

Agenda

- **Java Methods**
- **Declaring and calling java method**
- **Java method return type**
- **Passing parameters in method**
- **Standard library methods**
- **Advantages of using methods**

Java Methods

In Java, the word **method** refers to the same kind of thing that the word *function* is used for in other languages. Specifically, a **method** is a function that belongs to a class. In Java, *every* function belongs to a class.

A method is a block of code that performs a specific task.

Suppose you need to create a program to create a circle and color it. You can create two methods to solve this problem:

- a method to draw the circle
- a method to color the circle

Dividing a complex problem into smaller chunks makes your program easy to understand and reusable.

In Java, there are two types of methods:

- **User-defined Methods:** We can create our own method based on our requirements.
- **Standard Library Methods:** These are built-in methods in Java that are available to use.

Let's first learn about user-defined methods.

Declaring a Java Method

The syntax to declare a method is:



```
returnType methodName() {  
    // method body  
}
```

Here,

- **returnType** - It specifies what type of value a method returns. For example, if a method has an int return type, then it returns an integer value. If the method does not return a value, its return type is void.
- **methodName** - It is an identifier that is used to refer to the particular method in a program.
- **method body** - It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the curly braces {}.

For example,



```
int addNumbers() {  
    // code  
}
```

In the above example, the name of the method is addNumbers(). And, the return type is int. We will learn more about return types later in this tutorial.

This is the simple syntax of declaring a method. However, the **complete syntax of declaring a method is**



```
modifier static returnType nameOfMethod (parameter1, parameter2, ...) {  
    // method body  
}
```

Here,

- **modifier** - It defines access types whether the method is public, private, and so on. To learn more, visit [Java Access Specifier](#).
- **static** - If we use the static keyword, it can be accessed without creating objects. For example, the sqrt() method of standard [Math class](#) is static. Hence, we can directly call Math.sqrt() without creating an instance of Math class.
- **parameter1/parameter2** - These are values passed to a method. We can pass any number of arguments to a method.

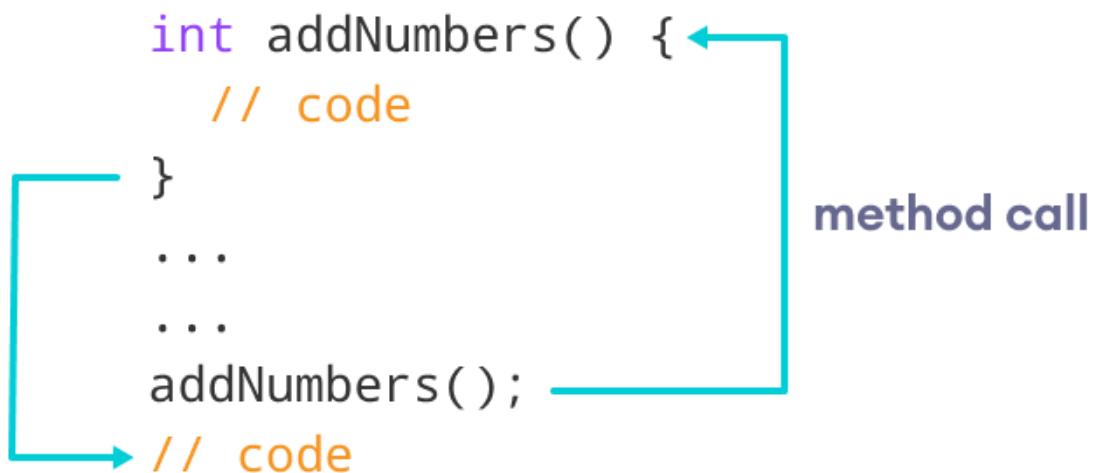
Calling a Method in Java

In the above example, we have declared a method named addNumbers(). Now, to use the method, we need to call it.

Here's how we can call the addNumbers() method.



```
// calls the method  
addNumbers();
```



Example 1: Java Methods



```
class Main {  
  
    // create a method  
    public int addNumbers(int a, int b) {  
        int sum = a + b;  
        // return value  
        return sum;  
    }  
  
    public static void main(String[] args) {
```

```
int num1 = 25;  
int num2 = 15;  
  
// create an object of Main  
Main obj = new Main();  
  
// calling method  
int result = obj.addNumbers(num1, num2);  
System.out.println("Sum is: " + result);  
}  
}
```

Output:



Sum is: 40

In the above example, we have created a method named `addNumbers()`. The method takes two parameters `a` and `b`. Notice the line,



```
int result = obj.addNumbers(num1, num2);
```

Here, we have called the method by passing two arguments `num1` and `num2`. Since the method is returning some value, we have stored the value in the `result` variable.

Note : The method is not static. Hence, we are calling the method using the object of the class.

Java Method Return Type

A Java method may or may not return a value to the function call. We use the **return statement** to return any value. For example,



```
int addNumbers() {  
    ...  
    return sum;  
}
```

Here, we are returning the variable sum. Since the return type of the function is int. The sum variable should be of int type. Otherwise, it will generate an error.

Example 2: Method Return Type



```
class Main {  
  
    // create a method  
    public static int square(int num) {  
  
        // return statement  
        return num * num;  
    }  
  
    public static void main(String[] args) {  
        int result;  
  
        // call the method  
        // store returned value to result  
        result = square(10);  
  
        System.out.println("Squared value of 10 is: " + result);  
    }  
}
```

Output:

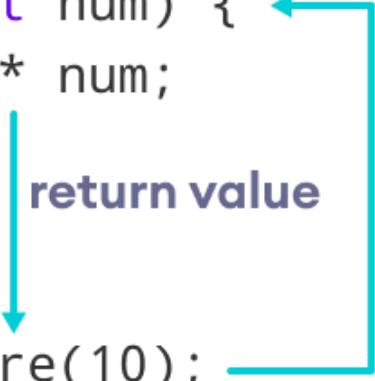


Squared value of 10 is: 100

In the above program, we have created a method named square(). The method takes a number as its parameter and returns the square of the number.

Here, we have mentioned the return type of the method as int. Hence, the method should always return an integer value.

```
int square(int num) { ←  
    return num * num;  
}  
...  
...  
result = square(10); ←  
// code
```



Note: If the method does not return any value, we use the void keyword as the return type of the method. For example,



```
public void square(int a) {  
    int square = a * a;  
    System.out.println("Square is: " + square);  
}
```

Method Parameters in Java

A method parameter is a value accepted by the method. As mentioned earlier, a method can also have any number of parameters. For example,



```
// method with two parameters  
int addNumbers(int a, int b) {  
    // code  
}
```

```
// method with no parameter  
  
int addNumbers(){  
    // code  
}
```

If a method is created with parameters, we need to pass the corresponding values while calling the method. For example,



```
// calling the method with two parameters  
  
addNumbers(25, 15);
```

```
// calling the method with no parameters  
  
addNumbers()
```

Example 3: Method Parameters



```
class Main {  
  
    // method with no parameter  
  
    public void display1() {  
        System.out.println("Method without parameter");  
    }  
  
    // method with single parameter  
  
    public void display2(int a) {  
        System.out.println("Method with a single parameter: " + a);  
    }  
  
    public static void main(String[] args) {
```

```
// create an object of Main  
Main obj = new Main();  
  
// calling method with no parameter  
obj.display1();  
  
// calling method with the single parameter  
obj.display2(24);  
}  
}
```

Output:



Method without parameter

Method with a single parameter: 24

Here, the parameter of the method is int. Hence, if we pass any other data type instead of int, the compiler will throw an error. It is because Java is a strongly typed language.

Note: The argument 24 passed to the display2() method during the method call is called the actual argument.

The parameter num accepted by the method definition is known as a formal argument. We need to specify the type of formal arguments. And, the type of actual arguments and formal arguments should always match.

Standard Library Methods

The standard library methods are built-in methods in Java that are readily available for use. These standard libraries come along with the Java Class Library (JCL) in a Java archive (*.jar) file with JVM and JRE.

For example,

- `print()` is a method of `java.io.PrintSteam`. The `print("...")` method prints the string inside quotation marks.
- `sqrt()` is a method of `Math class`. It returns the square root of a number.

Here's a working example:

Example 4: Java Standard Library Method



```
public class Main {  
    public static void main(String[] args) {  
  
        // using the sqrt() method  
        System.out.print("Square root of 4 is: " + Math.sqrt(4));  
    }  
}
```

Output:



Square root of 4 is: 2.0

To learn more about standard library methods, visit [Java Library Methods](#).

What are the advantages of using methods?

1. The main advantage is **code reusability**. We can write a method once, and use it multiple times. We do not have to rewrite the entire code each time. Think of it as, "write once, reuse multiple times".

Example 5: Java Method for Code Reusability



```
public class Main {  
  
    // method defined  
    private static int getSquare(int x){  
        return x * x;  
    }  
  
    public static void main(String[] args) {  
        for (int i = 1; i <= 5; i++) {  
    }
```

```
// method call  
int result = getSquare(i);  
System.out.println("Square of " + i + " is: " + result);  
}  
}  
}
```

Output:



Square of 1 is: 1

Square of 2 is: 4

Square of 3 is: 9

Square of 4 is: 16

Square of 5 is: 25

In the above program, we have created the method named `getSquare()` to calculate the square of a number. Here, the method is used to calculate the square of numbers less than **6**.

Hence, the same method is used again and again.

2. Methods make code more **readable and easier** to debug. Here, the `getSquare()` method keeps the code to compute the square in a block. Hence, makes it more readable.

Agenda

- **Basics of Java**
- **Flow of basic code in Java**
- **If Else Statements**
- **Loops in Java**
- **Best coding practices**

In the last module, we have learnt about Input/Output and exceptions in Java, Now, in this module we are going to go a little deep into the basic fundamentals of coding with Java.

So, lets get started with learning about the basics of Java.

Basics of Java

In basics lets get started with learning about the Variables in Java.

Variables in Java

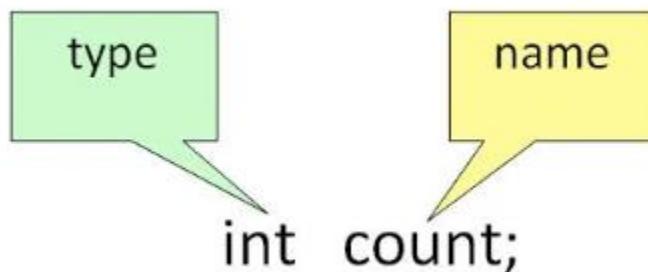
Variable in Java is a data container that stores the data values during Java program execution. Every variable is assigned data type which designates the type and quantity of value it can hold. Variable is a memory location name of the data. The Java variables have mainly three types : Local, Instance and Static.

In order to use a variable in a program you need to perform 2 steps

1. Variable Declaration
2. Variable Initialization

Variable Declaration:

To declare a variable, you must specify the data type & give the variable a unique name.



Examples of other Valid Declarations are:



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

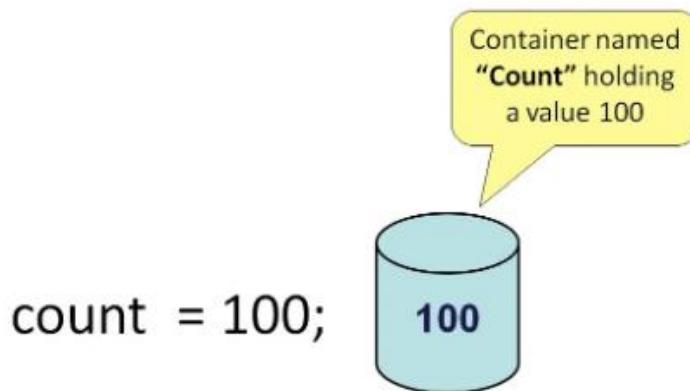
```
float pi;
```

```
double d;
```

```
char a;
```

Variable Initialization

To initialize a variable, you must assign it a valid value.



Example of other Valid Initializations are:



```
pi =3.14f;
```

```
do =20.22d;
```

```
a='v';
```

You can combine variable declaration and initialization as shown below:



```
int count = 100;
```

```
int a=2,b=4,c=6;
```

```
float pi=3.14f;
```

```
double do=20.22d;
```

```
char a='v';
```

Types of variables

In Java, there are three types of variables:

1. Local Variables
2. Instance Variables
3. Static Variables

1) Local Variables

Local Variables are a variable that are declared inside the body of a method.

2) Instance Variables

Instance variables are defined without the STATIC keyword .They are defined Outside a method declaration. They are Object specific and are known as instance variables.

3) Static Variables

Static variables are initialized only once, at the start of the program execution. These variables should be initialized first, before the initialization of any instance variables.

Consider the example shown below:



```
class AlmaBetter {  
    static int a = 1; //static variable  
    int data = 99; //instance variable  
    void method() {  
        int b = 90; //local variable  
    }  
}
```

Rules for Naming Variables in Java

Java programming language has its own set of rules and conventions for naming variables. Here's what you need to know:

- Java is case sensitive. Hence, age and Age are two different variables. For example,



```
int age = 24;
```

```
int AGE = 25;
```

```
System.out.println(age); // prints 24
```

```
System.out.println(AGE); // prints 25
```

- Variables must start with either a **letter** or an **underscore**, _ or a **dollar**, \$ sign. For example,



```
int age; // valid name and good practice
```

```
int _age; // valid but bad practice
```

```
int $age; // valid but bad practice
```

- Variable names cannot start with numbers. For example,



```
int 1age; // invalid variables
```

- Variable names can't use whitespace. For example,



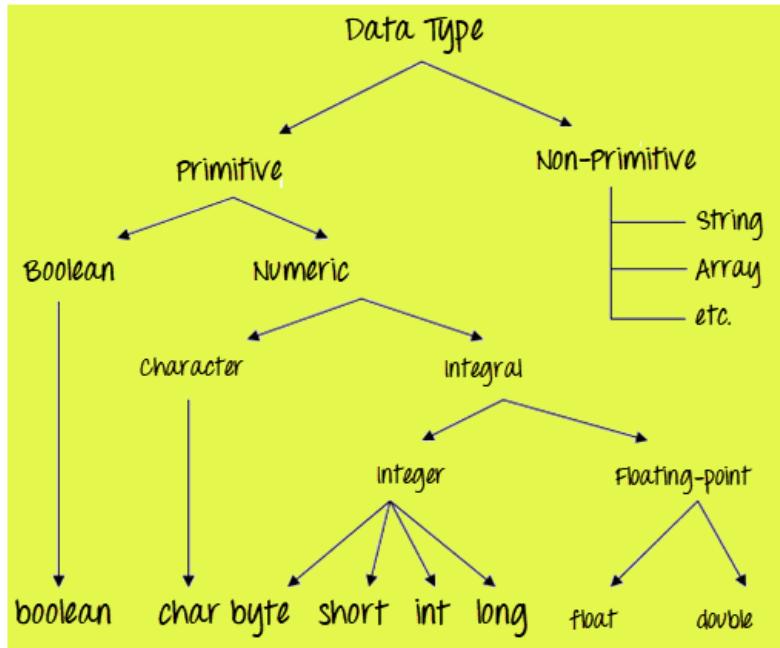
```
int my age; // invalid variables
```

What is Data Types in Java?

What is Data Types in Java?

Data Types in Java are defined as specifiers that allocate different sizes and types of values that can be stored in the variable or an identifier. Java has a rich set of data types. Data types in Java can be divided into two parts :

1. Primitive Data Types :- which include integer, character, boolean, and float
2. Non-primitive Data Types :- which include classes, arrays and interfaces.



Now, let's look at each data type one by one:

1. boolean type

- The boolean data type has two possible values, either true or false.
- Default value: false.
- They are usually used for **true/false** conditions.

Consider the example shown below:



```
class Main {
    public static void main(String[] args) {
```

```
        boolean flag = true;
        System.out.println(flag); // prints true
    }
}
```

2. byte type

- The byte data type can have values from **128** to **127** (8-bit signed two's complement integer).

- If it's certain that the value of a variable will be within -128 to 127, then it is used instead of int to save memory.
- Default value: 0

Consider the example shown below:



```
class Main {  
    public static void main(String[] args) {  
  
        byte range;  
        range = 124;  
        System.out.println(range); // prints 124  
    }  
}
```

3. short type

- The short data type in Java can have values from **32768** to **32767** (16-bit signed two's complement integer).
- If it's certain that the value of a variable will be within -32768 and 32767, then it is used instead of other integer data types (int, long).
- Default value: 0

Consider the example shown below:



```
class Main {  
    public static void main(String[] args) {  
  
        short temperature;  
        temperature = -200;  
        System.out.println(temperature); // prints -200  
    }  
}
```

4. int type

- The int data type can have values from **231** to **2311** (32-bit signed two's complement integer).
- If you are using Java 8 or later, you can use an unsigned 32-bit integer. This will have a minimum value of 0 and a maximum value of 21.
- Default value: 0

Consider the example shown below:



```
class Main {  
    public static void main(String[] args) {  
  
        int range = -4250000;  
        System.out.println(range); // print -4250000  
    }  
}
```

5. long type

- The long data type can have values from **263** to **2631** (64-bit signed two's complement integer).
- If you are using Java 8 or later, you can use an unsigned 64-bit integer with a minimum value of **0** and a maximum value of **2641**.
- Default value: 0

Consider the example shown below:



```
class LongExample {  
    public static void main(String[] args) {  
  
        long range = -42332200000L;  
        System.out.println(range); // prints -42332200000  
    }  
}
```

Notice, the use of L at the end of -42332200000. This represents that it's an integer of the long type.

6. double type

- The double data type is a double-precision 64-bit floating-point.
- It should never be used for precise values such as currency.
- Default value: 0.0 (0.0d)

Consider the example shown below:



```
class Main {  
    public static void main(String[] args) {  
  
        double number = -42.3;  
  
        System.out.println(number); // prints -42.3  
    }  
}
```

7. float type

- The float data type is a single-precision 32-bit floating-point. Learn more about [single-precision and double-precision floating-point](#) if you are interested.
- It should never be used for precise values such as currency.
- Default value: 0.0 (0.0f)

Consider the example shown below:



```
class Main {  
    public static void main(String[] args) {  
  
        float number = -42.3f;  
  
        System.out.println(number); // prints -42.3  
    }  
}
```

Notice that we have used -42.3f instead of -42.3in the above program. It's because -42.3 is a double literal.

To tell the compiler to treat -42.3 as float rather than double, you need to use f or F.

8. char type

- It's a 16-bit Unicode character.
- The minimum value of the char data type is '\u0000' (0) and the maximum value of the is '\uffff'.
- Default value: '\u0000'

Consider the example shown below:



```
class Main {  
    public static void main(String[] args) {  
  
        char letter = '\u0051';  
  
        System.out.println(letter); // prints Q  
    }  
}
```

Here, the Unicode value of Q is \u0051. Hence, we get Q as the output.

9. string type

In Java, a string is a sequence of characters. For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'.

We use **double quotes** to represent a string in Java. For example,



```
// create a string  
String type = "Java programming";
```

Here, we have created a string variable named type. The variable is initialized with the string Java Programming.

Consider the example shown below:



```
class Main {  
    public static void main(String[] args) {  
  
        // create strings  
  
        String first = "Java";  
  
        String second = "Python";  
  
        String third = "JavaScript";  
  
  
        // print strings  
  
        System.out.println(first); // print Java  
  
        System.out.println(second); // print Python  
  
        System.out.println(third); // print JavaScript  
    }  
}
```

In the above example, we have created three strings named first, second, and third. Here, we are directly creating strings like primitive types.

Java String Operations

Java String provides various methods to perform different operations on strings. We will look into some of the commonly used string operations.

1. Get length of a String

To find the length of a string, we use the length() method of the String. For example,



```
class Main {  
    public static void main(String[] args) {  
  
        // create a string  
  
        String greet = "Hello! World";  
  
        System.out.println("String: " + greet);  
    }  
}
```

```
// get the length of greet  
int length = greet.length();  
  
System.out.println("Length: " + length);  
}  
}
```

Output



String: Hello! World

Length: 12

In the above example, the `length()` method calculates the total number of characters in the string and returns it.

2. Join Two Java Strings

We can join two strings in Java using the `concat()` method. For example,



```
class Main {  
  
    public static void main(String[] args) {  
  
        // create first string  
        String first = "Java ";  
        System.out.println("First String: " + first);  
  
        // create second  
        String second = "Programming";  
        System.out.println("Second String: " + second);  
  
        // join two strings  
        String joinedString = first.concat(second);  
    }  
}
```

```
        System.out.println("Joined String: " + joinedString);
    }
}
```

Output



First String: Java

Second String: Programming

Joined String: Java Programming

In the above example, we have created two strings named first and second. Notice the statement,



```
String joinedString = first.concat(second);
```

Here, the concat() method joins the second string to the first string and assigns it to the joined String variable.

We can also join two strings using the + operator in Java.

3. Compare two Strings

In Java, we can make comparisons between two strings using the equals() method. For example,



```
class Main {
    public static void main(String[] args) {
        // create 3 strings
        String first = "java programming";
        String second = "java programming";
        String third = "python programming";

        // compare first and second strings
        boolean result1 = first.equals(second);
```

```
System.out.println("Strings first and second are equal: " + result1);

// compare first and third strings
boolean result2 = first.equals(third);

System.out.println("Strings first and third are equal: " + result2);
}
```

Output



Strings first and second are equal: true

Strings first and third are equal: false

In the above example, we have created 3 strings named first, second, and third. Here, we are using the equal() method to check if one string is equal to another.

10. array

An array is a collection of similar types of data.

For example, if we want to store the names of 100 people then we can create an array of the string type that can store 100 names.



```
String[] array = new String[100];
```

Here, the above array cannot store more than 100 names. The number of values in a Java array is always fixed.

How to declare an array in Java?

In Java, here is how we can declare an array.



```
dataType[] arrayName;
```

- `dataType` - it can be primitive data types like `int`, `char`, `double`, `byte`, etc. or Java objects
- `arrayName` - it is an identifier

For example,



```
double[] data;
```

Here, `data` is an array that can hold values of type `double`.

But, how many elements can array this hold?

Good question! To define the number of elements that an array can hold, we have to allocate memory for the array in Java. For example,



```
// declare an array
```

```
double[] data;
```

```
// allocate memory
```

```
data = new double[10];
```

Here, the array can store **10** elements. We can also say that the **size or length** of the array is 10.

In Java, we can declare and allocate the memory of an array in one single statement. For example,



```
double[] data = new double[10];
```

How to Initialize Arrays in Java?

In Java, we can initialize arrays during declaration. For example,



```
//declare and initialize an array
```

```
int[] age = {12, 4, 5, 2, 5};
```

Here, we have created an array named `age` and initialized it with the values inside the curly brackets.

Note that we have not provided the size of the array. In this case, the Java compiler automatically specifies the size by counting the number of elements in the array (i.e. 5).

In the Java array, each memory location is associated with a number. The number is known as an array index. We can also initialize arrays in Java, using the index number. For example,



```
// declare an array
```

```
int[] age = new int[5];
```

```
// initialize array
```

```
age[0] = 12;
```

```
age[1] = 4;
```

```
age[2] = 5;
```

```
..
```

Note:

- Array indices always start from 0. That is, the first element of an array is at index 0.
- If the size of an array is , then the last element of the array will be at index n-1 .

How to Access Elements of an Array in Java?

We can access the element of an array using the index number. Here is the syntax for accessing elements of an array,



```
// access array elements
```

```
array[index]
```

Let's see an example of accessing array elements using index numbers.

Example: Access Array Elements



```
class Main {
```

```
    public static void main(String[] args) {
```

```
// create an array  
int[] age = {12, 4, 5, 2, 5};  
  
// access each array elements  
System.out.println("Accessing Elements of Array:");  
System.out.println("First Element: " + age[0]);  
System.out.println("Second Element: " + age[1]);  
System.out.println("Third Element: " + age[2]);  
System.out.println("Fourth Element: " + age[3]);  
System.out.println("Fifth Element: " + age[4]);  
}  
}
```

Output



Accessing Elements of Array:

First Element: 12

Second Element: 4

Third Element: 5

Fourth Element: 2

Fifth Element: 5

In the above example, notice that we are using the index number to access each element of the array.

Now, we have learnt about variables and data types, so now lets start learning about if and else statements.

If...else statements

If Statement

The syntax of an **if-then** statement is:



```
if (condition) {
```

```
// statements  
}
```

Here, condition is a boolean expression such as age \geq 18.

- if evaluates to true, statements are executed

condition

- if evaluates to false, statements are skipped

condition

Working of if Statement

Condition is true

```
int number = 10;  
  
if (number > 0) {  
    // code  
}  
  
// code after if
```

Condition is false

```
int number = 10;  
  
if (number < 0) {  
    // code  
}  
  
// code after if
```

Consider the example shown below:



```
class IfStatement {  
    public static void main(String[] args) {  
  
        int number = 10;  
  
        // checks if number is less than 0
```

```
if (number < 0) {  
    System.out.println("The number is negative.");  
}  
  
System.out.println("Statement outside if block");  
}  
}
```

Output



Statement outside if block

In the program, `number < 0` is false. Hence, the code inside the body of the if statement is **skipped**.

Note: If you want to learn more about test conditions, visit [Java Relational Operators](#) and [Java Logical Operators](#).

if...else Statement

The if statement executes a certain section of code if the test expression is evaluated to true. However, if the test expression is evaluated to false, it does nothing.

In this case, we can use an optional else block. Statements inside the body of else block are executed if the test expression is evaluated to false. This is known as the **if...else** statement in Java.

The syntax of the **if...else** statement is:



```
if (condition) {  
    // codes in if block  
}  
  
else {  
    // codes in else block  
}
```

Here, the program will do one task (codes inside if block) if the condition is true and another task (codes inside else block) if the condition is false .

How the if...else statement works?

Condition is true

```
int number = 5;  
  
if (number > 0) {  
    // code  
}  
else {  
    // code  
}  
  
// code after if...else
```

Condition is false

```
int number = 5;  
  
if (number < 0) {  
    // code  
}  
else {  
    // code  
}  
  
// code after if...else
```

Consider the example shown below:



```
class Main {  
  
    public static void main(String[] args) {  
  
        int number = 10;  
  
        // checks if number is greater than 0  
        if (number > 0) {  
            System.out.println("The number is positive.");  
        }  
  
        // execute this block  
        // if number is not greater than 0  
        else {  
            System.out.println("The number is not positive.");  
        }  
  
        System.out.println("Statement outside if...else block");
```

```
}
```

```
}
```

Output



The number is positive.

Statement outside if...else block

In the above example, we have a variable named number. Here, the test expression `number > 0` checks if number is greater than 0.

Since the value of the number is 10, the test expression evaluates to true. Hence code inside the body of if is executed.

Now, change the value of the number to a negative integer. Let's say -5.



```
int number = -5;
```

If we run the program with the new value of number, the output will be:



The number is not positive.

Statement outside if...else block

Here, the value of number is -5. So the test expression evaluates to false. Hence code inside the body of else is executed.

Nested if...else statements

In Java, it is also possible to use if..else statements inside an if...else statement. It's called the nested if...else statement.

Here's a program to find the largest of 3 numbers using the nested if...else statement.



```
class Main {  
    public static void main(String[] args) {  
  
        // declaring double type variables  
    }  
}
```

```
Double n1 = -1.0, n2 = 4.5, n3 = -5.3, largest;

// checks if n1 is greater than or equal to n2
if (n1 >= n2) {

    // if...else statement inside the if block
    // checks if n1 is greater than or equal to n3
    if (n1 >= n3) {
        largest = n1;
    }

    else {
        largest = n3;
    }
} else {

    // if..else statement inside else block
    // checks if n2 is greater than or equal to n3
    if (n2 >= n3) {
        largest = n2;
    }

    else {
        largest = n3;
    }
}

System.out.println("Largest Number: " + largest);
}
```

}

Output:



Largest Number: 4.5

Now after learning conditional if else statements, let's start learning about the loops in Java.

Loops

In computer programming, loops are used to repeat a block of code. For example, if you want to show a message 100 times, then rather than typing the same code 100 times, you can use a loop.

In Java, there are three types of loops.

- for loop
- while loop
- do...while loop

For loop

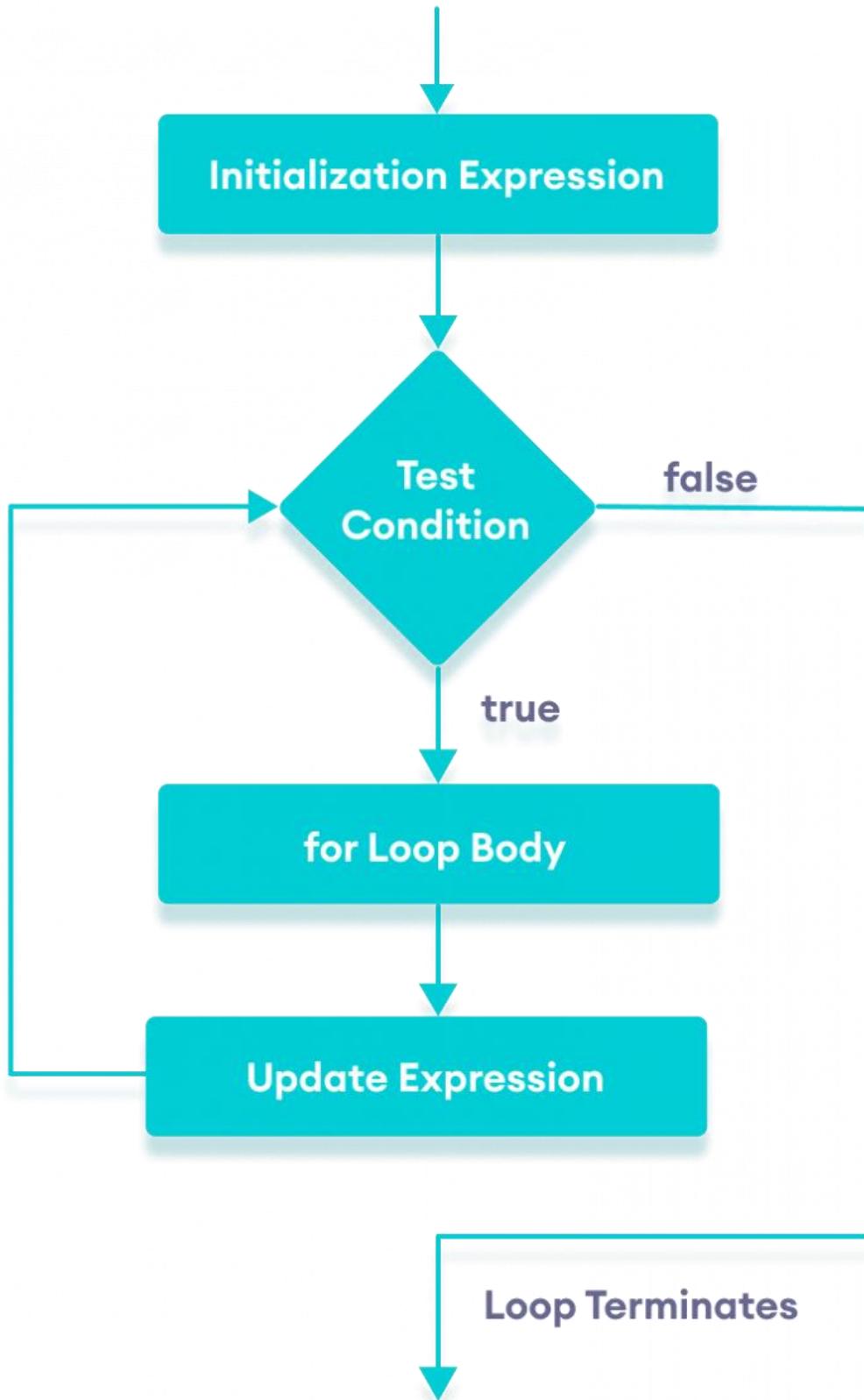
Java for loop is used to run a block of code for a certain number of times. The syntax of for loop is:



```
for (initialExpression; testExpression; updateExpression) {  
    // body of the loop  
}
```

Here,

1. The **initialExpression** initializes and/or declares variables and executes only once.
2. The **condition** is evaluated. If the **condition** is true, the body of the for loop is executed.
3. The **updateExpression** updates the value of **initialExpression**.
4. The condition is evaluated again. The process continues until the condition is **false**.



of Java for loop

Consider the example shown below, where we are displaying a text 5 times



```
// Program to print a text 5 times
```

```
class Main {  
    public static void main(String[] args) {  
  
        int n = 5;  
        // for loop  
        for (int i = 1; i <= n; ++i) {  
            System.out.println("Java is fun");  
        }  
    }  
}
```

Output:



```
Java is fun  
Java is fun  
Java is fun  
Java is fun  
Java is fun
```

Here is how this program works.

Iteration	Variable	Condition: $i \leq n$	Action
1st	$i = 1$ $n = 5$	true	Java is fun is printed. i is increased to 2.
2nd	$i = 2$ $n = 5$	true	Java is fun is printed. i is increased to 3.
3rd	$i = 3$ $n = 5$	true	Java is fun is printed. i is increased to 4.

Iteration	Variable	Condition: $i \leq n$	Action
4th	$i = 4n = 5$	true	Java is fun is printed. i is increased to 5.
5th	$i = 5n = 5$	true	Java is fun is printed. i is increased to 6.
6th	$i = 6n = 5$	false	The loop is terminated.

While loop

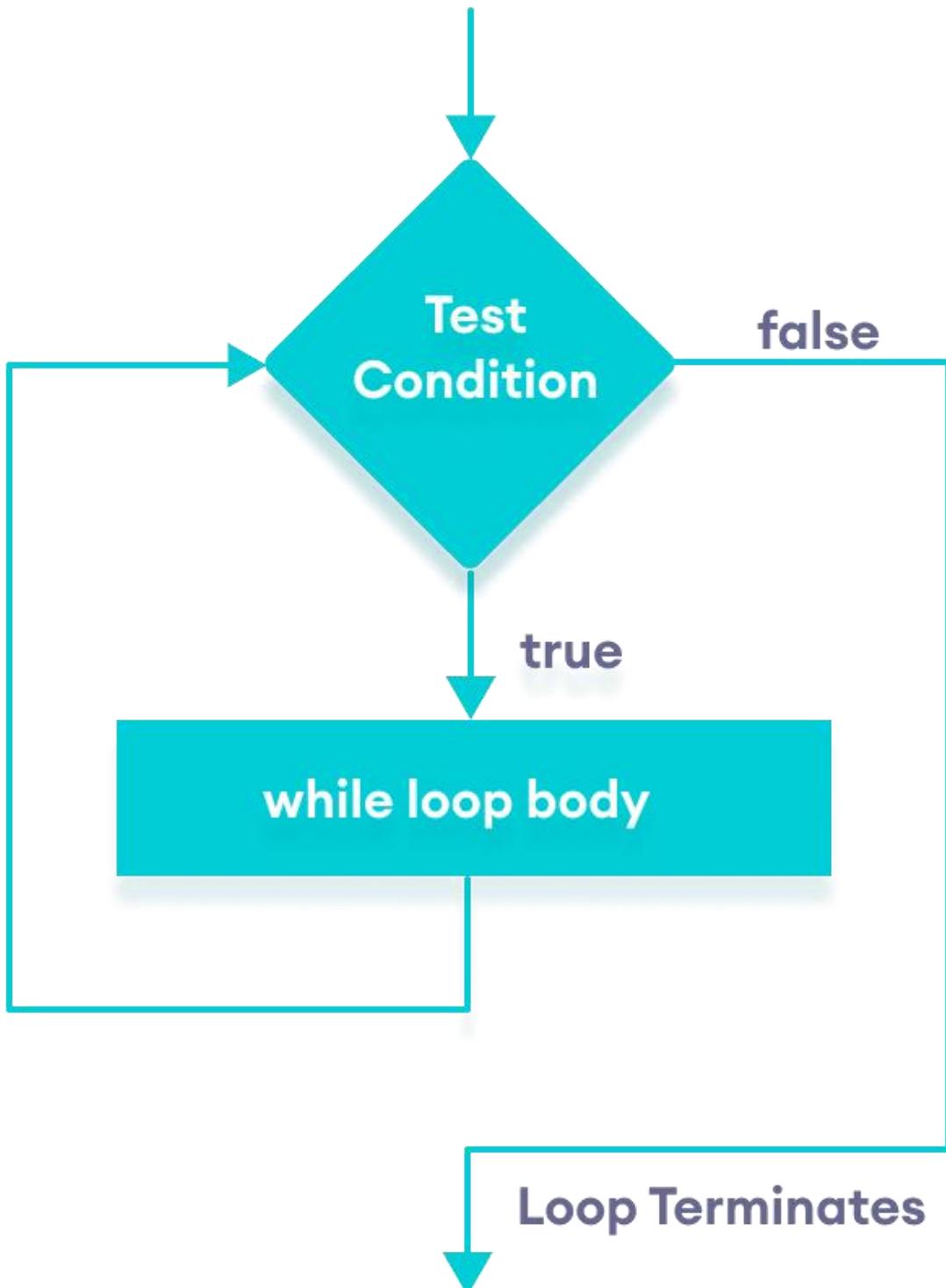
Java while loop is used to run a specific code until a certain condition is met. The syntax of the while loop is:



```
while (testExpression) {
    // body of loop
}
```

Here,

1. A while loop evaluates the **textExpression** inside the parenthesis () .
2. If the **textExpression** evaluates to true, the code inside the while loop is executed.
3. The **textExpression** is evaluated again.
4. This process continues until the **textExpression** is false.
5. When the **textExpression** evaluates to false, the loop stops.



Flowchart for while loop

Consider the example, where we are showing numbers from 1 to 5



```
// Program to display numbers from 1 to 5
```

```
class Main {  
    public static void main(String[] args) {  
  
        // declare variables  
        int i = 1, n = 5;  
  
        // while loop from 1 to 5  
        while(i <= n) {  
            System.out.println(i);  
            i++;  
        }  
    }  
}
```

Output:



```
1  
2  
3  
4  
5
```

Here is how this program works.

Iteration	Variable	Condition: $i \leq n$	Action
1st	$i = 1$ $n = 5$	true	1 is printed. i is increased to 2.

Iteration	Variable	Condition: $i \leq n$	Action
2nd	$i = 2n = 5$	true	2 is printed. i is increased to 3.
3rd	$i = 3n = 5$	true	3 is printed. i is increased to 4.
4th	$i = 4n = 5$	true	4 is printed. i is increased to 5.
5th	$i = 5n = 5$	true	5 is printed. i is increased to 6.
6th	$i = 6n = 5$	false	The loop is terminated

do...while loop

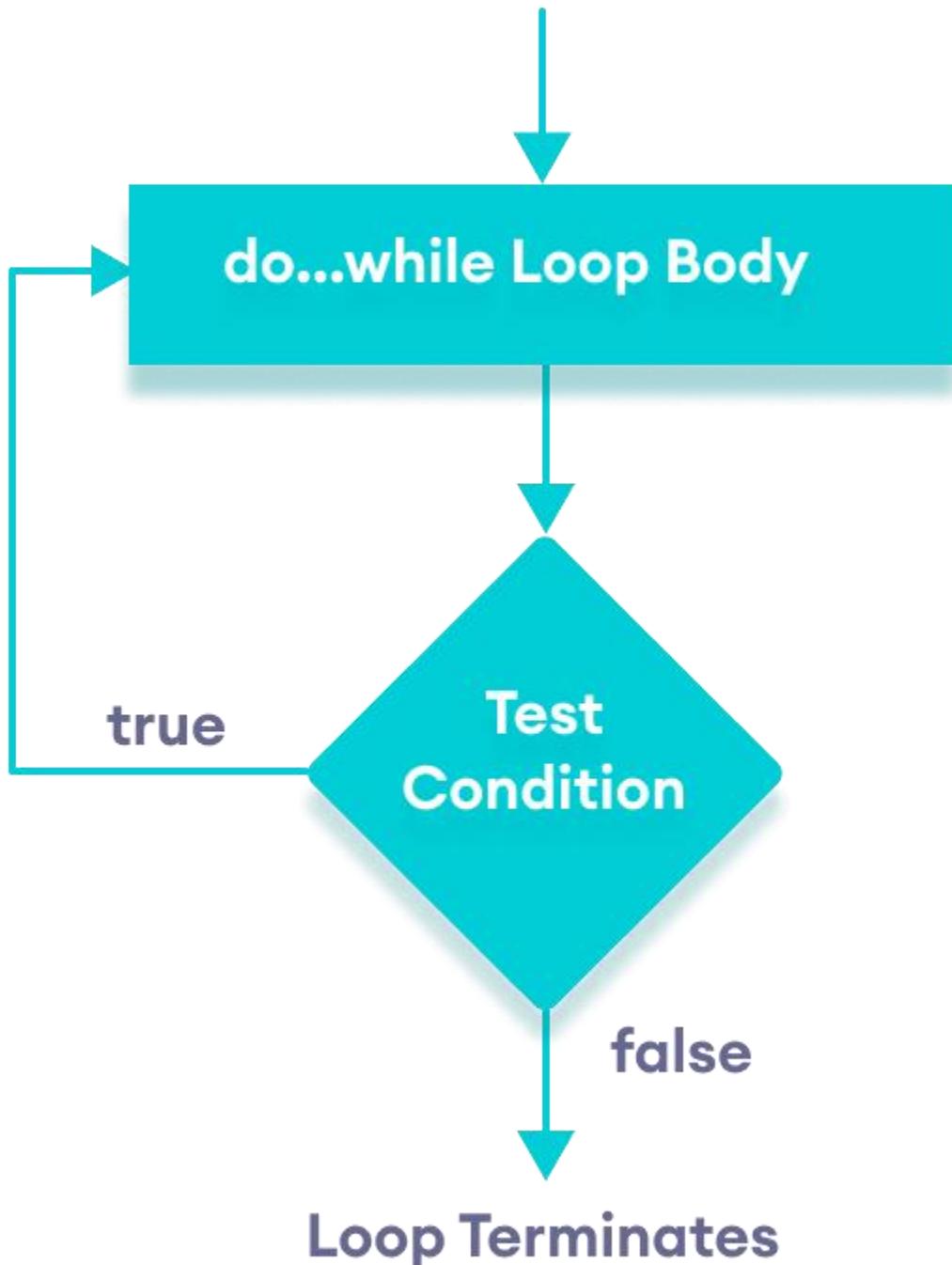
The do...while loop is similar to while loop. However, the body of do...while loop is executed once before the test expression is checked. For example,



```
do {
    // body of loop
} while(textExpression);
```

Here,

1. The body of the loop is executed at first. Then the **textExpression** is evaluated.
2. If the **textExpression** evaluates to true, the body of the loop inside the do statement is executed again.
3. The **textExpression** is evaluated once again.
4. If the **textExpression** evaluates to true, the body of the loop inside the do statement is executed again.
5. This process continues until the **textExpression** evaluates to false. Then the loop stops.



Flowchart of do...while loop

Consider the example shown below, where we are showing numbers from 1 to 5



```
// Java Program to display numbers from 1 to 5

import java.util.Scanner;

// Program to find the sum of natural numbers from 1 to 100.

class Main {
    public static void main(String[] args) {

        int i = 1, n = 5;

        // do...while loop from 1 to 5
        do {
            System.out.println(i);
            i++;
        } while(i <= n);
    }
}
```

Output



```
1
2
3
4
5
```

Here is how this program works.

Iteration	Variable	Condition: $i \leq n$	Action
	$i = 1$ $n = 5$	not checked	1 is printed. i is increased to 2.
1st	$i = 2$ $n = 5$	true	2 is printed. i is increased to 3.
2nd	$i = 3$ $n = 5$	true	3 is printed. i is increased to 4.
3rd	$i = 4$ $n = 5$	true	4 is printed. i is increased to 5.
4th	$i = 5$ $n = 5$	true	5 is printed. i is increased to 6.
5th	$i = 6$ $n = 5$	false	The loop is terminated

Let's end this session, now look at a bit on the best coding practices to improve coding efficiency:

Best coding practices

1. Variable Naming Conventions

Web developers often use simple variable names like A1 and B1 as temporary placeholders, but later forget to replace them with something more meaningful. This makes code less readable and ultimately leads to confusion.

One of the first things you learn when coding is that your variable names should be easy to understand and clearly represent the data they store. The way you name your variables is key to making your code readable. The idea of variable naming while coding is simple: to create variable names that are self-explanatory and follow a consistent theme throughout the code.

2. Add clear and concise comments in your code

It's pretty much guaranteed that your code will be modified or updated over time. It's also true that almost all developers will come across someone else's code at one time or another. A bad habit among inexperienced programmers is to include little or no comments while coding.

This poses a significant challenge for programmers working in a team, where more than one person may be working on a particular module.

Coding comments are segments of code that are ignored by the compiler. This means they are never passed to the computer and are not processed. Their sole purpose is to help the programmer understand the code, especially when returning to work on unfamiliar scripts in the future.

3. Indentation

Formatting and indentation are necessary to organize your code. Ideal coding formatting and indentation include correct spacing, line length, wraps and breaks. By employing indentations, white-spacing, and tabs within the code, programmers ensure their code is readable and organized.

Bear in mind there is no right or wrong way to indent your code. There are popular opinions but nothing is universally followed.

4. Reusability and Scalability

In coding, reusability is an essential design goal.

Because if modules and components have been tested already, a lot of time can be saved by reusing them. Software projects often begin with an existing framework or structure that contains a previous version of the project. Therefore, by reusing existing software components and modules, you can cut down on development cost and resources.

This directly results in faster delivery of the project, thereby increasing profitability.

Another key aspect to pay attention to is the ‘scalability’ of code. As user demands change, new features and improvements are constantly added to an application. Therefore, the ability to incorporate updates is an essential part of the software design process.

Interview Questions

Why Is Specifying DataType Mandatory In Java?

Data type of a variable is an attribute which tells what kind of data that variable can have. Every java variable takes up a certain amount of space in memory. How much memory a variable takes depends on its datatype.

For Example, As we see in our daily life, Different types of vehicle requires different amount of space in your parking area. For example, Our two wheeler may require two blocks where as Car may require 6 blocks of space. Same applies to variables in java. Storing a small number requires less space compared to large number. So, we need to declare a datatype of our variable accordingly.

Why Strings in Java are called as Immutable?

In java, string objects are called immutable as once value has been assigned to a string, it can't be changed and if changed, a new object is created.

In below example, reference str refers to a string object having value “Value one”.



```
String str="Value One";
```

When a new value is assigned to it, a new String object gets created and the reference is moved to the new object.



```
str="New Value";
```

What is an infinite Loop? How infinite loop is declared?

An infinite loop runs without any condition and runs infinitely. An infinite loop can be broken by defining any breaking logic in the body of the statement blocks.



```
for (;;) {  
    // Statements to execute
```

```
    // Add any loop breaking logic  
}
```

Thank You !

Agenda

- Writing your first Java program
- Compile and run a Java program
- Performing mathematical operations on Integers
- Taking Integers as user input
- Exception Handling

In this session, we will get you started on programming in Java, and study a variety of interesting programs.

First Java Program

In this section we'll take you through the three basic steps required to get a simple program running. It is understood that Java and an IDE or code editor is properly installed on your computer, according to the instructions specified in previous session.

Programming in Java.

We break the process of programming in Java into three steps:

1. *Create* the program by typing it into a text editor and saving it to a file named, say, MyProgram.java
2. *Compile* it by typing javac MyProgram.java in the terminal window.
3. *Execute* (or *run*) it by typing java MyProgram in the terminal window.

The first step creates the program; the second translates it into a language more suitable for machine execution (and puts the result in a file named MyProgram.class); the third actually runs the program.

Creating a Java program. A program is nothing more than a sequence of characters, like a sentence, a paragraph, or a poem. To create one, we need only define that sequence characters using a text editor in the same way as we do for email. HelloWorld.java is an example program. Type these character into your text editor and save it into a file named HelloWorld.java



```
public class HelloWorld {  
    public static void main(String[] args) {  
        // Prints "Hello, World" in the terminal window.  
        System.out.println("Hello, World");  
    }  
}
```

Compiling a Java program A *compiler* is an application that translates programs from the Java language to a language more suitable for executing on the computer. It takes a text file with the .java extension as input (your program) and produces a file with a .class extension (the computer-language version). To compile HelloWorld.java type the text below at the terminal.



```
javac HelloWorld.java
```

If you typed in the program correctly, you should see no error messages. Otherwise, go back and make sure you typed in the program exactly as it appears above.

Executing (or running) a Java program

Once you compile your program, you can execute it. This is the exciting part, where the computer follows your instructions. To run the program, type the following in the terminal window:



```
java HelloWorld
```

If all goes well, you should see the following response :



```
Hello, World
```

Understanding a Java program

The key line with System.out.println() prints the text "Hello, World" in the terminal window. When we begin to write more complicated programs, we will discuss the meaning of public, class, main, String[], args, System.out, and so on.

Creating your own Java Program

For the time being, all of our programs will be just like HelloWorld.java, except with a different sequence of statements in main(). The easiest way to write such a program is to:

- Copy HelloWorld.java into a new file whose name is program name followed by .java
- Replace HelloWorld with the program name everywhere
- Replace the print statement by a sequence of statements.

Errors

Most errors are easily fixed by carefully examining the program as we create it, in just the same way as we fix spelling and grammatical errors when we type an e-mail message.

- *Syntax errors.* A syntax error occurs when the code given does not follow the syntax rules of the programming language (so the compiler issues an error message that tries to explain why). Examples include:

- misspelling a statement, ex: mian() instead of main()
- missing brackets, ex: opening a bracket, but not closing it, etc.

A program cannot run if it has syntax errors. Any such errors must be fixed first. A good **integrated development environment (IDE)** usually points out any syntax errors to the programmer.

- *Run-time errors.* These errors are caught by the system when we execute the program, because the program tries to perform an invalid operation (e.g., division by zero).
- *Logical errors.* A logic error is an error in the way a program works. The program can run but does not do what it is expected to do. Some situations where a programmer can cause logical errors are:
 - Incorrectly using brackets in calculations
 - Incorrectly using mathematical operators
 - Unintentionally using wrong values, etc.

One of the very first skills that you will learn is to identify errors; one of the next will be to be sufficiently careful when coding to avoid many of them.

Integer Data type

A *data type* is a set of values and a set of operations defined on them. For example, we are familiar with numbers and with operations defined on them such as addition and multiplication. There are eight different built-in types of data in Java, mostly different kinds of numbers. For today's session we'll just be discussing about the int data type which is used to store integers, but as we proceed through the course we'll be discussing the other data types too while writing complex programs.

Terminology We use the following code fragment to introduce some terminology:



```
int a, b, c;  
a = 1234;  
b = 99;  
c = a + b;
```

The first line is a *declaration statement* that declares the names of three *variables* using the *identifiers* a, b, and c and their type to be int. The next three lines are *assignment statements* that change the values of the variables, using the *literals* 1234 and 99, and the *expression* a + b, with the end result that c has the value 1333.

Integers

An int is an integer (whole number) between -2^{31} and $2^{31} - 1$ ($-2,147,483,648$ to $2,147,483,647$). We use int frequently not just because they occur frequently in the real world, but also they naturally arise when expressing algorithms. Standard arithmetic operators for addition, multiplication, and division, for integers are built into Java, as illustrated in the following table:

<i>expression</i>	<i>value</i>	<i>comment</i>
99	99	<i>integer literal</i>
+99	99	<i>positive sign</i>
-99	-99	<i>negative sign</i>
5 + 3	8	<i>addition</i>
5 - 3	2	<i>subtraction</i>
5 * 3	15	<i>multiplication</i>
5 / 3	1	<i>no fractional part</i>
5 % 3	2	<i>remainder</i>
1 / 0		<i>run-time error</i>
3 * 5 - 2	13	* has precedence
3 + 5 / 2	5	/ has precedence
3 - 5 - 2	-4	left associative
(3 - 5) - 2	-4	better style
3 - (5 - 2)	0	unambiguous

Output in Java

In Java, you can simply use



System.out.println(); // or

System.out.print(); // or

System.out.printf();

to send output to standard output (screen).

Here,

- System is a class
- out is a public static field: it accepts output data.

Don't worry if you don't understand it. We will discuss class, public, and static in future sessions.

Difference between println(), print() and printf()

- print() - It prints string inside the quotes.
- println() - It prints string inside the quotes similar like print() method. Then the cursor moves to the beginning of the next line.
- printf() - It provides string formatting

Example: print() and println():



```
class Output {  
    public static void main(String[] args) {  
  
        System.out.println("1. println ");  
        System.out.println("2. println ");  
  
        System.out.print("1. print ");  
        System.out.print("2. print");  
    }  
}
```

```
    }  
}  
  
/*
```

OUTPUT :

1. `println`
 2. `println`
 1. `print` 2. `print`
- ```
*/
```

#### **Example: Printing Variables and Literals**



```
class Variables {
 public static void main(String[] args) {

 int number = -10;

 System.out.println(5);
 System.out.println(number);
 }
}
```

```
/*
 OUTPUT:
 5
 -10
*/
```

Here, you can see that we have not used the quotation marks. It is because to display integers, variables and so on, we don't use quotation marks.

#### **Example: Print Concatenated Strings**



```
class PrintVariables {
 public static void main(String[] args) {

 int number = -10;

 System.out.println("I am " + "awesome.");
 System.out.println("Number = " + number);
 }
}
```

/\*

OUTPUT :

I am awesome.

Number = -10

\*/

In the above example, notice the line,



```
System.out.println("I am " + "awesome.");
```

Here, we have used the + operator to concatenate (join) the two strings: "I am " and "awesome.".

And also, the line,



```
System.out.println("Number = " + number);
```

Here, first the value of variable number is evaluated. Then, the value is concatenated to the string: "Number = ".

Now, let's use what we've learned so far to write a program that illustrates integer operations in Java :



```

* Compilation: javac IntOps.java
* Execution: java IntOps a b
*
* Illustrates the integer operations a + b, a * b, a / b, and a % b.
*
* % java IntOps
* 10 + -3 = 7
* 10 * -3 = -30
* 10 / -3 = -3
* 10 % -3 = 1
* 10 = -3 * -3 + 1
*
*****/
```

```
public class IntOps {

 public static void main(String[] args) {
 int a = 10;
 int b = -3;
 int sum = a + b;
 int prod = a * b;
 int quot = a / b;
 int rem = a % b;

 System.out.println(a + " + " + b + " = " + sum);
 System.out.println(a + " * " + b + " = " + prod);
 System.out.println(a + " / " + b + " = " + quot);
 System.out.println(a + " % " + b + " = " + rem);
 }
}
```

```
 System.out.println(a + " = " + quot + " * " + b + " + " + rem);
 }
}
```

### Java Input

Now let's see how we can make our program dynamic and more useful by taking the input values from the user instead of using hardcoded values like in previous example. Java provides different ways to get input from the user. However, in this session, you will learn to get input from user using the object of Scanner class.

In order to use the object of Scanner, we need to import java.util.Scanner package.

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Built-in java packages consist of a large number of classes which are a part of Java API. java.util is one such package which contains the Scanner class. This class implements the code to get user data from command-line and can be imported and used in any java program.

Syntax:



```
import java.util.Scanner;
```

Then, we need to create an object of the Scanner class. We can use the object to take input from the user.



```
// create an object of Scanner
```

```
Scanner input = new Scanner(System.in);
```

```
// take input from the user
```

```
int number = input.nextInt()
```

### Example: Get Integer Input From the User



```
import java.util.Scanner;
```

```
class Input {
```

```

public static void main(String[] args) {

 Scanner input = new Scanner(System.in);

 System.out.print("Enter an integer: ");
 int number = input.nextInt();
 System.out.println("You entered " + number);

 // closing the scanner object
 input.close();
}

/*
Enter an integer: 23
You entered 23
*/

```

In the above example, we have created an object named `input` of the `Scanner` class. We then call the `nextInt()` method of the `Scanner` class to get an integer input from the user.

**Note:** We have used the `close()` method to close the object. It is recommended to close the scanner object once the input is taken.

Now that we have a clearer idea about taking user-inputs in Java and performing arithmetical operations, let's try to convert the pseudo-code to find the sum of two numbers from previous section to a working Java program. Being able to convert pseudo-code to language-specific code is one of the most important skills you'll need as a programmer.

Pseudo-code :



begin

  numeric nNum1,nNum2,nSum

  display "ENTER THE FIRST NUMBER : "

```
accept nNum1
display "ENTER THE SECOND NUMBER : "
accept nNum2
compute nSum=nNum1+nNum2
display "SUM OF THESE NUMBER : " nSum
end
```

Java :



```
import java.util.Scanner;

class CalcSum {
 public static void main(String[] args) {

 Scanner input = new Scanner(System.in);

 int nNum1, nNum2, nSum;

 System.out.print("ENTER THE FIRST NUMBER : ");
 nNum1 = input.nextInt();
 System.out.println("ENTER THE SECOND NUMBER : ");
 nNum2 = input.nextInt();

 nSum = nNum1 + nNum2;
 System.out.println("SUM OF THESE NUMBER : " + nSum);

 // closing the scanner object
 input.close();
 }
}
```

```
/*
OUTPUT
ENTER THE FIRST NUMBER : 12
ENTER THE SECOND NUMBER : 25
SUM OF THESE NUMBER : 37
*/
```

### **Exceptions**

When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an **exception** (throw an error).

### **Exception Handling in Java**

When an exception occurs in the program, the program execution is terminated. As this is an abrupt termination, the system generates a message and displays it. The message generated by the system may be cryptic like some codes or unreadable.

Thus the normal user should understand why the program stopped its execution abruptly, he/she should know the reason. The system generated messages as a result of exception may not be helpful. In Java, we can handle the exception and provide meaningful messages to the user about the issue.

This handling of exception, commonly known as "**Exception handling**" is one of the salient features of Java programming.

### **Reasons for the Exception to occur**

We can have various reasons due to which exceptions can occur. If its an exception related to input, then the reason may be that the input data is incorrect or unreadable.

If we get an exception for file I/O then it is quite possible that the files we are dealing with do not exist. At some other time, there may be errors like network issues, printer not available or functioning, etc.

In a program, apart from exceptions, we also get errors. Thus, to handle the exceptions effectively, we need to be aware of the differences between error and an exception.

An error indicates a more serious issue with the application and the application should not attempt to catch it. On the contrary, the exception is a condition that any reasonable application will try to catch.

Thus an error in the application is more severe and the applications would crash when they encounter an error. Exceptions on the other hand occur in code and can be handled by the programmer by providing corrective actions.

## **What is Exception Handling?**

The Exception Handling in Java is a mechanism using which the normal flow of the application is maintained. To do this, we employ a powerful mechanism to handle runtime errors or exceptions in a program.

A sequence of code that is used to handle the exception is called the “Exception handler”. An exception handler interrogates the context at the point when the exception occurred. This means that it reads the variable values that were in scope while the exception occurred and then restores the Java program to continue with normal flow.

## **Benefits of Exception Handling**

The major benefit of Exception handling is that it maintains the normal flow of the application despite the occurrence of an exception. When an exception occurs, the program usually terminates abruptly.

Having an exception handler in a program will not cause the program to terminate abruptly. Instead, an exception handler makes sure that all the statements in the program are executed normally and the program flow doesn't break abruptly.

## **Java Try and Catch**

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs. Syntax:



```
try {
 // Block of code to try
}

catch(Exception e) {
 // Block of code to handle errors
}
```

Consider the following example :



```
public class Main {

 public static void main(String[] args) {
 System.out.println(myNumber); // error!
 }
}
```

```
}
```

```
}
```

This will generate an error, because **myNumber** does not exist.

If an error occurs, we can use try...catch to catch the error and execute some code to handle it:



```
public class Main {
 public static void main(String[] args) {
 try {
 System.out.println(myNumber);
 } catch (Exception e) {
 System.out.println("Something went wrong.");
 }
 }
}
```

```
/*
```

OUTPUT :

Something went wrong.

```
*/
```

### Finally

The finally statement lets you execute code, after try...catch, regardless of the result:



```
public class Main {
 public static void main(String[] args) {
 try {
 System.out.println(myNumber);
 } catch (Exception e) {
 System.out.println("Something went wrong.");
 }
 }
}
```

```

 } finally {
 System.out.println("The 'try catch' is finished.");
 }
 }
}

/*
OUTPUT
Something went wrong.
The 'try catch' is finished.
*/

```

Now analyzing the code we wrote earlier, you'll notice that we're using `nextInt()` to read Integer values from the command line but the user may enter an alphabet or other non-integer values in which case the program ends abruptly with the following error :



```

Exception in thread "main" java.util.InputMismatchException
at java.base/java.util.Scanner.throwFor(Scanner.java:939)
at java.base/java.util.Scanner.next(Scanner.java:1594)
at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
at CalcSum.main(CalcSum.java:11)

```

The user won't understand the meaning behind this message generated by the system, so let's try to rewrite the same program and display appropriate message if a user enters an invalid value.



```

import java.util.Scanner;

class CalcSum {
 public static void main(String[] args) {
 Scanner input = new Scanner(System.in);
 int nNum1, nNum2, nSum;
 }
}

```

```

 try{

 System.out.print("ENTER THE FIRST NUMBER : ");
 nNum1 = input.nextInt();

 System.out.println("ENTER THE SECOND NUMBER : ");
 nNum2 = input.nextInt();

 nSum = nNum1 + nNum2;

 System.out.println("SUM OF THESE NUMBER : " +
nSum);

 } catch (Exception e) {

 System.out.println("Please enter Integer values only!");
 }

 // closing the scanner object
 input.close();
 }
}

```

This ensures that our program displays an appropriate error message to the user before its termination.

## **Conclusion**

In this session we learned about :

- Writing and Compiling a Java Program
- Taking inputs from the user
- Handling Exceptions

## **Interview Questions**

What is the difference between exception and error in Java?

Errors typically happen while an application is running. For instance, Out of Memory Error occurs in case the JVM runs out of memory. On the other hand, exceptions are mainly caused by the application. For instance, Null Pointer Exception happens when an app tries to get through a null object.

Why do we need exception handling in Java?

If there is no try and catch block while an exception occurs, the program will terminate. Exception handling ensures the smooth running of a program without program termination.

Thank You !

## Agenda

- What is programming and Why Programming?
- Programming use cases
- Writing pseudo code
- Differences in different programming languages
- Intro to Java and Development setup

### What is programming and Why Programming?



We all have heard about Computer Programming gaining a lot of popularity in the past 3 decades. So many students these days want to opt for a Computer Science stream in order to get a job at their dream tech company - Google, Facebook, Microsoft, Apple, and whatnot.

### Understanding Programming in layman terms

*Programming is a way to “instruct the computer to perform various tasks”.*

Confusing? Let us understand the definition deeply.

**“Instruct the computer”:** this basically means that you provide the computer a set of instructions that are written in a language that the computer can understand. The instructions could be of various types. For example:

- Adding 2 numbers,

- Rounding off a number, etc.

Just like we humans can understand a few languages (English, Spanish, Mandarin, French, etc.), so is the case with computers. Computers understand instructions that are written in a specific syntactical form called a programming language.

**“Perform various tasks”:** the tasks could be simple ones like we discussed above (adding 2 numbers, rounding off a number) or complex ones which may involve a sequence of multiple instructions. For example:

- Calculating simple interest, given principal, rate and time.
- Calculating the average return on a stock over the last 5 years.

The above 2 tasks require complex calculations. They cannot usually be expressed in simple instructions like adding 2 numbers, etc.

Hence, in summary, Programming is a way to tell computers to do a specific task.

### **Why should you bother about coding?**

You must be wondering - why does one need a computer for adding or rounding off numbers? Or even for simple interest calculation? After all, even an 8th standard kid can easily do such things even over large numbers. What is programming used for? What benefits do computers offer?

Well, computers offer so many benefits:

- **Computers are fast:** computers are amazingly fast. If you know how to properly utilize the power of Computer programming, you can do wonders with it. For a typical computer of today's time, an addition of 2 numbers which could be as big as a billion each takes hardly a nanosecond. Read again - nanosecond! That means that in 1 second, a computer can perform about a billion additions. Can any human ever do that? Forget a billion additions a second, typical human can't even do 10 additions per second. So, computers offer great speed.
- **Computers are cheap:** if you were a stock market analyst and you had to monitor the data of say 1000 stocks so that you can quickly trade them. Imagine the hassle that would create if you were to do it manually! It is just impractical. While you are performing your calculation on the stock's performance, the price may change. The other alternative is to hire people so that you can monitor more stocks in parallel. That means your cost goes up significantly. Not to mention the trouble you will face if some of your employees commit a calculation error in the process. You may end up losing money! Contrast that with the case where you use a computer. Computers can process a huge amount of information quickly and reliably. 1000 stocks are nothing for computers in the 21st century.
- **Computers can work 24x7:** Computers can work 24x7 without getting exhausted. So, if you have a task that is big enough, you can without worries allocate it to a computer by programming it and sleep peacefully.

### **What is Programming Language?**

As mentioned above, Computers understand instructions that are written in a specific syntactical form called a programming language. A programming language provides a way for a programmer to express a task so that it could be understood and executed by a computer. Some of the popular Programming languages are Python, C, C++, Java, etc.

### **Why should you learn Computer Programming?**

Now, after knowing so many things about programming, the big question to be answered is - why should you learn Computer Programming? Let us understand why:

- **Programming is fun:** Using Programming, you can create your own games, your personal blog/profile page, a social networking site like Facebook, a search engine like Google or an e-commerce platform like Amazon! Won't that be fun? Imagine creating your own game and putting it on Play Store and getting thousands and thousands of downloads!
- **The backbone of a Technology Company:** The backbones of today's technology companies like Google, Facebook, Microsoft, Apple, Amazon, and many others, are giant computer programs written by a collaboration of thousands of skilled programmers. If you have the right business acumen, knowing programming can help you create the next big tech company.
- **Pretty good salary:** Computer Programmers are paid extremely well almost all across the world. Top programmers in Silicon Valley make millions of dollars every year. Quite a few companies offer to start salaries as high as \$100,000 per year.

### **What is Programming Used For?**

Developers use a variety of programming languages to build websites and applications. Front-end developers typically use HTML, CSS, or JavaScript code to create website layouts and design functions.

Websites with user accounts, like Facebook, Instagram, or LinkedIn, require back-end developers to write computer code that connects websites to databases. Back-end developers often write code using languages such as SQL, Java, and Python.

Developers use other languages like [Python](#), Objective-C, [C#](#), Swift, or [Ruby on Rails](#) to create apps for cell phones and computer software.

Below, we list popular programming languages and their most common uses.

- **C:** Used for developing software operating systems and databases
- **Python:** Used for building websites and software programs and performing data analysis
- **HTML:** Used for creating the structures of a webpage, like paragraphs, links, and tables
- **Ruby on Rails:** Used for developing websites and applications and performing data analysis
- **C++:** Used for creating and developing games
- **C#:** Used for creating desktop applications and web services
- **Scala:** Used for data engineering, data processing, and web development support

- **Perl:** Used for text manipulation, web development, and network programming
- **PHP:** Used for managing databases and creating dynamic webpages
- **SQL:** Used for communicating with databases and managing and organizing data
- **JavaScript:** Used for creating webpages and supporting front-end and back-end development
- **Swift:** Used for creating apps, most commonly for Apple platforms
- **Objective-C:** Used for writing software for Apple products

## Pseudo Code

### What is Pseudocode?

Pseudocode literally means ‘fake code’. It is an **informal** and **contrived** way of writing programs in which you represent the sequence of actions and instructions (aka algorithms) in a form that humans can easily understand.

You see, computers and human beings are quite different, and therein lies the problem.

The language of a computer is very rigid: you are not allowed to make any mistakes or deviate from the rules. Even with the invention of high-level, human-readable languages like JavaScript and Python, it’s still pretty hard for an average human developer to reason and program in those coding languages.

With pseudocode, however, it’s the exact opposite. You make the rules. It doesn’t matter what language you use to write your pseudocode. All that matters is comprehension.

In pseudocode, you don’t have to think about semi-colons, curly braces, the syntax for arrow functions, how to define promises, DOM methods and other core language principles. You just have to be able to explain what you’re thinking and doing.

### Writing a Pseudo Code

For example, a **print** is a function in **python** to display the content whereas it is **System.out.println** in case of **java**, but as pseudocode display/output is the word which covers both the programming languages.

So that the program written in an informal language and could be understood by any programming background is pseudocode.

Hence we can say that the *purpose of writing pseudocode* is that it is easier for people to understand than any specific programming language code

No standard for pseudocode syntax exists, as a program in **pseudocode is not an executable program**.

Following are the basic rules before writing pseudocode :

- Write only one statement per line.
- Write what you mean, not how to program it
- Give proper indentation to show hierarchy and make code understandable.

- Make the program as simple as possible.
- Conditions and loops must be specified well ie. begun and ended explicitly as in given pseudocode examples :

**WRITE A PSEUDOCODE TO FIND THE SUM OF TWO NUMBERS:**



```
begin
 numeric nNum1,nNum2,nSum
 display "ENTER THE FIRST NUMBER : "
 accept nNum1
 display "ENTER THE SECOND NUMBER : "
 accept nNum2
 compute nSum=nNum1+nNum2
 display "SUM OF THESE NUMBER : " nSum
end
```

**WRITE A PSEUDOCODE TO FIND THE LARGEST OF TWO NUMBERS:**



BEGIN

```
 NUMERIC nNum1,nNum2
 DISPLAY "ENTER THE FIRST NUMBER : "
 INPUT nNum1

 DISPLAY "ENTER THE SECOND NUMBER : "
 INPUT nNum2

 IF nNum1 > nNum2
 DISPLAY nNum1 + " is larger than "+ nNum2
 ELSE
```

```
DISPLAY nNum2 + " is larger than " + nNum1
```

```
END
```

## **Types of Programming Languages**

### **Procedural Programming Language:**

The procedural programming language is used to execute a sequence of statements which lead to a result. Typically, this type of programming language uses multiple variables, heavy loops and other elements, which separates them from functional programming languages.

### **Functional Programming Language:**

Functional programming language typically uses stored data, frequently avoiding loops in favor of recursive functions. The functional programming's primary focus is on the return values of functions, and side effects and different suggests that storing state are powerfully discouraged. For example, in an exceedingly pure useful language, if a function is termed, it's expected that the function not modify or perform any o/p. It may, however, build algorithmic calls and alter the parameters of these calls.

### **Object-oriented Programming Language**

This programming language views the world as a group of objects that have internal data and external accessing parts of that data. The aim this programming language is to think about the fault by separating it into a collection of objects that offer services which can be used to solve a specific problem. One of the main principle of object oriented programming language is encapsulation that everything an object will need must be inside of the object.

### **Scripting Programming Language**

These programming languages are often procedural and may comprise object-oriented language elements, but they fall into their own category as they are normally not full-fledged programming languages with support for development of large systems. For example, they may not have compile-time type checking. Usually, these languages require tiny syntax to get started.

### **Logic Programming Language**

These types of languages let programmers make declarative statements and then allow the machine to reason about the consequences of those statements. In a sense, this language doesn't tell the computer how to do something, but employing restrictions on what it must consider doing.

## **The Difference Between Different Programming Languages**

### **C++ Language**

The C++ language has an object oriented structure which is used in large projects. Programmers can collaborate one program into different parts or even one individual work on each part of the program. The structure of object oriented also permit code to be reused many times. This language is an efficient language. But, many programmers will disagree



## C Language

The C language is a basic programming language and it is a very popular language, particularly used in game programming, Because C language includes the additional packing of the C++, Every programmer uses this language because it makes programs faster . However the value of this language gives the reusability of C++ to get the slight increase in performance with C language.



## Java Language

The Java language is a multi platform language that's particularly helpful in networking. Of course, mostly this language is used on the web with Java applets. However, this language is used to design cross platform programs, Since it similar to C++ in structure and syntax. For C++ programmers, Java language is very easy to learn and it offers some advantages provided by object oriented programming. Like reusability and it can be difficult to write efficient code in Java. But, nowadays the speed of the Java language has increased and 1.5 version offers some good features for easy program making.



### **PHP Language**

The PHP language is used to design web pages and sometimes it is also used as scripting language. This language is designed to develop a rapid website, and as a result comprises features which make it easy generate HTTP headers and link to databases. As a scripting language, it includes a set of components permit the programmer to easily get up to speed. However, it has more sophisticated object oriented features.



## **Intro to Java and Development setup**

### **What is Java?**

Java is a **programming language** and a **platform**. Java is a high level, robust, object-oriented and secure programming language.

Java was developed by *Sun Microsystems* (which is now the subsidiary of Oracle) in the year 1995. *James Gosling* is known as the father of Java. Before Java, its name was *Oak*. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

### **Application**

According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irctc.co.in, javatpoint.com, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games, etc.

### **Types of Java Applications**

There are mainly 4 types of applications that can be created using Java programming:

#### **1) Standalone Application**

Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

#### **2) Web Application**

An application that runs on the server side and creates a dynamic page is called a web application. Currently, [Servlet](#), [JSP](#), [Struts](#), [Spring](#), [Hibernate](#), [JSF](#), etc. technologies are used for creating web applications in Java.

#### **3) Enterprise Application**

An application that is distributed in nature, such as banking applications, etc. is called an enterprise application. It has advantages like high-level security, load balancing, and clustering. In Java, [EJB](#) is used for creating enterprise applications.

## 4) Mobile Application

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

### Setting up JAVA in windows

The Java Development Kit (**JDK**) is software used for Java programming, along with the Java Virtual Machine (**JVM**) and the Java Runtime Environment (**JRE**). The JDK includes the compiler and class libraries, allowing developers to create Java programs executable by the JVM and JRE.

**In this tutorial, you will learn to install the Java Development Kit on Windows.**

#### Check if Java Is Installed:

Before installing the Java Development Kit, check if a Java version is already installed on Windows. Follow the steps below:

1. Open a command prompt by typing *cmd* in the search bar and press **Enter**.
2. Run the following command:



```
java -version
```

```
C:\Users\boskom>java -version
'java' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\boskom>
```

The command outputs the Java version on your system. If Java isn't installed, the output is a message stating that Java isn't recognized as an internal or external command.

#### Download Java for Windows 10

Download the latest Java Development Kit installation file for Windows 10 to have the latest features and bug fixes.

1. Using your preferred web browser, navigate to the [Oracle Java Downloads page](#).
2. On the *Downloads* page, click the **x64 Installer** download link under the **Windows** category. At the time of writing this article, Java version 17 is the latest long-term support Java version.

| Linux                    | macOS     | Windows                                                                                                                                                            |
|--------------------------|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Product/file description | File size | Download                                                                                                                                                           |
| x64 Compressed Archive   | 170.66 MB | <a href="https://download.oracle.com/java/17/latest/jdk-17_windows-x64_bin.zip">https://download.oracle.com/java/17/latest/jdk-17_windows-x64_bin.zip</a> (sha256) |
| x64 Installer            | 152 MB    | <a href="https://download.oracle.com/java/17/latest/jdk-17_windows-x64_bin.exe">https://download.oracle.com/java/17/latest/jdk-17_windows-x64_bin.exe</a> (sha256) |
| x64 MSI Installer        | 150.89 MB | <a href="https://download.oracle.com/java/17/latest/jdk-17_windows-x64_bin.msi">https://download.oracle.com/java/17/latest/jdk-17_windows-x64_bin.msi</a> (sha256) |

## Install Java on Windows 10

After downloading the installation file, proceed with installing Java on your Windows system.

Follow the steps below:

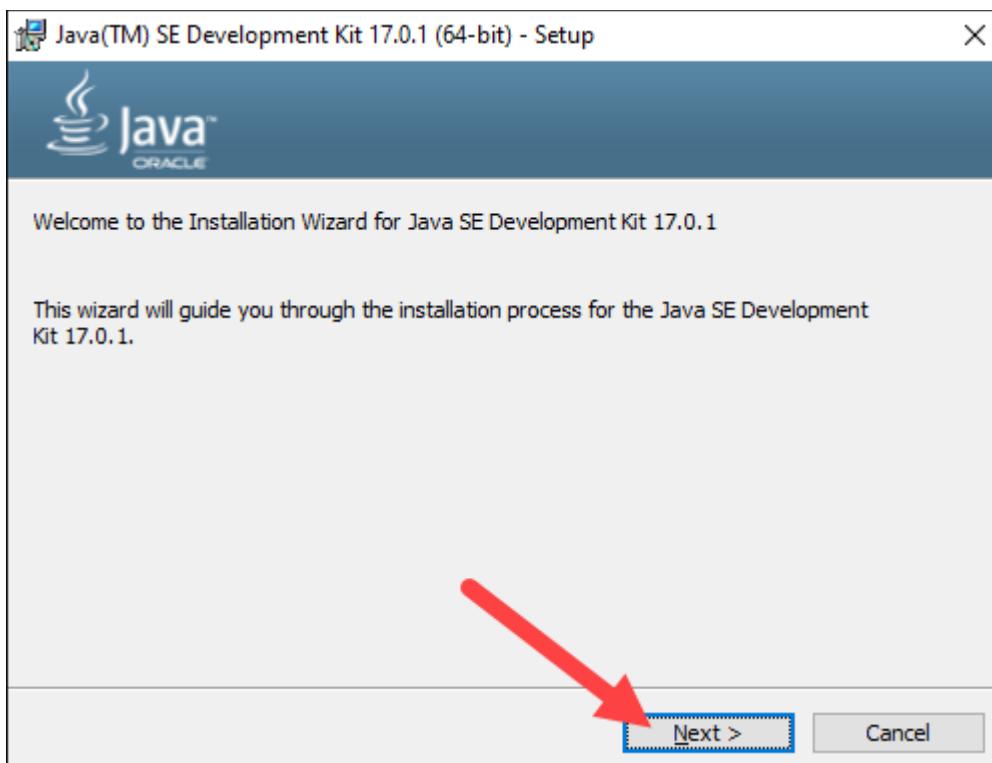
### Step 1: Run the Downloaded File

Double-click the **downloaded file** to start the installation.

