**Лаба 2**

1. Принципы ООП - наследование, полиморфизм, инкапсуляция. Наследование позволяет создавать новые классы на основе уже существующих, полиморфизм - использование одного и того же имени метода для разных классов, инкапсуляция - скрытие данных и методов внутри класса и предоставление доступа к ним только через определенные методы.

2. Класс .NET, от которого наследуются все классы - System.Object.

3. Открытые методы System.Object - Equals(), GetHashCode(), GetType(), ToString().

4. Закрытые методы System.Object - Finalize(), MemberwiseClone().

5. Пример определения класса:

public class MyClass {

private int myField;

public int MyProperty {

get { return myField; }

set { myField = value; }

}

public void MyMethod() {

//some code here

}

}

6. Ключевые слова при определении класса - public, private, protected, internal, static, abstract, sealed.

7. Объект - это экземпляр класса, созданный в памяти компьютера. Класс - это шаблон для создания объектов, содержащий описание его свойств и методов.

8. Конструктор - это специальный метод класса, который вызывается при создании нового объекта этого класса. Конструктор может принимать параметры или не принимать их вовсе.

9. Свойства конструктора - имя, параметры, тело метода.

10. Деструктор - это метод класса, который вызывается при уничтожении объекта этого класса. Деструктор не принимает параметров и не возвращает значения.

11. This - это ключевое слово, которое ссылается на текущий объект класса.

12. Результат выполнения программы: 5 5, 7 7. Переменная b ссылается на тот же объект, что и переменная a, поэтому изменение свойства Num у переменной a также отразится на переменной b.

13. Спецификаторы доступа для класса и методов класса в C# - public, private, protected, internal, protected internal.

14. Модификатор protected internal позволяет обращаться к членам класса из других классов в той же сборке или из производных классов в других сборках.

15. Ref и out параметры функции используются для передачи значений по ссылке. Ref используется для передачи уже инициализированной переменной, out - для передачи переменной, которая будет инициализирована внутри функции.

16. Пример необязательных и именованных параметров метода:

public void MyMethod(int a = 0, int b = 0) {

//some code here

}

public void MyMethod(int a, int b = 0, int c = 0) {

//some code here

}

17. Примеры полей класса - статические (static), константные (const), только для чтения (readonly).

18. Пример определения свойств класса:

public class MyClass {

private int myField;

public int MyProperty {

get { return myField; }

set { myField = value; }

}

}

Свойства связаны с инкапсуляцией, так как позволяют скрыть данные от прямого доступа и предоставить к ним доступ только через методы.

19. Явное имя параметра, передаваемого в метод set свойства класса - value.

20. Автоматические свойства - это свойства, которые не имеют явно определенного поля в классе, а хранятся внутри компилятора. Они определяются с помощью ключевого слова "prop" и могут иметь только геттеры и/или сеттеры.

21. Индексаторы класса - это методы, которые позволяют получать или устанавливать значения элементов массива или коллекции по индексу. Ограничения на индексатор - он должен иметь хотя бы один параметр, тип которого должен быть целочисленным или строковым.

22. Перегруженный метод - это метод с тем же именем, что и другой метод в классе, но с разными параметрами.

23. Частичный класс - это класс, который может быть разделен на несколько файлов. Преимущества частичных классов - удобство разделения больших классов на несколько файлов, возможность распределения работы между несколькими программистами.

24. Анонимный тип - это тип данных, который создается во время выполнения программы и не имеет явного имени. Он используется для хранения временных данных или для передачи данных между методами.

25. Статические классы используются для хранения методов и свойств, которые не требуют создания объекта этого класса.

26. Статическое поле принадлежит всем объектам класса, а экземплярное - только одному объекту это го класса.

27. Статический конструктор вызывается только один раз при первом обращении к статическому члену класса. Он используется для инициализации статических полей или выполнения других действий, которые требуются только один раз.

