

A collection of approximately 15 squares in light blue, medium blue, and grey, arranged in a sparse, abstract pattern across the top half of the slide.

# MDPA

Máster en Diseño y Programación de Apps

## Entornos de desarrollo

### Parte 3: Xamarin

A collection of approximately 10 squares in light blue, medium blue, and grey, arranged in a sparse, abstract pattern across the bottom half of the slide.

# Presentación asignatura/sesión

## Entorno de desarrollo Xamarin



### ■ Descripción y temario

- En esta primera parte de Xamarin, veremos el entorno usado para trabajar, Visual Studio y Xamarin Studio, los requisitos de hardware y software para cada tipo de proyecto, los diferentes tipos de proyecto que podemos crear y una breve introducción a como funciona cada uno con herramientas como MVVMCross.

### ■ Profesor

- Oriol Noya - *Freelance Frontend Web/Mobile Developer & Creative Director*

### ■ Evaluación

- Se realizará un ejercicio práctico al final de las sesiones donde se pondrán en práctica todos los conceptos aprendidos.

### ■ Entrega

- **07/01/2018:** Práctica final de las sesiones de Xamarin

# Introducción a C#

## Teoria + Ejemplos:

- Tipos de datos
- Boxing & Unboxing
- Construcciones básicas del lenguaje
- Generics
- Expresiones Lambda / Funciones anónimas
- LinQ
- SOLID
- async / await

# Tipos de datos

- C# implementa los tipos de datos soportados por el Base Common Language de .NET:
  - Int, Long.
  - Double, Decimal, Float.
  - String.
  - Date, Char, Boolean
  - Structures
  - Object
- También implementa tipos variables:
  - var
    - Obtiene el tipo en la primera asignación.
  - dynamic
    - Nos permite realizar late binding, asignándose su tipo en ejecución.

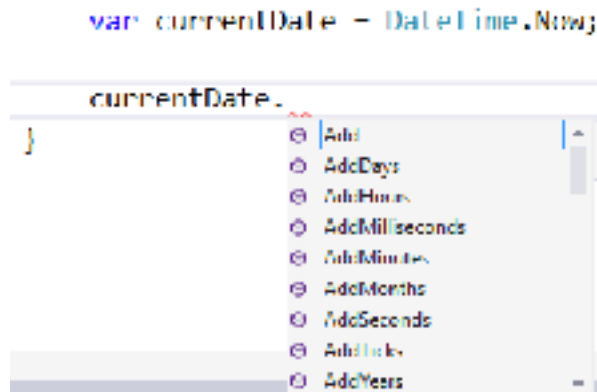
# Tipos de datos variables

## ■ var

- Nos permite asignarle cualquier valor.
- Se tipa con la primera asignación que le realicemos en tiempo de desarrollo.
- No puede variar su tipo.

```
var currentDate = DateTime.Now;  
currentDate = "cadena";
```

- Nos ofrece Intellisense de los miembros del tipo asignado.



# Tipos de datos variables

## ■ dynamic

- Nos permite asignarle cualquier valor.
- Nos permite realizar lo que se conoce como “late binding”
- Cada asignación puede variar su tipo.

```
dynamic currentDate =  
    "10/10/2015";  
currentDate = DateTime.Now;  
currentDate.ToUniversalTime();
```

- No nos ofrece Intellisense de los miembros del tipo asignado.
- Es muy útil trabajando con COM, para obtener objetos no tipados de los cuales conocemos sus miembros solo en tiempo de ejecución.

