



ТЕМА 1.2 МАССИВЫ В ПРИЛОЖЕНИЯХ JAVA




- ✓ Array is a container object and it contains elements of similar data type.
- ✓ Array is a data structure where we store similar elements.
- ✓ We can store only fixed set of elements in a java array.
- ✓ Array is a collection of similar type of elements that have contiguous memory location.
- ✓ An array is a collection of similar data types. Array is a container object that hold values of homogenous type. It is also known as static data structure because size of an array must be specified at the time of its declaration.




Orange = element

Box = Container = Array



Apple = element

Box = Container = Array



Первый индекс

Элемент (с индексом 8)

0	1	2	3	4	5	6	7	8	9

Индексы

Длина массива = 10

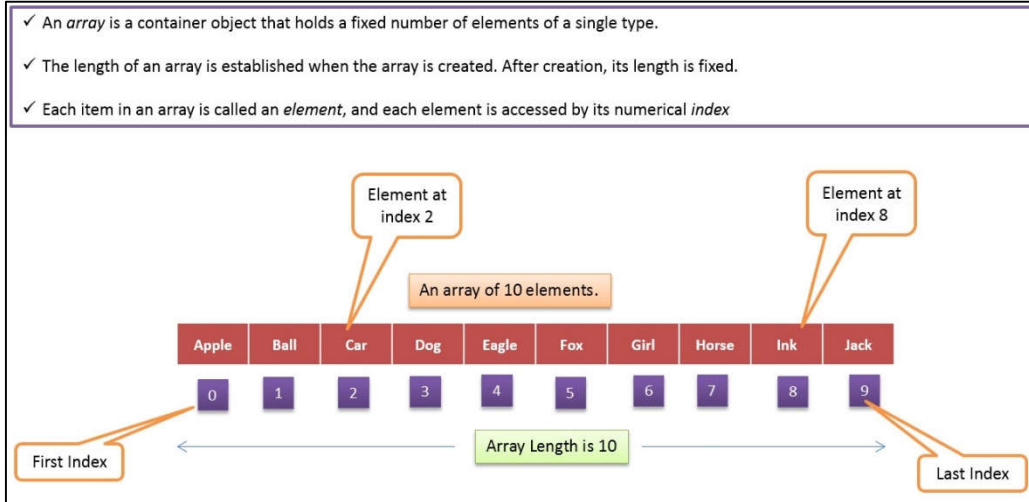
Массив (array) представляет собой совокупность однотипных переменных с общим именем. В *Java* массивы могут быть как одномерными, так и многомерными, хотя чаще всего используются одномерные массивы. Массивы могут применяться для самых разных целей, поскольку они предоставляют удобные средства для объединения связанных вместе переменных. Например, в массиве можно хранить максимальные суточные температуры, зарегистрированные в течение месяца, перечень биржевых курсов или же названия книг по программированию из домашней библиотеки.

Главное преимущество массива – возможность организации данных таким образом, чтобы ими было проще манипулировать. Так, если имеется массив данных о дивидендах, выплачиваемых по группе акций, то, организовав циклическое обращение к элементам этого массива, можно без особого труда рассчитать приносимый этими акциями средний доход. Кроме того, массивы позволяют организовать данные таким образом, чтобы облегчить их сортировку.

Массивами в *Java* можно пользоваться практически так же, как и в других языках программирования. Тем не менее у них имеется одна особенность: **массивы в *Java* реализованы в виде объектов**. Реализация массивов в виде объектов дает ряд существенных преимуществ, и далеко не самым последним среди них является возможность освобождения памяти, занимаемой массивами, которые больше не используются, средствами сборки мусора.

What is an array?

Dimensions	Example	Terminology																																												
1	<table><tr><td>0</td><td>1</td><td>2</td></tr></table>	0	1	2	Vector																																									
0	1	2																																												
2	<table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	6	7	8	Matrix																																			
0	1	2																																												
3	4	5																																												
6	7	8																																												
3	<table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	6	7	8	3D Array (3 rd order Tensor)																																			
0	1	2																																												
3	4	5																																												
6	7	8																																												
N	<table><tr><td><table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table></td><td>...</td><td><table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table></td></tr><tr><td></td><td><table><tr><td><table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table></td><td>...</td><td><table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table></td></tr></table></td><td>ND Array</td></tr></table>	<table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	6	7	8	...	<table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	6	7	8		<table><tr><td><table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table></td><td>...</td><td><table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table></td></tr></table>	<table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	6	7	8	...	<table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	6	7	8	ND Array
<table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	6	7	8	...	<table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	6	7	8																										
0	1	2																																												
3	4	5																																												
6	7	8																																												
0	1	2																																												
3	4	5																																												
6	7	8																																												
	<table><tr><td><table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table></td><td>...</td><td><table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table></td></tr></table>	<table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	6	7	8	...	<table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	6	7	8	ND Array																							
<table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	6	7	8	...	<table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	6	7	8																										
0	1	2																																												
3	4	5																																												
6	7	8																																												
0	1	2																																												
3	4	5																																												
6	7	8																																												



ОДНОМЕРНЫЕ МАССИВЫ

Одномерный массив представляет собой список связанных переменных. Например, в одномерном массиве можно хранить номера учетных записей активных пользователей сети или текущие средние результаты достижений бейсбольной команды.

Для объявления одномерного массива обычно применяется общий синтаксис следующего вида:

```
тип[] имя_массива = new тип[размер];
```

где **тип** объявляет конкретный тип массива. Тип массива, называемый также **базовым типом**, одновременно определяет тип данных каждого элемента, составляющего массив, а **размер** определяет число элементов массива.

Java Array Declaration

Declares an array of integers

As with variables of other types, the declaration does not actually create an array; it simply tells the compiler that this variable will hold an array of the specified type.

int[] anArray;

Array's type

Array's name

An array's type is written as `type[]`, where `type` is the data type of the contained elements; the brackets are special symbols indicating that this variable holds an array.

Creating, Initializing, and Accessing an Array

One way to create an array is with the `new` operator.

Create an array of integers

int[] intArray = new int[5];

Allocates an array with enough memory for 5 integer elements and assigns the array to the `intArray` variable.

Assign values to each element of the array:

```
intArray[0] = 10; // initialize first element
intArray[1] = 20; // initialize second element
intArray[2] = 30; // and so forth
```

Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + intArray[0]);
System.out.println("Element 2 at index 1: " + intArray[1]);
System.out.println("Element 3 at index 2: " + intArray[2]);
```



В связи с тем, что массивы реализованы в виде объектов, создание массива происходит в два этапа. Сначала объявляется переменная, ссылающаяся на массив, а затем выделяется память для массива, и ссылка на нее присваивается переменной массива. Следовательно, память для массивов в *Java* выделяется динамически с помощью оператора *new*.

Проиллюстрируем все сказанное выше на конкретном примере. В следующей строке кода создается массив типа *int*, состоящий из 10 элементов, а ссылка на него присваивается переменной *sample*:

```
int[] sample = new int[10];
```

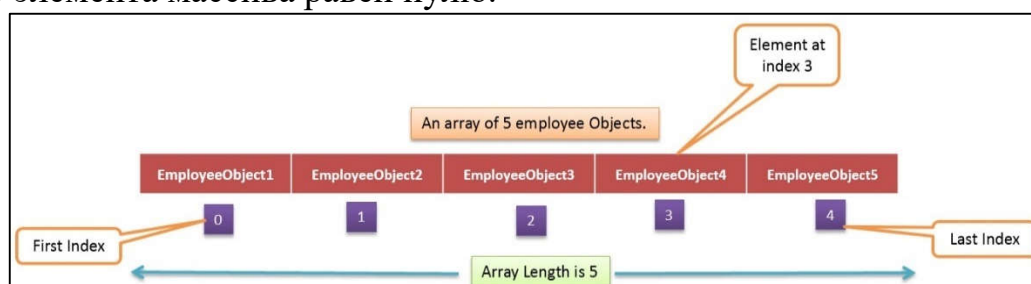
Объявление массива работает точно так же, как и объявление объекта. В переменной *sample* сохраняется ссылка на область памяти, выделяемую для массива оператором *new*. Этой памяти должно быть достаточно для размещения 10 элементов типа *int*.

Как и объявление объектов, приведенное выше объявление массива можно разделить на две отдельные составляющие.

```
int[] sample;  
sample = new int[10];
```

В данном случае сначала создается переменная *sample*, которая пока что не ссылается на конкретный объект. А затем переменная *sample* получает ссылку на конкретный массив.

Доступ к отдельным элементам массива осуществляется с помощью индексов. **Индекс** обозначает позицию элемента в массиве. В *Java* индекс первого элемента массива равен нулю.



Так, если массив *sample* содержит 10 элементов, то их индексы находятся в пределах от 0 до 9. Индексирование массива осуществляется по номерам его элементов, заключенным в квадратные скобки. Например, для доступа к первому элементу массива *sample* следует указать *sample[0]*, а для доступа к последнему элементу этого массива – *sample[9]*. В приведенном ниже примере программы в массиве *sample* сохраняются числа от 0 до 9.

```
//Пример №1. Демонстрация использования одномерного массива
class ArrayDemo {
    public static void main(String args[]) {
        int sample[] = new int[10];
        int i;
        for (i = 0; i < 10; i = i + 1) sample[i] = i;
        for (i = 0; i < 10; i = i + 1) //индексация массивов начинается
с нуля
            System.out.println("Элемент[" + i + "]: " + sample[i]);
    }
}
```

Результат выполнения программы:

Элемент[0]: 0
Элемент[1]: 1
Элемент[2]: 2
Элемент[3]: 3
Элемент[4]: 4
Элемент[5]: 5
Элемент[6]: 6
Элемент[7]: 7
Элемент[8]: 8
Элемент[9]: 9

Структура массива *sample* наглядно показана на рисунке ниже.

0	1	2	3	4	5	6	7	8	9
Sample [0]	Sample [1]	Sample [2]	Sample [3]	Sample [4]	Sample [5]	Sample [6]	Sample [7]	Sample [8]	Sample [9]

Массивы часто используются в программировании, поскольку они позволяют обрабатывать в цикле большое количество переменных. Например, в результате выполнения следующего примера программы определяется минимальное и максимальное значения из всех, хранящихся в массиве *nums*. Элементы этого массива перебираются в цикле *for*.

```
//Пример №2. Нахождение минимального и максимального элемента массива
class MinMax {
    public static void main(String args[]) {
        int nums[] = new int[10];
        int min, max;
        nums[0] = 99;
        nums[1] = -10;
        nums[2] = 100123;
        nums[3] = 18;
        nums[4] = -978;
        nums[5] = 5623;
```

```

        nums[6] = 463;
        nums[7] = 9;
        nums[8] = 287;
        nums[9] = 49;
        min = max = nums[0];
        for (int i = 1; i < 10; i++) {
            if (nums[i] < min) {min = nums[i];}
            if (nums[i] > max) {max = nums[i];}
        }
        System.out.println("min и max: " + min + " " + max);
    }}

```

Результат выполнения программы:

```

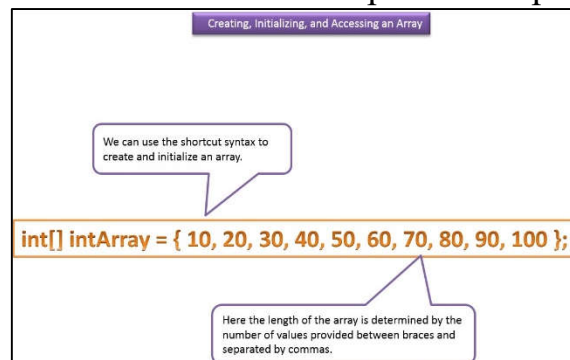
run:
min и max: -978 100123

```

В этом примере массив *nums* заполняется вручную в десяти операторах присваивания. И хотя в этом нет ничего неверного, существует более простой способ решения этой задачи. Массивы можно инициализировать в процессе их создания. Для этой цели служит приведенная ниже общая форма инициализации массива:

```
тип имя_массива[] = {val1, val2, val3, ..., valN};
```

где *val1-valN* обозначают начальные значения, которые поочередно присваиваются элементам массива слева направо в направлении роста индексов.



При инициализации массивов в процессе их создания *Java* автоматически выделяет объем памяти, достаточный для хранения всех инициализаторов массивов.

При этом необходимость в использовании оператора *new* явным образом отпадает сама собой. В качестве примера ниже приведена улучшенная версия программы, в которой определяются максимальное и минимальное значения в массиве.

