Лекция №1. Очереди, стеки, дженерики

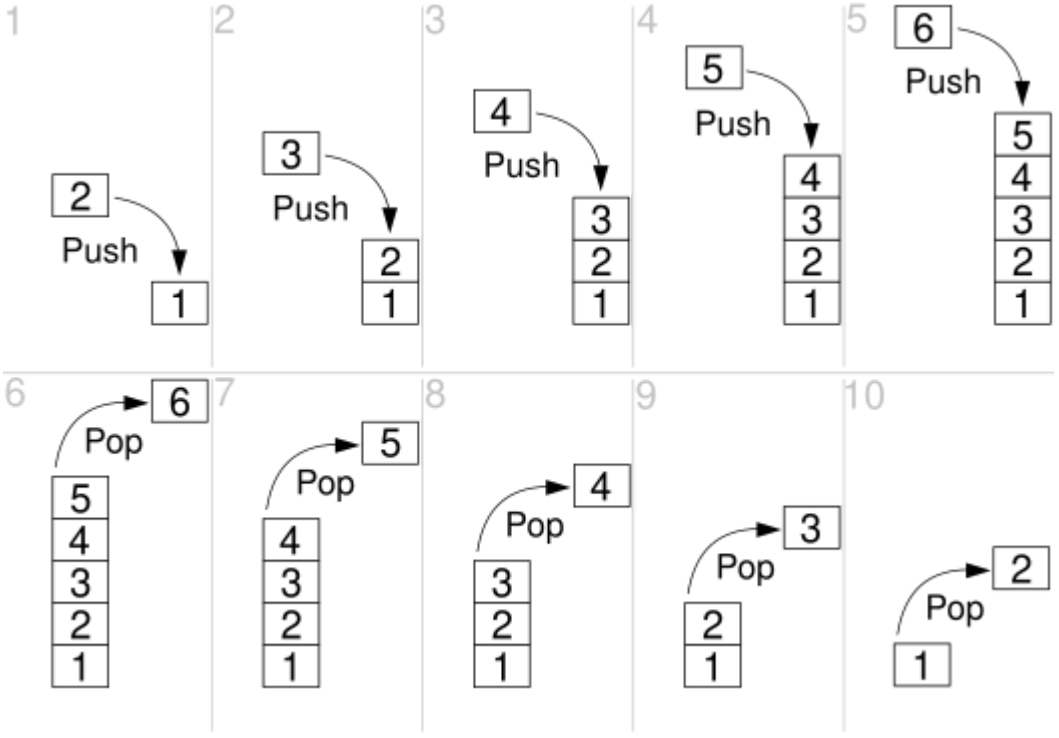
Стеки и очереди

Рассмотрим такие важные и часто используемые типы данных, как стек и очередь. Они оба являются коллекциями. Коллекция — это сущность, которая содержит в себе другие элементы. Мы уже работали с типами данных, которые являются коллекциями, — это массивы и списки (List). Элементы в стеке и очереди линейно упорядочены, их можно добавлять и удалять, но они делают это по-разному.

Стек работает по принципу LIFO (**L**ast **I**n, **F**irst **O**ut, «последним пришёл — первым вышел»). Стек можно сравнить со стопкой тарелок. Их складывают одну на другую так, что последняя тарелка оказывается на самом верху. Когда из этой стопки начнут доставать тарелки, первой достанут как раз последнюю.

Очередь работает по принципу FIFO (**F**irst **I**n, **F**irst **O**ut, «первый пришёл — первый вышел»). Если мы представим очередь на кассе в магазине, то очевидно, что первым совершит покупку и выйдет из очереди тот человек, который встал в неё первым.

