

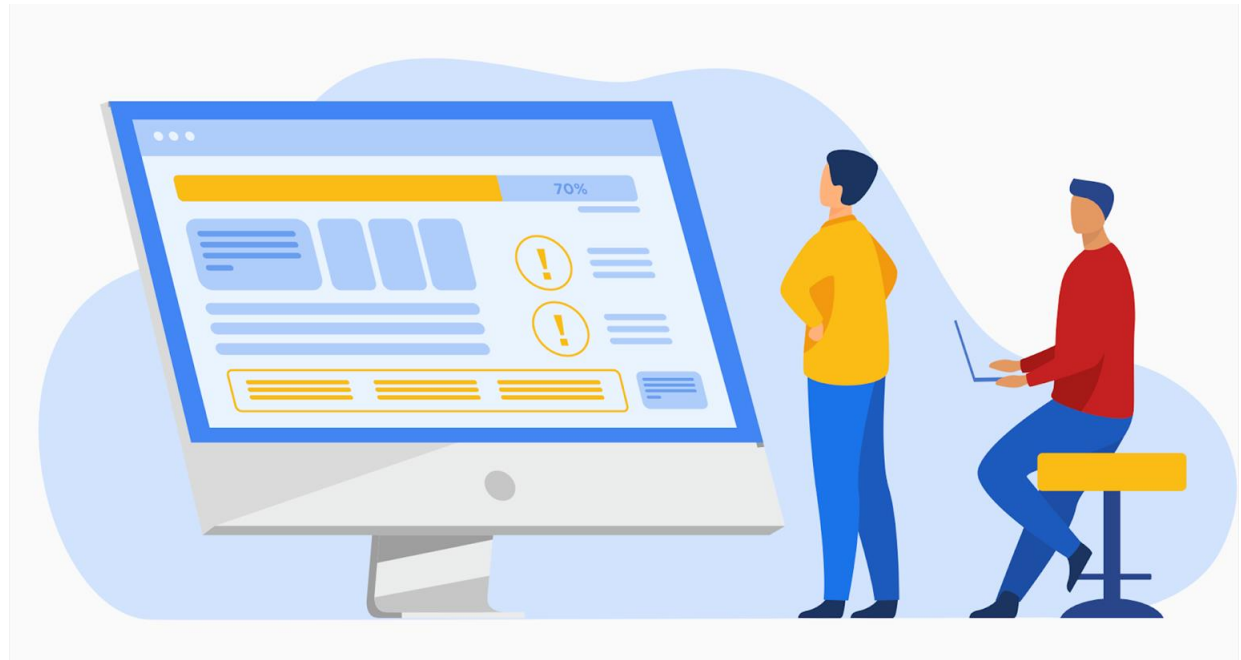
SOFTWARE ENGINEERING CAREER PROGRAM

Algorithms and Data Structures

Developed by

DIGITAL STACK
Knowledge. Performance. Growth.

With a little help from





Java for coding interviews

Content:

1. Motivation
2. Introduction to Java
3. Conditionals and loops in Java
4. Arrays and Strings
5. Methods
6. OOP in Java
7. Data Structures in Java
8. Stream API
9. Exceptions in Java
10. Unit testing - test driven development (TDD)

Motivation

What is the purpose?

1. **Versatility:** Java is a versatile language that can be used to solve a wide range of algorithmic problems. Whether it's sorting, searching, graph algorithms, or dynamic programming, Java provides the necessary tools and libraries to implement efficient solutions.
2. **Performance:** Java's robustness and optimized runtime make it an excellent choice for algorithmic problem-solving. The language's efficiency allows for the implementation of algorithms with fast execution times, making it suitable for handling large datasets or time-sensitive tasks.
3. **Rich Standard Library:** Java comes with a comprehensive standard library that includes data structures (e.g., ArrayList, HashMap), utility classes, and powerful APIs for handling strings, input/output, and mathematical operations. Leveraging these built-in features can greatly simplify algorithm implementation.
4. **Code Reusability:** Java's object-oriented programming paradigm promotes code reusability through the use of classes and interfaces. You can create modular and reusable components for algorithms, making it easier to maintain and update your codebase.
5. **Community Support:** Java has a large and active community of developers who are passionate about problem-solving and algorithms. Online forums, communities, and coding platforms like LeetCode and HackerRank offer a wealth of resources, discussions, and algorithmic challenges to help you improve your skills.

Introduction to Java

Introduction to Java

- 1. Object-Oriented Programming:** Java is an object-oriented programming language, which means it focuses on creating objects that encapsulate data and behavior. This approach promotes code organization, reusability, and modularity, making it easier to build complex software systems.
- 2. Platform Independence:** Java is platform-independent, meaning it can run on any operating system or platform that has a Java Virtual Machine (JVM) installed. This "write once, run anywhere" capability allows developers to write code on one platform and deploy it on multiple platforms without modification.
- 3. Robust and Secure:** Java is designed to be robust and secure, making it a popular choice for building enterprise-level applications. It incorporates features like automatic memory management (garbage collection), exception handling, and strong type-checking, which enhance the stability and reliability of Java applications.

Introduction to Java (2)

Data Types

Terminology

Data type = a set of **values** (definition domain) and a set of **operations** defined on them

8 **primitive** (built-in) data types in Java, mostly different types of **numbers**.

Other types are provided in Java **libraries**

OOP is centered around the idea of creating our own data types out of existing ones (we'll see later)

Introduction to Java (3)

Built-in data types

Terminology

```
int a, b, c;
```

```
a = 1000;
```

```
b = 100;
```

```
c = a + b;
```

The first statement **declares** 3 **variables** with the **identifiers** a, b, and c to be of **type** `int` (integer).

The next 2 **assignment statements** change the values of the variables using the **literals** 1000 and 100.

The last statement assigns c the value of the **expression** `a + b`.

Introduction to Java(4)

Integer numbers

int: whole number in range -2^{31} and $2^{31}-1$ (32 bits)

Used frequently in programs!

short: whole number in range -2^{15} and $2^{15}-1$ (16 bits)

long: whole number in range -2^{63} and $2^{63}-1$ (64 bits)

```
public class IntegerOperations {  
    public static void main (String... args) {  
        int a = Integer.parseInt(args[0]);  
        int b = Integer.parseInt(args[1]);  
        int sum = a + b;  
        int prod = a * b;  
        System.out.println(a + " + " + b + " = " + sum);  
        // implement other operations and print results!  
    }  
}
```

Introduction to Java(5)

Characters and Strings

char: alphanumeric symbol, like the ones on your keyboard

We mostly assign values to char variables (anything else, not very often)

String: sequence of characters; we mostly *concatenate* strings

Note: **String** is not built-in (it's part of the **java.lang** package)

```
String a, b, c; // we declare 3 variables of type String
a = "My name is "; // we assign values to variables a and b
b = "Paul";
c = a + b; // concatenate a and b and assign the expression value to c
```


Introduction to Java(6)

Real Numbers

float: 32 bit floating-point numbers with single precision

double: 64 bit floating-point numbers with double precision.

```
public class DoubleOperations {  
    public static void main(String... args) {  
        double a = Double.parseDouble(args[0]);  
        a = Math.toRadians(a);  
        System.out.println("Math.sin(" + a + ") = " +  
Math.sin(a));  
    }  
}
```

Introduction to Java(7)

Variables

- **Instance Variables (Non-Static Fields)** - objects store their individual states in "non-static fields", that is, fields declared without the `static` keyword.
- **Class Variables (Static Fields)** - a class variable is any field declared with the `static` modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated.
- **Local Variables** - similar to how an object stores its state in fields, a method will often store its temporary state in local variables.
- **Parameters** - Parameters are the variables that are listed as part of a method declaration.

Introduction to Java(8)

Variables – naming convention

Good variable names	Bad variable names
<i>numElements</i>	x
<i>sortedArray</i>	var1
<i>fibonacciSequence</i>	temp
<i>longestCommonSubsequence</i>	a
<i>minValue</i>	foo
<i>currentNode</i>	obj
<i>shortestPathDistance</i>	val

Introduction to Java(9)

Operators


Type	
Arithmetic	$+$, $-$, $/$, $*$, $\%$ $--$, $++$
Relational	$<$, $>$, $>=$, $<=$, $==$
Bitwise	$\&$, $ $, \wedge , \sim , $<<$, $>>$, $>>>$
Logical	$\&\&$, $ $, $!$
Assignment	$=$, $+=$, $-=$, $*=$, $/=$, $\%=$, $>>=$, $<<=$, $ =$, $\&=$, $\wedge=$
Misc	conditional: $(? :)$ instanceof

Introduction to Java(10)


Operator precedence

Operators	Associativity	Type
++ --	right to left	unary postfix
++ -- + - ! (type)	right to left	unary prefix
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&	left to right	boolean logical AND
^	left to right	boolean logical exclusive OR
	left to right	boolean logical inclusive OR
&&	left to right	conditional AND
	left to right	conditional OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

