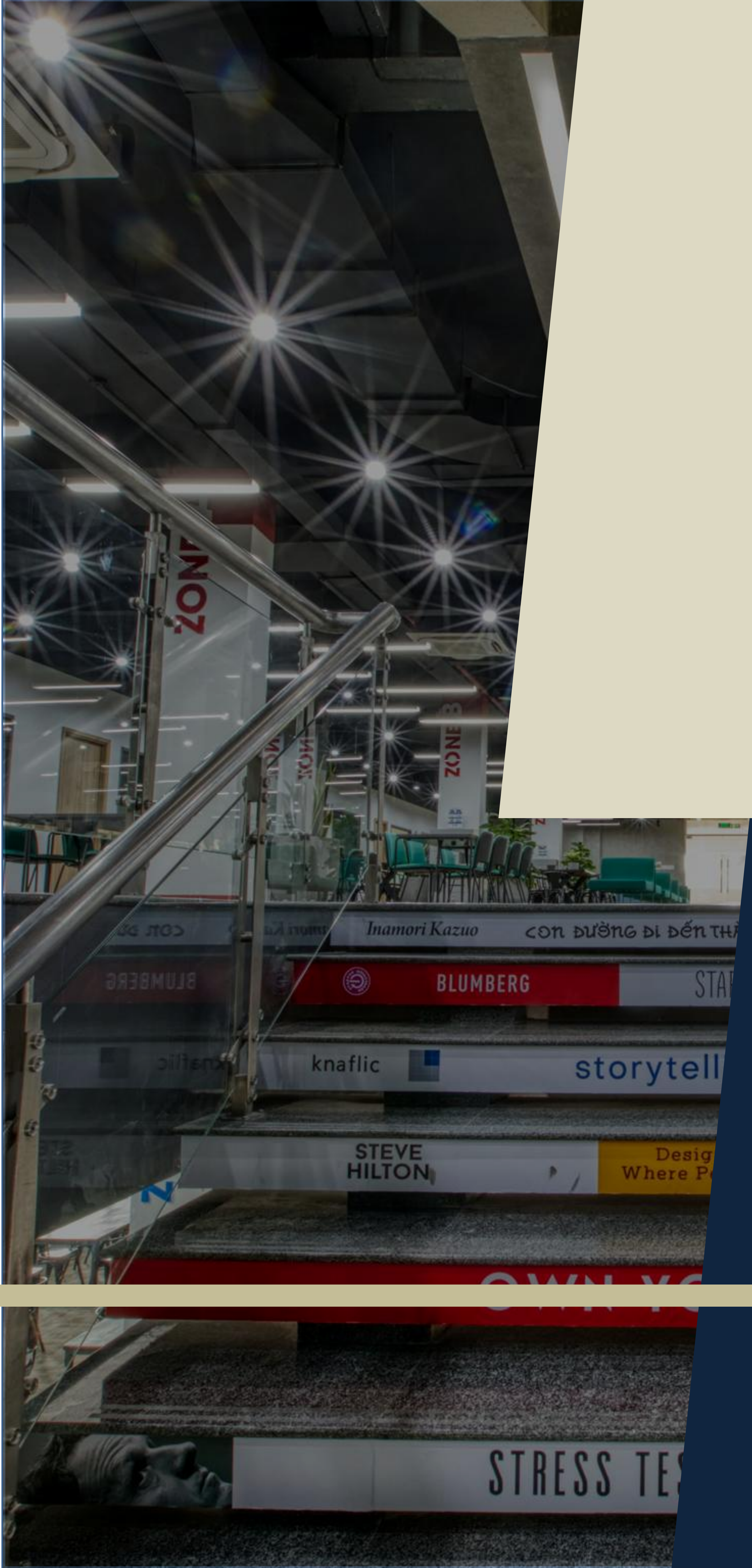


TRƯỜNG ĐẠI HỌC QUỐC TẾ MIỀN ĐÔNG
EASTERN INTERNATIONAL UNIVERSITY

Backend Development

Chapter 1: C# Programming languages

C# Advanced features

C# Version

C# version 12
C# version 11
C# version 10
C# version 9
C# version 8.0
C# version 7.3
C# version 7.2
C# version 7.1
C# version 7.0
C# version 6.0
C# version 5.0
C# version 4.0
C# version 3.0
C# version 2.0
C# version 1.2
C# version 1.0

12 version

C# Version 1

- Classes
- Structs
- Interfaces
- Events
- Properties
- Delegates
- Operators and expressions
- Statements
- Attributes

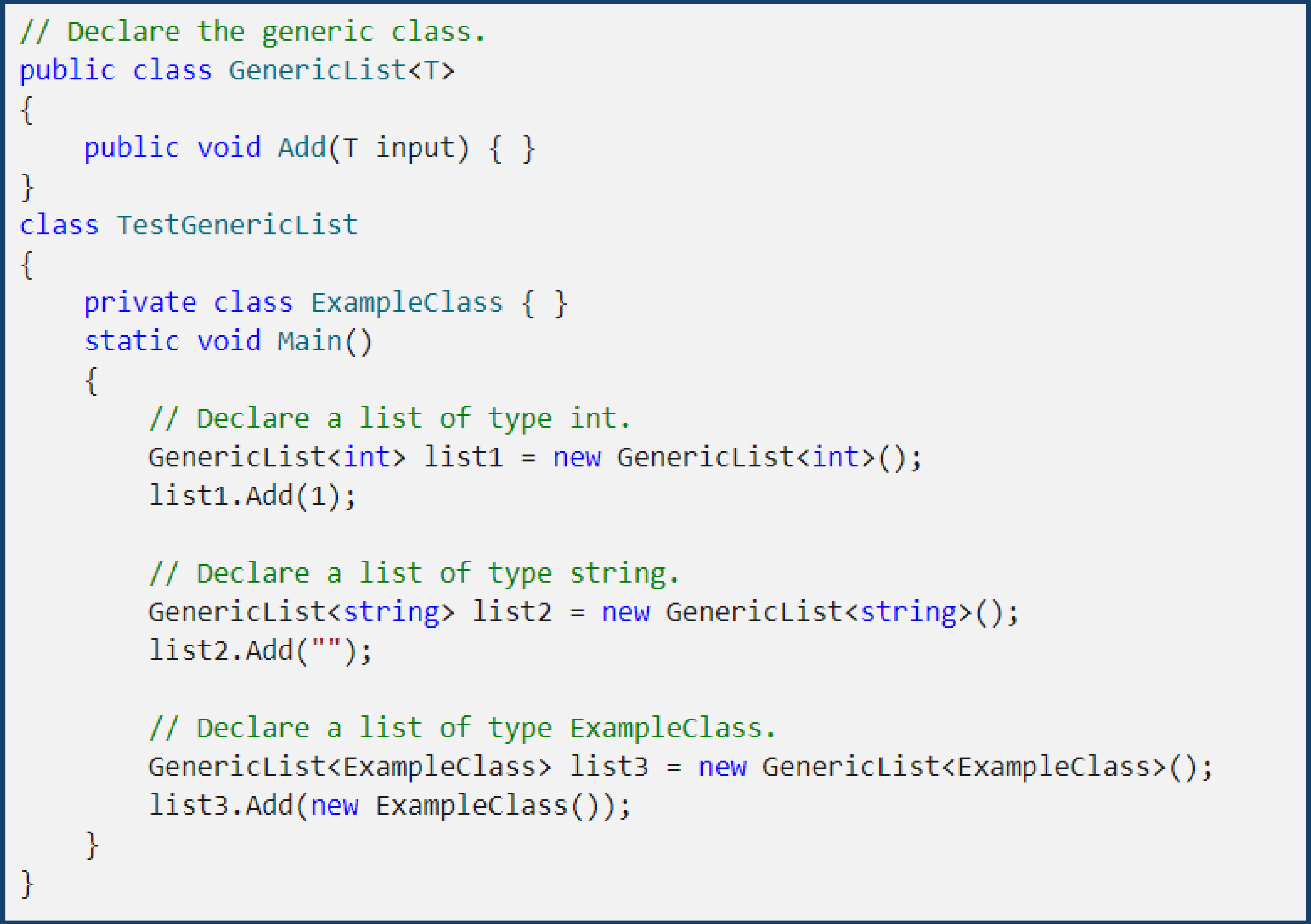
C# Version 2

- Generics
- Partial types
- Anonymous methods
- Nullable value types
- Iterators
- Covariance and contravariance

- Getter/setter separate accessibility
- Method group conversions (delegates)
- Static classes
- Delegate inference

C# Version 2 - Generics

Generics introduces the concept of type parameters to .NET, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code.



```csharp
// 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());
    }
}
```

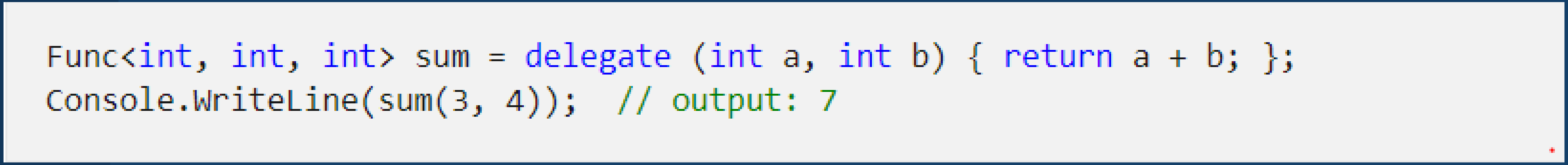
C# Version 2 - Partial types

It is possible to split the definition of a class, a struct, an interface or a method over two or more source files

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }

    partial class Nested
    {
        void Test2() { }
    }
}
```

C# versions 2 – Anonymous Methods

An anonymous function is an "inline" statement or expression that can be used wherever a delegate type is expected.
-> Used it to initialize a named delegate or pass it instead of a named delegate type as a method parameter.

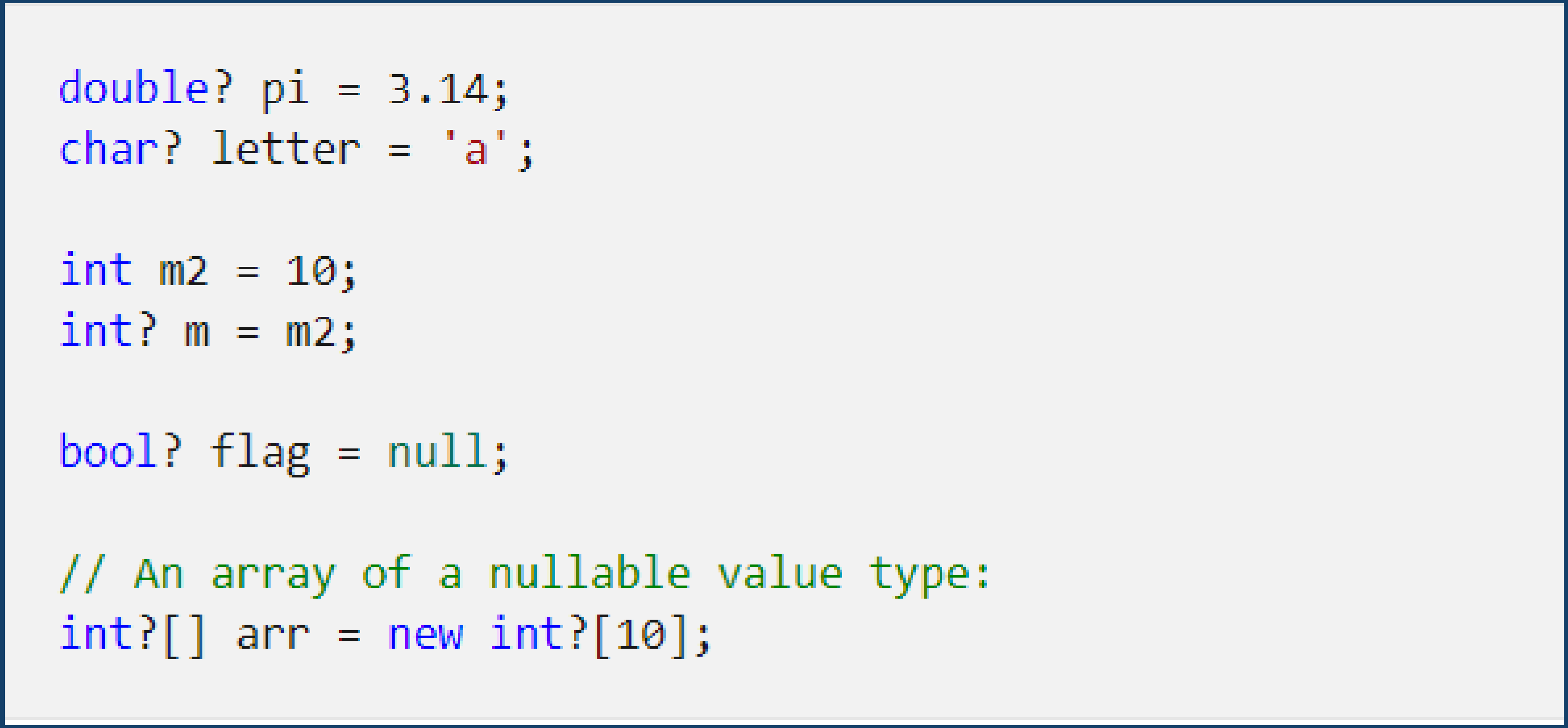


```csharp
Func<int, int, int> sum = delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(3, 4));  // output: 7
```

Beginning with C# 3, lambda expressions provide a more concise and expressive way to create an anonymous function. Use the **=> operator** to construct a lambda expression:

C# versions 2 – Nullable value types

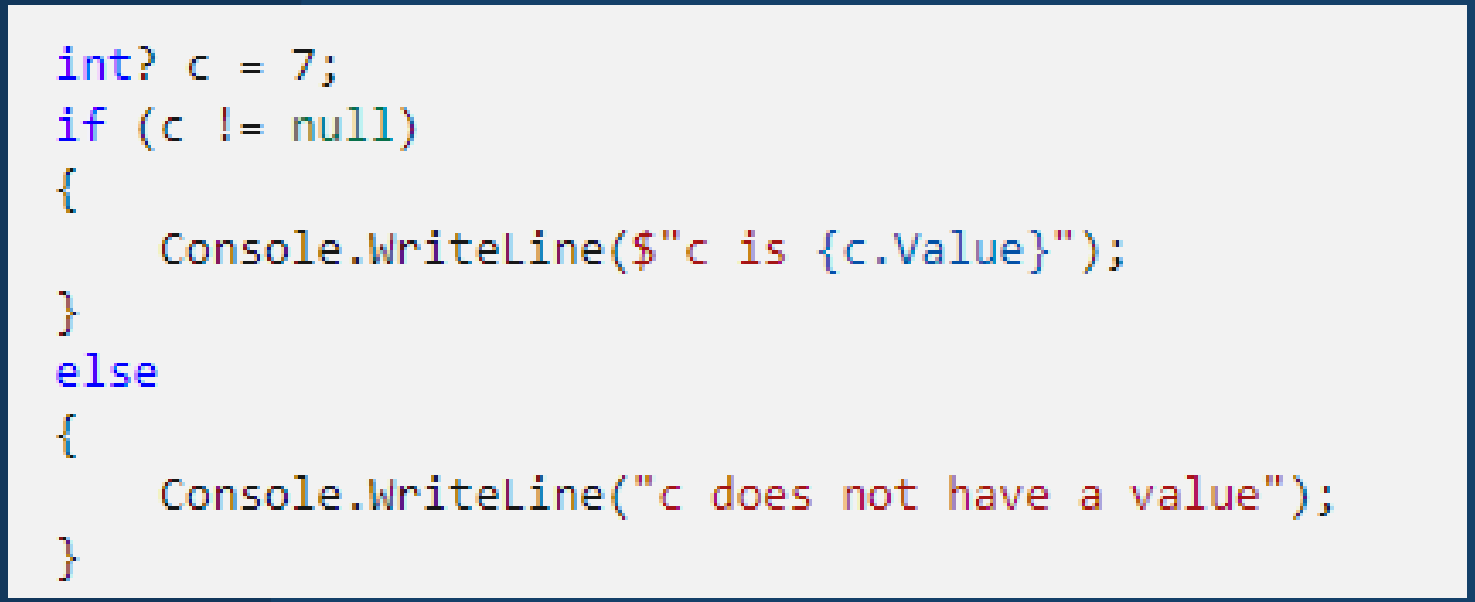
A nullable value type T? represents all values of its underlying value type T and an additional null value with default value is null.



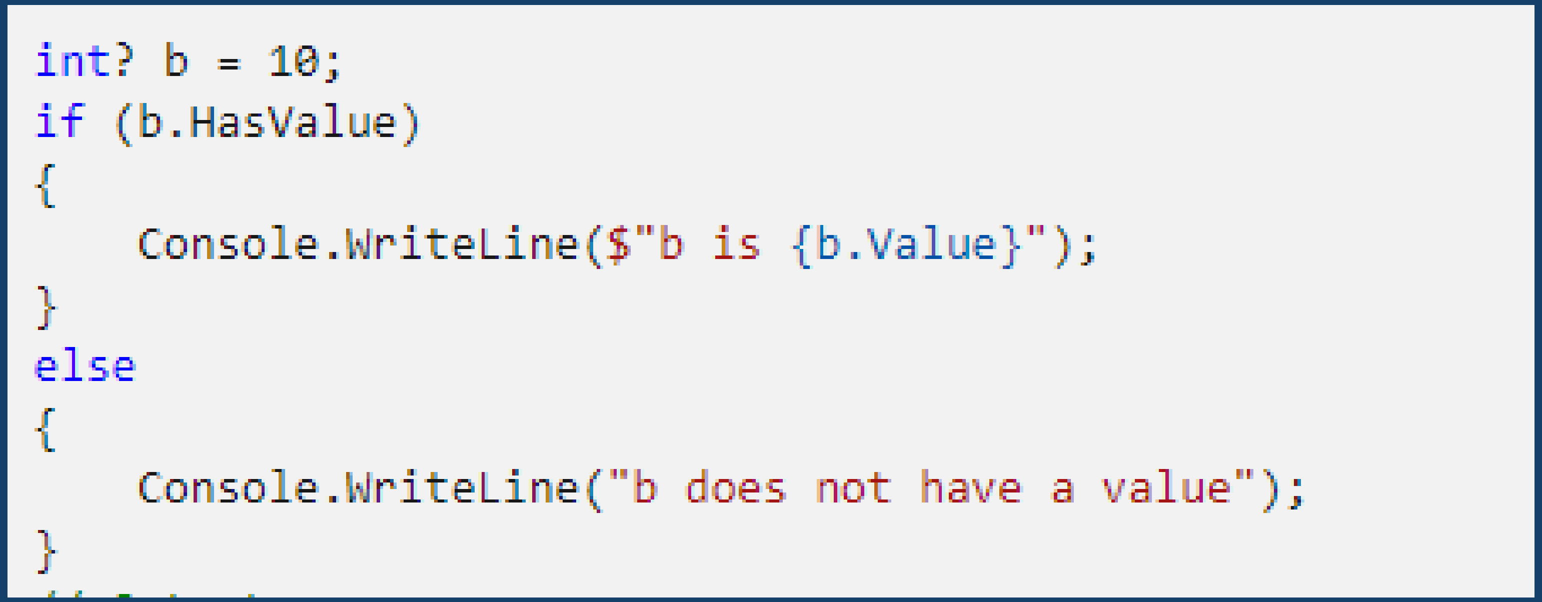
```csharp
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int?[] arr = new int?[10];
```

C# versions 2 – Nullable value types

Examination of an instance of a nullable value type

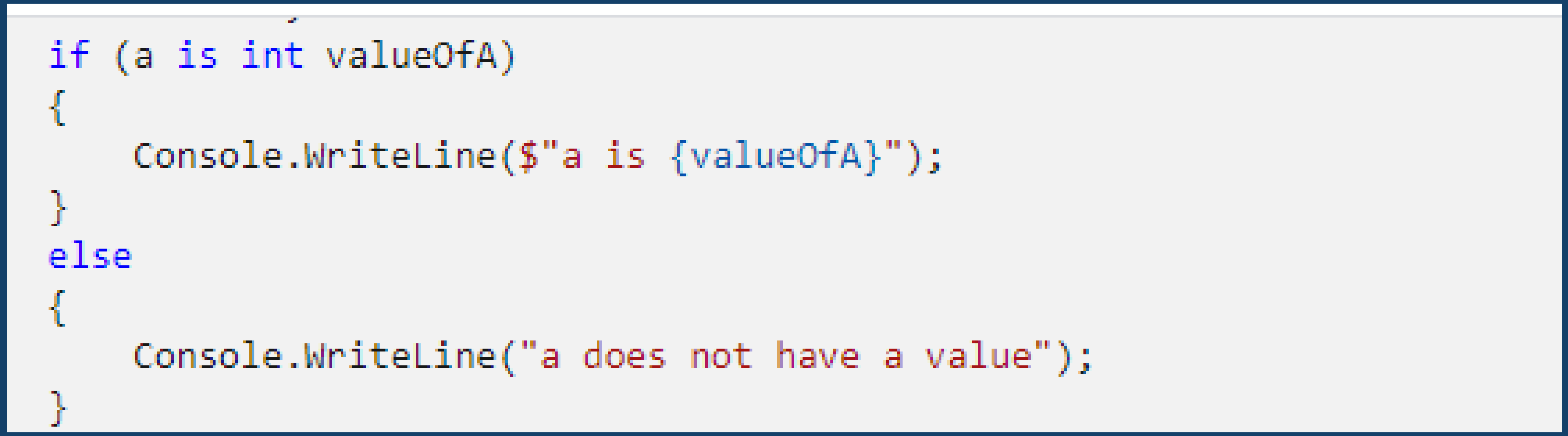


```
int? c = 7;
if (c != null)
{
    Console.WriteLine($"c is {c.Value}");
}
else
{
    Console.WriteLine("c does not have a value");
}
```



```
int? b = 10;
if (b.HasValue)
{
    Console.WriteLine($"b is {b.Value}");
}
else
{
    Console.WriteLine("b does not have a value");
}
```

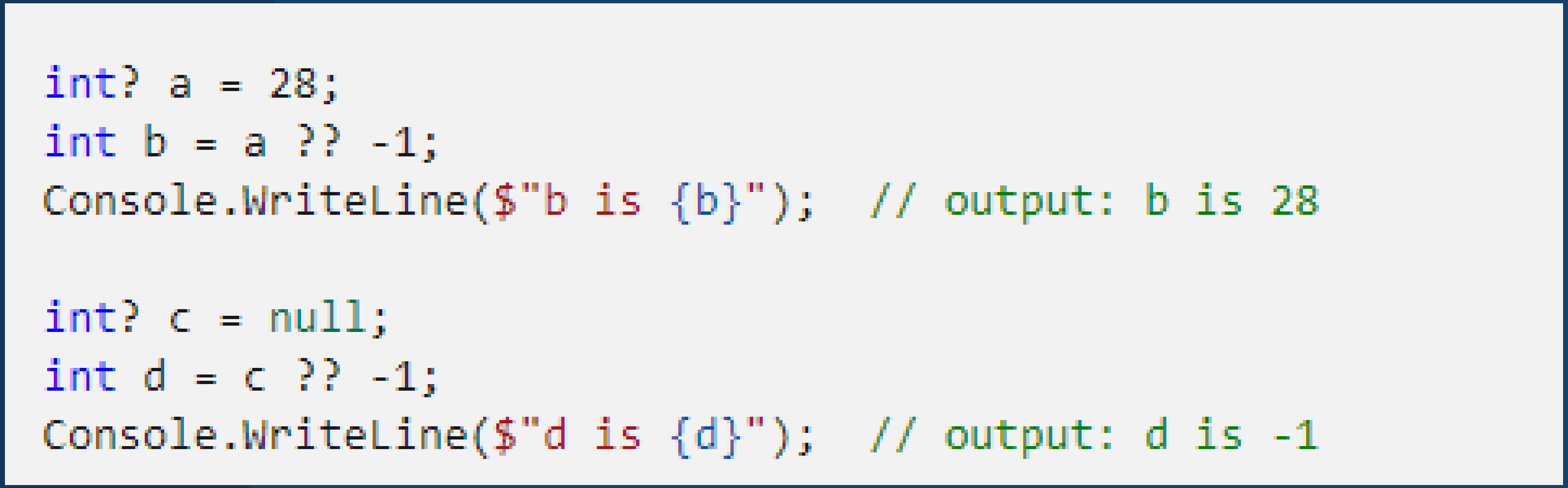
Beginning with C# 7.0, you can use the is operator with a type pattern to both examine an instance of a nullable value type for null and retrieve a value of an underlying type:



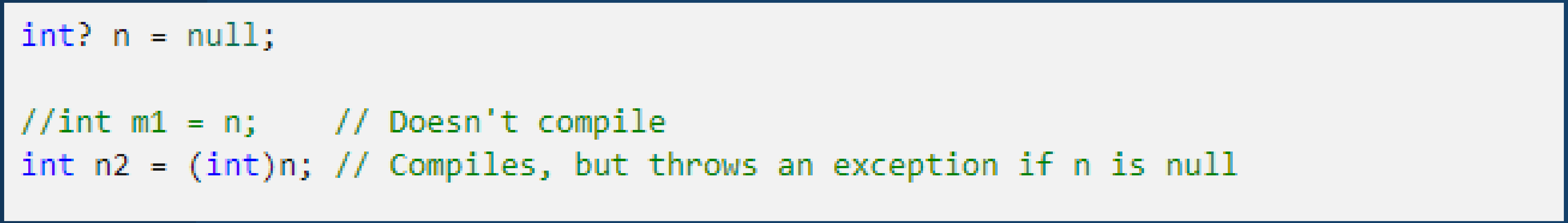
```
if (a is int valueOfA)
{
    Console.WriteLine($"a is {valueOfA}");
}
else
{
    Console.WriteLine("a does not have a value");
}
```

C# versions 2 – Nullable value types

Conversion from a nullable value type to an underlying type



