

В.В. Подбельский

Использованы материалы пособия Daniel Solis, Illustrated C#

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

05

Массивы и Кортежи Значений. Индексы и Диапазоны

Массивы – Основные Определения

Синтаксис: *<Тип>[] <Имя Ссылки> [= <инициализация>];*

Элементы (*Elements*) – отдельные единицы данных массива. Все элементы массива должны иметь один тип или должны быть унаследованы от одного типа.

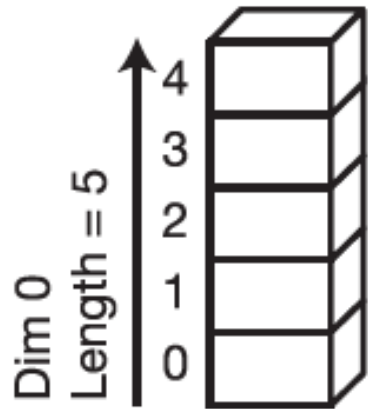
Размерность (*Rank*) – число измерений массива (amount of dimensions) – количество индексов для обращения к элементу массива (положительное число).

Длина измерения (*Dimension Length*) – число элементов в данном измерении массива. Каждое измерение массива имеет длину.

Размер массива (*Array Length*) – общее количество элементов, содержащееся во всех измерениях массива.

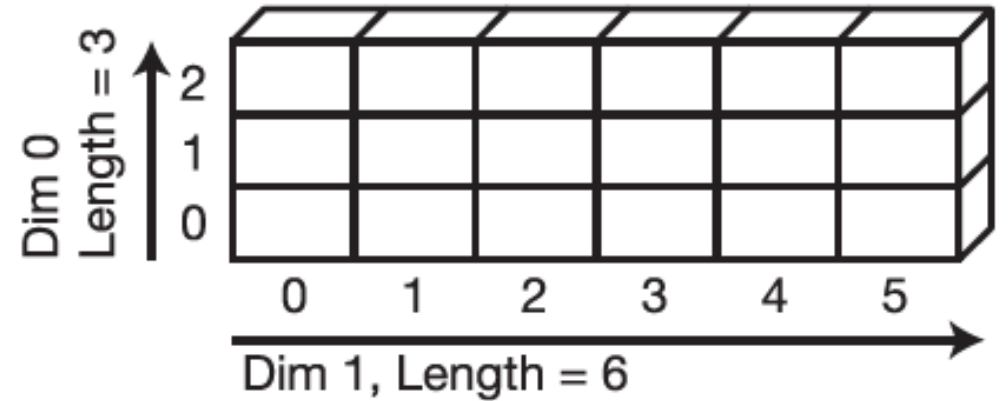
Запомните: имя массива – ошибочный термин, т. к. имена даются ссылкам на массивы, а не самим массивам.

Визуализация Массивов



Одномерный массив, `int[5]`:

- Размерность = 1
- Размер массива = 5



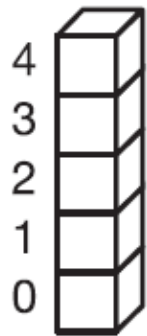
Двумерный прямоугольный массив, `int[3, 6]`:

- Размерность = 2
- Размер массива = 18

Классификация Типов Массивов в С#

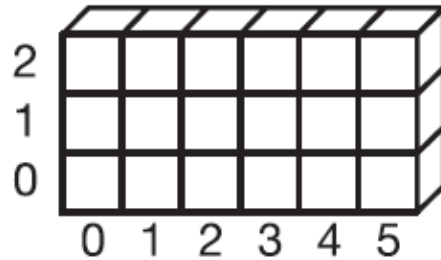
```
int x = myArray2[4, 6, 1]      // 1 пара квадратных скобок – многомерный массив.  
int y = jaggedArray1[2][7][4] // 3 пары квадратных скобок – зубчатый массив.
```

Одномерные
Массивы

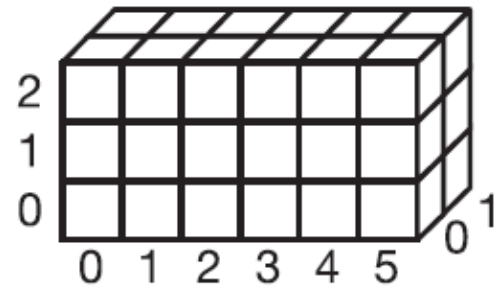


Одномерный:
`int[5]`

Прямоугольные Массивы

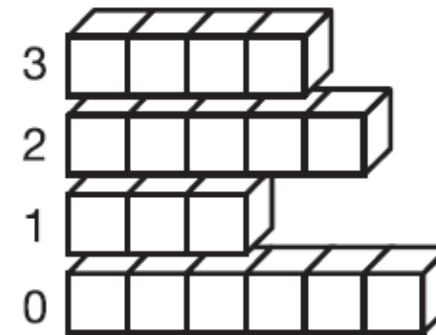


Двумерный:
`int[3,6]`



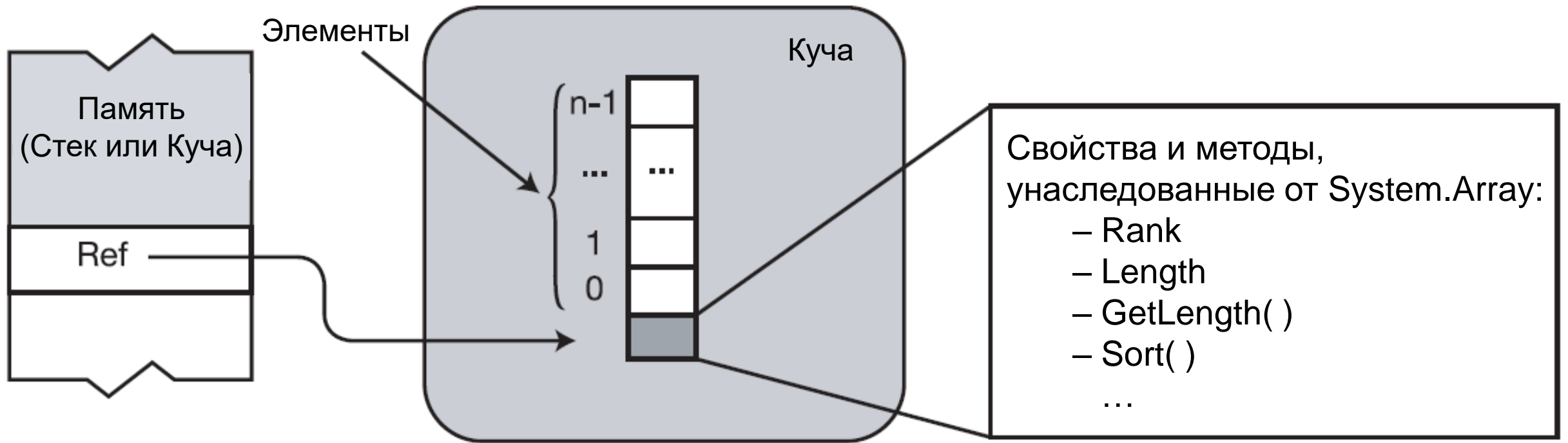
Трёхмерный:
`int[3,6,2]`

Зубчатые Массивы



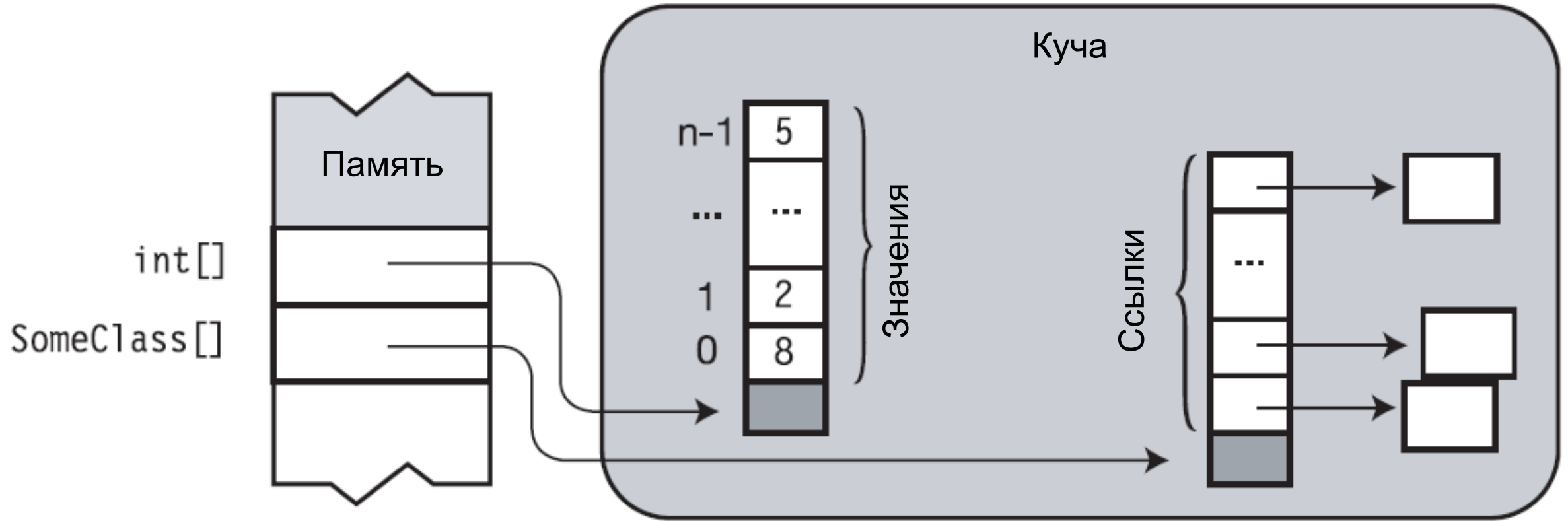
Зубчатый:
`int[4] []`

Массив как Объект



Запомните: все массивы являются ссылочными типами, поэтому хранятся всегда в куче.