Давайте попробуем сделать свою реализацию стека и очереди на C# на основе списка. Для создания стека нам понадобится реализовать два метода: `Push` и `Pop`. Метод `Push` кладёт новый элемент на верхушку стека, метод `Pop` снимает верхний элемент. Схематично это можно представить так:



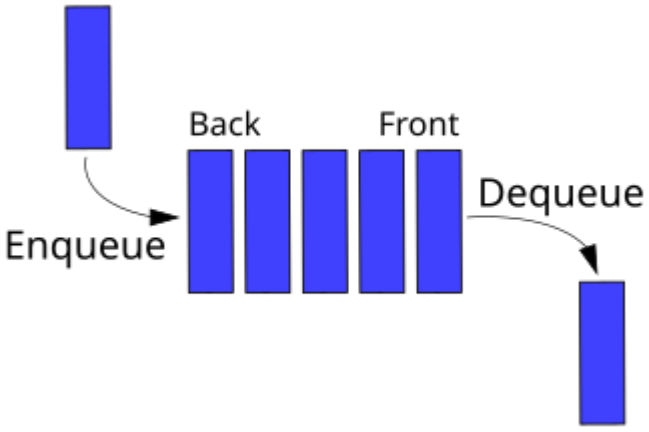
- Создадим класс `Stack`.
- Данные стека будут храниться в списке `list`.
- Метод `Push` просто добавляет новый элемент в конец списка.
- Метод `Pop` устроен немного сложнее:
  - Сначала проверяем, что список не пустой, иначе выбрасываем ошибку.
  - Получаем последний элемент в списке.
  - Удаляем последний элемент.

```
public class Stack
{
    List<int> list = new List<int>();

    public void Push(int value)
    {
        list.Add(value);
    }

    public int Pop()
    {
        if (list.Count == 0) throw new InvalidOperationException();
        var result = list[list.Count - 1];
        list.RemoveAt(list.Count - 1);
        return result;
    }
}
```

Для реализации очереди понадобится создать два метода: `Enqueue` и `Dequeue`, которые, соответственно, добавляют элемент в очередь и достают его из неё. Схематично это можно представить так:



- Создадим класс `Queue`.
- Данные очереди также будут храниться в списке `list`.
- Метод `Enqueue` аналогичен методу `Push` у стека.
- Метод `Dequeue` похож на метод `Pop`, с тем отличием, что достаём и удаляем не последний, а первый элемент в списке.
    - Удаление элемента из начала списка является неэффективной операцией.

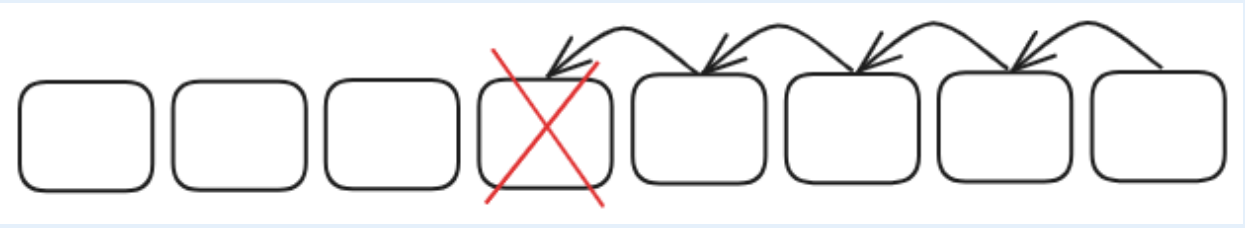
```
public class Queue
{
    List<int> list = new List<int>();

    public void Enqueue(int value)
    {
```

```
        list.Add(value);
    }

    public int Dequeue()
    {
        if (list.Count == 0) throw new InvalidOperationException();
        var result = list[0];
        // Удаление из начала списка имеет линейную сложность.
        list.RemoveAt(0);
        return result;
    }
}
```

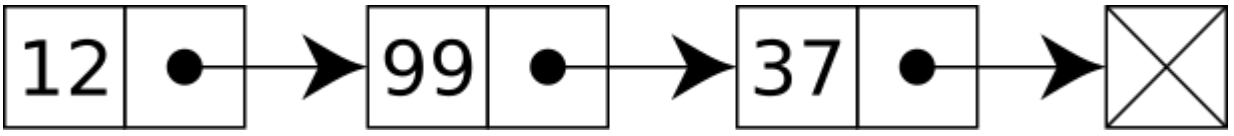
Почему удаление из начала списка — неэффективная операция? Когда мы удаляем элемент из списка, необходимо сдвинуть все элементы после него на одну позицию влево. Когда мы удаляем элемент из конца списка, нам не нужно ничего смещать. Если же мы удалим элемент из начала списка, то придётся сдвигать все элементы, которые есть в списке.



