Multithreading

- Multithreading
  - Thread
    - System.Threading.Thread
  - Thread Pool
    - ExecutionContext
    - Cancellation
  - TPL
    - Класс Task
      - Continuation
    - async / await
    - SyncronizationContext

Thread

- Thread в винде:
  - Объект ядра потока (thread kernel object). контекст потока, набор регистров процессора ~ 1KB
  - Блок окружения потока (Thread Environment Block, TEB). 4KB, содержит заголовок цепочки обработки исключений, локальное хранилище данных для потока и некоторые структуры данных, используемые интерфейсом графических устройств (GDI) и графикой OpenGL.
  - Стек пользовательского режима (user-mode stack). По умолчанию на каждый стек Windows выделяет 1 Мбайт памяти.
  - Стек режима ядра (kernel-mode stack). x86 - 12 KB, X64 — 24 Кбайт.

System.Threading.Thread

- Thread - соответствует потоку в ОС
- Самый низкоуровневый объект в C# для работы с потоками
- Запрещен в приложениях для windows store

```
public static void Main()
{
    Console.WriteLine("Main thread");
    Thread dedicatedThread = new Thread(Compute);
    dedicatedThread.Start(5);
    Console.WriteLine("Main thread: Doing other work");
    Thread.Sleep(2000);     // Имитация другой работы
    dedicatedThread.Join(); // Ожидание завершения потока
    Console.WriteLine("Main thread: ending");
}

// Передаем делегат ParameterizedThreadStart в конструктор Thread
private static void Compute(Object state)
{
    Console.WriteLine("Compute: state={0}", state);
    Thread.Sleep(1000);
}
```

Методы:

- Abort уведомить CLR, что надо прекратить поток (для проверки завершенности следует опрашивать свойство ThreadState)
- Interrupt прервать поток на время
- Join остановить вызвающий поток до завершения потока, экземляру которого был вызван данный метод
- Resume возобновить работу потока
- Start запустить поток (при этом поток непосредственно создается в ОС)
- Suspend приостановить
- static Sleep останавливить поток
- static GetDomain ссылка на домен приложения
- static GetDomainId id текущего домена приложения

Приоритет потоков:

```
dedicatedThread.Priority = ThreadPriority.AboveNormal;
```

- Табличка, как связаны C# приоритеты потоков, приоритеты процессов в винде с реальными приоритетами потоков в ОС
- Левая колонка - приоритеты потоков в c#, Заголовки колонок - Process priority
- 17+ - драйвера устройств

CLR Priority	Idle	Below Normal	Normal	Above Normal	High	Realtime
Time-Critical	15	15	15	15	15	31
Highest	6	8	10	12	15	26
Above Normal	5	7	9	11	14	25
Normal	4	6	8	10	13	24
Below Normal	3	5	7	9	12	23
Lowest	2	4	6	8	11	22
Idle	1	1	1	1	1	16

Idle	1	1	1	1	1	16

В CLR все потоки делятся на foreground / background

- При завершении активного потока:
  - Принудительно завершит все фоновые потоки
  - Все фоновые потоки завершатся немедленно и без появления исключений
- Thread - по-умолчанию foreground, ThreadPool - background
- IsBackground - можно изменять в процессе работы
- Общая рекомендация - лучше использовать фоновые потоки

```
public static void Main()
{
    Thread t = new Thread(Worker);
    t.IsBackground = true;
    t.Start();

    // Активный поток (IsBackground = false) - приложение будет работать около 10
секунд
    // Фоновый поток (IsBackground = true) - немедленно прекратит работу
    // В LINQPad5 работает криво, в студии работает нормально :)
    Console.WriteLine("Returning from Main");
}
private static void Worker()
{
    Thread.Sleep(10000);
    Console.WriteLine("Returning from Worker");
}
```

Thread Pool

- Создавать потоки руками - очень низкоуровневый подход
- CLR умеет управлять собственным пулом потоков, чтобы не плодить лишние потоки
- На каждый объект CLR создается свой пул потоков, который используют все AppDomain
- Пулл потоков динамически определяет количество реальных потоков, которые необходимы приложению и сам добавляет / удаляет потоки в зависимости от того, какие задачи ставит приложение

```
public static class ThreadPool
{
    static Boolean QueueUserWorkItem(WaitCallback callBack);
    static Boolean QueueUserWorkItem(WaitCallback callBack, Object state);

    delegate void WaitCallback(Object state);
    ...
}
```

