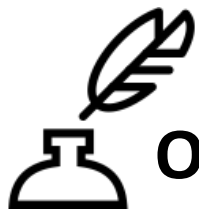# TDDD49/725G66
C# and .NET Programming

(Lecture 02)

Sahand Sadjadee
Linköping University

Outline

1. An introduction to Camedin
2. A deeper look into object-oriented programming in C#
3. 3 tier architecture
4. Exception handling

Camedin

Camedin is a tool used during the lab sessions to facilitate the assistance process.

1. Open the following link when you are in the lab rooms.

   http://www.camedin.com/live/b651b7af532c43b6abf458e083a9cc27
2. Page one of the assistants by clicking on the page button.
3. Provide a complete set of information as requested upon paging.

- Try to page the assistant with the least number of students in his queue!
- No login is required.

Camedin

DEMO

Object-oriented Programming
in C#

Inheritance

Inheritance, together with **encapsulation** and **polymorphism**, is one of the three primary characteristics (or *pillars*) of object-oriented programming. Inheritance enables you to create new classes that reuse, extend, and modify the behavior that is defined in other classes. The class whose members are inherited is called the ***base class***, and the class that inherits those members is called the ***derived class***.

1. A derived class can have only one direct base class
2. Inheritance is transitive

LINKÖPINGS UNIVERSITET

Inheritance

A class can inherit from other classes and interfaces defined by the programmer itslef, provided by the .Net framework or a third party.

Inheritance is a good choice when:

● Your inheritance hierarchy represents an "is-a" relationship and not a "has-a" relationship.

● You can reuse code from the base classes.

● You need to apply the same class and methods to different data types.

● The class hierarchy is reasonably shallow, and other developers are not likely to add many more levels.

● You want to make global changes to derived classes by changing a base class.

LINKÖPINGS UNIVERSITET

Inheritance- System.Object

Supports all classes in the .NET Framework class hierarchy and provides low-level services to derived classes. This is the ultimate base class of all classes in the .NET Framework; it is the root of the type hierarchy.

https://msdn.microsoft.com/en-us/library/system.object(v=vs.110).aspx

Inheritance- class hierarchy

1. Class hierarchies are defined from the general to the specific.
2. Be generous in defining data types and storage to avoid difficult changes later on.
3. Only expose items that are needed by derived classes.
4. Members that are only needed by derived classes should be marked as Protected.
5. Make sure that base class methods do not depend on Overridable members, whose functionality can be changed by inheriting classes.

https://msdn.microsoft.com/en-us/library/6csyy24x(v=vs.90).aspx

LINKÖPINGS UNIVERSITET

Inheritance- syntax

```
Class Creature
{
        public int LifeSpan;
}


Class Human : Creature
{
        Public void walk(int meters)
        {
        }
}
```

Inheritance syntax

https://msdn.microsoft.com/en-us/library/6csyy24x(v=vs.90).aspx

Inheritance- Sealed classes

https://msdn.microsoft.com/en-us/library/88c54tsw.aspx

When applied to a class, the **sealed** modifier prevents other classes from inheriting from it.

```
class A {}
sealed class B : A {}
```

Interfaces

- An interface contains definitions for a group of related functionalities that a class or a struct can implement.

- A solution to the limitation of single inheritance in classes.

- A solution to the limitation of no inheritance in structs.

- A class/struct can inherit from multiple interfaces.

- **Interfaces contain abstract methods which do not have any implementation.**

- Interfaces cannot be instantiated.

-

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

Class Text : IEquatable<Text>
{
        public bool Equals(Text obj) //implementing Equals
        {
        }
}
```

LINKÖPINGS UNIVERSITET

Interfaces

- Any class or struct that implements the interface must implement all its members.
- A class or struct can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.