Очередь на связных списках

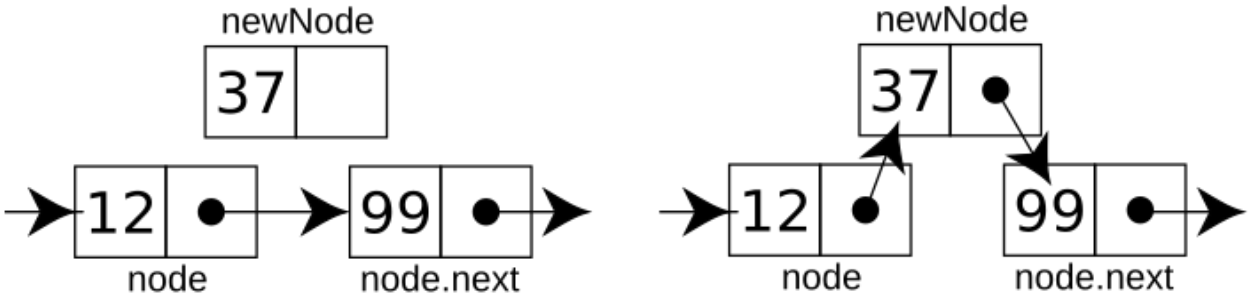
Как отмечалось выше, реализация очереди на основе списка является неэффективной. Поэтому давайте реализуем очередь на основе связного списка.

Связный список — это структура данных, которая состоит из узлов, содержащих данные и ссылки на следующий узел списка. Схематично это можно представить так:

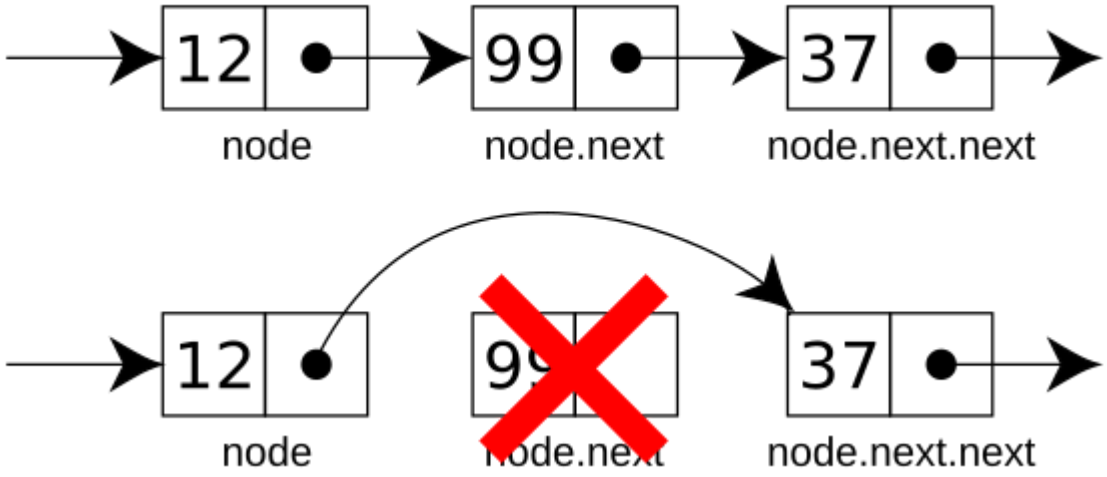


Преимуществом связного списка по сравнению с обычным списком является эффективное (за константное время) добавление и удаление элементов:

