1. Делегат (delegate) - это тип данных в C#, который представляет ссылку на метод. Делегаты позволяют передавать методы в качестве параметров, сохранять их в переменных, вызывать их асинхронно и создавать цепочки вызовов методов. Они являются важным механизмом для реализации событий, обратных вызовов и асинхронного программирования.
2. Делегаты в C# используются для представления и передачи методов как объектов. Они позволяют реализовать обратные вызовы, события и асинхронное программирование, делая код более гибким и расширяемым.
3. **Прямое присваивание метода:** Просто присвоить делегату метод напрямую, используя оператор присваивания.
4. **Использование анонимных методов:** Вы можете определить методы анонимно и назначить их делегату.
5. **Использование лямбда-выражений:** Лямбда-выражения предоставляют более краткую и удобную форму записи анонимных методов.
6. **Использование метода Delegate.CreateDelegate:** Этот метод позволяет создавать делегаты на основе методов по их именам и типам.
7. **Составные делегаты (Multicast Delegates):** Вы можете объединить несколько делегатов в один с помощью оператора **+=**. Это позволяет вызывать несколько методов при вызове делегата.

SomeMethod; myDelegate += AnotherMethod;

1. Метод **Invoke** используется для явного вызова делегата в C#. Этот метод может быть вызван на экземпляре делегата и предназначен для выполнения методов, на которые делегат ссылается.

Метод **Invoke** используется для явного вызова делегата в C#. Этот метод может быть вызван на экземпляре делегата и предназначен для выполнения методов, на которые делегат ссылается. Назначение метода **Invoke** включает в себя следующие аспекты:

1. **Выполнение делегата:** **Invoke** вызывает метод (или методы), на которые ссылается делегат. Это позволяет вам вызывать методы через делегат так же, как если бы вы вызывали их напрямую.
2. **Параметры и возвращаемое значение:** Вы можете передать параметры методу через **Invoke** и получить возвращаемое значение метода, если оно есть.
3. **Управление исключениями:** Использование **Invoke** позволяет ловить и обрабатывать исключения, которые могут возникнуть при выполнении метода через делегат.
4. Групповая адресация делегата (Multicast Delegates) представляет собой возможность объединения нескольких методов под одним делегатом так, что при вызове этого делегата будут вызваны все методы в порядке, в котором они были добавлены. Это позволяет создавать цепочки методов, которые будут вызываться последовательно.
5. Создание делегата: Определение делегата с сигнатурой, соответствующей методам, которые будут обрабатывать событие.
6. Создание события: Объявление события с использованием ключевого слова **event** и указанием делегата для события.
7. Вызов события: При возникновении события вызывается событие через проверку на **null** и вызов его. Все подписанные методы получают уведомление и выполняются в порядке подписки.
8. Подписка на событие: Другие части программы могут подписаться на событие, используя оператор **+=** и предоставив методы, соответствующие сигнатуре делегата.
9. Удаление подписки: Подписанные методы могут отписаться от события, используя оператор **-=**.
10. Лямбда-выражения в C# - это краткая и удобная форма для определения анонимных методов. Они позволяют создавать функции или делегаты на месте без явного определения имени метода.

Ковариантность и контравариантность делегатов - это особенности, представленные в C# 3.0 и выше, которые позволяют более гибко работать с делегатами, в частности, при работе с интерфейсами и обобщенными делегатами. Эти концепции позволяют расширить возможности и улучшить читаемость кода при работе с делегатами.

1. **Ковариантность делегатов**:
   * Ковариантный делегат позволяет вернуть более производный тип результата, чем был определен в оригинальном делегате.
   * Это означает, что вы можете использовать подтипы результата вместо базовых типов.
   * Пример: Вы можете использовать делегат, ожидающий возвращение **Animal**, с методом, который возвращает **Cat**.
2. **Контравариантность делегатов**:
   * Контравариантный делегат позволяет принимать параметры более базового типа, чем было определено в оригинальном делегате.
   * Это означает, что вы можете использовать супертипы параметров вместо более конкретных типов.
   * Пример: Вы можете использовать делегат, ожидающий **Func<Cat, string>**, с методом, принимающим **Func<Animal, string>**.
3. Встроенные делегаты **Action** и **Func** представляют собой обобщенные делегаты в C#, которые позволяют определить методы без явного создания пользовательских делегатов. Основная разница между ними заключается в том, что **Action** предназначен для методов, которые не возвращают значения, а **Func** для методов, которые возвращают значения.
4. **На какие основные виды/типы делятся все коллекции .NET? Охарактеризуйте каждый из них.**
   * **Необобщенные;**

- наличие разнотипных данных

- ссылки на данные типа object (не обеспечивают типовую безопасность)

- System.Collections

* + **Обобщенные;**

- обеспечивают обобщенную реализацию нескольких стандартных структур

данных, включая связные списки, стеки, очереди и словари

- типизированные в силу своего обобщенного характера (обеспечивают типовую безопасность)

