PROGRAMACIÓN MULTIMEDIA

Y

DISPOSITIVOS MÓVILES

UT1. Aspectos avanzados del lenguaje C#

Departamento de Informática y Comunicaciones

CFGS Desarrollo de Aplicaciones Multiplataforma

Segundo Curso

IES Virrey Morcillo

Villarrobledo (Albacete)

Índice

1. Espacios de nombres

Un **espacio de nombres** es un contenedor lógico en el que un grupo de uno o más identificadores únicos pueden existir. Un identificador definido en un espacio de nombres está asociado con ese espacio de nombres. El mismo identificador puede independientemente ser definido en múltiples espacios de nombres, eso es, el sentido asociado con un identificador definido en un espacio de nombres es independiente del mismo identificador declarado en otro espacio de nombres. Los lenguajes que manejan espacio de nombres especifican las reglas que determinan a qué espacio de nombres pertenece una instancia de un identificador.

Los programas en C# se organizan mediante espacios de nombres. Los espacios de nombres se utilizan como un sistema de organización interna para un programa y como un sistema de organización externa para presentar los elementos de programa que se exponen a otros programas.

La palabra clave **namespace** se usa para declarar un ámbito que contiene un conjunto de objetos relacionados. Puede usar un espacio de nombres para organizar los elementos de código y crear tipos únicos globales.

Los espacios de nombres tienen las propiedades siguientes:

- ✓ Organizan proyectos de código de gran tamaño.
- ✓ Se delimitan mediante el operador punto (.).
- ✓ La directiva using obvia la necesidad de especificar el nombre del espacio de nombres para cada clase.
- ✓ El espacio de nombres global es el espacio de nombres raíz: global::System siempre hará referencia al espacio de nombres System de .NET.

La directiva **using** permite el uso de tipos en un espacio de nombres de manera que no tenga que calificar el uso de un tipo en dicho espacio de nombres.

Los espacios de nombres se usan mucho en programación de C# de dos maneras. En primer lugar, .NET Framework usa espacios de nombres para organizar sus clases, de la siguiente manera:

```
System.Console.WriteLine("Hello World!");
```

System es un espacio de nombres, Console es una clase de ese espacio de nombres y WriteLine es un método de esa clase. La palabra clave using se puede usar para que no se necesite el nombre completo, como en el ejemplo siguiente:

```
using System;

Console.WriteLine("Hola");
Console.WriteLine("Mundo!");
```

En segundo lugar, declarar tus propios espacios de nombres puede ayudarte a controlar el ámbito de nombres de clase y método en proyectos de programación grandes.

La palabra clave contextual **global**, cuando se encuentra antes que el operador **doble dos puntos (::)**, hace referencia al espacio de nombres global, que es el espacio de nombres predeterminado para cualquier programa de C# y no tiene nombre.

La capacidad de tener acceso a un miembro en el espacio de nombres global es útil cuando el miembro puede estar oculto por otra entidad del mismo nombre.

Por ejemplo, en el código siguiente, Console se resuelve como TestApp.Console en lugar de como el tipo Console en el espacio de nombres System:

```
using System;

class TestApp {
    // Define a new class called 'System' to cause problems.
    public class System { }

    // Define a constant called 'Console' to cause more problems.
    const int Console = 7;
    const int number = 66;

    static void Main() {
        // The following line causes an error. It accesses TestApp.Console,
        // which is a constant.
        //Console.WriteLine(number);
    }
}
```

Usar System.Console todavía provoca un error porque el espacio de nombres System está oculto mediante la clase TestApp.System:

```
// The following line causes an error. It accesses TestApp.System,
// which does not have a Console.WriteLine method.
System.Console.WriteLine(number);
```

En cambio, puede solucionar este error con global::System.Console, como se muestra aquí:

```
// OK
global::System.Console.WriteLine(number);
```

Cuando el identificador izquierdo es global, la búsqueda del identificador derecho comienza en el espacio de nombres global.

Obviamente, no se recomienda crear sus propios espacios de nombres denominados System, y es poco probable que encuentre cualquier código donde haya ocurrido esto. En cambio, en proyectos más grandes, es una posibilidad muy real que pueda producirse una duplicación de espacio de nombres de una forma u otra. En estas situaciones, el calificador de espacio de nombres global es su garantía de que puede especificar el espacio de nombres raíz.

2. Interfaces

Una **interfaz** define un contrato que se puede implementar mediante clases y estructuras. Una interfaz puede contener métodos, propiedades, eventos e indexadores. Una interfaz no proporciona implementaciones de los miembros que define, simplemente determina los miembros que se deben proporcionar mediante clases o estructuras que implementan la interfaz.