- Добавление элемента:



- Удаление элемента:



Реализация очереди на связном списке

- Создадим класс `QueueItem`, который будет являться узлом связного списка. В нём будет два свойства: `Value` и `Next`.
  - `Value` — значение, которое хранится в узле.
  - `Next` — ссылка на следующий узел.
- Создадим класс `Queue`:
  - В этом классе будет две ссылки: `head` и `tail`, которые указывают на первый и последний узел.
  - Метод `Enqueue`:
    - Создаём новый узел с переданным значением, который не будет указывать на следующий узел (`null`).
    - Если наша очередь изначально пустая, устанавливаем новый элемент как начало (`head`) и конец (`tail`) очереди.
    - Если очередь не пустая, берём текущий последний узел и создаём ссылку на только что созданный.
    - Заменяем последний узел.
  - Метод `Dequeue`:
    - Если очередь пустая — выбрасываем ошибку.
    - Если не пустая, возвращаем значение первого узла и берём следующий элемент, на который ссылается первый.

- Заменяем первый элемент на следующий.
- Если первый элемент ни на что не ссылается, это означает, что в очереди не осталось элементов.

```
public class QueueItem
{
    public int Value { get; set; }
    public QueueItem Next { get; set; }
}

public class Queue
{
    QueueItem head;
    QueueItem tail;

    public void Enqueue(int value)
    {
        var newItem = new QueueItem { Value = value, Next = null };
        if (head == null)
            tail = head = newItem;
        else
        {
            tail.Next = newItem;
            tail = newItem;
        }
    }

    public int Dequeue()
    {
        if (head == null) throw new InvalidOperationException();
        var result = head.Value;
        head = head.Next;
        if (head == null)
            tail = null;
        return result;
    }
}
```

Универсальная очередь

В предыдущем примере мы реализовали очередь, которая может хранить только целые числа. Но очередь — достаточно сложная структура данных, и хочется сделать её сразу для всех типов: очередь чисел, строк и т. д., а не переписывать для каждого типа данных. Простейшее решение — хранить `Value` в виде `object`.

```
public class QueueItem
{
```

```
    public object Value { get; set; }
    public QueueItem Next { get; set; }
}

public class Queue
{
    QueueItem head;
    QueueItem tail;

    public bool IsEmpty { get { return head == null; } }

    public void Enqueue(object value)
    {
        var newItem = new QueueItem { Value = value, Next = null };
        if (IsEmpty)
            tail = head = newItem;
        else
        {
            tail.Next = newItem;
            tail = newItem;
        }
    }

    public object Dequeue()
    {
        if (head == null) throw new InvalidOperationException();
        var result = head.Value;
        head = head.Next;
        if (head == null)
            tail = null;
        return result;
    }
}
```

Теперь нам не нужно писать реализацию очереди под каждый тип данных — у нас есть универсальная очередь.

```
public class Program
{
    static void Main()
    {
        var myIntQueue = new Queue();
        myIntQueue.Enqueue(10);
        myIntQueue.Enqueue(20);
        myIntQueue.Enqueue(30);

        int sum = 0;
        while (!myIntQueue.IsEmpty)
        {
```

```
            // Так как в очереди хранится тип object,
            // нужно делать downcast к нужному типу.
            int value = (int)myIntQueue.Dequeue();
            sum += value;
        }
    }
}
```

Но у такой реализации есть ряд минусов. При работе с коллекцией мы ожидаем, что все элементы в ней будут одного типа. Однако в данной реализации нет никаких гарантий этого, так как это очередь `object`, то есть в неё можно записать всё что угодно.