```

//Пример №3. Применение инициализаторов массива. САМОСТОЯТЕЛЬНО
class MinMax {
    public static void main(String args[]) {
        int nums[]={99, -10, 100123, 18, -978, 5623, 463, -9, 287, 49};
        int min, max;
        min = max = nums[0];
        for (int i = 1; i < 10; i++) {
            if (nums[i] < min) {min = nums[i];}
            if (nums[i] > max) {max = nums[i];}
        }
    }
}

```

```

    }
    System.out.println("min и max: " + min + " " + max);
}
}

```

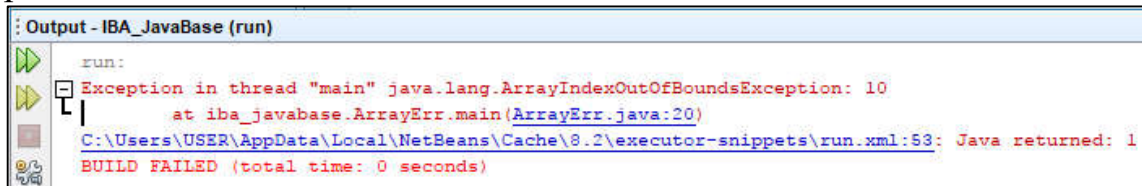
Границы массива в *Java* строго соблюдаются. В случае выхода индекса за верхнюю или нижнюю границу массива при выполнении программы возникает ошибка. Для того чтобы убедиться в этом, попробуйте выполнить приведенную ниже программу, в которой намеренно превышаются границы массива.

```

//Пример №4. Демонстрация превышения границ массива
class ArrayErr {
    public static void main(String args[]) {
        int sample[] = new int[10];
        int i;
        // Искусственно создать выход индекса за границы массива
        for (i = 0; i < 100; i = i + 1) {sample[i] = i;}
    }
}

```

Как только значение переменной *i* достигнет 10, будет сгенерировано исключение *ArrayIndexOutOfBoundsException*, и выполнение программы прекратится.



СОРТИРОВКА МАССИВА

Как пояснялось выше, данные в одномерном массиве организованы в виде индексированного линейного списка. Такая структура как нельзя лучше подходит для сортировки. В следующем примере предстоит реализовать простой алгоритм сортировки массива. Существуют разные алгоритмы сортировки, в том числе быстрая сортировка, сортировка перемешиванием, сортировка методом Шелла и т.п. Но самым простым и общеизвестным алгоритмом является пузырьковая сортировка. Этот алгоритм не очень эффективен, но отлично подходит для сортировки небольших массивов. Поэтапное описание процесса создания программы приведено ниже.

Создайте новый файл *Bubble.java*.

В алгоритме пузырьковой сортировки соседние элементы массива сравниваются между собой и, если требуется, меняются местами. При этом меньшие значения сдвигаются к одному краю массива, а большие – к другому. Этот процесс напоминает всплывание пузырьков воздуха на разные уровни в емкости с жидкостью, откуда и произошло название данного алгоритма.

исходный массив		обмен 2 и 3		обмен 2 и 7		обмен 2 и 5		нет обмена
1		1		1		1		1
5		5		5	↻	2		2
7		7	↻	2	↻	5		5
3	↻	2		7		7		7
2		3		3		3		3
первый проход циклом по массиву								

Пузырьковая сортировка предполагает обработку массива в несколько проходов. Элементы, взаимное расположение которых отличается от требуемого, меняются местами. Число проходов, необходимых для упорядочения элементов этим способом, на единицу меньше количества элементов в массиве.

Ниже приведен исходный код, составляющий основу алгоритма пузырьковой сортировки. Сортируемый массив называется *nums*.

```
// Реализация алгоритма пузырьковой сортировки по возрастанию
for (a = 1; a < size; a++) {
    for (b = size - 1; b >= a; b--) {
        if (nums[b - 1] > nums[b]){// если требуемый порядок
            t = nums[b - 1];// следования элементов не соблюдается,
            nums[b - 1] = nums[b];// то поменять элементы местам
            nums[b] = t;
        }
    }
}
```

Как видите, в приведенном выше фрагменте кода используются два цикла *for*. Во внутреннем цикле сравниваются соседние элементы массива и выявляются элементы, находящиеся не на своих местах. При обнаружении элемента, нарушающего требуемый порядок, два соседних элемента меняются местами. На каждом проходе наименьший элемент перемещается на одну позицию в нужное положение. Внешний цикл обеспечивает повторение описанного процесса до завершения всего процесса сортировки.

Ниже приведен полный исходный код программы из файла *Bubble.java*.

```
//Пример №5. Демонстрация алгоритма пузырьковой сортировки
class Bubble {
    public static void main(String args[]) {
        int nums[]={99, -10, 100123, 18, -978, 5623, 463, -9, 287, 49};
        int a, b, t;
        int size = 10; // количество сортируемых элементов
        // Отобразить исходный массив
        System.out.print("Исходный массив:");
        for (int i = 0; i < size; i++) {
            System.out.print(" " + nums[i]);
        }
    }
}
```

```

        System.out.println();
// Реализация алгоритма пузырьковой сортировки
        for (a = 1; a < size; a++) {
            for (b = size - 1; b >= a; b--) {
                if (nums[b - 1] > nums[b]){// если требуемый порядок
                    t = nums[b - 1];// следования не соблюдается,
                    nums[b - 1] = nums[b];// поменять элементы местам
                    nums[b] = t;// Отобразить отсортированный массив
                }
            }
        }
        System.out.print("Отсортированный массив:");
        for (int i = 0; i < size; i++) {
            System.out.print(" " + nums[i]);
        }
        System.out.println();
    }
}

```

Результат выполнения программы:

```

Output - IBA_JavaBase (run)

run:
Исходный массив: 99 -10 100123 18 -978 5623 463 -9 287 49
Отсортированный массив: -978 -10 -9 18 49 99 287 463 5623 100123

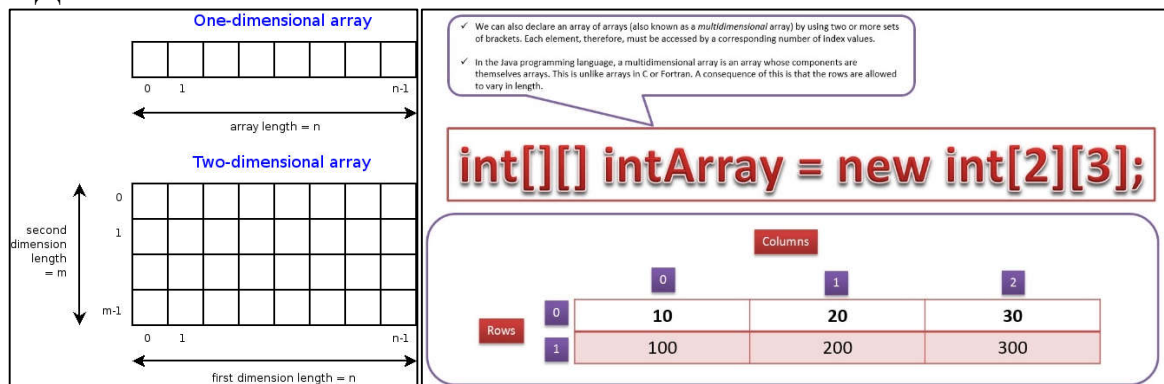
```

Как упоминалось выше, пузырьковая сортировка отлично подходит для обработки небольших массивов, но при большом числе элементов массива она становится неэффективной. Более универсальным является алгоритм быстрой сортировки, но для его эффективной реализации необходимы языковые средства *Java*, которые будут рассматриваться далее.

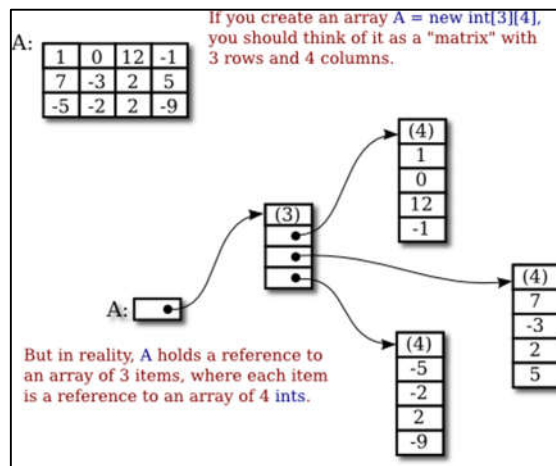
МНОГОМЕРНЫЕ МАССИВЫ

Несмотря на то, что одномерные массивы применяются чаще всего, в программировании, безусловно, применяются и многомерные (двух-, трехмерные и т.д.) массивы. В *Java* многомерные массивы представляют собой массивы массивов.

ДВУМЕРНЫЕ МАССИВЫ



Среди многомерных массивов наиболее простыми являются двумерные массивы. **Двумерный массив, по существу, представляет собой ряд одномерных массивов.**



Для того чтобы объявить двумерный целочисленный табличный массив *table* с размерами 10x20, следует использовать следующую строку:

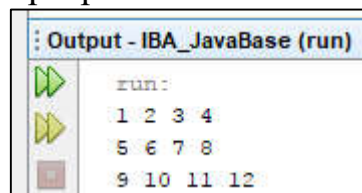
```
int[][] table = new int[10][20];
```

Внимательно посмотрите на это объявление. В отличие от некоторых других языков программирования, где размеры массива разделяются запятыми, в *Java* они заключаются в отдельные квадратные скобки. Так, для обращения к элементу массива *table* по индексам 3 и 5 следует указать *table[3][5]*.

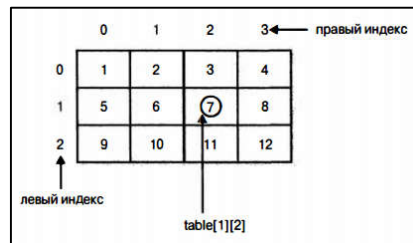
В следующем примере двумерный массив заполняется числами от 1 до 12.

```
//Пример №6. Демонстрация использования двумерного массива
class TwoDimensionArray {
    public static void main(String args[]) {
        int t, i;
        int table[][] = new int[3][4];
        for (t = 0; t < 3; ++t) {
            for (i = 0; i < 4; ++i) {
                table[t][i] = (t * 4) + i + 1;
                System.out.print(table[t][i] + " ");
            }
            System.out.println();
        }
    }
}
```

Результат выполнения программы:



В данном примере элемент *table[0][0]* будет содержать значение 1, элемент *table[0][1]* – значение 2, элемент *table[0][2]* – значение 3 и так далее, а элемент *table[2][3]* – значение 12. Структура данного массива представлена ниже.



НЕРЕГУЛЯРНЫЕ (НЕПРЯМОУГОЛЬНЫЕ, ЗУБЧАТЫЕ) МАССИВЫ

При выделении памяти для многомерного массива достаточно указать лишь первый (крайний слева) размер. Память, соответствующую остальным измерениям массива, можно будет выделять отдельно. Например, в приведенном ниже фрагменте кода память выделяется только под первый размер двумерного массива *table*. Дальнейшее выделение памяти, соответствующей второму измерению, осуществляется вручную.

