Object Oriented Programming with Applications
Lecture 3

David Šiška & Witold Gawlikowicz

School of Mathematics, University of Edinburgh

2018-19[1]

[1] Last updated 10th October 2018

Lecture 3

- Object-oriented programming paradigm
  - Structs
  - Classes & Objects
  - Value types and reference types
- Generics
- Anonymous functions

Read:
Wright, P. - Beginning Visual C# 2005 Express Edition. Chapter 4 and 6.
Duffy, D. J. and Germani, A. - C# for Financial Markets, Chapter 3, and Chapter 4 Sections 4.1–4.5.

Procedural Programming Paradigm

To solve a given problem

- Split it into separate sub-problems; each sub-problem is taken care of by a “method” a.k.a. “procedure” a.k.a. “function”.
- Use struct to group data and create data structures.

Struct

A `struct` is a value type that is used to encapsulate (group) small number of related variables together. Examples:

```
public struct BlackScholesModelParams
{
    public double S; // price of underlying stock
    public double sigma; // volatility
    public double r; // risk-free rate
}

public struct EuropeanOptionParams
{
    public char type; // 'c' for call, 'p' for put
    public double T; // expiry time as a year fraction
    public double K; // strike
}
```

Then an implementation of the Black–Scholes formula could be

```
public static double BlackScholesOptionPrice(BlackScholesModelParams model,
                                             EuropeanOptionParams option)
{  // Code goes here }
```

Object Oriented Programming

Main ideas:

- Group together related data and methods for manipulating the data into *classes*.
- Split the methods and data into *public* and *private*.
    - *public* methods and data can be accessed by program code *outside* the class safely without knowledge about how the data is organised and how the methods work.
    - *private* methods and data can only be accessed by program code *inside* the class.
- Instances of classes are *objects*.

Example class: Date

```
class Date
{
    private int day;
    private int month;
    private int year;

    // Default constructor
    public Date()
    {   day = 1; month = 1; year = 1;  }

    // Copy constructor
    public Date(Date d)
    {  day = d.day; month = d.month; year = d.year; }
```

Constructors: methods for creating *instances* of our class.

- The *default constructor* - the most basic setup for a working instance of our class.
- The *copy constructor* - useful for combining our classes with built-in data structure e.g. lists, trees etc. that may need to create copies of objects.

Example class: Date, continued

So far our Date class is not useful.

```
private static bool IsValidDate(int day, int month, int year)
{
    bool isValid = false;
    // do some checks
    return isValid;
}

public static Date CreateDate(int day, int month, int year)
{
    Date d = new Date();
    if (IsValidDate(day, month, year))
    {
        d.day = day; d.month = month; d.year = year;
        return d;
    }
    else
        throw new System.ArgumentOutOfRangeException("Invalid date!");
}
```

Keyword `static`: Methods do not access any instance variables.

Example class: Date, continued

Few more things needed. . .

```
    public int GetDayOfMonth() { return day; }

    public void SetDayOfMonth(int day)
    {
        if (IsValidDate(day, this.month, this.year))
        {
            this.day = day;
        }
        else
            throw new System.ArgumentOutOfRangeException("Invalid date!");
    }
}
```

Example class: Date, continued

What else would you like a date class be able to do?

- Get / Set months, years.
- Add a number of days, months, years.

The structure of a class

- Data
- Methods
  - Constructors, with the special *default* and *copy* constructors.
  - Selector e.g. `public int GetDayOfMonth()`
  - Modifier e.g. `public void SetDayOfMonth(int day)`
  - Other e.g. `public void AddDays(int numDaysToAdd)`
  - Destructor: (not used so far). Used to close open files / network connections.

More object-oriented techniques

- Method overloading
- Inheritance
- Interfaces

Method overloading

This technique involves having the same method name and return type for methods with different arguments.

We say that such methods have different signatures.

Example: `Console.WriteLine` has 19 overloaded methods:

```
public static void WriteLine();
public static void WriteLine(ulong value);
public static void WriteLine(uint value);
public static void WriteLine(string value);


...


public static void WriteLine(string format, params[] obj arg);
```

Inheritance

Say you have class `B` called the *base class*.