Массивы Ссылок и Значений



Массивы могут хранить в себе как элементы ссылочных типов, так и элементы типов значений. Если массив хранит ссылочные типы, то каждым значением его элемента становится ссылка на объект соответствующего типа.

Одномерные Массивы. Создание

При объявлении массивов используется следующий синтаксис:

<Тип>[] <Идентификатор> = new[<Длина>] {<Значения...>;}

// Без инициализации.

```
int[] arr0;
```

// Неявная инициализация, используется значения типа int по умолчанию, т. е. 0.

```
int[] arr1 = new int[2];
```

// Явная инициализация без new.

```
int[] arr2 = { 1, 2, 3 };
```

// Явная инициализация, обязательно указать ровно 4 элемента в фигурных скобках.

```
int[] arr3 = new int[4] { 4, 5, 6, 7 };
```

// Явная инициализация, размер массива определяется автоматически.

```
int[] arr4 = new int[] { 0, 0, 0 };
```

// Явная инициализация, тип массива определяется автоматически компилятором.

```
var arr5 = new[] { 8, 9, 10 };
```

// Элементы массивов могут быть результатами вычислений выражений.

```
int[] arr6 = { (int)Math.Round(3.14), 20 + 5 };
```

Ошибки при Создании Одномерных Массивов

Такой код НЕ скомпилируется:

// Размер массива может указываться только при инициализации.

```
int[5] arrIncorrect1;
```

// Нужны квадратные скобки после int – иначе компилятор считает это
// попыткой создания значения типа int, а не int[].

```
int[] arrIncorrect2 = new int { 0, 1, 2 };
```

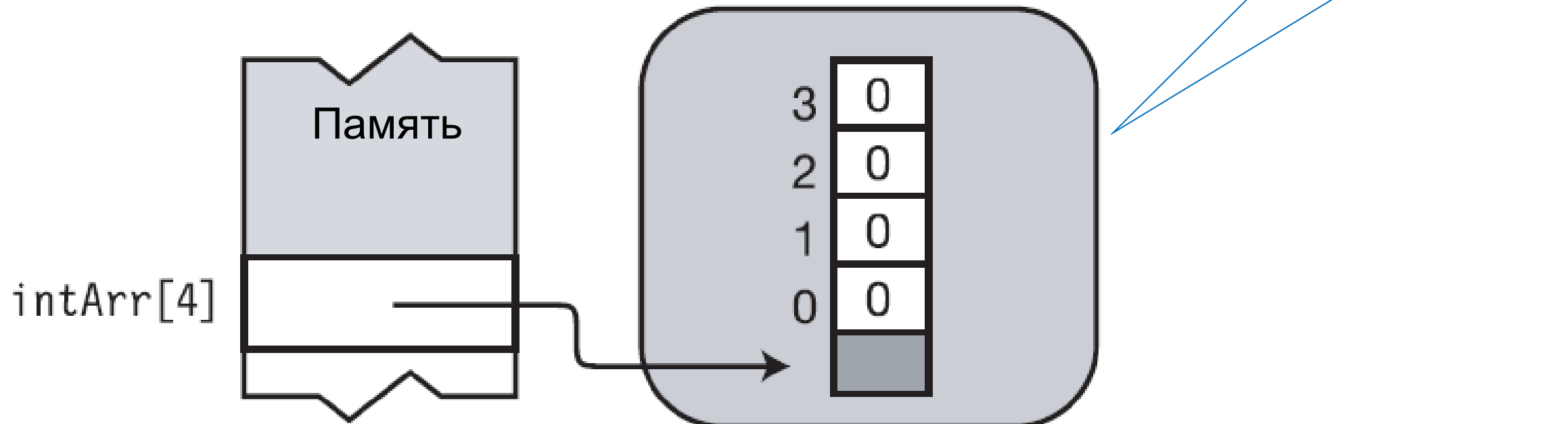
// «Массив var'ов» - синтаксически неверная конструкция.

```
var[] arrIncorrect3 = { 1, 2, 3 };
```


Визуализация Неявной Инициализации Массива

```
int[] intArr = new int[4];
```

```
int[] arr1 = new int[3] {10, 20, 30};  
int[] arr1 = {10, 20, 30};
```

 Эквивалентно

Одномерные Массивы. Индексация

Вы можете обращаться к элементам массива по индексу – соответствующему номеру элемента в массиве.

Важно: нумерация элементов массива в C# начинается с 0.

<Имя ссылки на массив>.Length – длина массива. Индекс последнего элемента непустого массива равен Length – 1.

При индексации массивов необходимо учитывать несколько вещей:

- Попытка обратиться по индексу, выходящему за границы массива приводит к **IndexOutOfRangeException**;
- Массивы – ссылочные типы, поэтому перед обращением по индексу возникает необходимость **проверки на null** (или использования **?[]**).

Одномерные Массивы: Пример

```
static bool TryFindMaxElement(int[] array, out int max_value) {  
    // Для пустых массивов и null максимума нет.  
    if (array == null || array.Length == 0) {  
        // Значение типа int по умолчанию, т. е. 0.  
        max_value = default;  
        return false;  
    }  
    max_value = array[0];  
    for (int i = 1; i < array.Length; i++) {  
        // Сравниваем элемент при обращении по индексу.  
        if (array[i] > max_value) {  
            max_value = array[i];  
        }  
    }  
    return true;  
}
```

Прямоугольные Массивы

Прямоугольные массивы – массивы, которые имеют более одного измерения.

Объявление двумерного массива с инициализацией будет иметь вид:
*<Тип>[,] <Идентификатор> = new[<Длина Измерения 1>,
<Длина Измерения 2>] { {<Значения...>}, ... };*

Размерность прямоугольного массива равна количеству измерений, а **длина** – количеству элементов массива *во всех измерениях*.

Обратите внимание: Вы можете использовать цикл **foreach** с прямоугольными массивами. При итерации будут последовательно перебираться все элементы всех измерений, начиная с нулевого.

Прямоугольные Массивы. Создание

// Двумерный массив без инициализации.

```
int[,] arr0;
```

// Двумерный массив с инициализацией.

```
int[,] arr2 = new int[2, 2] { { 2, 3 }, { 2, 3 } };
```

// Трёхмерный массив.

```
int[,,] arr1 = new int[2, 3, 5];
```

// Укороченный синтаксис инициализации.

```
int[,] arr3 = { { 0, 0 }, { 0, 1 }, { 1, 0 }, { 1, 1 } };
```

// Трёхмерный массив с явной инициализацией.

// Все элементы всех измерений заданы явно.

// arr4 состоит из четырёх групп, состоящих из трёх групп по два элемента.

```
int[,,] arr4 = new int[4, 3, 2] {  
    { {8, 6}, {5, 2}, {12, 9} },  
    { {6, 4}, {13, 9}, {18, 4} },  
    { {7, 2}, {1, 13}, {9, 3} },  
    { {4, 6}, {3, 2}, {23, 8} }  
};
```