Una interfaz se define mediante la palabra clave **interface**, como se muestra en el ejemplo siguiente:

```
interface IEquatable<T> {
    bool Equals(T obj);
}
```

El nombre de la estructura debe ser un nombre de identificador de C# válido. Por convención, los nombres de interfaz comienzan con una letra I mayúscula.

Cualquier clase o estructura que implementa la interfaz IEquatable<T> debe contener una definición para un método Equals que coincida con la firma que la interfaz especifica. Como resultado, puede contar con una clase que implementa IEquatable<T> para contener un método Equals con el que una instancia de la clase puede determinar si es igual a otra instancia de la misma clase.

La definición de IEquatable<T> no proporciona una implementación para Equals. La interfaz define sólo la firma. De esa manera, una interfaz en C# es similar a una clase abstracta en la que todos los métodos son abstractos. Sin embargo, una clase o estructura puede implementar varias interfaces, pero una clase solo puede heredar una clase única, ya sea abstracta o no.

Las interfaces pueden contener métodos, propiedades, eventos, indizadores o cualquier combinación de estos cuatro tipos de miembros.

Una interfaz no puede contener constantes, campos, operadores, constructores de instancias, finalizadores ni tipos.

Los miembros de interfaz son públicos automáticamente y no pueden incluir modificadores de acceso.

Cuando una clase o estructura implementa una interfaz, la clase o estructura debe proporcionar una implementación para todos los miembros que define la interfaz. La propia interfaz no proporciona ninguna funcionalidad que una clase o estructura puedan heredar de la misma la forma en que pueden heredar la funcionalidad de la clase base. Sin embargo, si una clase base implementa una interfaz, cualquier clase que se derive de la clase base hereda esta implementación.

En el siguiente ejemplo se muestra una implementación de la interfaz IEquatable<T>. La clase de implementación Car debe proporcionar una implementación del método Equals:

```
public class Car : IEquatable<Car> {
    public string Make { get; set; }
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car) {
        return this.Make == car.Make &&
               this.Model == car.Model &&
               this.Year == car.Year;
    }
}
```

Las interfaces se puede usar herencia múltiple. En el ejemplo siguiente, la interfaz IComboBox hereda de ITextBox y IListBox:

```
interface IControl {
    void Paint();
}

interface ITextBox : IControl {
    void SetText(string text);
}

interface IListBox : IControl {
    void SetItems(string[] items);
}
```

```
interface IComboBox : ITextBox, IListBox {
}
```

Las clases y las estructuras pueden implementar varias interfaces. En el ejemplo siguiente, la clase EditBox implementa IControl y IDataBound:

```
interface IDataBound {
    void Bind(Binder b);
}
public class EditBox : IControl, IDataBound {
    public void Paint() {
    }
    public void Bind(Binder b) {
    }
}
```

Cuando una clase implementan una interfaz determinada, las instancias de esa clase se pueden convertir implícitamente a ese tipo de interfaz. Por ejemplo:

```
EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;
```

3. Delegados

Los **delegados** proporcionan un mecanismo de enlace en tiempo de ejecución en .NET. Un enlace en tiempo de ejecución significa que se crea un algoritmo en el que el llamador también proporciona al menos un método que implementa parte del algoritmo.

Por ejemplo, se puede considerar la ordenación de una lista de estrellas en una aplicación de astronomía. Se puede decidir ordenar las estrellas por su distancia con respecto a la Tierra, por la magnitud de la estrella o por su brillo percibido. En todos estos casos, el método Ordenar() hace básicamente lo mismo, organiza los elementos en la lista en función de una comparación. El código que compara dos estrellas es diferente para cada uno de los criterios de ordenación.

Los delegados forman parte de los tipos de referencia del lenguaje C# y representa las referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Los delegados permiten tratar métodos como entidades que se puedan asignar a variables y se puedan pasar como parámetros.

Los delegados son similares al concepto de punteros de función en otros lenguajes, pero a diferencia de los punteros de funciones, los delegados están orientados a objetos y presentan seguridad de tipos.

Cuando se crea una instancia de un delegado, se puede asociar su instancia a cualquier método mediante una signatura compatible y un tipo de valor devuelto. Se puede invocar o llamar al método a través de la instancia del delegado.

