### 1. Состав .NET Framework. Структура среды выполнения CLR

\*\*.NET Framework\*\* традиционно включает в себя несколько ключевых компонентов:

1. \*\*CLR (Common Language Runtime)\*\* – среда выполнения, отвечающая за управление жизненным циклом объектов, сборку мусора (Garbage Collection), Just-In-Time (JIT) компиляцию, безопасность и т.д.

2. \*\*FCL (Framework Class Library)\*\* или \*\*BCL (Base Class Library)\*\* – базовая библиотека классов, предоставляющая типы и API для разработки (коллекции, ввод-вывод, работа со строками, сетевые операции, LINQ и т.д.).

3. \*\*Языковые компиляторы\*\* (C#, VB.NET, F# и другие), которые транслируют код в промежуточный язык (CIL).

4. \*\*Вспомогательные инструменты\*\* – утилиты (например, MSBuild, CSC, ILDASM и др.), позволяющие управлять процессом сборки и анализа кода.

\*\*Структура CLR\*\* включает:

- \*\*Классический Loader\*\* – загружает сборки, управляет доменами приложений.

- \*\*JIT-компилятор\*\* – компилирует промежуточный язык (CIL) в машинный код для конкретной платформы.

- \*\*GC (Garbage Collector)\*\* – осуществляет сборку мусора, автоматизируя управление памятью.

- \*\*Безопасность (Code Access Security, раньше CAS)\*\* – ранее позволяла управлять правами, сейчас в .NET Core/5+ механизм изменён, но принцип остаётся: CLR обеспечивает определённый уровень защиты и валидации кода.

- \*\*Управление типами\*\* – система метаданных и проверок, которая гарантирует типобезопасность.

---

### 2. Структура управляемого модуля (PE). Понятие и исполнение сборки. CIL

1. \*\*Управляемый модуль (PE – Portable Executable)\*\* в .NET представляет собой файл (обычно .exe или .dll), содержащий:

- Заголовок PE/COFF (стандартный для Windows).

- Метаданные .NET (описание типов, методов, сборки и т.д.).

- Код на CIL (Common Intermediate Language), ранее называвшийся MSIL.

- Маршрутизацию к исполняющему модулю CLR.

2. \*\*Сборка (Assembly)\*\* – это логическая единица распределения и повторного использования кода. Может состоять из одного или нескольких модулей (чаще всего – один файл), имеет метаданные (Manifest), в которых описаны версия, список файлов, ресурсы, strong name (если подписана). Исполняется в среде CLR.

3. \*\*CIL (Common Intermediate Language)\*\* – промежуточный язык, в который компилируются все .NET-языки. При запуске приложения CIL-команды JIT-компилятором транслируются в машинный код конкретной платформы.

---

### 3. CTS (Common Type System). Типы данных C#. Ссылочные и типы значений

1. \*\*CTS (Common Type System)\*\* – общая система типов, гарантирующая, что во всех .NET-языках типы совместимы на уровне CLR. В рамках CTS существуют две основных категории: \*\*Value Types\*\* (значимые типы) и \*\*Reference Types\*\* (ссылочные типы).

2. \*\*Значимые типы\*\* (value types) в C#:

- Простые (int, float, double, bool и т.д.),

- Структуры (struct),

- Перечисления (enum).

При передаче по значению копируется само значение.

3. \*\*Ссылочные типы\*\* (reference types) в C#:

- Классы (class),

- Интерфейсы (interface),

- Делегаты (delegate),

- Массивы (array),

- Строки (string).

При передаче по ссылке копируется ссылка (указатель) на объект в куче.

---

### 4. Понятие упаковки и распаковки типов. Типы Nullable: преобразование, проверка, null-объединение

1. \*\*Упаковка (boxing)\*\* – преобразование значимого типа (value type) в ссылочный тип (object). CLR выделяет память в куче и копирует значение. Например:

```csharp

int x = 5;

object obj = x; // упаковка

```

2. \*\*Распаковка (unboxing)\*\* – обратный процесс преобразования объекта (object) обратно в value type. Требует явного приведения:

```csharp

int y = (int)obj; // распаковка

```

3. \*\*Nullable-типы\*\* (например, `int?`, `double?`) позволяют value type принимать дополнительное состояние `null`.

- Проверка, есть ли значение, – через `.HasValue` или простую проверку `== null`.

- Преобразование: неявное из `T` в `T?`, и явное из `T?` в `T`.

- \*\*Оператор ?? (null-объединение)\*\* – позволяет задать значение по умолчанию, если `T?` равно `null`.

```csharp

int? a = null;

int b = a ?? -1; // если a == null, вернётся -1

```

---

### 5. Тип данных String: операции, литералы, пустые и нулевые строки, форматированный вывод

1. \*\*String\*\* в C# – это ссылочный неизменяемый тип (immutable). Операции над строками создают новые объекты.

2. \*\*Операции\*\*: конкатенация (`+`), сравнение (`==` и `.Equals()`), получение подстроки, поиск, замена и т.д.

3. \*\*Литералы\*\* строк: `"Hello"`, интерполированные строки: `$"Hello {name}"`, вербатим-строки: `@"C:\folder\file.txt"`.

4. \*\*Пустая строка\*\* (`String.Empty` или `""`) не равна `null`.

5. \*\*Нулевая строка\*\* (`null`) – отсутствие объекта.

6. \*\*Форматированный вывод\*\*: методы `String.Format(...)`, интерполяция строк `($"{var}")`, метод `Console.WriteLine("{0}", var)`.

