Лекция №2
Массивы.

Создание массива

Во многих языках программирования (даже в объектно-ориентированных, как C++) массивы считаются примитивными типами, но в Java они относятся к объектам. Соответственно для создания массива необходимо использовать оператор **new**:

```
int[] intArray;                 // Определение ссылки на массив
intArray = new int[100]; // Создание массива и сохранение ссылки на
                                // него в intArray
```

Оператор [] сообщает компилятору, что имя принадлежит объекту массива, а не обычной переменной.

```
int intArray[] = new int[100]; // Альтернативный синтаксис
```

Создание массива

Поскольку массив является объектом, его имя (intArray в предшествующем коде) содержит *ссылку* на массив. Сам массив хранится где-то в памяти, а intArray содержит только адрес блока данных.

Массивы содержат поле **length**, которое может использоваться для определения размера (количества элементов) массива:

int arrayLength = intArray.length; // Определение размера массива

Обращение к элементам массива,Инициализация

При обращении к элементу массива необходимо указать его индекс в квадратных скобках.

temp = intArray[3]; // Получение содержимого четвертого элемента
// массива

intArray[7] = 66; // Вставка значения 66 в восьмую ячейку

Если в программе явно не указано иное, массив целых чисел автоматически инициализируется 0 при создании.

Допустим, вы создаете массив объектов следующего вида:

autoData[] carArray = new autoData[4000];

Пока элементам массива не будут присвоены явные значения, они содержат специальный объект null.

int[] intArray = { 0, 3, 6, 9, 12, 15, 18, 21, 24, 27 };

Вставка, Поиск, Удаление

Вставка нового элемента в массив выполняется просто — с использованием обычного синтаксиса массива:

Arr[0] = 77;

Переменная searchKey содержит искомое значение. Чтобы найти нужный элемент, мы перебираем содержимое массива, сравнивая текущий элемент с searchKey. Если переменная цикла j достигает последней занятой ячейки, а совпадение так и не найдено, значит, значение отсутствует в массиве.

Удаление начинается с поиска заданного элемента. Для простоты мы (оптимистично) предполагаем, что элемент в массиве присутствует. Обнаружив его, мысдвигаем все элементы с большими значениями индекса на один элемент вниз,чтобы заполнить «дыру», оставшуюся от удаленного элемента, а значение nElems уменьшается на 1.

Задача №1

Создать класс, который будет инкапсулировать массив.
Показать на примере работу класса.

