Асинхронное программирование

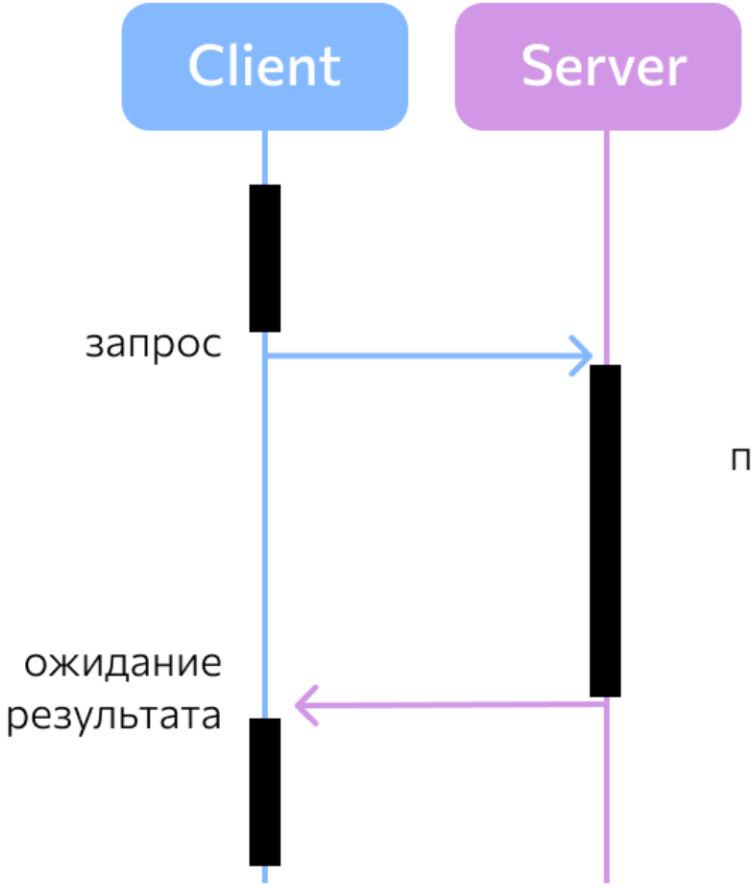
Синхронное программирование

Синхронное программирование - это стиль программирования, при котором операции выполняются последовательно и синхронно. Это означает, что программа ждет завершения одной операции, прежде чем перейти к следующей. Если операция занимает много времени, то она блокирует поток, и программа не может выполнять другие операции до тех пор, пока текущая не завершится.

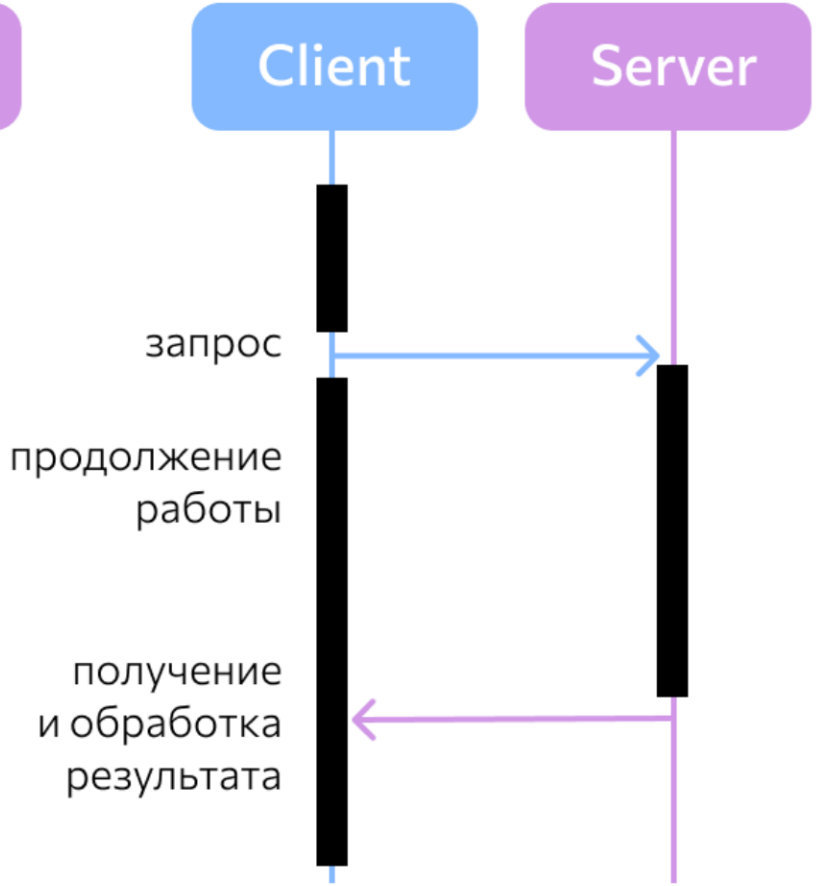
Асинхронное программирование

Асинхронное программирование — концепция программирования, при которой результат выполнения функции доступен спустя некоторое время в виде асинхронного (нарушающего стандартный порядок выполнения) вызова. Запуск длительных операций происходит без ожидания их завершения и не блокирует дальнейшее выполнение программы.

синхронно



асинхронно



Асинхронность

используется для улучшения пользовательского опыта. Например, когда мобильное приложение запрашивает и загружает данные с сервера, экран смартфона демонстрирует активность программы — анимированную заставку или раздел помощи. Основной поток приложения при этом находится в режиме ожидания данных, загружаемых с удаленного сервера.