Явная Инициализация Прямоугольного Массива

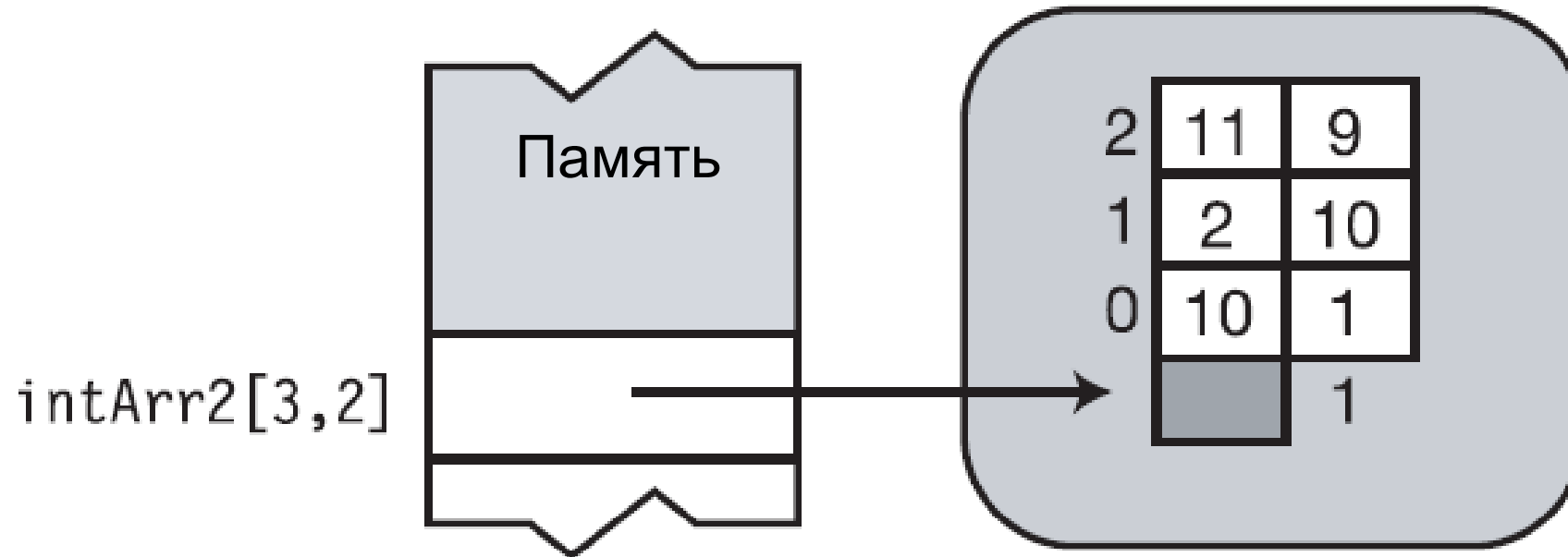
```
int[,] arr = new int[2,3] {{0, 1, 2}, {10, 11, 12}}; }  
int[,] arr = {{0, 1, 2}, {10, 11, 12}}; }
```

 Эквивалентно

Пример:

```
int[,] intArray2 = new int[,] { { 10, 1 }, { 2, 10 }, { 11, 9 } };
```

Список инициализации
массива, разделён запятыми.



Обращение к Элементам Массивов

Одномерные массивы

intArr1					
0	0	10	0	0	0

```
int[] intArr1 = new int[6]; // Объявление одномерного массива.  
intArr1[2] = 10;           // Запись элемента с индексом 2.  
int var1 = intArr1[2];     // Чтение элемента с индексом 2.
```

Многомерные массивы

intArr2				
0	0	0	0	0
0	0	0	0	0
0	0	0	7	0
0	0	0	0	0

```
int[,] intArr2 = new int[4, 5]; // Объявление двумерного м.  
intArr2[2, 3] = 7;              // Запись элемента [2, 3].  
int var2 = intArr2[2, 3];       // Чтение элемента [2, 3].
```

Прямоугольные Массивы: Пример

// Объявляем и инициализируем двумерный массив.

```
int[, ] arr = new int[4, 4] {  
    { 1, 2, 3, 4 },      {5, 6, 7, 8},  
    { 9, 10, 11, 12 },   { 13, 14, 15, 16 }  
};
```

1	2	3	4
5	6	7	8
9	10	11	12
13	13	15	16

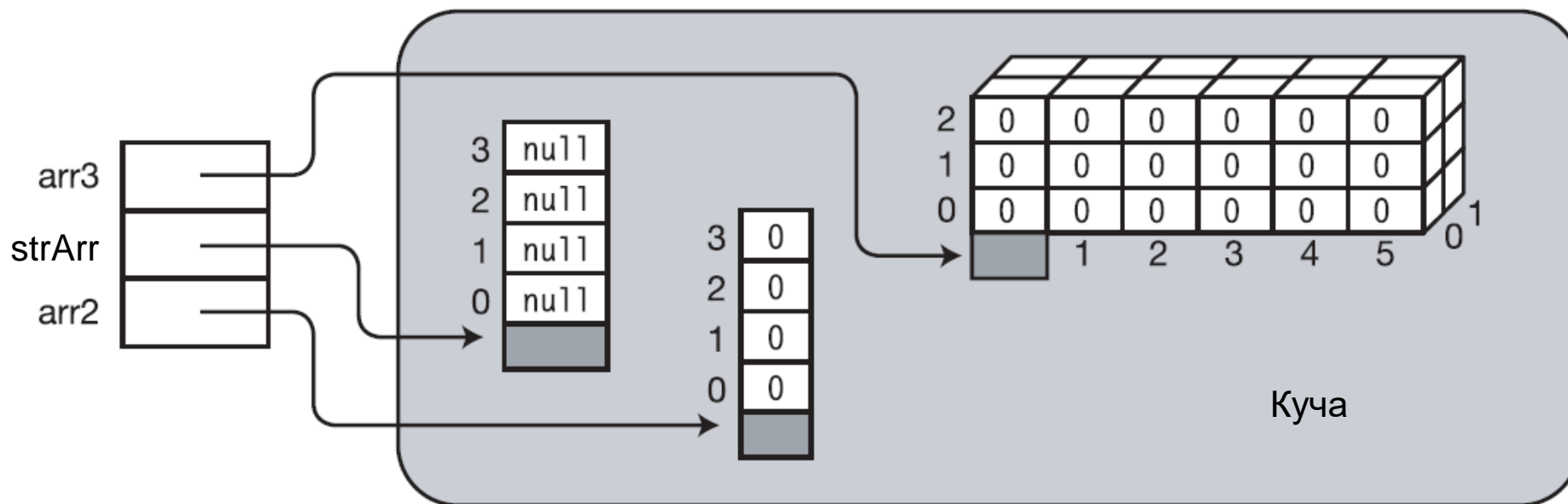
// Проходимся по каждому из элементов в цикле.

```
for (int i = 0; i < 4; ++i)  
{  
    for (int j = 0; j < 4; ++j)  
    {  
        // Выводим все элементы выше главной диагонали.  
        if (j > i)  
        {  
            Console.Write(arr[i, j] + " ");  
        }  
    }  
}
```

Результат выполнения:
2 3 4 7 8 12

Значения Элементов Массивов по Умолчанию

```
int[] arr2 = new int[4];           // Одномерный массив целых чисел из 4 элементов.  
string[] strArr = new string[4];  // Одномерный массив строк из 4 элементов.  
int[, ,] arr3 = new int[3, 6, 2]; // Трёхмерный массив с измерениями длиной 3, 6 и 2.
```



Запомните: элементы массива при неявной инициализации будут всегда иметь значения типа элемента по умолчанию.

Зубчатые Массивы (Jagged Arrays)

Зубчатые массивы – массивы, каждый элемент которых сам по себе является ссылкой на массив (вложенные массивы). Каждый вложенный массив может иметь различную длину.

Объявление зубчатого массива с инициализацией будет иметь вид:

```
<Тип>[][] <Идентификатор> = new [<Длина>][ ]  
{ new [<Длина Вложенного Массива 1>] {<Значения...>}, ...};
```

Обратите внимание: Вы можете комбинировать многомерные и зубчатые массивы.

```
// Массив arr состоит из 3 одномерных массивов.  
int[][] arr = new int[3][];  
// Инициализируем каждый вложенный одномерный массив отдельно:  
arr[0] = new int[] { 1, 2, 3 };  
arr[1] = new int[] { 4, 5, 6, 7 };  
arr[2] = new int[] { 8, 9, 10, 11, 12 };
```

Зубчатые Массивы. Создание

// Зубчатый массив одномерных зубчатых массивов без инициализации.

```
int[][][] arr0;
```

// Массив двумерных массивов с явной инициализацией.

```
int[][[],] arr2 =
```

```
{
```

```
    new int[,] { { 10, 20 }, { 100, 200 } },
```

```
    new int[2, 3] { { 30, 40, 50 }, { 300, 400, 500 } },
```

```
    new int[2, 4] { { 60, 70, 80, 90 }, { 600, 700, 800, 900 } }
```

```
};
```

Такой вариант **НЕ компилируется**:

```
int[][] arrIncorrect = new int[2][3];
```

