1. Что такое TPL? Как и для чего используется тип Task?

\*\*TPL (Task Parallel Library)\*\* — это библиотека в .NET, которая упрощает разработку многопоточных и параллельных приложений. Она предоставляет абстракцию в виде задач (`Task`), которые позволяют выполнять код асинхронно без необходимости управлять потоками напрямую.

\*\*Тип `Task`\*\* используется для представления асинхронной операции. Он позволяет:

- Запускать задачи.

- Ожидать их завершения.

- Обрабатывать результаты и исключения.

2. Почему эффект от распараллеливания наблюдается на большом количестве элементов?

Эффект от распараллеливания лучше всего заметен при обработке большого количества элементов, потому что:

- \*\*Нагрузка на систему\*\*: Распараллеливание позволяет распределить вычислительные задачи между несколькими ядрами процессора, что значительно ускоряет выполнение.

- \*\*Снижение времени ожидания\*\*: При большом объёме данных время, затраченное на обработку, становится значительным, и параллельное выполнение помогает уменьшить это время.

3. В чем основные достоинства работы с задачами по сравнению с потоками?

- \*\*Упрощённое управление\*\*: `Task` автоматически управляет потоками в пуле потоков, что упрощает код.

- \*\*Легковесность\*\*: Задачи легче, чем потоки, и требуют меньше ресурсов.

- \*\*Удобство использования\*\*: TPL предоставляет удобные методы для обработки результатов, исключений и продолжений.

### 4. Приведите три способа создания и/или запуска Task.

1. \*\*С помощью конструктора `Task` и метода `Start()`\*\*:

```csharp

Task task1 = new Task(() => { /\* код \*/ });

task1.Start();

```

2. \*\*С помощью `Task.Factory.StartNew()`\*\*:

```csharp

Task task2 = Task.Factory.StartNew(() => { /\* код \*/ });

```

3. \*\*С помощью `Task.Run()`\*\*:

```csharp

Task task3 = Task.Run(() => { /\* код \*/ });

```

### 5. Как и для чего используют методы Wait(), WaitAll() и WaitAny()?

- \*\*`Wait()`\*\*: Ожидает завершения одной задачи.

task.Wait();

```

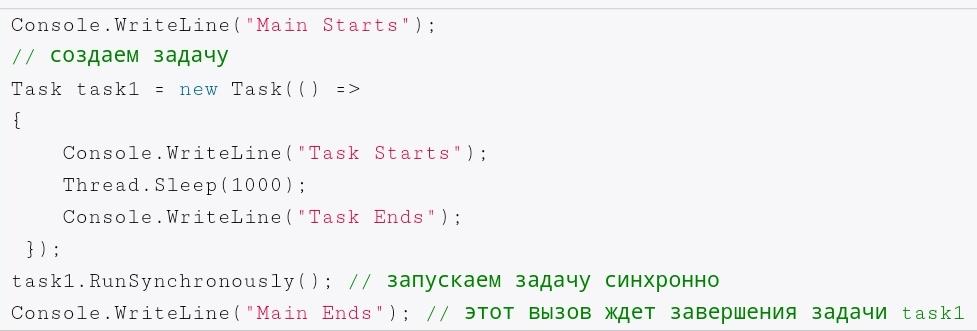
- \*\*`WaitAll()`\*\*: Ожидает завершения всех переданных задач.

Task.WaitAll(task1, task2);

- \*\*`WaitAny()`\*\*: Ожидает завершения любой из переданных задач.

Task.WaitAny(task1, task2);

### 6. Приведите пример синхронного запуска Task.



Синхронный запуск задачи можно реализовать с помощью метода `Wait()`:

Task task = Task.Run(() => { /\* код \*/ });

task.Wait(); // Синхронно ждём завершения задачи

### 7. Как создать задачу с возвратом результата?

Для создания задачи с возвратом результата используйте `Task<T>`:

Task<int> task = Task.Run(() =>

{

return 42; // Возвращаемое значение

});

int result = await task; // Ожидаем результат

```

### 8. Как обработать исключение, если оно произошло при выполнении Task?

Исключения, возникшие в задаче, могут быть обработаны с помощью блока `try-catch`:

try

{

await task;

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

```

### 9. Что такое CancellationToken и как с его помощью отменить выполнение задач?

\*\*`CancellationToken`\*\* — это структура, которая используется для управления отменой задач. Вы можете передать токен в задачу и проверить его состояние для безопасного завершения.

Пример использования:

var cts = new CancellationTokenSource();

CancellationToken token = cts.Token;

Task task = Task.Run(() =>

{

while (!token.IsCancellationRequested)

{

// Выполнение работы

}

}, token);

// Отмена

cts.Cancel();

```

### 10. Как организовать задачу продолжения (continuation task)?

Задача продолжения создаётся с помощью метода `ContinueWith`:

Task task = Task.Run(() => { /\* код \*/ });

task.ContinueWith(t => { /\* код продолжения \*/ });

```

### 11. Как и для чего используется объект ожидания при создании задач продолжения?

Объект ожидания (например, `TaskAwaiter`) позволяет работать с продолжениями задач асинхронно и управлять их выполнением:

var awaiter = task.GetAwaiter();

awaiter.OnCompleted(() => {

// Код, который выполнится после завершения задачи

});

```

### 12. Поясните назначение класса System.Threading.Tasks.Parallel?

Класс `System.Threading.Tasks.Parallel` предоставляет методы для выполнения параллельных операций, таких как `Parallel.For` и `Parallel.ForEach`, которые позволяют эффективно обрабатывать коллекции, распределяя работу между несколькими потоками.

### 13. Приведите пример задачи с Parallel.For(int, int, Action<int>).

Parallel.For(0, 100, i =>

{

// Код для обработки каждого элемента

Console.WriteLine(i);

});

```

### 14. Приведите пример задачи с Parallel.ForEach.

var numbers = new List<int> { 1, 2, 3, 4, 5 };

Parallel.ForEach(numbers, number =>

{

Console.WriteLine(number);

});

```

### 15. Приведите пример с Parallel.Invoke().

Parallel.Invoke(

() => { /\* Код для первой задачи \*/ },

() => { /\* Код для второй задачи \*/ },

() => { /\* Код для третьей задачи \*/ }

);

```

### 16. Как с использованием CancellationToken отменить параллельные операции?

При использовании `CancellationToken` в методах `Parallel` можно передать токен и проверять его состояние в теле задач:

var cts = new CancellationTokenSource();

CancellationToken token = cts.Token;

Parallel.For(0, 100, new ParallelOptions { CancellationToken = token }, i =>

{

if (token.IsCancellationRequested)

return; // Выход из задачи при отмене

// Код для обработки

});

// Отмена

cts.Cancel();

### 17. Для чего используют BlockingCollection<T>, в чем ее особенность?

\*\*`BlockingCollection<T>`\*\* — это потокобезопасная коллекция, которая обеспечивает блокировку при добавлении и удалении элементов. Она используется для реализации паттернов "производитель-потребитель", позволяя безопасно передавать данные между потоками.

### 18. Как используя async и await организовать асинхронное выполнение метода?

Используйте ключевые слова `async` и `await` для организации асинхронного выполнения:

async Task SomeAsyncMethod()

{

await Task.Delay(1000); // Асинхронная операция

Console.WriteLine("Async method completed.");

}