

**Лекция 7.**  
**Обертки над примитивными**  
**типами. Boxing и unboxing.**  
**Список на массиве.**  
**Хэш-таблица**

**Обертки над примитивными  
типами.**

**Boxing и unboxing**

# Обертки над примитивными типами

- Как мы помним, примитивные типы в Java не являются объектами
- **Какие есть примитивные типы?**
- `byte, short, int, long, char, boolean, float, double`
- Чтобы избавиться от этого ограничения, в Java есть специальные классы-обертки над примитивными типами
- Они есть для всех примитивных типов
- Объект-обертка хранит в себе значение примитивного типа

# Обертки над примитивными типами

- `Integer x = new Integer(3);` // хранит в себе `int`  
`System.out.println(x);` // 3, реализован `toString`
- `int y = x.intValue();` // получение значения
- Сокращенный синтаксис – **boxing** (упаковка) и **unboxing** (распаковка)
- `Integer x = 3;` // **boxing**, аналогично вызову `Integer.valueOf(3)`  
`int y = x;` // **unboxing**, вызывается `x.intValue()`

# Обертки над примитивными типами

- Сокращенный синтаксис – **boxing** (упаковка) и **unboxing** (распаковка)
- `Integer x = 3;` // boxing, аналогично `Integer.valueOf(3)`  
`int y = x + 4;` // unboxing  
// вызывается `x.intValue()`, затем сложение
- **Коварный момент:**
- `Integer x = null;`  
`int y = x + 4;` // ошибка `NullPointerException` во время  
// исполнения
- Ошибка будет, потому что для ссылки `null` будет вызван метод
- Перед использованием, обертки следует проверять на `null`

# Неизменяемость оберток

- Объекты оберток являются неизменяемыми (как, например, строки)
- То есть, создав объект обертки, поменять его значение уже нельзя
- `Integer x = 3;` // создается объект 3  
`x = 4;` // создается новый объект 4, а объект 3  
// остается неизменным

# Поля и методы классов-обертток

- Получение значения из строки:  
`Integer x = new Integer("3");`  
`Integer y = Integer.valueOf("16", 16);`  
`// второй необязательный параметр – система счисления. То`  
`есть это число 22, а не 16`
- Преобразование типов:  
`double y = x.doubleValue();`  
`byte z = x.byteValue();`  
и т.д.
- Константы для минимального и максимального значения:  
`Integer.MAX_VALUE` и `Integer.MIN_VALUE`

# Все классы-обертки

- `Character` для `char`
- `Boolean` для `boolean`
- `Byte` для `byte`
- `Short` для `short`
- `Integer` для `int`
- `Long` для `long`
- `Double` для `double`
- `Float` для `float`

Обычно имя класса-обертки такое же как у примитивного типа, но с заглавной буквы

Исключение – `Integer` и `Character`



# Сравнение объектов-обертки

- Как и любые объекты, сравнивать обертки нужно только при помощи метода **equals**, а не сравнением через **==**
- Но, как и для строк, тут есть тонкости

- `Integer a1 = 1;`  
`Integer b1 = 1;`  
`a1 == b1    // true`

Java на самом деле хранит в себе все объекты-обертки со значениями от -128 до 127

- `Integer a2 = 127;`  
`Integer b2 = 127;`  
`a2 == b2    // true`

Поэтому они будут ссылаться на одни и те же объекты, и сравнение ссылок даст true

- `Integer a3 = 128;`  
`Integer b3 = 128;`  
`a3 == b3    // false`

Это достаточно существенная оптимизация по скорости и памяти, т.к. маленькие числа нужны часто, и было бы дорого их создавать каждый раз заново

# Использование кэша оберток

- На предыдущем слайде говорилось про кэш объектов оберток от -128 до 127
- Такой кэш есть также для `Byte`, `Short`, `Long`
- Этот кэш используется в 2 случаях:
  - При **boxing**'е:
    - `Integer x = 100;`
  - При использовании метода **valueOf**:
    - `Integer x = Integer.valueOf(100);`
- Кэш не используется при использовании конструктора:
  - `Integer x = new Integer(100);`
- Поэтому этот вариант следует избегать