Зубчатые Массивы в Памяти

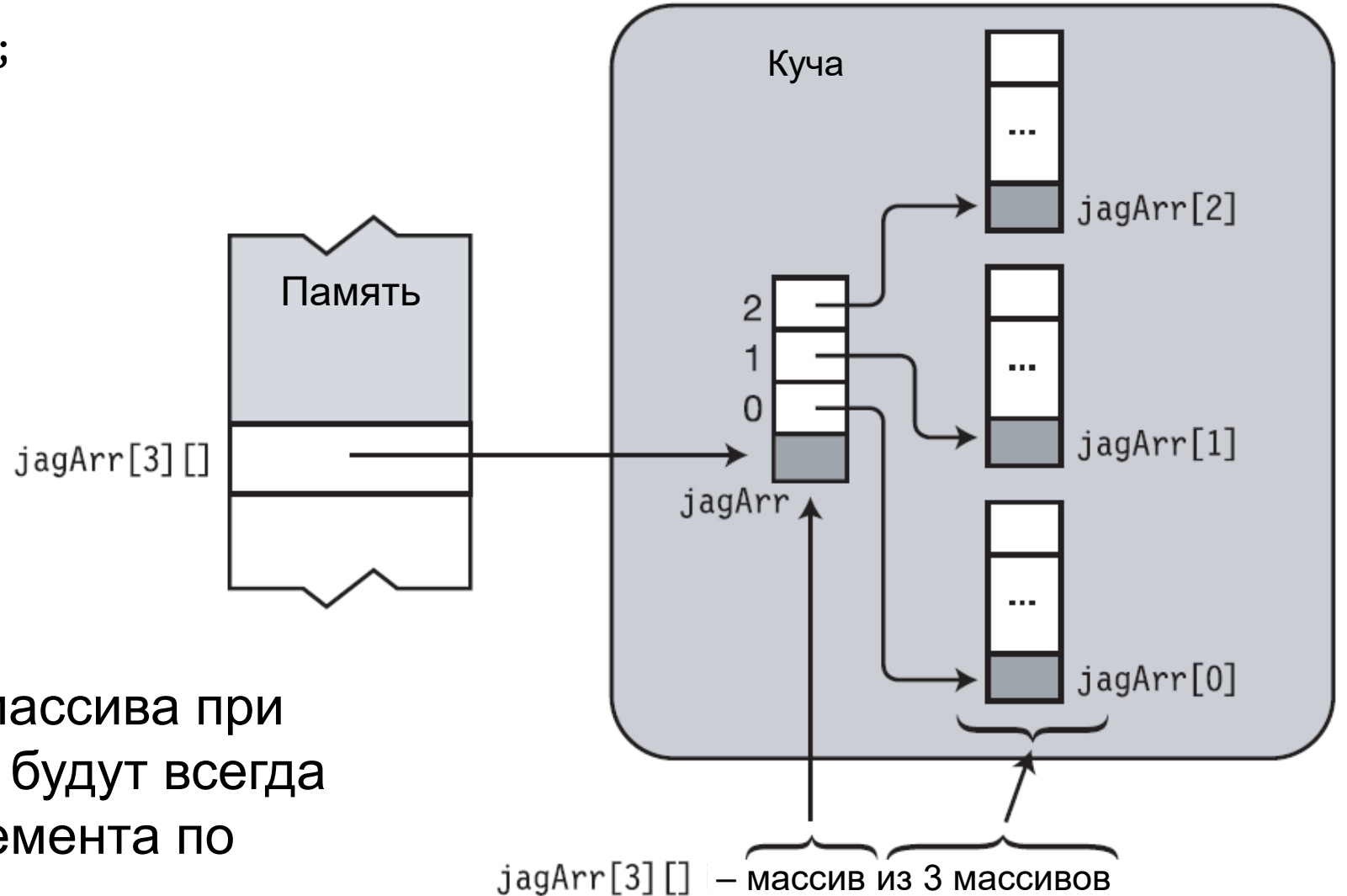
// Создадим зубчатый массив:

```
int[][] jagArr = new int[3][];
```

```
jagArr[0] = new int[3];
```

```
jagArr[1] = new int[5];
```

```
jagArr[2] = new int[15];
```



Запомните: элементы массива при неявной инициализации будут всегда иметь значения типа элемента по умолчанию.

Создание Зубчатого Массива в Памяти: Пример 1

// 1) Создать внешний массив:

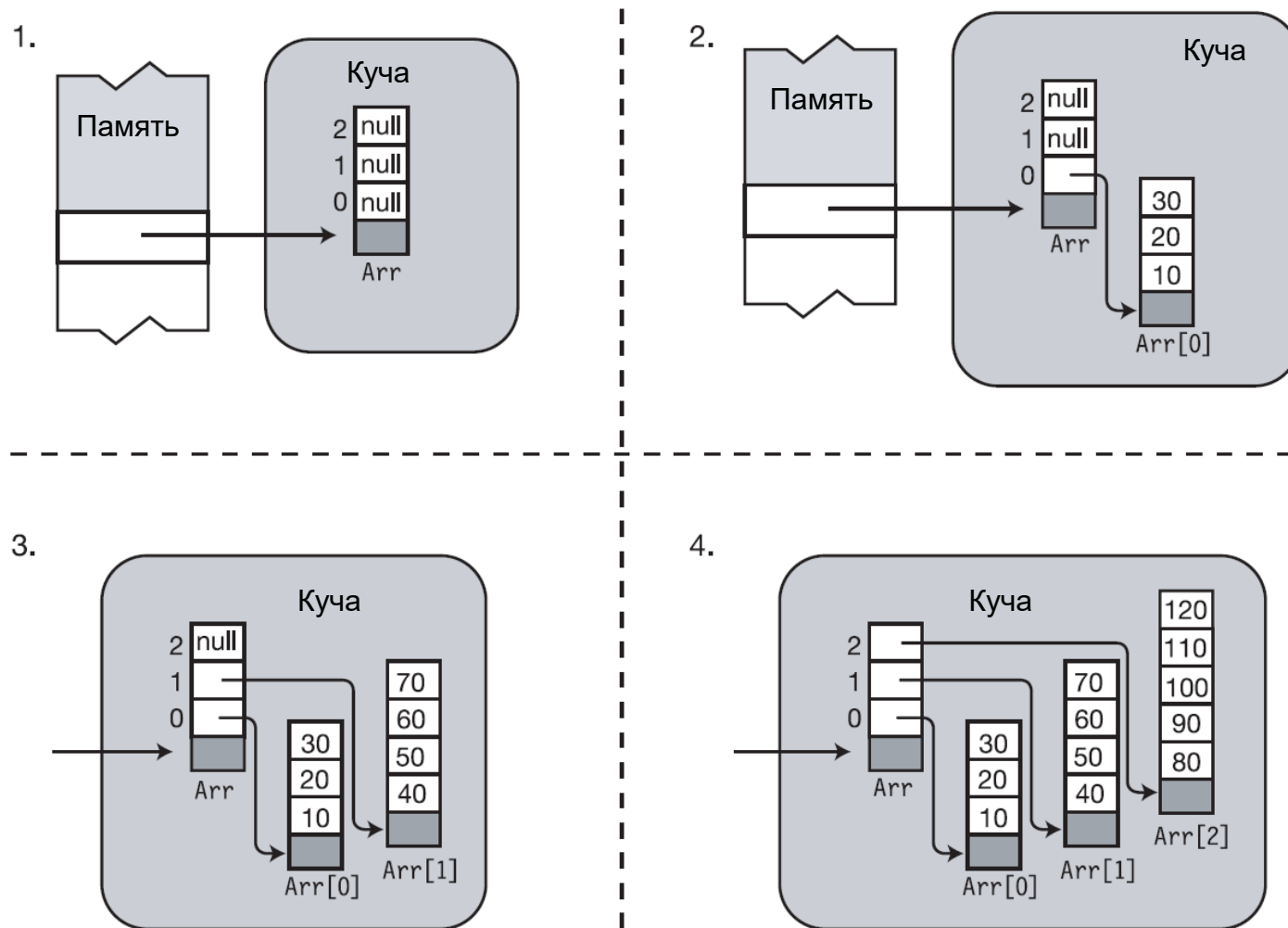
```
int[][] Arr = new int[3][];
```

// 2) Создать вложенные массивы:

```
Arr[0] = new int[]{ 10, 20, 30 };
```

```
Arr[1] = new int[]{ 40, 50, 60, 70 };
```

```
Arr[2] = new int[]{80, 90, 100, 110, 120}; 3.
```