28. При поверхностном копировании копируются только ссылки на объекты, а не сами объекты. При глубоком копировании копируются все объекты и их содержимое.

29. Равенство объектов проверяет, равны ли ссылки на объекты, тождество объектов проверяет, равны ли значения объектов.

30. Частичные классы и методы используются для разделения кода класса на несколько файлов или для разделения кода метода на несколько частей.

**Лаба 3**

1. Перечислите свойства внутренних и вложенных классов?

- Set:

- elements (внутреннее поле)

- Production (вложенный класс) - Id, OrganizationName

- Developer (вложенный класс) - FullName, Id, Department

2. Что такое статический класс и какие у него свойства?

- Статический класс в C# - это класс, который не может быть инстанциирован, и его члены (методы, поля, свойства) могут вызываться или использоваться без создания объекта этого класса. Статический класс может содержать только статические члены и не может иметь экземпляров. Он используется, например, для создания утилитных классов или хранения глобальных данных.

3. Каково назначение перегрузки операторов?

- Перегрузка операторов позволяет определить пользовательское поведение для операторов в C#. Это позволяет создавать более интуитивные и удобные синтаксические конструкции для пользовательских типов данных.

4. Как используется ключевое слово operator?

- Ключевое слово operator используется для определения перегруженных операторов в пользовательских типах данных. Оно следует за модификатором доступа (public, static, и т. д.) и указывает, какой оператор будет перегружен.

5. Какие операции нельзя перегружать в C#?

- В C# нельзя перегружать следующие операции:

- sizeof

- typeof

- -> (стрелочный оператор)

- Присваивание = (можно перегрузить только для пользовательских классов)

- Вызов метода ()

- Постфиксные операции x++ и x--

6. Можно ли перегрузкой отменить очередность выполнения операции?

- Нет, перегрузкой оператора нельзя изменить порядок выполнения операции. Перегрузка операторов влияет только на способ обработки операций для пользовательских типов, но не меняет их базовое поведение.

7. Истинно ли следующее утверждение: операция >= может быть перегружена.

- Да, операция >= может быть перегружена для пользовательских типов.

8. Сколько аргументов требуется для определения перегруженной унарной операции?

- Для определения перегруженной унарной операции требуется только один аргумент, который представляет объект, к которому применяется унарная операция.

9. Можно ли перегружать операцию []?

- Да, операцию [] можно перегружать в C#. Она позволяет создавать пользовательские индексаторы для доступа к элементам объекта, подобно массивам.

10. Можно ли перегружать операцию ->?

- Нет, операцию -> нельзя перегружать в C#. Она предназначена для работы с указателями и используется в низкоуровневых операциях.

11. Приведите пример оператора приведения типа.

- Например, оператор приведения типа из double в int:

public static explicit operator int(double value)

{

return (int)value;

}

12. Что такое метод расширения? Как и где его можно использовать?

- Метод расширения (extension method) - это статический метод, который добавляет новую функциональность существующему типу данных без изменения его исходного кода. Он может быть использован для любого класса, включая классы .NET Framework и ваши собственные классы. Метод расширения определяется в статическом классе и должен быть в том же пространстве имен, что и класс, к которому он применяется.

13. Пример оператора преобразования типа:

public static implicit operator int(Point3D point)

{ return point.Z; // Преобразование Point3D в int, используя значение Z}

14. Выберите верное утверждение. Метод расширения может:

- Метод расширения может получать доступ только к public членам расширяемого класса и не может получать доступ к protected, internal или private членам.

15. Выберите из списка неверное правило перегрузки операторов для C#.

- Неверное правило перегрузки операторов для C#: \*\*4) должны объявляться как protected\*\*. Перегрузка операторов может происходить с использованием различных модификаторов доступа, таких как `public`, `private`, `internal`, и `protected` не является обязательным модификатором. Выбор модификатора доступа зависит от того, какой доступ вы хотите предоставить к перегруженному оператору, и может быть адаптирован к вашим потребностям.

