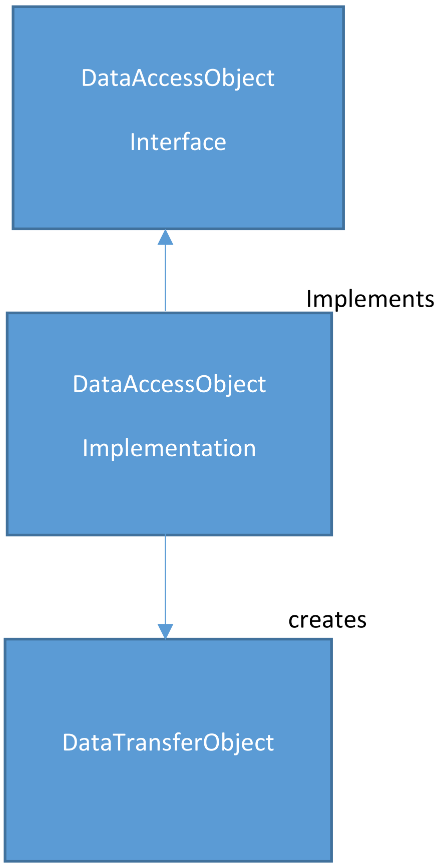
```
              public PriceDTO(double price, PricingModelEnum model, ContractEnum
optionType, double strike, double riskFreeRate, DateTime maturity,
              double volatility, string underlying, double spot, UnderlyingTypeEnum
udlType)
        {
                     this.price = price;
                     this.model = model;
                     this.optionType = optionType;
                     this.strike = strike;
                     this.riskFreeRate = riskFreeRate;
                     this.maturity = maturity;
                     this.volatility = volatility;
                     this.underlying = underlying;
                     this.spot = spot;
                     this.underlyingType = udlType;
        }

              public PriceDTO(OptionParametersDTO optionParametersDTO, double price,
PricingModelEnum model)
        {
                     this.price = price;
                     this.model = model;
                     this.optionType = optionParametersDTO.OptionType;
                     this.strike = optionParametersDTO.Strike;
                     this.riskFreeRate = optionParametersDTO.RiskFreeRate;
                     this.maturity = optionParametersDTO.Maturity;
                     this.volatility = optionParametersDTO.Volatility;
                     this.underlying = optionParametersDTO.Underlying;
                     this.spot = optionParametersDTO.Spot;
                     this.underlyingType = optionParametersDTO.UnderlyingType;
              }

        public override bool Equals(object obj)
        {
            return obj is PriceDTO dTO &&
                   price == dTO.price &&
                   model.Equals(dTO.model) &&
                   optionType.Equals(dTO.optionType) &&
                   strike == dTO.strike &&
                   riskFreeRate == dTO.riskFreeRate &&
                   maturity.CompareTo(dTO.maturity) == 0 &&
                   volatility == dTO.volatility &&
                   underlying == dTO.underlying &&
                   spot == dTO.spot &&
                    underlyingType.Equals(dTO.underlyingType);
        }

        public override int GetHashCode()
        {
            int hashCode = 1808784971;
            hashCode = hashCode * -1521134295 + price.GetHashCode();
            hashCode = hashCode * -1521134295 + model.GetHashCode();
            hashCode = hashCode * -1521134295 + optionType.GetHashCode();
            hashCode = hashCode * -1521134295 + strike.GetHashCode();
            hashCode = hashCode * -1521134295 + riskFreeRate.GetHashCode();
            hashCode = hashCode * -1521134295 + maturity.GetHashCode();
            hashCode = hashCode * -1521134295 + volatility.GetHashCode();
            hashCode = hashCode * -1521134295 + underlying.GetHashCode();
            hashCode = hashCode * -1521134295 + spot.GetHashCode();
            hashCode = hashCode * -1521134295 + underlyingType.GetHashCode();
            return hashCode;
        }
    }
}
```

```
namespace OptionPricingDAO.DTOs
{
    public class OptionParametersDTO
    {
            private ContractEnum optionType;
            private double strike;
            private double riskFreeRate;
            private DateTime maturity;
            private double volatility;
            private string underlying;
            private double spot;
            private UnderlyingTypeEnum underlyingType;
            public OptionParametersDTO(ContractEnum optionType, double strike, double
riskFreeRate, DateTime maturity,
                    double volatility, string underlying, double spot, UnderlyingTypeEnum
underlyingType)
        {
                    this.optionType = optionType;
                    this.strike = strike;
                    this.riskFreeRate = riskFreeRate;
```

```
                this.maturity = maturity;
                this.volatility = volatility;
                this.underlying = underlying;
                this.spot = spot;
                this.underlyingType = underlyingType;
        }
        public ContractEnum OptionType {
                get { return this.optionType; }
                set { this.optionType = value; }
            }
            public double Strike {
                get { return this.strike; }
                set { this.strike = value; }
            }
            public double RiskFreeRate {
                get { return this.riskFreeRate; }
                set { this.riskFreeRate = value; }
            }
            public DateTime Maturity {
                get { return this.maturity; }
                set { this.maturity = value; }
            }
            public double Volatility {
                get { return this.volatility; }
                set { this.volatility = value; }
            }
            public string Underlying {
                get { return this.underlying; }
                set { this.underlying = value; }
        }
        public double Spot {
                get { return this.spot; }
                set { this.spot = value; }
            }
            public UnderlyingTypeEnum UnderlyingType {
                get { return this.underlyingType; }
                set { this.underlyingType = value; }
            }

        public override bool Equals(object obj)
        {
                return obj is OptionParametersDTO dTO &&
                       optionType.Equals(dTO.optionType) &&
                                   strike == dTO.strike &&
                                   riskFreeRate == dTO.riskFreeRate &&
                                   maturity.CompareTo(dTO.maturity) == 0 &&
                                   volatility == dTO.volatility &&
                                   underlying.Equals(dTO.underlying) &&
                                   spot == dTO.spot &&
                                   underlyingType.Equals(dTO.UnderlyingType);
         }

         public override int GetHashCode()
         {
             int hashCode = -594186788;
             hashCode = hashCode * -1521134295 + optionType.GetHashCode();
             hashCode = hashCode * -1521134295 + strike.GetHashCode();
             hashCode = hashCode * -1521134295 + riskFreeRate.GetHashCode();
             hashCode = hashCode * -1521134295 + maturity.GetHashCode();
             hashCode = hashCode * -1521134295 + volatility.GetHashCode();
             hashCode = hashCode * -1521134295 + underlying.GetHashCode();
                     hashCode = hashCode * -1521134295 + spot.GetHashCode();
             hashCode = hashCode * -1521134295 + underlyingType.GetHashCode();
                     return hashCode;
         }

         public override string ToString()
              {
                     return new StringBuilder().Append("Option type ")
                                   .Append(optionType.Value)
                                   .Append(" Strike ")
                                   .Append(strike)
                                   .Append(" Risk free rate ")
                                   .Append(riskFreeRate)
                                   .Append(" Maturity ")
                                   .Append(maturity)
                                   .Append(" Volatility ")
                                   .Append(volatility)
                                   .Append(" Underlying ")
                                   .Append(underlying)
                                   .Append(" Spot ")
                                   .Append(spot)
                                   .Append(" Underlying type ")
                                   .Append(underlyingType.Value).ToString();
              }
        }
```

```
}
```

DAO pattern

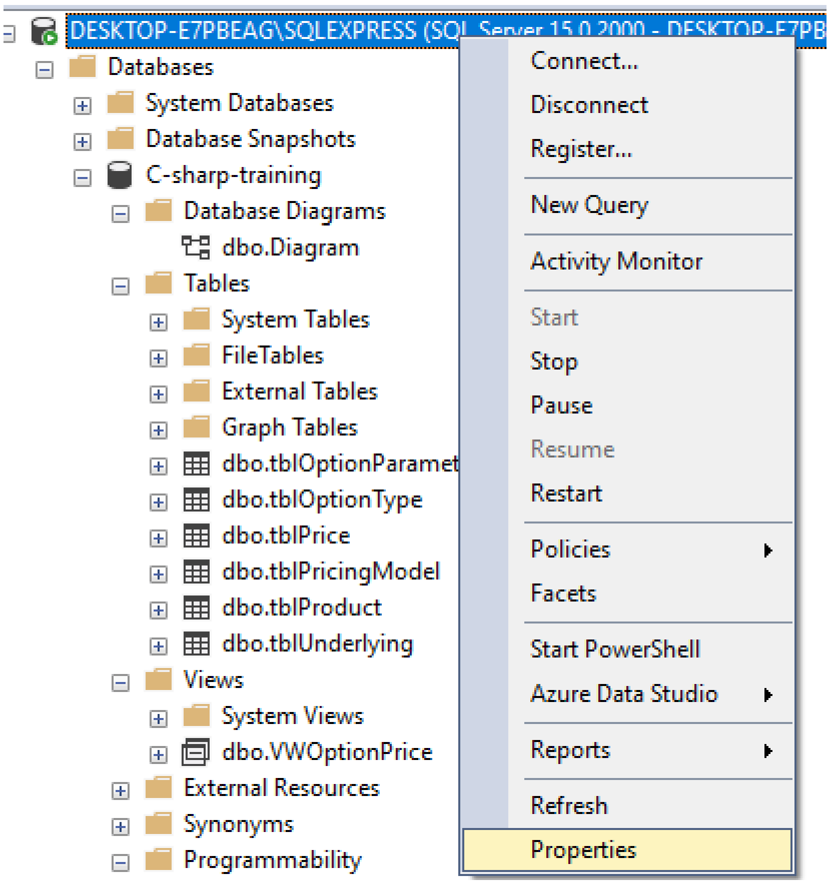


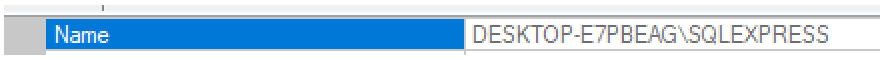
The DAO interface should expose database manipulation operations like inserting prices/options, deleting prices/option, getting prices …

```
public interface IOptionDAO
    {
        void InsertOptionParameters(OptionParametersDTO optionDTO);
        void InsertPrice(PriceDTO price);
        List<OptionParametersDTO> GetAllOptions();
        double? GetPriceByOptionAndPricingModel(OptionParametersDTO optionParameters, 
PricingModelEnum pricingModel);
        void DeleteOption(OptionParametersDTO optionDTO, PricingModelEnum pricingModel);
        void DeletePrice(PriceDTO price);
    }
```

C# DB connection

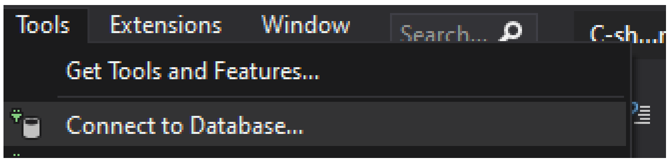
SQL Server:





Name	DESKTOP-E7PBEAG\SQLEXPRESS

Visual Studio



1] In Visual Studio go to **Tools -> Connect to Database**.

2] Under Server Name Select your Database Server Name (Let the list Populate if its taking time).

3] Under Connect to a Database, Select **Select or enter a database name**.

4] Select your Database from Dropdown.

5] After selecting Database try Test Connection.

6] If Test Connection Succeeds, Click Ok.

7] In Visual Studio go to **View -> Server Explorer**.

8] In Server Explorer window, Under Data Connections Select your Database. **Right Click your Database -> Click Properties**.

9] In **Properties** window you will see your **Connection String**.

