CS106 Data Structure & Java Fundamentals

BitTiger丨来自硅谷的终身学习平台

Claire & Bob

BITTIGER

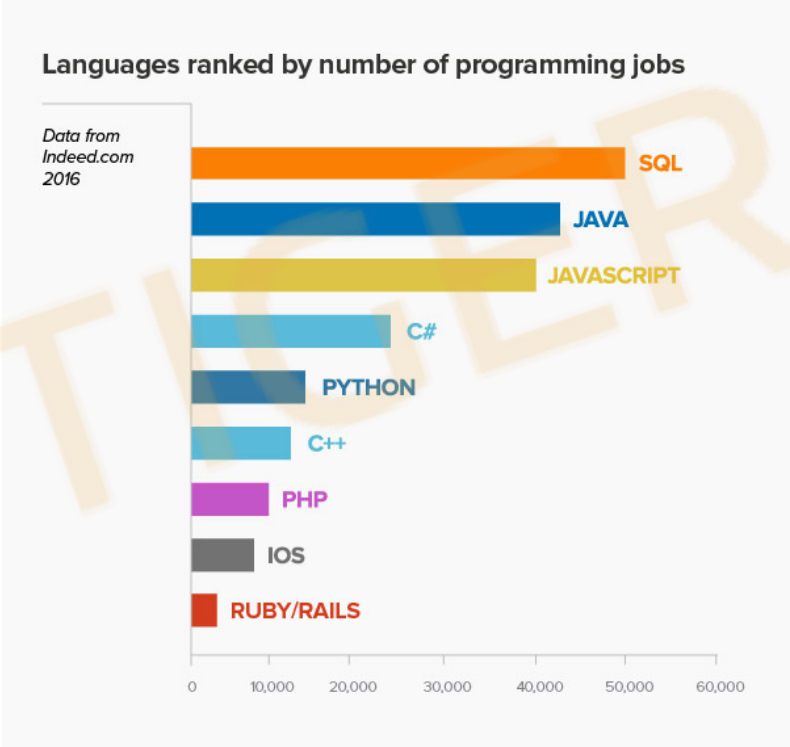
版权声明

Copyright Policy

Why Java?

Java Is ...

- Simple
- Easier to Maintain
- Large Community
- More Job Opportunities



Why should we take this course?

What we can help

- Java Fundamentals
- Data Structures in Java
- Algorithms
- Java New Features
- Hands-on Experience

Syllabus & Timeline

4.1	4.7	4.8	4.14	4.15
Introduction	**Java Fundamentals** - Class - Static - Final - Multithreading - Generic	**Data Structure** - Array - LinkedList - Tree - Stack/Heap	**Algorithm** - Binary Search - Sorting - BFS & DFS - DP	**Project & Java 8 & Java 9** - Project - Lambda - Functional Interface - Stream API

Java Fundamentals

- Basic Syntax
- Modifier Types
- Java Object Oriented Programming
- Multithreading Basics
- Generics

Java Fundamentals - Today's

- Basic Syntax
- Modifier Types

Lecture...

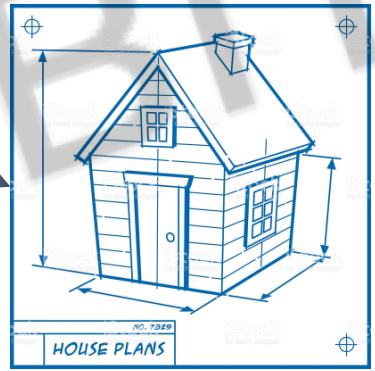
Basic Syntax

- Object
- Class
- Methods

Object House2

Object House1

Class House



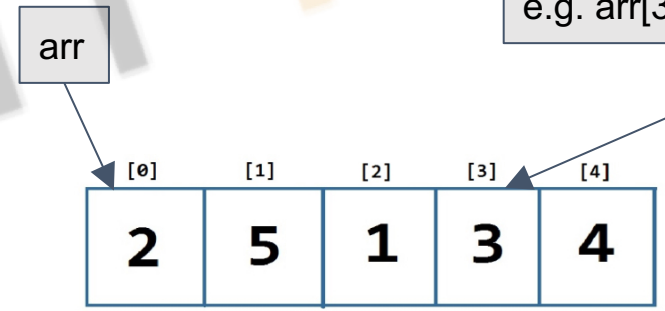
Variable Datatypes

- Variables are reserved memory locations to store values
- Primitive type
  - Predefined by Java: int, double, boolean, byte, ...
- Reference/Object type
  - Class Objects, array variables

arr[i] -> address(arr) + size(int) * i

e.g. arr[3]

int[] arr = new int[5];