**Лаба 8**

**1. Что такое делегат? Как определить делегат?**

**Делегат** - это тип данных в C#, который представляет собой ссылку на метод. Он позволяет передавать метод как параметр другого метода, хранить его в переменной и вызывать его по этой переменной.

Делегат можно определить с использованием ключевого слова `delegate`. Например:

delegate void MyDelegate(int x);

**2. Назначение делегатов**

Делегаты используются для обеспечения гибкости и расширяемости кода. Они позволяют передавать методы в качестве параметров, создавать цепочки методов (множественные делегаты), а также использовать их в событиях для обработки событий.

**3. Какие есть способы присваивания делегату адреса метода?**

Делегату можно присвоить адрес метода следующими способами:

- С использованием имени метода: `MyDelegate del = MyMethod;`

- С использованием конструктора делегата: `MyDelegate del = new MyDelegate(MyMethod);`

- С использованием анонимных методов.

- С использованием лямбда-выражений.

**4. Поясните назначение метода Invoke.**

Метод `**Invoke**` используется для явного вызова делегата. Вместо того чтобы вызывать делегат как метод (например, `myDelegate()`), вы можете использовать `myDelegate.Invoke()`. Это предоставляет явный способ вызова делегата, но в большинстве случаев вызов делегата напрямую считается более читаемым и предпочтительным.

**5. Что такое групповая адресация делегата?**

Групповая адресация (**multicast**) делегата позволяет объединять несколько методов в одном делегате. Вызов такого делегата приведет к вызову всех объединенных методов в порядке, в котором они были добавлены.

**6. Как создать событие?**

Событие в C# создается с использованием ключевого слова `event`. Пример:

class MyClass

{public event Action MyEvent;}

**7. Как события связаны с делегатами? Опишите и поясните схему взаимодействия.**

**Событие в C#** на самом деле представляет собой делегат, который ссылается на методы обработчики события. Определенный делегат используется для хранения этих обработчиков, и событие предоставляет интерфейс для добавления, удаления и вызова обработчиков.

**8. Что такое лямбда-выражения? Приведите пример лямбда-выражения с несколькими параметрами.**

**Лямбда-выражение** - это краткий способ определения анонимных методов (функций). Пример:

Func<int, int, int> add = (a, b) => a + b;

**9. Что такое ковариантность и контравариантность делегатов? Что это дает?**

Ковариантность позволяет использовать производный тип вместо базового типа возвращаемого значения делегата, а контравариантность позволяет использовать базовый тип вместо производного типа параметра делегата. Это делает код более гибким и позволяет использовать разные типы, не создавая дополнительных делегатов.

**10. Поясните разницу между встроенными делегатами Action и Func.**

**Action** - это делегат, предназначенный для методов, которые не возвращают значение. Func - это делегат, предназначенный для методов, возвращающих значение. Различие между ними заключается в том, что Action не имеет возвращаемого значения, в то время как Func имеет. Например, `Action<int, string>` принимает два параметра (целое число и строку) и не возвращает значение, в то время как `Func<int, string, bool>` принимает два параметра и возвращает булево значение.

**Подготовка к зачету**

Вот краткое изложение основных аспектов, касающихся C# и .NET Framework, по вашим пунктам:

1. Состав .NET Framework. Структура среды выполнения CLR

- \*\*.NET Framework\*\* включает в себя:

- \*\*CLR (Common Language Runtime)\*\*: управляет выполнением программ, обеспечивает сборку мусора, безопасность и обработку исключений.

- \*\*Библиотеки классов (Framework Class Library)\*\*: набор классов и API для решения общих задач.

- \*\*ASP.NET, ADO.NET, Windows Forms и WPF\*\*: технологии для разработки веб-приложений, работы с базами данных и создания пользовательских интерфейсов.

2. Структура управляемого модуля - Portable Executable (PE)

- \*\*PE\*\* — формат файлов, используемый в Windows для исполняемых файлов и библиотек.