# Когда использовать обертки?

- **Обертки следует использовать, если нужны объекты**
- Как мы потом увидим, коллекции в Java, например, списки, не умеют работать с примитивными типами. В них придется передавать обертки
- Generic'и в Java работают только с классами
- **Обычно лучше использовать примитивные типы, если это возможно**
- Скорость работы с ними выше, кроме того управление памятью для примитивных типов гораздо более эффективно, локальные переменные примитивных типов уничтожаются сразу при выходе из блока, где они объявлены

# Список на массиве

# Коллекции

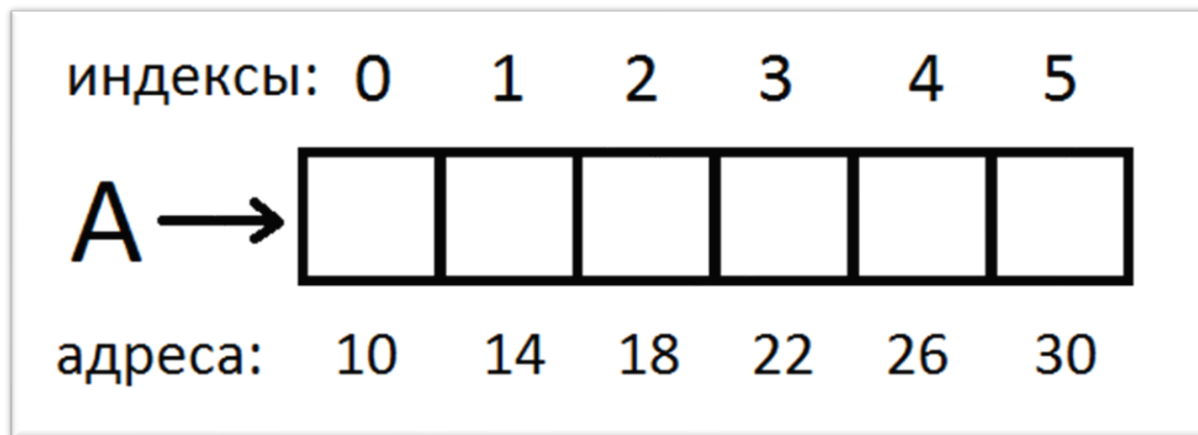
- **Коллекция** – это объект, который содержит в себе набор других объектов
- Например, **массив** является коллекцией
- Кроме массива, есть много других видов коллекций:
  - списки
  - стеки
  - деки
  - очереди
  - множества
  - ассоциативные массивы
  - и др.

# Массивы

- **Массив** – набор элементов одного типа, имеющий фиксированную длину
- Элементы массива хранятся последовательным куском памяти
- Если массив хранит объекты, то сами ссылки хранятся последовательно в памяти, а вот объекты, на которые они ссылаются, могут располагаться в памяти произвольным образом

# Доступ к массиву по индексу

- Массив знает, с какого адреса в памяти он начинается и каков размер типа данных, элементы которого он содержит
- Тогда, зная это, можно легко вычислить адрес элемента по его индексу
- Пусть, например, адрес начала массива  $A$  равен 10, массив хранит `int`, значит размер элемента – 4 байта
- Тогда адрес элемента  $A[5]$  равен  $10 + 5 * 4 = 30$
- Поэтому в массиве очень быстрый доступ по индексу



# Массивы

- **Достоинства массивов:**
  - Высокая скорость доступа к элементу по индексу:  $O(1)$  – константное время
- **Недостатки массивов:**
  - Невозможность менять размер массивов
  - Скорость вставки/удаления элементов для середины и начала –  $O(N)$



# Списки

- **Список (list)** – это структура данных, представляющая собой упорядоченный набор значений, в котором значения могут повторяться
- Подобно массиву, список хранит в себе элементы одного типа в некотором порядке
- В отличие от массива, размер списка можно изменять – можно добавлять или удалять элементы
- Это делает список очень удобным для многих задач

# Виды списков

- В курсе мы рассмотрим:
  - Список на массиве
  - Связные списки:
    - Односвязный
    - Двусвязный

# Виды списков в Java

- Стандартные классы списков в Java:
  - `ArrayList<E>` - список на массиве
  - `LinkedList<E>` - двусвязный список
- Устаревший класс, не рекомендуется к использованию:
  - `Vector<E>` – список на массиве
  - У него хуже производительность

