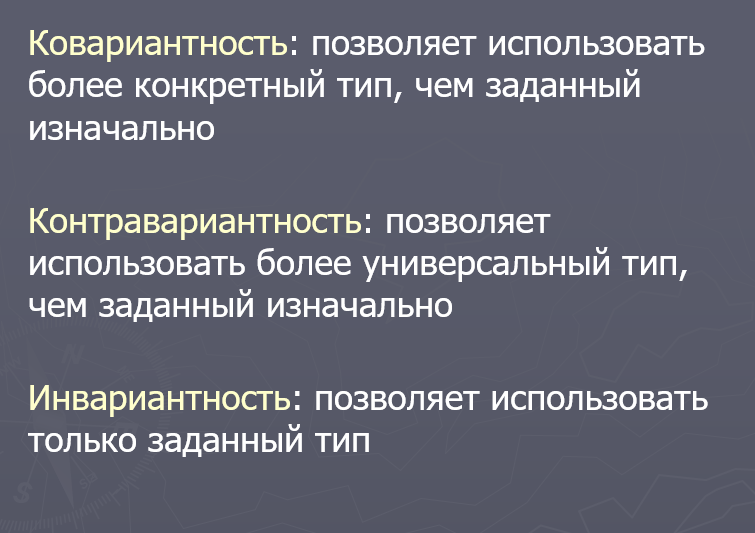
**31. Ковариантность и контравариантность интерфейсов**

****

**Ковариантность**

Ковариантность позволяет использовать более конкретные типы в месте ожидания более общего. Это важно, когда вы работаете с интерфейсами, которые возвращают данные. Например, если у вас есть интерфейс, который возвращает объекты типа Animal, вы можете использовать его для возвращения объектов типа Dog.

**Пример:**

public interface ICovariant<out T>

{

T GetValue();

}

// Реализация с конкретным типом

public class CovariantClass : ICovariant<Dog>

{

public Dog GetValue() => new Dog();

}

**Контравариантность**

Контравариантность позволяет использовать более общие типы в месте ожидания более конкретных. Это полезно в ситуациях, когда вы передаете данные. Например, если у вас есть интерфейс, который принимает объекты типа Animal, вы можете использовать его для передачи объектов типа Dog.

**Пример:**

public interface IContravariant<in T>

{

void SetValue(T value);

}

// Реализация с более общим типом

public class ContravariantClass : IContravariant<Animal>

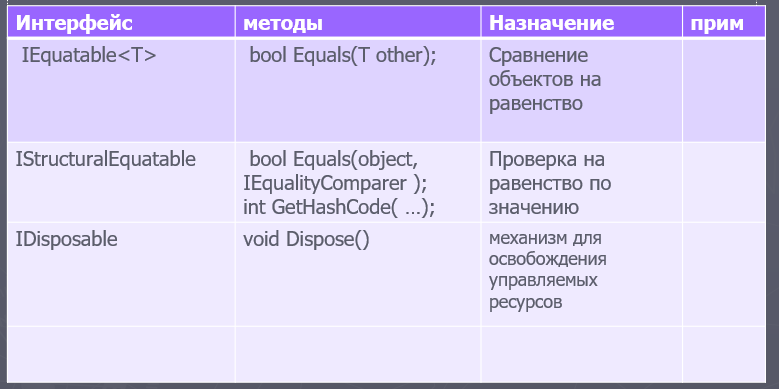
{

public void SetValue(Animal value) { /\* реализация \*/ }

}

**32. Стандартные интерфейсы .NET**

****

****

**Назначение и применение**

Стандартные интерфейсы .NET предоставляют единообразный способ работы с коллекциями и другими структурами данных. Они обеспечивают гибкость и возможность использования различных типов данных без необходимости менять код.

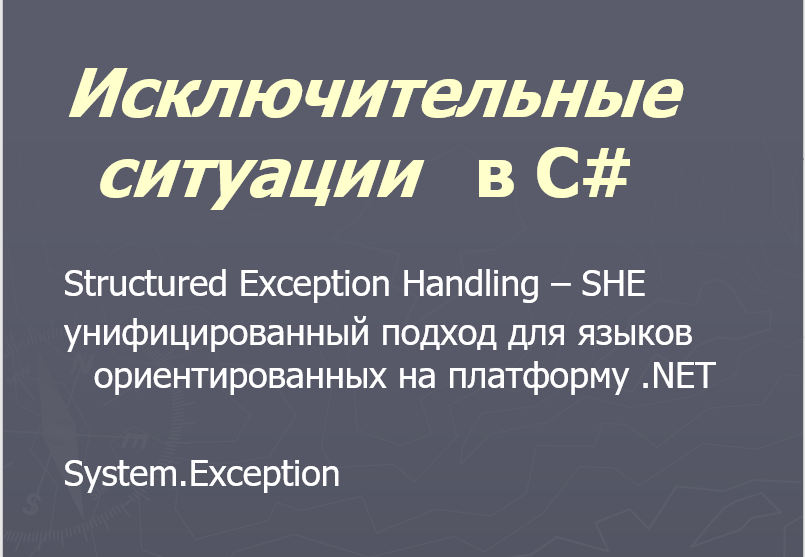
* **IEnumerable**: Позволяет перечислять элементы коллекции.
* **ICollection**: Расширяет IEnumerable, добавляя методы для управления коллекцией, такие как добавление и удаление элементов.
* **IList**: Расширяет ICollection, предоставляя возможность индексировать элементы.
* **IDictionary**: Определяет коллекцию пар "ключ-значение", позволяя эффективный доступ к элементам по ключу.