# Boxing & Unboxing

- El boxing es el acto de convertir un tipo value en un tipo de referencia.
  - Los tipos value almacenan su valor en memoria manejada, dentro de la variable.
  - Los tipos reference almacenan en memoria manejada la dirección de memoria nativa donde reside la información.
  - Al hacer boxing, cogemos un tipo value y lo almacenamos en memoria nativa dentro de un objeto:

```
int answerToLive = 42;  
object boxingLive = answerToLive;
```

- El unboxing es el acto de convertir un tipo referencia en un tipo value.
  - Sacamos su valor de la memoria nativa y lo almacenamos en memoria manejada:

```
object boxedLive = 42;  
int answerToLive = (int)boxedLive;
```

# Boxing & Unboxing

- El tipo genérico para boxing es Object.
- Como no es un tipo fuertemente tipado como var, tendremos que convertirlo al tipo de origen.
- Al hacer esta conversión tendremos una penalización en el rendimiento de la aplicación y en el consumo de memoria.
- Se creará un nuevo objeto de tipo valor con el tipo indicado, permaneciendo los dos en memoria.
- Además, tendremos que tener mucho cuidado, pues al modificar el valor del nuevo elemento, no se modificará el valor original.

```
object boxedLive = 42;  
int answerToLive = (int)boxedLive;  
answerToLive = 41;  
  
Console.WriteLine(boxedLive); //42  
Console.WriteLine(answerToLive); //41
```



# Construcciones básicas del lenguaje

- Para comenzar a trabajar con C# debemos conocer ciertas construcciones básicas:
  - Namespaces
  - Clases
  - Interfaces
  - Propiedades
  - Variables
  - Métodos

# Construcciones básicas del lenguaje: Namespaces

- Los namespaces delimitan el ámbito en el que se contienen clases e interfaces.
- Su objetivo primordial es el de funcionar como organizador de nuestro código, encapsulándolo.
- En un mismo namespace no pueden existir dos objetos con el mismo nombre.
- En distintos namespaces pueden existir objetos con el mismo nombre.
- Podemos añadir usings a nuestro código para acceder a objetos dentro de otros namespaces

```
namespace TiposVariables
{
    using System;
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Console pertenece al namespace System.");
        }
    }
}
```

# Construcciones básicas del lenguaje: Clases

- Las Clases son unidades básicas de creación de objetos.
- Pueden contener miembros públicos o privados, ya sean métodos, propiedades o variables.
- Cuando queremos crear un nuevo tipo complejo, usamos una clase para definirlo y aportarle funcionalidad.
- Puede ser estática, sin necesidad de instanciarla.
- Si no es estática, tendremos que crear una instancia usando la palabra clave `new` para poder acceder a su funcionalidad.

```
public class Student
{
    public string Name;
    public string PhoneNumber;
}
```

# Construcciones básicas del lenguaje: Interfaces

- Las interfaces proporcionan la capacidad de definir contratos que pueden ser implementados por clases o estructuras.
- Permite que el contrato y la implementación de la funcionalidad queden desacoplados.
- Pueden contener eventos, métodos, indizadores y propiedades.
- Las clases o estructuras que implementen una interfaz deben implementar todos los miembros.
- **NO** se puede crear una instancia de una interfaz.
- Todos sus miembros son públicos.
- Una clase puede implementar varias interfaces.
- Solo pueden tener como modificadores de acceso **internal** o **public**.

# Construcciones básicas del lenguaje: Interfaces

- Las interfaces son vitales en el desarrollo moderno por su uso con patrones de inversión del control.
- Podemos declarar un objeto del tipo de una interface y asignarle un tipo que implemente esa interface. De esta forma nos aseguramos que el tipo asignado cumple el contrato establecido.

```
public interface IStudent
{
    string GetName();
}
```

```
public class Student :
    IStudent
{
    public string Name;
    public string GetName()
    {
        return Name;
    }
}
```

# Construcciones básicas del lenguaje: Interfaces

```
class Program
{
    static void Main(string[] args)
    {
        IStudent student = new Student();

        string name = student.GetName();
    }
}
```

# Construcciones básicas del lenguaje: Propiedades

- Es un miembro que proporciona un mecanismo flexible para leer, escribir o calcular el valor de un campo privado.
- Se pueden usar las propiedades como si fueran miembros de datos públicos.
- Permite acceder a los datos de forma segura y flexible.
- Se utiliza el descriptor de acceso **get** para devolver el valor de la propiedad.
- Se utiliza el descriptor **set** para asignar un valor a la propiedad.
- La palabra clave **value** se utiliza para definir el valor asignado por el descriptor de acceso **set**.
- Existen propiedades de solo lectura, estas no implementan el descriptor de acceso **set** o lo implementan privado.
- En una interface solo podemos declarar propiedades, no variables.
- Existen propiedades auto implementadas.