```
int table[][] = new int[3][];
table[0] = new int[4];
table[1] = new int[4];
table[2] = new int[4];
```

Объявляя массив подобным способом, мы не получаем никаких преимуществ, но в некоторых случаях такое объявление оказывается вполне оправданным. В частности, это дает возможность установить разную длину массива по каждому индексу и сэкономить память.

Ragged Arrays

Since a two-dimensional array in Java is an array of arrays, each row can have a different number of elements (columns).

Arrays in which rows have different numbers of elements are called *ragged arrays*.

Многомерный массив реализован в виде массива массивов, что позволяет контролировать размер каждого из них. Допустим, требуется написать программу, в процессе работы которой будет сохраняться число пассажиров, перевезенных автобусом-экспрессом в аэропорт. Если автобус делает по десять рейсов в будние дни и по два рейса в субботу и воскресенье, то массив *trips* можно объявить так, как показано в приведенном ниже фрагменте кода. Обратите внимание на то, что длина массива по второму индексу для первых пяти элементов равна 10, а для двух последних – 2.

```
//Пример №7. Выделение памяти по второму индексу массива
class Ragged {
    public static void main(String args[]) {
        int trips[][] = new int[7][];
        //для первых пяти элементов длина массива по второму индексу равна
        10
        trips[0] = new int[10];
```

```

        trips[1] = new int[10];
        trips[2] = new int[10];
        trips[3] = new int[10];
        trips[4] = new int[10];
//для последних двух элементов длина массива по второму индексу равна
2
        trips[5] = new int[2];
        trips[6] = new int[2];
        int i, j;
// Сформировать данные о количестве перевезенных пассажиров
        for (i = 0; i < 5; i++) {
            for (j = 0; j < 10; j++) {
                trips[i][j] = i + j + 10;
            }
        }
        for (i = 5; i < 7; i++) {
            for (j = 0; j < 2; j++) {
                trips[i][j] = i + j + 10;
            }
        }
        System.out.println("Количество пассажиров, перевезенных
каждым рейсом в будние дни:");
        for (i = 0; i < 5; i++) {
            for (j = 0; j < 10; j++) {
                System.out.print(trips[i][j] + " ");
            }
            System.out.println();
        }
        System.out.println();
        System.out.println("Количество пассажиров, перевезенных
каждым рейсом в выходные дни:");
        for (i = 5; i < 7; i++) {
            for (j = 0; j < 2; j++) {
                System.out.print(trips[i][j] + " ");
            }
            System.out.println();
        }
    }
}
}
}

```

Результаты работы программы:

```

Количество пассажиров, перевезенных каждым рейсом в будние дни:
10 11 12 13 14 15 16 17 18 19
11 12 13 14 15 16 17 18 19 20
12 13 14 15 16 17 18 19 20 21
13 14 15 16 17 18 19 20 21 22
14 15 16 17 18 19 20 21 22 23

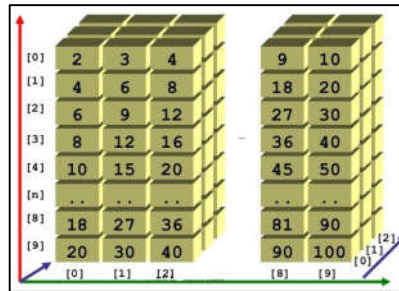
Количество пассажиров, перевезенных каждым рейсом в выходные дни:
15 16
16 17

```

Для большинства приложений использовать нерегулярные массивы не рекомендуется, поскольку это затрудняет восприятие кода другими программистами. Но в некоторых случаях такие массивы вполне уместны и могут существенно повысить эффективность программ. Так, **если вам требуется создать большой двумерный массив, в котором используются не все элементы, нерегулярный массив позволит существенно сэкономить память.**

ТРЕХМЕРНЫЕ, ЧЕТЫРЕХМЕРНЫЕ И МНОГОМЕРНЫЕ МАССИВЫ

В *Java* допускаются массивы размерностью больше двух. Трёхмерный массив можно представить как набор матриц, каждую из которых мы записали на библиотечной карточке или как куб, состоящий из слоёв (*layer*), каждый слой состоит из рядов и столбцов, т.е. каждый слой — это двумерный массив. Тогда чтобы добраться до конкретного числа сначала нужно указать номер карточки (первый индекс трёхмерного массива), потом указать номер строки (второй индекс массива) и только затем номер элемента в строке (третий индекс, номер столбца).



Ниже приведена общая форма объявления многомерного массива.

```
тип[][]...[] имя_массива = new тип[размер_1][размер_2]... [размер_N];
```

В качестве примера ниже приведено объявление трехмерного целочисленного массива с размерами 4x10x3.

```
int[][][] multiDim= new int [4][10][3];
```

ИНИЦИАЛИЗАЦИЯ МНОГОМЕРНЫХ МАССИВОВ

Многомерный массив можно инициализировать, заключая инициализирующую последовательность для каждого размера массива в отдельные фигурные скобки, как показано ниже.

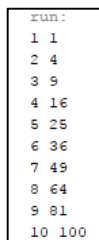
```
тип[][] имя_массива= {  
    { val, val, val, val },  
    { val, val, val,..., val },  
    .  
    .  
    .  
    { val, val, val,..., val }  
};
```

Здесь *val* обозначает начальное значение, которым инициализируются элементы многомерного массива. Каждый внутренний блок многомерного массива соответствует отдельной строке. В каждой строке первое значение сохраняется в первом элементе подмассива, второе значение — во втором элементе и т.д. Обратите внимание на запятые, разделяющие блоки инициализаторов многомерного массива, а также на точку с запятой после закрывающей фигурной скобки.

В следующем фрагменте кода двумерный массив *sgrs* инициализируется числами от 1 до 10 и их квадратами.

```
//Пример №8. Инициализация двумерного массива
class Squares {
    public static void main(String args[]) {
        int sgrs[][] = { //у каждой строки свой набор инициализаторов
            {1, 1},
            {2, 4},
            {3, 9},
            {4, 16},
            {5, 25},
            {6, 36},
            {7, 49},
            {8, 64},
            {9, 81},
            {10, 100}};
        int i, j;
        for (i = 0; i < 10; i++) {
            for (j = 0; j < 2; j++) {
                System.out.print(sgrs[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Результаты работы программы:



```
run:
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
```

АЛЬТЕРНАТИВНЫЙ СИНТАКСИС ОБЪЯВЛЕНИЯ МАССИВОВ

Для объявления массива можно также пользоваться двумя альтернативными синтаксическими конструкциями:

```
тип[] имя_массива;
тип имя_массива[];
```

Квадратные скобки могут указываться как после спецификатора типа, так и после имени массива. Поэтому следующие два объявления равнозначны.

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

Равнозначными являются и приведенные ниже строки кода.

```
char table[][] = new char[3][4];
char[][] table = new char[3][4] ;
```

Объявление массива с указанием квадратных скобок после спецификатора типа оказывается удобным в тех случаях, когда требуется объявить несколько массивов одного типа. Например:

```
int[] nums, nums2, nums3; // создать три массива
```

В этом объявлении создаются три массивы типа *int*. Тот же результат можно получить с помощью следующей строки кода:

```
int nums[], nums2[], nums3[]; // создать три массива
```

Объявление массива с указанием квадратных скобок после спецификатора типа оказывается удобным и в тех случаях, когда в качестве типа, возвращаемого методом, требуется указать массив. Например:

```
int[] someMeth() {...}
```

В этой строке кода объявляется метод *someMeth()*, возвращающий целочисленный массив.

Обе рассмотренные выше формы объявления массивов широко распространены в программировании на *Java*.

ПРИСВАИВАНИЕ ССЫЛОК НА МАССИВЫ

Присваивание значения одной переменной, ссылающейся на массив, другой переменной, означает, что обе переменные ссылаются на один и тот же массив, и в этом отношении массивы ничем не отличаются от любых других объектов. Такое присваивание не приводит ни к созданию копии массива, ни к копированию содержимого одного массива в другой. Это продемонстрировано в приведенном ниже примере программы.

```
//Пример №9. Присваивание ссылок на массивы
class AssignRef {
    public static void main(String args[]) {
        int i;
        int nums1[] = new int[10];
        int nums2[] = new int[10];
        for (i = 0; i < 10; i++) {nums1[i] = i;}
        for (i = 0; i < 10; i++) {nums2[i] = -i;}
        System.out.print("Массив nums1: ");
        for (i = 0; i < 10; i++) {System.out.print(nums1[i] + " ");}
        System.out.println();
        System.out.print("Массив nums2: ");
        for (i = 0; i < 10; i++) {System.out.print(nums2[i] + " ");}
        System.out.println();
        nums2 = nums1; //теперь обе переменные ссылаются на массив nums1
        System.out.print("Массив nums2 после присваивания: ");
        for (i = 0; i < 10; i++) {System.out.print(nums2[i] + " ");}
        System.out.println();
        //Выполнить операции над массивом nums1 через переменную nums2
        nums2[3] = 99;
```

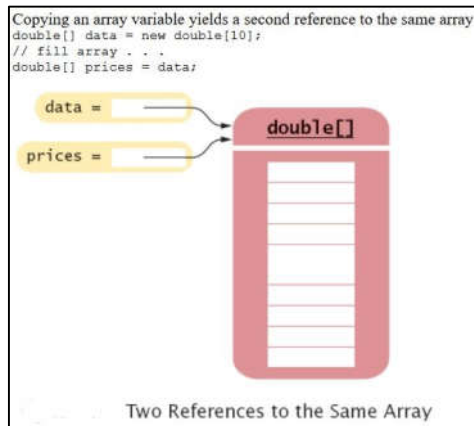


```
System.out.print("Массив nums1 после изменения через nums2: ");
for (i = 0; i < 10; i++) {System.out.print(nums1[i] + " ");}
System.out.println();
}}
```

Результаты работы программы:

```
run:
Массив nums1: 0 1 2 3 4 5 6 7 8 9
Массив nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
Массив nums2 после присвоения: 0 1 2 3 4 5 6 7 8 9
Массив nums1 после изменения через nums2: 0 1 2 99 4 5 6 7 8 9
```

В результате присваивания ссылки на массив *nums1* переменной *nums2* обе переменные ссылаются на один и тот же массив.



ПРИМЕНЕНИЕ ПЕРЕМЕННОЙ ЭКЗЕМПЛЯРА LENGTH

В связи с тем, что массивы реализованы в виде объектов, в каждом массиве содержится переменная экземпляра *length*. Значением этой переменной является число элементов, которые может содержать массив (в переменной *length* содержится размер массива).

1. Длину любого массива можно получить через свойство *length*.
2. `int[] days = {31,29,31,30};`
3. `System.out.println(days.length); // 4`
4. Длину массива нельзя менять
5. Удалять и добавлять новые элементы тоже нельзя
6. При обращении к элементу которого нет, Java выдаст ошибку `IndexOutOfBoundsException`
7. `int tenthDay = days[10]; // Exception`

Ниже приведен пример программы, демонстрирующий свойство *length* массивов.

```
//Пример №10. Использование переменной экземпляра length
class LengthDemo {
    public static void main(String args[]) {
        int list[] = new int[10];
        int nums[] = {1, 2, 3};
        int table[][] = { // таблица со строками переменной длины
            {1, 2, 3},
            {4, 5},
            {6, 7, 8, 9}};
        System.out.println("Размер массива list: " + list.length);
        System.out.println("Размер массива nums: " + nums.length);
    }
}
```

```

        System.out.println("Размер массива table: " + table.length);
        System.out.println("Размер массива table[0]: " +
table[0].length);
        System.out.println("Размер массива table[1]: " +
table[1].length);
        System.out.println("Размер массива table[2]: " +
table[2].length);
        System.out.println();
        // Использовать переменную length для управления массивом
        for (int i = 0; i < list.length; i++){list[i] = i * i;}
        System.out.print("Содержимое массива list:");
        for (int i = 0; i < list.length; i++) {
            System.out.print(list[i] + " ");
        }
        System.out.println();
    }
}

```

Результаты работы программы:

```

Размер массива list: 10
Размер массива nums: 3
Размер массива table: 3
Размер массива table[0]: 3
Размер массива table[1]: 2
Размер массива table[2]: 4

Содержимое массива list:0 1 4 9 16 25 36 49 64 81

```

Обратите внимание на то, каким образом переменная *length* используется в двумерном массиве.