Pass-by-reference/value

- **Passed by reference**: the caller and the callee **use the same variable** for the parameter. If the callee modifies the parameter variable, the effect is visible to the caller's variable.
- **Passed by value**, the caller and callee have **two independent variables** with the same value.

Is Java passing by reference or by value?

Pass-by-value

- Java **always** passes everything by value
- Cannot modify value of any parameter passed

Modifier Types

```
public class Car {
    private static final int NUM_OF_WHEELS = 4;
    private String owner;
    private String model;
    private int age;

    public Car(String owner, String model) {
        this.owner = owner;
        this.model = model;
        age = 0;
    }

    protected void start() {
        ...
    }
}
```


Access-control

- **private**: this Class.
- **package-private (default)**: Classes in same package
- **protected**: Subclasses + Classes in same package
- **public**: All

Modifier

Note: As inaccessible as possible.

Getters and Setters

```
public class Car {
    private String owner;
    private String model;
    private int age;

    public void setOwner(String owner) {
        this.owner = owner;
    }

    public String getOwner() {
        return owner;
    }

    …
}
```

Non-access Modifier

- **static**: create class method and variables
- **final**: finalize the implementation
- **abstract**: create abstract class and methods
- **synchronized**: prevent threads interference

Keyword - static

- Static variable
- Static method
- Static class
- Static initializer

Static Variables

- Refer the common property of all objects of a class
- Gets memory only once in class area at the time of class loading

Static Method

```
class Arrays {
        public static void sort(int[] a) {
                …
        }
}

public static void main(Strings[] args) {
        int[] a = {2, 3, 1};
        Arrays.sort(a[]);
}
```

Static Method

- Belongs to class rather than object of the class
- Can be invoked without creating an instance of a class
- Can access and update static member of the class
- Cannot access non-static members

Static Class

- Outer class cannot be static
- Inner class can be static
- Use inner class without creating an instance of outer class

Static Initializer

```
public class Solution {

    public static void main(String[] args) {
        System.out.println("Main Method.");
    }

    static {
        System.out.println("Inside Static Initializer.");
    }
}
```

Keyword - Static

- **Static variable**: isolated from enclosing class instance.
- **Static method**: the same, cannot access non-static members.
- **Static class**: static inner class, isolated from enclosing class instance.
- **Static initializer**:
  - Run only one time before constructor/main method called
  - No return
  - No arguments
  - No this/super support

Keyword - final

- Final class
- Final method
- Final variable

Keyword - final

- **Final class**: class cannot be extended
- **Final method**: method cannot be override
- Final variable

Question

- final variable is immutable?

Keyword - final

- **Final class**: class cannot be extended.
- **Final method**: method cannot be override.
- **Final variable**: variable can only be assigned once. (Reference!)

Data Structures & Algorithms

Why do we need to learn data structures & Algorithms?

- Fundamental blocks in programming
- 90% Interview questions

Data Structures & Algorithms

This course is nothing about…

- Tons of concepts
- Complicated maths

This course is all about…

- Hands-on coding challenges
- Learning by doing
- Questions you'll see in tech interviews

Data Structures

What you'll learn in this course:

- ArrayList
- LinkedList
- HashTable/Map/Set
- Tree
- Stack/Queue
- Heap

Data Structures

What you'll learn in this course:

- ArrayList
- LinkedList
- HashTable/Map/Set
- Tree
- Stack/Queue
- Heap

Data Structures - Array

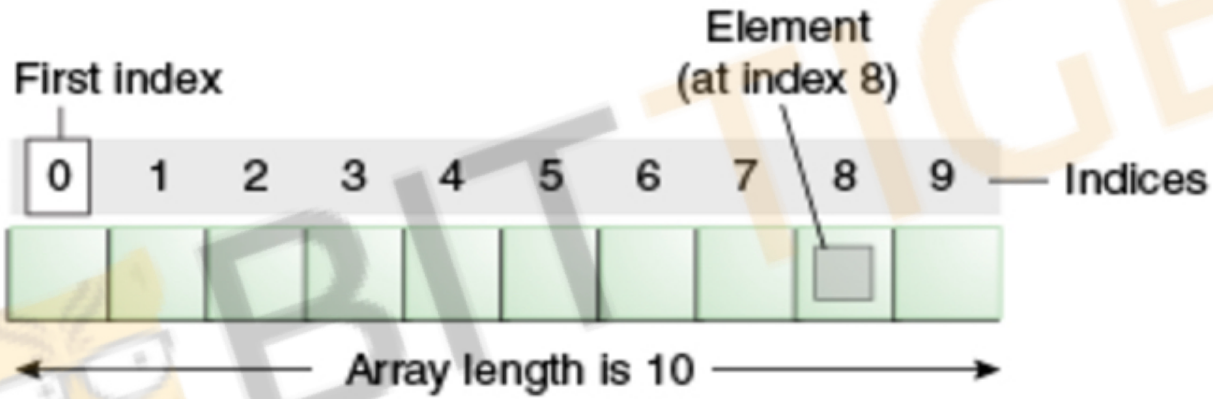


Image: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

ArrayList in Java - method review

- create - List<String> list = new ArrayList<>();
- add - list.add(e) - return: boolean
- remove - list.remove(index) - return: element removed
- get - list.get(index) - return: element
- size - list.size() - return: int

ArrayList - time complexity analysis

size: 0
elementData.length: 0

```
import java.util.ArrayList;

ArrayList<Integer> a = new ArrayList<>();
```

What's the time complexity for "add" and "remove" operations?

ArrayList - add

What's the time complexity of 'add'?

A. O(1)

B. O(n)

C. Amortized O(1)

ArrayList - add

What's the time complexity of 'add'?

A. O(1)

B. O(n)

C. Amortized O(1)

ArrayList - add

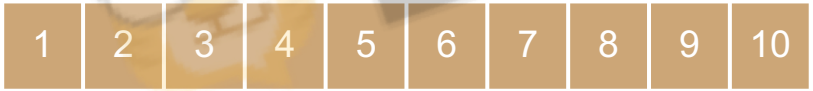
size: 1
elementData.length: 10

```
ArrayList<Integer> a = new ArrayList<>();

a.add(1);
```

ArrayList - add

size: 10
elementData.length: 10

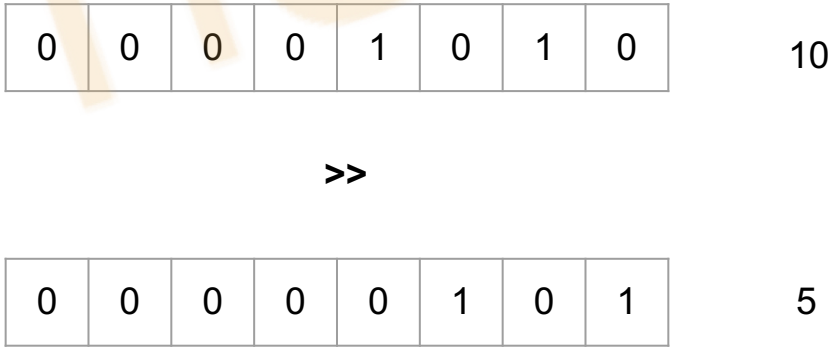
```
ArrayList<Integer> a = new ArrayList();

a.add(1);

for (int i = 2; i <= 10; i++) {
  a.add(i);
}
```



ArrayList - resize

```
int oldCapacity = elementData.length;
int newCapacity = oldCapacity + oldCapacity >> 1;
```

```
oldCapacity = 10;
newCapacity = 10 + 10/2;
```