---

### 6. Неявная типизация – назначение и использование

- В C# неявная типизация объявляется с помощью ключевого слова `var`:

```csharp

var x = 10; // x будет int

var list = new List<string>(); // list будет List<string>

```

- \*\*Назначение\*\* – упростить код, не писать явно тип при очевидном присвоении.

- \*\*Использование\*\*: переменная, объявленная через `var`, должна быть сразу инициализирована, иначе компилятор не сможет вывести тип.

---

### 7. Массивы C# одномерные, прямоугольные и ступенчатые

1. \*\*Одномерный массив\*\*:

```csharp

int[] arr = new int[5];

```

2. \*\*Прямоугольный (многомерный) массив\*\* – хранится как “таблица” в памяти:

```csharp

int[,] matrix = new int[3,4];

```

3. \*\*Ступенчатый (jagged) массив\*\* – массив массивов:

```csharp

int[][] jagged = new int[3][];

jagged[0] = new int[4];

jagged[1] = new int[2];

jagged[2] = new int[5];

```

Каждая “строка” может иметь разную длину.

---

### 8. Понятие кортежей. Свойства, создание

- \*\*Кортеж\*\* (tuple) в C# – структура, позволяющая группировать несколько значений в один объект без создания отдельного класса/структуры.

- Создание кортежа:

```csharp

var person = (Name: "Alice", Age: 30);

```

- Доступ к элементам: `person.Name` или `person.Item1`.

- Могут использоваться как возвращаемые значения из методов, для удобного группирования нескольких результатов.

---

### 9. Принципы объектно-ориентированного программирования

Классические \*\*ООП-принципы\*\*:

1. \*\*Инкапсуляция\*\* – скрытие деталей реализации от внешнего мира, предоставление чётко определённого интерфейса.

2. \*\*Наследование\*\* – способность одного класса (потомка) наследовать поля и методы другого класса (родителя).

3. \*\*Полиморфизм\*\* – способность методов с одним именем иметь разное поведение в зависимости от типов данных (виртуальные методы, интерфейсы, переопределения).

4. Дополнительно часто выделяют \*\*Абстракцию\*\* – упрощённое представление объекта, выделяющее только значимые характеристики.

---

### 10. Класс. Элементы класса. Свойства и индексаторы

1. \*\*Класс\*\* – основной ссылочный тип, содержащий: поля, методы, свойства, индексаторы, события, константы, вложенные типы и т.д.

2. \*\*Свойства (properties)\*\* – механизм, позволяющий инкапсулировать логику чтения/записи полей. Пример:

```csharp

public class Person {

private string name;

public string Name {

get { return name; }

set { name = value; }

}

}

```

3. \*\*Индексаторы\*\* – дают возможность обращаться к элементам класса по индексу, подобно массиву:

```csharp

public class MyCollection {

private int[] data = new int[10];

public int this[int index] {

get { return data[index]; }

set { data[index] = value; }

}

}

```

---

### 11. Класс. Константы. Поля только для чтения. Инициализаторы класса

1. \*\*Константы\*\* (`const`) – значение задаётся на этапе компиляции и не может изменяться во время выполнения. Например:

```csharp

public const double PI = 3.14;

```

2. \*\*Поля только для чтения\*\* (`readonly`) – значение задаётся либо при объявлении, либо в конструкторе. Менять после инициализации нельзя.

3. \*\*Инициализаторы класса\*\* – синтаксис, позволяющий присвоить начальные значения полям при объявлении:

```csharp

public class Example {

private int x = 10;

private readonly int y = 20;

}

```

---

### 12. Спецификаторы доступа C#. Видимость типов. Доступ к членам типов

В C# есть несколько уровней доступности:

1. `public` – доступен везде.

2. `private` – доступен только внутри текущего класса/структуры.

3. `protected` – доступен внутри класса и его потомков.

4. `internal` – доступен внутри сборки (assembly).

5. `protected internal` – доступен внутри сборки и в классах-наследниках из других сборок.

6. `private protected` – доступен только внутри текущей сборки и только в производных классах.

\*\*Видимость типов\*\* (классов, структур) определяется аналогично: тип может быть `public` или `internal` (по умолчанию `internal`, если не указано иное).

---

### 13. Класс. Конструкторы и их свойства. Деструкторы

1. \*\*Конструктор\*\* – специальный метод, вызываемый при создании объекта:

- Имеет то же имя, что и класс.

- Не имеет возвращаемого типа.

- Может быть \*\*перегружен\*\* (несколько конструкторов с разными параметрами).

- Может вызывать конструктор базового класса через `: base(...)`.

- Может вызывать другой конструктор собственного класса через `: this(...)`.

2. \*\*Статический конструктор\*\* – вызывается один раз при первом обращении к классу или его членам.

3. \*\*Деструктор\*\* (финализатор) – метод `~ClassName()`, который вызывается сборщиком мусора перед уничтожением объекта. Применяется редко, в основном для освобождения неуправляемых ресурсов. В .NET обычно используют `IDisposable` вместо деструктора.

---

### 14. Класс и методы System.Object

Все типы в C# неявно наследуются от `System.Object` (кроме указателей на уровне unsafe-кода). Важные методы:

1. `ToString()` – возвращает строковое представление объекта.

2. `Equals(object obj)` – сравнение объектов на равенство. По умолчанию сравнивает ссылки. Может быть переопределён.

