C# Advanced Course

Generics

- Allows for code reusability without performance penalties brought on by casting object types
- Most likely, you will use existing generics, *not* creating them

```
public class GenericList<T>
{
  public void Add(T value)
  {

  }

  public T this[int index]
  {
    get { ... }
  }
}
```

Constraints

```
 public int Max(int a, int b)
{
    return a > b ? a : b;
}

// OR

public T Max<T>(T a, T b) where T : IComparable
{
    return a.CompareTo(b) > 0 ? a : b;
}
```

- Can be applied to class or method:
  - `where T : IComparable`
  - `where T : Product`
  - `where T : struct`
  - `where T : class`
  - `where T : new()`

Dictionaries

- Use a hash table to store and retrieve objects (great performance advantages)
- Key / value pairs

Delegates

- An object that knows how to call a method (*or group of methods*)
- A reference to a function

Why use delegates?

- For designing extensible and flexible applications (e.g. frameworks)

- `public delegate void PhotoFilterHandler(Photo photo);`

  - This delegate can handle methods with a `void` signature that take `Photo` as a parameter

Existing Delegates

- `System.Action<>`
- `System.Func<>`

Interfaces or Delegates?

- Use a delegate when:
  - An eventing design pattern is used
  - The caller doesn't need to access the other properties or methods on the object implementing the method

Lambda Expression

- An anonymous method
- No access modifier
- No name
- No return statement
- Similar to `arrow functions` in JS
- Has access to all arguments passed, as well as all properties within class expression is

called in

Why?

- Convenience
- `[args] => [expression]`
- `x => ...`
- `() => ...`
- `(x, y, z) => ...`

```
func<int, int> square = number => number*number;
square(5; // 25
```

Events and Delegates

Events

- A mechanism for communicating between objects
- Used in building *Loosely Coupled Applications*
- Helps extend applications
- Allows for publishers of event to alert subscribers of event when necessary

Delegates

- Agreement / Contract between **Publisher** and **Subscriber**
- Determines the signature of the event handler method in **Subscriber**

1. Define a delegate
2. Define an event based on that delegate
3. Raise the event

```
class Program
{
  static void Main(string[] args)
  {
    var video = new Video() { Title = "Video 1"}
    var videoEncoder = new VideoEncoder(); // Publisher
    var mailService = new MailService(); // Subscriber

    videoEncoder.VideoEncoded += mailService.OnVideoEncoded;

    videoEncoder.Encode(video);
  }
}
```

```
public class VideoEncoder
{
  public delegate void VideoEncodedEventHandler(object source, EventArgs args);

  public event VideoEncodedEventHandler VideoEncoded;

  public void Encode(Video video)
  {
    Console.WriteLine("Encoding Video...")
    Thread.Sleep(3000); // Simulates encoding implementation...

    OnVideoEncoded();
  }

  protected virtual void OnVideoEncoded()
  {
    if (VideoEncoded != null)
      VideoEncoded(this, EventArgs.Empty);
  }
}
```

- Can use `public event EventHandler VideoEncoded` without needing `public delegate void VideoEncodedEventHandler(object source, EventArgs args);`
- `OnVideoEncoded()`
  - Fires `VideoEncoded` event handler

```
public class MailService
{
  public void OnVideoEncoded(object source, EventArgs e)
  {
    Console.WriteLine("MailService: Sending an email..."); // Simulates MailService
  }
}
```

Extension Methods

- Allow for adding methods to an existing class without:
    - Changing its source code
    - Creating a new class that inherits from it

```
public static class StringExtensions
{
  public static string Shorten(this String str, int numberOfWords)
  {

  }
}
```

```
var str = "This is an exceptionally long string ripe with meandering prose, ultimat
            var shortened = str.Shorten(5);
```

LINQ

- **L** anguage **In** tegrated **Q** uery
- Grants capability to query objects natively
- Can query…
    - Objects in memory (*LINQ to Objects*)
    - Databases (*LINQ to entities*)
    - XML (*LINQ to XML*)
    - ADO.NET Data Sets (*LINQ to Data Sets*)

LINQ Extension Methods

- `Select()`
- `Where()`
- `Take()` // used for pagination
- `Skip()` // used for pagination
- `OrderBy()`
- `Single()`
- `SingleOrDefault()`
- `First()`
- `FirstOrDefault()`
- `Last()`
- `LastOrDefault()`

- `Count()`
- `Max()`
- `Min()`
- `Sum()`

```
var cheapBooks = books
                        .Where(b => b.Price < 10)
                        .OrderBy(b => b.Title)
                        .Select(b => b.Title);
```

LINQ Query Operators

```
var cheapBooks =
    from b in books
    where b.Price < 10
    orderby b.Title
    select b.Title;
```

Nullable Types

- Value types cannot be null
- `DateTime? date = null;`

Null Coalescing Operator

- `DateTime date2 = date ?? DateTime.Today;`
  - If date is null, set date2 to today
  - Else, set date2 to date

Dynamic

- Static Languages: *C#, Java*
  - Interpreted at compile-time
- Dynamic Languages: *Ruby, Javascript, Python*
  - Interpreted at runtime

```
dynamic name = "Miller";
name = 10;
```

- **Dynamics** allow for implicit conversion and casting to target variable

Exception Handling

- **Stack Trace** - Sequence of methods called until exception is thrown
- In .NET namespace, an exception is essentially a class
- Nest catch blocks from most specific to most generic:
  `try { var calculator = new Calculator(); var result = calculator.Divide(5, 0); } catch (DivideByZeroException ex) { ... } catch (ArithmeticException ex) { ... } catch (Exception ex) { ... }`

Finally

- Use a `finally` block to handle resources that are not managed by the **CLR** (aka, *On Manage Resources*)
  - E.G. Database connections, streamReaders
    `finally { streamReader.Dispose(); }`
  - Can be done with the `using` statement:
    `try { using (var streamReader = new StreamReader(@"c:\file.zip")) { var content = streamReader.ReadToEnd(); } }`
  - Automatically invokes `finally` as soon as using block ends

Custom Exception Handling

```
public class YoutubeException : Exception
{
  public YoutubeException(string message, Exception innerException)
    : base(message, innerException)
  {
  }
}
```

Asynchronous Programming

Synchronous Program Execution

- Program is executed line by line, one at a time
- When a function is called, program execution has to wait until the function returns

Async Program Execution

- When a function is called, program execution continues to the next line *without* waiting for the function to complete

When to use Async?

- Accessing the web
- Working with files and Databases
- Working with images

```
// ASYNCHRONOUS
public async Task DownloadHtmlAsync(string url)
{
    var webClient = new WebClient();
    var html = await webClient.DownloadStringTaskAsync(url);

    using (var streamWriter = new StreamWriter(@"c:\projects\result.html"))
    {
      await streamWriter.WriteAsync(html);
    }
}

// SYNCHRONOUS
public void DownloadHtml(string url)
{
  var webClient = new WebClient();
  var html = webClient.DownloadStringTask(url);

  using (var streamWriter = new StreamWriter(@"c:\projects\result.html"))
  {
    streamWriter.Write(html);
  }
}
```

Await

- Signifies that the rest of an async method or lambda cannot continue execution until await operation is completed
- Immediately passes control back to outer context