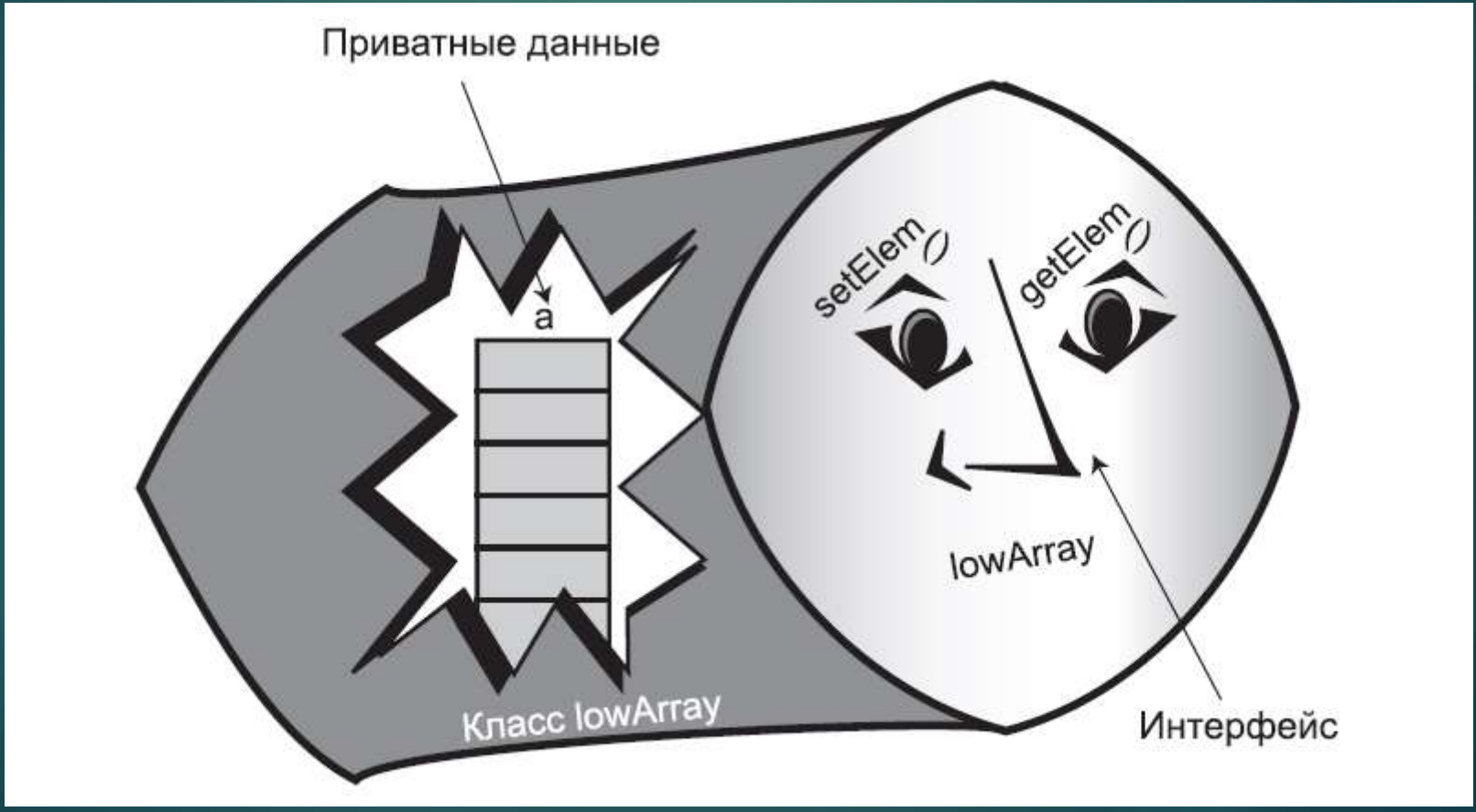
Ответ к задаче №1

```
class LowArray{
        private long[] a; // Ссылка на массив a
        public LowArray(int size) // Конструктор
        { a = new long[size]; } // Создание массива

        public void setElem(int index, long value) // Запись элемента
        { a[index] = value; }
        public long getElem(int index) // Чтение элемента
        { return a[index]; }
}
```

Интерфейсы классов

Если класс будет использоваться разными программистами, его необходимо спроектировать так, чтобы с ним было удобно работать. Способ взаимодействия пользователя с классом называется *интерфейсом* класса.



Задача №2.

Убрать методы setElem() и getElem() из класса LowArray. Реализовать методы insert(int value), find(int key), delete(value).

Ответ к задаче №2

```
private int nElems; // Количество элементов в массиве

public boolean find(long searchKey){ // Поиск заданного значения
    int j;
    for(j=0; j<nElems; j++) // Для каждого элемента
        if(a[j] == searchKey) // Значение найдено?
            break; // Да - выход из цикла
        if(j == nElems) // Достигнут последний элемент?
            return false; // Да
        else
    return true; // Нет
}
```

Ответ к задаче №2

```
public void insert(long value) // Вставка элемента в массив
{
    a[nElems] = value; // Собственно вставка
    nElems++; // Увеличение размера
}
```

Ответ к задаче №2

```
public boolean delete(long value){
        int j;
        for(j=0; j<nElems; j++) // Поиск заданного значения
            if( value == a[j] )
                    break;
            if(j==nElems) // Найти не удалось
                return false;
           else{  // Значение найдено
               for(int k=j; k<nElems; k++) // Сдвиг последующих элементов
                            a[k] = a[k+1];
               nElems--; // Уменьшение размера
              return true;}}
```