3. `GetHashCode()` – возвращает числовой хеш-код. При переопределении `Equals` желательно переопределять `GetHashCode`.

4. `GetType()` – возвращает тип объекта во время выполнения.

5. `MemberwiseClone()` – создаёт поверхностную копию объекта (shallow copy).

---

### 15. Статические методы и статические конструкторы класса

- \*\*Статические методы\*\* принадлежат классу, а не экземпляру. Их вызов не требует создания объекта:

```csharp

public static void Foo() { ... }

MyClass.Foo();

```

- \*\*Статический конструктор\*\*:

- Объявляется без модификаторов доступа и без параметров:

```csharp

static MyClass() { ... }

```

- Вызывается автоматически при первом обращении к любому члену класса или при создании первого экземпляра.

- Инициализирует статические поля или выполняет другую логику начальной загрузки.

---

### 16. Статические классы. Методы расширения и правила их определения

1. \*\*Статический класс\*\* (`static class`) – содержит только статические члены, не может быть создан как объект, не может быть унаследован.

2. \*\*Методы расширения\*\* (extension methods) – позволяют добавлять новые методы к уже существующим типам без модификации их кода.

- Определяются внутри \*\*статического\*\* класса и объявляются \*\*статическим\*\* методом с первым параметром, начинающимся с `this`:

```csharp

public static class StringExtensions {

public static int WordCount(this string str) {

return str.Split(new char[] {' ', '.', '?'}, StringSplitOptions.RemoveEmptyEntries).Length;

}

}

```

- Вызываются будто бы это методы экземпляра:

```csharp

string text = "Hello World";

int count = text.WordCount();

```

---

### 17. Анонимные типы

- \*\*Анонимные типы\*\* позволяют создавать объекты без явного определения класса, используя ключевое слово `new`:

```csharp

var person = new { Name = "Alice", Age = 30 };

```

- Свойства анонимного типа доступны только для чтения. Используются для временных структур данных (часто в LINQ-запросах).

---

### 18. Модификаторы параметров - ref , out, params. Необязательные и именованные аргументы

1. \*\*ref\*\* – позволяет передавать параметр по ссылке с сохранением текущего значения; на входе переменная должна быть инициализирована.

2. \*\*out\*\* – передача по ссылке без необходимости инициализировать переменную на входе, но метод обязан присвоить ей значение внутри.

3. \*\*params\*\* – позволяет передавать переменное число аргументов:

```csharp

public void Sum(params int[] numbers) { ... }

```

4. \*\*Необязательные параметры\*\* – указываются со значением по умолчанию:

```csharp

public void Foo(int x = 10) { ... }

```

5. \*\*Именованные аргументы\*\* – можно передавать аргументы в любом порядке, указывая имя: `Foo(x: 10)`.

---

### 19. Перегрузка методов и операторов. Правила перегрузки операторов

1. \*\*Перегрузка методов\*\* – создание нескольких методов с одинаковым именем, но разными сигнатурами (набор параметров).

2. \*\*Перегрузка операторов\*\* – в C# можно переопределить поведение некоторых операторов (`+`, `-`, `==`, `>`, `<`, `true/false` и др.) для пользовательских типов.

- Оператор перегружается как статический метод с ключевым словом `operator`:

```csharp

public static MyClass operator +(MyClass a, MyClass b) { ... }

```

- Правила: перегружать можно не все операторы, некоторые перегружаются “поровну” (если перегрузили `==`, надо перегрузить `!=`).

---

### 20. Операции преобразования типа. Явная и неявная форма. Ограничения

- \*\*Неявное преобразование\*\* (`implicit`) – безопасное преобразование без потери данных. Например, `int -> long`.

- \*\*Явное преобразование\*\* (`explicit`) – используется, когда возможно частичное или полное несоответствие типов. Требует оператора приведения в коде `(Type)`.

- В пользовательских типах можно определять собственные операторы преобразования:

```csharp

public static explicit operator MyClass(int value) { ... }

public static implicit operator MyClass(AnotherClass ac) { ... }

```

- Нужно соблюдать логику “безопасности”: \*\*implicit\*\* не должен приводить к неожиданной потере данных.

---

### 21. Вложенные типы. Вложенные объекты

1. \*\*Вложенный тип\*\* – класс, структура, перечисление или интерфейс, определённый внутри другого класса/структуры. Может иметь модификаторы доступа (`private`, `public`, и др.).

```csharp

public class OuterClass {

private class InnerClass {

...

}

}

```

2. \*\*Вложенные объекты\*\* – объекты, создаваемые на основе вложенного класса. Их доступность зависит от модификаторов доступа.

3. Вложенные типы полезны, когда их использование логически ограничено внешним классом.

---

### 22. Правила наследования C#

1. В C# отсутствует множественное наследование классов, разрешено \*\*единственное\*\* наследование (класс может иметь только одного родителя).

2. Класс может реализовывать \*\*сколько угодно интерфейсов\*\*.

3. Корневой класс иерархии – `System.Object`.

4. Наследование оформляется двоеточием:

```csharp

public class Derived : Base {

...

}

```

5. Модификаторы доступа (public, protected, private) регулируют доступ к членам базового класса при наследовании.