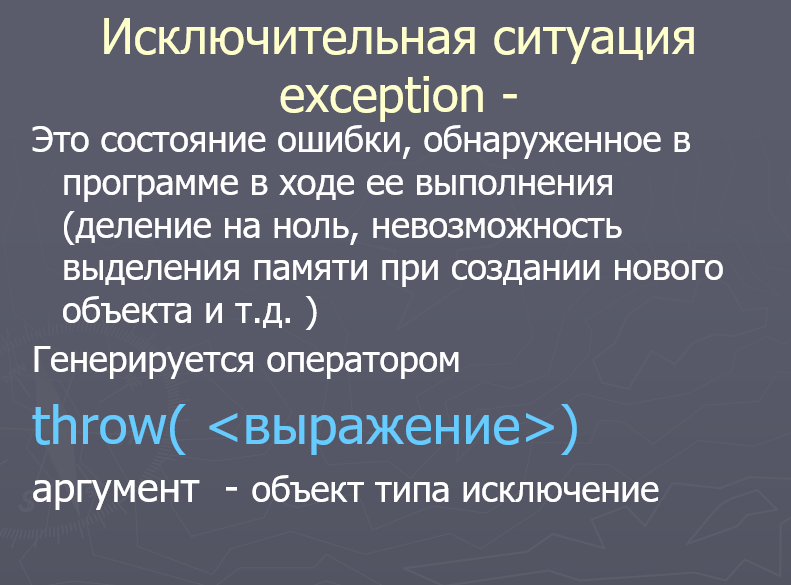
**Пример:**

List<string> list = new List<string> { "one", "two", "three" };

IEnumerable<string> enumerable = list; // Работает с IEnumerable

**33. Исключительные ситуации**

****

****

**Генерация и повторная генерация исключений**

Исключения в .NET позволяют обрабатывать ошибки и исключительные ситуации. Когда возникает ошибка, вы можете сгенерировать исключение с помощью оператора throw. Если необходимо, вы можете повторно сгенерировать исключение, используя throw;, что сохраняет исходное исключение.

**Пример:**

try

{

throw new Exception("Ошибка!");

}

catch (Exception ex)

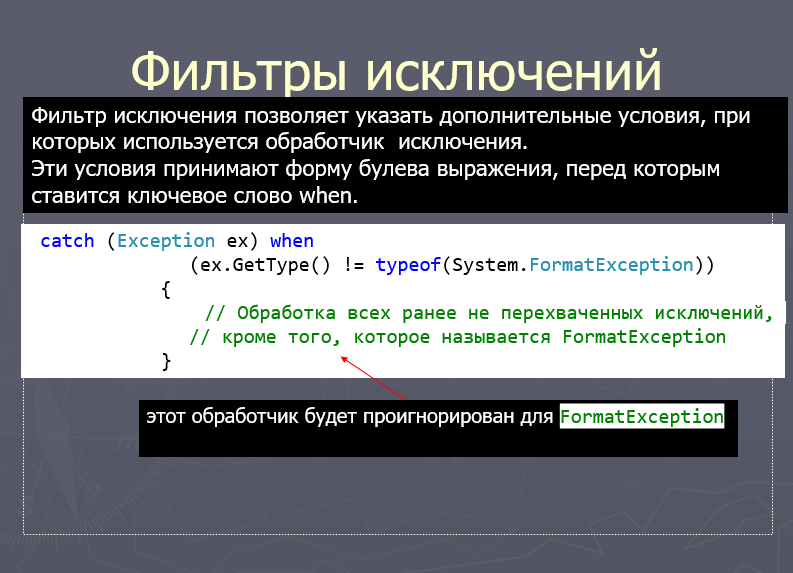
{

Console.WriteLine(ex.Message);

throw; // Повторная генерация исключения

}

**34. Варианты обработки исключений. Фильтры исключений**

****

.NET предоставляет механизмы для обработки исключений с помощью блоков try, catch и finally. Фильтры исключений позволяют задавать условия для обработки, что делает код более гибким и читаемым.

**Пример:**

try

{

// Код, который может вызвать исключение

}

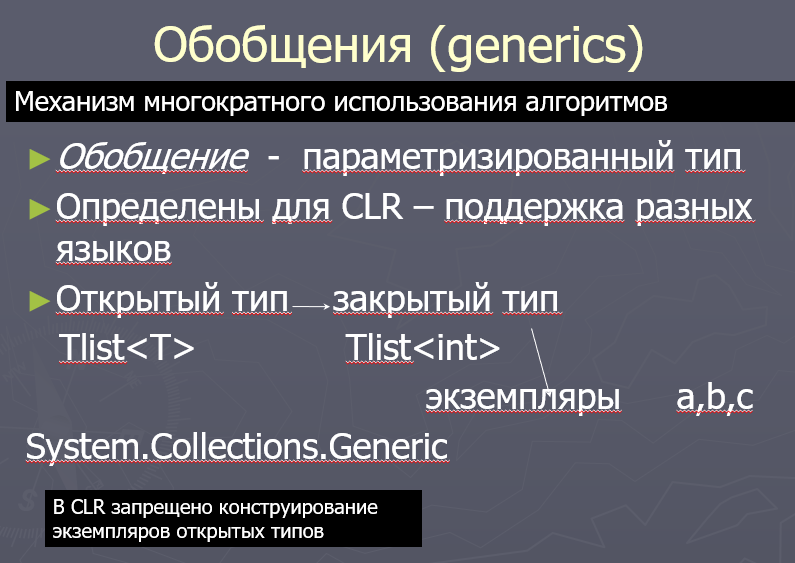
catch (Exception ex) when (ex.Message.Contains("неправильно"))

{

Console.WriteLine("Обработано специфическое исключение.");

}

**35. Обобщения (generics)**

****

Обобщения позволяют создавать классы, интерфейсы и методы, которые могут работать с любыми типами данных. Это повышает повторное использование кода и обеспечивает типобезопасность, избегая использования object.