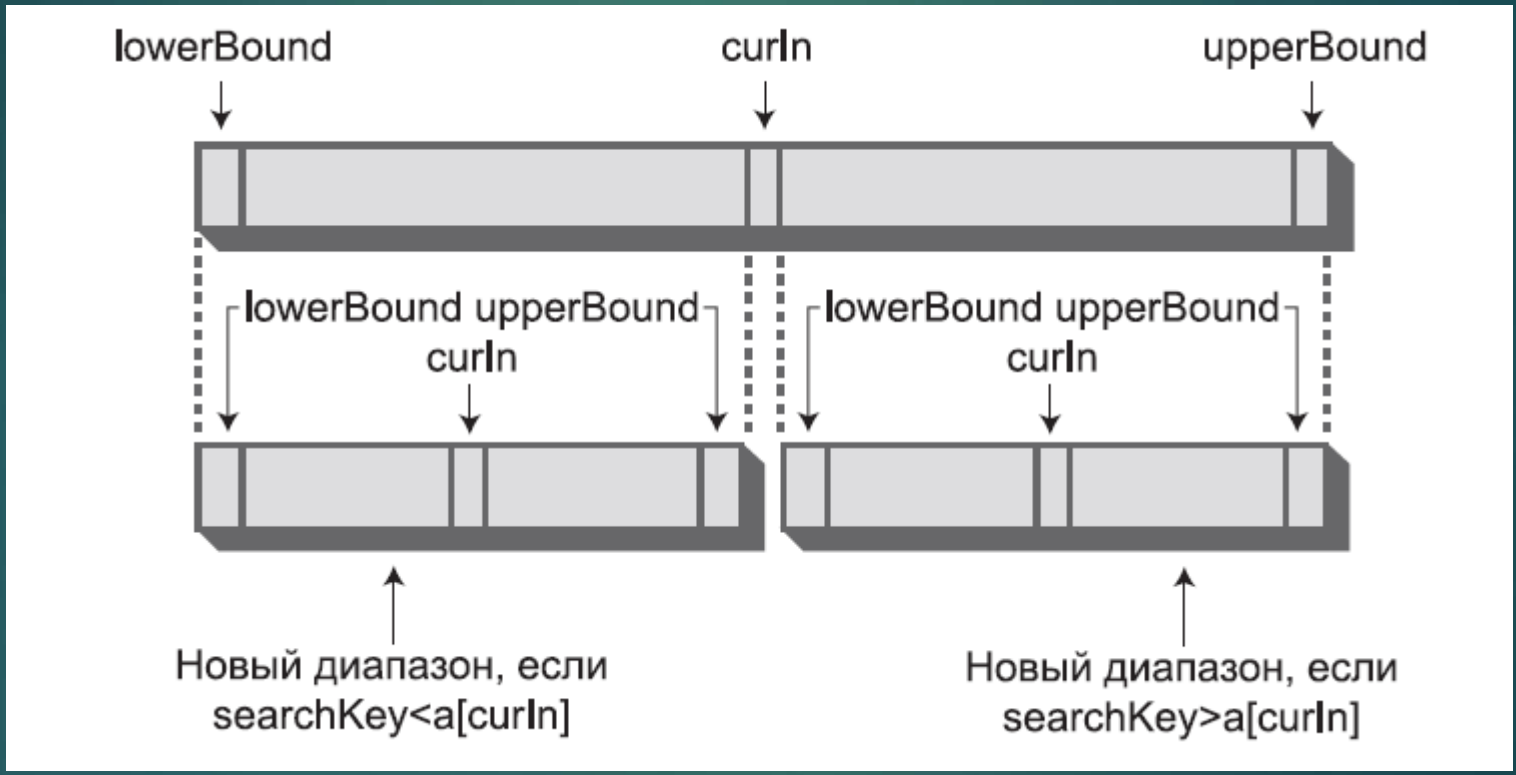
Абстракция

Процесс отделения «как» от «что» (то есть конкретного способа выполнения операции внутри класса от внешнего интерфейса, видимого пользователю класса) называется *абстракцией*. Абстракция принадлежит к числу важных аспектов программотехники. Абстрагирование функциональности класса упрощает проектирование программы, поскольку нам не приходится беспокоиться о подробностях реализации на ранней стадии процесса проектирования.

Двоичный поиск с методом find()

```
public int find(long searchKey) {
        int lowerBound = 0; int upperBound = nElems-1; int curIn;
        while(true) {
            curIn = (lowerBound + upperBound ) / 2;
           if(a[curIn]==searchKey)
               return curIn; // Элемент найден
          else if(lowerBound > upperBound)
              return nElems; // Элемент не найден
          else {  // Деление диапазона
              if(a[curIn] < searchKey)
                  lowerBound = curIn + 1; // В верхней половине
              else
                 upperBound = curIn - 1; // В нижней половине}}}
```

Деление диапазона при двоичном поиске



Задача №3

Реализовать класс инкапсулирующий упорядоченный массив.