Если вернуться к примеру выше, то мы подразумевали, что эта очередь должна содержать только целые числа. Поэтому, когда мы доставали элементы из очереди, мы приводили их из типа `object` к типу `int`. Однако у этой очереди нет ограничений на запись в нее данных другого типа, например строки. В этом случае в коде произойдет ошибка.

```
public class Program
{
    static void Main()
    {
        var myIntQueue = new Queue();
        myIntQueue.Enqueue(10);
        myIntQueue.Enqueue(20);
        myIntQueue.Enqueue(30);

        //Но что, если кто-то напишет так?
        myIntQueue.Enqueue("Surprise!");

        int sum = 0;
        while (!myIntQueue.IsEmpty)
        {
            //здесь будет InvalidCastException
            int value = (int)myIntQueue.Dequeue();
            sum += value;
        }

    }
}
```

Особенно плохо то, что эта ошибка произойдет в runtime, а не на этапе компиляции. То есть из-за этой ошибки пострадают пользователи, у которых "упадет" приложение. Также проблемой является отложенная ошибка, которую будет сложно исправить. Сообщение об ошибке будет указывать на место, где мы пытаемся сделать downcast,

хотя причина заключается в том, что в очередь был добавлен неправильный тип данных, а это может произойти в совершенно другой части проекта.

Дженерик-классы

Решением проблемы, показанной в предыдущем разделе, является создание дженерик-класса. Дженерик-класс — это класс, в котором мы указываем дженерик-параметр, то есть обобщённый тип, который используется внутри класса вместо конкретного типа.

```
public class QueueItem<T> // T - это какой-то тип данных
{
    //Внутри класса QueueItem, T может использоваться везде,
    //где может использоваться тип данных:
    //при объявлении свойств, в аргументах методов, и т.д.
    public T Value { get; set; }
    public QueueItem<T> Next { get; set; }
}

public class Queue<T>
{
    QueueItem<T> head;
    QueueItem<T> tail;

    public bool IsEmpty { get { return head == null; } }

    public void Enqueue(T value)
    {
        if (IsEmpty)
            tail = head = new QueueItem<T>
            { Value = value, Next = null };
        else
        {
            var item = new QueueItem<T>
            { Value = value, Next = null };
            tail.Next = item;
            tail = item;
        }
    }

    public T Dequeue()
    {
        if (head == null) throw new InvalidOperationException();
        var result = head.Value;
        head = head.Next;
        if (head == null)
            tail = null;
        return result;
```

```
    }
}
```

При создании объектов дженерик-класса, вместо обобщенного типа нужно указать конкретный тип, который подставится на его место.

```
static void Main()
{
    var myIntQueue = new Queue<int>();
    // здесь мы создаем очередь с уже конкретным T=int.
    // всюду, где в определении класса Queue<T> был написан T,
    // для объекта myIntQueue будет как бы написан int.


    myIntQueue.Enqueue(10);
    myIntQueue.Enqueue(20);
    myIntQueue.Enqueue(30);

    // myIntQueue.Enqueue("Surprise");
    // - здесь будет ошибка компиляции, т.к. метод Enqueue принимает
значение T
    // а T для myIntQueue равно int.
}
```

Таким образом, с помощью дженерик-классов мы смогли написать очередь, которую можно использовать для любого типа данных. Если вспомнить списки, то они также реализованы с помощью дженериков.

```
List<int> list = new List<int>();
```

Использования стеков и очередей

Стеки и очереди могут использоваться в большом количестве различных ситуаций, поэтому в языке C# уже существуют стандартные классы `Stack` и `Queue` в пространстве имен `System.Collections`, благодаря чему не нужно реализовывать их каждый раз заново.

Давайте рассмотрим следующую задачу, которую можно решить с помощью стека: нам дана последовательность скобок различных видов `()`, `[]`, `{}`, и нужно проверить корректность этого выражения. Корректным оно считается, когда у каждой открывающей скобки есть соответствующая закрывающая. Алгоритм следующий:

- Идем по всем символам в строке.
    - Если встречаем открывающую скобку, добавляем её в стек.

