ASYNCHRONOUS TYPES

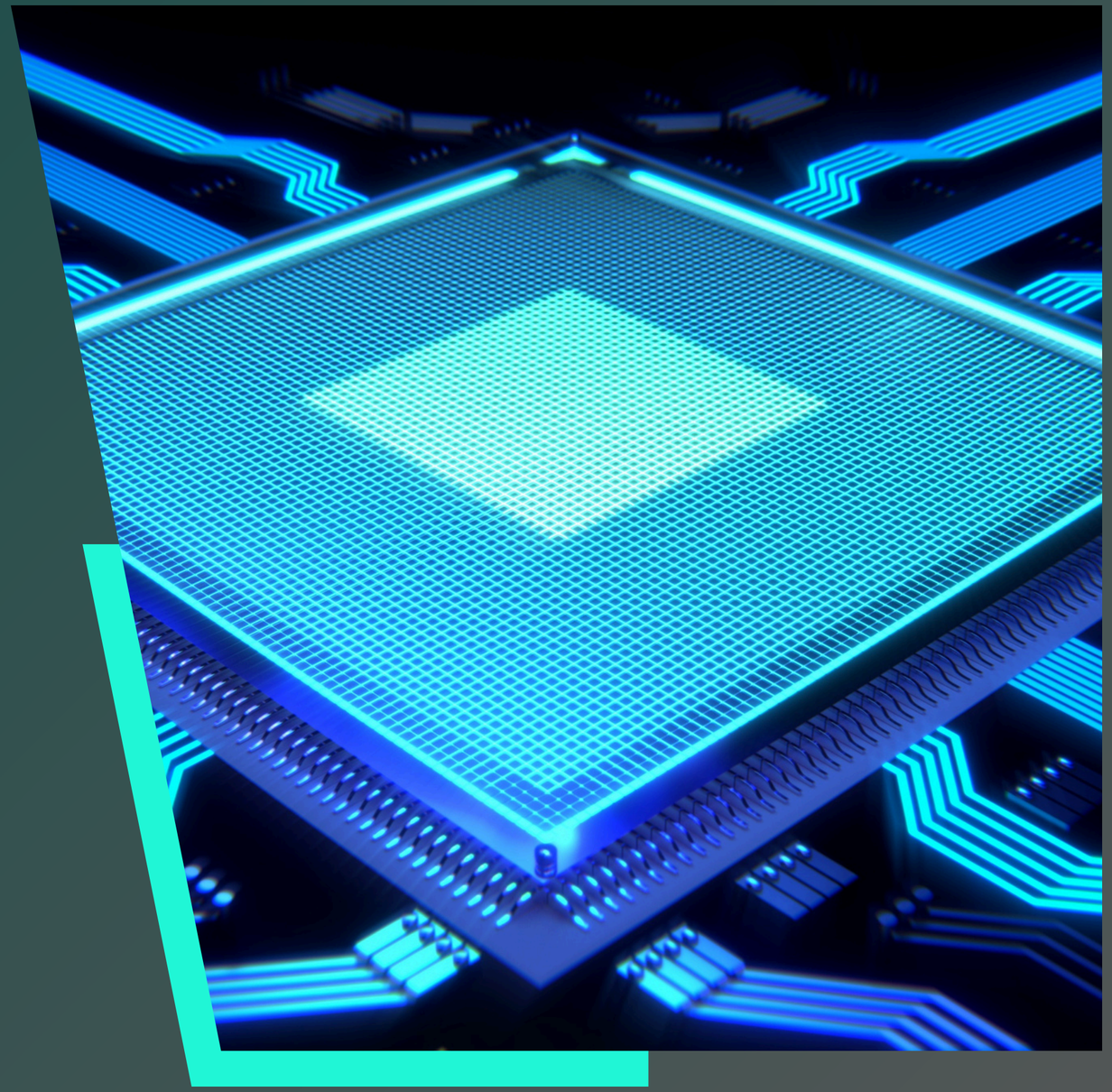
PROG6221

WHAT IS IT?