Как пояснялось ранее, двумерный массив представляет собой массив одномерных массивов. Поэтому приведенное ниже выражение позволяет определить число массивов, содержащихся в массиве *table*:

```
table.length
```

В примере *LengthDemo* число таких массивов равно 3. Для того чтобы получить размер отдельного массива, содержащегося в массиве *table*, потребуется выражение, аналогичное следующему:

```
table[0].length
```

Это выражение возвращает размер первого одномерного массива.

Анализируя код класса *LengthDemo*, следует также заметить, что выражение *list.length* используется в цикле *for* для определения требуемого количества итераций.

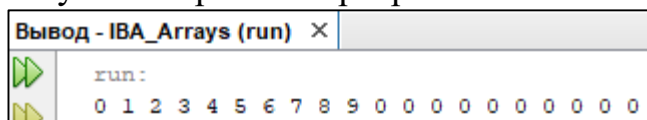
Учитывая то, что у каждого подмассива своя длина, пользоваться таким выражением удобнее, чем отслеживать вручную размеры массивов.

Переменная *length* не имеет никакого отношения к количеству фактически используемых элементов массива. Она содержит лишь данные о том, сколько элементов может содержать массив.

Использование переменной экземпляра *length* позволяет упростить многие алгоритмы. Так, в приведенном ниже примере программы эта переменная используется при копировании одного массива в другой и предотвращает возникновение исключений времени исполнения в связи с выходом за границы массива.

```
//Пример №11.1 Использование переменной length для копирования массивов
class ACopy {
    public static void main(String args[]) {
        int i;
        int nums1[] = new int[10];
        int nums2[] = new int[20];
        for (i = 0; i < nums1.length; i++) {nums1[i] = i;}
// копирование массива nums1 в массив nums2
//Использование переменной length для сравнения размеров массивов
        if (nums2.length >= nums1.length) {
            for (i = 0; i < nums1.length; i++) {nums2[i] = nums1[i];}
        }
        for (i = 0; i < nums2.length; i++) {
            System.out.print(nums2[i] + " ");
        }
    }
}
```

Результаты работы программы:

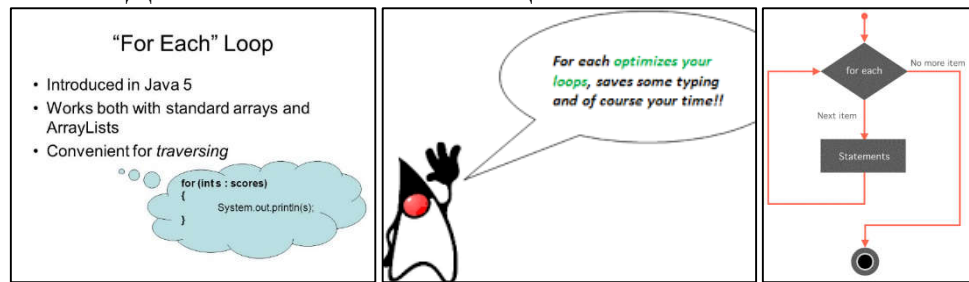


В данном примере переменная экземпляра *length* помогает решить две важные задачи. Во-первых, позволяет убедиться в том, что размера целевого массива (*nums2*) достаточно для хранения содержимого исходного массива (*nums1*). И во-вторых, с ее помощью формируется условие завершения цикла, в котором выполняется копирование массива. Конечно, в столь простом примере размеры массивов нетрудно отследить и без переменной экземпляра *length*, но подобный подход может быть применен для решения более сложных задач.

```
//Пример №11.2 Использование класса Arrays для копирования массивов
import java.util.Arrays;
class Main {
    public static void main(String[] args) {
        float [] numbers = {167.5f, -2, 16.6f, 99.8f, 26, 92, 43.4f,
-234, 35, 80};
```

```
float [] numbersCopy = Arrays.copyOf(numbers, numbers.length);
System.out.println(Arrays.toString(numbersCopy));
}}
```

РАЗНОВИДНОСТЬ FOR-EACH ЦИКЛА FOR



При выполнении операций с массивами очень часто возникают ситуации, когда должен быть обработан каждый элемент массива. Например, для расчета суммы всех значений, содержащихся в массиве, нужно обратиться ко всем его элементам. То же самое приходится делать при расчете среднего значения, поиске элемента и решении многих других задач. В связи с тем, что задачи, предполагающие обработку всего массива, встречаются очень часто, в *Java* была реализована еще одна разновидность цикла *for*, рационализирующая подобные операции с массивами.

Вторая разновидность оператора *for* реализует цикл типа *for-each*. В этом цикле происходит последовательное обращение к каждому элементу совокупности объектов (например, массива, контейнера). За последние годы циклы *for-each* появились практически во всех языках программирования. Изначально в *Java* подобный цикл не был предусмотрен и был реализован лишь в пакете *JDK 5*. Разновидность *for-each* цикла *for* называется также **расширенным циклом *for***. Ниже приведена общая форма разновидности *for-each* цикла *for*.

```
for(тип итр_пер : коллекция) блок_операторов
```

Здесь **тип** обозначает конкретный тип **итр_пер** — **итерационной переменной**, в которой сохраняются поочередно перебираемые элементы набора данных, обозначенного как **коллекция**. В данной разновидности цикла *for* могут быть использованы разные типы коллекций. На каждом шаге цикла очередной элемент извлекается из коллекции и сохраняется в итерационной переменной. Выполнение цикла продолжается до тех пор, пока не будут обработаны все элементы коллекции. Таким образом, при обработке массива размером N в расширенном цикле *for* будут последовательно извлечены элементы с индексами от 0 до $N-1$.

Итерационная переменная получает значения из коллекции, и поэтому ее тип должен совпадать (или, по крайней мере, быть совместимым) с типом элементов, которые содержит коллекция. В частности, при обработке массива тип итерационной переменной должен совпадать с типом массива.

Для того чтобы стали понятнее причины, побудившие к внедрению разновидности *for-each* цикла *for* в *Java*, рассмотрим приведенный ниже

фрагмент кода, в котором традиционный цикл *for* используется для вычисления суммы значений элементов массива.

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0; i < 10; i++) sum += nums[i];
```

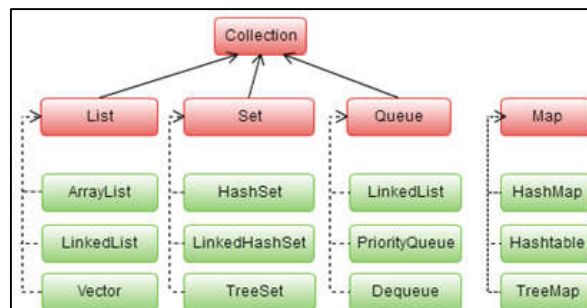
Для того чтобы вычислить сумму элементов массива, придется перебрать все элементы массива *nums* от начала до конца. Перебор элементов осуществляется благодаря использованию переменной цикла *i* в качестве индекса массива *nums*. Кроме того, нужно явно указать начальное значение переменной цикла, шаг ее приращения на каждой итерации и условие завершения цикла.

При использовании разновидности *for-each* данного цикла некоторые перечисленные выше действия выполняются автоматически. В частности, отпадает необходимость в использовании переменной цикла, задании ее исходного значения и условия завершения цикла, а также в индексировании массива. Вместо этого массив автоматически обрабатывается в цикле от начала до конца. Код, позволяющий решить ту же самую задачу с помощью разновидности *for-each* цикла *for*, выглядит следующим образом:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums) sum += x;
```

На каждом шаге этого цикла переменная *x* автоматически принимает значение, равное очередному элементу массива *nums*. Сначала ее значение равно 1, на втором шаге цикла итерации оно становится равным 2 и т.д. В данном случае не только упрощается синтаксис, но и исключается ошибка, связанная с превышением границ массива.

Какие типы коллекций, помимо массивов, можно обрабатывать с помощью разновидности *for-each* цикла *for*? **Наиболее важное применение разновидность *for-each* цикла *for* находит в обработке содержимого коллекции, определенной в *Collections Framework* – библиотеке классов, реализующих различные структуры данных, в том числе списки, векторы, множества и отображения.**



Ниже приведен весь исходный код программы, демонстрирующей решение описанной выше задачи с помощью разновидности *for-each* цикла *for*.

```
//Пример №12. Использование разновидности for-each цикла for
class ForEach {
```

```

public static void main(String args[]) {
    int nums[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int sum = 0;
    //Использовать разновидность for-each цикла for для суммирования и
    отображения значений
    for (int x : nums) {
        System.out.println("Значение: " + x);
        sum += x;
    }
    System.out.println("Сумма: " + sum);
}
}

```

Результат выполнения данной программы выглядит следующим образом.

```

sum:
Значение: 1
Значение: 2
Значение: 3
Значение: 4
Значение: 5
Значение: 6
Значение: 7
Значение: 8
Значение: 9
Значение: 10
Сумма: 55

```

Нетрудно заметить, что в данной разновидности цикла *for* элементы массива автоматически извлекаются один за другим в порядке возрастания индекса.

Несмотря на то, что в расширенном цикле *for* обрабатываются все элементы массива, этот цикл можно завершить преждевременно, используя оператор *break*. Так, в цикле, используемом в следующем примере, вычисляется сумма только пяти элементов массива *nums*.

```

//Суммирование первых 5 элементов массива
for(int x : nums) {
    System.out.println("Значение: " + x);
    sum += x;
    if(x == 5) break; //прервать цикл по достижении значения элемента
    массива со значением 5
}

```

Следует иметь в виду одну важную особенность разновидности *for-each* цикла *for*. **Итерационная переменная в этом цикле обеспечивает только чтение элементов массива, но ее нельзя использовать для записи значения в какой-либо элемент массива.** Иными словами, изменить содержимое массива, присвоив итерационной переменной новое значение, не удастся. Рассмотрим в качестве примера следующую программу.

```

//Пример №13. Циклы for-each предназначены только для чтения.
САМОСТОЯТЕЛЬНО. ИЗМЕНИТЬ ПРОГРАММУ ТАК, ЧТОБЫ В ПЕРВОМ ЦИКЛЕ ЗНАЧЕНИЯ
ЭЛЕМЕНТОВ МАССИВА NUMS БЫЛИ ИЗМЕНЕНЫ
class NoChange {
    public static void main(String args[]) {
        int nums[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        for (int x : nums) {
            System.out.print(x + " ");
        }
    }
}

```



```

        x = x * 10; // Эта операция фактически не изменяет содержимое
массива nums
    }
    System.out.println();
    for (int x : nums) { System.out.print(x + " "); }
    System.out.println();
}
}

```

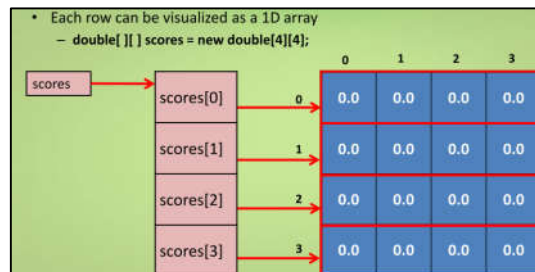
В первом цикле *for* значение итерационной переменной увеличивается на 10, но это не оказывает никакого влияния на содержимое массива *nums*, что и демонстрирует второй цикл *for*. Это же подтверждает и результат выполнения программы. Результаты работы программы:

```

run:
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10

```

ЦИКЛИЧЕСКОЕ ОБРАЩЕНИЕ К ЭЛЕМЕНТАМ МНОГОМЕРНЫХ МАССИВОВ



Расширенный цикл *for* используется также при работе с многомерными массивами. Выше уже говорилось, что в *Java* многомерный массив представляет собой массив массивов. Например, двумерный массив – это массив, элементами которого являются одномерные массивы. **Эту особенность важно помнить, организуя циклическое обращение к многомерным массивам, поскольку на каждом шаге цикла извлекается очередной массив, а не отдельный его элемент.** Более того, итерационная переменная в расширенном цикле *for* должна иметь тип, совместимый с типом извлекаемого массива. Так, при обращении к двумерному массиву итерационная переменная должна представлять собой ссылку на одномерный массив. При использовании разновидности *for-each* цикла *for* для обработки *N*-мерного массива извлекаемый объект представляет собой (*N*-1)-мерный массив.