---

### 23. Сокрытие имен при наследовании. Обращение к скрытым членам

1. Если в производном классе объявлен член (поле, метод) с тем же именем, что и в базовом, он \*\*скрывает\*\* базовый член. В C# используется ключевое слово `new`:

```csharp

public class Base {

public void Foo() { ... }

}

public class Derived : Base {

public new void Foo() { ... }

}

```

2. Чтобы вызвать скрытый член базового класса, используют явное указание типа:

```csharp

((Base)this).Foo();

```

3. Сокрытие отличается от переопределения (override). Сокрытие не использует механизм виртуальности, а полностью скрывает член родителя.

---

### 24. Использование операций is и as

1. \*\*is\*\* – проверяет, относится ли объект к заданному типу (возвращает `true`/`false`):

```csharp

if (obj is MyClass mc) {

// mc уже приведён к MyClass

}

```

С C# 7 можно использовать “pattern matching” (после is сразу получить переменную нужного типа).

2. \*\*as\*\* – пытается привести объект к заданному типу, при неудаче возвращает `null`, не вызывает исключения:

```csharp

MyClass mc = obj as MyClass;

if (mc != null) {

// Успешное приведение

}

```

---

### 25. Полиморфизм. Виртуальные методы, свойства и индексаторы. Правила переопределения

1. \*\*Полиморфизм\*\* – способность методов (и других членов) иметь различную реализацию при одном имени.

2. \*\*Виртуальные методы\*\* – в базовом классе помечаются ключевым словом `virtual`:

```csharp

public virtual void Foo() { ... }

```

В производном классе переопределяются с `override`:

```csharp

public override void Foo() { ... }

```

3. \*\*Свойства\*\* и \*\*индексаторы\*\* тоже могут быть `virtual` и `override`.

4. Если необходимо запретить дальнейшее переопределение, используется `sealed override`.

---

### 26. Понятие раннего и позднего связывания

1. \*\*Раннее связывание (Early Binding)\*\* – выбор вызываемого метода происходит во время компиляции, известен статически (например, вызов невиртуального метода).

2. \*\*Позднее связывание (Late Binding)\*\* – выбор вызываемого метода делается на этапе выполнения (runtime) в зависимости от реального типа объекта. Это ключ к полиморфизму (виртуальные методы, интерфейсы).

---

### 27. Абстрактные классы и методы. Бесплодные классы

1. \*\*Абстрактный класс\*\* – не может быть создан напрямую (через `new`), служит как “шаблон” для наследников. Объявляется с ключевым словом `abstract`.

2. \*\*Абстрактные методы\*\* – объявляются без тела (только сигнатура), реализацию дают наследники:

```csharp

public abstract class Shape {

public abstract void Draw();

}

```

3. \*\*Бесплодные (sealed) классы\*\* – запрещают дальнейшее наследование. Объявляются с `sealed`:

```csharp

public sealed class FinalClass { ... }

```

Они могут быть полезны, когда вы хотите исключить полиморфное расширение.

---

### 28. Структуры в C#

1. \*\*struct\*\* – значимый тип (value type), обычно используется для небольших “простых” объектов, не требующих наследования.

2. Структуры не могут наследовать от других структур или классов (но наследуют от `System.ValueType`).

3. Могут реализовывать интерфейсы.

4. Поддерживают поля, методы, свойства, конструкторы (без `public struct() { }` по умолчанию), но не поддерживают деструкторы и наследование (кроме `interface`).

5. Инициализация через конструктор либо инициализаторы.

---

### 29. Интерфейсы. Свойства интерфейсов. Реализация интерфейсов

1. \*\*Интерфейс\*\* – контракт, который определяет набор членов (методы, свойства, события, индексаторы), но не содержит реализацию (до C# 8.0).

2. В C# 8.0+ появились \*\*статические и приватные члены\*\* и \*\*реализация по умолчанию\*\* в интерфейсах, но классический подход – без реализации.

3. \*\*Реализация интерфейса\*\* – класс/структура, которая обязана реализовать все методы и свойства, объявленные в интерфейсе.

```csharp

public interface IFoo {

void DoSomething();

}

public class Foo : IFoo {

public void DoSomething() { ... }

}

```

---

### 30. Явная и неявная реализация интерфейсов. Работа с объектами через интерфейсы

1. \*\*Неявная реализация\*\* – члены интерфейса объявляются как обычные публичные методы/свойства класса:

```csharp

public void IFoo.DoSomething() { ... } // вариант явной реализации

public void DoSomething() { ... } // вариант неявной реализации

```

При неявной реализации метод доступен как `obj.DoSomething()`.

2. \*\*Явная реализация\*\* – методы интерфейса указываются вместе с именем интерфейса, и они не видны извне через переменную класса, только через переменную интерфейса:

```csharp

void IFoo.DoSomething() { ... }

```

Чтобы вызвать, нужно привести к интерфейсу: `((IFoo)obj).DoSomething();`.

---

### 31. Ковариантность интерфейсов. Контравариантность интерфейсов

- \*\*Ковариантность\*\* (out-параметр): позволяет использовать более “конкретный” тип, чем задан в интерфейсе. Например, `IEnumerable<out T>`.

- Это значит: `IEnumerable<string>` можно присвоить переменной типа `IEnumerable<object>`.

- \*\*Контравариантность\*\* (in-параметр): позволяет использовать более “обобщённый” тип, например, `IComparer<in T>`.

- `IComparer<object>` может сравнивать `string`, поэтому его можно присвоить переменной `IComparer<string>`.

- Ключевые слова `out` и `in` в обобщённых интерфейсах определяют ковариантность и контравариантность.

---

### 32. Стандартные интерфейсы .NET. Назначение и применение

1. \*\*IComparable\*\*, \*\*IComparable<T>\*\* – определяют метод `CompareTo`, используемый для сортировки.

