1. **Активные (foreground) and фоновые (background) потоки**.
   1. В чем отличие? К какому виду потоков по умолчанию относятся:

* потоки, созданные при конструировании объекта Thread;
* потоки в ThreadPool;
* потоки управляемой среды, созданные неуправляемым кодом?

**Активный поток** – поток, обеспечивающий предохранение текущего приложения от завершения: CLR не остановит приложение, пока не прекратят свою работу все активные потоки.

По умолчанию:

1. Основной поток приложения
2. Созданные явно с помощью System.Thread

**Фоновые поток** – поток, не поддерживающий работу CLR и будет прекращен в момент завершения всех активных потоков.

По умолчанию:

1. Потоки в ThreadPool (для которых свойство [*IsThreadPoolThread*](https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.thread.isthreadpoolthread) равно *true*)
2. Потоки, которые включаются в управляемую среду выполнения из неуправляемого кода.
   1. Приведите примеры, когда следует использовать foreground-, а когда background потоки?

**Активные потоки** имеет смысл использовать для исполнения заданий, которые **обязательно** требуется завершить при завершении приложения: часто что-то связанное с сохранением текущего состояния в память.

**Фоновые потоки:** работа анализатора кода или текста в редакторе.

* 1. Какие средства предоставляет .Net Framework для создания background/foreground потоков?

1. Свойство **IsBackground** для получения/переключения статуса потока.  
   **NB:** Изменение статуса потока с активного на фоновый не изменяет его приоритет или статус в планировщика потоков.
2. Создания активного потока: создание экземпляра System.Thread.
3. Создание фонового потока: использование потока из ThreadPool.
   1. Вызов статического метода ThreadPool.QueueUserWorkItem.
   2. Создание Task.

Джефри Рихтер. Глава 26. “Фоновые и активные потоки”

1. **Конструкции синхронихации (locking)**
   1. Что такое Mutex? Отличие от других конструкций синхронизации.

**Синхронизация** - акт координации параллельно выполняемых действий с целью получения предсказуемых результатов. Особенно важна в случае, когда потоки получают доступ к одним и тем же данным.

**Примитивы синхронизации** - простые программные механизмы, предоставляемые платформой (например, операционной системой) для поддержки синхронизации потоков или процессов. Они обычно строятся с использованием механизмов более низкого уровня.

**Обзор примитивов синхронизации в .NET**: <https://docs.microsoft.com/en-us/dotnet/standard/threading/overview-of-synchronization-primitives>

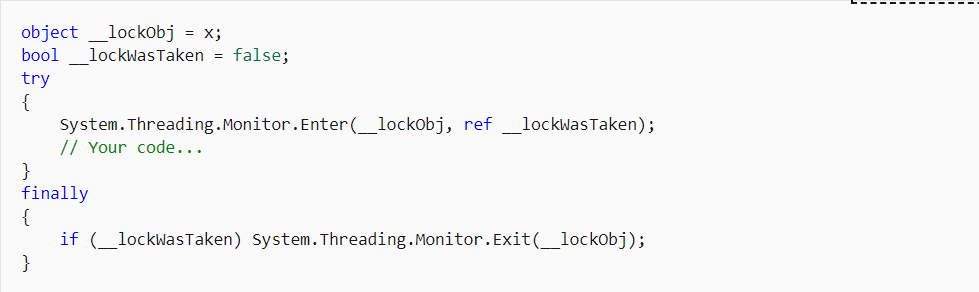
**Mutex** – один из примитивов синхронизации, реализован в виде класса [**System.Threading.Mutex**](https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.mutex) в .NET.

**Свойства:**

* Предоставляет монопольный доступ к общему ресурсу.
* Может использоваться в межпроцессной синхронизации (в отличие от Monitor).
* Может быть освобожден только тем потоком, который им владеет (в NET – метод ReleaseMutex)  
  1. Отличие между lock и Monitor.