Рассмотрим приведенный ниже пример программы, где для извлечения элементов двумерного массива используются вложенные циклы *for*. Обратите внимание на то, каким образом объявляется переменная *x*.

```

//Пример №14. Использование расширенного цикла for для обработки
двумерного массива
class ForEach2 {
    public static void main(String args[]) {
        int sum = 0;
        int nums[][] = new int[3][5];
        // Ввести ряд значений в массив nums
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 5; j++) {
                nums[i][j] = (i + 1) * (j + 1);
            }
        }
    }
}

```

```

    }
}
//Использовать разновидность for-each цикла for для суммирования и
отображения значений, обратите внимание на объявление переменной x
    for (int x[] : nums) {
        for (int y : x) {
            System.out.print(y+" ");
            sum += y;
        }
        System.out.println();
    }
    System.out.println("Сумма: " + sum);
}
}

```

Результаты выполнения программы:

```

run:
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
Сумма: 90

```

Обратите внимание на следующую строку кода:

```
for(int x[] : nums) {
```

Здесь переменная *x* представляет собой ссылку на одномерный целочисленный массив. Это очень важно, поскольку на каждом шаге цикла *for* из двумерного массива *nums* извлекается очередной массив, начиная с *nums[0]*. А во внутреннем цикле *for* перебираются элементы полученного массива и отображаются их значения.

ИСПОЛЬЗОВАНИЕ РАСШИРЕННОГО ЦИКЛА FOR

Разновидность *for-each* цикла *for* обеспечивает лишь последовательный перебор элементов от начала до конца массива, поэтому может создаться впечатление, будто такой цикл имеет ограниченное применение. Но это совсем не так. Данный механизм циклического обращения применяется в самых разных алгоритмах. Один из самых характерных тому примеров – организация поиска. В приведенном ниже примере программы расширенный цикл *for* используется для поиска значения в неотсортированном массиве. Выполнение цикла прерывается, если искомый элемент найден.

```

//Пример №15. Поиск в массиве с использованием цикла for-each
class Search {
    public static void main(String args[]) {
        int nums[] = {6, 8, 3, 7, 5, 6, 1, 4};
        int val = 5;
        boolean found = false;
        //Использовать цикл for-each для поиска значения переменной val в
        массиве nums
        for (int x : nums) {
            if (x == val) {
                found = true; break;
            }
        }
        if (found) {

```

```

        System.out.println("Значение найдено!");
    }
}

```

Результаты работы программы:

```

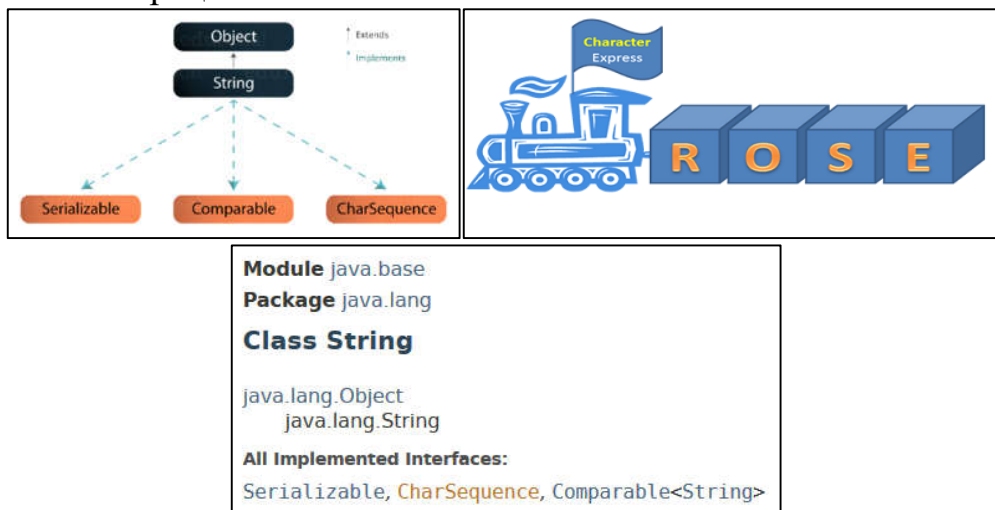
run:
Значение найдено!

```

В данном случае применение расширенного цикла *for* вполне оправданно, поскольку найти значение в неотсортированном массиве можно лишь, перебрав все его элементы. Если бы содержимое массива было предварительно отсортировано, то лучше было бы применить более эффективный алгоритм поиска, например, двоичный поиск. В этом случае нам пришлось бы использовать другой цикл. Расширенным циклом *for* удобно также пользоваться для расчета среднего значения, нахождения минимального и максимального элементов множества, выявления дубликатов значений и т.п.

СИМВОЛЬНЫЕ СТРОКИ

В повседневной работе каждый программист обязательно встречается с объектами типа *String*. Объект типа *String* определяет символьную строку и поддерживает операции с ней.



Во многих языках программирования символьная строка (или просто строка) – это массив символов, но в Java строки – это объекты.

Персистентность строк в Java

Стандартные строки в Java являются **персистентными** (constant, immutable), и после создания строки в ней уже не получится что-либо изменить. Причина – обеспечение целостности текстовых данных при работе приложения в режиме нескольких потоков. Все строковые методы возвращают новые строки, не меняя оригинальные данные.

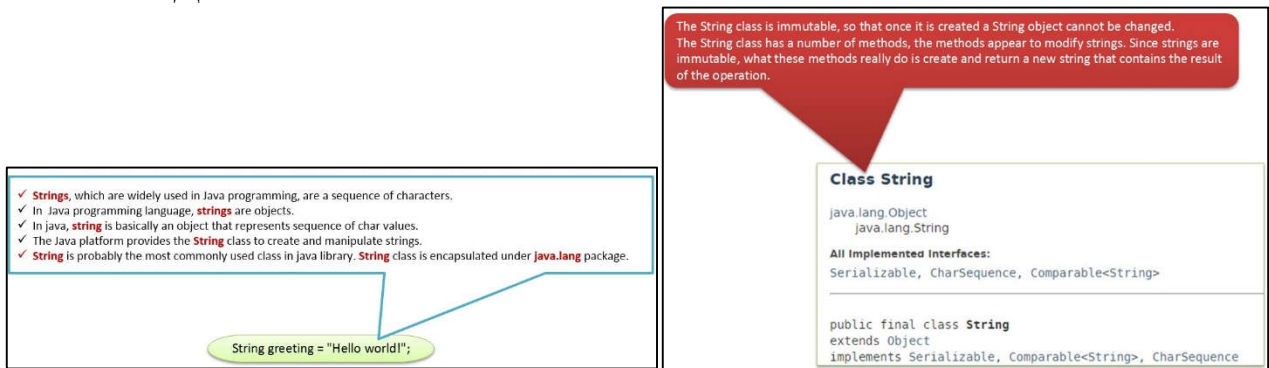
Класс *String* фактически уже использовался в примерах рассмотренных программ. При создании строкового литерала на самом деле генерировался объект типа *String*. Рассмотрим приведенный ниже оператор:

```
System.out.println("В Java строки - объекты.");
```

Наличие в нем строки "В Java строки – объекты." автоматически приводит к созданию объекта типа *String*. Следовательно, класс *String* незримо присутствовал в предыдущих примерах программ.

Следует отметить, что класс *String* настолько обширен, что мы сможем рассмотреть лишь незначительную его часть. Большую часть функциональных возможностей класса *String* вам предстоит изучить самостоятельно.

СОЗДАНИЕ СТРОК



Объекты типа *String* создаются таким же образом, как и объекты других типов. Для этой цели используется конструктор, как показано в следующем примере:

```
String str = new String("Привет");
```

В данном примере создается объект *str* типа *String*, содержащий строку "Привет". Объект типа *String* можно создать и на основе другого объекта такого же типа, как показано ниже.

```
String str = new String("Привет");  
String str2 = new String(str);
```

После выполнения этих строк кода объект *str2* будет также содержать строку "Привет". Ниже представлен еще один способ создания объекта типа *String*.

```
String str = "Строки Java эффективны.";
```

В данном случае объект *str* инициализируется последовательностью символов "Строки Java эффективны."

Создав объект типа *String*, можете использовать его везде, где допускается строковый литерал (последовательность символов, заключенная в двойные кавычки). Например, объект типа *String* можно передать в качестве параметра методу *println()* при его вызове.

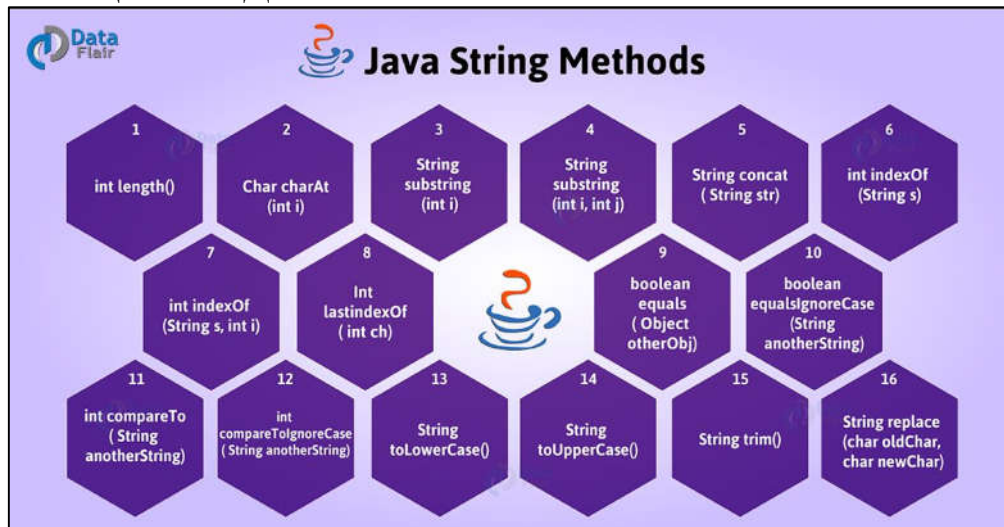
```
//Пример №16. Знакомство с классом String  
class StringDemo {  
    public static void main(String args[]) {  
        // Различные способы объявления строк
```

```
String str1 = new String("В Java строки - объекты.");
String str2 = "Их можно создавать разными способами.";
String str3 = new String(str2);
System.out.println(str1);
System.out.println(str2);
System.out.println(str3);
}}
```

Результат выполнения программы:

```
run:
В Java строки - объекты.
Их можно создавать разными способами.
Их можно создавать разными способами.
```

ОПЕРАЦИИ НАД СИМВОЛЬНЫМИ СТРОКАМИ



Класс *String* содержит ряд методов, предназначенных для манипулирования строками. Ниже описаны некоторые из них.