You can use it to define a class `D` called the *derived class* which contains all the methods and data of `B` but also any other methods and data you add.

```
public class D : B
{
    public D() : base() // the default constructor starts by
                        // calling the default constructor in B
    {
        // some more code specific to D
    }

    // more data for class D
    // more methods for class D
}
```

Inheritance: abstract and sealed

- `abstract` denotes a class that cannot be instantiated (i.e. you cannot do `new AbstractClass();`. It is meant as a basis for creating derived class.
- `sealed` denotes a class that cannot serve as a base class i.e. one cannot create derived classes based on a `sealed` class.

Inheritance: example

`abstract` denotes that one cannot create instances of this class: you first have to create a derived class.

```
public abstract class AbstractBond
{
    private Vector<Date> couponDates;
    // ...
    public abstract YieldToPrice(double yield);
    public abstract PriceToYield(double price);
}
```

Inheritance: abstract and example

Fixed coupon bond: the coupon is fixed

```
public class Bond : AbstractBond
{
    // we already have coupon dates from AbstractBond
    private double coupon; // annualized

    public YieldToPrice(double yield)
    {
        // Actual implementation of the calculation
    }
    public PriceToYield(double price)
    {
        // Actual implementation of the calculation
    }
}
```

Inheritance: example

Floating rate note: coupon is “floating” typically expressed as a LIBOR + spread, but formula could be more complex.

```
public class FloatingRateNote : AbstractBond
{
    // we already have coupon dates from AbstractBond
    private string couponFormula; // e.g. LIBOR3M + 50bp.

    public abstract YieldToPrice(double yield)
    {
        // Actual implementation of the calculation
    }
    public abstract PriceToYield(double price)
    {
        // Actual implementation of the calculation
    }
}
```

Interfaces

Interfaces are a way of specifying functionality without specifying how such functionality is achieved.

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

Now if we have a class whose instances can be checked for equality we can “broadcast” that by

```
class Date : IEquatable<Date>
{
    // ... previous Date code
    bool Equals(T d)
    {
        return (day == d.day && month == d.month && year == d.year);
    }
}
```

Why not just use `d1 == d2` instead?

Interfaces

Interfaces: why bother?

Allows, together with *generics* creation of generic algorithms e.g.

```
static void MySort<Type>(Type[] array)
     where Type : System.IComparable<Type>
{
    // sorting algorithm can now use
    // array[i].CompareTo(array[i + 1])>0 to see what is bigger
}
```

Generics

Write code once that works for multiple data types.[2]

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T tmp = rhs;
    rhs = lhs;
    lhs = tmp;
}
public static void Main (string[] args)
{
    int lhs = -10, rhs = 10;
    Console.WriteLine ("lhs={0}, rhs={1}", lhs, rhs);
    Swap<int> (ref lhs, ref rhs);
    Console.WriteLine ("lhs={0}, rhs={1}", lhs, rhs);

    double left = -10.1, right = 10.1;
    Console.WriteLine ("lhs={0}, rhs={1}", left, right);
    Swap<double> (ref left, ref right);
    Console.WriteLine ("lhs={0}, rhs={1}", left, right);
}
```

[2]Similar to Templates in C++.

Anonymous functions

This is really cool for writing numerical code!

- Create an object that is itself a function.[3]
- Use as:

```
Func<double, double> normalDensity
    = (x) => Math.Exp (-x * x / 2.0) / Math.Sqrt (2 * Math.PI);
```

- Notation

```
Func<input type, return type> arbitraryName
    = (x) => arbitrary code;
```

[3]In C you could have used function pointer.

Anonymous functions

- Then to write a method calculating e.g. integral you can have

```
static double IntegrateCompTrapezium(
               Func<double, double> f, double a, double b, int N)
{
    double h = (b - a) / N;
    double integral = 0;
    for (int i = 0; i < N; i++) {
        double x0 = a + i * h, x1 = a + (i + 1) * h;
        integral += (0.5 * (f (x0) + f (x1)))*h;
    }
    return integral;
}
```

- Use as

```
double integral = IntegrateCompTrapezium(normalDensity, -10,10,10000);
```