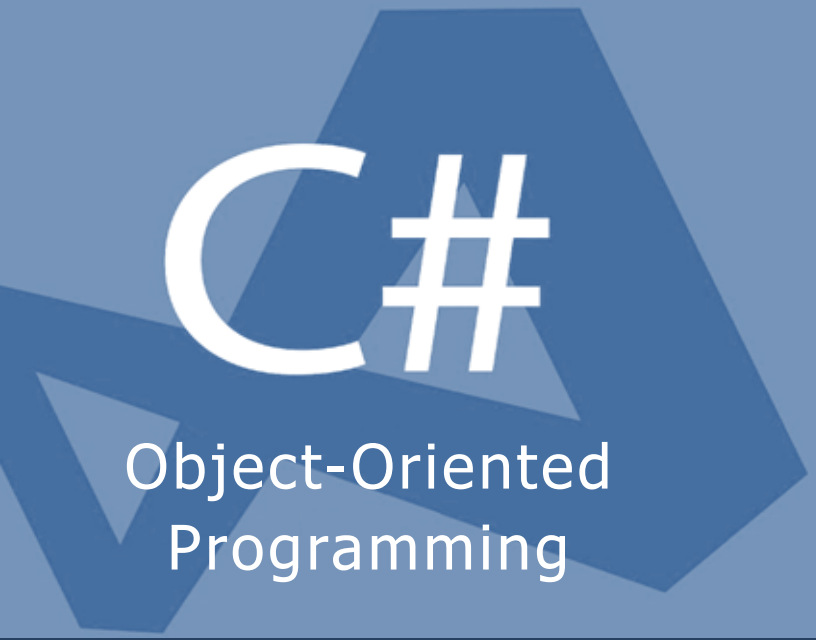


Object-Oriented Programming

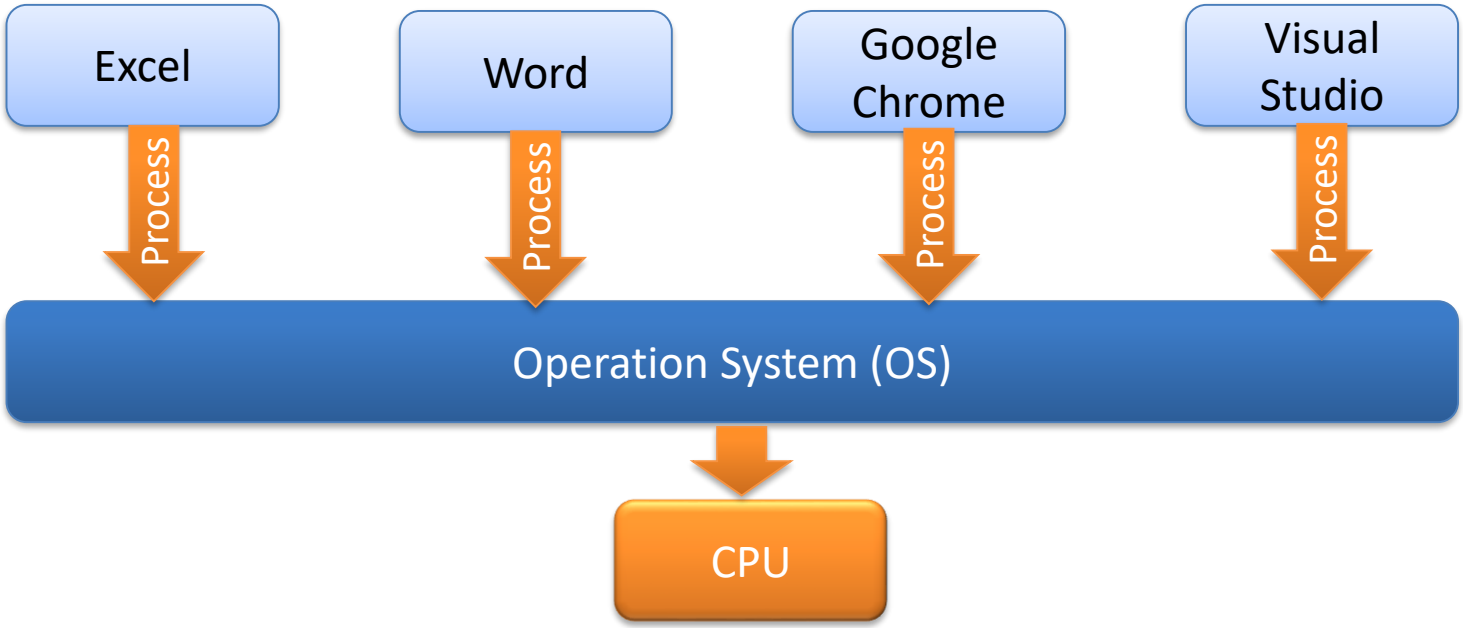
Multitasking & Multithreading

д.т.н. Емельянов Виталий Александрович

: v.yemelyanov@gmail.com

Многозадачность

Многозадачность, ***multitasking*** – это такой способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются несколько программ.



