Async Return Types

https://msdn.microsoft.com/en-us/library/hh524395.aspx

Async methods have three possible return types: Task<TResult>, Task, and void.
Each return type is examined in one of the following sections, and you can find a full example that uses all three types at the end of the topic.

Task(T) Return Type

The Task<TResult> return type is used for an async method that contains a return statement in which the operand has type **TResult**.
In the following example, the `TaskOfTResultMethodAsync` async method contains a return statement that returns an integer. Therefore, the method declaration must specify a return type of **Task<int>** .

C#
```
// TASK<TResult> EXAMPLE
async Task<int> TaskOfTResultMethodAsync()
{
    // The body of the method is expected to contain an awaited asynchronous
    // call.
    // Task.FromResult is a placeholder for actual work that returns a string.
    var today = await Task.FromResult<string>(DateTime.Now.DayOfWeek.ToString());

    // The method then can process the result in some way.
    int leisureHours;
    if (today.First() == 'S')
        leisureHours = 16;
    else
        leisureHours = 5;

    // Because the return statement specifies an operand of type int, the
    // method must have a return type of Task<int>.
    return leisureHours;
}
```

When `TaskOfTResultMethodAsync` is called from within an await expression, the await expression retrieves the integer value (the value of `leisureHours`) that's stored in the task that's returned by `TaskOfTResultMethodAsync`.
The following code calls and awaits method `TaskOfTResultMethodAsync`. The result is assigned to the `result1` variable.

C#
```
// Call and await the Task<T>-returning async method in the same statement.
int result1 = await TaskOfTResultMethodAsync();
```

You can better understand how this happens by separating the call to `TaskOfTResultMethodAsync` from the application of **Await**or **await**, as the following code shows. A call to method `TaskOfTResultMethodAsync` that isn't immediately awaited returns a**Task(Of Integer)** or **Task<int>**, as you would expect from the declaration of the method. The task is assigned to the `integerTask` variable in the example. Because `integerTask` is a Task<TResult>, it contains a Result property of type**TResult**. In this case, TResult represents an integer type. When **Await** or **await** is applied to `integerTask`, the await expression evaluates to the contents of the Result property of `integerTask`. The value is assigned to the `result2` variable.

Warning

The Result property is a blocking property. If you try to access it before its task is finished, the thread that's currently active is blocked until the task completes and the value is available. In most cases, you should access the value by using **await** instead of accessing the property directly.

```
C#
// Call and await in separate statements.
Task<int> integerTask = TaskOfTResultMethodAsync();

// You can do other work that does not rely on integerTask before awaiting.
textBox1.Text += String.Format("Application can continue working while the Task<T>"+
                               " runs. . . . \r\n");

int result2 = await integerTask;
```

The display statements in the following code verify that the values of the `result1` variable, the `result2` variable, and the**Result** property are the same. Remember that the **Result** property is a blocking property and shouldn't be accessed before its task has been awaited.

```
C#
// Display the values of the result1 variable, the result2 variable, and
// the integerTask.Result property.
textBox1.Text += String.Format("\r\nValue of result1 variable:   {0}\r\n", result1);
textBox1.Text += String.Format("Value of result2 variable:   {0}\r\n", result2);
textBox1.Text += String.Format("Value of integerTask.Result: {0}\r\n",
integerTask.Result);
```

Task Return Type

Async methods that **don't contain a return statement** or that c**ontain a return statement that doesn't return an operand** usually have a return type of Task. Such methods would be void-returning methods if they were written to run synchronously. If you use a **Task** return type for an async method, a calling method can use an await operator to suspend the caller's completion until the called async method has finished.

