Иллюстрации к курсу лекций по дисциплине «Программирование на C#»

Упаковка / распаковка.

Паттерн одиночка (Singleton)

Записи.

Кортежи. Коллекции.

Упаковка и распаковка. Когда возникает?

Операция присваивания:
target = value

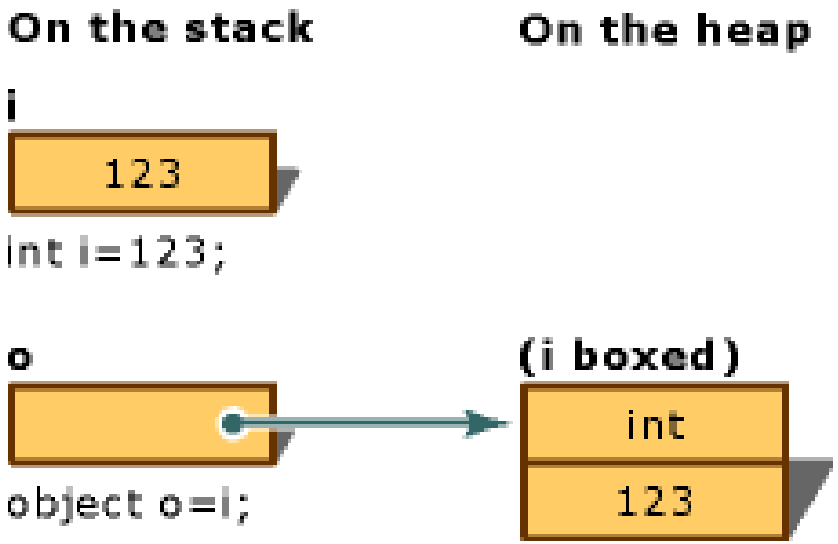
	Target	***Value***
1.	*Val*	*Val*
2.	*Ref*	*Ref*
3.	*Ref*	*Val (**boxing** – **упаковка**)*
4.	*Val*	*Ref (**unboxing** – **распаковка**)*

Явное приведение типов при распаковке:
target = (тип_target) value

Упаковка (boxing)

- **Значимый тип => Object;**
- **Должен быть создан новый объект и размещен в куче;**
- **Дорогая операция (в вычислительном смысле);**

```
int i = 123;
object o = i; // неявная упаковка
```

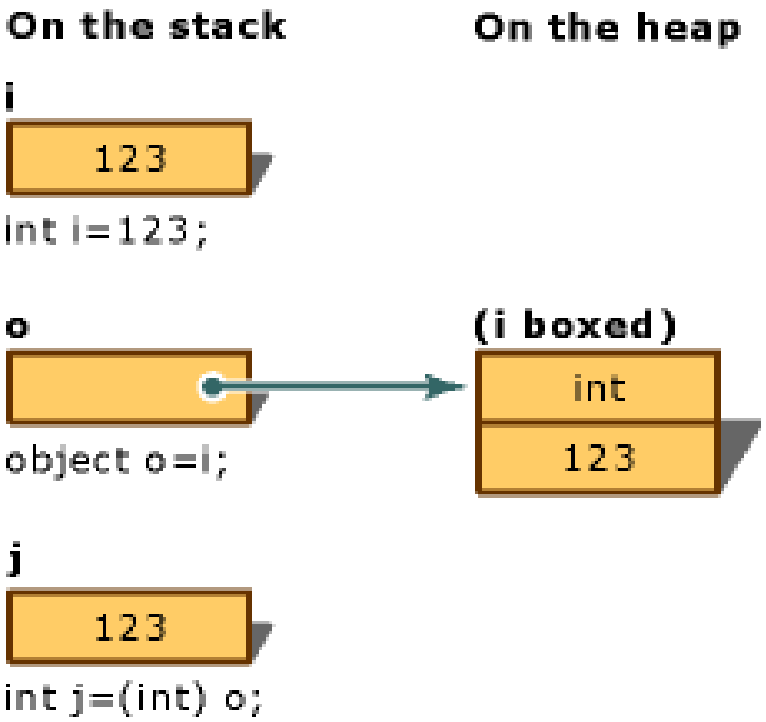


https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/types/boxing-and-unboxing

