### 1. Что такое обобщение (generic)?

\*\*Обобщение\*\* (generics) — это механизм в C#, позволяющий создавать универсальные классы, интерфейсы, методы и делегаты, которые могут работать с любым типом данных. Вместо конкретного типа данных используется параметр типа, который задается при создании экземпляра или вызове метода. Это повышает гибкость и повторное использование кода, а также обеспечивает типобезопасность.

Пример:

```csharp

class GenericClass<T>

{

public T Data { get; set; }

}

```

---

### 2. Пусть дан фрагмент листинга. В какой строчке содержится ошибка?

Для анализа нужно видеть сам листинг. Однако, типичные ошибки в обобщениях:

- Отсутствие указания параметра типа при создании экземпляра.

- Несоответствие типов при ограничениях (`where`).

- Использование неподходящих типов для ограничений `struct` или `class`.

Пример ошибки:

```csharp

class Test<T> where T : class { }

Test<int> obj = new Test<int>(); // Ошибка, так как int — это struct, а не class.

```

---

### 3. Как можно наложить определенное ограничение на параметр?

Используя ключевое слово `where`. Ограничения позволяют уточнить, какие типы можно использовать в качестве параметра.

Пример ограничения:

```csharp

class Test<T> where T : class

{

public T Data { get; set; }

}

```

---

### 4. Как можно наложить несколько ограничений на параметр?

Для нескольких ограничений используется ключевое слово `where` с перечислением условий через пробел.

Пример:

```csharp

class Test<T> where T : class, new()

{

public T CreateInstance()

{

return new T();

}

}

```

---

### 5. Перечислите все существующие ограничения на типы данных обобщения.

1. \*\*`class`\*\* — параметр типа должен быть ссылочным типом.

2. \*\*`struct`\*\* — параметр типа должен быть значимым типом.

3. \*\*`new()`\*\* — параметр типа должен иметь конструктор без параметров.

4. \*\*Тип или интерфейс\*\* — параметр типа должен быть либо указанным типом, либо реализовать указанный интерфейс.

5. \*\*Многоуровневое наследование\*\* — тип должен быть производным от указанного базового класса.

---

### 6. Какое ограничение на тип задано в следующем фрагменте листинга?

```csharp

class A { }

class B : A { }

class C { }

class Test<T> where T : A { }

```

\*\*Ответ\*\*: Тип `T` должен быть либо классом `A`, либо наследоваться от него (например, `B`). Тип `C` использовать нельзя, так как он не наследует `A`.

---

### 7. Какое ограничение на тип задано в следующем фрагменте листинга?

```csharp

interface A { }

class Test<T> where T : class { }

```

\*\*Ответ\*\*: Тип `T` должен быть ссылочным типом (`class`).

---

### 8. Какое ограничение на тип задано в следующем фрагменте листинга?

```csharp

interface A { }

class Test<T> where T : struct { }

```

\*\*Ответ\*\*: Тип `T` должен быть значимым типом (`struct`), то есть типом-значением (например, `int`, `float`, `DateTime`).

---

### 9. Приведите примеры, когда обобщенный класс может действовать как базовый или производный класс.

Обобщенный класс может выступать базовым классом, если нужно передать параметр типа производным классам.

Пример:

```csharp

class BaseClass<T>

{

public T Data { get; set; }

}

class DerivedClass<T> : BaseClass<T>

{

public void Print()

{

Console.WriteLine(Data);

}

}

```

---

### 10. В каких случаях в обобщениях может использоваться оператор `default`?

Оператор `default` используется для инициализации значений параметров типа, когда неизвестно, какой это тип:

- Для значимых типов возвращается `0`, `false` или `DateTime.MinValue`.

- Для ссылочных типов возвращается `null`.

Пример:

```csharp

class GenericClass<T>

{

public T GetDefault()

{

return default(T);

}

}

```

---

### 11. Поясните, как использовать статические переменные в обобщенных классах.

Статические переменные обобщенного класса принадлежат конкретному типу параметра, а не самому классу. Для каждого типа `T` создается отдельная копия статической переменной.