```
int? a = 28;
int b = a ?? -1;
Console.WriteLine($"b is {b}");  // output: b is 28

int? c = null;
int d = c ?? -1;
Console.WriteLine($"d is {d}");  // output: d is -1
```

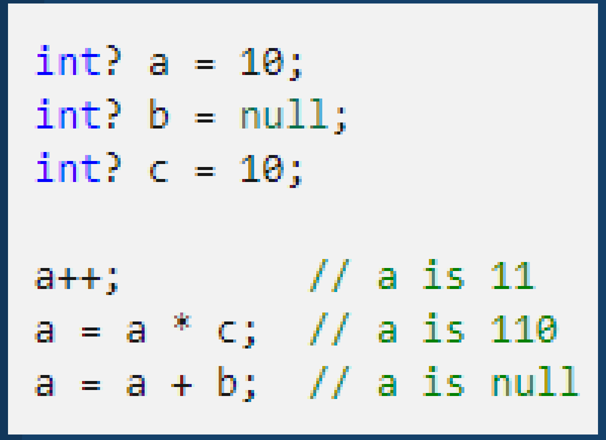


```
int? n = null;

//int m1 = n;    // Doesn't compile
int n2 = (int)n; // Compiles, but throws an exception if n is null
```

C# versions 2 – Nullable value types

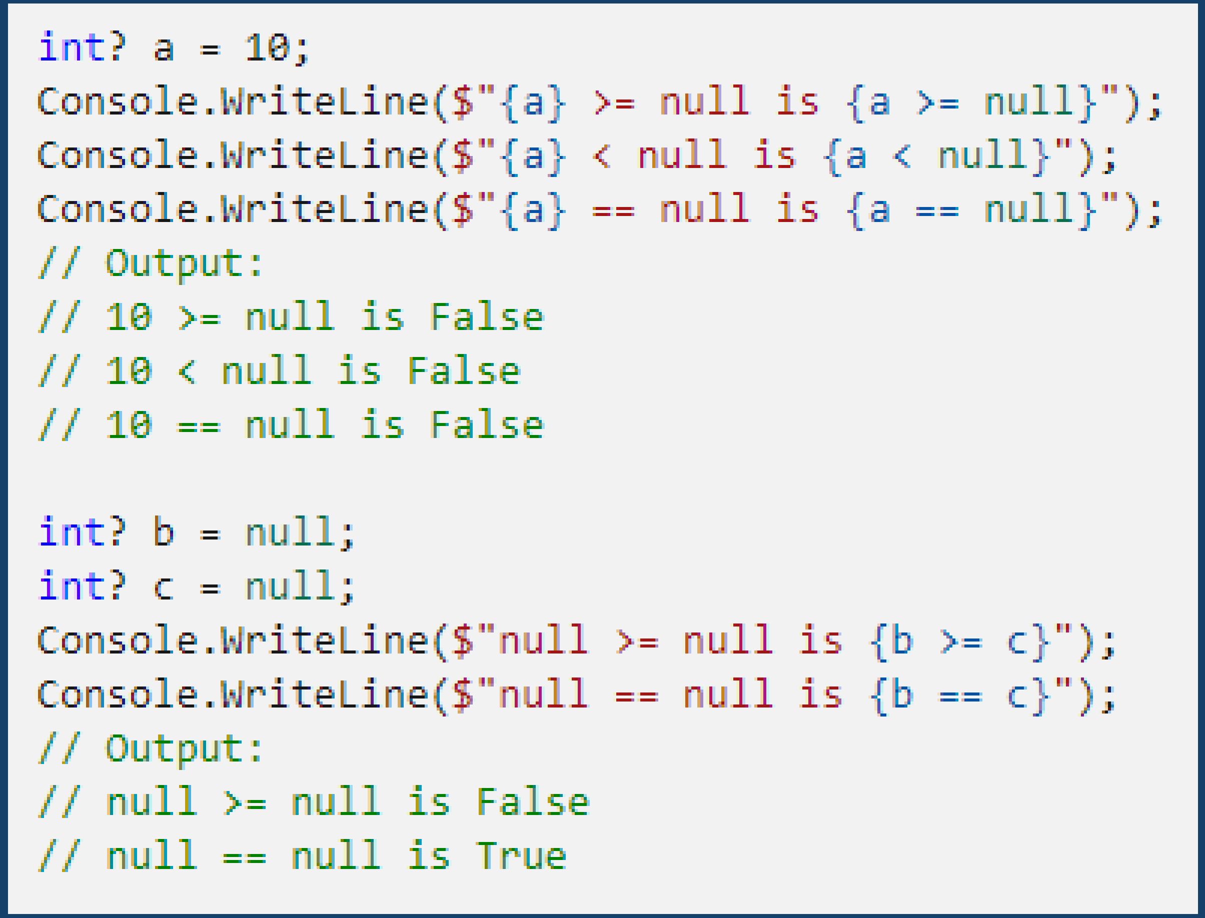
Lifted operators

- The predefined unary and binary operators or any overloaded operators that are supported by a value type T are also supported by the corresponding nullable value type T?.
- They produce null if one or both operands are null; otherwise, the operator uses the contained values of its operands to calculate the result (except Boolean & and | operators



```
int? a = 10;
int? b = null;
int? c = 10;

a++;         // a is 11
a = a * c;   // a is 110
a = a + b;   // a is null
```

- For the comparison operators <, >, <=, and >=, if one or both operands are null, the result is false;

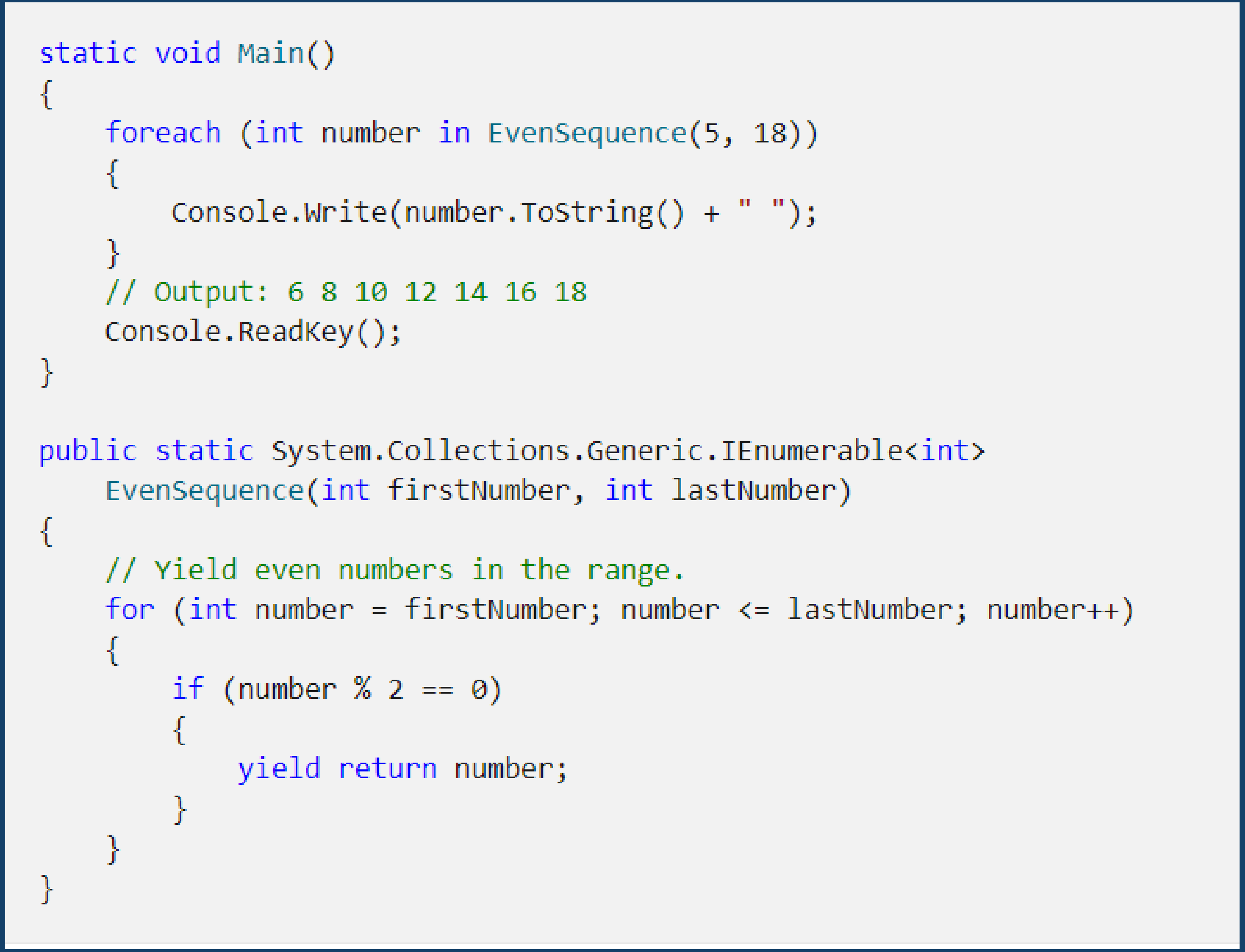


```
int? a = 10;
Console.WriteLine($"{a} >= null is {a >= null}");
Console.WriteLine($"{a} < null is {a < null}");
Console.WriteLine($"{a} == null is {a == null}");
// Output:
// 10 >= null is False
// 10 < null is False
// 10 == null is False

int? b = null;
int? c = null;
Console.WriteLine($"null >= null is {b >= c}");
Console.WriteLine($"null == null is {b == c}");
// Output:
// null >= null is False
// null == null is True
```

EIU

C# versions 2 – Iterators

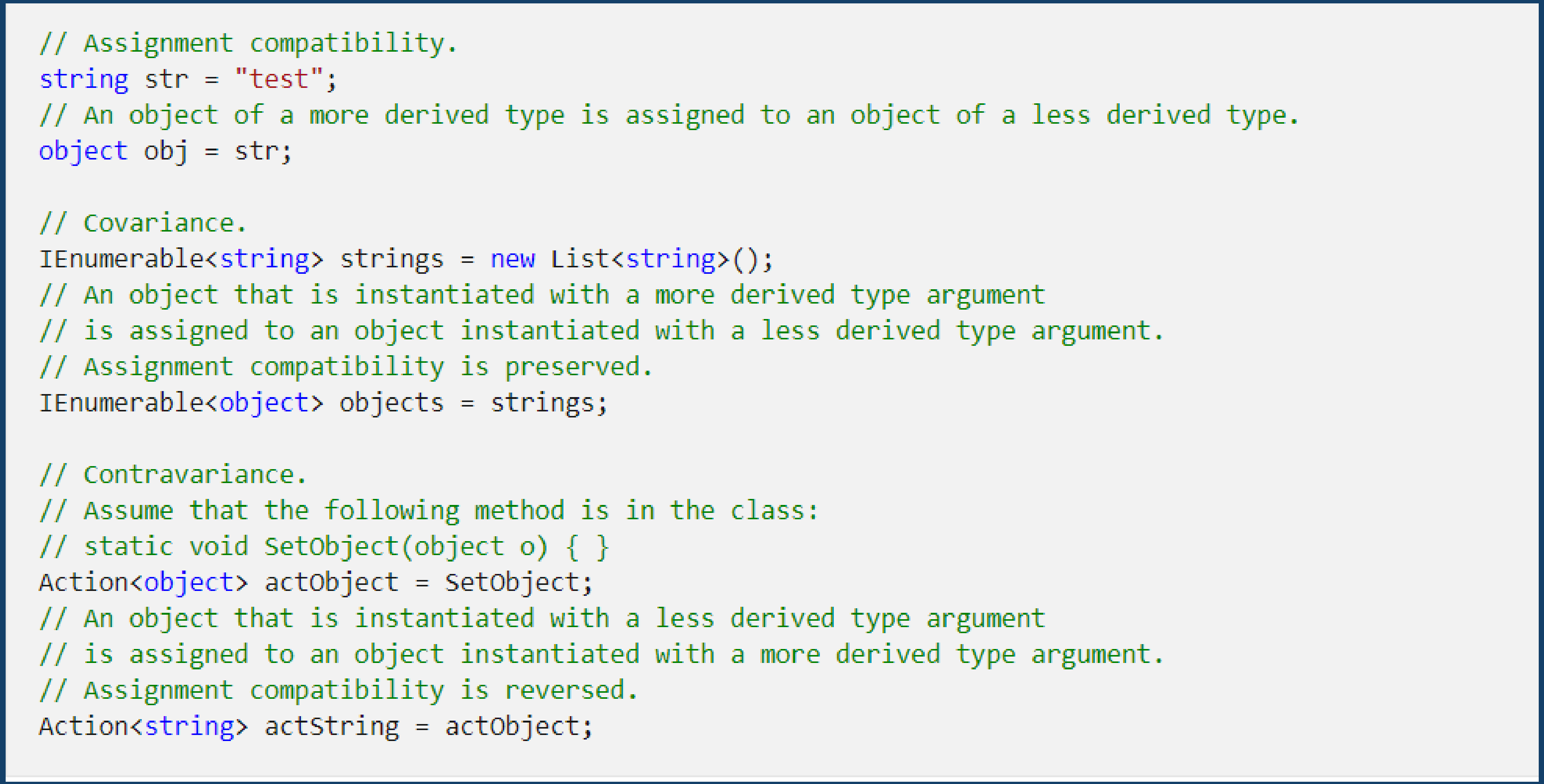
An iterator can be used to step through collections such as lists and arrays.



```
static void Main()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 6 8 10 12 14 16 18
    Console.ReadKey();
}

public static System.Collections.Generic.IEnumerable<int>
    EvenSequence(int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (int number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

C# versions 2 - Covariance and Contravariance

- In C#, covariance and contravariance enable implicit reference conversion for array types, delegate types, and generic type arguments. Covariance preserves assignment compatibility and contravariance reverses it.



```
// Assignment compatibility.
string str = "test";
// An object of a more derived type is assigned to an object of a less derived type.
object obj = str;

// Covariance.
IEnumerable<string> strings = new List<string>();
// An object that is instantiated with a more derived type argument
// is assigned to an object instantiated with a less derived type argument.
// Assignment compatibility is preserved.
IEnumerable<object> objects = strings;

// Contravariance.
// Assume that the following method is in the class:
// static void SetObject(object o) { }
Action<object> actObject = SetObject;
// An object that is instantiated with a less derived type argument
// is assigned to an object instantiated with a more derived type argument.
// Assignment compatibility is reversed.
Action<string> actString = actObject;
```

C# version 3.0

- Auto-implemented properties
- Anonymous types
- Query expressions
- Lambda expressions
- Expression trees
- Extension methods
- Implicitly typed local variables
- Partial methods
- Object and collection initializers

C# version 3.0 - Auto-Implemented Properties

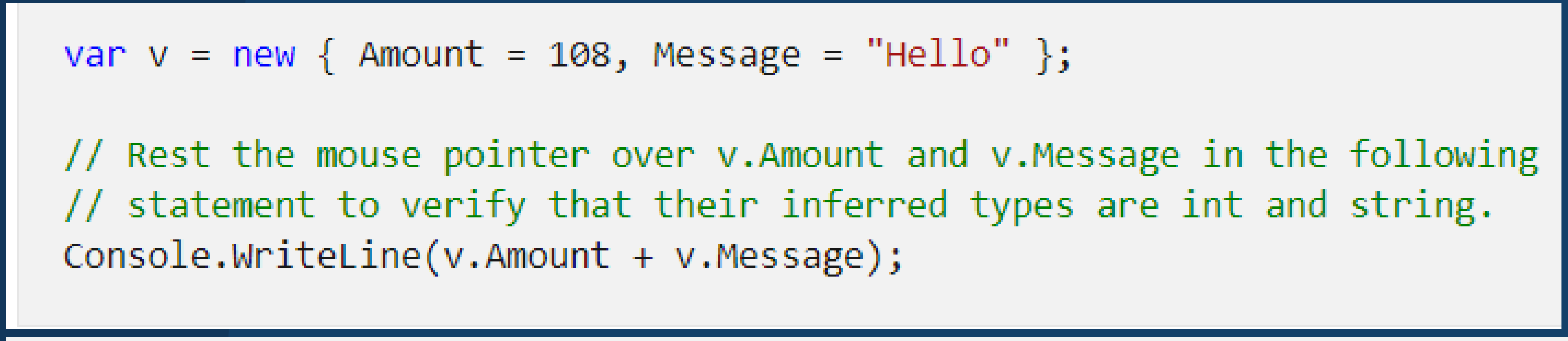
- Auto-implemented properties make property-declaration more concise when no additional logic is required in the property accessors.
- Can't declare auto-implemented properties in interfaces.
- C# 6 and later, you can initialize auto-implemented properties similarly to fields:

```
public string FirstName { get; set; } = "Jane";
```

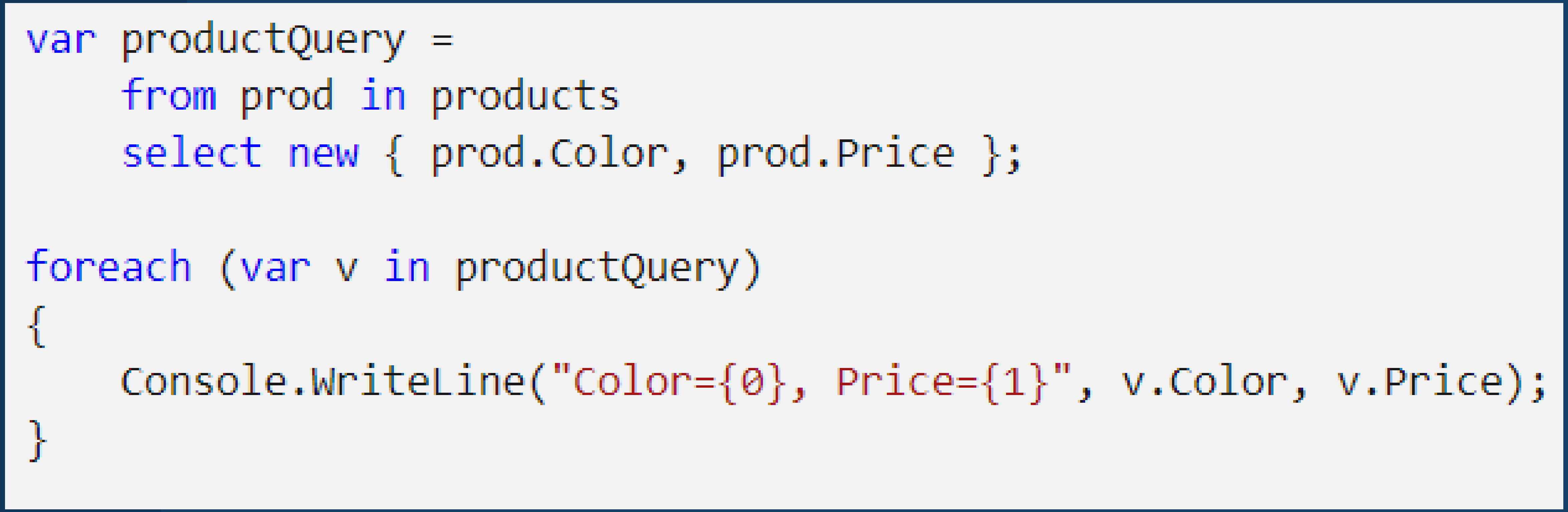
```
// This class is mutable. Its data can be modified from
// outside the class.
class Customer
{
    // Auto-implemented properties for trivial get and set
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerId { get; set; }

    // Constructor
    public Customer(double purchases, string name, int id)
    {
        TotalPurchases = purchases;
        Name = name;
        CustomerId = id;
    }

    // Methods
    public string GetContactInfo() { return "ContactInfo"; }
    public string GetTransactionHistory() { return "History"; }

    // .. Additional methods, events, etc.
}

class Program
{
    static void Main()
    {
        // Intialize a new object.
        Customer cust1 = new Customer(4987.63, "Northwind", 90108);

        // Modify a property.
        cust1.TotalPurchases += 499.99;
    }
}
```

C# version 3.0 - Anonymous Types

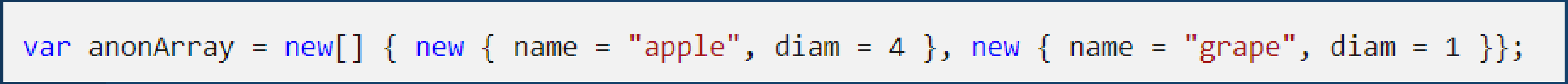
- Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first.



```csharp
var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following
// statement to verify that their inferred types are int and string.
Console.WriteLine(v.Amount + v.Message);
```



```csharp
var productQuery =
    from prod in products
    select new { prod.Color, prod.Price };

foreach (var v in productQuery)
{
    Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);
}
```



```csharp
var anonArray = new[] { new { name = "apple", diam = 4 }, new { name = "grape", diam = 1 }};
```

C# version 3.0 - Anonymous Types

- Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first.

```csharp
var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following
// statement to verify that their inferred types are int and string.
Console.WriteLine(v.Amount + v.Message);
```

```csharp
var productQuery =
    from prod in products
    select new { prod.Color, prod.Price };

foreach (var v in productQuery)
{
    Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);
}
```

```csharp
var anonArray = new[] { new { name = "apple", diam = 4 }, new { name = "grape", diam = 1 }};
```

C# version 3.0 - Query expression - LINQ

- Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language
- The most visible "language-integrated" part of LINQ is the query expression
- Query expressions can be used to query and to transform data from any LINQ-enabled data source
  - A query is not executed until you iterate over the query variable, for example, in a foreach statement.
  - Used to interact with numerous data types
  - Converted to expression trees at compile time and evaluated at runtime

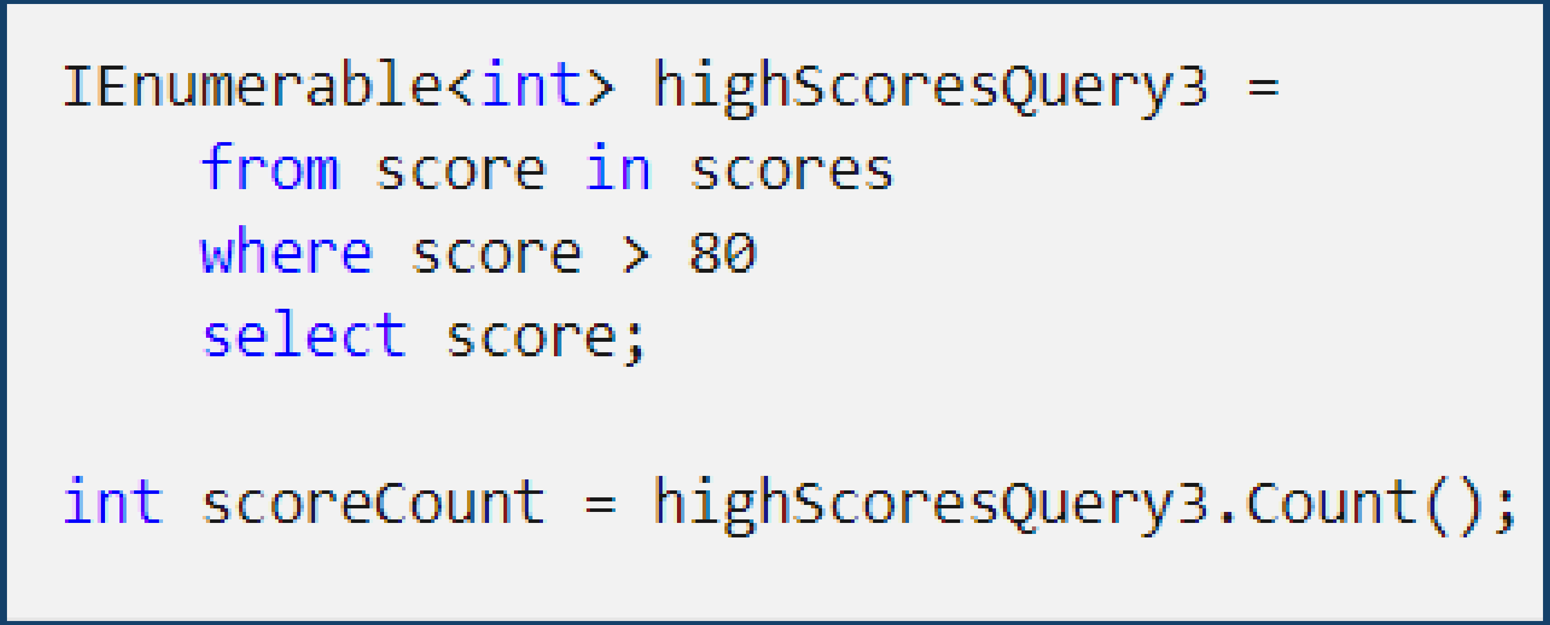
C# version 3.0 - Query expression

- A *query expression* is a query expressed in query syntax. A query expression is a first-class language construct.
  - Must begin with a from clause
  - Must end with a select or group clause.
  - It can contain one or more of these optional clauses: where, orderby, join, let and even additional from clauses.
  - You can also use the into keyword to enable the result of a join or group clause to serve as the source for additional query clauses in the same query expression

In LINQ, a query variable is any variable that stores a query instead of the results of a query.

