Useful Features and Methods

Null Handling

- Standard if statements are very useful, but there are also several other tools.
- For strings, you can test both for null and blank strings with one method.

```
// handling general reference-types
String test = null;

if (test != null) { }

if (String.IsNullOrEmpty(test)) { }
if (String.IsNullOrWhiteSpace(test)) { }

var text = test ?? "default value if null";


// handling nullable types
int? nullableValue = null;

if (nullableValue.HasValue) { }
var result = nullableValue.GetValueOrDefault();

List<int> list = null;
var x = list?[0];
var y = list?.First();
```

Default Values

- All reference types default to 'null'.
- Others can be checked with default().

```
var defaultValue = default(T);
defaultValue.WriteLine();

default(int).WriteLine(); // 0
default(char).WriteLine(); // ''
default(double).WriteLine(); // 0
default(bool).WriteLine(); // False

// null
//default(string).WriteLine();
//default(object).WriteLine();
//default(SimpleClass).WriteLine();
```

String Comparison

- The default equality test doesn't work well for reference types like strings – use the .Equals() method instead.

```
public void stringCompare()
{
    String s1 = "ABCD";
    String s2 = "abcd";

    bool result = s1 == s2; // Don't use this!
    result = s1.Equals(s2); // false
    result = s1.Equals(s2, StringComparison.OrdinalIgnoreCase); // true

}
```

Parsing and Casting

- Often we need to treat a variable that has type A as though it was actually type B.
- Some conversions happen automatically – smaller numeric types can always be converted to larger ones, and derived classes can always be converted to the base class.
- If the compiler can't be sure that the conversion is definitely safe, you need to give an explicit instruction.

Convert methods

- If you have a variable representing a value type, you can use the Convert class's static methods to parse it or the Parse method on the type you want.
  - Particularly useful for parsing user input or file data.

```
string value = "123";
int i = int.Parse(value);
int i2 = Convert.ToInt32(value);

double d = Convert.ToDouble(value);

string datevalue = "2018-01-01";
DateTime date = Convert.ToDateTime(datevalue);
```

Explicit Casts

- You can also instruct the compiler to 'cast' a variable to another type.
  - Casting with (type) will throw an exception if the cast is invalid; using the as keyword will just return null.

```
double d = 10.0;
float f = (float) d;

BaseClass obj = new DerivedClass();
DerivedClass obj2 = (DerivedClass)obj;
DerivedClass obj3 = obj as DerivedClass;
```

Using 'var'

- The var keyword can be used to tell the compiler to infer a variable's type automatically.
  - Variables are still strongly-typed!

```
// All of these pairs are equivalent
int i = 123;
var i2 = 123;

double d = 9.9;
var d2 = 9.9;

List<String> list = new List<string>();
var list2 = new List<string>();
```

Typeof

- To check the type of a variable at run-time, you can use .GetType() or the typeof keyword.

```
int i = 10;
var intType = i.GetType(); // System.Int32

double d = 10.1;
var doubleType = d.GetType(); // System.Double

var stringType = typeof(string); // System.String
```

Params as a parameter

- A method can take a variable number of inputs of the same type by using the params keyword.
  - All the values will be stored in an array and that array will be passed to the method.

```
public void usingParams(params int[] values)
{
}

public void main()
{
    usingParams(1, 2, 3, 4, 5);
}
```

String Concatenation

- Use the String.Format method to concatenate strings.
  - Null values will show an empty string.
  - Using the manual concatenation will throw an exception on null values.

```
string name = null;
int age = 50;
bool isHealthy = true;

// Concatenate with String.Format: safe and easier to read.
var formated = String.Format("Name: {0}, age: {1}, healthy: {2}", name, age, isHealthy);
Console.WriteLine(formated);

// Manual concatenation: can throw errors
var manual = "Name: " + name + ", age: " + age + ", healthy: " + isHealthy;
Console.WriteLine(manual);
```

Stringbuilder

- If you need to dynamically build large strings, use StringBuilder.
  - It has several internal optimisations over using regular strings.

```
StringBuilder builder = new StringBuilder();

builder.Append("Hello");
builder.AppendLine(" World");
builder.AppendFormat("{0} is {1}", "C#", "fun!");

// Hello World
// C# is fun!
```