```
Interface A
{
}
Interface B
{
}
Class Base
{
}
Class Derived : Base, A, B
{
}
```

LINKÖPINGS UNIVERSITET

Abstract classes/Methods

- An abstract class is a class which is declared using the keyword abstract.
- An abstract class has **usually** at least one abstract method.
- An abstract method is a method which does not have any implementation and is declared using the abstract keyword.
- It is possible to have an abstract class without any abstract methods.
- Like interfaces, abstract classes **cannot** be instantiated.

Virtual methods

- Virtual methods are declared using the virtual keyword.
- Unlike abstract methods, virtual methods do have an implementation but the implementation might not be useful for the derived classes.
- Virtual methods **usually** need to be overridden using the override keyword in the derived classes.
- A base class can implement a method, inherited from an interface, using the virtual keyword. In result the derived class can also override the inherited method from the base class,.

Overriding vs hiding

● Concrete and non-virtual methods can only be **hidden** using the `new` keyword in the derived class.
● Concrete and Virtual methods can only be **overridden** by using the `override` keyword in the derived class.

Difference? Please check the provided link for more information?

LINKÖPINGS UNIVERSITET

Polymorphism(many-shaped)

- At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this occurs, the object's declared type is no longer identical to its run-time type.

- Base classes may define and implement virtual *methods*, and derived classes can override them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. Thus in your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

Polymorphism

```
Interface Animal{
        int move(int meter);
}

Class Cat : Animal{
        Public override int move(int meter)
        {
                System.Console.Write("Cat is moving...")
                // implementation
        }
}
Class Dog : Animal{
        Public override int move(int meter)
        {
                System.Console.Write("Dog is moving...")
                // implementation
        }
}
```

An array of creatures

C	C	D	C	D	C

```
Creature[] cr = new Creature[6];
Cr[0] = new Cat();
Cr[1] = new Cat();
Cr[2] = new Dog();
Cr[3] = new Cat();
Cr[4] = new Dog();
Cr[5] = new Cat();

For (int i = 0 ; i < cr.Length ; i++)
{
        cr[i].move();
}
```

LINKÖPINGS UNIVERSITET

Casting

- Implicit conversions
- **Explicit conversions (casts)**
- User-defined conversions
- Conversions with helper classes

Check the provided link for more information!

```csharp
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
// Output: 1234
//=============================================================
// Create a new derived type.
Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.
Animal a = g;

// Explicit conversion is required to cast back
// to derived type. Note: This will compile but will
// throw an exception at run time if the right-side
// object is not in fact a Giraffe.
Giraffe g2 = (Giraffe) a;
```

LINKÖPINGS UNIVERSITET

Is operator

- Checks if an object is compatible with a given type.
- An is expression evaluates to true if the provided expression is non-null, and the provided object can be cast to the provided type without causing an exception to be thrown.

LINKÖPINGS UNIVERSITET

Composition/Aggregation

In a composition relationship, an object holds a reference to another object. In other words ObjectA has ObjectB.

Composition/Aggregation are considered a "has-a" relationship.
Composition and aggregation are almost similar concepts except for that a composited object's existence is tied with the owning object.

LINKÖPINGS UNIVERSITET

Composition/Aggregation - example

```
Class Human
{
     Public Hand RightHand = null;
     Public Hand LeftHand = null;
     Public Human(){
         RightHand = new Hand();
         LeftHand = new Hand();
     }

}


//Notes:
// Composition or aggregation?
// No encapsulation
// No dependency injection => tight
coupling
```

```
Class Hand
{
     Public Finger[] Fingers = new
     Finger[5];
     Public Hand()
     {
             for(int i = 0; i <
     Fingers.Length; ++i)
             {
                 Fingers[i] = new Finger();
             }
     }
}

Class Finger
{
...
}
```

Encapsulation

- *Encapsulation* means that a group of related properties, methods, and other members are treated as a single unit or object.
- There shall be limited and controlled access to the fields.
- Avoid using public access modifier for non-static fields.
- Use properties, setter/getter methods and/or constructors for allowing access to the fields.

LINKÖPINGS UNIVERSITET

Encapsulation