Понятие процесса

Упрощенно: ***процесс (process)*** **- работающий в текущий момент экземпляр программы.**

Для процесса требуется ряд *ресурсов*:

- время процессора,
- память,
- файлы,
- устройства ввода-вывода,
- сетевые устройства
- и др.

При создании процесса для него создается новое пространство виртуальной памяти.

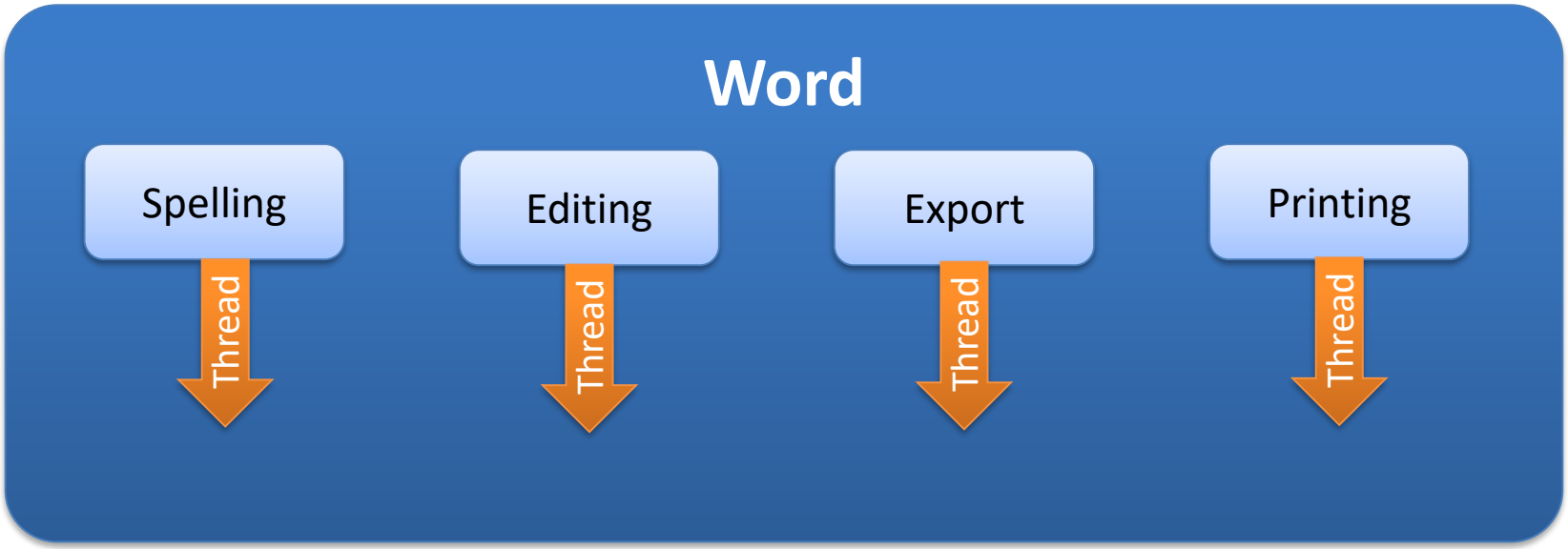
Многозадачность

Многозадачность:

- Многопроцессное выполнение подразумевает оформление каждой подзадачи в виде отдельной программы (процесса).
- Каждый процесс функционирует в своем виртуальном адресном пространстве, не пересекающемся с адресным пространством другого процесса.
- Для взаимодействия подзадач необходимо использовать специальные средства межпроцессной коммуникации (интерфейсы передачи сообщений, общие файлы, объекты ядра операционной системы).

Многопоточность

Поток – отдельная исполняемая часть кода для решения подзадачи внутри **процесса**. **Процесс** может содержать множество исполняемых **потоков**



Многопоточность

Многопоточность:

- Потоки позволяют выделить подзадачи в рамках одного процесса.
- Все потоки одного приложения работают в рамках одного адресного процесса. Для взаимодействия потоков не нужно применять какие-либо средства коммуникации.
- Потоки могут непосредственно обращаться к общим переменным, которые изменяют другие потоки.
- Работа с общими переменными приводит к необходимости использования средств синхронизации, регулирующими порядок работы потоков с данными.