**Пример:**

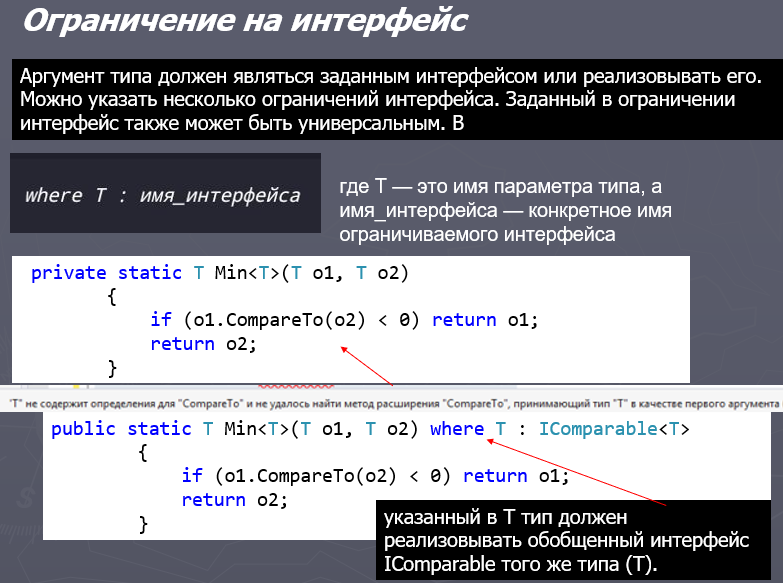
public class GenericClass<T>

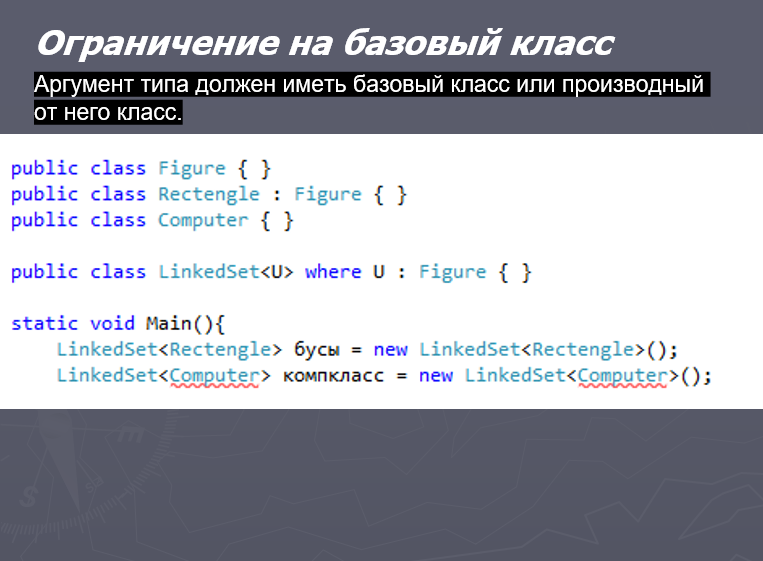
{

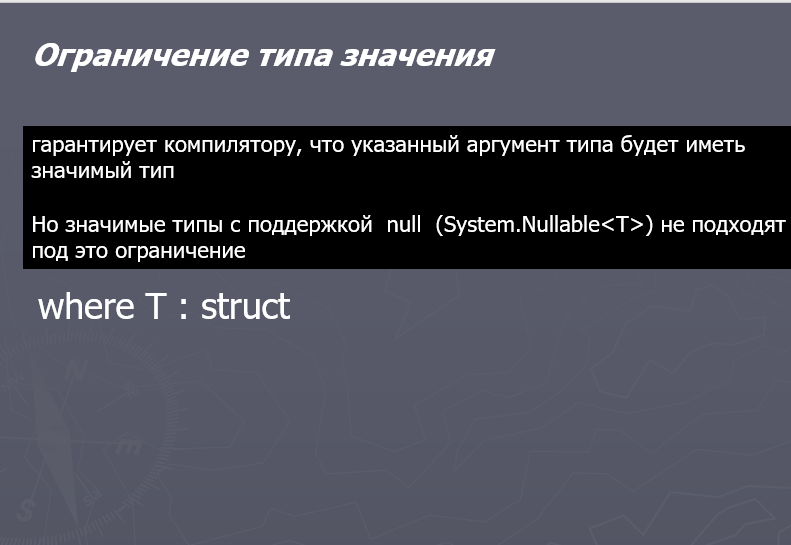
private T data;

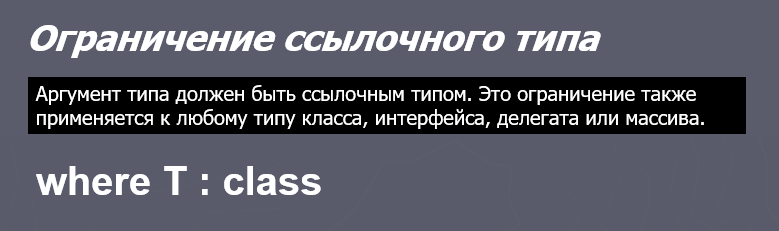
public GenericClass(T value) { data = value; }