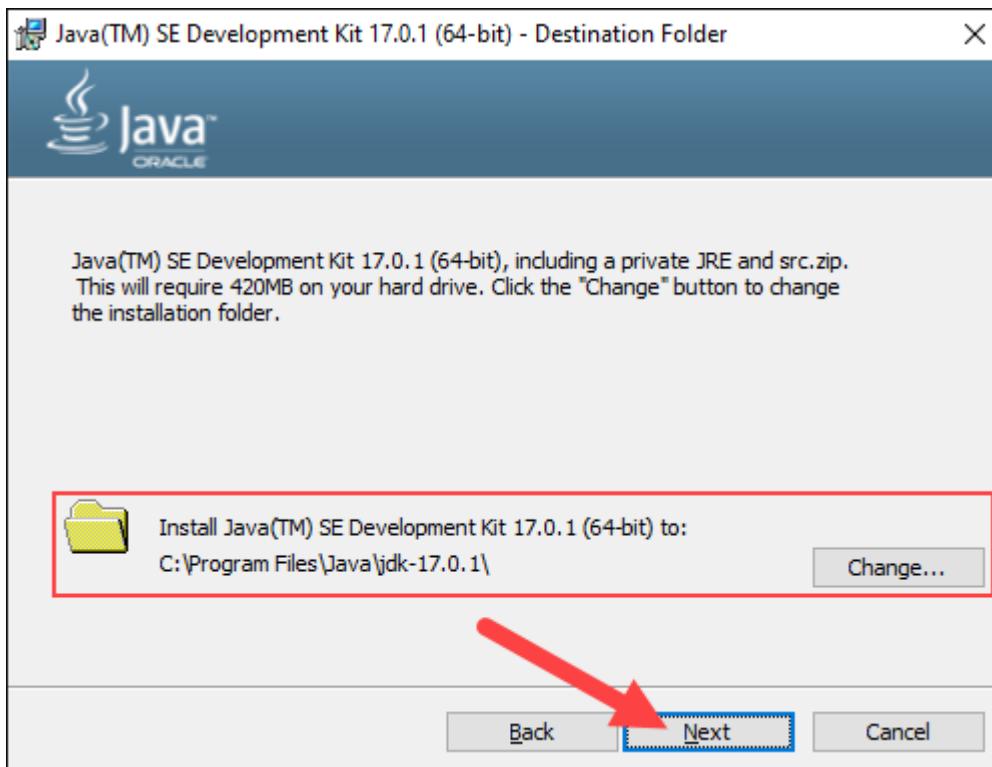
### Step 2: Configure the Installation Wizard

After running the installation file, the installation wizard welcome screen appears.

1. Click **Next** to proceed to the next step.



2. Choose the destination folder for the Java installation files or stick to the default path.  
Click Next to proceed.



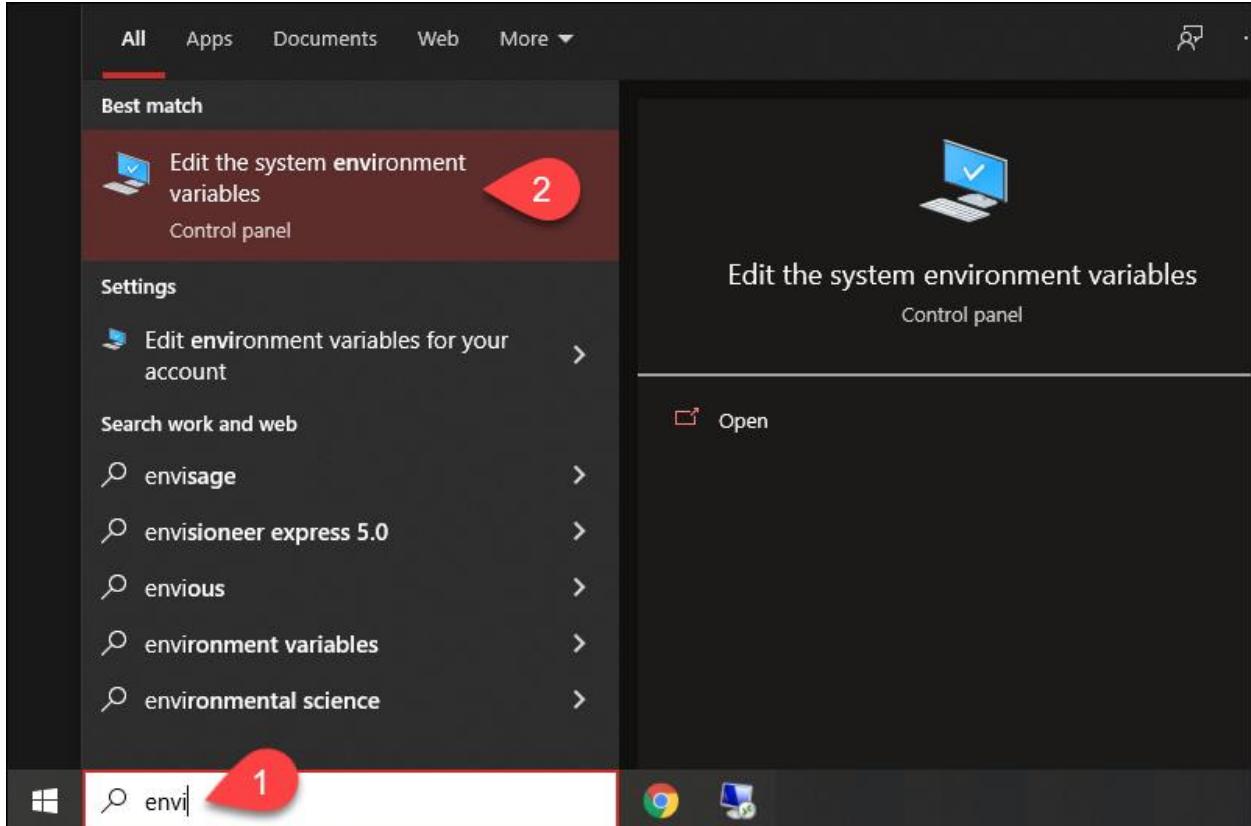
3. Wait for the wizard to finish the installation process until the Successfully Installed message appears. Click Close to exit the wizard.



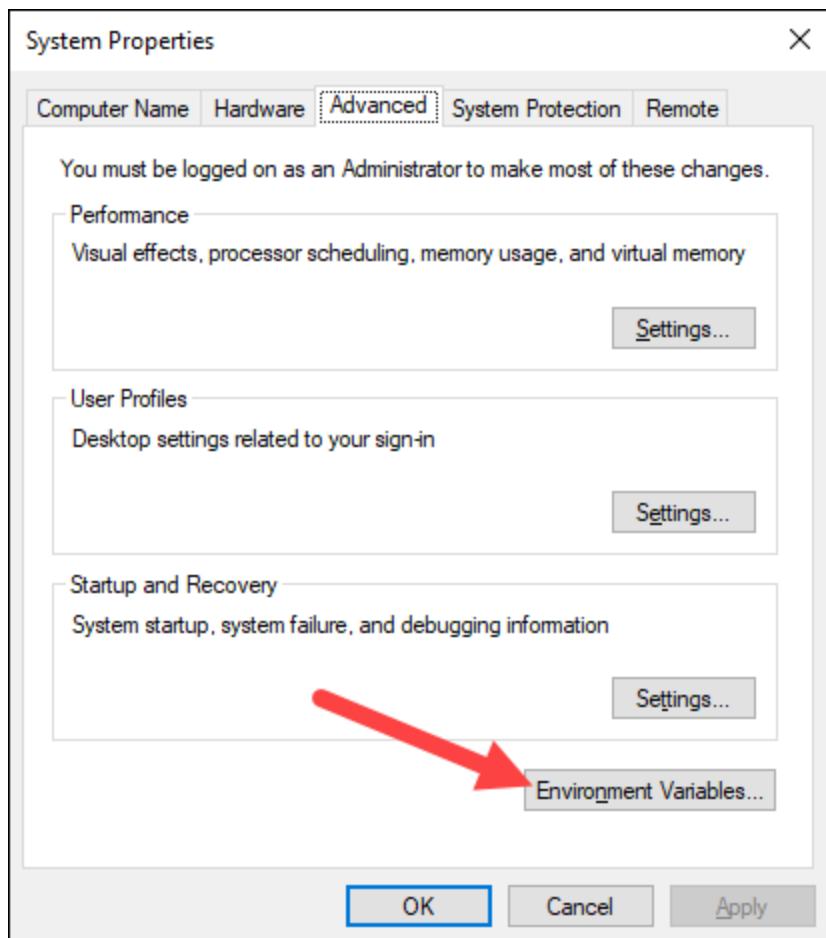
## Set Environmental Variables in Java

## Step 1: Add Java to System Variables

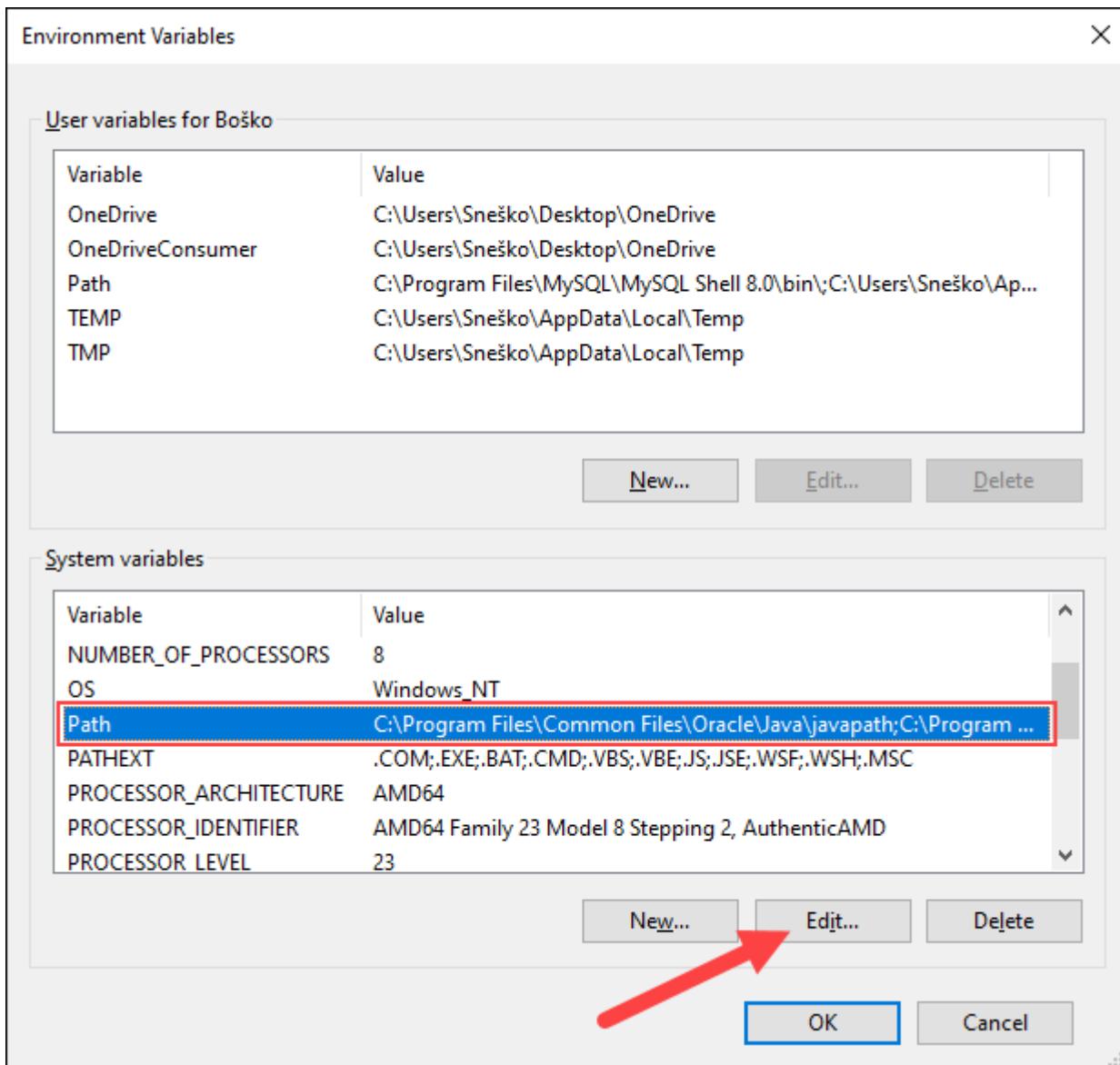
1. Open the **Start** menu and search for *environment variables*.
2. Select the **Edit the system environment variables** result.



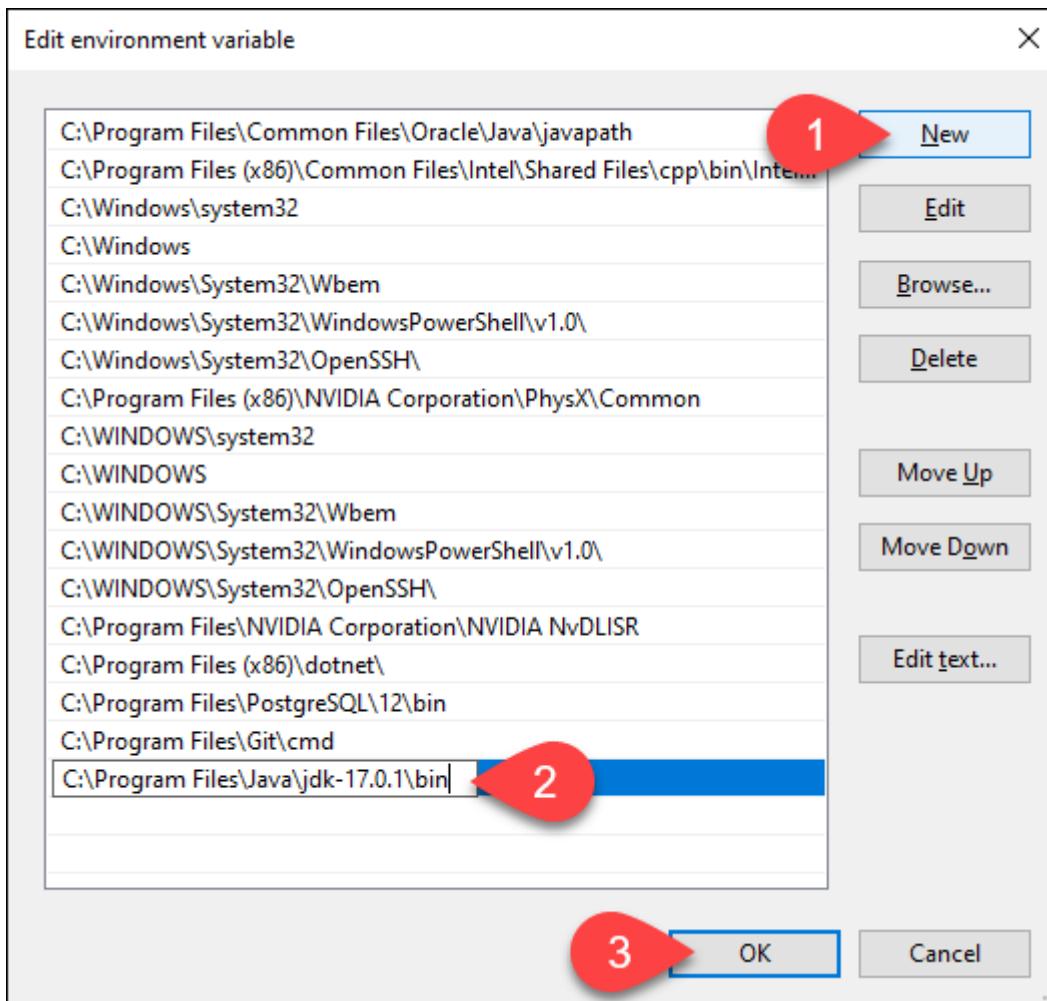
3. In the System Properties window, under the Advanced tab, click Environment Variables...



4. Under the System variables category, select the Path variable and click Edit:



5. Click the New button and enter the path to the Java bin directory:



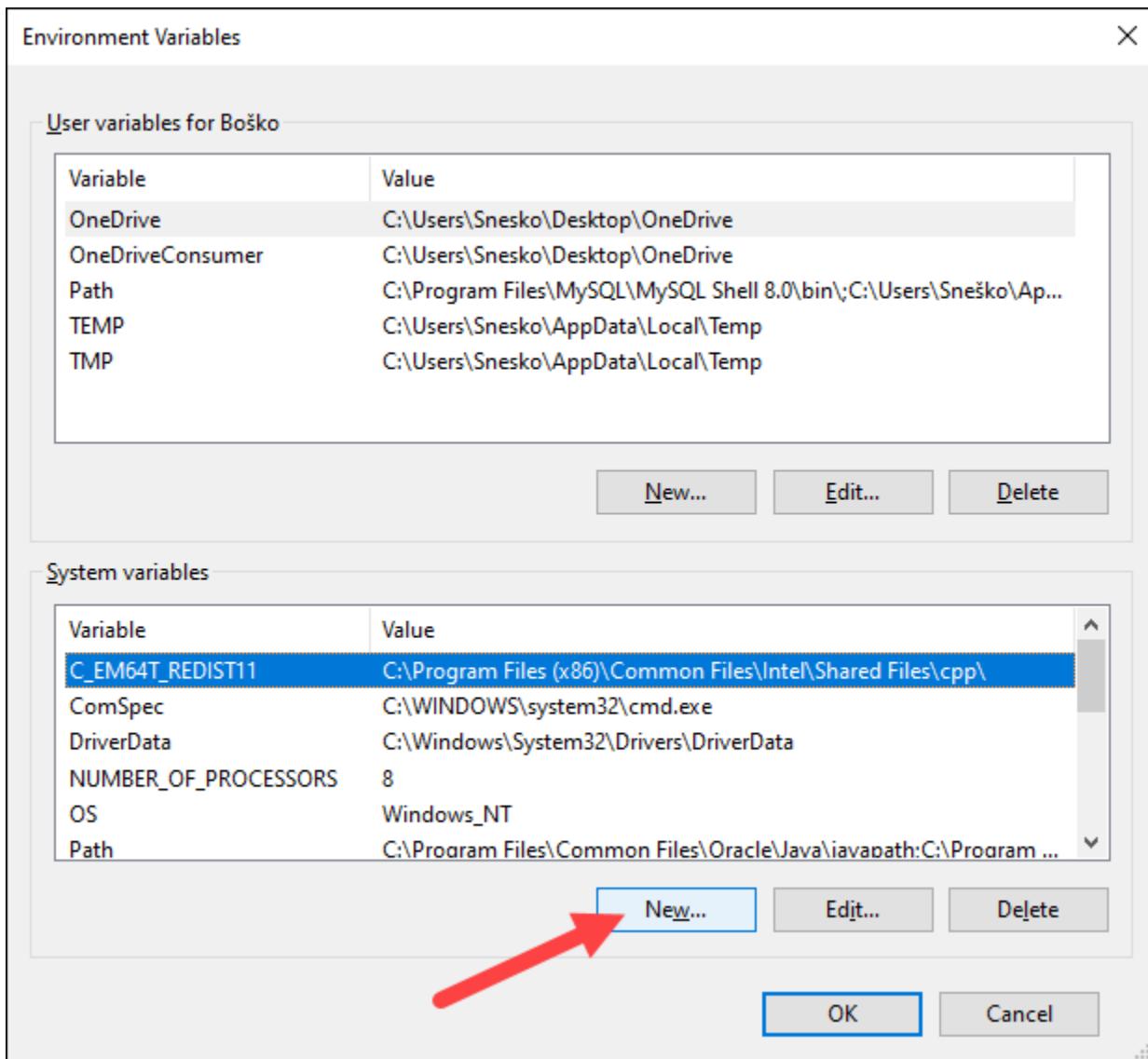
**Note:** The default path is usually `C:\Program Files\Java\jdk-17.0.1\bin`.

6. Click **OK** to save the changes and exit the variable editing window.

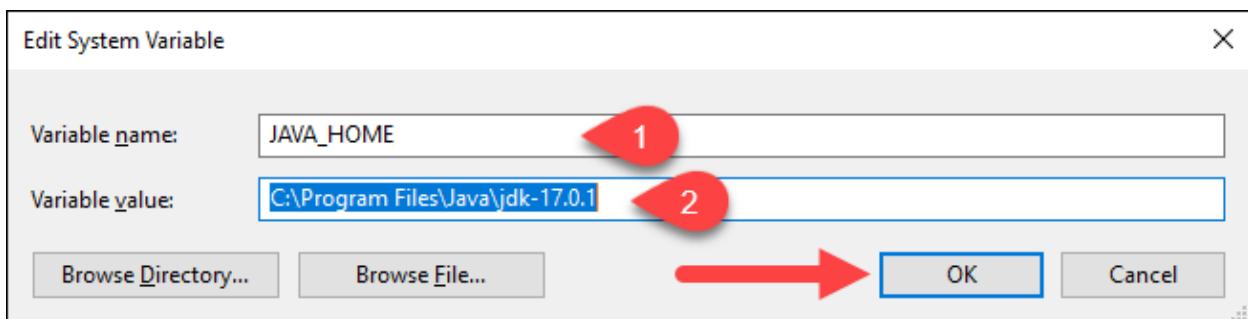
## Step 2: Add JAVA\_HOME Variable

Some applications require the `JAVA_HOME` variable. Follow the steps below to create the variable:

1. In the *Environment Variables* window, under the *System variables* category, click the **New...** button to create a new variable.



2. Name the variable as **JAVA\_HOME**.
3. In the variable value field, paste the path to your Java jdk directory and click **OK**.



4. Confirm the changes by clicking **OK** in the *Environment Variables* and *System properties* windows.

## Test the Java Installation

Run the **java -version** command in the command prompt to make sure Java installed correctly:

```
C:\Users\boskom>java -version
java version "17.0.1" 2021-10-19 LTS
Java(TM) SE Runtime Environment (build 17.0.1+12-LTS-39)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.1+12-LTS-39, mixed mode, sharing)
```

If installed correctly, the command outputs the Java version. Make sure everything works by writing a simple program and compiling it. Follow the steps below:

## Setting up IntelliJ IDEA for JAVA

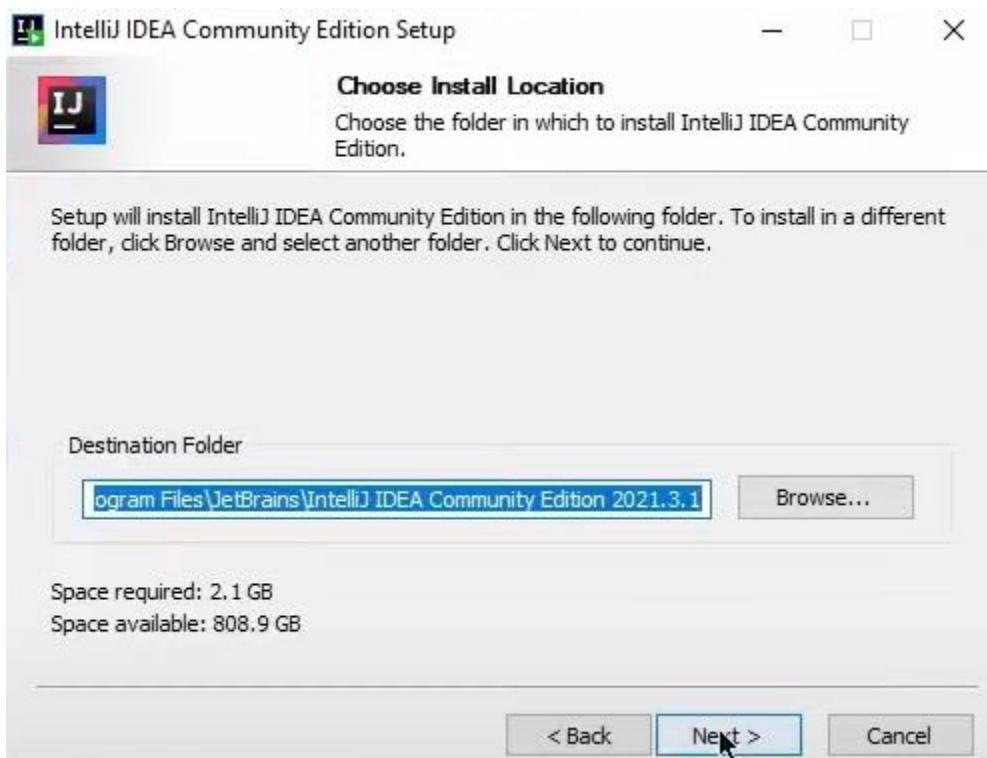
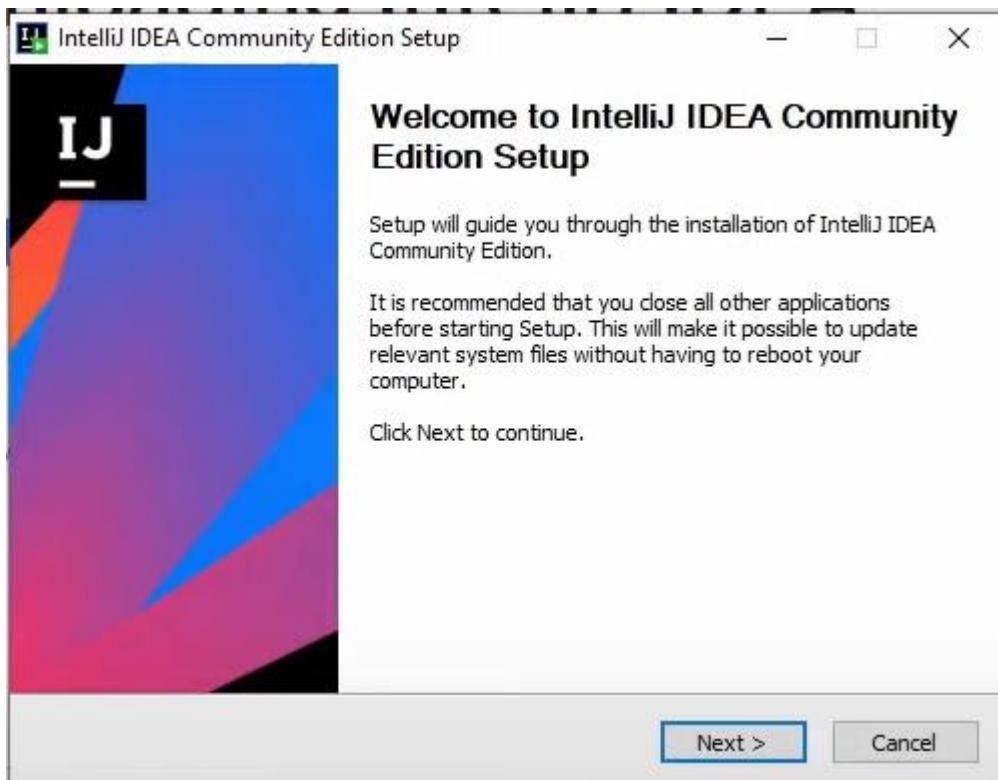
IntelliJ IDEA is an intelligent, context-aware IDE for **working with Java and other JVM languages like Kotlin, Scala, and Groovy on all sorts of applications.**

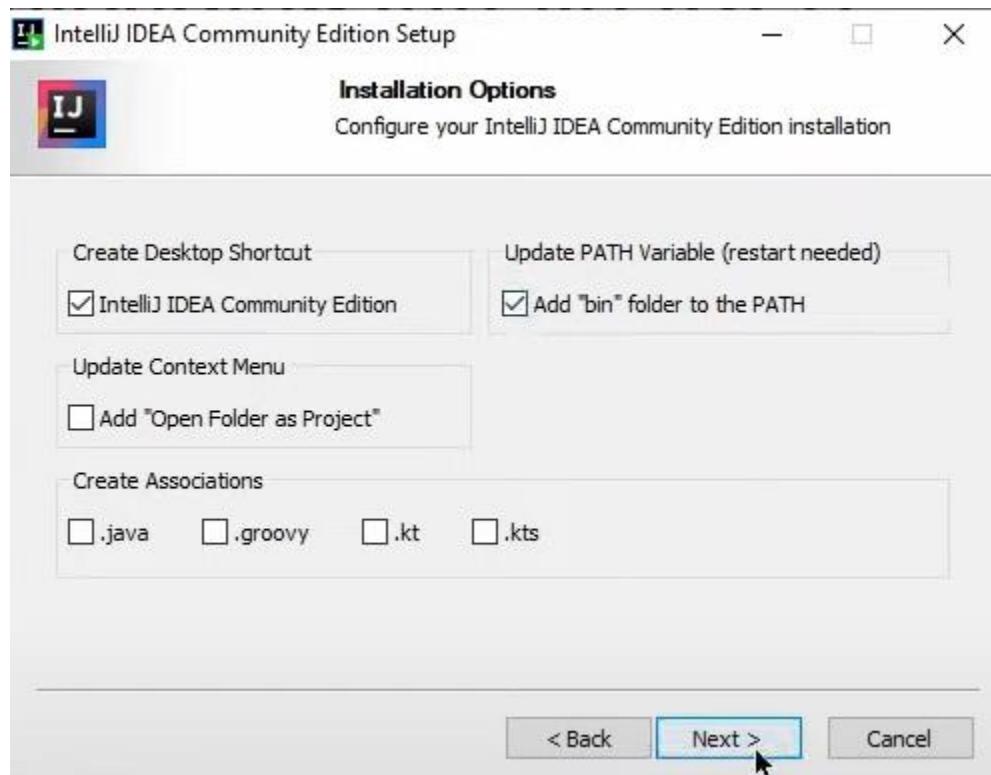
Download & Install Community version of IntelliJ Idea from here

- <https://www.jetbrains.com/idea/download/#section=windows>

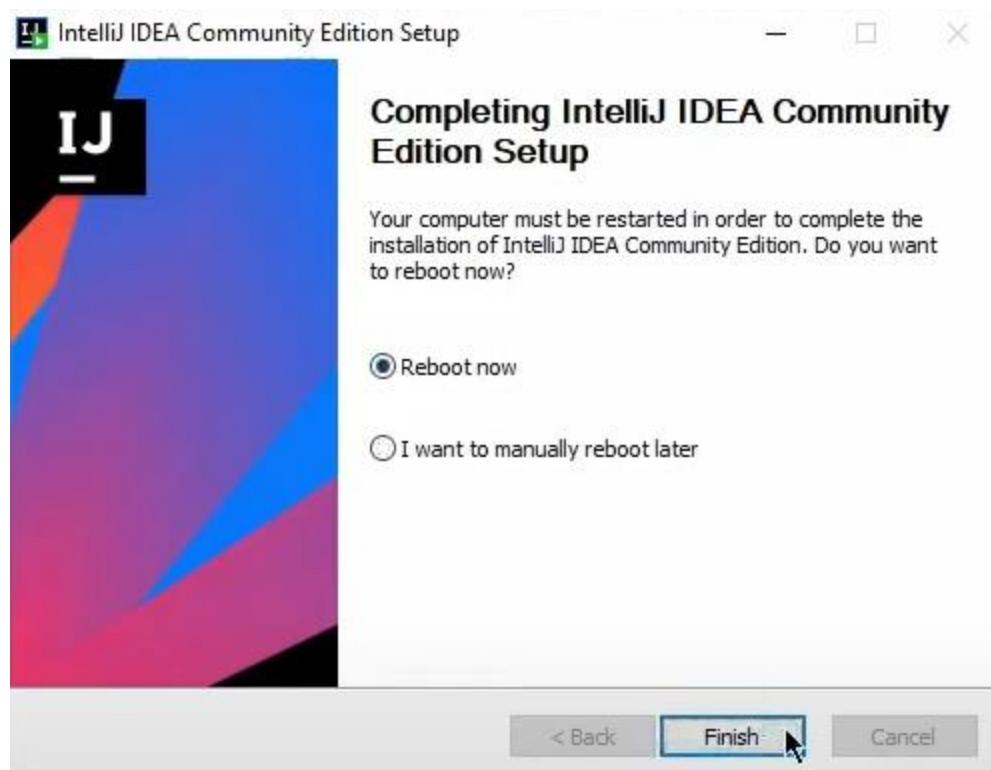
The screenshot shows the IntelliJ IDEA download page. At the top, there's a navigation bar with links for 'What's New', 'Features', 'Resources', 'Pricing' (which is highlighted), and a 'Download' button. Below the navigation, there's a large 'Download IntelliJ IDEA' section. It features two main download options: 'Ultimate' (for web and enterprise development) and 'Community' (for JVM and Android development). Both options have a 'Download' button and a dropdown menu for file type (.exe). Below each download section, there's a note: 'Free 30-day trial available' for the Ultimate edition and 'Free, built on open source' for the Community edition. To the left of the download sections, there's a logo for IntelliJ IDEA and some release information: Version: 2022.2.2, Build: 222.4167.29, 14 September 2022. Below that, there are links for 'Release notes', 'System requirements', 'Installation instructions', 'Other versions', and 'Third-party software'. A comparison table follows, showing which features are available in both editions. The table has three columns: 'IntelliJ IDEA Ultimate', 'IntelliJ IDEA Community Edition', and a column for 'Differences'. The features listed are Java, Kotlin, Groovy, Scala; Maven, Gradle, sbt; Git, GitHub, SVN, Mercurial, Perforce; and Databases.

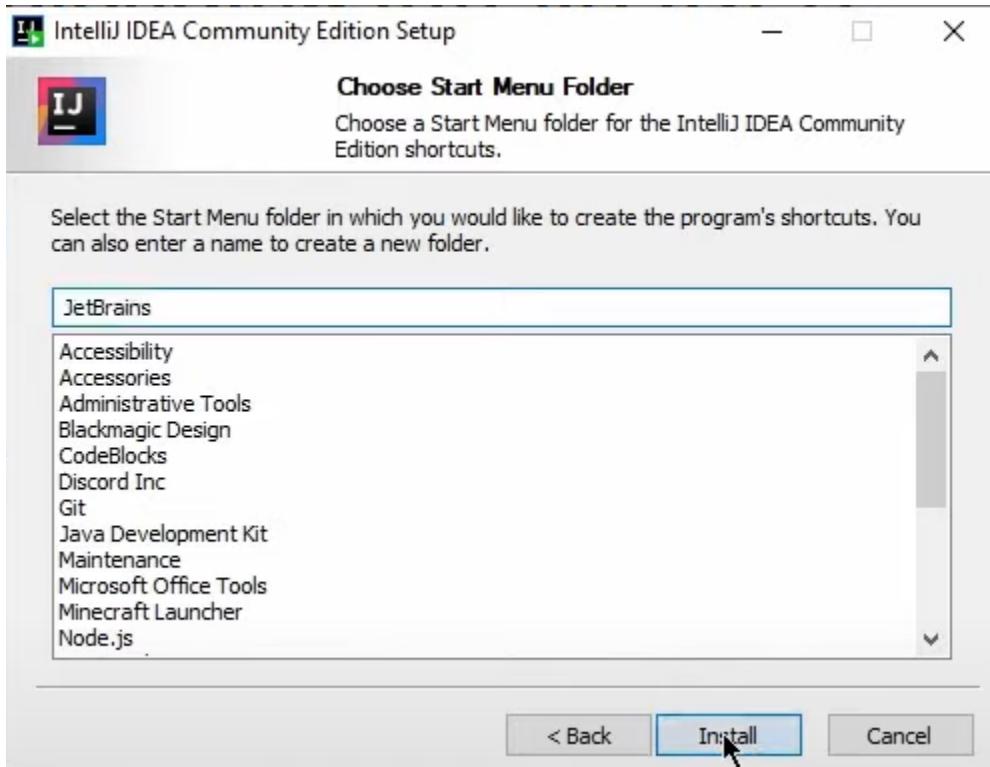
|                                       | IntelliJ IDEA Ultimate | IntelliJ IDEA Community Edition | Differences |
|---------------------------------------|------------------------|---------------------------------|-------------|
| Java, Kotlin, Groovy, Scala           | ✓                      | ✓                               |             |
| Maven, Gradle, sbt                    | ✓                      | ✓                               |             |
| Git, GitHub, SVN, Mercurial, Perforce | ✓                      | ✓                               |             |
| Databases                             | ✓                      | ✓                               |             |



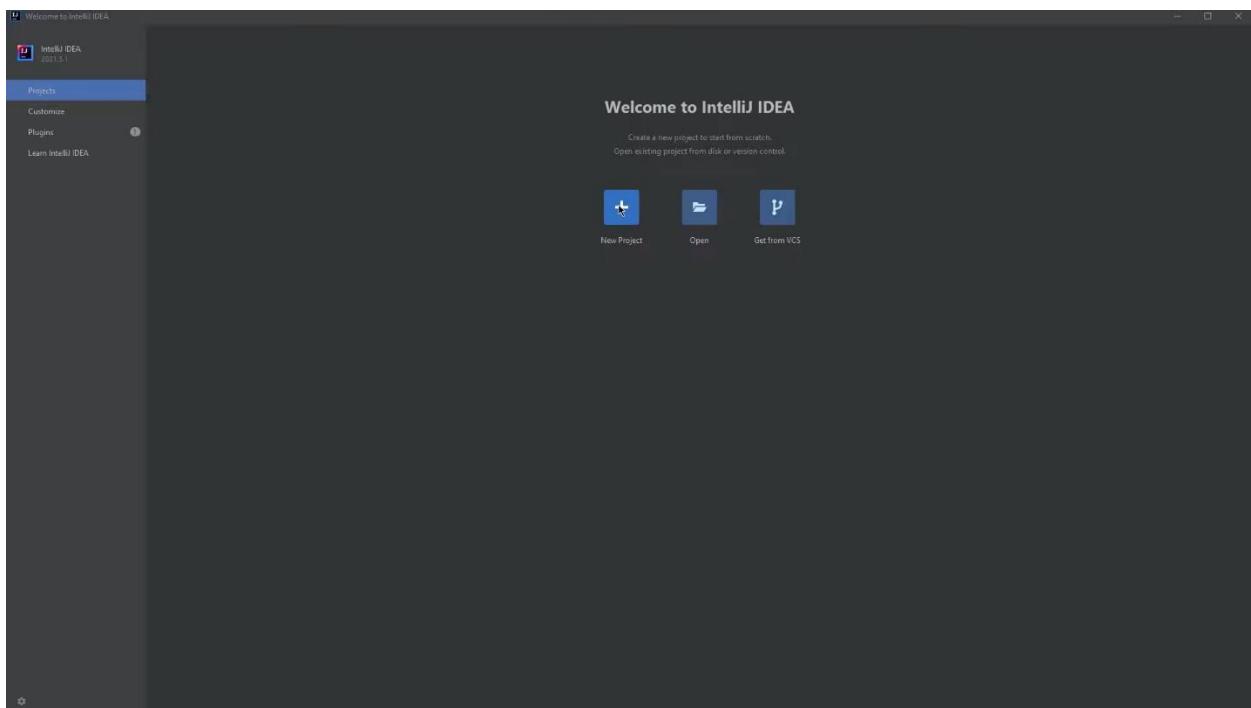


Click the option to update path variable.

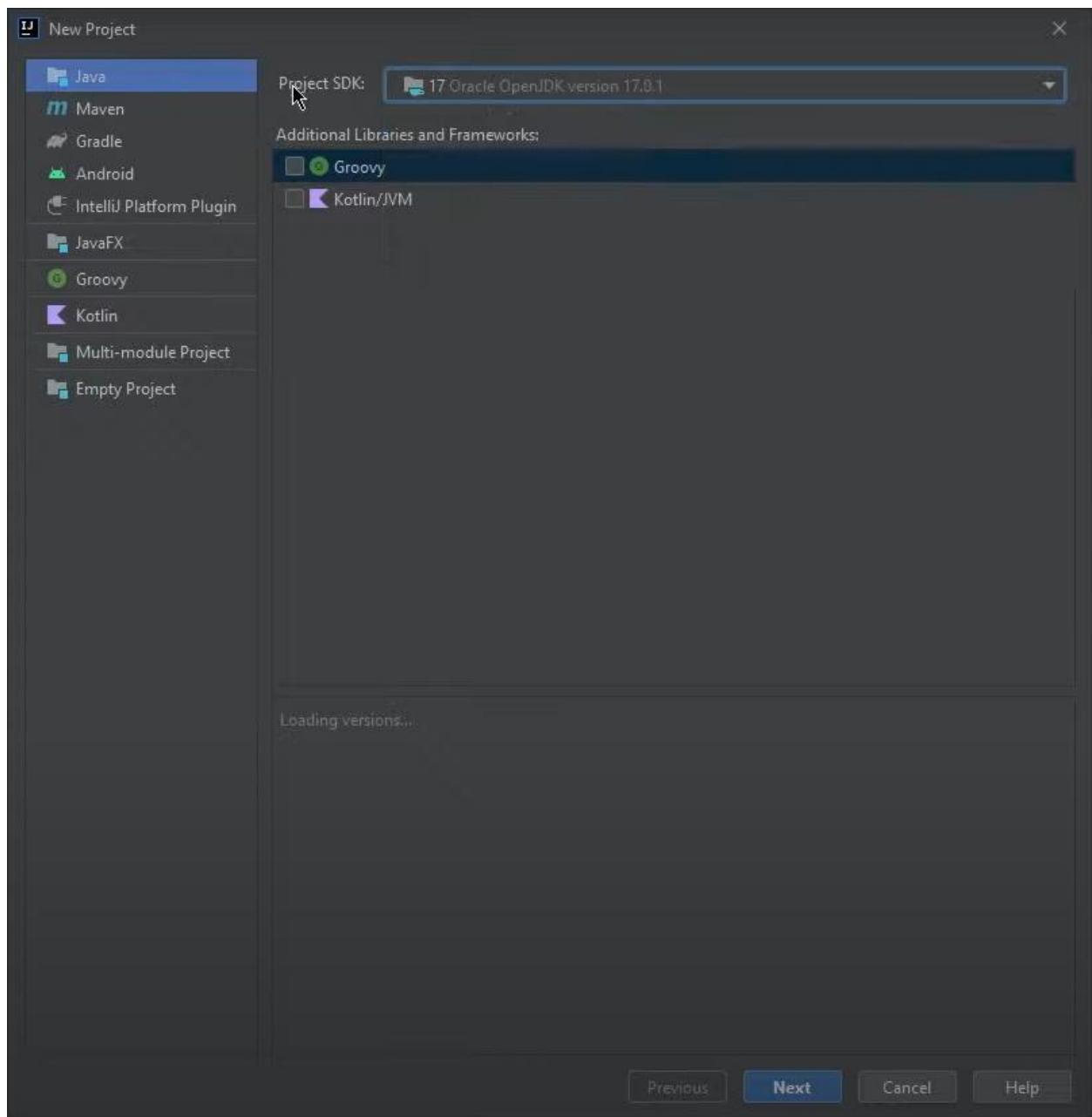




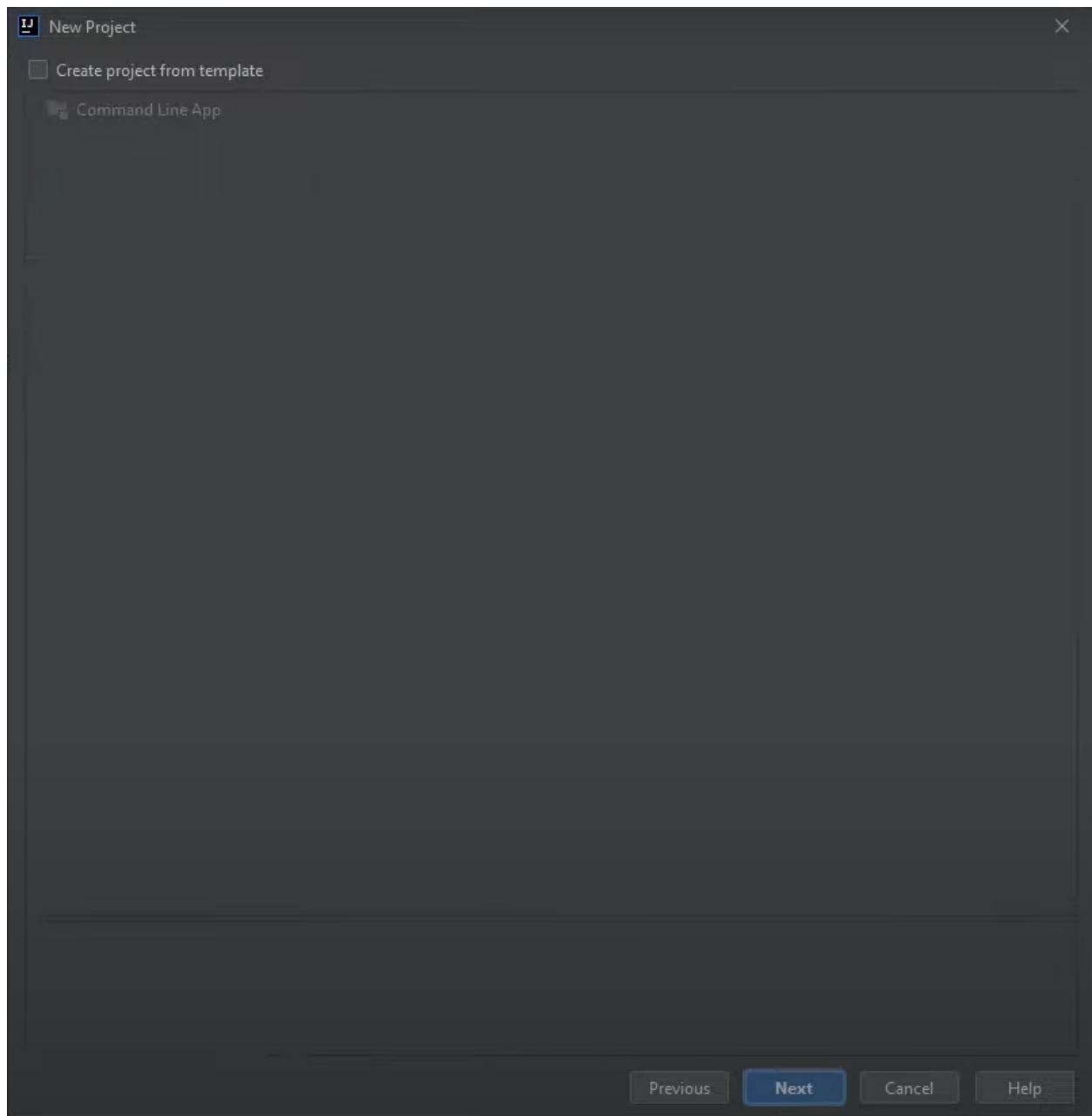
After successful installation click open the app and click on New Project.



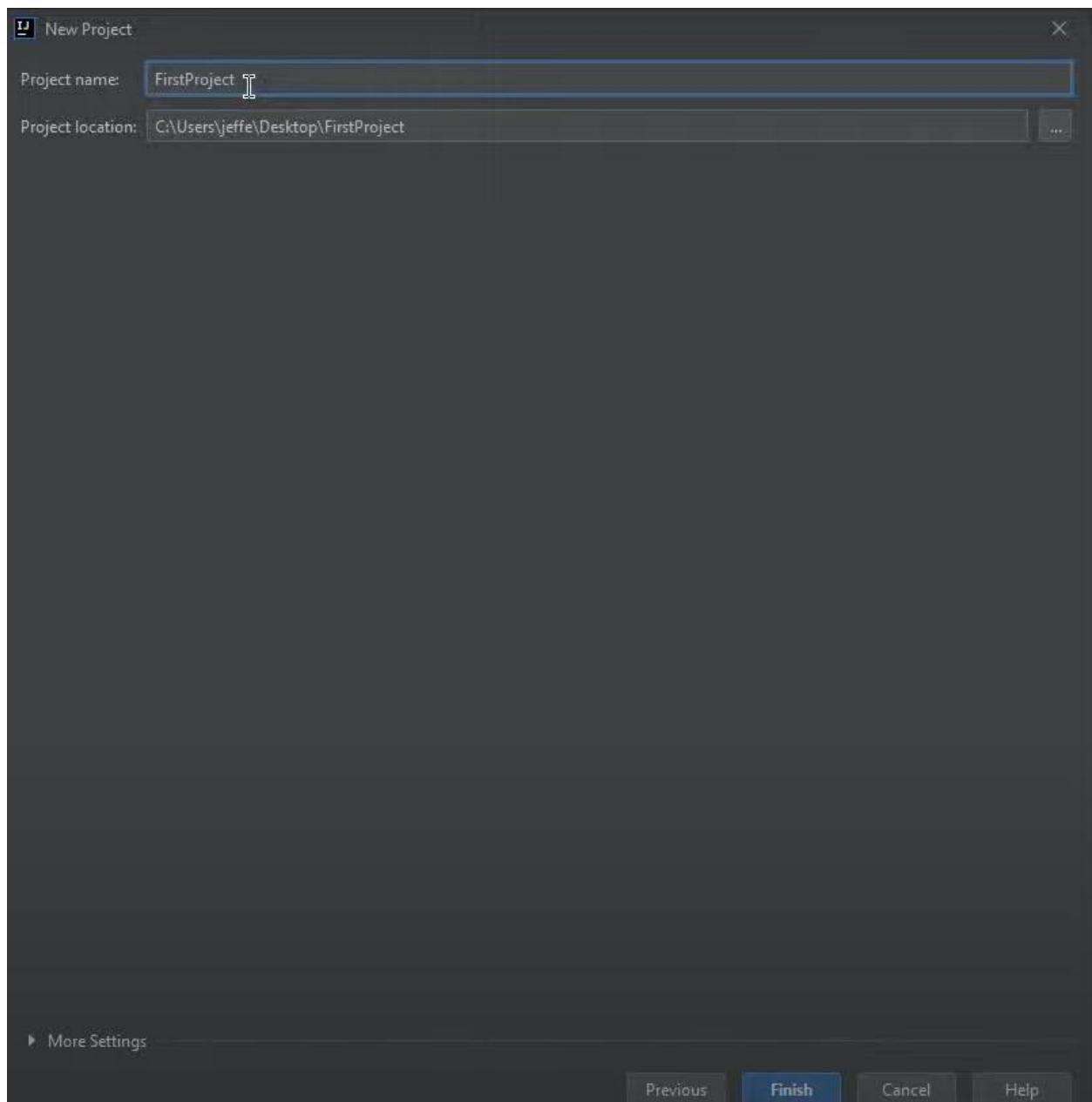
Select Java from sidebar if not already selected. You will see your java-jdk automatically selected in the dropdown. If not then choose the path to your java jdk you downloaded.



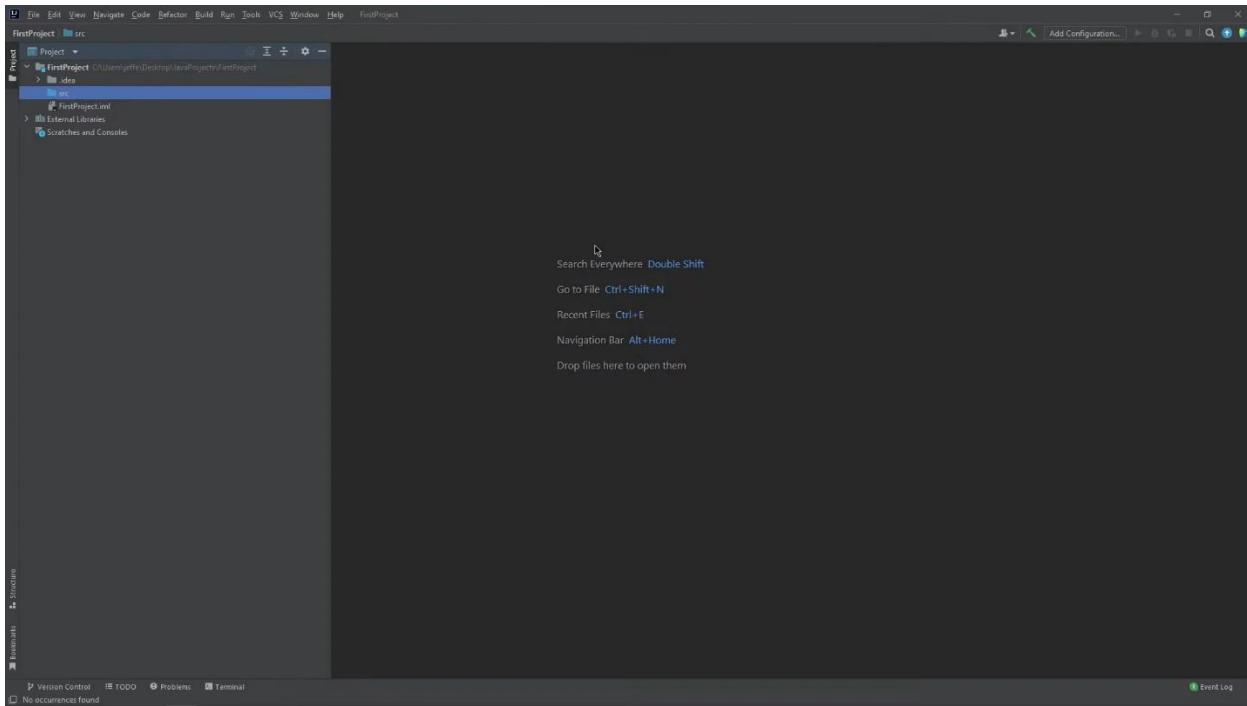
Click next.



Click next.



Give a name to your project and click finish. DONE!!



## Conclusion

In this module we went through basics of programming and completed java setup in our machines.

In further modules we will dive deep into java and its features.

## Interview Questions

Please explain an algorithm. What are some of its important features?

An algorithm can be defined as a set of finite steps that when followed helps in accomplishing a particular task. Important features of an algorithm are clarity, efficiency, and finiteness.

Explain low-level and high-level programming languages. Also, give some examples.

Any programming language that offers no generalization from the computer's instruction set architecture is a low-level programming language. Assembly language and machine language are two typical examples of low-level programming languages.

A programming language that offers high generalization from the computer's instruction set architecture is termed a high-level programming language. Typically, a high-level programming language has elements resembling natural language to make program development easier.

Another definition of a high-level programming language is one that is independent of the underlying processor of the system in which it is running. C++, Java, and Python are some of the most popular high-level programming languages.

Thank You !



## Agenda

In this session we'll be covering:

- Intro to Arrays
- Defining Arrays
- Array initialization
- Looping on Arrays

Computers get a lot of their power from working with data structures. A data structure is an organized collection of related data. An object is a data structure, but this type of data structure—consisting of a fairly small number of named instance variables—is just the beginning. In many cases, programmers build complicated data structures by hand, by linking objects together. But there is one type of data structure that is so important and so basic that it is built into every programming language: the array.

An array is a data structure consisting of a numbered list of items, where all the items are of the same type. In Java, the items in an array are always numbered from zero up to some maximum value, which is set when the array is created. For example, an array might contain 100 integers, numbered from zero to 99. The items in an array can belong to one of Java's primitive types. They can also be references to objects, so that you could, for example, make an array containing all the buttons in a GUI program.

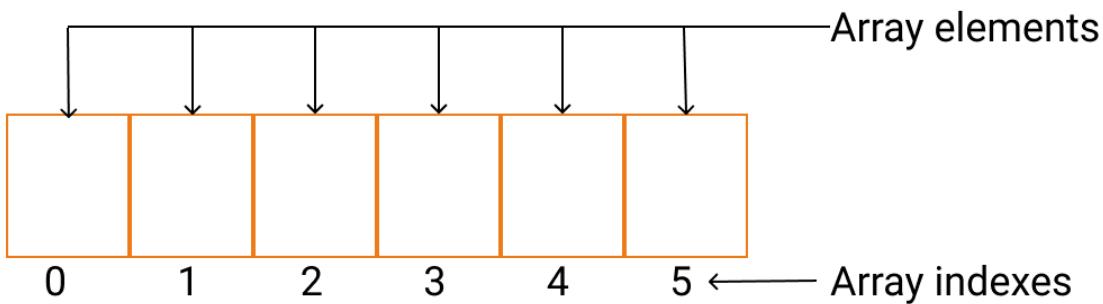
### Creating and Using Arrays

When a number of data items are chunked together into a unit, the result is a data structure. Data structures can be very complex, but in many applications, the appropriate data structure consists simply of a sequence of data items. An array is a data structure of this simple variety.

An array is a sequence of items. These items in an array are numbered, and individual items are referred to by their position number. Furthermore, all the items in an array must be of the same type. The definition of an array is: a numbered sequence of items, which are all of the same type. The number of items in an array is called the **length** of the array. The position number of an item in an array is called the **index** of that item. The type of the individual items in an array is called the **base type** of the array.

The base type of an array can be any Java type, that is, one of the primitive types, or a class name, or an interface name. If the base type of an array is int, it is referred to as an “array of ints.” An array with base type String is referred to as an “array of Strings.” However, an array is not, properly speaking, a list of integers or strings or other values. It is better thought of as a list of variables of type int, or of type String, or of some other type. As always, there is some potential for confusion between the two uses of a variable: as a name for a memory location and as a name for the value stored in that memory location. Each position in an array acts as a variable. Each position can hold a value of a specified type (the base type of the array). The value can be changed at any time. Values are stored in an array.

The items in an array—really, the individual variables that make up the array—are more often referred to as the elements of the array. In Java, the elements in an array are always numbered starting from zero. That is, the index of the first element in the array is zero. If the length of the array is N, then the index of the last element in the array is N-1. Once an array has been created, its length cannot be changed.



## Using Arrays

Suppose that A is a variable that refers to an array. Then the element at index k in A is referred to as A[k]. The first element is A[0], the second is A[1], and so forth. “A[k]” is really a variable, and it can be used just like any other variable. You can assign values to it, you can use it in expressions, and you can pass it as a parameter to a subroutine. All of this will be discussed in more detail below. For now, just keep in mind the syntax :



`<*array variable>* ***[* *<integer expression>* ***]**`

for referring to an element in an array.

Although every array, as an object, belongs to some class, array classes never have to be defined. Once a type exists, the corresponding array class exists automatically. If the name of the type is BaseType, then the name of the associated array class is BaseType[ ]. That is to say, an object belonging to the class BaseType[ ] is an array of items, where each item is a variable of type BaseType. The brackets, [], are meant to recall the syntax for referring to the individual items in the array. BaseType[ ] is read as “array of BaseType” or “BaseType array.” It might be worth mentioning here that if ClassA is a subclass of ClassB, then the class ClassA[ ] is automatically a subclass of ClassB[ ]. The base type of an array can be any legal Java type. From the primitive type int, the array type int[ ] is derived. Each element in an array of type int[ ] is a variable of type int, which holds a value of type int. From a class named Shape, the array type Shape[ ] is derived. Each item in an array of type Shape[ ] is a variable of type Shape, which holds a value of type Shape. This value can be either null or a reference to an object belonging to the class Shape. (This includes objects belonging to subclasses of Shape)

Let’s try to get a little more concrete about all this, using arrays of integers as our first example. Since int[] is a class, it can be used to declare variables. For example,



`int[] list;`

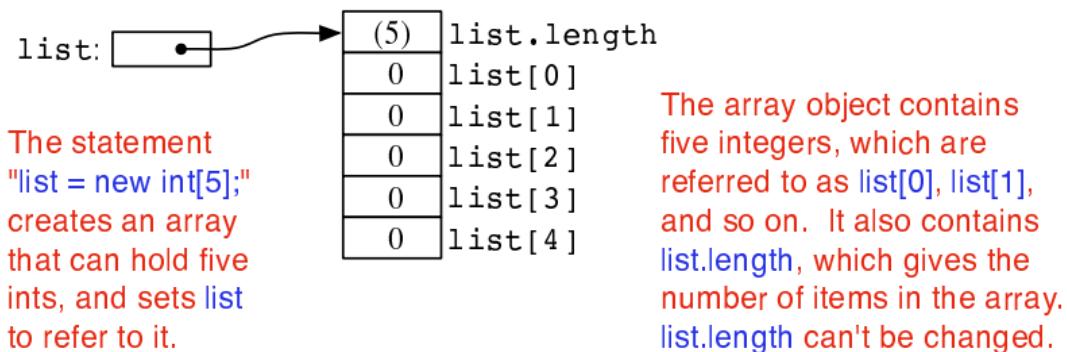
creates a variable named list of type int[]. This variable is capable of referring to an array of ints, but initially its value is null (if list is a member variable in a class) or undefined (if list is a local variable in a method). The new operator is used to create a new array object, which can then be assigned to list. The syntax for using new with arrays is different from the syntax you learned previously. As an example,



```
list = new int[5];
```

creates an array of five integers. More generally, the constructor new BaseType[N] is used to create an array belonging to the class BaseType[]. The value N in brackets specifies the length of the array, that is, the number of elements that it contains. Note that the array “knows” how long it is. The length of the array is an instance variable in the array object.

The situation produced by the statement list = new int[5] can be pictured like:



The elements in the array, list, are referred to as list[0], list[1], list[2], list[3] and list[4].

Note that the newly created array of integers is automatically filled with zeros. In Java, a newly created array is always filled with a known, default value: zero for numbers, false for boolean, the character with Unicode number zero for char, and null for objects.

The brackets in an array reference can contain any expression whose value is an integer. For example if indx is a variable of type int, then list[indx] and list[2\*indx+7] are syntactically correct references to elements of the array list. Thus, the following loop would print all the integers in the array, list, to standard output:



```
for(int i = 0; i < list.length; i++){
 System.out.println(list[i]);
}
```

The first time through the loop, *i* is 0, and *list[i]* refers to *list[0]*. So, it is the value stored in the variable *list[0]* that is printed. The second time through the loop, *i* is 1, and the value stored in *list[1]* is printed. The loop ends after printing the value of *list[4]*, when *i* becomes equal to 5 and the continuation condition *i < list.length* is no longer true. This is a typical example of using a loop to process an array.

Every use of a variable in a program specifies a memory location. Think for a moment about what the computer does when it encounters a reference to an array element, *list[k]*, while it is executing a program. The computer must determine which memory location is being referred to. To the computer, *list[k]* means something like this:

- Get the pointer that is stored in the variable, *list*
- Follow this pointer to find an array object
- Get the value of *k*
- Go to the *k*-th position in the array, this is the required location

There are two things that can go wrong here,

1. Suppose that the value of *list* is null. If that is the case, then *list* doesn't even refer to an array. The attempt to refer to an element of an array that doesn't exist is an error that will cause an exception of type `NullPointerException` to be thrown.
2. The second possible error occurs if *list* does refer to an array, but the value of *k* is outside the legal range of indices for that array. This will happen if *k < 0* or if *k >= list.length*. This is called an "array index out of bounds" error. When an error of this type occurs, an exception of type `ArrayIndexOutOfBoundsException` is thrown.

When you use arrays in a program, you should be mindful that both types of errors are possible.

### Array Initialization

For an array variable, just as for any variable, you can declare the variable and initialize it in a single step. For example,



```
int[] list = new int[5];
```

If *list* is a local variable in a subroutine, then this is exactly equivalent to the two statements:



```
int[] list;
```

```
list = new int[5];
```

The new array is filled with the default value appropriate for the base type of the array—zero for `int` and null for class types, for example. However, Java also provides a way to initialize an array variable with a

new array filled with a specified list of values. In a declaration statement that creates a new array, this is done with an array initializer. For example,



```
int[] list = { 1, 4, 9, 16, 25, 36, 49 };
```

creates a new array containing the seven values 1, 4, 9, 16, 25, 36, and 49, and sets list to refer to that new array. The value of list[0] will be 1, the value of list[1] will be 4, and so forth. The length of list is seven, since seven values are provided in the initializer.

A list initializer of this form can be used only in a declaration statement, to give an initial value to a newly declared array variable. It cannot be used in an assignment statement to assign a value to a variable that has been previously declared.

However, there is another, similar notation for creating a new array that can be used in an assignment statement or passed as a parameter to a subroutine. The notation uses another form of the new operator to both create and initialize a new array object at the same time. For example to assign a new value to an array variable, list, that was declared previously, you could use:



```
list = new int[] { 1, 8, 27, 64, 125, 216, 343 };
```

## Programming with Arrays

### Arrays and for loops

In many cases, processing an array means applying the same operation to each item in the array. This is commonly done with a for loop. A loop for processing all the elements of an array A has the form:



```
// do any necessary initialization
for (int i = 0; i < A.length; i++) {
 ... // process A[i]
}
```

Suppose, for example, that A is an array of type double[ ]. Suppose that the goal is to add up all the numbers in the array. The pseudo-code for this type of an algorithm would be:



Start with 0;

Add A[0]; (process the first item in A)

Add A[1]; (process the second item in A)

.

.

Add A[ A.length - 1 ]; (process the last item in A)

Putting the obvious repetition into a loop and giving a name to the sum, this becomes:



```
double sum; // The sum of the numbers in A.
```

```
sum = 0; // Start with 0.
```

```
for (int i = 0; i < A.length; i++)
```

```
 sum += A[i]; // add A[i] to the sum, for
```

```
// i = 0, 1, ..., A.length - 1
```

Note that the continuation condition,  $i < A.length$ , implies that the last value of  $i$  that is actually processed is  $A.length-1$ , which is the index of the final item in the array. It's important to use  $<$  here, not  $\leq$ , since  $\leq$  would give an array index out of bounds error. There is no element at position  $A.length$  in  $A$ .

Here is another example of a loop that will count the number of items in Array A which are less than zero:



```
int count; // For counting the items.
```

```
count = 0; // Start with 0 items counted.
```

```
for (int i = 0; i < A.length; i++) {
```

```
 if (A[i] < 0.0) // if this item is less than zero...
```

```
 count++; // ...then count it
```

```
}
```

```
// At this point, the value of count is the number
```

```
// of items that have passed the test of being < 0
```

Here is a variation on the same theme. Suppose you want to count the number of times that an item in the array A is equal to the item that follows it. The item that follows  $A[i]$  in the array is  $A[i+1]$ , so the test in this case is if ( $A[i] == A[i+1]$ ). But there is a catch: This test cannot be applied when  $A[i]$  is the last item

in the array, since then there is no such item as A[i+1]. The result of trying to apply the test in this case would be an `ArrayIndexOutOfBoundsException`. This just means that we have to stop one item short of the final item:



```
int count = 0;
for (int i = 0; i < A.length - 1; i++) {
 if (A[i] == A[i+1])
 count++;
}
```

### Arrays and for-each loop

Java 5.0 introduced a new form of the for loop, the “for-each loop”. The for-each loop is meant specifically for processing all the values in a data structure. When used to process an array, a for-each loop can be used to perform the same operation on each value that is stored in the array. If `anArray` is an array of type `BaseType[ ]`, then a for-each loop for `anArray` has the form:



```
for (BaseType item : anArray) {
 . . .
 . // process the item
 . . .
}
```

In this loop, `item` is the loop control variable. It is being declared as a variable of type `BaseType`, where `BaseType` is the base type of the array. (In a for-each loop, the loop control variable must be declared in the loop.) When this loop is executed, each value from the array is assigned to `item` in turn and the body of the loop is executed for each value. Thus, the above loop is exactly equivalent to:



```
for (int index = 0; index < anArray.length; index++) {
 BaseType item;
 item = anArray[index]; // Get one of the values from the array
 . . .
 . // process the item
```

```
}
```

For example, if A is an array of type int[], then we could print all the values from A with the for-each loop:



```
for (int item : A)
 System.out.println(item);
```

And we could add up all the positive integers in A with



```
int sum = 0; // This will be the sum of all the positive numbers in A
for (int item : A) {
 if (item > 0)
 sum = sum + item;
}
```

The for-each loop is not always appropriate. For example, there is no simple way to use it to process the items in just a part of an array. However, it does make it a little easier to process all the values in an array, since it eliminates any need to use array indices.

It's important to note that a for-each loop processes the values in the array, not the elements (where an element means the actual memory location that is part of the array). For example, consider the following incorrect attempt to fill an array of integers with 17's:



```
int[] intList = new int[10];
for (int item : intList) { // INCORRECT! DOES NOT MODIFY THE ARRAY!
 item = 17;
}
```

The assignment statement item = 17 assigns the value 17 to the loop control variable, item. However, this has nothing to do with the array. When the body of the loop is executed, the value from one of the elements of the array is copied into item. The statement item = 17 replaces that copied value but has no effect on the array element from which it was copied; the value in the array is not changed.

## Conclusion

In this session we've covered :

- Intro to Arrays
- Creating and Using Arrays
- Programming with Arrays

In the next session, we'll dive deeper into how arrays are used to solve programming problems.

### **Interview Questions**

Mention some advantages and disadvantages of Arrays.

#### **Advantages:**

- Multiple elements of Array can be sorted at the same time.
- Using the index, we can access any element in O(1) time.

#### **Disadvantages:**

- You need to specify how many elements you're going to store in your array ahead of time and We can not increase or decrease the size of the Array after creation.
- You have to shift the other elements to fill in or close gaps, which takes worst-case O(n) time.

What will happen if you do not initialize an Array?

The array will take default values depending upon the data type.

Can you declare an array without assigning the size of an array?

No, we cannot declare an array without assigning size. If we declare an array without size, it will throw compile time error.

Thank You !

## Agenda

- Traversing 2D Arrays: Row-Major Order
- Traversing 2D Arrays: Column-Major Order
- Combining Traversal and Conditional Logic
- Examples

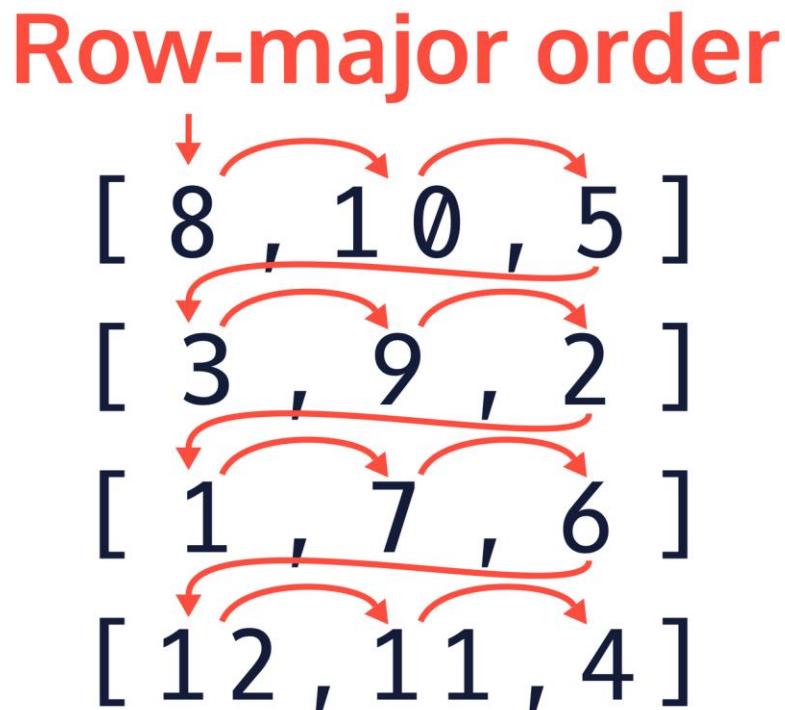
In the last session, we have gone over how to think about 2D array traversal in terms of arrays of arrays, but there are two main ways of thinking about traversal in terms of rows and columns. These are called row-major order and column-major order, which we will see in this session.

### Traversing 2D Arrays: Row-Major Order

Row-major order for 2D arrays refers to a traversal path which moves horizontally through each row starting at the first row and ending with the last.

Although we have already looked at how 2D array objects are stored in Java, this ordering system conceptualizes the 2D array into a rectangular matrix and starts the traversal at the top left element and ends at the bottom right element.

Here is a diagram which shows the path through the 2D array:



This path is created by the way we set up our nested loops. In the previous exercise, we looked at how we can traverse the 2D array by having nested loops in a variety of formats, but if we want to control the indices, we typically use standard **for** loops.

Let's take a closer look at the structure of the nested **for** loops when traversing a 2D array:

Given this 2D array of strings describing the element positions:



```
String[][] matrix = {[["[0][0]", "[0][1]", "[0][2]"],
 ["[1][0]", "[1][1]", "[1][2]"],
 ["[2][0]", "[2][1]", "[2][2]"],
 ["[3][0]", "[3][1]", "[3][2]"]};
```

Lets keep track of the total number of iterations as we traverse the 2D array:



```
int stepCount = 0;
```

```
for(int a = 0; a < matrix.length; a++) {
 for(int b = 0; b < matrix[a].length; b++) {
 System.out.print("Step: " + stepCount);
 System.out.print(", Element: " + matrix[a][b]);
 System.out.println();
 stepCount++;
 }
}
```

Here is the output of the above code:



```
Step: 0, Element: [0][0]
Step: 1, Element: [0][1]
Step: 2, Element: [0][2]
Step: 3, Element: [1][0]
```

Step: 4, Element: [1][1]

Step: 5, Element: [1][2]

Step: 6, Element: [2][0]

Step: 7, Element: [2][1]

Step: 8, Element: [2][2]

Step: 9, Element: [3][0]

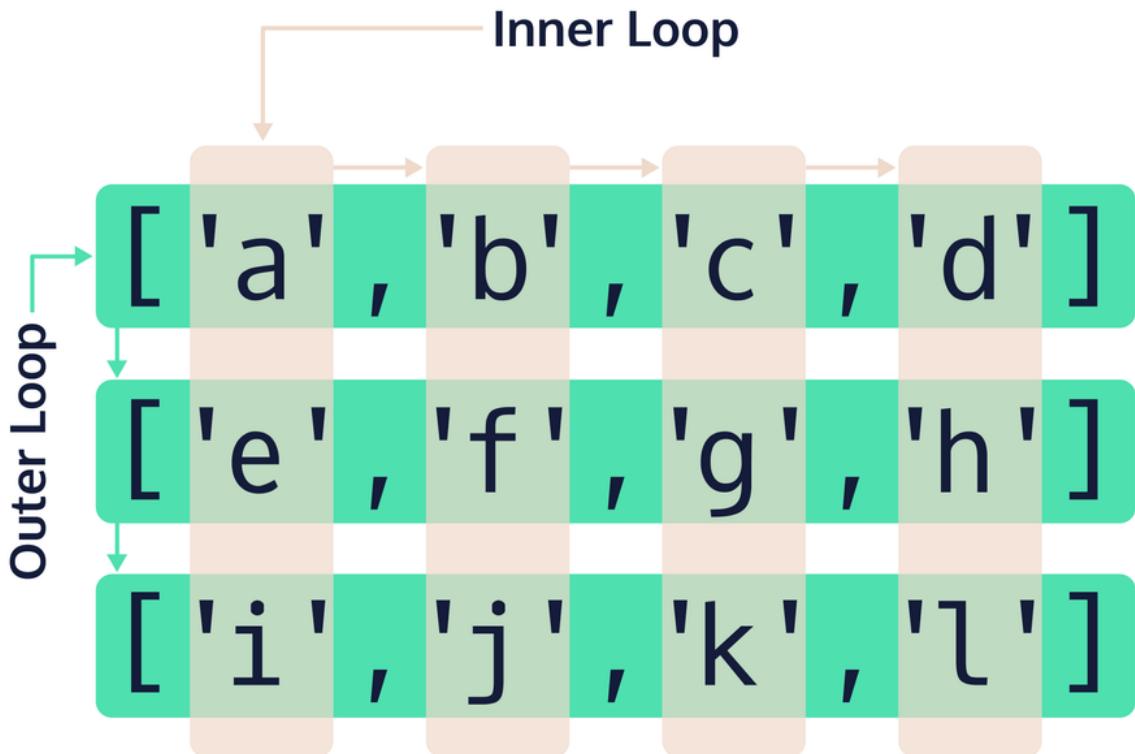
Step: 10, Element: [3][1]

Step: 11, Element: [3][2]

The step value increases with every iteration within the inner **for** loop. Because of this, we can see the order in which each element is accessed. If we follow the step value in the output shows us that the elements are accessed in the same order as the row-major diagram above. Now why is that?

This is because in our **for** loop, we are using the number of rows as the termination condition within the outer **for** loop header `a < matrix.length`; Additionally, we are using the number of columns `b < matrix[a].length` as the termination condition for our inner loop. Logically we are saying: “For every row in our matrix, iterate through every single column before moving to the next row”. This is why our above example is traversing the 2D array using row-major order.