```
Class Human
{
      private Hand RightHand = null;
      private Hand LeftHand = null;
      Public Human(){
            RightHand = new Hand();
            LeftHand = new Hand();
      }
      Public Hand GetRightHand(){
            Return RightHand;
      }
      Public Hand GetLeftHand(){
            Return LeftHand;
      }


}
```

Some points:

1. As no setters are provided then the hands cannot be replaced and will be permanent.
2. Properties can be used instead

Dependency Injectio https://msdn.microsoft.com/en-us/library/hh323705(v=vs.100).aspx

- Dependency Injection (DI) is a design pattern that demonstrates how to create loosely coupled classes.
- An object can be dependent on other objects or even primitive values (Composition/Aggregation)
- The dependency shall be first created and then injected into the object via the setter methods and/or the constructors.

Example:

A car is dependent on the engine in order to move.
The engine is created first and then injected into the car in the factory.

Dependency Injection + Encapsulation = correct impl

```
Class Human
{
        private Hand RightHand = null;
        private Hand LeftHand = null;
        Public Human(Hand lh, Hand, rh){
                RightHand = rh;
                LeftHand = lh;
        }
        Public Hand GetRightHand(){
                Return RightHand;
        }
        Public Hand GetLeftHand(){
                        Return LeftHand;
        }
}

Main:
Fingers[] fs1 = new Fingers[5];
Fingers[] fs2 = new Fingers[5];
Hand LeftHand = new Hand(fs1);
Hand RightHand = new Hand(fs2);
Human Sahand = new Human(LeftHand, RightHand);

//Notes:
// Composition or aggregation?
// Encapsulation and programming to interface is in place!
// Dependency Injection implemented => Loose coupling
```

```
Class Hand
{
        Public Finger[] Fingers = null;

        Public Hand(Fingers[] fs)
        {
                Fingers = fs
        }
}

Class Finger
{
...
}
```

LINKÖPINGS UNIVERSITET

Programming to interface

- The focus shall be on what the object does, not how it is done.
- The public methods form the interface of the object.
- "Interfaces" shall be used to define the interface of the object.
- Interfaces contain what is going to be available to the outside
- The inheriting classes override the interface and implement how the methods are going to do the tasks.

Example:
In case of a setter method, we don't care how the data is stored in the class. What we do care that the setter method does the storage and the getter method retrieves the data without any loss.

Boxing and unboxing

- Boxing is the process of converting a value type to the type **object** or to any interface type implemented by this value type. When the CLR boxes a value type, it wraps the value inside a System.Object and stores it on the managed heap. Unboxing extracts the value type from the object.
- Boxing is implicit; unboxing is explicit.

```
int i = 123;
// The following line boxes i.
object o = i;

Object o = 123;
i = (int)o;   // unboxing
```

Generics

by using a generic type parameter T you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations, as shown here:

```csharp
// Declare the generic class.
public class GenericList<T>
{
    void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
    }
}
```

LINKÖPINGS UNIVERSITET

Generics

- Use generic types to maximize code reuse, type safety, and performance.

- The most common use of generics is to create collection classes.

- The .NET Framework class library contains several new generic collection classes in the System.Collections.Generic namespace. These should be used whenever possible instead of classes such as ArrayList in the System.Collections namespace.

- You can create your own generic interfaces, classes, methods, events and delegates.

- Generic classes may be constrained to enable access to methods on particular data types.

LINKÖPINGS UNIVERSITET

System.Collections namespace

The System.Collections namespace contains interfaces and classes that define various collections of objects, such as lists, queues, bit arrays, hash tables and dictionaries.

https://msdn.microsoft.com/en-us/library/system.collections(v=vs.110).aspx
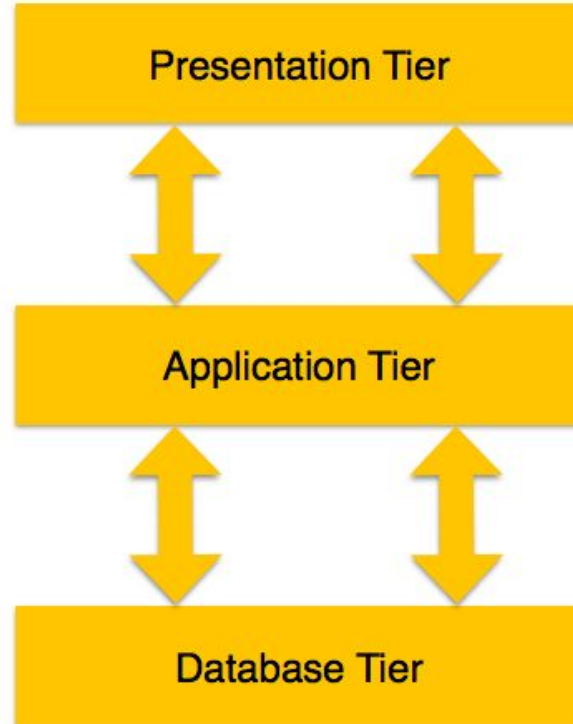
Three-Tier Architecture

Three-Tier Architecture

The three-tier architecture model, which is the fundamental framework for the logical design model, segments an application's components into three tiers of services. These tiers do not necessarily correspond to physical locations on various computers on a network, but rather to logical layers of the application.

1. The presentation tier, or user services layer, gives a user access to the application.
2. The middle tier, or business services layer, consists of business and data rules.
3. The data tier, or data services layer, interacts with persistent data usually stored in a database or in permanent storage.

LINKÖPINGS UNIVERSITET

Three-Tier Architecture

Three-Tier Architecture https://msdn.microsoft.com/en-us/library/windows/desktop/ms685068(v=vs.85).aspx

How Composition can help to implement the Three-Tire architecture?

LINKÖPINGS UNIVERSITET

Three-Tier Architecture

How Composition can help to implement the Three-Tire architecture?

Some sketchings…

```
class Presentation
{
      private Logic logic = null;
      public Presentation(Logic l)
      {
            Logic = l;
      }

}
```

```
class Logic
{
      private Data datal = null;
      public Logic(Data d)
      {
            Data = d;
      }

}
```

```
class Data
{
      File xml = null
      public Data(File f)
      {
            Xml = f;
      }

