CS 2530

Midterm Prep

WPF

WPF

- WPF is a presentation framework for building Windows client applications

WPF

- WPF is a presentation framework for building Windows client applications
- Core of WPF is a resolution-independent and vector-based rendering engine

WPF

- WPF is a presentation framework for building Windows client applications
- Core of WPF is a resolution-independent and vector-based rendering engine
- Application-development features:
  XAML, controls, styles, templates, data binding, . .

WPF

- WPF is a presentation framework for building Windows client applications
- Core of WPF is a resolution-independent and vector-based rendering engine
- Application-development features:
  XAML, controls, styles, templates, data binding, . .
- Supports strong separation of content and presentation => software developers and graphical designers can work simultaneously

WPF: Creating Layout

- Layout provides ordered way to place UI elements
- Manages size and position of those UI elements when UI is resized
- Layout controls:
  - Canvas
  - StackPanel
  - WrapPanel
  - DockPanel
  - Grid

SPEED THINK - TEST ROUND

Pair up - **stand up**

You have 1 min to brain storm everything that you know

about . . . .

Count as you go.

How many things could you come up with?

SPEED THINK - TEST ROUND

Pair up - **stand up**

You have 1 min to brain storm everything that you know

about **Event Handling**

Count as you go.

How many things could you come up with?

I/O

using

using directive

Allows the use of types in a namespace without having to fully qualify it

using statement

Provides a convenient syntax that ensures the correct use of IDisposable objects.

TODO:

Will the method Dispose be called on the variable reader?

```
using (StreamReader reader =
      new StreamReader(dialog.OpenFile()))
{
     ContentTb.Text = reader.ReadToEnd();
     throw new Exception();
}
```

TODO:

Will the method Dispose be called on the variable reader?

```
using (StreamReader reader =
      new StreamReader(dialog.OpenFile()))
{
     ContentTb.Text = reader.ReadToEnd();
     throw new Exception();
}
```

Y E S

TODO:

- *(IDisposable, IEnumerable, ISerializable)*

  The using statement provides a convenient syntax that ensures the correct use of

  _____ objects.

TODO:

- (ensures, can't ensure)

  The using statement _____

  that Dispose is called when an exception occurs

TODO:

- *(class, method, namespace, project)*

  The using directive allows you to use types in a

  _____ without fully

  qualifying them

TODO:

Take 1 min to discuss:

Event handling in Java vs C#

Serialization

To serialize an object you need 3 things:

1. The object to be serialized
2. A stream to contain the serialized object
3. A formatter

Apply the **Serializable** attribute to a type to indicate that instances of this type can be serialized

Serialization

- Serialization is the process of converting an object into a serial format (e.g. stream of bytes or XML) so it can be stored or transmitted.
- 2 main purposes:

  To save the state of an object in order to be able to recreate it when needed.

  To transport the state of an object over a network
- The reverse process is called deserialization.

TODO:

Take 1 min to discuss:

Binary vs XML Serialization

Type (formatter)	Binary Formatter	XmlSerializer class
Best performance	**Yes**	No
Readable with other plataforms (non .NET)	**No**	Yes
Serialize object private members	Yes (deep serialization)	**No** (shallow serialization)
Serialize generic collections (*)	Yes	Yes
Easy control of how each member is serialized	No (**)	**Yes** (by using attributes)
Saves metadata in output stream for deserialization	Yes	No

(**) … possible but not easy

File I/O

- StreamReader
- StreamWriter

File I/O

- StreamReader .. TextReader
- StreamWriter .. TextWriter

FileStream vs StreamWriter

- FileStream is a Stream

  Streams only deal with byte[] data

- SteamWriter is a TextWriter

  It converts text (string or char) to byte[] data for the underlying stream

ad Files TODO:

An example of when a file should * *NOT* * be used is:

a) to save the status of a video game for another session.

b) to save the value of a local variable at runtime.

c) to log errors during the execution of a program.

d) these are all good examples of when files should be used.

TODO:

- Which of the following is NOT required for serialization?

a) Formatter

b) Network Connection

c) Object

d) Stream

DELEGATES

Delegates

- A delegate is a type that defines a method signature.
- When you create an instance of a delegate, you can associate it with any method that matches its signature.
- You can invoke (or call) the method through the delegate instance.

Delegates

- Conceptually like function templates - they express a contract a function must adhere to in order to be considered of the 'type' of the delegate.
- Similar to function pointers but type-safe
- Once a delegate is assigned a method, it behaves exactly like that method.
- Derive from System.Delegate (abstract class)

Generic delegates

- Used to pass methods as a parameter
- `Action<T>` delegate

  Return type: void; One parameter of type T

  Has overloaded versions for multiple parameters
- `Func<T, TResult>` delegate

  Return type: TResult; one parameter of type T

  Has overloaded versions for multiple parameters

  Return type is always last parameter

Example

```
Action<int, int>

void PrintProduct (int x1, int x2)

void RingBell (int pitchInHz, int durationInSeconds)
```

Example

```
Action<int, int>

void PrintProduct (int x1, int x2)

void RingBell (int pitchInHz, int durationInSeconds)

Func<int, double>

double CalculateOneSeventh(int number)
```

Example

```
Action<int, int>

void PrintProduct (int x1, int x2)

void RingBell (int pitchInHz, int durationInSeconds)

Func<int, double>

double CalculateOneSeventh(int number)

Func<int, double, bool>

bool Ran100Miles (int numberOfTrips, doulbe distance);
```