Ответ к задаче №3

```
public void insert(long value) // Вставка элемента в массив
{
        int j;
        for(j=0; j<nElems; j++) // Определение позиции вставки
                if(a[j] > value) // (линейный поиск)
                        break;
                for(int k=nElems; k>j; k--) // Перемещение последующих
//элементов
                        a[k] = a[k-1];
        a[j] = value; // Вставка
        nElems++; // Увеличение размера
}
```

Ответ к задаче №3

```
public boolean delete(long value){
        int j = find(value);
       if(j==nElems) // Найти не удалось
            return false;
        else // Элемент найден
       {
           for(int k=j; k<nElems; k++) // Перемещение последующих
//элементов
                  a[k] = a[k+1];
           nElems--; // Уменьшение размера
           return true;
        }
}
```

Количество необходимых сравнений при двоичном поиске

Диапазон	Необходимо сравнений
10	4
100	7
1 000	10
10 000	14
100 000	17
1 000 000	20
10 000 000	24
100 000 000	27
1 000 000 000	30

Количества необходимых шагов при двоичном поиске

Поиск в 10 элементах требует 5 сравнений при линейном поиске (N/2) и максимум 4 сравнения при двоичном поиске. Однако чем больше элементов в массиве, тем заметнее становится разница. Со 100 элементами линейный поиск требует 50 сравнений, а двоичный — только 7. С 1000 элементов разрыв увеличивается до соотношения 500/10, а с 1 000 000 элементов — до соотношения 500 000/20.

Количества необходимых шагов при двоичном поиске

Последовательное удвоение размера диапазона создает числовой ряд, члены которого равны двум в соответствующей степени Если *s* — количество шагов (количество умножений на 2, то есть степень, в которую воз водится 2), а *r* — размер диапазона, то формула выглядит так:

$$r = 2^s$$

Количества необходимых шагов при двоичном поиске

Количество шагов (сравнений) равно логарифму размера диапазона по основанию 2.

$$s = \log_2(r).$$

Обычно логарифм вычисляет по основанию 10, но результат легко преобразуется к основанию 2 — достаточно умножить его на 3,322.

О-синтаксис

В США автомобили делятся в зависимости от размера на несколько категорий: субкомпакты, компакты, среднеразмерные и т. д. Эти категории позволяют получить общее представление о габаритах автомобиля без указания конкретных размеров. Подобная система приблизительных обозначений существует и для описания эффективности компьютерных алгоритмов. В информатике она называется «**О-синтаксисом**».

Критерий сравнения (**сложность алгоритма**) должен связывать скорость алгоритма с количеством элементов.

О-синтаксис

Вставка в неупорядоченный массив: постоянная сложность.

Вставка всегда выполняется с постоянной скоростью независимо от *N* — количества элементов в массиве. Можно сказать, что время вставки элемента в неупорядоченный массив *T* является константой *K*:

$$T = K.$$

O-синтаксис

Линейный поиск: сложность пропорциональна N

$$T = K \times N/2.$$

$$T = K \times N.$$

Из этой формулы следует, что среднее время линейного поиска пропорционально размеру массива. Если увеличить размер массива вдвое, то поиск займет вдвое больше времени.

O-синтаксис

Двоичный поиск: сложность пропорциональна log(N)

Аналогичным образом строится формула, вязывающая T с N для двоичного поиска:

$$T = K \times \log_2(N).$$

$$T = K \times \log(N).$$

Константа не нужна?

«О-синтаксис» почти не отличается от приведенных формул, если не считать того, что в нем не используется константа *K*. При сравнении алгоритмов особенности конкретного микропроцессора или компилятора несущественны; важна только закономерность изменения *T* для разных значений *N*, а не конкретные числа. А это значит, что постоянный коэффициент не нужен.

Константа не нужна?