Асинхронность нужна если:

- Среди множества операций будет та, которая должна работать, когда остальные блокируются.
- Задания выполняют множество операций по вводу/выводу, заставляя синхронную программу впустую тратить время на блокировки.
- Задания в основном независимы друг от друга, и необходимость обмена данными между операциями практически отсутствует.

C#

Асинхронность позволяет вынести отдельные задачи из основного потока в специальные асинхронные методы и при этом более экономно использовать потоки. Асинхронные методы выполняются в отдельных потоках.

В C# асинхронный код строится вокруг ключевых слов async и await.

async

Модификатор метода, который указывает на компилятору что метод является асинхронным. Внутри таких методов можно использовать оператор await.

await

Оператор, который «ожидает» завершение асинхронной операции, не блокируя текущий поток. После завершения операции выполнения продолжается с той же точки, где было приостановлено.

Асинхронный метод

Также стоит отметить, что слово async, которое указывается в определении метода, НЕ делает автоматически метод асинхронным. Оно лишь указывает, что данный метод может содержать одно или несколько выражений await.

```
await PrintAsync();    // вызов асинхронного метода
Console.WriteLine("Некоторые действия в методе Main");
Ссылок: 1
void Print()
{
    Thread.Sleep(3000);      // имитация продолжительной работы
    Console.WriteLine("Куча каких-то данных");
}
Ссылок: 1
async Task PrintAsync()//определение асинхронного метода
{
    Console.WriteLine("Начало метода PrintAsync");//выполняется синхронно
    await Task.Run(Print);//выполняется асинхронно
    Console.WriteLine("Конец метода PrintAsync");
}
```

Пример

```
PrintName("Tom");
PrintName("Bob");
PrintName("Sam");
```

Ссылок: 3

```
void PrintName(string name)
{
    Thread.Sleep(3000);// имитация продолжительной работы
    Console.WriteLine(name);
}
```

Пример

```
await PrintNameAsync("Tom");
await PrintNameAsync("Bob");
await PrintNameAsync("Sam");
Ссылок: 3
async Task PrintNameAsync(string name)//асинхронный метод
{
    await Task.Delay(3000);// имитация продолжительной работы
    Console.WriteLine(name);
}
```

Пример

```
var tomTask = PrintNameAsync("Tom");
var bobTask = PrintNameAsync("Bob");
var samTask = PrintNameAsync("Sam");
await tomTask;
await bobTask;
await samTask;
```

Ссылок: 3

```
async Task PrintNameAsync(string name)//асинхронный метод
{
    await Task.Delay(3000);      // имитация продолжительной работы
    Console.WriteLine(name);
}
```

Возвращаемый тип

В качестве возвращаемого типа используется один из следующих:

- void
- Task
- Task<T>
- ValueTask<T>

Task<T>

```
int n1 = await SquareAsync(5);
int n2 = await SquareAsync(6);
Console.WriteLine($"n1={n1}  n2={n2}"); // n1=25  n2=36
```

Ссылок: 2

```
async Task<int> SquareAsync(int n)
{
    await Task.Delay(0);
    return n * n;
}
```

Task<T>

```
Person person = await GetPersonAsync("Tom");
Console.WriteLine(person.Name); // Tom
// определение асинхронного метода
Ссылок: 1
async Task<Person> GetPersonAsync(string name)
{
    await Task.Delay(0);
    return new Person(name);
}
Ссылок: 3
record class Person(string Name);
```

```
var square5 = SquareAsync(5);
var square6 = SquareAsync(6);

Console.WriteLine("Остальные действия в методе Main");

int n1 = await square5;
int n2 = await square6;
Console.WriteLine($"n1={n1}  n2={n2}"); // n1=25  n2=36

Ссылок: 2
async Task<int> SquareAsync(int n)
{
    await Task.Delay(0);
    var result = n * n;
    Console.WriteLine($"Квадрат числа {n} равен {result}");
    return result;
}
```

Графическое приложение

Программе с графическим интерфейсом нужно загрузить данные из интернета. Если делать это синхронно, весь интерфейс программы заблокируется на время загрузки, и пользователь не сможет взаимодействовать с программой. Это очень плохо сказывается на удобстве использования.

```
<Window x:Class="WpfApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="450" Width="800">
    <Grid>
        <Button Content="Загрузить данные" Click="Button_Click"/>
    </Grid>
</Window>
```

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    await LoadDataAsync();
}

private async Task LoadDataAsync()
{
    using (HttpClient client = new HttpClient())
    {
        string url = "https://jsonplaceholder.typicode.com/todos/1";
        string response = await client.GetStringAsync(url);
        MessageBox.Show(response);
    }
}
```

Графическое приложение

Пользователь нажимает кнопку, начинается асинхронная загрузка данных из интернета. Благодаря использованию await, основной поток не блокируется, и пользователь может продолжать взаимодействовать с интерфейсом программы (например, перемещать окно, нажимать другие кнопки и т.д.), пока данные загружаются.

Преимущества асинхронного кода

- Улучшает производительность
- Повышение отзывчивости интерфейсов
- Упрощение параллелизма
- Экономия ресурсов
- Простота написания и поддержки кода
- Упрощенное управление состоянием

Недостатки асинхронного кода

- Сложность понимания и отладки
- Риск ошибок синхронизации
- Требует особого подхода к проектированию
- Не всегда оправдан

Задание

Расширить функциональность приложения со слайда 20, добавив дополнительную кнопку производящую какое-то другое действие.