- \*\*Сборка\*\*: единица развертывания и версионирования, содержащая метаданные, код и ресурсы.

- \*\*CIL (Common Intermediate Language)\*\*: промежуточный язык, на который компилируются языки .NET, исполняемый CLR.

3. CTS (Common Type System)

- \*\*CTS\*\* определяет, как типы данных используются в .NET, обеспечивая совместимость между языками.

- \*\*Типы данных C#\*\*:

- \*\*Ссылочные типы\*\*: хранят ссылки на объекты (например, классы).

- \*\*Типы значений\*\*: хранят значения непосредственно (например, int, float).

4. Упаковка и распаковка типов

- \*\*Упаковка\*\*: преобразование типа значения в ссылочный (например, `int` в `object`).

- \*\*Распаковка\*\*: преобразование ссылочного типа обратно в значение.

- \*\*Nullable\*\*: позволяет типам значений принимать значение `null`. Использует `Nullable<T>`. Примеры: проверка на наличие значения и использование оператора `??`.

5. Тип данных String

- \*\*String\*\*: ссылочный тип, представляющий последовательность символов.

- \*\*Операции\*\*: конкатенация, сравнение, форматирование, методы `Substring`, `IndexOf`.

- \*\*Литералы\*\*: строки заключаются в двойные кавычки.

- \*\*Пустые и нулевые строки\*\*: пустая строка — `""`, нулевая — `null`.

6. Неявная типизация

- \*\*var\*\*: позволяет компилятору выводить тип переменной на основе присвоенного значения. Пример: `var x = 10;`.

7. Массивы C#

- \*\*Одномерные массивы\*\*: простой список элементов одного типа.

- \*\*Прямоугольные массивы\*\*: двумерные массивы с равным количеством элементов в каждой строке.

- \*\*Ступенчатые массивы\*\*: массив массивов (размеры могут отличаться).

8. Понятие кортежей

- \*\*Кортежи\*\*: позволяют хранить набор значений разных типов в одной переменной.

- \*\*Создание\*\*: `var tuple = (1, "пример");`.

9. Принципы объектно-ориентированного программирования

- \*\*Инкапсуляция\*\*: сокрытие деталей реализации.

- \*\*Наследование\*\*: создание нового класса на основе существующего.

- \*\*Полиморфизм\*\*: возможность использовать объекты разных типов через общий интерфейс.

10. Класс. Элементы класса

- \*\*Элементы класса\*\*: поля, методы, свойства, конструкторы.

- \*\*Свойства\*\*: позволяют контролировать доступ к полям.

11. Класс. Константы

- \*\*Константы\*\*: значения, которые не могут изменяться после инициализации.

- \*\*Поля только для чтения\*\*: могут быть инициализированы только в конструкторе.

12. Спецификаторы доступа C#

- \*\*public, private, protected, internal\*\*: определяют видимость типов и членов классов.

13. Класс. Конструкторы и их свойства

- \*\*Конструкторы\*\*: специальные методы для инициализации объектов.

- \*\*Деструкторы\*\*: освобождают ресурсы перед удалением объекта.

14. Класс и методы System.Object

- \*\*Методы\*\*: `Equals()`, `GetHashCode()`, `ToString()`, которые могут переопределяться в пользовательских классах.

15. Статические методы и статические конструкторы класса

- \*\*Статические методы\*\*: принадлежат классу, а не его экземплярам.

- \*\*Статические конструкторы\*\*: инициализируют статические члены класса.

16. Статические классы

- \*\*Статические классы\*\*: не могут быть инстанцированы, содержат только статические члены.

- \*\*Методы расширения\*\*: позволяют добавлять методы к существующим типам без изменения их кода.

17. Анонимные типы

- \*\*Анонимные типы\*\*: используются для создания объектов без явного объявления класса, например, `var person = new { Name = "John", Age = 30 };`.