El tipo de un delegado se define por el nombre del delegado. En el ejemplo siguiente, se declara un delegado denominado Del que puede encapsular un método que toma una string como argumento y devuelve void:

```
public delegate void Del(string message);
```

Normalmente, un objeto delegado se construye al proporcionar el nombre del método que el delegado encapsulará o con un método anónimo. Una vez que se crea una instancia de delegado, el delegado pasará al método una llamada de método realizada al delegado. Los parámetros pasados al

delegado por el autor de la llamada se pasan a su vez al método, y el valor devuelto desde el método, si lo hubiera, es devuelto por el delegado al autor de la llamada. Esto se conoce como invocar al delegado. Un delegado con instancias se puede invocar como si fuera el propio método encapsulado. Por ejemplo:

```
delegate int Operar(int x1, int x2);

class Program {
    public int Sumar(int x, int y) {
        return x + y;
    }

    public int Restar(int x, int y) {
        return x - y;
    }

    static void Main(string[] args) {
        Program p = new Program();
        Console.WriteLine("Suma y resta de dos valores llamando directamente a los
                           métodos.");
        Console.WriteLine(p.Sumar(10, 5));
        Console.WriteLine(p.Restar(10, 5));
        Operar delegado = p.Sumar;
        Console.WriteLine("Suma y resta de dos valores llamando a los métodos a través
                           de delegados");
        Console.WriteLine(delegado(10, 5));
        delegado = p.Restar;
        Console.WriteLine(delegado(10, 5));
        Console.ReadKey();
    }
}
```

Se declara el delegado Operar que recibe dos parámetros de tipo int y retorna un int. En la clase Program se definen dos métodos Sumar y Restar que tienen una firma idéntica al delegado Operar. En el método Main se define un objeto de la clase Program y se llama directamente sus dos métodos. Lo nuevo aparece cuando se define un objeto del tipo delegado Operar y se le carga la referencia del método Sumar. La variable delegado tiene ahora la referencia del método Sumar y se puede llamar a dicho método a través del delegado. Luego se modifica la variable delegado y se almacena la referencia al método Restar y se llama a dicho método a través del delegado.

Un método puede recibir como parámetro un delegado y a partir de esto llamar al método que almacena el delegado:

```
delegate int Operar(int x1, int x2);

class Program {
    public int Sumar(int x, int y) {
        return x + y;
    }

    public int Restar(int x, int y) {
        return x - y;
    }

    public void operacion(Operar d, int x, int y) {
```

```
        Console.WriteLine(d(10, 5));
    }

    static void Main(string[] args) {
        Program p = new Program();
        Console.WriteLine("Suma y resta de 10 y 5.");
        p.operacion(p.Sumar, 10, 5);
        p.operacion(p.Restar, 10, 5);
        Console.ReadKey();
    }
}
```

Se declara el delegado. Se declara la clase Program y los dos métodos que suman y restan dos enteros. Lo nuevo aparece en el método operacion que recibe en el primer parámetro un delegado de tipo Operar y dos enteros. Dentro del algoritmo del método se llama al método que almacena el delegado. En la función Main después de crear un objeto de la clase Program se procede a llamar solo al método operacion y pasar en el primer parámetro la referencia del método Sumar o Restar según la actividad que se necesita hacer con los dos enteros restantes.

Una propiedad interesante y útil de un delegado es que no sabe ni necesita conocer la clase del método al que hace referencia; lo único que importa es que el método al que se hace referencia tenga los mismos parámetros y el tipo de valor devuelto que el delegado.

4. Eventos

Un **evento** es un mensaje que envía un objeto cuando ocurre una acción. La acción podría deberse a la interacción del usuario, como hacer clic en un botón, o podría derivarse de cualquier otra lógica del programa, como el cambio del valor de una propiedad. El objeto que provoca el evento se conoce como emisor del evento. El emisor del evento no sabe qué objeto o método recibirá y controlará los eventos que genera.

El evento normalmente es un miembro del emisor del evento. Por ejemplo, el evento Click es un miembro de la clase Button, y el evento PropertyChanged es un miembro de la clase que implementa la interfaz INotifyPropertyChanged.

En otras palabras, un evento es un miembro que permite que una clase u objeto proporcionen notificaciones. Un evento se declara como un campo, excepto por el hecho de que la declaración incluye una palabra clave **event** y el tipo debe ser un tipo delegado.

Dentro de una clase que declara un miembro de evento, el evento se comporta como un campo de un tipo delegado, siempre que el evento no sea abstracto y no declare descriptores de acceso. El campo almacena una referencia a un delegado que representa los controladores de eventos que se han agregado al evento. Si no existen controladores de eventos, el campo es null.