Отличия многопоточности и многозадачности

Многозадачность	Многопоточность
Многозадачность позволяет процессору выполнять несколько задач одновременно.	Многопоточность позволяет процессору одновременно выполнять несколько потоков 1 процесса.
В многозадачной системе приходится выделять отдельную память и ресурсы для каждой программы, которую выполняет ЦП.	В многопоточной системе приходится выделять память для процесса, несколько потоков этого процесса совместно используют одну и ту же память и ресурсы, выделенные процессу.

Класс Thread

Класс **Thread** (пространство имен System.Threading) определяет ряд методов и свойств, которые позволяют управлять потоком и получать информацию о нем. В программе на C# есть как минимум один поток - главный поток, в котором выполняется метод Main.

Свойства класса Thread	
IsAlive	указывает, работает ли поток в текущий момент
Name	содержит имя потока (по умолчанию свойство не установлено)
ManagedThreadId	возвращает числовой идентификатор текущего потока
Priority	хранит приоритет потока (значения **Lowest**, **BelowNormal**, **Normal**, **AboveNormal**, **Highest**) По умолчанию потоку задается значение Normal.

Класс Thread

Свойства класса Thread	
ThreadState	возвращает состояние потока. Значения: • **Aborted**: поток остановлен, но пока еще окончательно не завершен • **AbortRequested**: для потока вызван метод Abort, но остановка потока еще не произошла • **Background**: поток выполняется в фоновом режиме • **Running**: поток запущен и работает (не приостановлен) • **Stopped**: поток завершен • **StopRequested**: поток получил запрос на остановку • **Suspended**: поток приостановлен • **SuspendRequested**: поток получил запрос на приостановку • **Unstarted**: поток еще не был запущен • **WaitSleepJoin**: поток заблокирован в результате действия методов Sleep или Join

Ценности качественного кода

Методы класса Thread	
GetDomain()	Статический метод. Возвращает ссылку на домен приложения
Sleep()	Статический метод. Останавливает поток на определенное количество миллисекунд
Interrupt()	прерывает поток, который находится в состоянии WaitSleepJoin
Join()	блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод
Start()	запускает поток

Создание и запуск потока

C#

```
1         static void Main(string[] args)
2         {
3             Thread Thread1 = new Thread(GetInfo);    // создаем новый поток Thread1
4
5             Thread1.Start();                         // запускаем поток Thread1
6
7             Console.ReadKey();
8
9         }
10
11        static void GetInfo()
12        {
13            Console.WriteLine("Новый поток!");
14        }
15
16
17
18
19
20
21
```

Передача параметров в поток

C#

```
static void Main(string[] args)
{
    int n = 10;
    Thread Thread1 = new Thread(CalculateInThread);   // создаем новый поток Thread1

    Thread1.Start(n);    // запускаем поток Thread1



    Console.ReadKey();

}

static void CalculateInThread(object obj)
{
    if (obj is int n)
    {
        if(n<10)
            Console.WriteLine("Значение параметра < 10");
    }
}
```

Одновременное выполнение потоков

```
Вычисления внутри главного потока: 0
Вычисления внутри главного потока: 1
Вычисления внутри второго потока: 0
Вычисления внутри второго потока: 1
Вычисления внутри второго потока: 2
Вычисления внутри второго потока: 3
Вычисления внутри второго потока: 4
Вычисления внутри главного потока: 2
Вычисления внутри главного потока: 3
Вычисления внутри главного потока: 4
```

```
// создаем новый поток Thread1
// запускаем поток Thread1


потока: {i}");
```

```
12
13  static void Calcul
14  {
15      for (int i = 0
16      {
17          Console.Wr
18          Thread.Sle
19      }
20  }
21
```

```
: 0
Вычисления внутри второго потока: 0
Вычисления внутри главного потока: 1
Вычисления внутри второго потока: 1
Вычисления внутри главного потока: 2
Вычисления внутри второго потока: 2
Вычисления внутри главного потока: 3
Вычисления внутри главного потока: 4
Вычисления внутри второго потока: 3
Вычисления внутри второго потока: 4
```