Asynchronous programming is designed to improve the efficiency and responsiveness of applications, particularly when dealing with operations that take time to complete, such as I/O-bound tasks (e.g., reading files, making web requests, or interacting with databases).

Instead of blocking the main thread, asynchronous programming allows the application to continue running other tasks while waiting for the operation to finish.

This leads to a smoother user experience, especially in UI applications where freezing or lagging is undesirable.



EXAMPLE:

This example demonstrates how to perform an asynchronous operation using Task.Delay to simulate a long-running operation.

```
1   using System;
2   using System.Threading.Tasks;
3
    0 references
4   class Program
5   {
        0 references
6       static async Task Main(string[] args)
7       {
8           Console.WriteLine("Starting async operation...");
9           string result = await PerformLongRunningOperationAsync();
10          Console.WriteLine($"Operation completed: {result}");
11      }
12
        1 reference
13      static async Task<string> PerformLongRunningOperationAsync()
14      {
15          // Simulate a long-running operation with Task.Delay (e.g., downloading data)
16          await Task.Delay(3000); // Waits for 3 seconds asynchronously
17          return "Operation Successful";
18      }
19  }
```

- Main Method: The Main method is marked as async because it calls an asynchronous method (PerformLongRunningOperationAsync).
- Task.Delay: Inside PerformLongRunningOperationAsync, Task.Delay(3000) is used to simulate a delay of 3 seconds, representing a long-running operation like downloading data.
- Await: The await keyword ensures that the method waits for the Task.Delay to complete before proceeding but does so without blocking the main thread.

```
Microsoft Visual Studio Debug Console
Starting async operation...
Operation completed: Operation Successful
```

PARALLEL PROGRAMMING

Parallel programming is a technique used to perform multiple operations simultaneously.

It is particularly beneficial for CPU-bound tasks that can be divided into independent operations, such as data processing, mathematical computations, or any task that can be split into smaller chunks that can run concurrently.

By running these operations in parallel, you can significantly reduce the total execution time, especially on systems with multi-core processors.

EXAMPLE:

This example demonstrates how to use Parallel.For to perform a simple computation in parallel.

- Main Method: The Main method calls ParallelSum and prints the result.
- Parallel.For: The loop calculates the sum of all elements in the array. The Parallel.For loop allows the summing operation to be split across multiple threads.
- Interlocked.Add: Safely adds each thread's local sum to the totalSum variable.

```
using System;
using System.Linq;
using System.Threading.Tasks;

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("Starting parallel operation...");
        long totalSum = ParallelSum();
        Console.WriteLine($"Total sum: {totalSum}");
    }

    1 reference
    static long ParallelSum()
    {
        int[] numbers = Enumerable.Range(1, 1000000).ToArray(); // Create an array of numbers
        long totalSum = 0;

        // Parallel.For splits the work across multiple threads
        Parallel.For(0, numbers.Length, () => 0L, (i, loopState, localSum) =>
        {
            localSum += numbers[i]; // Calculate local sum
            return localSum;
        },
        localSum => Interlocked.Add(ref totalSum, localSum)); // Safely add local sum to totalSum

        return totalSum;
    }
}
```

Microsoft Visual Studio Debug Console

```
Starting parallel operation...
Total sum: 500000500000
```

MULTI-THREADING

Multithreading involves creating and managing multiple threads within a single application.

Each thread runs independently but shares the same memory space, allowing for parallel execution of tasks.

Multithreading is particularly useful for tasks that can be performed independently but may require sharing data between threads. However, it introduces complexity in managing the synchronization between threads to avoid issues like race conditions, deadlocks, and thread contention.

EXAMPLE:

This example simulates a simple bank account where multiple threads attempt to deposit and withdraw money. This will illustrate thread synchronization and the potential issues that can arise if threads are not properly managed.

```
35  class BankAccount
36  {
37      private object balanceLock = new object(); // Lock object for thread synchronization
38      private decimal balance;
39
        1 reference
40      public BankAccount(decimal initialBalance)
41      {
42          balance = initialBalance;
43      }
44
        2 references
45      public decimal Balance
46      {
47          get
48          {
49              lock (balanceLock)
50              {
51                  return balance;
52              }
53          }
54      }
```

BANK ACCOUNT CLASS:

Purpose:

- The BankAccount class represents a simple bank account with methods to deposit and withdraw money. It ensures thread safety when multiple threads access or modify the account balance.