# ArrayList<E> - список на массиве

- `ArrayList<String> names = new ArrayList<String>();`
- Обратим внимание на особый синтаксис при объявлении списка: `ArrayList<String>`
- Здесь в имени класса `ArrayList` участвует другой тип – `String`
- Тем самым мы говорим, что хотим создать список строк, т.е. `ArrayList` объектов `String`
- Такая параметризация классов другими классами называется **generic (джэнэрик)**
- Подробнее рассмотрим **generic**'и в одной из последующих лекций

# ArrayList<E>

- В качестве хранимого типа нельзя указывать примитивные типы, можно использовать только классы
- Поэтому, если нам хочется иметь список, например, `int`'ов, нам нужно объявлять список оберток `Integer`:
- `ArrayList<Integer> numbers = new ArrayList<Integer>();`
- Пример списка списков целых чисел:
- `ArrayList<ArrayList<Integer>> list =  
 new ArrayList<ArrayList<Integer>>();`

# Сокращенный синтаксис

- Если в левой и правой части объявления локальной переменной или поля класса полностью совпадает тип **generic** переменной, то можно использовать сокращенный синтаксис
- Вместо:
- `ArrayList<ArrayList<Integer>> list =  
 new ArrayList<ArrayList<Integer>>();`
- Можно писать:
- `ArrayList<ArrayList<Integer>> list = new ArrayList<>();`

# Добавление элементов в список

- `ArrayList<String> names = new ArrayList<>();`  
`names.add("First");` // добавление в конец  
`names.add(0, "Second");` // добавление по индексу
- `names.addAll(someCollection);`  
/\* добавляет в конец все элементы из переданной коллекции.  
туда можно передавать любые коллекции совместимого типа, т.е. чтобы в коллекции были элементы того же типа, либо дочерних типов \*/

# Удаление элементов из списка

- `ArrayList<String> names = new ArrayList<>();`
- `boolean isDeleted = names.remove("Second");`  
// удаление первого вхождения, совпадение  
// проверяется по equals  
// этот метод возвращает boolean – true, если такой  
// элемент нашелся и был удален
- `names.remove(0);`    // удаление по индексу
- `names.clear();`        // удаление всех элементов
- `names.removeAll(collection);`  
// удаление всех элементов списка, которые есть в  
// указанной коллекции



# Удаление элементов из списка

- `ArrayList<String> names = new ArrayList<>();`
- `names.removeRange(0, 5);`  
// удаление элементов с индексами из диапазона  
// [start, end)  
// то есть, исключая правую границу диапазона
- `names.retainAll(collection);`  
// оставляет в списке только элементы, которые  
// встречаются в переданной коллекции и  
// удаляет остальные элементы

# Инициализация списка

- Для списков нет удобного синтаксиса инициализации подобно массивам с фигурными скобками
- Но есть полезная функция `Arrays.asList` с неопределенным числом параметров
- Функция выдает список из переданных аргументов, а его можно передать в конструктор другого списка
- ```
ArrayList<String> lines  
    = new ArrayList<>(Arrays.asList("line 1", "line 2"));  
// список с элементами "line 1", "line 2"
```

# Получение, задание элементов

- Для списков нет удобного синтаксиса подобно массивам с квадратными скобками
- Чтобы получить и задать элемент списка по индексу, нужно использовать методы **get()** и **set()**:
- `ArrayList<String> lines = new ArrayList<>();`  
`lines.add("Test 1");`

```
String s = lines.get(0);
```

```
// получение элемента с индексом 0
```

```
lines.set(0, "Test 2");
```

```
// заменяет значение элемента по индексу 0
```

# Печать списка

- У класса `ArrayList<T>` нормально определен метод **`toString`**, поэтому можно печатать список
- ```
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(1, 4, 2));  
System.out.println(list);  
// [1, 4, 2]
```

# Поиск элементов в списке

- `ArrayList<String> lines = new ArrayList<>();`  
`lines.add("Test 1");`  
`lines.add("Test 1");`

```
int firstIndex = lines.indexOf("Test 1");    // 0
int lastIndex = lines.lastIndexOf("Test 1"); // 1
int notFound = lines.indexOf("123");         // -1
```

- Как обычно, методы **indexOf()** и **lastIndexOf()** ищут индекс первого и последнего вхождения в список соответственно
- Если элемент не найден, то возвращают -1

