[Asynchronous programming is a technique that lets you offload long running operations to background threads so that the main thread of your application remains available to respond to new input from your user.](https://www.linkedin.com/learning/asynchronous-programming-in-c-sharp/what-is-asynchronous-programming?contextUrn=urn%3Ali%3AlyndaLearningPath%3A6706e94e498e180fcd4c8b19&focused=false&leis=AICC&resume=false&u=2965546)

[When you write asynchronous code, you're able to send those long-running tasks to separate threads. Since the main user interface thread isn't busy performing that work, it can always be available to respond to new user events. When the worker threads are done, they can report their changes back to the main thread so the interface can be updated with their results](https://www.linkedin.com/learning/asynchronous-programming-in-c-sharp/what-is-asynchronous-programming?contextUrn=urn%3Ali%3AlyndaLearningPath%3A6706e94e498e180fcd4c8b19&focused=false&leis=AICC&resume=false&u=2965546)

Event Based Async Pattern (EAP):

* EventHandler Delegate
* EventArg Derived Types
* ReadAsync,ReadCompleted, ReadAsyncCompleted

Asynchrounous Programming Model (APM):

* Uses IASyncResult interface
* Async operations require Begin and End Methods.

Task Based Async Pattern

* Task
* Task<T>
* Async/Await

Var processData = Task.Run(ProcessData);

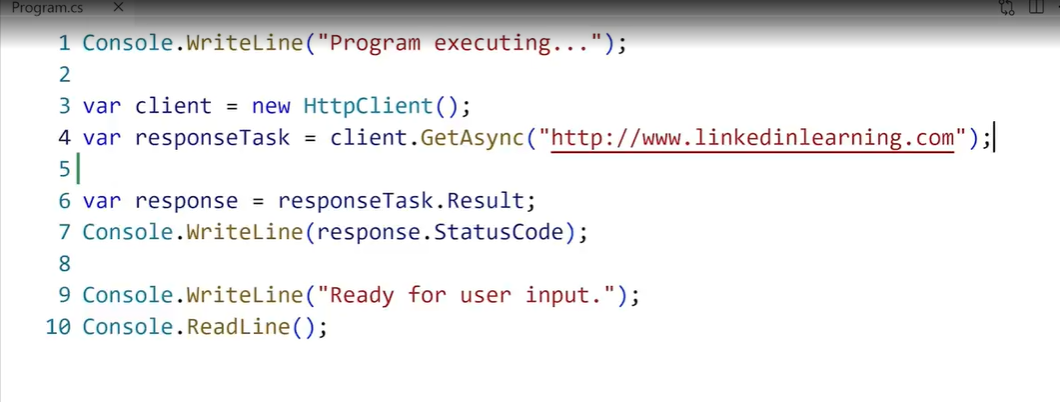
Task<string> dataTask = Task.Run(ProcessData);

Console.WriteLine(dataTask.Result);

Continuations:

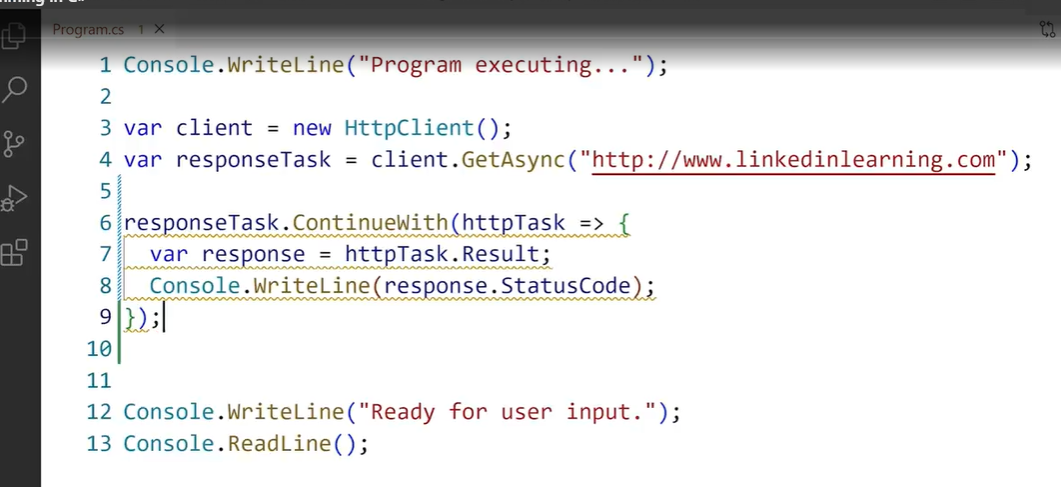
1. Tasks started by another task.
2. These tasks are started when previous task completes.