Verbatim String Literals

- You can simplify strings that would need escape characters by placing @ in front of the string.
  - A double-quote character still needs to be escaped: double them.

```
var s1 = "C:\\scripts\\javascript\\file.js";
var s2 = @"C:\scripts\javascript\file.js";

var s3 = "But \"Quotes\" still need to be escaped...";
var s4 = @"But ""Quotes"" still need to be escaped...";
```

Object Initializers

- You can initialize properties of an object as you create it, rather than afterwards.
  - This is atomic – the object is never half-initialized; it's either null or completed.
  - Can be used for anonymous types.

```
var bob = new Person
{
    Name = "Bob",
    DOB = new DateTime(2018, 1, 1)
};


var bill = new Person();
bill.Name = "Bill";
bill.DOB = new DateTime(2018, 1, 2);
```

Printing Arrays

- Printing arrays in C# requires an extra step – you need to manually create a string containing the result you want.
- Use String.Join(separator, data);

```
int[] data = { 1, 2, 3, 4, 5 };
Console.WriteLine(String.Join(", ", data));
// 1, 2, 3, 4, 5
```

Try/Catch/Finally

- If you have code that might throw an exception, place it inside a try block.
  - All try blocks must be followed by either a catch block or a finally block (or both).
  - Catch is for exception-handling only; if the try block does not throw an exception, catch does not run.
  - Finally is for clean-up; it runs regardless of exceptions.

Try/Catch/Finally

```
try
{
    // risky code here...
}
catch (Exception e)
{
    // error-handling here...
}
finally
{
    // cleanup here...
}
```

Exception type	Base type	Description	Example
Exception	Object	Base class for all exceptions.	None (use a derived class of this exception).
IndexOutOfRangeException	Exception	Thrown by the runtime only when an array is indexed improperly.	Indexing an array outside its valid range: `arr[arr.Length+1]`
NullReferenceException	Exception	Thrown by the runtime only when a null object is referenced.	`object o = null; o.ToString();`
InvalidOperationException	Exception	Thrown by methods when in an invalid state.	Calling `Enumerator.GetNext()` after removing an Item from the underlying collection.
ArgumentException	Exception	Base class for all argument exceptions.	None (use a derived class of this exception).
ArgumentNullException	Exception	Thrown by methods that do not allow an argument to be null.	`String s = null;` `"Calculate".IndexOf (s);`
ArgumentOutOfRangeException	Exception	Thrown by methods that verify that arguments are in a given range.	`String s = "string";` `s.Substring(s.Length+1);`

Using keywords as variable names

- If you prefix a C# keyword with the @ symbol, you can use it as a variable name.
  - Only do this with a very, ***very*** good reason!

```
var @return = "A string variable named return";

var @int = "Another string";
```

Enums

- Enums are a list of named numeric constants.

```
public enum XeroStatus
{
    Draft = 54,
    Submitted = 56,
    Authorised = 57,
    Deleted = 59,
    Voided = 60,
    Confirmed = 1093,
    Error = 1095,
    Paid = 1096
}
```

Generic Collections

- Classes and interfaces from the System.Collections.Generic library.
- These classes are strongly-typed: a list of strings rather than a list of objects.
- IEnumerable<T>
  - A generic interface for any collection that can be iterated.
- ICollection<T>
  - Allows modification of the values and counting the contents.
- IList<T>
  - Allows access by index as well as the above.

.NET Libraries

- There are many libraries available for .NET, both built-in and 3rd-party.
- It usually makes no sense to write your own List class, file access tool, etc.
  - Don't reinvent the wheel!
  - The first version *always* has problems – libraries have already solved them.

Important .NET libraries

- System
- System.Collections.Generic
- System.IO
- System.Threading
- System.Drawing
- System.Web
- System.XML
- And many more...

TFS

- TFS is an integrated Version Control System in Visual Studio.
- It includes many tools to help teams to collaborate:
  - Merge changes from multiple devs
  - Rollback changes
  - Track who made certain changes and when
  - Handle build tasks