**Advance API Notes**

**Debugging and Breakpoints in Visual Studio**

**1. Setting Up Debugging in Visual Studio**

To begin debugging in Visual Studio, you need to have a project open. Ensure that you have a **Debug Configuration** set up for your project (instead of a **Release** configuration). This enables debugging symbols, which are essential for stepping through your code effectively.

**Steps to Start Debugging:**

1. Open Visual Studio and load your project.
2. Select **Debug** from the toolbar.
3. Set breakpoints (explained below) to mark where execution should stop.
4. Press **F5** or choose **Debug > Start Debugging**.

**2. Using Breakpoints**

Breakpoints are markers set within the code to instruct the debugger to pause execution at that specific location. This allows you to inspect the program's state.

**Setting Breakpoints**

1. In the **Code Editor**, locate the line where you want to pause execution.
2. Click on the **left margin** next to the line of code (or press **F9**) to set a breakpoint.
   * A **red dot** will appear, indicating the breakpoint.

**Conditional Breakpoints**

Conditional breakpoints allow the debugger to pause only when a specified condition is met.

1. Right-click on a breakpoint (red dot).
2. Choose **Condition**.
3. In the dialog that appears, enter a condition (e.g., x == 10).
4. The breakpoint will now only trigger when the condition is true.

**Hit Count Breakpoints**

You can configure breakpoints to trigger only after they have been hit a specific number of times.

1. Right-click the breakpoint.
2. Select **Hit Count**.
3. Specify the number of times the breakpoint must be hit before it triggers.

**Data Breakpoints**

Data breakpoints are useful for pausing the execution when a specific variable's value changes.

1. Set a breakpoint on the line where the variable is being modified.
2. Right-click the breakpoint and select **Conditions** > **When value of variable changes**.

**3. Running the Debugger**

**Starting Debugging**

To start debugging:

1. Press **F5** or go to the menu and select **Debug > Start Debugging**.
   * The debugger will launch, and the code will run until it hits a breakpoint.

**Pausing the Debugger**

If the program is running, you can pause execution at any time.

* Click the **Pause** button on the toolbar, or press **Ctrl + Alt + Break**.

**4. Stepping Through Code**

**Step Over (F10)**

Step Over allows you to execute the current line of code without stepping into any method calls it may contain.

1. Press **F10** to step over the current line.

**Step Into (F11)**

Step Into allows you to go into the method or function call on the current line.

1. Press **F11** to step into the method.

**Step Out (Shift + F11)**

Step Out is used when you're inside a function and want to finish execution of the current function and return to the calling function.

1. Press **Shift + F11** to step out of the current function.

**5. Inspecting Code During Debugging**

**Watch Window**

The Watch Window allows you to monitor the values of specific variables or expressions during debugging.

1. To add a variable to the Watch Window, right-click on the variable in the code and select **Add Watch**.

**Locals Window**

The Locals Window displays local variables in the current scope.

1. To view this, go to **Debug > Windows > Locals**.

**Autos Window**

The Autos Window shows variables that are relevant to the current statement.

1. Open the Autos Window from **Debug > Windows > Autos**.

**6. Common Debugging Features**

**Call Stack Window**

The Call Stack Window shows the stack of function calls that led to the current breakpoint.

1. Open it by going to **Debug > Windows > Call Stack**.

**Immediate Window**

The Immediate Window allows you to evaluate expressions and execute commands while debugging.

1. Open it from **Debug > Windows > Immediate**.

**7. Stopping and Detaching from Debugging**

**Stopping Debugging**

To stop debugging:

1. Press **Shift + F5**, or click the **Stop Debugging** button in the toolbar.

**Detaching Debugger**

If you wish to detach from the program without stopping it:

1. Go to **Debug > Detach All**.

**Conditional Compilation**

Conditional compilation in C# allows developers to include or exclude code from the compilation process based on specific conditions. Only works at compile-time, not run-time.

C# uses preprocessor directives to control which parts of code are included in the compilation process. The primary directives for conditional compilation are:

* **#if**: Starts a conditional block based on whether a symbol is defined.
* **#else**: Provides an alternative block if the condition in #if is not met.
* **#elif**: Specifies an additional condition to check in the same block.
* **#endif**: Marks the end of the conditional block.

**Types of Classes in C#**

**1. Abstract Class**

* **Purpose**: Defines a base class that cannot be instantiated directly.
* **Usage**: Used to provide a common interface and shared functionality for derived classes.
* **Key Feature**: Can contain abstract methods (without implementation) that must be overridden in derived classes.

**2. Sealed Class**

* **Purpose**: Prevents a class from being inherited.
* **Usage**: Used when you want to lock the class design and ensure it cannot be extended further.
* **Key Feature**: No class can inherit from a sealed class.

**3. Static Class**

* **Purpose**: Contains only static members and cannot be instantiated.
* **Usage**: Used for utility or helper classes that provide methods or properties, such as Math or Console.
* **Key Feature**: All members must be static; cannot have instance members.

**4. Partial Class**

* **Purpose**: Allows a class to be split into multiple files.
* **Usage**: Commonly used for large classes or when auto-generated code needs to be extended without modifying the original file.
* **Key Feature**: All parts of a partial class must be defined with the partial keyword and within the same assembly or namespace.