# Construcciones básicas del lenguaje: Propiedades

`public string Name { get; set; }` Auto implementada

```
private string name;
public string Name
{
    get { return this.name; }
    set { this.name = value; }
}
```

Lectura/Escritura

Solo lectura

```
private string name;
public string Name
{
    get { return this.name; }
}

private string name;
public string Name
{
    get { return this.name; }
    private set { this.name =
value; }
}
```



# Generics

- Desde .NET 2.0 C# contempla el concepto de tipos genéricos.
- Los genéricos nos permiten trabajar con objetos tipados en tiempo de desarrollo sobre construcciones genéricas.
- Se usan principalmente:
  - Reutilización de código.
  - Listas de tipos complejos.
  - Funciones anónimas.
- Se suelen identificar con la letra T encuadrada entre los símbolos < y >
- Se puede aplicar genéricos a clases y a métodos, no a propiedades.
- Para obtener el valor por defecto de un genérico usamos el método default.

# Generics - Ejemplos

- Imaginemos un caso práctico.
- Necesitamos crear una clase que pueda almacenar una lista de elementos.
- Queremos que soporte distintos tipos de elementos.
- Queremos trabajar de forma fuertemente tipada.

```
public class MyList
{
    public IList<Object> Items { get;
set; }

    public void Add(Object newElement)
    {
        Items.Add(newElement);
    }

    public Object GetFirst()
    {
        return Items.First();
    }
}
```

```
public class MyList<T>
{
    public IList<T> Items { get; set;
}

    public void Add(T newElement)
    {
        Items.Add(newElement);
    }

    public T GetFirst()
    {
        return Items.First();
    }
}
```

# Generics - Ejemplos

```
static void Main(string[] args)
{
    var list = new MyList();
    list.Add("1");
    list.Add(23);
    int firstNotTyped = (int)list.GetFirst();

    var typedList = new MyList<int>();
    typedList.Add(1);
    typedList.Add(23);
    int firstTyped = typedList.GetFirst();
}
```

# Expresiones Lambda / Funciones anónimas

- Con la introducción de genéricos en C# 2.0, también se introdujeron las expresiones lambda, también conocidas como funciones anónimas.
- Una expresión lambda nos permite escribir una pieza de código que se autocontenga, con su propio scope, sin necesidad de definir un método.
- Podemos almacenar este código en una variable:

- Action si no devuelve ningún valor.

```
Action lambdaOne = () =>
{
    Console.WriteLine("Texto desde una función anónima");
};
```

- Func<T> si necesitamos devolver un valor.

```
Func<bool> lambdaTwo = () =>
{
    Console.WriteLine("Texto desde una función anónima");
    return true;
};
```

# Expresiones Lambda / Funciones anónimas

- Después de almacenarla, podemos ejecutarla tantas veces como necesitemos.
- El contexto de ejecución será el original donde hemos definido la función anónima, pudiendo acceder a valores de su contexto, incluso si lo ejecutamos en un contexto diferente.
- Esto nos abre múltiples posibilidades de abstracción de código, patrones de ejecución y desacoplamiento de código.
- Podemos tener una clase que solo sea infraestructura de ejecución y definir el código de lógica en otra, separando la arquitectura de la ejecución real.

# Expresiones Lambda / Funciones anónimas

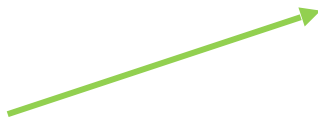
```
public class LambdaTest
{
    public Func<bool> Lambda { get; set; }

    public bool ExecuteLambda()
    {
        return Lambda();
    }
}
```