Conditionals (IF)



```
1      boolean isPassedInterview = true;
2
3      if (isPassedInterview) {
4          System.out.println("You passed the interview at Google.");
5      }
```



```
1      String interviewFeedback = "positive";
2
3      if (interviewFeedback.equals("positive")) {
4          System.out.println("The interview feedback at Google was positive");
5      }
```

Conditionals (IF ELSE)



```
1    boolean hasTechnicalSkills = true;
2    int yearsOfExperience = 5;
3
4    if (hasTechnicalSkills && yearsOfExperience >= 3) {
5        System.out.println("You meet the technical requirements for the Google interview.");
6    } else {
7        System.out.println(
8            "You need to further develop your technical skills and gain more experience.");
9    }
```

Conditionals (IF/ELSE IF/ELSE)



```
1      int codingScore = 90;
2      int problemSolvingScore = 80;
3
4      if (codingScore >= 90 && problemSolvingScore >= 90) {
5          System.out.println(
6              "Your coding and problem-solving skills are exceptional for the Google interview.");
7      } else if (codingScore >= 80 && problemSolvingScore >= 80) {
8          System.out.println(
9              "Your coding and problem-solving skills are strong for the Google interview.");
10     } else {
11         System.out.println(
12             "You need to improve your coding and problem-solving skills for the Google interview.");
13     }
```


Switch/Case

```
1 Scanner scanner = new Scanner(System.in);
2     System.out.println(
3         "Enter a step number (1-5) for preparing for coding interviews at Google:");
4     int step = scanner.nextInt();
5     switch (step) {
6         case 1:
7             System.out.println(
8                 "Step 1: Understand the Google interview process and the types of questions asked.");
9             break;
10        case 2:
11            System.out.println("Step 2: Study data structures and algorithms extensively.");
12            break;
13        case 3:
14            System.out.println(
15                "Step 3: Practice solving coding problems, including Google interview-style questions.");
16            break;
17        case 4:
18            System.out.println(
19                "Step 4: Review system design concepts and practice designing scalable solutions.");
20            break;
21        case 5:
22            System.out.println(
23                "Step 5: Prepare for behavioral interviews by researching Google's core values and practicing STAR method responses.");
24            break;
25        default:
26            System.out.println("Invalid step number. Please enter a number between 1 and 5.");
27    }
```

Loops – classic for



```
1      for (int i = 0; i < 5; i++) {  
2          System.out.println("Iteration: " + i);  
3      }
```

Loops – enhanced for loop



```
1      int[] numbers = {1, 2, 3, 4, 5};  
2      for (int num : numbers) {  
3          System.out.println("Number: " + num);  
4      }
```

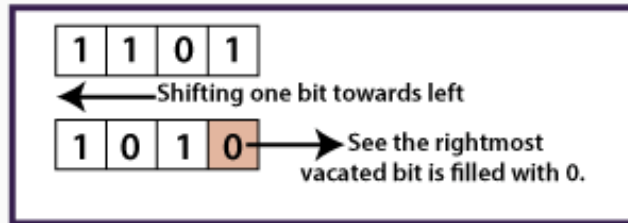
Loops – while



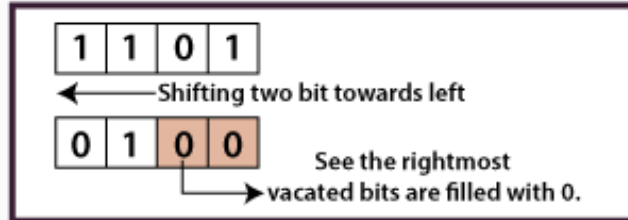
```
1      int count = 0;
2      while (count < 5) {
3          System.out.println("Count: " + count);
4          count++;
5      }
```

Workshop 1

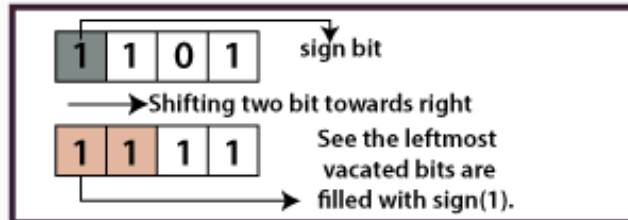
1101 << 1 becomes
1010 Left Shift



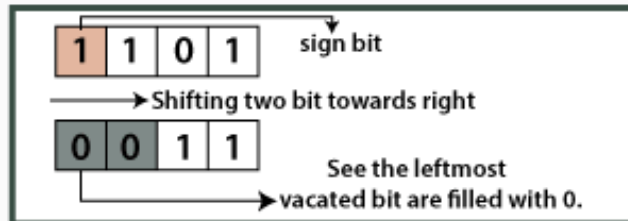
1101 << 2 becomes
0100 Left Shift



1101 >> 2 becomes
1111 Signed Right Shift




1101 >>> 2 becomes
0011 Unsigned Right Shift




<https://leetcode.com/problems/counting-bits/>

Loops – iterator/collection



```
1      List<String> names = new ArrayList<>();
2      names.add("John");
3      names.add("Mary");
4      names.add("Tom");
5
6      Iterator<String> iterator = names.iterator();
7      while (iterator.hasNext()) {
8          String name = iterator.next();
9          System.out.println("Name: " + name);
10     }
```

Loops – streams API



```
1
2 // Iteration with Stream API
3 List<Integer> numbersList = Arrays.asList(1, 2, 3, 4, 5);
4 numbersList.stream().forEach(num -> System.out.println("Number: " + num));
5
6 // Iteration with forEach method in a collection
7 List<String> colors = new ArrayList<>();
8 colors.add("Red");
9 colors.add("Green");
10 colors.add("Blue");
11
12 colors.forEach(color -> System.out.println("Color: " + color));
```

Methods

Java methods

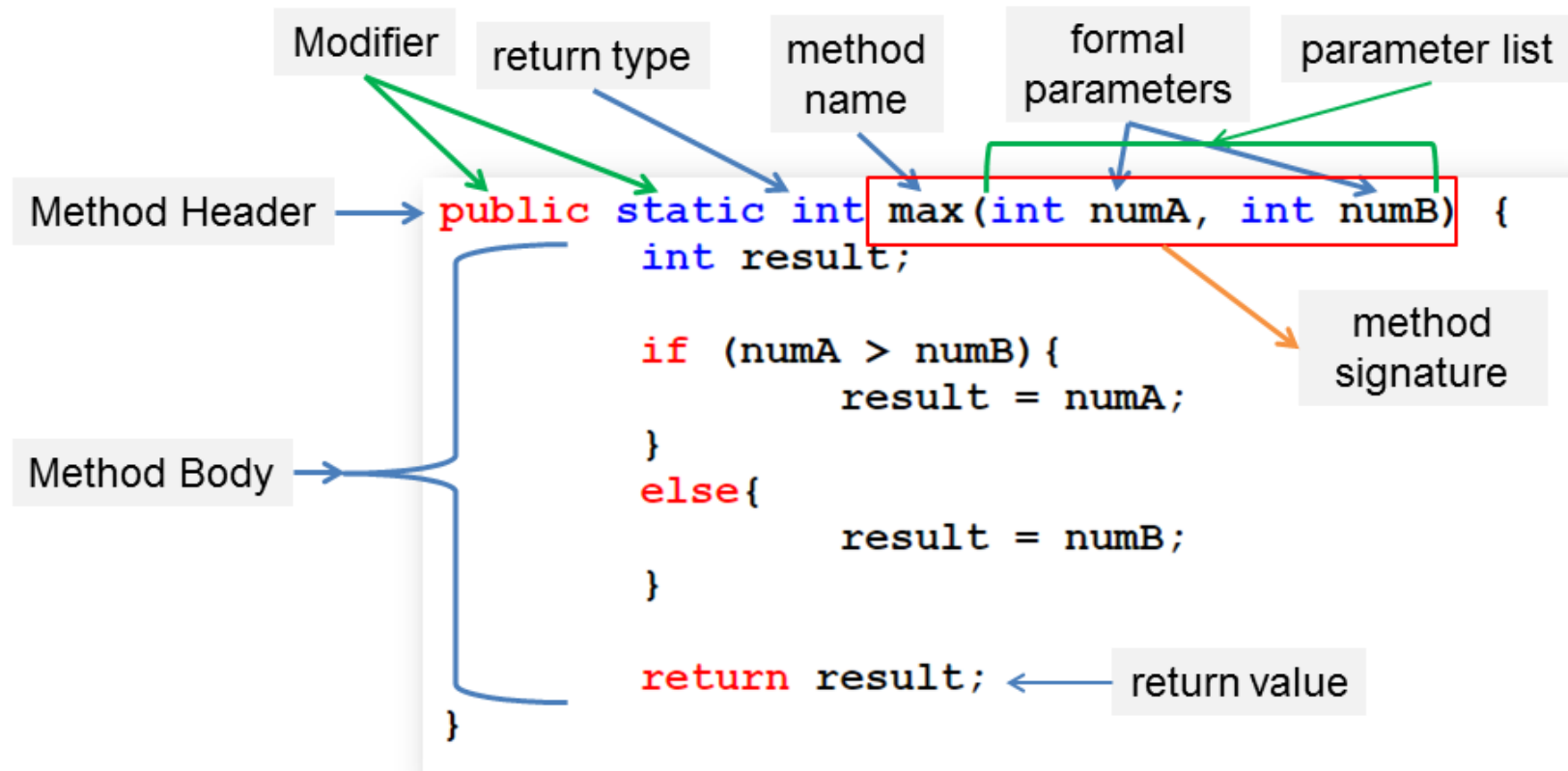


Figure: How to define a method

Methods(2)