18. Модификаторы параметров

- \*\*ref, out, params\*\*: позволяют передавать параметры по ссылке, возвращать несколько значений и передавать переменное количество аргументов, соответственно.

- \*\*Необязательные и именованные аргументы\*\*: позволяют задавать параметры по умолчанию и вызывать методы с указанием имен параметров.

19. Перегрузка методов и операторов

- \*\*Перегрузка методов\*\*: позволяет создавать несколько методов с одинаковым именем, но с разными параметрами.

- \*\*Перегрузка операторов\*\*: позволяет определять поведение операторов для пользовательских типов.

20. Операции преобразования типа

- \*\*Явная и неявная форма\*\*: явное преобразование требует явного указания, неявное — происходит автоматически.

- \*\*Ограничения\*\*: необходимо учитывать совместимость типов при преобразовании.

Вот краткое изложение основных аспектов, касающихся C# и .NET Framework по вашим пунктам с 21 по 63:

21. Вложенные типы. Вложенные объекты

- \*\*Вложенные типы\*\*: классы, структуры или интерфейсы, определенные внутри другого класса. Это позволяет организовать код и логически сгруппировать связанные элементы.

- \*\*Вложенные объекты\*\*: экземпляры вложенных типов, которые могут использоваться как обычные объекты, но имеют доступ к членам внешнего класса.

22. Правила наследования C#

- \*\*Наследование\*\*: позволяет создавать новый класс на основе существующего. Поддерживает как одиночное, так и множественное интерфейсное наследование.

- \*\*Запрещено наследование\*\*: от классов, помеченных как `sealed`.

23. Сокрытие имен при наследовании

- \*\*Сокрытие имен\*\*: использование ключевого слова `new` для сокрытия членов базового класса в производном.

- \*\*Обращение к сокрытым членам\*\*: возможно через базовый класс, но рекомендуется избегать использования сокрытых членов, чтобы избежать путаницы.

24. Использование операций `is` и `as`

- \*\*Операция `is`\*\*: проверяет, является ли объект экземпляром определенного типа.

- \*\*Операция `as`\*\*: пытается привести объект к заданному типу и возвращает `null`, если приведение невозможно, что предотвращает исключения.

25. Полиморфизм

- \*\*Полиморфизм\*\*: возможность использовать объекты разных типов через общий интерфейс.

- \*\*Виртуальные методы\*\*: методы, которые могут быть переопределены в производных классах.

- \*\*Правила переопределения\*\*: необходимо использовать ключевое слово `override` для переопределения виртуальных методов.

26. Понятие раннего и позднего связывания

- \*\*Раннее связывание\*\*: определение методов и свойств во время компиляции (обычное поведение).

- \*\*Позднее связывание\*\*: определение методов и свойств во время выполнения, чаще всего через виртуальные методы.

27. Абстрактные классы и методы

- \*\*Абстрактные классы\*\*: классы, которые не могут быть инстанцированы и могут содержать абстрактные методы (методы без реализации).

- \*\*Бесплодные классы\*\*: классы, которые не могут быть использованы для создания экземпляров и служат только для наследования.

28. Структуры в C#

- \*\*Структуры\*\*: типы значений, которые могут содержать данные и методы. Обладают большей производительностью по сравнению с классами, но не поддерживают наследование.

29. Интерфейсы

- \*\*Интерфейсы\*\*: определяют набор методов и свойств, которые должен реализовать класс. Не могут содержать реализацию.

- \*\*Реализация интерфейсов\*\*: класс, реализующий интерфейс, должен реализовать все его члены.

30. Явная и неявная реализация интерфейсов

- \*\*Неявная реализация\*\*: члены интерфейса реализуются как обычные методы класса.

- \*\*Явная реализация\*\*: члены интерфейса реализуются с указанием имени интерфейса, что позволяет скрыть их от других пользователей класса.

31. Ковариантность и контравариантность интерфейсов

- \*\*Ковариантность\*\*: позволяет использовать более производные типы (выходные параметры).