Contexto invocación



Contexto ejecución



```
class Program
{
    static void Main(string[] args)
    {
        int number = 0;

        Func<bool> lambdaTwo = () =>
        {
            number += 1;
            Console.WriteLine("Número: " + number);
            return true;
        };

        LambdaTest test = new LambdaTest();
        test.Lambda = lambdaTwo;
        test.ExecuteLambda();
        test.ExecuteLambda();
        test.ExecuteLambda();

        Console.WriteLine("Number es igual a " +
            number);
        Console.ReadKey();
    }
}
```

# LinQ

- LINQ es un lenguaje de consultas en línea. Nos permite usar un lenguaje parecido al SQL, pero escrito en C# y fuertemente tipado, para trabajar con datos, listas y objetos complejos.
- Para ello se apoya en las funciones anónimas y los tipos genéricos que ya hemos visto.
- Existen dos formas de escribir consultas LINQ:
  - Lenguaje fluido
  - Lenguaje de consultas
- Podemos usar cualquiera de los dos en nuestro código, aunque por claridad y brevedad es más usada la construcción fluida.

# LinQ – Lenguaje usado

- Lenguaje Fluido

```
Student YoungerStudent = (from stu in students  
                           orderby stu.Age  
                           select stu).FirstOrDefault();
```

- Lenguaje de consultas

```
Student YoungerStudent =  
    students.OrderBy(stu =>  
stu.Age).FirstOrDefault();
```



# LinQ

- Operadores
  - Where
  - OrderBy
  - OrderByDescending
  - Any
  - GroupBy
  - Sum
  - Max
  - Min
  - Select
  - SelectMany
- Namespace System y System.Linq

## LinQ – Ejemplo paso a paso

- Empecemos con un ejemplo, queremos construir una lista de objetos Student donde almacenemos todos los estudiantes de nuestra clase.
- Por cada estudiante tendremos su nombre, nota media y edad.
- Queremos poder ver:
  - El estudiante de menor nota media.
  - El estudiante de mayor nota media.
  - El estudiante más joven.
- Para todas estos filtros usaremos LINQ en vez de iterar la colección en busca de los valores.

# SOLID

- **SOLID** es un acrónimo (compuesto de más acrónimos)
- Enunciado por Robert C. Martín a comienzo de la década de 2000.
- **SOLID** expone una serie de principios consideradas como “beneficiosos” en el mundo del desarrollo en general. No es aplicable solo a C# o Xamarin, cualquiera que fuese el lenguaje orientado a objetos que usemos, deberíamos aplicarlas.
  - **S**ingle Responsibility Principle
  - **O**pen/Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inversion Principle

# Single Responsibility Principle

- El primero de los cinco principios, la responsabilidad única, nos indica que una clase debe tener solo una responsabilidad. También se conoce a este principio por su acrónimo en inglés: SRP.
- Esta responsabilidad debe ser gestionada solo por esa clase. ¿A qué nos referimos por responsabilidad?
- Razón para cambiar. Una clase debe tener solo una razón para cambiar.
- Por poner un ejemplo, piensa en una clase que se encarga de formatear textos, mostrarlos al usuario en pantalla y procesar la respuesta de éste

# Single Responsibility Principle

- Por poner un ejemplo, piensa en una clase que se encarga de formatear textos, mostrarlos al usuario en pantalla y procesar la respuesta de éste
- Esta clase ficticia tiene 3 responsabilidades (razones para cambiarla) distintas:
  - Formatear una cadena de texto.
  - Mostrarla al usuario.
  - Procesar la respuesta del usuario.
- Tendríamos que dividirla en tres clases diferentes:
  - Una clase formatea las cadenas de texto.
  - Una clase muestra los mensajes al usuario.
  - Una clase que procese la respuesta.
- La primera ventaja de esto es que, cualquier otra clase que necesite mostrar o formatear mensajes, puede reaprovechar estas clases, al estar aisladas de la lógica.