Meaningful method names

Problem	Suitable names	Unsuitable names
Binary Search	<i>binarySearch, findElementBinary</i>	<i>binary_search, find</i>
Sorting a Array	<i>quickSort, mergeSort</i>	<i>sort, quick</i>
Fibonacci	<i>fibonacci, fiboSequence</i>	<i>fib, calculatefib</i>
Longest Common Subsequence	<i>longestCommonSubsequence, findLCS,</i>	<i>find_common, lcs</i>
Finding the Minimum Element in an Unsorted Array	<i>findMinimumElement, minElement</i>	<i>find_min, smallest</i>
Binary Tree Traversal	<i>traverseBinaryTree, inOrderTraversal</i>	<i>traverse, inorder</i>
Maximum Subarray Sum Problem	<i>maximumSubarraySum, findMaxSubarraySum</i>	<i>subarray_sum, sum</i>
Shortest Path Algorithm	<i>shortestPath, dijkstraAlgorithm,</i>	<i>shortest, dijkstra</i>

Workshop 2 - FizzBuzz

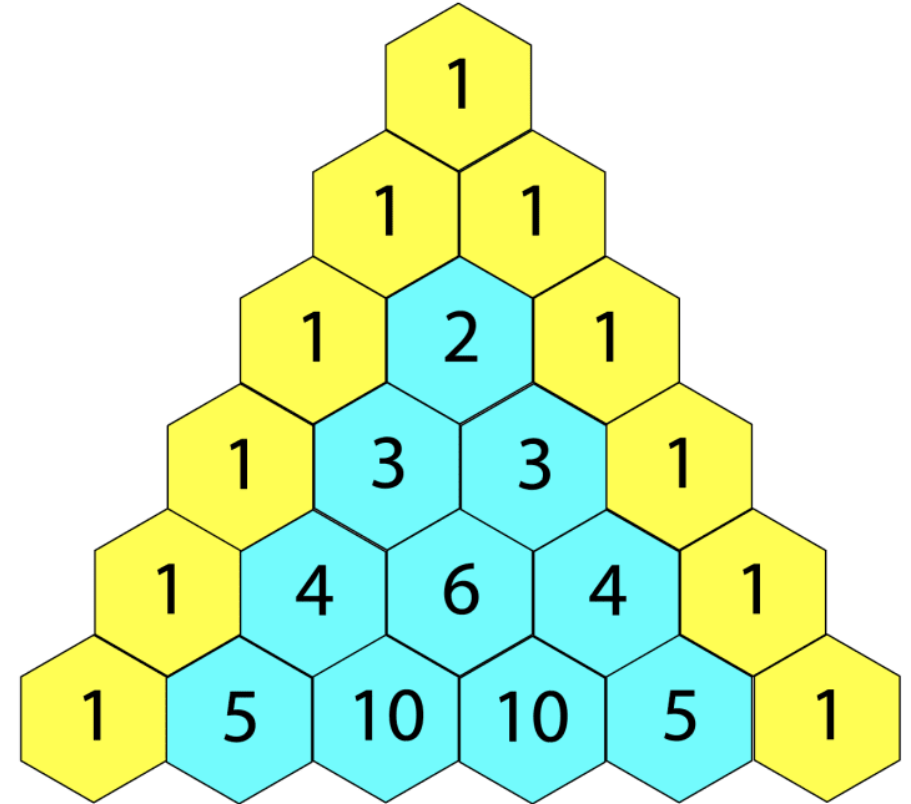
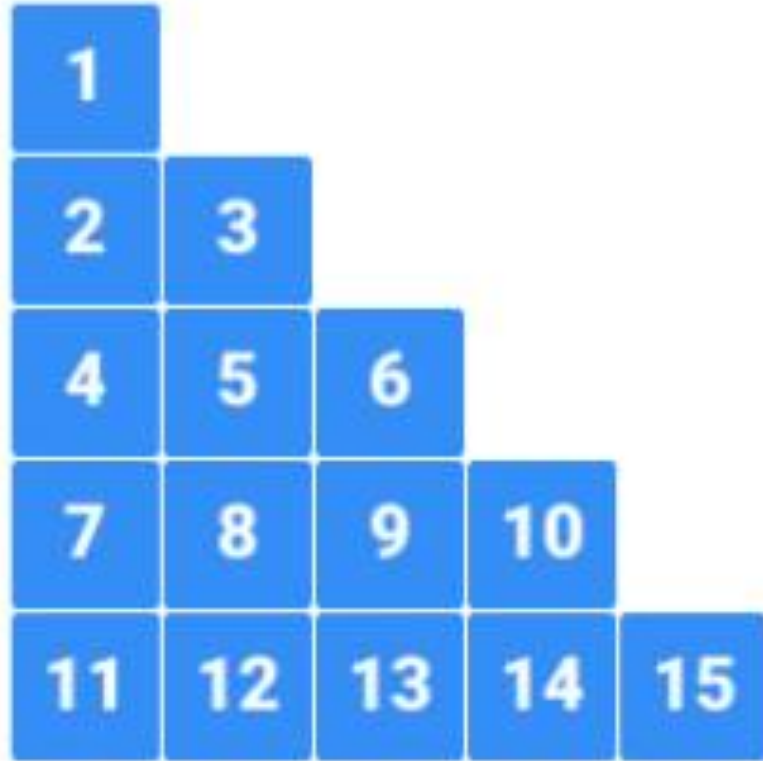
Please solve the **FizzBuzz** problem using the following methods in Java:

- *for-each loop*
- *for loop*
- *while loop*
- *Iterator*
- *Stream API.*

The program should iterate from 1 to 100 and print 'Fizz' for multiples of 3, 'Buzz' for multiples of 5, 'FizzBuzz' for multiples of both 3 and 5, and the number itself for all other cases."

Create a FizzBuzz class and a method for each iteration type. If you need "helper methods", please add them and use meaningful names.

Workshop 3 – Printing patterns



Arrays

1 D ARRAY:

C	O	D	I	N	G	E	E	K
0	1	2	3	4	5	6	7	8

← single row of elements

2 D ARRAY:

		col 0	col 1	col 2
	i \ j	0	1	2
row 0	0	A	A	A
row 1	1	B	B	B
row 2	2	C	C	C

↑ ROWS

← column

} array elements

Arrays

How to create arrays in Java



```
1 // 1. Initialization with static values
2 int[] arr1 = {1, 2, 3, 4, 5};
3 String[] names1 = {"Alice", "Bob", "Charlie"};
4
5 // 2. Initialization using array constructor
6 int[] arr2 = new int[5];
7 String[] names2 = new String[3];
8
9 // 3. Initialization using Arrays.fill() method
10 int[] arr3 = new int[5];
11 Arrays.fill(arr3, 10);
```



```
1 // 4. Initialization using for loop
2 int[] arr4 = new int[5];
3 for (int i = 0; i < arr4.length; i++) {
4     arr4[i] = i + 1;
5 }
6
7 // 5. Initialization using Arrays.copyOf() method
8 int[] arr5 = {1, 2, 3};
9 int[] newArr5 = Arrays.copyOf(arr5, arr5.length);
10
11 // 6. Initialization using Array.newInstance() method
12 int[] arr6 = (int[]) Array.newInstance(int.class, 5);
13 String[] names6 = (String[]) Array.newInstance(String.class, 3);
```

Arrays

Single dimensional array



```
1 String[] steps = {  
2     "1. Understand the problem statement and requirements.",  
3     "2. Break down the problem into smaller subproblems.",  
4     "3. Design an algorithm to solve each subproblem.",  
5     "4. Implement your solutions using the appropriate data structures and programming techniques."  
6 };
```

Multidimensional array



```
1 String[][] interviewResults = {  
2     {"Step 1", "A", "Pass"},  
3     {"Step 2", "B", "Pass"},  
4     {"Step 3", "C", "Fail"},  
5     {"Step 4", "A", "Pass"}  
6 };  
7
```

Strings – Interview problems

1. <https://leetcode.com/problems/reverse-words-in-a-string/>
2. <https://leetcode.com/problems/count-and-say/>
3. <https://leetcode.com/problems/longest-common-prefix/>
4. <https://leetcode.com/problems/add-binary/>
5. <https://leetcode.com/problems/valid-palindrome-ii/>
6. <https://leetcode.com/problems/reverse-string-ii/>
7. <https://leetcode.com/problems/valid-anagram/>
8. <https://leetcode.com/problems/reverse-words-in-a-string-iii/>
9. <https://leetcode.com/problems/goat-latin/>

Strings

- Strings in Java are objects that represent a sequence of characters.
- Strings are immutable, meaning they cannot be changed once created. Any modification to a string results in a new string object.
- Strings in Java are widely used for storing and manipulating text-based data.
- Java provides a rich set of methods for working with strings, allowing you to perform various operations such as concatenation, searching, replacing, and more.
- Here are five commonly used methods for manipulating strings in Java:

length(): Returns the length (number of characters) of the string.

charAt(int index): Returns the character at the specified index in the string.

substring(int beginIndex, int endIndex): Returns a new string that is a substring of the original string, starting from the beginIndex and ending at the endIndex - 1.

toLowerCase(): Converts the string to lowercase.

toUpperCase(): Converts the string to uppercase.

equals(Object obj): Checks if the current string is equal to the specified object. It returns true if the strings are equal, and false otherwise.

contains(CharSequence sequence): Checks if the current string contains the specified sequence of characters. It returns true if the sequence is found, and false otherwise.

replace(CharSequence target, CharSequence replacement): Replaces all occurrences of the target sequence with the replacement sequence in the current string and returns a new string with the replacements.

split(String regex): Splits the current string into an array of substrings based on the specified regular expression. It returns an array of strings.

trim(): Removes leading and trailing whitespace from the current string and returns a new string.

