## **Utility Classes: Essential Tools for System Interaction and Diagnostics**

Beyond core language features and data structures, the .NET Framework provides a rich set of utility classes that facilitate interaction with the operating system, manage processes, and handle common application-level tasks.

### **Console Class: Standard Input/Output for Console Applications**

The static System.Console class is your primary interface for standard input and output in console-based applications.

* **Input:** Methods like Read(), ReadKey(), and ReadLine() allow you to receive input from the keyboard.
* **Output:** Write() and WriteLine() methods send output to the console text window. They are overloaded to accept composite format strings (similar to string.Format), but they do not accept IFormatProvider directly, meaning they always use CultureInfo.CurrentCulture for formatting. If you need specific culture formatting, you must explicitly use string.Format beforehand.
* **Console Customization:** You can control the console window's appearance and cursor:
  + WindowLeft, WindowTop, WindowHeight, WindowWidth for dimensions.
  + BackgroundColor, ForegroundColor for colors.
  + CursorLeft, CursorTop, CursorSize for cursor manipulation.

| Console.WindowWidth = Console.LargestWindowWidth; Console.ForegroundColor = ConsoleColor.Green; Console.Write("test... 50%"); Console.CursorLeft -= 3; // Move cursor back 3 positions Console.Write("90%"); // Result: "test... 90%" |
| --- |

* **Stream Redirection:** Console.Out returns a TextWriter, which is useful for directing console output to other streams for diagnostic or logging purposes. You can also entirely redirect the console's input and output streams using Console.SetIn() and Console.SetOut().

| System.IO.TextWriter oldOut = Console.Out; // Save current output writer using (System.IO.TextWriter w = System.IO.File.CreateText("e:\\output.txt")) {  Console.SetOut(w); // Redirect console output to a file  Console.WriteLine("Hello world"); } Console.SetOut(oldOut); // Restore original console output |
| --- |

* *(Note: In Visual Studio debug mode, Console output is often redirected to the Output window, making Console.Write useful for quick debugging, though System.Diagnostics.Debug and System.Diagnostics.Trace are generally more appropriate for structured diagnostics.)*

### **Environment Class: System and User Information**

The static System.Environment class provides a wealth of information about the current operating environment, including the computer, operating system, user, and application.

* **Files and Folders:** CurrentDirectory, SystemDirectory, CommandLine.
* **Computer and Operating System:** MachineName, ProcessorCount, OSVersion, NewLine.
* **User Logon:** UserName, UserInteractive, UserDomainName.
* **Diagnostics:** TickCount, StackTrace, WorkingSet, Version.
* **Special Folders:** GetFolderPath() provides paths to common system folders (e.g., "My Documents", "Program Files").
* **Environment Variables:** You can access, retrieve, and set OS environment variables using GetEnvironmentVariable(), GetEnvironmentVariables(), and SetEnvironmentVariable().
* **Program Exit:**
  + ExitCode: Sets the program's return code for external callers (e.g., batch files).
  + FailFast(): Immediately terminates the application without performing cleanup, typically used for critical, unrecoverable errors.

### **Process Class: Launching and Interacting with Processes**

The System.Diagnostics.Process class allows you to launch new processes, interact with existing ones, and manage their input/output streams.

* **Launching Processes:** The static Process.Start() method has several overloads. The simplest takes a filename and optional arguments.

| Process.Start("notepad.exe"); Process.Start("notepad.exe", "e:\\file.txt"); |
| --- |

* **ProcessStartInfo for Flexibility:** For more control, use an instance of ProcessStartInfo. This allows you to configure redirection of standard input, output, and error streams, among other settings.

| ProcessStartInfo psi = new ProcessStartInfo {  FileName = "cmd.exe",  Arguments = "/c ipconfig /all", // /c executes command and exits  RedirectStandardOutput = true, // Crucial for capturing output  UseShellExecute = false // Must be false for stream redirection }; Process p = Process.Start(psi)!; string result = p.StandardOutput.ReadToEnd(); Console.WriteLine(result); |
| --- |

* **Waiting for Exit:** If you want your application to wait for the launched process to complete, call WaitForExit() on the Process object, optionally with a timeout.
* **Redirecting Output and Error Streams:** When UseShellExecute is false (the default in .NET 5+ / .NET Core, but true in .NET Framework), you can capture StandardInput, StandardOutput, and StandardError streams.
  + **Handling Interleaved Output:** If you redirect both standard output and standard error, data from these streams might be interleaved unpredictably. The solution is to read from at least one of the streams **asynchronously** using the OutputDataReceived and ErrorDataReceived events, paired with BeginOutputReadLine() and BeginErrorReadLine().

| (string output, string errors) Run(string exePath, string args = "") {  using var p = Process.Start(new ProcessStartInfo(exePath, args)  {  RedirectStandardOutput = true,  RedirectStandardError = true,  UseShellExecute = false, // Essential for stream redirection  })!;   var errors = new StringBuilder();  p.ErrorDataReceived += (sender, errorArgs) =>  {  if (errorArgs.Data != null) errors.AppendLine(errorArgs.Data);  };  p.BeginErrorReadLine(); // Start asynchronous read for errors   string output = p.StandardOutput.ReadToEnd(); // Synchronously read output  p.WaitForExit(); // Wait for the process to exit  return (output, errors.ToString()); } |
| --- |

### **AppContext Class: Application Context and Feature Switches**

The static System.AppContext class provides information about the application's environment and a mechanism for library authors to expose feature switches.

* **Key Properties:**
  + BaseDirectory: Returns the directory where the application's executable is located. Crucial for resolving assembly dependencies and locating configuration files.
  + TargetFrameworkName: Reports the name and version of the .NET runtime the application *targets* (as defined in its configuration), which might be different from the actual runtime it's currently executing on.
* **Feature Switches:** AppContext manages a global dictionary of boolean-valued switches. This allows library developers to provide a way for consumers to enable or disable new or experimental features, especially those that might involve breaking changes.
  + Library Consumer: AppContext.SetSwitch("MyLibrary.SomeBreakingChange", true);
  + Library Code:

| bool isDefined, switchValue; isDefined = AppContext.TryGetSwitch("MyLibrary.SomeBreakingChange", out switchValue); if (isDefined && switchValue) {  // New feature is enabled } else {  // Old behavior or feature is disabled } |
| --- |

* + TryGetSwitch returns false if the switch is not defined, allowing you to differentiate between an undefined switch and one explicitly set to false.