Гонка данных

C#

```
1  static int x = 1;
2  static void Main(string[] args)
3  {
4      Thread Thread1 = new Thread(Calc);    // создаем новый поток Thread1
5      Thread1.Name = "Поток 1";
6      Thread1.Start();                      // запускаем поток Thread1
7
8      Thread Thread2 = new Thread(Calc);    // создаем новый поток Thread2
9      Thread2.Name = "Поток 2";
10     Thread2.Start();                      // запускаем поток Thread2
11  }
12
13 static void Calc()
14 {
15     for (int i = 1; i <
16     {
17         Console.WriteLir
18         x++;
19         Thread.Sleep(100
20     }
21 }
```

```
Поток 1: 1
Поток 2: 1
Поток 2: 3
Поток 1: 3
Поток 1: 5
Поток 2: 5
Поток 1: 7
Поток 2: 7
Поток 2: 9
Поток 1: 9
```

Синхронизация потоков: lock

C#

```
1    static int x=1;
2    static object locker = new object(); // объект-заглушка для оператора lock
3    static void Main(string[] args)
4    {
5        Thread Thread1 = new Thread(Calc);   // создаем новый поток Thread1
6        Thread1.Name = "Поток 1";
7        Thread1.Start();                     // запускаем поток Thread1
8
9        Thread Thread2 = new Thread(Calc);   // создаем новый поток Thread2
10       Thread2.Name = "Поток 2";
11       Thread2.Start();                     // запускаем поток Thread2
12   }
13   static void Calc()
14   {
15       lock (locker)  //определяем
16       {
17           for (int i = 1; i < 6;
18           {
19               Console.WriteLine($
20               x++;
21           }
22       }
23   }
```

```
Поток 1: 1
Поток 1: 2
Поток 1: 3
Поток 1: 4
Поток 1: 5
Поток 2: 6
Поток 2: 7
Поток 2: 8
Поток 2: 9
Поток 2: 10
```

Синхронизация потоков: класс AutoResetEvent

Класс **AutoResetEvent** представляет событие синхронизации потоков, который позволяет при получении сигнала переключить данный объект-событие из сигнального в несигнальное состояние.

Событие синхронизации может находиться в сигнальном и несигнальном состоянии. Если состояние события несигнальное, поток, который вызывает метод WaitOne, будет заблокирован, пока состояние события не станет сигнальным. Метод Set, наоборот, задает сигнальное состояние события.

Методы класса AutoResetEvent	
Reset()	задает несигнальное состояние объекта, блокируя потоки
Set()	задает сигнальное состояние объекта, позволяя одному или несколким ожидающим потокам продолжить работу
WaitOne()	задает несигнальное состояние и блокирует текущий поток, пока текущий объект AutoResetEvent не получит сигнал

Синхронизация потоков: класс AutoResetEvent

C#

```
1    static int x=1;
2    static AutoResetEvent WaitingEvent = new AutoResetEvent(true);
3
4    static void Main(string[] args)
5    {
6        Thread Thread1 = new Thread(Calc);    // создаем новый поток Thread1
7        Thread1.Name = "Поток 1";
8        Thread1.Start();                      // запускаем поток Thread1
9
10       Thread Thread2 = new Thread(Calc);    // создаем новый поток Thread2
11       Thread2.Name = "Поток 2";
12       Thread2.Start();
13   }
14   static void Calc()
15   {
16       WaitingEvent.WaitOne(
17       for (int i = 1; i < 6
18           {
19               Console.Write
20               x++;
21           }
22       WaitingEvent.Set();
23   }
```

```
Поток 1: 1
Поток 1: 2
Поток 1: 3
Поток 1: 4
Поток 1: 5
Поток 2: 6
Поток 2: 7
Поток 2: 8
Поток 2: 9
Поток 2: 10
```

Синхронизация потоков: класс Mutex

C#

```
1    static int x=1;
2    static Mutex mutex1 = new Mutex();
3
4    static void Main(string[] args)
5    {
6        Thread Thread1 = new Thread(Calc);    // создаем новый поток Thread1
7        Thread1.Name = "Поток 1";
8        Thread1.Start();                      // запускаем поток Thread1
9
10       Thread Thread2 = new Thread(Calc);    // создаем новый поток Thread2
11       Thread2.Name = "Поток 2";
12       Thread2.Start();                      // запускаем поток Thread2
13   }
14   static void Calc()
15   {
16       mutex1.WaitOne();          // приостанавливаем поток до получения мьютекса
17       for (int i = 1; i < 6; i++)
18           {
19               Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
20               x++;
21           }
22       mutex1.ReleaseMutex();     // освобождаем мьютекс
23   }
```

```
Поток 1: 1
Поток 1: 2
Поток 1: 3
Поток 1: 4
Поток 1: 5
Поток 2: 6
Поток 2: 7
Поток 2: 8
Поток 2: 9
Поток 2: 10
```

Параллельное программирование и библиотека TPL

Библиотека параллельных задач TPL (Task Parallel Library) - пространство имен **System.Threading.Tasks**

В основе библиотеки TPL лежит концепция задач, каждая из которых описывает отдельную продолжительную операцию.