# Поиск элементов в списке

- `ArrayList<String> lines = new ArrayList<>();`

`lines.add("Test 1");`

`lines.add("Test 1");`

`boolean hasTest = lines.contains("Test 1");`

`// проверяет, есть ли в списке указанный элемент`

# Цикл foreach. Длина списка

- Для списков и всех других коллекций работает **foreach**:  

```
for (String s : names) {  
    System.out.println(s);  
}
```
- Длину списка можно получить, вызвав метод **size()**:  

```
ArrayList<String> names = new ArrayList<String>();  
System.out.println(names.size());
```
- Есть метод **isEmpty()** для проверки, что коллекция пуста:  

```
boolean isEmpty = names.isEmpty();
```

# Преобразование в массив

- При помощи метода **toArray**(**T[]** array) можно заполнить свой массив значениями из списка
- Правильный вариант использования метода:
- ```
ArrayList<String> names = new ArrayList<String>();  
String[] namesArray = names.toArray(new String[names.size()]);
```
- Замечания:
  - Если размер переданного массива меньше, чем размер списка, то тогда метод создаст новый массив
  - Если размер достаточен, то будет заполнен переданный массив



# Идея реализации списка на массиве

- Пусть для простоты, у нас будет список объектов **Object**
- Создадим класс списка, который содержит внутри себя поле-массив, т.е. набор своих элементов
- Создадим также поле **int** для хранения длины списка
- ```
public class ArrayList {  
    private Object[] items = new Object[10];  
    private int size;    // 0 по умолчанию  
}
```

У массива элементов зададим некоторый начальный размер, например, 10

Важно, что длина массива и длина списка могут отличаться

# Идея реализации списка на массиве

- ```
public class ArrayList {  
    private Object[] items = new Object[10];  
    private int size;    // 0 по умолчанию
```

```
    public int size() {  
        return size;  
    }
```

```
    public boolean isEmpty() {  
        return size == 0;  
    }  
}
```

Размер этого массива называют вместимостью, **capacity**

Массив **items** у нас с некоторым запасом. Если запас истощается, то есть добавляют всё больше элементов, то приходится создавать новый массив большего размера и копировать туда элементы из старого массива

При этом копирование довольно дорогая операция –  $O(N)$

# Идея реализации списка на массиве

- ```
public class ArrayList {  
    private Object[] items = new Object[10];  
    private int size;    // 0 по умолчанию  
  
    public Object get(int index) {  
        //TODO бросить исключение если выход за size  
        return items[index];  
    }  
  
    public void set(int index, Object element) {  
        //TODO бросить исключение если выход за size  
        items[index] = element;  
    }  
}
```

# Идея реализации списка на массиве

- ```
public class ArrayList {  
    public void add(Object element) {  
        if (size >= items.length) {  
            increaseCapacity();  
        }  
  
        items[size] = element;  
        ++size;  
    }  
  
    private void increaseCapacity() {  
        items = Arrays.copyOf(items, items.length * 2);  
    }  
}
```

Если массив не пришлось расширять, то вставка в конце происходит за  $O(1)$ .  
Если пришлось расширять, то за  $O(N)$

# Идея реализации списка на массиве

- ```
public class ArrayList {  
    public void remove(int index) {  
        // TODO выход за границы  
        if (index < size - 1) {  
            System.arraycopy(items, index + 1,  
                items, index, size - index - 1);  
        }  
  
        items[size - 1] = null;  
        --size;  
    }  
}
```

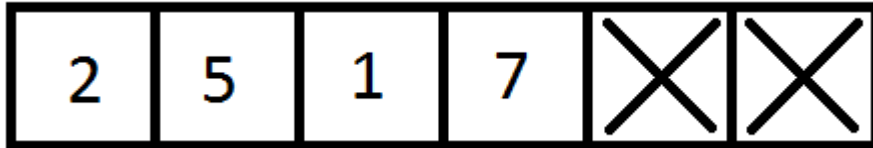
При удалении нужно убрать элемент по индексу и переместить «хвост» на 1 шаг назад

Вставка элемента в середину или начало была бы примерно аналогичной – пришлось бы «раздвигать» массив

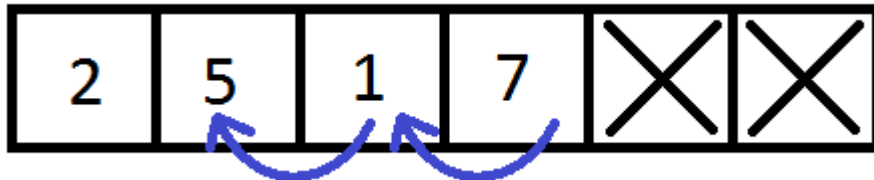
При удалении важно занулить ссылку на освобожденный элемент, чтобы сборщик мусора его собрал