Метод	Назначение
<i>boolean equals(Object anObject)</i>	Сравнение строк с учетом регистра. Возвращает логическое значение <i>true</i> , если текущая строка содержит ту же последовательность символов, что и параметр <i>anObject</i> с учетом регистра.
<i>boolean equalsIgnoreCase(String anotherString)</i>	Сравнение строк без учета регистра. Возвращает логическое значение <i>true</i> , если текущая строка содержит ту же последовательность символов, что и параметр <i>anotherString</i> без учета регистра
<i>int length()</i>	Возвращает длину строки
<i>char charAt(int index)</i>	Возвращает символ, занимающий в строке позицию, указываемую параметром <i>index</i>
<i>int compareTo(String anotherString)</i>	Лексикографическое сравнение строк с учетом регистра. Возвращает отрицательное значение, если текущая строка меньше строки <i>anotherString</i> ; нуль, если эти строки равны, и положительное значение, если текущая строка больше строки <i>anotherString</i> .
<i>int compareToIgnoreCase(String anotherString)</i>	Лексикографическое сравнение строк без учета регистра. Возвращает отрицательное значение, если текущая строка меньше строки <i>anotherString</i> ; нуль, если эти строки равны, и положительное значение, если текущая строка больше строки <i>anotherString</i>

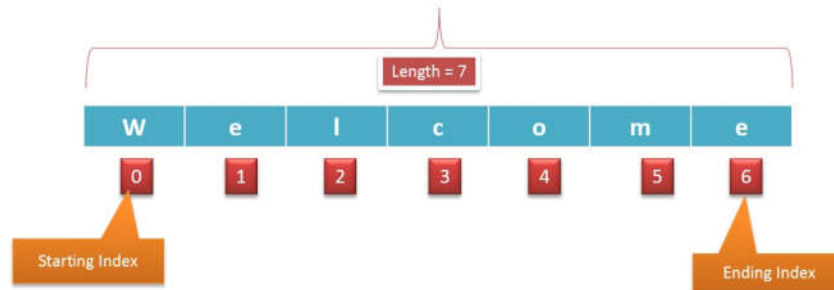
<i>int indexOf(String str)</i>	Выполняет в текущей строке поиск подстроки, определяемой параметром <i>str</i> . Возвращает индекс первого вхождения подстроки <i>str</i> или -1, если поиск завершается неудачно
<i>int lastIndexOf(String str)</i>	Производит в текущей строке поиск подстроки, определяемой параметром <i>str</i> . Возвращает индекс последнего вхождения подстроки <i>str</i> или -1, если поиск завершается неудачно

public boolean equals (Object string)

Метод `equals`(строка) – возвращает **true**, если сравниваемые строки равны, т.е. содержит те же символы и в том же порядке с учётом регистра.

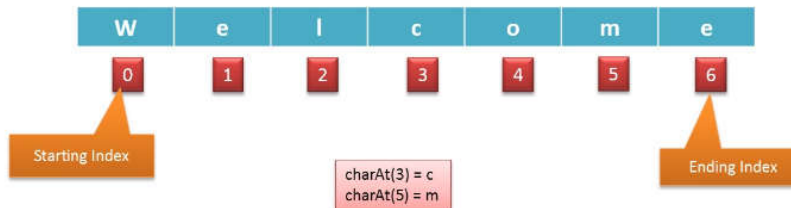
String – length Method

`length()` method returns the length of this string.



Getting Characters

We can get the character at a particular index within a string by invoking the `charAt()` accessor method. The index of the first character is 0, while the index of the last character is `length()-1`.



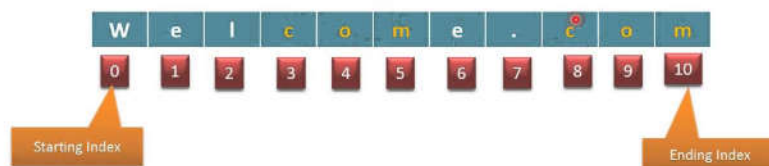
Comparing Strings

- We cannot use the relational operators to compare strings
- The `String` class contains a method called `compareTo` to determine if one string comes before another
- A call to `name1.compareTo(name2)`
 - returns zero if `name1` and `name2` are equal (contain the same characters)
 - returns a negative value if `name1` is less than `name2`
 - returns a positive value if `name1` is greater than `name2`

String – indexOf method

Method	Description
<code>int indexOf(String str)</code>	Returns the index of the first occurrence of the specified substring.

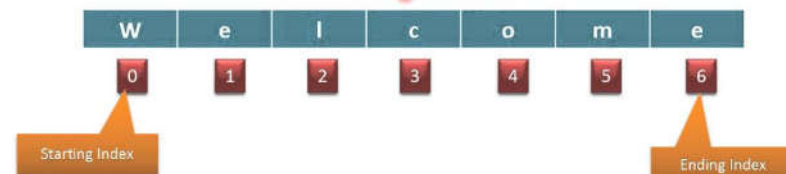
"Welcome.com".indexOf("com") = 3



String – lastIndexOf method

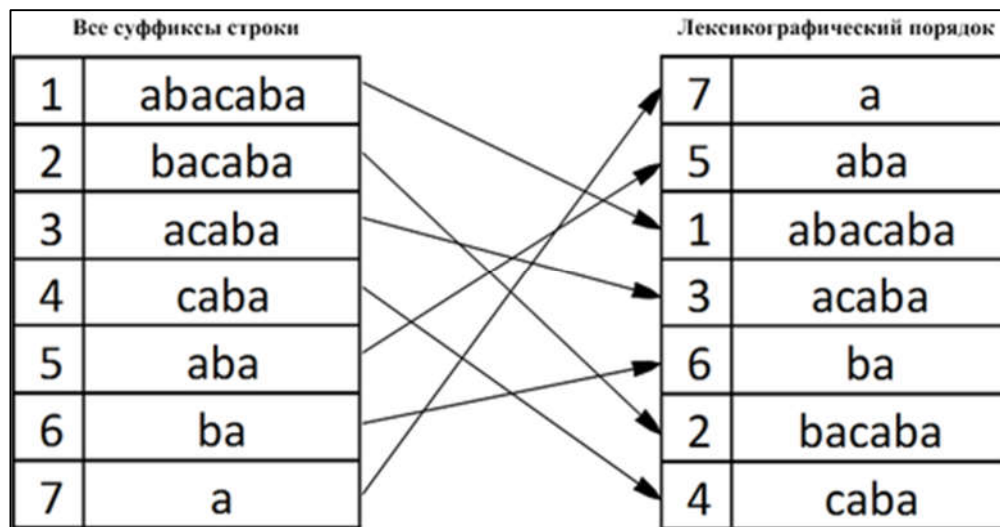
Method	Description
<code>int lastIndexOf(int ch)</code>	Returns the index of the last occurrence of the specified character.

"Welcome".lastIndexOf('e') = 6



Метод *equals()* и оператор `==` выполняют две совершенно различных проверки. Если метод *equals()* сравнивает символы внутри строк, то оператор `==` сравнивает две переменные-ссылки на объекты и проверяет, указывают ли они на разные объекты или на один и тот же.

Зачастую бывает недостаточно просто знать, являются ли две строки идентичными. Для приложений, в которых требуется сортировка, нужно знать, какая из двух строк меньше другой. Для ответа на этот вопрос нужно воспользоваться методом *compareTo()* класса *String*. Этот метод сравнивает две строки лексикографически. Если целое значение, возвращенное методом, отрицательно, то строка, для которой был вызван метод, меньше (находится раньше по алфавиту) строки-параметра, если положительно — больше (находится позже по алфавиту). Если же метод *compareTo()* вернул значение 0, то строки идентичны, в таком случае метод *equals(String str)* так же вернет *true*.



В приведенном ниже примере программы демонстрируется применение перечисленных выше методов, оперирующих строками.

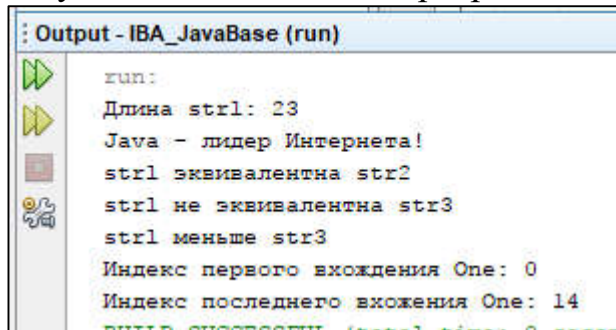
```
//Пример №17. Некоторые операции над строкам
class StrOps {
    public static void main(String args[]) {
        String str1 = "Java - лидер Интернета!";
        String str2 = new String(str1);
        String str3 = "Строки Java эффективны.";
        int result, idx;
        char ch;
        System.out.println("Длина str1: " + str1.length());
        // Отобразить строку str1 посимвольно
        for (int i = 0; i < str1.length(); i++) {
            System.out.print(str1.charAt(i));
        }
        System.out.println();
        //Проверка эквивалентности строк
```

```

if (str1.equals(str2)) {
    System.out.println("str1 эквивалентна str2");
} else {
    System.out.println("str1 не эквивалентна str2");
}
if (str1.equals(str3)) {
    System.out.println("str1 эквивалентна str3");
} else {
    System.out.println("str1 не эквивалентна str3");
}
result = str1.compareTo(str3);
if (result == 0) {
    System.out.println("str1 и str3 равны");
} else if (result < 0) {
    System.out.println("str1 меньше str3");
} else {
    System.out.println("str1 больше str3");
}
//Присвоить переменной str2 новую строку
str2 = "One Two Three One";
idx = str2.indexOf("One");
System.out.println("Индекс первого вхождения One: " + idx);
idx = str2.lastIndexOf("One");
System.out.println("Индекс последнего вхождения One: " + idx);
}}

```

Результаты выполнения программы:

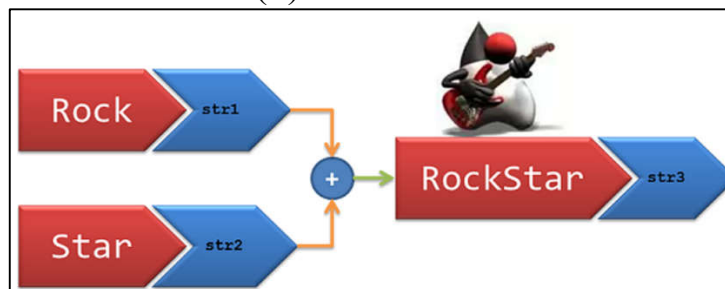


```

Output - IBA_JavaBase (run)
run:
Длина str1: 23
Java - лидер Интернета!
str1 эквивалентна str2
str1 не эквивалентна str3
str1 меньше str3
Индекс первого вхождения One: 0
Индекс последнего вхождения One: 14
BUILD SUCCESSFUL (total time: 0 sec)

```

Конкатенация – операция, позволяющая объединить две строки. В коде она обозначается знаком "плюс" (+).



Например, в приведенном ниже коде переменная *str4* инициализируется строкой "OneTwoThree".

```
String str1 ="One";
String str2 ="Two";
String str3 ="Three";
String str4= str1 + str2 + str3;
```

МАССИВЫ СТРОК

Подобно другим типам данных, строки можно объединять в массивы. Ниже приведен соответствующий демонстрационный пример.