Создание Зубчатого Массива в Памяти: Пример 2

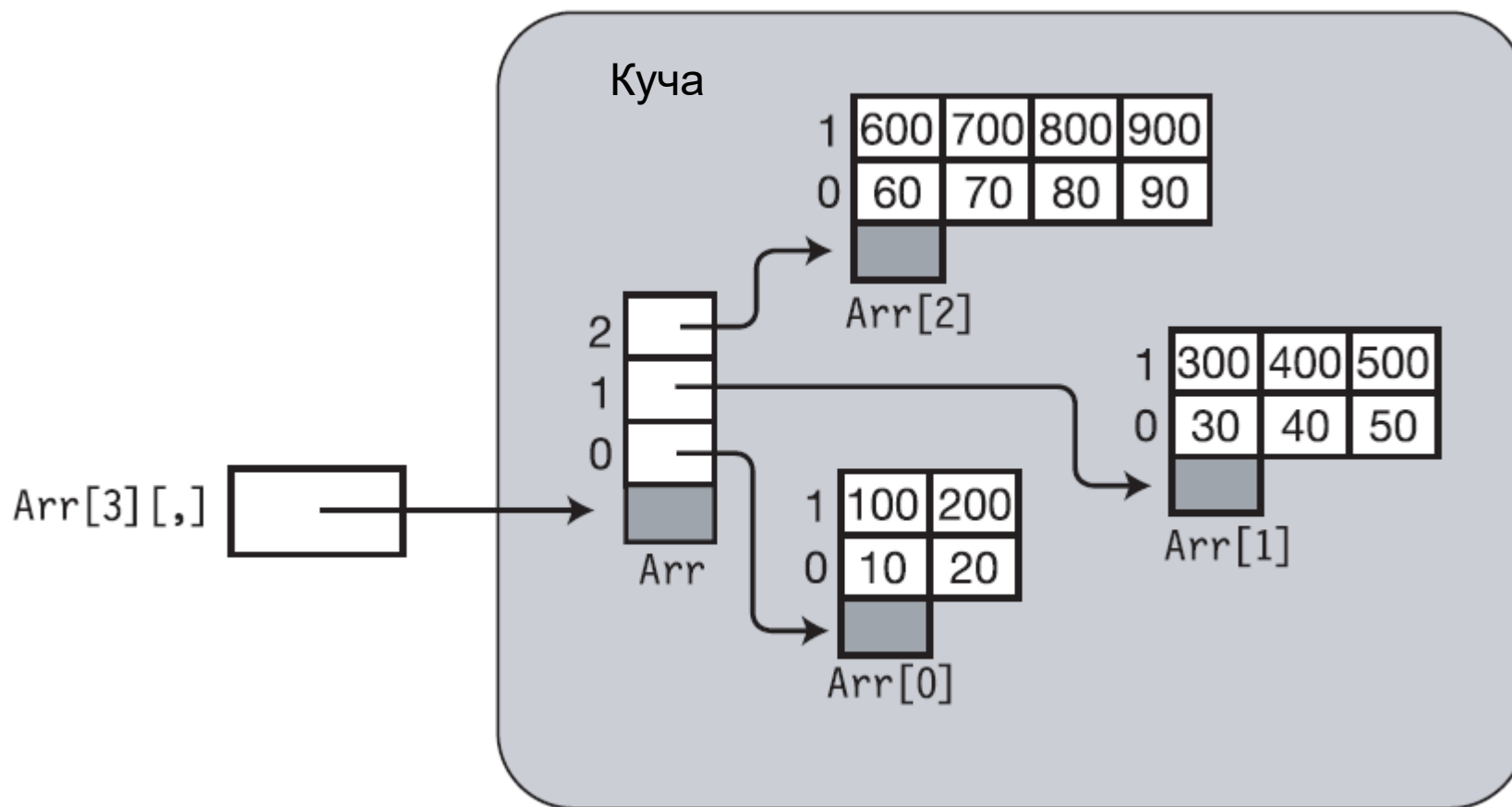
// В данном примере создаётся массив двумерных массивов.

```
int[][,] Arr = new int[3][,];
```

```
Arr[0] = new int[,] { { 10, 20 }, { 100, 200 } };
```

```
Arr[1] = new int[,] { { 30, 40, 50 }, { 300, 400, 500 } };
```

```
Arr[2] = new int[,] { { 60, 70, 80, 90 }, { 600, 700, 800, 900 } };
```



Обход Элементов Зубчатых Массивов – for

```
using System;
```

```
int[,] arr = {           // Массив двумерных массивов.  
    new[,] { { 10, 20 }, { 100, 200 } },  
    new int[,] { { 30, 40, 50 }, { 300, 400, 500 } },  
    new int[,] { { 60, 70, 80, 90 }, { 600, 700, 800, 900 } }  
};  
for (int i = 0; i < arr.GetLength(0); i++) {  
    for (int j = 0; j < arr[i].GetLength(0); j++) {  
        for (int k = 0; k < arr[i].GetLength(1); k++) {  
            Console.WriteLine($"[{i}][{j},{k}] = {arr[i][j, k]}\t");  
        }  
        Console.WriteLine();  
    }  
    Console.WriteLine();  
}
```

Метод GetLength()
позволяет получить
длину измерения
вложенных массивов.

Обход Элементов Зубчатых Массивов – for:

Результат

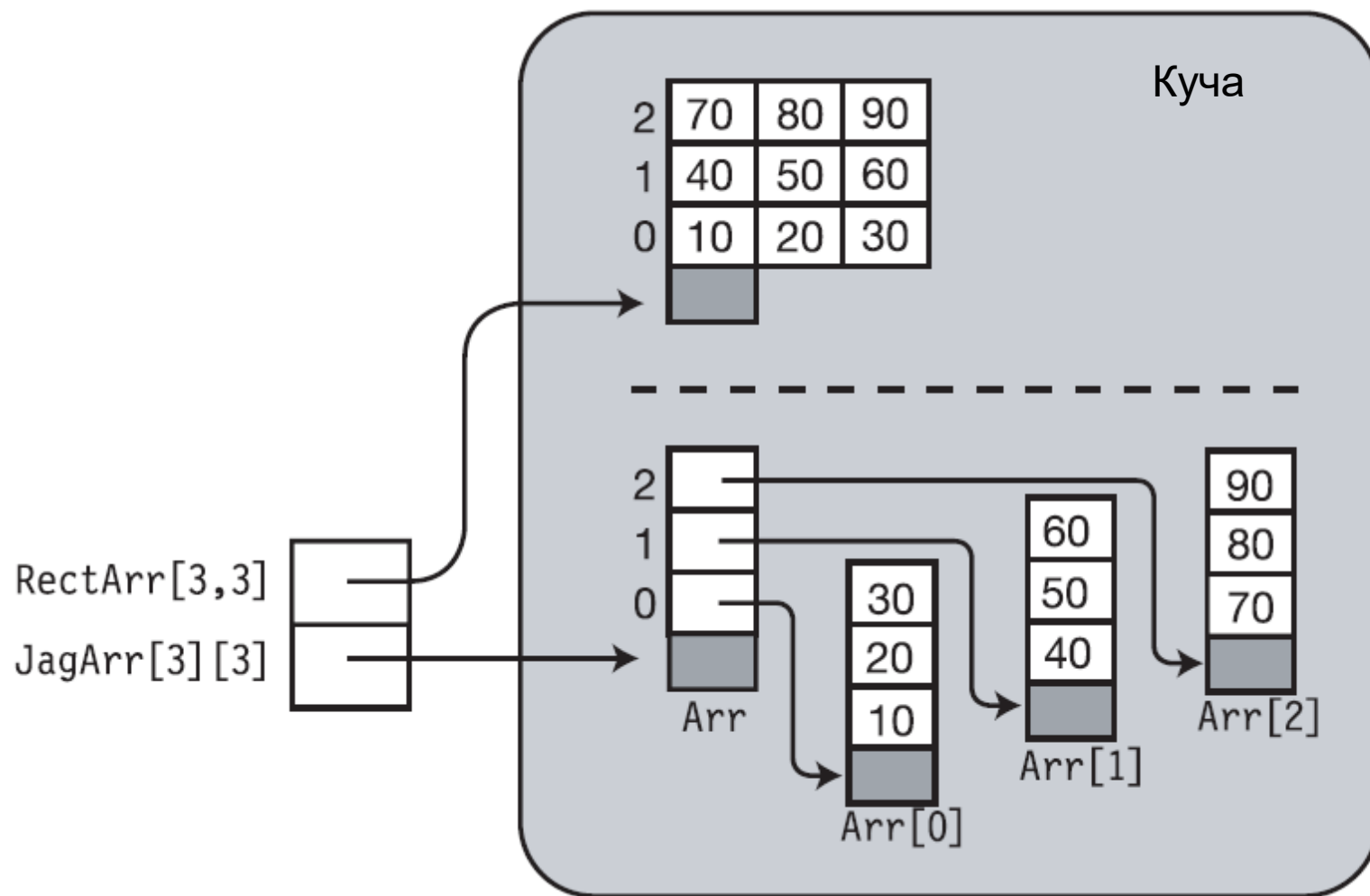
Результат выполнения:

$[0][0,0] = 10$	$[0][0,1] = 20$
$[0][1,0] = 100$	$[0][1,1] = 200$

$[1][0,0] = 30$	$[1][0,1] = 40$	$[1][0,2] = 50$
$[1][1,0] = 300$	$[1][1,1] = 400$	$[1][1,2] = 500$

$[2][0,0] = 60$	$[2][0,1] = 70$	$[2][0,2] = 80$	$[2][0,3] = 90$
$[2][1,0] = 600$	$[2][1,1] = 700$	$[2][1,2] = 800$	$[2][1,3] = 900$

Сравнение Прямоугольных и Зубчатых Массивов




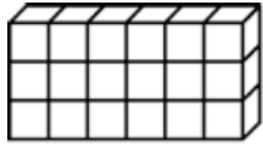
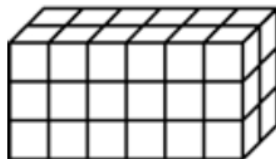
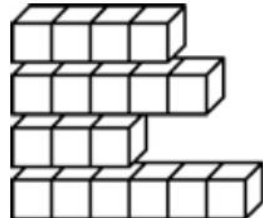
Прямоугольный массив размера 3 на 3:

- Один объект-массив;
- Тип ссылки – `System.Int32[,]`;
- Прямоугольные массивы не оптимизируются в IL.

Зубчатый массив из 3 элементов:

- Четыре объекта-массива;
- Тип ссылки – `System.Int32[][]`;
- Более сложная структура;
- Одномерные массивы оптимизируются в IL.

Сравнение Разных Видов Массивов

Тип Массива	Количество объектов-массивов	Синтаксис	Особенности	Форма
Одномерный	1	1 пара скобок	Оптимизируется в IL.	 <p>Одномерный: <code>int[3]</code></p>
Прямоугольный	1	1 пара скобок и запятые	Многомерный, все «вложенные массивы» должны быть одинаковой длины.	 <p>Двумерный: <code>int[3,6]</code></p>  <p>Трёхмерный: <code>int[3,6]</code></p>
Зубчатый	Несколько	Несколько пар скобок.	Многомерный, вложенные массивы могут быть разной длины	 <p>Зубчатый: <code>int[4][]</code></p>

Оператор Цикла foreach

Оператор цикла **foreach** позволяет итерироваться по всем элементам массива.