# Удаление элемента из списка на массиве

- Хотим удалить 5 из списка



- Для этого нужно сдвинуть все элементы, которые правее пятёрки, на 1 индекс влево



- Результат



# Временные оценки для ArrayList

- Из реализации следуют временные оценки:
  - Обращение по индексу –  **$O(1)$**
  - Вставка в конец –  **$O(1)$** , если не пришлось пересоздавать массив
  - Удаление с конца –  **$O(1)$**
  - Вставка/удаление в начало/середину –  **$O(N)$**
- Вывод:
  - **ArrayList** имеет высокую производительность при доступе по индексу
  - **ArrayList** позволяет быстро добавлять и удалять элементы в конец
  - **ArrayList** плохо подходит, если нужно часто вставлять/удалять элементы из середины

# Управление вместимостью ArrayList

- Для того, чтобы при работе с `ArrayList` приходилось реже пересоздавать внутренний массив с элементами, в классе `ArrayList` предусмотрены специальные методы и конструкторы
- Есть конструктор, который принимает **capacity** – начальную вместимость списка. Это очень полезно, когда мы примерно знаем, сколько элементов будет в списке
- Метод **ensureCapacity(int)** гарантирует, что вместимость списка будет  $\geq$  указанного числа. Если вместимость и так не меньше, то ничего не будет сделано. Иначе – массив пересоздастся, но будет иметь вместимость не менее указанной
- **trimToSize()** урезает внутренний массив до размера списка – полезно, если в списке было много элементов, но стало мало



# Задача “ArrayListHome”

1. Прочитать в список все строки из файла
2. Есть список из целых чисел. Удалить из него все четные числа. В этой задаче новый список создавать нельзя
3. Есть список из целых чисел, в нём некоторые числа могут повторяться. Надо создать новый список, в котором будут элементы первого списка в таком же порядке, но без повторений

Например, был список [1, 5, 2, 1, 3, 5], должен получиться новый список [1, 5, 2, 3]