Базовый пример:

```
public static void Main()
{
    Console.WriteLine("Main thread: starting");
    ThreadPool.QueueUserWorkItem(Compute, 5);

    Console.WriteLine("Main thread: 10 sec waiting");
    Thread.Sleep(10000);

    Console.WriteLine("Main thread: exit");
}

private static void Compute(Object state)
{
    Console.WriteLine($"Compute: state = {state}");
    Thread.Sleep(1000);
}
```

ExecutionContext

Не лишним будет упомянуть, что есть контекст выполнения потока:

- Параметры безопасности, Principal
- Контекстные данные логического вызова
- Копирование контекста занимает много ресурсов
- По-умолчанию для новых потоков копируется контекст безопасности
- Можно запретить копирование контекста

```
public sealed class ExecutionContext : IDisposable, ISerializable
{
    [SecurityCritical] public static AsyncFlowControl SuppressFlow();
    public static void RestoreFlow();
    public static Boolean IsFlowSuppressed();
    // Не показаны редко применяемые методы
}
```

Cancellation

- В C# используется стандартный паттерн отмены операций скоординированная отмена - оба класса должны явно поддерживать отмену
- CancellationTokenSource - специальный класс для управления токеном и непосредственно отменой операции

```
public sealed class CancellationTokenSource : IDisposable
{
    public CancellationTokenSource();
    public Boolean IsCancellationRequested { get; }

    // Токен, который передается в параллельный поток, считывая его свойства, поток
может реагировать на отмену и прекращать действие
    public CancellationToken Token { get; }

    // Отменить операцию!
    public void Cancel(); // Вызывает Cancel с аргументом false
    public void Cancel(Boolean throwOnFirstException);
    ...
}
```

- CancellationToken - класс, который передается в параллельный поток и по-которому можно понять, отменили задачу или еще нет.

```
public struct CancellationToken // Значимый тип
{
    // Статическое поле, используется, когда мы не хотим отменять метод, но он
поддерживает прием CancellationToken - передаем ему CancellationToken.None
    public static CancellationToken None { get; }

    // Непосредственные методы для реагирования на отмену
    Boolean IsCancellationRequested { get; }
    public void ThrowIfCancellationRequested();

    // Позволяет регистрировать дополнительные делегаты на событие отмены
    public CancellationTokenRegistration Register(Action<Object> callback, Object
state, Boolean useSynchronizationContext);

    // опущены другие методы, более простые перегрузки Register, GetHashCode,
Equals, == и != и прочее
}
```

Пример использования CancellationTokenSource и CancellationToken:

```
public static void Main()
{
    using (CancellationTokenSource cts = new CancellationTokenSource())
    {
        ThreadPool.QueueUserWorkItem(o => Count(cts.Token, 1000));
        Thread.Sleep(1000);
        cts.Cancel(); // Если метод Count уже вернул управления, Cancel не
оказывает никакого эффекта

        Thread.Sleep(1000);
        Console.WriteLine("Quit the programm");
    }
}

private static void Count(CancellationToken token, Int32 countTo)
{
    for (Int32 count = 0; count < countTo; count++)
    {
        if (token.IsCancellationRequested)
        {
            Console.WriteLine("Count is cancelled");
            break;
```

```
            }
            Console.WriteLine(count);
            Thread.Sleep(200);
        }
        Console.WriteLine("Count is done");
    }
```

Пример регистрации дополнительных событий на отмену действия:

```
public static void Main()
{
    CancellationTokenSource cts = new CancellationTokenSource();
    CancellationToken token = cts.Token;

    var obj1 = new CancelableObject("1");
    var obj2 = new CancelableObject("2");

    token.Register(() => obj1.Cancel());
    token.Register(() => obj2.Cancel());

    cts.Cancel();
    cts.Dispose();
}

class CancelableObject
{
    public string id;

    public CancelableObject(string id)
    {
        this.id = id;
```

```
    }

    public void Cancel()
    {
        Console.WriteLine("Object {0} Cancel callback", id);
    }
}
```