- \*\*Контравариантность\*\*: позволяет использовать более базовые типы (входные параметры).

32. Стандартные интерфейсы .NET

- \*\*IEnumerable, IList, IDictionary\*\*: интерфейсы для работы с коллекциями. Обеспечивают стандартные методы для обхода и манипуляции данными.

33. Исключительные ситуации

- \*\*Генерация исключений\*\*: с помощью ключевого слова `throw`.

- \*\*Повторная генерация\*\*: использование `throw;` для повторной генерации текущего исключения.

34. Варианты обработки исключений

- \*\*try-catch-finally\*\*: блоки для обработки исключений.

- \*\*Фильтры исключений\*\*: позволяют обрабатывать только определенные типы исключений.

35. Обобщения (generics)

- \*\*Обобщения\*\*: позволяют создавать классы, методы и интерфейсы, которые могут работать с любыми типами данных, обеспечивая безопасность типов и повторное использование кода.

36. Концепция ограничений обобщений

- \*\*Ограничения обобщений\*\*: позволяют задать условия для типов, которые могут быть использованы в обобщениях. Например, `where T : class` указывает, что T должен быть ссылочным типом.

37. Делегаты

- \*\*Определение\*\*: тип, представляющий ссылку на метод.

- \*\*Назначение\*\*: позволяет передавать методы как параметры.

- \*\*Обобщенные делегаты\*\*: `Func<T>`, `Action<T>`, которые позволяют передавать параметры и возвращать значения.

38. Анонимные функции и лямбда-выражения

- \*\*Анонимные функции\*\*: функции без имени, которые можно использовать для создания делегатов.

- \*\*Лямбда-выражения\*\*: сокращенный синтаксис для определения анонимных функций, например, `(x, y) => x + y`.

39. Обобщенные делегаты .NET

- \*\*Action\*\*: делегат без возвращаемого значения. Например, `Action<string>`.

- \*\*Func\*\*: делегат с возвращаемым значением. Например, `Func<int, int>`.

- \*\*Predicate\*\*: делегат, который возвращает `bool`. Например, `Predicate<string>`.

40. События и делегаты

- \*\*События\*\*: механизм, позволяющий объектам уведомлять другие объекты о произошедших изменениях. События основаны на делегатах.

41. Стандартные коллекции .NET

- \*\*Типы коллекций\*\*: массивы, списки, множества, словари, очереди и стеки. Каждая коллекция имеет свои особенности и предназначение.

42. Стандартные интерфейсы коллекций

- \*\*ICollection, IList, IDictionary\*\*: интерфейсы, предоставляющие базовые операции для работы с коллекциями.

43. IEnumerable и IEnumerator

- \*\*IEnumerable\*\*: интерфейс, который позволяет перечислять элементы коллекции.

- \*\*IEnumerator\*\*: интерфейс, который предоставляет методы для перемещения по элементам коллекции.

44. LINQ to Objects

- \*\*LINQ\*\*: язык запросов для работы с данными. Позволяет использовать синтаксис SQL для работы с коллекциями в памяти.

- \*\*Отложенные и неотложенные операции\*\*: отложенные операции выполняются только при необходимости, неотложенные — сразу.

45. LINQ to Objects: операции

- \*\*Where, Select, Take, OrderBy, Join, GroupBy\*\*: основные операции для фильтрации, проекции, сортировки и группировки данных.

46. Рефлексия. System Type

- \*\*Рефлексия\*\*: возможность получения информации о типах в программе во время выполнения.

- \*\*System.Type\*\*: класс, представляющий тип данных и предоставляющий методы для работы с рефлексией.

47. Классы для работы с файловой системой

- \*\*File, Directory, FileInfo, DirectoryInfo\*\*: классы для работы с файлами и директориями, позволяют выполнять операции чтения, записи, копирования и удаления.