- System.Collections.Generic

* + **Специальные;**

**-** оперируют данными конкретного типа или же делают это каким-то особым образом

- System.Collections.Specialized

* + **С поразрядной организацией (поразрядная коллекция);**

- BitArray

- System.Collections

* + **Параллельные;**

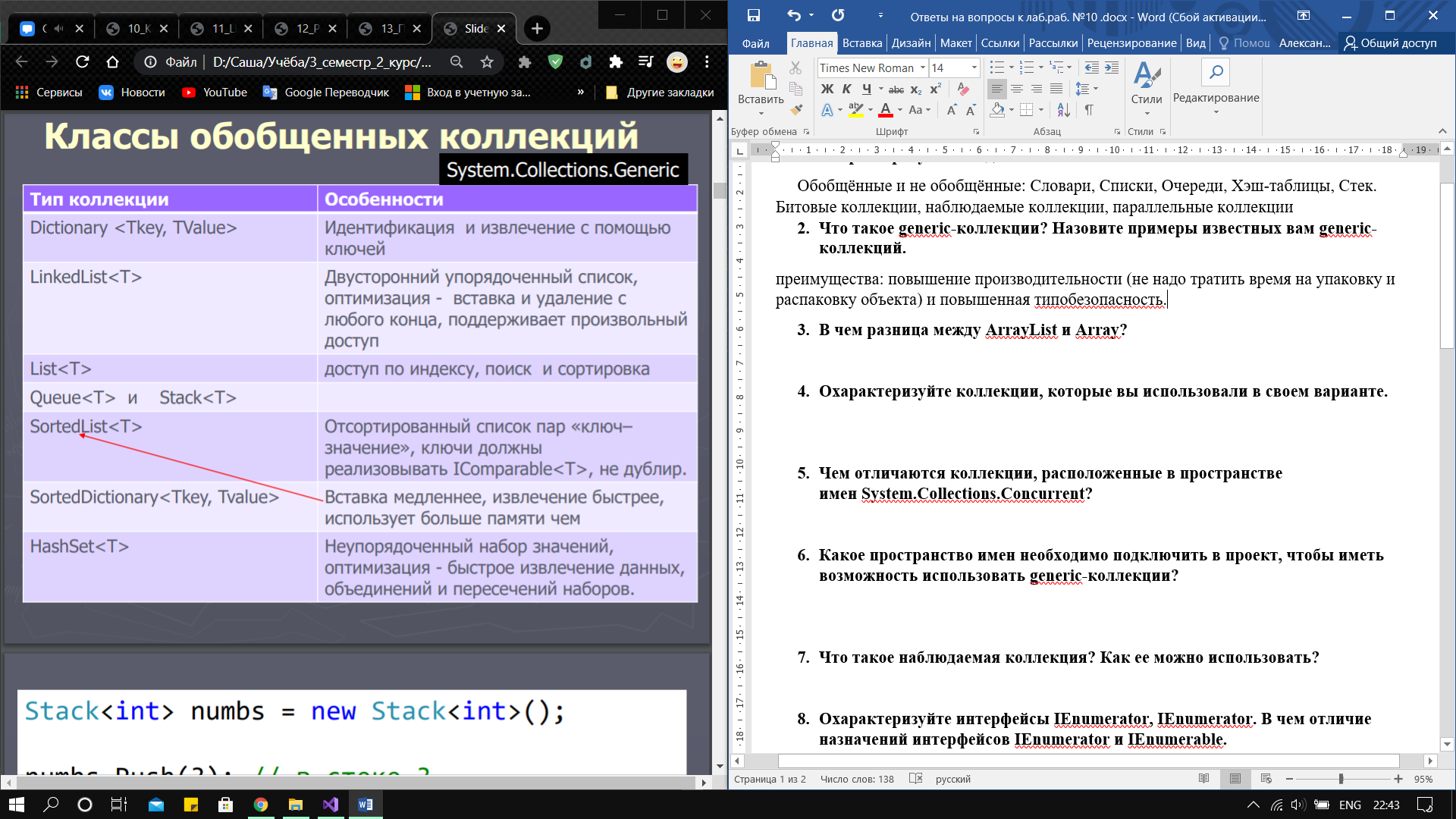
- многопоточный доступ к коллекции

- System.Collections.Concurrent

Обобщённые и необобщённые: Словари, Списки, Очереди, Хэш-таблицы, Стек.

1. **Что такое Generic-коллекции? Назовите примеры известных вам generic-коллекций.**

**Обобщённые коллекции** – это те же самые обобщённые классы (классы, которые изменяют своё поведение в зависимости от приписываемого им типа). Их использование перед необобщёнными коллекциями имеет те же преимущества: повышение производительности и повышенная типобезопасность.



1. **В чем разница между ArrayList и Array?**

Класс **ArrayList** определяется как массив переменной длины, который состоит из ссылок на объекты и может динамически увеличивать и уменьшать свой размер. Это список, каждый элемент связан друг с другом через указатели.

**Array** – это массив, представляющий из себя низкоуровневую структуру данных, которая по существу отображает область в памяти и имеет фиксированный размер.