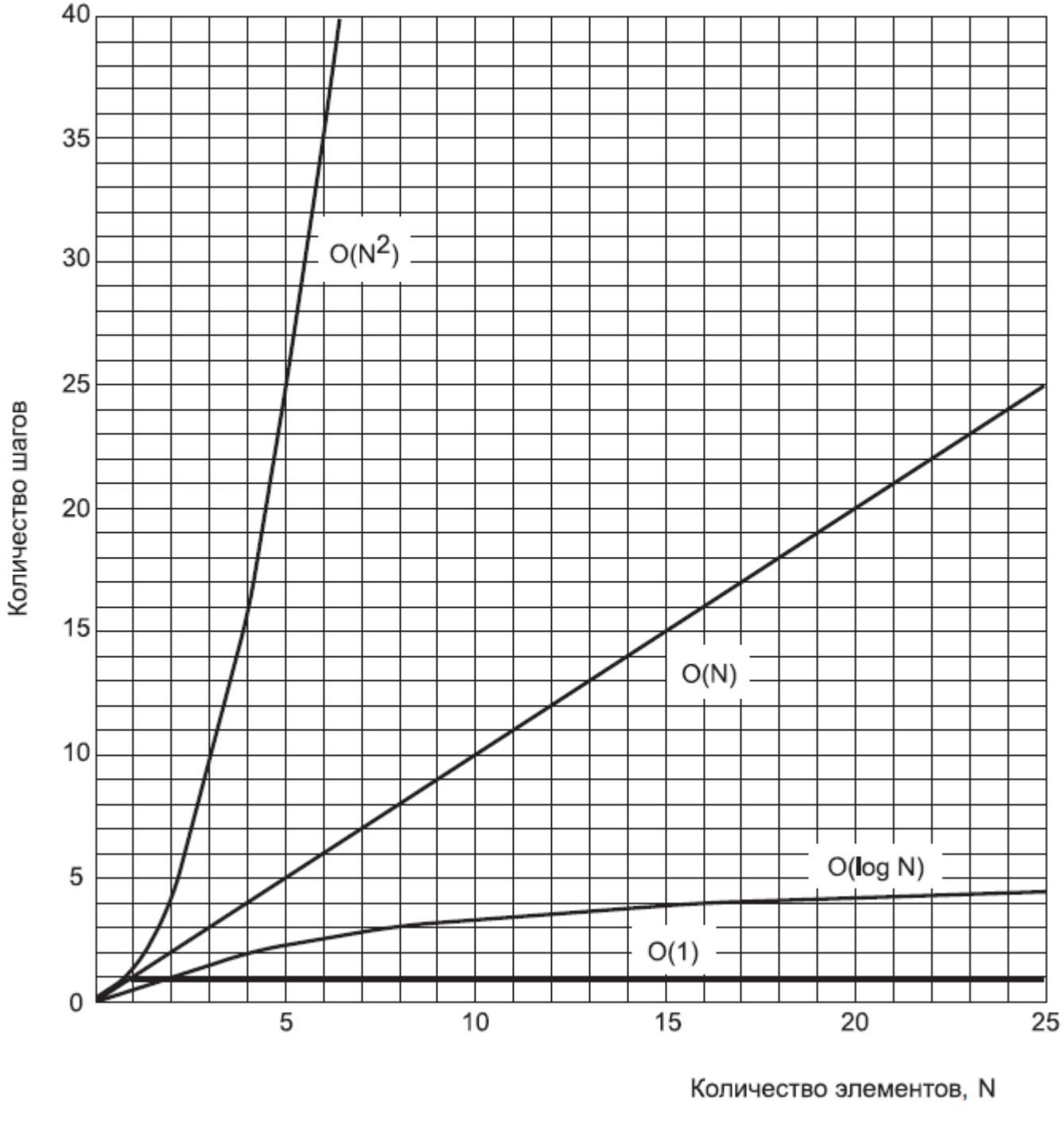
В «О - синтаксисе» используется прописная буква O, которая обозначает порядок «Order of».

В «О-синтаксисе» линейный поиск выполняется за

время $O(N)$, а двоичный поиск — за время $O(\log N)$. Вставка в неупорядоченный массив выполняется за время O(1), то есть за постоянное время (обозначаемое константой 1).

Время выполнения операций в О-синтаксисе

Алгоритм	Время выполнения в О-синтаксисе
Линейный поиск	$O(N)$
Двоичный поиск	$O(\log N)$
Вставка в неупорядоченном массиве	$O(1)$
Вставка в упорядоченном массиве	$O(N)$
Удаление в неупорядоченном массиве	$O(N)$
Удаление в упорядоченном массиве	$O(N)$


40
35
30
25
20
15
10
5
0
O(N2)
O(N)
O(log N)
O(1)
5
10
15
20
25
Количество шагов
Количество элементов, N

Время выполнения операций в O-синтаксисе

На основании этого графика можно (весьма условно) оценить разные сложности: $O(1)$ — отлично, $O(\log N)$ — хорошо, $O(N)$ — неплохо, $O(N2)$ — плохо. Сложность $O(N2)$ встречается при пузырьковой сортировке и в некоторых алгоритмах работы с графами.

Итоги

- Массивы в Java являются объектами и создаются оператором new.
- В неупорядоченных массивах вставка выполняется быстрее, а поиск и удаление — медленнее.
- Инкапсуляция массива в классе защищает массив от случайных изменений.
- Линейный поиск выполняется за время, пропорциональное количеству элементов в массиве.
- Двоичный поиск выполняется за время, пропорциональное логарифму количества элементов.

Спасибо!