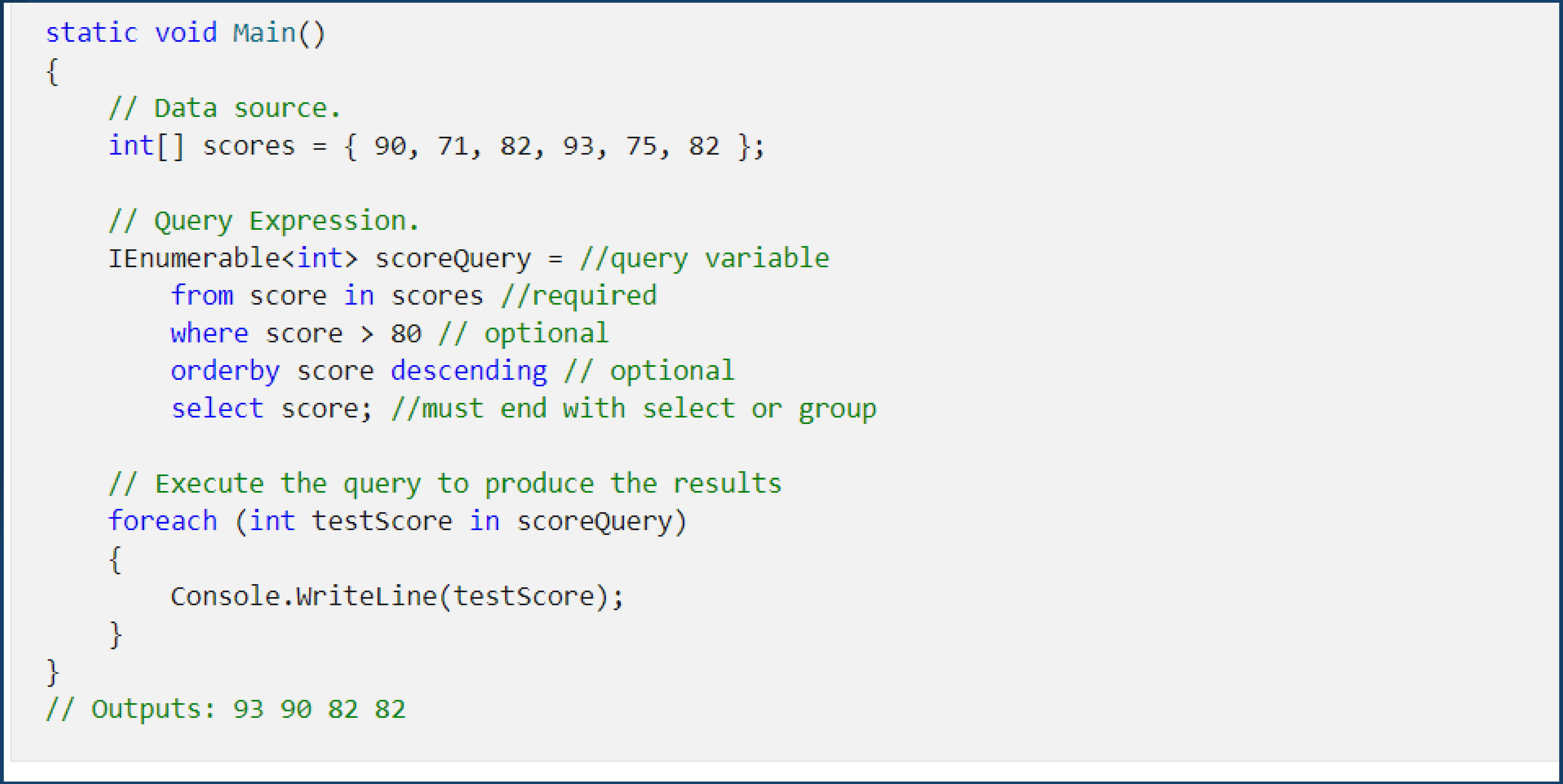
C# version 3.0 - Query expression

- A query is a set of instructions that describes what data to retrieve from a given data source (or sources) and what shape and organization the returned data should have. A query is distinct from the results that it produces.



```
IEnumerable<int> highScoresQuery3 =
    from score in scores
    where score > 80
    select score;

int scoreCount = highScoresQuery3.Count();
```

C# version 3.0 - Query expression



```
static void Main()
{
    // Data source.
    int[] scores = { 90, 71, 82, 93, 75, 82 };

    // Query Expression.
    IEnumerable<int> scoreQuery = //query variable
        from score in scores //required
        where score > 80 // optional
        orderby score descending // optional
        select score; //must end with select or group

    // Execute the query to produce the results
    foreach (int testScore in scoreQuery)
    {
        Console.WriteLine(testScore);
    }
}
// Outputs: 93 90 82 82
```

C# version 3.0 - Lambda expressions

You use a *lambda expression* to create an anonymous function. Use the lambda declaration operator => to separate the lambda's parameter list from its body.
A lambda expression can be of any of the following two forms:

- Expression lambda that has an expression as its body:

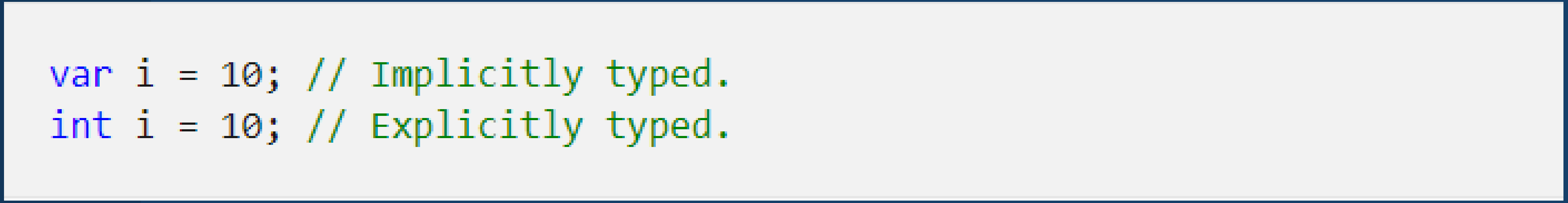
```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

- Statement lambda that has a statement block as its body:

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

C# version 3.0 - Implicitly typed local variable

- Implicitly typed local variable is strongly typed just as if you had declared the type yourself, but the compiler determines the type.
- An implicitly typed local variable is strongly typed just as if you had declared the type yourself, but the compiler determines the type



```
var i = 10; // Implicitly typed.
int i = 10; // Explicitly typed.
```

C# version 3.0 - partial method

- A partial method has its signature defined in one part of a partial type, and its implementation defined in another part of the type.

```
namespace PM
{
    partial class A
    {
        partial void OnSomethingHappened(string s);
    }

    // This part can be in a separate file.
    partial class A
    {
        // Comment out this method and the program
        // will still compile.
        partial void OnSomethingHappened(String s)
        {
            Console.WriteLine("Something happened: {0}", s);
        }
    }
}
```

C# version 3.0 - Object and Collection Initializers

- C# lets you instantiate an object or collection and perform member assignments in a single statement.
- Object initializers let you assign values to any accessible fields or properties of an object at creation time without having to invoke a constructor followed by lines of assignment statements.

```
public class Cat
{
    // Auto-implemented properties.
    public int Age { get; set; }
    public string Name { get; set; }

    public Cat()
    {
    }

    public Cat(string name)
    {
        this.Name = name;
    }
}
```

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
Cat sameCat = new Cat("Fluffy"){ Age = 10 };
```

C# version 3.0 - Object and Collection Initializers

- Collection initializers let you specify one or more element initializers when you initialize a collection type that implements IEnumerable and has Add with the appropriate signature as an instance method or an extension method.

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };
```

```
List<Cat> cats = new List<Cat>
{
    new Cat{ Name = "Sylvester", Age=8 },
    new Cat{ Name = "Whiskers", Age=2 },
    new Cat{ Name = "Sasha", Age=14 }
};
```

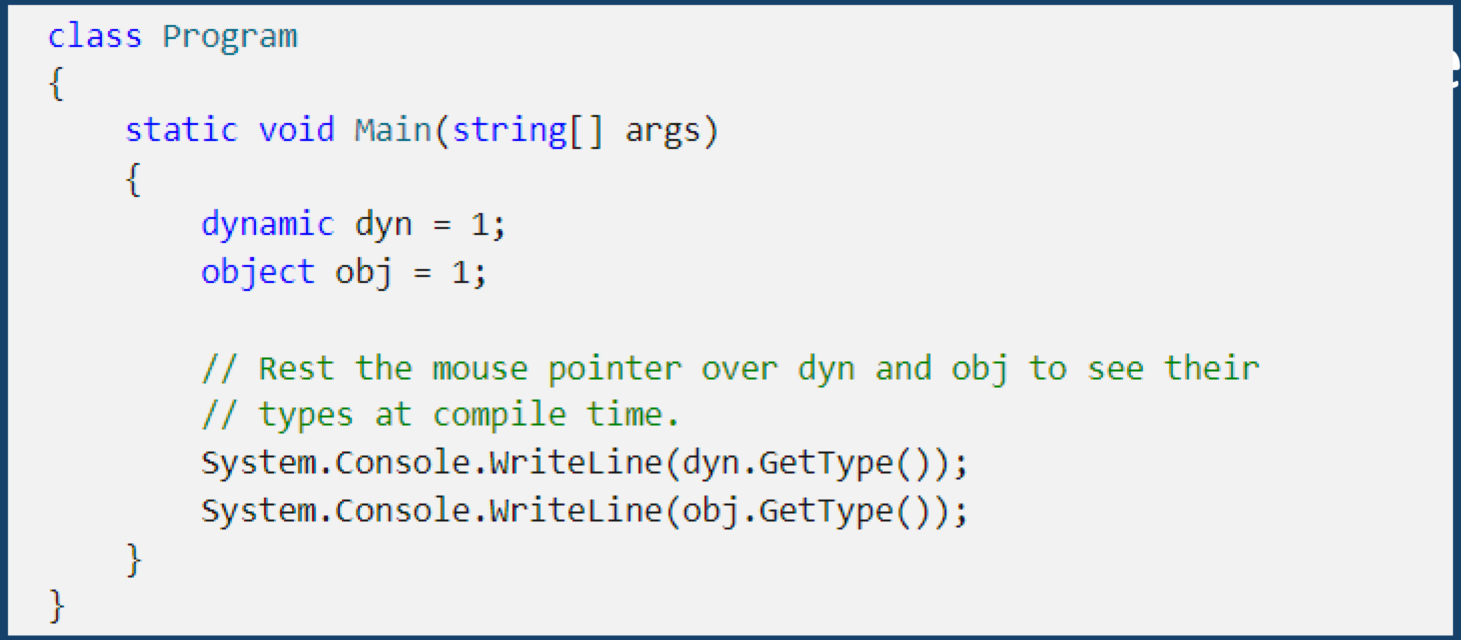
C# version 4.0

- Dynamic binding
- Named/optional arguments
- Generic covariant and contravariant
- Embedded interop types

C# version 4.0 – Dynamic binding

- The dynamic type indicates that use of the variable and references to its members bypass

se

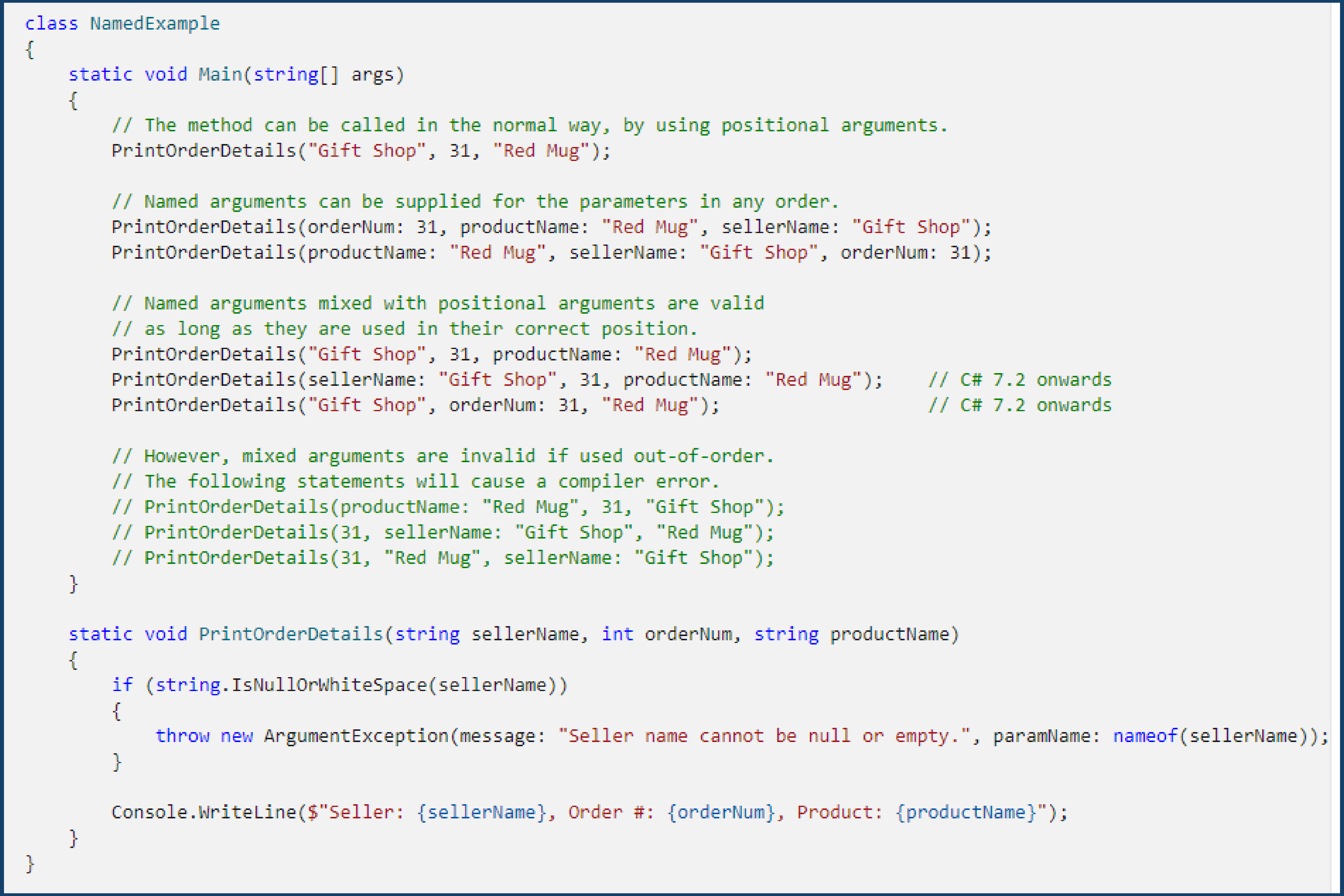


```
class Program
{
    static void Main(string[] args)
    {
        dynamic dyn = 1;
        object obj = 1;

        // Rest the mouse pointer over dyn and obj to see their
        // types at compile time.
        System.Console.WriteLine(dyn.GetType());
        System.Console.WriteLine(obj.GetType());
    }
}
```

C# version 4.0 – Named and Optional Arguments

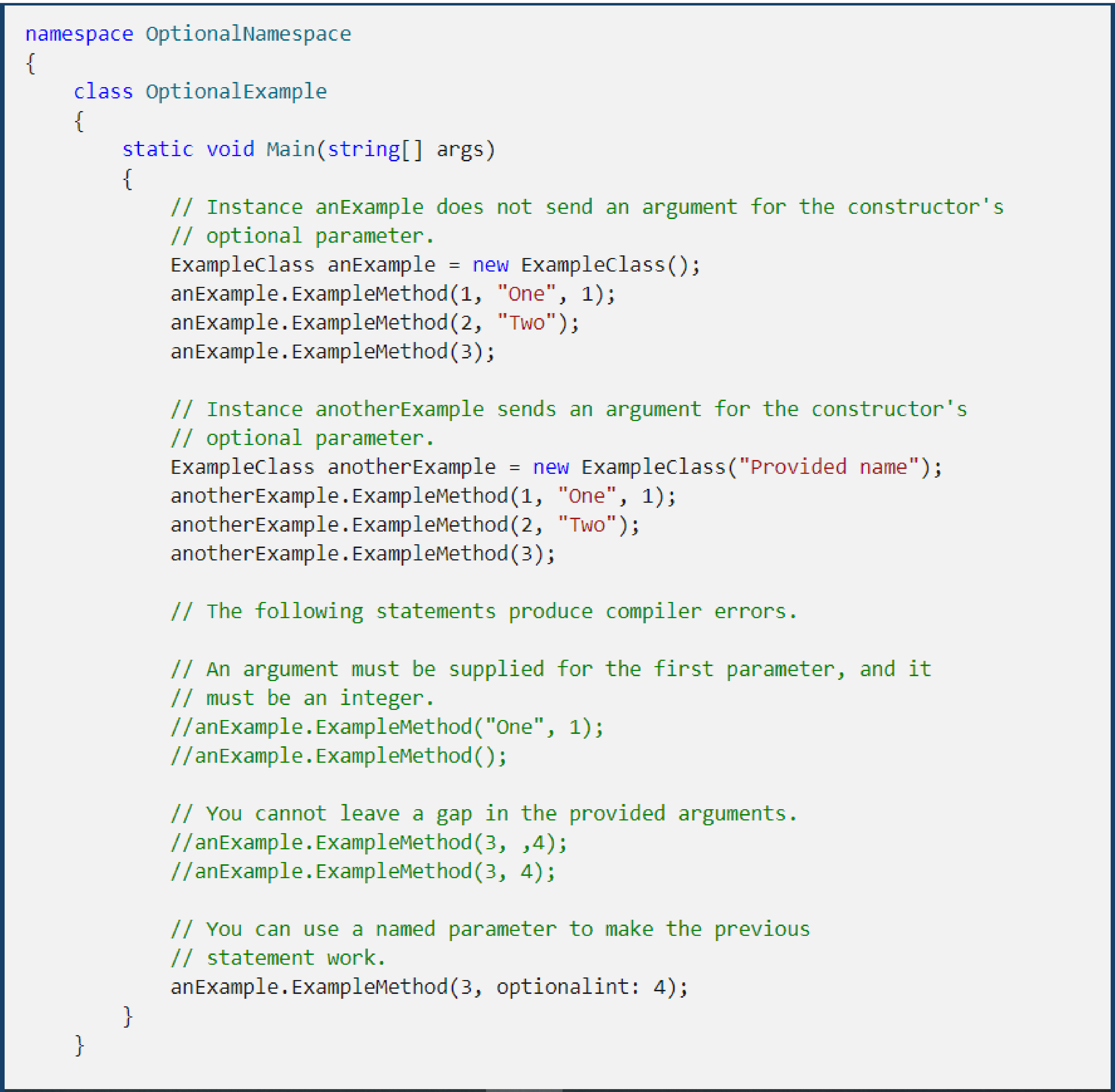
- *Named arguments* enable you to specify an argument for a parameter by matching the argument with its name rather than with its position in the parameter list.
- *Optional arguments* enable you to omit arguments for some parameters. Both techniques can be used with methods, indexers, constructors, and delegates.

Named arguments



```
class NamedExample
{
    static void Main(string[] args)
    {
        // The method can be called in the normal way, by using positional arguments.
        PrintOrderDetails("Gift Shop", 31, "Red Mug");

        // Named arguments can be supplied for the parameters in any order.
        PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
        PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);

        // Named arguments mixed with positional arguments are valid
        // as long as they are used in their correct position.
        PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
        PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");    // C# 7.2 onwards
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");                   // C# 7.2 onwards

        // However, mixed arguments are invalid if used out-of-order.
        // The following statements will cause a compiler error.
        // PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
        // PrintOrderDetails(31, sellerName: "Gift Shop", "Red Mug");
        // PrintOrderDetails(31, "Red Mug", sellerName: "Gift Shop");
    }

    static void PrintOrderDetails(string sellerName, int orderNum, string productName)
    {
        if (string.IsNullOrWhiteSpace(sellerName))
        {
            throw new ArgumentException(message: "Seller name cannot be null or empty.", paramName: nameof(sellerName));
        }

        Console.WriteLine($"Seller: {sellerName}, Order #: {orderNum}, Product: {productName}");
    }
}
```

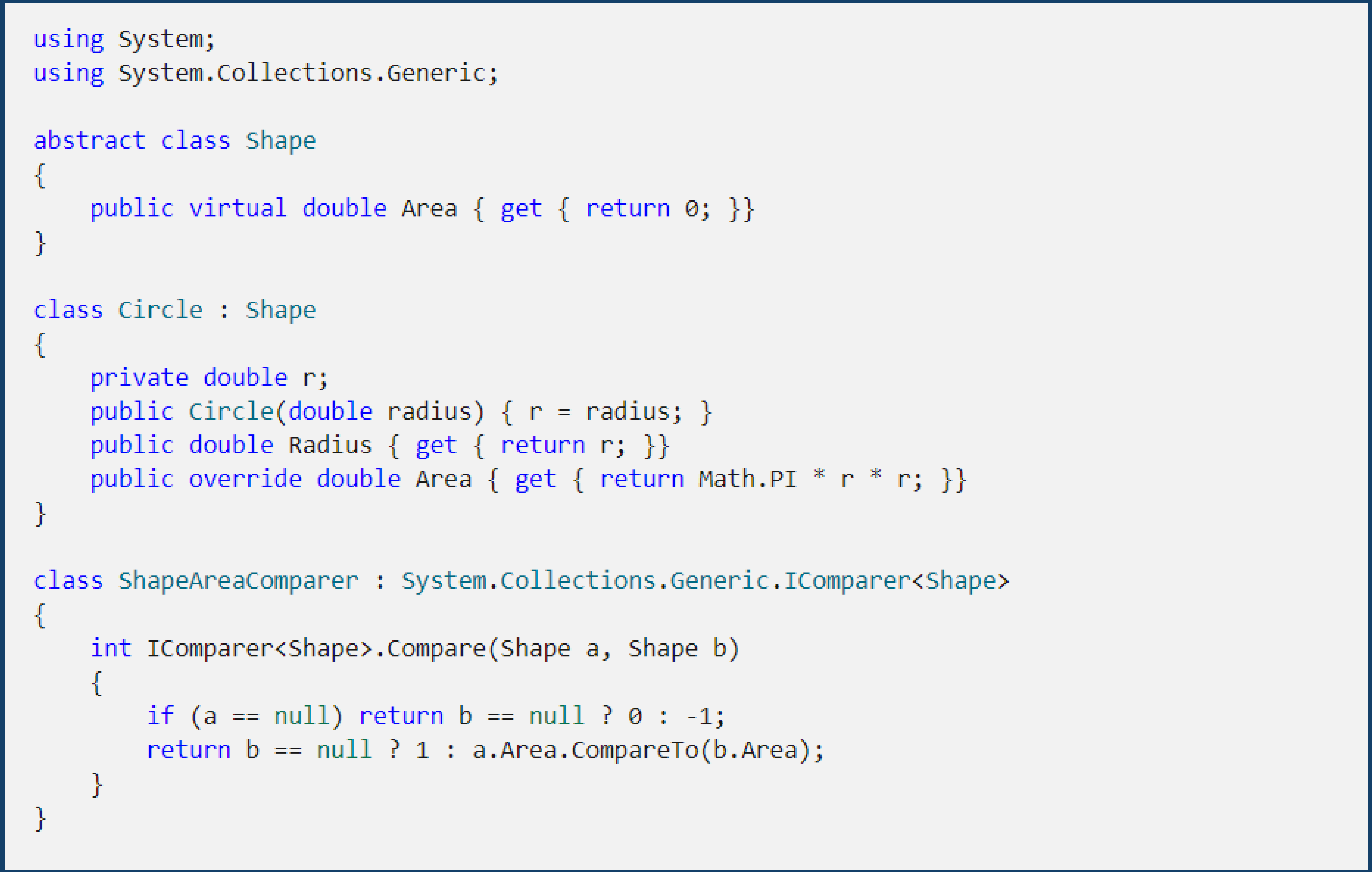
Optional arguments



```
namespace OptionalNamespace
{
    class OptionalExample
    {
        static void Main(string[] args)
        {
            // Instance anExample does not send an argument for the constructor's
            // optional parameter.
            ExampleClass anExample = new ExampleClass();
            anExample.ExampleMethod(1, "One", 1);
            anExample.ExampleMethod(2, "Two");
            anExample.ExampleMethod(3);

            // Instance anotherExample sends an argument for the constructor's
            // optional parameter.
            ExampleClass anotherExample = new ExampleClass("Provided name");
            anotherExample.ExampleMethod(1, "One", 1);
            anotherExample.ExampleMethod(2, "Two");
            anotherExample.ExampleMethod(3);

            // The following statements produce compiler errors.

            // An argument must be supplied for the first parameter, and it
            // must be an integer.
            //anExample.ExampleMethod("One", 1);
            //anExample.ExampleMethod();

            // You cannot leave a gap in the provided arguments.
            //anExample.ExampleMethod(3, ,4);
            //anExample.ExampleMethod(3, 4);

            // You can use a named parameter to make the previous
            // statement work.
            anExample.ExampleMethod(3, optionalint: 4);
        }
    }
```

C# version 4.0 – Covariance and contravariance in generics

- *Covariance* and *contravariance* are terms that refer to the ability to use a more derived type (more specific) or a less derived type (less specific) than originally specified.



```csharp
using System;
using System.Collections.Generic;

abstract class Shape
{
    public virtual double Area { get { return 0; }}
}

class Circle : Shape
{
    private double r;
    public Circle(double radius) { r = radius; }
    public double Radius { get { return r; }}
    public override double Area { get { return Math.PI * r * r; }}
}

class ShapeAreaComparer : System.Collections.Generic.IComparer<Shape>
{
    int IComparer<Shape>.Compare(Shape a, Shape b)
    {
        if (a == null) return b == null ? 0 : -1;
        return b == null ? 1 : a.Area.CompareTo(b.Area);
    }
}
```

C# version 4.0 – Covariance and contravariance in generics

- *Covariance* enables you to use a more derived type than originally specified.

```csharp
IEnumerable<Derived> d = new List<Derived>();
IEnumerable<Base> b = d;
```

- *Contravariance* enables you to use a more generic (less derived) type than originally specified.