Класс **[System.Threading.Monitor](https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.monitor)** предоставляет монопольный доступ к общему ресурсу, блокируя или разблокируя объект, определяющий ресурс.

Оператор **lock**  является синтаксическим сокращением для вызова методов **Monitor.Enter** и **Monitor.Exit** с добавленным блоком try/finally.



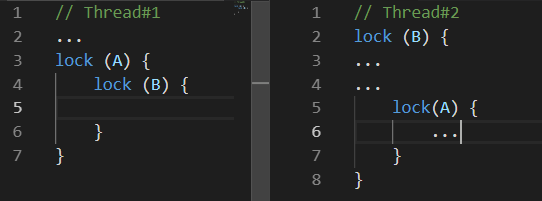
* 1. Как много времени затрачивают lock-конструкции на служебные операции (overhead)? Когда следует их использовать (примеры)?

**Советы по работе с потоками (в том числе про lock):**   
<https://docs.microsoft.com/en-us/dotnet/standard/threading/managed-threading-best-practices>

**Про overhead:**  
https://stackoverflow.com/questions/4673618/how-expensive-is-the-lock-statement

* 1. Как не допустить deadlock при использовании конструкций блокировки? Опишите, что может вызвать deadlock в multithreading. Напишите пример кода, проверьте его в Visual Studio (не следует путать с deadlock в async/await).

**Пример deadlock:** Пока **Thread#1** что-то делал, **Thread#2** заблокировал ресурс **B**, немного позднее **Thread#1** заблокировал ресурс **A** и пытается заблокировать ресурс **B**, к сожалению это никогда не произойдет, т.к. **Thread#2** освободит ресурс **B** лишь после того как заблокирует ресурс **А**.

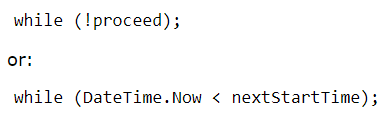


**Одно из решений (из Википедии):** Классический способ борьбы с проблемой — разработка иерархии блокировок: между блокировками устанавливается отношение сравнения и вводится правило о запрете захвата «большей» блокировки в состоянии, когда уже захвачена «меньшая». Таким образом, если процессу нужно несколько блокировок, ему нужно всегда начинать с самой «большой» — предварительно освободив все захваченные «меньшие», если такие есть — и затем в нисходящем порядке. Это может привести к лишним действиям (если «меньшая» блокировка нужна и уже захвачена, она освобождается только чтобы тут же быть захваченной снова), зато гарантированно решает проблему.

Joseph Albahari ["Locking"](http://www.albahari.com/threading/part2.aspx#_Locking)

1. **Blocking vs Spinning**
   1. Что такое spinning и его отличие от блокировки?

**Spinning** – один из способов приостановки выполнения потока, предполагающий наличие цикла с условием, например:

****

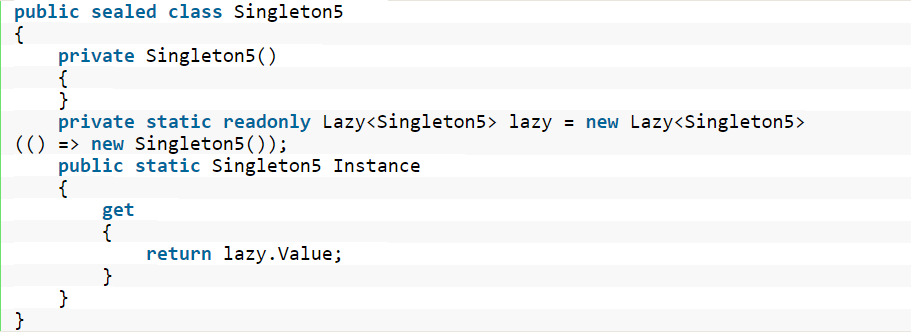
**Spinning** имеет смысл использовать, если условие будет выполнено достаточно быстро (несколько микросекунд). В данном случае можно избежать накладных расходов и задержки из-за переключения контекста.