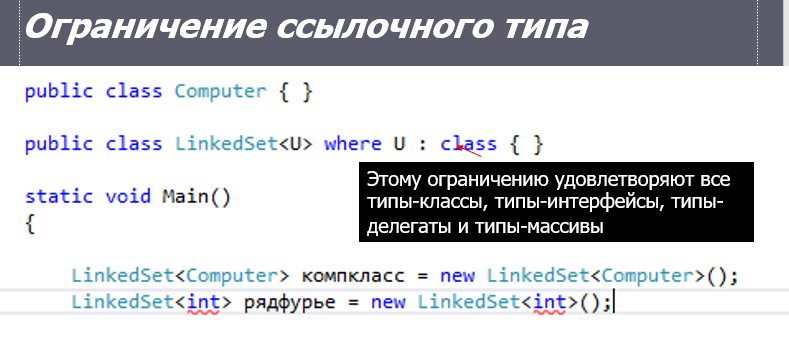
}

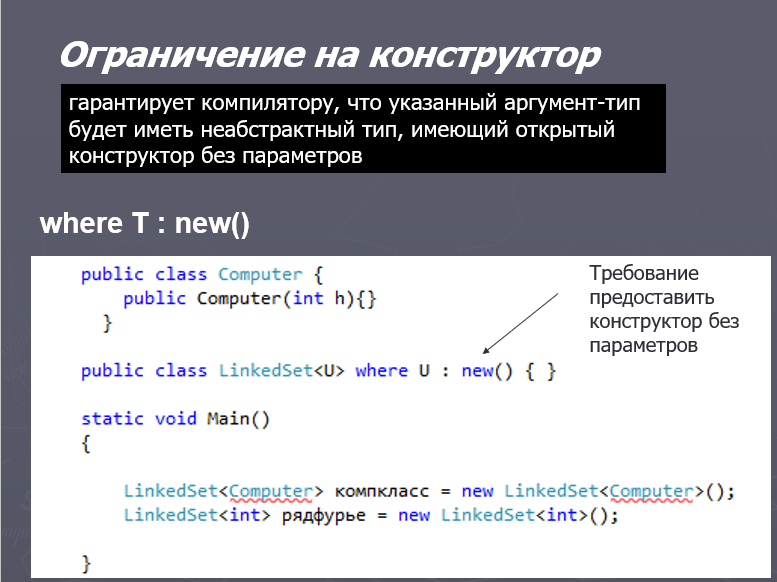
**36. **

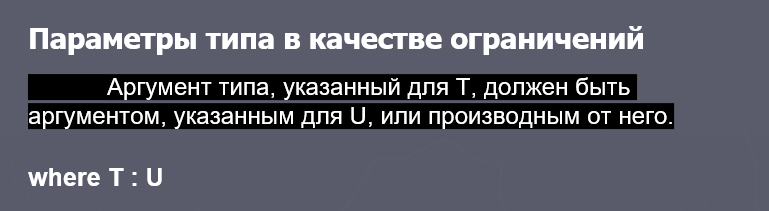
****

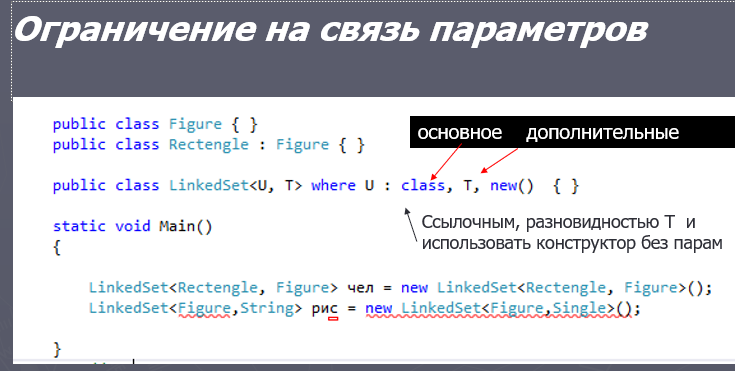
****

****

****

****

****

**Концепция ограничений обобщений**

Ограничения обобщений позволяют задавать условия для типов, которые могут быть использованы в обобщениях. Это гарантирует, что передаваемые типы соответствуют определённым требованиям.

**Пример:**

public class GenericClass<T> where T : class

{

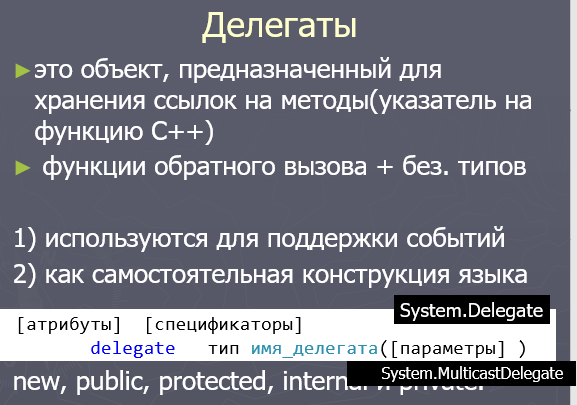
// Только ссылочные типы могут быть использованы

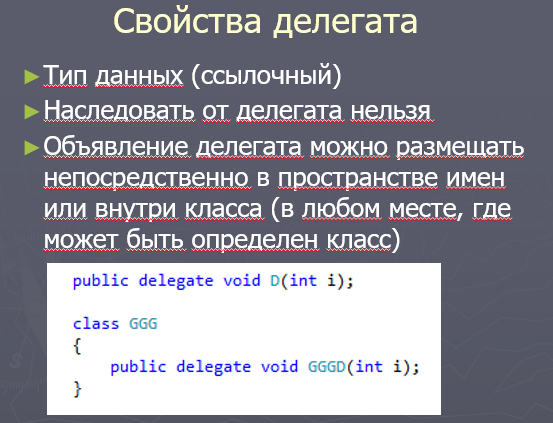
}

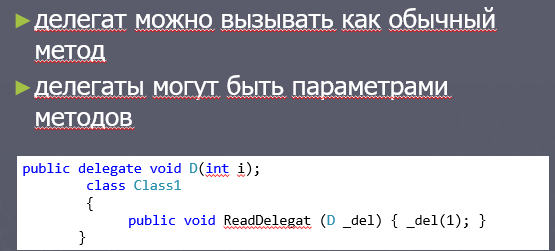
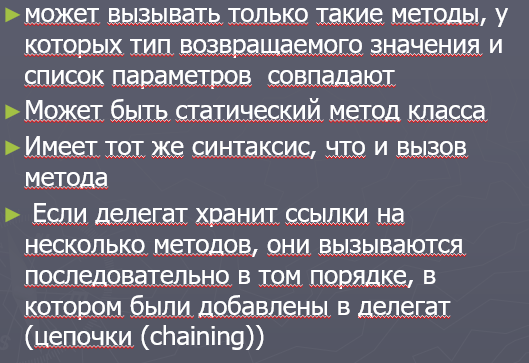
**Статические члены обобщений**