Strings – good for solving algorithms?

StringBuilder or **StringBuffer** are **preferred** in certain **coding interview** scenarios due to their specific features:

Efficient string building: The major difference between **StringBuilder** (or **StringBuffer**) and **String** is that strings built with **StringBuilder are mutable**, while strings built with **String are immutable**. This means that when you modify a string with **StringBuilder, it does not create a new object** with every modification, resulting in **improved efficiency** when constructing large strings or when performing frequent modifications.

Better performance for string concatenation: **StringBuilder** provides **efficient** methods for string **concatenation**, such as **append()**, which allows you to add content to an existing string. This is **more efficient than using the + operator** with strings because the **+** operator creates a new string with each concatenation, which can become costly for multiple concatenations.

Synchronization (only for **StringBuffer**): If you are working with **concurrency** and there is a possibility of **multiple threads** accessing and modifying the same string, **then it is recommended to use the StringBuffer** class because it is synchronized and provides thread-safe operations. If synchronization is not required, **StringBuilder** is preferred as it offers better performance but is not synchronized.

In general, **in coding interviews, using StringBuilder or StringBuffer is recommended when you need to construct or modify strings**, especially when performance and efficiency are important. However, in many other situations, using the **String** class is sufficient and easier to manage.

OOP

Class example

```
1 public class Interview {
2     public String company;
3     public String problem;
4     public String programingLanguage;
5
6     public Interview(String company, String problem, String programingLanguage) {
7         this.company = company;
8         this.problem = problem;
9         this.programingLanguage = programingLanguage;
10    }
11
12    public static void main(String[] args) {
13        Interview interview = new Interview("Google", "fibonacci", "java");
14
15        System.out.println("Company: " + interview.company);
16        System.out.println("Question: " + interview.problem);
17        System.out.println("ProgramingLanguage: " + interview.programingLanguage);
18    }
19 }
```

OOP(1)

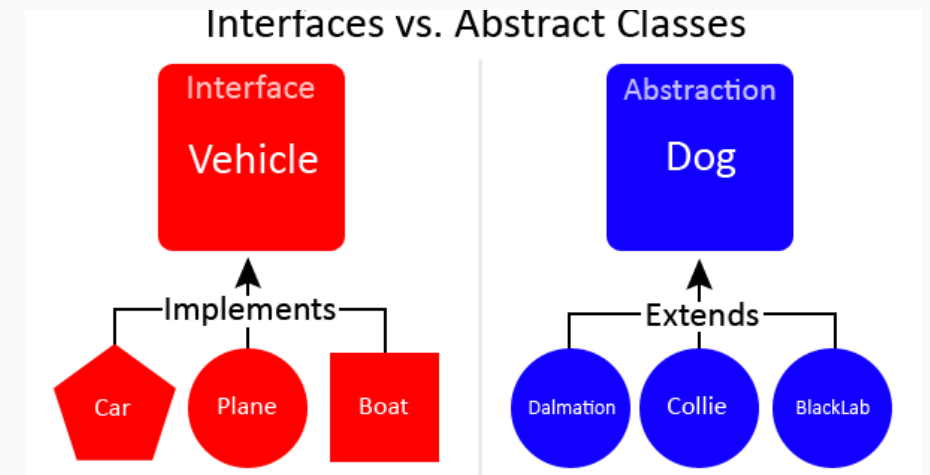
OOP Principles

- **Abstraction:** Simplifying complex systems by representing only essential features and hiding unnecessary details.
- **Encapsulation:** Bundling data and methods that operate on that data into a single unit, and hiding the internal workings of an object.
- **Composition:** Combining objects to create more complex structures or systems by creating objects as member variables of another class.
- **Inheritance:** Mechanism that allows a class to inherit properties and behaviors from a superclass, promoting code reuse and creating "is-a" relationships between classes.
- **Polymorphism:** Treating objects of different classes as objects of a common superclass, enabling objects to be processed in a generic way, irrespective of their specific types.

OOP(2)

Abstraction

Interface	Abstract class
Supports multiple inheritance	Doesn't support multiple inheritance
Doesn't contain data members	Can contain data members
Doesn't contain constructors	Contains constructors
All interfaces are public	Access modifiers for class, fields and methods
Members cannot be static	Can contain static methods



OOP(3)

Encapsulation Java

Access modifiers

```
1 public class RemoteControl {
2     private boolean powerOn;
3     private int volume;
4
5     // Constructor
6     public RemoteControl() {
7         powerOn = false;
8         volume = 0;
9     }
10
11    // Getter and Setter methods
12    public boolean isPowerOn() {
13        return powerOn;
14    }
15
16    public void setPowerOn(boolean powerOn) {
17        this.powerOn = powerOn;
18    }
19
20    public int getVolume() {
21        return volume;
22    }
23
24    public void setVolume(int volume) {
25        this.volume = volume;
26    }
27 }
```

	private	default	protected	public
Same Class	✓	✓	✓	✓
Same package subclass	✗	✓	✓	✓
Same package non-sub class	✗	✓	✓	✓
Different packages subclass	✗	✗	✓	✓
Different package non-sub class	✗	✗	✗	✓

OOP(4)

Composition

```
1 public class Engine {
2     private boolean running;
3
4     public Engine() {
5         running = false;
6     }
7
8     public void start() {
9         if (!running) {
10             System.out.println("Engine started");
11             running = true;
12         } else {
13             System.out.println("Engine is already running");
14         }
15     }
16
17     public void stop() {
18         if (running) {
19             System.out.println("Engine stopped");
20             running = false;
21         } else {
22             System.out.println("Engine is already stopped");
23         }
24     }
25 }
```

```
1 public class Car {
2     private Engine engine;
3     private String model;
4
5     public Car(String model) {
6         this.model = model;
7         engine = new Engine();
8     }
9
10    public void start() {
11        engine.start();
12        System.out.println(model + " started");
13    }
14
15    public void stop() {
16        engine.stop();
17        System.out.println(model + " stopped");
18    }
19
20    public String getModel() {
21        return model;
22    }
23
24    public void setModel(String model) {
25        this.model = model;
26    }
27 }
```



OOP(5)

Inheritance

```
1 class Animal {
2     public void eat() {
3         System.out.println("Animal is eating");
4     }
5
6     public void sleep() {
7         System.out.println("Animal is sleeping");
8     }
9 }
10
11 class Dog extends Animal {
12     public void bark() {
13         System.out.println("Woof! Woof!");
14     }
15 }
16
17 class Cat extends Animal {
18     public void meow() {
19         System.out.println("Meow! Meow!");
20     }
21 }
```

```
1 public class Main {
2     public static void main(String[] args) {
3         Animal animal = new Animal();
4         Dog dog = new Dog();
5         Cat cat = new Cat();
6
7         animal.eat();
8         animal.sleep();
9
10        dog.eat();
11        dog.sleep();
12        dog.bark();
13
14        cat.eat();
15        cat.sleep();
16        cat.meow();
17    }
18 }
```



OOP(6)

Polymorphism – compile time – method overloading

```
1 class Calculator {
2     public int add(int num1, int num2) {
3         return num1 + num2;
4     }
5
6     public double add(double num1, double num2) {
7         return num1 + num2;
8     }
9
10    public int add(int num1, int num2, int num3) {
11        return num1 + num2 + num3;
12    }
13 }
```