В библиотеке классов .NET задача представлена специальным классом - классом **Task**, который находится в пространстве имен **System.Threading.Tasks**. Данный класс описывает отдельную задачу, которая запускается асинхронно в одном из потоков из пула потоков. Хотя ее также можно запускать синхронно в текущем потоке.

Класс Task

Свойства класса Task	
Id	возвращает идентификатор текущей задачи
Status	возвращает статус задачи. Значения: **Canceled:** задача отменена **Created:** задача создана, но еще не запущена **Faulted:** в процессе работы задачи произошло исключение **RanToCompletion:** задача успешно завершена **Running:** задача запущена, но еще не завершена **WaitingForActivation:** задача ожидает активации и постановки в график выполнения **WaitingForChildrenToComplete:** задача завершена и теперь ожидает заврешения прикрепленных к ней дочерних задач **WaitingToRun:** задача поставлена в график выполнения, но еще не начала свое выполнение
IsCompleted	возвращает true, если задача завершена
IsCanceled	возвращает true, если задача была отменена.

Класс Task

Библиотека параллельных задач TPL (Task Parallel Library) - пространство имен **System.Threading.Tasks**

В основе библиотеки TPL лежит концепция задач, каждая из которых описывает отдельную продолжительную операцию.

В библиотеке классов .NET задача представлена специальным классом - классом **Task**, который находится в пространстве имен **System.Threading.Tasks**. Данный класс описывает отдельную задачу, которая запускается асинхронно в одном из потоков из пула потоков. Хотя ее также можно запускать синхронно в текущем потоке.

Класс Task

Создание и запуск объекта Task:

C#

```
1
2   Task task1 = new Task(Calc);
3   task1.Start();
4
5
```

Статический метод **Task.Factory.StartNew()** сразу же запускает задачу:

C#

```
1
2   Task task2 = Task.Factory.StartNew(Calc);
3
4
5
```

Возврат результата из задач

C#

```
1   static void Main(string[] args)
2   {
3       int x1 = 4, x2 = 5;
4
5       Task<int> CalcTask = new Task<int>(()=>Calc(x1, x2));
6       CalcTask.Start();
7
8       int result = CalcTask.Result;
9
10      Console.WriteLine($"{x1} + {x2} = {result}");
11
12
13  }
14
15  static int Calc(int a, int b)
16  {
17      return a + b;
18  }
19
20
21
22
23
```

Задачи продолжения

Задачи продолжения (**continuation task**) позволяют определить задачи, которые выполняются после завершения других задач.

C#

```
1   static void Main(string[] args)
2   {
3       Task<int> Task1 = new Task<int>(() => Calc(1, 2));
4
5       // задача продолжения
6       Task Task2 = Task1.ContinueWith(task => GetResult(task.Result));
7
8       Task1.Start();
9   }
10  static void GetResult(int sum)
11  {
12      Console.WriteLine($"Sum: {sum}");
13  }
14
15  static int Calc(int a, int b)
16  {
17      return a + b;
18  }
19
```

Класс Parallel

Класс **Parallel** является частью TPL и предназначен для упрощения параллельного выполнения кода

Методы класса Parallel	
Invoke()	позволяет осуществлять параллельное выполнение задач
For()	позволяет выполнять итерации цикла параллельно.
ForEach()	осуществляет итерацию по коллекции, реализующей интерфейс **IEnumerable**, подобно циклу foreach, только осуществляет параллельное выполнение перебора.

Класс Parallel

Метод **Invoke()**:

C#

```
1  static void Main(string[] args)
2  {
3      Parallel.Invoke(Calc1, Calc2);
4  }
5
6  static void Calc1()
7  {
8      int y;
9      for (int i = 0; i < 5; i++)
10     {
11         y=i+1;
12     }
13
14 }
15
16 static void Calc2()
17 {
18     int x=10;
19     x = x * 10;
20 }
```

Метод **For()**:

C#

```
1  static void Main(string[] args)
2  {
3      Parallel.For(1,11, Calc);
4  }
5
6
7
8
9  static void Calc(int x)
10 {
11     x=10;
12     x = x * 10;
13     Console.WriteLine(x);
14 }
15
16
17
18
19
20
```