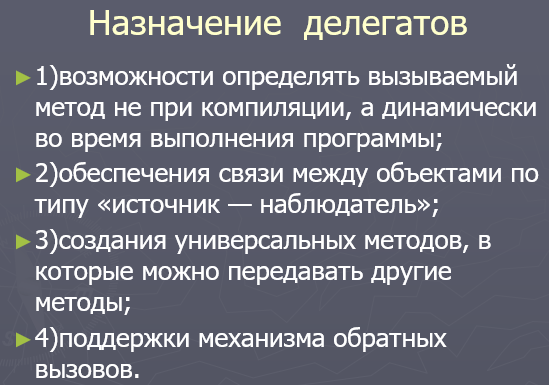
Обобщенные классы могут содержать статические члены, однако они должны быть определены с использованием ограничений.

**37. Делегаты**

****

****

****

****

Делегаты — это типы, которые представляют ссылки на методы. Они обеспечивают возможность передачи методов как параметров, что делает код более гибким и модульным. Делегаты могут вызывать методы с определённой сигнатурой.

**Пример:**

public delegate void MyDelegate(string message);

public class Program

{

public static void PrintMessage(string message) => Console.WriteLine(message);

static void Main()

{

MyDelegate del = PrintMessage;

del("Hello, World!");

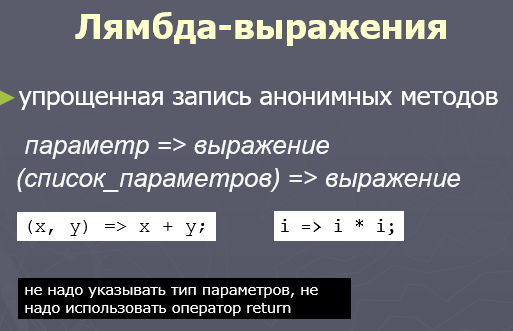
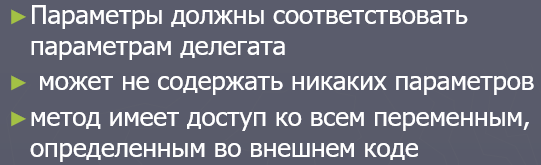
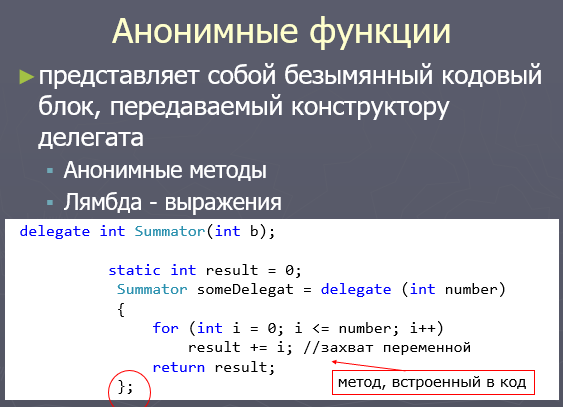
}

}

**Обобщенные делегаты**

.NET предоставляет обобщенные делегаты, такие как Action, Func и Predicate, которые упрощают работу с делегатами.

**38. Анонимные функции и лямбда-выражения**

****

Анонимные функции — это функции без имени, которые могут быть определены на месте. Лямбда-выражения представляют собой более лаконичный синтаксис для создания анонимных функций.

**Пример:**

Func<int, int> square = x => x \* x;

Console.WriteLine(square(5)); // 25

**39. Обобщённые делегаты .NET**

**Action, Func, Predicate**

* **Action**: Делегат, представляющий метод, который не возвращает значение. Например, Action<string> принимает строку и ничего не возвращает.
* **Func**: Делегат, представляющий метод, который возвращает значение. Например, Func<int, int> принимает целое число и возвращает целое число.
* **Predicate**: Делегат, представляющий метод, который принимает один аргумент и возвращает bool. Используется для фильтрации.

**Пример:**

Action<string> greet = name => Console.WriteLine($"Hello, {name}!");

greet("Alice");

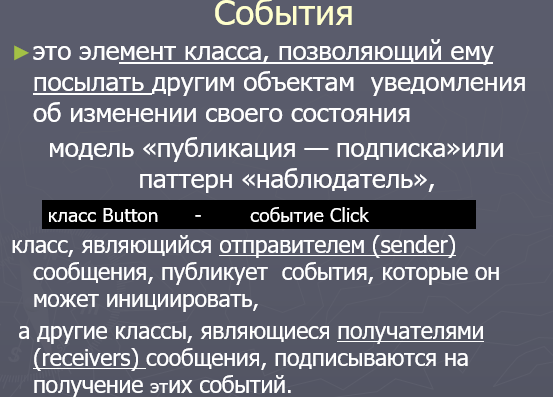
Func<int, int> add = (x, y) => x + y;

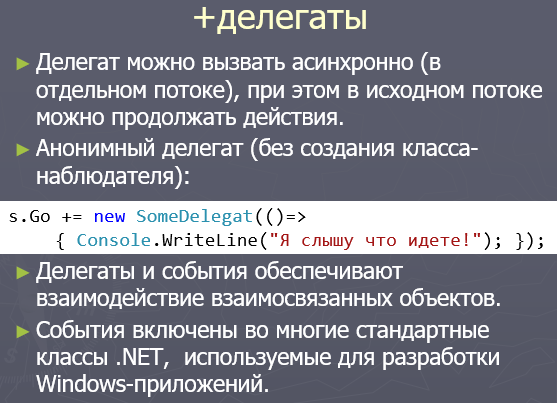
Console.WriteLine(add(2, 3)); // 5

Predicate<int> isEven = x => x % 2 == 0;

Console.WriteLine(isEven(4)); // true

**40. События и делегаты**

****

****

События используются для уведомления подписчиков об изменениях состояния. Они основаны на делегатах и позволяют создавать гибкие и расширяемые архитектуры.