Key Parts:

- balanceLock: This is an object used to synchronize access to the balance field, preventing race conditions where multiple threads might try to change the balance at the same time.
- Deposit and Withdraw Methods: These methods use the lock statement to ensure that only one thread can execute the method at a time, preventing errors like withdrawing more money than is available.

EXAMPLE:

This example simulates a simple bank account where multiple threads attempt to deposit and withdraw money. This will illustrate thread synchronization and the potential issues that can arise if threads are not properly managed.

PERFROMTRANSACTIONS METHOD:

Purpose:

- This method simulates a series of transactions (deposits and withdrawals) on the bank account. Each thread will run this method, performing the same operations but independently of the others.

Key Points:

- Thread Name: The method takes a thread name as an argument, which helps identify which thread is performing the transactions in the console output.
- Simulated Work: A small delay (Thread.Sleep) simulates the time it might take to perform these operations in a real-world scenario.

```
2 references
23  static void PerformTransactions(BankAccount account, string threadName)
24  {
25      for (int i = 0; i < 10; i++)
26      {
27          account.Deposit(100); // Deposit 100
28          account.Withdraw(50); // Withdraw 50
29          Console.WriteLine($"{threadName}: Balance after transactions: {account.Balance}");
30          Thread.Sleep(100); // Simulate some delay
31      }
32  }
33 }
```

EXAMPLE:

This example simulates a simple bank account where multiple threads attempt to deposit and withdraw money. This will illustrate thread synchronization and the potential issues that can arise if threads are not properly managed.

MAIN METHOD:

Purpose:

- The Main method sets up the environment, starts the threads, and waits for them to complete their operations.

Key Parts:

- Creating the Bank Account: The program starts by creating a BankAccount object with an initial balance.
- Starting Threads: Two threads are created, each assigned to perform transactions on the shared bank account.
- Thread Synchronization: The Join method ensures that the main program waits for both threads to complete their work before printing the final account balance.

```
using System;
using System.Threading;

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        BankAccount account = new BankAccount(1000); // Starting balance of 1000

        // Creating threads for deposit and withdrawal
        Thread thread1 = new Thread(() => PerformTransactions(account, "Thread 1"));
        Thread thread2 = new Thread(() => PerformTransactions(account, "Thread 2"));

        thread1.Start();
        thread2.Start();

        thread1.Join(); // Wait for thread1 to finish
        thread2.Join(); // Wait for thread2 to finish

        Console.WriteLine($"Final account balance: {account.Balance}");
    }
```

PROGRAM AS A WHOLE:

Multithreading Demonstration:

- The program shows how multiple threads can operate on the same data simultaneously. Without proper synchronization (using lock), this could lead to incorrect results, such as an incorrect balance.

Thread Safety:

- By using the lock keyword, the program ensures that only one thread can modify the account balance at a time, which is crucial in avoiding errors when multiple threads access shared resources.

COMPARISON: ASYNCHRONOUS PROGRAMMING, PARALLEL PROGRAMMING, AND MULTITHREADING

Feature:	***Asynchronous Programming***	***Parallel Programming***	***Multithreading***
Purpose:	Non-blocking operations, improving responsiveness.	Executing multiple tasks simultaneously for better performance.	Running multiple threads concurrently to perform different tasks.
Typical Use Case:	I/O-bound tasks like file reading or network calls.	CPU-bound tasks like data processing.	Tasks that can be divided into smaller independent operations, or require sharing data.
Example:	Downloading a file while still using the UI.	Summing large arrays in parallel.	Running separate tasks on different threads.
Complexity:	Easier to implement, especially with async/await keywords.	Can be complex, depended on conditions handled.	Requires careful management of threads to avoid issues like deadlocks.
Performance Impact:	Improves UI responsiveness, but not necessarily raw performance.	Significant performance gains for CPU-bound tasks.	Can improve performance but adds complexity.

THANK YOU