El publicador determina el momento en el que se genera un evento, los suscriptores determinan la acción que se lleva a cabo en respuesta al evento. Un evento puede tener varios suscriptores. Un suscriptor puede controlar varios eventos de varios publicadores. Nunca se generan eventos que no tienen suscriptores.

En la biblioteca de clases de .NET Framework, los eventos se basan en el delegado EventHandler y en la clase base EventArgs.

4.1. Publicar eventos

Todos los eventos de la biblioteca de clases .NET Framework se basan en el delegado EventHandler, que se define de la siguiente manera:

```
public delegate void EventHandler(object sender, EventArgs e);
```

Para publicar eventos basados en el patrón EventHandler se deben seguir los pasos siguientes:

1. Declara la clase de los datos personalizados en un ámbito que sea visible para las clases de publicador y suscriptor. Luego, agrega los miembros necesarios para almacenar los datos de eventos personalizados. En este ejemplo se devuelve una cadena simple.

```
public class CustomEventArgs : EventArgs {
    public CustomEventArgs(string s) {
        msg = s;
    }
    private string msg;
    public string Message {
        get { return msg; }
    }
}
```

2. Declara un delegado en la clase de publicación. Asígnale un nombre que acabe por EventHandler. El segundo parámetro especifica el tipo EventArgs personalizado.

```
public delegate void CustomEventHandler(object sender, CustomEventArgs a);
```

3. Declara el evento en la clase de publicación llevando a cabo uno de los siguientes pasos:
   a. Si no tiene ninguna clase EventArgs personalizada, el tipo Event será el delegado EventHandler no genérico. No es necesario declarar el delegado, porque ya está declarado en el espacio de nombres System que se incluye al crear el proyecto de C#. Agrega el código siguiente a la clase de publicador:

```
public event EventHandler RaiseCustomEvent;
```

   b. Si usas la versión no genérica de EventHandler y tienes una clase personalizada derivada de EventArgs, declara el evento dentro de la clase de publicación y use el delegado del paso 2 como tipo:

```
public event CustomEventHandler RaiseCustomEvent;
```

   c. Si usas la versión genérica, no necesitas ningún delegado personalizado. En su lugar, en la clase de publicación, especifica el tipo de evento como EventHandler<CustomEventArgs>, sustituyendo el nombre de su propia clase que aparece entre corchetes angulares:

```
public event EventHandler<CustomEventArgs> RaiseCustomEvent;
```

4.2. Suscripción a un evento

Para suscribirse a eventos mediante programación se deben seguir los pasos siguientes:

1. Define un método de controlador de eventos cuya firma coincida con la firma de delegado del evento. Por ejemplo, si el evento se basa en el tipo de delegado EventHandler, el siguiente código representa el código auxiliar del método:

```
void HandleCustomEvent(object sender, CustomEventArgs a) {
    // Do something useful here.
```

```
}
```

2. Utiliza el operador de suma y asignación (+=) para asociar el controlador de eventos al evento. En el ejemplo siguiente, se asume que un objeto denominado publisher tiene un evento denominado RaiseCustomEvent. Observe que la clase de suscriptor necesita una referencia a la clase de editor para suscribirse a sus eventos:

```
publisher.RaiseCustomEvent += HandleCustomEvent;
```

4.3. Cancelar una suscripción

Para impedir que se invoque el controlador de eventos cuando se produce el evento, puede cancelar la suscripción al evento. Para evitar que se pierdan recursos, debe cancelar la suscripción a los eventos antes de eliminar un objeto suscriptor.

Hasta que se cancela la suscripción a un evento, el delegado multidifusión subyacente al evento en el objeto de publicación tiene una referencia al delegado que encapsula el controlador de eventos del suscriptor. Mientras el objeto de publicación mantenga esa referencia, la recolección de elementos no utilizados no eliminará el objeto suscriptor.

Para cancelar la suscripción a un evento se deben seguir los pasos siguientes:

1. Utiliza el operador de resta y asignación (-=) para cancelar la suscripción a un evento:

```
publisher.RaiseCustomEvent -= HandleCustomEvent;
```

5. Genéricos

Los **genéricos** se han agregado a la versión 2.0 del lenguaje C# y Common Language Runtime (CLR). Los genéricos introducen en .NET Framework el concepto de parámetros de tipo, lo que le permite diseñar clases y métodos que aplazan la especificación de uno o varios tipos hasta que el código de cliente declare y cree una instancia de la clase o el método.

Las clases y métodos genéricos combinan reusabilidad, seguridad de tipos y eficacia de una manera en que sus homólogos no genéricos no pueden. Los genéricos se usan frecuentemente con colecciones y los métodos que funcionan en ellas. La versión 2.0 de la biblioteca de clases .NET Framework proporciona un nuevo espacio de nombres, System.Collections.Generic, que contiene varias clases de colección nuevas basadas en genéricos.