Blocking Code



Var response is waiting till the responseTask is resolved and then fetch it’s result property. Thus making it a blocking code.

Non Blocking Version:





Async Await Example



The ability to implement cancellation of a task is optional but included with most asynchronous methods. Cancellation is managed by the cancellation token source class. It provides a cancellation token via its token property that can be passed as a parameter to asynchronous methods that support cancellation.

To put it simply, managed code is just that: code whose execution is managed by a runtime. In this case, the runtime in question is called the **Common Language Runtime** or CLR, regardless of the implementation.

The managed execution process includes the following steps, which are discussed in detail later in this topic:

1. Choosing a compiler. To obtain the benefits provided by the common language runtime, you must use one or more language compilers that target the runtime.
2. Compiling your code to intermediate language. Compiling translates your source code into common intermediate language (CIL) and generates the required metadata.
3. Compiling CIL to native code. At execution time, a just-in-time (JIT) compiler translates the CIL into native code. During this compilation, code must pass a verification process that examines the CIL and metadata to find out whether the code can be determined to be type safe.
4. Running code. The common language runtime provides the infrastructure that enables execution to take place and services that can be used during execution.

Of course, the CLR allows passing the boundaries between managed and unmanaged world, and there's a lot of code that does that, even in the [.NET class libraries](https://learn.microsoft.com/en-us/dotnet/standard/class-library-overview). This is called *interoperability*, or *interop* for short.

Automatic memory management is one of the services that the Common Language Runtime provides during [Managed Execution](https://learn.microsoft.com/en-us/dotnet/standard/managed-execution-process). The Common Language Runtime's garbage collector manages the allocation and release of memory for an application. For developers, this means that you do not have to write code to perform memory management tasks when you develop managed applications. Automatic memory management can eliminate common problems, such as forgetting to free an object and causing a memory leak, or attempting to access memory for an object that has already been freed.

**Allocating Memory**

When you initialize a new process, the runtime reserves a contiguous region of address space for the process. This reserved address space is called the managed heap. The managed heap maintains a pointer to the address where the next object in the heap will be allocated. Initially, this pointer is set to the managed heap's base address. All [reference types](https://learn.microsoft.com/en-us/dotnet/standard/base-types/common-type-system) are allocated on the managed heap.

**Releasing Memory**

The garbage collector's optimizing engine determines the best time to perform a collection based on the allocations being made. When the garbage collector performs a collection, it releases the memory for objects that are no longer being used by the application. It determines which objects are no longer being used by examining the application's roots. Every application has a set of roots. Each root either refers to an object on the managed heap or is set to null. An application's roots include static fields, local variables and parameters on a thread's stack, and CPU registers.

**Generations and Performance**

To optimize the performance of the garbage collector, the managed heap is divided into three generations: 0, 1, and 2. The runtime's garbage collection algorithm is based on several generalizations that the computer software industry has discovered to be true by experimenting with garbage collection schemes. First, it is faster to compact the memory for a portion of the managed heap than for the entire managed heap. Secondly, newer objects will have shorter lifetimes and older objects will have longer lifetimes. Lastly, newer objects tend to be related to each other and accessed by the application around the same time.

The runtime's garbage collector stores new objects in generation 0. Objects created early in the application's lifetime that survive collections are promoted and stored in generations 1 and 2.

In reality, the garbage collector performs a collection when generation 0 is full. If an application attempts to create a new object when generation 0 is full, the garbage collector discovers that there is no address space remaining in generation 0 to allocate for the object. The garbage collector performs a collection in an attempt to free address space in generation 0 for the object. The garbage collector starts by examining the objects in generation 0 rather than all objects in the managed heap. This is the most efficient approach, because new objects tend to have short lifetimes, and it is expected that many of the objects in generation 0 will no longer be in use by the application when a collection is performed.

After the garbage collector performs a collection of generation 0, it compacts the memory for the reachable objects as explained in [Releasing Memory](https://learn.microsoft.com/en-us/dotnet/standard/automatic-memory-management#cpconautomaticmemorymanagementreleasingmemoryanchor1) earlier in this topic. The garbage collector then promotes these objects and considers this portion of the managed heap generation 1. Because objects that survive collections tend to have longer lifetimes, it makes sense to promote them to a higher generation. As a result, the garbage collector does not have to reexamine the objects in generations 1 and 2 each time it performs a collection of generation 0.