2. \*\*IEnumerable\*\*, \*\*IEnumerable<T>\*\* – позволяют перебирать коллекции в `foreach`.

3. \*\*IEnumerator\*\*, \*\*IEnumerator<T>\*\* – для итерации по коллекции, возвращают текущий элемент и двигают курсор.

4. \*\*IDisposable\*\* – определяет метод `Dispose()` для освобождения неуправляемых ресурсов.

5. \*\*ICloneable\*\* – метод `Clone()` для создания копии объекта (не всегда рекомендуется, т.к. неясно, глубокая или поверхностная копия).

6. \*\*IEquatable<T>\*\* – метод `Equals(T other)` для типобезопасного сравнения.

---

### 33. Исключительные ситуации. Генерация и повторная генерация исключений

1. \*\*Исключение (Exception)\*\* – механизм обработки ошибок, при котором выполнение кода прерывается и управление передаётся обработчику исключений.

2. Генерация (`throw`) исключения:

```csharp

throw new ArgumentException("Invalid argument");

```

3. Повторная генерация:

```csharp

catch(Exception ex) {

// какая-то логика

throw; // бросает то же исключение, сохраняя стэк

}

```

или

```csharp

throw ex; // перезаписывает стек-трейс, обычно не рекомендуется

```

4. Исключения бывают системные (`System.Exception` и наследники) и пользовательские (собственные классы).

---

### 34. Исключительные ситуации. Варианты обработки исключений. Фильтры исключений

1. \*\*try-catch-finally\*\* – основной способ:

```csharp

try {

// код, который может вызвать исключение

}

catch(SomeException ex) {

// обработка

}

finally {

// блок, который выполняется всегда (если не произошло "жёсткого" сбоя)

}

```

2. \*\*Фильтры исключений\*\* (C# 6 и выше) – позволяют отфильтровывать исключения по условию:

```csharp

catch (SomeException ex) when (ex.ErrorCode == 10) {

...

}

```

Код в `when(...)` проверяется до входа в сам блок `catch`.

---

### 35. Обобщения (generics). Свойства обобщений

1. \*\*Generics\*\* – механизм, позволяющий создавать классы, методы и интерфейсы с параметрами типов (`<T>`). Например:

```csharp

public class GenericList<T> {

private T[] items;

...

}

```

2. \*\*Свойства\*\*:

- Сильная типизация (тип определяется при использовании обобщённого типа).

- Отсутствие необходимости боксинга (напр. в Generic коллекциях).

- Повышение безопасности и производительности по сравнению с `ArrayList` и другими неуниверсальными коллекциями.

---

### 36. Концепция ограничений обобщений. Статические члены обобщений

1. \*\*Ограничения (constraints)\*\* позволяют задавать требования к типу-параметру:

- `where T : struct` (T должен быть значимым типом),

- `where T : class` (T – ссылочный тип),

- `where T : SomeBaseClass` (T – должен наследовать класс/реализовывать интерфейс),

- `where T : new()` (T должен иметь публичный конструктор без параметров).

2. \*\*Статические члены обобщений\*\* – каждый закрытый обобщённый тип (т.е. с подставленным типом T) будет иметь свою копию статических полей. Это значит, что статические поля \*не\* являются общими для всех специализаций, а привязаны к конкретному типу T.

---

### 37. Делегаты. Определение, назначение и варианты использования. Обобщенные делегаты

1. \*\*Делегат\*\* – тип-обёртка для метода, позволяющая хранить ссылку на метод и вызывать его. Объявляется с помощью сигнатуры:

```csharp

public delegate int MyDelegate(string s);

```

2. \*\*Назначение\*\*:

- Передача методов как параметров в другие методы (callback).

- Событийная модель (события используют делегаты).

3. \*\*Обобщённые делегаты\*\* позволяют указать параметры и возвращаемый тип как типы `<T>`:

```csharp

public delegate T Transformer<T>(T arg);

```

4. Делегаты могут ссылаться на несколько методов (multicast delegate). Порядок вызова определяется порядком добавления.

---

### 38. Анонимные функции. Лямбда-выражения

1. \*\*Анонимная функция\*\* – блок кода без имени, который можно присвоить делегату или переменной:

```csharp

delegate(int x) { return x \* x; }

```

2. \*\*Лямбда-выражения\*\* – более краткая форма записи:

```csharp

x => x \* x

```

- Позволяют легко описывать “инлайн”-логику.

- Могут использоваться в LINQ, в методах высшего порядка.

- Лямбда-выражения могут быть как `statement-lambda` (с телом в `{}`), так и `expression-lambda` (короткая запись).

---

### 39. Обобщённые делегаты .NET. Action, Func, Predicate

1. \*\*Action<T>\*\* – делегат, который ничего не возвращает, но принимает параметры. Пример: `Action<int>` – метод, принимающий `int`, не возвращающий значения.

2. \*\*Func<T, TResult>\*\* – делегат, принимающий параметры (до 16 параметров в .NET) и возвращающий `TResult`. Пример: `Func<int, string>` – метод, принимающий `int` и возвращающий `string`.

3. \*\*Predicate<T>\*\* – делегат, принимающий один параметр типа `T` и возвращающий `bool`. Часто используется для фильтрации списков.