SqlConnection. The SqlConnection class handles database connections. It initiates a connection to your SQL database. This class is best used in a using resource acquisition statement.

```
public static string CONNECTION_STRING = @"Data Source=DESKTOP-E7PBEAG\SQLEXPRESS;Initial
Catalog=C-sharp-training;Integrated Security=True";


        public void InsertOptionParameters(OptionParametersDTO optionDTO)
        {
            using (SqlConnection connection = new SqlConnection(CONNECTION_STRING))
            {
                connection.Open();
                SqlCommand cmd = new SqlCommand("SPInsertOptionParameters", connection);
                cmd.CommandType = CommandType.StoredProcedure;
                cmd.Parameters.AddWithValue("@optionType", optionDTO.OptionType.Value);
                cmd.Parameters.AddWithValue("@strike", optionDTO.Strike);
                cmd.Parameters.AddWithValue("@riskFreeRate", optionDTO.RiskFreeRate);
                cmd.Parameters.AddWithValue("@maturity",
MaturityToSqlDateTime(optionDTO.Maturity));
                cmd.Parameters.AddWithValue("@volatility", optionDTO.Volatility);
                cmd.Parameters.AddWithValue("@underlying", optionDTO.Underlying);
                cmd.Parameters.AddWithValue("@spot", optionDTO.Spot);
                cmd.Parameters.AddWithValue("@product", optionDTO.UnderlyingType.Value);
                try
                {
                    cmd.ExecuteNonQuery();
                }
                catch (Exception e)
                {
                    logger.Error($"Error while running InsertOptionParameters {e}");
                }
            }
        }
```

The full implementation of the DAO class is available in GitHub.

8- Multithreading

Multitasking is the simultaneous execution of multiple tasks or processes over a certain time interval. Windows operating system is an example of multitasking because it is capable of running more than one process at a time like running Google Chrome, Notepad, VLC player, etc. at the same time. The operating system uses a term known as a *process* to execute all these applications at the same time. A process is a part of an operating system that is responsible for executing an application. Every program that executes on your system is a process and to run the code inside the application a process uses a term known as a *thread*.
A thread is a lightweight process, or in other words, a thread is a unit which executes the code under the program. So every program has logic and a thread is responsible for executing this logic. Every program by default carries one thread to executes the logic of the program and the thread is known as the *Main Thread*, so every program or application is by default single-threaded model. This single-threaded model has a drawback. The single thread runs all the process present in the program in synchronizing manner, means one after another. So, the second process waits until the first process completes its execution, it consumes more time in processing.

Multi-threading is a process that contains multiple threads within a single process. Here each thread performs different activities. For example, we have a class and this call contains two different methods, now using multithreading each method is executed by a separate thread. So the major advantage of multithreading is it works simultaneously, which means multiple tasks execute at the same time. And also maximizing the utilization of the CPU because multithreading works on time-sharing concept mean each thread takes its own time for execution and does not affect the execution of another thread, this time interval is given by the operating system.

Thread:

- Lightweight process with independent execution path, can execute tasks without blocking its parent
- Small set of executable instructions

Implement multithreading:

- Create multiple threads using ThreadStart delegate
- Asynchronous methods like BeginInvoke

Task:

- Class that let you create tasks and run them asynchronously => C#4.0
- Task is an object that represents some work that should be done
- Task can tell you if the work is completed and if operation returns result, it gives results

Why task:

- Can be used whenever you want to run parallel
- Asynchronous implementation easy with 'async' keyword

Why thread:

- App required to perform tasks at the same time

Tasks vs Threads:

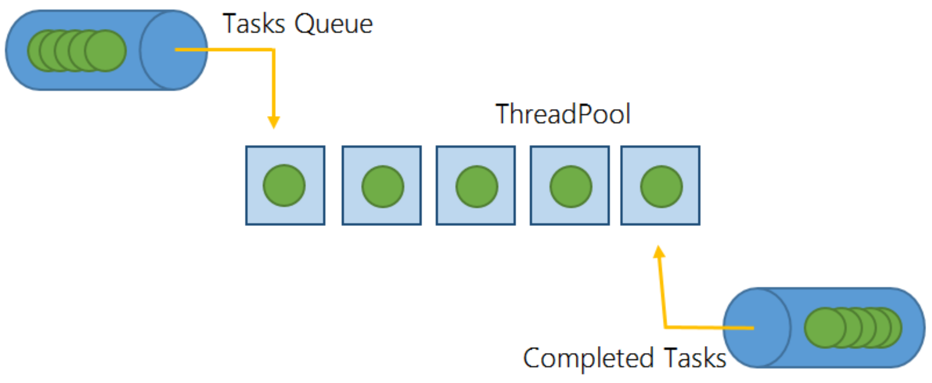
- Task represents asynchronous operations

- Task is part of Task Parallel Library
- Task can return result, while there's no direct mechanism for thread to return result
- Task supports cancellation using cancellation tokens, Thread doesn't
- Thread can have 1 task running, task can have multiple processes at same time
- Task uses Thread pool, thread doesn't
- Task is a higher-level concept than Thread

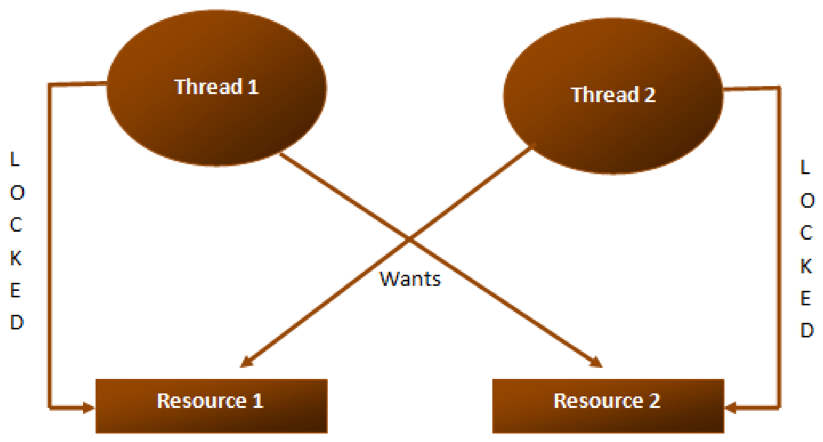
Thread pool:

Thread pool is a collection of threads which can be used to perform no of task in background. Once thread completes its task then it sent to the pool to a queue of waiting threads, where it can be reused. This reusability avoids an application to create more threads and this enables less memory consumption.

Once task is completed in thread pool, .NET framework sends these task completed threads to thread pool rather than sending these threads to garbage collector. Further which can be reused for other new task.



Deadlock



A deadlock is a situation where an application locks up because two or more activities are waiting for each other to finish. This occurs in multithreading software where a shared resource is locked by one thread and another thread is waiting to access it and something occurs so that the thread holding the locked item is waiting for the other thread to execute.

How to avoid that?

Thread safe collection

- Concurrent dictionary
- For lists => concurrent bag

Thread synchronization

1- **Lock**
Exclusive locking ensures only 1 thread can enter a specific section of code at the same time
Both lock and mutex are exclusive lockings constructs

```
public class ThreadSync
    {
        int a = 4, b = 2;

        public void Divide()
        {
            if (b!=0)
            {
                Console.WriteLine(a / b);
            }
            b = 0;
        }
    }
```

If 2 threads call Divide(), 1 could throw divide by 0 exception
⇨ 1 thread should lock this section, all other threads are in blocked state

⇨ Thread safe version :

```
public static readonly object _lock = new object();
        int a = 4, b = 2;

        public void Divide()
        {
            lock (_lock)
            {
                if (b != 0)
                    Console.WriteLine(a / b);
                b = 0;
            }
        }
```

2- **Monitor**

Similar to Lock

It prevents blocks of code from simultaneous execution by more than 1 thread

```
public class ThreadSync
    {

        public static readonly object _lock = new object();
        int a = 4, b = 2;
        public void Divide()
        {
            Monitor.Enter(_lock);
            try
            {
                if (b != 0)
                    Console.WriteLine(a / b);
                b = 0;
            }
            finally
            {
                Monitor.Exit(_lock);
            }
        }
    }
```

`Monitor.TryEnter` allows to specify for how much time we try to get the lock

3- **Mutex**

Like Lock but can work across multiple processes, computer-wide as well as application-wide

It is slower than Lock

Methods WaitOne, ReleaseMutex, closing or disposing mutex auto releases the lock

```
private Mutex _mut = new Mutex();
        public void DoSomething()
        {
            try
            {
                _mut.WaitOne();
                //perform operation
            }
            finally
            {
                _mut.ReleaseMutex();
            }
        }
```

Sync between process => use Mutex named that's used in the whole Operating System

4- **Semaphore**

It allows a specified number of threads to access a resource
All other threads are blocked until a thread releases the lock
Semaphore with capacity 1 = lock or monitor with exception: semaphore has **no owner**
Any thread can release the lock
Lock and Monitor : only the tread that obtained the lock can release it

```
private Semaphore _sem = new Semaphore(0, 3);
        public void DoSomething()
        {
            try
            {
                _sem.WaitOne();
                //perform action
            }
            finally
            {
                _sem.Release();
            }
        }
```

Class Parallel

Parallel.For / Parallel.ForEach: static methods that allows to execute a loop in parallel

Async & await

Async: means that a method contains asynchronous code

Await:

- mark the beginning of asynchronous code

- There can be several in a method
- If the method is asynchronous and if it returns a type **Result** then the method return signature is **Task<Result>**

Cancel tasks

Class **CancellationToken** and **CancellationTokenSource**

```
Var cancellationTokenSource = new CancellationTokenSource();
Var cancellationToken = cancellationTokenSource.Token;
// pass cancellationToken as argument to the asynchronous method
```

Multithreading example:

Producer – consumer

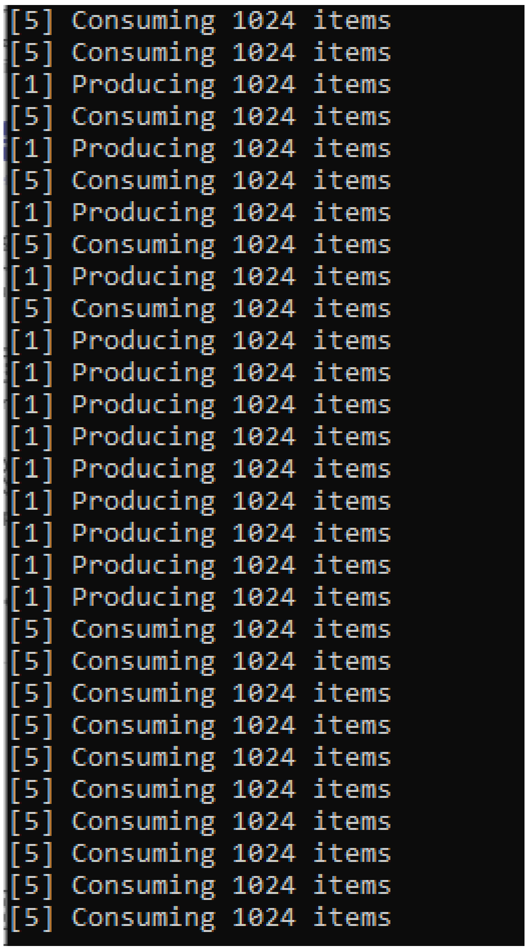
```
namespace ProducerConsumer
{
    public class Producer
    {
        public void Produce(ITargetBlock<byte[]> table)
        {
            Console.WriteLine("Starting producer...");
            Random rnd = new Random();
            for (int i = 0; i < 1000; i++)
            {
                byte[] buffer = new byte[1024];
                rnd.NextBytes(buffer);
                Console.WriteLine($"[{Thread.CurrentThread.ManagedThreadId}] Producing 
{buffer.Length} items");
                table.Post(buffer);
            }
            table.Complete();
        }
    }
}
```

```
namespace ProducerConsumer
{
    public class Consumer
    {
        /*
```

```
         * async modifier to specify that a method, lambda expression, or anonymous method
is asynchronous
         * An async method runs synchronously until it reaches its first await expression,
         * at which point the method is suspended until the awaited task is complete.
         * In the meantime, control returns to the caller of the method : do not block the
caller's thread
         * Return types :
         *      Task : if no meaningful value is returned when the method is completed.
That is, a call to the method returns a Task, but when the Task is completed, any await
expression that's awaiting the Task evaluates to void
         *      Task<TResult> : if the return statement of the method specifies an operand
of type TResult
         *      void : async void methods are generally discouraged for code other than
event handlers because callers cannot await those methods and must implement a different
mechanism to report successful completion or error conditions.
        */
        public async Task<int> ConsumeAsync(ISourceBlock<byte[]> table)
        {
            Console.WriteLine("Starting consumer...");
            int byteProcessed = 0;
            /*
             * If the method that the async keyword modifies doesn't contain an await
expression or statement, the method executes synchronously
             * The await operator suspends evaluation of the enclosing async method until
the asynchronous operation represented by its operand completes. When the asynchronous
operation completes, the await operator returns the result of the operation, if any.
             * When the await operator is applied to the operand that represents an already
completed operation, it returns the result of the operation immediately without suspension
of the enclosing method.
             * The await operator doesn't block the thread that evaluates the async method.
             * When the await operator suspends the enclosing async method, the control
returns to the caller of the method.
             */
            while (await table.OutputAvailableAsync())
            {
                byte[] data = await table.ReceiveAsync();
                Console.WriteLine($"[{Thread.CurrentThread.ManagedThreadId}] Consuming
{data.Length} items");
                byteProcessed += data.Length;
            }
            return byteProcessed;
        }
    }
}
```

```
namespace ProducerConsumer
{
    public class Program
    {
        public static void Main(string[] args)
        {

            var buffer = new BufferBlock<byte[]>(); //TPL Dataflow BufferBlock<T> type can
be thought of as an unbounded buffer for data that enables
                                                    //synchronous and asynchronous
producer/consumer scenarios
            Consumer consumer = new Consumer();
            Producer producer = new Producer();
            var consumerTask = consumer.ConsumeAsync(buffer);
            producer.Produce(buffer);
            int byteProcessed = consumerTask.Result;
            Console.WriteLine($"Byte processed : {byteProcessed}");
        }
    }
}
```

Console :



```
[5] Consuming 1024 items
[5] Consuming 1024 items
[1] Producing 1024 items
[5] Consuming 1024 items
[1] Producing 1024 items
[5] Consuming 1024 items
[1] Producing 1024 items
[5] Consuming 1024 items
[1] Producing 1024 items
[5] Consuming 1024 items
[1] Producing 1024 items
[1] Producing 1024 items
[1] Producing 1024 items
[1] Producing 1024 items
[1] Producing 1024 items
[1] Producing 1024 items
[1] Producing 1024 items
[1] Producing 1024 items
[1] Producing 1024 items
[5] Consuming 1024 items
[5] Consuming 1024 items
[5] Consuming 1024 items
[5] Consuming 1024 items
[5] Consuming 1024 items
[5] Consuming 1024 items
[5] Consuming 1024 items
[5] Consuming 1024 items
[5] Consuming 1024 items
[5] Consuming 1024 items
```

DIGISTRAT
CONSULTING

8-Option pricing

Create a new .NET FRAMEWORK Class Library

First define the common objects

They are :

- Option type enum
- Interfaces :
    - For greeks
    - For the pricing

Option type enum

```
public enum ContractEnum
       {
              EUROPEANCALL = 1,
              EUROPEANPUT,
              AMERICANCALL,
              AMERICANPUT

}
```

Interface for the Greeks

```
public interface IGreeks
    {
        double delta();
        double gamma();
        double theta();
        double vega();
        double rho();
    }
```

Interface for the pricing

```
public interface IPricer
    {
        double? premium();
    }
```

In this document, I will only show how to implement Black Scholes pricing model.

The binomial tree and Monte carlo pricing are also implemented and available in the GitHub

Black scholes formulas are :

The Black-Scholes Option Pricing Formula

$$c = SN(d_1) - Xe^{-rT}N(d_2)$$

$$p = Xe^{-rT}N(-d_2) - SN(-d_1).$$

$$d_1 = \frac{\ln(S/X) + (r+\sigma^2/2)T}{\sigma\sqrt{T}}$$

$$d_2 = \frac{\ln(S/X) + (r-\sigma^2/2)T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T}$$

S = Stock price.
X = Strike price of option.
r = Risk-free interest rate.
T = Time to expiration in years.
σ = Volatility of the relative price change of the underlying stock price.
$N(x)$ = The cumulative normal distribution function.

```
public class BlackScholesPricer : IPricer
    {
        private static readonly ILogger logger = 
LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);


        protected double S;
        protected double K;
        protected double r;
        protected double q;
        protected double sigma;
        protected double T;
        protected ContractEnum contractType;


        public BlackScholesPricer(ContractEnum contractType, double S, double K, double T, 
double sigma, double r, double q)
        {
            this.contractType = contractType;
            this.S = S;
            this.K = K;
            this.T = T;
            this.sigma = sigma;
            this.r = r;
            this.q = q;
        }

        public double s { get { return S; } set { S = value; } }
        public double k { get => K; set => K = value; }
        public double R { get => r; set => r = value; }
        public double Q { get => q; set => q = value; }
        public double Sigma { get { return sigma; } set { sigma = value; } }
        public double t { get { return T; } set { T = value; } }
        public ContractEnum ContractType { get { return contractType; } set { contractType
= value; } }



        public double? premium()
        {
            return premiumForSpecificSpot(S);
        }

        private double? premiumForSpecificSpot(double spot)
        {
            logger.Debug($"Computing Black Scholes price for option type : {contractType},
spot : {S}, " +
                $"strike : {K}, risk free rate : {R}, dividend rate : {Q}, Volatility :
{Sigma}, time to maturity : {T}");
            if (!areOptionParamValid(spot, K, T, sigma, r, q))
            {
                logger.Error("Option parameters not valid");
                return null;
            }
            double d1Param = d1(spot, K, T, sigma, r, q);
            double d2Param = d2(d1Param, sigma, T);

            switch (contractType)
            {
                case ContractEnum.EUROPEANCALL:
                    return spot * Math.Exp(-q * T) * Normal.CDF(0, 1, d1Param) - K *
Math.Exp(-r * T) * Normal.CDF(0, 1, d2Param);

                case ContractEnum.EUROPEANPUT:
                    return K * Math.Exp(-r * T) * Normal.CDF(0, 1, -d2Param) - spot *
Math.Exp(-q * T) * Normal.CDF(0, 1, -d1Param);
```

```
                default:
                    throw new NotSupportedException(" Option Type Error 1 " + contractType
+ "Type does not exist!");
            }
        }
        public static double d1(double S, double K, double T, double sigma, double r,
double q)
        {
            return (Math.Log(S / K) + (r - q + Math.Pow(sigma, 2) / 2) * T) / (sigma *
Math.Sqrt(T));
        }

        public static double d2(double d1, double sigma, double T)
        {
            return d1 - sigma * Math.Sqrt(T);
        }

        public static double timeToMaturity(DateTimeOffset contractMaturity, DateTimeOffset
time)
        {
            return (contractMaturity - time).TotalDays / 365.0;
        }

        private static bool areOptionParamValid(double S, double K, double T, double sigma,
double r, double q)
        {
            if (T < 0 || S < 0 || K < 0 || sigma < 0)
            {
                return false;
            }
            if (r < 0)
            {
                logger.Warn("Continuously compounded risk-free interest rate negative");
                return false;
            }

            else if (q < 0)
            {
                logger.Warn("Continuously compounded dividend yield negative");
                return false;
            }
            return true;
        }
        public void premiumChart()
        {
            DataTable dt = new DataTable();
            dt.Columns.Add("Spot_Value", typeof(double));
            dt.Columns.Add("PV_Value", typeof(double));

            double increment = 0.1 * K;
            logger.Info("Spot | PV");
            for (double spot = increment; spot < 2 * K; spot += increment)
            {
                double? pv = premiumForSpecificSpot(spot);
                dt.Rows.Add(spot, pv);
                logger.Info($"{spot} | {pv}");
            }

        }
    }
```

The Greeks

```
public class GreeksBS : BlackScholesPricer, IGreeks
    {
        private static readonly ILogger logger = 
LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);
        public GreeksBS(ContractEnum contractType, double S, double K, double T, double 
sigma, double r, double q) :
            base(contractType, S, K, T, sigma, r, q)
        {


        }
        public double delta()
        {
            logger.Debug("Computing Black Scholes Delta");
            switch (contractType)
            {
                case ContractEnum.EUROPEANCALL:
                    return Math.Exp(-q * T) * Normal.CDF(0, 1, d1(S, K, T, sigma, r, q));

                case ContractEnum.EUROPEANPUT:
                    return -Math.Exp(-q * T) * Normal.CDF(0, 1, -d1(S, K, T, sigma, r, q));
                default:
                    throw new NotSupportedException(" Option Type Error " + contractType + 
" no delta computation available for this contract kind");
            }
        }

        public double gamma()
        {
            logger.Debug("Computing Black Scholes Gamma");
            switch (contractType)
            {
                case ContractEnum.EUROPEANCALL:
                case ContractEnum.EUROPEANPUT:
                    return (Math.Exp(-q * T) * Math.Exp(-Math.Pow(d1(S, K, T, sigma, r, q), 
2) / 2)) / (Math.Sqrt(2 * Math.PI) * S * sigma * Math.Sqrt(T));

                default:
                    throw new NotSupportedException(" Option Type Error " + contractType + 
" no gamma computation available for this contract kind");
            }
        }

        public double rho()
        {
            logger.Debug("Computing Black Scholes Rho");
            switch (contractType)
            {
                case ContractEnum.EUROPEANCALL:
                    return K * T * Math.Exp(-r * T) * Normal.CDF(0, 1, d2(d1(S, K, T, 
sigma, r, q), Sigma, T));

                case ContractEnum.EUROPEANPUT:
                    return -T * K * Math.Exp(-r * T) * Normal.CDF(0, 1, -d2(d1(S, K, T, 
sigma, r, q), Sigma, T));

                default:
                    throw new NotSupportedException(" Option Type Error " + contractType + 
" no rho computation available for this contract kind");
            }
        }

        public double theta()
        {
            logger.Debug("Computing Black Scholes Theta");
            double d1Param = d1(S, K, T, sigma, r, q);
            double d2Param = d2(d1(S, K, T, sigma, r, q), Sigma, T);
            switch (contractType)
            {
                case ContractEnum.EUROPEANCALL:
                    return (1f / T) * (-(S * sigma * Math.Exp(-q * T) * Math.Exp(-
Math.Pow(d1Param, 2) / 2) / (2 * Math.Sqrt(T) * Math.Sqrt(2 * Math.PI))) -
                        r * K * Math.Exp(-r * T) * Normal.CDF(0, 1, d2Param) + q * S *
Math.Exp(-q * T) * Normal.CDF(0, 1, d1Param));


                case ContractEnum.EUROPEANPUT:
                    return (1f / T) * (-(S * sigma * Math.Exp(-q * T) * Math.Exp(-
Math.Pow(d1Param, 2) / 2) / (2 * Math.Sqrt(T) * Math.Sqrt(2 * Math.PI))) +
                        r * K * Math.Exp(-r * T) * Normal.CDF(0, 1, -d2Param) + q * S *
Math.Exp(-q * T) * Normal.CDF(0, 1, -d1Param));
                default:
                    throw new NotSupportedException(" Option Type Error " + contractType +
" no theta computation available for this contract kind");
            }
        }


        public double vega()
        {
            logger.Debug("Computing Black Scholes Vega");
            switch (contractType)
            {
                case ContractEnum.EUROPEANCALL:
                case ContractEnum.EUROPEANPUT:
                    return S * Math.Exp(-q * T) * Math.Sqrt(T) * Math.Exp(-Math.Pow(d1(S,
K, T, sigma, r, q), 2) / 2) / Math.Sqrt(2 * Math.PI);


                default:
                    throw new NotSupportedException(" Option Type Error " + contractType +
" no vega computation available for this contract kind");
            }
        }
    }
```

