Asynchronous Programming

- We have any I/O-bound needs (such as requesting data from a network, accessing a database, or reading and writing to a file system), we want to utilize asynchronous programming.
- We could also have CPU-bound code, such as performing an expensive calculation, which is also a good scenario for writing async code.
- The async and await keywords in C# are used in async programming.
- It follows what is known as the Task-based Asynchronous Pattern (TAP).
- The core of async programming is the Task and Task<T> objects, which model asynchronous operations.
- They are supported by the **async** and **await** keywords.

CONTD…

The model is fairly simple in most cases:

- For I/O-bound code, you await an operation that returns a Task or Task<T> inside of an async method.
- For CPU-bound code, you await an operation that is started on a background thread with the Task.Run method.

- **await** yields control to the caller of the method that performed await, and it ultimately allows a UI to be responsive or a service to be elastic.

Asynchronous Method

- An **async** method is represented by using **async** modifier in the method signature.
- If the method has any return types they are enclosed as part of **Task<TResult>** object.
- If the method does not return any values then the return type is just Task.
- Void is also a valid return type and it is used for asynchronous event handlers.
- Every async method should include at least one await operator in the method body to take the advantage of asynchronous programming.

```
class Program {
static async Task Main(string[] args) {
    await callMethod();
    Console.ReadKey();
}
public static async Task callMethod() {
    Method2(); var count = await Method1();
    Method3(count);
}
public static async Task<int> Method1() {
    int count = 0; await Task.Run(() => { for (int i = 0; i < 100; i++) {
    Console.WriteLine(" Method 1"); count += 1;}});
    return count;
}
public static void Method2() {
 for (int i = 0; i < 25; i++) { Console.WriteLine(" Method 2"); } }
public static void Method3(int count) { Console.WriteLine("Total count is " +
count); } }
```

Task API

- Task API needs to retrieve multiple pieces of data concurrently.
- The Task API contains two methods, **Task.WhenAll** and **Task.WhenAny**, that allow us to write asynchronous code that performs a non-blocking wait on multiple background jobs.

Wait for multiple tasks to complete

```
public async Task<User> GetUserAsync(int userId)
{
// Given a user Id {userId}, retrieves a User object corresponding
// to the entry in the database with {userId} as its Id.
 }
public static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int> userIds)
{
var getUserTasks = new List<Task<User>>();
foreach (int userId in userIds)
{
    getUserTasks.Add(GetUserAsync(userId));
}
return await Task.WhenAll(getUserTasks);
}
```