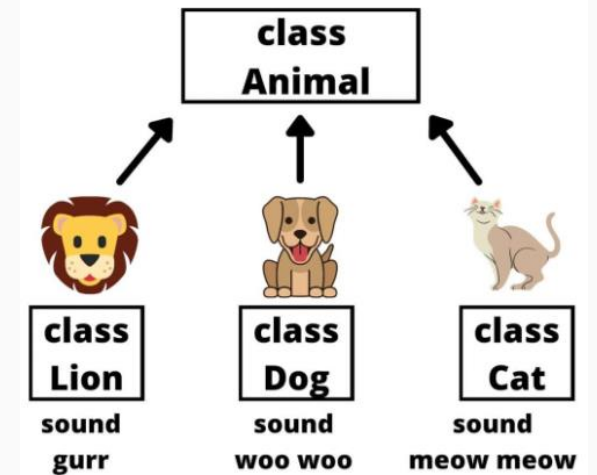
```
1 public class Main {
2     public static void main(String[] args) {
3         Calculator calculator = new Calculator();
4
5         int sum1 = calculator.add(2, 3);
6         double sum2 = calculator.add(2.5, 3.7);
7         int sum3 = calculator.add(2, 3, 4);
8
9         System.out.println("Sum1: " + sum1);
10        System.out.println("Sum2: " + sum2);
11        System.out.println("Sum3: " + sum3);
12    }
13 }
```


OOP(7)

Polymorphism – runtime – method overriding

```
1 class Animal {
2     public void makeSound() {
3         System.out.println(
4             "Animal is making a sound");
5     }
6 }
7 class Dog extends Animal {
8     @Override
9     public void makeSound() {
10        System.out.println("Dog is barking");
11    }
12 }
13
14 class Cat extends Animal {
15     @Override
16     public void makeSound() {
17        System.out.println("Cat is meowing");
18    }
19 }
```

```
1 public class Main {
2     public static void main(String[] args) {
3         Animal animal1 = new Animal();
4         Animal animal2 = new Dog();
5         Animal animal3 = new Cat();
6
7         animal1.makeSound();
8         animal2.makeSound();
9         animal3.makeSound();
10    }
11 }
```



Interfaces

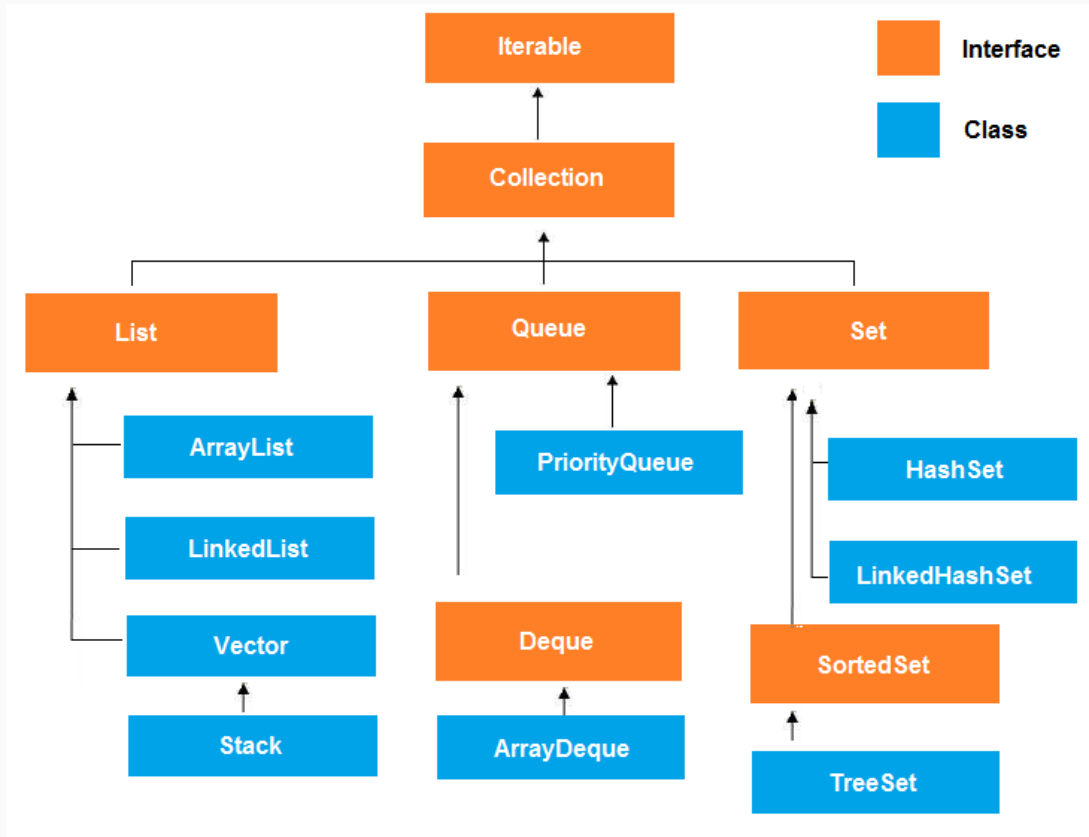
Example

```
1 interface Drawable {
2     void draw();
3 }
4
5 class Rectangle implements Drawable {
6     private double width;
7     private double height;
8
9     public Rectangle(double width, double height) {
10         this.width = width;
11         this.height = height;
12     }
13
14     public void draw() {
15         System.out.println(
16             "Drawing a rectangle with width " + width +
17             " and height " + height);
18     }
19 }
```

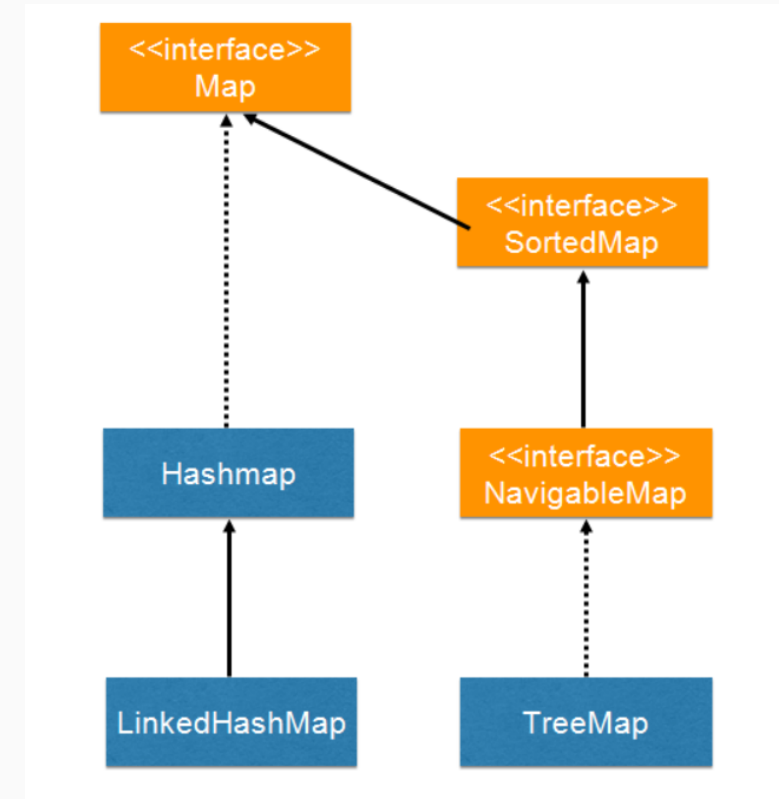
```
1 class Circle implements Drawable {
2     private double radius;
3
4     public Circle(double radius) {
5         this.radius = radius;
6     }
7
8     public void draw() {
9         System.out.println(
10             "Drawing a circle with radius " + radius);
11     }
12 }
13
14 public class Main {
15     public static void main(String[] args) {
16         Drawable circle = new Circle(5.0);
17         circle.draw();
18
19         Drawable rectangle = new Rectangle(3.0, 4.0);
20         rectangle.draw();
21     }
22 }
```