También se pueden crear tipos y métodos genéricos personalizados para proporcionar soluciones y patrones de diseño generalizados propios con seguridad de tipos y eficaces.

5.1. Parámetros de tipos genéricos

En un tipo genérico o en una definición de método, un parámetro de tipo es un marcador de posición para un tipo específico que un cliente especifica cuando crean instancias de una variable del tipo genérico.

Por ejemplo, al usar un parámetro de tipo genérico T puede escribir una clase única que otro código de cliente puede usar sin incurrir en el costo o riesgo de conversiones en tiempo de ejecución u operaciones de conversión boxing, como se muestra aquí:

```
// Declare the generic class.
public class GenericList<T> {
    public void Add(T input) { }
}
```

```
class TestGenericList {
    private class ExampleClass { }
    static void Main() {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

5.2. Clases genéricas

Las clases genéricas encapsulan operaciones que no son específicas de un tipo de datos determinado. El uso más común de las clases genéricas es con colecciones como listas vinculadas, tablas hash, pilas, colas y árboles, entre otros. Las operaciones como la adición y eliminación de elementos de la colección se realizan básicamente de la misma manera independientemente del tipo de datos que se almacenan.

5.3. Interfaces genéricas

A menudo es útil definir interfaces para las clases de colección genéricas o para las clases genéricas que representan los elementos de la colección. Lo preferible para las clases genéricas es usar interfaces genéricas, como IComparable<T> en lugar de IComparable, para evitar las operaciones de conversión boxing y unboxing en los tipos de valor.

5.4. Métodos genéricos

Un método genérico es un método que se declara con parámetros de tipo, de la manera siguiente:

```
static void Swap<T>(ref T lhs, ref T rhs) {
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

En el siguiente ejemplo de código se muestra una manera de llamar al método con int para el argumento de tipo:

```
public static void TestSwap() {
    int a = 1;
    int b = 2;

    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
```

También puede omitir el argumento de tipo y el compilador lo deducirá. La siguiente llamada a Swap es equivalente a la llamada anterior:

```
Swap(ref a, ref b);
```

6. Control de excepciones

Las características de **control de excepciones** del lenguaje C# le ayudan a afrontar cualquier situación inesperada o excepcional que se produce cuando se ejecuta un programa. El control de excepciones usa las palabras clave try, catch y finally para intentar realizar acciones que pueden no completarse correctamente, para controlar errores cuando decide que es razonable hacerlo y para limpiar recursos más adelante. Las excepciones las puede generar Common Language Runtime (CLR), .NET Framework, cualquier biblioteca de terceros o el código de aplicación. Las excepciones se crean mediante el uso de la palabra clave throw.