Несколько CancellationTokenSource можно объединить в один и обрабатывать несколько причин отмены сразу:

```
var cts1 = new CancellationTokenSource();
cts1.Token.Register(() => Console.WriteLine("cts1 canceled"));
var cts2 = new CancellationTokenSource();
cts2.Token.Register(() => Console.WriteLine("cts2 canceled"));

var linkedCts = CancellationTokenSource.CreateLinkedTokenSource(cts1.Token,
cts2.Token);
linkedCts.Token.Register(() => Console.WriteLine("linkedCts canceled"));

cts2.Cancel(); // Отмена любого из дочерних приводит к отмене общего

Console.WriteLine("cts1 canceled={0}, cts2 canceled={1}, linkedCts={2}",
    cts1.IsCancellationRequested, cts2.IsCancellationRequested,
linkedCts.IsCancellationRequested);
```

Есть стандартные методы, для отмены операций по Тайм-ауту:

```
public sealed class CancellationTokenSource : IDisposable
{
    public CancellationTokenSource(Int32 millisecondsDelay);
    public CancellationTokenSource(TimeSpan delay);
    public void CancelAfter(Int32 millisecondsDelay);
    public void CancelAfter(TimeSpan delay);
}
```

TPL

System.Threading.Tasks

Класс Task

- с ThreadPool есть глобальные проблемы:
  - возврат результатов из потока
  - как узнать о завершении операции
- Упрощенно: Task - типизированная обертка над пуллом потоков с кучей удобных методов
- Напоминаю, что все Task - берутся из пула и фоновые

```
ThreadPool.QueueUserWorkItem(ComputeBoundOp, 5);

new Task(ComputeBoundOp, 5).Start();

Task.Run(() => ComputeBoundOp(5));
```

Простейший пример создания задачи:

```
Task taskA = Task.Run( () => Console.WriteLine("Hello from thread '{0}'.",
Thread.CurrentThread.ManagedThreadId ));

Console.WriteLine("Hello from thread '{0}'.", Thread.CurrentThread.ManagedThreadId
);
taskA.Wait();
```

Есть типизированная версия Task<TResult> для возвращения конкретного результата:

```
// Создание задания Task (автоматически оно пока не выполняется)
Task<Int32> t = new Task<Int32>(n => Sum((Int32)n), 1000000000);

t.Start();  // Можно начать выполнение задания через некоторое время

t.Wait();   // Можно ожидать завершения задания в явном виде
            // !!!!!! Существует перегруженная версия, принимающая тайм-
аут/CancellationToken

Console.WriteLine("The Sum is: " + t.Result); // Получение результата (свойство
```

```
Result вызывает метод Wait)
```

- .Result
  - Автоматически внутри вызывает метод .Wait()
- .Wait()
  - если метод еще не начал выполняться - может выполнять его прямо в текущем потоке, что потенциально может приводить к дедлокам
- Исключения, сделанные в методах задачи, сохраняются в отдельную коллекцию и при вызове .Wait() / .Result возвращаются исходному коду в виде AggregateException, который будет содержать коллекцию со всеми исключениями
  - Если не вызвать wait / result, то основной поток не узнает об ошибке

Пример, когда мы запускаем несколько задач и ждем, пока завершатся все:

```
Task[] tasks = new Task[3]
{
    new Task(() => Console.WriteLine("First")),
    new Task(() => Console.WriteLine("Second")),
    new Task(() => Console.WriteLine("Third"))
};
foreach (var t in tasks)
{
    t.Start();
}
Task.WaitAll(tasks);

Console.WriteLine("End");
```

Continuation

- ContinueWith - позволяет сразу настроить продолжение действия, вместо блокирования текущего потока
- возвращает Task, который частенько не используется

```
public static void Main()
{
    Task task1 = new Task(()=>{ Console.WriteLine($"current: {Task.CurrentId}");
});
    Task task2 = task1.ContinueWith(Display);
    Task task3 = task1.ContinueWith((Task t) => { Console.WriteLine($"current:
{Task.CurrentId}"); });
    Task task4 = task2.ContinueWith((Task t) => { Console.WriteLine($"current:
{Task.CurrentId}"); });

    task1.Start();
    task1.Wait(); // Будет ждать только task1, а не всю цепочку!
    Console.WriteLine("After task1 wait");
    Thread.Sleep(5000);
    Console.WriteLine("End");
}
```

```
static void Display(Task t)
{
    Console.WriteLine($"current: {Task.CurrentId}, previous: {t.Id} ");
    Thread.Sleep(3000);
}
```