**Пример:**

public class EventPublisher

{

public event Action OnEvent;

public void TriggerEvent()

{

OnEvent?.Invoke();

}

}

public class Subscriber

{

public void Subscribe(EventPublisher publisher)

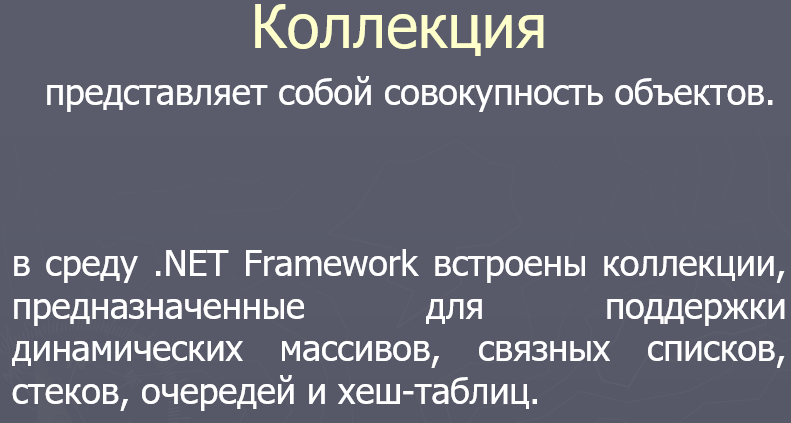
{

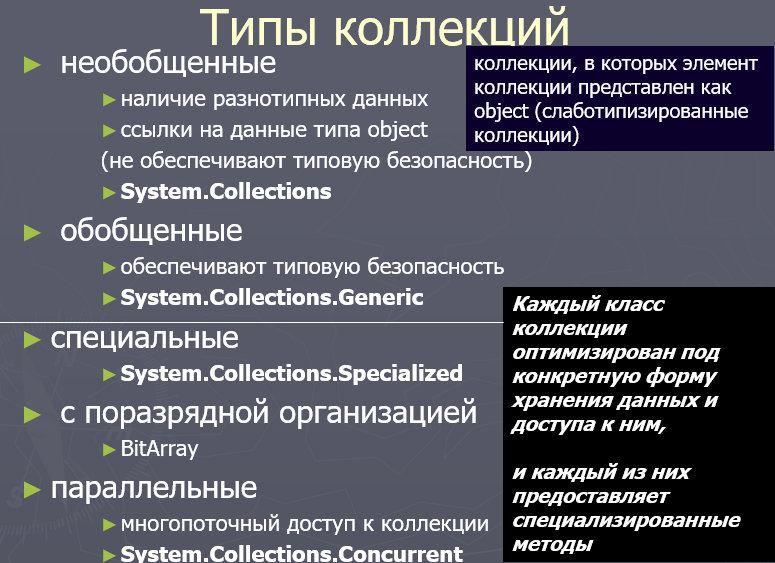
publisher.OnEvent += () => Console.WriteLine("Событие произошло!");

}

}

**41. Стандартные коллекции .NET**

****

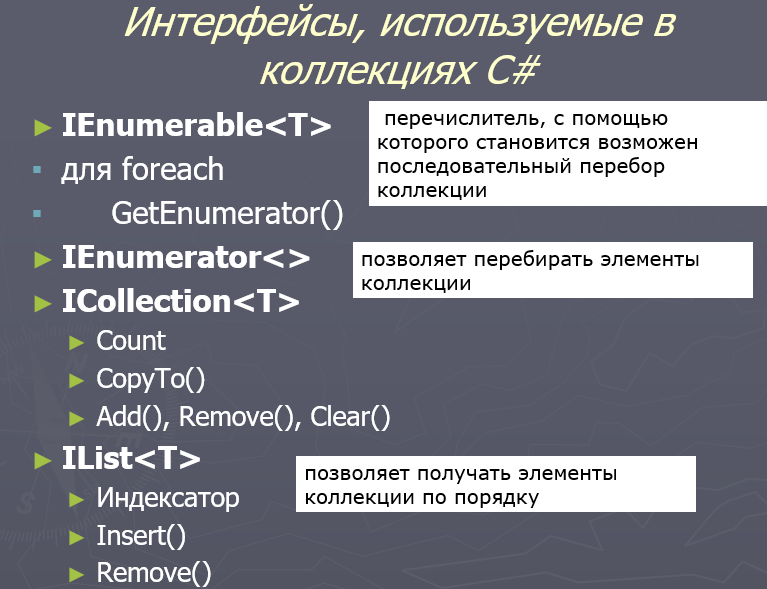
****

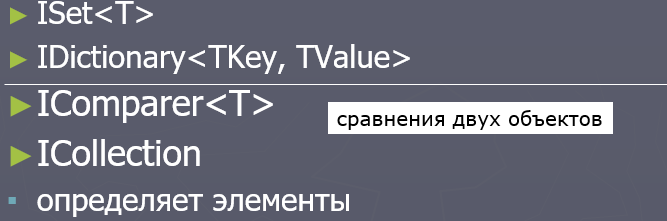
**Типы коллекций**

Стандартные коллекции .NET включают:

* **Array**: Фиксированный размер, быстрый доступ по индексу.
* **List<T>**: Динамический массив, поддерживающий добавление и удаление элементов.
* **Dictionary<TKey, TValue>**: Коллекция пар "ключ-значение", обеспечивающая быстрый доступ по ключу.
* **HashSet<T>**: Коллекция уникальных элементов, не допускающая дубликатов.

