



DIDA 2023/24

Lab. 2

Additional C# Topics

Summary

1. Properties
2. Exceptions
3. Delegates
4. Lambda Expressions
5. Generics
6. Threads and synchronization

1. Properties

Get/Set

Properties

- Simple way to control the access to the private attributes of classes, structs and interfaces:

```
public class X {  
    private string name;  
    private int age;  
  
    public string Name {  
        get { return name; }  
        set { name = value; }  
    }  
}
```

(...)

```
x.Name = "Smith";  
Console.WriteLine("I'm called {0}", x.Name);
```

- Formalizes the concept of Get/Set methods
- Makes code more readable

2. Exceptions

Syntax
Throwing
Intercepting

Exceptions

- Should correspond to **exceptional** situations.
- Thrown by the system or using the `throw` command.
- Used by putting code inside a `try-catch` block:

```
try {  
    //code that generates the exception  
} catch (<ExceptionType> e) {  
    // handling of exception e  
} finally {  
    // is always executed  
    // usually the release of allocated resources  
}
```

- New exceptions can be created by deriving `System.ApplicationException`

Exception Throwing

```
class MyException: ApplicationException {  
    // arbitrary methods and attributes  
}  
  
...  
  
if (<error condition>) {  
    throw new MyException();  
}
```

Exception Interception

- Go up the stack searching for a `try` block,
- Look for a matching `catch` block.
- If there is none, run the `finally` block
- And continue up the stack, executing the `finally` blocks until the exception is caught or the program is terminated. ☹

Sample System Exceptions

- `System.ArithmeticException`
- `System.ArrayTypeMismatchException`
- `System.DivideByZeroException`
- `System.IndexOutOfRangeException.`
- `System.InvalidCastException`
- `System.MulticastNotSupportedException`
- `System.NullReferenceException`
- `System.OutOfMemoryException`
- `System.OverflowException`
- `System.StackOverflowException`
- `System.TypeInitializationException`

Exception Tips

- Don't throw exceptions for normal program control flow.
- Don't throw exceptions you're not going to handle.
- When you catch an exception but you need to throw it again, use the isolated `throw` clause. It avoids rewriting the exception stacktrace.

```
try {  
    //code that may generate the exception  
} catch (<ExceptionType> e) {  
    // handling of exception e  
    throw; // instead of "throw e;"  
}
```

3. Delegates

Delegates

- Similar to function pointers:

```
bool (*myFunction) (int) /* in C */
```

- Pointers to object or class methods:

```
delegate bool MyDelegate(int x);  
MyDelegate md = new MyDelegate(a method);
```

- Delegates keep a list of methods.
- Can be manipulated with arithmetic operations:
Combine (+), Remove (-)
- An empty delegate is equal to null.

Delegates (cont.)

```
delegate void MyDelegate(string s);

class MyClass {
public static void Hello(string s) {
    Console.WriteLine(" Hello, {0}!", s);
}

public static void Goodbye(string s) {
    Console.WriteLine(" Goodbye, {0}!", s);
}
```

```
public static void Main() {
    MyDelegate a, b, c;
    a = new MyDelegate(Hello);
    b = new MyDelegate(Goodbye);
    c = a + b;
    a("A");
    b("B");
    c("C");
}
}
```

```
Hello, A!
GoodBye, B!
Hello, C!
GoodBye, C!
```

4. C#: Lambda Expressions

A lambda is an anonymous function of the format:

```
(input-parameters) => expression
```

or

```
(input-parameters) => { <sequence-of-statements> }
```

```
// Example 1
```

```
Func<int, int, bool> testForEquality = (x, y) => x == y;  
Console.WriteLine(testForEquality(4,4));
```

```
//Example 2
```

```
int[] numbers = { 2, 3, 4, 5 };  
var squaredNumbers = numbers.Select(x => x * x);
```

5. Generics

Collections

Classes

Methods

Generics (C# 2.0)

- Allow the definition of **strongly typed** structures.

Instead of:

```
using System.Collections;
ArrayList list = new ArrayList();
list.Add(1); // should check if 1 is int
int i = (int) list[0];
```

,we can have:

```
using System.Collections.Generic;
List<int> list = new List<int>();
list.Add(1); // no check needed
int i = list[0]; // no cast needed
```


Generics (classes)

- Programmers can create generic classes:

```
public class MyLista<T> { // T is any type
    T[] m_Items;

    public MyLista ():this(100) {}
    public MyLista (int size) { m_Items = new T[m_Size]; }
    public void Add(T item) { ... }
    public T Remove(int index) { ... }
    public T Get(int index) { ... }
}
```

- And use it:

```
MyLista<char> characters = new MyLista<char>(10);
characters Add('c');
char character = characters.Get(0); // no cast needed
```

Generics (methods)

- Programmers can also define methods using Generics:

```
public class AClass {  
    public T Add<T>(T item) { ... }  
}
```

- The type used is inferred during compile time.
- Use example:

```
AClass c = new AClass();  
c.Add('c'); // Add(char)  
c.Add(6); // Add(int)
```

6. Threads and Monitors

Threads

- When to use threads:
 - Simultaneous tasks
 - Sharing data
 - Performance is more important than fault tolerance

- **Construction (traditional):**

```
//ThreadStart is a public delegate void ThreadStart();  
ThreadStart ts = new ThreadStart(y.xpto);  
Thread t = new Thread(ts);  
t.Start(); // start execution  
t.Join(); // wait for termination
```

Threads (cont.)

- Construction with a lambda as parameter:

```
public Thread StartTheThread(SomeType param1, SomeOtherType
param2)
    var t = new Thread(() => RealStart(param1, param2));
    t.Start();
    return t;
}
```

```
private static void RealStart(SomeType param1, SomeOtherType
param2) {
    ...
}
```

Threads (cont.)

- **Other Thread methods:** Abort, Sleep, Join
- **Wait/Join example :**

```
using System;  
using System.Threading;
```

```
public class Alpha  
{  
    public void Beta()  
    {  
        while (true)  
        {  
            Console.WriteLine("A.B is running in its own thread.");  
        }  
    }  
};
```

Threads (cont.)

```
public class Simple
{
    public static int Main()
    {
        Alpha oAlpha = new Alpha();
        Thread oThread = new Thread(new ThreadStart(oAlpha.Beta));
        oThread.Start();

        // Spin for a while waiting for the started thread to become alive:
        while (!oThread.IsAlive);

        // Put the Main thread to sleep for 1 ms to allow oThread to work:
        Thread.Sleep(1);

        // Request that oThread be stopped
        oThread.Abort();
    }
}
```

Synchronization: Monitors

- Thread concurrency requires synchronization.
- `lock` primitive provides mutual exclusion.
- Two standard options:
 - `lock(this)`, mutual exclusion for all methods of one object.
 - `lock(typeof(this))`, mutual exclusion for all methods of one class.

Synchronization (cont.)

- Monitors:

- `Monitor.Enter(this);` [equivalent to `lock(this)`]
 - Gets an exclusive lock on the current object (`this`)
- `Monitor.Wait(this);`
 - Releases the lock over the current object and blocks until it receives a Pulse.
- `Monitor.Pulse(this);`
 - Wakes up one of the threads that called `Wait`. It will run again when it's alone in the current object.
- `Monitor.PulseAll(this);`
 - Wakes up all the threads that called `Wait` on the current object. One will run again when it's alone in the current object. The others will block again.
- `Monitor.Exit(this);`
 - Releases the exclusive lock on the current object.

- Recommended reading:

MSDN (

<http://msdn2.microsoft.com/en-au/library/system.threading.monitor.pulse.aspx>

)