* 1. SpinLock and SpinWait - для чего нужны, привести примеры, где они могут быть эффективны.

По ссылке достаточн подробно: http://www.albahari.com/threading/part5.aspx#\_SpinLock\_and\_SpinWait

Joseph Albahari ["Blocking vs Spinning"](http://www.albahari.com/threading/part2.aspx#_Blocking_Versus_Spinning), “[SpinLock and SpinWait](http://www.albahari.com/threading/part5.aspx" \l "_SpinLock_and_SpinWait)”.

1. **Thread safety**
   1. Какие способы используются для написания потокобезопасного кода?
2. Использование примитивов синхронизации (Monitor, Semaphor, Mutex, …)
3. Использование атомарных переменных (Класс Interlocked)
4. Использование потокобезопасных типов .NET.
5. Писать однопоточный код =)
   1. Как реализовать потокобезопасный Singleton?



Joseph Albahari ["Thread safety"](http://www.albahari.com/threading/part2.aspx#_Thread_Safety)

[“Thread safe singleton” Design Pattern](https://medium.com/peaceful-programmer/thread-safe-singleton-design-pattern-in-net-c-utilizing-the-c-properties-efficiently-25221fcd3b92).

1. **Event wait handles**
   1. Что такое signaling в многопоточности?

**Сигнализация** – примитив синхронизации, позволяющий потоку блокироваться вплоть до получения одного или большего числа уведомлений от другого потока (потоков).

В .NET: ManualResetEvent (ManualResetEventSlim), AutoResetEvent, CountdownEvent и Barrier. Первые три называются дескрипторами ожидания событий.

* 1. Отличие между WaitAny, WaitAll and SignalAndWait.

Методы **WaitAny, WaitAll и SignalAndWait** (все статические метода WaitHandle) выполняют операции сигнализирования и ожидания на множестве дескрипторов. Дескрипторы ожидания могут быть разных типов (в том числе Mutex и Sernphore, поскольку они также являются производными от абстрактного класса WaitHandle).

Метод **WaitHandle.WaitAny** ожидает любого дескриптора ожидания из массива таких дескрипторов; Метод **WaitHandle.WaitAll** ожидает все указанные дескрипторы атомарным образом. Это означает, что в случае ожидания двух объектов AutoResetEvent:

• метод WaitAny никогда не закончится "защелкиванием" обоих событий;

• метод Wai tAll никогда не закончится "защелкиванием" только одного события.

Метод SignalAndWait вызывает Set на WaitHandle и затем WaitOne на другом WaitHandle. После сигнализирования первого дескриптора произойдет переход в начало очереди в ожидании второго дескриптора; это помогает ему двигаться вперед.

* 1. ManualResetEvent vs AutoResetEvent.

Применение для **ManualResetEvent**: Поток, ожидающий некоторое событие, вызывает метод WaitOne() для событийного объекта, представляющего данное событие. Если событийный объект находится в сигнальном состоянии, то происходит немедленный возврат из метода WaitOne(). В противном случае выполнение вызывающего потока приостанавливается до тех пор, пока не будет получено уведомление о событии. Как только событие произойдет в другом потоке, этот поток установит событийный объект в сигнальное состояние, вызвав метод Set(). Поэтому метод Set() следует рассматривать как уведомляющий о том, что событие произошло.

После установки событийного объекта в сигнальное состояние произойдет немедленный возврат из метода WaitOne(), и первый поток возобновит свое выполнение. А в результате вызова метода Reset() событийный объект возвращается в несигнальное состояние.

Событие **AutoResetEvent** отличается от события типа ManualResetEvent лишь **способом установки в исходное состояние.** Если для события типа ManualResetEvent событийный объект остается в сигнальном состоянии до тех пор, пока не будет вызван метод Reset(), то для события типа AutoResetEvent событийный объект автоматически переходит в несигнальное состояние, как только поток, ожидающий это событие, получит уведомление о нем и возобновит свое выполнение. Поэтому если применяется событие типа AutoResetEvent, то вызывать метод Reset() необязательно.