Here is a diagram showing which loop accesses which part of the 2D array for row-major order:



### Why Use Row-Major Order?

Row-major order is important when we need to process data in our 2D array by row. You can be provided data in a variety of formats and you may need to perform calculations of rows of data at a time instead of individual elements. Let's take one of our previous checkpoint exercises as an example. You were asked to calculate the sum of the entire 2D array of integers by traversing and accessing each element. Now, if we wanted to calculate the sum of each row, or take the average of each row, we can use row-major order to access the data in the order that we need. Let's look at an example!

Given a 6X3 2D array of doubles:



```
double[][] data = {{0.51,0.99,0.12},
 {0.28,0.99,0.89},
 {0.05,0.94,0.05},
 {0.32,0.22,0.61},
```

```
{1.00,0.95,0.09},
{0.67,0.22,0.17});
```

Calculate the sum of each row using row-major order:



```
double rowSum = 0.0;

for(int o = 0; o < data.length; o++) {

 rowSum = 0.0;

 for(int i = 0; i < data[o].length; i++) {

 rowSum += data[o][i];

 }

 System.out.println("Row: " + o + ", Sum: " + rowSum);

}
```

The output of the above code is:



```
Row: 0, Sum: 1.62
Row: 1, Sum: 2.16
Row: 2, Sum: 1.04
Row: 3, Sum: 1.15
Row: 4, Sum: 2.04
Row: 5, Sum: 1.06
```

An interesting thing to note is that, due to the way 2D arrays are structured in Java, enhanced **for** loops are always in row-major order. This is because an enhanced **for** loop iterates through the elements of the outer array which causes the terminating condition to be the length of the 2D array which is the number of rows.

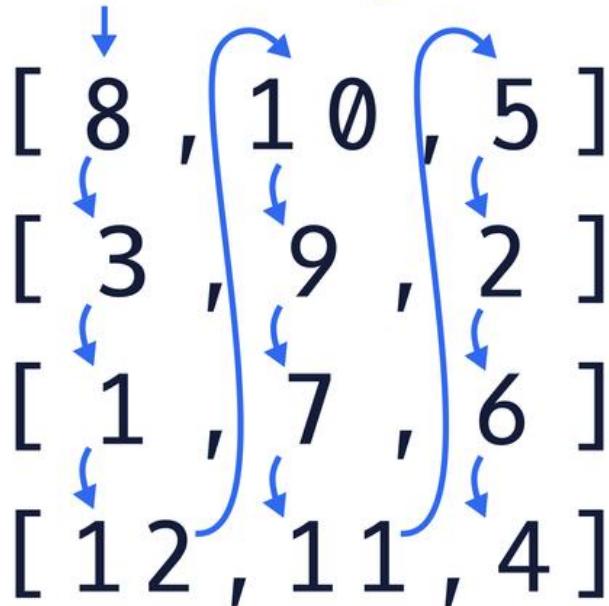
### Traversing 2D Arrays: Column-Major Order

Column-major order for 2D arrays refers to a traversal path which moves vertically down each column starting at the first column and ending with the last.

This ordering system also conceptualizes the 2D array into a rectangular matrix and starts the traversal at the top left element and ends at the bottom right element. Column-major order has the same starting and finishing point as row-major order, but its traversal is completely different

Here is a diagram which shows the path through the 2D array:

## Column-major order



In order to perform column-major traversal, we need to set up our nested loops in a different way. We need to change the outer loop from depending on the number of rows, to depending on the number of columns. Likewise we need the inner loop to depend on the number of rows in its termination condition.

Let's look at our example 2D array from the last exercise and see what needs to be changed.

Given this 2D array of strings describing the element positions:



```
String[][] matrix = {[["0"][0], ["0"][1], ["0"][2]],
 {[["1"][0], ["1"][1], ["1"][2]},
 {[["2"][0], ["2"][1], ["2"][2]},
 {[["3"][0], ["3"][1], ["3"][2]}}};
```

Let's keep track of the total number of iterations as we traverse the 2D array. We also need to change the termination condition (middle section) within the outer and inner **for** loop.



```
int stepCount = 0;

for(int a = 0; a < matrix[0].length; a++) {
 for(int b = 0; b < matrix.length; b++) {
 System.out.print("Step: " + stepCount);
 System.out.print(", Element: " + matrix[b][a]);
 System.out.println();
 stepCount++;
 }
}
```

Here is the output of the above code:



Step: 0, Element: [0][0]

Step: 1, Element: [1][0]

Step: 2, Element: [2][0]

Step: 3, Element: [3][0]

Step: 4, Element: [0][1]

Step: 5, Element: [1][1]

Step: 6, Element: [2][1]

Step: 7, Element: [3][1]

Step: 8, Element: [0][2]

Step: 9, Element: [1][2]

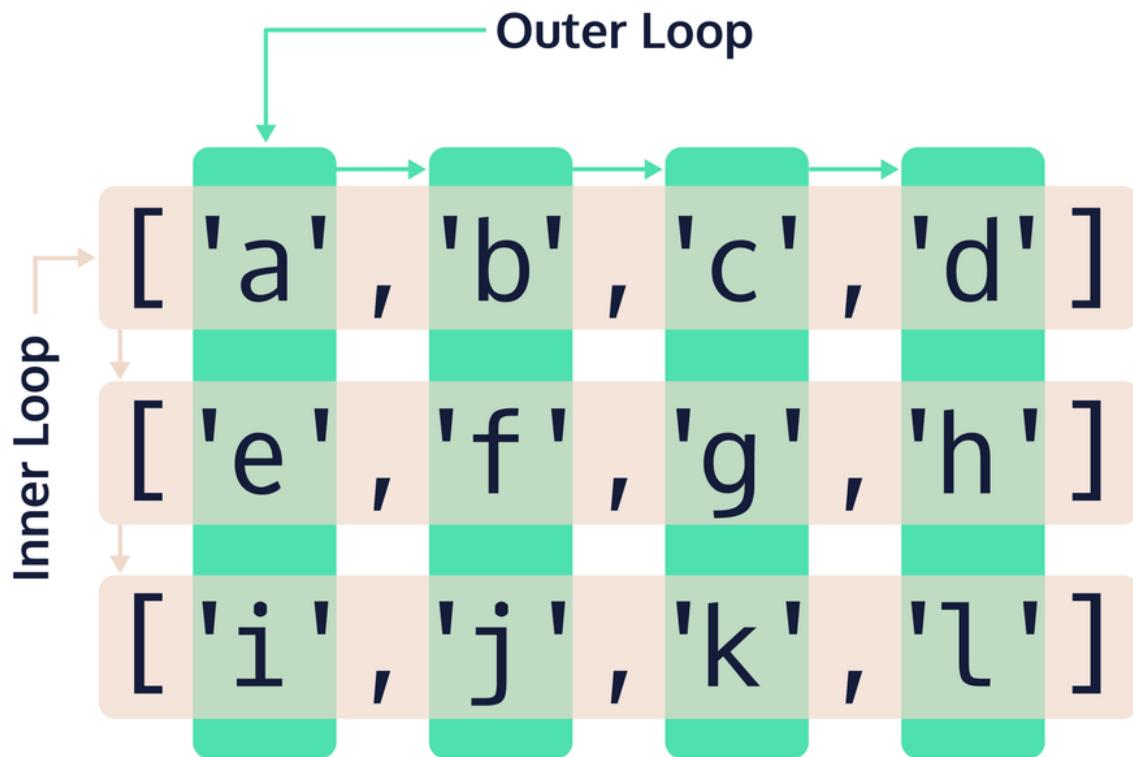
Step: 10, Element: [2][2]

Step: 11, Element: [3][2]

As you can see in the code above, the way we accessed the elements from our 2D array of strings called matrix is different from the way we accessed them when using row-major order. Let's remember that the way we get the number of columns is by using matrix[0].length and the way we get the number of rows is by using matrix.length. Because of these changes to our **for** loops, our iterator a now iterates through every column while our iterator b iterates through every row. Since our iterators now represent the opposite values, whenever we access an element from our 2D array, we need to keep in mind what

indices we are passing to our accessor. Remember the format we use for accessing the elements  $\text{matrix}[\text{row}][\text{column}]$ ? Since  $\text{a}$  now iterates through our column indices, we place it in the right set of brackets, and the  $\text{b}$  is now placed in the left set of brackets.

Here is a diagram showing which loop accesses which part of the 2D array for column-major order:



### Why Use Column-Major Order?

Column major order is important because there are a lot of cases when you need to process data vertically. Let's say that we have a chart of information which includes temperature data about each day. The top of each column is labeled with a day, and each row represents an hour. In order to find the average temperature per day, we would need to traverse the data vertically since each column represents a day. As mentioned in the last exercise, data can be provided in many different formats and shapes and you will need to know how to traverse it accordingly.

Let's look at our sum example from the last exercise, but now using column-major order.

Given a 6X3 2D array of doubles:



```
double[][] data = {{0.51,0.99,0.12},
 {0.28,0.99,0.89},
 {0.05,0.94,0.05},
 {0.32,0.22,0.61},
 {1.00,0.95,0.09},
 {0.67,0.22,0.17}};
```

Calculate the sum of each column using column-major order:



```
double colSum = 0.0;

for(int o = 0; o < data[0].length; o++) {
 colSum = 0.0;
 for(int i = 0; i < data.length; i++) {
 colSum += data[i][o];
 }
 System.out.println("Column: " + o + ", Sum: " + colSum);
}
```

The output of the above code is:



Column: 0, Sum: 2.83

Column: 1, Sum: 4.31

Column: 2, Sum: 1.93

### Combining Traversal and Conditional Logic

When working with 2D arrays, it is important to be able to combine traversal logic with conditional logic in order to effectively navigate and process the data. Here are a few ways in how conditional logic can affect 2D array traversal:

- Skipping or selecting certain rows and columns
- Modifying elements only if they meet certain conditions
- Complex calculations using the 2D array data

- Formatting the 2D array
- Avoiding exceptions / smart processing

Let's go over a few examples which use these ideas:

First, let's think about a situation where you have some string data inside a 2D array. We have an application which allows users to input events on a calendar. This is represented by a 5x7 2D array of strings. Due to the fact that the number of days in each month is slightly different and that there are less than 35 days in a month, we know that some of our elements are going to be empty. We want our application to do a few things:

- Detect which days of which weeks have something planned and alert us about the event.
- Count the number of events for each week
- Count the number of events for each day

Here is a visualization of what our calendar data looks like after a user has entered in some event information:

|      | Sun          | Mon      | Tues | Wed      | Thurs  | Fri     | Sat    |
|------|--------------|----------|------|----------|--------|---------|--------|
| Wk 1 | Volunteer    | Delivery |      |          | Doctor |         | Soccer |
| Wk 2 |              | Exam 1   |      | Mechanic |        |         | Soccer |
| Wk 3 | Volunteer    | Off Work |      | Birthday |        | Concert |        |
| Wk 4 |              | Exam 2   |      |          | Doctor |         | Soccer |
| Wk 5 | Visit Family |          |      |          |        |         |        |

Here's what our calendar data looks like in our application



```
String[][] calendar = {{"volunteer", "delivery", null, null, "doctor", null, "soccer"}, {null, "exam 1", null, "mechanic", null, null, "soccer"}, {"volunteer", "off work", null, "birthday", null, "concert", null}, {null, "exam 2", null, null, "doctor", null, "soccer"}, {"visit family", null, null, null, null, null}};
```

Let's look at some code which accomplishes the requirements above. Carefully look through each line of code and read all of the comments.

There are a few things to note:

- Row-major or column-major order can be used to access the individual events
- Row-major order must be used to count the number of events per week since each row represents a week

Let's take care of the first 2 requirements in one set of nested row-major loops



```
for(int i = 0; i < calendar.length; i++) {
 numberOfEventsPerWeek = 0;
 for(int j = 0; j < calendar[i].length; j++) {
 // We need conditional logic to ensure that we do not count the empty days
 String event = calendar[i][j];
 if(event!=null && !event.equals("")) {
 // If the day does not have a null value and an empty string for an event, then we print it and
 // count it
 System.out.println("Week: " + (i+1) + ", Day: " + (j+1) + ", Event: " + event);
 numberOfEventsPerWeek++;
 }
 }
 System.out.println("Total number of events for week "+ (i+1) +": " + numberOfEventsPerWeek + "\n");
}
```

The above code produces this output:



```
Week: 1, Day: 1, Event: volunteer
```

Week: 1, Day: 2, Event: delivery

Week: 1, Day: 5, Event: doctor

Week: 1, Day: 7, Event: soccer

Total number of events for week 1: 4

Week: 2, Day: 2, Event: exam 1

Week: 2, Day: 4, Event: mechanic

Week: 2, Day: 7, Event: soccer

Total number of events for week 2: 3

Week: 3, Day: 1, Event: volunteer

Week: 3, Day: 2, Event: off work

Week: 3, Day: 4, Event: birthday

Week: 3, Day: 6, Event: concert

Total number of events for week 3: 4

Week: 4, Day: 2, Event: exam 2

Week: 4, Day: 5, Event: doctor

Week: 4, Day: 7, Event: soccer

Total number of events for week 4: 3

Week: 5, Day: 1, Event: visit family

Total number of events for week 5: 1

Now let's complete the third requirement. Since we need to count all of the events for each of the weekdays, we will need to traverse the calendar vertically.



```
int numberOfEventsPerWeekday = 0;
```

```
// We will use this array of day strings for our output later on so we don't have (day: 1)
```

```
String[] days = {"Sundays", "Mondays", "Tuesdays", "Wednesdays", "Thursdays", "Fridays", "Saturdays"};
```

```

for(int i = 0; i < calendar[0].length; i++) {
 numberOfEventsPerWeekday = 0;
 for(int j = 0; j < calendar.length; j++) {
 // Don't forget to flip the iterators in the accessor since we are flipping the direction we are
 // navigating.

 // Remember, i now controls columns and j now controls rows
 String event = calendar[j][i];
 if(event!=null && !event.equals("")) {
 // Make sure we have an event for the day before counting it
 numberOfEventsPerWeekday++;
 }
 }
 // Use the days string array from earlier to convert the day index to a real weekday string
 System.out.println("Number of events on " + days[i] + ": " + numberOfEventsPerWeekday);
}

```

The output is:



```

Number of events on Sundays: 3
Number of events on Mondays: 4
Number of events on Tuesdays: 0
Number of events on Wednesdays: 2
Number of events on Thursdays: 2
Number of events on Fridays: 1
Number of events on Saturdays: 3

```

This example uses many of the concepts we have learned before. We use row-major order, column-major order, as well as including conditional logic to ensure that we have data for the elements we are accessing.

Additionally, we can use conditional logic to skip portions of the 2D array. For example, let's say we wanted to print the events for weekdays only and skip the weekends.

We could use a conditional statement such as if( $j \neq 0 \ \&\& j \neq 6$ ) in order to skip Sunday (0) and Saturday (6).

These modifications to our 2D array traversal are very common when processing data in applications. We need to know which cells to look at (skipping column titles for example), which cells to ignore (empty data, invalid data, outliers, etc.), and which cells to convert (converting string input from a file to numbers).

### Example Questions on 2D array

1. Write a Java program to print an array after changing the rows and columns of a given two-dimensional array.

Original Array:



10 20 30

40 50 60

After changing the rows and columns of the said array:



10 40

20 50

30 60

Solution:



```
import java.util.Scanner;

public class Solution {

 public static void main(String[] args) {

 int[][] twodm = {

 {10, 20, 30},
```

```

 {40, 50, 60}

 };

 System.out.print("Original Array:\n");
 print_array(twodm);

 System.out.print("After changing the rows
 and columns of the said array:");

 transpose(twodm);

}

private static void transpose(int[][] twodm) {

 int[][] newtwodm = new int[twodm[0].length][twodm.length];

 for (int i = 0; i < twodm.length; i++) {
 for (int j = 0; j < twodm[0].length; j++) {
 newtwodm[j][i] = twodm[i][j];
 }
 }

 print_array(newtwodm);
}

private static void print_array(int[][] twodm) {
 for (int i = 0; i < twodm.length; i++) {
 for (int j = 0; j < twodm[0].length; j++) {
 System.out.print(twodm[i][j] + " ");
 }
 System.out.println();
 }
}

```

```
 }
}
}
```

Output:



Original Array:

10 20 30

40 50 60

After changing the rows and columns of the said array: 10 40

20 50

30 60

2. In this problem, you have to implement the `int [] removeEven(int[] arr)` method, which removes all the even elements from the array and returns back updated array.



Solution:



```
class CheckRemoveEven {
 public static int[] removeEven(int[] arr) {
 int oddElements = 0;

 //Find number of odd elements in arr
 for (int i = 0; i < arr.length; i++) {
 if (arr[i] % 2 != 0) oddElements++;
 }

 //Create result array with size equal to the number of odd
 //elements in arr
```

```

int[] result = new int[oddElements];
int result_index = 0;

//Put odd values from arr to the resulted array
for (int i = 0; i < arr.length; i++) {
 if (arr[i] % 2 != 0)
 result[result_index++] = arr[i];
} //end of for loop

return result;
} //end of removeEven
}

```

### **Interview Questions**

- **What do you understand by row-major order while traversing 2D arrays ?**

“Row-major order” refers to an ordering of 2D array elements where traversal occurs across each row - from the top left corner to the bottom right. In Java, row major ordering can be implemented by having nested loops where the outer loop variable iterates through the rows and the inner loop variable iterates through the columns. Note that inside these loops, when accessing elements, the variable used in the outer loop will be used as the first index, and the inner loop variable will be used as the second index.

- **What do you understand by column-major order while traversing 2D arrays ?**

“Column-major order” refers to an ordering of 2D array elements where traversal occurs down each column - from the top left corner to the bottom right. In Java, column major ordering can be implemented by having nested loops where the outer loop variable iterates through the columns and the inner loop variable iterates through the rows. Note that inside these loops, when accessing elements, the variable used in the outer loop will be used as the second index, and the inner loop variable will be used as the first index.

## Agenda

- **Introduction to 2-Dimensional array**
- **Decelaration of 2-Dimensional array**
- **Initialization of 2-Dimensional array**
- **Java 2-Dimensional array of Primitive Type**
- **Java 2-Dimensional array of Objects**
- **Accessing elements in a 2-Dimensional array**
- **Modifying Elements in a 2-Dimensional Array**
- **Example of 2-Dimensional Array**
- **Traversing 2D Arrays: Introduction**
- **Traversing 2D Arrays: Practice with Loops**

We have learnt about the Arrays through Java, which was basically a single dimensional array till now, but now we are going to learn about the multi-dimensional arrays in Java i.e., 2-dimensional arrays. So, let's start by learning about the basics of 2-dimensional arrays.

### Introduction to 2-dimensional array

As we have learned previously, an array is a group of data consisting of the same type. This means that we can have an array of primitive data types (such as integers):



[1, 2, 3, 4, 5]

We can even have an array of Objects. For example, the following example shows an array of String Objects:



["hello", "world", "how", "are", "you"]

In Java, arrays are considered Objects; therefore, we can also have an array of arrays:



[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

These are called 2D arrays since we can logically view them as a two-dimensional matrix of values containing both rows and columns.

[1, 2, 3]

[4, 5, 6]

[7, 8, 9]

Additionally, we can have 2D arrays which are not rectangular in shape. These are called jagged arrays:



```
[['a', 'b', 'c', 'd'], ['e', 'f'], ['g', 'h', 'i', 'j'], ['k']]
```

```
['a' , 'b' , 'c' , 'd']
```

```
['e' , 'f']
```

```
['g' , 'h' , 'i' , 'j']
```

```
['k']
```

We won't be covering jagged arrays in this lesson, but be aware that 2D arrays don't always have to have the same number of subarrays in each array. This would cause the shape of the 2D array to not be rectangular.

Why use 2D arrays?

- It is useful to use 2D arrays for situations where you need to store and organize data by rows and columns. For example, exporting data to be used in a spreadsheet.
- You can condense multiple arrays down to a single variable using 2D arrays. For example, if you have 10 students who each have 10 different quiz grades, you can represent the overall class quiz grades as a 10x10 2D array by having each row represent a student and each column represent one of the quizzes they have taken.
- 2D arrays can be used to map out data. For example, if you want to create a game of tic-tac-toe, you can represent the game state by using a 3x3 2D array.

There are many other ways to use 2D arrays depending on the application. The only downside is that once initialized, no new rows or columns can be added or removed without copying the data to a newly initialized 2D array. This is because the length of arrays in Java are immutable (unable to be changed after creation).

### Deceleration of 2-Dimensional array

When declaring 2D arrays, the format is similar to normal, one-dimensional arrays, except that you include an extra set of brackets after the data type. In this example, int represents the data type, the first set of brackets [] represent an array, and the second set of brackets [] represent that we are declaring an array of arrays.



```
int[][] intTwoDArray;
```

You can think of this as creating an array ([]) of int arrays (int[]). So we end up with int[][].

Let's understand the creation of Java's two-dimensional array with an example:



```
//Declaring 2D array
```

```
int[][] a;
```

```
//Creating a 2D array
```

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

Here, the reference variable a points to a two-dimensional array object that represents a 3X3 integer matrix i.e., the array object contains 3 rows and 3 columns columns, and can only store integer (int) values.

**Note:** When we create a 2D array object using the new keyword, the JVM (Java Virtual Machine) allocates the memory required for the two-dimensional array and initializes the memory spaces with the default values according to the data type of the array object. For example, in the case of the Integer array (int[][]), every element of the array is initialized with the default value of 0.

### Initialization of 2-Dimensional array

Now that we've declared a 2D array, let's look at how to initialize it with starting values. When initializing arrays, we define their size. Initializing a 2D array is different because, instead of only including the number of elements in the array, you also indicate how many elements are going to be in the sub-arrays. This can also be thought of as the number of rows and columns in the 2D matrix.



```
int[][] intArray1;
```

```
intArray1 = new int[row][column];
```

Here is an example of initializing an empty 2D array with 3 rows and 5 columns.



```
int[][] intArray2;
```

```
intArray2 = new int[3][5];
```

This results in a matrix which looks like this:

If you already know what values are going to be in the 2D array, you can initialize it and write all of the values into it at once. We can accomplish this through initializer lists

- In Java, initializer lists are a way of initializing arrays and assigning values to them at the same time
- We can use this for 2D arrays as well by creating an initializer list of initializer lists

An example of an initializer list for a regular array would be:



```
char[] charArray = {'a', 'b', 'c', 'd'};
```

Similar to how a regular initializer list defines the size and values of the array, nested initializer lists will define the number of rows, columns, and the values for a 2D array.

There are three situations in which we can use initializer lists for 2D arrays:

1. In the case where the variable has not yet been declared, we can provide an abbreviated form since Java will infer the data type of the values in the initializer lists:



```
double[][] doubleValues = {{1.5, 2.6, 3.7}, {7.5, 6.4, 5.3}, {9.8, 8.7, 7.6}, {3.6, 5.7, 7.8}};
```

2. If the variable has already been declared, you can initialize it by creating a new 2D array object with the initializer list values:



```
String[][] stringValues;
```

```
stringValues = new String[][] {"working", "with"}, {"2D", "arrays"}, {"is", "fun"};
```

3. The previous method also applies to assigning a new 2D array to an existing 2D array stored in a variable.

### Java 2-Dimensional array of Primitive Type

Arrays are a collection of elements that have similar data types. Hence, we can create an array of primitive data types as well as objects. A 2D array of a primitive data type in Java (let's say int) is simply an array of Integer arrays.

We can declare a 2D array in Java for all the primitive Java data types in the following manner:



```
int[][] AIntegerArray; // 2D Integer Array
```

```
byte[][] AByteArray; // 2D Byte Array
```

```
short[][] AShortArray; // 2D Short Array
long[][] ALongArray; // 2D Long Array
float[][] AFloatArray; // 2D Float Array
double[][] ADoubleArray; // 2D Double Array
boolean[][] ABooleanArray; // 2D Boolean Array
char[][] ACharArray; // 2D Character Array
```

### Java 2-Dimensional array of Objects

As discussed earlier, we can create an array of objects. A two-dimensional array of objects in Java is simply a collection of arrays of several reference variables. We can declare a 2D array of objects in the following manner:



```
ClassName[][] ArrayName;
```

This syntax declares a two-dimensional array having the name ArrayName that can store the objects of class ClassName in tabular form. These types of arrays that store objects as their elements are typically used while dealing with String data objects.

### Accessing elements in a 2-Dimensional array

Let's first review how to access elements in regular arrays.

For a normal array, all we need to provide is an index (starting at 0) which represents the position of the element we want to access. Let's look at an example!

Given an array of five Strings:



```
String[] words = {"cat", "dog", "apple", "bear", "eagle"};
```

We can access the first element using index 0, the last element using the length of the array minus one (in this case, 4), and any of the elements in between. We provide the index of the element we want to access inside a set of brackets. Let's see those examples in code:



```
// Store the first element from the String array
String firstWord = words[0];
```

```
// Store the last element of the String array
```

```
String lastWord = words[words.length-1];

// Store an element from a different position in the array
String middleWord = words[2];
```

Now for 2D arrays, the syntax is slightly different. This is because instead of only providing a single index, we provide two indices. Take a look at this example:



```
// Given a 2D array of integer data
int[][] data = {{2,4,6}, {8,10,12}, {14,16,18}};
```

```
// Access and store a desired element
int stored = data[0][2];
```

There are two ways of thinking when accessing a specific element in a 2D array.

- The first way of thinking is that the first value represents a row and the second value represents a column in the matrix
- The second way of thinking is that the first value represents which subarray to access from the main array and the second value represents which element of the subarray is accessed

The above example of the 2D array called data can be visualized like so. The indices are labeled outside of the matrix:

| Index | 0               | 1 | 2 |
|-------|-----------------|---|---|
| 0     | [ 2 , 4 , 6 ]   |   |   |
| 1     | [ 8 , 10 , 12 ] |   |   |
| 2     | [14, 16, 18]    |   |   |

Using this knowledge, we now know that the result of `int stored = data[0][2];` would store the integer 6. This is because the value 6 is located on the first row (index 0) and the third column (index 2). Here is a template which can be used for accessing elements in 2D arrays:



```
datatype variableName = existing2DArray[row][column];
```

Here is another way to visualize the indexing system for our example integer array seen above. We can see what row and column values are used to access the element at each position.

[data[0][0],data[0][1],data[0][2]]

[data[1][0],data[1][1],data[1][2]]

[data[2][0],data[2][1],data[2][2]]

When accessing these elements, if either the row or column value is out of bounds, then an `ArrayIndexOutOfBoundsException` will be thrown by the application.

### Modifying Elements in a 2-Dimensional Array

Now let's review how to modify elements in a normal array.

For a one dimensional array, you provide the index of the element which you want to modify within a set of brackets next to the variable name and set it equal to an acceptable value:



```
storedArray[5] = 10;
```

For 2D arrays, the format is similar, but we will provide the outer array index in the first set of brackets and the subarray index in the second set of brackets. We can also think of it as providing the row in the first set of brackets and the column index in the second set of brackets if we were to visualize the 2D array as a rectangular matrix:



```
twoDArray[1][3] = 150;
```

To assign a new value to a certain element, make sure that the new value you are using is either of the same type or is castable to the type already in the 2D array.

Let's say we wanted to replace four values from a new 2D array called intTwoD. Look at this example code to see how to pick individual elements and assign new values to them.



```
int[][] intTwoD = new int[4][3];
```

```
intTwoD[3][2] = 16;
```

```
intTwoD[0][0] = 4;
```

```
intTwoD[2][1] = 12;
```

```
intTwoD[1][1] = 8;
```

Here is a before and after image showing when the 2D array was first initialized compared to when the four elements were accessed and modified.

| Before    | After      |
|-----------|------------|
| [0, 0, 0] | [4, 0, 0]  |
| [0, 0, 0] | [0, 8, 0]  |
| [0, 0, 0] | [0, 12, 0] |
| [0, 0, 0] | [0, 0, 16] |

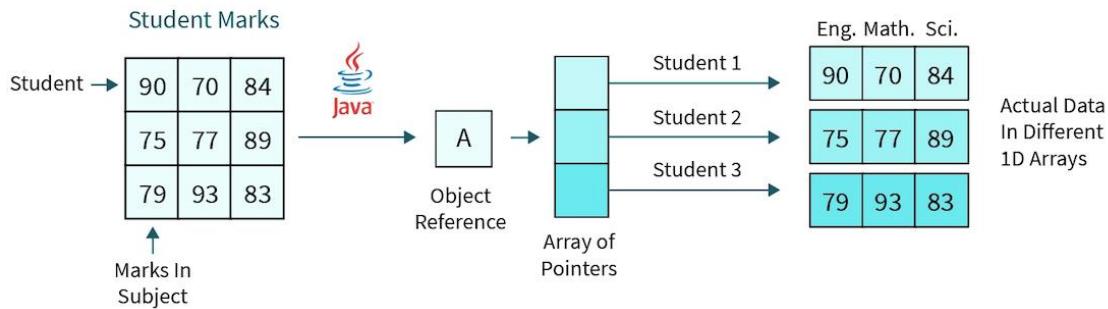
### Example of 2-Dimensional Array

Now, let's look at an example to completely understand 2D arrays in Java.

Consider a scenario where we wish to store the marks attained by 3 students in Maths, English, and Science Subjects. Here, we are storing a collection of marks (marks of English, Maths, and Science) for each student. Hence, we can use a table to represent and store student marks. In the Student Marks

table, the rows will represent the marks of a particular student and the columns will represent the subject in which the student attained those marks as shown below:

#### Java's Two Dimensional Arrays Example -



This Student Marks table can be mapped into Java using 2D arrays. We can use a two-dimensional integer array to represent the tabular form for the marks of students. This can be implemented in the following manner:



```
import java.util.Arrays;
```

```
public class Main {
 public static void main(String args[]) {
 int[][] StudentMarks = new int[3][3];

 // Marks Attained By Student 1
 StudentMarks[0][0] = 90; // English
 StudentMarks[0][1] = 70; // Maths
 StudentMarks[0][2] = 84; // Science

 // Marks Attained By Student 2
 StudentMarks[1][0] = 75; // English
 StudentMarks[1][1] = 77; // Maths
```

```

StudentMarks[1][2] = 89; // Science

// Marks Attained By Student 3
StudentMarks[2][0] = 69; // English
StudentMarks[2][1] = 93; // Maths
StudentMarks[2][2] = 83; // Science

// Displaying Marks of Students
System.out.println("Student Marks Matrix");
System.out.println(Arrays.deepToString(StudentMarks));

}

}

```

Output:



Student Marks Matrix

`[[90, 70, 84], [75, 77, 89], [69, 93, 83]]`

Here, we are declaring an Integer array `StudentMarks` that represents a 3X3 matrix. Now, to assign the marks of each student, we are accessing the corresponding elements using indexing and then assigning the marks to that particular element.

For example, to store the marks attained by student 1 in English, we are accessing the **0th** row (First student) and the **0th** column (First subject i.e., English) and then replacing the default integer value of 0 with the marks of the student (`StudentMarks[0][0] = 90`).

Finally, we are using the built-in method `deepToString()` of the `Arrays` class to display the `StudentMarks` two-dimensional array.

### Traversing 2D Arrays: Introduction

Traversing 2D arrays using loops is important because it allows us to access many elements quickly, access elements in very large 2D arrays, and even access elements in 2D arrays of unknown sizes.

Let's remember the structure of 2D arrays in Java:



```
char[][] letterBlock = {{'a','b','c'},{'d','e','f'},{'g','h','i'},{'j', 'k', 'l'}};
```

In Java, 2D arrays are like normal arrays, but each element is another array. This is shown by the initialized 2D array above. The outer array consists of four elements, where each element consists of a three element subarray.

Let's see what happens when we access elements of the outer array



```
System.out.println(Arrays.toString(letterBlock[0]) + "\n");
System.out.println(Arrays.toString(letterBlock[1]) + "\n");
System.out.println(Arrays.toString(letterBlock[2]) + "\n");
System.out.println(Arrays.toString(letterBlock[3]) + "\n");
```

Here is the output of the above code:



[a, b, c]

[d, e, f]

[g, h, i]

[j, k, l]

As you can see, we can retrieve the entire subarray from each of the outer array elements. If you look at how we are accessing these subarrays, we are just increasing the index. This means we can access each sub-array in the 2D array using a loop!

Let's take a look at an example which produces the same output, but can handle any sized 2D array.



```
for(int index = 0; index < letterBlock.length; index++){
 System.out.println(Arrays.toString(letterBlock[index]) + "\n");
}
```

Here is the result:



[a, b, c]

[d, e, f]

[g, h, i]

[j, k, l]

Now let's remember how to access a value from the subarray. Previously, we learned that we can use the double brackets `[][]`, where the first set of brackets contains the index of the element of the outer array and the second set of brackets contains the index of the element in the subarray. If we wanted to retrieve the letter 'f' we would use:

```
char storedLetter = letterBlock[1][2];
```

Since we know we can use a loop to retrieve each of the subarrays stored in the outer array, we can then use a nested loop to access each of the elements from the sub-array.

You might be wondering how we can figure out the number of iterations needed in order to fully traverse the 2D array.

- In order to find the number of elements in the outer array, we just need to get the length of the 2D array.
  - `int lengthOfOuterArray = letterBlock.length;`
  - When thinking about the 2D array in matrix form, this is the height of the matrix (the number of rows)
- In order to find the number of elements in the subarray, we can get the length of the subarray after it has been retrieved from the outer array.
  - Remember that we retrieved the sub array earlier using this format:
    - `char[] subArray = letterBlock[0];`
  - Therefore, we can use this to get the length of the first subarray in the 2D array
    - `int lengthOfSubArray = letterBlock[0].length;`
    - When thinking about the 2D array in matrix form, this is the width of the matrix (the number of columns)
  - In most cases, getting the length of the first subarray in the 2D array will apply to the rest of the subarrays (if it is rectangular in shape), but there are rare occasions where

the length of the subarrays could be different. This occurs if the 2D array is a jagged array. We won't be working with any jagged 2D arrays in this lesson, but it's something to keep in mind.

Let's look at an example!



```
for(int a = 0; a < letterBlock.length; a++) {
 for(int b = 0; b < letterBlock[a].length; b++) {
 System.out.print("Accessed: " + letterBlock[a][b] + "\t");
 }
 System.out.println();
}
```

You can think of the variable `a` as being the outer loop index, and the variable `b` as being the inner loop index.

Here is the output:

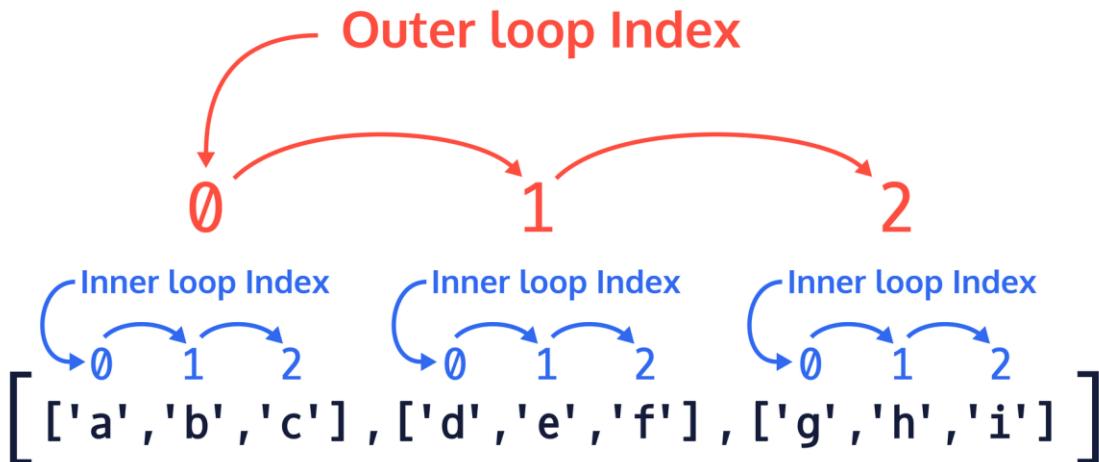


```
Accessed: a Accessed: b Accessed: c
Accessed: d Accessed: e Accessed: f
Accessed: g Accessed: h Accessed: i
```

Within the nested `for` loop, we can see that each of the subarray elements are being accessed by using the outer loop index for the outer array, and the inner loop index for the subarray.

Here is a diagram to help visualize how the 2D array is traversed using nested loops:

# 2D Array Traversal



We don't have to only use regular **for** loops for traversing 2D arrays. We can use enhanced **for** loops if we do not need to keep track of the indices. Since enhanced **for** loops only use the element of the arrays, it is a bit more cumbersome to keep track of which index we are at. This same idea applies to while and do-while loops as well. This is why we usually use regular **for** loops except for when we want to do something simple like printing.

We have gone over how to think about 2D array traversal in terms of arrays of arrays, but there are two main ways of thinking about traversal in terms of rows and columns. These are called row-major order and column-major order, which we will see in the next session.

## Traversing 2D Arrays: Practice with Loops

We have seen how to traverse 2D arrays using standard **for** loops, but in this part, we will practice traversing them using some other loop types. For example, you may want to only retrieve elements without keeping track of the indices using enhanced **for** loops, or you could continuously update the 2D array until a condition is met using **while** loops.

In enhanced **for** loops, each element is iterated through until the end of the array. When we think about the structure of 2D arrays in Java (arrays of array objects) then we know that the outer enhanced **for** loop elements are going to be arrays.

Let's take a look at an example:

Given this 2D array of character data:



```
char[][] charData = {{'a', 'b', 'c', 'd', 'e', 'f'}, {'g', 'h', 'i', 'j', 'k', 'l'}};
```

Print out every character using enhanced **for** loops:



```
for(char[] charRow : charData) {
 for(char c : charRow) {
 System.out.print(c + " ");
 }
 System.out.println();
}
```

Remember that the syntax for enhanced **for** loops looks like so: `for( datatype elementName : arrayName){}`. Since 2D arrays in Java are arrays of arrays, each element in the outer enhanced **for** loop is an entire row of the 2D array. The nested enhanced **for** loop is then used to iterate through each element in the extracted row. Here is the output of the above code:



```
a b c d e f
g h i j k l
```

Here is an example which accomplishes the same thing, but using **while** loops:



```
int i = 0, j=0;

while(i<charData.length) {
 j = 0;
 while(j<charData[i].length) {
 System.out.print(charData[i][j] + " ");
 j++;
 }
 System.out.println();
 i++;
}
```

Here is the output of the above code:



a b c d e f

g h i j k l

Notice how we can use different loop types for traversal, but still receive the same result.

### **Interview Questions**

What is the difference between a one-dimensional array and a two-dimensional array?

One-dimensional array stores a single sequence of elements and has only one index. A two-dimensional array stores an array of arrays of elements and uses two indices to access its elements.

What does it mean to be two dimensional?

Two-dimensional means having only two dimensions. In a geometric world, objects that have only height and width are two-dimensional or 2D objects. These objects do not have thickness or depth.

Triangle, rectangles, etc. are 2D objects. In software terms, two dimensional still means having two dimensions and we usually define data structures like arrays which can have 1, 2 or more dimensions.

How you can modify elements in two-dimensional array ?

In Java, elements in a 2D array can be modified in a similar fashion to modifying elements in a 1D array. Setting arr[i][j] equal to a new value will modify the element in row i column j of the array arr.

Thank You !

## Agenda

- OOP in JAVA
- Abstraction
- Inheritance
- Polymorphism
- Encapsulation

### What is OOPs Concept?

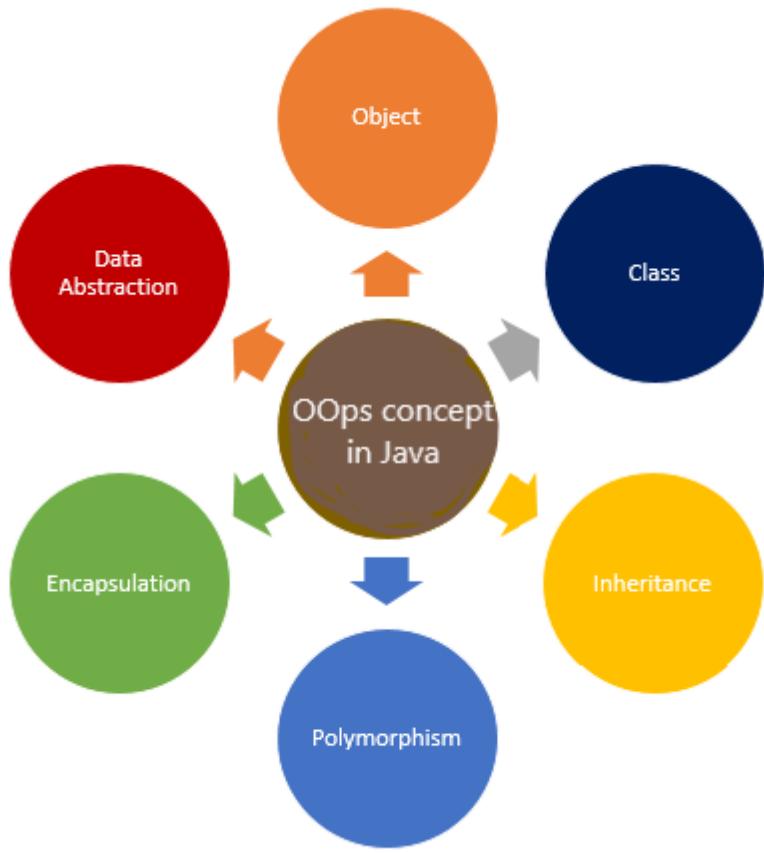
Object-oriented programming is a core of Java Programming, which is used for designing a program using classes and objects. OOPs, can also be characterized as data controlling for accessing the code. In this approach, programmers define the data type of a data structure and the operations that are applied to the data structure.

### What is OOPs in java?

OOps in java is to improve code readability and reusability by defining a Java program efficiently. The main principles of object-oriented programming are **abstraction, encapsulation, inheritance, and polymorphism**. These concepts aim to implement real-world entities in programs.

### List of OOPs Concepts in Java

- Objects
- Classes
- Object
- Class
- Abstraction
- Inheritance
- Polymorphism
- Encapsulation



### **What are Objects?**

Objects are always called instances of a class which are created from a class in java or any other language. They have states and behavior.

These objects always correspond to things found in the real world, i.e., real entities. So, they are also called run-time entities of the world. These are self-contained which consists of methods and properties which make data useful. Objects can be both physical and logical data. It contains addresses and takes up some space in memory. Some examples of objects are a dog, chair, tree etc.

When we treat animals as objects, it has states like color, name, breed etc., and behaviors such as eating, wagging the tail etc.

Suppose, we have created a class called My book, we specify the class name followed by the object name, and we use the keyword new.

### **Object Example 1:**



```
Public class Mybook {
```

```
int x=10;

Public static void main (String args []) {

Mybook Myobj= new Mybook ();

System.out.println(MyObj.x);

}

}
```

In the above example, a new object is created, and it returns the value of x which may be the number of books.

**Mybook Myobj= new Mybook ();**

This is the statement used for creating objects.

**System.out.println(Myobj.x);**

This statement is used to return the value of x of an object.

We can also create multiple objects in the same class and we can create in one class and access it in another class. This method is used for better organization of classes and always remember that name of the java file and the class name remains the same.

#### **Example 2:**

The below example shows how multiple objects are created in the same class and how they are accessed from another class.

- Mybook.java



```
Public class Mybook {
```

```
int x=10;
```

```
int y=8;
```

```
}
```

- Count.java



```
Class Count {
```

```
Public static void main (String [] args)
```

```
{
```

```
Mybook myobj1 = new myobj1();

Mybook myobj2 = new myobj2();

System.out.println (myobj1.x);

System.out.println (myobj2.y);

}

}
```

When this program is compiled, it gives the result as 10, and 8 respectively.

### **What are Classes?**

Classes are like object constructors for creating objects. The collection of objects is said to be a class. Classes are said to be logical quantities. Classes don't consume any space in the memory. Class is also called a template of an object. Classes have members which can be fields, methods and constructors. A class has both static and instance initializers.

A class declaration consists of:

1. **Modifiers:** These can be public or default access.
2. **Class name:** Initial letter.
3. **Superclass:** A class can only extend (subclass) one parent.
4. **Interfaces:** A class can implement more than one interface.
5. **Body:** Body surrounded by braces, { }.

A class keyword is used to create a class. A simplified general form of the class definition is given below:



```
class classname {

type instance variable 1;

type instance variable 2;

.

.

.

type instance variable n;

type methodname 1 (parameter list) {

// body od method
```

```
}

type methodname 2 (parameter list) {
 // body od method
}

type methodnamen (parameter list) {
 // body od method
}

}
```

The variables or data defined within a class are called instance variables. Code is always contained in the methods. Therefore, the methods and variables defined within a class are called members of the class. All the methods have the same form as the main () these methods are not specified as static or public.

### **What is Abstraction?**

Abstraction is a process which displays only the information needed and hides the unnecessary information. We can say that the main purpose of abstraction is data hiding. Abstraction means selecting data from a large number of data to show the information needed, which helps in reducing programming complexity and efforts.

There are also abstract classes and abstract methods. An abstract class is a type of class that declares one or more abstract methods. An abstract method is a method that has a method definition but not implementation. Once we have modelled our object using data abstraction, the same sets of data can also be used in different applications—abstract classes, generic types of behaviors and object-oriented programming hierarchy. Abstract methods are used when two or more subclasses do the same task in different ways and through different implementations. An abstract class can have both methods, i.e., abstract methods and regular methods.

Now let us see an example related to abstraction.

Suppose we want to create a student application and ask to collect information about the student.

We collect the following information.

- Name
- Class
- Address
- Dob
- Fathers name
- Mothers' names and so on.

We may not require every information that we have collected to fill out the application. So, we select the data that is required to fill out the application. Hence, we have fetched, removed, and selected the data, the student information from large data. This process is known as abstraction in the oops concept.

### **Abstract class example:**



```
//abstract parent class

Abstract class animal {
 //abstract method

 public abstract void sound () ;
}

Public class lion extends animal {
 Public void sound () {
 System.out.println (" roar ");
 }

 public Static void main (String args []) {
 animal obj = new lion ();
 obj. sound ();
 }
}
```

**Output:** Roar

### **What is Inheritance?**

Inheritance is a method in which one object acquires/inherits another object's properties, and inheritance also supports hierarchical classification. The idea behind this is that we can create new classes built on existing classes, i.e., when you inherit from an existing class, we can reuse methods and fields of the parent class. Inheritance represents the parent-child relationship. To know more about this concept check the free inheritance in java course.

For example, a whale is a part of the classification of marine animals, which is part of class mammal, which is under that class of animal. We use hierarchical classification, i.e., top-down classification. If we want to describe a more specific class of animals such as mammals, they would have more specific attributes such as teeth; cold-blooded, warm-blooded, etc. This comes under the subclass of animals whereas animals come under the superclass. The subclass is a class which inherits properties of the superclass. This is also called a derived class. A superclass is a base class or parental class from which a subclass inherits properties.

We use inheritance mainly for method overriding and R:

To inherit a class, we use the extend keyword.

There are five types of inheritance single, multilevel, multiple, hybrid and hierarchical.

- **Single level**

In this one class i.e., the derived class inherits properties from its parental class. This enables code reusability and also adds new features to the code. Example: class b inherits properties from class a.

Class A is the base or parental class and class b is the derived class.

**Syntax:**



Class a {

...

}

Class b extends class a {

...

}

- **Multilevel**

This one class is derived from another class which is also derived from another class i.e., this class has more than one parental class, hence it is called multilevel inheritance.

**Syntax:**



Class a {

....

}

Class b extends class a {

....

}

Class c extends class b {

...

}

- **Hierarchical level**

In this one parental class has two or more derived classes or we can say that two or more child classes have one parental class.

**Syntax:**



Class a {

...

}

Class b extends class a {

..

}

Class c extends class a {

..

}

- **Hybrid inheritance**

This is the combination of multiple and multilevel inheritances and in java, multiple inheritances are not supported as it leads to ambiguity and this type of inheritance can only be achieved through interfaces.

Consider that class a is the parental or base class of class b and class c and in turn, class b and class c are parental or a base class of class d. Class b and class c are derived classes from class a and class d is derived class from class b and class c.

The following program creates a superclass called add and a subclass called sub, using extend keyword to create a subclass add.



```
// a simple example of inheritance
```

```
//create a superclass
```

```
Class Add {
```

```
int my;
```

```
int by;
```

```
void setmyby (int xy, int hy) {
```

```

my=xy;
by=hy;
}
}

/ create a sub class

class b extends add {
int total;

void sum () {

public static void main (String args []) {

b subOb= new b ();
subOb. Setmyby (10, 12);
subOb. Sum ();
System.out.println("total =" + subOb. Total);

}
}

```

It gives output as – **total = 22**

### **What is Polymorphism?**

Polymorphism refers to many forms, or it is a process that performs a single action in different ways. It occurs when we have many classes related to each other by inheritance. Polymorphism is of two different types, i.e., compile-time polymorphism and runtime polymorphism. One of the examples of Compile time polymorphism is that when we overload a static method in java. Run time polymorphism also called a dynamic method dispatch is a method in which a call to an overridden method is resolved at run time rather than compile time. In this method, the overridden method is always called through the reference variable. By using method overloading and method overriding, we can perform polymorphism. Generally, the concept of polymorphism is often expressed as one interface, and multiple methods. This reduces complexity by allowing the same interface to be used as a general class of action.

### **Example:**



```
public class Bird {
```

...

```

Public void sound () {
 System.out.println (“ birds sounds ”);
}

}

public class pigeon extends Bird {

 ...
 @override
 public void sound () {
 System.out.println(“ cooing ”);
 }
}

public class sparrow extends Bird () {

 @override
 Public void sound () {
 System.out.println(“ chip ”);
 }
}

```

In the above example, we can see common action sound () but there are different ways to do the same action. This is one of the examples which shows polymorphism.

Polymorphism in java can be classified into two types:

1. Static / Compile-Time Polymorphism
2. Dynamic / Runtime Polymorphism

### **What is Compile-Time Polymorphism in Java?**

Compile-Time polymorphism in java is also known as Static Polymorphism. to resolved at compile-time which is achieved through the Method Overloading.

### **What is Runtime Polymorphism in Java?**

Runtime polymorphism in java is also known as Dynamic Binding which is used to call an overridden method that is resolved dynamically at runtime rather than at compile time.

### **What is Encapsulation?**

Encapsulation is one of the concepts in OOPs concepts; it is the process that binds together the data and code into a single unit and keeps both from being safe from outside interference and misuse. In this process, the data is hidden from other classes and can be accessed only through the current class's methods. Hence, it is also known as data hiding. Encapsulation acts as a protective wrapper that prevents the code and data from being accessed by outsiders. These are controlled through a well-defined interface.

Encapsulation is achieved by declaring the variables as private and providing public setter and getter methods to modify and view the variable values. In encapsulation, the fields of a class are made read-only or write-only. This method also improves reusability. Encapsulated code is also easy to test for unit testing.

**Example:**



```
class animal {
 // private field
 private int age;
 //getter method
 Public int getage () {
 return age;
 }
 //setter method
 public void setAge (int age) {
 this. Age = age;
 }
}

class Main {
 public static void main (String args []);
 //create an object of person
 Animal a1= new Animal ();
 //change age using setter
 A1. setAge (12);
 // access age using getter
```

```
System.out.println(" animal age is " + a1.getage ());
}
}
```

**Output:** Animal age is 12

In this example, we declared a private field called age that cannot be accessed outside of the class.

To access age, we used public methods. These methods are called getter and setter methods. Making age private allows us to restrict unauthorized access from outside the class. Hence this is called data hiding.

### **Can Polymorphism, Encapsulation and Inheritance work together?**

When we combine inheritance, polymorphism and encapsulation to produce a programming environment, this environment supports the development of far more robust and scalable programs that do the program-oriented model.

Let us consider a real-world example:

Humans are a form of inheritance from one standpoint, whereas cars are more like programs we write. All drivers rely on inheritance to drive different types of vehicles. People interface with the features of cars of all types as we have many different types of vehicles, and some have differences. The implementation of engines, brakes etc., comes under encapsulation and finally comes to polymorphism. We get a wide area of options on the same vehicle as to the anti-lock braking system, traditional braking system or power braking system. The same vehicle as many forms of the braking system is called polymorphism. This example shows us how encapsulation, inheritance and polymorphism are combined.

### **Advantages of OOPs Concept**

Some of the advantages are:

- **Re-usability**

When we say re-usability, it means that “write once, use it multiple times” i.e., reusing some facilities rather than building it again and again, which can be achieved by using class. We can use it n number of times whenever required.

- **Data redundancy**

It is one of the greatest advantages in oops. This is the condition which is created at the data storage when the same piece of data is held at two different places. If we want to use similar functionality in multiple classes, we can just write common class definitions for similar functionalities by inheriting them.

- **Code maintenance**

It is easy to modify or maintain existing code as new objects which can be created with small differences from the existing ones. This helps users from doing rework many times and modifying the existing codes by incorporating new changes to it.

- **Security**

Data hiding and abstraction are used to filter out limited exposure which means we are providing only necessary data to view as we maintain security.

- **Design benefits**

The designers will have a long and more extensive design phase, which results in better designs. At a point of time when the program has reached critical limits, it will be easier to program all non-oops separately.

- **Easy troubleshooting**

Using encapsulation objects is self-constrained. So, if developers face any problem easily it can be solved. And there will be no possibility of code duplicity.

- Flexibility
- Problem-solving

### **Disadvantages of OOPs Concept**

- Effort – A lot of work is put into creating these programs.
- Speed – These programs are slower compared to other programs.
- Size – OOPs programs are bigger when compared to other programs.

### **Interview Questions**

What is polymorphism in OOPs?

In OOPs, Polymorphism is the process that allows us to perform a single action in multiple ways. This occurs when there are several classes related to each other through inheritance. In polymorphism, there are two types. Namely, compile-time polymorphism and runtime polymorphism. It helps us in reducing complexity.

What are the advantages of OOPs?

There are several benefits of implementing OOPs Concepts in Java. A few of the major advantages are as follows: Reusability, Code maintenance, Data Redundancy, Security, Easy troubleshooting, Problem-Solving, Flexibility and Design Benefits. Java OOPs Concepts are one of the core development approaches that is widely accepted.

What are the main features of OOPs?

The main features of OOPs concepts in Java are Classes, Objects, Encapsulation, Data Abstraction, Polymorphism, and Inheritance.

Thank You !

## Agenda

- Deeper understanding of Arrays through problem solving

In the previous session, we've learned about what arrays are and how to use them. Let's now solve some computational problems using arrays to gain a deeper understanding of this data structure.

### Problem 1 - Remove Duplicates from Sorted Array

#### Problem Statement :

Given an integer array `nums` sorted in **non-decreasing order**, remove the duplicates **in-place** such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**.

Since it is impossible to change the length of the array in some languages, you must instead have the result be placed in the **first part** of the array `nums`. More formally, if there are  $k$  elements after removing the duplicates, then the first  $k$  elements of `nums` should hold the final result. It does not matter what you leave beyond the first  $k$  elements.

Return  $k$  *after placing the final result in the first  $k$  slots of `nums`*.

Do **not** allocate extra space for another array. You must do this by **modifying the input array in-place** with  $O(1)$  extra memory.

#### Custom Judge:

The judge will test your solution with the following code:



```
int[] nums = [...]; // Input array
int[] expectedNums = [...]; // The expected answer with correct length
int k = removeDuplicates(nums); // Calls your implementation
assert k == expectedNums.length;
for (int i = 0; i < k; i++) {
 assert nums[i] == expectedNums[i];
}
```

If all assertions pass, then your solution will be **accepted**.

#### Example 1 :



Input: `nums = [1,1,2]`

Output:  $2$ , `nums = [1,2,_]`

Explanation: Your function should return  $k = 2$ , with the first two elements of `nums` being 1 and 2 respectively.

It does not matter what you leave beyond the returned  $k$  (hence they are underscores).

#### Example 2 :



Input: `nums` = [0,0,1,1,1,2,2,3,3,4]

Output: 5, `nums` = [0,1,2,3,4,\_,\_,\_,\_,\_]

Explanation: Your function should return  $k = 5$ , with the first five elements of `nums` being 0, 1, 2, 3, and 4 respectively.

It does not matter what you leave beyond the returned  $k$  (hence they are underscores).

#### Constraints:

- $1 \leq \text{nums.length} \leq 3 * 10^4$
- $100 \leq \text{nums}[i] \leq 100$
- `nums` is sorted in **non-decreasing** order.

#### Solution :

This problem can be easily solved by using extra array, but in question you can see it is clearly mentioned that do **not** allocate extra space for another array.

- So we need to do operation in given array only.
- Here array will be always in sorted and in ascending order.

Suppose, we have given following an **nums** array.



Here number of unique elements are 2, so the output will be 2.



You can see first number will be always be counted as a unique number, so, we need to start comparing with second number which is also “1” over here.



## **unique\_count = 1**

1. Now we are starting to transverse given array, as we have counted first element as a uniqueCount = 1, we will start reading an array from 2nd element, so starting index position will 1, like below

|         |   |   |   |                                                   |
|---------|---|---|---|---------------------------------------------------|
| index = | 0 | 1 | 2 | <b>for (int i = 1; i &lt; nums.length; i++) {</b> |
| nums =  |   |   |   | <b>if(nums[i] != nums[i-1]) {</b>                 |
|         |   |   |   | <b>    nums[uniqueCount] = nums[i];</b>           |
|         |   |   |   | <b>    unique_count += 1</b>                      |
|         |   |   |   | <b>}</b>                                          |
|         |   |   |   | <b>}</b>                                          |

**uniqueCount = 1;**

Now i = 1, so we are comparing first index num[1]’s value = 1 with it’s previous index zero index num[0]’s value = 1.

|                  |   |   |   |                                                                                                                                                             |
|------------------|---|---|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| index =          | 0 | 1 | 2 | <pre>for (int i = 1; i &lt; nums.length; i++) {     if(nums[i] != nums[i-1]) {         nums[uniqueCount] = nums[i];         unique_count += 1     } }</pre> |
| nums =           | 1 | 1 | 2 |                                                                                                                                                             |
| uniqueCount = 1; |   |   |   |                                                                                                                                                             |

As both value are same so our if condition will be failed, now we are moving forward.  $i = 2$

|                  |   |   |   |                                                                                                                                                                      |
|------------------|---|---|---|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| index =          | 0 | 1 | 2 | <pre>for (int i = 1; i &lt; nums.length; i++) {    i = 2     if(nums[i] != nums[i-1]) {         nums[uniqueCount] = nums[i];         unique_count += 1     } }</pre> |
| nums =           | 1 | 1 | 2 |                                                                                                                                                                      |
| uniqueCount = 1; |   |   |   |                                                                                                                                                                      |

Now  $i = 2$ , so we are comparing second index num[2]'s value = 2 with it's previous index first index num[1]'s value = 1.

|                  |   |   |   |                                                                                                                                                                      |
|------------------|---|---|---|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| index =          | 0 | 1 | 2 | <pre>for (int i = 1; i &lt; nums.length; i++) {    i = 2     if(nums[i] != nums[i-1]) {         nums[uniqueCount] = nums[i];         unique_count += 1     } }</pre> |
| nums =           | 1 | 1 | 2 |                                                                                                                                                                      |
| uniqueCount = 1; |   |   |   |                                                                                                                                                                      |

Here, if condition will be **True**. Now we are moving inside if condition.

$\text{nums}[1] = \text{nums}[2] \rightarrow \text{num}[1]$  value will be overwrite by  $\text{num}[2]$ .

|                  |   |   |   |                                                                                      |
|------------------|---|---|---|--------------------------------------------------------------------------------------|
| index =          | 0 | 1 | 2 | for (int i = 1; i < nums.length; i++) {     i = 2                                    |
| nums =           | 1 | 2 | 2 | if(nums[i] != nums[i-1]) {<br>nums[uniqueCount] = nums[i];<br>unique_count += 1<br>} |
| uniqueCount = 1; |   |   |   | }                                                                                    |

Now, we have found another uniqueCount so, we are updating value by +1.

|                  |   |   |   |                                                                                      |
|------------------|---|---|---|--------------------------------------------------------------------------------------|
| index =          | 0 | 1 | 2 | for (int i = 1; i < nums.length; i++) {     i = 2                                    |
| nums =           | 1 | 2 | 2 | if(nums[i] != nums[i-1]) {<br>nums[uniqueCount] = nums[i];<br>unique_count += 1<br>} |
| uniqueCount = 2; |   |   |   | }                                                                                    |

Now, our next iteration, i will be 3, but given array's length is also 3, so our for loop condition will become false,

|                  |   |   |   |                                                                                      |
|------------------|---|---|---|--------------------------------------------------------------------------------------|
| index =          | 0 | 1 | 2 | for (int i = 1; i < nums.length; i++) {     i = 3                                    |
| nums =           | 1 | 2 | 2 | if(nums[i] != nums[i-1]) {<br>nums[uniqueCount] = nums[i];<br>unique_count += 1<br>} |
| uniqueCount = 2; |   |   |   | }                                                                                    |

We can return result as a 2, which is True, you can see, in our given array there are only two, unique elements and array we can get updated.

|                |   |   |   |
|----------------|---|---|---|
| <b>index =</b> | 0 | 1 | 2 |
| <b>nums =</b>  | 1 | 2 | 2 |

**uniqueCount = 2;**

**Code :**



```
class Solution {

 public int removeDuplicates(int[] nums) {

 int uniqueCount = 1;

 for (int i = 1; i < nums.length; i++){

 if(nums[i] != nums[i-1]){

 nums[uniqueCount] = nums[i];
 uniqueCount += 1;
 }
 }

 return uniqueCount;
 }
}
```

### **Problem 2 - Search Insert Position**

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

**Example 1:**



Input: nums = [1,3,5,6], target = 5

Output: 2

**Example 2:**



Input: nums = [1,3,5,6], target = 2

Output: 1

**Example 3:**



Input: nums = [1,3,5,6], target = 7

Output: 4

**Solution :**

**Approach:** Follow the steps below to solve the problem:

- Iterate over every element of the array **nums[]** and search for **target**.
- If any array element is found to be equal to **target**, then print index of **target**.
- Otherwise, if any array element is found to be greater than **target**, print that index as the insert position of **target**. If no element is found to be exceeding **target**, **target** must be inserted after the last array element.

**Code :**



```
class Solution {
```

```
 public int searchInsert(int[] nums, int target) {
 int n = nums.length;
 // Traverse the array
 for(int i = 0; i < n; i++){
 // If K is found
 if (arr[i] == K)
```

```

 return i;

 // If current array element

 // exceeds K

 else if (arr[i] > K)

 return i;

 }

 // If all elements are smaller

 // than K

 return n;

}

}

```

### **Problem 3 - Max Sum Contiguous Subarray**

#### **Problem Statement :**

Find the **contiguous** subarray within an array, **A** of length **N** which has the **largest sum**.

#### **Input Format:**

The first and the only argument contains an integer array, A.

#### **Output Format:**

Return an integer representing the maximum possible sum of the contiguous subarray.

#### **Constraints:**

$1 \leq N \leq 1e6$   $-1000 \leq A[i] \leq 1000$

#### **For example:**

##### **Input 1:**

$A = [1, 2, 3, 4, -10]$  Output 1:

##### **Explanation 1:**

The subarray  $[1, 2, 3, 4]$  has the maximum possible sum of 10.

##### **Input 2:**

$A = [-2, 1, -3, 4, -1, 2, 1, -5, 4]$  Output 2:

##### **Explanation 2:**

The subarray  $[4, -1, 2, 1]$  has the maximum possible sum of 6.

## Solution

One of the most efficient ways to solve the given problem is using **Kadane's Algorithm**. The idea of \*\*\*\*Kadane's algorithm \*\*\*\*is to maintain a variable max\_ending\_here \*\*\*\*that stores the maximum sum contiguous subarray ending at current index and a variable max\_so\_far stores the maximum sum of contiguous subarray found so far, Everytime there is a positive-sum value in max\_ending\_here compare it with max\_so\_far and update max\_so\_far \*\*if it is greater than max\_so\_far. Pseudo-code to implement this approach :



Initialize:

```
max_so_far = INT_MIN
```

```
max_ending_here = 0
```

Loop for each element of the array

```
(a) max_ending_here = max_ending_here + a[i]
```

```
(b) if(max_so_far < max_ending_here)
```

```
 max_so_far = max_ending_here
```

```
(c) if(max_ending_here < 0)
```

```
 max_ending_here = 0
```

```
return max_so_far
```

**Note:** The above algorithm only works if and only if at least one positive number should be present otherwise it does not work i.e if an Array contains all negative numbers it doesn't work.

An implementation of this algorithm in Java:



```
public int maxSubArray(int[] A) {
 int size = A.length;
 int max_so_far = Integer.MIN_VALUE
 int max_ending_here = 0;

 for (int i = 0; i < size; i++) {
```

```

 max_ending_here = max_ending_here + A[i];

 if (max_so_far < max_ending_here)
 max_so_far = max_ending_here;

 if (max_ending_here < 0)
 max_ending_here = 0;
 }

 return max_so_far;
}

```

#### **Problem 4 - Merge Sorted Array**

You are given two integer arrays `nums1` and `nums2`, sorted in **non-decreasing order**, and two integers `m` and `n`, representing the number of elements in `nums1` and `nums2` respectively.

**Merge** `nums1` and `nums2` into a single array sorted in **non-decreasing order**.

The final sorted array should not be returned by the function, but instead be *stored inside the array* `nums1`. To accommodate this, `nums1` has a length of  $m + n$ , where the first  $m$  elements denote the elements that should be merged, and the last  $n$  elements are set to 0 and should be ignored. `nums2` has a length of  $n$ .

#### **Example 1:**



Input: `nums1` = [1,2,3,0,0,0],  $m$  = 3, `nums2` = [2,5,6],  $n$  = 3

Output: [1,2,2,3,5,6]

Explanation: The arrays we are merging are [1,2,3] and [2,5,6].

The result of the merge is [1,2,2,3,5,6] with the underlined elements coming from `nums1`.

#### **Example 2:**



Input: `nums1` = [1],  $m$  = 1, `nums2` = [],  $n$  = 0

Output: [1]

Explanation: The arrays we are merging are [1] and [].

The result of the merge is [1].

#### **Example 3:**



Input:  $\text{nums1} = [0]$ ,  $m = 0$ ,  $\text{nums2} = [1]$ ,  $n = 1$

Output: [1]

Explanation: The arrays we are merging are [] and [1].

The result of the merge is [1].

Note that because  $m = 0$ , there are no elements in  $\text{nums1}$ . The 0 is only there to ensure the merge result can fit in  $\text{nums1}$ .

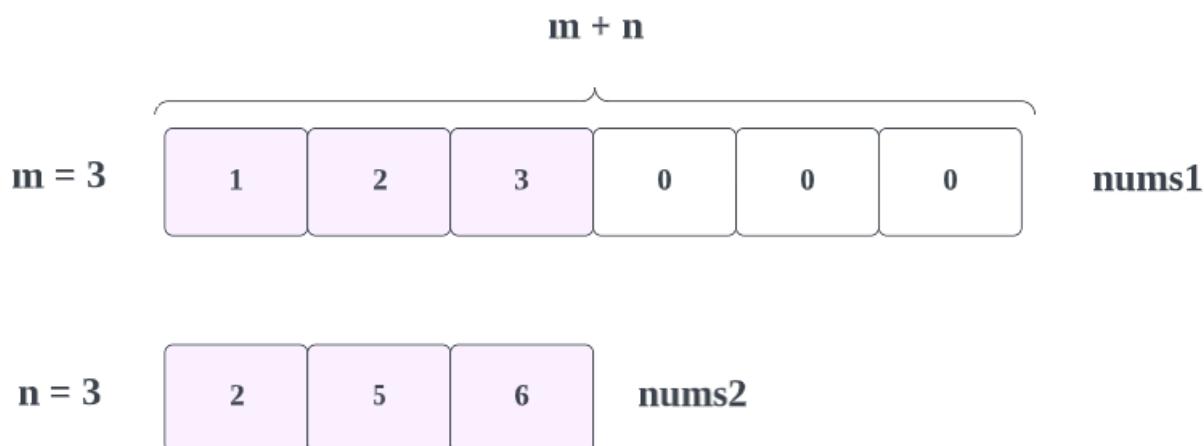
#### Constraints:

- $\text{nums1.length} == m + n$
- $\text{nums2.length} == n$
- $0 \leq m, n \leq 200$
- $1 \leq m + n \leq 200$
- $109 \leq \text{nums1}[i], \text{nums2}[j] \leq 109$

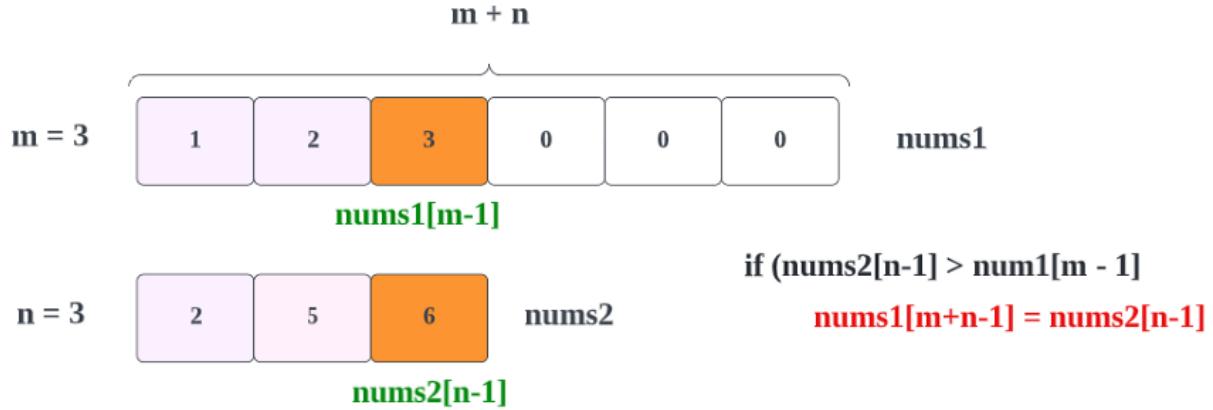
#### Solution

Lets get solution by an example.

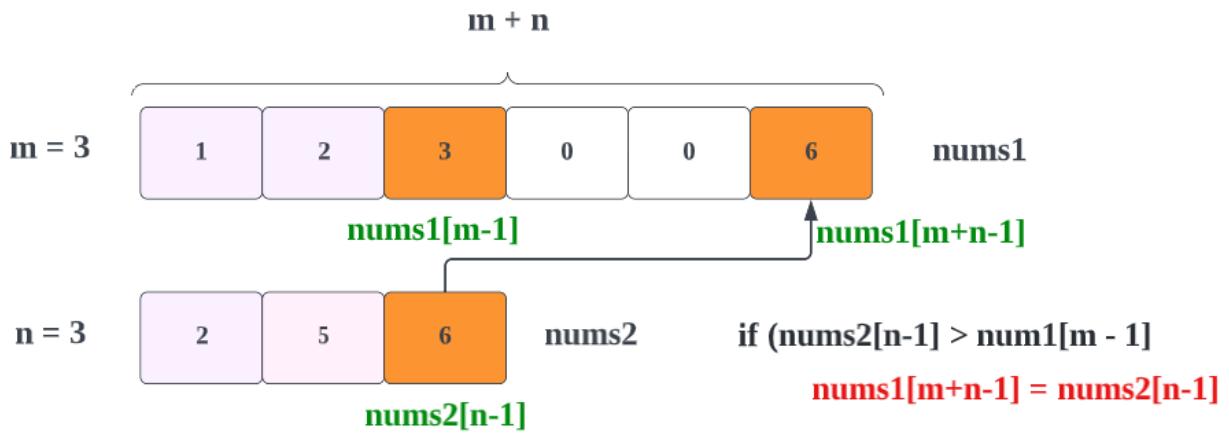
Suppose we have given two sorted arrays like below,



Over here we will compare, last two digit  $\text{nums1}[m-1] == \text{nums2}[n-1]$

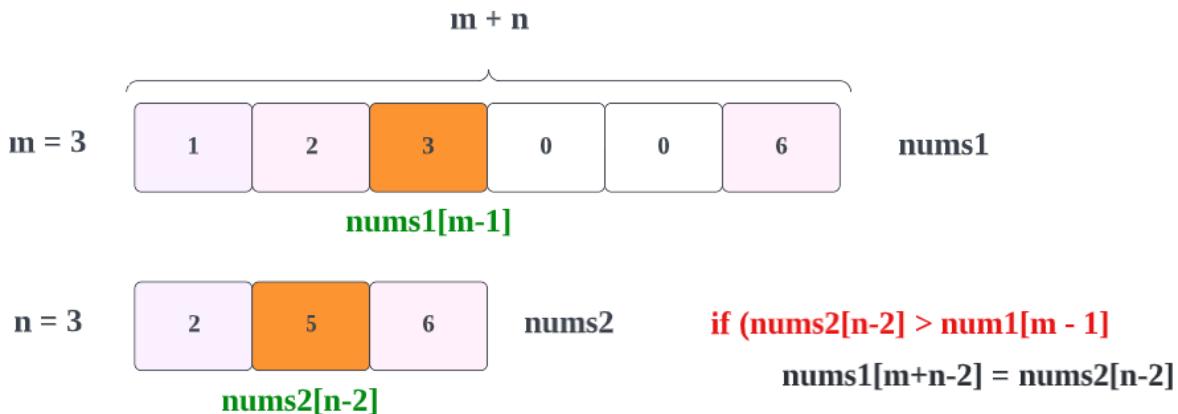


Whatever the biggest value is, will be append to  $\text{nums1}[m+n-1]$ ,

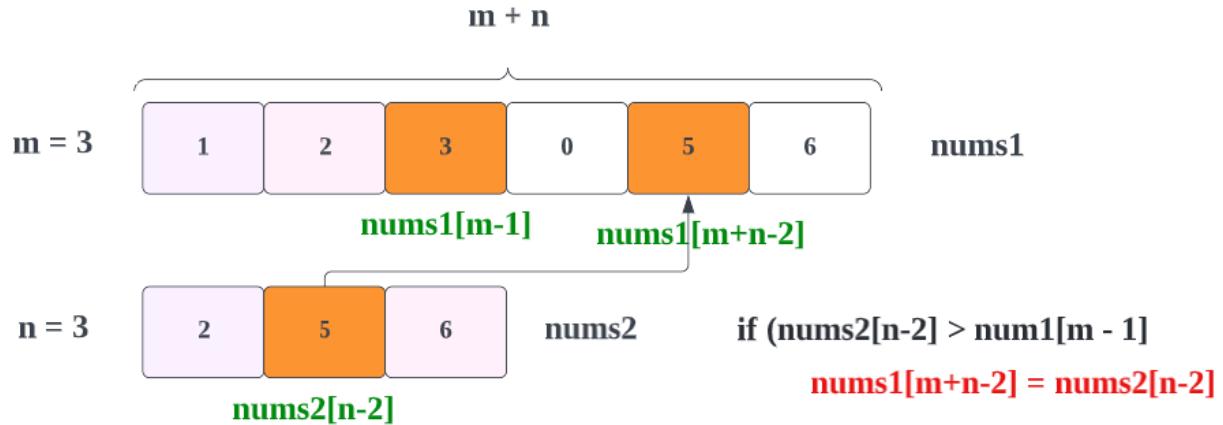


Because we found  $\text{nums2}[n-1]$  greater value, so we will move to it's previous element which is  $\text{nums2}[n-2]$  and compare with  $\text{nums1}[m-1]$ .