1. **Охарактеризуйте коллекции, которые вы использовали в своем варианте.**

Еще один распространенный тип коллекции представляют словари. Словарь хранит объекты, которые представляют пару ключ-значение. Класс словаря **Dictionary<K, V>** типизируется двумя типами: параметр **K** представляет тип ключей, а параметр **V** предоставляет тип значений.

* **void Add(K key, V value)**: добавляет новый элемент в словарь
* **void Clear()**: очищает словарь
* **bool ContainsKey(K key)**: проверяет наличие элемента с определенным ключом и возвращает true при его наличии в словаре
* **bool ContainsValue(V value)**: проверяет наличие элемента с определенным значением и возвращает true при его наличии в словаре
* **bool Remove(K key)**: удаляет по ключу элемент из словаря

Другая версия этого метода позволяет получить удленный элемент в выходной параметр: **bool Remove(K key, out V value)**

* **bool TryGetValue(K key, out V value)**: получает из словаря элемент по ключу key. При успешном получении передает значение элемента в выходной параметр value и возвращает true
* **bool TryAdd(K key, V value)**: добавляет в словарь элемент с ключом key и значением value. При успешном добавлении возвращает true

Класс List<T> из пространства имен System.Collections.Generic представляет простейший список однотипных объектов. Класс List типизируется типом, объекты которого будут хранится в списке.

* **void Add(T item)**: добавление нового элемента в список
* **void AddRange(IEnumerable<T> collection)**: добавление в список коллекции или массива
* **int BinarySearch(T item)**: бинарный поиск элемента в списке. Если элемент найден, то метод возвращает индекс этого элемента в коллекции. При этом список должен быть отсортирован.
* **void CopyTo(T[] array)**: копирует список в массив array
* **void CopyTo(int index, T[] array, int arrayIndex, int count)**: копирует из списка начиная с индекса index элементы, количество которых равно count, и вставляет их в массив array начиная с индекса arrayIndex
* **bool Contains(T item)**: возвращает true, если элемент item есть в списке
* **void Clear()**: удаляет из списка все элементы

1. **Чем отличаются коллекции, расположенные в пространстве имен System.Collections.Concurrent?**

**Параллельные коллекции** - коллекции классов, предназначенные для безопасной работы в многопоточной среде, которыми можно воспользоваться при создании многопоточных приложений.

1. **Какое пространство имен необходимо подключить в проект, чтобы иметь возможность использовать generic-коллекции?**

System.Collections.Generic

1. **Что такое наблюдаемая коллекция? Как ее можно использовать?**

**ObservableCollection<T>**

В случае если нужна информация о том, когда элементы коллекции удаляются или добавляются, можно использовать класс ObservableCollection<T>. Этот класс предназначен для того, чтобы пользовательский интерфейс мог получать информацию об изменениях коллекции. Пространство имен этого класса — System.Collections.ObjectModel.

Класс ObservableCollection<T> унаследован от базового класса Collection<T>, который может применяться для создания специальных коллекций; он использует внутри себя List<T>. Методы базового класса SetItem() и RemoveItem() переопределены для инициации события CollectionChanged.

var data = new ObservableCollection<string>();

data.CollectionChanged += Data\_CollectionChanged;

data.Add("One");

data.Add("Two");

data.Insert(1, "Three");

data.Remove("One");

1. **Охарактеризуйте интерфейсы IEnumerator, IEnumerator. В чем отличие назначений интерфейсов IEnumerator и IEnumerable.**

Интерфейс ***IEnumerable*** имеет метод GetEnumerator(), возвращающий ссылку на другой интерфейс IEnumerator – перечислитель. Коллекция, которая реализует данный интерфейс, может использоваться с оператором foreach.

А интерфейс ***IEnumerator*** определяет функционал для перебора внутренних объектов в контейнере. Этот интерфейс может перечислять: имеет свойство Current и методы MoveNext() и Reset().

public interface IEnumerator

{

    bool MoveNext(); // перемещение на одну позицию вперед в контейнере элементов

    object Current {get;}  // текущий элемент в контейнере

    void Reset(); // перемещение в начало контейнера

}

1. **Поясните принцип работы коллекций:**
2. **LinkedList <T> -** Сохраняет элементы в двунаправленном списке.
3. **HashSet <T> -** Сохраняет ряд уникальных значений, используя хеш-таблицу.
4. **Dictionary <Tkey, TValue> -** Сохраняет пары "ключ-значение". Обеспечивает такие же функциональные возможности, как и необобщенный класс Hashtable.
5. **ConcurrentBag <Tkey, TValue> -** Этот класс не определяет никакого порядка для добавления или извлечения элементов. Он реализует концепцию отображения потоков на используемые внутренние массивы. Для доступа к элементам применяются методы Add(), TryPeek() и TryTake().
6. **Stack<t>, Queue<T> -** Создает стек; обеспечивает такие же функциональные возможности, как и необобщенный класс Stack.

- Создает очередь. Обеспечивает такие же функциональные возможности, как и необобщенный класс Queue.

1. **SortedList -** хранит наборы пар ключ-значение, отсортированные по ключу.