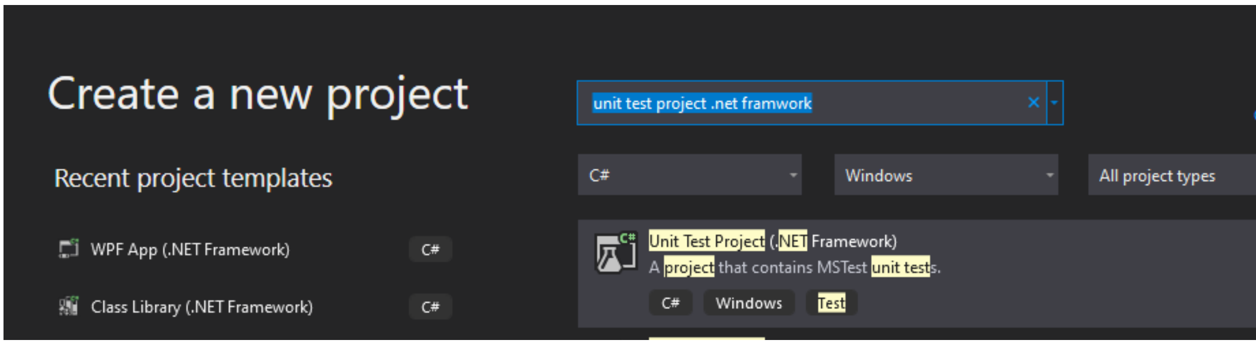
9-DAO layer unit tests, Domain layer, SOLID principles

Unit tests

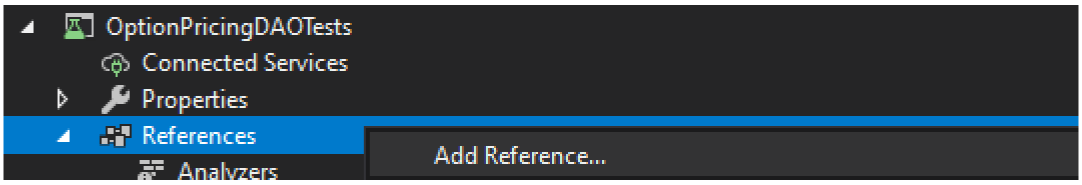
In the software development process Unit Tests basically test individual parts (also called as Unit) of code (mostly methods) and make it work as expected by programmer. A Unit Test is a code written by any programmer which test small pieces of functionality of big programs.

Generally the tests cases are written in the form of functions that will evaluate and determine whether a returned value after performing Unit Test is equals to the value you were expecting when you wrote the function. The main objective in unit testing is to isolate a unit part of code and validate its to correctness and reliable.

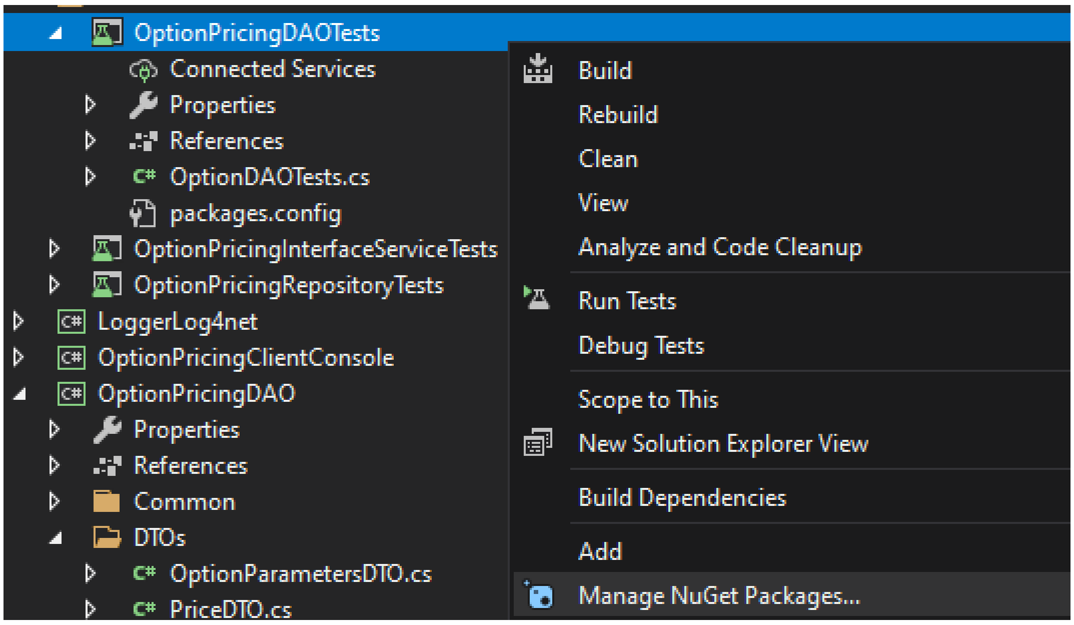
1- Create new solution


Create a new project
unit test project .net framwork
Recent project templates
WPF App (.NET Framework) C#
Class Library (.NET Framework) C#
C#
Windows
All project types
Unit Test Project (.NET Framework)
A project that contains MSTest unit tests.
C# Windows Test

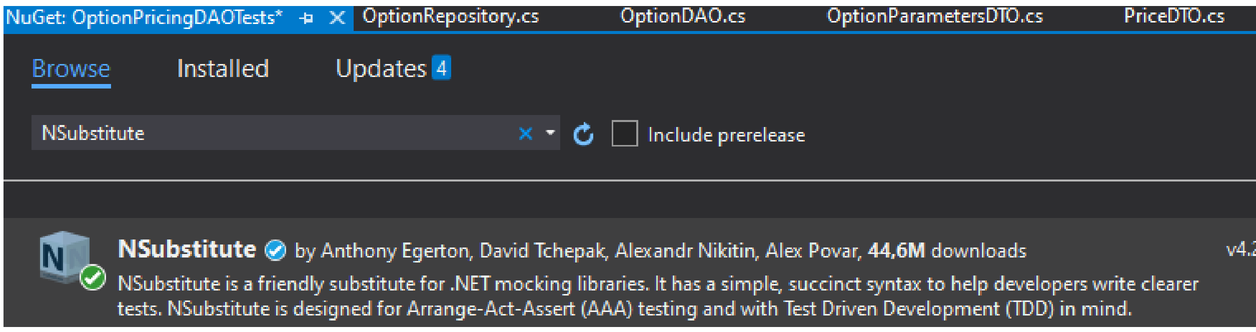
a. Add reference to OptionPricingDAO



b. Manage NuGet packages



2- Add NSubstitute



NSubstitute by Anthony Egerton, David Tchepak, Alexandr Nikitin, Alex Povar, **44,6M** downloads v4.2

NSubstitute is a friendly substitute for .NET mocking libraries. It has a simple, succinct syntax to help developers write clearer tests. NSubstitute is designed for Arrange-Act-Assert (AAA) testing and with Test Driven Development (TDD) in mind.

NSubstitute is designed for Arrange-Act-Assert (**AAA**) testing, so you just need to arrange how it should work, then assert it received the calls you expected once you're done. Because you've got more important code to write than whether you need a mock or a stub.

You will get a screen as shown below

```
namespace OptionPricingDAOTests
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
       }
    }
}
```

TEST CLASS REQUIREMENTS

The minimum requirements for a test class while writing Unit Test case is given below:

- If you are using Unit Test to write test case then the [TestClass] attribute is highly required in the Microsoft unit testing framework for any class that contains unit test methods that you would like to run in Visual Studio Test Explorer.
- Each and every test method that you want to run must having the [TestMethod]attribute above it.

TEST METHOD REQUIREMENTS

A test method must meet the given requirements:

- The method must be defined with the [TestMethod] attribute just above method name.
- The method must have return type void.
- The method cannot have any parameters.

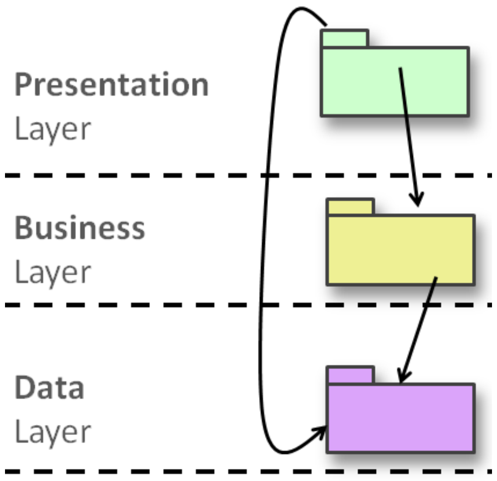
Writing first Unit Test method

Arrange-Act-Assert rule

```
[TestMethod]
public void InsertAndGetOptionParameters()
        {
            // Arrange
            OptionParametersDTO optionParametersDTO = DummyOption("DE_DAX_TEST");
            int initialOptionsNbr = optionDao.GetAllOptions().Count;
            // Act
            optionDao.InsertOptionParameters(optionParametersDTO);
            List<OptionParametersDTO> result = optionDao.GetAllOptions();
            // Assert
            Assert.AreEqual(result.Count, initialOptionsNbr+1);
            OptionParametersDTO option = result.Last(); // GetAllOptions keeps insertion 
order
            Assert.AreEqual(option.Spot, optionParametersDTO.Spot);
            Assert.AreEqual(option.Strike, optionParametersDTO.Strike);
            Assert.AreEqual(option.Underlying, optionParametersDTO.Underlying);
            Assert.IsNotNull(option.UnderlyingType);
            Assert.AreEqual(option.UnderlyingType.Value, 
optionParametersDTO.UnderlyingType.Value);
            Assert.AreEqual(option.Maturity, optionParametersDTO.Maturity);
            Assert.IsNotNull(option.OptionType);
            Assert.AreEqual(option.OptionType.Value, optionParametersDTO.OptionType.Value);
            Assert.AreEqual(option.RiskFreeRate, optionParametersDTO.RiskFreeRate);
            Assert.AreEqual(option.Volatility, optionParametersDTO.Volatility);
            CleanOption(option, null); // no price inserted => no pricing model
        }
```

The remaining tests are available in GitHub

Business Layer



The Data Access Layer (DAL) created in the first tutorial cleanly separates the data access logic from the presentation logic. However, while the DAL cleanly separates the data access details from the presentation layer, it does not enforce any business rules that may apply.

