Asynchronous Programming Code

Build asynchronous script logic.

Reference **Code** **Code Library** Async Programming

Asynchronous programming is a paradigm that allows for the execution of tasks without blocking the main execution thread. This approach is essential for developing stable and scalable applications, especially when dealing with operations that are I/O-bound or time-consuming, such as network requests, database operations, or file I/O.

Synchronous and Asynchronous
- Scripts Asynchronous Methods
- Debugging Asynchronous Code
- Error handling

Best Practices

Synchronous and Asynchronous

Programming allows running tasks one after another or running tasks without waiting for others to finish. When tasks run one after another, each task waits for the task before it finishes. When tasks run without waiting, tasks start and proceed without delay, allowing operations to occur at the same time.

Benefits:

- **Responsiveness**: Keeps applications responsive, especially in UI scenarios where blocking the main thread can lead to a frozen interface.
- **Scalability**: Efficiently handles multiple concurrent operations without the overhead of creating new threads for each task.
- **Resource Efficiency**: Frees up threads to perform other work while awaiting long-running operations, optimizing CPU and memory usage.

Scripts Asynchronous Methods

Version 10 uses a more modern and efficient programming pattern called asynchronous methods. This improves performance on displays and is one of the technologies that enables most Windows WPF displays to run on web HTML5 with no modifications.

When creating new solutions, this is done automatically for you. However, legacy displays with heavy CodeBehind scripts may require modification of methods to use the async operator.

You can find more information on the Microsoft Documentation about Async programming here.

Calling Server Classes from Displays and Client Classes

Now when calling Script Class Server method from Client scripts (Script Class Client or Display CodeBehind) should add "await" (C#) "Await" (VB.NET) for the HTML5 to work successfully.

CSharp

```
public async Task DisplayOpening()
        {
                await @Script.Class.ServerMain.MethodExample();
        }
```

VB.NET

```
Public Async Function DisplayOpening() As Task
        Await @Script.Class.ServerMain.Method1()
End Function
```

Correction of Logon() on CodeBehind

The method Logon() should be replaced by LogonAsync(), as well other synchronous methods shall be modified to their async versions. Here is the modification of the Logon() method, as it was very commonly used in the projects.

JS

```
this.DialogOnOK = async function()
{
    var result = await @Client.LogOnAsync("user1", "pass1");
    return true;
};
```

CSharp

```
public async Task<bool> DialogOnOK()
{
    var result = await @Client.LogOnAsync("Username", "Password");
    return true;
}
```

VB.NET

```
Public Async Function DialogOnOK() As Task(Of Boolean)
    Dim result As Integer = Await @Client.LogOnAsync("user1", "pass1")
    Return True
End Function
```

Calling Async Methods

One reason the update of the code to use async methods is not fully automated is that when a method is modified to become async, other methods that call it need to be updated as well. These calling methods must also be updated in their signatures and return values to accommodate the new async method. There are different ways to implement this, typically using `await` or `Result` operators. It would not be feasible for the upgrade tool to automatically and safely modify all dependencies and application logic to use the method asynchronously.

Here is complete example, on how a Code Behind of a display was in earlier versions, with synchronous call, and the modified code using async calls.

Legacy CodeBehind

```
Public Sub DisplayOpening()
    ' Add your code here

End Sub

Public Sub DisplayIsOpen()
    ' Add your code here

End Sub

Public Sub DisplayClosing()
    ' Add your code here

End Sub

'This method is only called on Dialog displays
Public Function DialogOnOK() As Boolean
    Dim log As eSecurityErrors
    log = DirectCast(@Client.LogOn(@Client.InputUserName, @Client.InputPassword), eSecurityErrors)
    If log = eSecurityErrors.OK Then
        @Display.LogOn.Close()
        Return True
    End If
    Dim msg As String = @Client.Locale("Could not logon") + " (" + @Client.Locale(log.ToString()) + ")"
    MessageBox.Show(msg)
    Return False
End Function

Public Sub ExecuteLogOff(ByVal sender As Object, ByVal e As MouseButtonEventArgs)
    @Client.LogOnGuest()
    @Display.LogOn.Close()
End Sub

Public Sub MouseLeftButtonOk(ByVal sender As Object, ByVal e As System.Windows.Input.InputEventArgs)
     DialogOnOK()
End Sub
```

The previous code in version 10 will issue warnings, not errors, as it can still be used in displays that are WPF (Windows) only.

The warnings are issued because the code is no longer acceptable for pages targeting Web HTML5. Therefore, it cannot be used in portable pages (Web and Windows ready), which are the preferred option moving forward.

It's important to emphasize that using async also benefits WPF-Windows applications by making UI interactions more responsive.

The upgrade tool will only attempt to automatically update the LogOn page if it was kept as provided in the templates. The following code shows the changes needed to manually correct the LogOnPage if necessary, and the patterns of using `async` and `await` can be applied to other pages that require changes.