Распаковка (unboxing)

- **Object => Значимый тип;**
- **Дорогая операция (в вычислительном смысле);**

```
int i = 123;
object o = i; // неявная упаковка
int j = (int)o; // распаковка
```



Потери в производительности при упаковке

```
const int count = 100_000_000;

Console.WriteLine("С упаковкой: {0} мс",                // 1072
DoTest(count, () => {
    int value = 123;
    object objValue = value;
}));

Console.WriteLine("Без упаковки: {0} мс",               // 520
DoTest(count, () => {
    int value = 123;
    int value2 = value;
}));

static long DoTest(int count, Action action) {
    var sw = Stopwatch.StartNew();
    for (int i = 0; i < count; i++) action();
    return sw.ElapsedMilliseconds;
}
```

Типы значений, допускающие null (Nullable<T>)

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

int?[] arr = new int?[10];

int? b = 10;
if (b.HasValue)
    Console.WriteLine($"b is {b.Value}");   // b is 10
else
    Console.WriteLine("b does not have a value");
```

https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/nullable-value-types

Типы значений, допускающие null (Nullable<T>)

```
int? a = 28;
int b = a ?? -1;
Console.WriteLine($"b is {b}");  // b is 28

int? c = null;
int d = c ?? -1;
Console.WriteLine($"d is {d}");  // d is -1

int? n = null;

//int m1 = n;    // не скомпилируется
int n2 = (int)n; // скомпилируется, но выбросит исключение
```

https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/nullable-value-types

Упаковка и распаковка Nullable<T>

Экземпляр типа значения, допускающего значение null, упаковывается следующим образом:

- Если HasValue возвращает false, создается пустая ссылка.
- Если HasValue возвращает true, упаковывается соответствующее значение базового типа T, а не экземпляр Nullable<T>.

```
int a = 41;
object aBoxed = a;
int? aNullable = (int?)aBoxed;
Console.WriteLine($"aNullable: {aNullable}"); // 41

object aNullableBoxed = aNullable;
if (aNullableBoxed is int valueOfA)
{
    Console.WriteLine($"aNullableBoxed is boxed int: {valueOfA}"); // 41
}
```

https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/nullable-value-types

Запись (Record)

Запись – это класс, в котором компилятор:

• добавляет защищенный конструктор копирования (и скрытый метод Clone), для неразрушающего изменения (nondestructive mutation).

• переопределяет функционал, связанный со сравнением для обеспечения структурного равенства (structural equality).

• переопределяет метод ToString() для отображения всех общедоступных свойств, как в анонимных типах: <record type name> { <property name> = <value>, <property name> = <value>, ...}

```
record Point
{
    public Point(double x, double y)
        => (X, Y) = (x, y);    // { this.X = x; this.Y = y; }

    public double X { get; init; }

    public double Y { get; init; }
}
```

https://docs.microsoft.com/ru-ru/dotnet/csharp/whats-new/tutorials/records

Итоговый код записи

```
class Point
{
    public Point(double x, double y) => (X, Y) = (x, y);

    public double X { get; init; }

    public double Y { get; init; }

    protected Point(Point original) { // Copy constructor
        this.X = original.X; this.Y = original.Y
    }

    // Метод со "странным" именем:
    public virtual Point<Clone>$() => new Point(this); // метод Clone

    // код для переопределения Equals, ==, !=, GetHashCode, ToString()
}
```

Опциональные параметры и совместимость

```
// Старый код:
new Foo(123, 234);

record Foo
{
    public Foo(int required1, int required2) { //...
    }

    public int Required1 { get; init; }
    public int Required2 { get; init; }

    public int Optional1 { get; init; } // не добавляйте в конструктор
    public int Optional2 { get; init; } // не добавляйте в конструктор
}

// Новый код:
new Foo(123, 234) { Optional2 = 345 };
```

Позиционный синтаксис для определения свойств