Example

```
Action<int, int>

void PrintProduct (int x1, int x2)

void RingBell (int pitchInHz, int durationInSeconds)

Func<int, double>

double CalculateOneSeventh(int number)

Func<int, double, bool>

bool Ran100Miles (int numberOfTrips, doulbe distance);
```

EXERCISE A

Lambda Expressions

- They are anonymous functions
- Can be associated with delegates
- Lambda operator => reads as “goes to”

Expression Lambdas

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

- Returns the result of an expression
- Examples:

```
(x, y) => x == y
(int x, string s) => s.Length > x
() => SomeMethod()
x => x * x
```

Statement Lambdas

```
(input parameters) => { statement; }
```

- Can include multiple statements; typically <= 3
- Examples:

```
(i1, i2) => { return i1 + i2;}
(int x) => {
      x++;
      Console.Writeline("x ..{0}", x);
}
```

EXERCISE B

if time: EXERCISE C

ITERATOR

Iterator Design Pattern

- Encapsulated in the IEnumerable and IEnumerator interfaces and their generic counterparts.
- Basic idea:

  As a data consumer, you can ask an IEnumerable for an IEnumerator by calling the GetEnumerator() method. Once you have an IEnumerator you can iterate over the content using the IEnumerator members Reset, Current, and MoveNext

Iterator

- An iterator is a section of code that returns an ordered sequence of values of the same type
- Iterator use the `yield return` statement to return each element in turn
- Return type must be one of these 4 types:

```
IEnumerable        IEnumerable<T>

IEnuemrator        IEnumerator<T>
```

Iterator named GetEnumerator

- Most common way to create an iterator
- Foreach can iterate directly on the class instance

```
public IEnumerator GetEnumerator()
 {
      for (int i = 0; i < 10; i++)
    {
          yield return i;
    }
 }
```

Iterator named GetEnumerator

- Most common way to create an iterator
- Foreach can iterate directly on the class instance

```
public IEnumerator GetEnumerator()
 {
     for (int i = 0; i < 10; i++)
    {
         yield return i;
    }
 }
```

Returns IEnumerator without explictily implementing the interface (compiler does that)

Iterator named GetEnumerator

- Most common way to create an iterator
- Foreach can iterate directly on the class instance

Can be called implicitly

```
public IEnumerator GetEnumerator()
 {
     for (int i = 0; i < 10; i++)
    {
         yield return i;
    }
 }
```

Returns IEnumerator without explictily implementing the interface (compiler does that)

yield

- When `yield return` statement is reached, the current location is stored
- Execution is started from that location the next time the iterator is called
- `yield break` ends the iteration
- There can be multiple `yield return` statements in an iterator

- FYI: The compiler translates the iterator into a nested class that is, in effect, a state machine

LINQ

LINQ

- **L**anguage **IN**tegrated **Q**uery
- Enables you to query and manipulate data independent of data sources
- Allows native data querying in C# (and VB)

Why LINQ

- Single Query Language for multiple data sources
- Compile-Time Name and Type Checking
- Wide range of operators provided
- Concise and clear way to express a query
- Declarative approach makes queries easier to understand

Query Syntax

- Declarative - similar to SQL
- For many easier to read

```
IEnumerable<int> sortedEvenNumbers =
   from num in numbers
   where num % 2 == 0
   orderby num
   select num;
```

Method Syntax

- Calls Standard Query Operators directly on source collections
- More powerful. Some query operators can only be called in method syntax

Example:

```
IEnumerable<int> numQuery2 =
    (from num in Numbers
     where (num % 2 == 0)
     select num).Distinct();
```

Mixed Syntax

Method Syntax

- Calls Standard Query Operators directly on source collections
- More powerful. Some query operators can only be called in method syntax

Example:

```
IEnumerable<int> numQuery2 =
    Numbers.Where(num => num % 2 == 0).Distinct();
```

Method Syntax

Query / Method Syntax

- No semantic or performance difference between query syntax and method syntax

Standard Query Operators

- Set of methods declared in the static class System.Query.Sequence
- Most Standard Query Operators are extension methods extending IEnumerable<T>
- API that enables querying of any .NET array or collection, that implements IEnumerable<T>.

Exercise E : Mark the following language features:

Anonymous Type, Extension Method, Lambda Expression, Object Initializer, T
ype Inference,

```
e.g.  // Query Syntax
var result = from s in students
             where s.DateOfBirth.Year > 1989
             select new { s.Name, s.DateOfBirth };

e.g.  // OO or Method Syntax
var result = students
            .Where(s => s.DateOfBirth.Year > 1989)
            .Select(s => new {s.Name, s.DateOfBirth});
```

[EXERCISE E]

LINQ - Language Features

e.g.

```
var result = from s in students
             where s.DateOfBirth.Year > 1989
             select new {s.Name, s.DateOfBirth};
```

Type Inference

Anonymnous type

Object Initializer

e.g.

```
var result = students
             .Where(s => s.DateOfBirth.Year > 1989)
             .Select(s => new {s.Name, s.DateOfBirth});
```

Lambda Expression

Extension methods

Delegates	• Anonymous functions
Lambda Expressions	• Types that define a method signature
Iterator	• Set of methods that enables querying of any .NET array or collection, that implements IEnumerable<T>.
Standard Query Operators	• Section of code that returns an ordered sequence of values of the same type