**42. Стандартные интерфейсы коллекций**

****

****

Стандартные интерфейсы коллекций в .NET включают:

* **IEnumerable<T>**: Позволяет перечислять коллекцию.
* **ICollection<T>**: Расширяет IEnumerable, добавляя методы для управления коллекцией.
* **IList<T>**: Расширяет ICollection, добавляя возможность индексирования.
* **IDictionary<TKey, TValue>**: Определяет коллекцию пар "ключ-значение".

**43. IEnumerable и IEnumerator**

* **IEnumerable**: Интерфейс, который определяет метод для получения итератора(Итератор — это объект, который позволяет последовательно перебрать элементы коллекции без необходимости раскрывать внутреннюю реализацию этой коллекции.).
* **IEnumerator**: Интерфейс, который позволяет перебирать коллекцию. Он предоставляет методы для перемещения по коллекции и получения текущего элемента.

**Пример:**

List<int> numbers = new List<int> { 1, 2, 3 };

IEnumerator<int> enumerator = numbers.GetEnumerator();

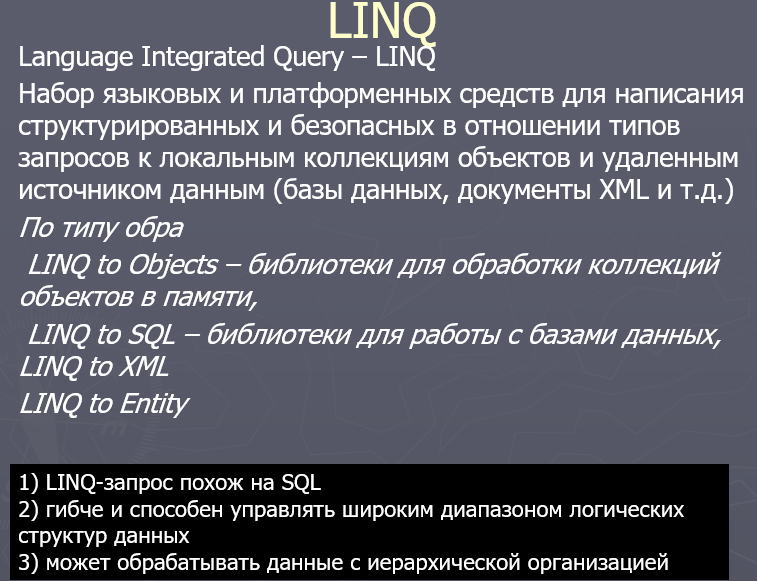
while (enumerator.MoveNext())

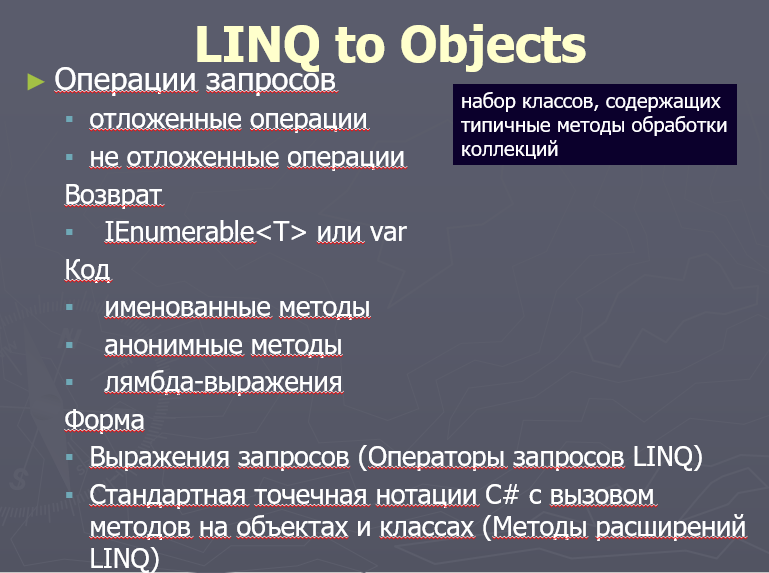
{

Console.WriteLine(enumerator.Current);

}

**44. LINQ to Objects**

****

****

**Синтаксис**

LINQ (Language Integrated Query) позволяет выполнять запросы к коллекциям .NET. Он обеспечивает простой и интуитивно понятный синтаксис для работы с данными.

**Пример:**

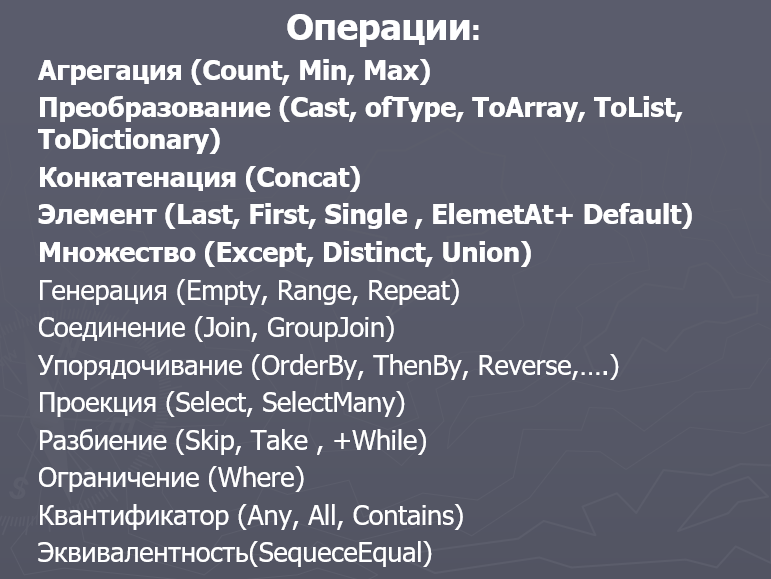
var numbers = new List<int> { 1, 2, 3, 4, 5 };