# Open/Closed Principle

- El principio Abierto/Cerrado, conocido por sus siglas en inglés OCP, nos indica la forma correcta de extender nuestras clases.
- Según este principio, nuestras clases deben cambiarse, solamente, creando nuevas clases que hereden de las antiguas. De esta forma, mantenemos esas clases originales cerradas mientras abrimos la posibilidad de añadirles nuevas funcionalidades. En definitiva: Nuestras entidades de software deben estar abiertas a ser extendidas, pero cerradas a ser modificadas.
- ¿Cómo podemos conseguir este comportamiento? Actualmente podemos aplicar este principio analizando los puntos que pueden ser extensibles en nuestra clase. Una vez que identifiquemos esos puntos, podemos marcarlos como virtuales, de forma que, una clase derivada pueda sobre escribirlos y añadirles funcionalidad.

# Open/Closed Principle

```
public class OriginalClass
{
    protected int number;

    public virtual void DoWork()
    {
        number = 30;
    }
}
```

```
public class ExtendedOriginalClass :
OriginalClass
{
    public override void DoWork()
    {
        base.DoWork();

        this.number += 12;
    }
}
```

# Liskov Substitution Principle.

- Este principio obtiene su nombre de Barbara Liskov, profesora de ingeniería en el prestigioso MIT y directora del grupo de metodología de desarrollo en el mismo.
- Originalmente enunciado como la definición de subtipos, es más conocido como el principio de sustitución de Liskov o LSP por sus siglas en inglés.
- Este principio trata sobre la capacidad de sustituir entre si dos objetos mutables. Si tenemos una clase B que es un subtipo de una clase A, deberíamos poder sustituir un objeto de tipo A por otro de tipo B sin producir ningún impacto negativo en la aplicación.
- En la práctica esto se consigue mediante el uso de interfaces. En una interfaz definiremos las propiedades y métodos básicos que toda clase debe implementar.
- Una vez hecho esto, podremos crear tantas clases que implementen ese interfaz como deseemos. El tipo de datos que usaremos en todo momento será la interfaz, con lo que nos abstraemos de la clase exacta que instanciemos.



# Liskov Substitution Principle.

```
public interface IVehicle
{
    bool IsStarted { get; set; }

    void StartCar();
}
```

```
public class SportCar : IVehicle
{
    private bool ensureEnoughGas;
    private int maxSpeed;

    public bool IsStarted { get; set; }

    public void StartCar()
    {
        maxSpeed = 280;
        ensureEnoughGas = true;
        IsStarted = ensureEnoughGas;
    }
}
```

```
public class UtilityCar : IVehicle
{
    private bool securityBeltCheck;
    private bool doorLocksCheck;
    public bool IsStarted { get; set; }

    public void StartCar()
    {
        securityBeltCheck = true;
        doorLocksCheck = true;
        IsStarted = securityBeltCheck &&
                    doorLocksCheck;
    }
}
```

# Interface Segregation Principle

- El cuarto de los principios SOLID. Conocido por sus siglas en inglés, ISP. Este principio enuncia que es mucho mejor tener muchos interfaces específicos que un interfaz global y lleno de métodos.
- Ningún cliente debería tener métodos que no va a necesitar. Estos interfaces más pequeños y específicos son conocidos como interfaces de rol.
- A esto podemos añadir, que nadie nos va a cobrar las interfaces por el número de ellas que tengamos y si es correcto que una interfaz tenga un solo método, ¿Por qué debería preocuparnos? No es una cuestión de números, ni existe un mínimo de miembros.
- Debemos intentar dividir nuestras interfaces por funcionalidad que aportan y escapar de grandes interfaces globales, siempre que podamos.

# Dependency Inversion Principle

- Este principio habla de cómo desacoplar nuestro código de las dependencias directas que tenga.
- Tradicionalmente en el desarrollo de software, nuestras clases eran monolíticas, es decir, contenían todo el código necesario para funcionar sin dependencias externas.
- Con el crecimiento de los proyectos en tamaño y funcionalidades era necesario reaprovechar código de la forma más sencilla posible. Para esto, se empezó a dividir estas clases monolíticas en componentes reaprovechables.
- Cada clase dependía de un número X de clases de apoyo, resueltas directamente en la clase principal, por lo que la lógica de la clase y el código de unión de sus dependencias estaban mezcladas.