---

### 40. События и делегаты

1. \*\*Событие (event)\*\* – механизм, основанный на делегатах, который обеспечивает инкапсуляцию подписки/отписки обработчиков. Объявляется обычно так:

```csharp

public event EventHandler SomethingHappened;

```

или с обобщённым делегатом:

```csharp

public event EventHandler<MyEventArgs> OnDataReceived;

```

2. Класс, в котором объявлено событие (источник события), вызывает это событие (`SomethingHappened?.Invoke(this, EventArgs.Empty)`), а внешние объекты подписываются на событие (`obj.SomethingHappened += HandlerMethod`).

3. Таким образом реализуется стандартная паттерн-наблюдатель (Publisher-Subscriber) модель.

---

### 41. Стандартные коллекции .NET. Типы коллекций

1. \*\*ArrayList\*\*, \*\*Hashtable\*\* – устаревшие неуниверсальные коллекции (до .NET 2.0).

2. \*\*List<T>\*\*, \*\*Dictionary<TKey, TValue>\*\*, \*\*Queue<T>\*\*, \*\*Stack<T>\*\*, \*\*LinkedList<T>\*\* – типичные обобщённые коллекции.

3. \*\*ObservableCollection<T>\*\*, \*\*BindingList<T>\*\* – коллекции, которые уведомляют об изменениях (чаще используются в UI/данных).

4. \*\*ConcurrentBag<T>\*\*, \*\*ConcurrentQueue<T>\*\*, \*\*ConcurrentDictionary<TKey,TValue>\*\* – потокобезопасные коллекции (для параллельных сценариев).

---

### 42. Стандартные интерфесы коллекций

1. \*\*ICollection\*\*, \*\*ICollection<T>\*\* – базовый набор методов и свойств для коллекций (Count, Add, Remove и т.д.).

2. \*\*IList\*\*, \*\*IList<T>\*\* – коллекция, к элементам которой можно обращаться по индексу.

3. \*\*IDictionary\*\*, \*\*IDictionary<TKey, TValue>\*\* – коллекция пар ключ-значение.

4. \*\*IReadOnlyList<T>\*\*, \*\*IReadOnlyCollection<T>\*\*, \*\*IReadOnlyDictionary<TKey,TValue>\*\* – “только для чтения” версии.

---

### 43. IEnumerable и IEnumerator

1. \*\*IEnumerable\*\* – интерфейс, предоставляющий метод `GetEnumerator()`. Позволяет использовать `foreach`.

2. \*\*IEnumerator\*\* – интерфейс итератора, возвращаемого методом `GetEnumerator()`. Имеет:

- `MoveNext()` – продвижение к следующему элементу,

- `Current` – текущий элемент,

- `Reset()` – сброс (не всегда реализован).

3. Для `foreach` компилятор под капотом использует `GetEnumerator()` и вызывает `MoveNext()`/`Current`.

---

### 44. LINQ to Objects. Синтаксис. Форма. Возврат результата. Грамматика выражений запросов. Отложенные и неотложенные операции

1. \*\*LINQ to Objects\*\* – это способ писать запросы к коллекциям (IEnumerable<T>) на C#.

2. Два основных синтаксиса:

- \*\*Выражения запроса\*\* (query expressions):

```csharp

var result = from x in collection

where x > 10

select x\*x;

```

- \*\*Метод-цепочки\*\* (method syntax, “fluent API”):

```csharp

var result = collection

.Where(x => x > 10)

.Select(x => x \* x);

```

3. \*\*Возврат результата\*\* может быть в виде `IEnumerable<T>` (отложенное выполнение) или с материализацией в массив/список (`ToArray()`, `ToList()`).

4. \*\*Отложенные операции (deferred)\*\* – `Where()`, `Select()` и т.д. не выполняются сразу, а только при переборе.

5. \*\*Неотложенные\*\* – `Count()`, `ToList()`, `Sum()` и т.д. выполняются сразу и возвращают окончательный результат.

---

### 45. LINQ to Objects. Операции Where, Select, Take, OrderBy, Join, GroupBy

1. \*\*Where\*\* – фильтрация:

```csharp

collection.Where(x => x > 10)

```

2. \*\*Select\*\* – проекция (преобразование):

```csharp

collection.Select(x => x \* x)

```

3. \*\*Take\*\* – взять первые n элементов:

```csharp

collection.Take(5)

```

4. \*\*OrderBy / OrderByDescending\*\* – сортировка:

```csharp

collection.OrderBy(x => x).ThenBy(x => x.Prop)

```

5. \*\*Join\*\* – соединение двух последовательностей по ключу:

```csharp

collectionA.Join(collectionB,

a => a.Id,

b => b.Id,

(a,b) => new { a, b });

```

6. \*\*GroupBy\*\* – группировка по ключу:

```csharp

collection.GroupBy(x => x.Category)

.Select(g => new { Category = g.Key, Items = g.ToList() });

```

---

### 46. Рефлексия. System.Type

1. \*\*Рефлексия\*\* – механизм исследования метаданных о типах в процессе выполнения программы.

2. В C# рефлексия реализована через пространство имён `System.Reflection`.

3. \*\*System.Type\*\* – основной класс, предоставляющий методы для анализа типа (получить поля, методы, свойства, конструкторы). Пример:

```csharp

Type t = typeof(MyClass);

MethodInfo[] methods = t.GetMethods();

```

4. Позволяет динамически создавать объекты (`Activator.CreateInstance`) и вызывать методы, что даёт гибкость, но может снижать производительность и безопасность.