In the following example, async method `Task_MethodAsync` doesn't contain a return statement. Therefore, you **specify a return type** of **Task** for the method, which enables `Task_MethodAsync` to be awaited. The definition of the **Task** type doesn't include a **Result** property to store a return value.

```
C#
// TASK EXAMPLE
async Task Task_MethodAsync()
{
    // The body of an async method is expected to contain an awaited
    // asynchronous call.
    // Task.Delay is a placeholder for actual work.
    await Task.Delay(2000);
    // Task.Delay delays the following line by two seconds.
    textBox1.Text += String.Format("\r\nSorry for the delay. . . .\r\n");

    // This method has no return statement, so its return type is Task.
}
```

`Task_MethodAsync` is called and awaited by using an await statement instead of an await expression, similar to the calling statement for a void-returning method. The application of an await operator in this case doesn't produce a value.
The following code calls and awaits method `Task_MethodAsync`.

C#
```
// Call and await the Task-returning async method in the same statement.
await Task_MethodAsync();
```

As in the previous Task<TResult> example, you can separate the call to `Task_MethodAsync` from the application of an await operator, as the following code shows. However, remember that a **Task** doesn't have a **Result** property, and that no value is produced when an await operator is applied to a **Task**.
The following code separates calling `Task_MethodAsync` from awaiting the task that `Task_MethodAsync` returns.

C#
```
// Call and await in separate statements.
Task simpleTask = Task_MethodAsync();

// You can do other work that does not rely on simpleTask before awaiting.
textBox1.Text += String.Format("\r\nApplication can continue working while the Task runs. . . .\r\n");

await simpleTask;
```

Void Return Type

The primary use of the void return type is in event handlers, where a void return type is required. A void return also can be used to override void-returning methods or for methods that perform activities that can be categorized as "fire and forget." However, you should return a **Task** wherever possible, because a void-returning async method can't be awaited. Any caller of such a method must be able to continue to completion without waiting for the called async method to finish, and the caller must be independent of any values or exceptions that the async method generates.
The caller of a void-returning async method can't catch exceptions that are thrown from the method, and such unhandled exceptions are likely to cause your application to fail. If an exception occurs in an async method that returns a Task orTask<TResult>, the exception is stored in the returned task, and rethrown when the task is awaited. Therefore, make sure that any async method that can produce an exception has a return type of Task or Task<TResult> and that calls to the method are awaited..
The following code defines an async event handler.

C#
```
// VOID EXAMPLE
private async void button1_Click(object sender, RoutedEventArgs e)
{
    textBox1.Clear();

    // Start the process and await its completion. DriverAsync is a
    // Task-returning async method.
    await DriverAsync();

    // Say goodbye.
    textBox1.Text += "\r\nAll done, exiting button-click event handler.";
}
```

Complete Example

The following Windows Presentation Foundation (WPF) project contains the code examples from this topic.

To run the project, perform the following steps:

1. Start Visual Studio.
2. On the menu bar, choose **File**, **New**, **Project**.
   The **New Project** dialog box opens.
3. In the **Installed**, **Templates** category, choose **Visual Basic** or **Visual C#**, and then choose **Windows**. Choose **WPF Application** from the list of project types.
4. Enter **AsyncReturnTypes** as the name of the project, and then choose the **OK** button.
   The new project appears in **Solution Explorer**.
5. In the Visual Studio Code Editor, choose the **MainWindow.xaml** tab.
   If the tab is not visible, open the shortcut menu for MainWindow.xaml in **Solution Explorer**, and then choose **Open**.
6. In the **XAML** window of MainWindow.xaml, replace the code with the following code.

   C#

   ```
   <Window x:Class="AsyncReturnTypes.MainWindow"
           xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
           Title="MainWindow" Height="350" Width="525">
       <Grid>
           <Button x:Name="button1" Content="Start" HorizontalAlignment="Left"
   Margin="214,28,0,0" VerticalAlignment="Top" Width="75"
   HorizontalContentAlignment="Center" FontWeight="Bold" FontFamily="Aharoni"
   Click="button1_Click"/>
           <TextBox x:Name="textBox1" Margin="0,80,0,0" TextWrapping="Wrap"
   FontFamily="Lucida Console"/>

       </Grid>
   </Window>
   ```

   A simple window that contains a text box and a button appears in the **Design** window of MainWindow.xaml.
7. In **Solution Explorer**, open the shortcut menu for MainWindow.xaml.vb or MainWindow.xaml.cs, and then choose**View Code**.
8. Replace the code in MainWindow.xaml.vb or MainWindow.xaml.cs with the following code.