Для итерации с помощью foreach используется следующий синтаксис:

```
foreach (<Тип> <Идентификатор> in <Идентификатор Массива>) {...}
```

```
foreach (var <Идентификатор> in <Идентификатор Массива>) {...}
```

Важно: Итерационная переменная foreach доступна только для чтения.

```
using System;
```

```
int[,] arr = { { 1, 2, 3 }, { 4, 5, 6 } };
```

```
// Итерация осуществляется по всем измерениям!
```

```
foreach (int value in arr) {
```

```
    // ++value; // Если убрать комментарий – возникнет ошибка компиляции.
```

```
    Console.Write(value + " ");
```

```
}
```

Результат выполнения:

1 2 3 4 5 6

Оператор foreach: Пример (Зубчатые Массивы)

```
int sum = 0;
int[][] arr1 = {
    new int[] { 10, 11 },
    new int[] { 12, 13, 14 }
};
// Цикл foreach отдельно проходит по
// каждому из вложенных в arr1 массивов.
foreach (int[] array in arr1) {
    System.Console.WriteLine("Starting new array:");
    // Обход по каждому из элементов вложенных массивов.
    foreach (int item in array) {
        sum += item;
        System.Console.WriteLine($"Item: {item}, Current sum: {sum}");
    }
}
```

Результат выполнения:

Starting new array:
Item: 10, Current sum: 10
Item: 11, Current sum: 21
Starting new array:
Item: 12, Current sum: 33
Item: 13, Current sum: 46
Item: 14, Current sum: 60

Члены Класса Array-1

Член	Тип Члена	static?	Краткое описание
<u>Length</u>	свойство	нет	Количество элементов во всех измерениях массива.
<u>Rank</u>	свойство	нет	Количество измерений массива.
<u>GetLength</u>	метод	нет	Длина указанного измерения массива.
<u>Clear</u>	метод	да	Заменяет диапазон значений массива значениями типа элемента по умолчанию.
<u>Sort</u>	метод	да	Выполняет сортировку <i>одномерного</i> массива.
<u>BinarySearch</u>	метод	да	Выполняет поиск значения в одномерном массиве.
<u>Clone</u>	метод	нет	Выполняет <i>поверхностное</i> копирование массива – копирует только лежащие в массиве значения/ссылки.

Члены Класса Array-2 (Методы)

Член	static?	Краткое описание
Reverse	да	Разворачивает диапазон элементов массива.
Resize	да	Пересоздаёт массив большего/меньшего размера, содержащий элементы исходного.
IndexOf	да	Возвращает индекс первого вхождения элемента в массиве или -1 при его отсутствии.
LastIndexOf	да	Возвращает индекс последнего вхождения элемента в массиве или -1 при его отсутствии.
GetLowerBound	нет	Возвращает индекс первого элемента данного измерения массива.
GetUpperBound	нет	Возвращает индекс последнего элемента данного измерения массива.
CreateInstance	да	По указанному типу создаёт массив с заданной нижней границей и размерами.

Члены Класа Array: Пример

```
using System;
int[] arr = { 15, 20, 5, 25, 10 };
PrintArray(arr);
Array.Sort(arr);
PrintArray(arr);
Array.Reverse(arr);
PrintArray(arr);
Array.Resize(ref arr, 3);
PrintArray(arr);
Console.WriteLine($"Rank = {arr.Rank}, Length = {arr.Length}");
Console.WriteLine($"GetLength(0) = {arr.GetLength(0)}, GetType() = {arr.GetType()}");

static void PrintArray(int[] array) {
    foreach (var element in array) {
        Console.Write($"{element} ");
    }
    Console.WriteLine();
}
```

Результат выполнения:

15 20 5 25 10

5 10 15 20 25

25 20 15 10 5

25 20 15

Rank = 1, Length = 3

GetLength(0) = 3, GetType() = System.Int32[]

Метод Clone(): Пример 1 – Типы Значений

```
using System;
```

```
int[] intArr1 = { 1, 2, 3 };
```

```
// Требуется приведение - Clone возвращает ссылку типа Object.
```

```
int[] intArr2 = intArr1.Clone() as int[];
```

```
intArr2[0] = 100;
```

```
intArr2[1] = 200;
```

```
intArr2[2] = 300;
```

```
foreach (int item in intArr1) {
```

```
    Console.Write(item + " ");
```

```
}
```

```
Console.WriteLine();
```

```
foreach (int item in intArr2) {
```

```
    Console.Write(item + " ");
```

```
}
```

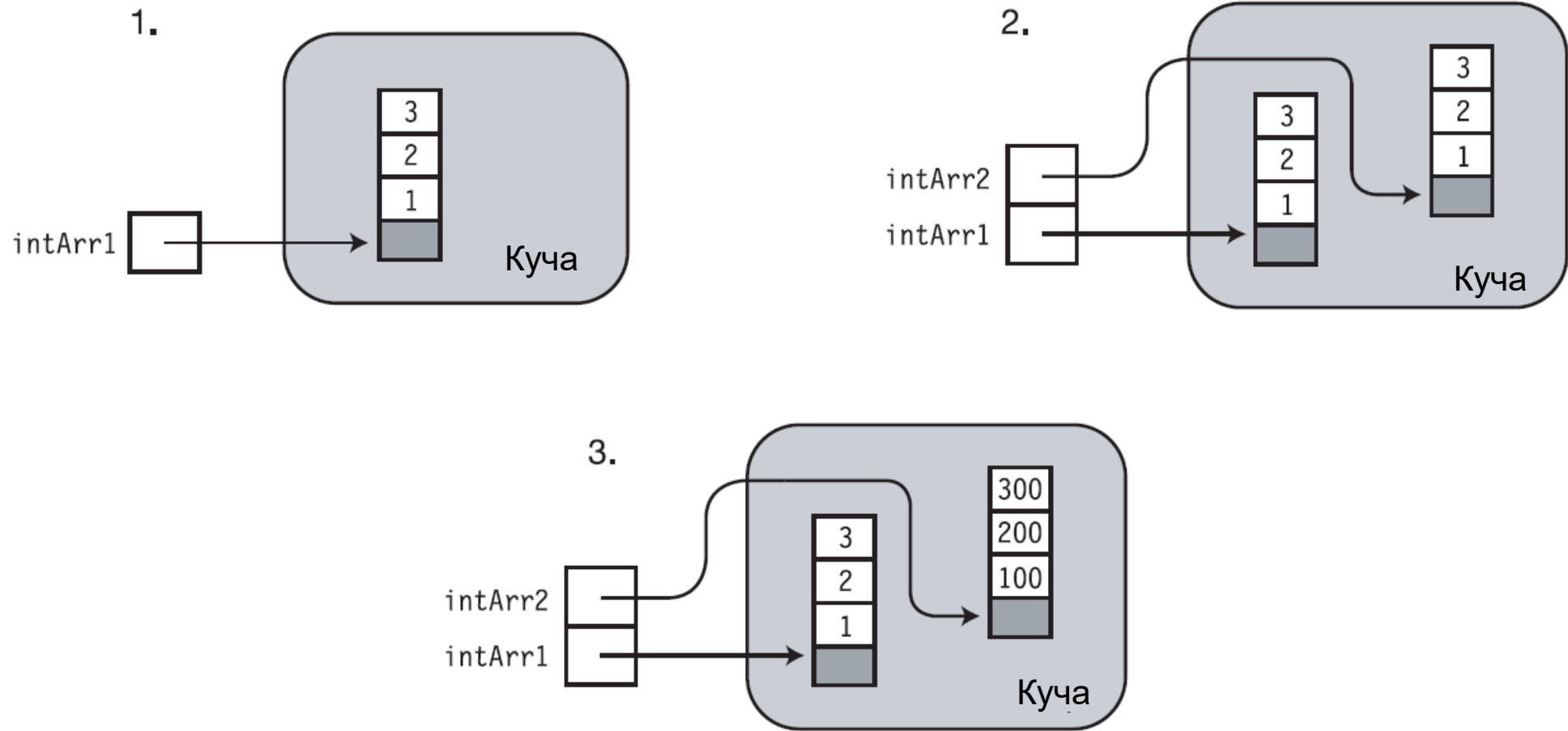
Операция as пробует выполнить приведение и возвращает результат в случае успеха, иначе - null.

Результат выполнения:

1 2 3

100 200 300

Иллюстрация к Примеру 1



Для одномерного массива `int[]` будут корректно скопированы значения – полученная копия никак не связана с оригиналом.