---

### 47. Классы для работы с файловой системой

1. \*\*System.IO.File\*\*, \*\*System.IO.FileInfo\*\* – для операций с файлами (создание, копирование, удаление, проверка существования).

2. \*\*System.IO.Directory\*\*, \*\*System.IO.DirectoryInfo\*\* – для операций с каталогами (создание, перечисление файлов, удаление и т.д.).

3. \*\*Path\*\* – вспомогательные методы для работы с путями (соединение путей, получение расширений, имён файлов).

---

### 48. Синтаксическая конструкция `using`. Чтение и запись файлов. Потоковые классы

1. \*\*Ключевое слово `using`\*\* – гарантирует вызов `Dispose()` по окончании блока, часто применимо к файловым и сетевым потокам:

```csharp

using (var stream = new FileStream("file.txt", FileMode.Open)) {

// работа с потоком

} // здесь Dispose() вызывается автоматически

```

2. \*\*Чтение файлов\*\*: `File.ReadAllText()`, `File.ReadAllLines()`, `StreamReader`.

3. \*\*Запись файлов\*\*: `File.WriteAllText()`, `File.WriteAllLines()`, `StreamWriter`.

4. \*\*Основные потоковые классы\*\*: `Stream`, `FileStream`, `MemoryStream`, `NetworkStream` и т.д.

---

### 49. Классы адаптеры потоков

- \*\*StreamReader\*\* / \*\*StreamWriter\*\* – адаптируют `Stream` для чтения/записи текстовых данных.

- \*\*BinaryReader\*\* / \*\*BinaryWriter\*\* – адаптируют `Stream` для чтения/записи двоичных данных (int, double и т.д.).

- Позволяют работать с потоком более высокоуровнево, предоставляя удобные методы чтения/записи.

---

### 50. Сериализация. Форматы сериализации

1. \*\*Сериализация\*\* – процесс преобразования объекта в поток байтов (или другой формат) для хранения или передачи по сети.

2. \*\*Десериализация\*\* – обратный процесс восстановления объекта.

3. Форматы:

- \*\*Бинарная\*\* (устаревший BinaryFormatter, сейчас не рекомендуется),

- \*\*XML\*\* (XmlSerializer),

- \*\*JSON\*\* (System.Text.Json, Newtonsoft.Json),

- \*\*SOAP\*\* (раньше использовался, сейчас устарел).

4. В .NET Core/.NET 5+ наиболее актуальны `System.Text.Json` и `Newtonsoft.Json`.

---

### 51. Сериализация контрактов данных. Интерфейс ISerializable

1. \*\*Сериализация контрактов данных\*\* (DataContractSerializer) – используется в WCF и подобных сценариях, атрибуты `[DataContract]` и `[DataMember]` определяют, что сериализовать:

```csharp

[DataContract]

public class Person {

[DataMember]

public string Name {get;set;}

}

```

2. \*\*ISerializable\*\* – интерфейс, позволяющий вручную управлять сериализацией (через методы `GetObjectData` и специальный конструктор):

```csharp

public void GetObjectData(SerializationInfo info, StreamingContext context) {

info.AddValue("Name", this.Name);

}

```

---

### 52. Атрибуты. Создание собственного атрибута

1. \*\*Атрибуты\*\* – механизмы, позволяющие добавлять метаданные к программным элементам (классам, методам, свойствам).

2. В .NET атрибуты – это классы, наследующиеся от `System.Attribute`.

3. \*\*Создание собственного атрибута\*\*:

```csharp

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]

public class MyAttribute : Attribute {

public string Info { get; }

public MyAttribute(string info) {

Info = info;

}

}

```

4. Затем можно использовать:

```csharp

[My("Some info")]

public class TestClass {

...

}

```

---

### 53. Процесс. Домен приложений. Поток выполнения

1. \*\*Процесс\*\* (Process) – изолированная единица выполнения приложения в операционной системе.

2. \*\*Домен приложений (AppDomain)\*\* – в .NET Framework был механизм для ещё более тонкой изоляции, внутри одного процесса могло быть несколько AppDomain. В .NET Core/5+ концепция AppDomain упрощена/изменена.

3. \*\*Поток (Thread)\*\* – наименьшая единица исполнения внутри процесса. Процесс может содержать один или несколько потоков.

---

### 54. Создание потоков, классы приоритетов. Состояния потоков

1. \*\*Создание потока\*\*:

```csharp

Thread thread = new Thread(SomeMethod);

thread.Start();

```

2. \*\*Приоритеты\*\* потока: `ThreadPriority.Highest`, `AboveNormal`, `Normal`, `BelowNormal`, `Lowest`. Обычно оставляют `Normal`.

3. \*\*Состояния\*\* потока: `Unstarted`, `Running`, `WaitSleepJoin`, `Stopped`, `Background`, `Aborted` (устарел). Многие состояния внутренние для CLR и не все доступны напрямую.

---

### 55. Синхронизация потоков. Lock. Monitor. Mutex. Semaphore

1. \*\*Lock\*\* – упрощённая форма `Monitor.Enter/Exit`.

```csharp

private object locker = new object();

lock(locker) {

// критическая секция

}

```

2. \*\*Monitor\*\* – класс с методами `Enter`, `Exit`, `Wait`, `Pulse`, даёт низкоуровневый контроль синхронизации.

3. \*\*Mutex\*\* – межпроцессная синхронизация, может использоваться для блокировки ресурсов между разными процессами.