Data Structures in Java

Collection Interface

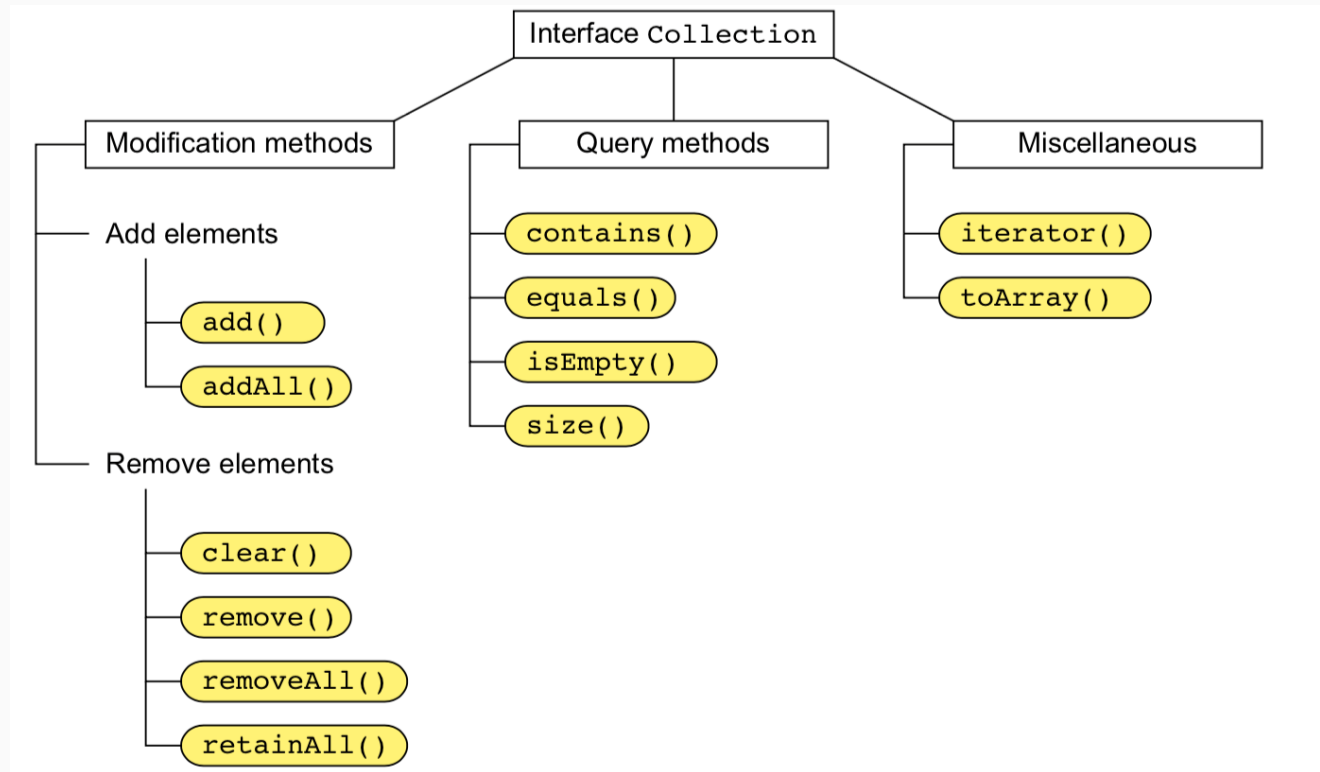


Map Interface



Data Structures in Java(2)

Operations - Collection



Data Structures in Java(3)

Operations - Map

Method	Behavior
put(K key,V value)	Inserts an entry in the map
putAll(Map map)	Inserts a specified map in the map.
putIfAbsent(K key, V value)	Inserts the entry only when the key is not present.
remove(K key)	Deletes the entry for the specified key.
remove(K key,V value)	Deletes the entry for the specified key and value.
clear	Removes all the mappings from the map.
isEmpty	Returns true if the map has no key-value mappings.
size	Returns the number of key-value mappings.
get(K key)	Returns the value associated with the given key.

Interview problems

ArrayList: <https://leetcode.com/problems/find-all-numbers-disappeared-in-an-array/>

LinkedList: <https://leetcode.com/problems/palindrome-linked-list/>

HashSet: <https://leetcode.com/problems/contains-duplicate/>

LinkedHashSet: <https://leetcode.com/problems/first-unique-character-in-a-string/>

TreeSet: <https://leetcode.com/problems/contains-duplicate-iii/>

HashMap: <https://leetcode.com/problems/two-sum/>

LinkedHashMap: <https://leetcode.com/problems/lru-cache/>

PriorityQueue: <https://leetcode.com/problems/kth-largest-element-in-an-array>

ArrayDeque: <https://leetcode.com/problems/sliding-window-maximum/>

Stream API

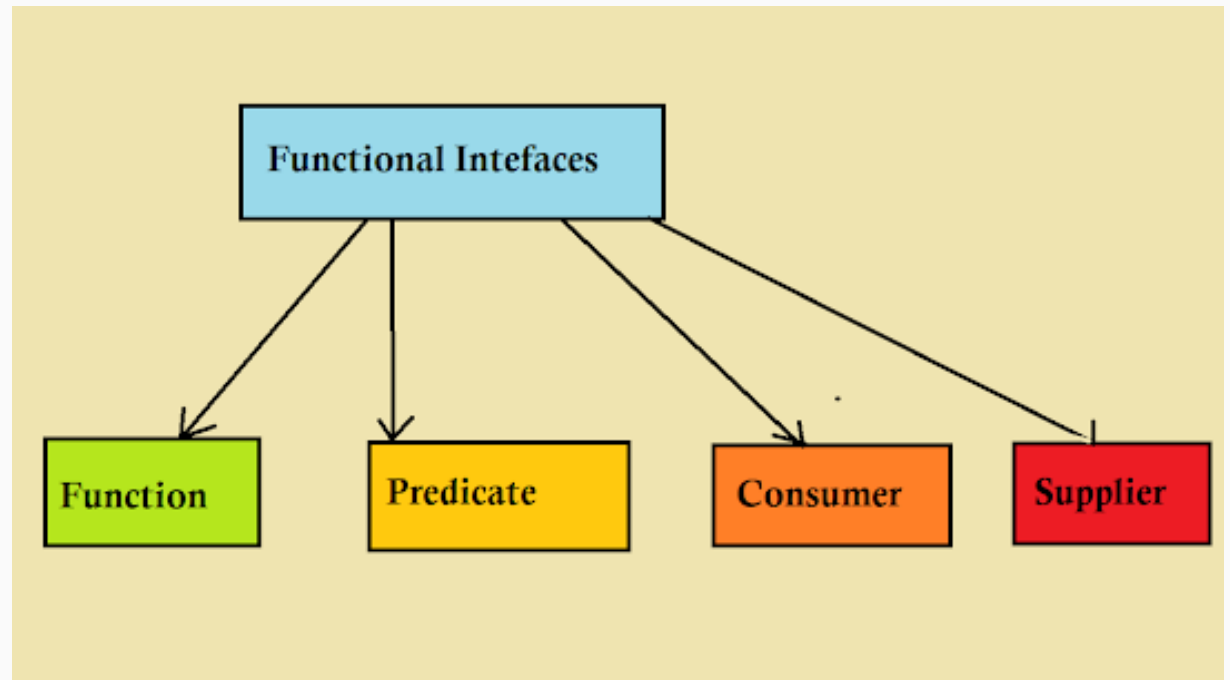
- **A stream** is a sequence of data that can be processed efficiently.
- **A lambda** is an anonymous function used for concise and inline code execution.
- **A functional interface** is an interface with a single abstract method used for lambda expressions.

Lambda Syntax

- No arguments: `() -> System.out.println("Hello")`
- One argument: `s -> System.out.println(s)`
- Two arguments: `(x, y) -> x + y`
- With explicit argument types:

```
(Integer x, Integer y) -> x + y
(x, y) -> {
    System.out.println(x);
    System.out.println(y);
    return (x + y);
}
```
- Multiple statements:

6



Stream API(1)

Functional interfaces

Function<T, R>:

Converts input of type T to output of type R.

Commonly used with **map()** and **flatMap()** methods in Stream API.

Predicate<T>:

Tests a condition on input of type T and returns a boolean.

Commonly used with **filter()** method in Stream API.

Consumer<T>:

Performs an action on input of type T.

Commonly used with **forEach()** method in Stream API.

Supplier<T>:

Supplies or generates values of type T.

Commonly used with **generate()** and **iterate()** methods in Stream API.

Stream API (2)

Function<T,R>

map():

Transforms each element of a stream using a specified function.

Returns a **new stream** with the results of the mapping.

Useful in situations where you want to **convert each element** to a different data type.

flatMap():

Flattens a stream of elements into **a single flat stream, removing nested structure**.

Applies a flattening function to each element, which returns a stream of results.

Useful in cases where you **have nested lists** or streams and want to flatten them into a simple stream.

collect():

Collects the elements of a **stream into a collection or a specified structure**.

Allows specifying an accumulator and a combination operation for aggregating the elements.

Useful in situations where you want to **gather, group, or process the results** of a stream into a final form.

```
1 List<String> words = Arrays.asList("apple", "banana", "orange", "grape");
2
3 List<Integer> wordLengths = words.stream()
4     .map(word -> word.length())
5     .collect(Collectors.toList());
6
7 // Output: Word lengths: [5, 6, 6, 5]
8 System.out.println("Word lengths: " + wordLengths);
```

```
1 List<String> list1 = Arrays.asList("a", "b", "c");
2 List<String> list2 = Arrays.asList("d", "e", "f");
3 List<String> list3 = Arrays.asList("g", "h", "i");
4 List<String>[] arrayOfLists = new List
5     []{list1, list2, list3};
6
7 List<String> flatList = Arrays.stream(arrayOfLists)
8     .flatMap(List::stream)
9     .collect(Collectors.toList());
10
11 // Output: [a, b, c, d, e, f, g, h, i]
12 System.out.println(flatList);
```

Stream API (3)

Predicate<T>

filter():

Filters the elements of a stream based on a **specified condition**. Returns a new stream that contains only **the elements that satisfy the condition**.

Useful **for selecting specific elements** from a stream.

```
1 List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
2 List<Integer> evenNumbers = numbers.stream()
3                                     .filter(num -> num % 2 == 0)
4                                     .collect(Collectors.toList());
```

allMatch():

Checks if **all elements in the stream satisfy a specified condition**. Returns a **boolean** value indicating whether the **condition holds for all elements**.

```
1 List<String> names = Arrays.asList("John", "Jane", "Jim");
2 boolean allNamesStartWithJ = names.stream()
3                                   .allMatch(name -> name.startsWith("J"));
```

noneMatch():

Checks **if none of the elements in the stream satisfy a specified condition**.