   C#

   ```
   using System;
   using System.Collections.Generic;
   using System.Linq;
   using System.Text;
   using System.Threading.Tasks;
   using System.Windows;
   using System.Windows.Controls;
   using System.Windows.Data;
   using System.Windows.Documents;
   using System.Windows.Input;
   using System.Windows.Media;
   ```

```
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace AsyncReturnTypes
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        // VOID EXAMPLE
        private async void button1_Click(object sender, RoutedEventArgs e)
        {
          textBox1.Clear();

          // Start the process and await its completion. DriverAsync is a
          // Task-returning async method.
          await DriverAsync();

          // Say goodbye.
          textBox1.Text += "\r\nAll done, exiting button-click event handler.";
        }

        async Task DriverAsync()
        {
    // Task<T>
    // Call and await the Task<T>-returning async method in the same statement.
            int result1 = await TaskOfTResultMethodAsync();

            // Call and await in separate statements.
            Task<int> integerTask = TaskOfTResultMethodAsync();

    // You can do other work that does not rely on integerTask before awaiting.
            textBox1.Text += String.Format("Application can continue working "+
                                           "while the Task<T> runs.  . \r\n");

            int result2 = await integerTask;

    // Display the values of the result1 variable, the result2 variable, and
    // the integerTask.Result property.
            textBox1.Text += String.Format("\r\nValue of result1 variable:  "+
                                           "{0}\r\n", result1);
            textBox1.Text += String.Format("Value of result2 variable: "+
                                           "  {0}\r\n", result2);
            textBox1.Text += String.Format("Value of integerTask.Result: "+
                                           "{0}\r\n", integerTask.Result);

     // Task
     // Call and await the Task-returning async method in the same statement.
            await Task_MethodAsync();

     // Call and await in separate statements.
            Task simpleTask = Task_MethodAsync();

    // You can do other work that does not rely on simpleTask before awaiting.
            textBox1.Text += String.Format("\r\nApplication can continue "+
                                "working while the Task runs. . . .\r\n");
```

```
            await simpleTask;
        }

    // TASK<T> EXAMPLE
        async Task<int> TaskOfTResultMethodAsync()
        {
    // The body of the method is expected to contain an awaited asynchronous
    // call.
    // Task.FromResult is a placeholder for actual work that returns a string.

            var today =
              await Task.FromResult<string>(DateTime.Now.DayOfWeek.ToString());

            // The method then can process the result in some way.
            int leisureHours;
            if (today.First() == 'S')
                leisureHours = 16;
            else
                leisureHours = 5;

        // Because the return statement specifies an operand of type int, the
        // method must have a return type of Task<int>.
            return leisureHours;
        }


        // TASK EXAMPLE
        async Task Task_MethodAsync()
        {
          // The body of an async method is expected to contain an awaited
          // asynchronous call.
          // Task.Delay is a placeholder for actual work.
          await Task.Delay(2000);
            // Task.Delay delays the following line by two seconds.
          textBox1.Text += String.Format("\r\nSorry for the delay. . . .\r\n");

          // This method has no return statement, so its return type is Task.
        }
    }
}
```

9. Choose the F5 key to run the program, and then choose the **Start** button.
   The following output should appear.

```
Application can continue working while the Task<T> runs. . . .

Value of result1 variable:   5
Value of result2 variable:   5
Value of integerTask.Result: 5

Sorry for the delay. . . .

Application can continue working while the Task runs. . . .

Sorry for the delay. . . .

All done, exiting button-click event handler.
```