```
record PositionalPoint(double X, double Y)
{
    // Дополнительные члены класса (опциональные)
}
```

Параметры могут включать модификаторы in и params (но не out или ref). При указании параметров записи компилятор выполняет дополнительные действия:

- добавляет свойства для каждого параметра (init-only) property per parameter.
- Определяет конструктор для инициализации всех свойств-параметров.
- Определяет деконструктор:

```
public void Deconstruct(out double X, out double Y) // Deconstructor
{
    X = this.X;
    Y = this.Y;
}
```

Наследование записей

```
record Point3D(double X, double Y, double Z) : Point(X, Y);
```

На самом деле компилятор создает класс:

```
class Point3D : Point
{
    public double Z { get; init; }

    public Point3D(double X, double Y, double Z) : base(X, Y)
       => this.Z = Z;
}
```

Обратите внимание:

Нельзя наследовать классы от записей и записи от классов!

Наследование записей и эквивалентность

```
public abstract record Person(string FirstName, string LastName);

public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

static void Main(string[] args)
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Person student = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(teacher == student); // False

    Student student2 = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(student2 == student); // True
}
```

Неразрушающее изменение записей

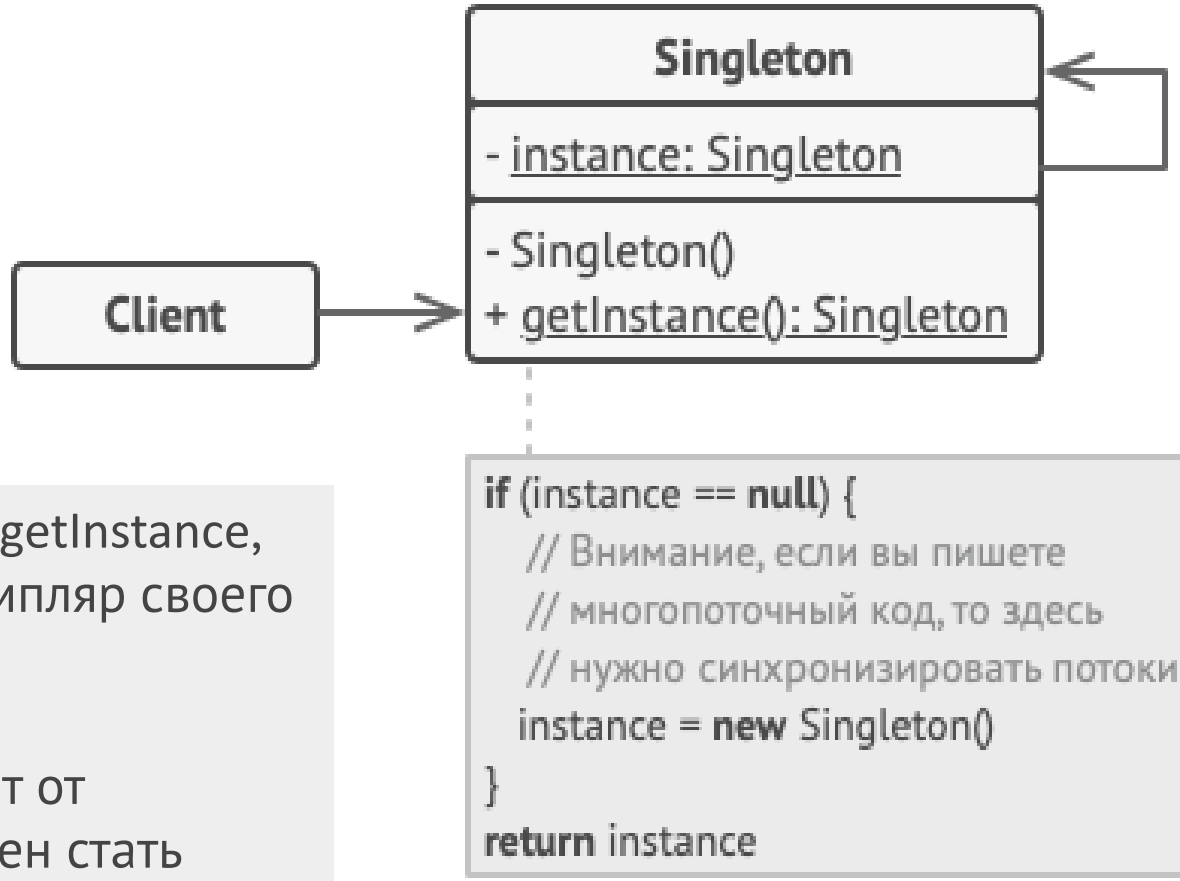
```
record Test(int A, int B, int C, int D, int E, int F, int G, int H);