After the garbage collector performs its first collection of generation 0 and promotes the reachable objects to generation 1, it considers the remainder of the managed heap generation 0. It continues to allocate memory for new objects in generation 0 until generation 0 is full and it is necessary to perform another collection. At this point, the garbage collector's optimizing engine determines whether it is necessary to examine the objects in older generations. For example, if a collection of generation 0 does not reclaim enough memory for the application to successfully complete its attempt to create a new object, the garbage collector can perform a collection of generation 1, then generation 2. If this does not reclaim enough memory, the garbage collector can perform a collection of generations 2, 1, and 0. After each collection, the garbage collector compacts the reachable objects in generation 0 and promotes them to generation 1. Objects in generation 1 that survive collections are promoted to generation 2. Because the garbage collector supports only three generations, objects in generation 2 that survive a collection remain in generation 2 until they are determined to be unreachable in a future collection.

**Releasing Memory for Unmanaged Resources**

For the majority of the objects that your application creates, you can rely on the garbage collector to automatically perform the necessary memory management tasks. However, unmanaged resources require explicit cleanup. The most common type of unmanaged resource is an object that wraps an operating system resource, such as a file handle, window handle, or network connection. Although the garbage collector is able to track the lifetime of a managed object that encapsulates an unmanaged resource, it does not have specific knowledge about how to clean up the resource. When you create an object that encapsulates an unmanaged resource, it is recommended that you provide the necessary code to clean up the unmanaged resource in a public **Dispose** method. By providing a **Dispose** method, you enable users of your object to explicitly free its memory when they are finished with the object. When you use an object that encapsulates an unmanaged resource, you should be aware of **Dispose** and call it as necessary

[.Net Garbage Collection and Finalization Queue](https://www.scholarhat.com/tutorial/net/net-garbage-collection-and-finalization-queue)

The [Dispose](https://learn.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) method is primarily implemented to release unmanaged resources. When working with instance members that are [IDisposable](https://learn.microsoft.com/en-us/dotnet/api/system.idisposable) implementations, it's common to cascade [Dispose](https://learn.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) calls. There are other reasons for implementing [Dispose](https://learn.microsoft.com/en-us/dotnet/api/system.idisposable.dispose), for example, to free memory that was allocated, remove an item that was added to a collection, or signal the release of a lock that was acquired.

**The Dispose() method**

Because the public, non-virtual , parameterless Dispose method is called when it's no longer needed (by a consumer of the type), its purpose is to free unmanaged resources, perform general cleanup, and to indicate that the finalizer, if one is present, doesn't have to run. Freeing the actual memory associated with a managed object is always the domain of the [garbage collector](https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/). Because of this, it has a standard implementation:

C#Copy

public void Dispose()

{

// Dispose of unmanaged resources.

Dispose(true);

// Suppress finalization.

GC.SuppressFinalize(this);

}

**The Dispose(bool) method overload**

In the overload, the disposing parameter is a [Boolean](https://learn.microsoft.com/en-us/dotnet/api/system.boolean) that indicates whether the method call comes from a [Dispose](https://learn.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) method (its value is true) or from a finalizer (its value is false).

In C#, both Dispose and Finalize are used for resource management, but they serve different purposes and are used in different scenarios:

**Dispose Method**

* **Purpose**: The Dispose method is part of the IDisposable interface and is used to release both managed and unmanaged resources deterministically.
* **Control**: You call Dispose explicitly in your code, usually when you are done using an object. This allows for the immediate freeing of resources.
* **Deterministic**: Dispose provides a deterministic way to release resources, meaning you know exactly when the resources are released.
* [**Pattern**: When implementing Dispose, it’s common to follow the dispose pattern, which includes a finalizer call (GC.SuppressFinalize(this)) to prevent the garbage collector from calling Finalize if Dispose has already been called1](https://stackoverflow.com/questions/732864/finalize-vs-dispose)[2](https://dotnettutorials.net/lesson/differences-between-finalize-and-dispose-in-csharp/).

**Finalize Method**

* **Purpose**: The Finalize method is used for cleanup operations before an object is garbage collected. It’s typically overridden to release unmanaged resources that the object holds.
* **Control**: You do not call Finalize directly. It’s invoked by the garbage collector.
* **Non-deterministic**: The exact time when Finalize is called is non-deterministic, depending on the garbage collector’s schedule.
* **Inheritance**: The Finalize method is inherited from the Object class. [It should always call the Finalize method of its base class if overridden to ensure that all resources are released properly1](https://stackoverflow.com/questions/732864/finalize-vs-dispose)[2](https://dotnettutorials.net/lesson/differences-between-finalize-and-dispose-in-csharp/).