Code Behind using async

```
Public Sub DisplayIsOpen()
End Sub

Public Sub DisplayClosing()
End Sub

Public Async Function DialogOnOK() As Task(Of Boolean)
    Dim log As eSecurityErrors
    log = DirectCast(Await @Client.LogOnAsync(@Client.InputUserName, @Client.InputPassword), eSecurityErrors)
    If log = eSecurityErrors.OK Then
    @Display.LogOn.Close()
    Return True
    End If
    Dim msg As String = @Client.Locale("Could not logon") + " (" + @Client.Locale(log.ToString()) + ")"
    MessageBox.Show(msg)
    Return False
End Function

Public Async Function ExecuteLogOff(ByVal sender As Object, ByVal e As MouseButtonEventArgs) As Task
    Await @Client.LogOnGuestAsync()
    @Display.LogOn.Close()
End Function

Public Async Function MouseLeftButtonOk(ByVal sender As Object, ByVal e As System.Windows.Input.
InputEventArgs) As Task
    Await DialogOnOK()
End Function
```

Modified CodeBehind to use async

```
public void DisplayIsOpen() {
}

public void DisplayClosing() {
}

public async Task<bool> DialogOnOK() {
    eSecurityErrors log;
    log = (eSecurityErrors) await @Client.LogOnAsync(@Client.InputUserName, @Client.InputPassword);
    if (log == eSecurityErrors.OK) {
        @Display.LogOn.Close();
        return true;
    }
    string msg = @Client.Locale("Could not logon") + " (" + @Client.Locale(log.ToString()) + ")";
    MessageBox.Show(msg);
    return false;
}

public async Task ExecuteLogOff(object sender, MouseButtonEventArgs e) {
    await @Client.LogOnGuestAsync();
    @Display.LogOn.Close();
}

public async Task MouseLeftButtonOk(object sender, System.Windows.Input.InputEventArgs e) {
    // Add your code here
    await DialogOnOK();
}
```

Async and Await

Both work together to implement an asynchronous execution model. async prepares the method for asynchronous operations, while await manages the wait points within the method.

You apply the `async` modifier to a method. It indicates that the function contains asynchronous operations and can use `await`. It only affects the method or expression where you declare it.

Characteristics:

- **Method declaration:** You use `async` to modify a method, indicating that it must return a `Task`, `Task<T>`, or `void` (in specific cases, such as event handlers).
- **Enables the use of `await`:** Without `async`, the compiler does not allow the use of `await` inside the method.
- **Return management:** A method with `async` always returns a `Task` or `Task<T>`. If the method would originally return `void`, the compiler transforms the return into a `Task` or `Task<T>`.

The `await` operator directly applies to a `Task` or `Task<T>` instance. It suspends the execution of the current method until the associated `Task` completes. It only affects the line or block of code where you use it.

Characteristics:

- **Waiting for the completion of an asynchronous operation:** It suspends execution at the point where it appears, freeing up control of the thread for other operations while it waits.
- **Asynchronous context:** You can only use it in methods or expressions marked as `async`.
- **Return of control:** After the awaited operation completes, the method's execution flow continues from the point of the `await`.

Controlling Tasks Without Await

You can control and monitor the status of an asynchronous operation without awaiting it immediately. This is useful when you want the program to continue executing other tasks while the asynchronous operation runs in the background.

Code example

```
Task taskName = FunctionAsync();

if (taskName.IsCompleted)
{
    // Perform actions after the task has completed
}
```

By checking the `IsCompleted` property of the task, you can determine if the asynchronous operation has finished and then execute additional code accordingly.

TRef<T> in Async Methods

In asynchronous methods in C#, `out` and `ref` parameters cannot be used due to the compiler's transformation of these methods into state machines for asynchronous execution. To overcome this limitation, `TRef<T>` is used as a substitute for `out` parameters. `TRef<T>` serves as a container for values that need to be returned in addition to the method's primary return value.

`TRef<T>` is used in asynchronous methods to simulate the behavior of `out` parameters, which are not allowed in async methods. It allows methods to modify and return additional values beyond the primary return type. Instantiate `TRef<T>` before use, and recognize that returned values can vary based on the method's implementation without a fixed default. This approach supports asynchronous programming while maintaining method signatures similar to their synchronous counterparts.

Why Use `TRef<T>` Instead of Out Parameters?

- **Technical Limitations**: Asynchronous methods do not support `out` or `ref` parameters because of the way the compiler processes async methods.
- **Value Encapsulation**: `TRef<T>` encapsulates a value of type `T`, allowing the method to modify this value and make it accessible after the asynchronous operation completes.
- **Method Signature Consistency**: Using `TRef<T>` maintains similar method signatures between synchronous and asynchronous versions, facilitating the transition to asynchronous programming.

Code Examples

Example of using TRef<int> in an asynchronous method

```
// Definition of TRef<T>
public class TRef<T>
{
    public T Value { get; set; }
}

// Calling the asynchronous method with TRef<int>
public async Task UseTRefAsync()
{
    TRef<int> refStatus = new TRef<int>();
    DataTable table1 = await dataTable1.SelectAsync(refStatus);
    int status = refStatus.Value;
    // Use 'status' as needed
}

// Asynchronous method that uses TRef<int>
public async Task<DataTable> SelectAsync(TRef<int> statusRef)
{
    DataTable result = new DataTable();
    int status = 1; // Example value
    statusRef.Value = status;
    return result;
}
```

Notes:

- Instantiate `TRef<T>` before passing it to the asynchronous method.
- Access the additional returned value through the `Value` property of `TRef<T>` after the method completes.

