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

Параллелизм на уровне задач. Библиотека параллельных задач (Task Parallel Library, TPL)
позволяет распараллелить задачи и выполнять их сразу на нескольких процессорах (для создания многопоточных приложений)
Задача (task) – абстракция более высокого уровня чем поток
using System.Threading.Tasks

2. Почему эффект от распараллеливания наблюдается на большом количестве элементов?
Эффект от распараллеливания проявляется при большом числе элементов, потому что накладные расходы на создание потоков и синхронизацию становятся незначительными. При малом числе элементов накладные расходы могут превышать выгоду от параллелизма.

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

Высокий уровень абстракции: Задачи проще в использовании, управление потоками не требуется.
Меньше накладных расходов: Задачи могут использовать пул потоков, экономя ресурсы.
Автоматическая синхронизация: Задачи упрощают управление синхронизацией.
Гибкость: Задачи лучше подходят для асинхронных операций и более эффективно используют ресурсы.

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

1)создание объекта Task и вызов у него метода Start:

2)использование статического метода Task.Factory.StartNew()

3)использование статического метода Task.Run():

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

`Wait()`	Приостанавливает текущий поток до завершения задачи
`WaitAll()`	Статический метод; приостанавливает текущий поток до заверш ния всех указанных задач
`WaitAny()`	Статический метод; приостанавливает текущий поток до заверш ния любой из указанных задач

6. Приведите пример синхронного запуска *Task*?

►По умолчанию задачи запускаются асинхронно

►RunSynchronously() - можно запускать синхронно

7. Как создать задачу с возвратом результата?

Чтобы создать задачу с возвратом результата, используется тип Task<T>, где T — это тип возвращаемого значения. Это позволяет создать асинхронную задачу, которая выполнит некоторую операцию и вернет результат в виде значения типа T.

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

Если при выполнении задачи возникает исключение, оно можно обработать, используя метод Task.Exception, который содержит исключения, возникшие в задаче. Можно также использовать try-catch блок, оборачивающий await для асинхронных задач или проверять свойства задачи, такие как IsFaulted, чтобы удостовериться, что задача завершена с ошибкой.

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

CancellationToken — это механизм, который позволяет отменить выполнение асинхронных или параллельных операций. Он используется в сочетании с CancellationTokenSource, который инициирует отмену. Во время выполнения задачи можно регулярно проверять CancellationToken.IsCancellationRequested для корректного завершения операции.

10. Как организовать задачу продолжения (continuation task) ?

Задачи продолжения создаются с помощью метода ContinueWith, который позволяет запланировать выполнение кода после завершения основной задачи. Задача продолжения выполняется в зависимости от состояния основной задачи (например, если основная задача завершилась успешно или с ошибкой).

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

Объект ожидания используется для синхронизации задач. Он позволяет блокировать выполнение до тех пор, пока одна или несколько задач не завершатся. Для этого можно использовать методы, такие как Task.WhenAll или Task.WhenAny, которые позволяют дождаться завершения одной или всех задач.

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

Класс Parallel предоставляет методы для параллельного выполнения операций, таких как циклы или наборы задач. Он позволяет выполнять код параллельно, распределяя работу между доступными потоками и абстрагируя управление многозадачностью.

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

Parallel.For используется для параллельного выполнения итераций цикла. Это позволяет ускорить выполнение циклов за счет распределения работы между несколькими потоками, каждый из которых выполняет отдельные итерации.

14. Приведите пример задачи с *Parallel.ForEach*

Parallel.ForEach используется для параллельной обработки элементов коллекции. Каждое выполнение действия на элементе коллекции происходит в отдельном потоке, что позволяет эффективно использовать многозадачность при обработке большого объема данных.

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

Parallel.Invoke используется для параллельного выполнения нескольких делегатов или методов. Все переданные в метод действия выполняются параллельно в разных потоках, что позволяет ускорить выполнение нескольких независимых операций.

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

В Parallel.For и Parallel.ForEach можно использовать CancellationToken, чтобы отменить выполнение параллельных операций. Для этого необходимо передать CancellationToken в соответствующие опции выполнения параллельных задач и периодически проверять флаг отмены

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

BlockingCollection<T> используется для организации потокобезопасной коллекции, которая поддерживает блокировку, когда коллекция пуста или переполнена. Это идеальный инструмент для реализации паттерна "производитель-потребитель", где один поток добавляет элементы в коллекцию, а другой извлекает их.

18. Как используя *async* и *await* организовать асинхронное выполенение метода

Ключевые слова async и await используются для упрощения написания асинхронного кода. async маркирует метод как асинхронный, а await приостанавливает выполнение метода до завершения асинхронной операции. Это позволяет выполнять операции без блокировки основного потока и улучшает производительность приложений.