## **Conditional Compilation**

Conditional compilation provides a mechanism to selectively include or exclude sections of source code during the compilation process. This is achieved through **preprocessor directives**, which are special instructions to the compiler that begin with the # symbol and must appear on their own line. Although they are processed during the lexical parsing phase, conceptually, they operate *before* the main compilation.

### **Directives for Conditional Compilation**

The primary preprocessor directives for conditional compilation are #if, #else, #endif, and #elif.

* **#if Directive:** This directive instructs the compiler to include the subsequent block of code *only if* a specified symbol has been defined. If the symbol is not defined, the code block is entirely ignored by the compiler, effectively as if it were commented out.

| #define TESTMODE // Defines the symbol TESTMODE for this file. Must be at the top.  using System;  class Program {  static void Main()  { #if TESTMODE  Console.WriteLine("in test mode!"); // This line compiles if TESTMODE is defined #endif  } } |
| --- |

* If the #define TESTMODE line were removed, the Console.WriteLine statement would be completely eliminated from the compiled executable.
* **#else and #elif Directives:**
  + The #else directive functions similarly to C#'s else statement, providing an alternative code block to compile if the preceding #if (or #elif) condition is false.
  + The #elif directive is analogous to else if, allowing you to test additional conditions in sequence.
* **Logical Operators:** You can combine symbols using standard logical operators: || (OR), && (AND), and ! (NOT).

| #if TESTMODE && !PLAYMODE // Compiles if TESTMODE is defined AND PLAYMODE is NOT defined // ... code for test mode only, when not in play mode ... #elif DEVMODE || STAGINGMODE // Compiles if DEVMODE OR STAGINGMODE is defined // ... code for development or staging ... #else // ... default code ... #endif |
| --- |

* It is important to remember that these are preprocessor symbols, entirely separate from runtime variables. They are evaluated strictly at compile time.

### **Defining Symbols**

Symbols can be defined in two primary ways:

1. **In Source Code (#define):** Using the #define directive directly within a source file (as shown above). When defined this way, the symbol's scope is limited to that specific file. These directives must appear at the top of the file, before any actual code.
2. **In the Project File (.csproj):** For symbols that should apply to an entire assembly (all files in a project), you can define them in the .csproj file using the <DefineConstants> element within a <PropertyGroup>:

| <PropertyGroup>  <DefineConstants>TESTMODE;PLAYMODE</DefineConstants> </PropertyGroup> |
| --- |

1. Multiple symbols are separated by semicolons. You can also manage these via Visual Studio's Project Properties (Build tab). If a symbol is defined at the assembly level, you can "undefine" it for a specific file using the #undef directive within that file.

### **Conditional Compilation Versus Static Variable Flags**

One might consider using a static boolean variable flag for similar conditional logic:

| static internal bool TestMode = true; static void Main() {  if (TestMode) Console.WriteLine("in test mode!"); } |
| --- |

This approach offers runtime configurability. However, conditional compilation provides capabilities that variable flags cannot, as it operates at the compile-time level:

* **Conditional Inclusion of Attributes:** You can conditionally include or exclude attributes based on symbols.
* **Changing Declared Types:** You can alter the declared type of a variable.
* **Switching Namespaces/Type Aliases:** You can switch between different using directives (namespaces or type aliases).

| using TestType = #if V2  MyCompany.Widgets.GadgetV2; #else  MyCompany.Widgets.Gadget; #endif |
| --- |

* **Major Refactoring:** It enables parallel development or instant switching between old and new versions of code, facilitating significant refactoring.
* **Multi-Runtime Version Libraries:** Allows libraries to compile against different runtime versions, leveraging newer features when available.
* **Debugging Code Inclusion:** Debugging-specific code can refer to types in assemblies that are not included in the final deployment.

### **The [Conditional] Attribute**

The [Conditional] attribute, found in the System.Diagnostics namespace, offers a more elegant way to conditionally include method calls. It instructs the compiler to ignore any calls to a method if the specified symbol has not been defined.

Consider a logging method:

| static void LogStatus(string msg) {  string logFilePath = "log.txt"; // simplified  System.IO.File.AppendAllText(logFilePath, msg + "\r\n"); } |
| --- |

**Problem with manual #if:**

Wrapping every call with #if LOGGINGMODE is tedious:

| #if LOGGINGMODE LogStatus("Message Headers: " + GetMsgHeaders()); #endif |
| --- |

Putting #if *inside* LogStatus is problematic because arguments would always be evaluated, potentially incurring performance penalties for complex argument expressions:

| // Problematic: GetComplexMessageHeaders() always runs, even if logging is off LogStatus("Message Headers: " + GetComplexMessageHeaders());  static void LogStatus(string msg) { #if LOGGINGMODE // This check is too late!  string logFilePath = "log.txt";  System.IO.File.AppendAllText(logFilePath, msg + "\r\n"); #endif } |
| --- |

**Solution with [Conditional] Attribute:**

By applying the [Conditional] attribute to the LogStatus method, you instruct the compiler to treat all calls to LogStatus as if they were wrapped in an #if LOGGINGMODE directive:

| using System.Diagnostics; // Required for [Conditional]  [Conditional("LOGGINGMODE")] static void LogStatus(string msg) {  string logFilePath = "log.txt";  System.IO.File.AppendAllText(logFilePath, msg + "\r\n"); } |
| --- |

Now, if LOGGINGMODE is not defined, any calls to LogStatus are entirely **eliminated during compilation**, including the evaluation of their arguments. This means any side effects within argument expressions are bypassed. This mechanism works even if the LogStatus method and its callers are in different assemblies.

An additional benefit is that the conditionality check is performed when the *caller* is compiled, allowing you to build and distribute a single version of a library that includes conditional methods. The [Conditional] attribute is purely a compile-time instruction and is ignored at runtime.

### **Alternatives to the [Conditional] Attribute (for Runtime Control)**

The [Conditional] attribute is suitable only when you need to control functionality at compile time. If you require the ability to dynamically enable or disable functionality at runtime, you must use a variable-based approach.

To elegantly circumvent the evaluation of arguments for logging methods in a runtime-controlled scenario, you can use a functional approach by accepting a Func<T> delegate for the message:

| using System; using System.Linq; // For brevity, not strictly needed for this example  class Program {  public static bool EnableLogging; // Runtime flag   static void LogStatus(Func<string> messageFactory)  {  string logFilePath = "log.txt"; // simplified  if (EnableLogging)  {  // The messageFactory() is only invoked if logging is enabled  System.IO.File.AppendAllText(logFilePath, messageFactory() + "\r\n");  }  } } |
| --- |

You can then call this method using a lambda expression, which defers the message generation until it's actually needed:

| // Example call: LogStatus(() => "Message Headers: " + GetComplexMessageHeaders()); |
| --- |

If EnableLogging is false, the lambda expression () => "Message Headers: " + GetComplexMessageHeaders() is never executed, meaning GetComplexMessageHeaders() is never called, effectively bypassing any associated performance hit.