**Key Differences**

* **Timing**: Dispose is called explicitly at a known point in the program, while Finalize is called by the garbage collector in a non-deterministic manner.
* [**Usage**: Dispose is used for both managed and unmanaged resources, whereas Finalize is primarily used for unmanaged resources](https://stackoverflow.com/questions/732864/finalize-vs-dispose)[2](https://dotnettutorials.net/lesson/differences-between-finalize-and-dispose-in-csharp/)[3](https://codetosolutions.com/blog/36/difference-between-finally,-finalize-and-dispose-in-c).

Here’s a simple example to illustrate the use of both methods:

**C#**

public class ResourceHolder : IDisposable

{

private bool \_disposed = false;

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

protected virtual void Dispose(bool disposing)

{

if (!\_disposed)

{

if (disposing)

{

// Free other managed objects

}

// Free unmanaged resources

\_disposed = true;

}

}

~ResourceHolder()

{

Dispose(false);

}

}

AI-generated code. Review and use carefully. [More info on FAQ](https://www.bing.com/new#faq).

In this example, the Dispose method is called explicitly to release resources, while the finalizer (~ResourceHolder) ensures that resources are released if Dispose is not called.

Reflection in C# is a powerful feature that allows you to inspect and interact with the metadata of assemblies, modules, and types at runtime. This can be useful for various tasks such as dynamically creating instances of types, invoking methods, and accessing fields and properties.

**Key Concepts of Reflection**

1. **Assemblies and Modules**: Reflection can be used to load assemblies and modules, and to discover the types defined within them.
2. **Types**: You can obtain information about classes, interfaces, structures, and enumerations.
3. **Members**: Reflection allows you to inspect members of a type, such as constructors, methods, fields, properties, and events.

**Common Uses of Reflection**

* **Loading Assemblies**: You can load an assembly and explore its types.
* **Creating Instances**: Dynamically create instances of types.
* **Invoking Methods**: Invoke methods on objects dynamically.
* **Accessing Fields and Properties**: Get or set the values of fields and properties

In C#, a Func is a delegate type that represents a method that returns a value. It can have one or more parameters, but it always returns a value (it can't return void). The Func delegate can be used to represent methods with any return type, making it very versatile.

In C#, the Action delegate represents a method that takes parameters but does **not** return a value (i.e., the return type is void). It's a versatile delegate type often used for methods that perform operations or side effects but don’t need to produce a result.

In C#, the Predicate<T> delegate represents a method that takes a parameter of type T and returns a bool value. This delegate is useful when you need to evaluate a condition and determine if the given parameter satisfies that condition.

In C#, a **converter delegate** typically refers to a delegate that transforms or converts an object of one type into an object of another type. A common type for such a delegate is Func<TInput, TOutput>, which represents a method that takes an input of type TInput and returns an output of type TOutput.

This can be used for tasks like mapping, transformation, or conversion between different types. In many scenarios, you’ll use a **converter delegate** when you need to apply a transformation function to elements in a collection, or when you need to convert between types

In C#, a **comparison delegate** typically refers to a delegate that defines a method to compare two objects. This is useful in sorting algorithms, finding the order of objects, or filtering collections based on some criteria. A common delegate used for comparison is Comparison<T>, which is built into the .NET framework.

In C#, an **Expression Tree** is a data structure that represents code in a tree-like format, where each node of the tree represents a part of a C# expression (such as a method call, a variable, or a literal). Expression trees allow you to work with code as data, enabling you to dynamically build and execute code, or manipulate it before execution. This is particularly useful in scenarios like LINQ providers, dynamic querying, and compiling code at runtime.

**Key Concepts of Expression Trees:**

1. **Expression**: In C#, System.Linq.Expressions.Expression is the base class for expression trees. It represents an abstract syntax tree (AST) for code.
2. **Expression Types**: You can build various types of expressions such as:
   * **BinaryExpression**: Represents binary operations like addition or subtraction.
   * **ConstantExpression**: Represents a constant value.
   * **ParameterExpression**: Represents a parameter in an expression.
   * **MethodCallExpression**: Represents a method call.
   * **LambdaExpression**: Represents a lambda expression.
3. **Execution of Expression Trees**: Expression trees can be compiled into executable code and executed via Lambda.Compile or similar methods.

**Why Use Expression Trees?**

* **Dynamic Queries**: For example, in LINQ-to-SQL or Entity Framework, expression trees are used to represent the underlying SQL queries dynamically.
* **Code Generation**: Expression trees allow you to generate and compile C# code dynamically, which can then be executed.
* **Reflection**: You can inspect and manipulate code structure, such as creating dynamic methods or analyzing existing code.