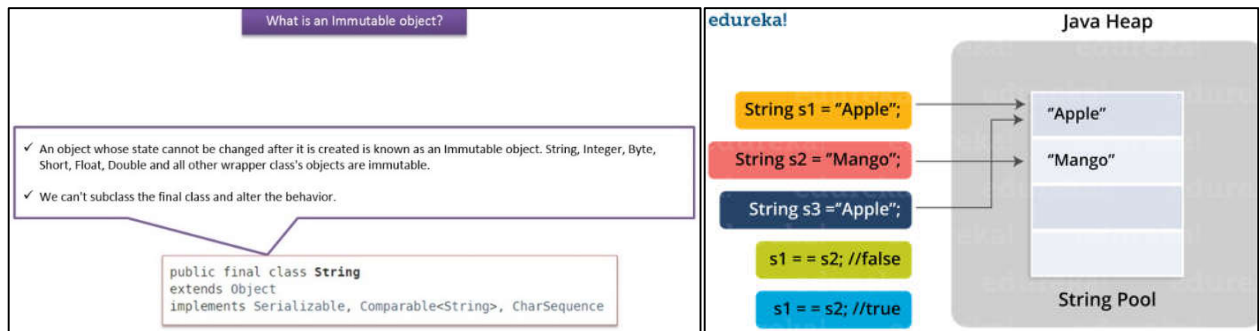
```
//Пример №18. Демонстрация использования массивов строк
class StringArrays {
    public static void main(String args[]) {
        String strs[] = {"Эта", "строка", "является", "тестом."};
        System.out.println("Исходный массив: ");
        for (String s : strs) {System.out.print(s + " ");}
        System.out.println("\n");
        // Изменить строку
        strs[2] = "также является";
        strs[3] = "тестом!";
        System.out.println("Измененный массив: ");
        for (String s : strs) {System.out.print(s + " ");}
    }
}
```

Результат работы программы:

```
run:
Исходный массив:
Эта строка является тестом.

Измененный массив:
Эта строка также является тестом!
```

НЕИЗМЕНЯЕМОСТЬ СТРОК



Объекты типа *String* являются неизменяемыми объектами. Это означает, что состояние такого объекта не может быть изменено после его создания. Такое ограничение способствует наиболее эффективной реализации строк. Поэтому на первый взгляд очевидный недостаток на самом деле превращается в преимущество. Так, если требуется видоизменение уже существующей строки, то для этой цели следует создать новую строку, содержащую все необходимые изменения. А поскольку неиспользуемые строковые объекты автоматически

удаляются сборщиком мусора, то о дальнейшей судьбе ненужных строк можно не беспокоиться.

Следует, однако, иметь в виду, что содержимое ссылочных переменных типа *String* может изменяться, приводя к тому, что переменная будет ссылаться на другой объект, но содержимое самих объектов типа *String* остается неизменным после того, как они были созданы.

Для того чтобы стало понятнее, почему неизменяемость строк не является помехой, воспользуемся еще одним способом обработки строк класса *String*. Здесь имеется в виду метод *substring()*, возвращающий новую строку, которая содержит часть вызывающей строки. В итоге создается новый строковый объект, содержащий выбранную подстроку, тогда как исходная строка не меняется, а, следовательно, соблюдается принцип постоянства строк. Ниже приведен общий синтаксис объявления метода *substring()*:

```
string substring(int начальный_индекс, int конечный_индекс)
```

где **начальный_индекс** обозначает начало извлекаемой подстроки, а **конечный_индекс** – ее окончание.

String – Substrings by Index	
Method	Description
String substring(int beginIndex)	Returns a new string that is a substring of this string. The integer argument specifies the index of the first character. Here, the returned substring extends to the end of the original string.

str.substring(..) → returns a new String by copying characters from an existing String.
• str.substring (i, k) – returns substring of chars from pos i to k-1
• str.substring (i); – returns substring from the i-th char to the end

Ниже приведен пример программы, демонстрирующий применение метода *substring()* и принцип неизменяемости строк.

```
//Пример №19. Применение метода substring()
class SubStr {
    public static void main(String args[]) {
        String orgStr = "Java - двигатель Интернета.";
        //Сформировать подстроку
        String subStr = orgStr.substring(7, 25); //Здесь создается
        новая строка, содержащая нужную подстроку
        System.out.println("orgStr: " + orgStr);
        System.out.println("subStr: " + subStr);
    }
}
```

Результат выполнения программы:

```
run:
orgstr: Java - двигатель Интернета.
substr: двигатель Интернет
```


Исходная строка *orgStr* остается неизменной, а новая строка *subStr* содержит сформированную подстроку.

Как пояснялось выше, **содержимое однажды созданного объекта типа *String* не может быть изменено после его создания**. С практической точки зрения это не является серьезным ограничением, но что если нужно создать строку, которая может изменяться? В *Java* предоставляется класс *StringBuffer*, который создает символьные строки, способные изменяться.

String objects are immutable	String Buffer objects are mutable
<p>StringBuffer – это строковый буфер переменной длины. Создать объект класса StringBuffer можно только с помощью конструкторов: StringBuffer() - пустой строковый буфер с емкостью 16 символов. StringBuffer(int length) - пустой строковой буфер с емкостью length StringBuffer(String str) - строковый буфер емкостью str.length()+16, содержащий строку str. Если строковый буфер начинает переполняться, его емкость автоматически увеличивается.</p>	

Так, в дополнение к методу *charAt()*, возвращающему символ из указанного места в строке, в кассе *StringBuffer* определен метод *setCharAt()*, вставляющий символ в строку. Но для большинства целей вполне подходит класс *String*.

- ✓ Java StringBuffer class is used to create **mutable (modifiable)** string. The StringBuffer class in java is same as String class except it is **mutable** i.e. it can be changed.
- ✓ Java StringBuffer class is **thread-safe** i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.
- ✓ StringBuffer represents growable and writable character sequence.

StringBuffer – setCharAt method	
Method	Description
public void setCharAt(int index, char ch)	<p>The character at the specified index is set to ch. This sequence is altered to represent a new character sequence that is identical to the old character sequence, except that it contains the character ch at position index.</p> <p>The index argument must be greater than or equal to 0, and less than the length of this sequence.</p>

```
String str = "Dig";
StringBuffer sb = new StringBuffer(str);
sb.setCharAt(1, 'o'); // "Dig" will change to "Dio"
```

D	i	g
0	1	2

Starting Index
Ending Index

StringBuffer()	Constructs a string buffer with no characters in it and an initial capacity of 16 characters.
StringBuffer(CharSequence seq)	Constructs a string buffer that contains the same characters as the specified CharSequence.
StringBuffer(int capacity)	Constructs a string buffer with no characters in it and the specified initial capacity.
StringBuffer(String str)	Constructs a string buffer initialized to the contents of the specified string.

ИСПОЛЬЗОВАНИЕ СТРОК ДЛЯ УПРАВЛЕНИЯ ОПЕРАТОРОМ SWITCH

До появления версии *JDK 7* для управления оператором *switch* приходилось использовать только константы целочисленных типов, таких как *int* или *char*. Это препятствовало применению оператора *switch* в тех случаях, когда выбор варианта определялся содержимым строки. В качестве выхода из этого положения зачастую приходилось обращаться к многоступенчатой конструкции *if-else-if*. И хотя эта конструкция семантически правильна, для организации подобного выбора более естественным было бы применение оператора *switch*.

Этот недостаток был исправлен. После выпуска комплекта *JDK 7* появилась возможность управлять оператором *switch* с помощью объектов типа *String*. Во многих ситуациях это способствует написанию более удобочитаемого и рационально организованного кода.

The switch Statement

```

1 class SwitchDemo
2 {
3     public static void main(String[] args)
4     {
5         int month = 3;
6         String monthValue;
7         switch (month)
8         {
9             case 1:
10                monthValue = "January";
11                break;
12             case 2:
13                monthValue = "February";
14                break;
15             case 3:
16                monthValue = "March";
17                break;
18             case 4:
19                monthValue = "April";
20                break;
21             case 5:
22                monthValue = "May";
23                break;
24             case 6:
25                monthValue = "June";
26                break;
27             case 7:
28                monthValue = "July";
29                break;
30             case 8:
31                monthValue = "August";
32                break;
33             case 9:
34                monthValue = "September";
35                break;
36             case 10:
37                monthValue = "October";
38                break;
39             case 11:
40                monthValue = "November";
41                break;
42             case 12:
43                monthValue = "December";
44                break;
45             default:
46                monthValue = "Invalid month";
47                break;
48         }
49         System.out.println("Month is : " + monthValue);
50     }
51 }
                    
```

- ✓ Unlike if then and if then-else statements, the switch statement can have a number of possible execution paths.
- ✓ A switch works with the byte, short, char, and int primitive data types. It also works with enumerated types, the String class, and a few special classes that wrap certain primitive types: Character, Byte, Short, and Integer.
- ✓ The body of a switch statement is known as a switch block. A statement in the switch block can be labeled with one or more case or default labels. The switch statement evaluates its expression, then executes all statements that follow the matching case label.
- ✓ Each break statement terminates the enclosing switch statement. Control flow continues with the first statement following the switch block.

The switch Statement

```

6         int month = 3;
7         String monthValue;
8         if (month == 1)
9         {
10            monthValue = "January";
11        }
12        else if (month == 2)
13        {
14            monthValue = "February";
15        }
16        else if (month == 3)
17        {
18            monthValue = "March";
19        }
20        // and so on
                    
```

- ✓ Deciding whether to use if-then-else statements or a switch statement is based on readability and the expression that the statement is testing.
- ✓ An if-then-else statement can test expressions based on ranges of values or conditions, whereas a switch statement tests expressions based only on a single integer, enumerated value, or String object.

Ниже приведен пример программы, демонстрирующий управление оператором *switch* с помощью объектов типа *String*.

```

import java.util.Scanner;
//Пример №20. Использование строк для управления оператором switch
class StringSwitch {
    public static void main(String args[]) {
        do {
            Scanner in = new Scanner(System.in);
            System.out.println("Введите команду: ");
            String command = in.nextLine();
            switch (command) {
                case "connect":
                    System.out.println("Набор операторов для
выполнения команды \"Соединение\"");
                    break;
                case "cancel":
                    System.out.println("Набор операторов для
выполнения команды \"Отмена действия\"");
                    break;
                case "disconnect":
                    System.out.println("Набор операторов для
выполнения команды \"Разъединение\"");
                    break;
                default:
                    System.out.println("Неверная команда!");
                    break;
            }
        } while (true);
    }
}
                    
```

Результат выполнения программы:

```
Введите команду:
avava
Неверная команда!
Введите команду:
cancel
Набор операторов для выполнения команды "Отмена действия"
Введите команду:
disconnect
Набор операторов для выполнения команды "Разъединение"
Введите команду:
connect
Набор операторов для выполнения команды "Соединение"
Введите команду:
```

Строка, содержащаяся в переменной *command*, а в данном примере это строка "*cancel*" (отмена), проверяется на совпадение со строковыми константами в ветвях *case* оператора *switch*. Если совпадение обнаружено, как это имеет место во второй ветви *case*, то выполняется код, связанный с данным вариантом выбора.

Возможность использования строк в операторе *switch* очень удобна и позволяет сделать код более удобочитаемым. В частности, применение оператора *switch*, управляемого строками, является более совершенным решением по сравнению с эквивалентной последовательностью операторов *if-else*. Но если учитывать накладные расходы, то использование строк для управления переключателями оказывается менее эффективным по сравнению с целочисленными значениями. Поэтому **использовать строки для данной цели целесообразно лишь в тех случаях, когда управляющие данные уже являются строками**. Иными словами, пользоваться строками в операторе *switch* без особой надобности не следует.