En este ejemplo, un método prueba a hacer la división entre cero y detecta el error. Sin el control de excepciones, este programa finalizaría con un error DivideByZeroException no controlada:

```
class ExceptionTest {
    static double SafeDivision(double x, double y) {
        if (y == 0)
            throw new System.DivideByZeroException();
        return x / y;
    }
    static void Main() {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result = 0;

        try {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
        catch (DivideByZeroException e) {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

Las excepciones tienen las siguientes propiedades:

- ✓ Las excepciones son tipos que derivan en última instancia de System.Exception.
- ✓ Usa un bloque try alrededor de las instrucciones que pueden producir excepciones.
- ✓ Una vez que se produce una excepción en el bloque try, el flujo de control salta al primer controlador de excepciones asociado que está presente en cualquier parte de la pila de llamadas. En C#, la palabra clave catch se utiliza para definir un controlador de excepciones.
- ✓ Si no hay ningún controlador de excepciones para una excepción determinada, el programa deja de ejecutarse con un mensaje de error.
- ✓ No detectes una excepción a menos que puedas controlarla y dejar la aplicación en un estado conocido. Si se detecta System.Exception, reinícialo con la palabra clave throw al final del bloque catch.

- ✓ Si un bloque catch define una variable de excepción, puedes utilizarla para obtener más información sobre el tipo de excepción que se ha producido.
- ✓ Las excepciones puedes generarlas explícitamente un programa con la palabra clave throw.
- ✓ Los objetos de excepción contienen información detallada sobre el error, como el estado de la pila de llamadas y una descripción de texto del error.
- ✓ El código de un bloque finally se ejecuta incluso si se produce una excepción. Usa un bloque finally para liberar recursos, por ejemplo, para cerrar las secuencias o los archivos que se abrieron en el bloque try.

7. Programación asíncrona

El modelo de programación asincrónica de tareas (TAP) es una abstracción del código asincrónico. El código se escribe como una secuencia de instrucciones, como es habitual. Puede leerlo como si cada instrucción se completase antes de comenzar la siguiente. El compilador realiza una serie de transformaciones, debido a que algunas de estas instrucciones podrían empezar a funcionar y devolver una clase Task que representa el trabajo en curso.

Para explicar cómo se prepara un desayuno, probablemente escribirás unas instrucciones parecidas a las que se recogen en la lista siguiente:

1) Sirve una taza de café.
2) Calienta una sartén y fría dos huevos.
3) Fríe tres lonchas de beicon.
4) Tuesta dos rebanadas de pan.
5) Unta el pan con mantequilla y mermelada.
6) Sirve un vaso de zumo de naranja.

Si tienes experiencia en la cocina, lo más probable es que ejecutes estas instrucciones de forma asincrónica. Primero, calentarás la sartén para los huevos e irás friendo el beicon. Después, pondrás el pan en la tostadora y empezarás a freír los huevos. En cada paso del proceso, iniciarás una tarea y volverás la atención a las tareas que tiene pendientes.

La preparación del desayuno es un buen ejemplo de un trabajo asincrónico que no es paralelo. Una persona (o un subproceso) puede controlar todas estas tareas. Siguiendo con la analogía del desayuno, una persona puede preparar el desayuno asincrónicamente si comienza la tarea siguiente antes de que finalice la anterior. Los alimentos se cocinan tanto si una persona supervisa el proceso como si no. En cuanto se empieza a calentar la sartén para los huevos, se puede comenzar a freír el beicon. Una vez que el beicon se esté haciendo, se puede poner el pan en la tostadora.

En el caso de un algoritmo paralelo, necesitaría varios cocineros (o subprocesos). Uno se encargaría de los huevos, otro del beicon, etc. Cada uno de ellos se centraría en una sola tarea. Un cocinero (o subproceso) se bloqueará al esperar asincrónicamente a que el beicon se dore para darle la vuelta, o al esperar a que las tostadas estén listas.

Piensa ahora en estas mismas instrucciones escritas como instrucciones de C#:

```
static void Main(string[] args) {
    Coffee cup = PourCoffee();
```

```
    Console.WriteLine("coffee is ready");
    Egg eggs = FryEggs(2);
    Console.WriteLine("eggs are ready");
    Bacon bacon = FryBacon(3);
    Console.WriteLine("bacon is ready");
    Toast toast = ToastBread(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");
    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");

    Console.WriteLine("Breakfast is ready!");
}
```

Los equipos no interpretan estas instrucciones de la misma manera que las personas. El equipo se bloqueará en cada instrucción hasta que el trabajo se complete antes de pasar a la instrucción siguiente. Podría decirse que esto da lugar a un desayuno poco satisfactorio. Las tareas posteriores no se pueden iniciar mientras no se completen las anteriores. Así pues, se tardará mucho más en preparar el desayuno y algunos alimentos se habrán enfriado incluso antes de servirse.

Si quieres que el equipo ejecute las instrucciones anteriores de forma asincrónica, debe escribir **código asincrónico**.

Estas cuestiones son importantes para los programas que se escriben hoy en día. Al escribir programas cliente, nos interesa que la interfaz de usuario responda a la entrada del usuario. La aplicación no debería hacer que un teléfono parezca congelado mientras descarga datos de la Web. Al escribir programas de servidor, no nos conviene que los subprocesos se bloqueen. La intención es que puedan atender también otras solicitudes.

7.1. Uso de await para evitar los bloqueos

Empecemos por actualizar este código para que el subproceso no se bloquee mientras se ejecutan las tareas. La palabra clave **await** proporciona un modo sin bloqueo para iniciar una tarea y, después, proseguir la ejecución cuando dicha tarea se complete. Una versión asincrónica sencilla del código para preparar el desayuno tendría un aspecto parecido al del fragmento siguiente:

```
static async Task Main(string[] args) {
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    Egg eggs = await FryEggs(2);
    Console.WriteLine("eggs are ready");
    Bacon bacon = await FryBacon(3);
    Console.WriteLine("bacon is ready");
    Toast toast = await ToastBread(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");
    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");