# Хэш-таблица

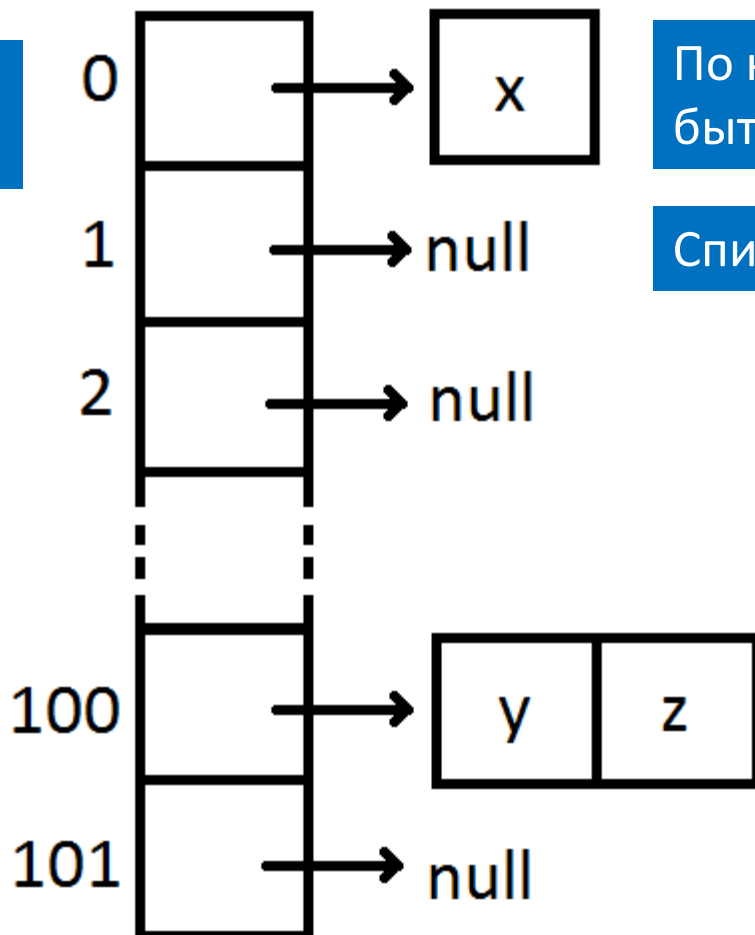
# Хэш-таблица

- Реализовать её можно при помощи массива списков

# Хэш-таблица на массиве списков

- Вертикально нарисован массив. Его элементы – списки

Массив  
списков

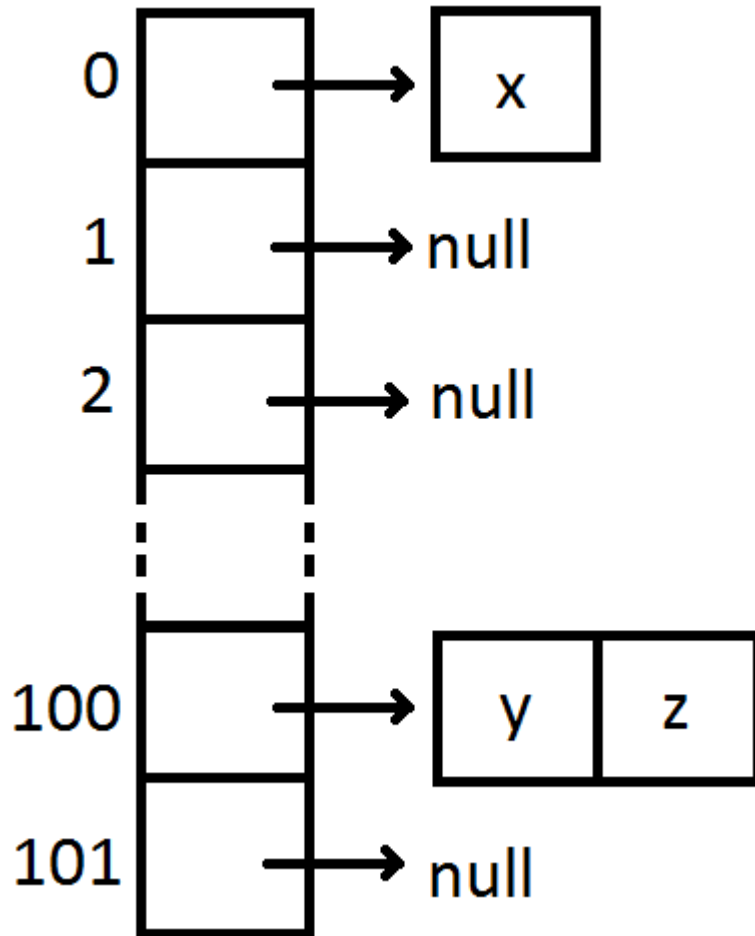


По некоторому индексу может  
быть один элемент

Список может быть null

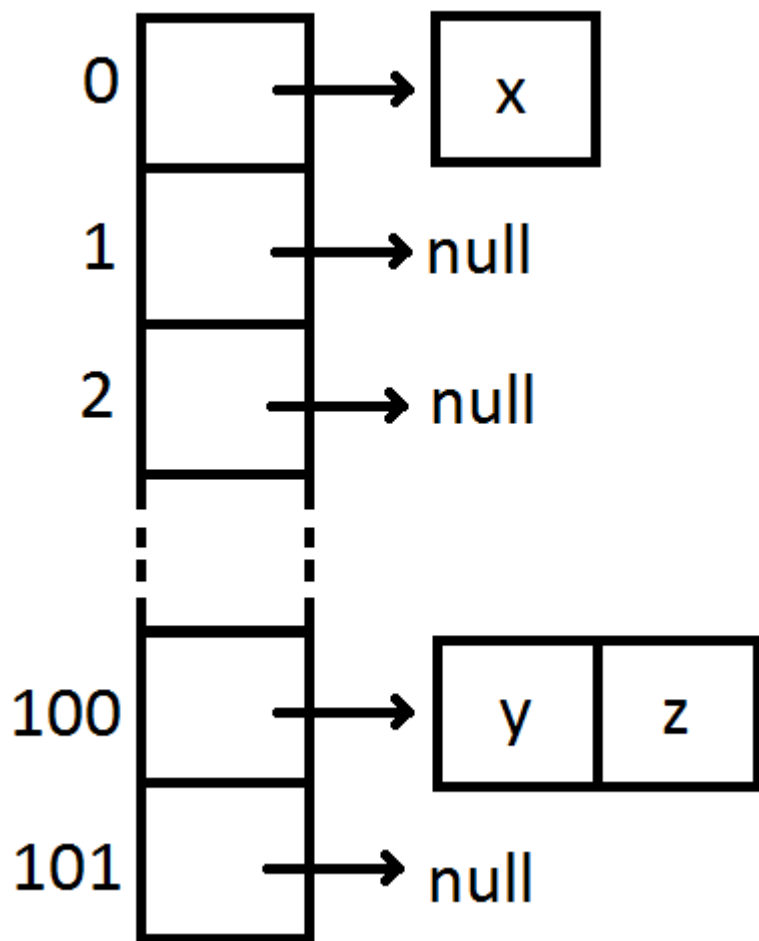
По некоторым индексам –  
много элементов