* 1. Как много времени затрачивают signaling-конструкции на служебные операции (overhead)?

Порядка 1 микросекунда.   
Классы **ManualResetEventSlim** и **CountdownEvent** могут быть до 50 раз быстрее в сценариях с кратким ожиданием, поскольку они не зависят от операционной системы и благоразумно используют конструкции зацикливания.

* 1. Какие существуют механизмы для отправки сигналов между процессами (cross-process signaling)?

Создание именованного объекта **EventWaitHandle.**

Joseph Albahari ["Signaling with Event Wait Handles"](http://www.albahari.com/threading/part2.aspx#_Signaling_with_Event_Wait_Handles)

1. **Concurrent collections**
   1. Назовите разновидности ConcurrentCollections, в каких случаях следует применять тот или иной вид коллекции?

Microsoft Docs "[System.Collections.Concurrent Namespace](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent?view=netframework-4.8)".

Joseph Albahari “[Concurrent Collections](http://www.albahari.com/threading/part5.aspx#_Concurrent_Collections)”

1. **Parallelism**
   1. Что такое параллелизм? Какие средства предоставляет .Net Framework для параллельного выполнения кода?

**Различие понятий параллелизм и конкурентность:**

**Конкурентность (concurrency)** – парадигма программирования, предполагающая возможность одновременного запуска нескольких задач и что таким образом за определенный промежуток времени будет решено более одной задачи. При этом **ничего** не говориться о том, как конкурентность реализована: путем приостановки каких-то вычислительных элементов и их переключение на другую задачу или путем действительно одновременного исполнения (параллелизм) или еще как-то.

**Параллелизм – парадигма программирования, предполагающая** одновременное выполнение нескольких операций на нескольких вычислительных устройствах (например, процессорах).

**Параллельное исполнение программы – подмножество конкурентного исполнения!**

**Параллелизм в .NET:**

1. Библиотека параллельных задач (TPL).
2. Параллельный язык интегрированных запросов (PLINQ).
   1. Может ли параллельное исполнение кода быть менее эффективным, чем последовательное? Приведите примеры в подтверждение.

**Из msdn** (<https://docs.microsoft.com/ru-ru/dotnet/standard/parallel-programming/potential-pitfalls-in-data-and-task-parallelism#do-not-assume-that-parallel-is-always-faster>):  
**Не считайте, что параллельные процессы всегда быстрее.** В некоторых случаях параллельный цикл может выполняться медленнее, чем аналогичный последовательный. Первое правило состоит в том, что параллельные циклы с небольшим числом итераций и быстрыми пользовательскими делегатами, скорее всего, большого ускорения не дадут. Но в связи с тем, что на производительность влияет множество факторов, рекомендуем всегда оценивать фактические результаты.

**Примеры:** Перемножение матриц. Во-первых, для параллельного выполнения кода потребуется создание потоков. Метод ParallelFor использует потоки из ThreadPool, но нужные потоки требуется создать. Таким образом, если суть программы состоит в единичном вызове метода перемножения, то последовательный метод будет эффективнее, так как не придется создавать потоки.   
Во-вторых, последовательный код все еще будет эффективнее при малых размерностях матриц (до 20 на моем компьютере) за счет отсутствия накладных расходов на работу с потоками (не только создание).

* 1. Какую роль играет AggregateException в Parallel Programming?

Исключения, полученные в потоке, созданном с помощью TPL, перехватываются и повторно генерируются для основного потока. Поскольку параллельные библиотеки задействуют множество

потоков, вполне возможна одновременная генерация двух и более исключений. Чтобы обеспечить получение сведений обо всех исключениях, исключения помещаются в контейнер **AggregateException**, свойство **InnerExceptions** которого содержит каждое из перехваченных исключений.