Пример:

```csharp

class GenericClass<T>

{

public static int Counter;

}

GenericClass<int>.Counter = 1;

GenericClass<string>.Counter = 2;

Console.WriteLine(GenericClass<int>.Counter); // 1

```

---

### 12. Приведите пример обобщенного интерфейса.

Пример:

```csharp

interface IGenericInterface<T>

{

T Process(T input);

}

class GenericImplementation : IGenericInterface<int>

{

public int Process(int input) => input \* 2;

}

```

---

### 13. В чем отличие обобщенных классов от обобщенных структур?

- \*\*Обобщенные классы\*\* — ссылочные типы, размещаются в управляемой куче.

- \*\*Обобщенные структуры\*\* — значимые типы, размещаются в стеке, что повышает производительность.

Пример обобщенной структуры:

```csharp

struct GenericStruct<T>

{

public T Data { get; set; }

}

```

---

### 14. Какие классы для работы с файлами вы знаете? Приведите пример.

Классы:

1. \*\*`File`\*\* — предоставляет методы для работы с файлами.

2. \*\*`FileInfo`\*\* — аналог `File`, но с дополнительными методами.

3. \*\*`StreamReader`/`StreamWriter`\*\* — чтение/запись текста в поток.

4. \*\*`BinaryReader`/`BinaryWriter`\*\* — работа с двоичными данными.

Пример:

```csharp

using System.IO;

class Program

{

static void Main()

{

string path = "example.txt";

File.WriteAllText(path, "Hello, world!");

string content = File.ReadAllText(path);

Console.WriteLine(content);

}

}

```

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

\*\*Делегат\*\* — это тип, который представляет ссылку на метод. Делегаты позволяют вызывать методы через эту ссылку, обеспечивая гибкость и расширяемость программ.

\*\*Определение делегата\*\*:

Делегаты объявляются с помощью ключевого слова `delegate`, указывая возвращаемый тип и сигнатуру метода, который делегат может представлять.

Пример:

```csharp

delegate int MyDelegate(int x, int y); // Делегат, который принимает два int и возвращает int

```

---

### 2. Назначение делегатов

1. \*\*Передача методов как параметров\*\*: Позволяют передавать методы в качестве аргументов другим методам.

2. \*\*Обработка событий\*\*: Делегаты используются для определения и вызова событий.

3. \*\*Многообразие вызовов\*\*: Позволяют вызывать разные методы с одинаковой сигнатурой через один делегат.

4. \*\*Реализация обратных вызовов (callbacks)\*\*: Используются для асинхронного программирования или обратных вызовов.

---

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

1. \*\*Присвоение метода вручную\*\*:

```csharp

MyDelegate del = new MyDelegate(MyMethod);

```

2. \*\*Неявное создание делегата\*\*:

```csharp

MyDelegate del = MyMethod;

```

3. \*\*Присвоение анонимного метода\*\*:

```csharp

MyDelegate del = delegate(int x, int y) { return x + y; };

```

4. \*\*Присвоение лямбда-выражения\*\*:

```csharp

MyDelegate del = (x, y) => x + y;

```

---

### 4. Поясните назначение метода `Invoke`

Метод `Invoke` используется для вызова метода, адрес которого хранится в делегате. Это фактический способ выполнить метод через делегат.

Пример:

```csharp

MyDelegate del = (x, y) => x + y;

int result = del.Invoke(3, 4); // Результат: 7

```

Сокращенный вариант (эквивалент):

```csharp

int result = del(3, 4); // Тоже вызов делегата

```

---

### 5. Что такое групповая адресация делегата?

\*\*Групповая адресация\*\* — это возможность делегатов хранить ссылки на несколько методов. При вызове делегата все методы в списке вызываются по порядку.

Пример:

```csharp

delegate void MyDelegate(string message);

void Method1(string msg) => Console.WriteLine($"Method1: {msg}");

void Method2(string msg) => Console.WriteLine($"Method2: {msg}");

MyDelegate del = Method1;

del += Method2; // Групповая адресация

del("Hello"); // Вызов обеих методов

```

Вывод:

```

Method1: Hello

Method2: Hello

```

---

### 6. Как создать событие?

Для создания события используется ключевое слово `event`. Событие — это тип, который основан на делегате.