Пример отмены задания с выбрасыванием исключения и обработкой такой ситуации в основном коде:

```
public static void Main()
{
    CancellationTokenSource cts = new CancellationTokenSource();
    Task<int> t = new Task<int>(() => Sum(cts.Token, 2), cts.Token);

    t.Start();

    cts.Cancel(); // кстати задача уже может быть завершена
    try
    {
        // В случае отмены задания именно метод Result генерирует исключение
AggregateException
        Console.WriteLine("The sum is: " + t.Result);
    }
    catch (AggregateException x)
    {
        // Пометить все OperationCanceledException ошибки, как обработанные,
        // остальные записать в новый объект AggregateException, выбросить его
заново, если он не пуст
        x.Handle(e => e is OperationCanceledException);

        Console.WriteLine("Sum was canceled");  // Строка выполняется, если все
```

```
исключения уже обработаны
    }
}
```

```
private static Int32 Sum(CancellationToken ct, int n)
{
    int sum = 0;
    for (; n > 0; n--)
    {
        ct.ThrowIfCancellationRequested(); // исключение OperationCanceledException
        checked { sum += n; }
        // исключение System.OverflowException
    }
    return sum;
}
```

- Проблемы чистых Task
  - .Result блокирует поток, что не хорошо.
  - Писать реальный код с ContinueWith очень сложно и код получается тяжелый. Пример msdn

Если совсем упрощенно:

```
Factory.StartNew(() => DoSomeAsyncWork())
    .ContinueWith(
        (antecedent) =>
        {
            DoSomeWorkAfter();
        },
        TaskScheduler.FromCurrentSynchronizationContext());
```

Заменяется:

```
await DoSomeAsyncWork();
DoSomeWorkAfter();
```

- Ключевое слово async
  - Включает в методе поддержку await
  - Изменяет как обрабатывается результат
  - Не создает отдельных потоков или любой другой магии
- Ключевое слово await
  - Что-то в духе asynchronous wait
  - Некоторый унарный оператор, который принимает асинхронную операцию
  - Откладывает остаток метода до завершения операции, если она еще не выполнена
  - То есть метод ставится на паузу (что-то похожее на yielding) до завершения операции
  - Поток не блокируется

```
public async Task DoSomethingAsync()
{
  // In the Real World, we would actually do something...
  // For this example, we're just going to (asynchronously) wait 100ms.
  await Task.Delay(100);
}
```

```
async Task<int> AccessTheWebAsync()
{
    HttpClient client = new HttpClient();

    // GetStringAsync returns a Task<string>. That means that when you await the
    // task you'll get a string (urlContents).
    Task<string> getStringTask =
client.GetStringAsync("http://msdn.microsoft.com");

    // You can do work here that doesn't rely on the string from GetStringAsync.
    DoIndependentWork();

    // The await operator suspends AccessTheWebAsync.
    //  - AccessTheWebAsync can't continue until getStringTask is complete.
    //  - Meanwhile, control returns to the caller of AccessTheWebAsync.
    //  - Control resumes here when getStringTask is complete.
    //  - The await operator then retrieves the string result from getStringTask.
    string urlContents = await getStringTask;

    return urlContents.Length;
}
```

```
public async Task NewStuffAsync()
{
  // Use await and have fun with the new stuff.
  await ...
}

public Task MyOldTaskParallelLibraryCode()
{
  // Note that this is not an async method, so we can't use await in here.
  ...
}

public async Task ComposeAsync()
{
  // We can await Tasks, regardless of where they come from.
  await NewStuffAsync();
  await MyOldTaskParallelLibraryCode();
}
```

- Что может возвращать async метод
  - Task<T>
  - Task
  - void

Всегда лучше возвращать Task вместо void, который используется в очень специфичных сценариях типа EventHandler / static async void MainAsync()

```
public async Task<int> CalculateAnswer()
{
  await Task.Delay(100); // (Probably should be longer...)

  return 42; // Return a type of "int", not "Task<int>"
}
```

Before	After	Comments
task.Wait	await task	Wait/await for a task to complete
task.Result	await task	Get the result of a completed task
Task.WaitAny	await Task.WhenAny	Wait/await for one of a collection of tasks to complete
Task.WaitAll	await Task.WhenAll	Wait/await for every one of a collection of tasks to complete
Thread.Sleep	await Task.Delay	Wait/await for a period of time
Task constructor	Task.Run or TaskFactory.StartNew	Create a code-based task