In this tutorial we'll see how to centralize these business rules into a Business Logic Layer (BLL) that serves as an intermediary for data exchange between the presentation layer and the DAL. The BLL should be implemented as a separate Class Library project

Add a new solution named **OptionPricingDomain**, target : .NET FRAMEWOR, output type : Class Library

In this library, we’ll be implementing all the business classes needed :

- ContractEnum
- Maturity
- Option
- Price
- PricingModelEnum

- Underlying
- UnderlyingTypeEnum

We will be adding business rules, for instance option maturities cannot be in the past, option type cannot be unknown, option’s underlying cannot be null …

```
public class Maturity
    {
        public int Year { get; private set; }
        public int Month { get; private set; }
        public int Day { get; private set; }

        public Maturity(int year, int month, int day)
        {
            IsMaturityValid(year, month, day);
            this.Year = year;
            this.Month = month;
            this.Day = day;
        }

        private void IsMaturityValid(int year, int month, int day)
        {
            if (year < DateTime.Today.Year)
            {
                throw new ArgumentException("Year cannot be in the past");
            }
            var d = new DateTime(year, month, day);
            if (d.CompareTo(DateTime.Now) <= 0)
            {
                throw new ArgumentException("Maturity cannot be in the past");
            }
        }

        public bool Equals(Maturity that)
        {
            if (that == null)
                return false;

            if (ReferenceEquals(that, this))
                return false;
            if (that.GetType() != this.GetType())
                return false;

            return this.Year.Equals(that.Year)
                && this.Month == that.Month
                && this.Day == that.Day;
        }
        public override int GetHashCode()
        {
            return this.Year.GetHashCode()
                 ^ this.Month.GetHashCode()
                 ^ this.Day.GetHashCode();
        }

        public DateTime ToDateTime()
        {
            return new DateTime(this.Year, this.Month, this.Day);
        }
    }
```

The remaining classes implementation is available in GitHub

SOLID principles

S — SINGLE RESPONSIBILITY

A CLASS SHOULD HAVE A SINGLE RESPONSIBILITY

O — OPEN-CLOSED

CLASSES SHOULD BE OPEN FOR EXTENSION BUT CLOSED FOR MODIFICATION

L — LISKOV SUBSTITUTION

IF S IS A SUBTYPE OF T, THEN OBJECTS OF TYPE T IN A PROGRAM MAY BE REPLACED WITH OBJECTS OF TYPE S WITHOUT ALTERING ANY OF THE DESIRABLE PROPERTIES OF THAT PROGRAM.

I — INTERFACE SEGREGATION

CLIENTS SHOULD NOT BE FORCED TO DEPEND ON METHODS THAT THEY DO NOT USE.

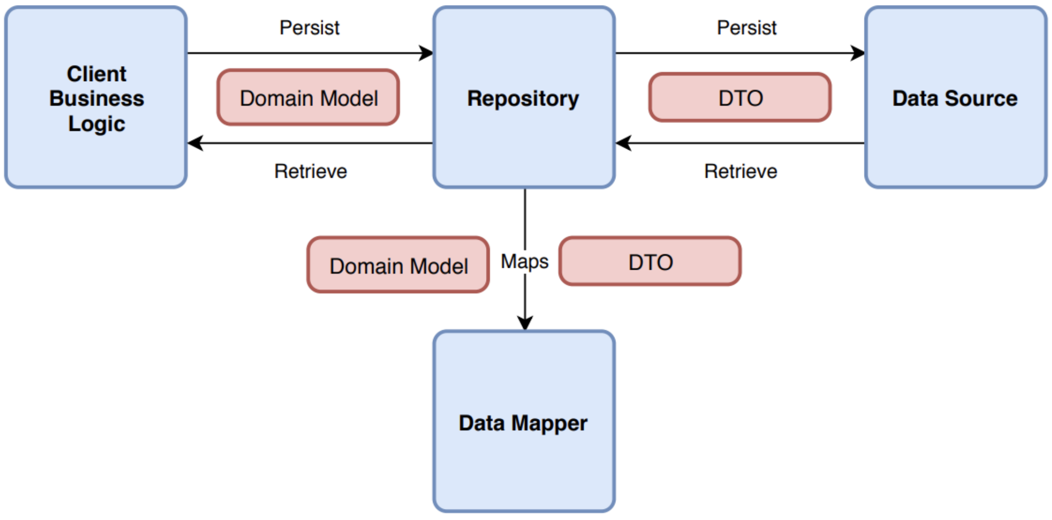
D — DEPENDENCY INVERSION

- HIGH-LEVEL MODULES SHOULD NOT DEPEND ON LOW-LEVEL MODULES. BOTH SHOULD DEPEND ON THE ABSTRACTION.

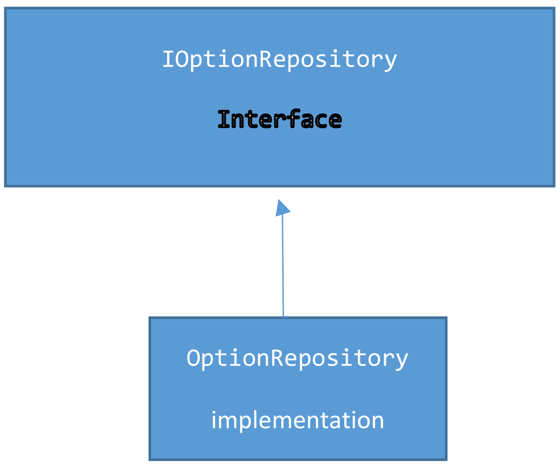
- ABSTRACTIONS SHOULD NOT DEPEND ON DETAILS. DETAILS SHOULD DEPEND ON ABSTRACTIONS.

10-Repository layer

Repositories are classes or components that encapsulate the logic required to access data sources. They centralize common data access functionality, providing better maintainability and decoupling the infrastructure or technology used to access databases from the domain model layer. If you use an Object-Relational Mapper (ORM) like Entity Framework, the code that must be implemented is simplified, thanks to LINQ and strong typing. This lets you focus on the data persistence logic rather than on data access plumbing.



Pattern



The repository interface should expose functions to the business layer to manipulate business objects (Option, Price)

It should convert business objects to DTO and use the DAO layer.

```
public interface IOptionRepository
    {
        void InsertOptionParameters(Option option);
        void InsertPrice(Price price);
        List<Option> GetAllOptions();
        double? GetPriceByOptionAndPricingModel(Option option, PricingModelEnum 
pricingModel);
        void DeleteOption(Option optionDTO, PricingModelEnum pricingModel);
        void DeletePrice(Price price);
    }
```

Data mapper

For Domain ⇔ DTO object conversion, a class has been implemented with static methods that provides the conversions needed

```
public class OptionUtils
    {
        public static Option GetOptionFromDTO(OptionParametersDTO optionParameters)
        {
            ContractEnum contractType = (ContractEnum)Enum.Parse(typeof(ContractEnum), 
optionParameters.OptionType.Value);
            DateTime optionMatu = optionParameters.Maturity;
            Maturity maturity = new Maturity(optionMatu.Year, optionMatu.Month, 
optionMatu.Day);
            UnderlyingTypeEnum udlType = 
(UnderlyingTypeEnum)Enum.Parse(typeof(UnderlyingTypeEnum), 
optionParameters.UnderlyingType.Value.ToUpper());
            Underlying udl = new Underlying(optionParameters.Underlying, 
optionParameters.Spot, udlType, optionParameters.Volatility);
            return new Option(contractType, maturity, optionParameters.Strike, 
optionParameters.RiskFreeRate, udl);
        }
…
```

Repository Unit Tests

Since we already unit tested the DAO layer. We will be using a substitute for the DAO class.

The idea is to test that the DAO class is receiving the correct calls, correct number of calls and the arguments are well received.

Now that we have our **Repository** up and running, we shall implement the Domain services

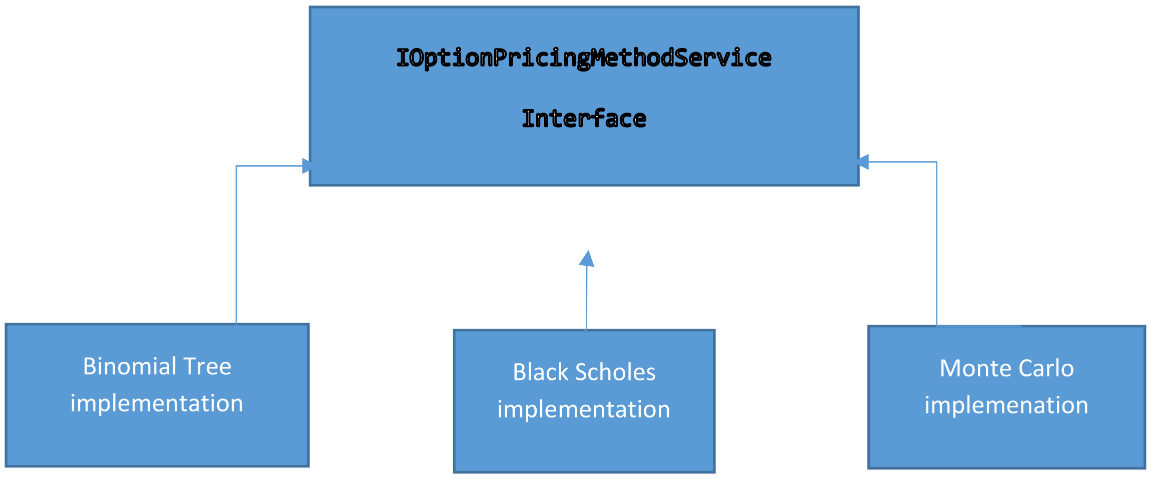
```
public class OptionRepositoryTest
    {
        private readonly IOptionRepository optionRepo;
        private readonly IOptionDAO optionDAO;

        public OptionRepositoryTest()
        {
            this.optionDAO = Substitute.For<IOptionDAO>();
            this.optionRepo = new OptionRepository(optionDAO);
        }

        [TestMethod]
        public void InsertAndGetOption()
        {
            // Arrange
            Option option = DummyOption("US_SPX_TEST");
            OptionPricingDAO.Common.ContractEnum contractType =
OptionPricingDAO.Common.ContractEnum.FromString(option.OptionType.ToString());
            Underlying udl = option.UnderlyingObj;
            OptionPricingDAO.Common.UnderlyingTypeEnum udlType =
OptionPricingDAO.Common.UnderlyingTypeEnum.FromString(option.UnderlyingObj.UnderlyingType.T
oString());
            OptionParametersDTO expectedOptionDTO =  new OptionParametersDTO(contractType,
option.Strike, option.RiskFreeRate, option.Maturity.ToDateTime(), udl.Volatility,
                                                udl.UnderlyingName, udl.Spot, udlType);
            //Act
            optionRepo.InsertOptionParameters(option);
            //Assert
            optionDAO.Received(1).InsertOptionParameters(expectedOptionDTO); // Check a
call was received a specific number of times with a specific arg
        }



…
}
```

Domain services

Now that we have our pricing library and our domain objects, we shall implement the domain services (business logic)

We need to expose services that allow to price options depending on the pricing model selected (and available)

Pattern


IOptionPricingMethodService
Interface
Binomial Tree implementation
Black Scholes implementation
Monte Carlo implemenation

Interface:

It exposes the pricing of a Domain Option.

```
namespace OptionPricingDomainService
{
    public interface IOptionPricingMethodService
    {
        double Price(Option option);
    }
}
```

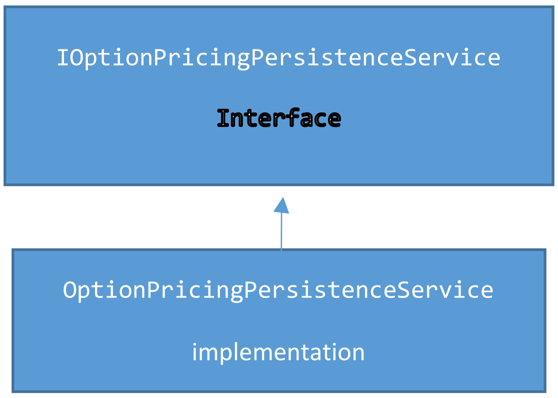
Black Scholes pricing

```
namespace OptionPricingDomainService
{
    public class OptionPricingBlackScholesService : IOptionPricingMethodService
    {
        private static readonly ILogger logger = 
LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);
```

```
        public double Price(Option option)
        {
            logger.Info("Starting Black Scholes pricing");
            Stopwatch stopwatch = new Stopwatch();
            stopwatch.Start();
            double timeToMatu = OptionUtils.TimeToMaturity(option);
            BlackScholesPricer bs = new 
BlackScholesPricer(OptionUtils.OptionTypeToPricingEnum(option), option.UnderlyingObj.Spot, 
option.Strike,
                                    timeToMatu, option.UnderlyingObj.Volatility, 
option.RiskFreeRate, 0d);
            stopwatch.Stop();
            logger.Info($"Black Scholes pricing done in {stopwatch.Elapsed}");
            return bs.premium().Value;
        }
    }
}
```

The domain services should also expose a service to persist data in the database from Domain objects.