```csharp
Action<Base> b = (target) => { Console.WriteLine(target.GetType().Name); };
Action<Derived> d = b;
d(new Derived());
```

In general, a covariant type parameter can be used as the return type of a delegate, and contravariant type parameters can be used as parameter types. For an interface, covariant type parameters can be used as the return types of the interface's methods, and contravariant type parameters can be used as the parameter types of the interface's methods.

C# version 4.0 – Type equivalence and embedded interop types

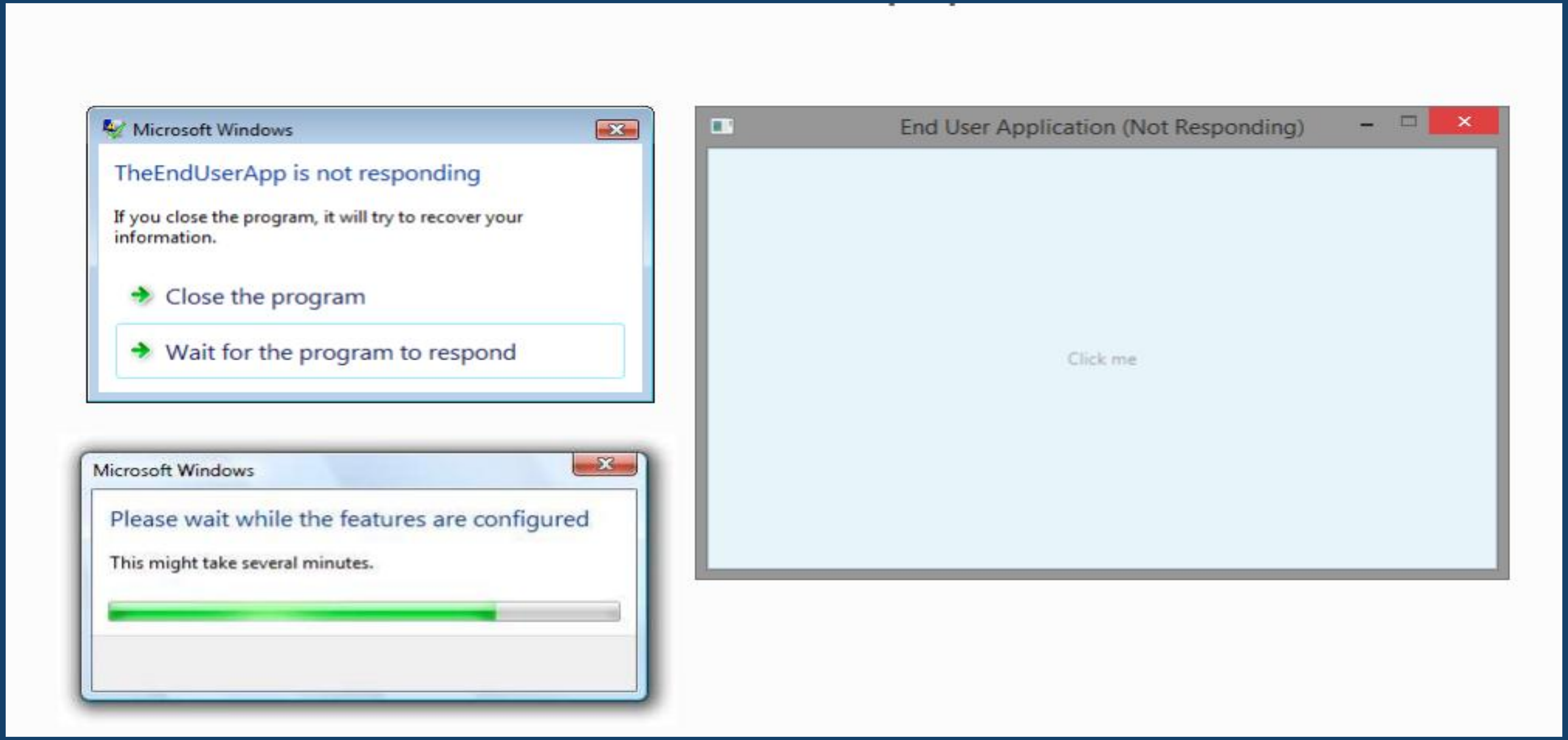
- Embedded interop types eased the deployment pain of creating COM interop assemblies for your application. Generic covariance and contravariance give you more power to use generics

C# version 5.0

- Asynchronous members
- Caller info attributes

Asynchronous Programming – Why?

- Avoid unreliable applications
- Give users a better experience
- Utilize resources



Microsoft Windows

TheEndUserApp is not responding

If you close the program, it will try to recover your information.

Close the program

Wait for the program to respond

Microsoft Windows

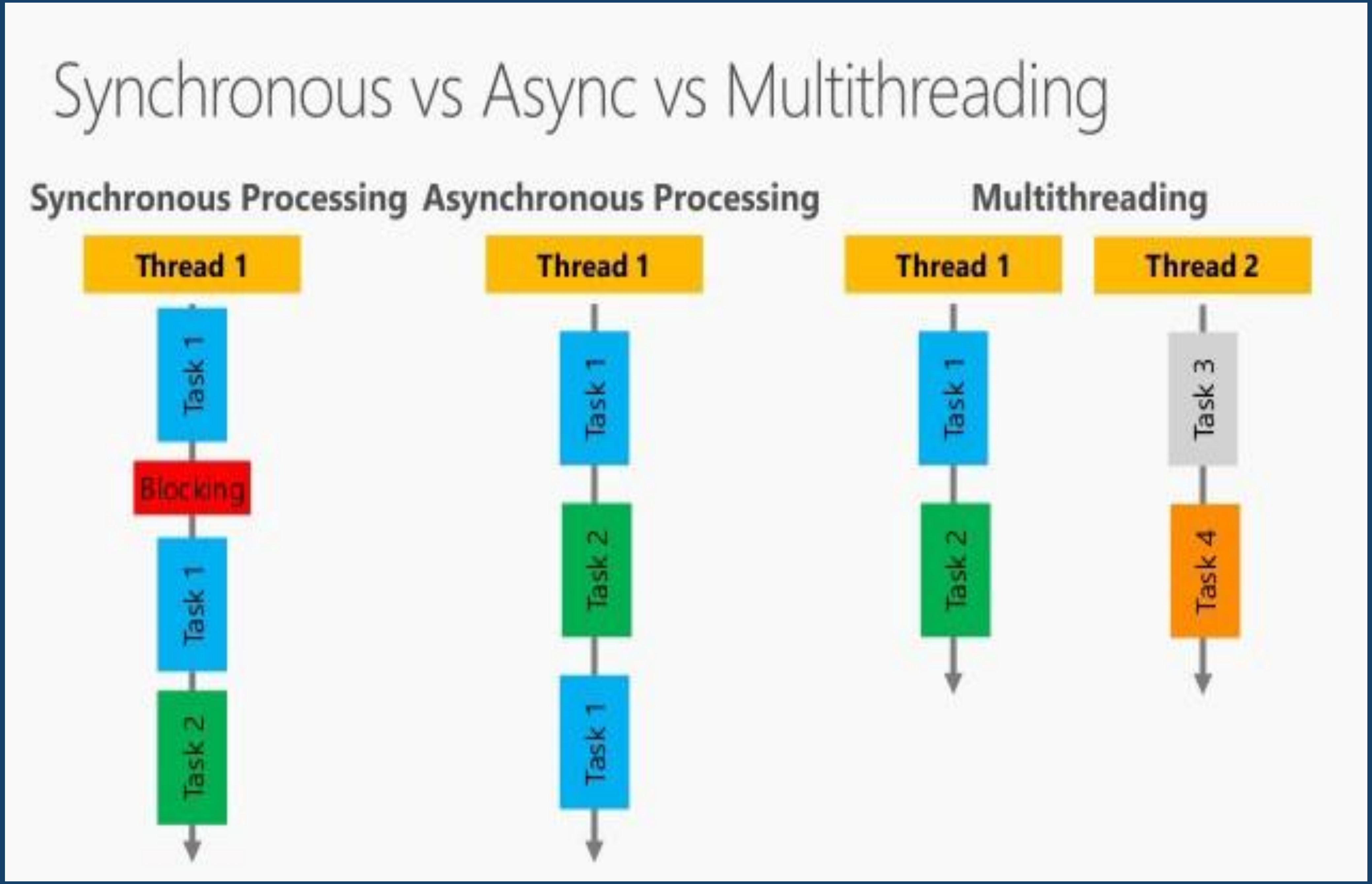
Please wait while the features are configured

This might take several minutes.

End User Application (Not Responding)

Asynchronous Programming

- Processing allowed before the current execution is done, such as read/write to disk.
- Asynchronous Javascript (AJAX)

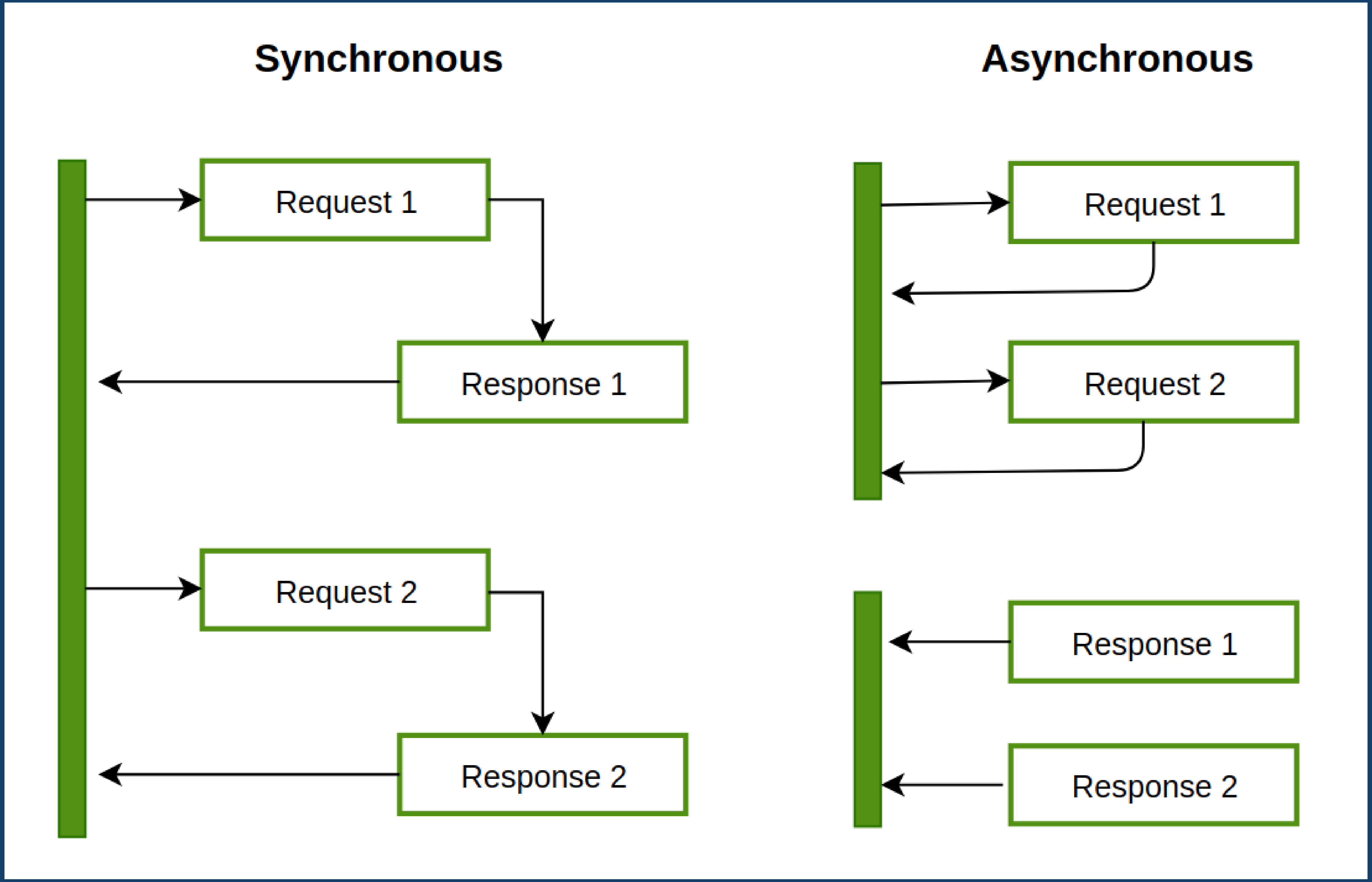


Asynchronous vs Parallel

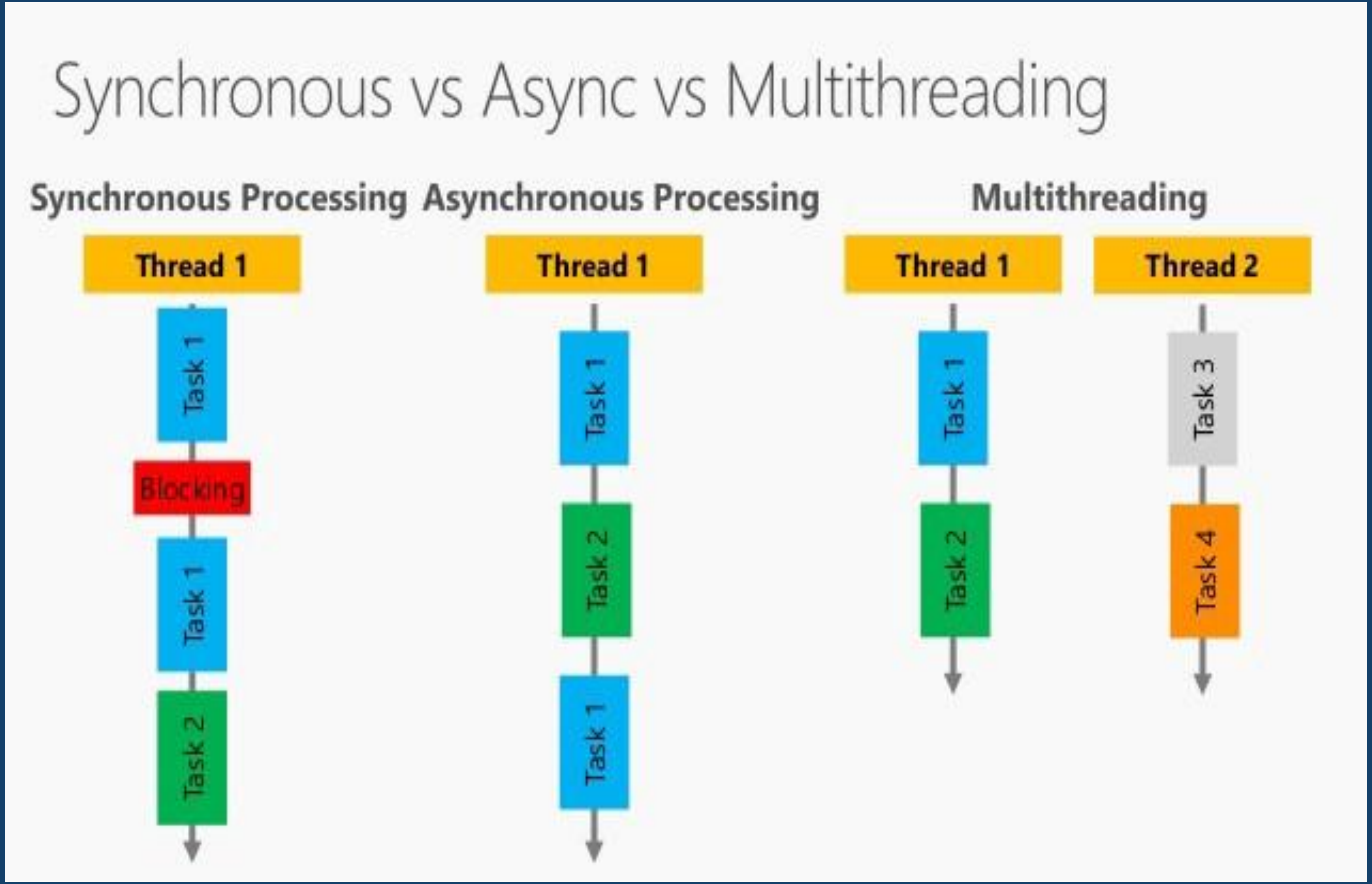
- Run something asynchronously it means it is non-blocking
- Execute it without waiting for it to complete and carry on with other things

- Parallelism means to run multiple things at the same time, in parallel.
- Parallelism works well when tasks are separated

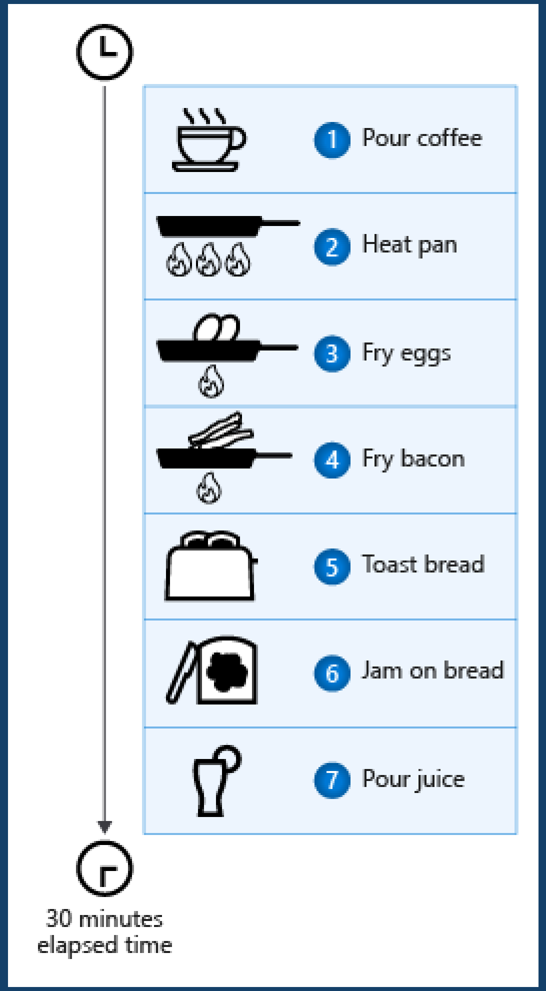
Synchronous vs Asynchronous



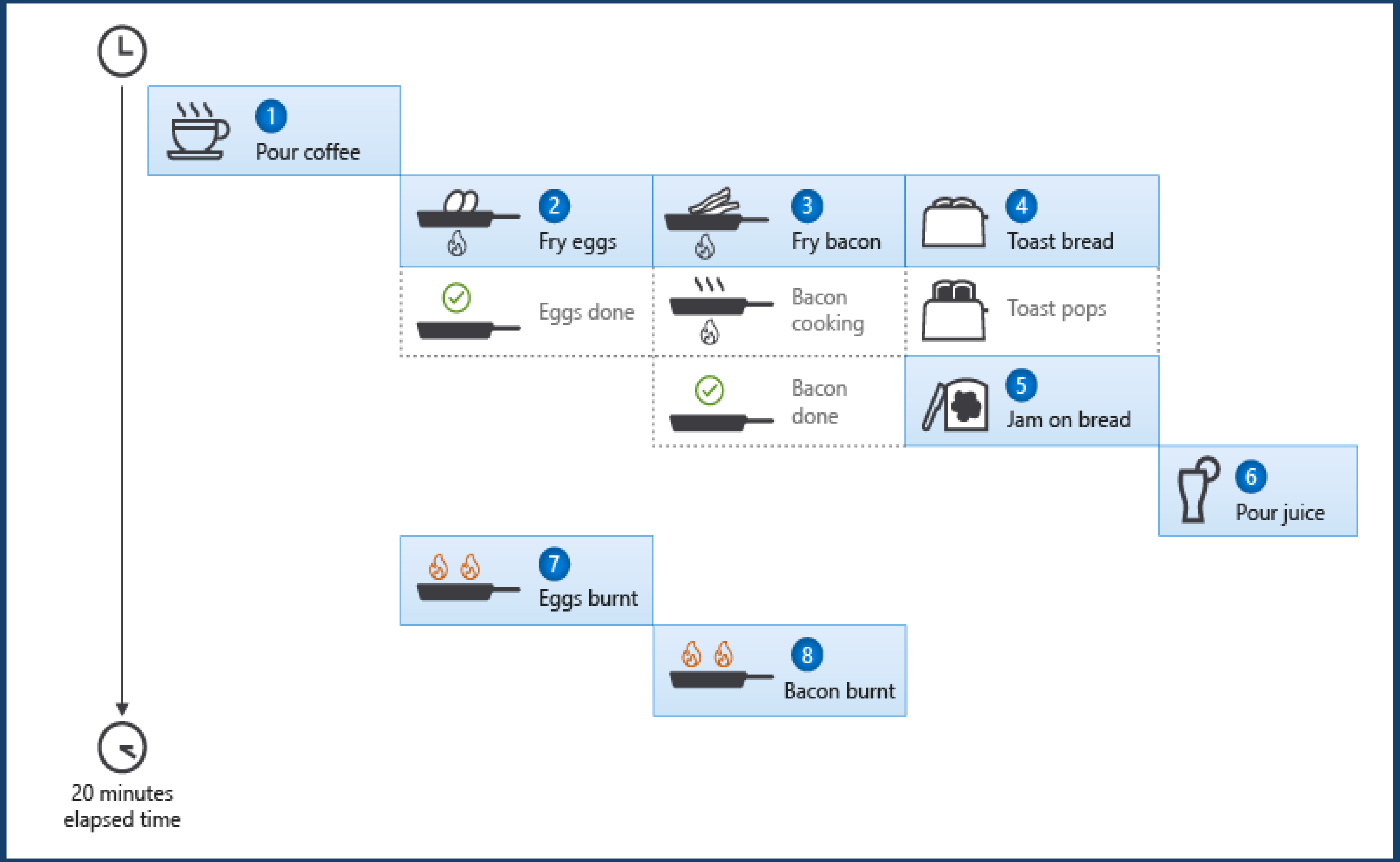
Synchronous vs Asynchronous vs Multithreading



C# version 5.0 - Asynchronous programming with Async and Await



Synchronous



Asynchronous

Asynchronous Programming in .NET

- Traditionally two different patterns have been used for asynchronous operations
  - Asynchronous Programming Model
    - BeginRead, EndRead… methods
  - Even-based asynchronous pattern
    - WebClient.DownloadStringCompleted

Task Parallel Library (TPL)

- Simplify the work of writing concurrent and asynchronous code
- TPL was introduced in .NET 4.0

C# version 5.0 - Asynchronous programming

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. The model is fairly simple in most cases

C# version 5.0 - Asynchronous programming

- **Making a method as async**
  - Indicate that your method body will be executed within a state machine
  - Adds overhead and hidden complexity
  - Not using await? => Don’t mark as async!
  - Code can be still be synchronous in a method marked as async.
  - Async and Await need to be used together.

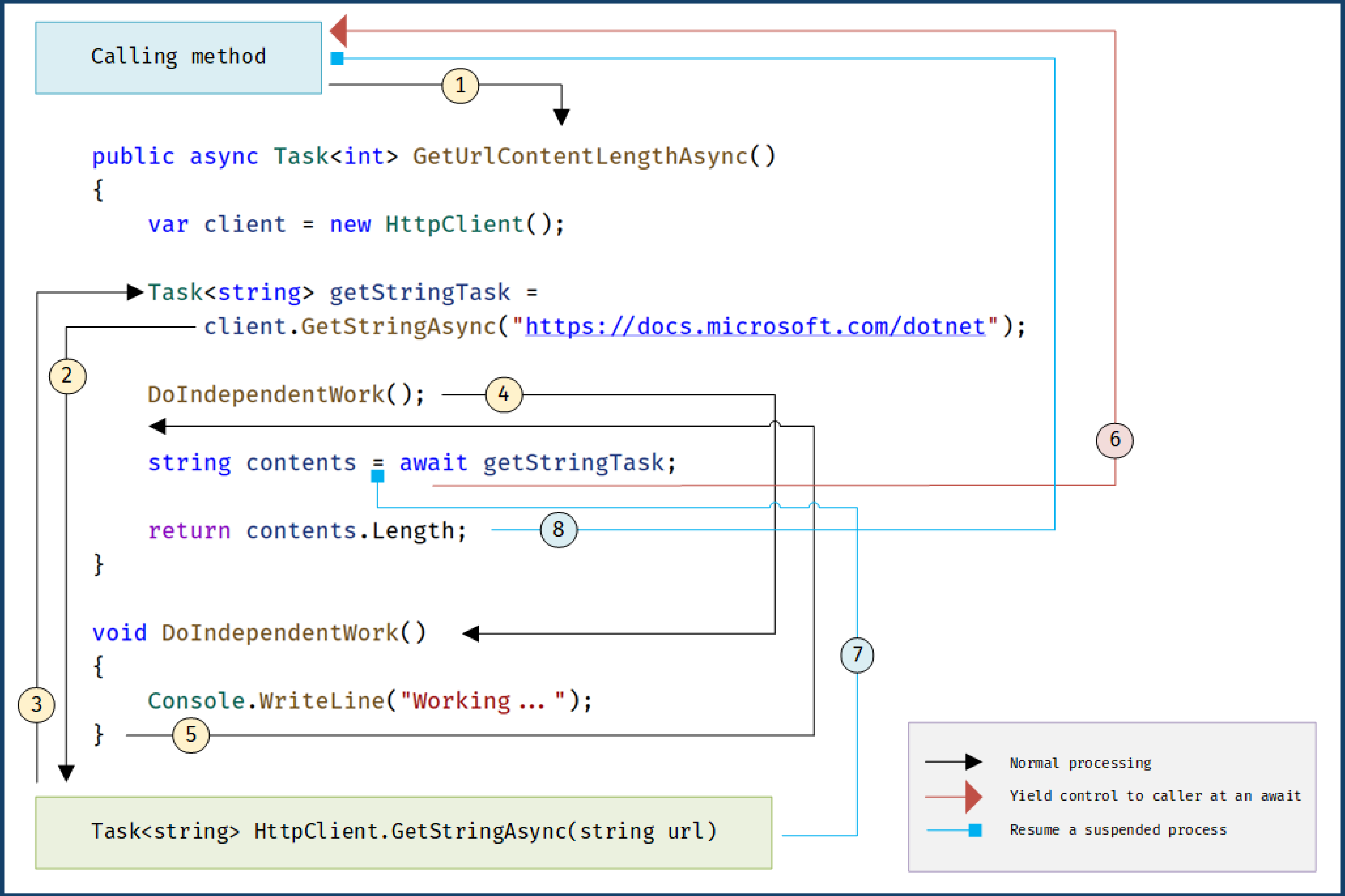
C# version 5.0 - Asynchronous programming

- **Async methods can have the following return types:**
  - Task, for an async method that performs an operation but returns no value.
  - Task<TResult>, for an async method that returns a value.
  - void, for an event handler (only)
    - can't be awaited, can't catch exceptions
  - Starting with C# 7.0, any type that has an accessible GetAwaiter method. The object returned by the GetAwaiter method must implement the System.Runtime.CompilerServices.ICriticalNotifyCompletion interface.
  - Starting with C# 8.0, IAsyncEnumerable<T>, for an async method that returns an async stream.

C# version 5.0 - Asynchronous programming

- **Where do you use await?:**
  - Everything after the statement is awaited will be in the continuation
  - Await returns the executing task to the caller
  - The continuation will be processed on the caller(main/UI) thread
  - Await will give you the result of the asynchronous Task which is processed.

C# version 5.0 - Asynchronous programming

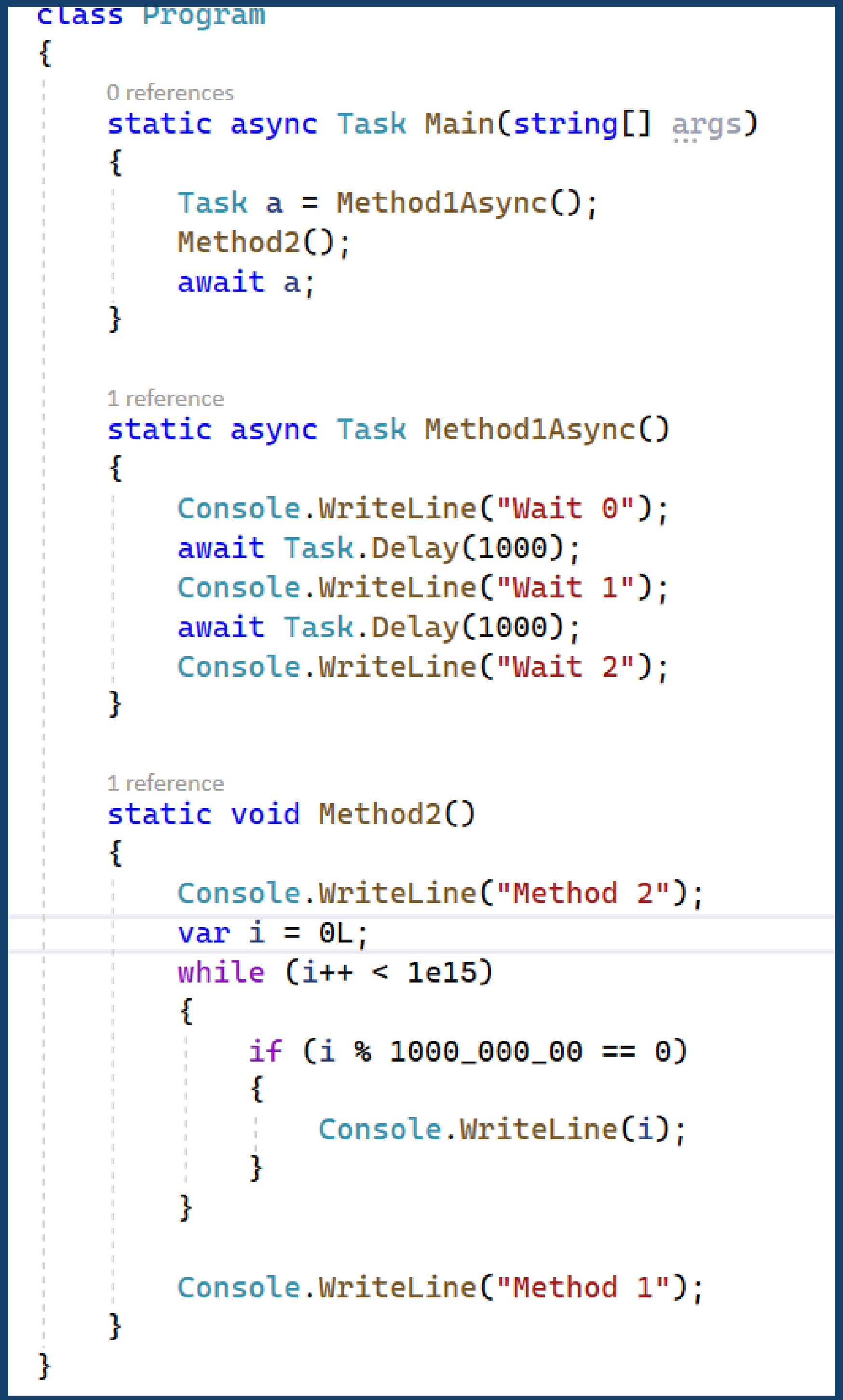


