**Three use cases we need asynchronous programming**

Asynchronous programming in .NET is crucial for enhancing application performance and responsiveness. Here are three common use cases where asynchronous programming is beneficial:

1. I/O Bound Operations

Applications frequently perform I/O operations such as file reading/writing, database queries, and API calls. Using asynchronous programming allows these operations to run without blocking the main thread, improving application responsiveness.

Example:

public async Task<string> ReadFileAsync(string filePath)

{

using StreamReader reader = new StreamReader(filePath);

return await reader.ReadToEndAsync();

}

2. Web Applications Handling Multiple Requests

In ASP.NET Core applications, handling multiple concurrent requests efficiently is essential. Using asynchronous controllers ensures the application can serve more users with minimal thread usage.

Example:

public async Task<IActionResult> GetData()

{

var data = await FetchDataFromDatabaseAsync();

return Ok(data);

}

3. Parallel Processing for Computational Tasks

When handling CPU-intensive operations, parallel execution can speed up computations. Asynchronous tasks can be used with Task.Run to leverage multiple cores.

Example:

public async Task<int> ComputeSumAsync(List<int> numbers)

{

return await Task.Run(() => numbers.Sum());

}

**What is the difference between thread and task .**

**1. Thread**

* A Thread represents an independent execution unit.
* Threads are managed by the OS and are expensive to create.
* Suitable for long-running background processes.

Example:

Thread thread = new Thread(() => Console.WriteLine("Running in a thread"));

thread.Start();

**2. Task**

* A Task represents an asynchronous operation that runs on a thread pool.
* More efficient than manually managing threads.
* Used in modern async programming with async and await.

Example:

Task task = Task.Run(() => Console.WriteLine("Running in a task"));

await task;

**Key Differences**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Thread** | **Task** |
| Creation Cost | High | Low |
| Management | Manual | Automatic |
| Execution | Dedicated Thread | Thread Pool |
| Async Support | No | Yes |

**What is the new features in .net 9 to make operations on Generic type**

.NET 9 introduces several enhancements to working with generic types. Some of the key improvements include:

**1. Primary Constructors for Generic Types**

Allows defining constructors directly in generic types.

public class DataStore<T>(T data)

{

public T Data { get; } = data;

}

**2. Improved Generic Math Operations**

Extends support for mathematical operations on generic types.

T Add<T>(T a, T b) where T : INumber<T>

{

return a + b;

}

**3. Enhanced Type Inference for Generics**

Improves automatic type inference, reducing redundant type declarations.

var numbers = new List<int> { 1, 2, 3 };

var sum = numbers.Aggregate((a, b) => a + b);

**What is delegate built\_in types in details**

The built-in delegate types simplify development by providing predefined delegate signatures.

**1. Action**

Represents a method that takes parameters but returns void.

Action<string> printMessage = message => Console.WriteLine(message);

printMessage("Hello, World!");

**2. Func<T, TResult>**

Represents a method that takes parameters and returns a value.

Func<int, int, int> add = (a, b) => a + b;

Console.WriteLine(add(3, 4)); // Output: 7

**3. Predicate**

Represents a method that takes a parameter and returns a boolean.

Predicate<int> isEven = number => number % 2 == 0;

Console.WriteLine(isEven(10)); // Output: True

These built-in delegates reduce the need for custom delegate declarations, making code cleaner and more maintainable.