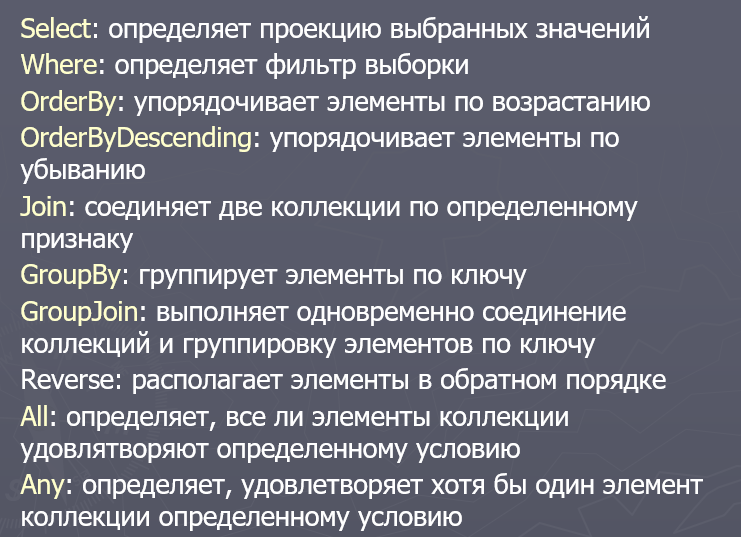
var evenNumbers = from n in numbers where n % 2 == 0 select n;

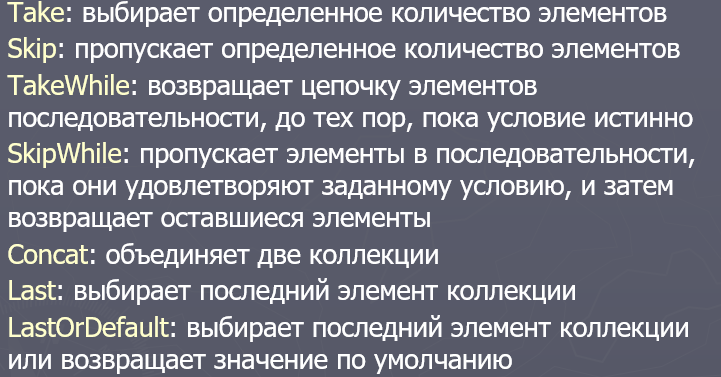
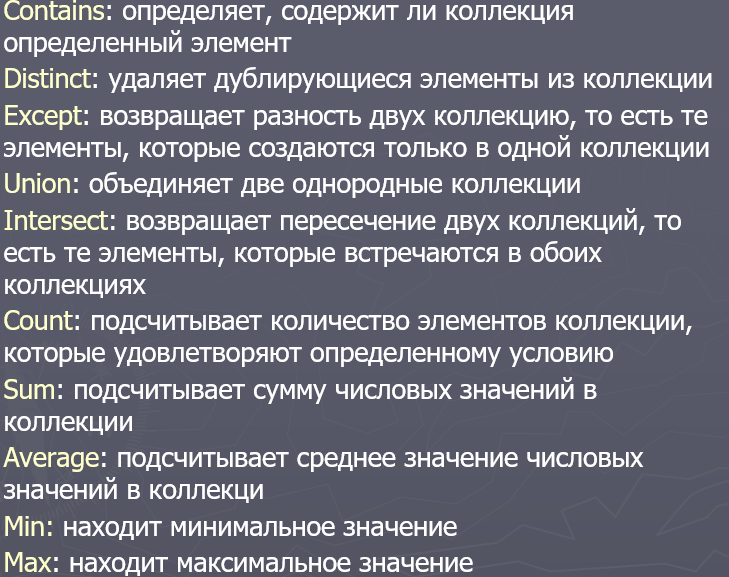
**Отложенные и неотложенные операции**

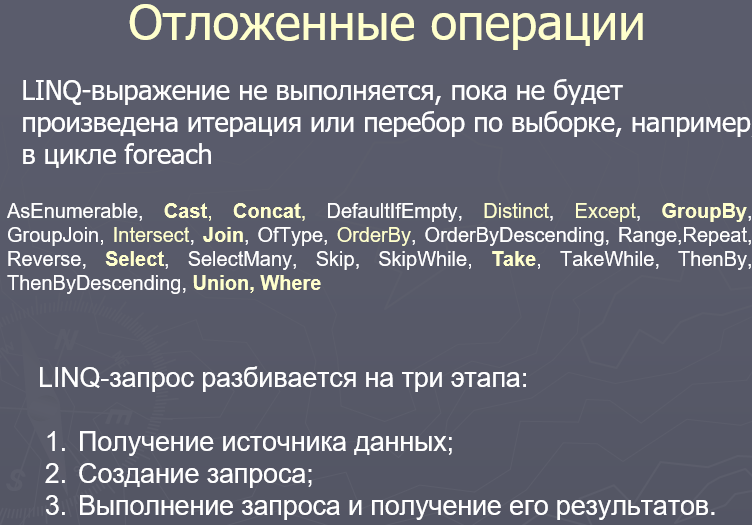
Отложенные операции выполняются только при необходимости, что позволяет оптимизировать использование ресурсов. Неотложенные операции выполняются сразу, возвращая результаты немедленно.

**45. LINQ to Objects. Операции**

****

****

****

****

**Операции Where, Select, Take, OrderBy, Join, GroupBy**

* **Where**: Фильтрует коллекцию по заданному условию.
* **Select**: Преобразует элементы коллекции, создавая новую коллекцию.
* **Take**: Возвращает заданное количество элементов из начала коллекции.
* **OrderBy**: Сортирует коллекцию по заданному критерию.
* **Join**: Объединяет две коллекции по ключу.
* **GroupBy**: Группирует элементы коллекции по заданному ключу.

**Пример:**

var evenNumbers = numbers.Where(n => n % 2 == 0).ToList();

var squaredNumbers = numbers.Select(n => n \* n).ToList();

var firstThree = numbers.Take(3).ToList();

var sortedNumbers = numbers.OrderBy(n => n).ToList();