    - Если встречаем закрывающую скобку, то проверяем элемент, находящийся на вершине стека:
        - Если там находится открывающая скобка того же типа — снимаем её со стека и идем дальше.
        - Если там находится скобка другого типа или стек пустой — выражение некорректное.
- Если по завершению цикла стек пуст — выражение корректное.
- Иначе — выражение некорректное.

```
public static bool IsCorrectString(string str)
{
    var stack = new Stack<char>();
    foreach (var e in str)
    {
        switch (e)
        {
            case '[':
            case '(':
                stack.Push(e);
                break;

            case ']':
                if (stack.Count == 0)
                    return false;
                if (stack.Pop() != '[')
                    return false;
                break;

            case ')':
                if (stack.Count == 0)
                    return false;
                if (stack.Pop() != '(')
                    return false;
                break;
            default:
                return false;
        }
    }
    return stack.Count == 0;
}
```

Этот алгоритм можно записать короче, с соблюдением принципа DRY:

```
public static bool IsCorrectString(string str)
{
    var pairs = new Dictionary<char, char>();
    pairs.Add('(', ')');
```

```
    pairs.Add('[', ']');
    var stack = new Stack<char>();
    foreach (var e in str)
    {
        if (pairs.ContainsKey(e)) stack.Push(e);
        else if (pairs.ContainsValue(e))
        {
            if (stack.Count == 0 || pairs[stack.Pop()] != e)
                return false;
        }
        else return false;
    }
    return stack.Count == 0;
}
```

Дженерики и сортировка массивов

Рассмотрим на примере сортировки массивов возможности дженериков. Отсортировать можно только те элементы, которые можно сравнивать друг с другом, то есть реализующие интерфейс `IComparable`. С помощью ключевого слова `where` мы можем наложить ограничения на тип, используемый в дженерике.

```
class Sorter<T>
    where T : IComparable // это - требование: в качестве T может быть
только класс
// реализующий IComparable
{
    public static void Sort(T[] array)
    {
        for (int i = array.Length - 1; i > 0; i--)
        for (int j = 1; j <= i; j++)
        {
            var element1 = array[j - 1];
            var element2 = array[j];
            //здесь не нужен каст к интерфейсу,
            //поскольку element1 имеет тип T,
            //а для него есть требование о том,
            //что IComparable должен быть реализован
            if (element1.CompareTo(element2) < 0)
            {
                array[j - 1] = element2;
                array[j] = element1;
            }
        }
    }
}
```

Посмотрим, как можно использовать этот класс:

```
class Point
{
    public int X { get; set; }
    public int Y { get; set; }
}

public class Program
{

    static void Main()
    {
        var intArray = new int[] { 1, 2, 3 };
        var pointArray = new[] { new Point { X = 1, Y = 2 }, new Point { X =
2, Y = 1 } };
        Sorter<int>.Sort(intArray);
        //отсортирует, т.к. int реализует IComparable
        //Sorter<Point>.Sort(pointArray);
        // покажет ошибку, т.к. Point не реализует IComparable
    }
}
```

В C# кроме дженерик-классов можно также использовать и дженерик-методы. Это синтаксический сахар, который позволяет немного упростить код за счет автоматического вывода типа в зависимости от переданного значения.

```
public static class Sorter
{
    //Это - дженерик метод. Компилятор превращает его в обычный метод
    //внутри дженерик-класса (то есть так, как написано выше)
    public static void Sort<T>(T[] array)
        where T : IComparable
    {
        for (int i = array.Length - 1; i > 0; i--)
        for (int j = 1; j <= i; j++)
        {
            var element1 = array[j - 1];
            var element2 = array[j];
            if (element1.CompareTo(element2) < 0)
            {
                array[j - 1] = element2;
                array[j] = element1;
            }
        }
    }
}
```

```
Sorter.Sort<int>(intArray); //при использовании дженерик-метода тип нужно указывать не у класса, а у метода
Sorter.Sort(intArray); //но можно и не писать тип, T автоматически выводится из типа переданного массива
//Sorter.Sort(pointArray); выдаст ошибку
```