Context

```
// WinForms example (it works exactly the same for WPF).
private async void DownloadFileButton_Click(object sender, EventArgs e)
{
  // Since we asynchronously wait, the UI thread is not blocked by the file
download.
  await DownloadFileAsync(fileNameTextBox.Text);

  // Since we resume on the UI context, we can directly access UI elements.
  resultTextBox.Text = "File downloaded!";
}
```

```
private async Task DownloadFileAsync(string fileName)
{
  // Use HttpClient or whatever to download the file contents.
  var fileContents = await
DownloadFileContentsAsync(fileName).ConfigureAwait(false);

  // Note that because of the ConfigureAwait(false), we are not on the original
context here.
  // Instead, we're running on the thread pool.

  // Write the file contents out to a disk file.
  await WriteToDiskAsync(fileName, fileContents).ConfigureAwait(false);
  // The second call to ConfigureAwait(false) is not *required*, but it is Good
Practice.
}

// WinForms example (it works exactly the same for WPF).
private async void DownloadFileButton_Click(object sender, EventArgs e)
{
  // Since we asynchronously wait, the UI thread is not blocked by the file
download.
  await DownloadFileAsync(fileNameTextBox.Text);
```

```
    // Since we resume on the UI context, we can directly access UI elements.
    resultTextBox.Text = "File downloaded!";
}
```

- В .NET Core контекста нет
    - Вся информация, которая раньше хранилась в контексте, теперь передается через DI
- В .NET Framework контекст есть
    - Если нам не нужен контекс смело везде фигачим .ConfigureAwait(false)
- Если .NET Standard библиотека предполагает широкое использование (в том числе на .NET Framework), лучше .ConfigureAwait(false) везде, где это возможно.

```
public async Task DoOperationsConcurrentlyAsync()
{
  Task[] tasks = new Task[3];
  tasks[0] = DoOperation0Async();
  tasks[1] = DoOperation1Async();
  tasks[2] = DoOperation2Async();

  // At this point, all three tasks are running at the same time.

  // Now, we await them all.
  await Task.WhenAll(tasks);
}
```

```
public async Task<int> GetFirstToRespondAsync()
{
  // Call two web services; take the first response.
  Task<int>[] tasks = new[] { WebService1Async(), WebService2Async() };

  // Await for the first one to respond.
  Task<int> firstTask = await Task.WhenAny(tasks);

  // Return the result.
  return await firstTask;
}
```

```
await someObject;
```

```
private class FooAsyncStateMachine : IAsyncStateMachine
{
    // Member fields for preserving “locals” and other necessary state
    int $state;
    TaskAwaiter $awaiter;
    …
    public void MoveNext()
    {
        // Jump table to get back to the right statement upon resumption
        switch (this.$state)
        {
            …
            case 2: goto Label2;
            …
        }
        …
        // Expansion of “await someObject;”
        this.$awaiter = someObject.GetAwaiter();
        if (!this.$awaiter.IsCompleted)
```

```
        {
            this.$state = 2;
            this.$awaiter.OnCompleted(MoveNext);
            return;
            Label2:
        }
        this.$awaiter.GetResult();
        …
    }
}
```

- await нельзя:
  - в property getter/setter
  - inside lock/synclock
  - inside catch/finally
  - some other situations
- await someTask; vs someTask.Wait;
- task.Result vs task.GetAwaiter().GetResult()
  - сами значения в позитивном сценарии абсолютно идентичны
  - при ошибке .Result - AggregationException, GetResult - конкретную ошибку

Links:

- async
  - C# Asynchronous programming
  - MSDN: Асинхронное программирование с использованием ключевых слов Async и Await (C#)
  - Async and await (Stephen Cleary)
  - MSDN: Asynchronous Programming - Async Performance: Understanding the Costs of Async and Await (By Stephen Toub | October 2011)
  - Teplyakov: Dissecting async
  - Jon skeet: тонна мыслей и примеров про async, которые тяжело осмыслить
- habr
  - Habrahabr: async C#
  - Habrahabr: Использование async и await в C# — лучшие практики
- Joseph Albahari (Teplyakov translate)
  - 3 - Thread, Cancellation, Lazy
  - 4 - Barrier, Locks
  - 5.1 PLINQ
  - 5.2 Parallel, TPL, Task, Параллельные коллекции, SpinLock и SpinWait
- SOF
  - SOF: What is the difference between asynchronous programming and multithreading?
  - SOF:Asynchronous vs synchronous execution, what does it really mean?
  - difference-between-await-and-continuewith, good example continue with problem