Returns a **boolean** value indicating **whether the condition is not met by any element**.

```
1 List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
2 boolean positiveNumbers = numbers.stream()
3                                   .noneMatch(num -> num <= 0);
```

Stream API (4)

Function<T,R>

map():

Transforms each element of a stream using a specified function.

Returns a **new stream** with the results of the mapping.

Useful in situations where you want to **convert each element** to a different data type.

flatMap():

Flattens a stream of elements into **a single flat** stream, **removing nested structure**.

Applies a flattening function to each element, which returns a stream of results.

Useful in cases where you **have nested lists** or streams and want to flatten them into a simple stream.

collect():

Collects the elements of a **stream into a collection or a specified structure**.

Allows specifying an accumulator and a combination operation for aggregating the elements.

Useful in situations where you want to **gather, group, or process the results** of a stream into a final form.

```
1 List<String> words = Arrays.asList("apple", "banana", "orange", "grape");
2
3 List<Integer> wordLengths = words.stream()
4     .map(word -> word.length())
5     .collect(Collectors.toList());
6
7 // Output: Word lengths: [5, 6, 6, 5]
8 System.out.println("Word lengths: " + wordLengths);
```

```
1 List<String> list1 = Arrays.asList("a", "b", "c");
2 List<String> list2 = Arrays.asList("d", "e", "f");
3 List<String> list3 = Arrays.asList("g", "h", "i");
4 List<String>[] arrayOfLists = new List
5     []{list1, list2, list3};
6
7 List<String> flatList = Arrays.stream(arrayOfLists)
8     .flatMap(List::stream)
9     .collect(Collectors.toList());
10
11 // Output: [a, b, c, d, e, f, g, h, i]
12 System.out.println(flatList);
```

Stream API (5)

Consumer<T>

forEach(Consumer<? super T> action):

Applies the specified consumer function to each element of the stream.

```
1 List<String> names = Arrays.asList("John", "Jane", "Alice", "Bob");
2
3 names.stream()
4     .forEach(name -> System.out.println("Hello, " + name + "!"));
5
6 // Output:
7 // Hello, John!
8 // Hello, Jane!
9 // Hello, Alice!
10 // Hello, Bob!
```

Stream API (6)

Supplier<T>

generate(Supplier<T> supplier):

Generates an **infinite stream** using a Supplier<T> to supply the elements.

```
1 Stream<String> randomStrings = Stream.generate(() -> UUID.randomUUID().toString());
2 // Output: An infinite stream of random strings
```

iterate(T seed, UnaryOperator<T> f):

Generates an **infinite stream** by repeatedly applying a function (UnaryOperator<T>) to a seed (a starting element).

```
1 Stream<Integer> powersOfTwo = Stream.iterate(1, n -> n * 2);
2 // Output: An infinite stream of powers of two: 1, 2, 4, 8, 16, ...
```

concat(Stream<? extends T> a, Stream<? extends T> b):
Concatenates two streams (Stream<T>) into a single stream.

```
1 Stream<Integer> stream1 = Stream.of(1, 2, 3);
2 Stream<Integer> stream2 = Stream.of(4, 5, 6);
3 Stream<Integer> combinedStream = Stream.concat(stream1, stream2);
4 // Output: A stream containing the elements 1, 2, 3, 4, 5, 6
```

empty():

Returns an **empty stream**.

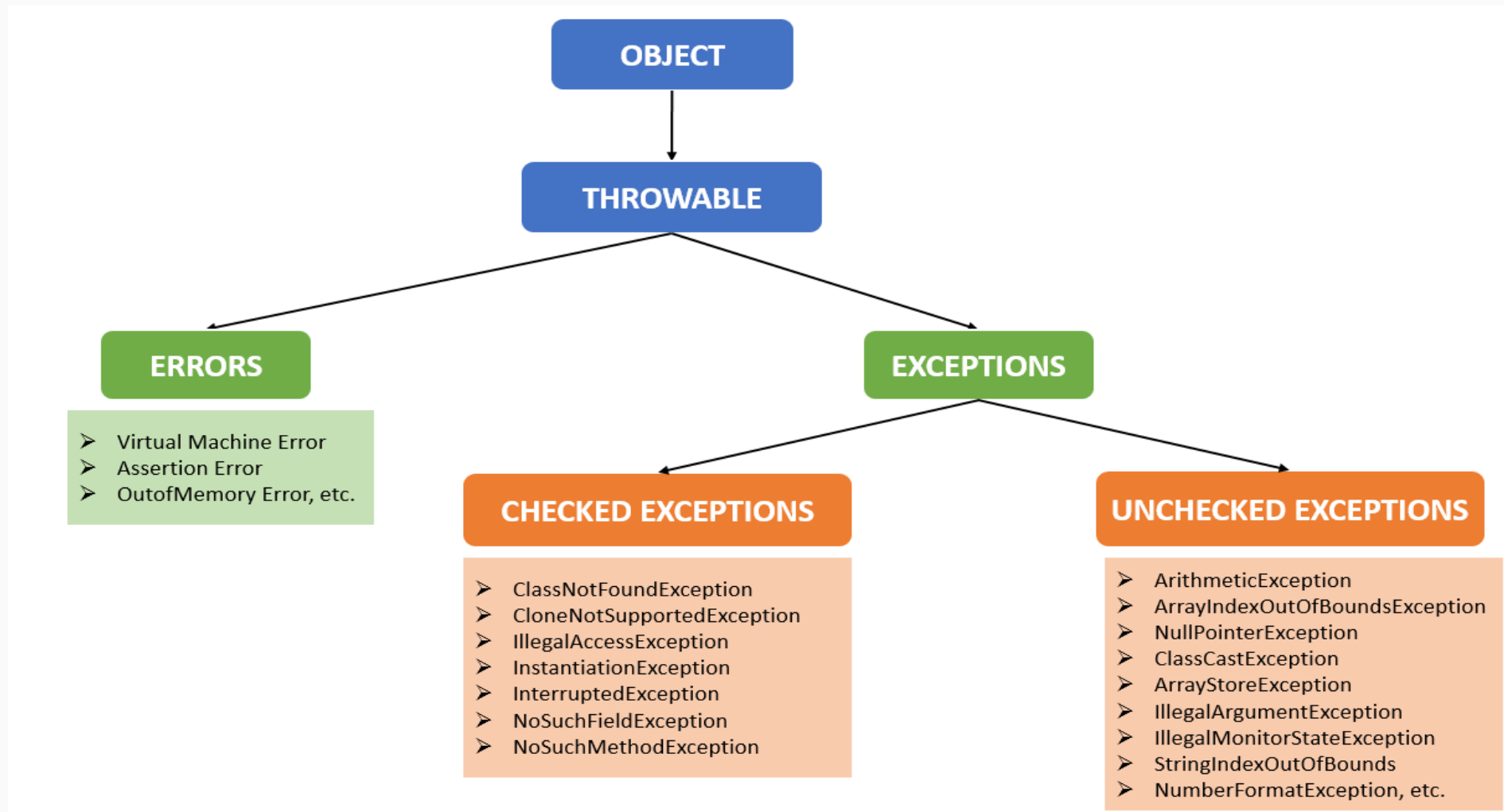
```
1 Stream<Object> emptyStream = Stream.empty();
2 // Output: An empty stream
```

Exceptions

- are used to handle and manage **exceptional conditions or errors** that may occur during the execution of a program.
- provide a way to **separate the normal flow of code from exceptional situations**, making error handling more structured and manageable.
- are represented by **classes in Java**, and they are categorized into two types: **checked exceptions** and **unchecked exceptions**.
- **Checked exceptions must be declared in the method signature** or handled using **try-catch blocks**, forcing the developer to **acknowledge and handle potential exceptions**.
- **Unchecked exceptions do not need to be explicitly declared or caught**, and they usually represent **programming errors** or unforeseen conditions.