public static void Demo_Record() {

    Test t1 = new Test(1, 2, 3, 4, 5, 6, 7, 8);
    Test t2 = t1 with { A = 10, C = 30 };
    Console.WriteLine(t2);
    // Test { A = 10, B = 2, C = 30, D = 4, E = 5, F = 6, G = 7, H = 8 }
}
```

Вычисляемые свойства и ленивая инициализация

```
record NotLazyPoint(double X, double Y) {
    public double DistanceFromOrigin => Math.Sqrt(X * X + Y * Y);
}
```

```
record LazyPoint(double X, double Y) {
    double? _distance;
    public double DistanceFromOrigin
    {
        get
        {
            if (_distance == null)
                _distance = Math.Sqrt(X * X + Y * Y);
            return _distance.Value;
        }
    }
}
```

Singleton (одиночка)



Singleton определяет статический метод getInstance, который возвращает единственный экземпляр своего класса.

Конструктор одиночки должен быть скрыт от клиентов. Вызов метода getInstance должен стать единственным способом получить объект этого класса.

https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff650849(v=pandp.10)

Классическая неправильная реализация

```
/// <summary>
/// потоконебезопасный одиночка
/// </summary>
public class SingletonNaive
{
    private static SingletonNaive instance;

    private SingletonNaive()
    { }

    public static SingletonNaive GetInstance()
    {
        if (instance == null)
            instance = new SingletonNaive();
        return instance;
    }
}
```

Плюсы

- Контролирует (и гарантирует) наличие единственного экземпляра класса
- Предоставляет к нему глобальную точку доступа
- Возможна реализация отложенной инициализации объекта-одиночки

Минусы

- Нарушает принцип единственной ответственности класса
- Проблемы с многопоточностью

Реализация потокобезопасная

```
/// <summary>
/// потокобезопасный одиночка (с блокировкой)
/// </summary>
public sealed class SingletonThreadSafe
{
    private static volatile SingletonThreadSafe instance;
    private static object syncRoot = new object();

    private SingletonThreadSafe() { }

    public static SingletonThreadSafe GetInstance()
    {
        if (instance == null) {
            lock (syncRoot) {
                if (instance == null)
                    instance = new SingletonThreadSafe();
            }
        }
        return instance;
    }
}
```

https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff650316(v=pandp.10)

Реализация с изъяном…

```
class Manager
{
    public static readonly Manager Instance = new Manager();
    public int data = 5;
}
class Program
{
    static void Main(string[] args)
    {
        System.Console.WriteLine(Manager.Instance.data);
       // Вопрос: Чем плоха эта реализация?
    }
}
```

Реализация через статический конструктор

```
/// <summary>
/// потокобезопасный одиночка (без блокировки)
/// </summary>
public sealed class SingletonSafe
{
    private static readonly SingletonSafe instance;

    private SingletonSafe()
    { }

    static SingletonSafe()
    {
        instance = new SingletonSafe();
    }

    public static SingletonSafe GetInstance()
    {
        return instance;
    }
}
```

Что такое кортежи?

Кортеж — упорядоченный набор фиксированной длины, состоящий из элементов одного или разных типов.

Примеры кортежей:
("первая строка", "вторая строка", "третья строка")
("просто строка", 1234)

Классификация:
1) Кортежи ссылочных типов Tuple<> (C# 4+).
2) Кортежи значимых типов ValueTuple<> (C# 7+).

Как мог бы класс выглядеть программно? (без обобщений)