ИСПОЛЬЗОВАНИЕ АРГУМЕНТОВ КОМАНДНОЙ СТРОКИ

Теперь, когда вы уже познакомились с классом *String*, можно пояснить назначение параметра *args* метода *main()*. Многие программы получают параметры, задаваемые в командной строке, в качестве параметров запуска приложения. Это так называемые **аргументы командной строки**. Они представляют собой данные, указываемые непосредственно после имени запускаемой на выполнение программы и используемые приложением для своего первоначального запуска. Для того чтобы получить доступ к аргументам командной строки из программы на *Java*, достаточно обратиться к массиву объектов типа *String*, который передается методу *main()*. Рассмотрим в качестве примера программу, отображающую параметры командной строки.

```
//Пример №21. Отображение всех данных, указываемых в командной строке
public class CLDemo {
    public static void main(String args[]) {
        System.out.println("Программе передано " + args.length + "
аргументов командной строки.");
        System.out.println("Список аргументов: ");
        for (int i = 0; i < args.length; i++) {
            System.out.println("arg[" + i + "]: " + args[i]);
        }
    }
}
```

```
}}}
```

Если класс *CLDemo* будет запущен на выполнение из командной строки без указания аргументов приложения, то результат его выполнения будет выглядеть следующим образом:

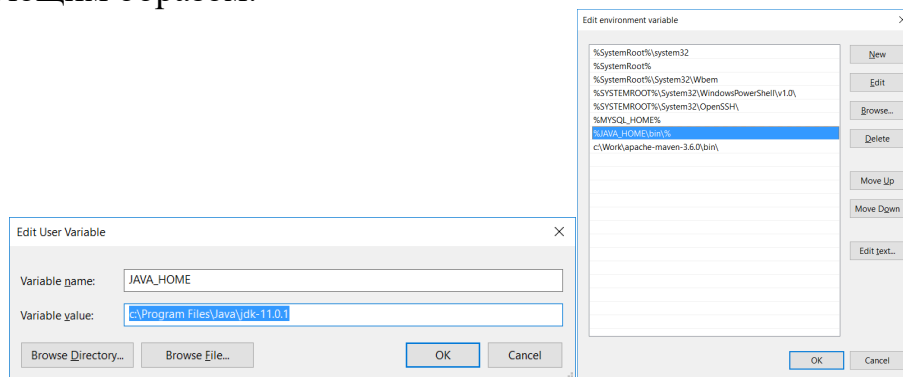
```
C:\Work\Java\Programs\Examples\untitled\src>javac -cp . MyPack/CLDemo.java
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF8

C:\Work\Java\Programs\Examples\untitled\src>java -cp . MyPack.CLDemo
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF8
Программе передано 0 аргументов командной строки.
Список аргументов:
```

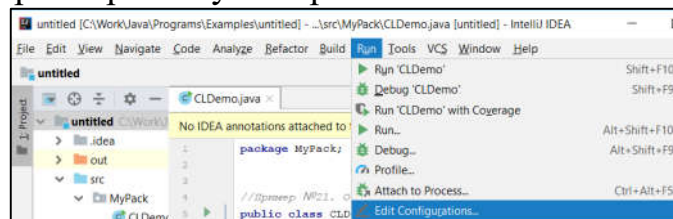
Если класс *CLDemo* будет запущен на выполнение из командной строки с указанием аргументов приложения, то результат его выполнения будет выглядеть следующим образом:

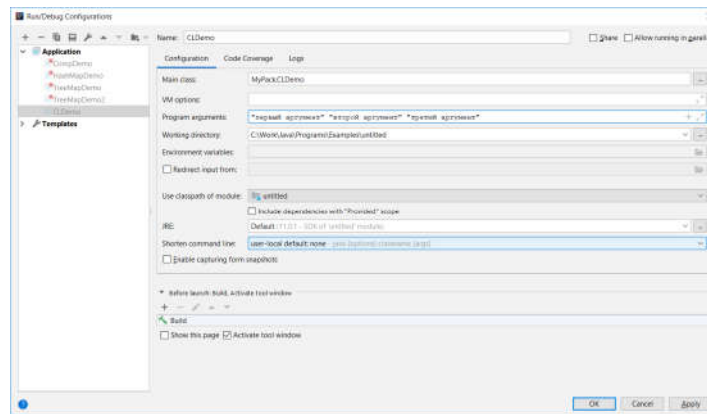
```
C:\Work\Java\Programs\Examples\untitled\src>java -cp . MyPack.CLDemo "первый аргумент" "второй аргумент" "третий аргумент"
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF8
Программе передано 3 аргументов командной строки.
Список аргументов:
arg[0]: первый аргумент
arg[1]: второй аргумент
arg[2]: третий аргумент
```

Для того, чтобы команды `java` и `javac` были видны из любой директории на компьютере, следует задать значения для переменных окружения *JAVA_HOME* и *PATH* следующим образом:



Для указания аргументов командной строки приложения в *IDE* необходимо отредактировать параметры запуска приложения:





При запуске приложения можно воспользоваться переданными аргументами:

```
Программе передано 3 аргументов командной строки.
Список аргументов:
arg[0]: первый аргумент
arg[1]: второй аргумент
arg[2]: третий аргумент
```

Обратите внимание на то, что первый аргумент содержится в строке, хранящейся в элементе массива с индексом 0. Для доступа ко второму аргументу следует воспользоваться индексом 1 и т.д.

Для того чтобы стало понятнее, как пользоваться аргументами командной строки, рассмотрим следующую программу. Эта программа принимает один аргумент, определяющий имя абонента, а затем производит поиск имени в двумерном массиве строк. Если имя найдено, программа отображает телефонный номер указанного абонента.

```
//Пример №22. Простейший автоматизированный телефонный справочник
class Phone {
    public static void main(String args[]) {
        String numbers[][] = {
            {"Tom", "555-3322"},
            {"Mary", "555-8976"},
            {"Jon", "555-1037"},
            {"Rachel", "555-1400"}};

        int i;
        //чтобы воспользоваться программой, ей нужно передать один аргумент
        //командной строки
        if (args.length != 1) {
            System.out.println("Использование: java Phone <имя>");
        } else {
            for (i = 0; i < numbers.length; i++) {
                if (numbers[i][0].equals(args[0])) {
                    System.out.println(numbers[i][0]+": " + numbers[i][1]);
                    break;
                }
            }
            if (i == numbers.length) {
```

```

        System.out.println("Имя не найдено.");
    }
}
}
}

```

Выполнение этой программы в командной строке и в *IDE*:

```

C:\Users\USER\Documents\NetBeansProjects\IBA_JavaBase\build\classes\iba_javabase>java -classpath c:\Users\USER\Documents\NetBeansProjects\IBA_JavaBase\build\classes\ iba_javabase.Phone
Использование: java Phone <имя>

C:\Users\USER\Documents\NetBeansProjects\IBA_JavaBase\build\classes\iba_javabase>java -classpath c:\Users\USER\Documents\NetBeansProjects\IBA_JavaBase\build\classes\ iba_javabase.Phone Mary
Mary: 555-8976

C:\Users\USER\Documents\NetBeansProjects\IBA_JavaBase\build\classes\iba_javabase>java -classpath c:\Users\USER\Documents\NetBeansProjects\IBA_JavaBase\build\classes\ iba_javabase.Phone Rachel
Rachel: 555-1400

C:\Users\USER\Documents\NetBeansProjects\IBA_JavaBase\build\classes\iba_javabase>

```

```

run:
Mary: 555-8976

```

ОПЕРАТОР ? :

Оператор ? является одним из самых удобных в *Java*. Он часто используется вместо операторов *if-else* следующего вида.

```

if (условие) переменная выражение_1;
else переменная = выражение_2;

```

Здесь значение, присваиваемое переменной, определяется условием оператора *if*. Оператор ? называется тернарным, поскольку он обрабатывает три операнда. Этот оператор записывается в следующей общей форме:

```

выражение_1 ? выражение_2 : выражение_3;

```

где выражение_1 должно быть логическим, т.е. возвращать тип *boolean*, а выражение_2 и выражение_3, разделяемые двоеточием, могут быть любого типа, за исключением *void*. Но типы второго и третьего выражений непременно должны совпадать.

Значение выражения ? определяется следующим образом. Сначала вычисляется **выражение_1**. Если оно дает логическое значение *true*, то выполняется **выражение_2**, а его значение становится результирующим для всего выражения ?. Если же **выражение_1** дает логическое значение *false*, то выполняется **выражение_3**, а его значение становится результирующим для всего выражения ?.

Рассмотрим пример, в котором сначала вычисляется абсолютное значение переменной *val*, а затем оно присваивается переменной *absVal*.

```

absVal = val < 0 ? -val : val; //получить абсолютное значение
переменной val

```

В данном примере переменной *absVal* присваивается значение переменной *val*, если это значение больше или равно нулю. А если значение переменной *val* отрицательное, то переменной *absVal* присваивается значение *val* со знаком "минус", что в итоге дает положительную величину. Код, выполняющий ту же самую функцию, но с помощью логической конструкции *if-else*, будет выглядеть следующим образом:

```
if(val < 0) absVal = -val;  
else absVal = val;
```

Рассмотрим еще один пример применения оператора *?*. В этом примере программы выполняется деление двух чисел, но не допускается деление на нуль.

```
//Пример №23. Предотвращение деления на нуль с помощью оператора?  
class NoZeroDiv {  
    public static void main(String args[]) {  
        int result;  
        for (int i = -5; i < 6; i++) {  
            //Деление на нуль предотвращается  
            result = (i != 0) ? 100/i : 0;  
            if (i != 0) {  
                System.out.println("100/" + i + " равно " + result);  
            }  
        }  
    }  
}
```

Результат работы программы:

```
run:  
100/-5 равно -20  
100/-4 равно -25  
100/-3 равно -33  
100/-2 равно -50  
100/-1 равно -100  
100/1 равно 100  
100/2 равно 50  
100/3 равно 33  
100/4 равно 25  
100/5 равно 20
```

Обратите внимание на следующую строку кода:

```
result = (i != 0) ? 100/i : 0;
```

где переменной *result* присваивается результат деления числа 100 на значение переменной *i*. Но деление выполняется только в том случае, если значение переменной *i* не равно нулю. В противном случае переменной *result* присваивается нулевое значение.

Если такого рода проверки не делать, то возникнет ошибка времени выполнения программы:

```
run:
100/-5 равно -20
100/-4 равно -25
100/-3 равно -33
100/-2 равно -50
100/-1 равно -100
Exception in thread "main" java.lang.ArithmeticException: / by zero
|   at iba_javabase.NoZeroDiv.main(NoZeroDiv.java:20)
|   C:\Users\USER\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 0 seconds)
```

Значение, возвращаемое оператором `?`, не обязательно присваивать переменной. Его можно, например, использовать в качестве параметра при вызове метода. Если же все три выражения оператора `?` имеют тип *boolean*, то сам оператор `?` может быть использован в качестве условия для выполнения цикла или оператора *if*. Ниже приведена немного видоизмененная версия предыдущего примера программы. Ее выполнение дает такой же результат, как и прежде.

```
//Пример №24. Предотвращение деления на нуль с помощью оператора?
class NoZeroDiv2 {
    public static void main(String args[]) {
        for (int i = -5; i < 6; i++) {
            if (i != 0 ? true : false) {
                System.out.println("100/" + i + " равно " + 100/i);
            }
        }
    }
}
```

Обратите внимание на выражение, определяющее условие выполнения оператора *if*. Если значение переменной *i* равно нулю, то оператор `?` возвращает логическое значение *false*, что предотвращает деление на нуль, и результат не отображается. В противном случае осуществляется обычное деление.