**Generics**

**1. Generic Class**

A **generic class** is a class that can operate with any data type. The type is specified when the class is instantiated, enabling the same class definition to work with multiple types.

**Key Features:**

* A generic class can handle various data types, providing flexibility in your code.
* Type parameters are placeholders for actual data types and are defined at the class level.
* You can use the type parameter throughout the class in fields, methods, and properties.

**Constraints on Generic Classes:**

You can apply constraints to restrict which types can be used with the generic class. For example, you can specify that the type must be a reference type or implement a certain interface.

**2. Generic Methods**

A **generic method** allows you to define methods that work with any data type. The type parameter is defined at the method level, so each method can have its own type parameter, independent of the class it's in.

**Key Features:**

* A generic method allows you to define methods that can work with any data type, without needing to create multiple overloads.
* The type parameter is defined when calling the method, making it versatile and reusable.

**Constraints on Generic Methods:**

Like with generic classes, you can place constraints on the types that can be used with generic methods. For example, you can require that a type implements a certain interface, ensuring it supports specific operations.

**3. Generic Interfaces**

A **generic interface** defines an interface that works with any data type. Implementing classes or structs specify the type when they implement the interface, ensuring that they work with the correct data type.

**Key Features:**

* A generic interface allows for defining methods or properties that can work with any data type.
* The type parameter is specified when the interface is implemented, making it flexible for various implementations.

**Constraints on Generic Interfaces:**

Just like with generic classes and methods, you can apply constraints to the type parameter of a generic interface. This ensures that only types meeting certain criteria (e.g., implementing a specific interface or being a certain type) can implement the interface.

**FileSystem**

C# provides several classes for handling files and directories. These classes are part of the System.IO namespace and enable you to interact with the file system, read and write files, and manipulate paths and directories.

**1. FileInfo**

**FileInfo** is a class that provides methods and properties to work with file system files. It offers more detailed information about a file than the static File class and allows you to perform operations on the file like copying, deleting, and moving.

Methods and Properties:

Exists, FullName, IsReadOnly, Name, Extension, CreationTime, Directory, DirectoryName, Length, MoveTo(filepath), Copyto(destinationPath), Delete()

**2. StreamReader**

**StreamReader** is a class designed for reading characters from a byte stream, primarily used to read text from files. It is ideal for reading line-by-line from text files or large data streams.

Methods and Properties

ReadLine(), ReadToEnd(), EndOfStream

**3. StreamWriter**

**StreamWriter** is a class used to write characters to a byte stream, typically for creating or appending to text files. It is an efficient way to write text data to files.

Methods and Properties

Write(), WriteLine()

**4. Path**

**Path** is a utility class that provides static methods for working with file and directory path strings. It allows you to perform operations such as combining paths, getting file extensions, and extracting the file name.

Methods and Properties

Combine(baseDir, subDirs), GetFileName(), GetExtension(), GetDirectoryName, GetFullPath()

**5. Directory**

**Directory** is a class that provides methods for creating, deleting, and manipulating directories. It is useful for working with directories and subdirectories.

Methods and Properties

Exists(), CreateDirectory(), GetFiles(), SetCurrentDirectory(), GetDirectories(), Delete()

**JSON Serialization and Deserialization**

**Newtonsoft JSON Serialization**

Newtonsoft.Json is a popular, high-performance library for working with JSON in C#. It provides easy-to-use methods for serializing objects to JSON and deserializing JSON back into objects.

Key Features:

* **Flexible**: Supports complex types and custom serialization.
* **High Performance**: Optimized for fast serialization and deserialization.
* **Customizable**: Supports custom converters, settings, and formatting.
* **Supports LINQ**: Allows querying and manipulating JSON data using LINQ.

Common Operations:

1. **Serialization**: Converting C# objects to JSON format (JsonConvert.SerializeObject()).
2. **Deserialization**: Converting JSON strings back into C# objects (JsonConvert.DeserializeObject<model>()).

**XML Serialization and Deserialization**

C# provides built-in support for **XML serialization** (XmlSerializer) using the System.Xml.Serialization namespace. This allows you to convert objects to XML format and back.

Key Features:

* **Built-In Support**: Integrated into the .NET Framework, no third-party libraries are required.
* **Supports Simple Types**: Works out-of-the-box for most simple types and common objects.
* **Customizable**: Allows attributes to control serialization behavior.

Common Operations:

1. **Serialization**: Converting objects into XML.
2. **Deserialization**: Converting XML back into objects.

**Lambdas**

A lambdaexpression is a shorthand way of defining an anonymous method. It allows you to define a function inline, without needing to explicitly define a method. A lambda expression can take parameters and return a value or perform an action.

* Func Delegates

Func<T> is a built-in delegate type in C# that represents a method that can take parameters and return a value.

* Action Delegates

Action<T> is another built-in delegate type in C# that represents a method that performs an action but does not return a value.

**Extension Methods**

To create an extension method, follow these steps:

1. Create a static class.
2. Define a static method within the class.
3. Use the this keyword as the first parameter to indicate which type is being extended.

To use the extended methods:

* Include the namespace where the extension methods are defined.
* Call the extension method using the instance of the type you extended.