# Вставка в хэш-таблицу



- Допустим, мы хотим вставить в таблицу объект **x**:
  - Вычисляем его хэш-код, допустим он равен **0**
  - Смотрим что находится в массиве по этому индексу
  - Если там **null**, то создаем список из одного элемента **x**
  - Если там уже есть список, то добавляем элемент в него

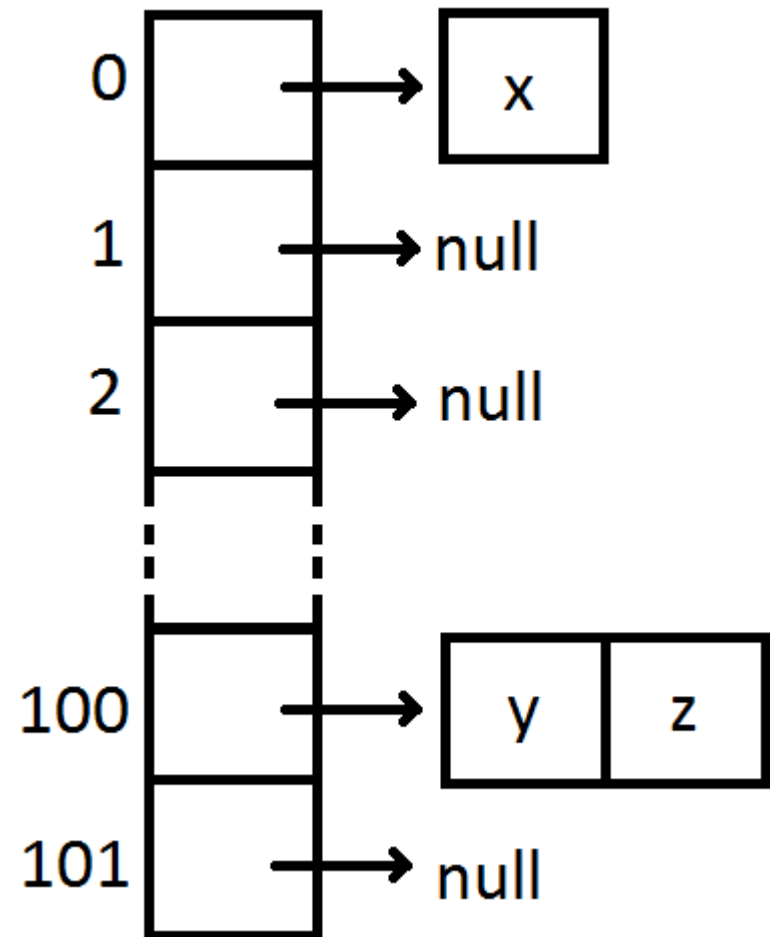
# Поиск в хэш-таблице



- Допустим, мы хотим найти объект **z**:
  - Вычисляем его хэш-код, допустим он равен **100**
  - Смотрим что находится в массиве по этому индексу
  - Если там **null**, то объекта нет
  - Если там уже есть список, то перебираем его элементы, чтобы найти **z**

