Armen Melkumyan • 1st
Technical / Solutions Architect
4mo •

From DotNet C# Technical Interviews: Explain the Difference Between Parallel.ForEach and Parallel.ForEachAsync

What's the Difference?

- Parallel.ForEach:

1) Designed for CPU-bound tasks.

2) Executes iterations in parallel using multiple threads.

3) Blocks the calling thread until all iterations complete.

4) Great for tasks that are synchronous and computationally intensive.

- Parallel.ForEachAsync:

1) Built for asynchronous workflows.

2) Leverages async/await, making it non-blocking.

3) Ideal for IO-bound or mixed operations where some tasks involve asynchronous calls.

Both methods process collections in parallel, but their use cases and internal mechanics are quite different. Let's understand them better with an example.

```
Console.WriteLine("Using Parallel.ForEach:");
ParallelForEachPrimeCheck(2, 50);

Console.WriteLine("\nUsing Parallel.ForEachAsync:");
await ParallelForEachAsyncPrimeCheck(2, 50);

static void ParallelForEachPrimeCheck(int start, int end)
{
 var primes = new ConcurrentBag<int>();

 Parallel.ForEach(Enumerable.Range(start, end - start + 1), number =>
```

```
 {
 if (IsPrime(number))
 primes.Add(number);
 });

 Console.WriteLine($"Primes: {string.Join(", ", primes.OrderBy(x => x))}");
}

static async Task ParallelForEachAsyncPrimeCheck(int start, int end)
{
 var primes = new ConcurrentBag<int>();

 await Parallel.ForEachAsync(Enumerable.Range(start, end - start + 1), async (number, _) =>
 {
 await Task.Delay(10); // Simulating async work
 if (IsPrime(number))
 primes.Add(number);
 });

 Console.WriteLine($"Primes: {string.Join(", ", primes.OrderBy(x => x))}");
}
```

Parallel.ForEach:

- Processes the range of numbers (2 to 50) in parallel using threads.
- Each number is checked for primality using IsPrime.
- Results are stored in a thread-safe collection (ConcurrentBag<int>).
- This method is optimal here because the IsPrime function is CPU-bound and doesn't require asynchronous operations.

Parallel.ForEachAsync:

- Does the same task but uses asynchronous processing for each number.
- Simulates an asynchronous workload (using Task.Delay) to mimic scenarios like IO-bound operations or

API calls.

- Again, results are safely stored in a ConcurrentBag<int>.

Full code you can see in the attached image

#DotNet #CSharp #TechnicalInterviews #ParallelProgramming #Async

```
using System.Collections.Concurrent;

Console.WriteLine("Using Parallel.ForEach:");
ParallelForEachPrimeCheck(2, 50);

Console.WriteLine("\nUsing Parallel.ForEachAsync:");
await ParallelForEachAsyncPrimeCheck(2, 50);

// Synchronous: Parallel.ForEach
static void ParallelForEachPrimeCheck(int start, int end)
{
    var primes = new ConcurrentBag<int>();

    Parallel.ForEach(Enumerable.Range(start, end - start + 1), number =>
    {
        if (IsPrime(number))
            primes.Add(number);
    });

    Console.WriteLine($"Primes: {string.Join(", ", primes.OrderBy(x => x))}");
}

// Asynchronous: Parallel.ForEachAsync
static async Task ParallelForEachAsyncPrimeCheck(int start, int end)
{
    var primes = new ConcurrentBag<int>();

    await Parallel.ForEachAsync(Enumerable.Range(start, end - start + 1), async (number, _) =>
    {
        await Task.Delay(10); // Simulating async work
        if (IsPrime(number))
            primes.Add(number);
    });

    Console.WriteLine($"Primes: {string.Join(", ", primes.OrderBy(x => x))}");
}

// Helper Function to Check if a Number is Prime
static bool IsPrime(int number)
{
    if (number < 2) return false;
    for (int i = 2; i <= Math.Sqrt(number); i++)
    {
        if (number % i == 0) return false;
    }
    return true;
}
```