Return Behavior

- **Variable Return Values**: The content of `TRef<T>` depends on the internal logic of the asynchronous method. There is no fixed rule for default values; it may return `null`, an empty string `""`, a success code, or any other context-specific value.
- **Primary Return Type**: The main method typically returns a `Task<T>`, where `T` is the type of the primary result (e.g., `DataTable`).
- **Use of Await**: Applying `await` pauses the execution of the calling method until the asynchronous method completes, without blocking the UI thread or other asynchronous operations.

Special Considerations

- **Instantiation of TRef<T>**: Always create a new instance of `TRef<T>` before using it to avoid `NullReferenceException`.

  ```
   // Correct usage
  TRef<int> refStatus = new TRef<int>();

  // Incorrect usage - refStatus is not initialized
  TRef<int> refStatus;
  ```

- **Thread Safety**: If the asynchronous method runs on multiple threads, ensure thread safety by providing each thread with its own instance of `TRef<T>` or by synchronizing access.
- **Avoid Blocking Calls**: Do not use methods like `.Result` or `.Wait()` on asynchronous tasks, as this can block the current thread and negate the benefits of asynchronous programming.
- **Exception Handling**: Implement `try-catch` blocks within asynchronous methods to handle exceptions appropriately.

Debugging Asynchronous Code

How to examine the asynchronous operations impact execution flow and application responsiveness. This section covers the execution flow when using await and outlines the differences from synchronous calls, emphasizing how the UI thread stays unblocked and allows users to interact with the application during asynchronous operations.

Execution Flow with Await

Using `await` in asynchronous code makes the execution flow during debugging appear sequential. The debugger pauses at the `await` expression and waits until the asynchronous method completes, mimicking the behavior of synchronous code. This approach allows developers to step through code line by line without disrupting the logical sequence of execution.

Differences from Synchronous Calls

Unlike synchronous calls, using `await` does not block the UI thread during asynchronous operations. The application remains responsive, allowing users to continue their activities while background tasks run. This behavior prevents the interface from freezing during operations that take longer to complete, enhancing the overall user experience.

Error handling

Error handling ensures predictable system behavior by managing failures and anomalies.

eErrorMessage Enum

The eErrorMessage enum standardizes error messages across methods, assigning explicit codes for specific issues like invalid filenames or timeouts. Methods must not return null. They use eErrorMessage values to indicate outcomes. Each method defines return values based on its functionality without relying on default success rules.

Code	Value	Description
0	`Success`	Indicates the operation completes successfully.
1	`Disabled`	Shows that the feature or functionality is disabled.
2	`ModuleStopped`	States that the related module stops and cannot perform the operation.
3	`ModulePaused`	Shows that the related module pauses, preventing the operation.
4	`ModuleReadOnly`	Indicates that the module runs in read-only mode, disallowing write or modification operations.
5	`Exception`	Signals that an exception occurs during execution.
6	`ServiceNotConnected`	States that the service required for the operation is not connected.
7	`NotAllowedInWebBrowser`	Indicates that the functionality is not available in a web browser environment.
8	`NotAvailableOnNetCore`	Indicates that the functionality is not supported on the .NET Core platform.
9	`InvalidFilename`	Denotes that the specified file name is invalid.
10	`ErrorWritingToFile`	Indicates that an error occurs while attempting to write to a file.
11	`NoObjectsConfigured`	Indicates that no objects are configured to execute the operation.
12	`InvalidContent`	Signals that the provided content is invalid.
13	`Timeout`	Shows that a timeout occurs during the execution of the operation.
14	`PreviousCommandStillRunning`	States that a previous command runs, blocking the execution of a new command.
15	`AccessRestricted`	Indicates that access to the functionality or resource is restricted.
16	`NoSelectExecuted`	Indicates that no selection operation executes, preventing further actions.
17	`InvalidURL`	States that the provided URL is invalid.
18	`HTTPError`	Signals that an HTTP error occurs during server communication.
19	`InvalidHandle`	Denotes that the handle used in the operation is invalid.
20	`AsyncCommand`	Indicates that an asynchronous command executes but remains in progress.

> Methods must not return null. Instead, methods should return values or status codes as defined by the eErrorMessage enum.

Best Practices

- **Avoid Blocking Calls**: Do not use blocking calls like `Task.Wait()` or `Task.Result`, which negate the benefits of asynchronous programming.
- **Async All the Way**: Ensure that asynchronous methods call other asynchronous methods to maintain non-blocking behavior throughout the call stack.
- **Exception Handling**: Use try-catch blocks within async methods to handle exceptions properly.
- **Cancellation Tokens**: Implement cancellation tokens to allow operations to be canceled if they are no longer needed.
- **Progress Reporting**: Utilize progress reporting mechanisms to provide feedback during long-running operations.

In this section...