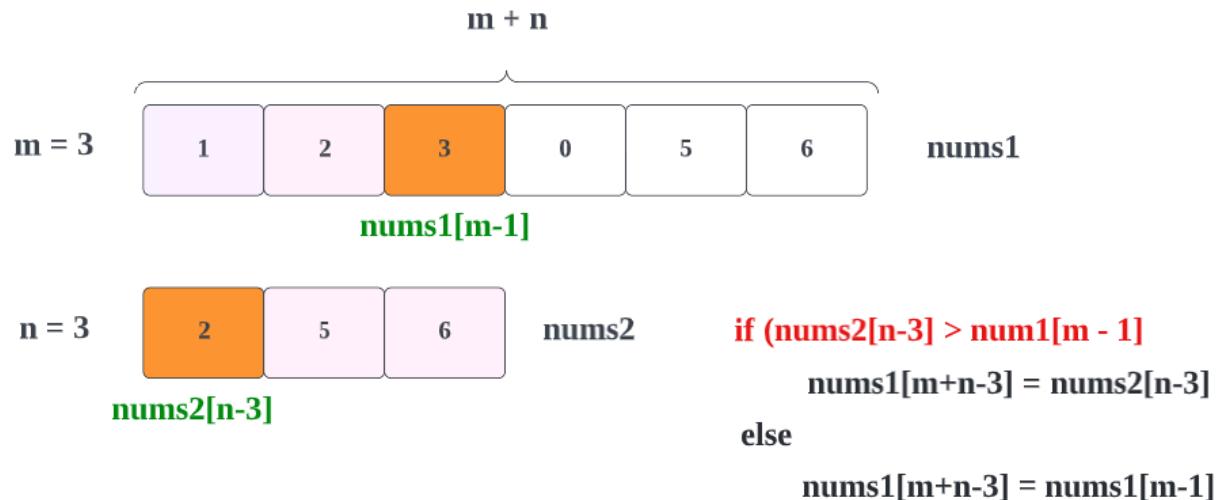
Also,  $\text{nums}[m+n-1]$  is already filled, so we will move to it's previous element, which is  $\text{nums}[m+n-2]$



As previous step, whatever the biggest value is, will be append to  $\text{nums1}[m+n-2]$ ,

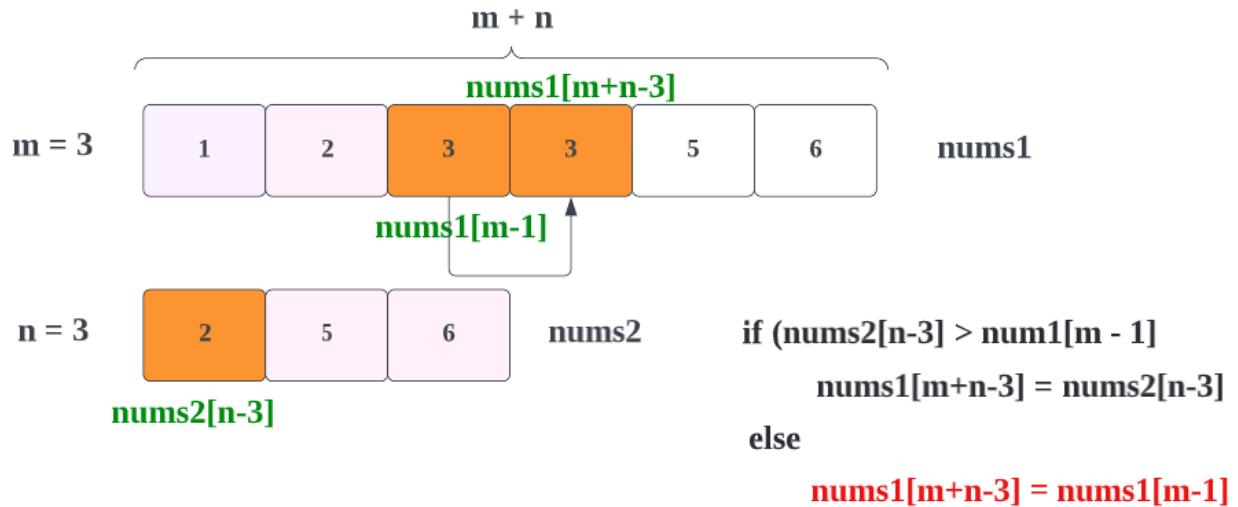


Here  $nums2[n-2]$  was greater value, so we will move to it's previous element which is  $nums2[n-3]$  and compare with  $nums1[m-1]$ .



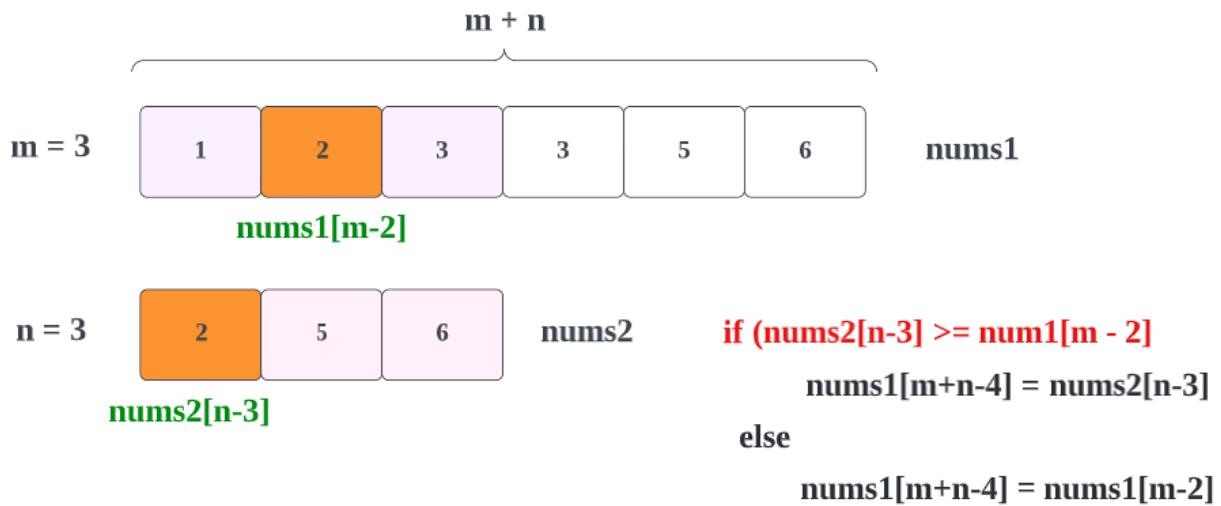
But here,  $nums2[n-3] > nums1[m-1] \Rightarrow 2 > 3$ , condition becomes **False**.

So, as previous step, whatever the biggest value is, will be append to  $nums1[m+n-3]$ ,



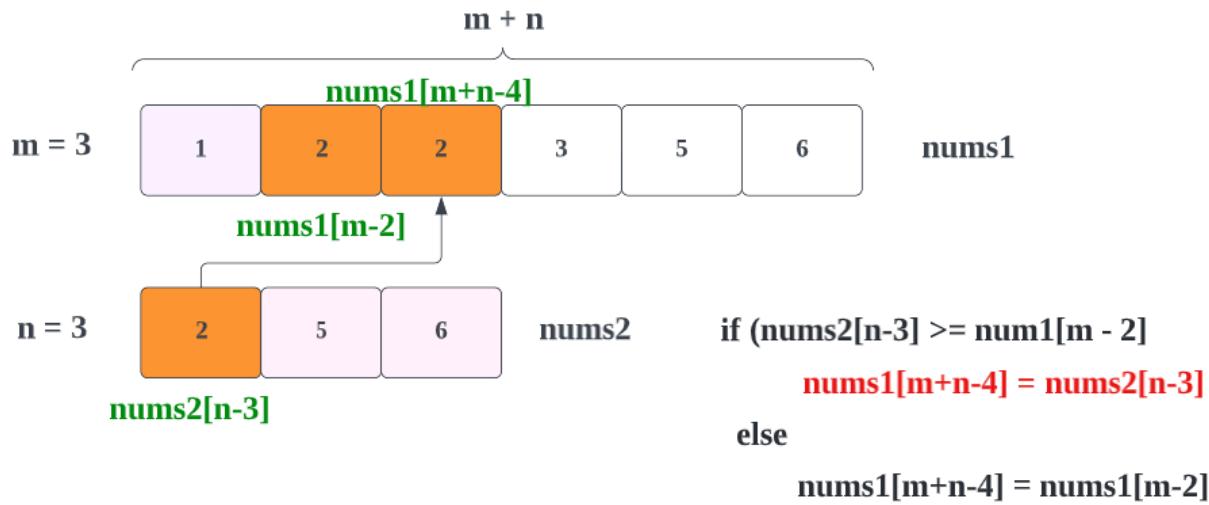
Here  $\text{nums1}[m-1]$  was \*\*\*\*greater value, so we will move to it's previous element which is  $\text{nums1}[m-2]$  and compare with  $\text{nums2}[n-3]$ .

Also,  $\text{nums}[m+n-3]$  is already filled, so we will move to it's previous element, which is  $\text{nums}[m+n-4]$



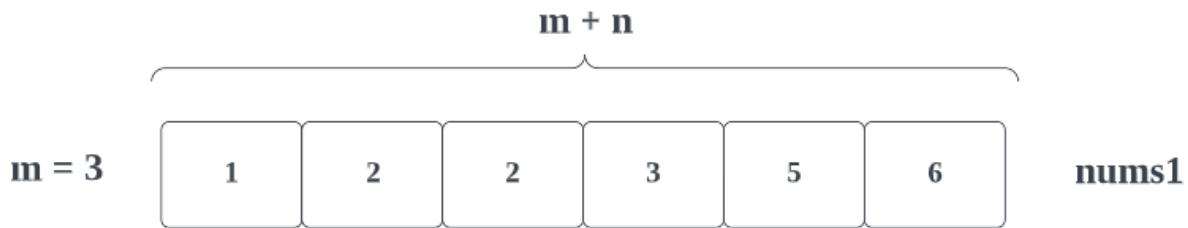
Here,  $\text{nums2}[n-3] \geq \text{nums1}[m-2] \Rightarrow 2 == 2**,**$  condition becomes True.

So, as previous step, what ever the biggest value is, will be append to  $\text{nums1}[m+n-4]$ ,



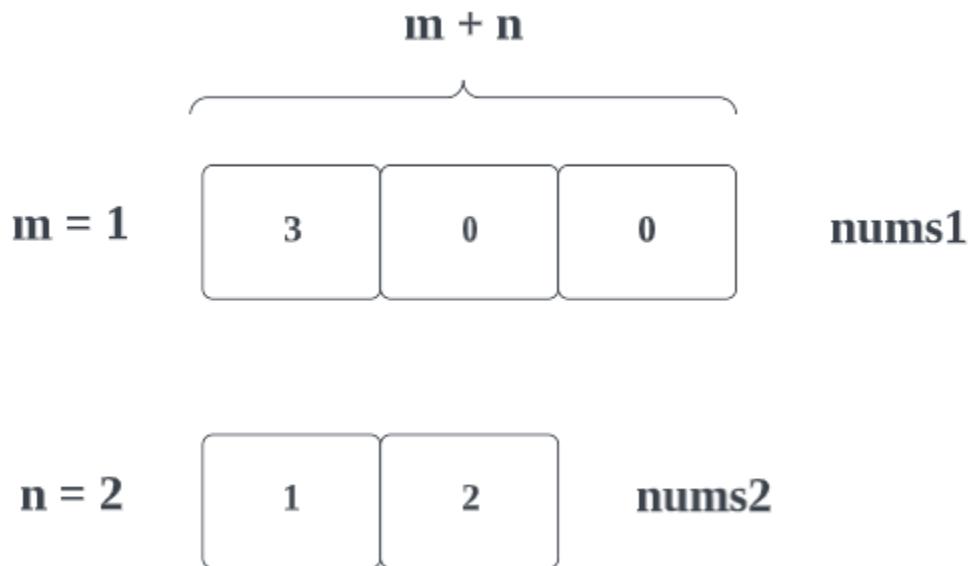
Here **nums2[n-3]** was \*\*\*\*greater value, so we will move to it's previous element which is **nums2[n-4]**, but **nums2 length is ended**. So, we can not go to it's previous element and we must need to avoid those steps by **if condition**.

So, we got our result which is below,

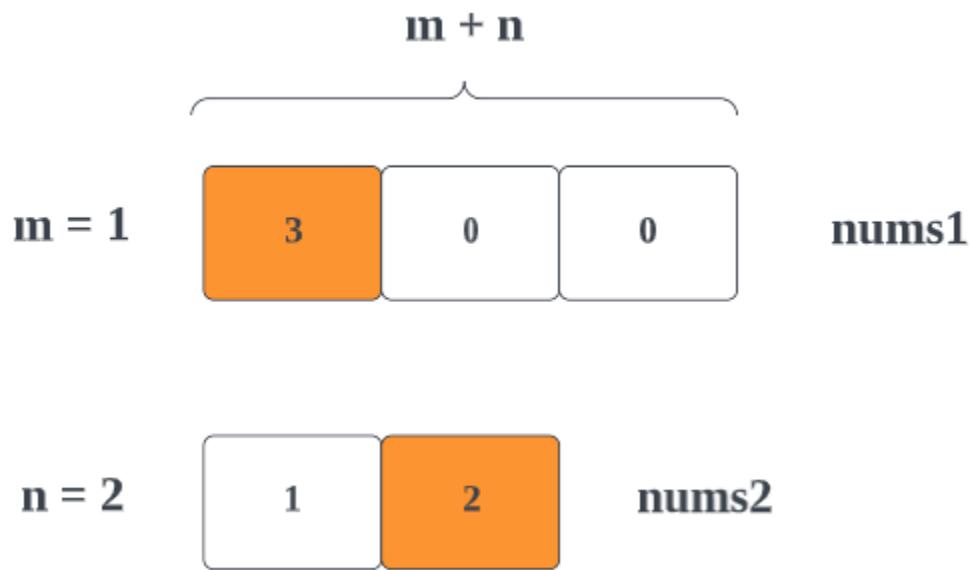


Note: In above example, you can see that **nums2** reached to **0** earlier then **nums1**. But what if **num1** reached to 0 earlier then **nums2**.

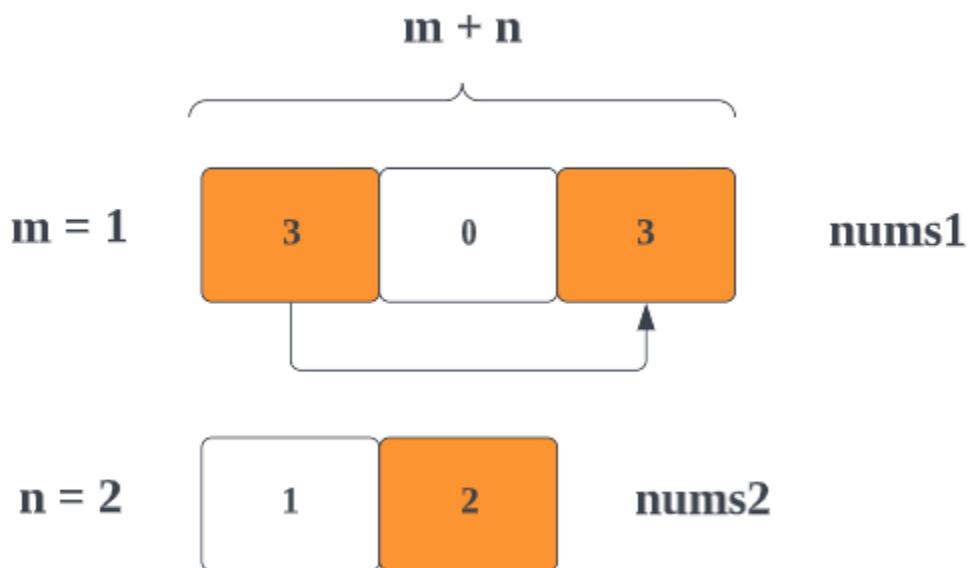
Like below,



In this case, first we will go step by step, like above we have seen,

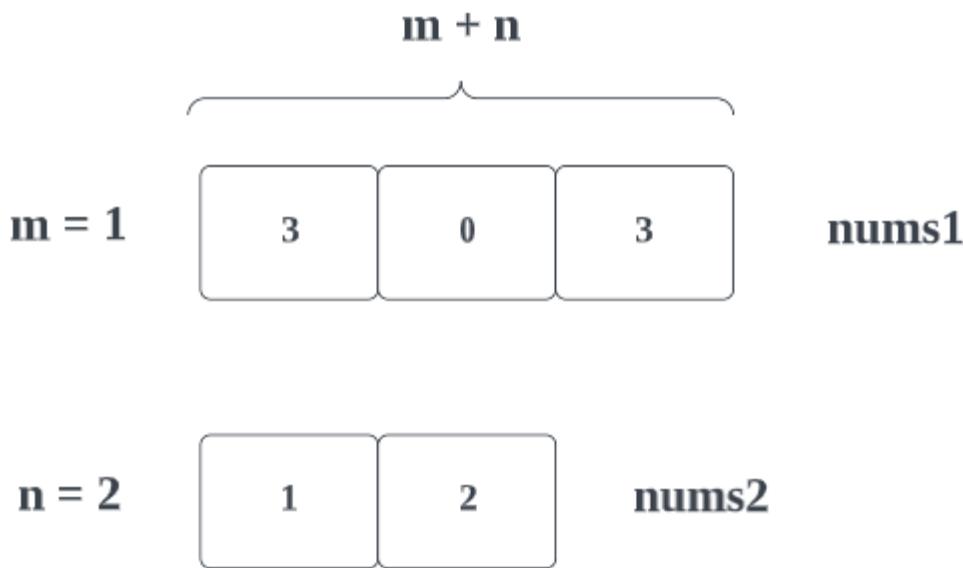


Here,  $nums1[m-1]$  which is 3 is bigger so,

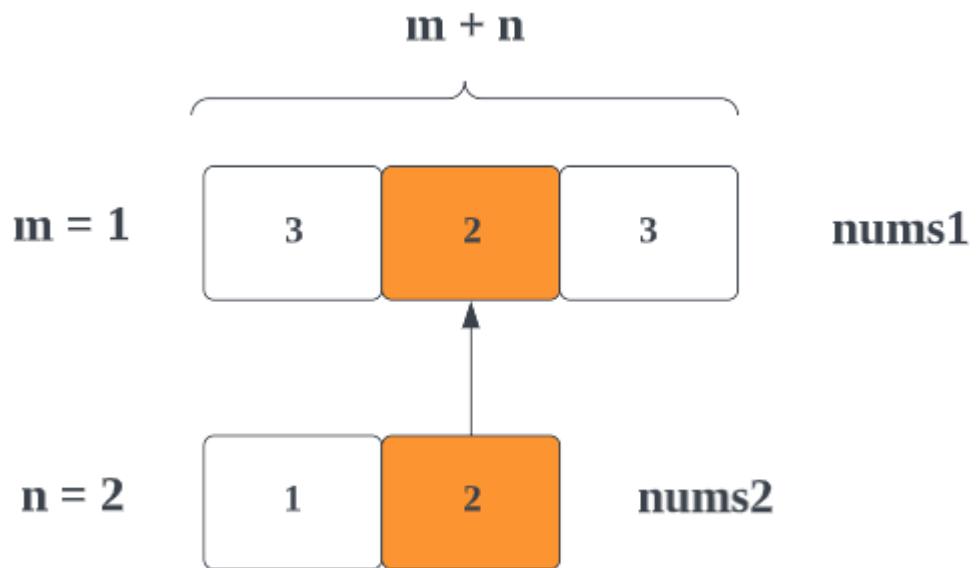


Here  $\text{nums1}[m-1]$  was \*\*\*\*greater value, so we will move to it's previous element which is  $\text{nums1}[m-2]$ , but  $\text{nums1}$  length is ended. So, we can not go to it's previous element and we must need to avoid those steps by **if condition**.

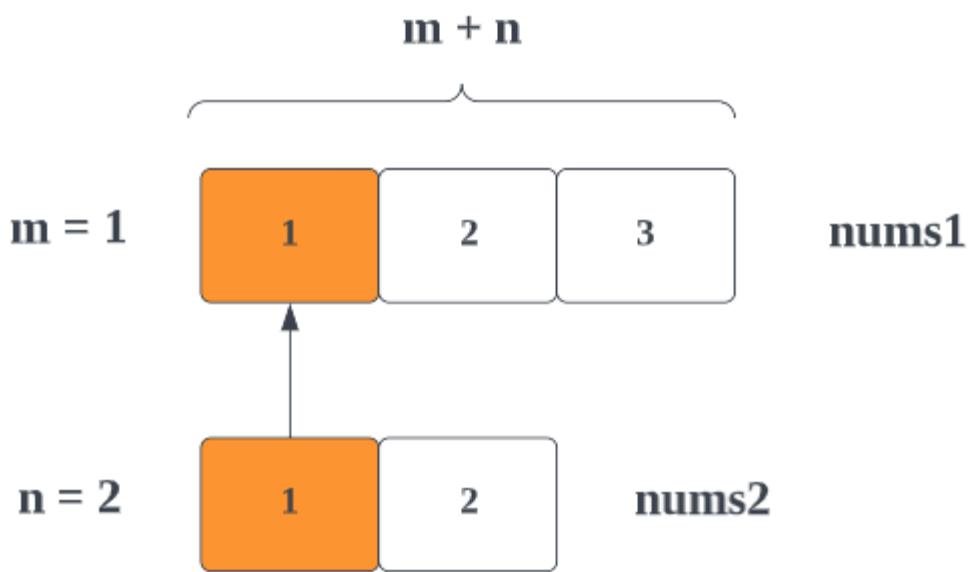
Here, we can not stop whole process, as we have not traversed all elements of  $\text{nums2}$ , and also, we do not get proper num1.



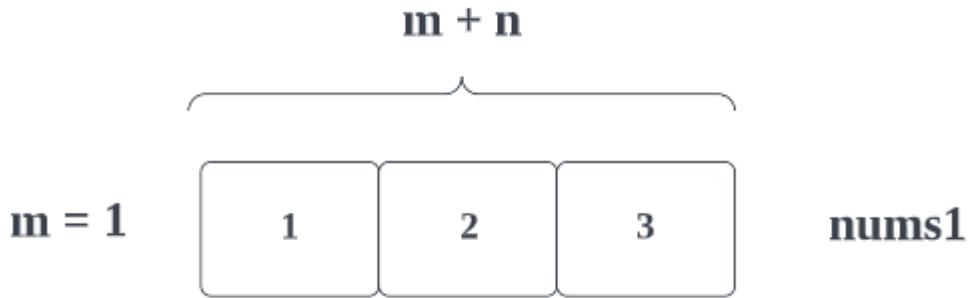
So, at the end, we need to place one by one of  $\text{nums2}$ 's element to remaining  $\text{nums1}$ 's element, like below,



Next,



Now, we get proper result,



Let's see the full source code:



```
class Solution {

 public void merge(int[] nums1, int m, int[] nums2, int n) {

 int i = m - 1;
 int j = n - 1;
 int k = m + n - 1;

 while (k >= 0) {

 if (i >= 0 && j >= 0) {

 if (nums1[i] >= nums2[j]) {
 nums1[k--] = nums1[i--];
 } else {
 nums1[k--] = nums2[j--];
 }
 } else if (j >= 0) {
 nums1[k--] = nums2[j--];
 }
 }
 }
}
```

```

 } else {
 break;
 }

}

}

```

### **Problem 5 - Best Time to Buy and Sell Stock**

You are given an array prices where prices[i] is the price of a given stock on the ith day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

#### **Example 1:**



Input: prices = [7,1,5,3,6,4]

Output: 5

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

#### **Example 2:**



Input: prices = [7,6,4,3,1]

Output: 0

Explanation: In this case, no transactions are done and the max profit = 0.

#### **Constraints:**

- $1 \leq \text{prices.length} \leq 105$
- $0 \leq \text{prices}[i] \leq 104$

#### **Solution**

Here, we will try to find **min value** and **max difference value**.

Lets understand step by step,

Suppose, we have given below example,

Here, we will take **two variables**,

```
● ● ●
//In constraints, it is mentioned
// 0 <= prices[i] <= 10^4
int min = 10000;

//If no difference found, maxDiff will be 0
int maxDiff = 0;
```

Step — 1. for first value `prices[0]` which is 7,

for **min**, we will compare **smaller value** and take that value,

**Min = 10000**

**Max = 0**



**Min**

**7 is Smaller than 10000**

### **Max Difference**

Here we find 7 is smaller than 10000. So smaller is 7.

**Min = 7**

**maxDiff = 0**



**Min**

**7**

**7 is Smaller than 10000**

### Max Difference

Now for, **maxDiff** =>  $\text{prices}[0]-\text{min} \Rightarrow 7-7 \Rightarrow 0$ ,

0 is **similar** with current **maxDiff**, so it is 0.

**Min = 7**

**maxDiff = 0**



**Min**

**7**

**Max Difference**

**0**

Step — 2. Now, for second  $\text{prices}[1] \Rightarrow 1$ .

**Min = 7**

**maxDiff = 0**



**Min**

**7**

**Max Difference**

**0**

For min:- 1 is smaller than 7, so min will be 1.

**Min = 1**

**maxDiff = 0**



**Min**

**7**

**1**

**Max Difference**

**0**

Now for, **maxDiff** :  $\text{prices}[1]-\text{min} \Rightarrow 1-1 \Rightarrow 0$ ,

0 is **similar** with current **maxDiff**, so it is 0.

**Min = 1**

**maxDiff = 0**

|                       |   |   |   |   |   |
|-----------------------|---|---|---|---|---|
|                       | 7 | 1 | 5 | 3 | 6 |
| <b>Min</b>            | 7 | 1 |   |   |   |
| <b>Max Difference</b> | 0 | 0 |   |   |   |

Step — 3. Now, for third prices[2] => 5.

**Min = 1**

**maxDiff = 0**

|                       |   |   |   |   |   |
|-----------------------|---|---|---|---|---|
|                       | 7 | 1 | 5 | 3 | 6 |
| <b>Min</b>            | 7 | 1 |   |   |   |
| <b>Max Difference</b> | 0 | 0 |   |   |   |

For Min:- 1 is not smaller than 5. So, we will take minimum value only so, it will be 1.

**Min = 1**

**maxDiff = 0**

|   |   |   |   |   |
|---|---|---|---|---|
| 7 | 1 | 5 | 3 | 6 |
|---|---|---|---|---|

|            |   |   |   |  |
|------------|---|---|---|--|
| <b>Min</b> | 7 | 1 | 1 |  |
|------------|---|---|---|--|

**Max Difference**

|   |   |  |
|---|---|--|
| 0 | 0 |  |
|---|---|--|

Now for, **maxDiff ,prices[2]-min => 5-1 => 4\*\*,\*\***

We get **4**, and **compare** with **maxDiff** which is **0**

Bigger one is **4**, so **maxDiff** will be **4**.

**Min = 1**

**maxDiff = 4**

|   |   |   |   |   |
|---|---|---|---|---|
| 7 | 1 | 5 | 3 | 6 |
|---|---|---|---|---|

|            |   |   |   |  |
|------------|---|---|---|--|
| <b>Min</b> | 7 | 1 | 1 |  |
|------------|---|---|---|--|

**Max Difference**

|   |   |   |  |
|---|---|---|--|
| 0 | 0 | 4 |  |
|---|---|---|--|

Step — 4. Now, for third **prices[3] => 3.**

**Min = 1**

**maxDiff = 4**

|   |   |   |   |   |
|---|---|---|---|---|
| 7 | 1 | 5 | 3 | 6 |
|---|---|---|---|---|

**Min**

|   |   |   |
|---|---|---|
| 7 | 1 | 1 |
|---|---|---|

**Max Difference**

|   |   |   |
|---|---|---|
| 0 | 0 | 4 |
|---|---|---|

For min:- 3 is not smaller than 1. So we will keep it 1.

**Min = 1**

**maxDiff = 4**

|   |   |   |   |   |
|---|---|---|---|---|
| 7 | 1 | 5 | 3 | 6 |
|---|---|---|---|---|

**Min**

|   |   |   |   |
|---|---|---|---|
| 7 | 1 | 1 | 1 |
|---|---|---|---|

**Max Difference**

|   |   |   |
|---|---|---|
| 0 | 0 | 4 |
|---|---|---|

Now for, **maxDiff**,

**prices[3]-min => 4-1 => 3\*\*\*, \*\***

We get **3**, and **compare** with **maxDiff** which is **4**,

We will take **max value** which is **4**.

**Min = 1**

**maxDiff = 4**

|                       |   |   |   |   |
|-----------------------|---|---|---|---|
| 7                     | 1 | 5 | 3 | 6 |
| <b>Min</b>            | 7 | 1 | 1 | 1 |
| <b>Max Difference</b> | 0 | 0 | 4 | 4 |

Step — 5. Now, for third prices[4] => 6.

**Min = 1**

**maxDiff = 4**

|                       |   |   |   |   |
|-----------------------|---|---|---|---|
| 7                     | 1 | 5 | 3 | 6 |
| <b>Min</b>            | 7 | 1 | 1 | 1 |
| <b>Max Difference</b> | 0 | 0 | 4 | 4 |

For min:- 6 is not smaller than 1. So we will keep it 1.

**Min = 1**

**maxDiff = 4**

|                       |   |   |   |   |
|-----------------------|---|---|---|---|
| 7                     | 1 | 5 | 3 | 6 |
| <b>Min</b>            | 7 | 1 | 1 | 1 |
| <b>Max Difference</b> | 0 | 0 | 4 | 4 |

Now for, **maxDiff**,

**prices[4]-min => 6-1 => 5\*\*,\*\***

We get 5, and **compare** with **maxDiff** which is 4,

We will take max value which is 5.

**Min = 1**

**maxDiff = 5**

|                       |   |   |   |   |
|-----------------------|---|---|---|---|
| 7                     | 1 | 5 | 3 | 6 |
| <b>Min</b>            | 7 | 1 | 1 | 1 |
| <b>Max Difference</b> | 0 | 0 | 4 | 4 |
|                       |   |   |   | 5 |

So, after end of an array, we will get maxDiff as a 5.

We will **return that maxDiff**, this will be our answer.

Now, lets see full source code.



```
class Solution {

 public int maxProfit(int[] prices) {

 //In constraints it is given that
 //0 <= prices[i] <= 10^4
 int min = 10000;

 //Profit will be 0, if no transaction are done.
 int maxDiff = 0;

 int size = prices.length;

 for (int i = 0; i < size; i++){
 //We need to find Min value
 min = Math.min(prices[i], min);
 //We need to find maxProfit which is Difference between
 //currentPrice - min, then compare with maxDiff
 maxDiff = Math.max(prices[i] - min, maxDiff);
 }
 return maxDiff;
 }
}
```

## Interview Questions

We know that Arrays are objects so why cannot we write strArray.length()?

We cannot write strArray.length() because length is not a method, it's a data item of an array. We can use the methods of Object like toString() and hashCode() against Array.

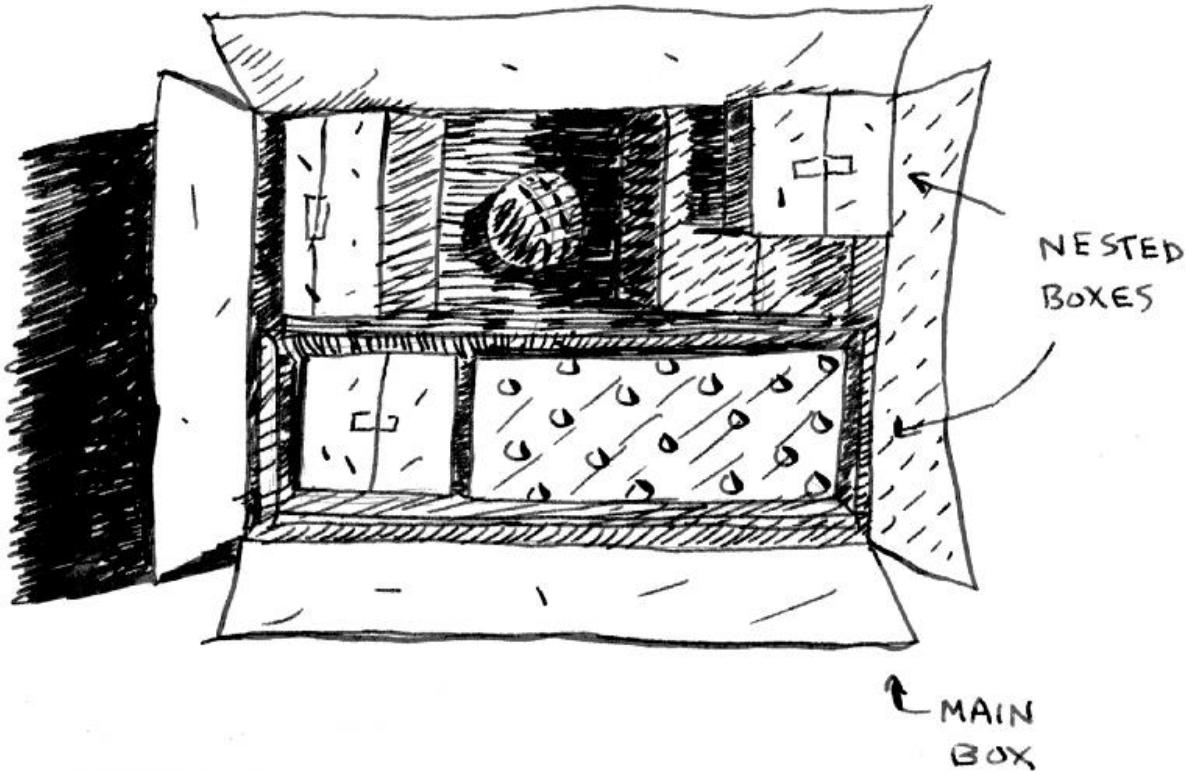
How do you remove a particular element from an array?

- You can't directly remove elements from the original array, as arrays are fixed sets and the size can't change therefore the interviewer is looking for you to suggest an alternate solution and address the issue that the question presents. The best way to remove an element would be to create a new array. In this array, you could include copies of elements of the first array and omit only the element you want to remove.
- Another approach is searching for the target element in the array and then moving all the elements in one position back which are on the right side of the target element.

Thank You !

## Tech it easy!!

Recursion can be tough to understand — especially for new programmers. In its simplest form, a recursive function is one that calls itself. Let me try to explain with an example.

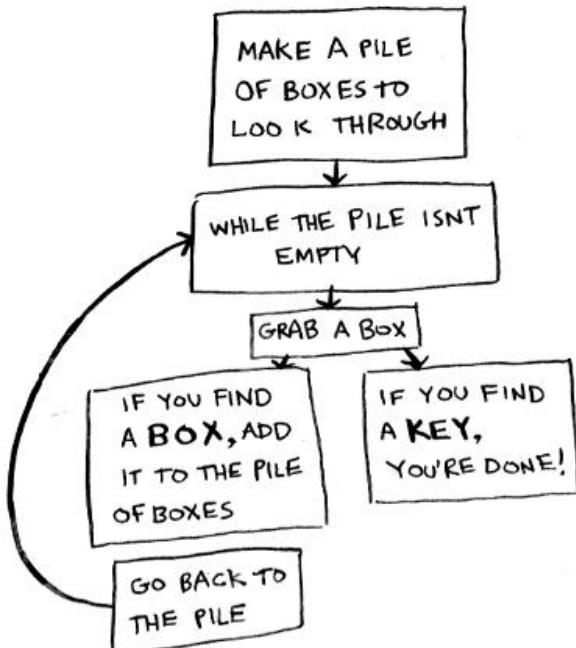


Imagine you go to open your bedroom door and it's locked. Your three-year-old son pops in from around the corner and lets you know he hid the only key in a box. ("Just like him," you think.) You're late for work and you really need to get in the room to get your shirt.

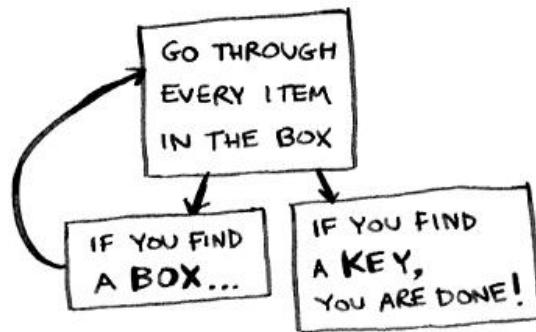
You open the box only to find... more boxes. Boxes inside of boxes. And you don't know which one has the key! You need to get that shirt soon, so you have to think of a good algorithm to find that key.

There are two main approaches to create an algorithm for this problem: iterative and recursive. Here are both approaches as flow charts:

## Iterative Approach



## Recursive Approach



- The first approach uses a while loop. While the pile isn't empty, grab a box and look through it.
- The second way uses recursion. Remember, recursion is where a function calls itself.

Both approaches accomplish the same thing. The main purpose for using the recursive approach is that once you understand it, it can be clearer to read. There is actually no performance benefit to using recursion. The iterative approach with loops can sometimes be faster. But mainly the simplicity of recursion is sometimes preferred.

### The Technical part!

Recursion is a process by which a function or a method calls itself again and again. This function that is called again and again either directly or indirectly is called the “recursive function”.

We will see various examples to understand recursion. Now let's see the syntax of recursion.

### Recursion Syntax:

There are two main requirements of a recursive function:

- **precondition** – the function returns a value when a certain condition is satisfied, without a further recursive call
- **The Recursive Call** – the function calls itself with an *input* which is a step closer to the stop condition

Note that a precondition is necessary for any recursive method as, if we do not break the recursion then it will keep on running infinitely and result in a stack overflow.

**The general syntax of recursion is as follows:**



```
methodName (T parameters...)
```

```
{
```

```
 if (precondition == true)
```

```
 //precondition or base condition
```

```
{
```

```
 return result;
```

```
}
```

```
return methodName (T parameters...);
```

```
//recursive call
```

```
}
```

Note that the precondition is also called base condition. We will discuss more about the base condition in the next section.

### Recursion Base Condition

While writing the recursive program, we should first provide the solution for the base case. Then we express the bigger problem in terms of smaller problems.

As an **example**, we can take a classic problem of calculating the factorial of a number. Given a number n, we have to find a factorial of n denoted by n!

Now let's implement the program to calculate the n factorial (n!) using recursion.



```
public class Main{
```

```
 static int fact(int n)
```

```
{
```

```
 if (n == 1)
```

```
 // base condition
```

```
 return 1;
```

```

 else
 return n*fact(n-1);

 }

 public static void main(String[] args) {
 int result = fact(10);

 System.out.println("10! = " + result);
 }
}

```

In this program, we can see that the condition ( $n \leq 1$ ) is the base condition and when this condition is reached, the function returns 1. The else part of the function is the recursive call. But every time the recursive method is called,  $n$  is decremented by 1.

Thus we can conclude that ultimately the value of  $n$  will become 1 or less than 1 and at this point, the method will return value 1. This base condition will be reached and the function will stop. Note that the value of  $n$  can be anything as long as it satisfies the base condition.

### **How to find if a condition can be solved using Recursion?**

Finding whether a programming problem can be solved using Recursion is a skill, not everybody sees issues in terms of Recursion. The best thing is to break the problem into a smaller set and see if the smaller problem is the same as the original problem or not like in order to calculate a factorial of 5, does it help to calculate a factorial of 4? This may be a guide to see if Recursion can be used or not.

Since a smaller linked list is a linked list, a smaller tree is a tree itself, problems like reversing the linked list, traversing the tree, etc. can be solved using Recursion in Java.

### **Steps to solve a problem using Recursion**

Once you have identified that a coding problem can be solved using Recursion, You are just two steps away from writing a recursive function.

1. Find the base case
2. Finding how to call the method and what to do with the return value.

For example, when you calculate factorial, the base case is  $\text{factorial}(0)$  which is 1, you mean you know the answer so you can directly return it and from there onwards recursion will unroll and calculate factorial for the given number.

Once you have done that, you need to find a way to call the recursive method and what to do with the result returned by the method. This will be more clear when you will do some examples of Recursion in Java.

## **Stack Overflow Error In Recursion**

We are aware that when any method or function is called, the state of the function is stored on the stack and is retrieved when the function returns. The stack is used for the recursive method as well.

But in the case of recursion, a problem might occur if we do not define the base condition or when the base condition is somehow not reached or executed. If this situation occurs then the stack overflow may arise.

**Let's consider the below example of factorial notation.**

Here we have given a wrong base condition, n==100.



```
public class Main
{
 static int fact(int n)
 {
 if (n == 100)
 // base condition resulting in stack overflow
 return 1;
 else
 return n*fact(n-1);
 }

 public static void main(String[] args) {
 int result = fact(10);

 System.out.println("10! = " + result);
 }
}
```

So when  $n > 100$  the method will return 1 but recursion will not stop. The value of  $n$  will keep on decrementing indefinitely as there is no other condition to stop it. This will go on till stack overflows.

Another case will be when the value of  $n < 100$ . In this case, as well the method will never execute the base condition and result in a stack overflow.

## **Recursion Types**

**Recursion is of two types based on when the call is made to the recursive method.**

### **1) Tail Recursion**

When the call to the recursive method is the last statement executed inside the recursive method, it is called “Tail Recursion”.

In tail recursion, the recursive call statement is usually executed along with the return statement of the method.

**The general syntax for tail recursion is given below:**



```
methodName (T parameters...){
{
 if (base_condition == true)
 {
 return result;
 }
 return methodName (T parameters ...) //tail recursion
}
```

### **2) Head Recursion**

Head recursion is any recursive approach that is not a tail recursion. So even general recursion is ahead recursion.

**Syntax of head recursion is as follows:**



```
methodName (T parameters...){
 if (some_condition == true)
 {
 return methodName (T parameters...);
 }
 return result;
}
```

Now let us consider another example for **Factorial of a Number Using Recursion** and see how recursion actually works



```
class Factorial {

 static int factorial(int n) {
 if (n != 0) // termination condition
 return n * factorial(n-1); // recursive call
 else
 return 1;
 }

}
```

```
public static void main(String[] args) {
 int number = 4, result;
 result = factorial(number);
 System.out.println(number + " factorial = " + result);
}
}
```

**Output:**



4 factorial = 24

In the above example, we have a method named factorial(). The factorial() is called from the main() method. with the number variable passed as an argument.

Here, notice the statement,



```
return n * factorial(n-1);
```

The factorial() method is calling itself. Initially, the value of n is 4 inside factorial(). During the next recursive call, 3 is passed to the factorial() method. This process continues until n is equal to 0.

When n is equal to 0, the if statement returns false hence 1 is returned. Finally, the accumulated result is passed to the main() method.

### **Working of Factorial Program**

The image below will give you a better idea of how the factorial program is executed using recursion.

```

public static void main(args: Array<String>) {
 ...
 result = factorial(number) ← 4
 ...
}

static int factorial(int n) {
 4
 if (n != 0)
 return n * factorial(n-1) ← 3
 else 4
 return 1
 }
}

static int factorial(int n) {
 3
 if (n != 0)
 return n * factorial(n-1) ← 2
 else 3
 return 1
 }
}

static int factorial(int n) {
 2
 if (n != 0)
 return n * factorial(n-1) ← 1
 else 2
 return 1
 }
}

static int factorial(int n) {
 1
 if (n != 0)
 return n * factorial(n-1) ← 1
 else 1
 return 1
 }
}

static int factorial(int n) {
 0
 if (n != 0)
 return n * factorial(n-1)
 else
 return 1
}

```

The diagram illustrates the execution flow of the factorial function through five levels of recursion. Each level is represented by a dashed box containing a stack frame with a value. The values represent the current state of the factorial calculation:

- Level 1:** The initial call to `factorial(number)` with value **4**.
- Level 2:** The first recursive call with value **3**, resulting in **returns 3\*2**.
- Level 3:** The second recursive call with value **2**, resulting in **returns 2\*1**.
- Level 4:** The third recursive call with value **1**, resulting in **returns 1\*1**.
- Level 5:** The fourth recursive call with value **0**, resulting in **returns 1**.

## Recursion Vs Iteration In Java

| Recursion                                                                                    | Iteration                                                                                                |
|----------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|
| Recursion is a process where a method calls itself repeatedly until a base condition is met. | Iteration is a process by which a piece of code is repeated number of times or until a condition is met. |
| Is the application for functions.                                                            | Is applicable for loops                                                                                  |
| Works well for smaller code size.                                                            | Works well for larger code size.                                                                         |
| Utilizes more memory as each recursive call is pushed to the stack                           | Comparatively less memory is used.                                                                       |
| Difficult to debug and maintain                                                              | Easier to debug and maintain                                                                             |
| Results in stack overflow if the base condition is not specified or not reached.             | May execute infinitely but will ultimately stop execution w                                              |

### Advantages and Disadvantages of Recursion

When a recursive call is made, new storage locations for variables are allocated on the stack. As, each recursive call returns, the old variables and parameters are removed from the stack. Hence, recursion generally uses more memory and is generally slow.

On the other hand, a recursive solution is much simpler and takes less time to write, debug and maintain.

### Interview Questions

What are the benefits of Recursion?

The benefits of Recursion include:

1. Recursion reduces redundant calling of function.
2. Recursion allows us to solve problems easily when compared to the iterative approach.

Which one is better – Recursion or Iteration?

Recursion makes repeated calls until the base function is reached. Thus there is a memory overhead as a memory for each function call is pushed on to the stack.

Iteration on the other hand does not have much memory overhead. Recursion execution is slower than the iterative approach. Recursion reduces the size of the code while the iterative approach makes the code large.

What are the Advantages of Recursion over Iteration?

1. Recursion makes the code clearer and shorter.
2. Recursion is better than the iterative approach for problems like the Tower of Hanoi, tree traversals, etc.

3. As every function call has memory pushed on to the stack, Recursion uses more memory.
4. Recursion performance is slower than the iterative approach.

Thank You !

## Agenda:

- **Problems on Recursion**
- **Divide and Conquer algorithm**

In the last session, we have learnt about the concept of Recursion, and now in this session we are going to solve and see some problems on recursion.

### Check if a number is Palindrome

Given an integer, write a function that returns true if the given number is palindrome, else false. For example, 12321 is palindrome, but 1451 is not palindrome.



### Solution:

Let the given number be *num*. A simple method for this problem is to first reverse digits of *num*, then compare the reverse of *num* with *num*. If both are same, then return true, else false.

The idea is to create a copy of *num* and recursively pass the copy by reference, and pass *num* by value. In the recursive calls, divide *num* by 10 while moving down the recursion tree. While moving up the recursion tree, divide the copy by 10. When they meet in a function for which all child calls are over, the last digit of *num* will be ith digit from the beginning and the last digit of copy will be ith digit from the end.



```
// A recursive Java program to
// check whether a given number
// is palindrome or not

import java.io.*;

import java.util.*;
```

```
public class CheckPalindromeNumberRecursion {

 // A function that returns true
 // only if num contains one digit

 public static int oneDigit(int num) {

 if ((num >= 0) && (num < 10))

 return 1;

 else

 return 0;

 }

 public static int isPalUtil

 (int num, int dupNum) throws Exception {

 // base condition to return once we

 // move past first digit

 if (num == 0) {

 return dupNum;

 } else {

 dupNum = isPalUtil(num / 10, dupNum);

 }

 // Check for equality of first digit of

 // num and dupNum

 if (num % 10 == dupNum % 10) {

 // if first digit values of num and

 // dupNum are equal divide dupNum

 // value by 10 to keep moving in sync

 }

 }

}
```

```
// with num.
 return dupNum / 10;
} else {
 // At position values are not
 // matching throw exception and exit.
 // no need to proceed further.
 throw new Exception();
}
}
```

```
public static int isPal(int num)
throws Exception {
```

```
if (num < 0)
 num = (-num);

int dupNum = (num);

return isPalUtil(num, dupNum);
```

```
}
```

```
public static void main(String args[]) {

 int n = 1242;
 try {
 isPal(n);
 System.out.println("Yes");
 } catch (Exception e) {
```

```
 System.out.println("No");
}
n = 1231;
try {
 isPal(n);
 System.out.println("Yes");
} catch (Exception e) {
 System.out.println("No");
}
```

```
n = 12;
try {
 isPal(n);
 System.out.println("Yes");
} catch (Exception e) {
 System.out.println("No");
}
```

```
n = 88;
try {
 isPal(n);
 System.out.println("Yes");
} catch (Exception e) {
 System.out.println("No");
}
```

```
n = 8999;
try {
 isPal(n);
```

```
 System.out.println("Yes");

 } catch (Exception e) {

 System.out.println("No");

 }

}

}
```

Output:



Yes

No

Yes

No

#### **Print all possible strings of length k that can be formed from a set of n characters**

Given a set of characters and a positive integer k, print all possible strings of length k that can be formed from the given set.

#### **Examples:**



Input:

```
set[] = {'a', 'b'}, k = 3
```

Output:

aaa

aab

aba

abb

baa

bab

bba

bbb

Input:

```
set[] = {'a', 'b', 'c', 'd'}, k = 1
```

Output:

a

b

c

d

For a given set of size n, there will be  $n^k$  possible strings of length k. The idea is to start from an empty output string (we call it *prefix* in following code). One by one add all characters to *prefix*. For every character added, print all possible strings with current prefix by recursively calling for k equals to k-1.



```
// Java program to print all
```

```
// possible strings of length k
```

```
class AlmaBetter {
```

```
// The method that prints all
```

```
// possible strings of length k.
```

```
// It is mainly a wrapper over
```

```
// recursive function printAllKLengthRec()
```

```
static void printAllKLength(char[] set, int k)
```

```
{
```

```
 int n = set.length;
```

```
 printAllKLengthRec(set, "", n, k);
```

```
}
```

```
// The main recursive method
```

```

// to print all possible
// strings of length k

static void printAllKLengthRec(char[] set,
 String prefix,
 int n, int k)
{

 // Base case: k is 0,
 // print prefix
 if (k == 0)
 {
 System.out.println(prefix);
 return;
 }

 // One by one add all characters
 // from set and recursively
 // call for k equals to k-1
 for (int i = 0; i < n; ++i)
 {

 // Next character of input added
 String newPrefix = prefix + set[i];

 // k is decreased, because
 // we have added a new character
 printAllKLengthRec(set, newPrefix,
 n, k - 1);
 }
}

```

```
}

// Driver Code
public static void main(String[] args)
{
 System.out.println("First Test");
 char[] set1 = {'a', 'b'};
 int k = 3;
 printAllKLength(set1, k);

 System.out.println("\nSecond Test");
 char[] set2 = {'a', 'b', 'c', 'd'};
 k = 1;
 printAllKLength(set2, k);
}
```

Output:



First Test

aaa  
aab  
aba  
abb  
baa  
bab  
bba  
bbb

## Second Test

a  
b  
c  
d

### **Count consonants in a string (Iterative and recursive methods)**

Given a string, count total number of consonants in it. A consonant is a English alphabet character that is not vowel (a, e, i, o and u). Examples of constants are b, c, d, f, g, ..

#### **Examples :**



Input : abc de

Output : 3

There are three consonants b, c and d.

Input : geeksforgeeks portal

Output : 12



```
// Iterative Java program
// to count total number
// of consonants
```

```
import java.io.*;
```

```
class AlmaBetter {
```

```
 // Function to check for consonant
 static boolean isConsonant(char ch)
 {
```

```

// To handle lower case

ch = Character.toUpperCase(ch);

return !(ch == 'A' || ch == 'E' ||
 ch == 'I' || ch == 'O' ||
 ch == 'U') && ch >= 65 && ch <= 90;
}

static int totalConsonants(String str)
{
 int count = 0;

 for (int i = 0; i < str.length(); i++)

 // To check is character is Consonant

 if (isConsonant(str.charAt(i)))
 ++count;

 return count;
}

// Driver code

public static void main(String args[])
{
 String str = "abc de";
 System.out.println(totalConsonants(str));
}

```

### **Recursive solution to count substrings with same first and last characters**

We are given a string S, we need to find count of all contiguous substrings starting and ending with same character.

**Examples :**



Input : S = "abcab"

Output : 7

There are 15 substrings of "abcab"

a, ab, abc, abca, abcab, b, bc, bca

bcab, c, ca, cab, a, ab, b

Out of the above substrings, there

are 7 substrings : a, abca, b, bcab,

c, a and b.

Input : S = "aba"

Output : 4

The substrings are a, b, a and aba



// Java program to count substrings

// with same first and last characters

```
class AlmaBetter
```

```
{
```

```
 // Function to count substrings
```

```
 // with same first and
```

```
 // last characters
```

```
 static int countSubstrs(String str, int i,
```

```
 int j, int n)
```

```
{
```

```
 // base cases
```

```
 if (n == 1)
```

```

 return 1;

 if (n <= 0)
 return 0;

 int res = countSubstrs(str, i + 1, j, n - 1) +
 countSubstrs(str, i, j - 1, n - 1) -
 countSubstrs(str, i + 1, j - 1, n - 2);

 if (str.charAt(i) == str.charAt(j))
 res++;

 return res;
}

// Driver code
public static void main (String[] args)
{
 String str = "abcab";
 int n = str.length();
 System.out.print(countSubstrs(str, 0, n - 1, n));
}

```

Output:



7

### **Divide and Conquer algorithm:**

This technique can be divided into the following three parts:

1. **Divide:** This involves dividing the problem into smaller sub-problems.

2. **Conquer:** Solve sub-problems by calling recursively until solved.
3. **Combine:** Combine the sub-problems to get the final solution of the whole problem.

**Example:** To find the maximum and minimum element in a given array.



Input: { 70, 250, 50, 80, 140, 12, 14 }

Output: The minimum number in a given array is : 12

The maximum number in a given array is : 250

#### **Approach:**

To find the maximum and minimum element from a given array is an application for divide and conquer. In this problem, we will find the maximum and minimum elements in a given array. In this problem, we are using a divide and conquer approach(DAC) which has three steps divide, conquer and combine.

#### **For Maximum:**

In this problem, we are using the recursive approach to find the maximum where we will see that only two elements are left and then we can easily use condition i.e. if( $a[index] > a[index+1]$ ).In a program line  $a[index]$  and  $a[index+1]$ )condition will ensure only two elements in left.



```
if(index >= l-2)
{
 if(a[index]>a[index+1])
 {
 // (a[index]
 // Now, we can say that the last element will be maximum in a given array.
 }
 else
 {
 // (a[index+1]
 // Now, we can say that last element will be maximum in a given array.
 }
}
```

In the above condition, we have checked the left side condition to find out the maximum. Now, we will see the right side condition to find the maximum. Recursive function to check the right side at the current index of an array.



```
max = DAC_Max(a, index+1, l);
```

```
// Recursive call
```

Now, we will compare the condition and check the right side at the current index of a given array. In the given program, we are going to implement this logic to check the condition on the right side at the current index.



```
// Right element will be maximum.
```

```
if(a[index]>max)
```

```
return a[index];
```

```
// max will be maximum element in a given array.
```

```
else
```

```
return max;
```

```
}
```

#### **For Minimum:**

In this problem, we are going to implement the recursive approach to find the minimum no. in a given array.



```
int DAC_Min(int a[], int index, int l)
```

```
//Recursive call function to find the minimum no. in a given array.
```

```
if(index >= l-2)
```

```
// to check the condition that there will be two-element in the left
```

```
then we can easily find the minimum element in a given array.
```

```
{
```

```
// here we will check the condition
```

```
if(a[index]<a[index+1])
```

```
return a[index];
else
return a[index+1];
}
```

Now, we will check the condition on the right side in a given array.



```
// Recursive call for the right side in the given array.
```

```
min = DAC_Min(a, index+1, l);
```

Now, we will check the condition to find the minimum on the right side.



```
// Right element will be minimum
```

```
if(a[index]<min)
```

```
return a[index];
```

```
// Here min will be minimum in a given array.
```

```
else
```

```
return min;
```

#### **Implementation:**



```
// Java code to demonstrate Divide and
```

```
// Conquer Algorithm
```

```
class AlmaBetter{
```

```
// Function to find the maximum no.
```

```
// in a given array.
```

```
static int DAC_Max(int a[], int index, int l)
```

```
{
```

```
int max;
```

```
if(l - 1 == 0)
{
 return a[index];
}
if (index >= l - 2)
{
 if (a[index] > a[index + 1])
 return a[index];
 else
 return a[index + 1];
}
```

```
// Logic to find the Maximum element
// in the given array.

max = DAC_Max(a, index + 1, l);
```

```
if (a[index] > max)
 return a[index];
else
 return max;
}
```

```
// Function to find the minimum no.
// in a given array.

static int DAC_Min(int a[], int index, int l)
```

```
{
 int min;
 if(l - 1 == 0)
 {
```

```

 return a[index];
 }

 if (index >= l - 2)
 {
 if (a[index] < a[index + 1])
 return a[index];
 else
 return a[index + 1];
 }

 // Logic to find the Minimum element
 // in the given array.

 min = DAC_Min(a, index + 1, l);

 if (a[index] < min)
 return a[index];
 else
 return min;
}

// Driver Code
public static void main(String[] args)
{
 // Defining the variables
 int min, max;

 // Initializing the array
 int a[] = { 70, 250, 50, 80, 140, 12, 14 };
}

```

```

// Recursion - DAC_Max function called
max = DAC_Max(a, 0, 7);

// Recursion - DAC_Max function called
min = DAC_Min(a, 0, 7);

System.out.printf("The minimum number in " +
 "a given array is : %d\n", min);
System.out.printf("The maximum number in " +
 "a given array is : %d", max);
}

}

```

Output:



Maximum: 120

Minimum: 11

#### **Conclusion:**

- We have learnt about some famous problems on recursion.
- We have also learnt about the divide and conquer algorithm related to recursion.

#### **Interview Questions:**

- **What is the divide-and-conquer strategy for solving problems with recursion?**

Divide the problem into a number of subproblems that are smaller instances of the same problem. Conquer the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases. Combine the solutions to the subproblems into the solution for the original problem.

- **How to Print Fibonacci Series in Java using Recursion.**



```
import java.util.Scanner;
/** * Java program to calculate and print Fibonacci number using both recursion
* and Iteration.
* Fibonacci number is sum of previous two Fibonacci numbers $fn = fn-1 + fn-2$
* first 10 Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 *
*/
```

```
public class FibonacciCalculator {
 public static void main(String args[]) {
 //input to print Fibonacci series upto how many numbers
 System.out.println("Enter number upto which Fibonacci series to print: ");
 int number = new Scanner(System.in).nextInt();
 System.out.println("Fibonacci series upto " + number + " numbers : ");
 //printing Fibonacci series upto number
 for(int i=1;i<=number; i++){
 System.out.print(fibonacci2(i) + " ");
 }
 }
```

```
/* * Java program for Fibonacci number using recursion.
* This program uses tail recursion to calculate Fibonacci number
* for a given number * @return Fibonacci number
*/
```

```
public static int fibonacci(int number){
 if(number == 1 || number == 2){ return 1;
 } return fibonacci(number-1) + fibonacci(number -2);
 //tail recursion
}
```

}

Thank You!

## **Agenda**

- What is Time Complexity
- Significance of Time Complexity
- Big O Notation
- Space Complexity
- Examples

### **What is Time and Space Complexity**

Time complexity is defined as the amount of time taken by an algorithm to run, as a function of the length of the input. It measures the time taken to execute each statement of code in an algorithm. It is not going to examine the total execution time of an algorithm. Rather, it is going to give information about the variation (increase or decrease) in execution time when the number of operations (increase or decrease) in an algorithm.

Space and Time complexity can define the effectiveness of an algorithm. While we know there is more than one way to solve the problem in programming, knowing how the algorithm works efficiently can add value to the way we do programming. To find the effectiveness of the program/algorithm, knowing how to evaluate them using Space and Time complexity can make the program behave in required optimal conditions.

### **Significance**

Let us first understand what defines an algorithm.

An Algorithm, in computer programming, is a finite sequence of well-defined instructions, typically executed in a computer, to solve a class of problems or to perform a common task. Based on the definition, there needs to be a sequence of defined instructions that have to be given to the computer to execute an algorithm/ perform a specific task. In this context, variation can occur the way how the instructions are defined, for example :

- There can be any number of ways, a specific set of instructions can be defined to perform the same task.
- With options available to choose any one of the available programming languages, the instructions can take any form of syntax along with the performance boundaries of the chosen programming language.
- Variations in terms of Operating System, Processor, Hardware, etc.

Now that we know different factors can influence the outcome of an algorithm being executed, it is wise to understand how efficiently such programs are used to perform a task. To gauge this, we require to evaluate both the Space and Time complexity of an algorithm.

By definition, the Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input. While Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.

To elaborate, Time complexity measures the time taken to execute each statement of code in an algorithm. If a statement is set to execute repeatedly then the number of times that statement gets executed is equal to N multiplied by the time required to run that function each time.

For example, look at the code below:

```
%time
print("This code is to demonstrate Time complexity!") # Only once this statement is executed (N = 1)
```

```
This code is to demonstrate Time complexity!
Wall time: 1 ms
```

```
%time
for i in range(10): # 10 times this statement is executed (N=10)
 print("This code is to demonstrate Time complexity!") # 10 times this statement is executed (N=10)
```

```
This code is to demonstrate Time complexity!
Wall time: 999 µs
```

The first algorithm is defined to print the statement only once. The time taken to execute is shown as **0 nanoseconds**. While the second algorithm is defined to print the same statement but this time it is set to run the same statement in FOR loop 10 times. In the second algorithm, the time taken to execute both the line of code – FOR loop and print statement, is **2 milliseconds**. And, the time taken increases, as the N value increases, since the statement is going to get executed N times.

When an algorithm has a combination of both single executed statements and LOOP statements or with nested LOOP statements, the time increases proportionately, based on the number of times each statement gets executed.

### Big O Notation

As we have discussed, Time complexity is given by time as a function of the length of the input. And, there exists a relation between the input data size (n) and the number of operations performed (N) with respect to time. This relation is denoted as Order of growth in Time complexity and given notation  $O[n]$  where O is the order of growth and n is the length of the input. It is also called as '**Big O Notation**'.

To understand what Big O notation is, we can take a look at a typical example,  $O(n^2)$ , which is usually pronounced "**Big O squared**". The letter "**n**" here represents the **input size**, and the function " **$g(n) = n^2$** " inside the " **$O()$** " gives us an idea of how complex the algorithm is with respect to the input size.

A typical algorithm that has the complexity of  $O(n^2)$  would be the **selection sort** algorithm. Selection sort is a sorting algorithm that iterates through the array to ensure every element at index i is the **i<sup>th</sup>** smallest/largest element of the array.

The algorithm can be described by the following code. In order to make sure the  $i$ th element is the  $i$ th smallest element in the list, this algorithm first iterates through the list with a for loop. Then for every element it uses another for loop to find the smallest element in the remaining part of the list.



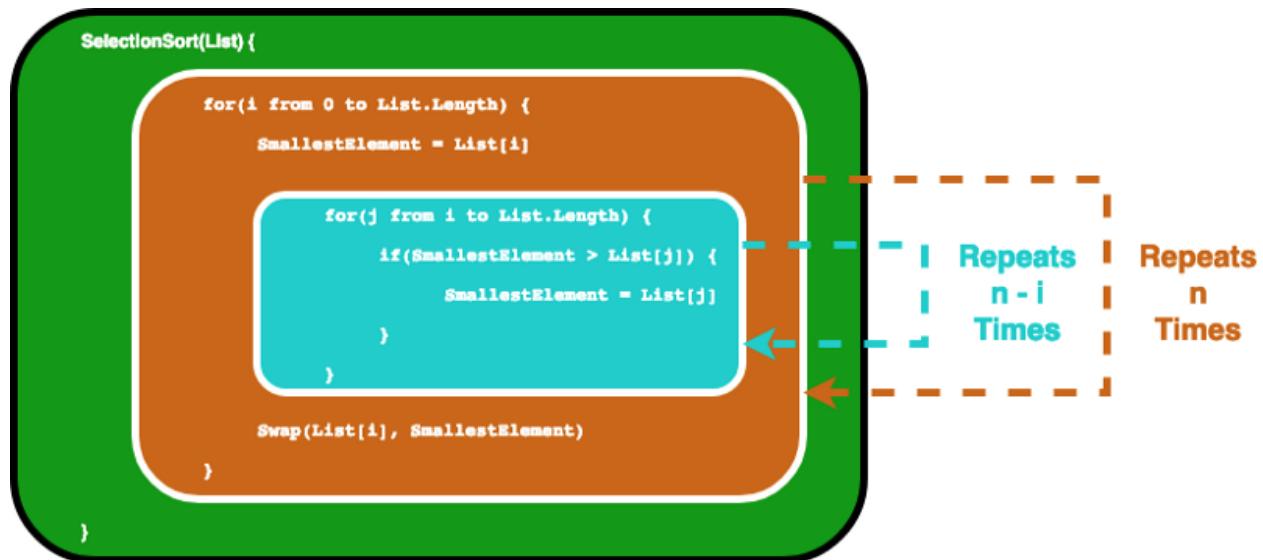
```
void sort(int arr[])
{
 int n = arr.length;

 // One by one move boundary of unsorted subarray
 for (int i = 0; i < n-1; i++)
 {
 // Find the minimum element in unsorted array
 int min_idx = i;
 for (int j = i+1; j < n; j++)
 if (arr[j] < arr[min_idx])
 min_idx = j;

 // Swap the found minimum element with the first
 // element
 int temp = arr[min_idx];
 arr[min_idx] = arr[i];
 arr[i] = temp;
 }
}
```

In this scenario, we consider the variable `arr` as the input, thus input size `n` is the **number of elements inside arr**. Assume the if statement, and the value assignment bounded by the if statement, takes constant time. Then we can find the big O notation for the `SelectionSort` function by analyzing how many times the statements are executed.

First the inner for loop runs the statements inside `n` times. And then after `i` is incremented, the inner for loop runs for `n-1` times... ...until it runs once, then both of the for loops reach their terminating conditions.



This actually ends up giving us a geometric sum, and with some [high-school math](#) we would find that the inner loop will repeat for  $1+2 \dots + n$  times, which equals  $n(n-1)/2$  times. If we multiply this out, we will end up getting  $n^2/2 - n/2$ .

When we calculate big O notation, we only care about the **dominant terms**, and we do not care about the coefficients. Thus we take the  $n^2$  as our final big O. We write it as  $O(n^2)$ , which again is pronounced "*Big O squared*".

Now you may be wondering, what is this "**dominant term**" all about? And why do we not care about the coefficients? Don't worry, we will go over them one by one. It may be a little bit hard to understand at the beginning, but it will all make a lot more sense as we progress through the session.

### Dominant Term

Once upon a time there was an Indian king who wanted to reward a wise man for his excellence. The wise man asked for nothing but some wheat that would fill up a chess board.

But here were his rules: in the first tile he wants 1 grain of wheat, then 2 on the second tile, then 4 on the next one...each tile on the chess board needed to be filled by double the amount of grains as the previous one. The naïve king agreed without hesitation, thinking it would be a trivial demand to fulfill, until he actually went on and tried it...



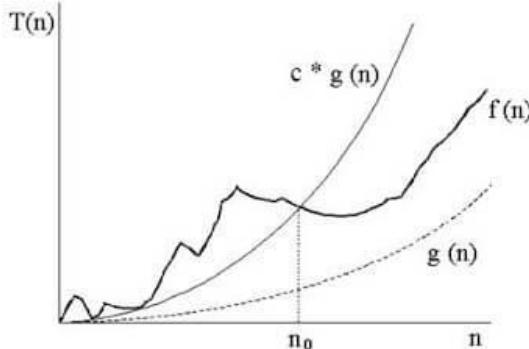
So how many grains of wheat does the king owe the wise man? We know that a chess board has 8 squares by 8 squares, which totals 64 tiles, so the final tile should have  $2^{64}$  grains of wheat. If you do a calculation online, you will end up getting  $1.8446744 \times 10^{19}$ , that is about 18 followed by 18 zeroes. Assuming that each grain of wheat weights 0.01 grams, that gives us 184,467,440,737 tons of wheat. And 184 billion tons is quite a lot, isn't it?

The numbers grow quite fast later for exponential growth don't they? The same logic goes for computer algorithms. If the required efforts to accomplish a task grow exponentially with respect to the input size, it can end up becoming enormously large.

Now the square of 64 is 4096. If you add that number to  $2^{64}$ , it will be lost outside the significant digits. This is why, when we look at the growth rate, we only care about the dominant terms. And since we want to analyze the growth with respect to the input size, the coefficients which only multiply the number rather than growing with the input size do not contain useful information.

# Big-Oh defined

- Big-Oh is about finding an *asymptotic upper bound*.
- Formal definition of Big-Oh:  
 $f(N) = O(g(N))$ , if there exists positive constants  $c, N_0$  such that  
$$f(N) \leq c \cdot g(N) \text{ for all } N \geq N_0.$$
  - We are concerned with how  $f$  grows when  $N$  is large.
    - not concerned with small  $N$  or constant factors
  - Lingo: " $f(N)$  grows no faster than  $g(N)$ ."



21

The formal definition is useful when you need to perform a math proof. For example, the time complexity for selection sort can be defined by the function  $f(n) = n^2/2 - n/2$  as we have discussed in the previous section.

If we allow our function  $g(n)$  to be  $n^2$ , we can find a constant  $c = 1$ , and a  $N_0 = 0$ , and so long as  $N > N_0$ ,  $N^2$  will always be greater than  $N^2/2 - N/2$ . We can easily prove this by subtracting  $N^2/2$  from both functions, then we can easily see  $N^2/2 > -N/2$  to be true when  $N > 0$ . Therefore, we can come up with the conclusion that  $f(n) = O(n^2)$ , in other words *selection sort is "big O squared"*.

You might have noticed a little trick here. That is, if you make  $g(n)$  grow super fast, way faster than anything,  $O(g(n))$  will always be great enough. For example, for any polynomial function, you can always be right by saying that they are  $O(2^n)$  because  $2^n$  will eventually outgrow any polynomials.

## Big O, Little O, Omega & Theta

Big O: " $f(n)$  is  $O(g(n))$ " if for some constants  $c$  and  $N_0$ ,  $f(N) \leq cg(N)$  for all  $N > N_0$

Omega: " $f(n)$  is  $\Omega(g(n))$ " if for some constants  $c$  and  $N_0$ ,  $f(N) \geq cg(N)$  for all  $N > N_0$

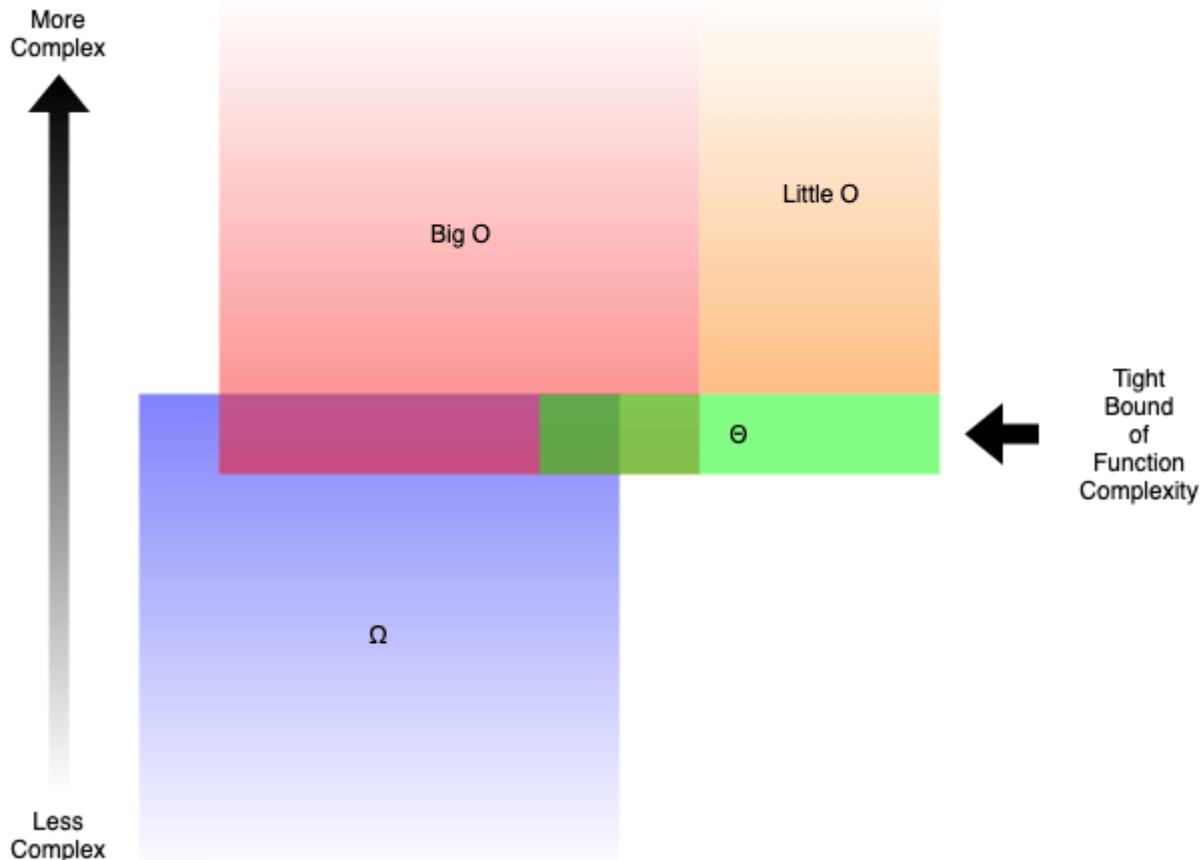
Theta: " $f(n)$  is  $\Theta(g(n))$ " if  $f(n)$  is  $O(g(n))$  and  $f(n)$  is  $\Omega(g(n))$

Little O: " $f(n)$  is  $o(g(n))$ " if  $f(n)$  is  $O(g(n))$  and  $f(n)$  is not  $\Theta(g(n))$

## —Mathematical Definition of Big O, Omega, Theta and Little O

In plain words:

- **Big O ( $O()$ )** describes the **upper bound** of the complexity.
- **Omega ( $\Omega()$ )** describes the **lower bound** of the complexity.
- **Theta ( $\Theta()$ )** describes the **exact bound** of the complexity.
- **Little O ( $o()$ )** describes the **upper bound excluding the exact bound**.



For example, the function  $g(n) = n^2 + 3n$  is  $O(n^3)$ ,  $o(n^4)$ ,  $\Theta(n^2)$  and  $\Omega(n)$ . But you would still be right if you say it is  $\Omega(n^2)$  or  $O(n^2)$ .

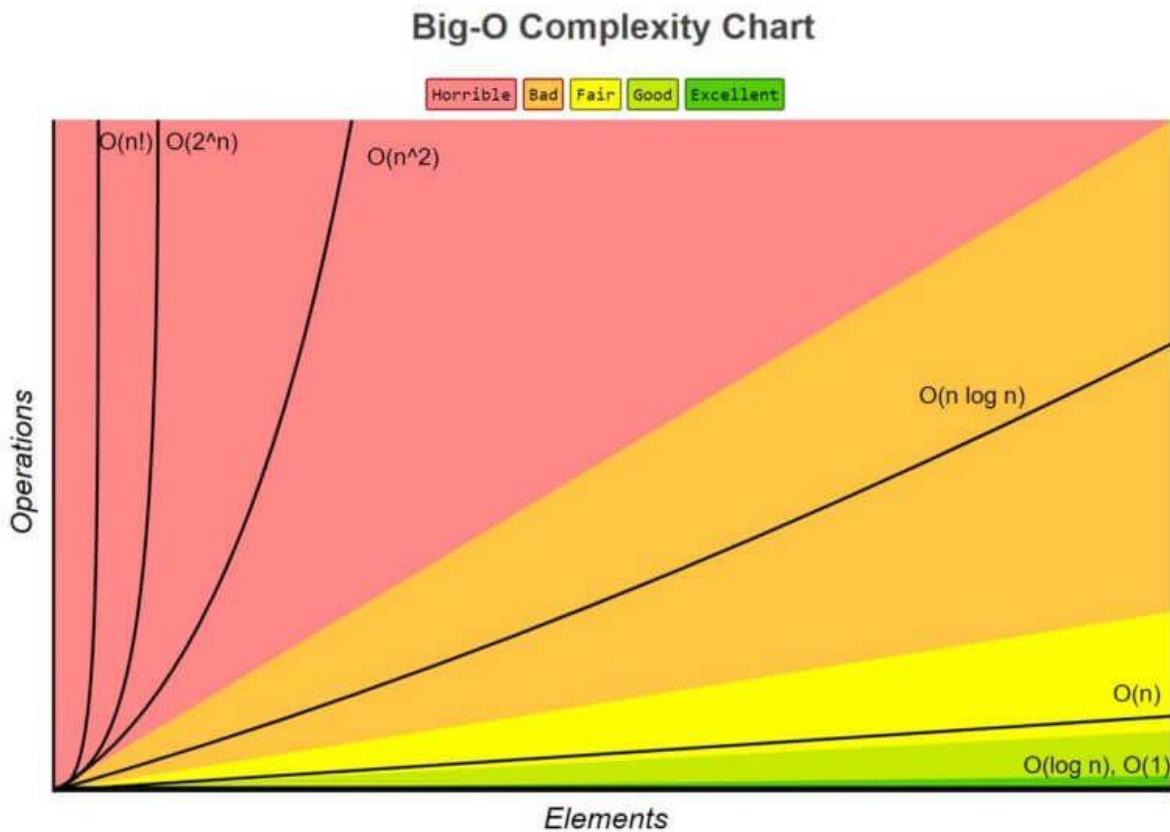
Generally, when we talk about Big O, what we actually meant is Theta. It is kind of meaningless when you give an upper bound that is way larger than the scope of the analysis. This would be similar to solving inequalities by putting  $\infty$  on the larger side, which will almost always make you right.

### Complexity Comparison Between Typical Big Os

When we are trying to figure out the Big O for a particular function  $g(n)$ , we only care about the **dominant term** of the function. The dominant term is the term that grows the fastest.

For example,  $n^2$  grows faster than  $n$ , so if we have something like  $g(n) = n^2 + 5n + 6$ , it will be big  $O(n^2)$ .

But which function grows faster than the others? There are actually quite a few rules.



#### 1. $O(1)$ has the least complexity

Often called "**constant time**", if you can create an algorithm to solve the problem in  $O(1)$ , you are probably at your best. In some scenarios, the complexity may go beyond  $O(1)$ , then we can analyze them by finding its  $O(1/g(n))$  counterpart. For example,  $O(1/n)$  is more complex than  $O(1/n^2)$ .