Метод Clone(): Пример 2 – Ссылочные Типы

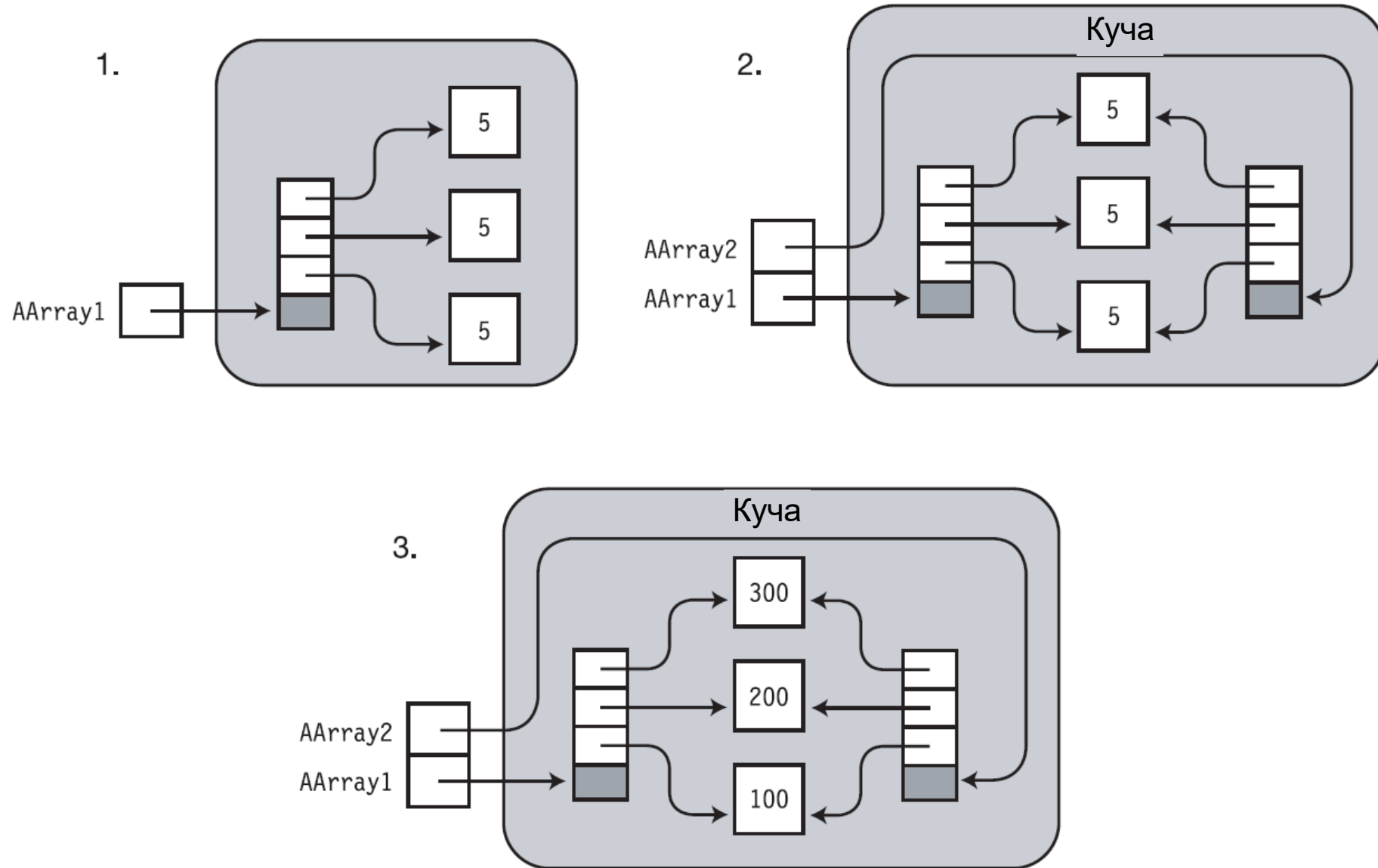
```
class A {  
    public int Value = 5;  
}  
  
A[] AArray1 = { new A(), new A(), new A() };  
A[] AArray2 = AArray1.Clone() as A[];  
AArray2[0].Value = 100;  
AArray2[1].Value = 200;  
AArray2[2].Value = 300;  
foreach (A item in AArray1) {  
    System.Console.Write(item.Value + " ");  
}  
System.Console.WriteLine();  
foreach (A item in AArray2) {  
    System.Console.Write(item.Value + " ");  
}
```

За счёт поверхностного копирования будут скопированы только ссылки на объекты-массивы, а сами вложенные массивы не будут скопированы (на них будут ссылаться дважды).

Результат выполнения:

```
100 200 300  
100 200 300
```

Иллюстрация к Примеру 2



Индексация с Конца. Index

В C# 8.0 была добавлена операция **^x** для получения **индекса с конца**.

Запомните: Для последовательности длины `length` индекс с конца **^x** будет вычисляться как **`length – x`**. Таким образом, при индексации с конца последний элемент имеет **индекс `^1`**, а первый – **`^length`**.

```
using System;

int[] arr = { 1, 2, 3, 4, 5 };
Console.WriteLine(arr[^1]);           // 5.
Console.WriteLine(arr[^5]);           // 1.
```

Операция **^x** возвращает элемент специального типа **System.Index**, который неявно преобразуется к типу `int`, благодаря чему `Index` применим везде, где ожидается `int`.

Индексы: Пример

```
using System;
```

```
int[] arr = { 35, 35, 555, 800, 8 };
```

```
// Обход массива в обратную сторону с помощью индексов с конца.
```

```
for (int i = 1; i <= arr.Length; i++)  
{  
    Console.Write(arr[^i] + " ");  
}
```

Следите за индексами при использовании индексации с конца.

Результат выполнения:

8 800 555 35 35

Диапазоны. Range

Ещё одним нововведением C# 8.0 стали **диапазоны** (System.Range). Вы можете использовать бинарную операцию **диапозона** «..**»** для удобной работы с поддиапазонами внутри последовательностей.

Оба операнда операции диапазона могут иметь тип *int* или *Index*.

```
Range rng1 = ..;    // Эквивалент диапазона [0;^0).  
Range rng2 = x..;   // Эквивалент диапазона [x;^0).  
Range rng2 = ..y;   // Эквивалент диапазона [0;y).
```

Диапазоны: Пример 1

```
using System;
```

```
int[] arr = { 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50 };
```

```
int[] newArr1 = arr[5..11];
```

```
// Изменения старого массива не влияют на
```

```
// новый - он хранит копии значений.
```

```
arr[^4] = 100000;
```

```
foreach (int val in newArr1)
```

```
{
```

```
    Console.Write(val + " ");
```

```
}
```

Результат выполнения:

25 30 35 40 45 50

Диапазоны: Пример 2

```
using System;

int[][] jaggedArr = {
    new int[]{ 1, 2, 3 }, new int[]{ 4, 5, 6 },
    new int[]{ 6, 5, 4 }, new int[]{ 3, 2, 1 }
};
// Диапазоны применимы и к зубчатым массивам:
int[][] newJagged = jaggedArr[1..^1];
jaggedArr[1][0] = 100;
for (int i = 0; i < newJagged.Length; i++)
{
    for (int j = 0; j < newJagged[i].Length; j++)
    {
        Console.Write(newJagged[i][j] + " ");
    }
    Console.WriteLine();
}
```

Результат выполнения:

100 5 6

6 5 4

Кортежи-Значения

В C# 7.0, были добавлены **кортежи-значения** (*System.ValueTuple*) как способ группировки нескольких элементов в одну простую структуру. Они являются **типами значений** и могут содержать сколь угодно много элементов.

C# предоставляет специальный синтаксис для создания кортежей значений:
(*<Tup 1>*, *<Tup 2>*, ... , *<Tup n>*) *<Идентификатор>* = (*<Значение 1>*,
<Значение 2>, ... , *<Tup n>*);

```
(int, double) valueTuple1 = (10, 15.55);  
Console.WriteLine(valueTuple1);  
// ValueTuple – тип значений, копирование поэлементное.  
var valueTuple2 = valueTuple1;  
valueTuple1.Item1++;  
Console.WriteLine(valueTuple2.Item1);
```

Результат выполнения:
(10, 15,55)
10

Именованние Полей Кортежей-Значений

Вы можете явно указывать имена полей кортежа-значения как при объявлении типа, так и при его инициализации (при использовании `var`).

Важно: имена полей кортежа-значения существуют только на этапе компиляции, не учитываются при сравнении на равенство и впоследствии заменяются именами по умолчанию (`Item1`, `Item2`, ...).

Начиная с C# 7.1, компилятор может выводить имена переменных кортежей-значений из имён соответствующих переменных при инициализации.

Кортежи -Значения: Именование

```
using System;
```

```
// Явное именование параметров при объявлении.
```

```
(double perimeter, double area) valueTuple1 = (11.1, 22.2);
```

```
// Автоматическое определение, с именованными полями.
```

```
var valueTuple2 = (x: 1, y: 2);
```

```
// Использование явно заданного имени и имени по умолчанию.
```

```
Console.WriteLine(valueTuple2.x + valueTuple2.Item2);
```

```
// Хотя имена различные, типы полностью соответствуют.
```

```
(double density, double weight) valueTuple3 = valueTuple1;
```

```
// C# 7.1: Определение имён полей по переданным переменным.
```

```
int mass = 30, energy = 9000;
```

```
var valueTuple4 = (mass, energy);
```

```
Console.WriteLine(valueTuple4.mass + valueTuple4.energy);
```

Результат:

3

9030

Кортежи-Значения в Методах: Пример

В типе возвращаемого значения можно указывать имена элементов.

```
static (int Min, int Max, bool HasMinMax) MinMaxElement(int[] array) {  
    if (array == null || array.Length == 0)  
        return (-1, -1, true);  
  
    var output = (Min: array[0], Max: array[0], IsEmpty: false);  
    for (int i = 1; i < array.Length; i++) {  
        if (array[i] < output.Min)  
            output.Min = array[i];  
        if (array[i] > output.Max)  
            output.Max = array[i];  
    }  
    return output;  
}
```

Явное именование элементов при инициализации.