```
Calling method

public async Task<int> GetUrlContentLengthAsync()
{
    var client = new HttpClient();

    Task<string> getStringTask =
        client.GetStringAsync("https://docs.microsoft.com/dotnet");

    DoIndependentWork();

    string contents = await getStringTask;

    return contents.Length;
}

void DoIndependentWork()
{
    Console.WriteLine("Working . . . ");
}

Task<string> HttpClient.GetStringAsync(string url)
```

1
2
3
4
5
6
7
8
Normal processing
Yield control to caller at an await
Resume a suspended process

Quick test

- What is the output?



```
class Program
{
    0 references
    static async Task Main(string[] args)
    {
        Task a = Method1Async();
        Method2();
        await a;
    }

    1 reference
    static async Task Method1Async()
    {
        Console.WriteLine("Wait 0");
        await Task.Delay(1000);
        Console.WriteLine("Wait 1");
        await Task.Delay(1000);
        Console.WriteLine("Wait 2");
    }

    1 reference
    static void Method2()
    {
        Console.WriteLine("Method 2");
        var i = 0L;
        while (i++ < 1e15)
        {
            if (i % 1000_000_00 == 0)
            {
                Console.WriteLine(i);
            }
        }

        Console.WriteLine("Method 1");
    }
}
```

C# version 5.0 - Asynchronous programming

- 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.

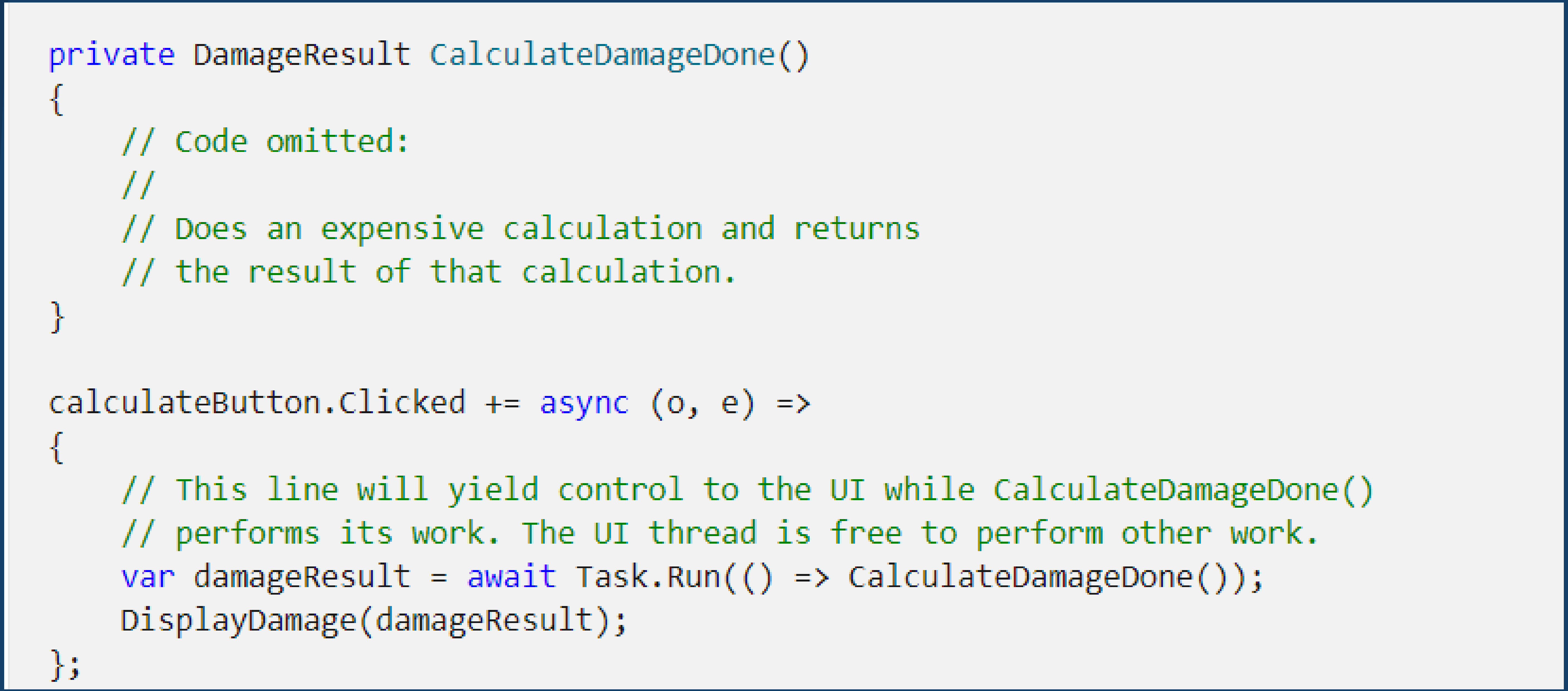
C# version 5.0 - Asynchronous programming

- **I/O-bound example: Download data from a web service**

```
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);
};
```

C# version 5.0 - Asynchronous programming

- **CPU-bound example: Perform a calculation for a game**



```csharp
private DamageResult CalculateDamageDone()
{
    // Code omitted:
    //
    // Does an expensive calculation and returns
    // the result of that calculation.
}

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);
};
```

C# version 5.0 - Asynchronous programming

- **Await all tasks to complete**

```csharp
public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // 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<User[]> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id));
    return await Task.WhenAll(getUserTasks);
}
```

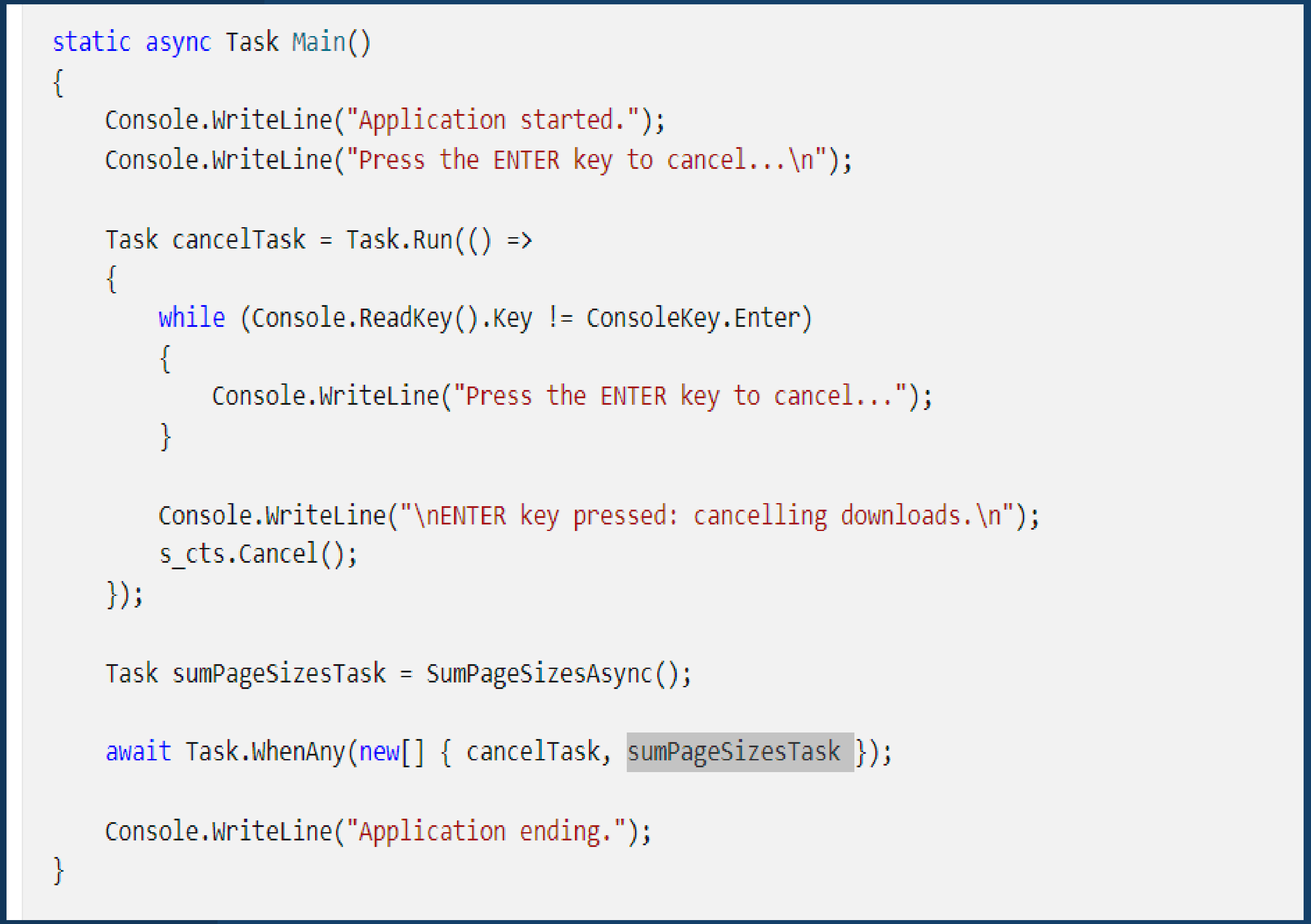
- **Await any task to complete**

```csharp
var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
while (breakfastTasks.Count > 0)
{
    Task finishedTask = await Task.WhenAny(breakfastTasks);
    if (finishedTask == eggsTask)
    {
        Console.WriteLine("Eggs are ready");
    }
    else if (finishedTask == baconTask)
    {
        Console.WriteLine("Bacon is ready");
    }
    else if (finishedTask == toastTask)
    {
        Console.WriteLine("Toast is ready");
    }
    breakfastTasks.Remove(finishedTask);
}
```

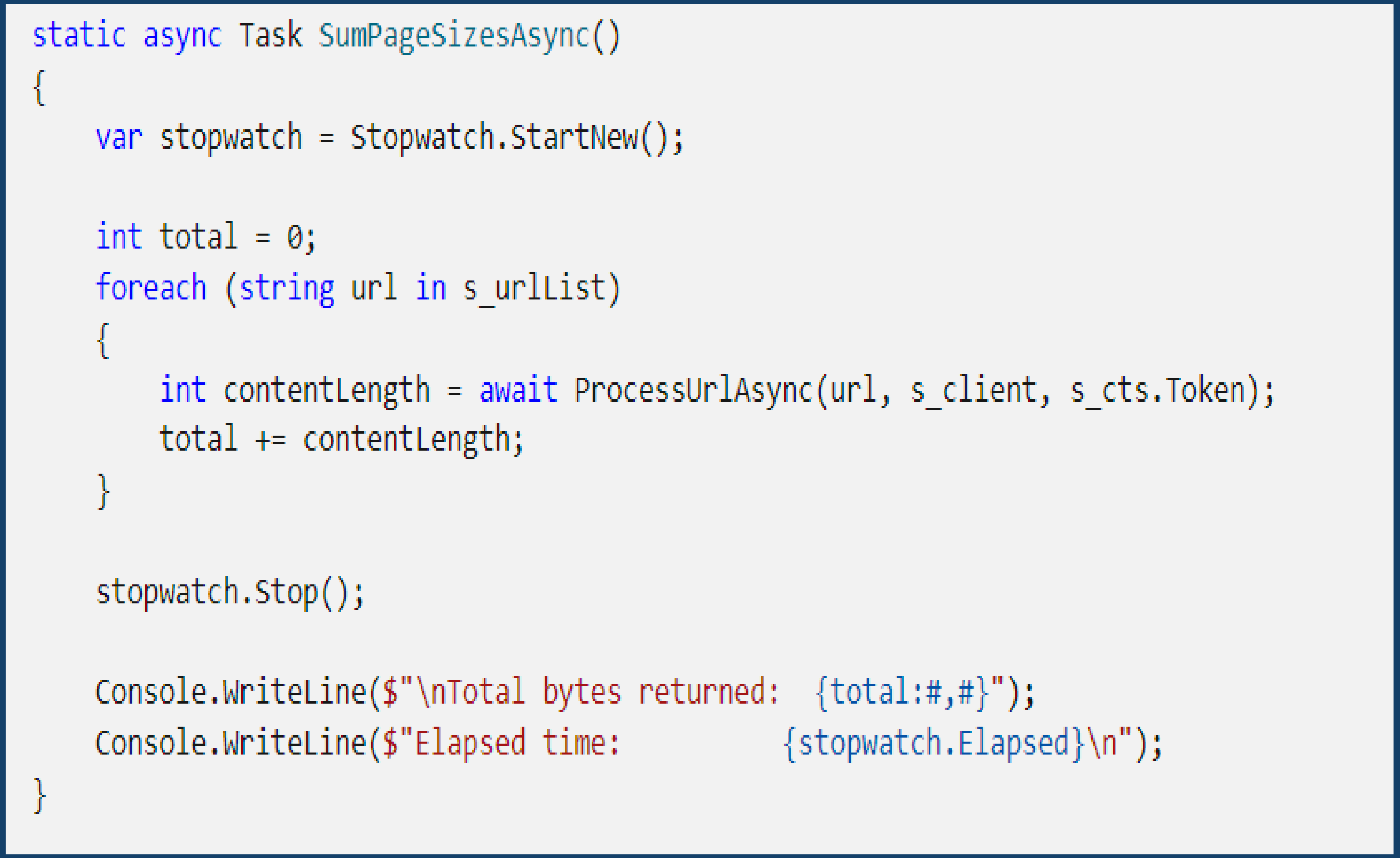
C# version 5.0 - Asynchronous programming

- **Cancel tasks**
- You can cancel an async console application if you don't want to wait for it to finish.
  - The CancellationTokenSource is used to signal a requested cancellation to a CancellationToken

static readonly CancellationTokenSource s_cts = new CancellationTokenSource();



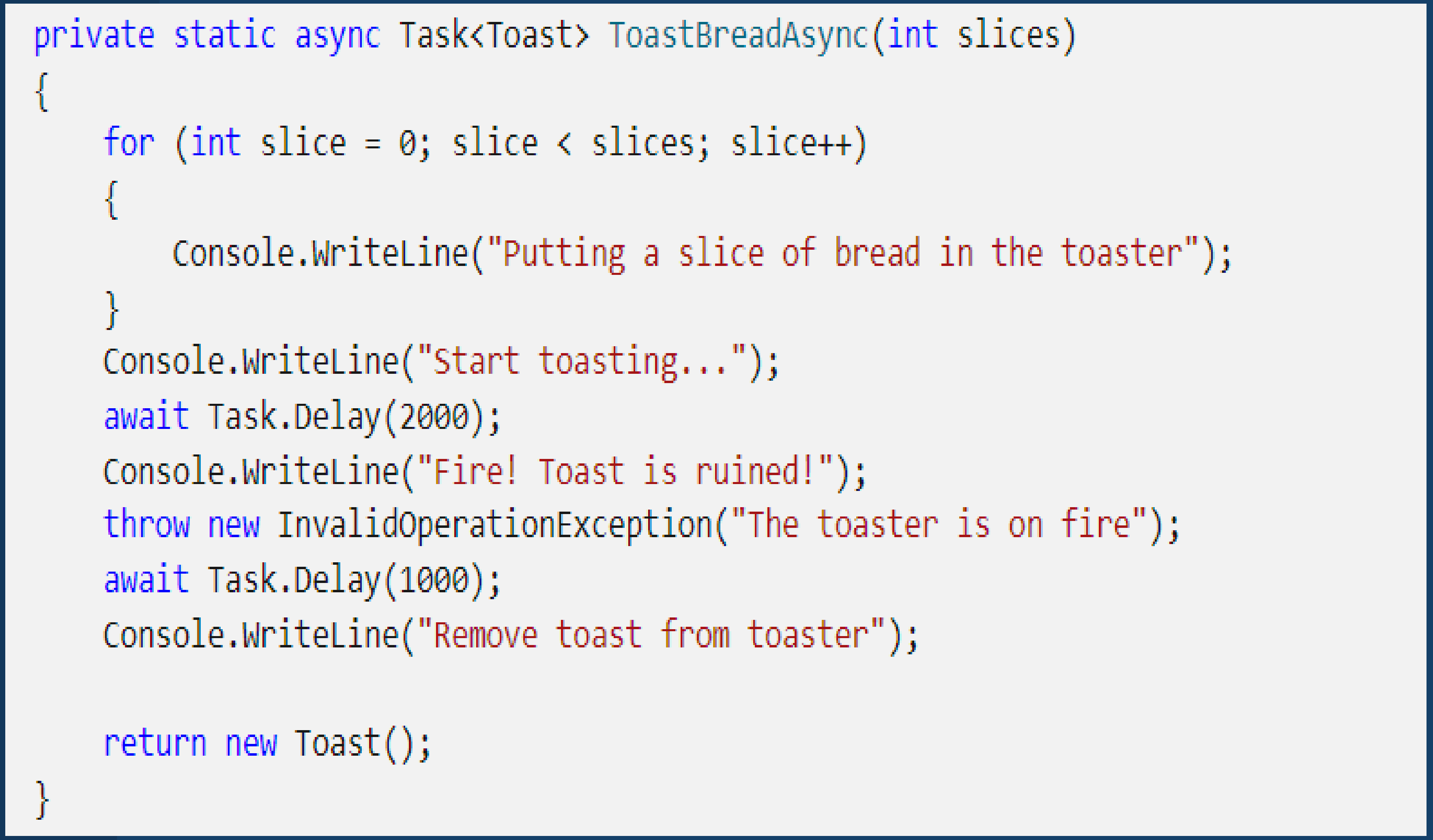
```
static async Task Main()
{
    Console.WriteLine("Application started.");
    Console.WriteLine("Press the ENTER key to cancel...\n");

    Task cancelTask = Task.Run(() =>
    {
        while (Console.ReadKey().Key != ConsoleKey.Enter)
        {
            Console.WriteLine("Press the ENTER key to cancel...");
        }

        Console.WriteLine("\nENTER key pressed: cancelling downloads.\n");
        s_cts.Cancel();
    });

    Task sumPageSizesTask = SumPageSizesAsync();

    await Task.WhenAny(new[] { cancelTask, sumPageSizesTask });

    Console.WriteLine("Application ending.");
}
```



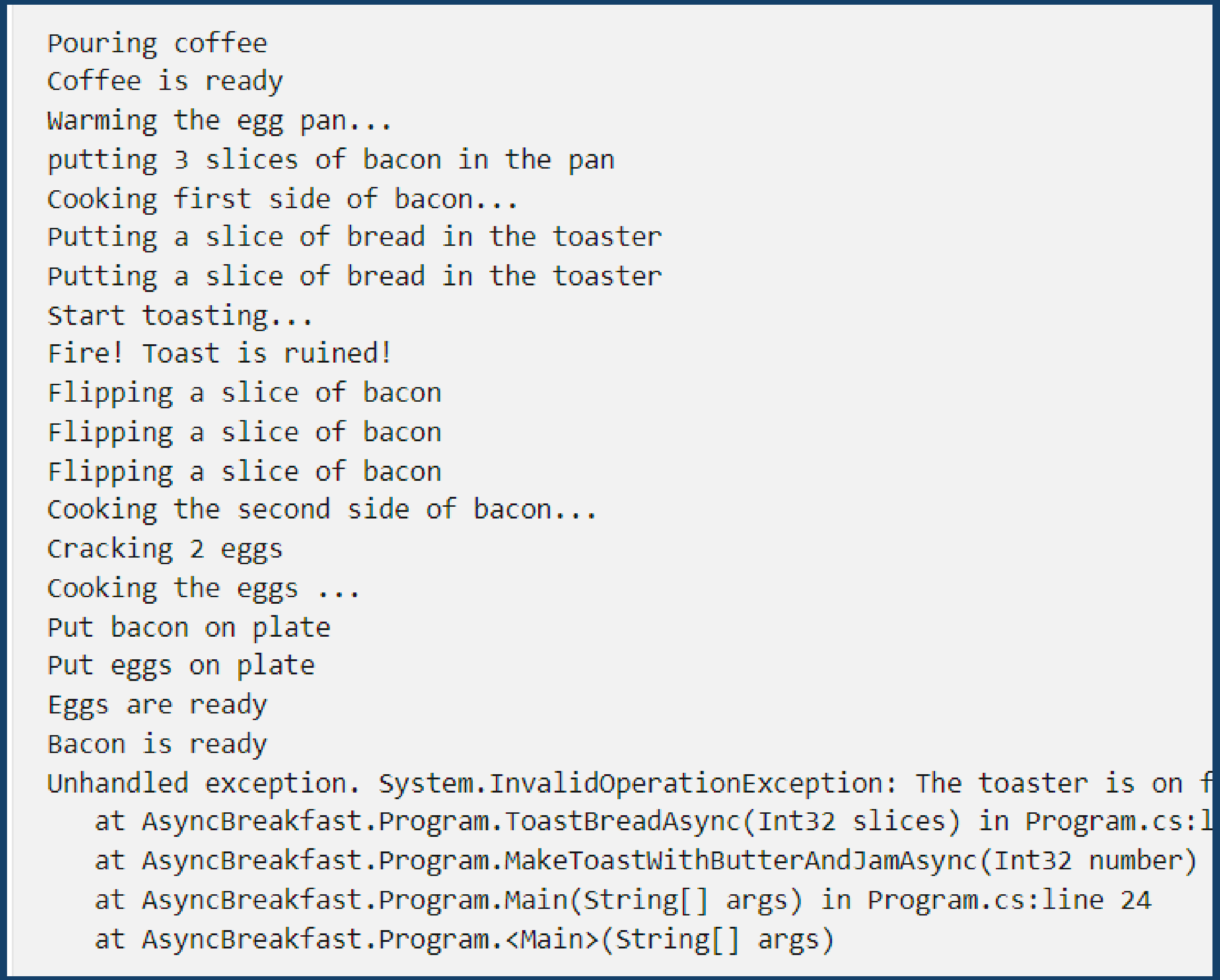
```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
    Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
}
```

Asynchronous programming

- **Asynchronous exceptions**
  - Asynchronous methods throw exceptions, just like their synchronous counterparts. Asynchronous support for exceptions and error handling strives for the same goals as asynchronous support in general.
    - You should write code that reads like a series of synchronous statements.
  - Tasks throw exceptions when they can't complete successfully.
  - The client code can catch those exceptions when a started task is **awaited**.