#### 2. $O(\log(n))$ is more complex than $O(1)$ , but less complex than polynomials

As complexity is often related to divide and conquer algorithms,  $O(\log(n))$  is generally a good complexity you can reach for sorting algorithms.  $O(\log(n))$  is less complex than  $O(\sqrt{n})$ , because the square root function can be considered a polynomial, where the exponent is 0.5.

#### 3. Complexity of polynomials increases as the exponent increases

For example,  $O(n^5)$  is more complex than  $O(n^4)$ . Due to the simplicity of it, we actually went over quite many examples of polynomials in the previous sections.

#### 4. Exponentials have greater complexity than polynomials as long as the coefficients are positive multiples of n

$O(2^n)$  is more complex than  $O(n^{99})$ , but  $O(2^n)$  is actually less complex than  $O(1)$ . We generally take 2 as base for exponentials and logarithms because things tends to be binary in Computer Science, but

exponents can be [changed](#) by changing the coefficients. If not specified, the base for logarithms is assumed to be 2.

## 5. Factorials have greater complexity than exponentials

If you are interested in the reasoning, look up the [Gamma function](#), it is an [analytic continuation](#) of a factorial. A short proof is that both factorials and exponentials have the same number of multiplications, but the numbers that get multiplied grow for factorials, while remaining constant for exponentials.

## 6. Multiplying terms

When multiplying, the complexity will be greater than the original, but no more than the equivalence of multiplying something that is more complex. For example,  $O(n * \log(n))$  is more complex than  $O(n)$  but less complex than  $O(n^2)$ , because  $O(n^2) = O(n * n)$  and  $n$  is more complex than  $\log(n)$ .

### Space Complexity

So far, we have only been discussing the time complexity of the algorithms. That is, we only care about how much time it takes for the program to complete the task. What also matters is the space the program takes to complete the task. The space complexity is related to how much memory the program will use, and therefore is also an important factor to analyze.

The space complexity works similarly to time complexity. For example, selection sort has a space complexity of  $O(1)$ , because it only stores one minimum value and its index for comparison, the maximum space used does not increase with the input size.

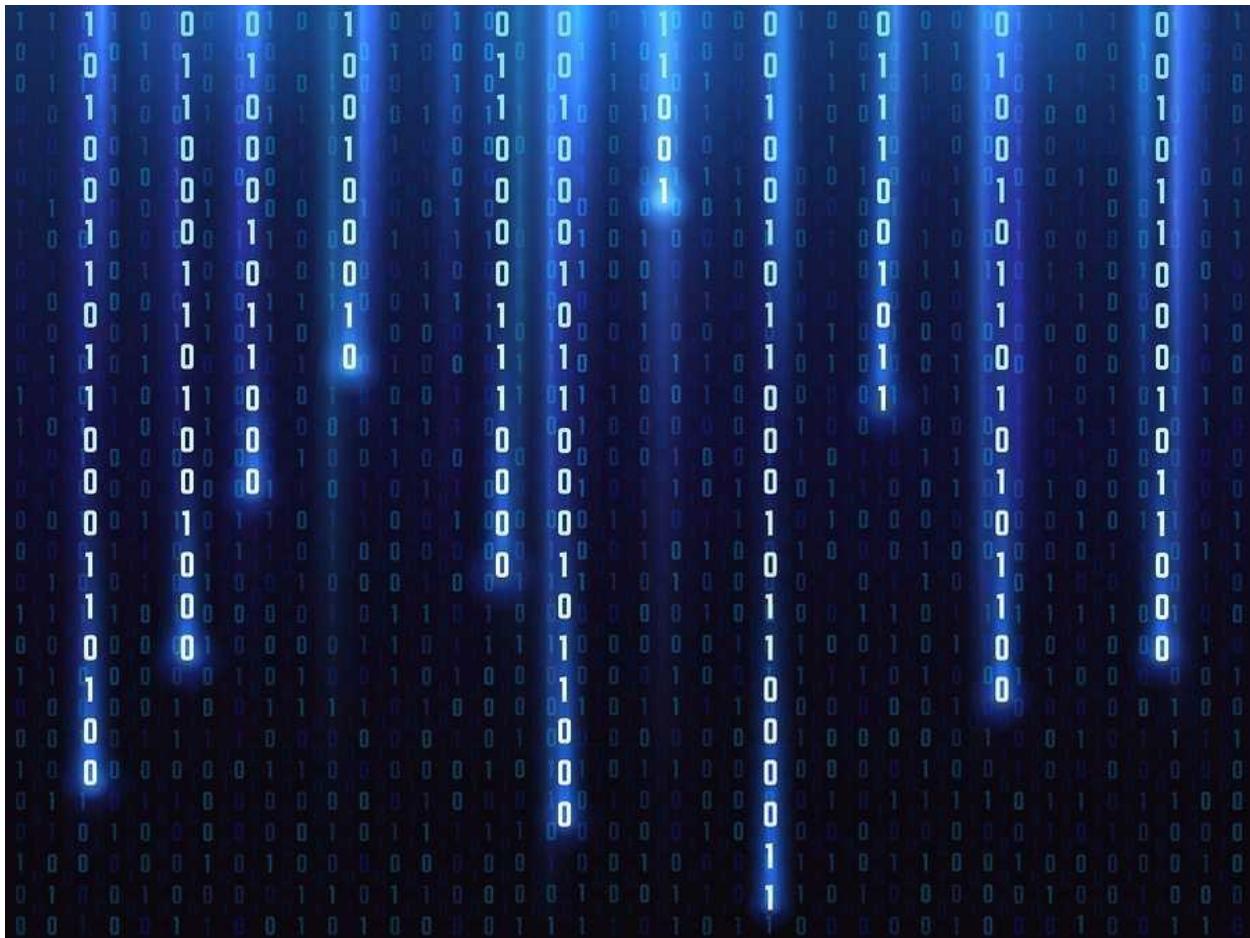
Agenda :

- **What is Bit Manipulation?**
- **Why and when does Bit Manipulation come in handy?**
- **Bitwise Operators in Java**
- **Tricks using Bitwise Operators**
- **Conclusion**

### What is Bit Manipulation?

#### Introduction

The process of applying logical operations to a series of bits (short for binary digits), a computer's smallest unit of data, in order to get a desired outcome is known as bit manipulation. It's essentially all there in it's name - manipulating the bit.



Contrast them with the arithmetic operators which you already know of. For example, what happens when you execute the following block of code?



```
class AddTwoNumbers {
 public static void main(String args[]) {
 int a = 4;
 int b = 5;
 System.out.println("Addition of (" + a + " , " + b + ") is: " + (a + b));
 }
}
```

**Output :**



Addition of (4 , 5) is: 9

Note that the int variable a (holding a value of 4) and the int variable b (holding a value of 5) are stored in the computer memory in these forms, where number in subscript represents the base :



$\$a = 4_{10} = 100_2\$$

$\$b = 5_{10} = 101_2\$$

(Note that the int variables a and b consist of 32 bits each but we need not mention the trailing zeroes in the representation since it has no effect on our result).

When we apply the operator (+) to these variables and print the resultant value (a+b) it internally computes the value of the result which is 9 in this case (represented as 1001 in binary) and displays it on screen. The hardware level implementation is somewhat complicated, and out of the scope for a high level language programmer to worry about. Basically exactly how did those bits in a and b turned into 9 is something we shouldn't worry about at the moment.

However, consider the similar code which computes the value of a&b :



```
class AndBitwise {
 public static void main(String args[]) {
 int a = 4;
```

```

int b = 5;

System.out.println("Bitwise AND of (" + a + " , " + b + ") is: " + (a & b));

}
}

```

**Output :**



Bitwise AND of (4 , 5) is: 4

The output here is 4, which we can easily figure out. We will later on study in detail what the AND operator (`&`) does, but for now, just assume that it returns a value of 1, if both the bits are 1, and returns a value of 0 in all other cases.

Now, let's see how we got the result 4 from the operation 4&5 :



$$a = 4_{\{10\}} = 100_{\{2\}}$$

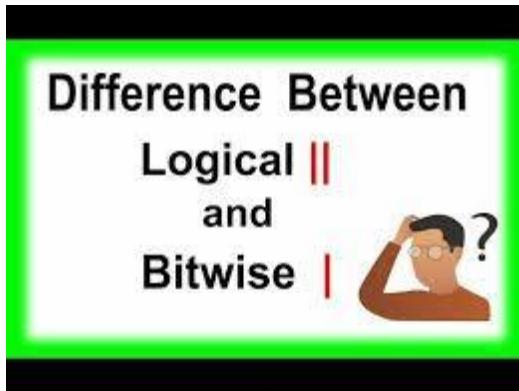
$$b = 5_{\{10\}} = 101_{\{2\}}$$

$$a \& b = 100_{\{2\}} = 4_{\{10\}}$$

The rightmost bits (also called the Least Significant Bit, or the LSB) for a and b are 0 and 1 respectively, Therefore, `a&b` becomes 0 for the LSB. The second bit from right is 0 for both a and b, hence the result is 0 as well. The third bits are 1 for both a and b, therefore we get the value 1 in the third position. This results in a number with the bits 1-0-0 or 100, which when converted back to decimal number system is 4.

Note that in this case, the result is computed by putting in together all the resultant bits, which in turn are computed individually by applying the bitwise operator '`&`' on all the respective bits of the two numbers.

### Bitwise Operators vs Logical Operators



Before we move on to the next topic, it is important that we understand that there is a difference between the Bitwise AND operator (`&`) and a logical AND operator (`&&`). It is possible to confuse one with the other at times, and is also a common mistake programmers usually make in the beginning of their coding journey.

Keep in mind that bitwise operators in Java work on binary digits of integer values (`long`, `int`, `short`, `char`, and `byte`) and return an integer, whereas logical operators work on boolean expressions and return boolean values (either true or false).

Same holds true for the Bitwise OR operator (`|`) and the logical OR operator (`||`).

#### **Why and when does Bit Manipulation come in handy?**

Bit manipulation is extremely effective on all systems due to its constant time complexity and parallel processing. Most programming languages will have you work with abstractions, like objects or variables, rather than the bits they represent. However, direct bit manipulation is very much needed to improve performance and reduce error in certain situations.

In simpler words, the bitwise operators which we are about to study in the next section are faster and more efficient because they work directly within memory rather than through a level of abstraction of software.

Some examples of tasks that require bit manipulation are shown here :

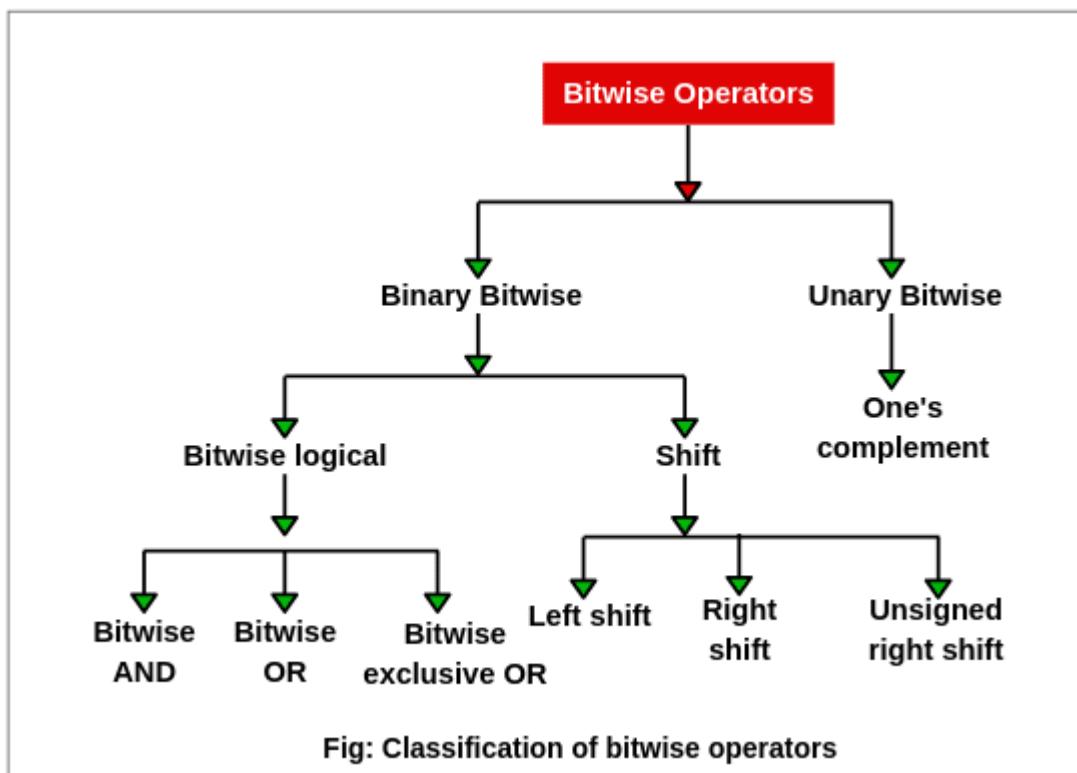
- Low-level device control
- Error detection and correction algorithms
- Data compression
- Encryption algorithms
- Optimisation

Bit manipulation is also a common topic in coding interviews. Many a times, you would not be able to solve a coding test question in an optimal way unless you use the concepts of Bit Manipulation. The interviewers expect you to have a basic understanding of bits, fundamental bit operators, and generally understand the thought process behind bit manipulation. So it becomes all the more important for you to seriously grasp the concepts.

## Bitwise Operators in Java

The various types of the bitwise operator in Java are :

- Bitwise AND
- Bitwise exclusive OR
- Bitwise inclusive OR
- Bitwise Complement
- Bit Shift Operators



Let's study each one of them in detail.

### Bitwise AND (&)

It is a binary operator denoted by the symbol **&**. It returns 1 if and only if both bits are 1, else returns 0.

| x | y | x & y |
|---|---|-------|
| 0 | 0 | 0     |
| 0 | 1 | 0     |
| 1 | 0 | 0     |
| 1 | 1 | 1     |

We have already seen the example of the use of the AND operator. Let's revisit that one :



```
class AndBitwise {
 public static void main(String args[]) {
 int a = 4;
 int b = 5;
 System.out.println("Bitwise AND of (" + a + " , " + b + ") is: " + (a & b));
 }
}
```

**Output :**



Bitwise AND of (4 , 5) is: 4

#### **Bitwise exclusive OR (^)**

It is a binary operator denoted by the symbol  $\wedge$  (pronounced as caret). It returns 0 if both bits are the same, else returns 1.

| x | y | $x \wedge y$ |
|---|---|--------------|
| 0 | 0 | 0            |
| 0 | 1 | 1            |
| 1 | 0 | 1            |
| 1 | 1 | 0            |

Example :



```
class XorBitwise {
 public static void main(String args[]) {
 int a = 6;
 int b = 10;
 System.out.println("Bitwise XOR of (" + a + " , " + b + ") is: " + (a ^ b));
 }
}
```

```
// 6^10 = 0110 ^ 1010 = 1100 (binary) = 12 (decimal)
}
```

**Output :**



Bitwise XOR of (6 , 10) is: 12

### Bitwise inclusive OR (|)

It is a binary operator denoted by the symbol | (pronounced as a pipe). It returns 1 if either of the bit is 1, else returns 0.

| x | y | x   y |
|---|---|-------|
| 0 | 0 | 0     |
| 0 | 1 | 1     |
| 1 | 0 | 1     |
| 1 | 1 | 1     |

Example :



```
class OrBitwise {

 public static void main(String args[]) {

 int a = 2;
 int b = 5;

 System.out.println("Bitwise OR of (" + a + " , " + b + ") is: " + (a | b));
 }

 // 2|5 = 0010 | 0101 = 0111 (binary) = 7 (decimal)
}
```

**Output :**



Bitwise OR of (2 , 5) is: 7

### Bitwise Complement (~)

It is a unary operator denoted by the symbol  $\sim$  (pronounced as the tilde). It returns the inverse or complement of the bit. In simple words, it turns every 1 into a 0 and every 0 into 1.

| X | $\sim X$ |
|---|----------|
| 0 | 1        |
| 1 | 0        |

Example :



```
class NotBitwise {
 public static void main(String args[]) {
 int a = 6;
 System.out.println("Bitwise NOT of " + a + " is: " + ~(a));
 }
}
```

Output :



Bitwise NOT of 6 is: -7

The output here is -7 because in Java, negative numbers are represented in **Two's Complement Notation**. When we flip all the bits present in the binary representation of 6, the resulting bit pattern which we get represents number -7 in Java.

You should refer to the *Additional Resources* section to understand how exactly Two's Complement works if you are not familiar with it already.

### Bit Shift Operators

Shift operators are used in shifting all the bits of the given number either right or left. The general format to shift the bit is as follows :



\*variable << or >> number of places to shift;\*

There are further 3 sub-types of shift operators in Java :

- Signed Right Shift Operator or Bitwise Right Shift Operator
- Unsigned Right Shift Operator

- Left Shift Operator or Bitwise Left Shift Operator

### **Signed Right Shift Operator ( >> )**

The signed right shift operator shifts the bits in a number towards the right with a specified number of positions and fills 0. The operator is denoted by the symbol `>>`.

It also preserves the leftmost bit (sign bit). If 0 is present at the leftmost bit, it means the number is positive. If 1 is present at the leftmost bit, it means the number is negative.

In general, if we write `a>>n`, it means to shift the bits of a number toward the right with a specified position (`n`). In the terms of mathematics, we can represent the signed right shift operator as follows:

$$b = a >> n \longrightarrow b = a / 2^n$$

Example :



```
class SignedRightShiftOperatorExample
{
 public static void main(String args[])
 {
 int x = 64, y = -64;
 System.out.println("x>>2 = " + (x >> 2));
 System.out.println("y>>2 = " + (y >> 2));
 }
}
```

**Output :**



`x>>2 = 16`

`y>>2 = -16`

### **Left Shift Operator ( << )**

The left shift operator (`<<`) shifts a bit pattern to the left. It is represented by the symbol `<<`.

In general, if we write `a<<n`, it means to shift the bits of a number toward the left with specified position (`n`). In the terms of mathematics, we can represent the left shift operator as follows:



`$b = a << n$` implies : `$b = a * 2 ^{n}$`



```
class SignedLeftShiftOperatorExample
```

```
{
```

```
 public static void main(String args[])
 {
 int x = 12, y = -12;
 System.out.println("x<<1 = " + (x << 1));
 System.out.println("y<<1 = " + (y << 1));
 }
}
```

**Output :**



```
x<<1 = 24
```

```
y<<1 = -24
```

*Note:\*\** For arithmetic left shift, since filling the right-most vacant bits with 0s will not affect the sign of the number, the vacant bits will always be filled with 0s, and the sign bit is not considered. Thus, it behaves in a way identical to the logical (unsigned) left shift. So there is no need for a separate unsigned left shift operator.

### **Unsigned Right Shift Operator ( `>>` )**

Unsigned Right Shift Operator moves the bits of the integer a given number of places to the right. The vacant bits are filled with 0s. It is denoted by the symbol `>>`. It does not preserve the sign bit.

**Example:** If `a=11110000` and `b=2`, what would we get from `a>>b`?

```
a >> b = 11110000 >> 2 = 00111100
```

The left operand value is moved right by the number of bits specified by the right operand and the shifted bits are filled up with zeros. Excess bits shifted off to the right are discarded. See the following example :



```
class UnsignedRightShiftOperatorExample
{
 public static void main(String args[])
 {
 int x = 64, y = -64;
 System.out.println("x>>>2 = " + (x >>>2));
 System.out.println("y>>>2 = " + (y >>>2));
 }
}
```

**Output :**



```
x>>>2 = 16
y>>>2 = 1073741808
```

Note that the output which we get when we apply (-64>>>2) is 1073741808 and not -16, since the sign bit was not preserved here.

### Tricks using Bitwise Operators

Now that you have familiarised yourself with the why-and-what part of this topic, let's jump to the how.

Sure we now know what are the Bitwise operators, why and when do we use them, but the thing which will help you the most in an interview setup would be knowing how to make use of the Bit Manipulation concepts to solve the problem-at-hand.

Do keep in mind that there is no exhaustive list covering all the possible techniques. You will keep finding them as you progress in your professional career in this field. Here we present to you two of the most useful tricks :

#### 1. Check for Even and Odd Numbers

Think for a moment - how can we use any Bitwise operator to determine if a given number is odd or even. To do that, observe one key difference which always exists between odd number and even number when we represent them in binary system.

The difference is that an odd number will always have a ‘1’ at the LSB position, while an even number will always have a ‘0’ at LSB. Now if we take that number and apply the operand AND (&) on the given number and the integer 1, we will always end up with 0 if the number is even, and with the number 1, if the given number is odd.

In short, if ‘n’ is an even number, then,



$$n \& 1 = (\dots\dots\dots 0) \& (00001) = 0$$

And if ‘n’ is an odd number, then,



$$n \& 1 = (\dots\dots\dots 1) \& (00001) = 1$$

The following program in Java checks whether a given number is odd or even using this property :



```
class OddEven
{
 static void checkOddEven(int n)
 {
 int b = n&1;

 if(b==1) System.out.println("The given integer is odd.");
 else System.out.println("The given integer is even.");
 }

 public static void main(String[] args)
 {
 int n = 5, m = 100;
 checkOddEven(n);
 checkOddEven(m);
 }
}
```

**Output :**



The given integer is odd.

The given integer is even.

## 2. Change Character to Uppercase or Lowercase

Given any character 'c', we can easily convert to the opposite case using the operation :



```
c = c ^ 32;
```

This works because the binary representation of lowercase and uppercase letters differ by only 1 bit at the 6th bit position (LSB being the first position).

Why is that so? Because if you look at the ASCII values of the characters, for the upper case alphabets, the values start at 65 (A) and for the lower case characters, the values start at 97 (a).

The difference between their integer values is exactly 32.

Using the XOR operation lets us toggle that single bit and swap it to the opposite value, therefore making a lowercase character uppercase or vice versa.



```
class Case
{
 static String shiftCase(char[] a)
 {
 for (int i=0; i<a.length; i++)
 {
 a[i] ^= 32;
 }
 return new String(a);
 }

 public static void main(String[] args)
 }
```

```
{
 String str = "aLMAbETTER";
 System.out.print("Shifted Case: ");
 String str2 = shiftCase(str.toCharArray());
 System.out.println(str2);
}
}
```

**Output :**



Shifted Case: AlmaBetter

**Conclusion**

In this session, we learned about :

- What exactly is Bit Manipulation?
- How Bitwise operators are different from arithmetic and logical operators?
- Why should we use the concepts of Bit Manipulation?
- What are the different Bitwise operators in Java?
- Some common tricks with Bitwise operators.

However, there is one important thing which you should know.

*"If Bit Manipulation is so handy, I should apply its concepts wherever possible."*

Every novice programmer has had this thought. But the one thing which you should always keep in mind is that everything in the programming world (actually, the real world too) always has both - pros and cons.

Bit manipulation no doubt can be very handy in some cases and is really efficient. This increased performance, however, comes at its cost, for example. The readability suffers a lot. And it can be really puzzling for somebody who is not familiar with the bit manipulation concepts.

If the scenario you are using is not highly-performance-critical, you may want to consider, whether the trade off of performance for readability is really worth it and maybe rewrite your solution in a more readable way.

Don't use bit manipulation everywhere possible just because you learned a cool new concept.

**Interview Questions**

- **What are the advantages of using Bitwise operators?**

Bitwise operations are incredibly simple and thus usually faster than arithmetic operations, because they work directly within memory rather than through a level of abstraction of software.

- **What are the different type of Bit Shift Operators in Java?**

There are 3 types of Bit Shift Operators in Java :



|                            |                                                                                                                      |  |
|----------------------------|----------------------------------------------------------------------------------------------------------------------|--|
| Signed Left Shift   <<     | The left shift operator moves all bits by a given number of bits to the left.                                        |  |
| ---   ---   ---            |                                                                                                                      |  |
| Signed Right Shift   >>    | The right shift operator moves all bits by a given number of bits to the right.                                      |  |
| Unsigned Right Shift   >>> | It is the same as the signed right shift, but the vacant leftmost position is filled with 0 instead of the sign bit. |  |

- **How is the Exclusive OR (^) operator different from Inclusive OR (|)?**

**Inclusive OR** allows both bits to be set as well as either of them to be set. So, if either of the bits is set to 1, or if both are set to 1, then the resulting bit is set to 1 as well.

Whereas **Exclusive OR** only allows one bit to be set to 1. So if either of the bits is set to 1, then and only then is the resulting bit set to 1. Otherwise, if both the bits are set to 1, or both are set to 0, then the resulting bit is set to 0.

## Additional Resources

1. How to set k-th bit to 1 in a given number?

[<https://www.geeksforgeeks.org/set-k-th-bit-given-number/>]

2. Negative Numbers in Java and Two's Complement :

[<https://medium.com/@jeanvillete/java-makes-use-of-the-twos-complement-for-representing-signed-numbers-31e421725c04>]

3. Two's Complement Calculator :

[<https://www.allmath.com/twos-complement.php>]

4. Binary - Decimal - Binary Conversion :

[<https://www.cuemath.com/numbers/binary-to-decimal/>]

5. Check whether k-th bit is set to 1 in a given number :

[<https://www.geeksforgeeks.org/check-whether-k-th-bit-set-not/>]

6. XOR of all numbers from 1 to n :

[<https://www.geeksforgeeks.org/calculate-xor-1-n/>]

## Agenda

- What is a Search Algorithm?
- Linear Search - Code and Time Complexity Analysis
- Binary Search - Code and Time Complexity Analysis
- Difference Between Linear Search and Binary Search

Search algorithms are a fundamental computer science concept that you should understand as a developer. They work by using a step-by-step method to locate specific data among a collection of data.

In this module, we'll learn how search algorithms work by looking at their implementations in Java

### ##What is a Search Algorithm?

According to Wikipedia, a search algorithm is:

Any algorithm which solves the search problem, namely, to retrieve information stored within some data structure, or calculated in the search space of a problem domain, either with discrete or continuous values.

Search algorithms are designed to check or retrieve an element from any data structure where that element is being stored. They search for a target (key) in the search space.

### ##Types of Search Algorithms

In this module, we are going to discuss two important types of search algorithms:

1. Linear or Sequential Search
2. Binary Search

Let's discuss these two in detail with examples, code implementations, and time complexity analysis.

### ##What is a linear search?

Linear search is a simple searching algorithm. Here, the searching occurs from one item after the other. That is; this algorithm checks every item and checks for a matching item of that. If the item is not present, searching continues until the end of the data. Therefore, linear search is an algorithm that allows going through each element in an array to locate the given item.

**Let's consider a simple example.**

**Suppose we have an array of 10 elements as shown in the below figure:**

The above figure shows an array of character type having 10 values. If we want to search 'E', then the searching begins from the 0th element and scans each element until the element, i.e., 'E' is not found. We cannot directly jump from the 0th element to the 4th element, i.e., each element is scanned one by one till the element is not found.

**Approach for Linear or Sequential Search**

- Start with index 0 and compare each element with the target
- If the target is found to be equal to the element, return its index
- If the target is not found, return -1

### Code Implementation

Let's now look at how we'd implement this type of search algorithm in JAVA



```
public class LinearSearch {

 public static void main(String[] args) {
 int[] nums = {2, 12, 15, 11, 7, 19, 45};
 int target = 7;
 System.out.println(search(nums, target));

 }

 static int search(int[] nums, int target) {
 for (int index = 0; index < nums.length; index++) {
 if (nums[index] == target) {
 return index;
 }
 }
 return -1;
 }
}
```

### Time Complexity Analysis

**The Best Case** occurs when the target element is the first element of the array. The number of comparisons, in this case, is 1. So, the time complexity is O(1).

**The Average Case:** On average, the target element will be somewhere in the middle of the array. The number of comparisons, in this case, will be  $N/2$ . So, the time complexity will be O(N) (the constant being ignored).

**The Worst Case** occurs when the target element is the last element in the array or not in the array. In this case, we have to traverse the entire array, and so the number of comparisons will be N. So, the time complexity will be O(N).

### What is Binary Search?

A binary search is a search in which the middle element is calculated to check whether it is smaller or larger than the element which is to be searched. The main advantage of using binary search is that it does not scan each element in the list. Instead of scanning each element, it performs the searching to the half of the list. So, the binary search takes less time to search an element as compared to a linear search.

The one **pre-requisite of binary search** is that an array should be in sorted order, whereas the linear search works on both sorted and unsorted array. The binary search algorithm is based on the divide and conquer technique, which means that it will divide the array recursively.

**There are three cases used in the binary search:**

**Case 1: data< a[mid] then left = mid+1.**

**Case 2: data> a[mid] then right=mid-1**

**Case 3: data = a[mid] // element is found**

In the above case, 'a' is the name of the array, **mid** is the index of the element calculated recursively, **data** is the element that is to be searched, **left** denotes the left element of the array and **right** denotes the element that occur on the right side of the array.

### Approach for Binary Search

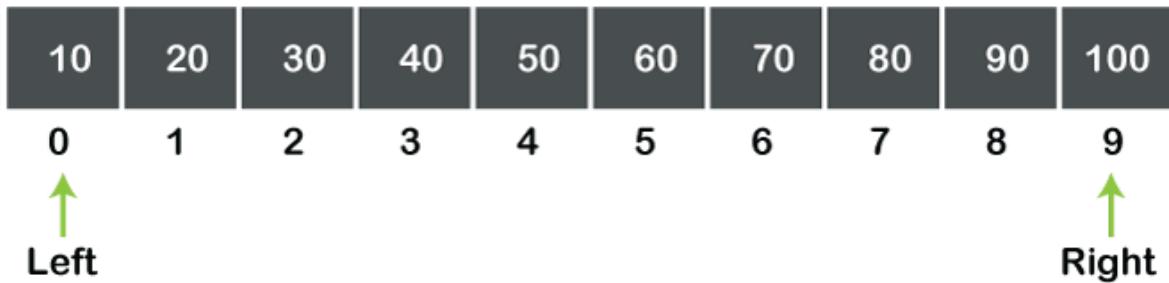
- Compare the target element with the middle element of the array.
- If the target element is greater than the middle element, then the search continues in the right half.
- Else if the target element is less than the middle value, the search continues in the left half.
- This process is repeated until the middle element is equal to the target element, or the target element is not in the array
- If the target element is found, its index is returned, else -1 is returned.

### Let's understand the working of binary search through an example.

Suppose we have an array of 10 size which is indexed from 0 to 9 as shown in the below figure:

We want to search for 70 element from the above array.

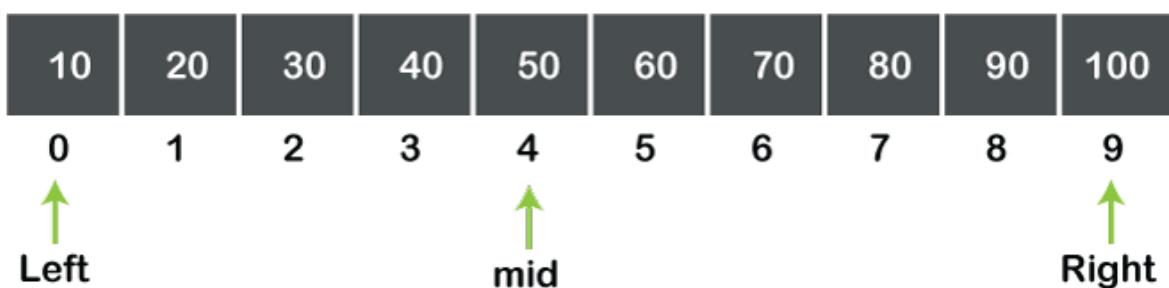
**Step 1:** First, we calculate the middle element of an array. We consider two variables, i.e., left and right. Initially, left =0 and right=9 as shown in the below figure:



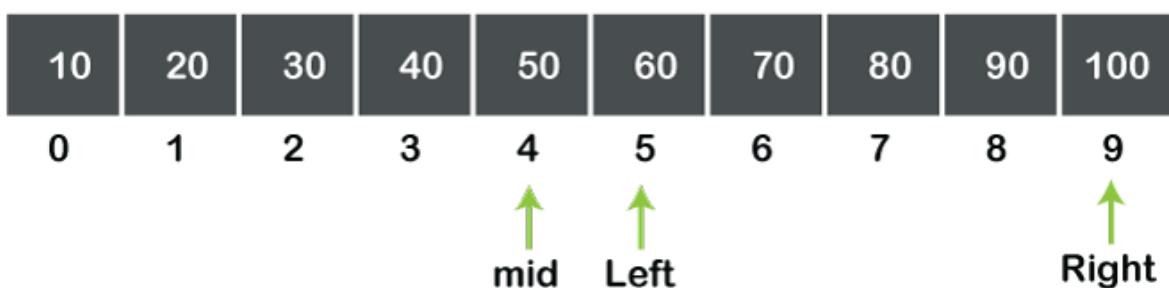
The middle element value can be calculated as:

$$mid = \frac{left + right}{2}$$

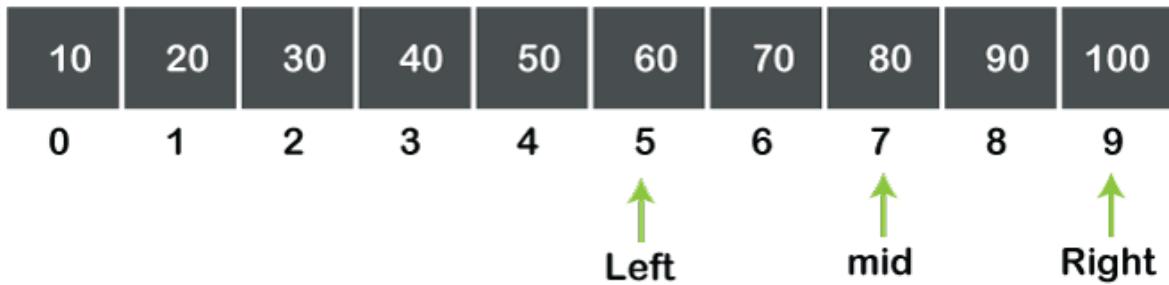
Therefore, mid = 4 and a[mid] = 50. The element to be searched is 70, so a[mid] is not equal to data. The case 2 is satisfied, i.e., data>a[mid].



**Step 2:** As data>a[mid], so the value of left is incremented by mid+1, i.e., left=mid+1. The value of mid is 4, so the value of left becomes 5. Now, we have got a subarray as shown in the below figure:



Now again, the mid-value is calculated by using the above formula, and the value of mid becomes 7. Now, the mid can be represented as:



In the above figure, we can observe that  $a[mid] > data$ , so again, the value of mid will be calculated in the next step.

**Step 3:** As  $a[mid] > data$ , the value of right is decremented by mid-1. The value of mid is 7, so the value of right becomes 6. The array can be represented as:



The value of mid will be calculated again. The values of left and right are 5 and 6, respectively. Therefore, the value of mid is 5. Now the mid can be represented in an array as shown below:



In the above figure, we can observe that  $a[mid] < data$ .

**Step 4:** As  $a[mid] < data$ , the left value is incremented by mid+1. The value of mid is 5, so the value of left becomes 6.

Now the value of mid is calculated again by using the formula which we have already discussed. The values of left and right are 6 and 6 respectively, so the value of mid becomes 6 as shown in the below figure:



We can observe in the above figure that  $a[mid] = \text{data}$ . Therefore, the search is completed, and the element is found successfully.

### Code Implementation



```

public class BinarySearch {

 public static void main(String[] args) {
 int[] nums = {2, 12, 15, 17, 27, 29, 45};
 int target = 17;
 System.out.println(search(nums, target));
 }

 static int search(int[] nums, int target) {
 int start = 0;
 int end = nums.length - 1;

 while (start <= end) {
 int mid = start + (end - start) / 2;

 if (nums[mid] > target)
 end = mid - 1;
 else if (nums[mid] < target)
 start = mid + 1;
 else
 }
 }
}

```

```

 return mid;
 }
 return -1;
}

```

### Time Complexity Analysis

**The Best Case** occurs when the target element is the middle element of the array. The number of comparisons, in this case, is 1. So, the time complexity is  $O(1)$ .

**The Average Case:** On average, the target element will be somewhere in the array. So, the time complexity will be  $O(\log N)$ .

**The Worst Case** occurs when the target element is not in the list or it is away from the middle element. So, the time complexity will be  $O(\log N)$ .

### What is the Difference Between Linear Search and Binary Search?

| Basis of comparison | Linear search                                                                                                                              | Binary search                                                                                                |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| Definition          | The linear search starts searching from the first element and compares each element with a searched element till the element is not found. | It finds the position of the searched middle element of the array.                                           |
| Sorted data         | In a linear search, the elements don't need to be arranged in sorted order.                                                                | The pre-condition for the binary search must be arranged in a sorted order.                                  |
| Implementation      | The linear search can be implemented on any linear data structure such as an array, linked list, etc.                                      | The implementation of binary search is implemented only on those data structures that support random access. |
| Approach            | It is based on the sequential approach.                                                                                                    | It is based on the divide and conquer approach.                                                              |
| Size                | It is preferable for the small-sized data sets.                                                                                            | It is preferable for the large-size data sets.                                                               |
| Efficiency          | It is less efficient in the case of large-size data sets.                                                                                  | It is more efficient in the case of large-size data sets.                                                    |
| Worst-case scenario | In a linear search, the worst-case scenario for finding the element is $O(n)$ .                                                            | In a binary search, the worst-case scenario for finding the element is $O(\log n)$ .                         |
| Best-case scenario  | In a linear search, the best-case scenario for finding the first element in the list is $O(1)$ .                                           | In a binary search, the best-case scenario for finding the first element in the list is $O(1)$ .             |
| Dimensional array   | It can be implemented on both a single and multidimensional array.                                                                         | It can be implemented only on a multidimensional array.                                                      |

### Interview Questions

- **Why use sequential search?**
  1. If your collection size is relatively small and you would be performing this search relatively a few times, then this might be an option.
  2. Another case is when you need to have constant insertion performance (like using a linked list) and the search frequency is less.
  3. In addition, linear search places very few restrictions on the complex data types. All that is needed is a match function of sorts.
- **Write pseudo code to search the array in the reverse order, returning 0 when the element is not found.**



Set i to n.

Repeat this loop

If  $i \leq 0$ , then exit the loop.

If  $A[i] = x$ , then exit the loop.

Set i to  $i - 1$ .

Return i

- **What are the various applications of linear search**

Linear search is usually very simple to implement, and is practical when the list has only a few elements, or when performing a single search in an unordered list. When many values have to be searched in the same list, it often pays to pre-process the latter in order to use a faster method.

- **What is binary search?**

Binary search is a type of search algorithm that programmers can use to locate target values in a sorted dataset. Some other names for binary search include binary chop, half-interval search and logarithmic search. Binary search requires a sorted or ordered array to operate.

Binary search works by dividing a dataset in half and comparing the middle value to the target value. If the target value is greater than the middle value, the algorithm eliminates the data to the left of the middle value. It continues this process of division and elimination until the middle value is equal to the target value or until it fails to locate the target value.

- **Explain the differences between binary and linear search algorithms.**

Binary and linear search algorithms are different in four primary ways. Binary search requires sorted data arranged by value, while linear search doesn't. Binary search needs ordering comparisons of 'greater than' or 'less than' to operate, and linear only requires equality comparisons. Finally, the two algorithms have different data access requirements and complexity. Binary requires random data access versus sequential access for linear search, and binary search has a complexity of  $O(\log n)$  while linear has a complexity of  $O(n)$ .

Basis of comparison      Linear search      Binary search

Definition      The linear search starts searching from the first element and compares each element with a searched element till the element is not found. It finds the position of the searched element by finding the middle element of the array.

Sorted data      In a linear search, the elements don't need to be arranged in sorted order.      The pre-condition for the binary search is that the elements must be arranged in a sorted order.

Implementation      The linear search can be implemented on any linear data structure such as an array, linked list, etc.      The implementation of binary search is limited as it can be implemented only on those data structures that have two-way traversal.

Approach      It is based on the sequential approach.      It is based on the divide and conquer approach.

Size      It is preferable for the small-sized data sets.      It is preferable for the large-size data sets.

Efficiency      It is less efficient in the case of large-size data sets.      It is more efficient in the case of large-size data sets.

Worst-case scenario      In a linear search, the worst-case scenario for finding the element is  $O(n)$ .  
In a binary search, the worst-case scenario for finding the element is  $O(\log 2n)$ .

Best-case scenario      In a linear search, the best-case scenario for finding the first element in the list is  $O(1)$ . In a binary search, the best-case scenario for finding the first element in the list is  $O(1)$ .

Dimensional array      It can be implemented on both a single and multidimensional array.      It can be implemented only on a multidimensional array.

## 1. Search a particular character in a string



String : "Almabetter"

Character : "l"

Output: true

CODE:



```
public class SearchInStrings {
 public static void main(String[] args) {
 String name = "Kunal";
 char target = 'u';
 // System.out.println(search(name, target));

 System.out.println(Arrays.toString(name.toCharArray()));
 }

 static boolean search(String str, char target) {
 if (str.length() == 0) {
 return false;
 }

 for (int i = 0; i < str.length(); i++) {
 if (target == str.charAt(i)) {
 return true;
 }
 }
 return false;
 }
}
```

}

**2. Find min number in the array**



Input: [18,12,-7,3,14,28]

Output: -7

CODE:



```
public class FindMin {
 public static void main(String[] args) {
 int[] arr = {18, 12, 7, 3, 14, 28};
 System.out.println(min(arr));
 }

 // assume arr.length != 0
 // return the minimum value in the array
 static int min(int[] arr) {
 int ans = arr[0];
 for (int i = 1; i < arr.length; i++) {
 if (arr[i] < ans) {
 ans = arr[i];
 }
 }
 return ans;
 }
}
```

**3. Search for 3 in the range index[1,4]**



Input: [18,12,-7,3,14,28]

Output: 3 //(Index of 3 in the array)

CODE:



```
public class SearchInRange {
 public static void main(String[] args) {
 int[] arr = {18, 12, -7, 3, 14, 28};
 int target = 3;
 System.out.println(linearSearch(arr, target, 1, 4));
 }

 static int linearSearch(int[] arr, int target, int start, int end) {
 if (arr.length == 0) {
 return -1;
 }

 // run a for loop
 for (int index = start; index <= end; index++) {
 // check for element at every index if it is = target
 int element = arr[index];
 if (element == target) {
 return index;
 }
 }
 // this line will execute if none of the return statements above have executed
 // hence the target not found
 return -1;
 }
}
```

```
}
```

```
}
```

4. Given a sorted array of n integers and a target value, determine if the target exists in the array in logarithmic time using the binary search algorithm. If target exists in the array, print the index of it.



Input:

```
nums[] = [2, 3, 5, 7, 9]
```

```
target = 7
```

Output: Element found at index 3

Input:

```
nums[] = [1, 4, 5, 8, 9]
```

```
target = 2
```

Output: Element not found

CODE:



```
class Main
```

```
{
```

```
// Function to determine if a `target` exists in the sorted array `nums`
// or not using a binary search algorithm
public static int binarySearch(int[] nums, int target)
{
```

```
// search space is nums[left...right]
int left = 0, right = nums.length - 1;

// loop till the search space is exhausted
while (left <= right)
{
 // find the mid-value in the search space and
 // compares it with the target

 int mid = (left + right) / 2;

 // overflow can happen. Use:
 // int mid = left + (right - left) / 2;
 // int mid = right - (right - left) / 2;

 // target is found
 if (target == nums[mid]) {
 return mid;
 }

 // discard all elements in the right search space,
 // including the middle element
 else if (target < nums[mid]) {
 right = mid - 1;
 }

 // discard all elements in the left search space,
 // including the middle element
 else {

```

```

 left = mid + 1;
 }
}

// `target` doesn't exist in the array
return -1;
}

public static void main(String[] args)
{
 int[] nums = { 2, 5, 6, 8, 9, 10 };
 int target = 5;

 int index = binarySearch(nums, target);

 if (index != -1) {
 System.out.println("Element found at index " + index);
 }
 else {
 System.out.println("Element not found in the array");
 }
}

```

5. Given a sorted array containing duplicates, efficiently find each element's frequency without traversing the whole array.



Input: [2, 2, 2, 4, 4, 4, 5, 5, 6, 8, 8, 9]

Output: {2: 3, 4: 3, 5: 2, 6: 1, 8: 2, 9: 1}

CODE:



```
import java.util.HashMap;
import java.util.Map;

class Main
{
 // Function to find the frequency of each element in a sorted array
 public static Map<Integer, Integer> findFrequency(int[] nums)
 {
 // Map to store the frequency of each array element
 Map<Integer, Integer> freq = new HashMap<>();

 // search space is nums[left...right]
 int left = 0, right = nums.length - 1;

 // loop till the search space is exhausted
 while (left <= right)
 {
 // if nums[left...right] consists of only one element, update its count
 if (nums[left] == nums[right])
 {
 freq.put(nums[left], freq.getOrDefault(nums[left], 0) +
 (right - left + 1));
 }

 // now discard nums[left...right] and continue searching in
 // nums[right+1... n-1] for nums[left]
 left = right + 1;
 right = nums.length - 1;
 }
 }
}
```

```

 }

 else {
 // reduce the search space
 right = (left + right) / 2;
 }

}

return freq;
}

public static void main(String[] args)
{
 int[] nums = { 2, 2, 2, 4, 4, 4, 5, 5, 6, 8, 8, 9};

 Map<Integer, Integer> freq = findFrequency(nums);
 System.out.println(freq);
}

```

## 6. Search an element in a 2D-Array



Input: {

```

{23, 4, 1},
{18, 12, 3, 9},
{78, 99, 34, 56},
{18, 12}
}
```

Output: // Row and column of element if exists

CODE:



```
import java.util.Arrays;

public class SearchIn2DArray {

 public static void main(String[] args) {
 int[][] arr = {
 {23, 4, 1},
 {18, 12, 3, 9},
 {78, 99, 34, 56},
 {18, 12}
 };
 int target = 56;
 int[] ans = search(arr,target); // format of return value {row, col}
 System.out.println(Arrays.toString(ans));

 System.out.println(max(arr));

 System.out.println(Integer.MIN_VALUE);
 }

 static int[] search(int[][] arr, int target) {
 for (int row = 0; row < arr.length; row++) {
 for (int col = 0; col < arr[row].length; col++) {
 if (arr[row][col] == target) {
 return new int[]{row, col};
 }
 }
 }
 }
}
```

```
 return new int[]{-1, -1};
}
}
```

7. Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.



Input: nums = [5,7,7,8,8,10], target = 8

Output: [3,4]

CODE:



```
public class FirstAndLastPosition {
 public static void main(String[] args) {

 }

 public int[] searchRange(int[] nums, int target) {

 int[] ans = {-1, -1};
 // check for first occurrence if target first
 ans[0] = search(nums, target, true);
 if (ans[0] != -1) {
 ans[1] = search(nums, target, false);
 }
 return ans;
 }

 // this function just returns the index value of target
 int search(int[] nums, int target, boolean findstartIndex) {
```

```

int ans = -1;

int start = 0;

int end = nums.length - 1;

while(start <= end) {

 // find the middle element

// int mid = (start + end) / 2; // might be possible that (start + end) exceeds the range of int in java

 int mid = start + (end - start) / 2;

if (target < nums[mid]) {

 end = mid - 1;

} else if (target > nums[mid]) {

 start = mid + 1;

} else {

 // potential ans found

 ans = mid;

 if (findStartIndex) {

 end = mid - 1;

 } else {

 start = mid + 1;

 }

}

return ans;

}

}

```

8. You are given an array of characters letters that is sorted in non-decreasing order, and a character target. There are at least two different characters in letters. Return *the smallest character in letters that is lexicographically greater than target*. If such a character does not exist, return the first character in letters.



Input: letters = ["c", "f", "j"], target = "a"

Output: "c"

Explanation: // The smallest character that is lexicographically greater than 'a' in letters is 'c'.

CODE:



```
public class SmallestLetter {
```

```
 public static void main(String[] args) {
```

```
 }
```

```
 public char nextGreatestLetter(char[] letters, char target) {
```

```
 int start = 0;
```

```
 int end = letters.length - 1;
```

```
 while(start <= end) {
```

```
 // find the middle element
```

```
// int mid = (start + end) / 2; // might be possible that (start + end) exceeds the range of int in java
```

```
 int mid = start + (end - start) / 2;
```

```
 if (target < letters[mid]) {
```

```
 end = mid - 1;
```

```
 } else {
```

```
 start = mid + 1;
```

```
 }
```

```
}
```

```
 return letters[start % letters.length];
}
}
```

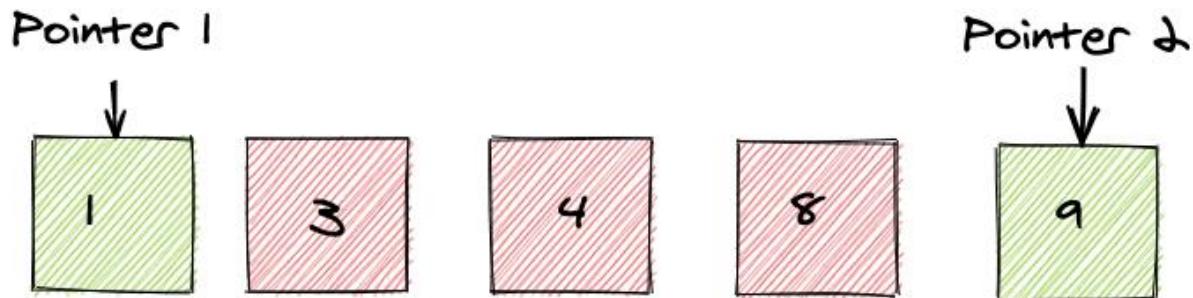
## Agenda:

- **Introduction**
- **Implementation**
- **How space is saved using this two pointer technique**
- **Examples**

## Introduction

Two pointer approach is an essential part of a programmer's toolkit, especially in technical interviews. The name does justice in this case, it involves using two pointers to save time and space. (Here, pointers are basically array indexes).

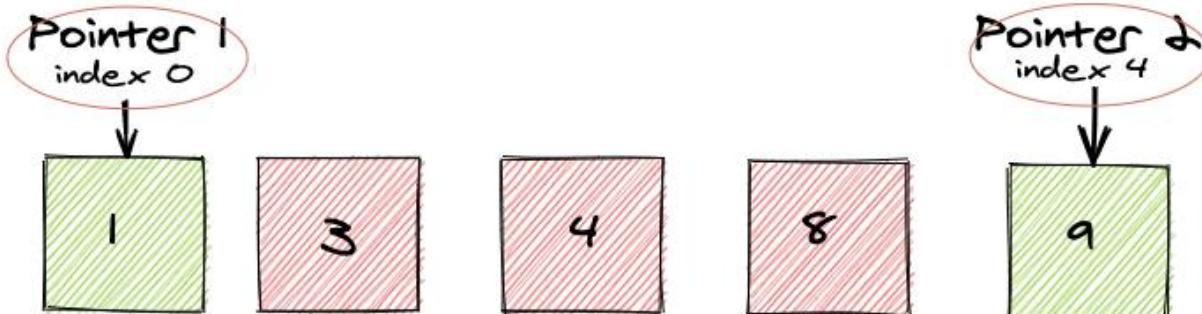
***The idea here is to iterate two different parts of the array simultaneously to get the answer faster.***



## What is Pattern ?

The name two pointers does justice in this case, as it is exactly as it sounds. It's the use of two different pointers (usually to keep track of array or string indices) to solve a problem involving said indices with the benefit of saving time and space. See the below for the two pointers highlighted in yellow.

But what are pointers? In computer science, a pointer is a reference to an object. In many programming languages, that object stores a memory address of another value located in computer memory, or in some cases, that of memory-mapped computer hardware.



## When do use it ?

In many problems **involving collections such as arrays or lists**, we have to analyze each element of the collection compared to its other elements.

There are many approaches to solving problems like these. For example we usually start from the first index and iterate through the data structure one or more times depending on how we implement our code.

Sometimes we may even have to create an additional data structure depending on the problem's requirements. This approach might give us the correct result, but it likely won't give us the most space and time efficient result.

This is why the two-pointer technique is efficient. We are able to process two elements per loop instead of just one. Common patterns in the two-pointer approach entail:

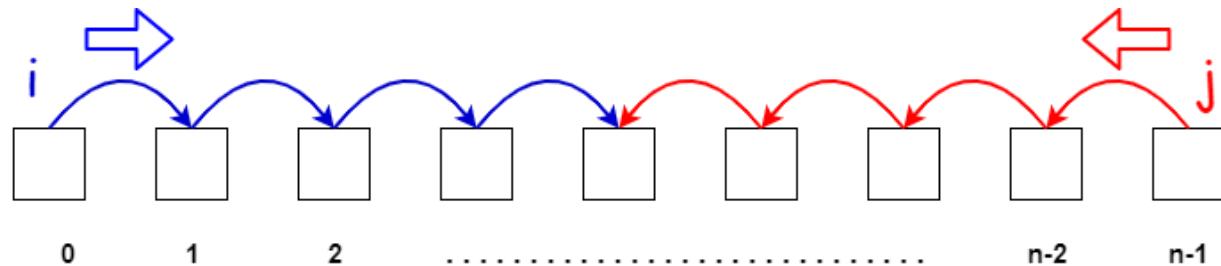
1. Two pointers, each starting from the beginning and the end until they both meet.
2. One pointer moving at a slow pace, while the other pointer moves at twice the speed.

## Implementation

There are primarily two ways of implementing the two-pointer technique:

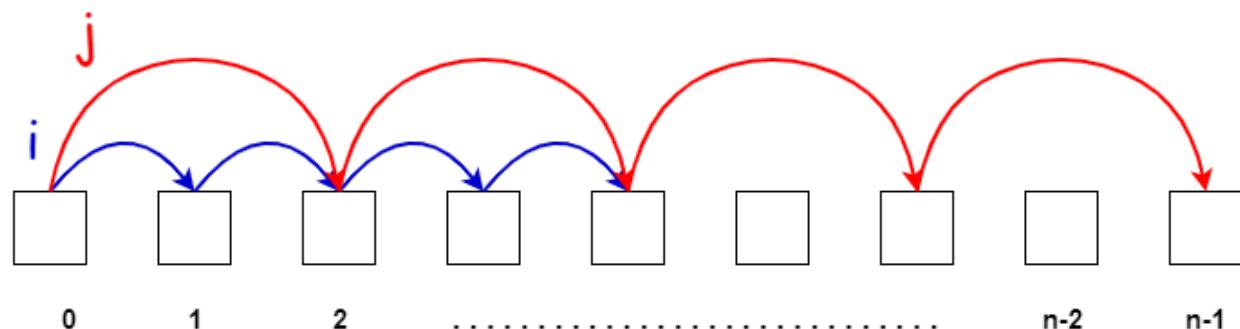
### 1. One pointer at each end

One pointer starts from beginning and other from the end and they proceed towards each other.



### 2. Different Paces

Both pointers start from the beginning but one pointer moves at a faster pace than the other one.



## How space is saved using this two pointer technique ?

There are several situations when a naive implementation of a problem requires additional space thereby increasing the space complexity of the solution. Two-pointer technique often helps to decrease the required space or remove the need for it altogether.

Let's take an example:

### Reverse an array

- **Naive Solution:** Using a temporary array and filling elements in it from the end



```
int[] reverseArray(int arr[], int n)
{
 int reverse[n]
 for (i = 0 to n-1)
 reverse[n-i-1] = arr[i]

 return reverse
}
```

*Space Complexity:  $O(n)$*

- **Efficient Solution:** Moving pointers towards each other from both ends and swapping elements at their positions



```
int[] reverseArray(int arr[], int n)
{
 i = 0
 j = n-1
 while (i < j)
 {
 swap(arr[i], arr[j])
 i = i + 1
 j = j - 1
 }
}
```

```
 return arr
}
```

*Space Complexity: O(1)*

**Examples:**

**Sum Exists in an Array**

**Problem:** Given a sorted array of integers, we need to see if there are two numbers in it such that their sum is equal to a specific value.

For example, if our input array is [1, 1, 2, 3, 4, 6, 8, 9] and the target value is 11, then our method should return *true*. However, if the target value is 20, it should return *false*.

Let's first see a naive solution:



```
public boolean twoSumSlow(int[] input, int targetValue) {

 for (int i = 0; i < input.length; i++) {
 for (int j = 1; j < input.length; j++) {
 if (input[i] + input[j] == targetValue) {
 return true;
 }
 }
 }
 return false;
}
```

In the above solution, we looped over the input array twice to get all possible combinations. We checked the combination sum against the target value and returned *true* if it matches. **The time complexity of this solution is  $O(n^2)$ .**

Now let's see how can we apply the two-pointer technique here:



```
public boolean twoSum(int[] input, int targetValue) {
```

```

int pointerOne = 0;

int pointerTwo = input.length - 1;

while (pointerOne < pointerTwo) {

 int sum = input[pointerOne] + input[pointerTwo];

 if (sum == targetValue) {

 return true;

 } else if (sum < targetValue) {

 pointerOne++;

 } else {

 pointerTwo--;

 }

}

return false;
}

```

Since the array is already sorted, we can use two pointers. One pointer starts from the beginning of the array, and the other pointer begins from the end of the array, and then we add the values at these pointers. If the sum of the values is less than the target value, we increment the left pointer, and if the sum is higher than the target value, we decrement the right pointer.

We keep moving these pointers until we get the sum that matches the target value or we have reached the middle of the array, and no combinations have been found. **The time complexity of this solution is O(n) and space complexity is O(1)**, a significant improvement over our first implementation.

### Rotate Array k Steps

**Problem:** Given an array, rotate the array to the right by  $k$  steps, where  $k$  is non-negative. For example, if our input array is  $[1, 2, 3, 4, 5, 6, 7]$  and  $k$  is 4, then the output should be  $[4, 5, 6, 7, 1, 2, 3]$ .

We can solve this by having two loops again which will make the time complexity  $O(n^2)$  or by using an extra, temporary array, but that will make the space complexity  $O(n)$ .

Let's solve this using the two-pointer technique instead:



```

public void rotate(int[] input, int step) {
 step %= input.length;
 reverse(input, 0, input.length - 1);
 reverse(input, 0, step - 1);
 reverse(input, step, input.length - 1);
}

private void reverse(int[] input, int start, int end) {
 while (start < end) {
 int temp = input[start];
 input[start] = input[end];
 input[end] = temp;
 start++;
 end--;
 }
}

```

In the above methods, we reverse the sections of the input array in-place, multiple times, to get the required result. For reversing the sections, we used the two-pointer approach where swapping of elements was done at both ends of the array section.

Specifically, we first reverse all the elements of the array. Then, we reverse the first  $k$  elements followed by reversing the rest of the elements. **The time complexity of this solution is  $O(n)$  and space complexity is  $O(1)$ .**

#### **Find a triplet such that sum of two equals to third element**

Given an array of integers, you have to find three numbers such that the sum of two elements equals the third element.

Examples:



Input : {5, 32, 1, 7, 10, 50, 19, 21, 2}

Output : 21, 2, 19

Input : {5, 32, 1, 7, 10, 50, 19, 21, 0}

Output : no such triplet exist

#### Simple approach:

Run three loops and check if there exists a triplet such that sum of two elements equals the third element.

Time complexity:  $O(n^3)$

#### Efficient approach:

- Sort the given array first.
- Start fixing the greatest element of three from the back and traverse the array to find the other two numbers which sum up to the third element.
- Take two pointers j(from front) and k(initially i-1) to find the smallest of the two number and from i-1 to find the largest of the two remaining numbers
- If the addition of both the numbers is still less than A[i], then we need to increase the value of the summation of two numbers, thereby increasing the j pointer, so as to increase the value of  $A[j] + A[k]$ .
- If the addition of both the numbers is more than A[i], then we need to decrease the value of the summation of two numbers, thereby decrease the k pointer so as to decrease the overall value of  $A[j] + A[k]$ .



```
// Java program to find three numbers
// such that sum of two makes the
// third element in array

import java.util.Arrays;

public class AlmaBetter {

 // utility function for finding
 // triplet in array

 static void findTriplet(int arr[], int n)
 {
 // sort the array
```

```
Arrays.sort(arr);

// for every element in arr
// check if a pair exist(in array) whose
// sum is equal to arr element
for (int i = n - 1; i >= 0; i--) {
 int j = 0;
 int k = i - 1;
 while (j < k) {
 if (arr[i] == arr[j] + arr[k]) {

 // pair found
 System.out.println("numbers are " + arr[i] + " "
 + arr[j] + " " + arr[k]);
 }

 return;
 }

 else if (arr[i] > arr[j] + arr[k])
 j += 1;
 else
 k -= 1;
}

// no such triplet is found in array
System.out.println("No such triplet exists");
}

// driver program
```

```
public static void main(String args[])
{
 int arr[] = { 5, 32, 1, 7, 10, 50, 19, 21, 2 };
 int n = arr.length;
 findTriplet(arr, n);
}
```

#### **Output:**



numbers are 21 2 19

**Time Complexity: O(N^2)**

**Auxiliary Space: O(1)**

#### **Conclusion**

We have learnt about:

- two pointer technique
- its implementation and how this saves our space
- seen some popular examples

#### **Interview Questions**

- **How many pointers are required to solve a problem using two pointer technique?**
  1. Two pointers, each starting from the beginning and the end until they both meet.
  2. One pointer moving at a slow pace, while the other pointer moves at twice the speed.
- **Can two pointer approach be used to solve any problem?**

**Two-pointer technique is commonly used to solve array problems very efficiently.** Whenever an array question deals with finding two numbers in an array that satisfy a certain condition, either directly or indirectly, two-pointer should be the first strategy that comes to mind.

## Agenda

- What is Sorting?
- Insertion Sort
- Selection Sort
- Bubble Sort

### What is Sorting?

Ever wonder how the products in an Amazon or any other e-commerce website get sorted when you apply filters like low-to-high or high-to-low, or alphabetically? Sorting algorithms play a vital role for such websites where you have an enormous amount of products listed and you have to make customer interactions easy.

A sorting algorithm is used to rearrange a given array or list of elements as per the comparison operator on the element. The comparison operator is used to decide the new order of elements in the respective data structure.



Mainly there are five basic algorithms used and you can derive multiple algorithms using these basic algorithms. Each of these algorithms has some pros and cons and can be chosen effectively depending on the size of data to be handled.

1. Insertion Sort
2. Selection Sort

3. Bubble Sort
4. Merge Sort
5. Quick Sort

We'll discuss Insertion, Selection and Bubble Sort in depth in this session while Merge Sort and Quick Sort will be discussed later.

### **Insertion Sort**

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

A similar approach is used by insertion sort.

### **Working**

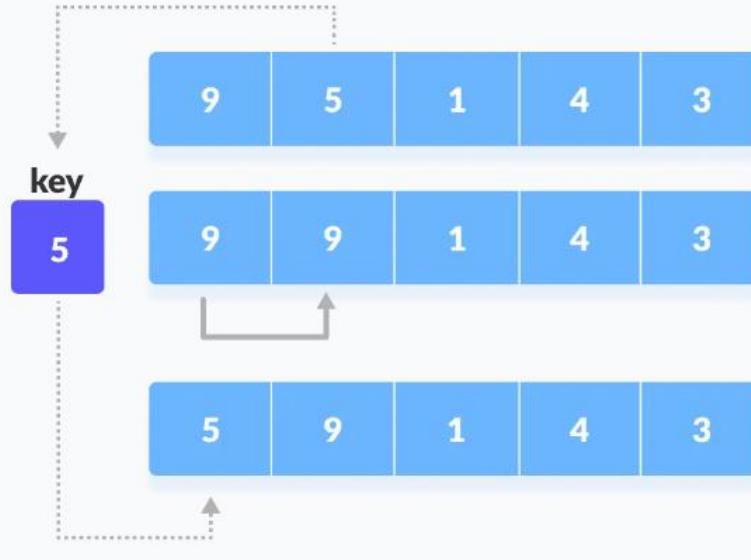
Suppose we need to sort the following array.

|   |   |   |   |   |
|---|---|---|---|---|
| 9 | 5 | 1 | 4 | 3 |
|---|---|---|---|---|

1. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.

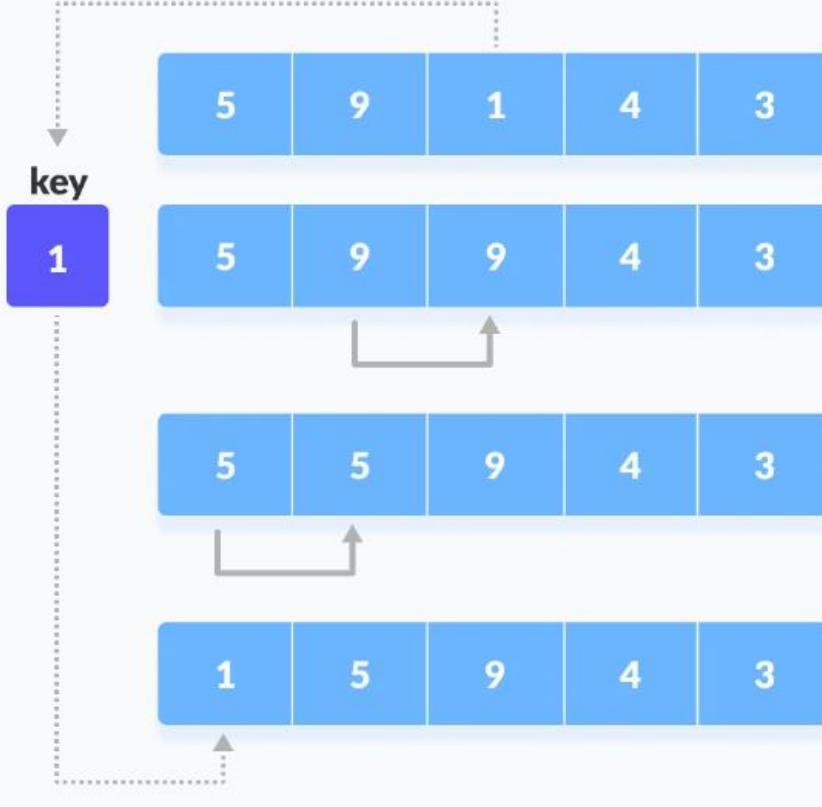
**step = 1**



2. Now, the first two elements are sorted.

Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.

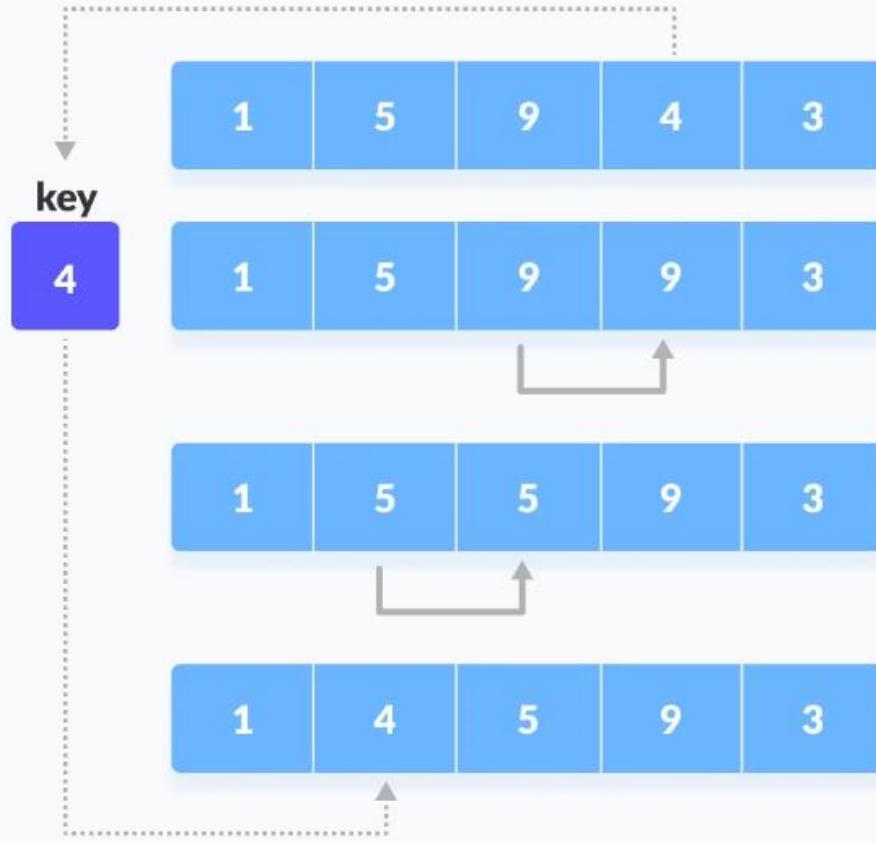
**step = 2**



Place 1 at the beginning

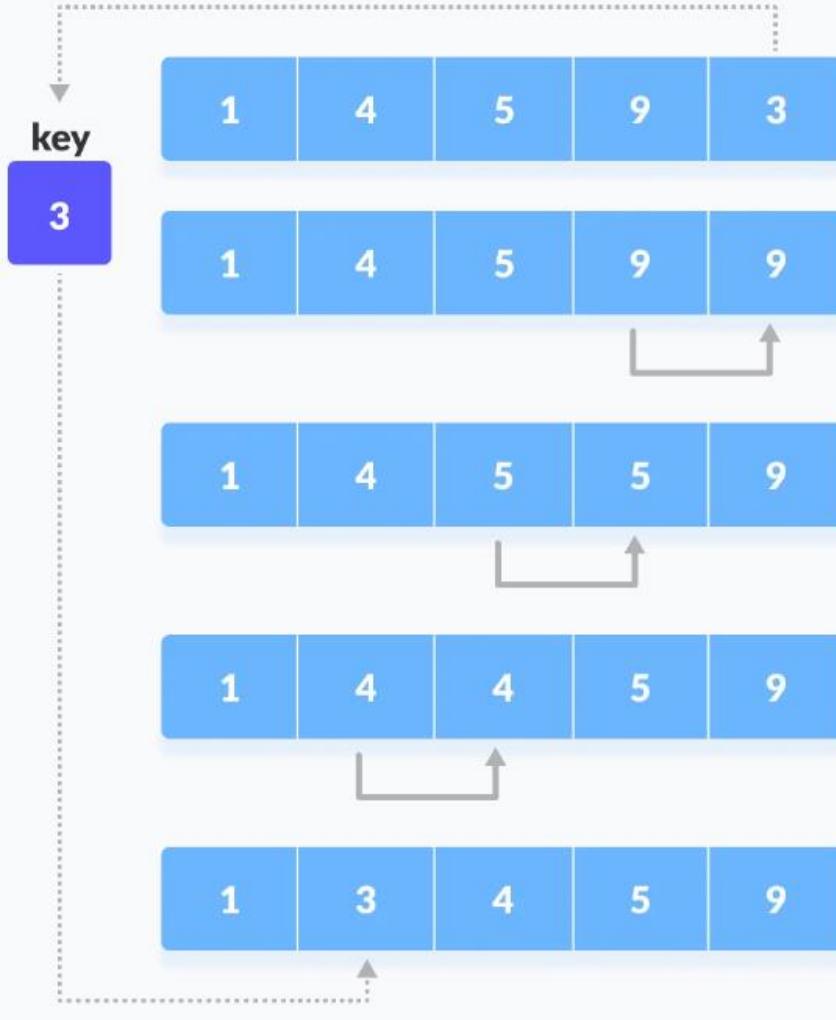
3. Similarly, place every unsorted element at its correct position.

**step = 3**



Place 1 at the beginning

**step = 4**



Place 3 behind 1 and the array is sorted

### Algorithm



insertionSort(array)

mark first element as sorted

for each unsorted element X

'extract' the element X

for j <- lastSortedIndex down to 0

```
if current element j > X
 move sorted element to the right by 1
 break loop and insert X here
end insertionSort
```

### Code

```
import java.util.Arrays;

class InsertionSort {

 void insertionSort(int array[]) { int size = array.length;

 for (int step = 1; step < size; step++) {
 int key = array[step];
 int j = step - 1;

 // Compare key with each element on the left of it until an element smaller than
 // it is found.
 // For descending order, change key<array[j] to key>array[j].
 while (j >= 0 && key < array[j]) {
 array[j + 1] = array[j];
 --j;
 }

 // Place key at after the element just smaller than it.
 array[j + 1] = key;
 }
 }

 // Driver code public static void main(String args[]) { int[] data = { 9, 5, 1, 4, 3 }; InsertionSort is = new
 InsertionSort(); is.insertionSort(data); System.out.println("Sorted Array in Ascending Order: ");
 System.out.println(Arrays.toString(data)); } }
```

### Complexity

| Time Complexity  |            |
|------------------|------------|
| Best             | O(n)       |
| Worst            | O( $n^2$ ) |
| Average          | O( $n^2$ ) |
| Space Complexity | O(1)       |

### Time Complexities

- **Worst Case Complexity:** O( $n^2$ ) Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs. Each element has to be compared with each of the other elements so, for every nth element, ( $n-1$ ) number of comparisons are made. Thus, the total number of comparisons =  $n*(n-1) \sim n^2$
- **Best Case Complexity:** O(n) When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear.
- **Average Case Complexity:** O( $n^2$ ) It occurs when the elements of an array are in jumbled order (neither ascending nor descending).

### Space Complexity

Space complexity is O(1) because an extra variable key is used.

### Applications

The insertion sort is used when:

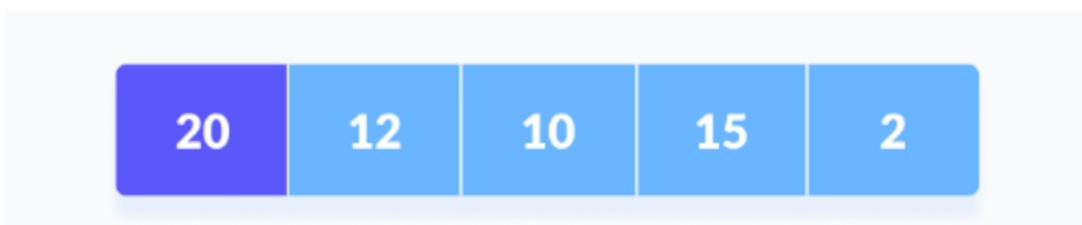
- the array is has a small number of elements
- there are only a few elements left to be sorted

### Selection Sort

Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

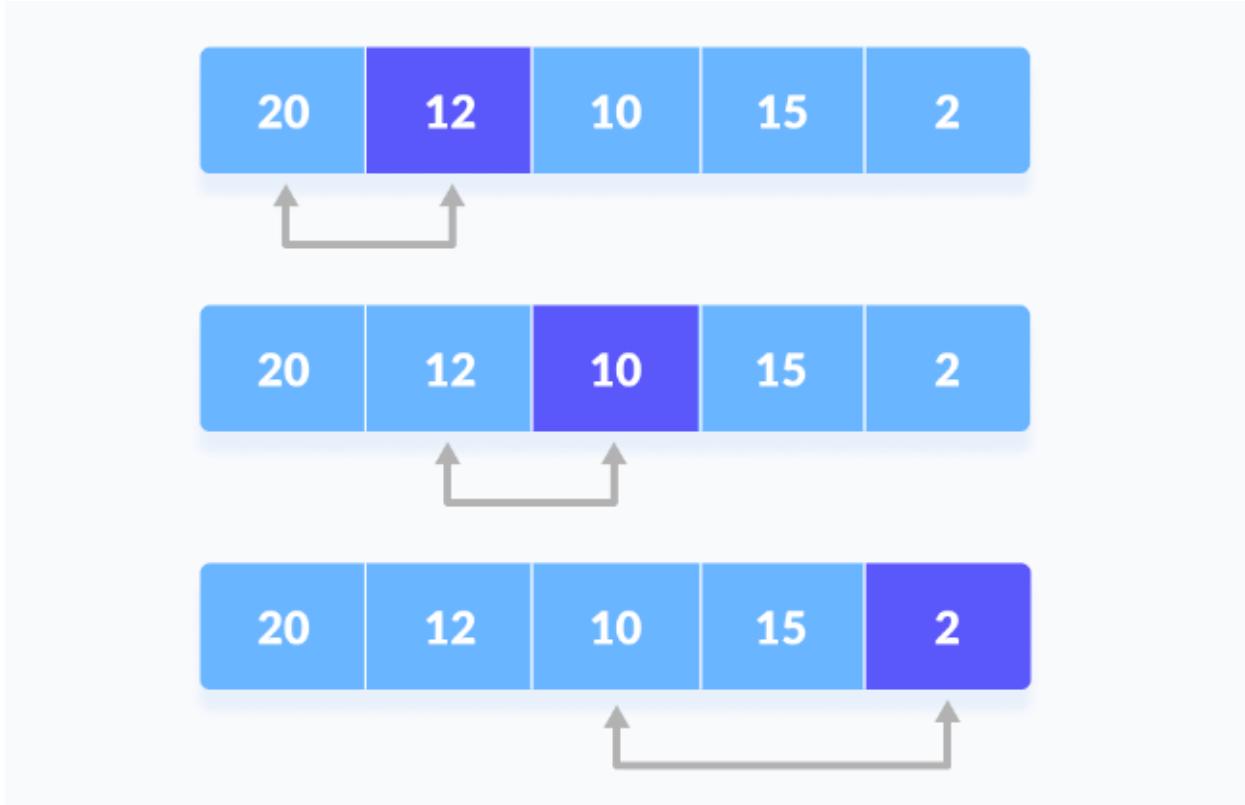
### Working

1. Set the first element as minimum.



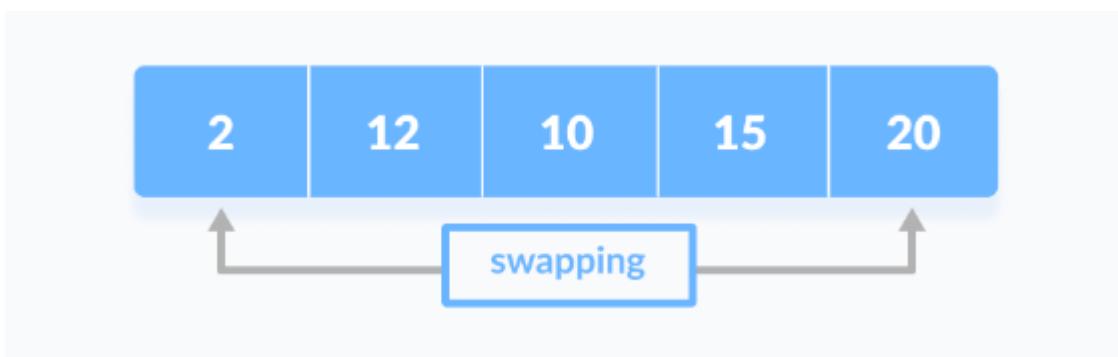
Select first element as minimum

2. Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum. Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.