Пример:

```csharp

delegate void MyEventHandler(string message);

class EventExample

{

public event MyEventHandler MyEvent;

public void TriggerEvent(string message)

{

MyEvent?.Invoke(message); // Вызов события

}

}

```

Использование:

```csharp

var example = new EventExample();

example.MyEvent += msg => Console.WriteLine($"Event triggered: {msg}");

example.TriggerEvent("Hello!");

```

---

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

\*\*Событие\*\* — это обертка над делегатом, обеспечивающая контроль доступа. Только класс-владелец может вызывать событие, а подписчики (подписываемые методы) могут быть добавлены или удалены.

\*\*Схема взаимодействия\*\*:

1. Событие объявляется на основе делегата.

2. Методы подписчиков добавляются через `+=`.

3. Событие вызывается (например, при наступлении какого-то условия), что вызывает все подписанные методы.

Пример:

```csharp

class Program

{

public delegate void Notify(string message);

public event Notify OnNotify;

public void RaiseEvent(string msg)

{

OnNotify?.Invoke(msg);

}

}

```

---

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

\*\*Лямбда-выражение\*\* — это краткая форма записи анонимного метода. Используется для упрощения кода, особенно в LINQ.

Пример с несколькими параметрами:

```csharp

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

Console.WriteLine(add(3, 5)); // 8

```

---

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

- \*\*Ковариантность\*\* позволяет возвращать производный тип вместо базового в методе, на который ссылается делегат.

- \*\*Контравариантность\*\* позволяет использовать более общий тип для параметров метода.

\*\*Пример ковариантности\*\*:

```csharp

delegate object MyDelegate();

string Method() => "Hello";

MyDelegate del = Method; // Ковариантность: string подходит для object

```

\*\*Пример контравариантности\*\*:

```csharp

delegate void MyDelegate(string input);

void Method(object obj) => Console.WriteLine(obj);

MyDelegate del = Method; // Контравариантность: object подходит для string

```

---

### 10. Поясните разницу между встроенными делегатами `Action` и `Func`.

1. \*\*`Action`\*\*: Представляет метод, который ничего не возвращает. Может принимать 0 или более параметров.

Пример:

```csharp

Action<string> print = msg => Console.WriteLine(msg);

print("Hello, Action!");

```

2. \*\*`Func`\*\*: Представляет метод, который возвращает значение. Последний параметр указывает возвращаемый тип.

Пример:

```csharp

Func<int, int, int> multiply = (x, y) => x \* y;

Console.WriteLine(multiply(3, 4)); // 12

```

### 1. На какие основные виды/типы делятся все коллекции .NET? Охарактеризуйте каждый из них.

Коллекции в .NET делятся на:

1. \*\*Несильно типизированные (негерерик) коллекции\*\*:

Расположены в пространстве имен `System.Collections`. Они могут содержать элементы любого типа. Использование таких коллекций может привести к проблемам с производительностью из-за операций упаковки и распаковки (`boxing` и `unboxing`).

Примеры: `ArrayList`, `Hashtable`.

2. \*\*Обобщенные (generic) коллекции\*\*:

Находятся в пространстве имен `System.Collections.Generic`. Эти коллекции типизированы, что обеспечивает безопасность типов и более высокую производительность.

Примеры: `List<T>`, `Dictionary<TKey, TValue>`, `HashSet<T>`.

3. \*\*Параллельные (конкурентные) коллекции\*\*:

Расположены в пространстве имен `System.Collections.Concurrent`. Предназначены для многопоточного использования и обеспечивают потокобезопасность.

Примеры: `ConcurrentBag<T>`, `ConcurrentQueue<T>`, `ConcurrentDictionary<TKey, TValue>`.

4. \*\*Наблюдаемые коллекции\*\*:

Используются для отслеживания изменений. Обычно применяются в связке с графическим интерфейсом для уведомления об изменениях данных.

Пример: `ObservableCollection<T>`.

---

### 2. Что такое generic-коллекции? Назовите примеры известных вам generic-коллекций.

\*\*Generic-коллекции\*\* — это коллекции, которые работают с определенным типом данных, обеспечивая безопасность типов на этапе компиляции и исключая необходимость приведения типов.

Примеры:

- `List<T>`

- `Dictionary<TKey, TValue>`

- `Queue<T>`

- `Stack<T>`

- `HashSet<T>`

---

### 3. В чем разница между `ArrayList` и `Array`?

1. \*\*`Array`\*\*:

- Фиксированного размера.