# Dependency Inversion Principle

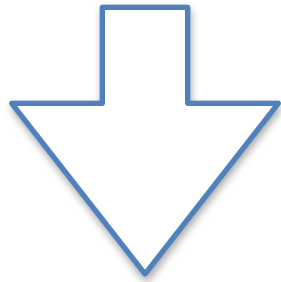
- Lo que propone la inversión de dependencias es la de separar más estos conceptos. Nuestra clase principal no tiene porqué contener la lógica de unión de estas clases, ni siquiera conocer sus implementaciones exactas.
- En su lugar usaremos interfaces para definir contratos que consumirá nuestra clase principal.
- Usando un patrón locator, que veremos más adelante también, podremos desligar esa resolución de dependencias de la lógica de nuestra clase

## Async / await

- A nivel de UX queremos que la UI responda rápidamente a las acciones del usuario (que no se “bloquee”)
- Muchas de las funcionalidades de una app dependen de tareas que pueden tardar mucho tiempo en terminarse:
  - Descarga de datos de un servicio web via 3G o Wifi
  - Acceso a datos de la base de datos
  - Procesos complejos para la CPU de un móvil
- Como desarrolladores queremos que estas tareas no nos afecten el curso natural de la app a nivel de UI
- Por este motivo, necesitamos hacer correr estas tareas en un thread diferente al de la UI, es decir, hacerlas **ASÍNCRONAS**

# Async / await

- Hasta ahora esto lo tenía que gestionar completamente el programador:
  - Cambiando de thread manualmente
  - Gestionando los callbacks manualmente
  - Gestionando los errores manualmente



- Demasiado código → Difícil de leer
- El flujo salta de un sitio a otro → Difícil de entender
- La gestión de errores → Difícil de implementar

# Async / await

```
var webClient = new WebClient();

webClient.DownloadStringCompleted += (sender, e) => {
    if(e.Cancelled || e.Error != null) {
        // do something with error
    }
    string contents = e.Result;

    int length = contents.Length;
    InvokeOnMainThread (() => {
        ResultTextView.Text += "Downloaded the html and found out the length \n\n";
    });

    webClient.DownloadDataCompleted += (sender1, e1) => {
        if(e1.Cancelled || e1.Error != null) {
            // do something with error
        }
        SaveBytesToFile(e1.Result, "team.jpg");

        InvokeOnMainThread (() => {
            ResultTextView.Text += "Downloaded the image.\n";
            DownloadedImageView.Image = UIImage.FromFile (localPath);
        });

        ALAssetsLibrary library = new ALAssetsLibrary();
        var dict = new NSDictionary();
        library.WriteImageToSavedPhotosAlbum (DownloadedImageView.Image.CGImage, dict, (s2,e2) => {
            InvokeOnMainThread (() => {
                ResultTextView.Text += "Saved to album assetUrl\n";
            });
            if (downloaded != null)
                downloaded(length);
        });
    });

    webClient.DownloadDataAsync(new Uri("http://xamarin.com/images/about/team.jpg"));
};

webClient.DownloadStringAsync(new Uri("http://xamarin.com/"));
```

# Async / await

- Con “Async”:
  - Las tareas se ejecutan fuera del flujo principal del programa
  - El código se ejecuta en otro thread, por lo que no bloquea la UI
  - Cuando una tarea termina su ejecución, ella misma devuelve el flujo al thread que la ha llamado
- Algunos ejemplos de tareas “Async”:
  - `HttpClient.GetStringAsync()`
  - `FileStream.ReadAsync()`
  - `ALAsset.SetImageDataAsync()`
  - `Android.Graphics.BitmapFactory.DecodeFileAsync()`
  - `Geolocator.GetPositionAsync()`