Compare minimum with the remaining elements

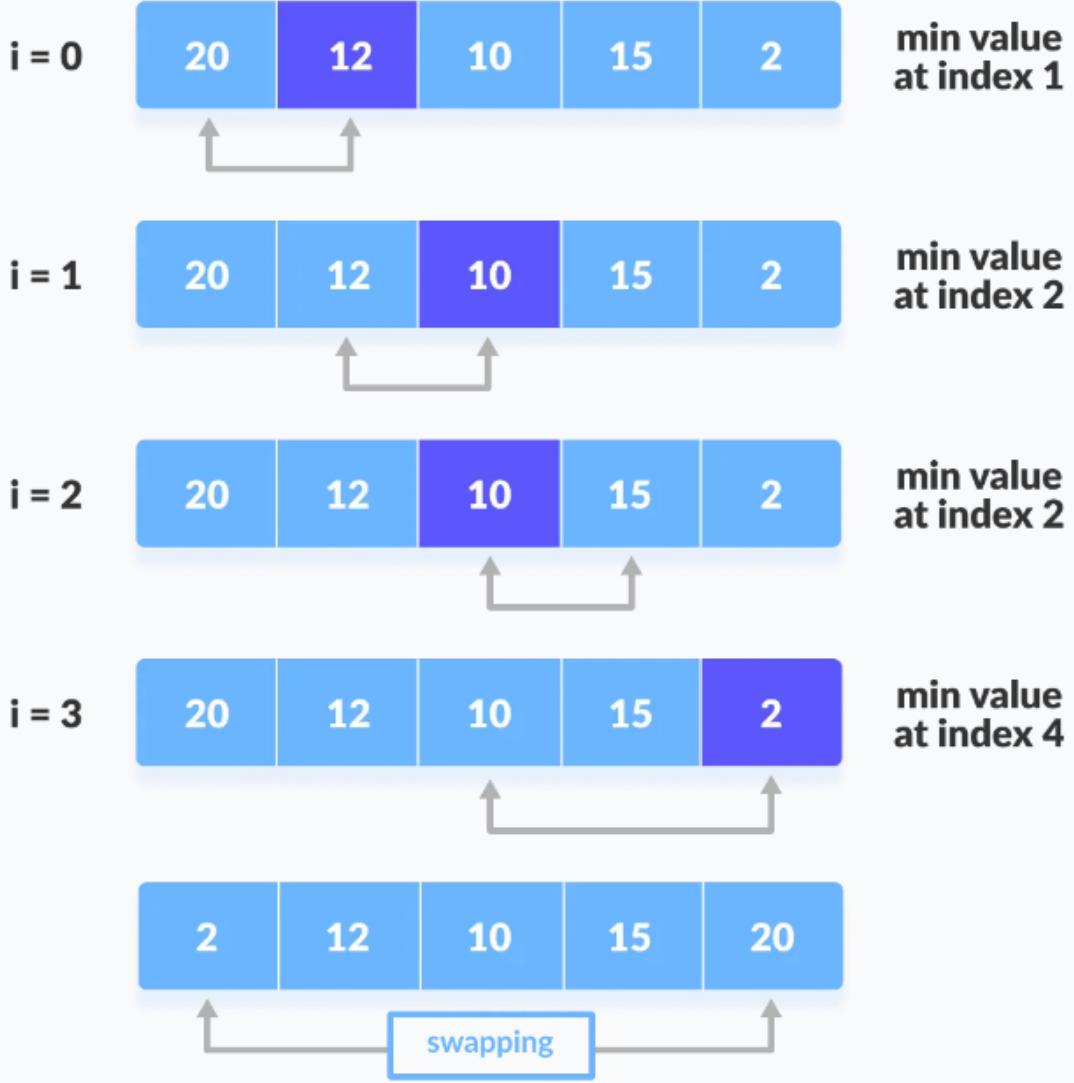
3. After each iteration, minimum is placed in the front of the unsorted list.



Swap the first with minimum

4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.

**step = 0**



The first iteration

**step = 1**

i = 0



**min value  
at index 2**

i = 1



**min value  
at index 2**

i = 2



**min value  
at index 2**



The second iteration

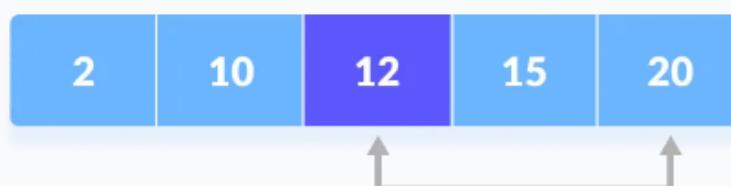
**step = 2**

**i = 0**



**min value  
at index 2**

**i = 2**



**min value  
at index 2**

**2**

**10**

**12**

**15**

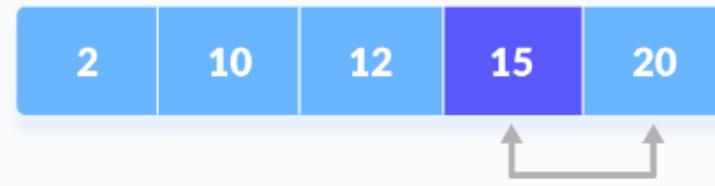
**20**

**already in place**

The third iteration

**step = 3**

**i = 0**



**min value  
at index 3**

**2**

**10**

**12**

**15**

**20**

**already in place**

The fourth iteration

### Algorithm



```
selectionSort(array, size)
repeat (size - 1) times
 set the first unsorted element as the minimum
 for each of the unsorted elements
 if element < currentMinimum
 set element as new minimum
 swap minimum with first unsorted position
end selectionSort
```

### Code



```
import java.util.Arrays;

class SelectionSort {
 void selectionSort(int array[]) {
 int size = array.length;

 for (int step = 0; step < size - 1; step++) {
 int min_idx = step;

 for (int i = step + 1; i < size; i++) {

 // To sort in descending order, change > to < in this line.

 // Select the minimum element in each loop.

 if (array[i] < array[min_idx]) {
 min_idx = i;
 }
 }
 }
 }
}
```

```

 }

}

// put min at the correct position
int temp = array[step];
array[step] = array[min_idx];
array[min_idx] = temp;
}

}

// driver code
public static void main(String args[]) {
 int[] data = { 20, 12, 10, 15, 2 };
 SelectionSort ss = new SelectionSort();
 ss.selectionSort(data);
 System.out.println("Sorted Array in Ascending Order: ");
 System.out.println(Arrays.toString(data));
}
}

```

### Complexity

Number of comparisons:  $(n - 1) + (n - 2) + (n - 3) + \dots + 1 = n(n - 1) / 2$  nearly equals to  $n^2$ .

**Complexity** =  $O(n^2)$

Also, we can analyze the complexity by simply observing the number of loops. There are 2 loops so the complexity is  $n*n = n^2$ .

### Time Complexities:

- **Worst Case Complexity:**  $O(n^2)$  If we want to sort in ascending order and the array is in descending order then, the worst case occurs.
- **Best Case Complexity:**  $O(n^2)$  It occurs when the array is already sorted
- **Average Case Complexity:**  $O(n^2)$  It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

The time complexity of the selection sort is the same in all cases. At every step, you have to find the minimum element and put it in the right place. The minimum element is not known until the end of the array is not reached.

### **Space Complexity:**

Space complexity is  $O(1)$  because an extra variable temp is used.

### **Applications**

The selection sort is used when

- a small list is to be sorted
- cost of swapping does not matter
- checking of all the elements is compulsory
- cost of writing to a memory matters like in flash memory (number of writes/swaps is  $O(n)$  as compared to  $O(n^2)$  of bubble sort)

### **Bubble Sort**

**Bubble sort** is a sorting algorithm that compares two adjacent elements and swaps them until they are in the intended order.

Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a bubble sort.

### **Working**

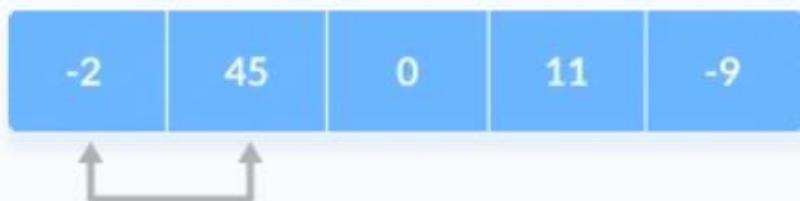
Suppose we are trying to sort the elements in **ascending order**.

#### **1. First Iteration (Compare and Swap)**

1. Starting from the first index, compare the first and the second elements.
2. If the first element is greater than the second element, they are swapped.
3. Now, compare the second and the third elements. Swap them if they are not in order.
4. The above process goes on until the last element.

**step = 0**

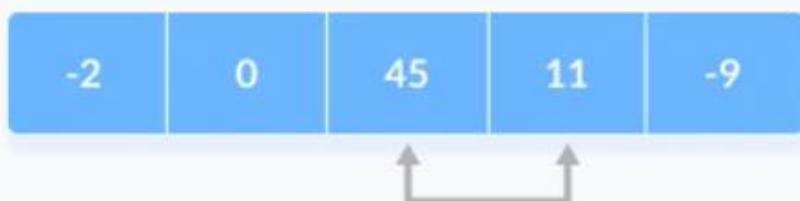
**i = 0**



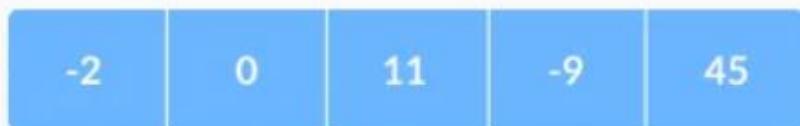
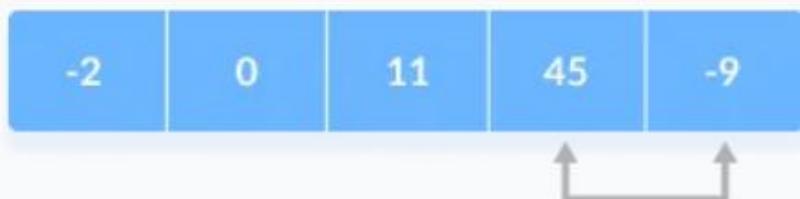
**i = 1**



**i = 2**



**i = 3**

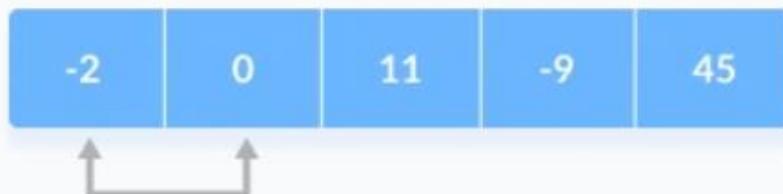


## 2. Remaining Iteration

The same process goes on for the remaining iterations. After each iteration, the largest element among the unsorted elements is placed at the end.

**step = 1**

**i = 0**



**i = 1**



**i = 2**



Put the largest element at the end

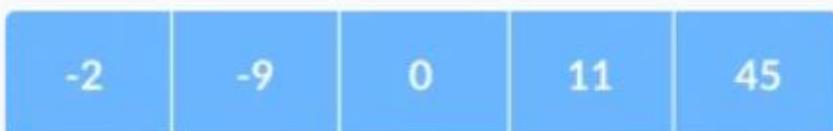
In each iteration, the comparison takes place up to the last unsorted element.

**step = 2**

**i = 0**



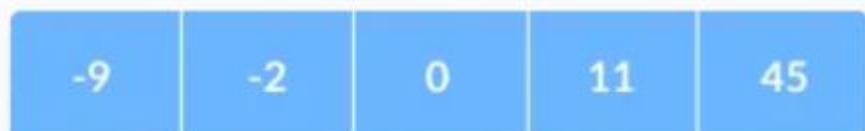
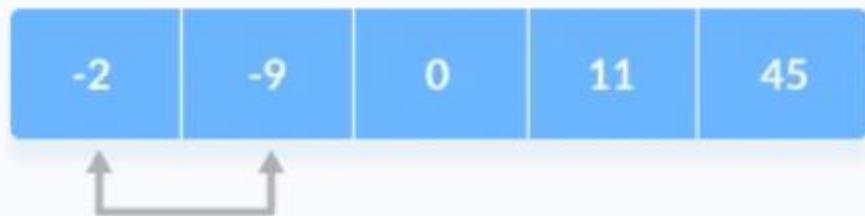
**i = 1**



The array is sorted when all the unsorted elements are placed at their correct positions.

**step = 3**

**i = 0**



**Algorithm**



```
bubbleSort(array)
for i <- 1 to indexOfLastUnsortedElement-1
 if leftElement > rightElement
 swap leftElement and rightElement
end bubbleSort
```

#### Code



```
import java.util.Arrays;
```

```
class Main {
```

```
 // perform the bubble sort
 static void bubbleSort(int array[]) {
 int size = array.length;

 // loop to access each array element
 for (int i = 0; i < size - 1; i++) {

 // loop to compare array elements
 for (int j = 0; j < size - i - 1; j++) {

 // compare two adjacent elements
 // change > to < to sort in descending order
 if (array[j] > array[j + 1]) {

 // swapping occurs if elements
 // are not in the intended order
 }
 }
 }
 }
}
```

```

 int temp = array[j];
 array[j] = array[j + 1];
 array[j + 1] = temp;
 }

}

public static void main(String args[]) {

 int[] data = { -2, 45, 0, 11, -9 };

 // call method using class name
 Main.bubbleSort(data);

 System.out.println("Sorted Array in Ascending Order:");
 System.out.println(Arrays.toString(data));
}
}

```

### **Optimized Bubble Sort**

In the above algorithm, all the comparisons are made even if the array is already sorted.

This increases the execution time.

To solve this, we can introduce an extra variable swapped. The value of swapped is set true if there occurs swapping of elements. Otherwise, it is set false.

After an iteration, if there is no swapping, the value of swapped will be false. This means elements are already sorted and there is no need to perform further iterations.

This will reduce the execution time and helps to optimize the bubble sort.

### **Algorithm**



bubbleSort(array)

swapped <- false

```
for i <- 1 to indexOfLastUnsortedElement-1
 if leftElement > rightElement
 swap leftElement and rightElement
 swapped <- true
 end bubbleSort
```

### Code



```
import java.util.Arrays;
```

```
class Main {

 // perform the bubble sort
 static void bubbleSort(int array[]) {
 int size = array.length;

 // loop to access each array element
 for (int i = 0; i < (size-1); i++) {

 // check if swapping occurs
 boolean swapped = false;

 // loop to compare adjacent elements
 for (int j = 0; j < (size-i-1); j++) {

 // compare two array elements
 // change > to < to sort in descending order
 if (array[j] > array[j + 1]) {
```

```

// swapping occurs if elements
// are not in the intended order

int temp = array[j];
array[j] = array[j + 1];
array[j + 1] = temp;

swapped = true;
}

}

// no swapping means the array is already sorted
// so no need for further comparison
if (!swapped)
 break;

}

}

public static void main(String args[]) {

 int[] data = { -2, 45, 0, 11, -9 };

 // call method using the class name
 Main.bubbleSort(data);

 System.out.println("Sorted Array in Ascending Order:");
 System.out.println(Arrays.toString(data));
}
}

```

## Complexity

Bubble Sort compares the adjacent elements.

Hence, the number of comparisons is



$$(n-1) + (n-2) + (n-3) + \dots + 1 = n(n-1)/2$$

nearly equals to  $n^2$

Hence, **Complexity:**  $O(n^2)$

Also, if we observe the code, bubble sort requires two loops. Hence, the complexity is  $n*n = n^2$

## 1. Time Complexities

- **Worst Case Complexity:**  $O(n^2)$  If we want to sort in ascending order and the array is in descending order then the worst case occurs.
- **Best Case Complexity:**  $O(n)$  If the array is already sorted, then there is no need for sorting.
- **Average Case Complexity:**  $O(n^2)$  It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

## 2. Space Complexity

- Space complexity is  $O(1)$  because an extra variable is used for swapping.
- In the **optimized bubble sort algorithm**, two extra variables are used. Hence, the space complexity will be  $O(2)$ .

## Applications

Bubble sort is used if

- complexity does not matter
- short and simple code is preferred

## Conclusion

We've discussed basic sorting algorithms in depth in this session, the following session will cover Merge Sort and Quick Sort algorithms.

## Interview Questions

### 1. Why Sorting algorithms are important?

Efficient sorting is important for **optimizing the efficiency of other algorithms** (such as search and merge algorithms) that require input data to be in sorted lists. Sorting is also often useful for canonicalizing data and for producing human-readable output. Sorting have direct applications in database algorithms, divide and conquer methods, data structure algorithms, and many more.

### 2. Explain what is ideal Sorting algorithm?

The **Ideal Sorting Algorithm** would have the following properties:

- **Stable:** Equal keys aren't reordered.
- **Operates in place:** requiring  $O(1)$  extra space.
- Worst-case  $O(n \log n)$  key comparisons.
- Worst-case  $O(n)$  swaps.
- **Adaptive:** Speeds up to  $O(n)$  when data is nearly sorted or when there are few unique keys.

There is **no** algorithm that has all of these properties, and so the choice of sorting algorithm depends on the application.

### 3. What is meant by "Sort in Place"?

## Agenda:

- Examples
- Comparator sort

In the last session, we have seen sorting algorithms and now in this session we will see different examples on these algorithms:

### Examples:

#### Maximum Subarray Sum using Divide and Conquer

Given an integer array, find the maximum sum among all subarrays possible.

The problem differs from the problem of finding the maximum subsequence sum. Unlike subsequences, subarrays are required to occupy consecutive positions within the original array.

For example,



\*\*Input:\*\* nums[] = [2, -4, \*\*1, 9, -6, 7\*\*, -3]

\*\*Output:\*\* The maximum sum of the subarray is 11 (Marked in \*\*Green\*\*)

A naive solution is to consider every possible subarray, find the sum, and take the maximum. The problem with this approach is that its worst-case time complexity is **O(n<sup>2</sup>)**, where n is the size of the input.

Following is the java program that demonstrates it:



```
class Main
{
 // A naive solution to finding maximum subarray sum using
 // divide-and-conquer
 public static int findMaximumSum(int[] nums)
 {
 int maxSum = Integer.MIN_VALUE;
 int sum = 0;
```

```
// do for each subarray starting with `i`
for (int i = 0; i < nums.length; i++)
{
 // calculate the sum of subarray `nums[i...j]`

 sum = 0; // reset sum

 // do for each subarray ending at `j`
 for (int j = i; j < nums.length; j++)
 {
 sum += nums[j];

 // if the current subarray sum is greater than the maximum
 // sum calculated so far, update the maximum sum
 if (sum > maxSum) {

 maxSum = sum;
 }
 }
}

return maxSum;
}

public static void main(String[] args)
{
 int[] nums = { 2, -4, 1, 9, -6, 7, -3 };
 System.out.println("The maximum sum of the subarray is "
 + findMaximumSum(nums));
}
```

}

Output:



The maximum sum of the subarray is 11

### How can we perform better?

The idea is to use Divide and Conquer technique to find the maximum subarray sum. The algorithm works as follows:

- Divide the array into two equal subarrays.
- Recursively calculate the maximum subarray sum for the left subarray,
- Recursively calculate the maximum subarray sum for the right subarray,
- Find the maximum subarray sum that crosses the middle element.
- Return the maximum of the above three sums.

The algorithm can be implemented as follows in java:



```
class Main
{
 // Function to find the maximum subarray sum using divide-and-conquer
 public static int findMaximumSum(int[] nums, int left, int right)
 {
 // If the array contains 0 or 1 element
 if (right == left) {
 return nums[left];
 }

 // Find the middle array element
 int mid = (left + right) / 2;

 // Find maximum subarray sum for the left subarray,
 }
}
```

```

// including the middle element

int leftMax = Integer.MIN_VALUE;

int sum = 0;

for (int i = mid; i >= left; i--)
{
 sum += nums[i];

 if (sum > leftMax) {
 leftMax = sum;
 }
}

// Find maximum subarray sum for the right subarray,
// excluding the middle element

int rightMax = Integer.MIN_VALUE;

sum = 0; // reset sum to 0

for (int i = mid + 1; i <= right; i++)
{
 sum += nums[i];

 if (sum > rightMax) {
 rightMax = sum;
 }
}

// Recursively find the maximum subarray sum for the left
// and right subarray, and take maximum

int maxLeftRight = Integer.max(findMaximumSum(nums, left, mid),
 findMaximumSum(nums, mid + 1, right));

// return the maximum of the three

```

```

 return Integer.max(maxLeftRight, leftMax + rightMax);
 }

// Wrapper over findMaximumSum() function

public static int findMaximumSum(int[] nums)
{
 // base case
 if (nums == null || nums.length == 0) {
 return 0;
 }

 return findMaximumSum(nums, 0, nums.length - 1);
}

public static void main(String[] args)
{
 int[] nums = { 2, -4, 1, 9, -6, 7, -3 };

 System.out.println("The maximum sum of the subarray is " +
 findMaximumSum(nums));
}
}

```

Output:



The maximum sum of the subarray is 11

### **Find the minimum and maximum element in an array using Divide and Conquer**

Given an integer array, find the minimum and maximum element present in it by making minimum comparisons by using the divide-and-conquer technique.

For example,



Input: nums = [5, 7, 2, 4, 9, 6]

Output:

The minimum array element is 2

The maximum array element is 9

We can easily solve this problem by using Divide and Conquer. The idea is to recursively divide the array into two equal parts and update the maximum and minimum of the whole array in recursion by passing minimum and maximum variables by reference. The base conditions for the recursion will be when the subarray is of length 1 or 2. The following solution in Java handles all the comparisons efficiently in base conditions:



```
// nums Pair class to wrap immutable primitive ints
class Pair
{
 public int max, min;

 public Pair(int max, int min)
 {
 this.max = max;
 this.min = min;
 }
}

class Main
{
 // Divide and conquer solution to find the minimum and maximum number
 // in an array
```

```
public static void findMinAndMax(int[] nums, int left, int right, Pair p)
{
 // if the array contains only one element

 if (left == right) // common comparison
 {
 if (p.max < nums[left]) { // comparison 1
 p.max = nums[left];
 }

 if (p.min > nums[right]) { // comparison 2
 p.min = nums[right];
 }
 }

 return;
}

// if the array contains only two elements

if (right - left == 1) // common comparison
{
 if (nums[left] < nums[right]) // comparison 1
 {
 if (p.min > nums[left]) { // comparison 2
 p.min = nums[left];
 }

 if (p.max < nums[right]) { // comparison 3
 p.max = nums[right];
 }
 }
}
```

```
 }

}

else {

 if (p.min > nums[right]) { // comparison 2
 p.min = nums[right];
 }

 if (p.max < nums[left]) { // comparison 3
 p.max = nums[left];
 }

}

return;

}

// find the middle element
int mid = (left + right) / 2;

// recur for the left subarray
findMinAndMax(nums, left, mid, p);

// recur for the right subarray
findMinAndMax(nums, mid + 1, right, p);

}

public static void main(String[] args)
{
 int[] nums = { 7, 2, 9, 3, 1, 6, 7, 8, 4 };
}
```

```
// initialize the minimum element by INFINITY and the
// maximum element by -INFINITY

Pair p = new Pair(Integer.MIN_VALUE, Integer.MAX_VALUE);

findMinAndMax(nums, 0, nums.length - 1, p);

System.out.println("The minimum array element is " + p.min);
System.out.println("The maximum array element is " + p.max);

}
```

Output:



The minimum array element is 1

The maximum array element is 9

### **Find the first or last occurrence of a given number in a sorted array**



Input:

nums = [2, 5, 5, 5, 6, 6, 8, 9, 9, 9]

target = 5

Output:

The first occurrence of element 5 is located at index 1

The last occurrence of element 5 is located at index 3

Input:

```
nums = [2, 5, 5, 5, 6, 6, 8, 9, 9, 9]
target = 4
```

Output:

Element not found in the array

Solution:



```
class Main
{
 // Function to find the first occurrence of a given number
 // in a sorted integer array

 public static int findFirstOccurrence(int[] nums, int target)
 {
 // search space is nums[left...right]
 int left = 0;
 int right = nums.length - 1;

 // initialize the result by -1
 int result = -1;

 // loop till the search space is exhausted
 while (left <= right)
 {
 // find the mid-value in the search space and compares it with the target
 int mid = (left + right) / 2;

 // if the target is located, update the result and
 }
 }
}
```

```
// search towards the left (lower indices)
if (target == nums[mid])
{
 result = mid;
 right = mid - 1;
}

// if the target is less than the middle element, discard the right half
else if (target < nums[mid]) {
 right = mid - 1;
}

// if the target is more than the middle element, discard the left half
else {
 left = mid + 1;
}

// return the leftmost index, or -1 if the element is not found
return result;
}

public static void main(String[] args)
{
 int[] nums = {2, 5, 5, 5, 6, 6, 8, 9, 9, 9};
 int target = 5;

 int index = findFirstOccurrence(nums, target);
```

```

if (index != -1)
{
 System.out.println("The first occurrence of element " + target +
 " is located at index " + index);
}
else {
 System.out.println("Element not found in the array");
}
}

```

### **Comparator Interface in Java**

In order to rank the objects of user-defined classes, a comparator interface is utilised. A comparator object can compare two objects that belong to the same class. The method below compares object 1 and object 2

#### **Syntax:**



```
public int compare(Object obj1, Object obj2):
```

Suppose we have an Array/ArrayList of our own class type, containing fields like roll no, name, address, DOB, etc, and we need to sort the array based on Roll no or name?

**Method 1:** One obvious approach is to write our own sort() function using one of the standard algorithms. This solution requires rewriting the whole sorting code for different criteria like Roll No. and Name.

**Method 2:** Utilizing the comparator interface: The objects of a user-defined class are sorted using the comparator interface. This interface is part of the java.util package and includes the methods equals and compare(Object obj1, Object obj2) (Object element). We can arrange the elements according to the data members using a comparator. It could be on the roll number, name, age, or anything else, for instance.

Method of Collections class for sorting List elements is used to sort the elements of List by the given comparator.



```
public void sort(List list, ComparatorClass c)
```

To sort a given List, ComparatorClass must implement a Comparator interface.

### **How do the sort() method of Collections class work?**

The Compare method of the classes that the Sort function is sorting does get called internally. It asks "Which is greater?" to compare two items. If one value is less than, equal to, or larger than the other, the compare method returns -1, 0, or 1. It then utilises this outcome to decide whether they ought to be swapped for their sort.

#### **Example:**



```
// Java Program to Demonstrate Working of
// Comparator Interface

// Importing required classes

import java.io.*;

import java.lang.*;

import java.util.*;

// Class 1
// A class to represent a Student
class Student {

 // Attributes of a student
 int rollno;
 String name, address;

 // Constructor
 public Student(int rollno, String name, String address)
 {

 // This keyword refers to current instance itself
```

```
this.rollno = rollno;
this.name = name;
this.address = address;
}

// Method of Student class
// To print student details in main()
public String toString()
{
 // Returning attributes of Student
 return this.rollno + " " + this.name + " "
 + this.address;
}

}

// Class 2
// Helper class implementing Comparator interface
class Sortbyroll implements Comparator<Student> {

 // Method
 // Sorting in ascending order of roll number
 public int compare(Student a, Student b)
 {
 return a.rollno - b.rollno;
 }
}
```

```
// Class 3

// Helper class implementing Comparator interface

class Sortbyname implements Comparator<Student> {

 // Method

 // Sorting in ascending order of name

 public int compare(Student a, Student b)

 {

 return a.name.compareTo(b.name);

 }

}
```

```
// Class 4

// Main class

class Alma {

 // Main driver method

 public static void main(String[] args)

 {

 // Creating an empty ArrayList of Student type

 ArrayList<Student> ar = new ArrayList<Student>();

 // Adding entries in above List

 // using add() method

 ar.add(new Student(111, "Mayank", "london"));

 ar.add(new Student(131, "Anshul", "nyc"));

 ar.add(new Student(121, "Solanki", "jaipur"));

 }

}
```

```
ar.add(new Student(101, "Aggarwal", "Hongkong"));

// Display message on console for better readability
System.out.println("Unsorted");

// Iterating over entries to print them
for (int i = 0; i < ar.size(); i++)
 System.out.println(ar.get(i));

// Sorting student entries by roll number
Collections.sort(ar, new Sortbyroll());

// Display message on console for better readability
System.out.println("\nSorted by rollno");

// Again iterating over entries to print them
for (int i = 0; i < ar.size(); i++)
 System.out.println(ar.get(i));

// Sorting student entries by name
Collections.sort(ar, new Sortbyname());

// Display message on console for better readability
System.out.println("\nSorted by name");

// // Again iterating over entries to print them
for (int i = 0; i < ar.size(); i++)
 System.out.println(ar.get(i));

}
```

}

**Output:**



Unsorted

111 Mayank london

131 Anshul nyc

121 Solanki jaipur

101 Aggarwal Hongkong

Sorted by rollno

101 Aggarwal Hongkong

111 Mayank london

121 Solanki jaipur

131 Anshul nyc

Sorted by name

101 Aggarwal Hongkong

131 Anshul nyc

111 Mayank london

121 Solanki jaipur

**Sort collection by more than one field**

In the previous example, we have discussed how to sort the list of objects on the basis of a single field using Comparable and Comparator interface But, what if we have a requirement to sort ArrayList objects in accordance with more than one field like firstly, sort according to the student name and secondly, sort according to student age.



// Java Program to Demonstrate Working of

// Comparator Interface Via More than One Field

```
// Importing required classes

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.Iterator;
import java.util.List;

// Class 1

// Helper class representing a Student

class Student {

 // Attributes of student

 String Name;
 int Age;

 // Parameterized constructor

 public Student(String Name, Integer Age)
 {

 // This keyword refers to current instance itself

 this.Name = Name;
 this.Age = Age;
 }

 // Getter setter methods

 public String getName() { return Name; }

 public void setName(String Name) { this.Name = Name; }
```

```
public Integer getAge() { return Age; }

public void setAge(Integer Age) { this.Age = Age; }

// Method

// Overriding toString() method

@Override public String toString()

{

 return "Customer{"

 + "Name=" + Name + ", Age=" + Age + '}';

}

// Class 2

// Helper class implementing Comparator interface

static class CustomerSortingComparator

 implements Comparator<Student> {

 // Method 1

 // To compare customers

 @Override

 public int compare(Student customer1,

 Student customer2)

 {

 // Comparing customers

 int NameCompare = customer1.getName().compareTo(

 customer2.getName());

 int AgeCompare = customer1.getAge().compareTo(
```

```
 customer2.getAge());

 // 2nd level comparison
 return (NameCompare == 0) ? AgeCompare
 : NameCompare;
 }
}

// Method 2
// Main driver method
public static void main(String[] args)
{

 // Create an empty ArrayList
 // to store Student
 List<Student> al = new ArrayList<>();

 // Create customer objects
 // using constructor initialization
 Student obj1 = new Student("Ajay", 27);
 Student obj2 = new Student("Sneha", 23);
 Student obj3 = new Student("Simran", 37);
 Student obj4 = new Student("Ajay", 22);
 Student obj5 = new Student("Ajay", 29);
 Student obj6 = new Student("Sneha", 22);

 // Adding customer objects to ArrayList
 // using add() method
 al.add(obj1);
```

```
al.add(obj2);
al.add(obj3);
al.add(obj4);
al.add(obj5);
al.add(obj6);

// Iterating using Iterator
// before Sorting ArrayList
Iterator<Student> custIterator = al.iterator();

// Display message
System.out.println("Before Sorting:\n");

// Holds true till there is single element
// remaining in List
while (custIterator.hasNext()) {

 // Iterating using next() method
 System.out.println(custIterator.next());
}

// Sorting using sort method of Collections class
Collections.sort(al,
 new CustomerSortingComparator());

// Display message only
System.out.println("\n\nAfter Sorting:\n");

// Iterating using enhanced for-loop
```

```
// after Sorting ArrayList

for (Student customer : al) {
 System.out.println(customer);
}
}
```

**Output:**



Before Sorting:

```
Customer{Name=Ajay, Age=27}
Customer{Name=Sneha, Age=23}
Customer{Name=Simran, Age=37}
Customer{Name=Ajay, Age=22}
Customer{Name=Ajay, Age=29}
Customer{Name=Sneha, Age=22}
```

After Sorting:

```
Customer{Name=Ajay, Age=22}
Customer{Name=Ajay, Age=27}
Customer{Name=Ajay, Age=29}
Customer{Name=Simran, Age=37}
Customer{Name=Sneha, Age=22}
Customer{Name=Sneha, Age=23}
```

## **Agenda:**

- **Divide and Conquer algorithm**
- **Merge Sort**
- **Quick Sort**

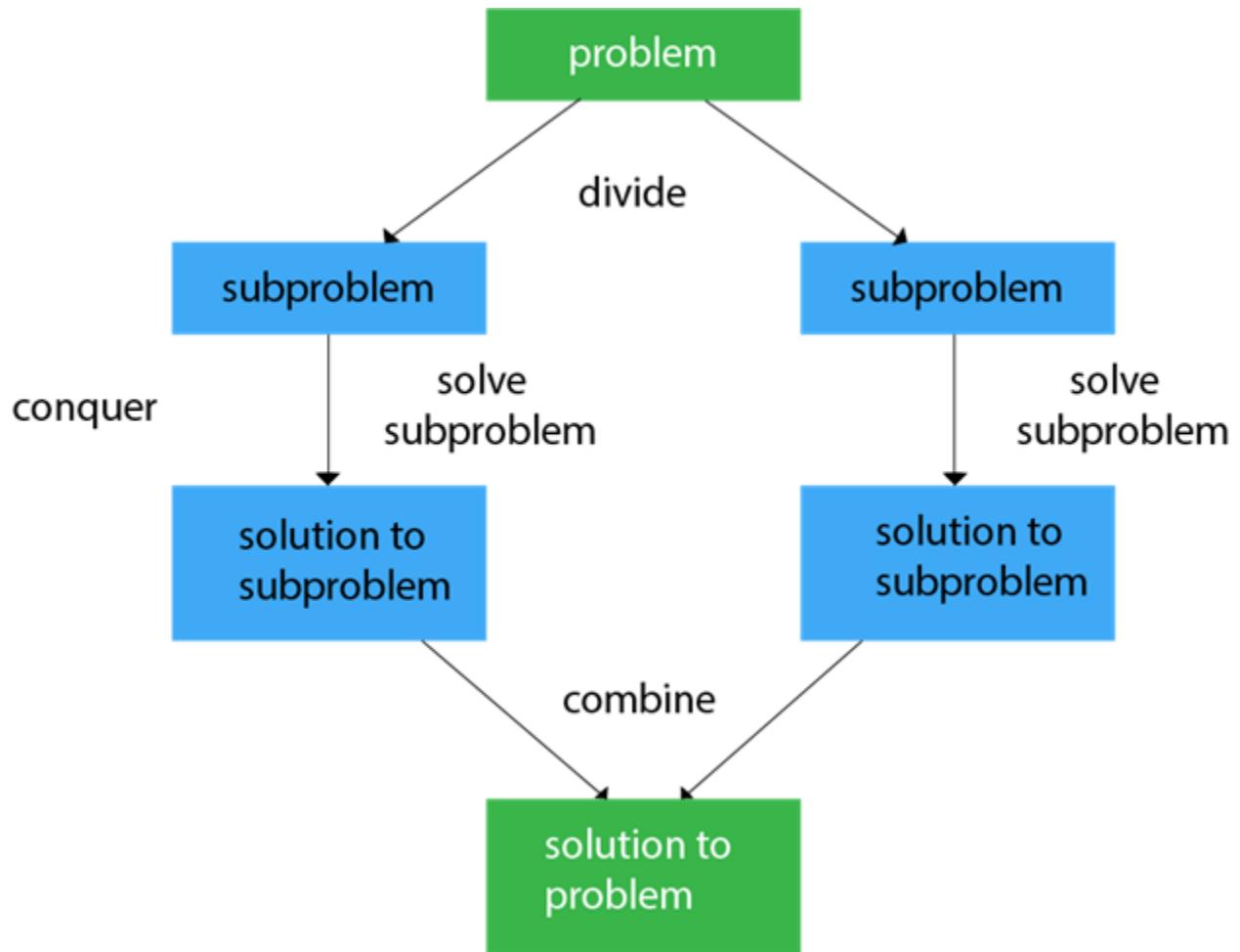
We have learnt about sorting in last session, now we are going learn different algorithms regarding sorting.

### **Divide and Conquer Algorithm**

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of subproblems.
2. **Conquer:** Solve every subproblem individually, recursively.
3. **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.



**Examples:** The specific computer algorithms are based on the Divide & Conquer approach:

1. Maximum and Minimum Problem
2. Binary Search
3. Sorting (merge sort, quick sort)

#### Fundamental of Divide and Conquer algorithm

There are two fundamental of Divide & Conquer Strategy:

1. Relational Formula
2. Stopping Condition

**1. Relational Formula:** It is the formula that we generate from the given technique. After generation of Formula we apply D&C Strategy, i.e. we break the problem recursively & solve the broken subproblems.

**2. Stopping Condition:** When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion steps of D&C is called as Stopping Condition.

## **Applications of Divide and Conquer algorithm**

Following algorithms are based on the concept of the Divide and Conquer Technique:

1. **Binary Search:** The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search. It works by comparing the target value with the middle element existing in a sorted array. After making the comparison, if the value differs, then the half that cannot contain the target will eventually eliminate, followed by continuing the search on the other half. We will again consider the middle element and compare it with the target value. The process keeps on repeating until the target value is met. If we found the other half to be empty after ending the search, then it can be concluded that the target is not present in the array.
2. **Quicksort:** It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub-arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sort the arrays recursively.
3. **Merge Sort:** It is a sorting algorithm that sorts an array by making comparisons. It starts by dividing an array into sub-array and then recursively sorts each of them. After the sorting is done, it merges them back.

## **Advantages of Divide and Conquer algorithm**

- Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle. It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively. This algorithm is much faster than other algorithms.
- It efficiently uses cache memory without occupying much space because it solves simple subproblems within the cache memory instead of accessing the slower main memory.
- It is more proficient than that of its counterpart Brute Force technique.
- Since these algorithms inhibit parallelism, it does not involve any modification and is handled by systems incorporating parallel processing.

## **Disadvantages of Divide and Conquer algorithm**

- Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.
- An explicit stack may overuse the space.
- It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

## **Merge Sort Algorithm**

\*\*Merge sort is a “divide and conquer” algorithm, wherein we first divide the problem into subproblems.\*\*When the solutions for the subproblems are ready, we combine them together to get the final solution to the problem.

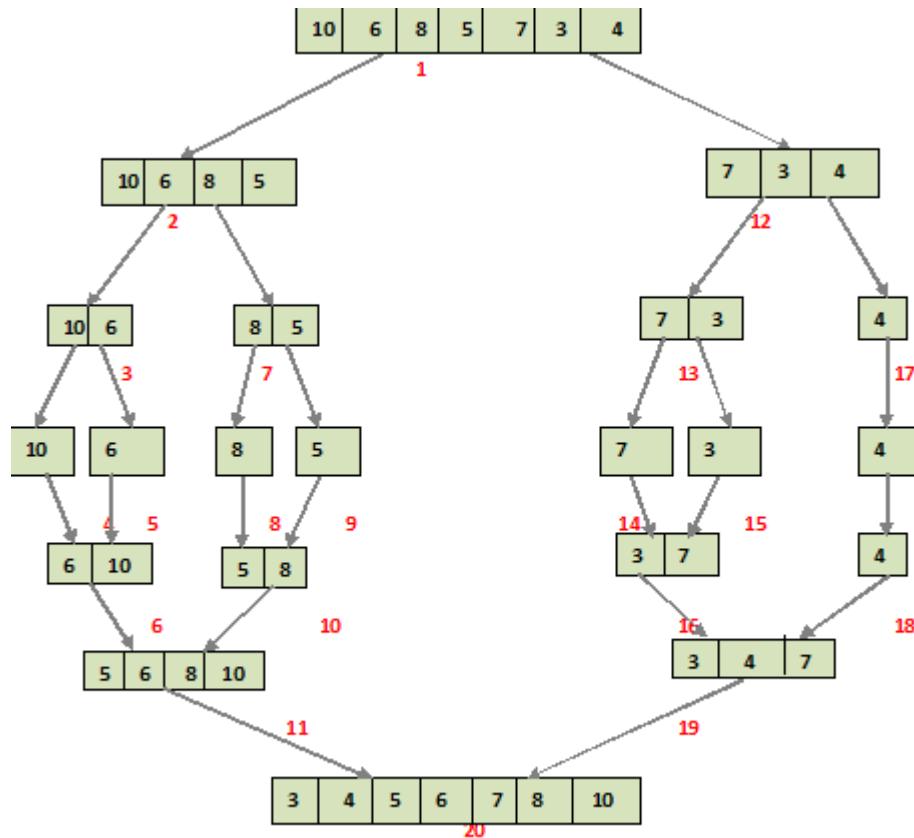
We can easily implement this algorithm using recursion, as we deal with the subproblems rather than the main problem.

We can describe the algorithm as the following 2 step process:

- **Divide:** In this step, we divide the input array into 2 halves, the pivot being the midpoint of the array. This step is carried out recursively for all the half arrays until there are no more half arrays to divide.
- **Conquer:** In this step, we sort and merge the divided arrays from bottom to top and get the sorted array.

The following diagram shows the complete merge sort process for an example array {10, 6, 8, 5, 7, 3, 4}.

If we take a closer look at the diagram, we can see that the array is recursively divided into two halves until the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back while sorting:



### Implementation of Merge sort

For the implementation, we'll write a *mergeSort* function that takes in the input array and its length as the parameters. This will be a recursive function, so we need the base and the recursive conditions.

The base condition checks if the array length is 1 and it will just return. For the rest of the cases, the recursive call will be executed.

**For the recursive case, we get the middle index and create two temporary arrays, *l[]* and *r[]*.** Then we call the *mergeSort* function recursively for both the sub-arrays:



```
public static void mergeSort(int[] a, int n) {
 if (n < 2) {
 return;
 }
 int mid = n / 2;
 int[] l = new int[mid];
 int[] r = new int[n - mid];

 for (int i = 0; i < mid; i++) {
 l[i] = a[i];
 }
 for (int i = mid; i < n; i++) {
 r[i - mid] = a[i];
 }
 mergeSort(l, mid);
 mergeSort(r, n - mid);

 merge(a, l, r, mid, n - mid);
}
```

**Next, we call the *merge* function, which takes in the input and both the sub-arrays, as well as the start and end indices of both the sub arrays.**

**The *merge* function compares the elements of both sub-arrays one by one and places the smaller element into the input array.**

When we reach the end of one of the sub-arrays, the rest of the elements from the other array are copied into the input array, thereby giving us the final sorted array:



```
public static void merge(
 int[] a, int[] l, int[] r, int left, int right) {

 int i = 0, j = 0, k = 0;

 while (i < left && j < right) {
 if (l[i] <= r[j]) {
 a[k++] = l[i++];
 } else {
 a[k++] = r[j++];
 }
 }

 while (i < left) {
 a[k++] = l[i++];
 }

 while (j < right) {
 a[k++] = r[j++];
 }
}
```

The unit test for the program is:



```
@Test
public void positiveTest() {
 int[] actual = { 5, 1, 6, 2, 3, 4 };
 int[] expected = { 1, 2, 3, 4, 5, 6 };
```

```
MergeSort.mergeSort(actual, actual.length);

assertArrayEquals(expected, actual);

}
```

### Complexity of Merge sort

As merge sort is a recursive algorithm, the time complexity can be expressed as the following recursive relation:



$$T(n) = 2T(n/2) + O(n)$$

$2T(n/2)$  corresponds to the time required to sort the sub-arrays, and  $O(n)$  is the time to merge the entire array.

When solved, **the time complexity will come to  $O(n \log n)$ .**

This is true for the worst, average, and best cases, since it'll always divide the array into two and then merge.

The space complexity of the algorithm is  $O(n)$ , as we're creating temporary arrays in every recursive call.

### Program to implement Merge Sort



```
import java.util.Arrays;
```

```
// Merge sort in Java
```

```
class Main {
```

```
 // Merge two sub arrays L and M into array
```

```
 void merge(int array[], int p, int q, int r) {
```

```
 int n1 = q - p + 1;
```

```
 int n2 = r - q;
```

```
 int L[] = new int[n1];
```

```

int M[] = new int[n2];

// fill the left and right array
for (int i = 0; i < n1; i++)
 L[i] = array[p + i];
for (int j = 0; j < n2; j++)
 M[j] = array[q + 1 + j];

// Maintain current index of sub-arrays and main array
int i, j, k;
i = 0;
j = 0;
k = p;

// Until we reach either end of either L or M, pick larger among
// elements L and M and place them in the correct position at A[p..r]
// for sorting in descending
// use if(L[i] >= M[j])
while (i < n1 && j < n2) {
 if (L[i] <= M[j]) {
 array[k] = L[i];
 i++;
 } else {
 array[k] = M[j];
 j++;
 }
 k++;
}

```

```

// When we run out of elements in either L or M,
// pick up the remaining elements and put in A[p..r]

while (i < n1) {
 array[k] = L[i];
 i++;
 k++;
}

while (j < n2) {
 array[k] = M[j];
 j++;
 k++;
}

// Divide the array into two sub arrays, sort them and merge them
void mergeSort(int array[], int left, int right) {
 if (left < right) {

 // m is the point where the array is divided into two sub arrays
 int mid = (left + right) / 2;

 // recursive call to each sub arrays
 mergeSort(array, left, mid);
 mergeSort(array, mid + 1, right);

 // Merge the sorted sub arrays
 merge(array, left, mid, right);
 }
}

```

```
}
```

```
public static void main(String args[]) {

 // created an unsorted array
 int[] array = { 6, 5, 12, 10, 9, 1 };

 Main ob = new Main();

 // call the method mergeSort()
 // pass argument: array, first index and last index
 ob.mergeSort(array, 0, array.length - 1);

 System.out.println("Sorted Array:");
 System.out.println(Arrays.toString(array));
}
}
```

Output:



Unsorted Array:

[6, 5, 12, 10, 9, 1]

Sorted Array:

[1, 5, 6, 9, 10, 12]

### Quick Sort algorithm

**Quick Sort is a sorting algorithm, which is leveraging the divide and conquer principle.** It has an average  $O(n \log n)$  complexity and it's one of the most used sorting algorithms, especially for big data volumes.

**It's important to remember that Quicksort isn't a stable algorithm.** A stable sorting algorithm is an algorithm where the elements with the same values appear in the same order in the sorted output as they appear in the input list.

The input list is divided into two sub-lists by an element called pivot; one sub-list with elements less than the pivot and another one with elements greater than the pivot. This process repeats for each sub-list.

Finally, all sorted sub-lists merge to form the final output.

### Algorithm Steps

1. We choose an element from the list, called the pivot. We'll use it to divide the list into two sub-lists.
2. We reorder all the elements around the pivot – the ones with smaller value are placed before it, and all the elements greater than the pivot after it. After this step, the pivot is in its final position. This is the important partition step.
3. We apply the above steps recursively to both sub-lists on the left and right of the pivot.

As we can see, **quicksort is naturally a recursive algorithm, like every divide and conquer approach.**

Let's take a simple example in order to better understand this algorithm.



Arr[] = {5, 9, 4, 6, 5, 3}

1. Let's suppose we pick 5 as the pivot for simplicity
2. We'll first put all elements less than 5 in the first position of the array: {3, 4, 5, 6, 5, 9}
3. We'll then repeat it for the left sub-array {3,4}, taking 3 as the pivot
4. There are no elements less than 3
5. We apply quicksort on the sub-array in the right of the pivot, i.e. {4}
6. This sub-array consists of only one sorted element
7. We continue with the right part of the original array, {6, 5, 9} until we get the final ordered array

### Choosing the Optimal pivot

The crucial point in QuickSort is to choose the best pivot. The middle element is, of course, the best, as it would divide the list into two equal sub-lists.

But finding the middle element from an unordered list is difficult and time-consuming, that is why we take as pivot the first element, the last element, the median or any other random element.

### Implementation of Quick Sort

The first method is *quickSort()* which takes as parameters the array to be sorted, the first and the last index. First, we check the indices and continue only if there are still elements to be sorted.

We get the index of the sorted pivot and use it to recursively call *partition()* method with the same parameters as the *quickSort()* method, but with different indices:



```
public void quickSort(int arr[], int begin, int end) {
 if (begin < end) {
 int partitionIndex = partition(arr, begin, end);

 quickSort(arr, begin, partitionIndex-1);
 quickSort(arr, partitionIndex+1, end);
 }
}
```

Let's continue with the *partition()* method. For simplicity, this function takes the last element as the pivot. Then, checks each element and swaps it before the pivot if its value is smaller.

By the end of the partitioning, all elements less than the pivot are on the left of it and all elements greater than the pivot are on the right of it. The pivot is at its final sorted position and the function returns this position:



```
private int partition(int arr[], int begin, int end) {
 int pivot = arr[end];
 int i = (begin-1);

 for (int j = begin; j < end; j++) {
 if (arr[j] <= pivot) {
 i++;

 int swapTemp = arr[i];
 arr[i] = arr[j];
 arr[j] = swapTemp;
 }
 }
}
```

```

 }

 int swapTemp = arr[i+1];
 arr[i+1] = arr[end];
 arr[end] = swapTemp;

 return i+1;
}

```

### **Complexity of Quick Sort**

In the best case, the algorithm will divide the list into two equal size sub-lists. So, the first iteration of the full  $n$ -sized list needs  $O(n)$ . Sorting the remaining two sub-lists with  $n/2$  elements takes  $2O(n/2)*$  each. As a result, the QuickSort algorithm has the complexity of  $O(n \log n)$ .

In the worst case, the algorithm will select only one element in each iteration, so  $O(n) + O(n-1) + \dots + O(1)$ , which is equal to  $O(n^2)$ .

On the average QuickSort has  $O(n \log n)$  complexity, making it suitable for big data volumes.

### **Program to implement Quick Sort**



```

import java.util.Arrays;

class Quicksort {

 // method to find the partition position
 static int partition(int array[], int low, int high) {

 // choose the rightmost element as pivot
 int pivot = array[high];

 // pointer for greater element
 int i = (low - 1);

 for (int j = low; j < high; j++) {
 if (array[j] < pivot) {
 i++;
 swap(array, i, j);
 }
 }

 swap(array, i + 1, high);
 return i + 1;
 }

 // utility function to swap
 static void swap(int arr[], int i, int j) {
 int temp = arr[i];
 arr[i] = arr[j];
 arr[j] = temp;
 }
}

```

```
// traverse through all elements
// compare each element with pivot
for (int j = low; j < high; j++) {

 // if element smaller than pivot is found
 // swap it with the greater element pointed by i
 i++;

 // swapping element at i with element at j
 int temp = array[i];
 array[i] = array[j];
 array[j] = temp;
}

}

// swapt the pivot element with the greater element specified by i
int temp = array[i + 1];
array[i + 1] = array[high];
array[high] = temp;

// return the position from where partition is done
return (i + 1);
}

static void quickSort(int array[], int low, int high) {
 if (low < high) {
```

```
// find pivot element such that
// elements smaller than pivot are on the left
// elements greater than pivot are on the right
int pi = partition(array, low, high);

// recursive call on the left of pivot
quickSort(array, low, pi - 1);

// recursive call on the right of pivot
quickSort(array, pi + 1, high);
}
}
}
```

```
// Main class
class Main {
 public static void main(String args[]) {

 int[] data = { 8, 7, 2, 1, 0, 9, 6 };
 System.out.println("Unsorted Array");
 System.out.println(Arrays.toString(data));

 int size = data.length;

 // call quicksort() on array data
 Quicksort.quickSort(data, 0, size - 1);

 System.out.println("Sorted Array in Ascending Order ");
```

```
 System.out.println(Arrays.toString(data));
 }
}
```

Output:



Unsorted Array

[8, 7, 2, 1, 0, 9, 6]

Sorted Array in Ascending Order

[0, 1, 2, 6, 7, 8, 9]

## Conclusion

In this session, we have learnt about

- divide and conquer algorithm for sorting
- merge sort algorithm
- quick sort algorithm

## Interview Questions

- What is the principle behind divide and conquer ?

The divide-and-conquer paradigm is often used to find an optimal solution of a problem. Its basic idea is to **decompose a given problem into two or more similar, but simpler, subproblems, to solve them in turn, and to compose their solutions to solve the given problem.**

- How does Merge Sort works ?

**Merge sort** is a **divide-and-conquer, comparison-based** sorting algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list. Most implementations produce a **stable** sort, which means that the order of equal elements is the same in the input and output. Can be used as **external** sorting (when the data to be sorted is too large to fit into memory).

- How does Quick Sort works ?

**Quick Sort** is a **divide and conquer, comparison, in-place** algorithm. Efficient implementations of Quicksort are **not a stable** sort. When implemented well, it can be about two or three times faster than its main competitors, merge sort and heapsort.

**Quicksort** determines something called a **pivot**, which is a somewhat **arbitrary** element in the collection. Using the pivot point, quicksort **partitions** (or **divides**) the larger unsorted collection into two, smaller lists. It moves all the elements smaller than the pivot to the left (before) the pivot element,

and moves all the elements larger than the pivot to the right (after) the pivot element. Even though the list isn't completely sorted yet, we know that the items are in the correct order in relation to the pivot. This means that we never have to compare elements on the left side of the partition to elements on the right side of the partition. We already know they are in their correct spots in relation to the pivot. The sub-lists are then sorted ***recursively***.

Ideally, partitioning would use the ***median*** of the given values (in the array), but the median can only be found by scanning the whole array and this would slow the algorithm down. In that case the two partitions would be of equal size; In the simplest versions of quick sort an arbitrary element, typically the ***last (rightmost) element*** is used as an estimate (guess) of the median.

### Additional Resources

- <https://medium.com/jeremy-gottfrieds-tech-blog/intro-to-divide-and-conquer-algorithms-9a2d9659c12#:~:text=The%20brute%20force%20version%20will,%20the%20brute%20force%20requires%2017>
- <https://www.codingninjas.com/codestudio/library/why-is-quick-sort-preferred-for-arrays-and-merge-sort-for-linked-lists>
- <https://medium.com/human-in-a-machine-world/quicksort-the-best-sorting-algorithm-6ab461b5a9d0>
- <https://www.freecodecamp.org/news/stability-in-sorting-algorithms-a-treatment-of-equality-fa3140a5a539#:~:text=A%20stable%20sorting%20algorithm%20maintains,after%20the%20collection%20is%20sorted>

Agenda :

- **Introduction to Java Collections**
- **HashMap**
- **HashSet**
- **Iterators**

## **1. Introduction to Java Collections**

Any group of individual objects which are represented as a single unit is known as the collection of the objects. In Java, a separate framework named the “*Java Collection Framework*” has been defined in JDK 1.2 which holds all the collection classes and interface in it.

### **What is a Framework?**

A framework is a set of classes and interfaces which provide a ready-made architecture. An optimal object-oriented design always includes a framework with a collection of classes such that all the classes perform the same kind of task.

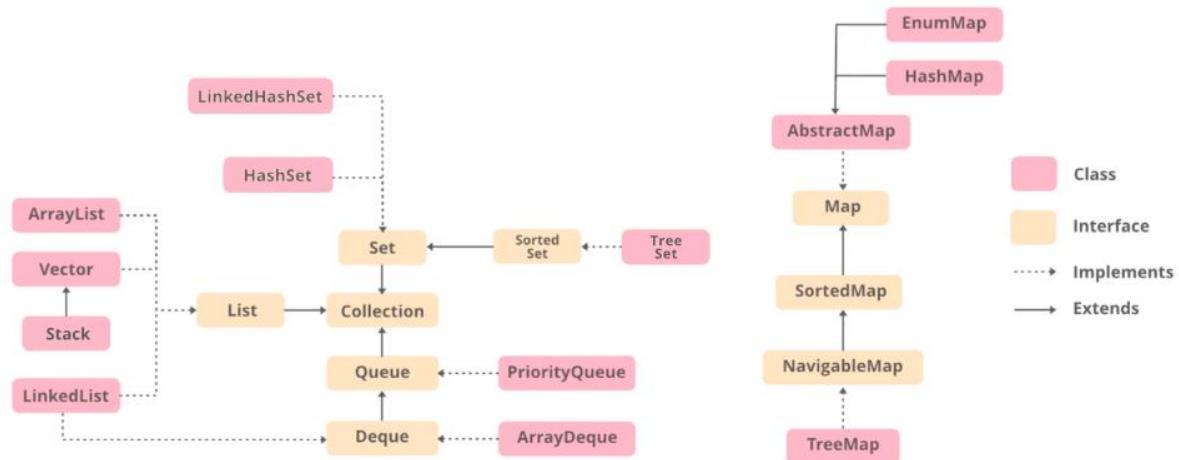
### **Need for a Separate Collection Framework?**

Before the Collection Framework was introduced, the standard methods for grouping Java objects (or collections) were **Arrays** or **Vectors**, or **Hashtables**. All of these collections had no common interface. Therefore, though the main aim of all the collections is the same, the implementation of all these collections was defined independently and had no correlation among them. And also, it is very difficult for the users to remember all the different methods, syntax, and constructors present in every collection class.

### **Hierarchy of the Collection Framework**

The utility package, (java.util) contains all the classes and interfaces that are required by the collection framework. The collection framework contains an interface named an iterable interface which provides the iterator to iterate through all the collections. This interface is extended by the main collection interface which acts as a root for the collection framework. All the collections extend this collection interface thereby extending the properties of the iterator and the methods of this interface.

The following figure illustrates the hierarchy of the collection framework.



The collection framework contains multiple interfaces where every interface is used to store a specific type of data. In this session, we are going to mainly focus on the following :

1. **HashSet** The HashSet class is an inherent implementation of the hash table data structure. The objects that we insert into the HashSet do not guarantee to be inserted in the same order. The objects are inserted based on their hashCode. This class also allows the insertion of NULL elements.
2. **HashMap**

HashMap provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs. To access a value in a HashMap, we must know its key. HashMap uses a technique called Hashing. Hashing is a technique of converting a large String to a small String that represents the same String so that the indexing and search operations are faster. HashSet also uses HashMap internally.

## 2. HashMap

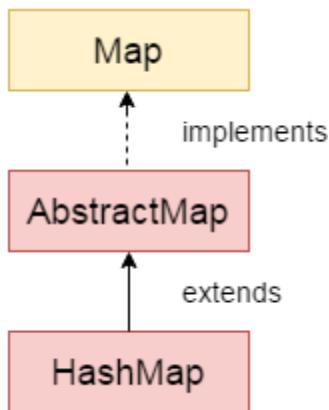
As mentioned earlier, Java **HashMap** class implements the Map interface which allows us *to store key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

### Important Points to keep in Mind

- Java HashMap contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap maintains no order.

### Hierarchy of HashMap class

HashMap class extends AbstractMap class and implements Map interface, as seen in the figure below :



### HashMap class Declaration

Let's see the declaration for java.util.HashMap class :



```
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable
```

### HashMap class Parameters

Let's see the Parameters for java.util.HashMap class.

- **K:** It is the type of keys maintained by this map.
- **V:** It is the type of mapped values.

### Constructors of Java HashMap class

The following table shows the various constructors of the HashMap class :

| Constructor                             | Description                                                                                                                                        |
|-----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| HashMap()                               | It is used to construct a default HashMap.                                                                                                         |
| HashMap(Map<? extends K,? extends V> m) | It is used to initialize the hash map by using the elements of the given Map object m.                                                             |
| HashMap(int capacity)                   | It is used to initializes the capacity of the hash map to the given integer value, capacity.                                                       |
| HashMap(int capacity, float loadFactor) | It is used to initialize both the capacity and load factor of the hash map by using the given integer value, capacity and float value, loadFactor. |

Constructor      Description

HashMap()      It is used to construct a default HashMap.

HashMap(Map<? extends K,? extends V> m)      It is used to initialize the hash map by using the elements of the given Map object m.

HashMap(int capacity)      It is used to initializes the capacity of the hash map to the given integer value, capacity.

`HashMap(int capacity, float loadFactor)` It is used to initialize both the capacity and load factor of the hash map by using its arguments.

### Methods in HashMap

There are various methods present in the `HashMap` class. The table below shows a few of them which are used frequently :

| METHOD                                   | DESCRIPTION                                                                                                  |
|------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| <code>clear()</code>                     | Removes all of the mappings from this map.                                                                   |
| <code>clone()</code>                     | Returns a shallow copy of this <code>HashMap</code> instance: the keys and values themselves are not cloned. |
| <code>containsKey(Object key)</code>     | Returns true if this map contains a mapping for the specified key.                                           |
| <code>containsValue(Object value)</code> | Returns true if this map maps one or more keys to the specified value.                                       |
| <code>get(Object key)</code>             | Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key. |
| <code>isEmpty()</code>                   | Returns true if this map contains no key-value mappings.                                                     |
| <code>put(K key, V value)</code>         | Associates the specified value with the specified key in this map.                                           |
| <code>remove(Object key)</code>          | Removes the mapping for the specified key from this map if present.                                          |
| <code>size()</code>                      | Returns the number of key-value mappings in this map.                                                        |
| <code>values()</code>                    | Returns a Collection view of the values contained in this map.                                               |

### METHOD DESCRIPTION

`clear()` Removes all of the mappings from this map.

`clone()` Returns a shallow copy of this `HashMap` instance: the keys and values themselves are not cloned.

`containsKey(Object key)` Returns true if this map contains a mapping for the specified key.

`containsValue(Object value)` Returns true if this map maps one or more keys to the specified value.

`get(Object key)` Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

`isEmpty()` Returns true if this map contains no key-value mappings.

`put(K key, V value)` Associates the specified value with the specified key in this map.

`remove(Object key)` Removes the mapping for the specified key from this map if present.

`size()` Returns the number of key-value mappings in this map.

`values()` Returns a Collection view of the values contained in this map.

## Performing Basic Operations on HashMap

**1. Adding Elements:** In order to add an element to the map, we can use the **put()** method. However, the insertion order is not retained in the HashMap. Internally, for every element, a separate hash is generated and the elements are indexed based on this hash to make it more efficient.



```
import java.io.*;
import java.util.*;

class AddElementsToHashMap {
 public static void main(String args[])
 {
 HashMap<Integer, String> hm1 = new HashMap<>();
 HashMap<Integer, String> hm2 = new HashMap<Integer, String>();

 // Add Elements using put method
 hm1.put(1, "Alma");
 hm1.put(2, "is");
 hm1.put(3, "Better");

 hm2.put(1, "Full");
 hm2.put(2, "Stack");
 hm2.put(3, "Development");

 System.out.println("Mappings of HashMap hm1 are : " + hm1);
 System.out.println("Mapping of HashMap hm2 are : " + hm2);
 }
}
```

**Output :**



Mappings of HashMap hm1 are : {1=Alma, 2=is, 3=Better}

Mapping of HashMap hm2 are : {1=Full, 2=Stack, 3=Development}

## 2.Removing Element:

In order to remove an element from the Map, we can use the **remove()** method. This method takes the key value and removes the mapping for a key from this map if it is present in the map.



```
import java.io.*;
import java.util.*;

class RemoveElementsOfHashMap{

 public static void main(String args[])
 {

 Map<Integer, String> hm = new HashMap<Integer, String>();

 hm.put(1, "Full");
 hm.put(2, "Stack");
 hm.put(3, "Web");
 hm.put(4, "Development");

 // Initial HashMap
 System.out.println("Mappings of HashMap are : " + hm);

 // remove element with a key
 // using remove method
 hm.remove(4);

 // Final HashMap
 }
}
```

```
 System.out.println("Mappings after removal are : " + hm);
 }
}
```

**Output :**



Mappings of HashMap are : {1=Full, 2=Stack, 3=Web, 4=Development}

Mappings after removal are : {1=Full, 2=Stack, 3=Web}

### **3.Other Useful Methods :**

Consider the following Java implementation to understand the working of other methods of the HashMap class :



```
import java.io.*;
import java.util.*;

class MethodsOfHashMap{

 public static void main(String args[])
 {

 Map<Integer, String> hm = new HashMap<Integer, String>();

 //To check if the HashMap is empty or not.
 System.out.println(hm.isEmpty());

 //Adding Elements in the HashMap.
 hm.put(1, "Full");
 hm.put(2, "Stack");
 hm.put(3, "Web");
 hm.put(4, "Development");
 }
}
```

```
//Printing the size.
System.out.println(hm.size());

//Checking if a key is present in the HashMap.
System.out.println(hm.containsKey(3));

//Checking if a value is present in the HashMap.
System.out.println(hm.containsValue("Stack"));

//Using the method 'get' to store the value corresponding to the provided key.
String val = hm.get(1);
System.out.println(val);
val = hm.get(5);
System.out.println(val);

// Printing all the values present in the HashMap.
System.out.println(hm.values());

//Clearing the HashMap.
hm.clear();
System.out.println(hm.size());
}
}
``
```

\*\*Output :\*\*

```java

```
true  
4  
true  
true  
Full  
null  
[Full, Stack, Web, Development]  
0
```

3. HashSet

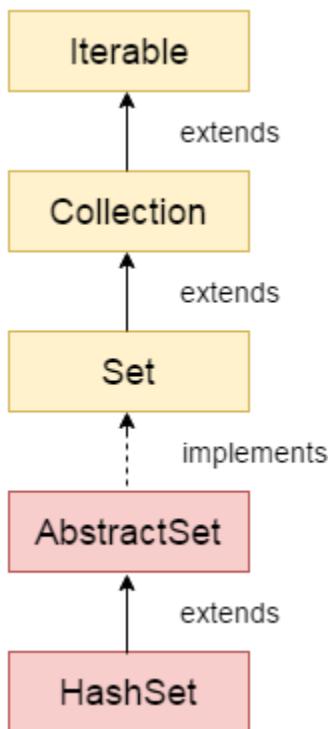
Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

Important Points to keep in Mind

- HashSet stores the elements by using a mechanism called **hashing**.
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashCode.
- HashSet is the best approach for search operations.

Hierarchy of HashSet class

The HashSet class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.



HashSet class declaration

Let's see the declaration for `java.util.HashSet` class.



```
public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable
```

Constructors of Java HashSet class

Methods in HashSet

Similar to the `HashMap`, we have many methods in `HashSet`. The table below shows a few of them which are used frequently :

Performing Basic Operations on HashSet

1.Adding Elements

In order to add an element to the `HashSet`, we can use the **add() method**. However, the insertion order is not retained in the `HashSet`. We need to keep a note that duplicate elements are not allowed and all the duplicate elements are ignored.



```
import java.io.*;
import java.util.*;

class AlmaBetter {

    public static void main(String[] args)
    {
        // Creating an empty HashSet of string entities
        HashSet<String> hs = new HashSet<String>();

        // Adding elements using add() method
        hs.add("Alma");
        hs.add("is");
        hs.add("Better");

        // Printing all string entries inside the Set
        System.out.println("HashSet elements : " + hs);
    }
}
```

Output:



HashSet elements : [Alma, Better, is]

2. Removing Elements

The values can be removed from the HashSet using the **remove()** method.



```
import java.io.*;
import java.util.*;
```

```
class AlmaBetter {  
  
    public static void main(String[] args)  
    {  
  
        HashSet<String> hs = new HashSet<String>();  
  
        // Adding elements to above Set  
        // using add() method  
        hs.add("Full");  
        hs.add("Stack");  
        hs.add("Web");  
        hs.add("3");  
        hs.add("Development");  
        hs.add("Complete");  
  
        // Printing the elements of HashSet elements  
        System.out.println("Initial HashSet " + hs);  
  
        // Removing the element B  
        hs.remove("3");  
  
        // Printing the updated HashSet elements  
        System.out.println("After removing element " + hs);  
  
        // Returns false if the element is not present  
        System.out.println("Element Data exists in the Set : " + hs.remove("Data"));  
    }  
}
```

}

Output :



Initial HashSet [3, Web, Complete, Development, Full, Stack]

After removing element [Web, Complete, Development, Full, Stack]

Element Data exists in the Set : false

3.Other Useful Methods:

Consider the following Java implementation to understand the working of other methods of the HashSet class :



```
import java.io.*;  
import java.util.*;
```

```
class AlmaBetter {
```

```
    public static void main(String[] args)  
    {  
  
        HashSet<String> hs = new HashSet<String>();  
  
        hs.add("Full");  
        hs.add("Stack");  
        hs.add("Course");  
  
        //Checking if the HashSet is empty or not.  
        System.out.println(hs.isEmpty());
```

```

//Checking if an element is present in the HashSet.
System.out.println(hs.contains("Course"));

System.out.println(hs.contains("course"));

//Clearing the HashSet and then printing it's size.

hs.clear();

System.out.println(hs.size());

}

}

```

Output :



false

true

false

0

4.Iterators

In Java, an **Iterator** is one of the Java cursors. **Java Iterator** is an interface that is practised in order to iterate over a collection of Java object components entirely one by one. It is free to use in the Java programming language since the Java 1.2 Collection framework. It belongs to java.util package.

The Java Iterator is also known as the **universal cursor** of Java as it is appropriate for all the classes of the Collection framework. The Java Iterator also helps in the operations like READ and REMOVE. When we compare the Java Iterator interface with the enumeration iterator interface, we can say that the names of the methods available in Java Iterator are more precise and straightforward to use.

Advantages of Java Iterator

. The advantages of Java Iterator are given as follows -

- The user can apply these iterators to any of the classes of the Collection framework.
- In Java Iterator, we can use both of the read and remove operations.

- If a user is working with a for loop, they cannot modernize(add/remove) the Collection, whereas, if they use the Java Iterator, they can simply update the Collection.
- The Java Iterator is considered the Universal Cursor for the Collection API.
- The method names in the Java Iterator are very easy and are very simple to use.

Disadvantages of Java Iterator

Despite the numerous advantages, the Java Iterator has various disadvantages also. The disadvantages of the Java Iterator are given below -

- The Java Iterator only preserves the iteration in the forward direction. In simple words, the Java Iterator is a uni-directional Iterator.
- The replacement and extension of a new component are not approved by the Java Iterator.
- In CRUD Operations, the Java Iterator does not hold the various operations like CREATE and UPDATE.
- In comparison with the Spliterator, Java Iterator does not support traversing elements in the parallel pattern which implies that Java Iterator supports only Sequential iteration.
- In comparison with the Spliterator, Java Iterator does not support more reliable execution to traverse the bulk volume of data.

How to use Java Iterator?

When a user needs to use the Java Iterator, then it's compulsory for them to make an instance of the Iterator interface from the collection of objects they desire to traverse over. After that, the received Iterator maintains the trail of the components in the underlying collection to make sure that the user will traverse over each of the elements of the collection of objects.

If the user modifies the underlying collection while traversing over an Iterator leading to that collection, then the Iterator will typically acknowledge it and will throw an exception in the next time when the user will attempt to get the next component from the Iterator.

Java Iterator Methods

The Java Iterator interface contains a total of four methods that are:

- `hasNext()`
- `next()`
- `remove()`
- `forEachRemaining()`

Let's consider in detail three of the primary methods :

- **boolean hasNext():** The method does not accept any parameter. It returns true if there are more elements left in the iteration. If there are no more elements left, then it will return false.If

there are no more elements left in the iteration, then there is no need to call the next() method. In simple words, we can say that the method is used to determine whether the next() method is to be called or not.

- **E next():** It is similar to hasNext() method. It also does not accept any parameter. It returns E, i.e., the next element in the traversal. If the iteration or collection of objects has no more elements left to iterate, then it throws the NoSuchElementException.
- **default void remove():** This method also does not require any parameters. There is no return type of this method. The main function of this method is to remove the last element returned by the iterator traversing through the underlying collection. The remove () method can be requested hardly once per the next () method call. If the iterator does not support the remove operation, then it throws the UnsupportedOperationException. It also throws the IllegalStateException if the next method is not yet called.

Examples using the Java Iterator

1. Using the method 'hasNext()' and 'next()':

A Java program that demonstrates the usage of the methods '**hasNext()**' and '**next()**' is shown below :



```
import java.io.*;
import java.util.*;

class JavalteratorExample {
    public static void main(String[] args)
    {
        ArrayList<String> cityNames = new ArrayList<String>();

        cityNames.add("Delhi");
        cityNames.add("Mumbai");
        cityNames.add("Kolkata");
        cityNames.add("Chandigarh");
        cityNames.add("Noida");

        // Iterator to iterate the cityNames
```

```
Iterator iterator = cityNames.iterator();

System.out.println("CityNames elements : ");

while (iterator.hasNext())
    System.out.print(iterator.next() + " ");

System.out.println();
}
```

Output :



CityNames elements:

Delhi Mumbai Kolkata Chandigarh Noida

2. Using the method ‘remove()’:

As mentioned earlier, an element can be removed from a Collection using the Iterator method **remove()**. This method removes the current element in the Collection.

Note that if the remove() method is not preceded by the next() method, then the exception *IllegalStateException* is thrown.

A program that demonstrates the usage of the method **‘remove()’** is given here :



```
import java.util.ArrayList;
import java.util.Iterator;

class Demo {
    public static void main(String[] args) {
        ArrayList<String> aList = new ArrayList<String>();
        aList.add("Apple");
    }
}
```

```
aList.add("Mango");
aList.add("Guava");
aList.add("Orange");
aList.add("Peach");

System.out.println("The ArrayList elements are: ");
for (String s: aList) {
    System.out.println(s);
}

Iterator i = aList.iterator();

String str = "";
while (i.hasNext()) {
    str = (String) i.next();
    if (str.equals("Orange")) {
        i.remove();
        System.out.println("\nThe element Orange is removed.\n");
        break;
    }
}

System.out.println("The ArrayList elements are: ");
for (String s: aList) {
    System.out.println(s);
}
```

Output :



The ArrayList elements are:

Apple

Mango

Guava

Orange

Peach

The element Orange is removed.

The ArrayList elements are:

Apple

Mango

Guava

Peach

Points to Remember

- The Java Iterator is an interface added in the Java Programming language in the Java 1.2 Collection framework. It belongs to `java.util` package.
- It is one of the Java Cursors that are practised to traverse the objects of the collection framework.
- The Java Iterator is used to iterate the components of the collection object one by one.
- The Java Iterator is also known as the Universal cursor of Java as it is appropriate for all the classes of the Collection framework.
- The Java Iterator also supports the operations like READ and REMOVE.
- The methods names of the Iterator class are very simple and easy to use compared to the method names of Enumeration Iterator.

Interview Questions

- **What are the advantages of using the Java Collections Framework?**

The Java Collections Framework provides the following benefits:



- Reduces programming effort
- Increases program speed and quality
- Allows interoperability among unrelated APIs

- Reduces effort to learn and to use new APIs
- Reduces effort to design new APIs
- Fosters software reuse

- **What is the difference between the HashMap and the HashSet Class?**

| Basic | HashSet | HashMap |
|--|---|--|
| Implements | Set interface | Map interface |
| Duplicates | No | Yes duplicates values are allowed but no duplicate allowed |
| Dummy values | Yes | No |
| Objects required during an add operation | 1 | 2 |
| Adding and storing mechanism | HashMap object | Hashing technique |
| Speed | It is comparatively slower than HashMap | It is comparatively faster than HashSet because of hashing technique has been used here. |
| Null | Have a single null value | Single null key and any number of null values |
| Insertion Method | Add() | Put() |

Basic HashSet HashMap

Implements Set interface Map interface

Duplicates No Yes duplicates values are allowed but no duplicate key is allowed

Dummy values Yes No

Objects required during an add operation 1 2

Adding and storing mechanism HashMap object Hashing technique

Speed It is comparatively slower than HashMap It is comparatively faster than HashSet because of hashing technique has been used here.

Null Have a single null value Single null key and any number of null values

Insertion Method Add() Put()

Additional Resources

1. Load Factor in HashMap :

[<https://www.javatpoint.com/load-factor-in-hashmap>]

2. Working of Java HashSet :

[<https://www.javatpoint.com/working-of-hashset-in-java>]

3. Iterator vs Enumeration in Java :

[<https://www.geeksforgeeks.org/difference-between-iterator-and-enumeration-in-java-with-examples/>]

4. HashMap.replace() method and it's examples:

[<https://www.geeksforgeeks.org/hashmap-replacekey-value-method-in-java-with-examples/>]

Agenda

- Z Algorithm
- KMP
- Aho Corasick
- String Hashing

Z Algorithm

This algorithm finds all occurrences of a pattern in a text in linear time. Let length of text be n and of pattern be m, then total time taken is $O(m + n)$ with linear space complexity. In this algorithm, we construct a Z array.

What is Z Array?

For a string $\text{str}[0..n-1]$, Z array is of same length as string. An element $Z[i]$ of Z array stores length of the longest substring starting from $\text{str}[i]$ which is also a prefix of $\text{str}[0..n-1]$. The first entry of Z array is meaningless as complete string is always prefix of itself.



Example:

| | |
|----------|---------------------------|
| Index | 0 1 2 3 4 5 6 7 8 9 10 11 |
| Text | a a b c a a b x a a a z |
| Z values | X 1 0 0 3 1 0 0 2 2 1 0 |

How is Z array helpful in Searching Pattern in Linear time?

The idea is to concatenate pattern and text, and create a string “P\$T” where P is pattern, \$ is a special character should not be present in pattern and text, and T is text. Build the Z array for concatenated string. In Z array, if Z value at any point is equal to pattern length, then pattern is present at that point.



Example:

Pattern P = "aab", Text T = "baabaa"

The concatenated string is = "aab\$baabaa"

Z array for above concatenated string is {x, 1, 0, 0, 0, 3, 1, 0, 2, 1}.

Since length of pattern is 3, the value 3 in Z array indicates presence of pattern.

How to construct Z array?

A Simple Solution is to run two nested loops, the outer loop goes to every index and the inner loop finds length of the longest prefix that matches the substring starting at the current index. The time complexity of this solution is $O(n^2)$.

We can construct Z array in linear time.



The idea is to maintain an interval $[L, R]$ which is the interval with max R such that $[L,R]$ is prefix substring (substring which is also prefix).

Steps for maintaining this interval are as follows –

1) If $i > R$ then there is no prefix substring that starts before i and ends after i , so we reset L and R and compute new $[L,R]$ by comparing $\text{str}[0..]$ to $\text{str}[i..]$ and get $Z[i]$ ($= R-L+1$).

2) If $i \leq R$ then let $K = i-L$, now $Z[i] \geq \min(Z[K], R-i+1)$ because $\text{str}[i..]$ matches with $\text{str}[K..]$ for atleast $R-i+1$ characters (they are in $[L,R]$ interval which we know is a prefix substring).

Now two sub cases arise –

a) If $Z[K] < R-i+1$ then there is no prefix substring starting at $\text{str}[i]$ (otherwise $Z[K]$ would be larger) so $Z[i] = Z[K]$ and interval $[L,R]$ remains same.

b) If $Z[K] \geq R-i+1$ then it is possible to extend the $[L,R]$ interval thus we will set L as i and start matching from $\text{str}[R]$ onwards and get new R then we will update interval $[L,R]$ and calculate $Z[i]$ ($=R-L+1$).

The algorithm runs in linear time because we never compare character less than R and with matching we increase R by one so there are at most T comparisons. In mismatch case, mismatch happen only once

for each i (because of which R stops), that's another at most T comparison making overall linear complexity.

Below is the implementation of Z algorithm for pattern searching.