- Типизированная структура (только один тип данных).

- Более производительна, так как не использует упаковку/распаковку.

2. \*\*`ArrayList`\*\*:

- Размер динамически изменяется.

- Не типизированная структура (может хранить элементы разных типов).

- Использует операции упаковки (`boxing`) и распаковки (`unboxing`), что снижает производительность.

Пример:

```csharp

ArrayList list = new ArrayList();

list.Add(1);

list.Add("string"); // Нет проверки типов

```

---

### 4. Охарактеризуйте коллекции, которые вы использовали в своем варианте.

- \*\*`List<T>`\*\*: Динамический список, поддерживает доступ по индексу и работу с элементами одного типа. Удобен для хранения последовательностей данных.

- \*\*`Dictionary<TKey, TValue>`\*\*: Хранит пары ключ-значение, обеспечивает быструю выборку по ключу.

- \*\*`HashSet<T>`\*\*: Коллекция уникальных элементов.

---

### 5. Чем отличаются коллекции, расположенные в пространстве имен `System.Collections.Concurrent`?

\*\*Concurrent-коллекции\*\* предназначены для многопоточного доступа. Они обеспечивают потокобезопасность без необходимости явной синхронизации.

Примеры:

- \*\*`ConcurrentDictionary<TKey, TValue>`\*\*: Потокобезопасная версия `Dictionary`.

- \*\*`ConcurrentBag<T>`\*\*: Коллекция для хранения элементов без порядка вставки.

- \*\*`ConcurrentQueue<T>` и `ConcurrentStack<T>`\*\*: Потокобезопасные очереди и стеки.

---

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

Необходимо подключить пространство имен:

```csharp

using System.Collections.Generic;

```

---

### 7. Что такое наблюдаемая коллекция? Как ее можно использовать?

\*\*Наблюдаемая коллекция (ObservableCollection<T>)\*\* — это коллекция, которая уведомляет подписчиков о своих изменениях (например, добавление, удаление элементов).

Применяется в приложениях с графическим интерфейсом (WPF), где изменения данных должны быть отражены в интерфейсе.

Пример:

```csharp

using System.Collections.ObjectModel;

ObservableCollection<string> collection = new ObservableCollection<string>();

collection.CollectionChanged += (sender, args) =>

{

Console.WriteLine("Collection changed!");

};

collection.Add("New item"); // Триггер события

```

---

### 8. Охарактеризуйте интерфейсы `IEnumerator`, `IEnumerable`. В чем отличие назначений?

1. \*\*`IEnumerable`\*\*:

- Предоставляет метод `GetEnumerator()`, который возвращает объект для итерации.

- Используется для коллекций, которые можно перечислить (например, в цикле `foreach`).

2. \*\*`IEnumerator`\*\*:

- Определяет методы для управления итерацией: `MoveNext()`, `Reset()` и свойство `Current`.

- Предназначен для реализации механизма итерации.

Пример:

```csharp

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

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

while (enumerator.MoveNext())

{

Console.WriteLine(enumerator.Current);

}

```

---

### 9. Поясните принцип работы коллекций:

\*\*a. `LinkedList<T>`\*\*:

- Реализация двусвязного списка.

- Элементы связаны ссылками на следующий и предыдущий элементы.

- Быстрая вставка и удаление, но медленный доступ по индексу.

\*\*b. `HashSet<T>`\*\*:

- Коллекция уникальных элементов без порядка.

- Использует хеширование для быстрого поиска, добавления и удаления.

\*\*c. `Dictionary<TKey, TValue>`\*\*:

- Хранит пары ключ-значение.

- Обеспечивает быстрый доступ по ключу.

\*\*d. `ConcurrentBag<T>`\*\*:

- Потокобезопасная коллекция без порядка.

- Подходит для хранения элементов в многопоточном окружении.

\*\*e. `Stack<T>` и `Queue<T>`\*\*:

- `Stack<T>`: Реализация стека (LIFO).

- `Queue<T>`: Реализация очереди (FIFO).

\*\*f. `SortedList` и `SortedList<TKey, TValue>`\*\*:

- Хранят элементы в отсортированном порядке.

- `SortedList<TKey, TValue>` — обобщенная версия, где ключи используются для сортировки.