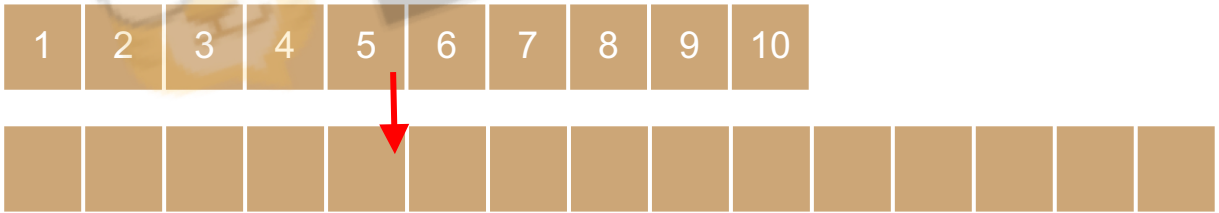


ArrayList - resize

size: 10
elementData.length: 10

```
newCapacity =
10 + 10 >> 1 = 15
```

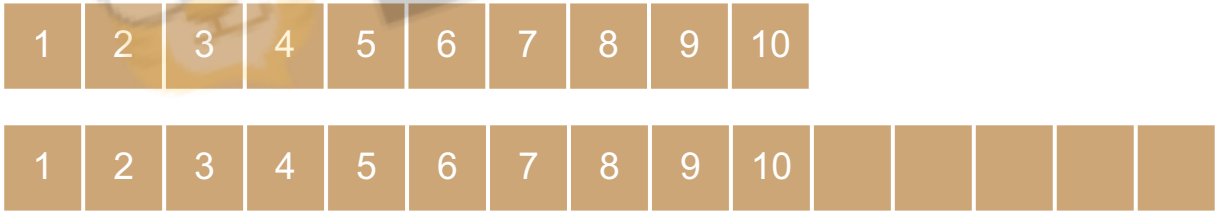
```
ArrayList<Integer> a = new ArrayList();

a.add(1);

for (int i = 2; i <= 10; i++) {
  a.add(i);
}

a.add(11);
```



ArrayList - resize

size: 10
elementData.length: 10

```
ArrayList<Integer> a = new ArrayList();

a.add(1);

for (int i = 2; i <= 10; i++) {
  a.add(i);
}

a.add(11);
```



ArrayList - resize

size: 10
elementData.length: **15**

```
ArrayList<Integer> a = new ArrayList();

a.add(1);

for (int i = 2; i <= 10; i++) {
  a.add(i);
}

a.add(11);
```

ArrayList - resize

size: 11
elementData.length: 15

```
ArrayList<Integer> a = new ArrayList();

a.add(1);

for (int i = 2; i <= 10; i++) {
  a.add(i);
}

a.add(11);
```

Amortized O(1)

ArrayList - remove

What's time complexity of 'remove'?

A. O(1）

B. O(n)

C. Amortized O(1)

ArrayList - remove

What's time complexity of 'remove'?

A. O(1)

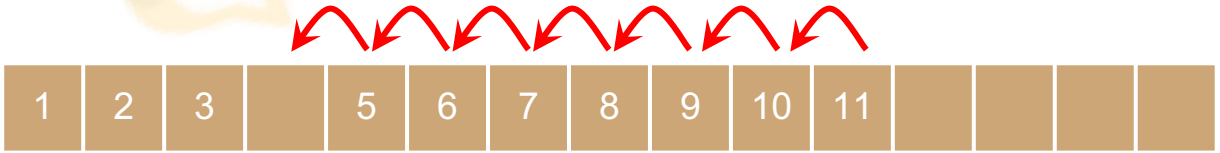
B. O(n)

C. Amortized O(1)

ArrayList - remove

size: 11
elementData.length: 15

```
ArrayList<Integer> a = new ArrayList();

a.add(1);

for (int i = 2; i <= 10; i++) {
  a.add(i);
}

a.add(11);
a.remove(3);
```



ArrayList - remove

size: 11
elementData.length: 15

```
ArrayList<Integer> a = new ArrayList();

a.add(1);

for (int i = 2; i <= 10; i++) {
  a.add(i);
}

a.add(11);
a.remove(3);
```

ArrayList - remove

size: **10**
elementData.length: 15

```
ArrayList<Integer> a = new ArrayList();

a.add(1);

for (int i = 2; i <= 10; i++) {
  a.add(i);
}

a.add(11);
a.remove(3);
```