```
// A Java program that implements Z algorithm for pattern
// searching

class ZAlgo {

    // prints all occurrences of pattern in text using
    // Z algo

    public static void search(String text, String pattern)
    {

        // Create concatenated string "P$T"
        String concat = pattern + "$" + text;

        int l = concat.length();

        int Z[] = new int[l];

        // Construct Z array
        getZarr(concat, Z);

        // now looping through Z array for matching condition
        for(int i = 0; i < l; ++i){

            // if Z[i] (matched region) is equal to pattern
            // length we got the pattern
        }
    }
}
```

```

        if(Z[i] == pattern.length()){

            System.out.println("Pattern found at index "
                + (i - pattern.length() - 1));

        }

    }

}

// Fills Z array for given string str[]

private static void getZarr(String str, int[] Z) {

    int n = str.length();

    // [L,R] make a window which matches with
    // prefix of s

    int L = 0, R = 0;

    for(int i = 1; i < n; ++i) {

        // if i>R nothing matches so we will calculate.
        // Z[i] using naive way.

        if(i > R){

            L = R = i;

            // R-L = 0 in starting, so it will start
            // checking from 0'th index. For example,
            // for "ababab" and i = 1, the value of R
            // remains 0 and Z[i] becomes 0. For string
            // "aaaaaa" and i = 1, Z[i] and R become 5
    }
}

```

```

while(R < n && str.charAt(R - L) == str.charAt(R))

    R++;

    Z[i] = R - L;

    R--;

}

else{

    // k = i-L so k corresponds to number which

    // matches in [L,R] interval.

    int k = i - L;

    // if Z[k] is less than remaining interval

    // then Z[i] will be equal to Z[k].

    // For example, str = "ababab", i = 3, R = 5

    // and L = 2

    if(Z[k] < R - i + 1)

        Z[i] = Z[k];

    // For example str = "aaaaaa" and i = 2, R is 5,

    // L is 0

    else{

        // else start from R and check manually

        L = i;

        while(R < n && str.charAt(R - L) == str.charAt(R))

            R++;

```

```

        Z[i] = R - L;
        R--;
    }
}

}

// Driver program
public static void main(String[] args)
{
    String text = "AlmaBetter";
    String pattern = "Alma";

    search(text, pattern);
}

/*
OUTPUT
Pattern found at index 0
*/

```

Time Complexity: $O(m+n)$, where m is length of pattern and n is length of text.

Auxiliary Space: $O(m+n)$

KMP (Knuth-Morris-Pratt) Algorithm

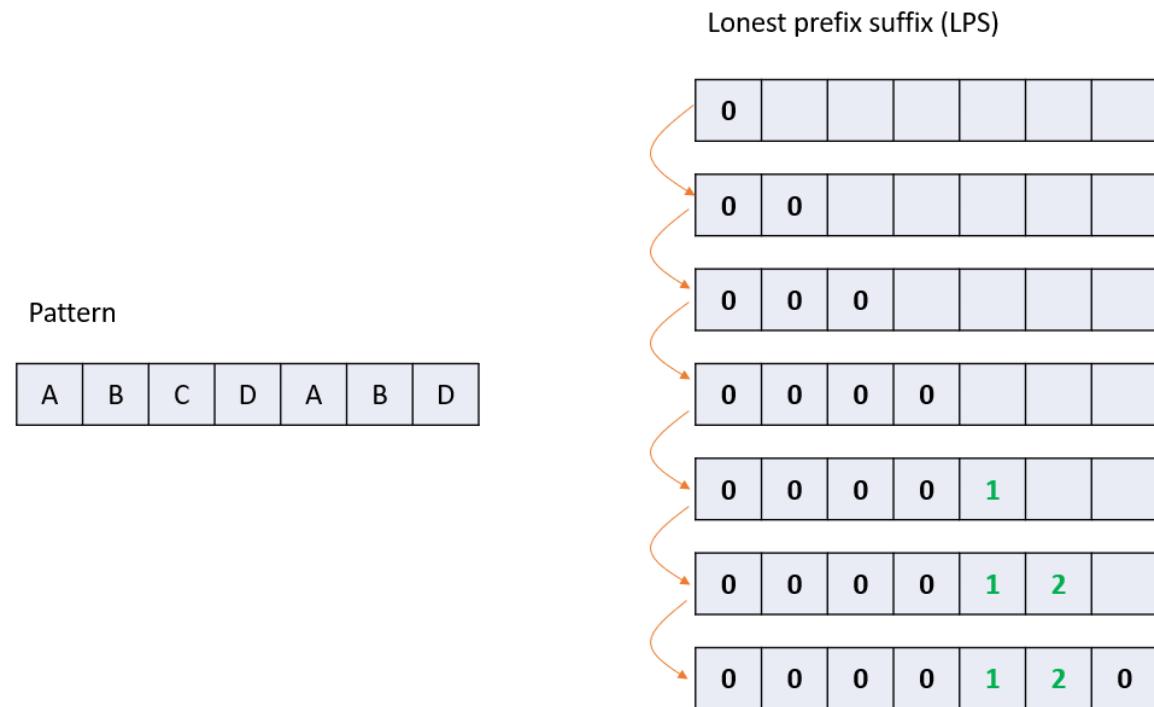
The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to $O(n)$. The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to

avoid matching the characters that we know will anyway match. Let us consider below example to understand this.

LPS Table

In the KMP algorithm, we prepare a table, called the LPS table. In the **LPS table**, we map every character of the pattern to a value. The value mapped to each character, i.e. $LPS[i]$ represents the length of the longest proper prefix that is also a suffix in the first i characters where $0 < i < \text{len(pattern)} - 1$.

- **Step 1** — Define a list with a size equal to the length of the pattern.
- **Step 2** — Define variables **i** & **j**. Set $i = 0$, $j = 1$ and $LPS[0] = 0$.
- **Step 3** — Compare the characters at `pattern[i]` and `pattern[j]`.
- **Step 4** — If both are matched then set $LPS[j] = i+1$ and increment both i & j values by one. Goto to Step 3.
- **Step 5** — If both are not matched then check the value of the variable i . If it is 0 then set $LPS[j] = 0$ and increment j value by one, if it is not 0 then set $i = LPS[i-1]$.
- **Step 6** — Repeat the above steps until all the values of $LPS[]$ are filled.



```
void computeLPSArray(String pat, int M, int lps[])
```

```

{

// length of the previous longest prefix suffix

int len = 0;

int i = 1;

lps[0] = 0; // lps[0] is always 0

// the loop calculates lps[i] for i = 1 to M-1

while (i < M) {

    if (pat.charAt(i) == pat.charAt(len)) {

        len++;

        lps[i] = len;

        i++;

    }

    else // (pat[i] != pat[len])

    {

        // This is tricky. Consider the example.

        // AAACAAAA and i = 7. The idea is similar

        // to search step.

        if (len != 0) {

            len = lps[len - 1];



            // Also, note that we do not increment

            // i here

        }

        else // if (len == 0)

        {

            lps[i] = len;

            i++;

        }

    }

}

}

```

```
    }  
}  
}
```

Search / Use of LPS Table

In KMP, when we encounter a mismatch, instead of moving one symbol forward in a target text and starting from the very beginning of a pattern string, KMP uses LPS to continue from exactly the same place where the mismatch occurred.

We use the LPS table to decide how many characters are to be skipped for comparison when a mismatch has occurred. When a mismatch occurs, check the LPS value of the previous character of the mismatched character in the pattern. If it is 0 then start comparing the first character of the pattern with the next character to the mismatched character in the text. If it is not 0 then start comparing the character which is at an index value equal to the LPS value of the previous character to the mismatched character in the pattern with the mismatched character in the Text.



```
void KMPSearch(String pat, String txt)  
{  
    int M = pat.length();  
    int N = txt.length();  
  
    // create lps[] that will hold the longest  
    // prefix suffix values for pattern  
    int lps[] = new int[M];  
    int j = 0; // index for pat[]  
  
    // Preprocess the pattern (calculate lps[]  
    // array)  
    computeLPSArray(pat, M, lps);  
  
    int i = 0; // index for txt[]  
    while ((N - i) >= (M - j)) {  
        if (pat.charAt(j) == txt.charAt(i)) {
```

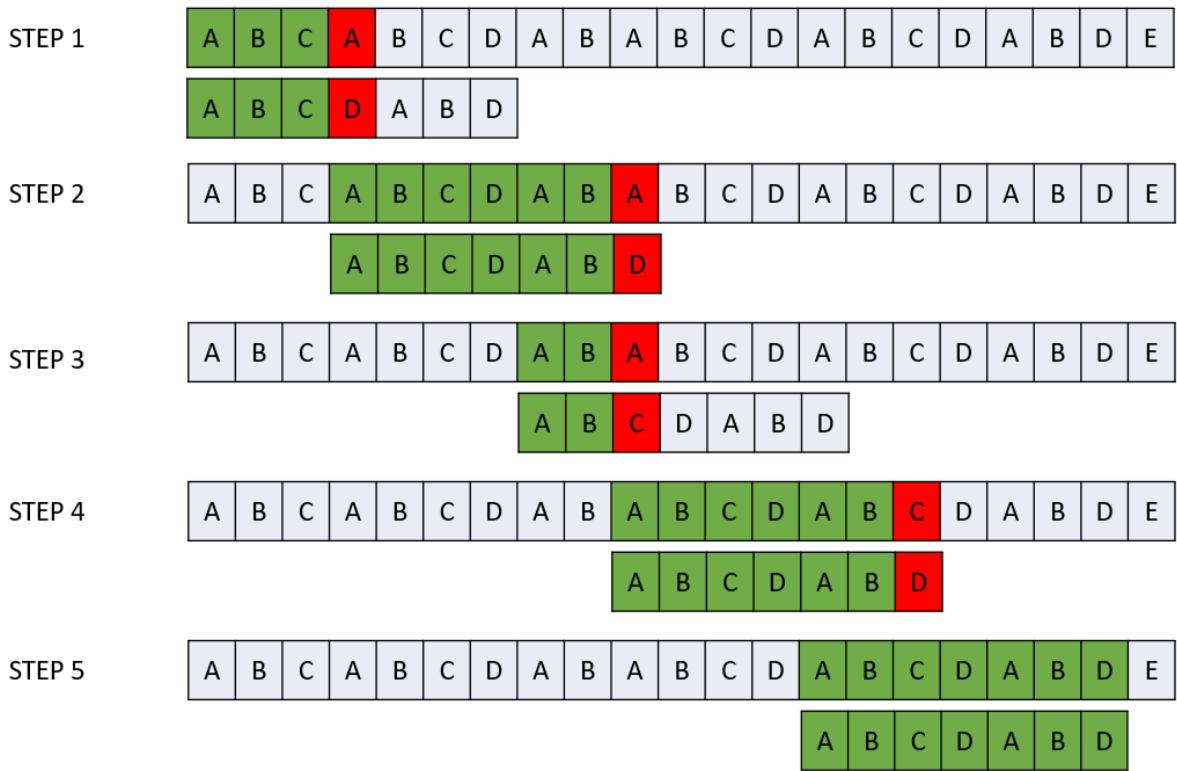
```

j++;
i++;
}

if (j == M) {
    System.out.println("Found pattern "
        + "at index " + (i - j));
    j = lps[j - 1];
}

// mismatch after j matches
else if (i < N && pat.charAt(j) != txt.charAt(i)) {
    // Do not match lps[0..lps[j-1]] characters,
    // they will match anyway
    if (j != 0)
        j = lps[j - 1];
    else
        i = i + 1;
}
}

```



Complete Solution



```
// JAVA program for implementation of KMP pattern
```

```
// searching algorithm
```

```
class KMP_String_Matching {
    void KMPSearch(String pat, String txt)
    {
        int M = pat.length();
        int N = txt.length();

        // create lps[] that will hold the longest
        // prefix suffix values for pattern
        int lps[] = new int[M];
```

```

int j = 0; // index for pat[]

// Preprocess the pattern (calculate lps[]
// array)
computeLPSArray(pat, M, lps);

int i = 0; // index for txt[]

while ((N - i) >= (M - j)) {
    if (pat.charAt(j) == txt.charAt(i)) {
        j++;
        i++;
    }
    if (j == M) {
        System.out.println("Found pattern "
            + "at index " + (i - j));
        j = lps[j - 1];
    }
}

// mismatch after j matches
else if (i < N && pat.charAt(j) != txt.charAt(i)) {
    // Do not match lps[0..lps[j-1]] characters,
    // they will match anyway
    if (j != 0)
        j = lps[j - 1];
    else
        i = i + 1;
}
}

```

```

void computeLPSArray(String pat, int M, int lps[])
{
    // length of the previous longest prefix suffix
    int len = 0;

    int i = 1;
    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    while (i < M) {
        if (pat.charAt(i) == pat.charAt(len)) {

            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {

            // This is tricky. Consider the example.
            // AAACAAAA and i = 7. The idea is similar
            // to search step.

            if (len != 0) {

                len = lps[len - 1];
            }

            // Also, note that we do not increment
            // i here
        }
        else // if (len == 0)
        {

            lps[i] = len;
        }
    }
}

```

```

        i++;
    }
}

}

// Driver program to test above function
public static void main(String args[])
{
    String txt = "ABABDABACDABABCABAB";
    String pat = "ABABCABAB";
    new KMP_String_Matching().KMPSearch(pat, txt);
}

```

/*

OUTPUT

Pattern found at index 10

*/

Time Complexity : $O(m+n)$

Space Complexity : $O(m)$

Aho Corasick Algorithm

Given an input text and an array of k words, arr[], find all occurrences of all words in the input text. Let **n** be the length of text and **m** be the total number characters in all words, i.e. $m = \text{length}(\text{arr}[0]) + \text{length}(\text{arr}[1]) + \dots + \text{length}(\text{arr}[k-1])$. Here **k** is total numbers of input words.

Example:



Input: text = "ahishers"

arr[] = {"he", "she", "hers", "his"}

Output:

Wordhis appears from 1 to 3

Wordhe appears from 4 to 5

Wordshe appears from 3 to 5

Wordhers appears from 4 to 7

If we use a linear time searching algorithm like **KMP**, then we need to one by one search all words in `text[]`. This gives us total time complexity as $O(n + \text{length}(\text{word}[0]) + O(n + \text{length}(\text{word}[1]) + O(n + \text{length}(\text{word}[2]) + \dots + O(n + \text{length}(\text{word}[k-1]))$. This time complexity can be written as $O(n*k + m)$.

Aho-Corasick Algorithm finds all words in $O(n + m + z)$ time where z is total number of occurrences of words in text. The Aho–Corasick string matching algorithm formed the basis of the original [Unix command fgrep](#).

- **Preprocessing :** Build an automaton of all words in `arr[]`. The automaton has mainly three functions:



Go To : This function simply follows edges

of Trie of all words in `arr[]`. It is

represented as 2D arrayg[][] where

we store next state for current state

and character.

Failure : This function stores all edges that are

followed when current character doesn't

have edge in Trie. It is represented as

1D arrayf[] where we store next state for

current state.

Output : Stores indexes of all words that end at

current state. It is represented as 1D

arrayo[] where we store indexes

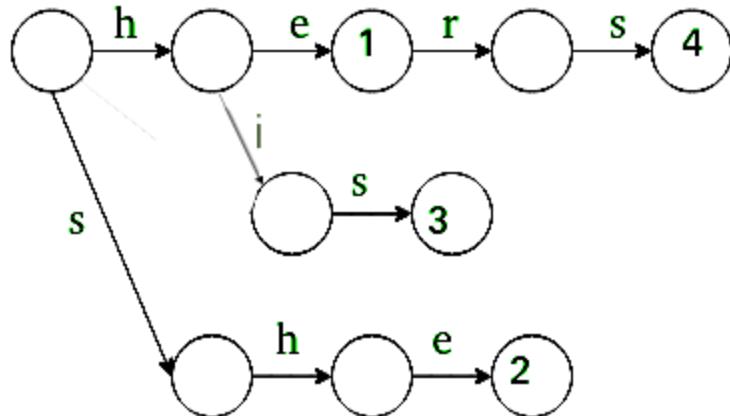
of all matching words as a bitmap for current state.

- **Matching :** Traverse the given text over built automaton to find all matching words.

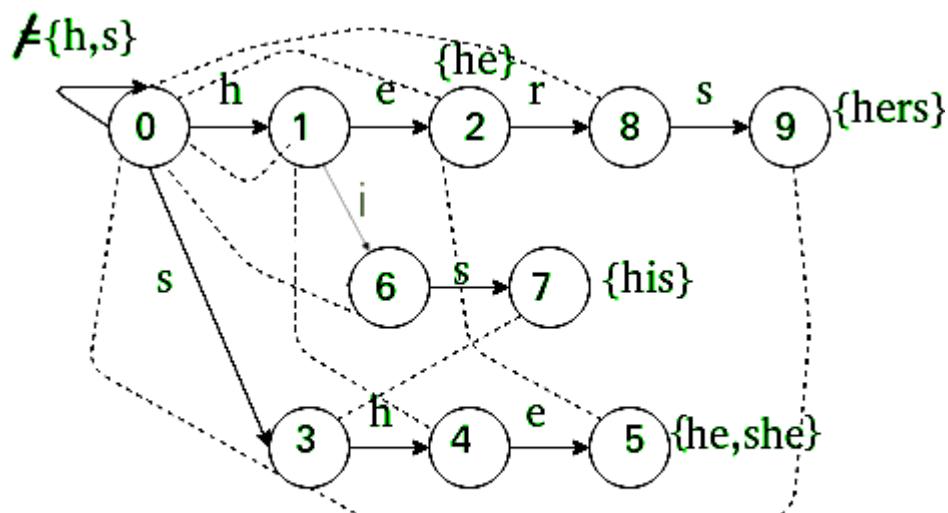
Preprocessing:

- We first Build a **Trie** (or Keyword Tree) of all words.

Trie for Arr[] = { he , she, his , hers }



- This part fills entries in goto g[][] and output o[].
- Next we extend Trie into an automaton to support linear time matching.



Dashed arrows are failed transactions.
Normal arrows are goto transactions.

- This part fills entries in failure f[] and output o[].

Go to : We build **Trie**. And for all characters which don't have an edge at root, we add an edge back to root.

Failure : For a state s, we find the longest proper suffix which is a proper prefix of some pattern. This is done using Breadth First Traversal of Trie.

Output : For a state s, indexes of all words ending at s are stored. These indexes are stored as bitwise map (by doing bitwise OR of values). This is also computing using Breadth First Traversal with Failure.

Below is the implementation of Aho-Corasick Algorithm



```
// Java program for implementation of
// Aho Corasick algorithm for String
// matching
import java.util.*;

class AhoCorasick{

    // Max number of states in the matching
    // machine. Should be equal to the sum
    // of the length of all keywords.
    static int MAXS = 500;

    // Maximum number of characters
    // in input alphabet
    static int MAXC = 26;

    // OUTPUT FUNCTION IS IMPLEMENTED USING out[]
    // Bit i in this mask is one if the word with
    // index i appears when the machine enters
    // this state.
    static int []out = new int[MAXS];
```

```

// FAILURE FUNCTION IS IMPLEMENTED USING f[]

static int []f = new int[MAXS];

// GOTO FUNCTION (OR TRIE) IS

// IMPLEMENTED USING g[][]

static int [][]g = new int[MAXS][MAXC];

// Builds the String matching machine.

// arr - array of words. The index of each keyword is important:

//           "out[state] & (1 << i)" is > 0 if we just found word[i]

//           in the text.

// Returns the number of states that the built machine has.

// States are numbered 0 up to the return value - 1, inclusive.

static int buildMatchingMachine(String arr[], int k)

{

    // Initialize all values in output function as 0.

    Arrays.fill(out, 0);

    // Initialize all values in goto function as -1.

    for(int i = 0; i < MAXS; i++)

        Arrays.fill(g[i], -1);

    // Initially, we just have the 0 state

    int states = 1;

    // Convalues for goto function, i.e., fill g[][]

    // This is same as building a Trie for arr[]

```

```

for(int i = 0; i < k; ++i)

{
    String word = arr[i];

    int currentState = 0;

    // Insert all characters of current
    // word in arr[]

    for(int j = 0; j < word.length(); ++j)

    {
        int ch = word.charAt(j) - 'a';

        // Allocate a new node (create a new state)
        // if a node for ch doesn't exist.

        if (g[currentState][ch] == -1)

            g[currentState][ch] = states++;

        currentState = g[currentState][ch];
    }

    // Add current word in output function
    out[currentState] |= (1 << i);
}

// For all characters which don't have
// an edge from root (or state 0) in Trie,
// add a goto edge to state 0 itself

for(int ch = 0; ch < MAXC; ++ch)

if (g[0][ch] == -1)

    g[0][ch] = 0;

```

```

// Now, let's build the failure function

// Initialize values in fail function

Arrays.fill(f, -1);

// Failure function is computed in
// breadth first order
// using a queue

Queue<Integer> q = new LinkedList<>();

// Iterate over every possible input
for(int ch = 0; ch < MAXC; ++ch)

{

    // All nodes of depth 1 have failure
    // function value as 0. For example,
    // in above diagram we move to 0
    // from states 1 and 3.

    if (g[0][ch] != 0)

    {

        f[g[0][ch]] = 0;

        q.add(g[0][ch]);

    }

}

// Now queue has states 1 and 3

while (!q.isEmpty())

{

```

```

// Remove the front state from queue

int state = q.peek();

q.remove();


// For the removed state, find failure

// function for all those characters

// for which goto function is

// not defined.

for(int ch = 0; ch < MAXC; ++ch)

{



// If goto function is defined for

// character 'ch' and 'state'

if (g[state][ch] != -1)

{



// Find failure state of removed state

int failure = f[state];





// Find the deepest node labeled by proper

// suffix of String from root to current

// state.

while (g[failure][ch] == -1)

    failure = f[failure];



failure = g[failure][ch];

f[g[state][ch]] = failure;



// Merge output values

```

```

        out[g[state][ch]] |= out[failure];

        // Insert the next level node
        // (of Trie) in Queue
        q.add(g[state][ch]);
    }

}

return states;
}

// Returns the next state the machine will transition to using goto
// and failure functions.

// currentState - The current state of the machine. Must be between
//                 0 and the number of states - 1, inclusive.

// nextInput - The next character that enters into the machine.

static int findNextState(int currentState, char nextInput)
{
    int answer = currentState;

    int ch = nextInput - 'a';

    // If goto is not defined, use
    // failure function

    while (g[answer][ch] == -1)
        answer = f[answer];

    return g[answer][ch];
}

```

```

// This function finds all occurrences of
// all array words in text.

static void searchWords(String arr[], int k,
                      String text)

{

    // Preprocess patterns.

    // Build machine with goto, failure
    // and output functions

    buildMatchingMachine(arr, k);

    // Initialize current state

    int currentState = 0;

    // Traverse the text through the
    // built machine to find all
    // occurrences of words in arr[]

    for(int i = 0; i < text.length(); ++i)

    {

        currentState = findNextState(currentState,
                                     text.charAt(i));

        // If match not found, move to next state

        if (out[currentState] == 0)

            continue;

        // Match found, print all matching
        // words of arr[]
        // using output function.

```

```

        for(int j = 0; j < k; ++j)
        {
            if ((out[currentState] & (1 << j)) > 0)
            {
                System.out.print("Word " + arr[j] +
                    " appears from " +
                    (i - arr[j].length() + 1) +
                    " to " + i + "\n");
            }
        }
    }

// Driver code
public static void main(String[] args)
{
    String arr[] = { "he", "she", "hers", "his" };
    String text = "ahishers";
    int k = arr.length;

    searchWords(arr, k, text);
}

/*
OUTPUT
Word his appears from 1 to 3
Word he appears from 4 to 5
Word she appears from 3 to 5

```

Word hers appears from 4 to 7

*/

Time Complexity: $O(n + l + z)$, where n is the length of the text, l is the length of keywords, and z is the number of matches.

Auxiliary Space: $O(l * q)$, where q is the length of the alphabet since that is the maximum number of children a node can have.

String Hashing

Hash Function

A Hash function is a function that maps any kind of data of arbitrary size to fixed-size values. The values returned by the function are called Hash Values or digests. There are many popular Hash Functions such as DJBX33A, MD5, and SHA-256. This post will discuss the key features, implementation, advantages and drawbacks of the Polynomial Rolling Hash Function.

The polynomial rolling hash function

Polynomial rolling hash function is a hash function that uses only multiplications and additions. The following is the function:

or simply,

Where,

- The input to the function is a string s of length n .
- p and m are positive integers.
- The choice of p and m affects the performance and the security of the hash function.
- If the string s consists only of lower-case letters, then $p = 31$ is a good choice
 - Competitive programmers often use a larger value of p . Examples include 29791, 11111, 111111.
- m shall necessarily be large prime.
- The output of the function is a hash value of the string s which ranges between 0 and $(m - 1)$ inclusive.

Below is an implementation of Polynomial Rolling Hash Function



```

class Hash {

    final int p = 31, m = 1000000007;

    int hash_value;

    Hash(String S) {

        int hash_so_far = 0;

        final char[] s = S.toCharArray();

        long p_pow = 1;

        final int n = s.length;

        for (int i = 0; i < n; i++) {

            hash_so_far = (int)((hash_so_far + (s[i] - 'a' + 1) * p_pow) % m);

            p_pow = (p_pow * p) % m;

        }

        hash_value = hash_so_far;

    }

}

```

```

class Main {

    public static void main(String[] args) {

        String s = "almabetter";

        Hash h = new Hash(s);

        System.out.println("Hash of " + s + " is " + h.hash_value);

    }

}

```

Collisions in Polynomial Rolling Hash

Since the output of the Hash function is an integer in the range $[0, m)$, there are high chances for two strings producing the same hash value.

For instance, the strings "countermand" and "furnace" produce the same hash value for $p=31$ and $m = 10^9 + 7$.

We can guarantee a collision within a very small domain. Consider a set of strings, s , consisting of only lower-case letters, such that the length of any string in s doesn't exceed 7.

We have $|s| = (26 + 26^2 + 26^3) = 835308258210^9 + 7$. Since the range of the Hash Function is $[0, m)$, one-one mapping is impossible. Hence, we can guarantee a collision by arbitrarily generating two strings whose length doesn't exceed 7.

Collision Resolution

We can note that the value of m affects the chances of collision. We have seen that the probability of collision is $1/m$. We can increase the value of m to reduce the probability of collision. But that affects the speed of the algorithm. Larger the value of m , the slower the algorithm. Also, some languages (C, C++, Java) have a limit on the size of the integer. Hence, we can't increase the value of m to a very large value.

Then how can we minimise the chances of a collision?

Note that the hash of a string depends on two parameters : p and m .We have seen that the strings "countermand" and "furnace" produce the same hash value for $p=31$ and $m = 10^9 + 7$. But for $p = 37$ and $m = 10^9 + 9$, they produce different hashes.

Observation



If two strings produce the same hash values for a pair (p_1, m_1) ,
they will produce different hashes for a different pair, (p_2, m_2) .

Strategy

We cannot, however, nullify the chances of collision because there are infinitely many strings. But, surely, we can reduce the probability of two strings colliding.

We can reduce the probability of collision by generating a pair of hashes for a given string. The first hash is generated using $p=31$ and $m = 10^9 + 7$, while the second hash is generated using $p=37$ and $m=10^9 + 9$.

Why will this work

We are generating two hashes using two different modulo values, m_1 and m_2 . The probability of a collision is now $1/m_1 * 1/m_2$. Since both m_1 and m_2 are greater than 10^9 , the probability that a collision occurs is now less than 10^{-18} which is so much better than the original probability of collision, 10^{-9} .

Below is the implementation for the same



```
class Hash {  
    final int p1 = 31, m1 = 1000000007;  
    final int p2 = 37, m2 = 1000000009;
```

```

int hash_value1, hash_value2;

Hash(String s) {
    compute_hash1(s);
    compute_hash2(s);
}

void compute_hash1(String s) {
    int hash_so_far = 0;
    final char[] s_array = s.toCharArray();
    long p_pow = 1;
    final int n = s_array.length;
    for (int i = 0; i < n; i++) {
        hash_so_far = (int)((hash_so_far + (s_array[i] - 'a' + 1) * p_pow) % m1);
        p_pow = (p_pow * p1) % m1;
    }
    hash_value1 = hash_so_far;
}

void compute_hash2(String s) {
    int hash_so_far = 0;
    final char[] s_array = s.toCharArray();
    long p_pow = 1;
    final int n = s_array.length;
    for (int i = 0; i < n; i++) {
        hash_so_far = (int)((hash_so_far + (s_array[i] - 'a' + 1) * p_pow) % m2);
        p_pow = (p_pow * p2) % m2;
    }
    hash_value2 = hash_so_far;
}

```

```

class Main {
    public static void main(String[] args) {
        String s = "almabetter";
        Hash h = new Hash(s);
        System.out.println("Hash values of " + s + " are: " + h.hash_value1 + ", " +
h.hash_value2);
    }
}

```

Output



Hash values of geeksforgeeks are: (609871790, 642799661)

Features of Polynomial rolling hash function

Calculation of Hashes of any substring of a given string in O(1).

Note that computing the hash of the string S will also compute the hashes of all of the prefixes. We just have to store the hash values of the prefixes while computing. Say $\text{hash}[i]$ denotes the hash of the prefix $S[0...i]$, we have

$$\text{hash}[i...j].p^i = \text{hash}[0...j] - \text{hash}[0...(i-1)]$$

This allows us to quickly compute the hash of the substring $s[i...j]$ in O(1) provided we have powers of p ready.

The behaviour of the hash when a character is changed

Recall that the hash of string s is given by

$$\text{hash}(s) = \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m$$

Say, we change a character ch_1 at some index i to some other character ch_2 . How will the hash change?

If some hash_{old} denotes the hash value before changing and hash_{new} is the hash value after changing, then the relation between them is given by

```
hash_new = hash_old - p^i * (ch1) + p^i * (ch2)
```

Therefore, queries can be performed very quickly instead of recalculating the hash from beginning, provided we have the powers of pready.

Interview Questions

- **How to compare two Strings in java program?**

Java String implements Comparable interface and it has two variants of compareTo() methods. compareTo(String anotherString) method compares the String object with the String argument passed lexicographically. If String object precedes the argument passed, it returns negative integer and if String object follows the argument String passed, it returns a positive integer. It returns zero when both the String have the same value, in this case equals(String str) method will also return true. compareToIgnoreCase(String str): This method is similar to the first one, except that it ignores the case. It uses String CASE_INSENSITIVE_ORDER Comparator for case insensitive comparison. If the value is zero then equalsIgnoreCase(String str) will also return true.

- **Why String is immutable or final in Java**

There are several benefits of String because it's immutable and final.

- String Pool is possible because String is immutable in java.
- It increases security because any hacker can't change its value and it's used for storing sensitive information such as database username, password etc.
- Since String is immutable, it's safe to use in multi-threading and we don't need any synchronization.
- Strings are used in java classloader and immutability provides security that correct class is getting loaded by Classloader.

Agenda

- Strings in Java
- Java String Methods
- Problem Solving with String in Java

Introduction

In Java, a string is a sequence of characters. For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'.

We use **double quotes** to represent a string in Java. For example,



```
// create a string  
String type = "Java programming";
```

Here, we have created a string variable named type. The variable is initialized with the string Java Programming.

Create a String in Java



```
class Main {  
  
    public static void main(String[] args) {  
  
        // create strings  
        String first = "Java";  
        String second = "Python";  
        String third = "JavaScript";  
  
        // print strings  
        System.out.println(first); // print Java  
        System.out.println(second); // print Python  
        System.out.println(third); // print JavaScript  
    }  
}
```

```
}
```

In the above example, we have created three strings named first, second, and third. Here, we are directly creating strings like primitive types.

However, there is another way of creating Java strings (using the new keyword). We will learn about that later.

Note: Strings in Java are not primitive types (like int, char, etc). Instead, all strings are objects of a predefined class named String.

And, all string variables are instances of the String class.

Escape character in Java Strings

The escape character is used to escape some of the characters present inside a string.

Suppose we need to include double quotes inside a string.



```
// include double quote
```

```
String example = "This is the "String" class";
```

Since strings are represented by **double quotes**, the compiler will treat "This is the " as the string. Hence, the above code will cause an error.

To solve this issue, we use the escape character \ in Java. For example,



```
// use the escape character
```

```
String example = "This is the \"String\" class.";
```

Now escape characters tell the compiler to escape **double quotes** and read the whole text.

Immutability

In Java, strings are **immutable**. This means, once we create a string, we cannot change that string.

To understand it more deeply, consider an example:



```
// create a string
```

```
String example = "Hello! ";
```

Here, we have created a string variable named example. The variable holds the string "Hello!".

Now suppose we want to change the string.



```
// add another string "World"  
// to the previous string example  
  
example = example.concat(" World");
```

Here, we are using the concat() method to add another string World to the previous string.

It looks like we are able to change the value of the previous string. However, this is not true.

Let's see what has happened here,

1. JVM takes the first string "Hello! "
2. creates a new string by adding to the first string "World"
3. assign the new string "Hello! World" to the variable example
4. the first string "Hello! " remains unchanged

The new keyword

So far we have created strings like primitive types in Java.

Since strings in Java are objects, we can create strings using the new keyword as well. For example,



```
// create a string using the new keyword  
  
String name = new String("Java String");
```

In the above example, we have created a string name using the new keyword.

Here, when we create a string object, the String() constructor is invoked.

Note: The String class provides various other constructors to create strings. To learn more, visit [Java String \(official Java documentation\)](#).

Create String using literals vs new keyword

Now that we know how strings are created using string literals and the new keyword, let's see what is the major difference between them.

In Java, the JVM maintains a **string pool** to store all of its strings inside the memory. The string pool helps in reusing the strings.

1. While creating strings using string literals,



```
String example = "Java";
```

Here, we are directly providing the value of the string (Java). Hence, the compiler first checks the string pool to see if the string already exists.

- **If the string already exists**, the new string is not created. Instead, the new reference, points to the already existed string (Java).

example

- **If the string doesn't exist**, the new string (Java is created).
2. While creating strings using the new keyword,



```
String example = new String("Java");
```

Here, the value of the string is not directly provided. Hence, a new "Java" string is created even though "Java" is already present inside the memory pool.

Java String Methods

toCharArray():

This method converts the string into a character array i.e first it will calculate the length of the given Java String including spaces and then create an array of char type with the same content. For example:



```
StringToCharArrayExample{  
public static void main(String args[]){  
String s1="Hello World";  
char[] ch=s1.toCharArray();  
for(int i=0;i<ch.length;i++){  
System.out.print(ch[i]);  
}}}
```

The above code will return “Hello World”.

length():

The Java String length() method tells the length of the string. It returns the count of the total number of characters present in the String. For example:



```
public class Example{  
    public static void main(String args[]){  
        String s1="hello";  
        String s2="whatsup";  
        System.out.println("string length is: "+s1.length());  
        System.out.println("string length is: "+s2.length());  
    }  
}
```

Here, String length() function will return the length 5 for s1 and 7 for s2 respectively.

compareTo():

The Java String compareTo() method compares the given string with current string. It is a method of '*Comparable*' interface which is implemented by String class. It either returns a positive number, negative number or 0. For example:



```
public class CompareToExample{  
    public static void main(String args[]){  
        String s1="hello";  
        String s2="hello";  
        String s3="hemlo";  
        String s4="flag";  
        System.out.println(s1.compareTo(s2)); // 0 because both are equal  
        System.out.println(s1.compareTo(s3)); // -1 because "l" is only one time lower than "m"  
        System.out.println(s1.compareTo(s4)); // 2 because "h" is 2 times greater than "f"  
    }  
}
```

This program shows the comparison between the various string. It is noticed that

if s1 > s2, it returns a positive number

if s1 < s2, it returns a negative number

if s1 == s2, it returns 0

concat() :

The Java String concat() method combines a specific string at the end of another string and ultimately returns a combined string. It is like appending another string. For example:



```
public class ConcatExample{  
    public static void main(String args[]){  
        String s1="hello";  
        s1=s1.concat("how are you");  
        System.out.println(s1);  
    }  
}
```

The above code returns “**hellohow are you**”.

trim() :

The Java string trim() method removes the leading and trailing spaces. It checks the Unicode value of space character ('\u0020') before and after the string. If it exists, then removes the spaces and return the omitted string. For example:



```
public class StringTrimExample{  
    public static void main(String args[]){  
        String s1=" hello ";  
        System.out.println(s1+"how are you"); // without trim()  
        System.out.println(s1.trim()+"how are you"); // with trim()  
    }  
}
```

In the above code, the first print statement will print “hello how are you” while the second statement will print “**hellohow are you**” using the trim() function.

toLowerCase() :

The java string toLowerCase() method converts all the characters of the String to lower case. For example:



```
public class StringLowerExample{
```

```
public static void main(String args[]){
    String s1="HELLO HOW Are You?";
    String s1lower=s1.toLowerCase();
    System.out.println(s1lower);
}
```

The above code will return “**hello how are you**”.

toUpperCase() :

The Java String `toUpperCase()` method converts all the characters of the String to upper case. For example:



```
public class StringUpperExample{
    public static void main(String args[]){
        String s1="hello how are you";
        String s1upper=s1.toUpperCase();
        System.out.println(s1upper);
    }
}
```

The above code will return “**HELLO HOW ARE YOU**”.

valueOf():

This method converts different types of values into a string. Using this method, you can convert int to string, long to string, Boolean to string, character to string, float to a string, double to a string, object to string and char array to string. The signature or syntax of string `valueOf()` method is given below:

```
public static String valueOf(boolean b)
public static String valueOf(char c)
public static String valueOf(char[] c)
public static String valueOf(int i)
public static String valueOf(long l)
public static String valueOf(float f)
public static String valueOf(double d)
public static String valueOf(Object o)
```

Let's understand this with a programmatic example:



```
public class StringValueOfExample{  
    public static void main(String args[]){  
        int value=20;  
        String s1=String.valueOf(value);  
        System.out.println(s1+17); //concatenating string with 10  
    }  
}
```

In the above code, it concatenates the Java String and gives the output — **2017**.

replace():

The Java String replace() method returns a string, replacing all the old characters or CharSequence to new characters. There are 2 ways to replace methods in a Java String.



```
public class ReplaceExample1{  
    public static void main(String args[]){  
        String s1="hello how are you";  
        String replaceString=s1.replace('h','t');  
        System.out.println(replaceString);  
    }  
}
```

In the above code, it will replace all the occurrences of '**h**' to '**t**'. Output to the above code will be '**tello tow are you**'. Let's see another type of using replace method in java string:

replace(CharSequence target, CharSequence replacement) method :



```
public class ReplaceExample2{  
    public static void main(String args[]){  
        String s1="Hello World!";  
        String replaceString=s1.replace("Hello","Hey");  
        System.out.println(replaceString);  
    }  
}
```

In the above code, it will replace all occurrences of "Hello" to "Hey". Therefore, the output would be " **Hey World!**".

contains() :

The Java string contains() method searches the sequence of characters in the string. If the sequences of characters are found, then it returns true otherwise returns false. For example:



```
class ContainsExample{  
    public static void main(String args[]){  
        String name=" hello how are you doing?";  
        System.out.println(name.contains("how are you")); // returns true  
        System.out.println(name.contains("hello")); // returns true  
        System.out.println(name.contains("fine")); // returns false  
    }  
}
```

In the above code, the first two statements will return true as it matches the String whereas the second print statement will return false because the characters are not present in the string.

equals() :

The Java String equals() method compares the two given strings on the basis of the content of the string i.e Java String representation. If all the characters are matched, it returns true else it will return false. For example:



```
public class EqualsExample{  
    public static void main(String args[]){  
        String s1="hello";  
        String s2="hello";  
        String s3="hi";  
        System.out.println(s1.equalsIgnoreCase(s2)); // returns true  
        System.out.println(s1.equalsIgnoreCase(s3)); // returns false  
    }  
}
```

equalsIgnoreCase():

This method compares two string on the basis of content but it does not check the case like equals() method. In this method, if the characters match, it returns true else false. For example:



```
public class EqualsIgnoreCaseExample{  
    public static void main(String args[]){  
        String s1="hello";  
        String s2="HELLO";  
        String s3="hi";  
        System.out.println(s1.equalsIgnoreCase(s2)); // returns true  
        System.out.println(s1.equalsIgnoreCase(s3)); // returns false  
    }  
}
```

In the above code, the first statement will return true because the content is the same irrespective of the case. Then, in the second print statement will return false as the content doesn't match in the respective strings.

getBytes() :

The Java string getBytes() method returns the sequence of bytes or you can say the byte array of the string. For example:



```
public class StringGetBytesExample {  
    public static void main(String args[]){  
        String s1="ABC";  
        byte[] b=s1.getBytes();  
        for(int i=0;i<b.length;i++){  
            System.out.println(b[i]);  
        }  
    }  
}
```

In the above code, it will return the value **65,66,67**.

isEmpty() :

This method checks whether the String is empty or not. If the length of the String is 0, it returns true else false. For example:



```
public class IsEmptyExample{  
    public static void main(String args[]) {  
        String s1="";  
        String s2="hello";  
        System.out.println(s1.isEmpty()); // returns true  
        System.out.println(s2.isEmpty()); // returns false  
    }  
}
```

In the above code, the first print statement will **return true** as it does not contain anything while the second print statement will **return false**.

endsWith() :

The Java String endsWith() method checks if this string ends with the given suffix. If it returns with the given suffix, it will return true else returns false. For example:



```
public class EndsWithExample{  
    public static void main(String args[]) {  
        String s1="hello how are you";  
        System.out.println(s1.endsWith("u")); // returns true  
        System.out.println(s1.endsWith("you")); // returns true  
        System.out.println(s1.endsWith("how")); // returns false  
    }  
}
```

Things to remember when using Strings

Use Equals methods for comparing

String class overrides equals method and provides content equality, which is based on characters, case and order. So if you want to compare two String objects, to check whether they are same or not, always use the *equals()* method instead of the equality operator(==).

“+” is overloaded for String concatenation

Java doesn't support operator overloading but String is special and + operator can be used to concatenate two Strings. It can even be used to convert int, char, long or double to convert into String by simply concatenating with empty string “”.

Don't store sensitive data in String

String poses a security threat if used for storing sensitive data like passwords, SSN or any other sensitive information. Since String is immutable in Java there is no way you can erase contents of String and since they are kept in the String pool (in the case of String literal) they stay longer on the Java heap, which exposes the risk of being seen by anyone who has access to Java memory, like reading from the memory dump. Instead, `char[]` should be used to store passwords or sensitive information.

Problem Solving with Strings in Java

Pattern Matching

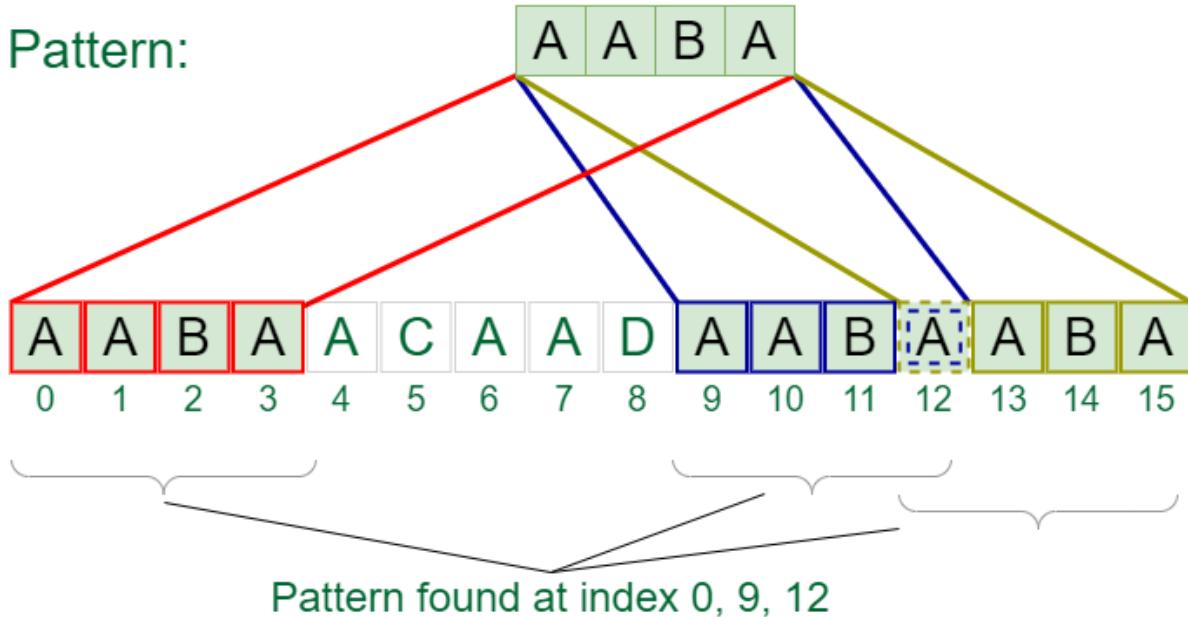
The string matching (or pattern matching) problem is defined as follows.

Given a text T and a pattern P, determine all the occurrences of the pattern in the text.

We assume that the text T is an array of length n and the pattern P is an array of m elements. We say that **pattern P occurs with shifts s in text T** if

- $0 \leq s \leq n-m$, and
- $T[s+1 : s+m] = P[1 : m]$

Text: A A B A A C A A D A A A B A A B A



A shift is valid only if pattern P occurs with shift s in text T. We can say that the string matching problem is the problem of finding all valid shifts with which the pattern P occurs in a text T.

The Naive Algorithm

Let's see now a very simple (and inefficient) algorithm for string matching, the so called **naive string-matching algorithm**.

We use a for loop from 0 to $n-m$ (respectively, the length of the text T and the length of the pattern P) to checks if P is equal to the substring of S taken from $s+1$ and $s+m$.

Pseudocode



NaiveStringMatcher(T, P):

$n = \text{length}(T)$

$m = \text{length}(P)$

 FOR $s = 0$ to $n-m$ do:

 IF $P[1 : m] == T[s+1 : s+m]$:

 print "Pattern occurs with shift " s

The pattern slides over the text and if the pattern is found in the text starting from s (the if statement is true), we print out the shift s .

This is an inefficient algorithm for long text. Why? Because for every possible $n-m+1$ shift we have to compare all the character of P with the m character of $T[s+1 : s+m]$.

The **worst case running time** is $\Theta((n-m+1)*m)$.

If m is equal to $n/2$, the running time is $\Theta(n^2)$.

Java Implementation



```
// Java program for Naive Pattern Searching
```

```
public class NaiveSearch {
```

```
    static void search(String pat, String txt)
```

```
{
```

```
    int l1 = pat.length();
```

```
    int l2 = txt.length();
```

```
    int i = 0, j = l2 - 1;
```

```

        for (i = 0, j = l2 - 1; j < l1;) {

            if (txt.equals(pat.substring(i, j + 1))) {
                System.out.println("Pattern found at index "
                    + i);
            }

            i++;
            j++;
        }

    }

// Driver's code
public static void main(String args[])
{
    String pat = "AABAACAAADAABAAABAA";
    String txt = "AABA";

    // Function call
    search(pat, txt);
}

```

Interview Questions

- **What is String in Java? String is a data type?**

String is a Class in java and defined in `java.lang` package. It's not a primitive data type like `int` and `long`. String class represents character Strings. String is used in almost all the Java applications and there are some interesting facts we should know about String. String is immutable and final in Java and JVM uses String Pool to store all the String objects. Some other interesting things about String is the way we can instantiate a String object using double quotes and overloading of "+" operator for concatenation.

- **How can we make String upper case or lower case?**

We can use String class toUpperCase and toLowerCase methods to get the String in all upper case or lower case. These methods have a variant that accepts Locale argument and use that locale rules to convert String to upper or lower case.

- **Write a method that will remove given character from the String?**

We can use replaceAll method to replace all the occurrence of a String with another String. The important point to note is that it accepts String as argument, so we will use Character class to create String and use it to replace all the characters with empty String.



```java

```
private static String removeChar(String str, char c) {
 if (str == null)
 return null;
 return str.replaceAll(Character.toString(c), "");
}
```

...

### Additional Resources

- **StringBuffer**

<https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuffer.html>

<https://www.scaler.com/topics/java/stringbuffer-in-java/>

Agenda :

- **Wrapper Classes**
- **Exceptions**
- **RegEx**
- **Java Threads**

## 1. Wrapper Classes

The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive. Let us see the different scenarios, where we need to use the wrapper classes.

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

### Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the `valueOf()` method of wrapper classes to convert the primitive into objects.



```
//Java program to convert primitive into objects
```

```
//Autoboxing example of int to Integer
```

```
class WrapperExample
```

```
{
```

```
 public static void main(String args[])
```

```
{
 //Converting int into Integer

 int a=20;

 Integer i=Integer.valueOf(a);//converting int into Integer explicitly

 Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

 System.out.println(a+" "+i+" "+j);
}
}
```

**Output :**



20 20 20

**Unboxing**

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.



```
//Java program to convert object into primitives

//Unboxing example of Integer to int

class WrapperExample
{

 public static void main(String args[])
 {

 //Converting Integer to int

 Integer a=new Integer(5);

 int i=a.intValue();//converting Integer to int explicitly

 int j=a;//unboxing, now compiler will write a.intValue() internally
```

```
 System.out.println(a+" "+i+" "+j);
 }
}
```

**Output :**



5 5 5

### All Wrapper Classes Example

Consider the following Java program to see how autoboxing and unboxing works for all the primitive data types in Java :



```
//Java Program to convert all primitives into its corresponding
//wrapper objects and vice-versa
```

```
class WrapperExample
{
 public static void main(String args[])
 {
 byte b=10;
 short s=20;
 int i=30;
 long l=40;
 float f=50.0F;
 double d=60.0D;
 char c='a';
 boolean b2=true;

 //Autoboxing: Converting primitives into objects
 Byte byteobj=b;
```

```
Short shortobj=s;
Integer intobj=i;
Long longobj=l;
Float floatobj=f;
Double doubleobj=d;
Character charobj=c;
Boolean boolobj=b2;

//Printing objects
System.out.println("---Printing object values---");
System.out.println("Byte object: "+byteobj);
System.out.println("Short object: "+shortobj);
System.out.println("Integer object: "+intobj);
System.out.println("Long object: "+longobj);
System.out.println("Float object: "+floatobj);
System.out.println("Double object: "+doubleobj);
System.out.println("Character object: "+charobj);
System.out.println("Boolean object: "+boolobj);

//Unboxing: Converting Objects to Primitives
byte bytevalue=byteobj;
short shortvalue=shortobj;
int intvalue=intobj;
long longvalue=longobj;
float floatvalue=floatobj;
double doublevalue=doubleobj;
char charvalue=charobj;
boolean boolvalue=boolobj;
```

```
//Printing primitives
System.out.println("");
System.out.println("byte value: "+bytevalue);
System.out.println("short value: "+shortvalue);
System.out.println("int value: "+intval);
System.out.println("long value: "+longvalue);
System.out.println("float value: "+floatvalue);
System.out.println("double value: "+doublevalue);
System.out.println("char value: "+charvalue);
System.out.println("boolean value: "+boolvalue);
}
}
```

**Output :**



---Printing object values---

Byte object: 10

Short object: 20

Integer object: 30

Long object: 40

Float object: 50.0

Double object: 60.0

Character object: a

Boolean object: true

---Printing primitive values---

byte value: 10

short value: 20

int value: 30

long value: 40

```
float value: 50.0
double value: 60.0
char value: a
boolean value: true
```

## 2. Exceptions

When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an **exception** (throw an error).

### Java try and catch

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:



```
try {
 // Block of code to try
}
catch(Exception e) {
 // Block of code to handle errors
}
```

### Example :



```
public class Main {
 public static void main(String[] args) {
 try {
 int[] myNumbers = {1, 2, 3};
 System.out.println(myNumbers[10]);
 } catch (Exception e) {
```

```
 System.out.println("Something went wrong.");
 }
}
}
```

**Output :**



Something went wrong.

**Finally**

The finally statement lets you execute code, after try...catch, regardless of the result.

**Example :**



```
public class Main {
 public static void main(String[] args) {
 try {
 int[] myNumbers = {1, 2, 3};
 System.out.println(myNumbers[10]);
 } catch (Exception e) {
 System.out.println("Something went wrong.");
 } finally {
 System.out.println("The 'try catch' is finished.");
 }
 }
}
```

**Output :**



Something went wrong.

The 'try catch' is finished.

## The throw keyword

The throw statement allows you to create a custom error.

The throw statement is used together with an **exception type**. There are many exception types available in Java

: ArithmeticException, FileNotFoundException, ArrayIndexOutOfBoundsException, SecurityException, etc.

**Example :** Throw an exception if **age** is below 18 (print "Access denied"). If age is 18 or older, print "Access granted".



```
public class Main {
 static void checkAge(int age) {
 if (age < 18) {
 throw new ArithmeticException("Access denied - You must be at least 18 years old.");
 }
 else {
 System.out.println("Access granted - You are old enough!");
 }
 }
}
```

```
public static void main(String[] args) {
 checkAge(15); // Set age to 15 (which is below 18...)
}
```

**Output :**



Exception in thread "main" java.lang.ArithmaticException: Access denied - You must be at least 18 years old.

```
at Main.checkAge(Main.java:4)
at Main.main(Main.java:12)
```

## 3. Java RegEx

A regular expression is a sequence of characters that forms a search pattern. When you search for data in a text, you can use this search pattern to describe what you are searching for.

A regular expression can be a single character, or a more complicated pattern.

Regular expressions can be used to perform all types of **text search** and **text replace** operations.

Java does not have a built-in Regular Expression class, but we can import the `java.util.regex` package to work with regular expressions. The package includes the following classes:

- Pattern Class - Defines a pattern (to be used in a search)
- Matcher Class - Used to search for the pattern
- PatternSyntaxException Class - Indicates syntax error in a regular expression pattern

#### **Example :**

Find out if there are any occurrences of the word "almabetter" in a sentence, where the case of the letters does not matter :



```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class AlmaBetter {
 public static void main(String[] args) {
 Pattern pattern = Pattern.compile("almabetter", Pattern.CASE_INSENSITIVE);
 Matcher matcher = pattern.matcher("Visit AlmaBetter!");
 boolean matchFound = matcher.find();
 if(matchFound) {
 System.out.println("Match found");
 } else {
 System.out.println("Match not found");
 }
 }
}
```

#### **Output :**



## Match Found

The first parameter of the Pattern.compile() method is the pattern. It describes what is being searched for.

## Flags

Flags in the compile() method change how the search is performed. Here are a few of them:

- Pattern.CASE\_INSENSITIVE - The case of letters will be ignored when performing a search.
- Pattern.LITERAL - Special characters in the pattern will not have any special meaning and will be treated as ordinary characters when performing a search.
- Pattern.UNICODE\_CASE - Use it together with the CASE\_INSENSITIVE flag to also ignore the case of letters outside of the English alphabet

## Example of Java Regular Expressions

There are three ways to write the regex example in Java :



```
import java.util.regex.*;
class AlmaBetter
{
 public static void main(String args[])
 {
 //1st way
 Pattern p = Pattern.compile(".s");//. represents single character
 Matcher m = p.matcher("as");
 boolean b1 = m.matches();

 //2nd way
 boolean b2=Pattern.compile(".s").matcher("as").matches();

 //3rd way
```

```

 boolean b3 = Pattern.matches(".s", "as");

 System.out.println(b1+" "+b2+" "+b3);
 }

}

```

**Output :**



true true true

### Regex Character Classes

The left-hand column specifies the regular expression constructs, while the right-hand column describes the conditions under which each construct will match.

| Expression | Description                                              |
|------------|----------------------------------------------------------|
| [abc]      | Find one character from the options between the brackets |
| [^abc]     | Find one character NOT between the brackets              |
| [0-9]      | Find one character from the range 0 to 9                 |

**For Example :**



```

import java.util.regex.*;

class AlmaBetter
{
 public static void main(String args[])
 {
 System.out.println(Pattern.matches("[amn]", "abcd")); //false (not a or m or n)

 System.out.println(Pattern.matches("[amn]", "a")); //true (among a or m or n)

 System.out.println(Pattern.matches("[amn]", "amn")); //false (anyone of a or m or n is
allowed only once)

 System.out.println(Pattern.matches("[amn][mno]good", "amgood")); //true (a is
followed by m, which is followed by good)
 }
}

```

```
 }
}
```

**Output :**



```
false
true
false
true
```

### Regex Quantifiers

The quantifiers specify the number of occurrences of a character.

| Regex  | Description                                     |
|--------|-------------------------------------------------|
| X?     | X occurs once or not at all                     |
| X+     | X occurs once or more times                     |
| X*     | X occurs zero or more times                     |
| X{n}   | X occurs n times only                           |
| X{n,}  | X occurs n or more times                        |
| X{y,z} | X occurs at least y times but less than z times |

**For Example :**



```
import java.util.regex.*;

class AlmaBetter
{
 public static void main(String args[])
 {
 System.out.println("? quantifier");
 System.out.println(Pattern.matches("[amn]?", "a"));
 }
}
```

```

//true (a comes exactly once)

 System.out.println(Pattern.matches("[amn]?", "aaa"));

//false (a comes more than one time)

 System.out.println(Pattern.matches("[amn]?", "amn"));

//false (a, m and n each occur one time)

 System.out.println(Pattern.matches("[amn]?", "n"));

//true (n comes exactly once)

System.out.println("+ quantifier");

 System.out.println(Pattern.matches("[amn]+", "a"));

//true (a or m or n come once or more times)

 System.out.println(Pattern.matches("[amn]+", "aaa"));

//true (a comes more than one time)

 System.out.println(Pattern.matches("[amn]+", "aaaammmmmnn"));

//true (a or m or n comes more than once)

 System.out.println(Pattern.matches("[amn]+", "manmanamnamn"));

//true (note that the order doesn't matter)

 System.out.println(Pattern.matches("[amn]+", "aazzta"));

//false (z and t are not matching pattern)

System.out.println("* quantifier");

 System.out.println(Pattern.matches("[amn]*", "ammmnna"));

//true (a or m or n may come zero or more times)

 System.out.println(Pattern.matches("[amn]*", "aaaaannnnn"));

//true (m occurs 0 times)

 System.out.println(Pattern.matches("[amn]*", "amnamnxyz"));

//false (x,y and z are not matching pattern)

 System.out.println(Pattern.matches("[amn]*", ""));

```

```
//true (empty string has a, m or n zero or more times)
```

```
 }
}
```

**Output :**



```
? quantifier
```

```
true
```

```
false
```

```
false
```

```
true
```

```
+ quantifier
```

```
true
```

```
true
```

```
true
```

```
true
```

```
false
```

```
* quantifier
```

```
true
```

```
true
```

```
false
```

```
true
```

## Metacharacters

Metacharacters are characters with a special meaning:

| Metacharacter | Description                                              |
|---------------|----------------------------------------------------------|
| .             | Find just one instance of any character                  |
| ^             | Finds a match at the beginning of a string as in: ^Hello |
| \$            | Finds a match at the end of the string as in: World\$    |

| Metacharacter | Description                                                                                          |
|---------------|------------------------------------------------------------------------------------------------------|
| \d            | Find a digit                                                                                         |
| \s            | Find a whitespace character                                                                          |
| \b            | Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b |
| \uxxxx        | Find the Unicode character specified by the hexadecimal number xxxx                                  |

Metacharacter Description

- . Find just one instance of any character
- ^ Finds a match as the beginning of a string as in: ^Hello
- \$ Finds a match at the end of the string as in: World\$
- \d Find a digit
- \s Find a whitespace character
- \b Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b
- \uxxxx Find the Unicode character specified by the hexadecimal number xxxx

### Examples using Metacharacters for matching patterns

#### 1. Digit & Non Digit related Metacharacters: (\d, \D)

- **d** metacharacter represents a digit from 0 to 9. So when we compare “**d**” within the range, it then returns **true**. Else return false.
- **D** metacharacter represents a non-digit that **accepts anything except numbers**. So when we compare “**D**” with **any number**, it **returns false**. Else True.



```
// Java program to demonstrate the
// Digit & Non Digit related Metacharacters
```

```
import java.io.*;
import java.util.regex.*;
class AlmaBetter {
 public static void main(String[] args)
 {
```

```

// \d represents a digit
// represents a number so return true
System.out.println(Pattern.matches("\d", "2")); //true

// Comparing a number with character so return false
System.out.println(Pattern.matches("\d", "a")); //false

// \D represents non digits
// Comparing a non digit with character so return
// true
System.out.println(Pattern.matches("\D", "a")); //true

// comparing a non digit with a digit so return
// false
System.out.println(Pattern.matches("\D", "2")); //false

}

}

```

**Output :**



true

false

true

false

## 2. Whitespace and Non-Whitespace Metacharacters: (\s, \S)

- **s** represents whitespace characters like space, tab space, newline, etc. So when we compare “**s**” with **whitespace characters, it returns true**. Else false.
- **S** represents a Non-whitespace character that \*\*\*\*accepts everything except \*\*\*\*whitespace, So when we compare “**S**” with **whitespace characters, it returns false**. Else true.



```
// Java program to demonstrate the
// Whitespace and Non-Whitespace Metacharacters

import java.io.*;
import java.util.regex.*;
class AlmaBetter{
 public static void main(String[] args)
 {
 // comparing any whitespace character with a white
 // space so return true else false
 System.out.println(Pattern.matches("\s", " ")); //true
 System.out.println(Pattern.matches("\s", "2")); //false

 // comparing any non whitespace character with a non
 // white space character so return true else false
 System.out.println(Pattern.matches("\S", "2")); //true
 System.out.println(Pattern.matches("\S", " ")); //false
 }
}
```

**Output :**



```
true
false
true
false
```

**Note :** The first example in this topic makes use of the method **find()** while later on we use the method **matches()**. It is important to note that both these methods have distinct use cases.

The main difference is that the **matches()** method tries to match the entire region of the given input i.e. if you are trying to search for digits in a line this method returns true only if the input has digits in all lines in the region.

Whereas, the **find()** method tries to find the next substring that matches the pattern i.e. if at least one match found in the region this method returns true.

#### 4. Java Threads

Typically, we can define threads as a sub process with lightweight with the smallest unit of processes and also has separate paths of execution. These threads use shared memory but they act independently hence if there is an exception in threads that do not affect the working of other threads despite them sharing the same memory.

Threads allows a program to operate more efficiently by doing multiple things at the same time. Threads can be used to perform complicated tasks in the background without interrupting the main program.

##### 4.1 Creating a Thread

There are two ways to create a thread.

It can be created by extending the Thread class and overriding its run() method:

###### 4.1.1 Extend Syntax



```
public class Main extends Thread
{
 public void run()
 {
 System.out.println("This code is running in a thread");
 }
}
```

Another way to create a thread is to implement the Runnable interface:

###### 4.1.2 Implement Syntax



```
public class Main implements Runnable
{
 public void run()
}
```

```
{
 System.out.println("This code is running in a thread");
}
}
```

### Differences between "extending" and "implementing" Threads

The major difference is that when a class extends the Thread class, you cannot extend any other class, but by implementing the Runnable interface, it is possible to extend from another class as well, like:  
class MyClass extends OtherClass implements Runnable.

## 4.2 Running a Thread

### 4.2.1 Extend Example

If the class extends the Thread class, the thread can be run by creating an instance of the class and call its start() method:



```
public class Main extends Thread
{
 public static void main(String[] args)
 {
 Main thread = new Main();
 thread.start();
 System.out.println("This code is outside of the thread");
 }
 public void run()
 {
 System.out.println("This code is running in a thread");
 }
}
```

**Output :**



This code is outside of the thread

This code is running in a thread

#### 4.2.2 Implement Example

If the class implements the Runnable interface, the thread can be run by passing an instance of the class to a Thread object's constructor and then calling the thread's start() method:



```
public class Main implements Runnable
{
 public static void main(String[] args)
 {
 Main obj = new Main();
 Thread thread = new Thread(obj);
 thread.start();
 System.out.println("This code is outside of the thread");
 }
 public void run()
 {
 System.out.println("This code is running in a thread");
 }
}
```

**Output :**



This code is outside of the thread

This code is running in a thread

#### 4.3 Concurrency Problems

Because threads run at the same time as other parts of the program, there is no way to know in which order the code will run. When the threads and main program are reading and writing the same variables, the values are unpredictable. The problems that result from this are called concurrency problems.

**Example :**

A code example where the value of the variable **amount** is unpredictable:



```
public class Main extends Thread
{
 public static int amount = 0;

 public static void main(String[] args)
 {
 Main thread = new Main();
 thread.start();
 System.out.println(amount);
 amount++;
 System.out.println(amount);
 }

 public void run() {
 amount++;
 }
}
```

**Output :**



```
0
2
```

### Solution

To avoid concurrency problems, it is best to share as few attributes between threads as possible. If attributes need to be shared, one possible solution is to use the `isAlive()` method of the thread to check whether the thread has finished running before using any attributes that the thread can change as shown in the following example :



```
public class Main extends Thread
{
 public static int amount = 0;

 public static void main(String[] args)
 {
 Main thread = new Main();
 thread.start();
 // Wait for the thread to finish
 while(thread.isAlive()) {
 System.out.println("Waiting...");
 }
 // Update amount and print its value
 System.out.println("Main: " + amount);
 amount++;
 System.out.println("Main: " + amount);
 }
}
```

```
public void run()
{
 amount++;
}
}
```

**Output :**



Waiting...

Main: 1

Main: 2

## Interview Questions

### 1 What is the purpose of using Wrapper Classes in Java?

- Wrapper Class will **convert primitive data types into objects**. The objects are necessary if we wish to modify the arguments passed into the method (because primitive types are **passed by value**).
- The classes in **java.util package** handles only objects and hence **wrapper classes** help in this case also.
- **Data structures** in the Collection framework such as **ArrayList and Vector** store only the objects (reference types) and not the **primitive types**.
- The object is needed to support **synchronization** in **multithreading**.

### 2 What Is an Exception?

The term *exception* is shorthand for the phrase "exceptional event." So, an *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called *throwing an exception*.

### 3 What do you understand by the term 'concurrency problems' particularly in the context of Multithreading?

When multiple threads try to read and write a shared variable concurrently, and these read and write operations overlap in execution, then the final outcome depends on the order in which the reads and writes take place, which is unpredictable. This is termed a concurrency problems.

## Additional Resources

1. Types of Exceptions in Java -

<https://www.geeksforgeeks.org/types-of-exception-in-java-with-examples/>

2. Metacharacters in Java in more detail -

<https://www.geeksforgeeks.org/metacharacters-in-java-regex/>

3. Comparison of Autoboxed Integer objects in Java -

<https://www.geeksforgeeks.org/comparison-autoboxed-integer-objects-java/>

4. More on the Matcher Class -

<https://jenkov.com/tutorials/java-regex/matcher.html>

## Agenda:

- **Introduction to Linked lists**
- **Insertion at positions in linked list**
- **Iteration in linked list**
- **Deletion at positions in linked list**

### Introduction to Linked Lists

Linked List is a very commonly used linear data structure which consists of group of **nodes** in a sequence.

Each node holds its own **data** and the **address of the next node** hence forming a chain like structure.

Linked Lists are used to create trees and graphs.



### Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

### Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

### Applications of Linked Lists

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.

- In Linked Lists we don't need to know the size in advance.

## Types of Linked Lists

There are 3 different implementations of Linked List available, they are:

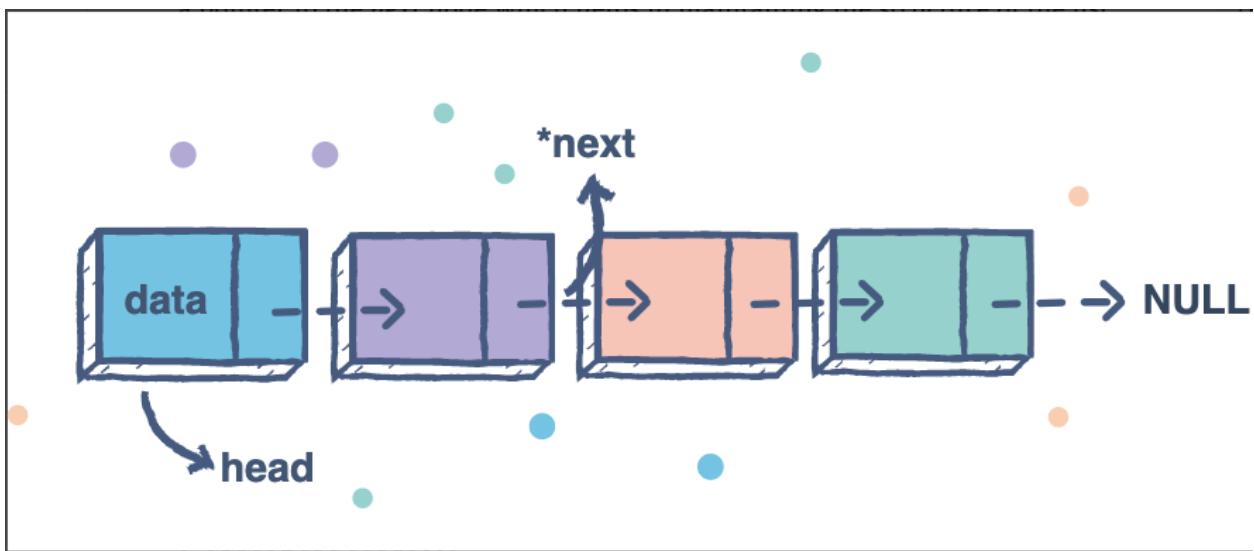
1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

### Introduction to Singly Linked list

A **singly linked list** is a type of linked list that is *unidirectional*, that is, it can be traversed in only one direction from head to the last node (tail).

Each element in a linked list is called a **node**. A single node contains *data* and a pointer to the *next* node which helps in maintaining the structure of the list.

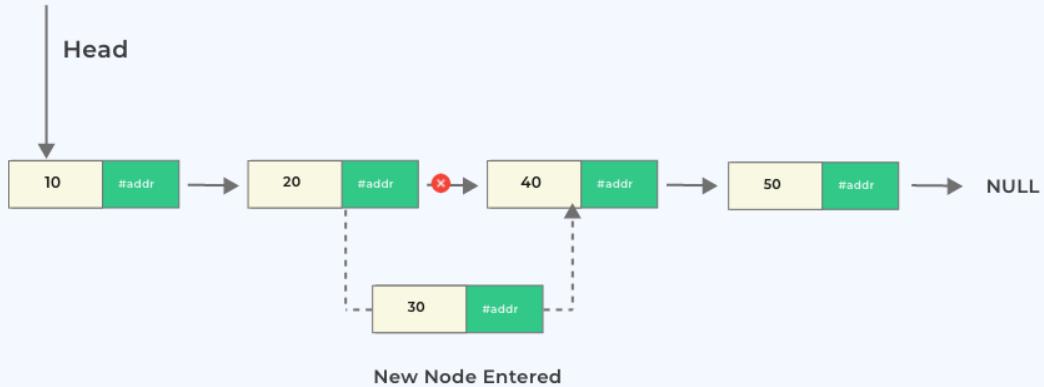
The first node is called the **head**; it points to the first node of the list and helps us access every other element in the list. The last node, also sometimes called the **tail**, points to *NULL* which helps us in determining when the list ends



### Insertion at positions in linked list

- Accept the position at which the user wants to enter the new node call this position : pos
- Visit the node just before that position i.e. pos - 1 node
- Change the next node of pos - 1 node this new node
- Change this new node's next node to pos + 1 node

## Insertion at a Specific Position in a Linked List in Java



Program to Insert a node at given position in linked list



```
import java.lang.*;
```

```
class LinkedList {
 Node head;
 int size = 0;
 // Node Class
 class Node{
 int data;
 Node next;

 Node(int x) // parameterized constructor
 {
 data = x;
 next = null;
 }
 }
```

```
}

public void insert(int data)
{
 Node new_node = new Node(data);

 new_node.data = data;
 new_node.next = head;
 head = new_node;
 size++;
}

public void insertPosition(int pos, int data) {
 Node new_node = new Node(data);
 new_node.data = data;
 new_node.next = null;

 // Invalid positions
 if(pos < 1 || pos > size + 1)
 System.out.println("Invalid\n");

 // inserting first node
 else if(pos == 1){
 new_node.next = head;
 head = new_node;
 size++;
 }

 else
 {
 Node temp = head;
```

```
// traverse till the current (pos-1)th node
while(--pos > 1){
 temp = temp.next;
}
new_node.next= temp.next;
temp.next = new_node;
size++;
}
```

```
public void display()
{
 System.out.print("Linked List : ");
```

```
Node node = head;
// as linked list will end when Node is Null
while(node!=null){
 System.out.print(node.data + " ");
 node = node.next;
}
System.out.println();
}
```

```
public static void main(String args[])
{
 LinkedList linked_list = new LinkedList();

 linked_list.insert(7);
```

```
linked_list.insert(6);
linked_list.insert(4);
linked_list.insert(3);
linked_list.insert(1);

linked_list.display();

// Inserts value: 2 at 2nd position
linked_list.insertPosition(2, 2);

// Inserts value: 5 at 5th position
linked_list.insertPosition(5, 5);

// Inserts value: 8 at 8th position
linked_list.insertPosition(8, 8);

linked_list.display();
}

}

Output:
```



Linked List : 1 3 4 6 7

Linked List : 1 2 3 4 5 6 7 8

#### **Iteration in Linked list**

Using loop:

- for loop
- while loop

- do while loop

Using iterators

- iterator() method

### Using For loop

The below program shows how to iterate a linked list in java using for loop.



```
import java.util.LinkedList;

public class AlmaBetter {

 public static void main(String[] args) {
 /*
 * Create an empty LinkedList
 */
 LinkedList<Integer> list = new LinkedList<Integer>();

 // Add values to the list
 list.add(1);
 list.add(2);
 list.add(3);
 list.add(4);
 list.add(9);
 list.add(11);

 // Iterate using for loop
 System.out.print("Iterating the linked list using for loop : ");
 for(int i = 0 ;i<list.size();i++) {
 System.out.print(list.get(i) + " ");
 }
 }
}
```

```
 }
 }
```

### Output

Iterating the linked list using for loop : 1 2 3 4 9 11

**Time Complexity:** O(n), as we are traversing the list, and n is the number of nodes in the linked list.

**Space Complexity:** O(1), Constant auxiliary space is being used.

### Using while loop

The below program shows how to iterate a linked list in java using while loop.



```
import java.util.LinkedList;

public class AlmaBetter {

 public static void main(String[] args) {
 /*
 * Create an empty LinkedList
 */

 LinkedList<integer> list = new LinkedList<integer>();

 // Add values to the list
 list.add(1);
 list.add(2);
 list.add(3);
 list.add(4);
 list.add(9);
 list.add(11);

 // Iterate using while loop
 System.out.print("Iterating the linked list using while loop : ");
 }
}
```

```
int i = 0;
while(i < list.size()) {
 System.out.print(list.get(i) + " ");
 i++;
}
}
}
```

### Output

Iterating the linked list using while loop : 1 2 3 4 9 11

**Time Complexity:** O(n), as we are traversing the list, and n is the number of nodes in the linked list.

**Space Complexity:** O(1), Constant auxiliary space is being used.

### Using do while loop

The below program shows how to iterate a linked list in java using do while loop.



```
import java.util.LinkedList;

public class AlmaBetter {

 public static void main(String[] args) {
 /*
 * Create an empty LinkedList
 */
 LinkedList<integer> list = new LinkedList<integer>();

 // Add values to the list
 list.add(1);
 list.add(2);
 list.add(3);
 list.add(4);
 }
}
```

```

list.add(9);

list.add(11);

// Iterate using do while loop

System.out.print("Iterating the linked list using do while loop : ");

int i = 0;

do {

 System.out.print(list.get(i) + " ");

 i++;

}

while(i < list.size());

}

}

```

### **Output**

Iterating the linked list using do while loop : 1 2 3 4 9 11

**Time Complexity:** O(n), as we are traversing the list, and n is the number of nodes in the linked list.

**Space Complexity:** O(1), Constant auxiliary space is being used.

### **Using iterator method**

The below program shows how to iterate a linked list in java using iterator() method.

