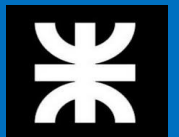


# Unidad 2

Fundamentos de los Lenguajes

## Capítulo 4

Programación Asíncrona



# Contenidos

- Generalidades
- Modelo asincrónico
- Trabajos *I/O-bound*
- Trabajos *CPU-bound*
- Aspectos clave
- Esperar a que se completen varios trabajos
- Mejores prácticas

# Generalidades

Si tenemos cualquier necesidad ***I/O-bound*** (por ejemplo, solicitar datos de una red, acceder a una base de datos, o leer y escribir un sistema de archivos), deberíamos usar la programación asincrónica.

También podríamos tener código ***CPU-bound***, como realizar un cálculo costoso, que también es un buen escenario para escribir código asincrónico.

# Modelo asincrónico

El núcleo de la programación asincrónica son los objetos `Task` y `Task<T>`, que modelan las operaciones asincrónicas. Son compatibles con las palabras clave `async` y `await`.

- Código ***I/O-bound***: se espera (`await`) una operación que devuelva `Task` o `Task<T>` dentro de un método `async`
- Código ***CPU-bound***: se espera (`await`) una operación que se inicia en un subproceso (`thread`) en segundo plano con el método `Task.Run`

# Trabajos *I/O-bound*

Descargar datos de un servicio web cuando se presione un botón, sin bloquear el subproceso de UI:

```
private readonly HttpClient _httpClient = new HttpClient();

downloadButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI as the request
    // from the web service is happening.
    //
    // The UI thread is now free to perform other work.
    var stringData = await _httpClient.GetStringAsync(URL);
    DoSomethingWithData(stringData);
};
```

# Trabajos *CPU-bound*

Calcular daño infligido en un juego para móviles en un subproceso en segundo plano, sin bloquear el subproceso de UI:

```
private DamageResult CalculateDamageDone()
{
    // Does an expensive calculation and returns the result.
}

calculateButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI while
    // CalculateDamageDone() performs its work. The UI thread is
    // free to perform other work.
    var damageResult = await Task.Run(() => CalculateDamageDone());
    DisplayDamage(damageResult);
};
```

# Aspectos Claves

- El código asincrónico puede usarse para código ***I/O-bound*** como ***CPU-bound***, pero de forma distinta en cada escenario.
- El código asincrónico usa **Task<T>** y **Task**, que son construcciones que se usan para modelar el trabajo que se realiza en segundo plano.
- La palabra clave **async** convierte un método en un método asincrónico, lo que permite usar la palabra clave **await** en su cuerpo.
- Cuando se aplica la palabra clave **await**, se suspende el método invocado y se cede el control de vuelta al autor de la invocación hasta que se completa la tarea esperada.
- **await** solo puede usarse dentro de un método asincrónico.

# Esperar a que se completen varios trabajos

Podemos escribir código asíncrono que realiza una espera sin bloqueo de varios trabajos en segundo plano:

```
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);
}
```



# Buenas prácticas

- Los métodos **async** deben tener una palabra clave **await** en el cuerpo o serán ineficientes.
- Agregar "Async" como sufijo al nombre de todos los métodos asincrónicos que se escriba.
- **async void** sólo se debe usar para manejadores de eventos.
- Escribir código menos "stateful": no depender del estado de objetos globales (shared state) sino solamente de los valores devueltos por los métodos.

# Más información

Para más información sobre **Programación Asíncrona** puede consultar:

<https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/>