```
private static async Task<Toast> ToastBreadAsync(int slices)
{
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    Console.WriteLine("Start toasting...");
    await Task.Delay(2000);
    Console.WriteLine("Fire! Toast is ruined!");
    throw new InvalidOperationException("The toaster is on fire");
    await Task.Delay(1000);
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}
```



```
Pouring coffee
Coffee is ready
Warming the egg pan...
putting 3 slices of bacon in the pan
Cooking first side of bacon...
Putting a slice of bread in the toaster
Putting a slice of bread in the toaster
Start toasting...
Fire! Toast is ruined!
Flipping a slice of bacon
Flipping a slice of bacon
Flipping a slice of bacon
Cooking the second side of bacon...
Cracking 2 eggs
Cooking the eggs ...
Put bacon on plate
Put eggs on plate
Eggs are ready
Bacon is ready
Unhandled exception. System.InvalidOperationException: The toaster is on f
   at AsyncBreakfast.Program.ToastBreadAsync(Int32 slices) in Program.cs:l
   at AsyncBreakfast.Program.MakeToastWithButterAndJamAsync(Int32 number)
   at AsyncBreakfast.Program.Main(String[] args) in Program.cs:line 24
   at AsyncBreakfast.Program.<Main>(String[] args)
```

Asynchronous Programming

- Read more:
  - **Asynchronous programming with async and await: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/**
  - **Parallel Processing, Concurrency, và Async Programming: https://viblo.asia/p/parallel-processing-concurrency-va-async-programming-OeVKBdj0lkW**
  - **Other online resources**

C# version 5.0 – Caller into attributes

- To obtain information about the caller to a method

```
public void DoProcessing()
{
    TraceMessage("Something happened.");
}

public void TraceMessage(string message,
        [CallerMemberName] string memberName = "",
        [CallerFilePath] string sourceFilePath = "",
        [CallerLineNumber] int sourceLineNumber = 0)
{
    Trace.WriteLine("message: " + message);
    Trace.WriteLine("member name: " + memberName);
    Trace.WriteLine("source file path: " + sourceFilePath);
    Trace.WriteLine("source line number: " + sourceLineNumber);
}

// Sample Output:
//  message: Something happened.
//  member name: DoProcessing
//  source file path: c:\Visual Studio Projects\CallerInfoCS\CallerInfoCS\Form1.cs
//  source line number: 31
```

C# version 6.0

- Static imports
- Exception filters
- Auto-property initializers
- Expression bodied members
- Null propagator
- String interpolation
- nameof operator

C# version 6.0 - Static imports

- Allows specifying a static class in a using clause
  - All its accessible static members become available without qualification

```
using System.Console;
using System.Math;
class Program
{
    static void Main()
    {
        WriteLine(Pow(2, 10));
    }
}
```

C# version 6.0 - Static imports

- *using static* imports only accessible static members and nested types declared in the specified type. Inherited members are not imported.
- *using static* makes extension methods declared in the specified type available for extension method lookup. However, the names of the extension methods are not imported into scope for unqualified reference in code.

C# version 6.0 - Exception filters

```
var streamTask = client.GetStringAsync("https://localHost:10000");
try
{
    var responseText = await streamTask;
    return responseText;
}
catch (HttpRequestException e) when (e.Message.Contains("301"))
{
    return "Site Moved";
}
catch (HttpRequestException e) when (e.Message.Contains("404"))
{
    return "Page Not Found";
}
catch (HttpRequestException e)
{
    return e.Message;
}
```

C# version 6.0 - Auto-property initializers

- You can add an initializer to an auto-property, just as you can to a field:

```
public class Person
{
    public string FirstName { get; set; } = "Nikolay";
    public string LastName { get; set; } = "Kostov";
}
```

C# version 6.0 - Getter-Only Auto-Properties

- Auto-properties can now be declared without a setter:

```
public string FirstName { get; } = "Nikolay";
```

C# version 6.0 - Expression bodied members

- The => token is supported in two forms: as the lambda operator and as a separator of a member name and the member implementation in an expression body definition.

```
string[] words = { "bot", "apple", "apricot" };
int minimalLength = words
  .Where(w => w.StartsWith("a"))
  .Min(w => w.Length);
Console.WriteLine(minimalLength);   // output: 5

int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (interim, next) => interim * next);
Console.WriteLine(product);   // output: 280
```

```
public override string ToString() => $"{fname} {lname}".Trim();
```

C# version 6.0 - Null propagator

- Lets you access members and elements only when the receiver is not-null
  - Providing a null result otherwise

  - Can be used together with the null coalescing operator ??:

```
int? length = customers?.Length; //null if customers is null
Customer first = customers?[0]; //null if customers is null
```

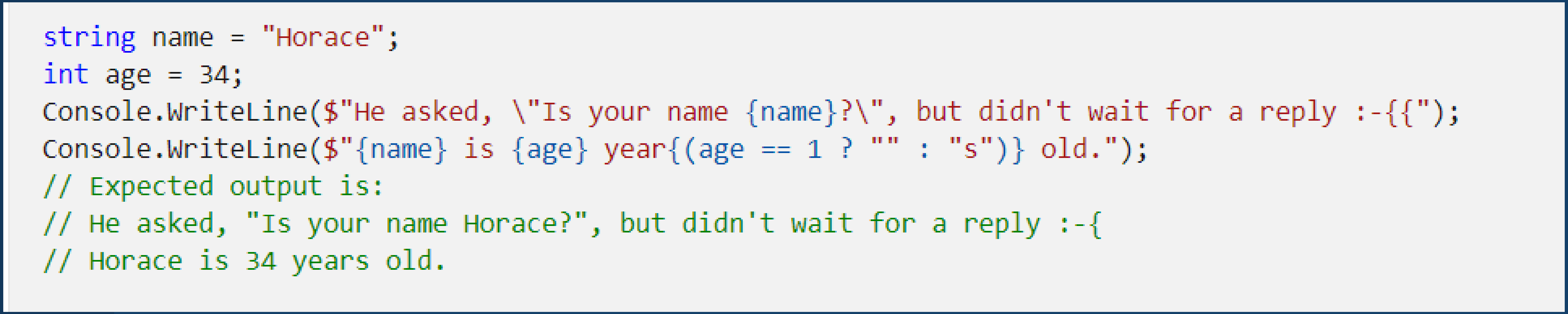
  - Can be used together with the null coalescing operator ??:

```
int length = customers?.Length ?? 0; // 0 if customers null
```

```
int? first = customers?[0].Orders?.Count();
```

C# version 6.0 - String interpolation

An interpolated string is a string literal that might contain *interpolation expressions*. When an interpolated string is resolved to a result string, items with interpolation expressions are replaced by the string representations of the expression results



```
string name = "Horace";
int age = 34;
Console.WriteLine($"He asked, \"Is your name {name}?\", but didn't wait for a reply :-{{");
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
// Expected output is:
// He asked, "Is your name Horace?", but didn't wait for a reply :-{
// Horace is 34 years old.
```

C# version 6.0 - String interpolation

- **Structure of an interpolated string**
- {<interpolationExpression>[,<alignment>][:<formatString>]}

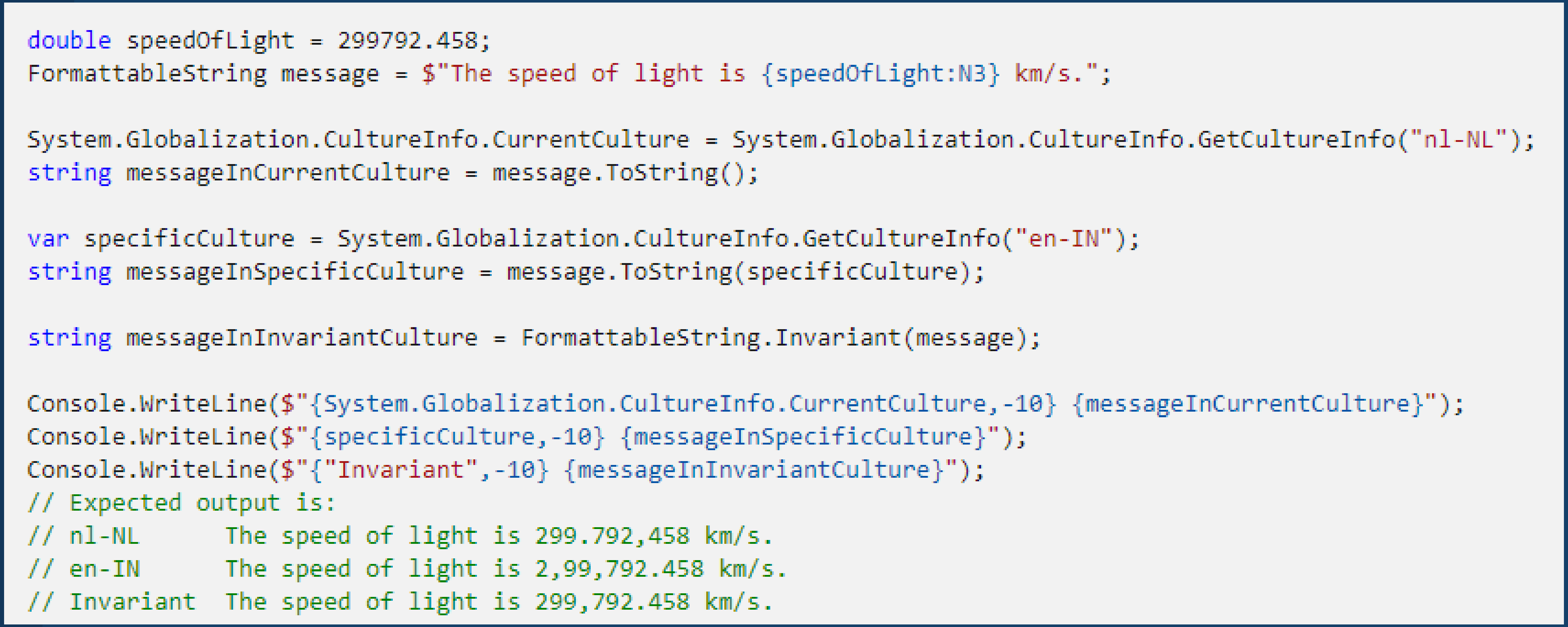
```
Console.WriteLine($"|{"Left",-7}|{"Right",7}|");

const int FieldWidthRightAligned = 20;
Console.WriteLine($"{Math.PI,FieldWidthRightAligned} - default formatting of the pi number");
Console.WriteLine($"{Math.PI,FieldWidthRightAligned:F3} - display only three decimal digits of the pi number");
// Expected output is:
// |Left   |  Right|
//     3.14159265358979 - default formatting of the pi number
//                3.142 - display only three decimal digits of the pi number
```

```
string name = "Horace";
int age = 34;
Console.WriteLine($"He asked, \"Is your name {name}?\", but didn't wait for a reply :-{{");
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
// Expected output is:
// He asked, "Is your name Horace?", but didn't wait for a reply :-{
// Horace is 34 years old.
```

C# version 6.0 - String interpolation

- **Implicit conversions and how to specify IFormatProvider implementation**
  - To a String instance: uses the CurrentCulture to format expression results.
  - To a FormattableString instance: represents a composite format string along with the expression results to be formatted. Allows you to create multiple result strings with culture-specific content from a single FormattableString instance.
  - Conversion of an interpolated string to an IFormattable instance that also allows you to create multiple result strings with culture-specific content from a single IFormattable instance.

C# version 6.0 - String interpolation



```csharp
double speedOfLight = 299792.458;
FormattableString message = $"The speed of light is {speedOfLight:N3} km/s.";

System.Globalization.CultureInfo.CurrentCulture = System.Globalization.CultureInfo.GetCultureInfo("nl-NL");
string messageInCurrentCulture = message.ToString();

var specificCulture = System.Globalization.CultureInfo.GetCultureInfo("en-IN");
string messageInSpecificCulture = message.ToString(specificCulture);

string messageInInvariantCulture = FormattableString.Invariant(message);

Console.WriteLine($"{System.Globalization.CultureInfo.CurrentCulture,-10} {messageInCurrentCulture}");
Console.WriteLine($"{specificCulture,-10} {messageInSpecificCulture}");
Console.WriteLine($"{"Invariant",-10} {messageInInvariantCulture}");
// Expected output is:
// nl-NL      The speed of light is 299.792,458 km/s.
// en-IN      The speed of light is 2,99,792.458 km/s.
// Invariant  The speed of light is 299,792.458 km/s.
```

C# version 6.0 - Nameof expression

- A nameof expression produces the name of a variable, type, or member as the string constant

```
Console.WriteLine(nameof(System.Collections.Generic));  // output: Generic
Console.WriteLine(nameof(List<int>));  // output: List
Console.WriteLine(nameof(List<int>.Count));  // output: Count
Console.WriteLine(nameof(List<int>.Add));  // output: Add

var numbers = new List<int> { 1, 2, 3 };
Console.WriteLine(nameof(numbers));  // output: numbers
Console.WriteLine(nameof(numbers.Count));  // output: Count
Console.WriteLine(nameof(numbers.Add));  // output: Add
```

C# version 6.0 – Index Initializers

- A new syntax to object initializers allowing to set values to keys through the indexers
  - C# 5.0 and earlier

```
var numbers = new Dictionary<int, string> {
   { 7, "seven" },
   { 9, "nine" },
   { 13, "thirteen" }
};
```

  - C# 6.0

```
var numbers = new Dictionary<int, string> {
   [7] = "seven",
   [9] = "nine",
   [13] = "thirteen"
};
```

C# version 6.0 – Await in catch/finally

- Await is now allowed in catch/finally C# 5.0 and earlier

```
var input = new StreamReader(fileName);
var log = new StreamWriter(logFileName);
try {
    var line = await input.ReadLineAsync();
}
catch (IOException ex) {
    await log.WriteLineAsync(ex.ToString());
}
finally {
    if (log != null) await log.FlushAsync();
}
```

C# version 7 - Tuples and deconstruction

- Tuples are lightweight data structures that contain multiple fields to represent the data members

```
(string Alpha, string Beta) namedLetters = ("a", "b");
Console.WriteLine($"{namedLetters.Alpha}, {namedLetters.Beta}");
```

```
var alphabetStart = (Alpha: "a", Beta: "b");
Console.WriteLine($"{alphabetStart.Alpha}, {alphabetStart.Beta}");
```

```
(int max, int min) = Range(numbers);
Console.WriteLine(max);
Console.WriteLine(min);
```

C# version 7 - Tuples and deconstruction

- Discards are temporary, write-only variables used in assignments when you don't care about the value assigned

```
public class Point
{
    public Point(double x, double y)
        => (X, Y) = (x, y);

    public double X { get; }
    public double Y { get; }

    public void Deconstruct(out double x, out double y) =>
        (x, y) = (X, Y);
}
```

```
var p = new Point(3.14, 2.71);
(double X, double Y) = p;
```

C# version 7 - Pattern matching

- Test variables for their type, values or the values of their properties.
- Supports is expressions and switch expressions. Each enables inspecting an object and its properties to determine if that object satisfies the sought pattern.
- Use the when keyword to specify additional rules to the pattern.
- The is pattern expression extends the familiar is operator to query an object about its type and assign the result in one instruction.

```
if (input is int count)
    sum += count;
```

C# version 7 - Pattern matching

```csharp
public static int SumPositiveNumbers(IEnumerable<object> sequence)
{
    int sum = 0;
    foreach (var i in sequence)
    {
        switch (i)
        {
            case 0:
                break;
            case IEnumerable<int> childSequence:
            {
                foreach(var item in childSequence)
                    sum += (item > 0) ? item : 0;
                break;
            }
            case int n when n > 0:
                sum += n;
                break;
            case null:
                throw new NullReferenceException("Null found in sequence");
            default:
                throw new InvalidOperationException("Unrecognized type");
        }
    }
    return sum;
}
```

C# version 7 - Async main

An *async main* method enables you to use await in yothod

- Without async main

```
static int Main()
{
    return DoAsyncWork().GetAwaiter().GetResult();
}
```

- With async main

```
static async Task<int> Main()
{
    // This could also be replaced with the body
    // DoAsyncWork, including its await expressions:
    return await DoAsyncWork();
}
```

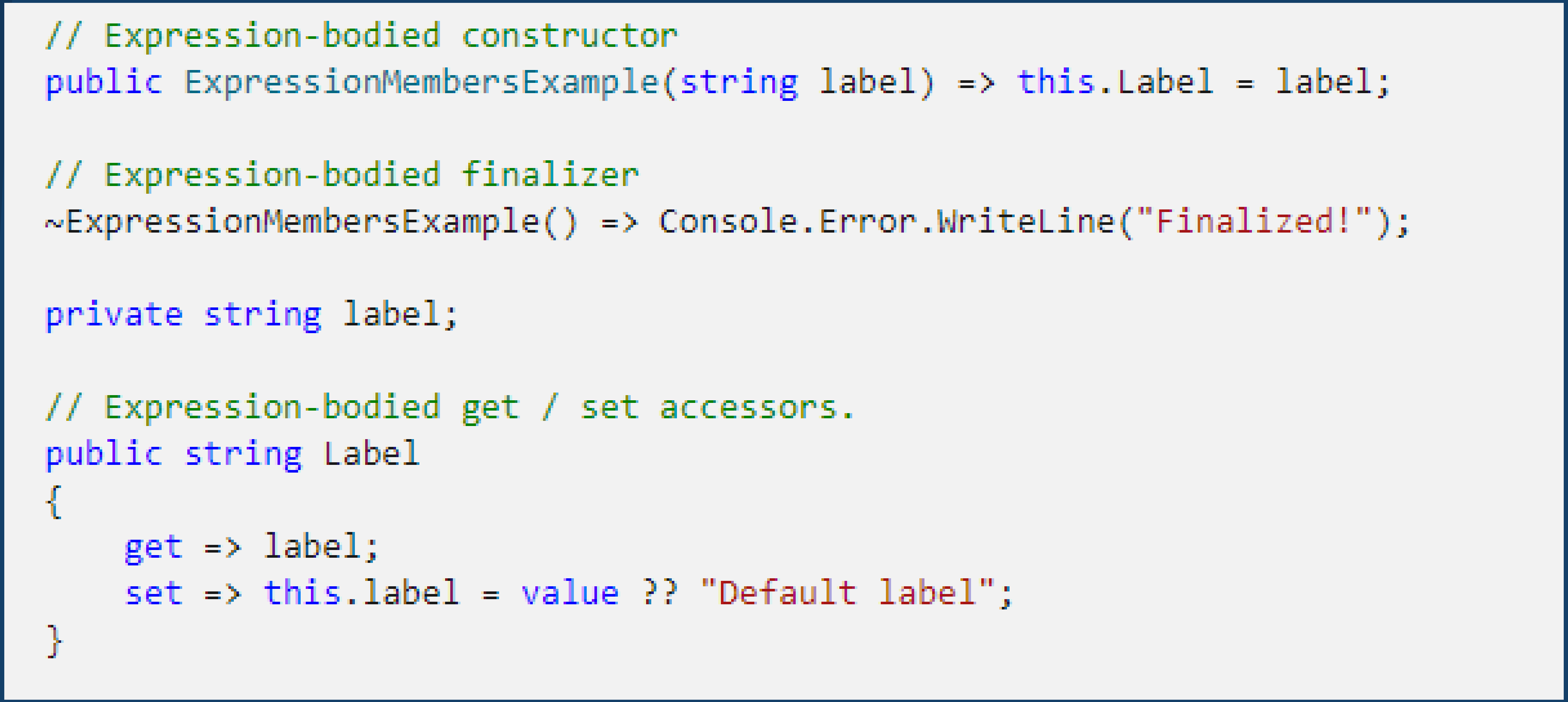
C# version 7 – Local functions

- Local functions enable you to declare methods inside the context of another method.

```
public Task<string> PerformLongRunningWork(string address, int index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required", paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name", paramName: nameof(name));

    return longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    }
}
```

C# version 7 – More expression-bodied members

- In C# 7.0, you can implement constructors, finalizers, and get and set accessors on properties and indexers



```
// Expression-bodied constructor
public ExpressionMembersExample(string label) => this.Label = label;

// Expression-bodied finalizer
~ExpressionMembersExample() => Console.Error.WriteLine("Finalized!");

private string label;

// Expression-bodied get / set accessors.
public string Label
{
    get => label;
    set => this.label = value ?? "Default label";
}
```

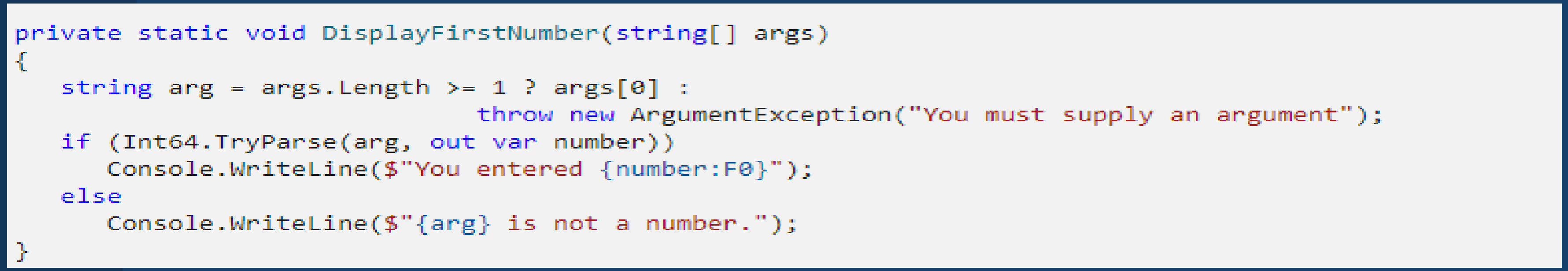
C# version 7 – More expression-bodied members

- In C# 7.0, you can implement constructors, finalizers, and get and set accessors on properties and indexers

```
// Expression-bodied constructor
public ExpressionMembersExample(string label) => this.Label = label;

// Expression-bodied finalizer
~ExpressionMembersExample() => Console.Error.WriteLine("Finalized!");

private string label;

// Expression-bodied get / set accessors.
public string Label
{
    get => label;
    set => this.label = value ?? "Default label";
}
```

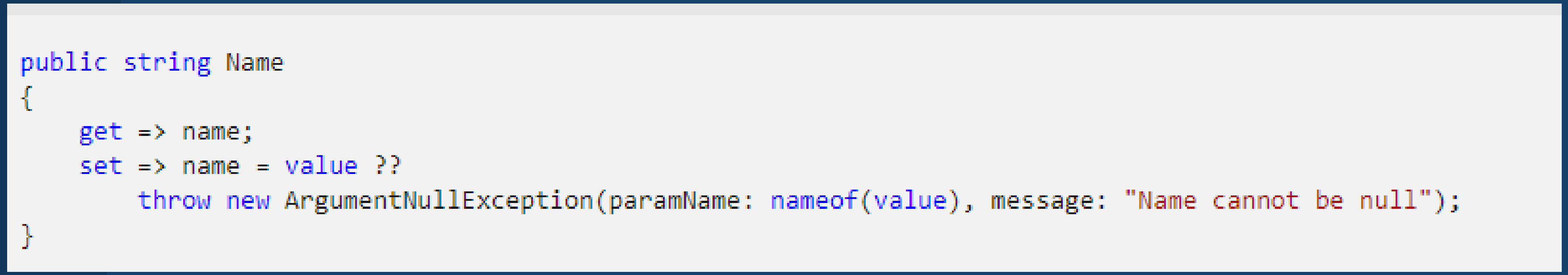
C# version 7 – throw expression

- Throw can be used as an expression as well as a statement. This allows an exception to be thrown in contexts that were previously unsupported
  - the conditional operator.



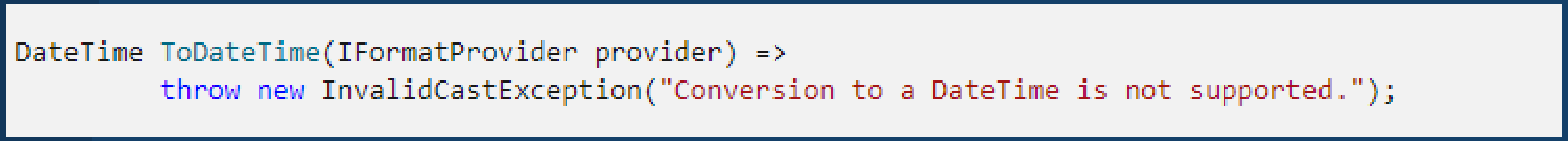
```
private static void DisplayFirstNumber(string[] args)
{
   string arg = args.Length >= 1 ? args[0] :
                              throw new ArgumentException("You must supply an argument");
   if (Int64.TryParse(arg, out var number))
      Console.WriteLine($"You entered {number:F0}");
   else
      Console.WriteLine($"{arg} is not a number.");
}
```

  - the null-coalescing operator