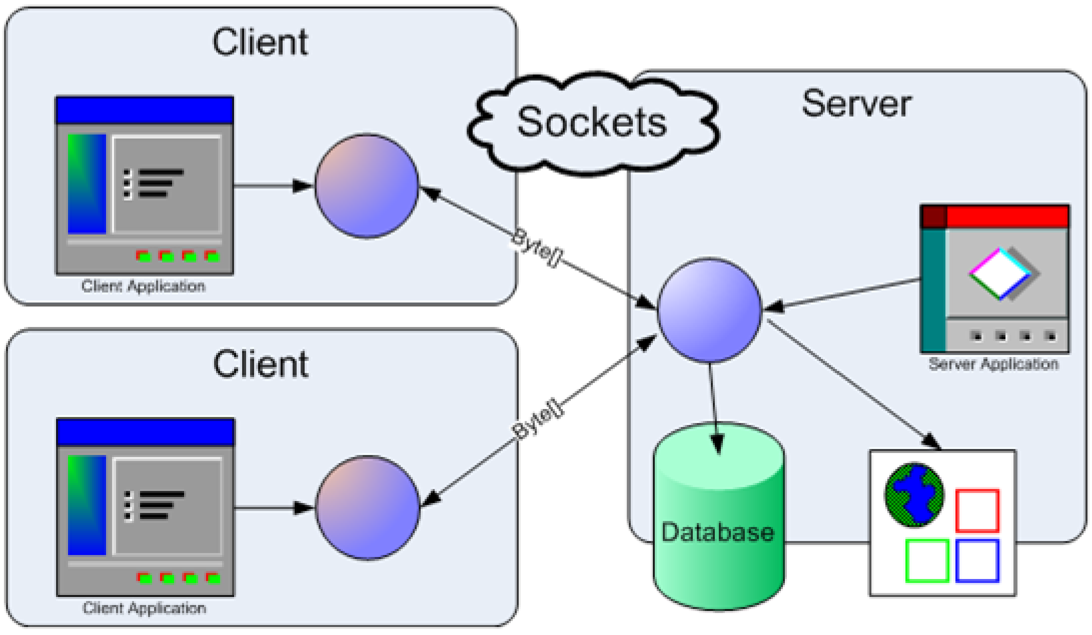
Pattern


IOptionPricingPersistenceService
Interface
OptionPricingPersistenceService
implementation

```
public interface IOptionPricingPersistenceService
    {
        void InsertOption(Option option);
        void InsertPrice(Price price);
    }
```

`OptionPricingPersistenceService` will use the OptionRepository

11- Service creation, NetMq



New .NET FRAMEWORK console application: **OptionPricingInfrastructure**

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket.

1- Send data over the network

In order to send data over the network, they should be serialized.

It can be XML, json …

Here, we use Json serialization.

First, we create a generic class to serialize and deserialize data (business objects)

As usual, we create an interface and then implementing it

```
public interface IOptionPricingJsonSerializer<T>
```

```
    {
        string Serialize(T obj);
        T Deserialize(string obj);
    }
```

Add package `JsonConvert` to the solution though nuget tool.

Here the implementation is better to manipulate generic type because we are going to use it to serialize/deserialize several types to be sent over the network (Option, Price)

```
public class OptionPricingJsonSerializer<T> : IOptionPricingJsonSerializer<T>
    {
        public string Serialize(T obj)
        {
            return JsonConvert.SerializeObject(obj);
        }

        public T Deserialize(string obj)
        {
            return JsonConvert.DeserializeObject<T>(obj);
        }
    }
```

2- Dependency injection pattern

Dependency Injection (DI) is a software design pattern that allows us to develop loosely coupled code. DI is a great way to reduce tight coupling between software components. DI also enables us to better manage future changes and other complexity in our software. The purpose of DI is to make code maintainable.

The Dependency Injection pattern uses a builder object to initialize objects and provide the required dependencies to the object means it allows you to "inject" a dependency from outside the class.

Advantages of Dependency Injection

Reduces class coupling

Increases code reusability

Improves code maintainability

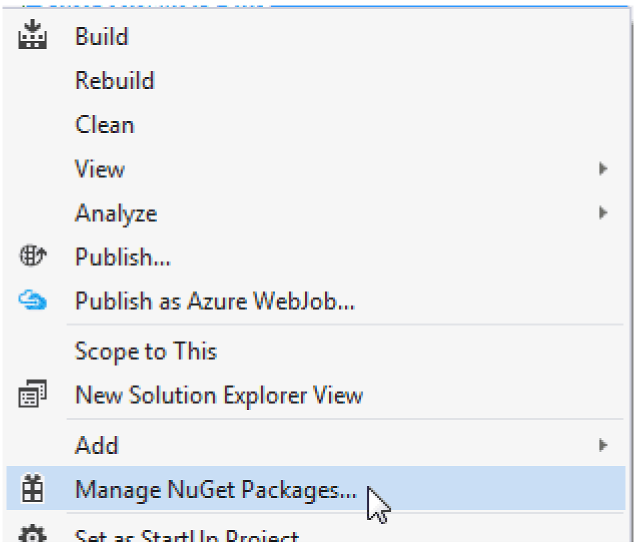
Make unit testing possible

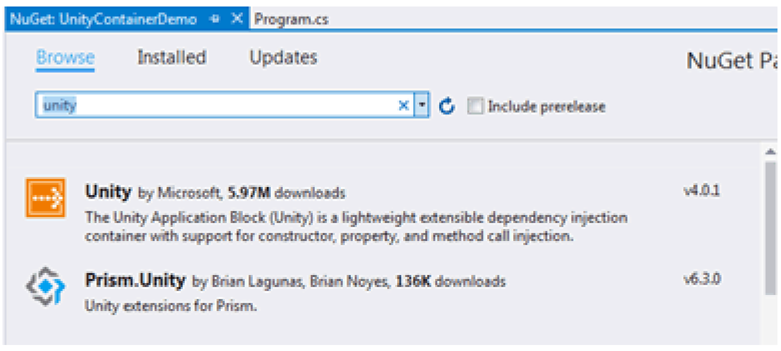
DI Container

The recommended way to implement DI is, you should use DI containers. If you compose an application without a DI CONTAINER, it is like a POOR MAN'S DI

Unity Container

Unity container is an open source IoC container for .NET applications supported by Microsoft. It is a lightweight and extensible IoC container.





Unity by Microsoft, 5.97M downloads v4.0.1
The Unity Application Block (Unity) is a lightweight extensible dependency injection container with support for constructor, property, and method call injection.

Prism.Unity by Brian Lagunas, Brian Noyes, 136K downloads v6.3.0
Unity extensions for Prism.

Every container must provide a way to register and resolve dependencies. Unity container provides the RegisterType() and Resolve() methods for this.

Register

Before Unity resolves the dependencies, we need to register the type-mapping with the container, so that it can create the correct object for the given type. Use the RegisterType() method to register a type mapping. Basically, it configures which class to instantiate for which interface or base class.

Resolve

Unity creates an object of the specified class and automatically injects the dependencies using the resolve() method.

Register Named Type

You can register a type-mapping with a name which you can use with the Resolve() method.

Register Instance

Unity container allows us to register an existing instance using the RegisterInstance() method. It will not create a new instance for the registered type and we will use the same instance every time.

As usual, we begin by creating an interface :

```
public interface IDependencyInjectionManager
    {
        void RegisterType<TInterface, TImplem>() where TImplem : TInterface;
        void RegisterInstance<T>(T instance);
        void RegisterTypeWithKey<TInterface, TImplem>(string key) where TImplem :
TInterface;
        TInterface ResolveWithKey<TInterface>(string key);
        TInterface Resolve<TInterface>();
    }
```

Then implementing it

```
public class DependencyInjectionManager : IDependencyInjectionManager
    {
        private readonly IUnityContainer container;

        public DependencyInjectionManager()
        {
            this.container = new UnityContainer();
        }

        public void RegisterType<TInterface, TImplem>() where TImplem : TInterface
        {
            container.RegisterType<TInterface, TImplem>();
        }

        public void RegisterInstance<T>(T instance)
        {
            container.RegisterInstance<T>(instance);
        }

        public void RegisterTypeWithKey<TInterface, TImplem>(string key) where
TImplem : TInterface
        {
            container.RegisterType<TInterface, TImplem>(key);
        }

        public TInterface ResolveWithKey<TInterface>(string key)
        {
            return container.Resolve<TInterface>(key);
        }

        public TInterface Resolve<TInterface>()
        {
            return container.Resolve<TInterface>();
        }
    }
```

3- TCP transport manager

Now that we have our serialized and our DI manager we can focus on the transport manager.
Basically, it will be responsible for sending requests to the server from clients and get back from the server the responses.

The transport will be managed by NetMQ messaging library.

NetMQ

NetMQ extends the standard socket interface with features traditionally provided by specialised messaging middleware products. NetMQ sockets provide an abstraction of asynchronous message queues, multiple messaging patterns, message filtering (subscriptions), seamless access to multiple transport protocols and more.

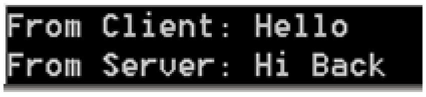
You need to get the package through NuGet

Example

The example is straight forward, we send a message from the client to the server, and the server sends a message back. This is a (extremely simple) example of the Request/Response pattern

```
using System;
using NetMQ;namespace HelloWorldDemo
{
class Program
{
private static void Main(string[] args)
{
using (NetMQContext ctx = NetMQContext.Create())
{
using (var server = ctx.CreateResponseSocket())
{
server.Bind("tcp://127.0.0.1:5556");
using (var client = ctx.CreateRequestSocket())
{
client.Connect("tcp://127.0.0.1:5556");


client.Send("Hello");

string fromClientMessage = server.ReceiveString();
Console.WriteLine("From Client: {0}", fromClientMessage);

server.Send("Hi Back");

string fromServerMessage = client.ReceiveString();

Console.WriteLine("From Server: {0}", fromServerMessage);

Console.ReadLine();
}
}
}
}
}
}
```

Console



```
From Client: Hello
From Server: Hi Back
```

Take away points there

1- We are able to create a request/response pattern, by using specialized sockets that are geared towards working in the request/response scenario.
2- We can use tcp as the protocol (ZeroMQ also supports others such as inproc)
3- We did not have to spin up any extra thread on the server to deal with the freshly connected client socket and then continue to accept other client sockets. In fact this code could pretty much talk to 1000nds of clients without much alteration at all (in fact I will show you an example of the using separate processes)

Running In Separate Threads

The server would spin up a worker for each client request

Topology:

Client: RequestSocket

Server: RouterSocket (TCP 5555) – DealerSocket (TCP 5556) & Poller

Worker: DealerSocket

The server has a frontend for the requests coming from the client.