48. Синтаксическая конструкция using

- \*\*using\*\*: позволяет автоматически освобождать ресурсы после использования объекта, реализующего интерфейс `IDisposable`.

- \*\*Чтение и запись файлов\*\*: классы `StreamReader`, `StreamWriter` используются для чтения и записи текстовых файлов.

49. Классы адаптеры потоков

- \*\*Классы адаптеров\*\*: позволяют использовать различные источники и приемники данных, например, `StreamAdapter`.

50. Сериализация

- \*\*Сериализация\*\*: процесс преобразования объекта в формат, который можно сохранить или передать по сети.

- \*\*Форматы сериализации\*\*: бинарный, XML, JSON.

51. Сериализация контрактов данных

- \*\*Интерфейс ISerializable\*\*: позволяет контролировать процесс сериализации и десериализации пользовательских объектов.

52. Атрибуты

- \*\*Атрибуты\*\*: метаданные, которые могут быть добавлены к классам, методам и другим элементам для передачи дополнительной информации.

- \*\*Создание собственного атрибута\*\*: создайте класс, унаследованный от `Attribute`.

53. Процесс. Домен приложений. Поток выполнения

- \*\*Процесс\*\*: экземпляр выполняемой программы.

- \*\*Домен приложений\*\*: изолированная среда для выполнения кода .NET.

- \*\*Поток выполнения\*\*: единица работы, выполняемая в процессе.

54. Создание потоков

- \*\*Создание потоков\*\*: использование класса `Thread`.

- \*\*Приоритеты потоков\*\*: определяют, насколько важен поток по сравнению с другими.

- \*\*Состояния потоков\*\*: новые, готовые, выполняющиеся, заблокированные и завершенные.

55. Синхронизация потоков

- \*\*Lock\*\*: блокировка объекта для предотвращения одновременного доступа к ресурсу.

- \*\*Monitor, Mutex, Semaphore\*\*: классы для управления доступом к ресурсам между потоками.

56. Библиотека параллельных задач TPL

- \*\*Класс Task\*\*: представляет асинхронную операцию.

- \*\*Состояние задачи\*\*: может быть создана, запущена, завершена или отменена.

57. Способы создания Task

- \*\*Создание Task\*\*: с использованием `Task.Run` или `Task.Factory.StartNew`.

- \*\*Возврат результата\*\*: использование `Task<T>` для получения результата.

- \*\*Отмена выполнения задач\*\*: использование `CancellationToken`.

- \*\*Продолжения\*\*: использование `ContinueWith` для выполнения действий после завершения задачи.

58. Параллелизм при императивной обработке данных

- \*\*Класс Parallel\*\*: предоставляет методы для параллельного выполнения операций, например, `Parallel.For`.

59. Асинхронные методы

- \*\*async и await\*\*: ключевые слова для реализации асинхронного программирования. `async` помечает метод как асинхронный, а `await` используется для ожидания завершения задачи.

60. Проектирование отношений

- \*\*Агрегация\*\*: "имеет" отношение между объектами, где один объект может существовать независимо от другого.

- \*\*Композиция\*\*: более сильное "имеет" отношение, где жизненный цикл объектов связан.

- \*\*Ассоциация\*\*: общее отношение между объектами, без зависимости.

61. Антипаттерны проектирования

- \*\*Антипаттерны\*\*: распространенные ошибки в проектировании, которые следует избегать.

- \*\*Рефакторинг\*\*: процесс улучшения кода без изменения его функциональности. Методы рефакторинга включают упрощение, сокращение дублирования и улучшение читаемости кода.

62. Чистый код

- \*\*Требования к именам\*\*: понятные и описательные названия.

- \*\*Функции\*\*: должны быть короткими и выполнять одну задачу.

- \*\*Форматирование\*\*: соблюдение единообразия в коде для улучшения читаемости.

63. Чистый код

- \*\*Требования к классам и объектам\*\*: классы должны быть ответственными за одну задачу и не должны содержать избыточных методов.