4. \*\*Semaphore\*\* / \*\*SemaphoreSlim\*\* – позволяет ограничить доступ к ресурсу нескольким потокам одновременно (счётчик пропускной способности).

---

### 56. Библиотека параллельных задач TPL. Класс Task. Состояние задачи

1. \*\*TPL (Task Parallel Library)\*\* – высокоуровневая библиотека для параллельного программирования (пространство имён `System.Threading.Tasks`).

2. \*\*Task\*\* – представляет асинхронную операцию. Может быть создана и запущена для выполнения кода в пуле потоков.

3. \*\*Состояния задачи\*\* – `Created`, `WaitingToRun`, `Running`, `Completed`, `Faulted`, `Canceled`.

- Доступны через свойство `Task.Status`.

---

### 57. Способы создания Task. Возврат результата. Отмена выполнения задач. Продолжения

1. \*\*Создание Task\*\*:

- `Task.Factory.StartNew(Action)`,

- `new Task(Action).Start()`,

- `Task.Run(Action)`. (наиболее часто используемый способ).

2. \*\*Возврат результата\*\*: `Task<TResult>` – возвращает значение типа `TResult`:

```csharp

Task<int> t = Task.Run(() => 42);

int result = t.Result; // блокирующее получение результата

```

3. \*\*Отмена задач\*\*: через `CancellationTokenSource` и `CancellationToken`. Код задачи должен периодически проверять `token.IsCancellationRequested`.

4. \*\*Продолжения\*\*: `t.ContinueWith(...)` – запуск следующей задачи после завершения предыдущей.

---

### 58. Параллелизм при императивной обработке данных. Класс `Parallel`

1. Класс \*\*Parallel\*\* (пространство имён `System.Threading.Tasks`) предоставляет статические методы, упрощающие параллельные операции над коллекциями:

- `Parallel.For`, `Parallel.ForEach` – цикл for или foreach, выполняемый в параллельных потоках.

2. Может ускорять вычисления на многоядерных процессорах, если итерации независимы друг от друга.

3. Нужно учитывать потокобезопасность при изменении общих данных внутри параллельных циклов.

---

### 59. Асинхронные методы. `async` и `await`

1. \*\*Асинхронные методы\*\* упрощают написание асинхронного кода. Объявляются ключевым словом `async`:

```csharp

public async Task SomeMethodAsync() {

await Task.Delay(1000);

// ...

}

```

2. `await` приостанавливает выполнение метода до завершения ожидаемой задачи, но не блокирует поток – даёт возможность выполнять другие задачи.

3. \*\*Возвращаемый тип\*\* асинхронного метода: `Task`, `Task<T>` или `void` (для обработчиков событий, но не рекомендуется).

4. При использовании `await` метод компилируется в состояние-машину, которая обрабатывает возобновление кода после окончания асинхронной операции.

---

### 60. Проектирование отношений. Агрегация, композиция и ассоциация

1. \*\*Ассоциация\*\* – “просто связь” между объектами, без жёсткой принадлежности (например, класс `Student` и класс `Teacher`: у студента может быть учитель, но они существуют независимо).

2. \*\*Агрегация\*\* – “слабая композиция”: один объект (контейнер) владеет другим, но «владеемый» может существовать отдельно. Пример: класс `Car` и объект `Engine` – можно вынуть двигатель и использовать отдельно.

3. \*\*Композиция\*\* – более сильная форма: владеемый объект не может существовать независимо от владельца. При уничтожении “владельца” уничтожаются и все составные части. Пример: класс `House` и “комнаты” – нет смысла существования комнаты вне дома в данной модели.

---

### 61. Антипаттерны проектирования. Рефакторинг. Методы рефакторинга

1. \*\*Антипаттерны\*\* – типичные ошибки и слабости в проектировании кода (God Object, Spaghetti Code, Lava Flow и др.). Они усложняют поддержку и развитие проекта.

2. \*\*Рефакторинг\*\* – последовательное улучшение внутренней структуры кода без изменения его внешнего поведения.

3. Основные методы рефакторинга:

- Извлечение метода (Extract Method),

- Перемещение метода (Move Method),

- Инкапсуляция полей (Encapsulate Field),

- Удаление дублирования (Remove Duplication),

- Замена условных операторов полиморфизмом и т.п.

---

### 62. Чистый код. Требования к именам, функциям, форматированию

1. \*\*Имена\*\*: описательные, отражающие назначение (классы, методы, переменные). Использовать единый стиль (CamelCase/PascalCase).

2. \*\*Функции (методы)\*\*: должны быть небольшими, выполнять одну задачу (Single Responsibility).

3. \*\*Форматирование\*\*: разумное использование отступов, пустых строк, ограничение длины строки, единообразие.

4. Цель – повышение читаемости и удобства поддержки кода.

---

### 63. Чистый код. Требования к классам и объектам

1. \*\*Принцип SRP (Single Responsibility Principle)\*\*: класс должен заниматься одной основной задачей.

2. \*\*Частные детали (приватность)\*\*: скрывать внутреннюю реализацию, оставлять публичными только действительно необходимые методы.

3. \*\*Модульность и понятная структура\*\*: классы разбиваются по функциональным областям, файлы и пространства имён соответствуют назначению.

4. \*\*Минимизация связности (coupling)\*\*: объекты не должны напрямую зависеть друг от друга, если это не требуется.

5. \*\*Читабельность\*\*: следование соглашениям об именовании, аккуратная и понятная иерархия, комментарии там, где нужно пояснить нестандартные решения.