Сравнение Кортежей-Значений

В С# 7.3 добавлено **сравнение кортежей-значений** с помощью операторов `==` и `!=`.

Сравнение производится *поэлементно*, без учёта имён параметров и возможно *только* для кортежей-значений с одинаковым количеством элементов.

Важно: Хотя сравнение производится по короткой схеме, в любом случае будут вычислены все выражения, результаты вычисления которых используются в качестве элементов.

Сравнение Кортежей-Значений: Пример

```
using System;
```

```
// Для сравнения кортежей-значений типы элементов не  
// обязаны совпадать - достаточно неявного приведения.
```

```
(int, short) valueTuple1 = (21, 12);
```

```
(long, byte) valueTuple2 = (21, 12);
```

```
// Выведет True.
```

```
Console.WriteLine(valueTuple1 == valueTuple2);
```

Результат:

True

False

```
// Все выражения будут вычислены для сравнения:
```

```
// Выведется False - по умолчанию Math.Round округляет
```

```
// до ближайшего чётного, кортежи равны.
```

```
Console.WriteLine((Math.Round(2.5), Math.Truncate(3.5))
```

```
    != (Math.Round(1.5), Math.Truncate(3.2)));
```

Деконструкция Кортежей-Значений

Для записи элементов кортежей-значений в переменных можно использовать **деконструкцию**.

Синтаксис деконструкции в схож с синтаксисом определения кортежей-значений:

```
(int, string, double) valueTuple = (101, "dalmatian", 46.68);
```

При деконструкции существует набор правил:

- Перед круглыми скобками нельзя указать общий тип для всех переменных, такой синтаксис корректен только для var;
- Каждый элемент кортежа-значения необходимо присвоить переменной, но можно использовать пустую переменную «_»;
- Смешение объявления переменных и присваивания значений уже существующим при деконструкции не допускается (до C# 10).

Деконструкция Кортежей-Значений: Пример

```
(int, string, double) valueTuple = (101, "dalmatian", 46.68);  
// Ниже показаны различные варианты деконструкции данного кортежа-значения:  
  
// I способ: типы всех переменных указываются явно.  
(int count1, string breed1, double mass1) = valueTuple;  
// II способ: тип определяется компилятором.  
var (count2, breed2, mass2) = valueTuple;  
  
// III способ: деконструкция в уже существующие переменные.  
int count3;  
string breed3;  
double mass3;  
(count3, breed3, mass3) = valueTuple;  
// IV способ: смешение явного определения типов и автоматического  
// Довольно громоздко, по этой причине не рекомендуется.  
(int count4, var breed4, var mass4) = valueTuple;  
// Деконструкция с использованием пустых переменных:  
var (_, breed, _) = valueTuple;
```


Ковариантность Элементов Массивов

Warning: данный слайд выходит за рамки темы лекции и в первую очередь предназначен для более продвинутой аудитории.

```
class A { }  
class B : A { }
```

```
// 2 Массива типа A[]:
```

```
A[] AArray1 = new A[2];
```

```
A[] AArray2 = new A[2];
```

```
// Присваивание объектов типа A по ссылке типа A.
```

```
AArray1[0] = new A();
```

```
AArray1[1] = new A();
```

```
// Ковариантность: присваивание объектов типа B по ссылке типа A.
```

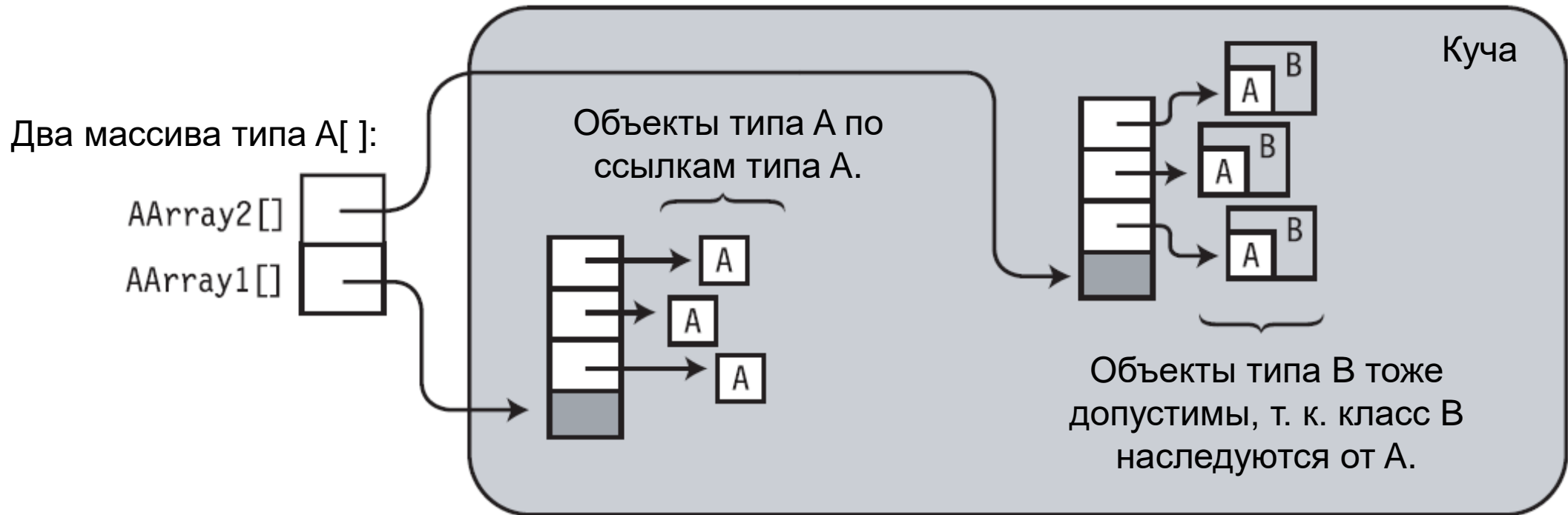
```
AArray2[0] = new B();
```

```
AArray2[1] = new B();
```

Все можете положить объект типа наследника по ссылке на объект типа родителя.

Схема Ковариантности Элементов Массивов

Warning: данный слайд выходит за рамки темы лекции и в первую очередь предназначен для более продвинутой аудитории.



Запомните: все ссылки в C# ковариантны. Если объект типа B – наследник типа A, то его можно сохранить по ссылке типа A. Иными словами, любой объект типа B также является и объектом типа A.

Ковариантность Массивов

Warning: данный слайд выходит за рамки темы лекции и в первую очередь предназначен для более продвинутой аудитории.

Ковариантность – сохранение иерархии наследования исходных типов в производных типах в том же порядке. Например, если у класса `Shape` есть наследник `Circle`, то ссылку типа-массива `Circle[]` также можно неявно привести к типу-массиву `Shape[]`.

```
class A { }  
class B : A { }      // B – наследник A.
```

```
A[] arrA = new A[5];  
for (int i = 0; i < arrA.Length - 1; i++) {  
    arrA[i] = new A();  
}  
arrA[arrA.Length - 1] = new B();           // OK - ссылки ковариантны.  
B[] b = new B[15];                         // OK - ковариантность массивов.  
arrA = b;  
System.Console.WriteLine(arrA.GetType()); // Выведет: B[].
```

Результат выполнения:
B[]

Проблемы Типобезопасности при Ковариантности

Warning: данный слайд выходит за рамки темы лекции и в первую очередь предназначен для более продвинутой аудитории.

Хотя `Circle[]` в прошлом примере можно неявно привести к `Shape[]`, данные типы **НЕ являются наследниками**.

Такое приведение **не типобезопасно**: по ссылке типа `A[]` можно попытаться записаться в массив с реальным типом `B[]` значение типа `A`, компилятор никак не сможет это проконтролировать – возникнет **`ArrayTypeMismatchException`**:

```
class A { }  
class B : A { }
```

```
// Выведется false – B[] не наследник A[]!  
System.Console.WriteLine(typeof(B[]).IsSubclassOf(typeof(A[])));  
B[] arrB = { new B(), new B(), null };  
A[] arrA = arrB; // OK: B - наследник A.  
// Никак не проверяется компилятором - ArrayTypeMismatchException.  
arrA[arrA.Length - 1] = new A();
```

Полезные Ссылки про Ковариантность

Warning: данный слайд выходит за рамки темы лекции и в первую очередь предназначен для более продвинутой аудитории.

Про ковариантность в программировании: [Википедия \(рус.\)](#)

Теория категорий (математическое обоснование): [Википедия \(рус.\)](#)

Про ковариантность в компьютерных науках: [Википедия \(англ.\)](#)

Про ковариантность в C#: [Официальная документация \(англ.\)](#)

Про ковариантность в C#: [Официальная документация \(рус.\)](#)