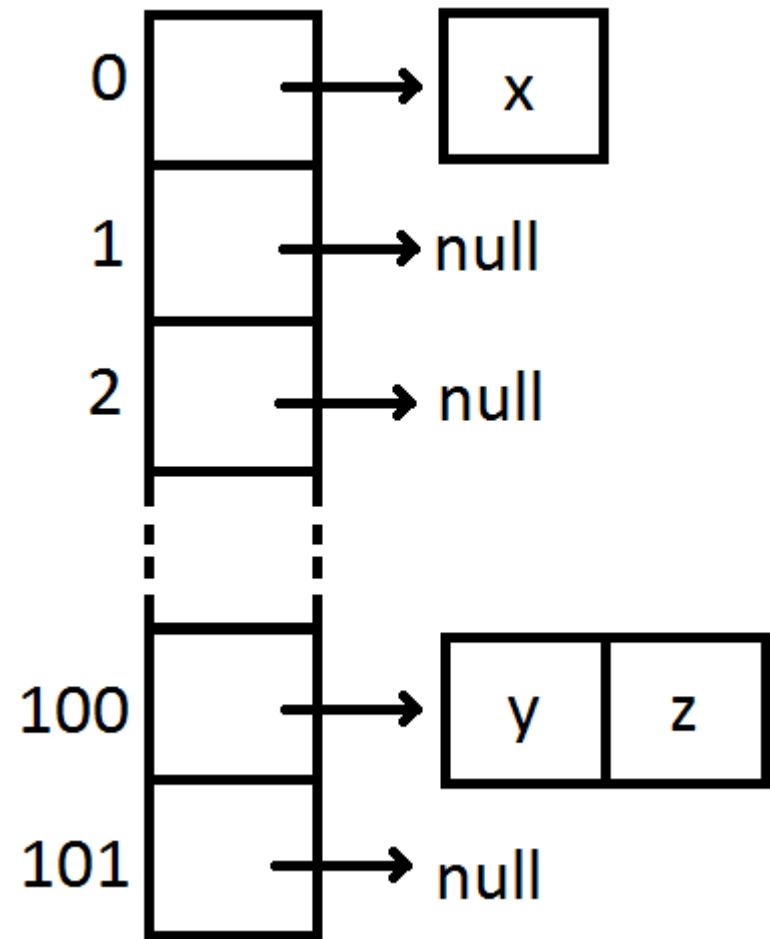
# Производительность хэш-таблицы

- Если предположить что для всех разных объектов хэш-функция дает разные значения, то тогда поиск выполняется за одну операцию – просто вычисляем хэш-функцию и берем единственный элемент из списка, который лежит там. Либо по этому индексу ничего нет
- То есть поиск практически мгновенный,  **$O(1)$**



# Производительность хэш-таблицы

- Но в реальной жизни, разные объекты могут дать одинаковый хэш-код (например, **y** и **z** на рисунке)
- В этом случае нужно дополнительно пройтись по списку, который лежит по индексу равному хэшу-коду
- Нужно стараться делать хэш-функцию такой, чтобы она давала как можно реже совпадений для разных объектов





# Размер массива хэш-таблицы

- Создавать большой массив для хэш-таблицы невыгодно по памяти
- Поэтому обычно размер массива делают небольшим, а если элементов становится много, то размер массива увеличивают и перестраивают таблицу. Тогда некоторые объекты уже могут попасть по разным индексам

# Размер массива хэш-таблицы

- Кроме того, результатом вызова **hashCode()** может быть любой **int** – в том числе и отрицательное число или очень большое положительное. Элемента с таким индексом может не быть в нашем массиве
- Чтобы привести это число к диапазону от 0 до длины массива – 1, удобно пользоваться остатком от деления
- Чтобы число получилось положительным, нужно взять модуль:
- **int** index = **Math**.abs(o.hashCode() % array.length);  
// получится число от 0 до array.length – 1  
// array – это массив списков внутри хэш-таблицы  
// но еще нужно будет учесть null

# Проблема с поиском

- Т.к. результат вызова **hashCode()** зависит от значений полей, то хэш-код изменится после изменения полей объекта
- Поэтому если мы положим объект в хэш-таблицу, а потом поменяем какое-нибудь поле, то есть риск, что мы уже не сможем найти этот объект в хэш-таблице – мы будем обращаться по новому индексу, а объект лежит по старому
- Поэтому не следует менять объекты, которые мы положили в хэш-таблицу, если мы потом планируем их искать

# Задача на курс «HashTable»

- Сделать свой класс хэш-таблицу
- Полное условие см. в файле со списком задач

# Задача на курс «ArrayList»

- Написать свой аналог `ArrayList<E>`
- Список должен реализовывать интерфейс `List<E>`
- Полное условие см. в файле со списком задач