ArrayList - time complexity analysis

- Resizable Array,
- Add: O(1) **Amortized**
- Remove: O(n)

Learning by Doing

Cracking the code interview - Missing ranges

```
/**
 * Given a sorted integer array where the range of elements are in the inclusive range
 * [lower, upper], return its missing ranges.
 *
 * For example,
 * given [0, 1, 3, 50, 75], lower = 0 and upper = 99,
 * return ["2", "4->49", "51->74", "76->99"]
 */
```

3 steps

1. **Define signatures and return values**
- List<String> findMissingRanges(int[] nums, int lower, int upper) {}
- List<String> rst = new Arraylist<>();
- return rst;

1. **Handle corner case**s
- If (nums == null || nums.length == 0) {}

1. **Business logic**

Missing ranges solution

```java
public class SolutionMissingRanges {
    private static List<String> findMissingRanges(int[] nums, int lower, int upper) {
        // Step1: Define return value
        List<String> rst = new ArrayList<>(); // Don't need to specify type right hand side

        // Step2: Handle corner cases
        if (nums == null || nums.length == 0) {
            addToList(rst, lower, upper);
            return rst;
        }

        // Step3: Fill in business logic
        // First, add the range before lower
        addToList(rst, lower, nums[0]-1);

        // Second, add all the ranges between lower and upper
        int prev = nums[0];
        int i = 1;
        while (i < nums.length) {
            int cur = nums[i];
            if (cur != prev + 1) {
                addToList(rst, prev+1, cur-1);
            }
            prev = cur;
            i++;
        }

        // Third, add the range after upper
        addToList(rst, nums[nums.length-1] + 1, upper);

        // Remember to return result
        return rst;
    }
```

Missing ranges solution

```
49      // Supporting method
50      private static void addToList(List<String> rst, int start, int end) {
51          // Cases 1: if lower = upper = 8, return ["8"]
52          if (start == end) {
53              rst.add(String.valueOf(start)); // Convert integer to string
54          } else if (start < end) {
55              rst.add(start + "->" + end);
56          }
57      }
58
59      public static void main(String[] args) {
60          int[] nums = {0, 1, 3, 50, 75};
61          int lower = 0;
62          int upper = 99;
63          List<String> rst = SolutionMissingRanges.findMissingRanges(nums, lower, upper);
64          System.out.println(rst.toString());
65      }
```

Algorithms

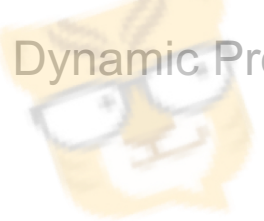
What you'll learn in this course:

- Binary Search
- Sort (quickSort/mergeSort)
- DFS & BFS
- Dynamic Programming

Algorithms

What you'll learn in this course:

- Binary Search
- Sort (quickSort/mergeSort)
- DFS & BFS
- Dynamic Programming



Linear Search vs Binary Search

Search target: 51

99	56	47	8	34	69	55	72	51

Linear Search

Search target: 51

Time complexity: n/2 -> O(n)

99	56	47	8	34	69	55	72	51

Binary Search

Search target: 51

To do a binary search, the array must be **SORTED**!

99	56	47	8	34	69	55	72	51

Binary Search

Search target: 51

Middle index = (start + end) / 2 = (0 + 8)/2 = 4

8	34	47	51	55	56	69	72	99

Binary Search

Search target: 51

Middle index = (start + end) / 2 = (0 + 8)/2 = 4

8	34	47	51	55

Binary Search

Search target: 51

Middle index = (start + end) / 2 = (0 + 4)/2 = 2

8	34	47	51	55

Binary Search

Search target: 51

Middle index = (start + end) / 2 = (0 + 4)/2 = 2

47	51	55

Binary Search

Search target: 51

Middle index = (start + end) / 2 = (0 + 2)/2 = 1

47	51	55

Binary Search

Search target: 51

Time complexity: **O(logN)**

Learning by Doing

Coding Challenge - FirstBadVersion

Learning by Doing

Let’s get to advanced level…

MediumInTwoSortedArray

Search

Linear Search: O(n)

Binary Search: O(logN)