Exceptions(2)



Exceptions(3)

Checked exception - example

```
1 public class CheckedExceptionExample {
2     public static void main(String[] args) {
3         File file = new File("myfile.txt");
4         try {
5             Scanner scanner = new Scanner(file);
6             // Perform operations on the file
7             scanner.close();
8         } catch (FileNotFoundException e) {
9             // Handle the checked exception
10            System.out.println("File not found: " + e.getMessage());
11        }
12    }
13 }
```

Unchecked exception - example

```
1 public class UncheckedExceptionExample {
2     public static void main(String[] args) {
3         int dividend = 10;
4         int divisor = 0;
5         try {
6             int result = dividend / divisor; // Division by zero
7             System.out.println("Result: " + result);
8         } catch (ArithmeticException e) {
9             // Handle the unchecked exception
10            System.out.println("Error: " + e.getMessage());
11        }
12    }
13 }
```


Exceptions(4)

Good practices

- **Use specific exception types:** Use specific exception types that **accurately represent** the nature of the error or exceptional condition. This **helps in understanding the cause** of the exception and facilitates targeted error handling.
- **Handle exceptions appropriately:** Handle exceptions at an appropriate level in your application. Catch exceptions where you can take corrective actions or **provide meaningful error messages to the user**. Avoid catching exceptions too early or too late, as it may lead to poor error handling.
- **Provide meaningful error messages:** When catching and handling exceptions, provide clear and meaningful error messages that describe the cause of the exception. This helps users and developers understand what went wrong and how to resolve the issue.
- **Avoid catching generic exceptions:** Avoid catching generic exception types like **Exception** unless absolutely necessary. Catching specific exceptions allows for more precise error handling and avoids masking other potential issues.
- **Log exceptions:** Logging exceptions is essential for debugging and troubleshooting. Use a logging framework (e.g., Java Logging API, Log4j, SLF4J) to log exceptions along with relevant information like timestamps, stack traces, and contextual details.
- **Avoid unnecessary checked exceptions:** Only use checked exceptions when it is necessary for the caller to handle or recover from the exception. Unchecked exceptions are generally more appropriate for unexpected or unrecoverable errors.
- **Separate business logic from exception handling:** Keep your business logic separate from exception handling code. This helps maintain cleaner and more readable code by separating concerns and improving code maintainability.
- **Use custom exceptions when needed:** When the standard exception types are not sufficient to convey the specific nature of an error or exceptional condition, consider creating custom exception classes. Custom exceptions can provide more contextual information and improve the clarity of your code.
- **Test exception scenarios:** Ensure that your code is tested for different exception scenarios, including both expected and unexpected exceptions. Write unit tests that cover exception handling code to ensure it behaves as expected.

Unit Testing(1)

A **unit test** is a **small, focused** test that **verifies the correctness of a specific unit of code**, such as a method or function. The purpose of unit testing is to **ensure that individual units of code work correctly** in isolation and to **catch bugs early in the development process**.

Unit tests help ensure **code quality, maintainability, and reliability** by providing a safety net during code changes or refactoring.

An **assertion** is a **statement** within a unit test that **checks if a certain condition is true**. It is used to **validate the expected behavior** of the code being tested.

In an algorithm interview, the focus on unit tests may involve **testing the algorithm implementation against various input cases** to validate its correctness.

Interviewers **may evaluate if the unit tests cover edge cases**, boundary conditions, and handle invalid inputs effectively. Candidates may be expected to **demonstrate their ability to write clear, concise, and effective unit tests** that validate the algorithm's behavior accurately.

The structure and organization of unit tests, including test **naming conventions and separation of concerns**, may be observed during an algorithm interview.

Interviewers may inquire about the candidate's understanding of **test-driven development (TDD)** principles and how they integrate unit testing into their development workflow.

Candidates may be evaluated on their ability to interpret test results, identify failures, and debug issues related to unit tests.

Unit Testing(2)

TDD – Test Driven Development

- is a software **development approach** that emphasizes **writing tests before** writing the implementation code.
- follows a **cycle of writing failing tests**, implementing the code to **make the tests pass**, and then refactoring the code.
- helps ensure that the **code meets the desired requirements** and **behaves as expected** by defining the **expected behavior through** tests.
- encourages developers to **think about the design and requirements of their code before writing** the implementation.
- promotes **shorter development cycles by catching defects early** and reducing debugging time.
- improves **code quality by enforcing modular** and testable code design.
- facilitates **better collaboration among team members** by providing clear and executable specifications in the form of tests.
- encourages a **mindset of continuous improvement** and testing, leading to a more robust and reliable codebase.

Unit Testing(3)

Junit - Setup

1. Open IntelliJ IDEA and click on "Create New Project" or go to "File" -> "New" -> "Project" to create a new project.
2. Select "Maven" as the project type and click "Next".
3. Choose a name and location for your project and click "Finish".
4. IntelliJ will generate a basic Maven project structure for you.
5. Open the pom.xml file in your project. This file contains the project configuration and dependencies.
6. Inside the <dependencies> tag, add the following dependency for JUnit:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.2</version>
  <scope>test</scope>
</dependency>
```

7. Save the pom.xml file to apply the changes.
8. Right-click on your project folder in the project explorer and select "New" -> "Directory" to create a new directory for your test files (e.g., "src/test/java").
9. Right-click on the newly created test directory and select "New" -> "Java Class" to create a new test class.
10. Write your test methods using the @Test annotation and various assertions provided by JUnit.
11. To run the tests, right-click on the test class or individual test methods and select "Run 'TestClassName'" or "Run 'TestMethodName'".

Unit Testing(4)

Junit - Example

```
package org.example;

public class Calculator {
    1 usage
    public int add(int a, int b) {
        return a + b;
    }

    public int subtract(int a, int b) {
        return a - b;
    }

    public int multiply(int a, int b) {
        return a * b;
    }

    1 usage
    public int divide(int a, int b) {
        if (b == 0) {
            throw new ArithmeticException(
                "Cannot divide by zero");
        }
        return a / b;
    }
}

import org.example.Calculator;
import org.junit.Test;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThrows;

public class CalculatorTest {

    @Test
    public void testAdd() {
        Calculator calculator = new Calculator();
        int result = calculator.add(2, 3);
        assertEquals("expected: 5, result:", result);
    }

    @Test
    public void testDivideByZero() {
        Calculator calculator = new Calculator();
        assertThrows(ArithmeticException.class,
            () -> calculator.divide(10, 0));
    }
}
```

Run: CalculatorTest

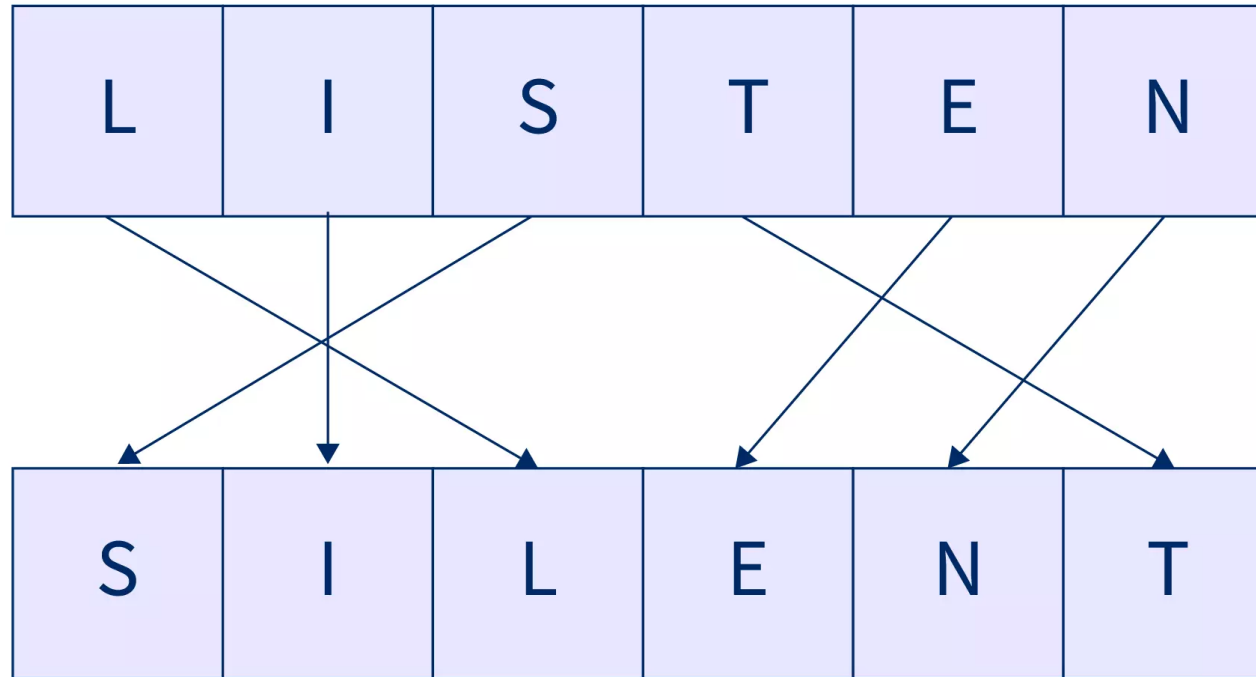
Tests passed: 2 of 2 tests - 7 ms

Test Name	Duration
testDivideByZero	7 ms
testAdd	0 ms

Process finished with exit code 0

Unit Testing - Workshop

Solve and write unit test for “isAnagram” problem



Resources

Java tutorial - <https://www.programiz.com/java-programming>

Junit - <https://www.javatpoint.com/junit-tutorial>

Data Structures in Java - <https://www.javatpoint.com/data-structure-tutorial>

Interview preparation - <https://neetcode.io/>

Interview preparation - <https://leetcode.com>

Youtube channels:

- <https://www.youtube.com/@amigoscode>
- <https://www.youtube.com/@CodingWithJohn>
- <https://www.youtube.com/@clem>
- <https://www.youtube.com/@laurspilca>
- https://www.youtube.com/@abdul_bari
- <https://www.youtube.com/@LifeatGoogle>
- <https://www.youtube.com/@LindaRaynier>