It has the backend for communicating with the workers.

Poller is used to handle the messages

```
public class NetMQServer
    {
        private readonly string frontEndPoint;
        private readonly int frontPort;
        private readonly string backEndPoint = "localhost";
        private readonly int backPort = 5556;
        private static readonly ILogger logger = 
LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);

        public NetMQServer(string frontEndPoint, int frontPort)
        {
            this.frontEndPoint = frontEndPoint;
            this.frontPort = frontPort;
        }
        public void StartListening()
        {
            logger.Info("Starting server");
            using (var front = new RouterSocket())
            {
                using (var back = new DealerSocket())
                {
                    using (var poller = new NetMQPoller { front, back })
                    {
                        front.Bind($"tcp://{frontEndPoint}:{frontPort}");
                        back.Bind($"tcp://{backEndPoint}:{backPort}");

                        poller.Add(front);
                        poller.Add(back);

                        // creating the workers
                        // When the server's frontend receives a message, we want to spin 
up a new worker
                        front.ReceiveReady += (sender, eventArgs) =>
                        {
                            var optionPricingInterfaceServiceRegistration = new 
OptionPricingInterfaceServiceRegistration();
                            optionPricingInterfaceServiceRegistration.Register();
                            IOptionPricingJsonSerializer<Price> serializer = 
optionPricingInterfaceServiceRegistration.DependencyInjectionManager.Resolve<IOptionPricing
JsonSerializer<Price>>();
                            IOptionPricingPersistenceService 
optionPricingPersistenceService = 
optionPricingInterfaceServiceRegistration.DependencyInjectionManager.Resolve<IOptionPricing
PersistenceService>();
```

```
                            var mqMessage = eventArgs.Socket.ReceiveMultipartMessage(3);
                            var id = mqMessage.First;
                            var content = mqMessage[2].ConvertToString();
                            logger.Debug("Front received " + content);

                            ThreadPool.QueueUserWorkItem(ctx =>
                             {
                                // The worker
                                // Parameters are available from the context.
                                var context = (Tuple<NetMQFrame, string>)ctx;

                                 var clientId = context.Item1;
                                 var message = context.Item2;

                                // Send message to server's backend which then will return
the reply to the client
                                using (var workerConnection = new DealerSocket())
                                 {

workerConnection.Connect($"tcp://{backEndPoint}:{backPort}");

logger.Debug($"[{Thread.CurrentThread.ManagedThreadId}] worker");
                                     var messageToClient = new NetMQMessage();
                                     messageToClient.Append(clientId);
                                     messageToClient.AppendEmptyFrame();

                                     Price price = serializer.Deserialize(message);

optionPricingPersistenceService.InsertOption(price.OptionObj);
                                     PricingModelEnum pricingModel = price.PricingModel;
                                     IOptionPricingMethodService pricer =
optionPricingInterfaceServiceRegistration.DependencyInjectionManager.ResolveWithKey<IOption
PricingMethodService>(pricingModel.ToString());
                                     Price priceComputed = new
Price(pricer.Price(price.OptionObj), price.PricingModel, price.OptionObj);

optionPricingPersistenceService.InsertPrice(priceComputed);
                                     string serializedPrice =
serializer.Serialize(priceComputed);
                                     messageToClient.Append(serializedPrice);


workerConnection.SendMultipartMessage(messageToClient);
                                 }
                             }, Tuple.Create(id, content));
```

```
                        };
                        // Returning the message to client
                        back.ReceiveReady += (sender, eventArgs) =>
                        {
                            logger.Debug("Back received message, route to client");
                            var mqMessage = eventArgs.Socket.ReceiveMultipartMessage();
                            front.SendMultipartMessage(mqMessage);
                        };
                        poller.Run();
                    }
                }
            }
        }
    }
```

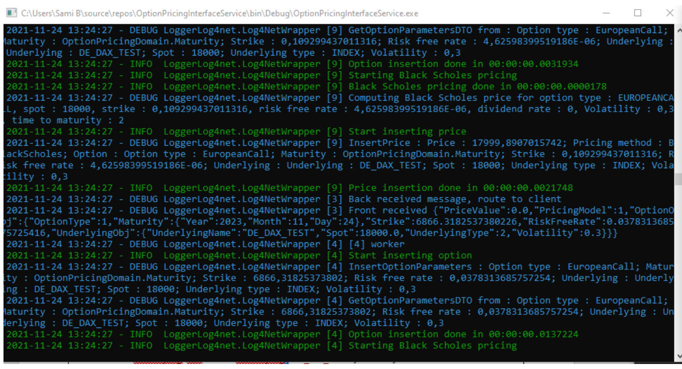
Launching several client that query several times the server

```
public static void Main(string[] args)
        {
            // Usually we use Thread.Join() method on all created threads to wait until all
threads are completed.
            // another way to join threads if threads are on thread pool : CountdownEvent
            int threadCount = 10;
            logger.Info("Starting Client Threads");
            using (CountdownEvent counter = new CountdownEvent(threadCount))
            {
                for (int task = 0; task < 10; task++)
                {
                    ThreadPool.QueueUserWorkItem(_ => NetMQClient.ClientTaskCallBack(task,
counter));
                    Thread.Sleep(1000);
                }
                counter.Wait();
            }
            logger.Info("All client threads finished execution");
        }
```

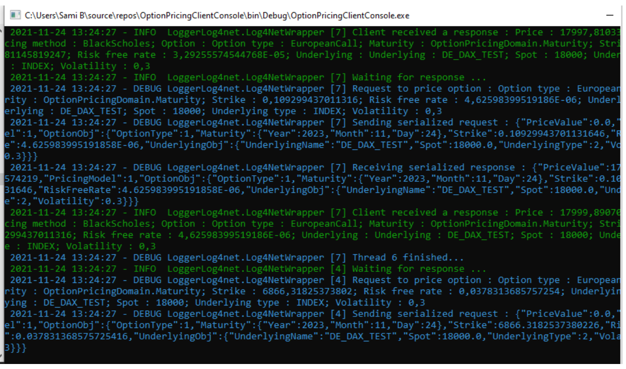
NetMQ client

```
public class NetMQClient
    {
        private static readonly ILogger logger =
LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);

        private readonly string endPoint;
        private readonly int port;

        public NetMQClient(string endPoint, int port)
        {
            this.endPoint = endPoint;
            this.port = port;
        }

        public Price StartSending(Price initialPrice)
        {
            using (var client = new RequestSocket())
            {
                client.Connect($"tcp://{endPoint}:{port}");
                IOptionPricingJsonSerializer <Price> optionPricingSerializer = new
OptionPricingJsonSerializer<Price>();
                IOptionPricingTcpTransportManager optionPricingTcpTransportManager = new
OptionPricingTcpTransportManager(endPoint, port, optionPricingSerializer);
                logger.Info("Waiting for response ...");
                Price computedPrice =
optionPricingTcpTransportManager.PriceOption(initialPrice);
                logger.Info($"Client received a response : {computedPrice}");
                return computedPrice;
            }
        }

        public static void ClientTaskCallBack(Object threadNumber, CountdownEvent evt)
        {
            string threadName = "Thread " + threadNumber.ToString();
            DateTime today = DateTime.Now;
            Maturity maturity = new Maturity(today.Year + 2, today.Month, today.Day);
            Underlying udl = new Underlying("DE_DAX_TEST", 18000d,
UnderlyingTypeEnum.INDEX, 0.3d);
            Option option = new Option(ContractEnum.EuropeanCall, maturity, 15000d, 0.04d,
udl);
            NetMQClient clientMQ = new NetMQClient("localhost", 5555);
            for (int task = 0; task < 10; task++)
            {
                clientMQ.StartSending(new Price(0d, PricingModelEnum.BlackScholes,
option.DummyOption()));
            }
            logger.Debug(threadName + " finished...");
            evt.Signal();
        }
    }
```