# Async / await

```
public async Task<int> DownloadHomepageAsync()
{
    try {
        var httpClient = new HttpClient();

        Task<string> contentsTask = httpClient.GetStringAsync("http://xamarin.com");

        string contents = await contentsTask;

        int length = contents.Length;
        ResultTextView.Text += "Downloaded the html and found out the length.\n\n";

        byte[] imageBytes = await httpClient.GetByteArrayAsync("http://xamarin.com/images/about/team.jpg");
        SaveBytesToFile(imageBytes, "team.jpg");
        ResultTextView.Text += "Downloaded the image.\n";
        DownloadedImageView.Image = UIImage.FromFile (localPath);

        ALAssetsLibrary library = new ALAssetsLibrary();
        var dict = new NSDictionary();
        var assetUrl = await library.WriteImageToSavedPhotosAlbumAsync
            (DownloadedImageView.Image.CGImage, dict);
        ResultTextView.Text += "Saved to album assetUrl = " + assetUrl + "\n";

        ResultTextView.Text += "\n\n\n" + contents; // just dump the entire HTML
        return length;
    } catch {
        // do something with error
        return -1;
    }
}
```

# Async / await

```
var webClient = new WebClient();

webClient.DownloadStringCompleted += (sender, e) => {
    if(e.Cancelled || e.Error != null) {
        // do something with error
    }
    string contents = e.Result;

    int length = contents.Length;
    InvokeOnMainThread (() => {
        ResultTextView.Text += "Downloaded the html and found out the length \n\n";
    });

    webClient.DownloadDataCompleted += (sender1, e1) => {
        if(e1.Cancelled || e1.Error != null) {
            // do something with error
        }
        SaveBytesToFile(e1.Result, "team.jpg");

        InvokeOnMainThread (() => {
            ResultTextView.Text += "Downloaded the image.\n";
            DownloadedImageView.Image = UIImage.FromFile (localPath);
        });

        ALAssetsLibrary library = new ALAssetsLibrary();
        var dict = new NSDictionary();
        library.WriteImageToSavedPhotosAlbum (DownloadedImageView.Image.CGImage, dict, (s2,e2) => {
            InvokeOnMainThread (() => {
                ResultTextView.Text += "Saved to album assetUrl\n";
            });
            if (downloaded != null)
                downloaded(length);
        });
    };

    webClient.DownloadDataAsync(new Uri("http://xamarin.com/images/about/team.jpg"));
};

webClient.DownloadStringAsync(new Uri("http://xamarin.com/"));
```

```
public async Task<int> DownloadHomepageAsync()
{
    try {
        var httpClient = new HttpClient();

        Task<string> contentsTask = httpClient.GetStringAsync("http://xamarin.com");

        string content = await contentsTask;

        int length = contents.Length;
        ResultTextView.Text += "Downloaded the html and found out the length.\n\n";

        byte[] imageBytes = await httpClient.GetByteArrayAsync("http://xamarin.com/images/about/team.jpg");
        SaveBytesToFile(imageBytes, "team.jpg");
        ResultTextView.Text += "Downloaded the image.\n";
        DownloadedImageView.Image = UIImage.FromFile (localPath);

        ALAssetsLibrary library = new ALAssetsLibrary();
        var dict = new NSDictionary();
        var assetUrl = await library.WriteImageToSavedPhotosAlbumAsync
            (DownloadedImageView.Image.CGImage, dict);
        ResultTextView.Text += "Saved to album assetUrl = " + assetUrl + "\n";

        ResultTextView.Text += "\n\n" + content; // just dump the entire HTML
        return length;
    } catch {
        // do something with error
        return -1;
    }
}
```

Después

Antes

# Async / await

## ■ Async:

- Poner el modificador “async” en los métodos y lambdas dónde se tenga que usar el “await”
- Usar el sufijo “Async” en el nombre del método. Ej: LoadAsync, SendAsync
- El método debe devolver una Task o Task<T>, dónde T es el tipo del valor a devolver. Ej: Task<int>, Task<bool>

## ■ Await:

- Usar la keyword “await” al llamar un método asíncrono
- Siempre dentro de un contexto que haya sido marcado cómo “async”

# Async / await

- Gestión de errores:
  - Mediante excepciones (try, catch, ...)
  - Si Task.State == Faulted
  - Devolviendo un bool indicando si la tarea se ha completado correctamente
- Además:
  - Se pueden cancelar tareas
  - Se puede hacer un seguimiento del progreso de la tarea
  - Se pueden agrupar y concatenar tareas