```
public class TupleNonGeneric
{
    public object Item1 { get; private set; }

    public object Item2 { get; private set; }

    public TupleNonGeneric(object item1, object item2)
    {
        Item1 = item1;
        Item2 = item2;
    }
}
```

Как мог бы класс выглядеть программно? (с использованием обобщений)

```
public class TupleGeneric<T1, T2>
{
    public T1 Item1 { get; private set; }

    public T2 Item2 { get; private set; }

    public TupleGeneric(T1 item1, T2 item2)
    {
        Item1 = item1;
        Item2 = item2;
    }
}
```

Как могла бы структура выглядеть программно? (с использованием обобщений)

```
public struct MyValueTuple<T1, T2>
{
    public T1 Item1;

    public T2 Item2;

    public MyValueTuple(T1 item1, T2 item2)
    {
        Item1 = item1;
        Item2 = item2;
    }
}
```

Важно помнить

- **Tuple** – ссылочный тип, а **ValueTuple** - значимый.
- **Tuple** – неизменяемый, а **ValueTuple** – изменяемый тип.
- **Tuple** – использует свойства, а **ValueTuple** – поля.
- Создать объект **Tuple** / **ValueTuple** можно так:
  - использовать конструктор.
  - использовать метод Create.
- Для доступа к элементам в **Tuple** можно использовать свойства: Item1, Item2 и т.д. до Item8 (Item0 – отсутствует).
- Нельзя просмотреть последовательно элементы в кортеже через foreach или с помощью индексатора.

Виды коллекций

System.Collections System.Collections.Generic

Type	Replacement
`ArrayList`	`List<T>`
`CaseInsensitiveComparer`	`StringComparer.OrdinalIgnoreCase`
`CaseInsensitiveHashCodeProvider`	`StringComparer.OrdinalIgnoreCase`
`CollectionBase`	`Collection<T>`
`Comparer`	`Comparer<T>`
`DictionaryBase`	`Dictionary<TKey, TValue>` or `KeyedCollection<TKey, TItem>`
`DictionaryEntry`	`KeyValuePair<TKey, TValue>`
`Hashtable`	`Dictionary<TKey, TValue>`
`Queue`	`Queue<T>`
`ReadOnlyCollectionBase`	`ReadOnlyCollection<T>`
`SortedList`	`SortedList<TKey, TValue>`
`Stack`	`Stack<T>`

List<T>

- Является массивом, размер которого динамически увеличивается по мере добавления элементов.
- Универсальный эквивалент ArrayList (есть доступ по индексу)
- Свойство Capacity – возвращает или задает общее число элементов, которые может вместить внутренняя структура данных без изменения размера.
- Метод TrimExcess() - Задает емкость, равную фактическому числу элементов в списке, если это число меньше порогового значения (90% - т.е если неиспользуемая емкость < 10%, то размер не изменится).

Немного про асимптотику:

Добавление: O(1);

Удаление, поиск: O(n).

Добавление элементов и автоматическое расширение

```
0
i = 0 => new Capacity = 4
i = 4 => new Capacity = 8
i = 8 => new Capacity = 16
i = 16 => new Capacity = 32
i = 32 => new Capacity = 64
```

```
using System;
using System.Collections.Generic;
class Program {
    static void Main()     {
        List<int> list = new List<int>();
        Console.WriteLine(list.Capacity);
        for (int i = 0, cap = list.Capacity; i < 50; i++) {
            list.Add(i);
            if (cap != list.Capacity)
            {
                cap = list.Capacity;
                Console.WriteLine($"i = {i} => new Capacity =
{list.Capacity}");
            }
        }
    }
}
```

Пример реализации List<T>

```
public class MyList<T>
{
    T[] data = new T[2];
    int count = 0;

    public void Add(T value)
    {
        if (count >= data.Length)
            Array.Resize(ref data, data.Length * 2);
        data[count++] = value;
    }

    public int Count => count;

    public int Capacity => data.Length;

    public T this[int i] {
        get => data[i];
        set => data[i] = value;
    }
}
```