}
```

```
Main:
Presentation p = new Presentation(new Logic(new Data(new File("c:\temp\database.xml"))));
```

Dependency injection to have loosely coupled layers!

LINKÖPINGS UNIVERSITET

Passing data among layers/tiers

Methods:
1. Non-uniform
2. Uniform
3. DTO
4. Hybrid(DTO + Entity)

http://www.codeproject.com/Articles/493389/Four-ways-of-passing-data-between-layers
https://msdn.microsoft.com/en-us/library/ff649585.aspx

LINKÖPINGS UNIVERSITET

Exception Handling

(Why is it called an "Exception"?)

Exception handling

The C# language's exception handling features help you deal with any unexpected or exceptional situations that occur when a program is running. Exception handling uses the **try**, **catch**, and **finally** keywords to try actions that may not succeed, to handle failures when you decide that it is reasonable to do so, and to clean up resources afterward.

Exceptions can be generated by:

● The Common Language Runtime (CLR)
● The .NET Framework
● Any third-party libraries
● The application code

LINKÖPINGS UNIVERSITET

Exception handling

- Exceptions are types that all ultimately derive from **System.Exception**.

- Use a **try** block around the statements that might throw exceptions.

- Once an exception occurs in the **try** block, the flow of control jumps to the first associated exception handler that is present anywhere in the call stack. In C#, the **catch** keyword is used to define an exception handler.

- If no exception handler for a given exception is present, the program stops executing with an error message.

- Do not catch an exception unless you can handle it and leave the application in a known state. If you catch**System.Exception**, rethrow it using the **throw** keyword at the end of the **catch** block.

- If a **catch** block defines an exception variable, you can use it to obtain more information about the type of exception that occurred.

- Exceptions can be explicitly generated by a program by using the **throw** keyword.

- Exception objects contain detailed information about the error, such as the state of the call stack and a text description of the error.

- Code in a **finally** block is executed even if an exception is thrown. Use a **finally** block to release resources, for example to close any streams or files that were opened in the **try** block.
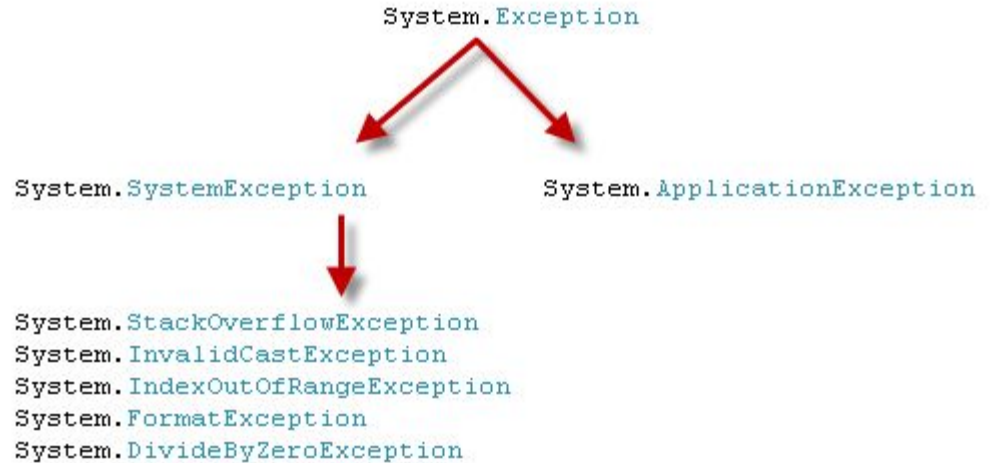
Exception handling

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
}
catch (SomeSpecificException2 ex)
{
    // Code to handle the exception goes here.
}
catch (SomeSpecificException3 ex)
{
    // Code to handle the exception goes here.
}
finally
{
    // Code to execute after the try (and possibly catch) blocks
    // goes here.
}
```

LINKÖPINGS UNIVERSITET

Exception handling - System.Exception

- The StackTrace property.
- The InnerException property.
- The Message property.
- The HelpLink property.

System.Exception

System.SystemException

System.ApplicationException

System.StackOverflowException
System.InvalidCastException
System.IndexOutOfRangeException
System.FormatException
System.DivideByZeroException

https://msdn.microsoft.com/en-us/library/5whzhsd2(v=vs.110).aspx

LINKÖPINGS UNIVERSITET

Exception handling - creating user-defined exceptions

- Inherit from ApplicationException or Exception.

- Each layer or component has its own problems and shall have a set of custom designed exceptions.

- Exceptions are used for controlling the execution flow, upon a problem, and passing some information about the occurred exception.

https://msdn.microsoft.com/en-us/library/z7cdya2t(v=vs.110).aspx

LINKÖPINGS UNIVERSITET

Exception handling - creating user-defined exceptions

```csharp
using System;

public class EmployeeListNotFoundException: Exception
{
    public EmployeeListNotFoundException()
    {
    }

    public EmployeeListNotFoundException(string message)
        : base(message)
    {
    }

    public EmployeeListNotFoundException(string message, Exception inner)
        : base(message, inner)
    {
    }
}
```

https://msdn.microsoft.com/en-us/library/87cdya3t(v=vs.110).aspx

LINKÖPINGS UNIVERSITET

Exception handling - Three-Tier architecture

http://www.codeproject.com/Articles/85569/Exception-Handling-in-Tier-Architecture

Thanks for listening!