Console (with thread ids)



```
C:\Users\Sami B\source\repos\OptionPricingInterfaceService\bin\Debug\OptionPricingInterfaceService.exe
 2021-11-24 13:24:27 - DEBUG LoggerLog4net.Log4NetWrapper [9] GetOptionParametersDTO from : Option type : EuropeanCall;
Maturity : OptionPricingDomain.Maturity; Strike : 0,109299437011316; Risk free rate : 4,62598399519186E-06; Underlying :
 Underlying : DE_DAX_TEST; Spot : 18000; Underlying type : INDEX; Volatility : 0,3
 2021-11-24 13:24:27 - INFO  LoggerLog4net.Log4NetWrapper [9] Option insertion done in 00:00:00.0031934
 2021-11-24 13:24:27 - INFO  LoggerLog4net.Log4NetWrapper [9] Starting Black Scholes pricing
 2021-11-24 13:24:27 - INFO  LoggerLog4net.Log4NetWrapper [9] Black Scholes pricing done in 00:00:00.0000178
 2021-11-24 13:24:27 - DEBUG LoggerLog4net.Log4NetWrapper [9] Computing Black Scholes price for option type : EUROPEANCA
LL, spot : 18000, strike : 0,109299437011316, risk free rate : 4,62598399519186E-06, dividend rate : 0, Volatility : 0,3
, time to maturity : 2
 2021-11-24 13:24:27 - INFO  LoggerLog4net.Log4NetWrapper [9] Start inserting price
 2021-11-24 13:24:27 - DEBUG LoggerLog4net.Log4NetWrapper [9] InsertPrice : Price : 17999,8907015742; Pricing method : B
lackScholes; Option : Option type : EuropeanCall; Maturity : OptionPricingDomain.Maturity; Strike : 0,109299437011316; R
isk free rate : 4,62598399519186E-06; Underlying : Underlying : DE_DAX_TEST; Spot : 18000; Underlying type : INDEX; Vola
tility : 0,3
 2021-11-24 13:24:27 - INFO  LoggerLog4net.Log4NetWrapper [9] Price insertion done in 00:00:00.0021748
 2021-11-24 13:24:27 - DEBUG LoggerLog4net.Log4NetWrapper [3] Back received message, route to client
 2021-11-24 13:24:27 - DEBUG LoggerLog4net.Log4NetWrapper [3] Front received {"PriceValue":0.0,"PricingModel":1,"OptionO
bj":{"OptionType":1,"Maturity":{"Year":2023,"Month":11,"Day":24},"Strike":6866.3182537380226,"RiskFreeRate":0.0378313685
75725416,"UnderlyingObj":{"UnderlyingName":"DE_DAX_TEST","Spot":18000.0,"UnderlyingType":2,"Volatility":0.3}}}
 2021-11-24 13:24:27 - DEBUG LoggerLog4net.Log4NetWrapper [4] [4] worker
 2021-11-24 13:24:27 - INFO  LoggerLog4net.Log4NetWrapper [4] Start inserting option
 2021-11-24 13:24:27 - DEBUG LoggerLog4net.Log4NetWrapper [4] InsertOptionParameters : Option type : EuropeanCall; Matur
ity : OptionPricingDomain.Maturity; Strike : 6866,31825373802; Risk free rate : 0,0378313685757254; Underlying : Underly
ing : DE_DAX_TEST; Spot : 18000; Underlying type : INDEX; Volatility : 0,3
 2021-11-24 13:24:27 - DEBUG LoggerLog4net.Log4NetWrapper [4] GetOptionParametersDTO from : Option type : EuropeanCall;
Maturity : OptionPricingDomain.Maturity; Strike : 6866,31825373802; Risk free rate : 0,0378313685757254; Underlying : Un
derlying : DE_DAX_TEST; Spot : 18000; Underlying type : INDEX; Volatility : 0,3
 2021-11-24 13:24:27 - INFO  LoggerLog4net.Log4NetWrapper [4] Option insertion done in 00:00:00.0137224
 2021-11-24 13:24:27 - INFO  LoggerLog4net.Log4NetWrapper [4] Starting Black Scholes pricing
```



```
C:\Users\Sami B\source\repos\OptionPricingClientConsole\bin\Debug\OptionPricingClientConsole.exe
 2021-11-24 13:24:27 - INFO  LoggerLog4net.Log4NetWrapper [7] Client received a response : Price : 17997,81033
cing method : BlackScholes; Option : Option type : EuropeanCall; Maturity : OptionPricingDomain.Maturity; Stri
81145819247; Risk free rate : 3,29255574544768E-05; Underlying : Underlying : DE_DAX_TEST; Spot : 18000; Under
 : INDEX; Volatility : 0,3
 2021-11-24 13:24:27 - INFO  LoggerLog4net.Log4NetWrapper [7] Waiting for response ...
 2021-11-24 13:24:27 - DEBUG LoggerLog4net.Log4NetWrapper [7] Request to price option : Option type : European
rity : OptionPricingDomain.Maturity; Strike : 0,109299437011316; Risk free rate : 4,62598399519186E-06; Underl
erlying : DE_DAX_TEST; Spot : 18000; Underlying type : INDEX; Volatility : 0,3
 2021-11-24 13:24:27 - DEBUG LoggerLog4net.Log4NetWrapper [7] Sending serialized request : {"PriceValue":0.0,"
el":1,"OptionObj":{"OptionType":1,"Maturity":{"Year":2023,"Month":11,"Day":24},"Strike":0.10929943701131646,"R
e":4.625983995191858E-06,"UnderlyingObj":{"UnderlyingName":"DE_DAX_TEST","Spot":18000.0,"UnderlyingType":2,"Vo
0.3}}}
 2021-11-24 13:24:27 - DEBUG LoggerLog4net.Log4NetWrapper [7] Receiving serialized response : {"PriceValue":17
574219,"PricingModel":1,"OptionObj":{"OptionType":1,"Maturity":{"Year":2023,"Month":11,"Day":24},"Strike":0.10
31646,"RiskFreeRate":4.625983995191858E-06,"UnderlyingObj":{"UnderlyingName":"DE_DAX_TEST","Spot":18000.0,"Und
e":2,"Volatility":0.3}}}
 2021-11-24 13:24:27 - INFO  LoggerLog4net.Log4NetWrapper [7] Client received a response : Price : 17999,89070
cing method : BlackScholes; Option : Option type : EuropeanCall; Maturity : OptionPricingDomain.Maturity; Stri
299437011316; Risk free rate : 4,62598399519186E-06; Underlying : Underlying : DE_DAX_TEST; Spot : 18000; Unde
e : INDEX; Volatility : 0,3
 2021-11-24 13:24:27 - DEBUG LoggerLog4net.Log4NetWrapper [7] Thread 6 finished...
 2021-11-24 13:24:27 - INFO  LoggerLog4net.Log4NetWrapper [4] Waiting for response ...
 2021-11-24 13:24:27 - DEBUG LoggerLog4net.Log4NetWrapper [4] Request to price option : Option type : European
rity : OptionPricingDomain.Maturity; Strike : 6866,31825373802; Risk free rate : 0,0378313685757254; Underlyin
ying : DE_DAX_TEST; Spot : 18000; Underlying type : INDEX; Volatility : 0,3
 2021-11-24 13:24:27 - DEBUG LoggerLog4net.Log4NetWrapper [4] Sending serialized request : {"PriceValue":0.0,"
el":1,"OptionObj":{"OptionType":1,"Maturity":{"Year":2023,"Month":11,"Day":24},"Strike":6866.3182537380226,"Ri
":0.037831368575725416,"UnderlyingObj":{"UnderlyingName":"DE_DAX_TEST","Spot":18000.0,"UnderlyingType":2,"Vola
3}}}
```

The transport manager will expose a method to pricing an option coming from clients :

```
public interface IOptionPricingTcpTransportManager
    {
        Price PriceOption(Price clientInitialPrice);
    }
```

Finally, we have our infrastructure and interface service.

We can now focus on our front end

13-Logger

If you have been writing code for any reasonable amount of time, then you surely have handled logging. It's one of the most essential parts of modern app development.

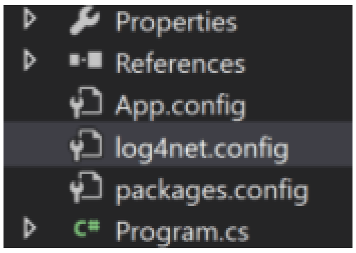
Here we will be covering a famous framework: **log4net**

The framework takes care of where to log to, whether to append to an existing file or create a new one, the formatting of the log message …

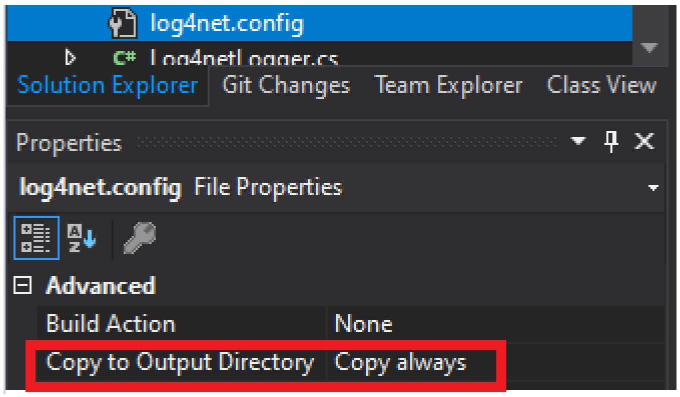
You can write your .NET logs to a file on disk, a database, a log management system … all without changing your code.

1- **Install log4net package**
First create a new solution (add it to your current project solution)
Use NuGet to download and install log4net

2- **Add log4net.config file**
Add a new file named log4net.config



Set **Copy to Output Directory** to **Copy Always** property



Build Action None
Copy to Output Directory Copy always

This is important because we need the **log4net.config** to be copied to the **bin** folder when you build and run your app

You can copy down below config, we'll discuss more about this further down

```
<log4net>
  <root>
    <level value="ALL" />
    <!--both colored-console-logging and file-logging is enabled-->
    <appender-ref ref="ColoredConsoleAppender" />
    <appender-ref ref="file" />
  </root>

<appender name="ColoredConsoleAppender"
type="log4net.Appender.ColoredConsoleAppender">
    <mapping>
      <level value="INFO" />
      <forecolor value="Green" />
    </mapping>
    <mapping>
      <level value="ERROR" />
      <forecolor value="Red" />
    </mapping>
    <mapping>
      <level value="DEBUG" />
      <forecolor value="Cyan" />
    </mapping>
    <mapping>
      <level value="WARN" />
      <forecolor value="Yellow" />
    </mapping>
    <layout type="log4net.Layout.PatternLayout">
      <param name="ConversionPattern" value="%d{ yyyy-MM-dd HH:mm:ss} – %-5p %type
[%thread] %m %n" />
    </layout>
  </appender>


  <!--log to file-->
  <appender name="file" type="log4net.Appender.RollingFileAppender">
    <file value="optionPricingApp.log" />
    <appendToFile value="true" />
    <rollingStyle value="Size" />
    <maxSizeRollBackups value="5" />
    <maximumFileSize value="10MB" />
    <staticLogFileName value="true" />
    <layout type="log4net.Layout.PatternLayout">
      <param name="ConversionPattern" value="%d{ yyyy-MM-dd HH:mm:ss} – %-5p %type
[%thread] %m %n stacktrace:%stacktrace{5} %n" />
    </layout>
  </appender>
</log4net>
```

3- How to tell log4net to load our config file

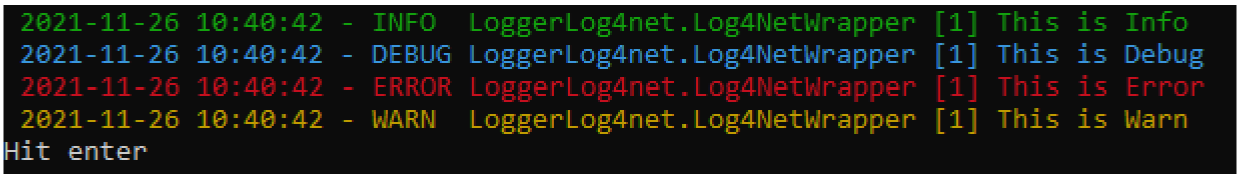
Now we need to tell log4net where to load its configuration from.

We'll put it in the AssemblyInfo.cs

```
[assembly: log4net.Config.XmlConfigurator(ConfigFile = "log4net.config")]
```

4- Test it out

```
public class Program
    {
        private static readonly ILogger log =
LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);

        static void Main(string[] args)
        {
            log.Info("This is Info");
            log.Debug("This is Debug");
            log.Error("This is Error");
            log.Warn("This is Warn");
            Console.WriteLine("Hit enter");
            Console.ReadLine();
        }
    }
```



```
 2021-11-26 10:40:42 - INFO  LoggerLog4net.Log4NetWrapper [1] This is Info
 2021-11-26 10:40:42 - DEBUG LoggerLog4net.Log4NetWrapper [1] This is Debug
 2021-11-26 10:40:42 - ERROR LoggerLog4net.Log4NetWrapper [1] This is Error
 2021-11-26 10:40:42 - WARN  LoggerLog4net.Log4NetWrapper [1] This is Warn
Hit enter
```

Log Appenders
Appenders are how you direct where you want your logs sent.
The most popular is the **RollingFileAppender** and **ConsoleAppender**
we are using the console; we used a cool appender: **ColoredConsoleAppender**

log levels

- all (log everything)
- Debug
- Info
- Warn
- Error
- Fatal
- Off (don't log anything)

Best practices

1- **Create a wrapper around log4net**
Expose only the methods that will be used by creating an interface and implementing it

```
public interface ILogger
    {
        void Debug(object message);
        bool IsDebugEnabled { get; }
        void Info(object message);
        void Error(object message);
        void Warn(object message);
    }
```

Wrapper implementation

```
public class Log4NetWrapper : ILogger
    {
        private readonly log4net.ILog logger;

        public Log4NetWrapper(Type type)
        {
            this.logger = log4net.LogManager.GetLogger(type);
        }

        public void Debug(object message)
        {
            logger.Debug(message);
        }

        public void Info(object message)
        {
            logger.Info(message);
        }

        public void Error(object message)
        {
            logger.Error(message);
        }

        public void Warn(object message)
        {
            logger.Warn(message);
        }

        public bool IsDebugEnabled
        {
            get { return logger.IsDebugEnabled; }
        }
    }
```

2- Define LogManager Object as Static

Declaring any variable has overhead.
Constructors on the LogManager object can use a lot of CPU
Declare it as static, plus use down below so you don't hard code the class name

```
public static class LogManager
    {
        public static ILogger GetLogger(Type type)
        {
            return new Log4NetWrapper(type);
        }
    }
```

```
private static readonly ILogger log =
LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod().DeclaringTy
pe);
```

3- Customize your layout in your logs with log4net pattern layouts

Write log level with **%p**
Current date time with **%d**
Thread with **%thread**
The message with **%m**
New line with **%n**
Name of the method where the log message was written with **%method**
Output a stack trace to show where the log message was written with **%stacktrace{level}**
Type of the caller issuing the log request. (class name) with **%type**
the line number from where your logging statement was logged with **%line**

14-WPF app

sami.benabidallah@digistratconsulting.com