```
public string Name
{
    get => name;
    set => name = value ??
        throw new ArgumentNullException(paramName: nameof(value), message: "Name cannot be null");
}
```

  - an expression-bodied lambda or method



```
DateTime ToDateTime(IFormatProvider provider) =>
         throw new InvalidCastException("Conversion to a DateTime is not supported.");
```

C# version 7 – Default literal expressions

- Default literal expressions are an enhancement to default value expressions. These expressions initialize a variable to the default value. Where you previously would write:
  - Before

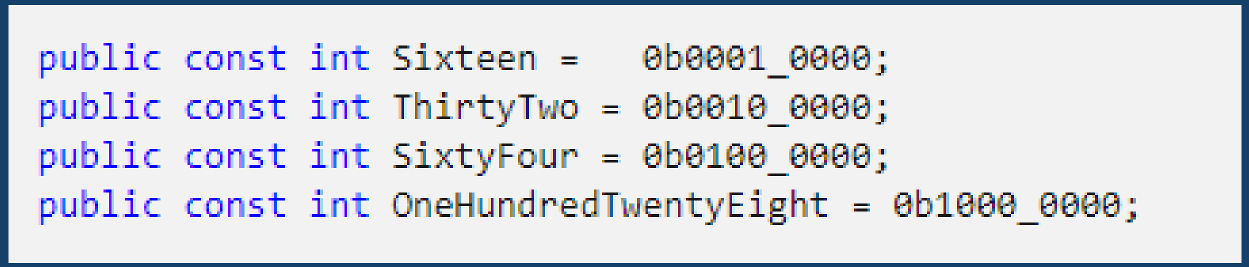
```
Func<string, bool> whereClause = default(Func<string, bool>);
```

  - Now

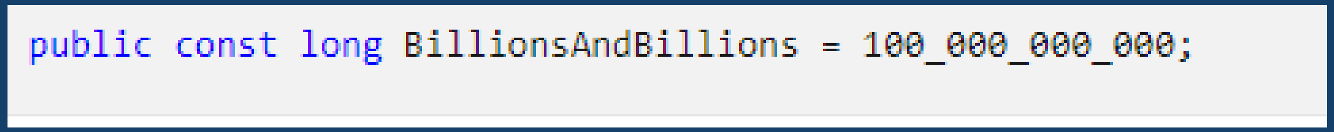
```
Func<string, bool> whereClause = default;
```

C# version 7 – Numeric literal syntax improvements

- C# 7.0 includes two new features to write numbers in the most readable fashion for the intended use: binary literals, and digit separators.
  - Binary *literals*

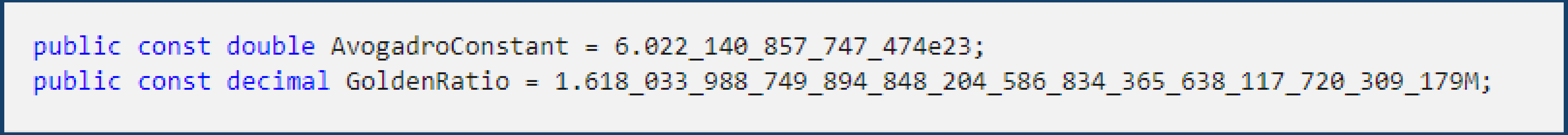


```
public const int Sixteen =   0b0001_0000;
public const int ThirtyTwo = 0b0010_0000;
public const int SixtyFour = 0b0100_0000;
public const int OneHundredTwentyEight = 0b1000_0000;
```



```
public const long BillionsAndBillions = 100_000_000_000;
```

  - *Digit separators*



```
public const double AvogadroConstant = 6.022_140_857_747_474e23;
public const decimal GoldenRatio = 1.618_033_988_749_894_848_204_586_834_365_638_117_720_309_179M;
```

C# version 7.0 – Ref locals and returns

- This feature enables algorithms that use and return references to variables defined elsewhere

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

```
ref var item = ref MatrixSearch.Find(matrix, (val) => val == 42);
Console.WriteLine(item);
item = 24;
Console.WriteLine(matrix[4, 2]);
```

C# version 8.0

- Readonly members
- Default interface methods
- Pattern matching enhancements
- Using declarations
- Static local functions
- Disposable ref structs
- Nullable reference types
- Asynchronous streams
- Asynchronous disposable
- Indices and ranges
- Null-coalescing assignment
- Unmanaged constructed types
- Stackalloc in nested expressions
- Enhancement of interpolated verbatim strings

C# version 8.0 - Readonly members

- Allows to apply the readonly modifier to any member of a struct. It indicates that the member doesn't modify the state.

```
public struct XValue
{
  private int X { get; set; }

  public readonly int IncreaseX()
  {
      // This will not compile: C# 8
      // X = X + 1;

       var newX = X + 1; // OK
       return newX;
   }
}
```

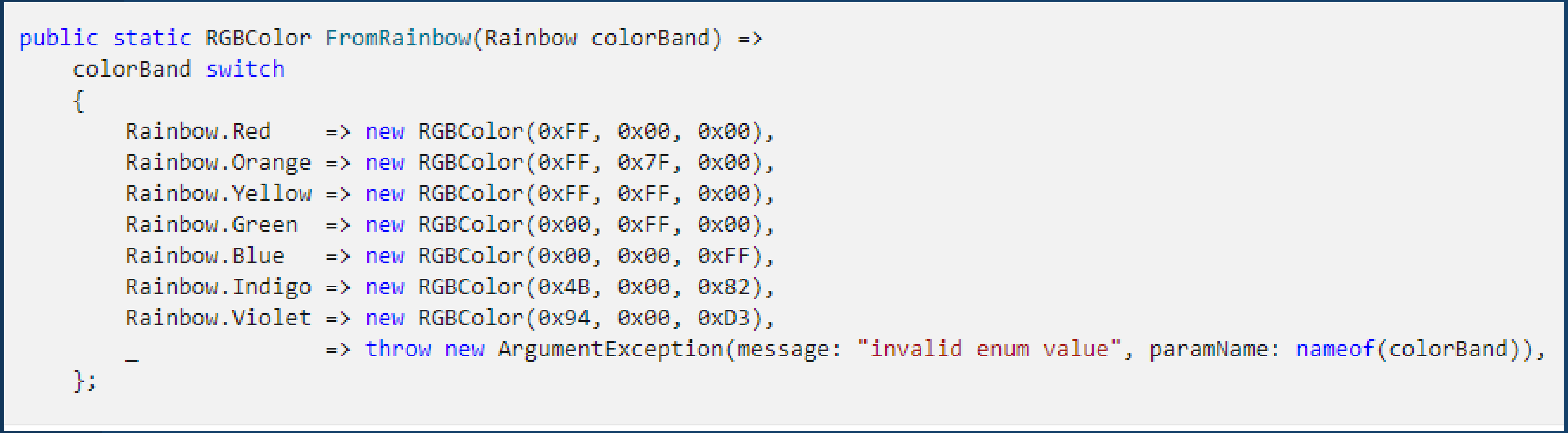
C# version 8.0 - Default interface methods

- Add new functionality to the interfaces of your libraries and ensure the backward compatibility with code written for older versions of those interfaces.

```
interface IWriteLine
{
 public void WriteLine()
 {
  Console.WriteLine("Wow C# 8!");
 }
}
```

C# version 8.0 - Pattern matching enhancements

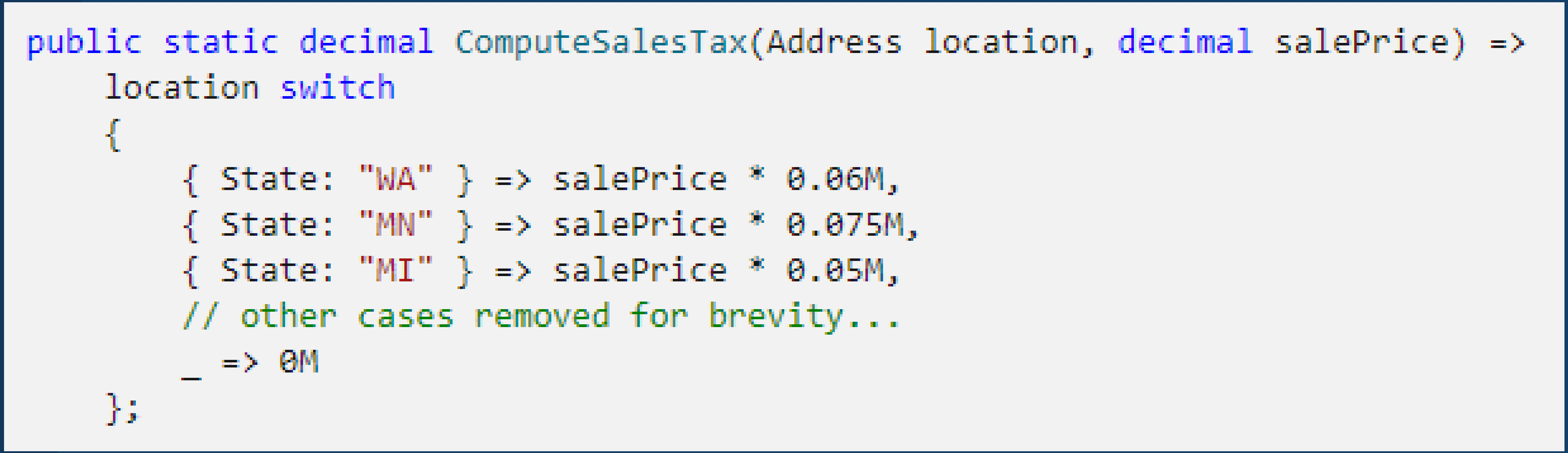
- *Switch* expressions enable you to use more concise expression syntax



```
public static RGBColor FromRainbow(Rainbow colorBand) =>
    colorBand switch
    {
        Rainbow.Red    => new RGBColor(0xFF, 0x00, 0x00),
        Rainbow.Orange => new RGBColor(0xFF, 0x7F, 0x00),
        Rainbow.Yellow => new RGBColor(0xFF, 0xFF, 0x00),
        Rainbow.Green  => new RGBColor(0x00, 0xFF, 0x00),
        Rainbow.Blue   => new RGBColor(0x00, 0x00, 0xFF),
        Rainbow.Indigo => new RGBColor(0x4B, 0x00, 0x82),
        Rainbow.Violet => new RGBColor(0x94, 0x00, 0xD3),
        _              => throw new ArgumentException(message: "invalid enum value", paramName: nameof(colorBand)),
    };
```

C# version 8.0 - Pattern matching enhancements

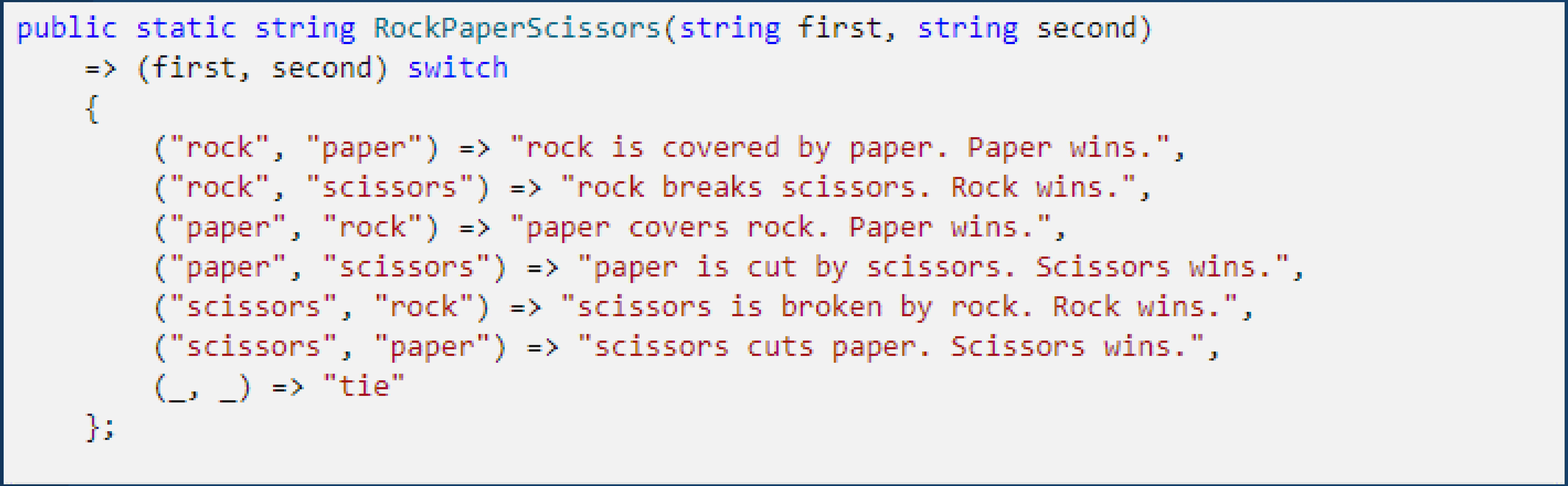
- The **property pattern** enables you to match on properties of the object examined



```csharp
public static decimal ComputeSalesTax(Address location, decimal salePrice) =>
    location switch
    {
        { State: "WA" } => salePrice * 0.06M,
        { State: "MN" } => salePrice * 0.075M,
        { State: "MI" } => salePrice * 0.05M,
        // other cases removed for brevity...
        _ => 0M
    };
```

C# version 8.0 - Tuple patterns

- **Tuple patterns** allow you to switch based on multiple values expressed as a tuple. The following code shows a switch expression for the game rock, paper, scissors:



```
public static string RockPaperScissors(string first, string second)
    => (first, second) switch
    {
        ("rock", "paper") => "rock is covered by paper. Paper wins.",
        ("rock", "scissors") => "rock breaks scissors. Rock wins.",
        ("paper", "rock") => "paper covers rock. Paper wins.",
        ("paper", "scissors") => "paper is cut by scissors. Scissors wins.",
        ("scissors", "rock") => "scissors is broken by rock. Rock wins.",
        ("scissors", "paper") => "scissors cuts paper. Scissors wins.",
        (_, _) => "tie"
    };
```

C# version 8.0 - Tuple patterns

- Some types include a **Deconstruct** method that deconstructs its properties into discrete variables. When a **Deconstruct** method is accessible, you can use **positional patterns** to inspect properties of the object and use those properties for a pattern

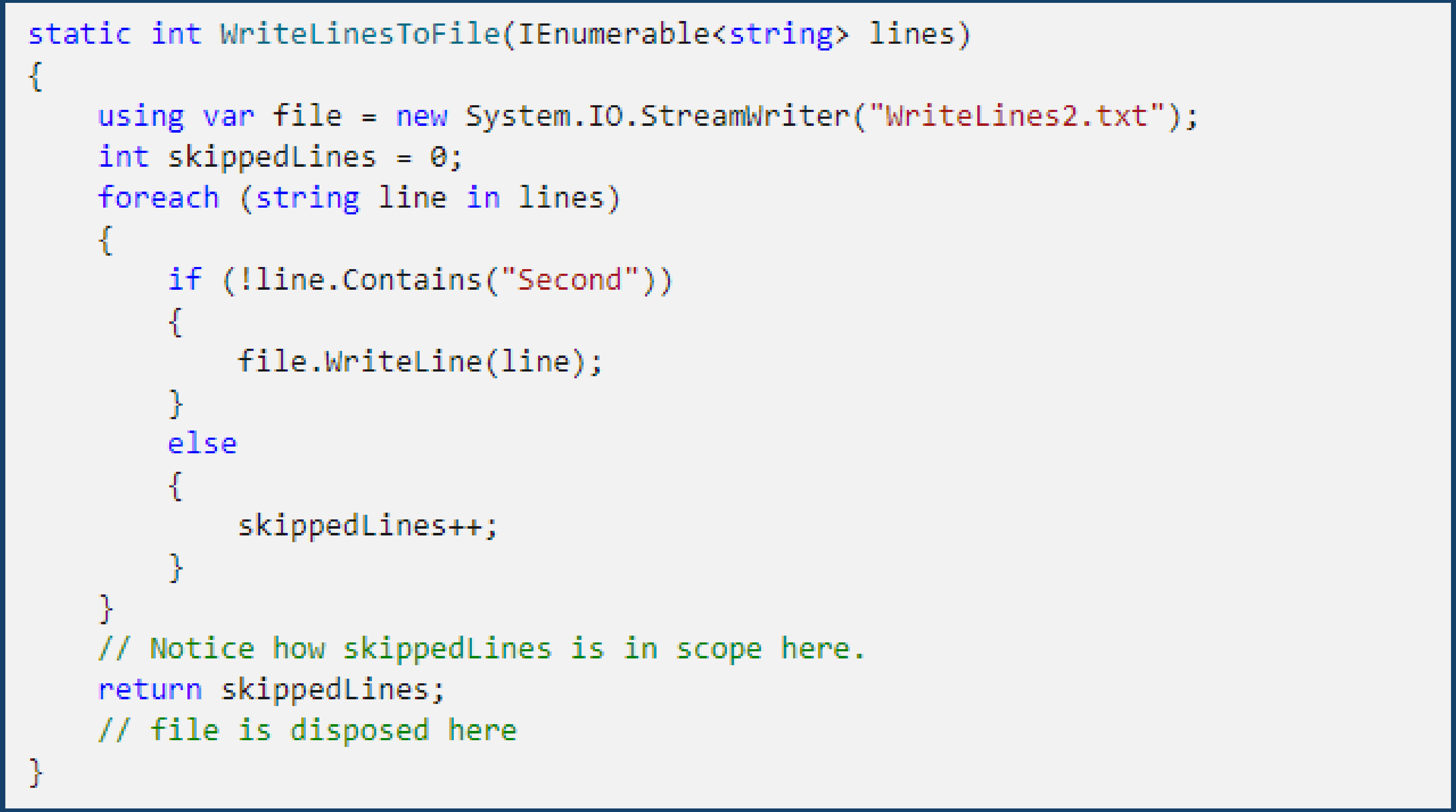
```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);

    public void Deconstruct(out int x, out int y) =>
        (x, y) = (X, Y);
}
```

```
public enum Quadrant
{
    Unknown,
    Origin,
    One,
    Two,
    Three,
    Four,
    OnBorder
}
```

```
static Quadrant GetQuadrant(Point point) => point switch
{
    (0, 0) => Quadrant.Origin,
    var (x, y) when x > 0 && y > 0 => Quadrant.One,
    var (x, y) when x < 0 && y > 0 => Quadrant.Two,
    var (x, y) when x < 0 && y < 0 => Quadrant.Three,
    var (x, y) when x > 0 && y < 0 => Quadrant.Four,
    var (_, _) => Quadrant.OnBorder,
    _ => Quadrant.Unknown
};
```

C# version 8.0 - Using declarations

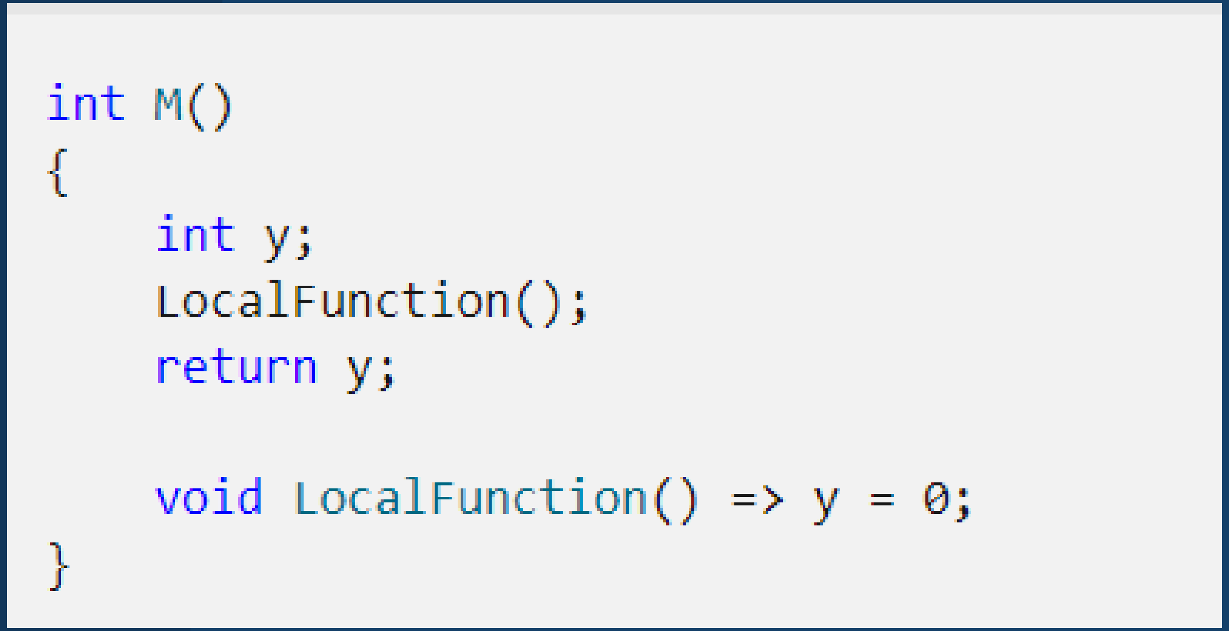
- A **using declaration** is a variable declaration preceded by the using keyword. It tells the compiler that the variable being declared should be disposed at the end of the enclosing scope



```
static int WriteLinesToFile(IEnumerable<string> lines)
{
    using var file = new System.IO.StreamWriter("WriteLines2.txt");
    int skippedLines = 0;
    foreach (string line in lines)
    {
        if (!line.Contains("Second"))
        {
            file.WriteLine(line);
        }
        else
        {
            skippedLines++;
        }
    }
    // Notice how skippedLines is in scope here.
    return skippedLines;
    // file is disposed here
}
```

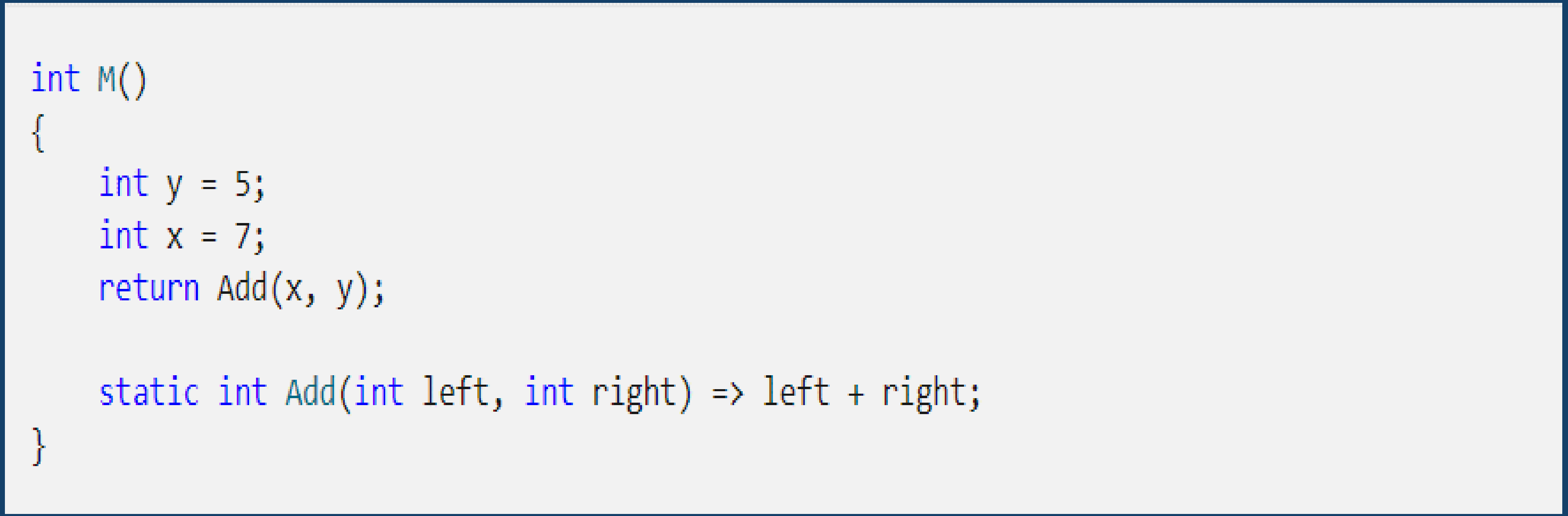
C# version 8.0 - Static local functions

- **Add** the static modifier to local functions to ensure that local function doesn't capture (reference) any variables from the enclosing scope



```
int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}
```

LocalFunction access variables in the enclosing scope



```
int M()
{
    int y = 5;
    int x = 7;
    return Add(x, y);

    static int Add(int left, int right) => left + right;
}
```

LocalFunction doesm'ta access variables in the enclosing scope

C# version 8.0 - Disposable ref structs

- Allows to use ref structs/read-only ref struct with “using” pattern.