    Console.WriteLine("Breakfast is ready!");
}
```

Este código no produce un bloqueo mientras se cocinan los huevos o el beicon, pero tampoco inicia otras tareas. Es decir, pondría el pan en la tostadora y se quedaría esperando a que estuviera listo, pero, por lo menos, si alguien reclamara su atención, le haría caso. En un restaurante en el que se atienden varios pedidos, el cocinero empezaría a preparar otro desayuno mientras se hace el primero.

El subproceso que se encarga del desayuno ya no se bloquearía mientras espera por las tareas iniciadas que aún no han terminado. En algunas aplicaciones, lo único que se necesita es este cambio. Una aplicación de interfaz gráfica de usuario seguirá respondiendo al usuario con este único cambio. Aun así, para este escenario, necesitas algo más. No nos interesa que todas las tareas de componente se ejecuten secuencialmente. Es mejor iniciar cada una de estas tareas sin esperar a que la tarea anterior se complete.

7.2. Inicio simultáneo de tareas

En muchos escenarios, necesitas iniciar varias tareas independientes de inmediato. Después, a medida que finalice cada tarea, podrás seguir con el trabajo que esté listo. Siguiendo con la analogía del desayuno, esta es la manera más rápida de prepararlo. Además, de este modo, todo estará listo aproximadamente en el mismo momento. Así podrás disfrutar de un desayuno caliente.

La clase System.Threading.Tasks.Task y los tipos relacionados se pueden usar para razonar sobre las tareas que están en curso. Esto nos permite escribir código que se asemeje más a la manera en que realmente se prepara el desayuno. Para ello, cocinarás los huevos, el beicon y las tostadas al mismo tiempo. Como cada uno de estos elementos requiere una acción, nos ocuparemos de esa tarea, nos encargaremos de la acción siguiente y esperaremos por todo lo que necesite nuestra atención.

Comenzaremos una tarea y conservará el objeto Task que representa el trabajo. Se llevará a cabo una instrucción await para esperar por cada tarea antes de trabajar con su resultado.

Se realizarán estos cambios en el código del desayuno. El primer paso consiste en almacenar las tareas de las operaciones cuando se inician, en lugar de esperar por ellas:

```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");
Task<Egg> eggTask = FryEggs(2);
Egg eggs = await eggTask;
Console.WriteLine("eggs are ready");
Task<Bacon> baconTask = FryBacon(3);
Bacon bacon = await baconTask;
Console.WriteLine("bacon is ready");
Task<Toast> toastTask = ToastBread(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");
Juice oj = PourOJ();
Console.WriteLine("oj is ready");

Console.WriteLine("Breakfast is ready!");
```

Después, puedes mover las instrucciones await del beicon y los huevos al final del método, antes de servir el desayuno:

```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");
Task<Egg> eggTask = FryEggs(2);
```

```
Task<Bacon> baconTask = FryBacon(3);
Task<Toast> toastTask = ToastBread(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");
Juice oj = PourOJ();
Console.WriteLine("oj is ready");

Egg eggs = await eggTask;
Console.WriteLine("eggs are ready");
Bacon bacon = await baconTask;
Console.WriteLine("bacon is ready");

Console.WriteLine("Breakfast is ready!");
```

El código anterior funciona mejor. Iniciará todas las tareas asincrónicas a la vez y esperará por una tarea solo cuando necesite los resultados. El código anterior se parece al código de una aplicación web que realiza solicitudes a diferentes microservicios y, después, combina los resultados en una sola página. Se podrán realizar todas las solicitudes de inmediato y, luego, se llevará a cabo una instrucción await para esperar por todas esas tareas y componer la página web.

7.3. Composición con tareas

Ya tienes todo listo al mismo tiempo para el desayuno, excepto las tostadas. La preparación de las tostadas es la composición de una operación asincrónica (tostar el pan) y varias operaciones sincrónicas (untar la mantequilla y la mermelada). La actualización de este código ilustra un concepto importante: la composición de una operación asincrónica seguida de un trabajo sincrónico es una operación asincrónica. Dicho de otra manera, si una parte cualquiera de una operación es asincrónica, toda la operación es asincrónica.

En el código anterior se muestra que se puede usar un objeto Task o Task<TResult> para conservar tareas en ejecución. Lleva a cabo una instrucción await para esperar por una tarea a fin de poder usar su resultado.

El siguiente paso consiste en crear métodos que representan la combinación de otro trabajo. Antes de servir el desayuno, quiere esperar por la tarea que representa tostar el pan antes de untar la mantequilla y la mermelada. Puede representar este trabajo con el código siguiente:

```
async Task<Toast> makeToastWithButterAndJamAsync(int number) {
    var plainToast = await ToastBreadAsync(number);
    ApplyButter(plainToast);
    ApplyJam(plainToast);
    return plainToast;
}
```

El método anterior tiene el modificador async en su firma, lo que indica al compilador que este método incluye una instrucción await, es decir, que contiene operaciones asincrónicas. Este método representa la tarea que tuesta el pan y, después, agrega la mantequilla y la mermelada. El método devuelve un objeto Task<TResult> que representa la composición de estas tres operaciones. El bloque principal de código se convierte en lo siguiente:

```
static async Task Main(string[] args) {
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
```

```
    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = makeToastWithButterAndJamAsync(2);

    var eggs = await eggsTask;
    Console.WriteLine("eggs are ready");
    var bacon = await baconTask;
    Console.WriteLine("bacon is ready");
    var toast = await toastTask;
    Console.WriteLine("toast is ready");
    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");

    Console.WriteLine("Breakfast is ready!");

    async Task<Toast> makeToastWithButterAndJamAsync(int number) {
        var plainToast = await ToastBreadAsync(number);
        ApplyButter(plainToast);
        ApplyJam(plainToast);
        return plainToast;
    }
}
```

7.4. Espera de la finalización de las tareas de forma eficaz

La serie de instrucciones await al final del código anterior se puede mejorar mediante el uso de métodos de la clase Task. Una de estas API es **WhenAll**, que devuelve un objeto Task que se completa cuando finalizan todas las tareas de la lista de argumentos, como se muestra en el código siguiente:

```
await Task.WhenAll(eggTask, baconTask, toastTask);
Console.WriteLine("eggs are ready");
Console.WriteLine("bacon is ready");
Console.WriteLine("toast is ready");
Console.WriteLine("Breakfast is ready!");
```

Otra opción consiste en usar WhenAny, que devuelve un objeto Task<Task> que se completa cuando finaliza cualquiera de sus argumentos. Puede esperar por la tarea devuelta, con la certeza de saber que ya ha terminado. En el código siguiente se muestra cómo se puede usar WhenAny para esperar a que la primera tarea finalice y, después, procesar su resultado. Después de procesar el resultado de la tarea completada, quítela de la lista de tareas que se pasan a WhenAny:

```
var allTasks = new List<Task> { eggsTask, baconTask, toastTask };

while (allTasks.Any()) {
    Task finished = await Task.WhenAny(allTasks);
    if (finished == eggsTask) {
        Console.WriteLine("eggs are ready");
        allTasks.Remove(eggsTask);
        var eggs = await eggsTask;
    }
    else if (finished == baconTask) {
        Console.WriteLine("bacon is ready");
        allTasks.Remove(baconTask);
        var bacon = await baconTask;
    }
    else if (finished == toastTask) {
```

```
        Console.WriteLine("toast is ready");
        allTasks.Remove(toastTask);
        var toast = await toastTask;
    }
    else
        allTasks.Remove(finished);
}

Console.WriteLine("Breakfast is ready!");
```

Después de todos estos cambios, la versión final de Main tiene un aspecto similar al código siguiente:

```
static async Task Main(string[] args) {
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = makeToastWithButterAndJamAsync(2);

    var allTasks = new List<Task> { eggsTask, baconTask, toastTask };

    while (allTasks.Any()) {
        Task finished = await Task.WhenAny(allTasks);
        if (finished == eggsTask) {
            Console.WriteLine("eggs are ready");
            allTasks.Remove(eggsTask);
            var eggs = await eggsTask;
        }
        else if (finished == baconTask) {
            Console.WriteLine("bacon is ready");
            allTasks.Remove(baconTask);
            var bacon = await baconTask;
        }
        else if (finished == toastTask) {
            Console.WriteLine("toast is ready");
            allTasks.Remove(toastTask);
            var toast = await toastTask;
        }
        else
            allTasks.Remove(finished);
    }
    Console.WriteLine("Breakfast is ready!");

    async Task<Toast> makeToastWithButterAndJamAsync(int number) {
        var plainToast = await ToastBreadAsync(number);
        ApplyButter(plainToast);
        ApplyJam(plainToast);
        return plainToast;
    }
}
```

Este código final es asincrónico. Refleja con más precisión la manera en que una persona prepara un desayuno. Las acciones principales siguen siendo claras cuando se lee el código. De hecho, se puede leer como si se tratara de las instrucciones para preparar el desayuno que se indican al principio. Las características del lenguaje para async y await proporcionan la traducción que cualquier persona haría

para seguir las instrucciones escritas, a saber: las tareas deben iniciarse a medida que sea posible y debe evitarse el bloqueo por esperar a que se completen las tareas.