Можно сделать generic extension метод, который будет доступен у всех массивов, чьи элементы реализуют `IComparable`.

```
public static class Sorter
{
    public static void BubbleSort<T>(this T[] array)
        where T : IComparable
    {
        for (int i = array.Length - 1; i > 0; i--)
        for (int j = 1; j <= i; j++)
        {
            var element1 = array[j - 1];
            var element2 = array[j];
            if (element1.CompareTo(element2) < 0)
            {
                array[j - 1] = element2;
                array[j] = element1;
            }
        }
    }
}
```

```
intArray.BubbleSort(); //теперь можно вызвать метод непосредственно из массива
//pointArray.BubbleSort(); выдаст ошибку
```

Кортежи

В языке C# функция может возвращать только одно значение. Но иногда возникает необходимость вернуть сразу несколько значений из одной функции. Для этого используются различные подходы, такие как создание объектов для упаковки нескольких значений. Рассмотрим этот способ:

```
class Result
{
    public int Sum { get; set; }
    public int Product { get; set; }
}

class Program
```

```
{
    static Result Calculate(int a, int b)
    {
        return new Result { Sum = a + b, Product = a * b };
    }

    static void Main()
    {
        var result = Calculate(3, 4);
        Console.WriteLine($"Sum: {result.Sum}, Product: {result.Product}");
    }
}
```

Здесь для возврата нескольких значений создается отдельный класс `Result`. Это решение работает, но требует явного определения типа, что увеличивает объем кода. В данной ситуации лучше использовать специальный тип данных — **кортежи**. Кортеж — это структура данных, которая позволяет объединить несколько значений разного типа в одну сущность без необходимости создания отдельного класса или структуры. Для создания кортежей в C# используется дженерик-класс `Tuple`.

```
class Program
{
    static Tuple<int, int> Calculate(int a, int b)
    {
        return Tuple.Create(a + b, a * b);
    }

    static void Main()
    {
        var result = Calculate(3, 4);
        Console.WriteLine($"Sum: {result.Item1}, Product: {result.Item2}");
    }
}
```

Недостаток такого подхода — обращение к элементам через свойства `Item1`, `Item2`, что делает код менее читаемым. Поэтому в современных версиях C# используется другой тип кортежей — `ValueTuple`. Он позволяет задавать имена возвращаемым значениям, благодаря чему код становится более читаемым и имеет более простой синтаксис.

```
class Program
{
    static (int Sum, int Product) Calculate(int a, int b)
    {
        return (a + b, a * b);
    }
```

```
    static void Main()
    {
        var result = Calculate(3, 4);
        Console.WriteLine($"Sum: {result.Sum}, Product: {result.Product}");
    }
}
```

Nullable

Типы значений в C# (например, `int`, `double`, `bool`) не могут содержать значение `null`. Это создает неудобства, когда нужно указать отсутствие значения для таких типов (например, при работе с базами данных или необязательными параметрами). Для решения этой проблемы используется специальная обертка — `Nullable<T>`.

```
class Program
{
    static Nullable<int> GetNullableInt(bool condition)
    {
        if (condition)
            return 42;
        else
            return null;
    }

    static void Main()
    {
        Nullable<int> result = GetNullableInt(false);

        if (result.HasValue)
            Console.WriteLine($"Value: {result.Value}");
        else
            Console.WriteLine("No value (null)");
    }
}
```

Класс `Nullable<T>` позволяет обернуть тип значений, чтобы он мог принимать значение `null`. Свойство `HasValue` позволяет проверить, было ли задано значение, а `Value` возвращает само значение (если оно присутствует).

В новых версиях C# был добавлен удобный синтаксис для работы с nullable-типами. Использование `int?` означает, что это тип `int`, который может принимать значение `null`. Это сокращает запись и делает код более читаемым.

```
class Program
{
    static int? GetNullableInt(bool condition)
```

```
    {
        if (condition)
            return 42;
        else
            return null;
    }

    static void Main()
    {
        int? result = GetNullableInt(false);

        if (result != null)
            Console.WriteLine($"Value: {result}");
        else
            Console.WriteLine("No value (null)");
    }
}
```