```
ref struct Test {
    public void Dispose() { ... }
}
using var local = new Test();
// local is disposed here!
```

C# version 8.0 - Nullable reference types

C# 8.0 introduces **nullable reference types** and **non-nullable reference types** that enable you to make important statements about the properties for reference type variables:

A reference isn't supposed to be null:

- The variable must be initialized to a non-null value.
- The variable can never be assigned the value null.

A reference may be null. The compiler enforces different rules to ensure that you've correctly checked for a null reference:

- The variable may only be dereferenced when the compiler can guarantee that the value isn't null.
- These variables may be initialized with the default null value and may be assigned the value null in other code.

C# version 8.0 - Nullable reference types

C# 8.0 introduces **nullable reference types** and **non-nullable reference types** that enable you to make important statements about the properties for reference type variables:

A reference isn't supposed to be null:

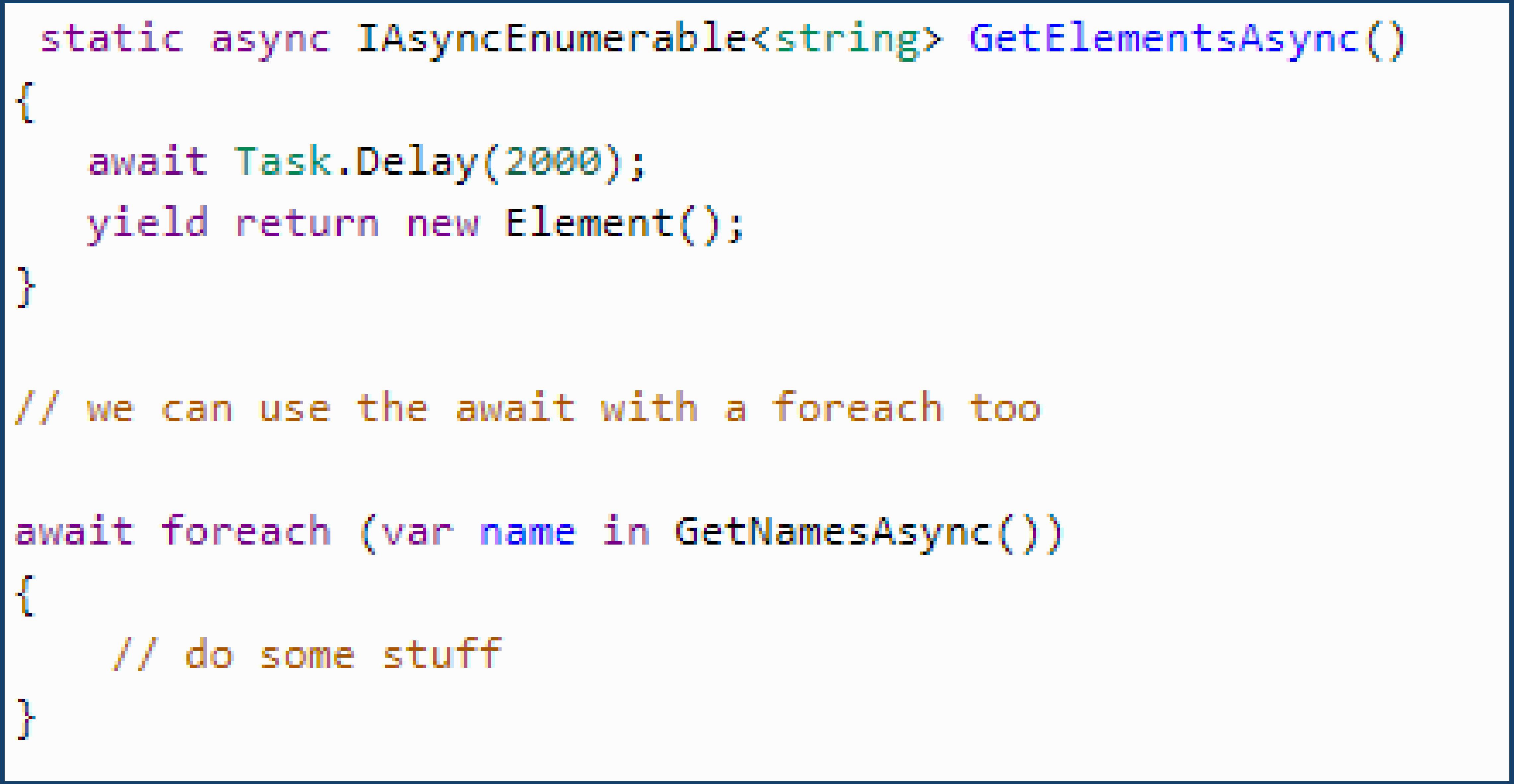
- The variable must be initialized to a non-null value.
- The variable can never be assigned the value null.

A reference may be null. The compiler enforces different rules to ensure that you've correctly checked for a null reference:

- The variable may only be dereferenced when the compiler can guarantee that the value isn't null.
- These variables may be initialized with the default null value and may be assigned the value null in other code.

C# version 8.0 - Asynchronous streams

- The IAsyncEnumerable which allows us to yield results asynchronously



```
 static async IAsyncEnumerable<string> GetElementsAsync()
{
   await Task.Delay(2000);
   yield return new Element();
}

// we can use the await with a foreach too

await foreach (var name in GetNamesAsync())
{
    // do some stuff
}
```

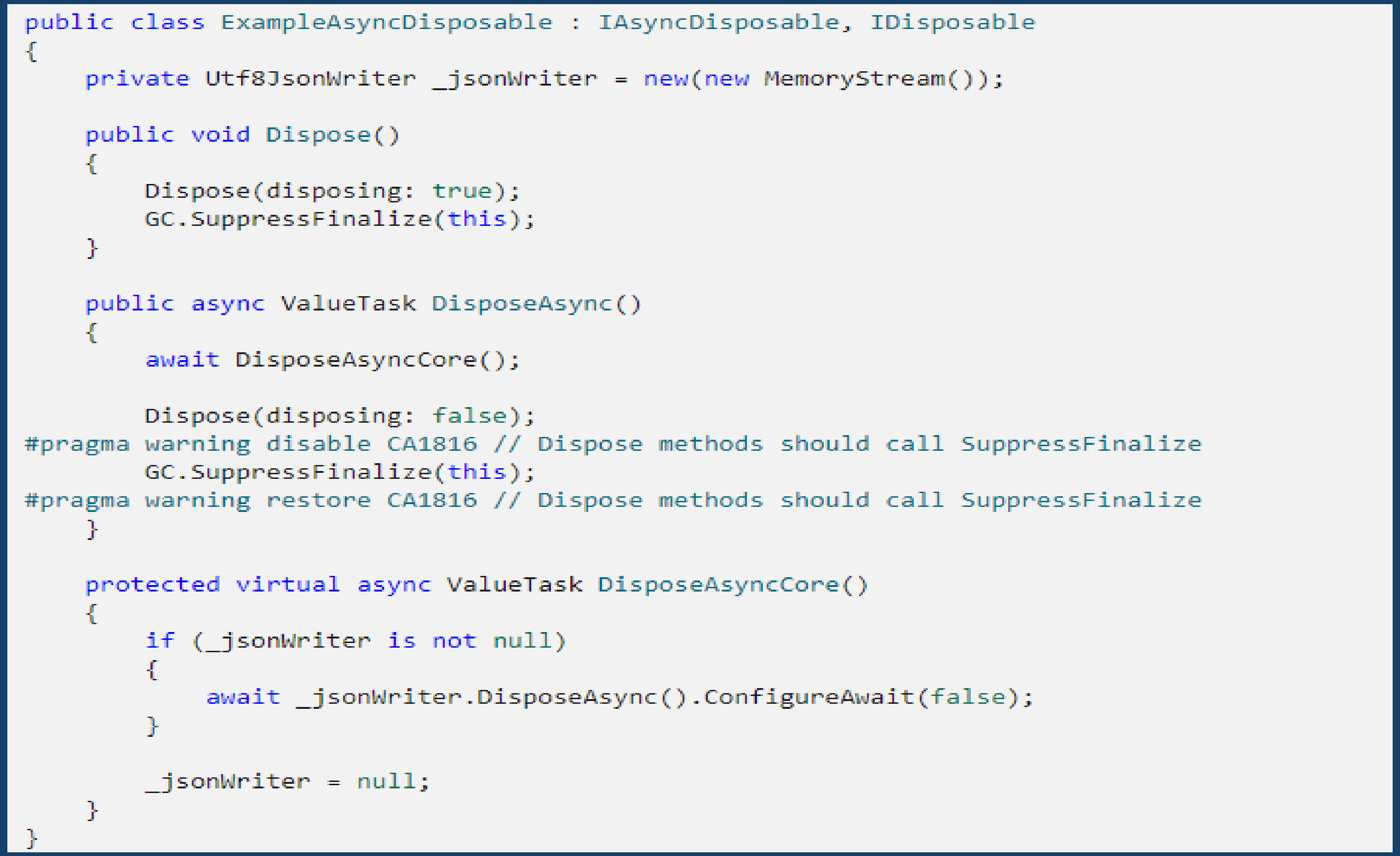
C# version 8.0 - Asynchronous disposable

- Supports asynchronous disposable types that implement the *System.IAsyncDisposable* interface
  - **The DisposeAsync() method:** to free unmanaged resources, perform general cleanup, and to indicate that the finalizer, if one is present, need not run. It has a standard implementation

```
public async ValueTask DisposeAsync()
{
    // Perform async cleanup.
    await DisposeAsyncCore();

    // Dispose of unmanaged resources.
    Dispose(false);

#pragma warning disable CA1816 // Dispose methods should call SuppressFinalize
    // Suppress finalization.
    GC.SuppressFinalize(this);
#pragma warning restore CA1816 // Dispose methods should call SuppressFinalize
}
```

C# version 8.0 - Asynchronous disposable

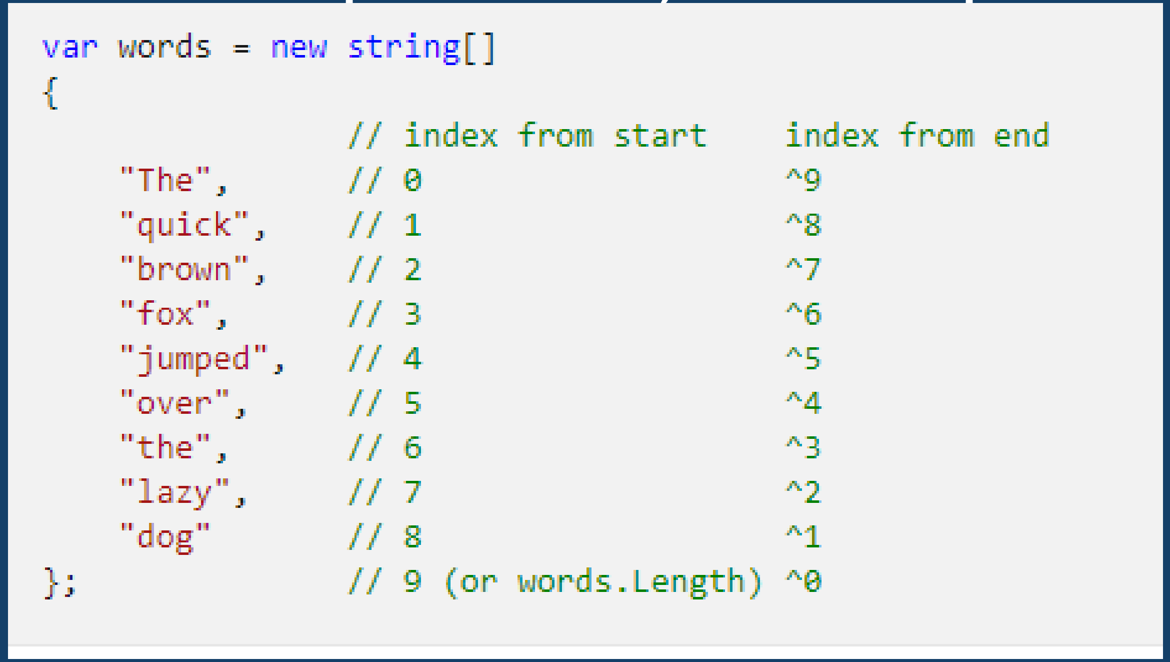
- **The DisposeAsyncCore() method:** to perform the asynchronous cleanup of managed resources or for cascading calls to DisposeAsync().
  - Encapsulates the common asynchronous cleanup operations when a subclass inherits a base class that is an implementation of IAsyncDisposable.
  - The DisposeAsyncCore() method is virtual so that derived classes can define additional cleanup in their overrides.



```
public class ExampleAsyncDisposable : IAsyncDisposable, IDisposable
{
    private Utf8JsonWriter _jsonWriter = new(new MemoryStream());

    public void Dispose()
    {
        Dispose(disposing: true);
        GC.SuppressFinalize(this);
    }

    public async ValueTask DisposeAsync()
    {
        await DisposeAsyncCore();

        Dispose(disposing: false);
#pragma warning disable CA1816 // Dispose methods should call SuppressFinalize
        GC.SuppressFinalize(this);
#pragma warning restore CA1816 // Dispose methods should call SuppressFinalize
    }

    protected virtual async ValueTask DisposeAsyncCore()
    {
        if (_jsonWriter is not null)
        {
            await _jsonWriter.DisposeAsync().ConfigureAwait(false);
        }

        _jsonWriter = null;
    }
}
```

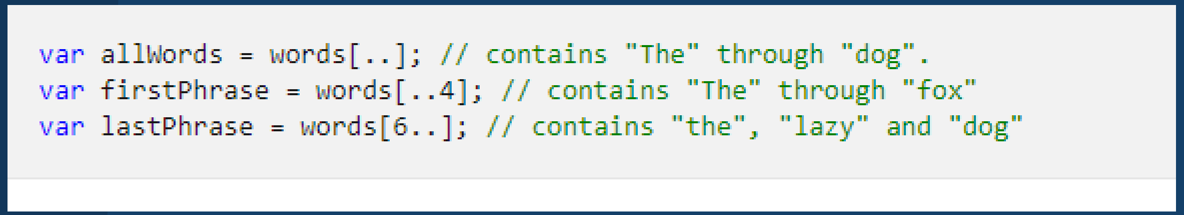
C# version 8.0 - Indices and ranges

- **Indices and ranges** provide a succinct syntax for accessing single elements or ranges in a sequence.
  - System.Index represents an index into a sequence.
  - The index from end operator ^, which specifies that an index is relative to the end of the sequence.

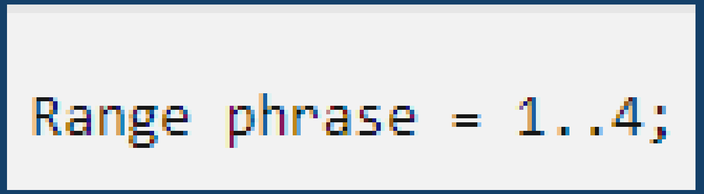


```
var words = new string[]
{
                // index from start    index from end
    "The",      // 0                   ^9
    "quick",    // 1                   ^8
    "brown",    // 2                   ^7
    "fox",      // 3                   ^6
    "jumped",   // 4                   ^5
    "over",     // 5                   ^4
    "the",      // 6                   ^3
    "lazy",     // 7                   ^2
    "dog"       // 8                   ^1
};              // 9 (or words.Length) ^0
```

  - System.Range represents a sub range of a sequence.
  - The range operator .., which specifies the start and end of a range as its operands.



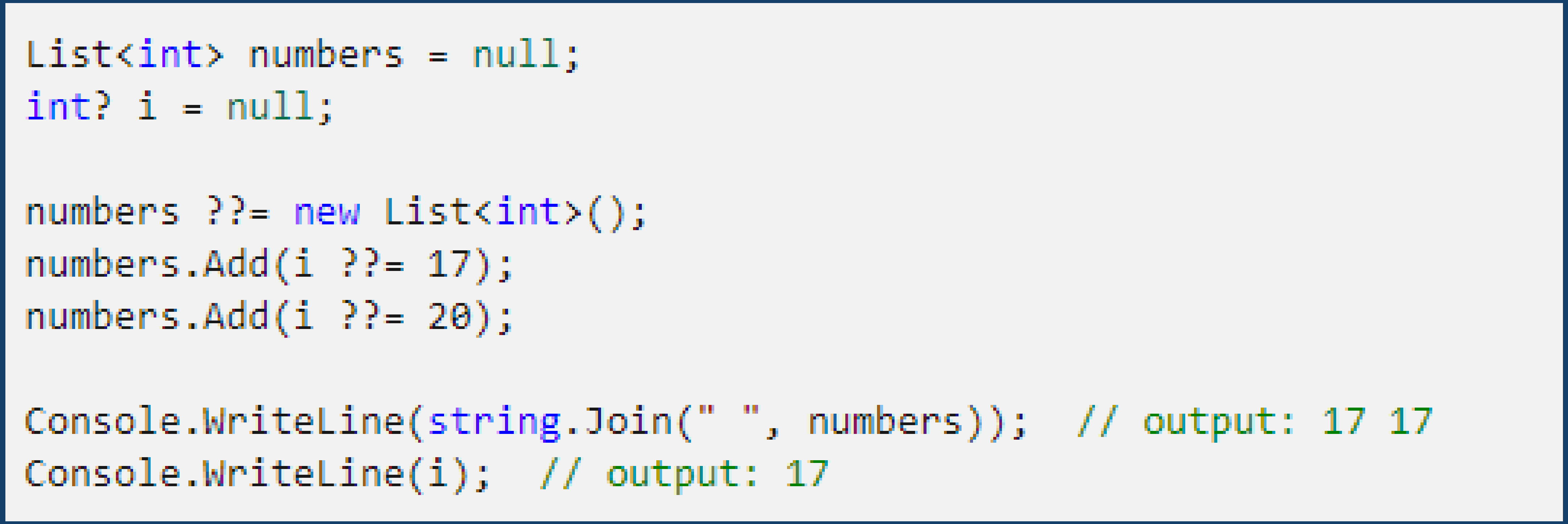
```
var allWords = words[..]; // contains "The" through "dog".
var firstPhrase = words[..4]; // contains "The" through "fox"
var lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
```



```
Range phrase = 1..4;
```

C# version 8.0 - Null-coalescing assignment

You can use the **??= operator** to assign the value of its right-hand operand to its left-hand operand only if the left-hand operand evaluates to null.



```
List<int> numbers = null;
int? i = null;

numbers ??= new List<int>();
numbers.Add(i ??= 17);
numbers.Add(i ??= 20);

Console.WriteLine(string.Join(" ", numbers));  // output: 17 17
Console.WriteLine(i);  // output: 17
```

C# version 8.0 - Unmanaged constructed types

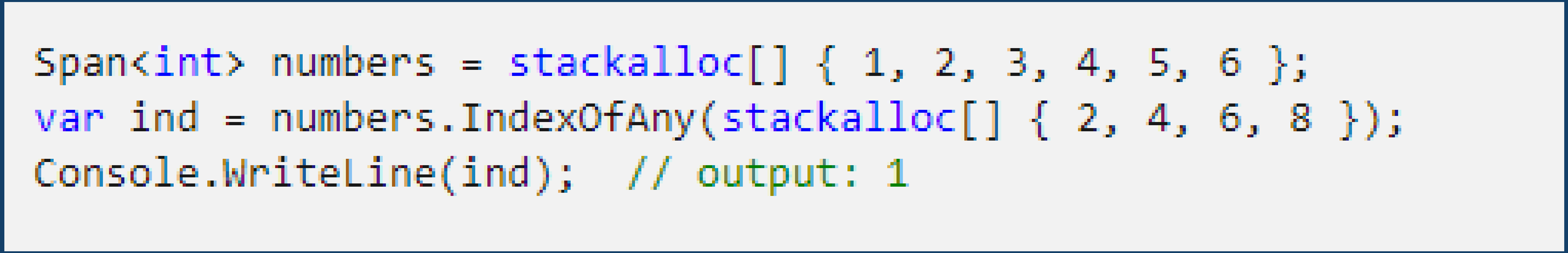
A constructed value type is unmanaged if it contains fields of unmanaged types only.

```
public struct Coords<T>
{
    public T X;
    public T Y;
}
```

```
Span<Coords<int>> coordinates = stackalloc[]
{
    new Coords<int> { X = 0, Y = 0 },
    new Coords<int> { X = 0, Y = 3 },
    new Coords<int> { X = 4, Y = 0 }
};
```

C# version 8.0 - Enhancement of interpolated verbatim strings

If the result of a stackalloc expression is of the System.Span<T> or System.ReadOnlySpan<T> type, you can use the stackalloc expression in other expressions:



```
Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
Console.WriteLine(ind);  // output: 1
```

C# version 8.0 - Stackalloc in nested expressions

- Order of the $ and @ tokens in interpolated verbatim strings can be any: both $@"..." and @$"..." are valid interpolated verbatim strings.
- In earlier C# versions, the $ token must appear before the @ token.

C# version 9.0

- Records
- Init only setters
- Top-level statements
- Pattern matching enhancements
- Performance and interop
  - Native sized integers
  - Function pointers
  - Suppress emitting localsinit flag
- Fit and finish features
  - Target-typed new expressions
  - static anonymous functions
  - Target-typed conditional expressions
  - Covariant return types
  - Extension GetEnumerator support for foreach loops
  - Lambda discard parameters
  - Attributes on local functions
- Support for code generators
  - Module initializers
  - New features for partial methods

C# version 10

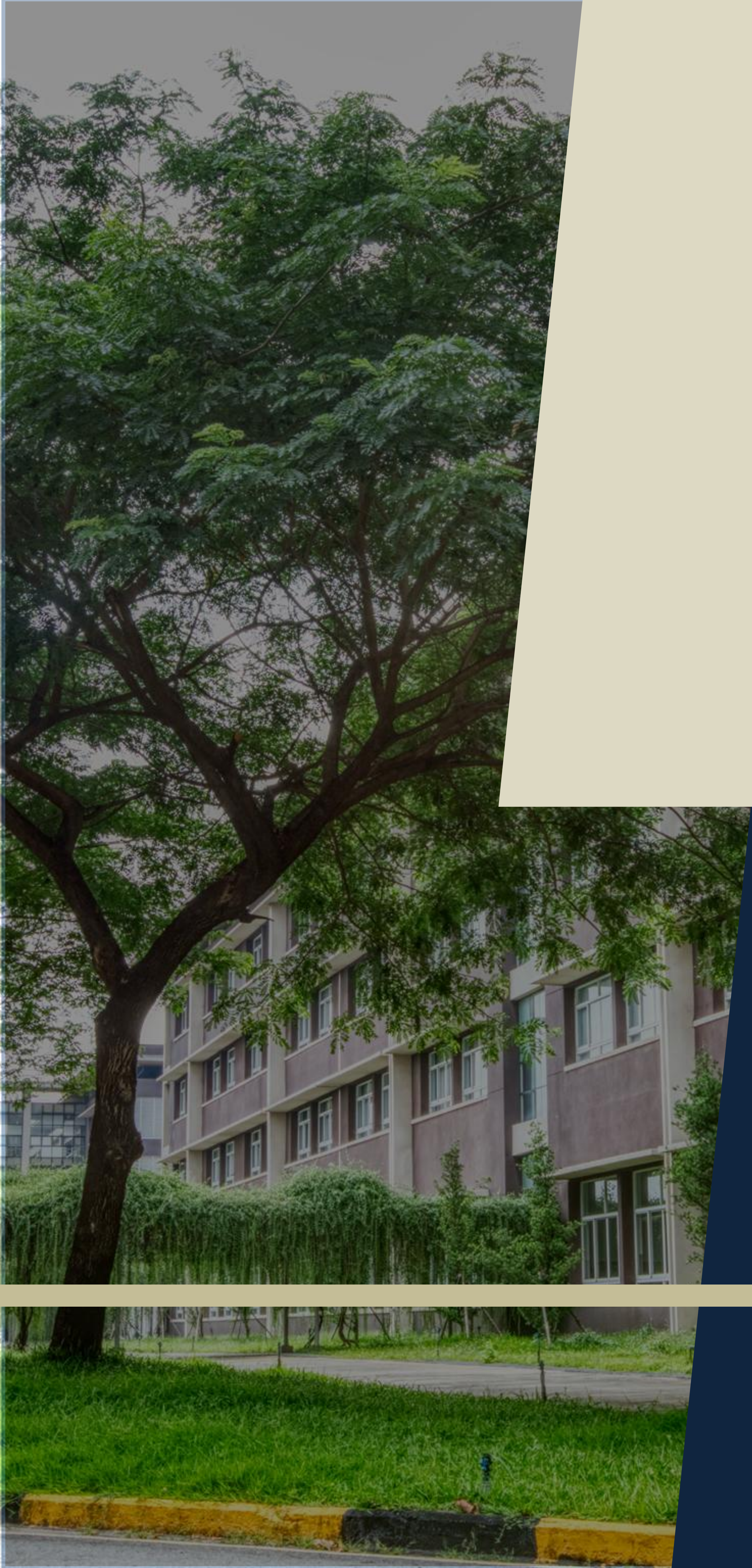
- Record structs
- Improvements of structure types
- Interpolated string handlers
- global using directives
- File-scoped namespace declaration
- Extended property patterns
- Improvements on lambda expressions
- Allow const interpolated strings
- Record types can seal ToString()
- Improved definite assignment
- Allow both assignment and declaration in the same deconstruction
- Allow AsyncMethodBuilder attribute on methods
- CallerArgumentExpression attribute
- Enhanced #line pragma

C# version 11

- Raw string literals
- Generic math support
- Generic attributes
- UTF-8 string literals
- Newlines in string interpolation expressions
- List patterns
- File-local types
- Required members
- Auto-default structs
- Pattern match Span<char> on a constant string
- Extended nameof scope
- Numeric IntPtr
- ref fields and scoped ref
- Improved method group conversion to delegate
- Warning wave 7

C# version 12

- Primary constructors - You can create primary constructors in any class or struct type.
- Collection expressions - A new syntax to specify collection expressions, including the spread element, (..e), to expand any collection.
- Inline arrays - Inline arrays enable you to create an array of fixed size in a struct type.
- Optional parameters in lambda expressions - You can define default values for parameters on lambda expressions.
- ref readonly parameters - ref readonly parameters enables more clarity for APIs that might be using ref parameters or in parameters.
- Alias any type - You can use the using alias directive to alias any type, not just named types.
- Experimental attribute - Indicate an experimental feature.



TRƯỜNG ĐẠI HỌC QUỐC TẾ MIỀN ĐÔNG
EASTERN INTERNATIONAL UNIVERSITY

Start your future at EIU

Thank You