### **Syntax**

LinkedList.iterator()

### **Return value**

The LinkedList.iterator() method returns an iterator over the sequence of elements of the Linked List in proper order.



```

import java.util.LinkedList;

import java.util.Iterator;

public class AlmaBetter {

```

```

public static void main(String[] args) {
 /*
 * Create an empty LinkedList
 */
 LinkedList<integer> list = new LinkedList<integer>();

 // Add values to the list
 list.add(1);
 list.add(2);
 list.add(3);
 list.add(4);
 list.add(9);
 list.add(11);

 // Iterate using iterator
 System.out.print("Iterating the linked list using iterator : ");

 Iterator<integer> itr = list.iterator();
 while(itr.hasNext()) {
 System.out.print(itr.next()+" ");
 }
}

```

### **Output**

Iterating the linked list using iterator : 1 2 3 4 9 11

**Time Complexity:** O(n), where n is the number of elements in the linked list.

**Space Complexity:** O(1), Constant auxiliary space is being used because iterator() creates an iterator over the elements of the list.

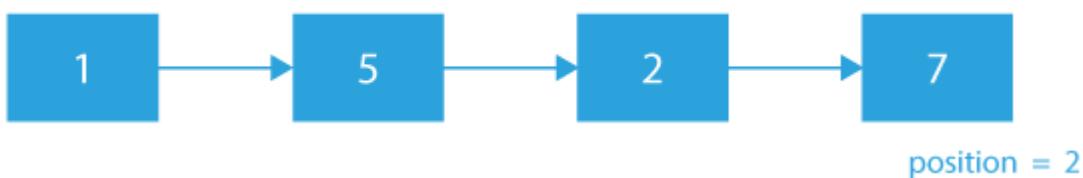
### Deletion at positions in linked list

Suppose the linked list is  $1 \rightarrow 5 \rightarrow 2 \rightarrow 7$  and the position given is 2. So, we have to delete the node present at the 2nd position.

**Note :** We are using 0 based indexing in this problem.

Here, in the given linked list, the node present at the 2nd position is 2. So, we have to delete 2 from the given list. The Final linked list after deleting 2 will be:  $1 \rightarrow 5 \rightarrow 7$

Input:



Output:



**Explanation :** The node at the 2nd position has been deleted.

To delete a node, we should also have the pointer to its previous node. We can tackle this by traversing till the  $(\text{position} - 1)$ th node.

First, let's think in terms of how to delete a node from the linked list at a given position.

Let the linked list be:

$1 \rightarrow 5 \rightarrow 2 \rightarrow 7$ .

So, here, if we delete the node (2), we get :

$1 \rightarrow 5 \rightarrow 7$ .

So, by deleting the node(2) we are making a connection between node(1) and node(5). That means the next of node(1) point to node(5).

### Observation

- If we take some more examples, we will notice that we need to access the previous and the next node of the node that we need to delete.

Say, if the node to be deleted is target, and its previous node is prev and its next node is next1. So, to do the deletion of target node from the linked list, we need to perform the following operations:

1.  $\text{prev} \rightarrow \text{next} = \text{next1}$ .
2. And finally free the target node.

By doing this, we are removing the target node at the given position and changing the necessary links.



```
public class LinkedList
```

```
{
```

```
 Node head;
```

```
 class Node
```

```
{
```

```
 int data;
```

```
 Node next;
```

```
 Node(int d)
```

```
{
```

```
 data = d;
```

```
 next = null;
```

```
}
```

```
}
```

```
 public void push(int new_data)
```

```
{
```

```
 Node new_node = new Node(new_data);
```

```
 new_node.next = head;
```

```
head = new_node;
}

void deleteNode(int position)
{

 if (head == null)
 return;

 Node temp = head;

 if (position == 0)
 {
 head = temp.next;
 return;
 }

 for (int i=0; temp!=null && i<position-1; i++)
 temp = temp.next;

 if (temp == null || temp.next == null)
 return;

 Node next1 = temp.next.next;

 temp.next = next1;
}

public void printList()
```

```
{
 Node tnode = head;
 while (tnode != null)
 {
 System.out.print(tnode.data + " ");
 tnode = tnode.next;
 }
}

public static void main(String[] args)
{
 LinkedList llist = new LinkedList();

 llist.push(7);
 llist.push(2);
 llist.push(5);
 llist.push(1);

 System.out.println("\nCreated Linked list is: ");
 llist.printList();

 llist.deleteNode(2);

 System.out.println("\nLinked List after Deletion at position 2: ");
 llist.printList();
}
}
Output:
```



Created Linked List: 1 5 2 7

Linked List after Deletion at position 2: 1 5 7

#### Conclusion:

In this session, we have learnt about:

- Introduction to Linked lists
- Insertion at positions in linked list
- Iteration in linked list
- Deletion at positions in linked list

#### Interview Questions:

- Explain Linked List in short.

A linked list may be defined as a linear data structure which can store a collection of items. In another way, the linked list can be utilized to store various objects of similar types. Each element or unit of the list is indicated as a node. Each node contains its data and the address of the next node. It is similar to a chain. Linked lists are used to create graphs and trees.

- Explain Singly Linked List in short.

The singly linked list includes nodes which contain a data field and next field. The next field further points to the next node in the line of nodes.

In other words, the nodes in singly linked lists contain a pointer to the next node in the list. We can perform operations like insertion, deletion, and traversal in singly linked lists.

A singly linked list is shown below whose nodes contain two fields: an integer value and a pointer value (a link to the next node).



## Singly Linked List

- How many pointers are necessary to implement a simple Linked List?

There are generally three types of pointers required to implement a simple linked list:

- A 'head' pointer which is used for pointing to the start of the record in a list.
- A 'tail' pointer which is used for pointing to the last node. The key factor in the last node is that its subsequent pointer points to nothing (NULL).
- A pointer in every node which is used for pointing to the next node element.

## Agenda:

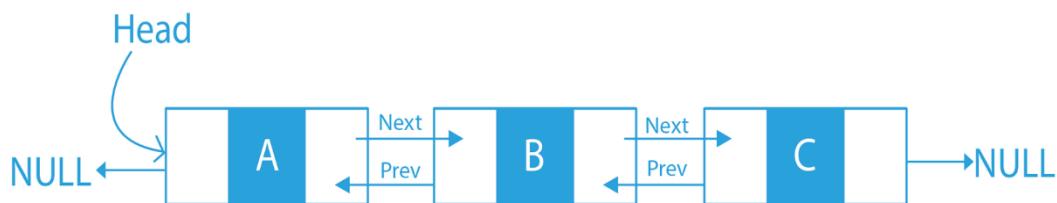
- Introduction to Double Linked list
- Double linked list creation
- Insertion at positions in Double linked list
- Deletion in Double linked list

In the last session, we have learnt about the linked list and mainly about single linked list, now in this session we will learn about double linked list:

### Introduction to Double linked list

A doubly linked list is a list that contains links to the next and previous nodes. We know that in a singly linked list, we can only traverse forward. But, in a doubly linked list, we can traverse both in a forward and backward manner.

### Doubly Linked List Representation



### Advantages over singly linked list

- A doubly linked list can be traversed both in the forward and backward directions.
- If the pointer to the node to be deleted is given, then the delete operation in a doubly-linked list is more efficient.
- Insertion of a new node before a given node is more efficient.

### Disadvantages over singly linked list

- Extra space is required to store the previous pointer.
- All operations of a doubly-linked list require an extra previous pointer to be maintained. We have to modify the previous pointers together with the next pointer.

### Creation of Double linked list

- Define a Node class which represents a node in the list. It will have three properties: data, previous which will point to the previous node and next which will point to the next node.
- Define another class for creating a doubly linked list, and it has two nodes: head and tail. Initially, head and tail will point to null.

- `addNode()` will add node to the list:
  - It first checks whether the head is null, then it will insert the node as the head.
  - Both head and tail will point to a newly added node.
  - Head's previous pointer will point to null and tail's next pointer will point to null.
  - If the head is not null, the new node will be inserted at the end of the list such that new node's previous pointer will point to tail.
  - The new node will become the new tail. Tail's next pointer will point to null.

a. `display()` will show all the nodes present in the list.

- Define a new node 'current' that will point to the head.
- Print `current.data` till `current` points to null.
- `Current` will point to the next node in the list in each iteration.



```
public class DoublyLinkedList {

 //Represent a node of the doubly linked list

 class Node{
 int data;
 Node previous;
 Node next;

 public Node(int data) {
 this.data = data;
 }
 }

 //Represent the head and tail of the doubly linked list
 Node head, tail = null;
```

```
//addNode() will add a node to the list
public void addNode(int data) {
 //Create a new node
 Node newNode = new Node(data);

 //If list is empty
 if(head == null) {
 //Both head and tail will point to newNode
 head = tail = newNode;
 //head's previous will point to null
 head.previous = null;
 //tail's next will point to null, as it is the
 // last node of the list
 tail.next = null;
 }
 else {
 //newNode will be added after tail such that tail's
 // next will point to newNode
 tail.next = newNode;
 //newNode's previous will point to tail
 newNode.previous = tail;
 //newNode will become new tail
 tail = newNode;
 //As it is last node, tail's next will point to null
 tail.next = null;
 }
}

//display() will print out the nodes of the list
```

```
public void display() {
 //Node current will point to head

 Node current = head;

 if(head == null) {
 System.out.println("List is empty");

 return;
 }

 System.out.println("Nodes of doubly linked list: ");

 while(current != null) {
 //Prints each node by incrementing the pointer.

 System.out.print(current.data + " ");

 current = current.next;
 }

}

public static void main(String[] args) {

 DoublyLinkedList dList = new DoublyLinkedList();

 //Add nodes to the list

 dList.addNode(1);
 dList.addNode(2);
 dList.addNode(3);
 dList.addNode(4);
 dList.addNode(5);

 //Displays the nodes present in the list
 dList.display();
}
```

}

Output:



Nodes of doubly linked list:

1 2 3 4 5

### **Insertion at positions in double linked list**

#### **Insert at the beginning.**

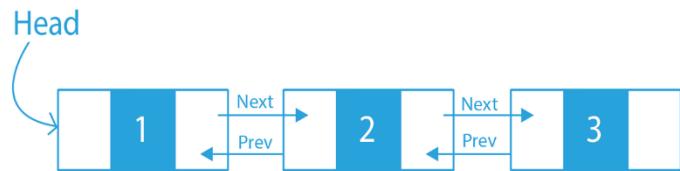
In this approach, the new node is always added at the start of the given doubly linked list, and the newly added node becomes the new head.

#### **Approach**

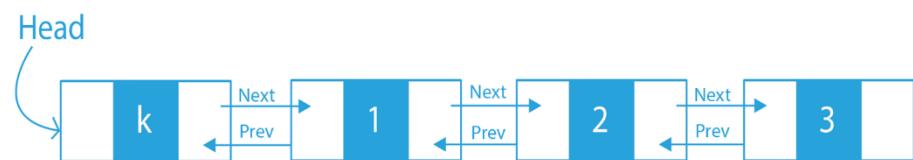
The approach is going to be very simple. We'll make the next of the new node point to the head. Then, we will make the previous pointer of head point to the new node. Lastly, we will make the new node the new head.

#### **Algorithm**

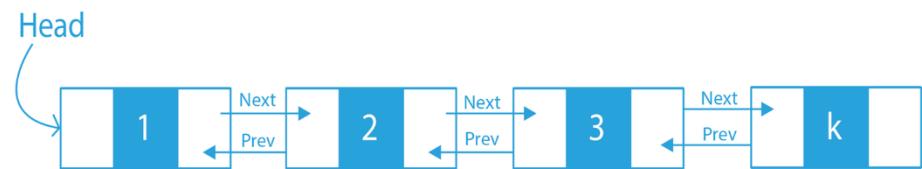
- Allocate the new node and put in the data
- Make the next of the new node as head and previous as NULL.
- If the head is not pointing to NULL, then change the previous pointer of the head node to the new node.
- Make the new node the new head.



Insert at beginning, value = k



Insert at end, value = k



```
public void push(int new_data)
{
 Node new_Node = new Node(new_data);

 new_Node.next = head;
 new_Node.prev = null;

 if (head != null)
 head.prev = new_Node;

 head = new_Node;
}
```

### **Time Complexity:**

O(1), as no traversal is needed.

### **Space Complexity:**

O(1), as no extra space is required.

#### **Insert at the end**

In this approach, the new node will always be added after the last node of the given doubly linked list.

#### **Approach**

The approach is going to be very simple. Firstly, we have to traverse till the end of the list. After reaching the last node, we will make it point to the new node. After this step, we will make the previous pointer of the new node point to the last node.

#### **Algorithm**

- Allocate the new node and put in the data.
- As the new node is going to be the last, so the next of this new node will be NULL.
- If the linked list is empty, then simply make the new node as the head node and return.
- Else, traverse till the end of the doubly linked list and make the next of the last node point to the new node.
- Lastly, make the prev of the new node point to the last node.



```
void append(int new_data)
{
 Node new_node = new Node(new_data);

 Node last = head;

 new_node.next = null;

 if (head == null) {
 new_node.prev = null;
```

```
 head = new_node;
 return;
}
```

```
while (last.next != null)
```

```
 last = last.next;
```

```
 last.next = new_node;
```

```
 new_node.prev = last;
```

```
}
```

#### **Time Complexity:**

$O(n)$ , as one traversal is needed.

#### **Space Complexity:**

$O(1)$ , as no extra space is required apart from the creation of node, which takes 20 bytes.

#### **Insert after a given node**

In this approach, the new node will always be added after a given node.

#### **Approach**

The approach is going to be very simple. We are given a pointer to a node as `prev_node`. The new node is to be inserted after `prev_node`.

As we have the pointer to that node (`prev_node`), we can perform this operation in  $O(1)$  time. Firstly, we will create the new node. Now, we will make the new node point to the next of `prev_node`. By doing this, we are making the new node point to the next node of `prev_node`.

Now, `prev` will point to the new node. Now, we just have to change the necessary links. So, we will make `prev_node` as the previous of the new node. In the end, if the new node is not the last node, we will make the new node's next node's prev as the new node. By doing this, we are making the previous of the next node of the new node point to the new node.

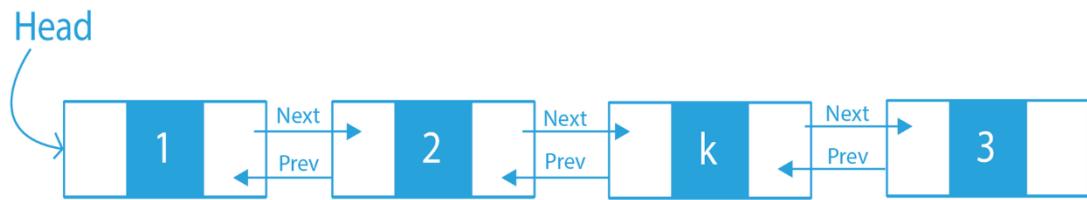
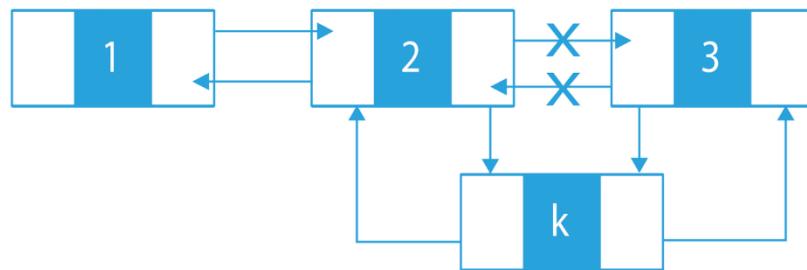
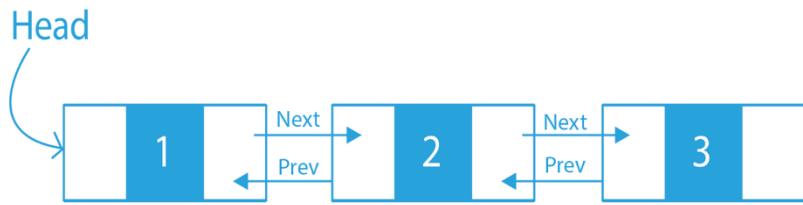
#### **Algorithm**

- Allocate the new node and put in the data.
- Make the new node point to the next of `prev_node`.
- Now, make the `prev_node` point to the new node.

- The previous of the new node will point to the prev\_node.
- If the new node is not the last node, it's next's previous point to the new node. By doing this, the new node becomes the previous of the next of the new node.
- If the linked list is empty, then simply make the new node as the head node and return.

## Insert after a given node

### Insert k after 2



```

public void InsertAfter(Node prev_Node, int new_data)
{

```

```

if (prev_Node == null) {
 System.out.println("The given previous node cannot be NULL ");
 return;
}

Node new_node = new Node(new_data);

new_node.next = prev_Node.next;

prev_Node.next = new_node;

new_node.prev = prev_Node;

if (new_node.next != null)
 new_node.next.prev = new_node;
}

```

### **Time Complexity:**

O(1), as no traversal is needed.

### **Space Complexity:**

O(1), as no extra space is required.

### **Insert before a given node**

In this approach, the new node will always be added before a given node.

### **Approach**

The approach is going to be very simple. We are given a pointer to a node as next\_node. The new node is to be inserted before next\_node.

As we have the pointer to that node (next\_node), we can perform this operation in O(1) time. Firstly, we will create the new node. Now, we will make the previous of the new node as the previous of the next\_node. By doing this, we are trying to add the new node in between the next\_node and the previous node of the next node.

Now, the previous of the next node will point to the new node and the next of the new node will point to the next\_node. Now, if the previous of the new node is not NULL, then the next of the previous node of the new node will point to the new node. By doing this, we are changing the appropriate links.

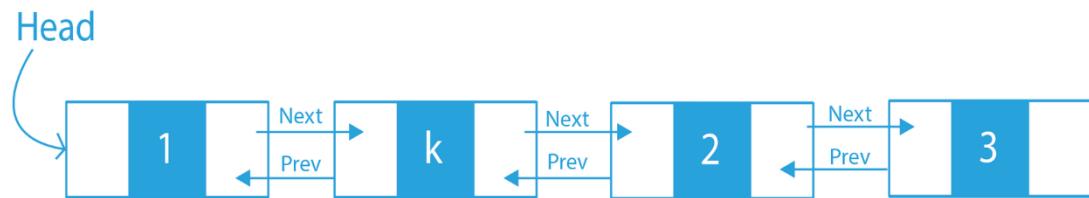
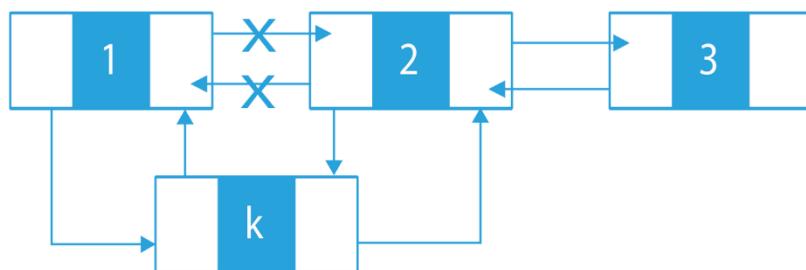
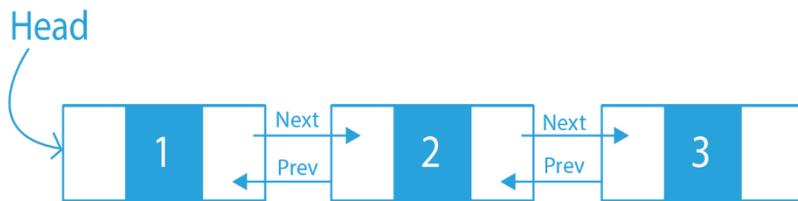
If the previous of the new node is NULL, it means that the new node is the first node in the list, hence it will become our new head.

### **Algorithm**

- Allocate the new node and put in the data.
- Make the previous of the new node point to the previous of the next node.
- The previous of the next\_node will point to the new node.
- The new node will point to the next\_node
- If the new node is not the head of the list, then the next of the previous of the new node will point to the new node. Else, the new node will become the head of the list.

Insert before a given node

Insert k before 2



```
void insertBefore(Node head_ref, Node next_node,int new_data)
{
 if(next_node==null)
 {
 System.out.println("the display next node cannot be null");
 return;
 }
}
```

```

Node new_node=new Node(new_data);

new_node.prev=next_node.prev;

next_node.prev=new_node;

new_node.next=next_node;

if(new_node.prev!=null)

{

 new_node.prev.next=new_node;

}

else

{

 head_ref=new_node;

}

}

```

#### **Time Complexity:**

O(1), as no traversal is needed.

#### **Space Complexity:**

O(1), as no extra space is required.

#### **Deletion in Double linked list**

If the given doubly linked list is head  $\leftrightarrow$  4  $\leftrightarrow$  5  $\leftrightarrow$  7  $\leftrightarrow$  6  $\leftrightarrow$  2  $\leftrightarrow$  1  $\leftrightarrow$  9 and the node to be deleted is head $\rightarrow$ next $\rightarrow$ next.

- According to the problem statement, we will have to delete the node head $\rightarrow$ next $\rightarrow$ next from the doubly linked list.
- In the above given linked list, the head $\rightarrow$ next $\rightarrow$ next is the node with value 7. So we will have to delete this node.
- So after deleting this node from the doubly linked list, our doubly linked list will be head  $\leftrightarrow$  4  $\leftrightarrow$  5  $\leftrightarrow$  6  $\leftrightarrow$  2  $\leftrightarrow$  1  $\leftrightarrow$  9.

While deleting a node **temp** in a doubly linked list, we must take care of the links of the following nodes, the node *temp* which we want to delete, the *immediate previous node to temp* and the *immediate next node of the temp* by changing the **next of the previous node of temp as the next of temp** and the **previous of the next node of temp as the previous of temp**.

- $\text{temp} \rightarrow \text{next} \rightarrow \text{prev} = \text{temp} \rightarrow \text{prev}$
- $\text{temp} \rightarrow \text{prev} \rightarrow \text{next} = \text{temp} \rightarrow \text{next}$
- $\text{delete}(\text{temp})$

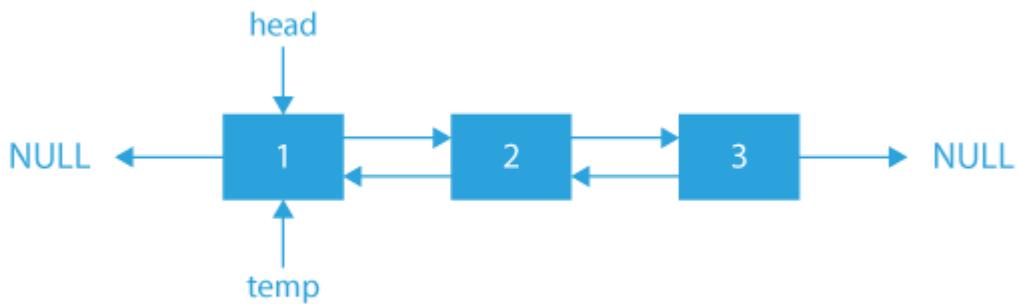
In the above example, the node we intended to delete was somewhere in the middle of the linked list. But if we wish to delete a node that is either the head or the tail of the doubly linked list, we will delete it a little differently, which we will see in the next section.

Now I think from the above example, the problem statement on how to delete a node in doubly linked list. is clear, so let's see in the next section how we can approach this problem.

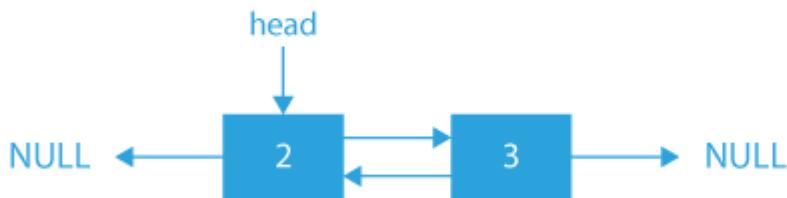
#### **Approach on deletion in doubly linked list.**

While deleting a node from a doubly linked list, there can be 3 cases:

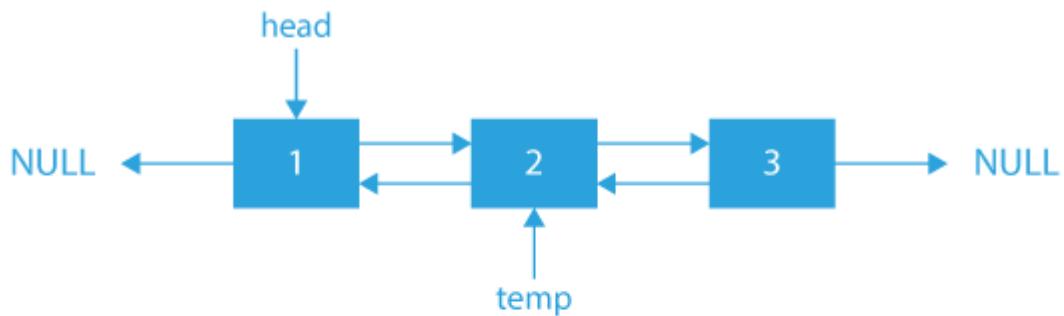
**Case1:** If the node to be deleted is the head node.



We have to delete node to which temp is pointing  
make,  $\text{head} = \text{head} \rightarrow \text{next}$   
and then delete ( $\text{temp}$ )



**Case2:** If the node to be deleted is somewhere in the middle of the linked list.



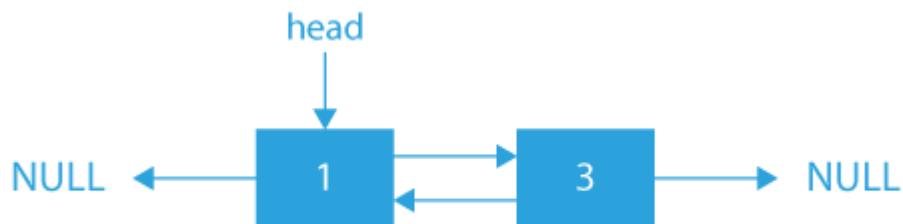
We have to delete node to which temp is pointing

As temp  $\rightarrow$  next  $\neq$  NULL

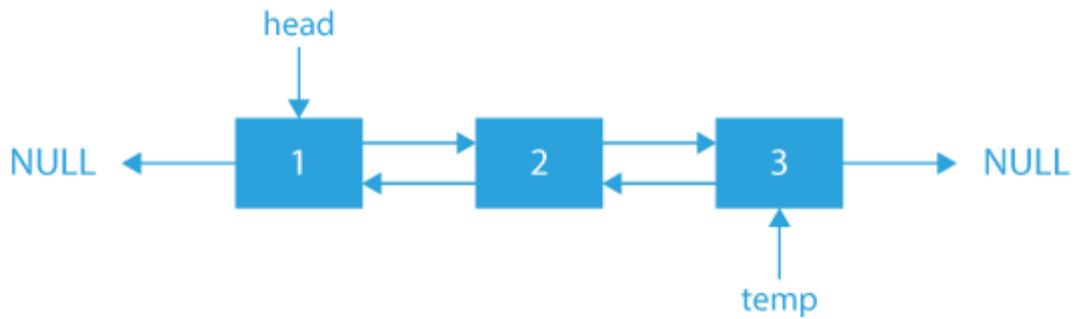
so make, temp  $\rightarrow$  next  $\rightarrow$  prev = temp  $\rightarrow$  prev

As temp  $\rightarrow$  prev  $\neq$  NULL

so make, temp  $\rightarrow$  prev  $\rightarrow$  next = temp  $\rightarrow$  next  
and then delete (temp)



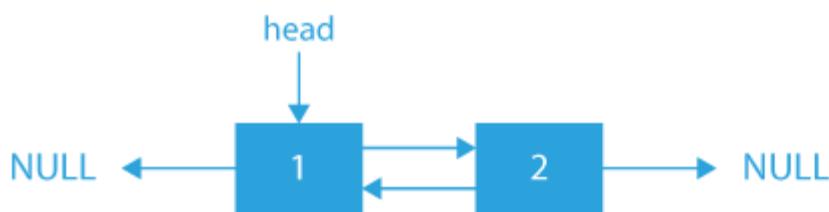
**Case3:** If the node to be deleted is the tail of the linked list.



We have to delete node to which temp is pointing

As temp → prev != NULL

so make, temp → prev → next = temp → next  
and then delete (temp)

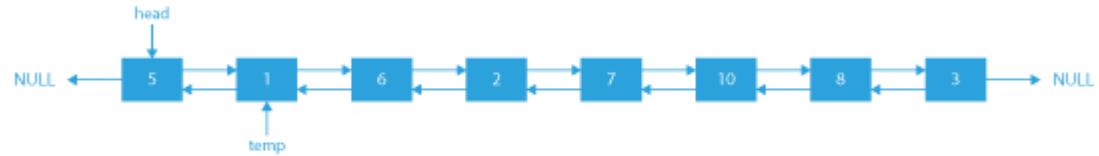


#### **Algorithm on deletion in doubly linked list.**

- If the node **temp** which we want to delete is NULL or the head is NULL, we will simply return.
- Then we will check if **temp** is a head node or not.
  - If **temp** is the head node, then we will move head to head→next.
- If the prev of **temp** is not NULL, we will change the prev of the next of **temp** to the prev of **temp**.
- If the next of **temp** is not NULL, we will change the next of the prev of **temp** to the next of **temp**.
- Finally, we will free the space allocated to the temp.
- After all the above steps, the node **temp** will be deleted from the Doubly Linked List.

#### **Dry Run on deletion in doubly linked list.**

## Initial doubly linked list



We have to delete this node temp from the doubly linked list

As temp → next != NULL

so make, temp → next → prev = temp → prev

As temp → prev != NULL

so make, temp → prev → next = temp → next  
and then free (temp)

After all the above steps temp has been deleted



Our output doubly linked list



```
public class AlmaBetter {
```

```
 public static class DLL {
```

```
 Node head;
```

```
 class Node {
```

```
 int data;
```

```
 Node prev;
```

```
 Node next;
```

```
Node(int d) {
 data = d;
}
}

public void push(int new_data) {
 Node new_Node = new Node(new_data);
 new_Node.next = head;
 new_Node.prev = null;

 if (head != null)
 head.prev = new_Node;

 head = new_Node;
}

public void printlist(Node node) {
 Node last = null;

 while (node.next != null) {
 System.out.print(node.data + " ");
 last = node;
 node = node.next;
 }

 System.out.println(node.data);
}

void deleteNode(Node temp) {
```

```
if (head == null || temp == null) {
 return;
}

if (head == temp) {
 head = head.next;
}

if (temp.next != null) {
 temp.next.prev = temp.prev;
}

if (temp.prev != null) {
 temp.prev.next = temp.next;
}

return;
}

}

public static void main(String[] args) {
 DLL dll = new DLL();
 dll.push(3);
 dll.push(8);
 dll.push(10);
 dll.push(7);
 dll.push(2);
 dll.push(6);
 dll.push(1);
}
```

```

 dll.push(5);

 dll.deleteNode(dll.head.next);

 dll.printlist(dll.head);

}

}

```

Output:



5 6 2 7 10 8 3

**Time Complexity:** O(1), as we do not need to do the traversal of the linked list.

#### Conclusion:

In this session, we have learnt about:

- Introduction to Double Linked list
- Double linked list creation
- Insertion at positions in Double linked list
- Deletion in Double linked list
- **What do you understand by Doubly Linked List?**

The doubly linked list includes a pointer (link) to the next node as well as to the previous node in the list. The two links between the nodes may be called "forward" and "backward," or "next" and "prev" (previous). A doubly linked list is shown below whose nodes consist of three fields: an integer value, a link that points to the next node, and a link that points to the previous node



## Doubly Linked List

- **Why would you use a doubly linked list instead of a single linked list?**

A doubly linked list offers a few advantages over a single linked list. First, because each node has pointers to both the next and previous nodes in the list, it is easier to navigate backwards through a doubly linked list than a single linked list. Additionally, because each node has a pointer to both the next and previous nodes, it is easier to insert and delete nodes from a doubly linked list than a single linked list.

- **How do you add an item to the end of a doubly linked list?**

You would add an item to the end of a doubly linked list by creating a new node with the data you want to add, and then linking that node to the last node in the list. The new node would then become the new last node in the list.

## Agenda:

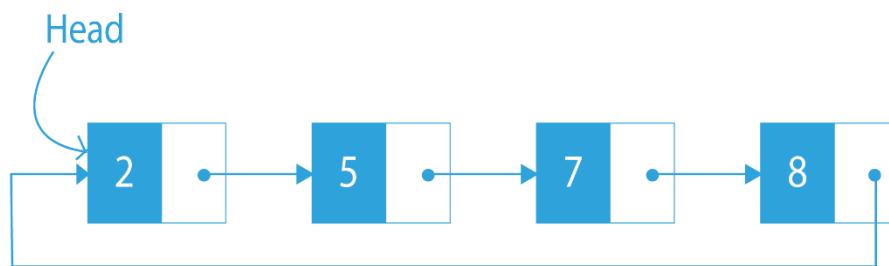
- **Introduction to Circular Linked List**
- **Creation of Circular Linked List**
- **Insertion at positions in Circular Linked List**
- **Deletion in Circular Linked List**

In the last session, we have learnt about the doubly linked list now in this session, we will learn about circular linked list:

### Introduction to Circular Linked List

Circular Linked List is a variation of a linked list where all the nodes are connected, forming a circle. This means that there is no NULL at the end. The last node, instead of pointing to NULL, points to the first node.

A singly linked list or a doubly linked list can be converted to a circular linked list.



### Advantages of Circular Linked Lists

- We can traverse the entire list using any node as the starting point. It means that any node can be the starting point. We can just end the traversal when we reach the starting node again.
- Useful for the implementation of a queue. We don't need to store two pointers that point to the front and the rear. Instead, we can just maintain a pointer to the last inserted node. By doing this. We can always obtain the front as the next of last.
- Some problems are circular and a circular data structure like a circular linked list is ideal for them.
- Circular doubly linked lists are very useful when it comes to the implementation Fibonacci Heap and many other advanced data structures.

### Disadvantages of Circular Linked Lists

- Finding the end of the list is harder, as there is no NULL to mark the end of the list.

### Creation of Circular Linked List

- Define a Node class which represents a node in the list. It has two properties data and next which will point to the next node.
- Define another class for creating the circular linked list and it has two nodes: head and tail. It has two methods: add() and display() .
- add() will add the node to the list:
  - It first checks whether the head is null, then it will insert the node as the head.
  - Both head and tail will point to the newly added node.
  - If the head is not null, the new node will be the new tail, and the new tail will point to the head as it is a circular linked list.

a. display() will show all the nodes present in the list.

- Define a new node 'current' that will point to the head.
- Print current.data till current will points to head
- Current will point to the next node in the list in each iteration.



```
public class CreateList {
 //Represents the node of list.

 public class Node{
 int data;
 Node next;
 public Node(int data) {
 this.data = data;
 }
 }

 //Declaring head and tail pointer as null.
 public Node head = null;
 public Node tail = null;

 //This function will add the new node at the end of the list.
}
```

```
public void add(int data){
 //Create new node
 Node newNode = new Node(data);
 //Checks if the list is empty.
 if(head == null) {
 //If list is empty, both head and tail would point to new node.
 head = newNode;
 tail = newNode;
 newNode.next = head;
 }
 else {
 //tail will point to new node.
 tail.next = newNode;
 //New node will become new tail.
 tail = newNode;
 //Since, it is circular linked list tail will point to head.
 tail.next = head;
 }
}
```

```
//Displays all the nodes in the list
public void display() {
 Node current = head;
 if(head == null) {
 System.out.println("List is empty");
 }
 else {
 System.out.println("Nodes of the circular linked list: ");
 do{
```

```

 //Prints each node by incrementing pointer.
 System.out.print(" " + current.data);

 current = current.next;

}while(current != head);

System.out.println();

}

}

```

```

public static void main(String[] args) {

 CreateList cl = new CreateList();

 //Adds data to the list

 cl.add(1);

 cl.add(2);

 cl.add(3);

 cl.add(4);

 //Displays all the nodes present in the list

 cl.display();

}

}

```

Output:

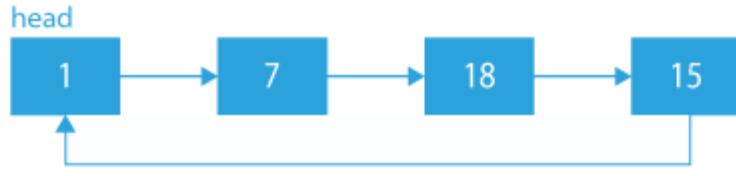


Nodes of the circular linked list:

1 2 3 4

### **Iteration in Circular Linked List**

Suppose the given circular linked list is:



Now, we have to traverse this list. So, the output after traversing the list and printing every element will be: 1 7 18 15

### Approach

Which node should we choose as our starting node?

- The head node will make our life easier as we already have a pointer to the head of the list.

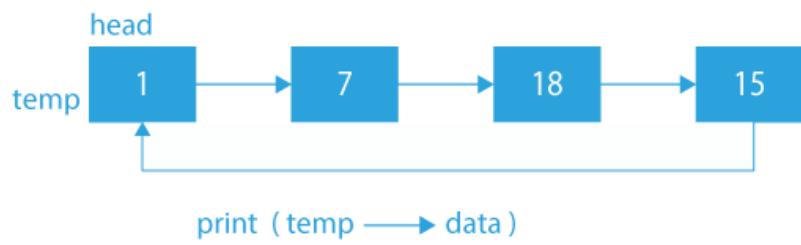
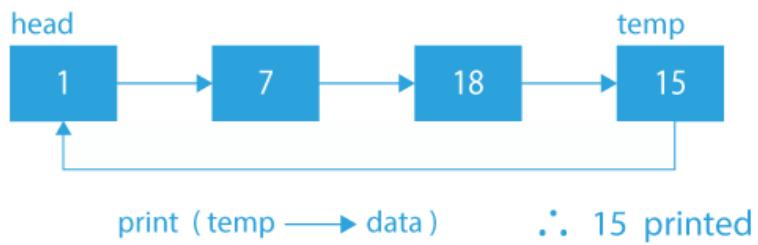
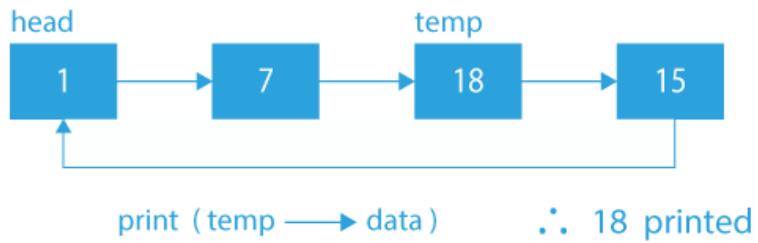
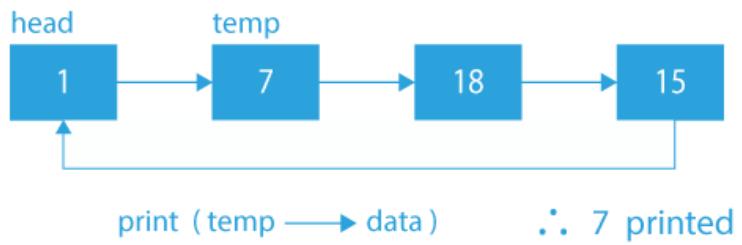
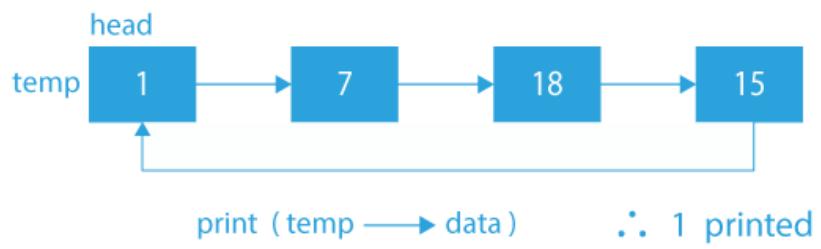
Create a node temp and make it point to the head. Now, keep incrementing temp while temp is not equal to the head of the list. In every iteration, print the temp's data.

As explained already, we are using the head as our starting node, and terminating the loop when we are reaching the head again.

### Algorithm

- If the head is NULL, simply return because the list is empty.
- Create a node temp and make it point to the head of the list.
- Now, with the help of a do-while loop, keep printing the temp → data and increment the temp, while temp is not equal to the head of the list.
- As we have chosen the head as our starting point, we are terminating the loop when we are reaching the head again.

### Dry Run



Now , as `temp = head` , loop will terminate.

Final Output

1            7            18            15



```
public class PrepBytes
{
 static class Node
 {
 int data;
 Node next;
 };

 static Node push(Node head_ref,
 int data)
 {
 Node ptr1 = new Node();
 Node temp = head_ref;
 ptr1.data = data;
 ptr1.next = head_ref;

 if (head_ref != null)
 {
 while (temp.next != head_ref)
 temp = temp.next;
 temp.next = ptr1;
 }
 else
 ptr1.next = ptr1;

 head_ref = ptr1;
 }
}
```

```
 return head_ref;
}

static void printList(Node head)
{
 Node temp = head;
 if (head != null)
 {
 do
 {
 System.out.print(temp.data + " ");
 temp = temp.next;
 }
 while (temp != head);
 }
}

public static void main(String args[])
{
 Node head = null;

 head = push(head, 15);
 head = push(head, 18);
 head = push(head, 7);
 head = push(head, 1);

 System.out.println("Contents of Circular " +
 "Linked List:");
}
```

```
 printList(head);
}
}
```

Output:



Contents of Circular Linked List:

1 7 18 15

### **Insertion in Circular Linked List**

There are four main insertion operations:

- Insert at the beginning of the list
- Insert at the end of the list.
- Insert in an empty list.
- Insert in between the nodes.

Let's have a glance at the approaches and algorithms for the above four insertion operations.

#### **Insert at the beginning**

##### **Approach**

While inserting at the beginning in a circular linked list, we have to keep in mind that the last node always points to the head node.

If we keep the above point in mind, we can say that firstly, we will make the new node point to the head node. Now, as we know that the last should point to the first node of the list, so we will make the last node point to the new node. In this way, the last node will point to the newly created node, and the newly created node will point to the head. Now, the newly created node will become the head.

##### **Algorithm**

- Create the node which is to be inserted, say NewNode.
- Do `NewNode -> next = last -> next`. This step ensures that the current node is being added at the beginning of the list, as the next of the last node is always the head of the list. The NewNode will become the head.
- As it is a circular linked list, the last will now point to the new node, so `last -> next = NewNode`. By doing this, the last node will now point to the newly created node, which is our new head. In this way, the tail is being updated.



```

Node addBegin(Node last,int data)
{
 if(last==null)
 {
 return addToEmpty(last,data);
 }

 Node newNode=new Node(data);
 newNode.next=last.next;
 last.next=newNode;

 return last;
}

```

### **Insert at the end**

#### **Approach**

While inserting at the end in a circular linked list, we have to keep in mind that the last node always points to the head node.

If we keep the above point in mind, we can say that firstly, we will make the new node point to the next of the last node. After doing this, we will make the last node point to the new node. In the end, the new node will become the last node.

#### **Algorithm**

- Create the node which is to be inserted, say NewNode.
- Make the new node point to the next of the last node  $\text{NewNode} \rightarrow \text{next} = \text{last} \rightarrow \text{next}$ .
- Make the last node point to the new node  $\text{last} \rightarrow \text{next} = \text{NewNode}$ . By doing this, the last node, instead of pointing to the head node, will point to the new node.
- In the end, the NewNode will become the last node. In this way, the tail is being updated.



```

Node addEnd(Node last,int data)
{
 if(last==null)
 {

```

```

 return addToEmpty(last,data);

 }

Node NEmode=new Node(data);

NewNode.next=last.next;

last.next=NewNode;

last=NewNode;

return last;
}

```

### **Insert in an empty list**

#### **Approach**

While inserting in an empty list, we have to keep in mind that the last node will be NULL initially.

If we keep the above point in mind, we can say that firstly, we will allocate memory for the new node. Now, the new node will become the new tail. As there is only a singly node, so this new node is the tail as well as the head of the list. So, the new node will point to itself. In this way, we can update the head and tail, and insert in an empty list.

#### **Algorithm**

- Create the node which is to be inserted, say NewNode.
- Make the new node the last node last = NewNode.
- As it is the only node in the list, it will be the tail as well as head.
- Keeping the above point in mind, we will make the NewNode point to the last(itself) NewNode – > next = last.
- By performing the above steps, the head and tail are being updated, as well as the NewNode is being added to the empty list.



```

Node addToEmpty(Node last,int data)

{
 if(last!=null)
 {
 return last;
 }
}

```

```

 }

 Node NewNode=new Node(data);

 last=NewNode;

 last.next=last;

 return last;
}

```

### **Insert in between the nodes**

#### **Approach**

In this approach, we will be given a node's data. We will have to insert the new node just after this given node.

To achieve this, we will first create the node which is to be inserted, say NewNode, then we will simply traverse the list till we find the the node with the given data. Let the node that is found is P. So, to insert this New Node just after P, we will simply do NewNode –> next = P –> next. After this, make P point to this NewNode. By doing this, we are successfully inserting the new node after a given node.

#### **Algorithm**

- Create the node which is to be inserted, say NewNode.
- Traverse through the list till the node with the given data is not found.
- Store the node in P.
- Make the next of NewNode point to the next of P, NewNode –> next = P –> next
- Make P point to the NewNode P –> next = NewNode.



```

public void insertAfter(Node prev_node, int new_data)
{
 /* 1. Check if the given Node is null */

 if (prev_node == null) {
 System.out.println(
 "The given previous node cannot be null");

 return;
 }
}

```

```
}
```

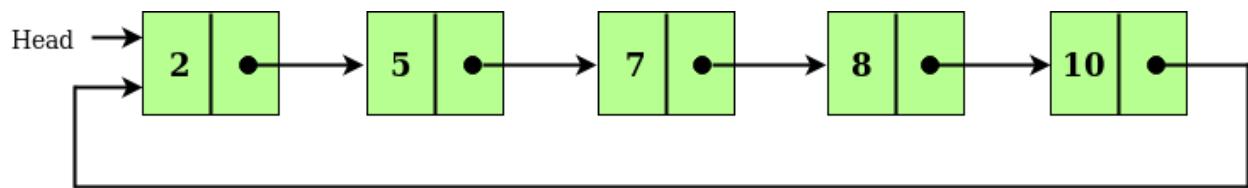
```
/* 2. Allocate the Node &
3. Put in the data*/
Node new_node = new Node(new_data);

/* 4. Make next of new Node as next of prev_node */
new_node.next = prev_node.next;

/* 5. make next of prev_node as new_node */
prev_node.next = new_node;
}
```

### **Deletion in Circular Linked List**

Consider the linked list as shown below:



We will be given a node and our task is to delete that node from the circular linked list.

#### **Examples:**



Input : 2->5->7->8->10->(head node)

data = 5

Output : 2->7->8->10->(head node)

Input : 2->5->7->8->10->(head node)

7

Output : 2->5->8->10->(head node)

#### **Algorithm:**

**Case 1:** List is empty.

- If the list is empty we will simply return.

**Case 2:** List is not empty

- If the list is not empty then we define two pointers **curr** and **prev** and initialize the pointer **curr** with the **head** node.
- Traverse the list using **curr** to find the node to be deleted and before moving to curr to the next node, every time set **prev = curr**.
- If the node is found, check if it is the only node in the list. If yes, set **head = NULL** and **free(curr)**.
- If the list has more than one node, check if it is the first node of the list. Condition to check this (**curr == head**). If yes, then move **prev** until it reaches the last node. After **prev** reaches the last node, set **head = head -> next** and **prev -> next = head**. Delete **curr**.
- If **curr** is not the first node, we check if it is the last node in the list. Condition to check this is (**curr -> next == head**).
- If **curr** is the last node. Set **prev -> next = head** and delete the node **curr** by **free(curr)**.
- If the node to be deleted is neither the first node nor the last node, then set **prev -> next = curr -> next** and delete **curr**.



```
// Java program to delete a given key from
// linked list.

class AlmaBetter {

 /* ure for a node */
 static class Node {
 int data;
 Node next;
 };

 /* Function to insert a node at the beginning of
 * a Circular linked list */
 static Node push(Node head_ref, int data)
```

```

{

 // Create a new node and make head as next
 // of it.

 Node ptr1 = new Node();
 ptr1.data = data;
 ptr1.next = head_ref;

 /* If linked list is not null then set the
 next of last node */

 if (head_ref != null) {
 // Find the node before head and update
 // next of it.

 Node temp = head_ref;
 while (temp.next != head_ref)
 temp = temp.next;

 temp.next = ptr1;
 }
 else
 ptr1.next = ptr1; /*For the first node */

 head_ref = ptr1;
 return head_ref;
}

/* Function to print nodes in a given
circular linked list */

static void printList(Node head)

{
 Node temp = head;

```

```

if (head != null) {
 do {
 System.out.printf("%d ", temp.data);
 temp = temp.next;
 } while (temp != head);
}

System.out.printf("\n");
}

/* Function to delete a given node from the list */
static Node deleteNode(Node head, int key)
{
 if (head == null)
 return null;

 // Find the required node
 Node curr = head, prev = new Node();
 while (curr.data != key) {
 if (curr.next == head) {
 System.out.printf("\nGiven node is not found"
 + " in the list!!!!");
 break;
 }

 prev = curr;
 curr = curr.next;
 }
}

```

```
// Check if node is only node
if (curr == head && curr.next == head) {
 head = null;
 return head;
}

// If more than one node, check if
// it is first node
if (curr == head) {
 prev = head;
 while (prev.next != head)
 prev = prev.next;
 head = curr.next;
 prev.next = head;
}

// check if node is last node
else if (curr.next == head) {
 prev.next = head;
}
else {
 prev.next = curr.next;
}
return head;
}

/* Driver code */
public static void main(String args[])
{
```

```

/* Initialize lists as empty */

Node head = null;

/* Created linked list will be 2.5.7.8.10 */

head = push(head, 2);
head = push(head, 5);
head = push(head, 7);
head = push(head, 8);
head = push(head, 10);

System.out.printf("List Before Deletion: ");
printList(head);

head = deleteNode(head, 7);

System.out.printf("List After Deletion: ");
printList(head);

}
}


```

Output:



List Before Deletion: 10 8 7 5 2

List After Deletion: 10 8 5 2

#### **Conclusion:**

In this session, we have learnt about:

- Introduction to Circular Linked List
- Creation of Circular Linked List
- Insertion at positions in Circular Linked List

- Deletion in Circular Linked List

## Interview Questions

- **What are circular linked lists?**

A circular linked list is a type of linked list where the last node in the list points back to the first node in the list. This creates a loop or cycle in the list. Circular linked lists are often used in applications where data needs to be processed in a continuous loop, such as in a video game or music player.

- **How do you delete an existing node in a circular linked list?**

You would need to find the node before the one you want to delete, and then change its next pointer to point to the node after the one you want to delete. Then, you can simply delete the node you wanted to delete without having to worry about any pointers becoming orphaned.

- **Can you explain how to traverse a circular linked list?**

To traverse a circular linked list, you would start at the head of the list and follow the links until you reach the head again. This will ensure that you visit every node in the list.

## Agenda

- Drawbacks of Array Implementation
- Implementation of Queue using Linked List
- Queue as a library

### Implementation of Queue using Linked List

Due to the drawbacks discussed, the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

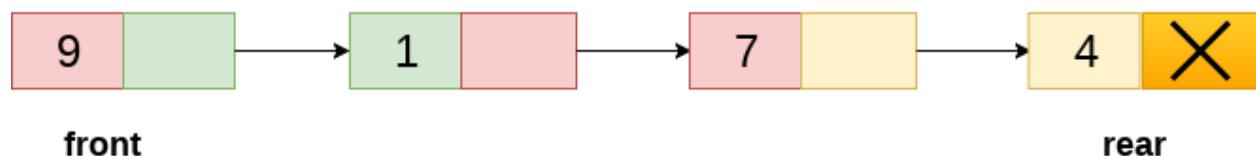
The storage requirement of linked representation of a queue with n elements is  $O(n)$  while the time requirement for operations is  $O(1)$ .

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



## Linked Queue

### Operation on Linked Queue

- **enQueue()** This operation adds a new node after **rear** and moves **rear** to the next node.
- **deQueue()** This operation removes the front node and moves **front** to the next node.

### Approach:

Follow the below steps to Implement the queue using linked list:

- Create a Struct QNode with data members integer data and QNode\* next.
  - A parameterized constructor that takes an integer **x** value as a parameter and sets data equal to **x** and next as **NULL**.

- Create a struct Queue with data members QNode **front** and **rear**.
- Default constructor Queue() sets front and rear as NULL.
- Enqueue Operation with parameter **x**:
  - Initialize QNode\* **temp** with data = x;
  - If **rear** is set to NULL then set **front** and **rear** to **temp** and return.
  - Else set **rear**'s next to **temp** and then move **rear** to **temp**.
- Dequeue Operation:
  - If **front** is set to NULL return.
  - Initialize QNode\* **temp** with **front** and set **front** to its next.
  - If **front** is equal to **NULL** then set **rear** to **NULL**.
  - Delete **temp**.

Below is the Implementation of the above approach:



```
// Java program for linked-list implementation of queue

// A linked list (LL) node to store a queue entry
class QNode {
 int key;
 QNode next;

 // constructor to create a new linked list node
 public QNode(int key)
 {
 this.key = key;
 this.next = null;
 }
}
```

```
// A class to represent a queue
// The queue, front stores the front node of LL and rear
// stores the last node of LL
class Queue {
 QNode front, rear;

 public Queue() { this.front = this.rear = null; }

 // Method to add an key to the queue.
 void enqueue(int key)
 {

 // Create a new LL node
 QNode temp = new QNode(key);

 // If queue is empty, then new node is front and
 // rear both
 if (this.rear == null) {
 this.front = this.rear = temp;
 return;
 }

 // Add the new node at the end of queue and change
 // rear
 this.rear.next = temp;
 this.rear = temp;
 }

 // Method to remove an key from queue.
```

```
void dequeue()
{
 // If queue is empty, return NULL.
 if (this.front == null)
 return;

 // Store previous front and move front one node
 // ahead
 QNode temp = this.front;
 this.front = this.front.next;

 // If front becomes NULL, then change rear also as
 // NULL
 if (this.front == null)
 this.rear = null;
}

// Driver class
public class Test {
 public static void main(String[] args)
 {
 Queue q = new Queue();
 q.enqueue(10);
 q.enqueue(20);
 q.dequeue();
 q.dequeue();
 q.enqueue(30);
 q.enqueue(40);
 }
}
```

```

 q.enqueue(50);
 q.dequeue();
 System.out.println("Queue Front : " + q.front.key);
 System.out.println("Queue Rear : " + q.rear.key);
 }
}

```

## **Complexity**

### **Time Complexity:**

$O(1)$ , The time complexity of both operations enqueue() and dequeue() is  $O(1)$  as it only changes a few pointers in both operations.

### **Auxiliary Space:**

$O(1)$ , Space complexity of both operations enqueue() and dequeue() is  $O(1)$  as constant extra space is required.

## **Java Queue Interface**

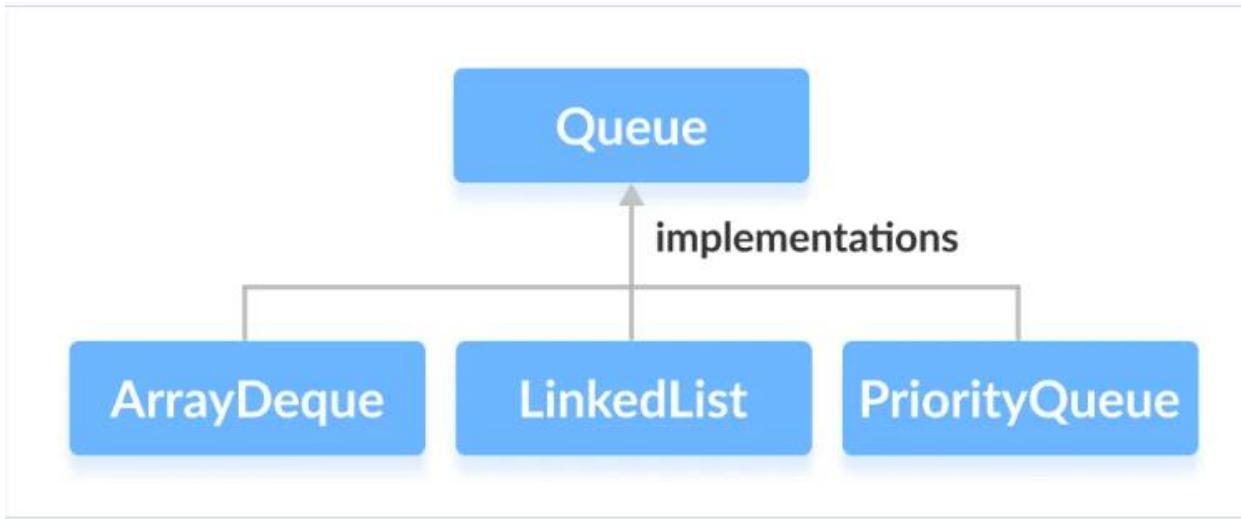
The Queue interface of the Java collections framework provides the functionality of the queue data structure. It extends the Collection interface

### **Classes that Implement Queue**

Since the Queue is an interface, we cannot provide the direct implementation of it.

In order to use the functionalities of Queue, we need to use classes that implement it:

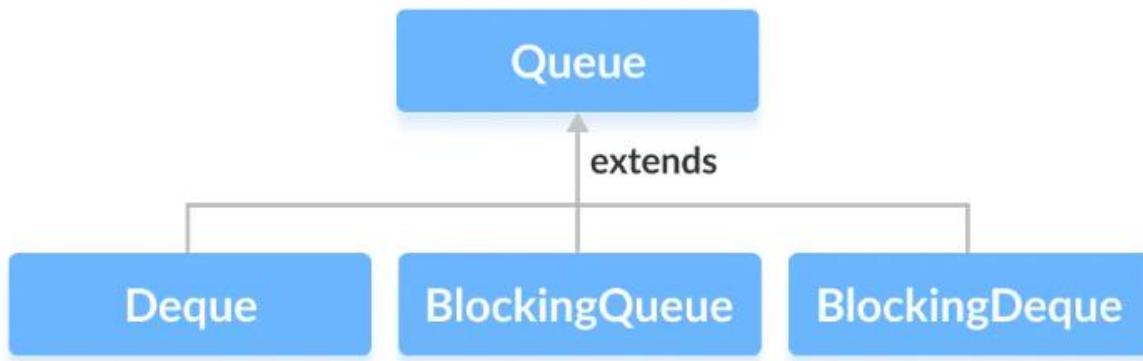
- ArrayDeque
- LinkedList
- PriorityQueue



### Interfaces that extend Queue

The `Queue` interface is also extended by various subinterfaces:

- `Deque`
- `BlockingQueue`
- `BlockingDeque`



### How to use Queue?

In Java, we must import `java.util.Queue` package in order to use `Queue`.



```
// LinkedList implementation of Queue
Queue<String> animal1 = new LinkedList<>();
```

```
// Array implementation of Queue
Queue<String> animal2 = new ArrayDeque<>();
```

```
// Priority Queue implementation of Queue
Queue<String> animal3 = new PriorityQueue<>();
```

Here, we have created objects animal1, animal2 and animal3 of classes LinkedList, ArrayDeque and PriorityQueue respectively. These objects can use the functionalities of the Queue interface.

### Methods of Queue

The Queue interface includes all the methods of the Collection interface. It is because Collection is the super interface of Queue.

Some of the commonly used methods of the Queue interface are:

- **add()** - Inserts the specified element into the queue. If the task is successful, add() returns true, if not it throws an exception.
- **offer()** - Inserts the specified element into the queue. If the task is successful, offer() returns true, if not it returns false.
- **element()** - Returns the head of the queue. Throws an exception if the queue is empty.
- **peek()** - Returns the head of the queue. Returns null if the queue is empty.
- **remove()** - Returns and removes the head of the queue. Throws an exception if the queue is empty.
- **poll()** - Returns and removes the head of the queue. Returns null if the queue is empty.

### Implementation of the Queue Interface

#### 1. Implementing the LinkedList Class



```
import java.util.Queue;
import java.util.LinkedList;

class Main {

 public static void main(String[] args) {
```

```

// Creating Queue using the LinkedList class

Queue<Integer> numbers = new LinkedList<>();

// offer elements to the Queue
numbers.offer(1);
numbers.offer(2);
numbers.offer(3);
System.out.println("Queue: " + numbers);

// Access elements of the Queue
int accessedNumber = numbers.peek();
System.out.println("Accessed Element: " + accessedNumber);

// Remove elements from the Queue
int removedNumber = numbers.poll();
System.out.println("Removed Element: " + removedNumber);

System.out.println("Updated Queue: " + numbers);

}

}

/*
OUTPUT
Queue: [1, 2, 3]
Accessed Element: 1
Removed Element: 1
Updated Queue: [2, 3]
*/

```

## 2. Implementing the PriorityQueue Class



```
import java.util.Queue;
import java.util.PriorityQueue;

class Main {

 public static void main(String[] args) {
 // Creating Queue using the PriorityQueue class
 Queue<Integer> numbers = new PriorityQueue<>();

 // offer elements to the Queue
 numbers.offer(5);
 numbers.offer(1);
 numbers.offer(2);
 System.out.println("Queue: " + numbers);

 // Access elements of the Queue
 int accessedNumber = numbers.peek();
 System.out.println("Accessed Element: " + accessedNumber);

 // Remove elements from the Queue
 int removedNumber = numbers.poll();
 System.out.println("Removed Element: " + removedNumber);

 System.out.println("Updated Queue: " + numbers);
 }
}
```

```
/*
OUTPUT
Queue: [1, 5, 2]
Accessed Element: 1
Removed Element: 1
Updated Queue: [2, 5]
*/
```

**Note :** Unlike normal queues, priority queue elements are retrieved in sorted order.

Suppose, we want to retrieve elements in the ascending order. In this case, the head of the priority queue will be the smallest element. Once this element is retrieved, the next smallest element will be the head of the queue.

It is important to note that the elements of a priority queue may not be sorted. However, elements are always retrieved in sorted order.

### Conclusion

In the last two sessions, Queue data structure has been covered in depth.

### Interview Questions

#### 1. What is Complexity Analysis of Queue operations?

- Queues offer random access to their contents by shifting the first element off the front of the queue. You have to do this repeatedly to access an arbitrary element somewhere in the queue. Therefore, **access** is  $*O*(n)$ .
- Searching for a given value in the queue requires iterating until you find it. So **search** is  $*O*(n)$ .
- Inserting into a queue, by definition, can only happen at the back of the queue, similar to someone getting in line for a delicious Double-Double burger at In 'n Out. Assuming an efficient queue implementation, queue **insertion** is  $*O*(1)$ .
- Deleting from a queue happens at the front of the queue. Assuming an efficient queue implementation, queue **deletion** is  $*O*(1)$ .

#### 2. List some Queue real-life applications

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

- Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

- In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

### **Additional Resources**

- Dequeue :

[<https://www.scaler.com/topics/dequeue-in-data-structure/>] (<https://www.scaler.com/topics/dequeue-in-data-structure/>)

- Priority Queue :

[<https://www.programiz.com/dsa/priority-queue>] (<https://www.programiz.com/dsa/priority-queue>)

## Agenda

- Implementation of Stack using Linked List
- Stack as a library

### Stack Implementation using Linked List

To implement a **stack** using the singly linked list concept, all the singly **linked list** operations should be performed based on Stack operations LIFO(last in first out) and with the help of that knowledge, we are going to implement a stack using a singly linked list.

So we need to follow a simple rule in the implementation of a stack which is **last in first out** and all the operations can be performed with the help of a top variable. Let us learn how to perform **Pop, Push, Peek, and Display** operations.

In the stack Implementation, a stack contains a top pointer. which is the “head” of the stack where pushing and popping items happens at the head of the list. The first node has a null in the link field and second node-link has the first node address in the link field and so on and the last node address is in the “top” pointer.

The main advantage of using a linked list over arrays is that it is possible to implement a stack that can shrink or grow as much as needed. Using an array will put a restriction on the maximum capacity of the array which can lead to stack overflow. Here each new node will be dynamically allocated. so overflow is not possible.

## Algorithm

### Push Operation



Initialise a node

Update the value of that node by data i.e. `node->data = data`

Now link this node to the top of the linked list

And update top pointer to the current node

### Pop Operation



First Check whether there is any node present in the linked list or not, if not then return

Otherwise make pointer let say temp to the top node and move forward the top node by 1 step

Now free this temp node

### **Peek Operation**



Check if there is any node present or not, if not then return

Otherwise return the value of top node of the linked list

### **Display Operation**



Take a temp node and initialize it with top pointer

Now start traversing temp till it encounters NULL

Simultaneously print the value of the temp node

### **Code**



```
// Driver code
class Solution {
 public static void main(String[] args) {
 // create Object of Implementing class
 StackUsingLinkedlist obj
 = new StackUsingLinkedlist();
 // insert Stack value
 obj.push(11);
```

```
obj.push(22);
obj.push(33);
obj.push(44);

// print Stack elements
obj.display();

// print Top element of Stack
System.out.printf("\nTop element is %d\n",
 obj.peek());

// Delete top element of Stack
obj.pop();
obj.pop();

// print Stack elements
obj.display();

// print Top element of Stack
System.out.printf("\nTop element is %d\n",
 obj.peek());
}

}

// Create Stack Using Linked list
class StackUsingLinkedlist {

 // A linked list node
 private class Node {
```

```
int data; // integer data
Node link; // reference variable Node type
}

// create global top reference variable global
Node top;
// Constructor
StackUsingLinkedlist() { this.top = null; }

// Utility function to add an element x in the stack
public void push(int x) // insert at the beginning
{
 // create new node temp and allocate memory
 Node temp = new Node();

 // check if stack (heap) is full. Then inserting an
 // element would lead to stack overflow
 if (temp == null) {
 System.out.print("\nHeap Overflow");
 return;
 }

 // initialize data into temp data field
 temp.data = x;

 // put top reference into temp link
 temp.link = top;

 // update top reference
```

```
 top = temp;
}

// Utility function to check if the stack is empty or
// not
public boolean isEmpty() { return top == null; }

// Utility function to return top element in a stack
public int peek()
{
 // check for empty stack
 if (!isEmpty()) {
 return top.data;
 }
 else {
 System.out.println("Stack is empty");
 return -1;
 }
}

// Utility function to pop top element from the stack
public void pop() // remove at the beginning
{
 // check for stack underflow
 if (top == null) {
 System.out.print("\nStack Underflow");
 return;
 }
}
```

```

// update the top pointer to point to the next node
top = (top).link;

}

public void display()
{
 // check for stack underflow
 if (top == null) {
 System.out.printf("\nStack Underflow");
 exit(1);
 }
 else {
 Node temp = top;
 while (temp != null) {

 // print node data
 System.out.print(temp.data);

 // assign temp link to temp
 temp = temp.link;
 if(temp != null)
 System.out.print(" -> ");
 }
 }
}

```

**Output**



44 -> 33 -> 22 -> 11

Top element is 44

22 -> 11

Top element is 22

### **Time Complexity**

$O(1)$ , for all `push()`, `pop()`, and `peek()`, as we are not performing any kind of traversal over the list. We perform all the operations through the current pointer only.

### **Auxiliary Space:**

$O(N)$ , where  $N$  is the size of the stack

### **Stack Library**

The Java collections framework has a class named `Stack` that provides the functionality of the stack data structure.

The `Stack` class extends the `Vector` class.



### Creating a Stack

In order to create a stack, we must import the `java.util.Stack` package first. Once we import the package, here is how we can create a stack in Java.



```
Stack<Type> stacks = new Stack<>();
```

Here, Type indicates the stack's type. For example,



```
// Create Integer type stack
```

```
Stack<Integer> stacks = new Stack<>();
```

```
// Create String type stack
```

```
Stack<String> stacks = new Stack<>();
```

## Stack Methods

Since Stack extends the Vector class, it inherits all the methods of Vector.

Besides these methods, the Stack class includes 5 more methods that distinguish it from Vector.

### push() Method

To add an element to the top of the stack, we use the push() method. For example,



```
import java.util.Stack;
```

```
class Main {
 public static void main(String[] args) {
 Stack<String> animals= new Stack<>();

 // Add elements to Stack
 animals.push("Dog");
 animals.push("Horse");
 animals.push("Cat");

 System.out.println("Stack: " + animals);
 }
}
```

```
/*
```

OUTPUT

```
Stack: [Dog, Horse, Cat]
```

```
*/
```

### pop() Method

To remove an element from the top of the stack, we use the pop() method. For example,



```
import java.util.Stack;

class Main {
 public static void main(String[] args) {
 Stack<String> animals= new Stack<>();

 // Add elements to Stack
 animals.push("Dog");
 animals.push("Horse");
 animals.push("Cat");

 System.out.println("Initial Stack: " + animals);

 // Remove element stacks
 String element = animals.pop();
 System.out.println("Removed Element: " + element);
 }
}

/*
OUTPUT
Initial Stack: [Dog, Horse, Cat]
Removed Element: Cat
*/
```

### peek() Method

The peek() method returns an object from the top of the stack. For example,



```
import java.util.Stack;
```

```

class Main {

 public static void main(String[] args) {
 Stack<String> animals= new Stack<>();

 // Add elements to Stack
 animals.push("Dog");
 animals.push("Horse");
 animals.push("Cat");
 System.out.println("Stack: " + animals);

 // Access element from the top
 String element = animals.peek();
 System.out.println("Element at top: " + element);

 }
}

/*
OUTPUT
Stack: [Dog, Horse, Cat]
Element at top: Cat
*/

```

### **search() Method**

To search an element in the stack, we use the search() method. It returns the position of the element from the top of the stack. For example,



```
import java.util.Stack;
```

```

class Main {
 public static void main(String[] args) {
 Stack<String> animals= new Stack<>();

 // Add elements to Stack
 animals.push("Dog");
 animals.push("Horse");
 animals.push("Cat");
 System.out.println("Stack: " + animals);

 // Search an element
 int position = animals.search("Horse");
 System.out.println("Position of Horse: " + position);
 }
}

/*
OUTPUT
Stack: [Dog, Horse, Cat]
Position of Horse: 2
*/

```

### **empty() Method**

To check whether a stack is empty or not, we use the `empty()` method. For example,



`import java.util.Stack;`

```

class Main {
 public static void main(String[] args) {

```

```

Stack<String> animals= new Stack<>();

// Add elements to Stack
animals.push("Dog");
animals.push("Horse");
animals.push("Cat");

System.out.println("Stack: " + animals);

// Check if stack is empty
boolean result = animals.empty();

System.out.println("Is the stack empty? " + result);
}

}

/*
OUTPUT
Stack: [Dog, Horse, Cat]
Is the stack empty? false
*/

```

### **Conclusion**

In the last two sessions, Stack data structure has been covered in depth.

### **Interview Questions**

- **What are some common scenarios where stacks are used?**

Following are some of the common scenarios in which Stacks can be Used.



- To check if delimiters (parenthesis, brackets etc.) are balanced in a computer program
- To reverse strings
- To traverse the nodes of a binary tree.

- To search the vertices of a graph.
- To process tasks
- To process messages

- **Reverse a String using Stack**

The followings are the steps to reversing a String using Stack:

1. String to char[].
2. Create a Stack.
3. **Push** all characters, one by one.
4. Then **Pop** all characters, one by one and put into the char[].
5. Finally, convert to the String.



```
public static String reverse(String str) {
 char[] charArr = str.toCharArray();
 int size = charArr.length;
 Stack stack = new Stack(size);

 int i;
 for (i = 0; i < size; ++i) {
 stack.push(charArr[i]);
 }

 for (i = 0; i < size; ++i) {
 charArr[i] = stack.pop();
 }

 return String.valueOf(charArr);
}
```

## **Additional Resources**

Tower of Hanoi :

[[https://docs.google.com/spreadsheets/d/1CaV2APUzFvNAgPpYsj6-L\\_LX6vS4-VUz0\\_H1vv-wnHQ/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1CaV2APUzFvNAgPpYsj6-L_LX6vS4-VUz0_H1vv-wnHQ/edit?usp=sharing)]

([https://docs.google.com/spreadsheets/d/1CaV2APUzFvNAgPpYsj6-L\\_LX6vS4-VUz0\\_H1vv-wnHQ/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1CaV2APUzFvNAgPpYsj6-L_LX6vS4-VUz0_H1vv-wnHQ/edit?usp=sharing))

Agenda :

- **Introduction**
- **Modular Arithmetic**
- **Modular Exponentiation**
- **Greatest Common Divisor**
- **Modular Multiplicative Inverse**
- **Primality Testing**
- **Sieve of Eratosthenes**

## **Introduction**

Many of the concepts of Mathematics form an integral part of the theory of Computer Science, and a good programmer should be well equipped to use those concepts in solving the problems encountered. In this section, we introduce you to some of the very basic ideas which you will find useful throughout your journey as a developer.

## **1. Modular Arithmetic**

Modular arithmetic is the branch of arithmetic mathematics related with the “mod” functionality. Basically, modular arithmetic is related with computation of “mod” of expressions. Expressions may have digits and computational symbols of addition, subtraction, multiplication, division or any other.

Let's begin with discussing about all modular arithmetic operations.

### **1.1. Quotient Remainder Theorem**

It states that, for any pair of integers a and b (b is positive), there exist two unique integers q and r such that :



$$*a = b*q + r*$$

\*where,  $0 \leq r < b$ \*

**For example :**



Let,  $a = 20$

$b = 7$

So we get,  $q = 2$ , and

$$r = 6$$

$$(20 = 7 \times 2 + 6)$$

In very simple terms, this theorem states that when any integer 'a' is divided by a positive integer 'b', we get an integer quotient 'q' and a remainder 'r' which lies in the range [0,b).

As you might be knowing already, the operator '%' is used in Java to calculate the remainder of the two numbers after division. For the given numbers, a and b, the expression (a%b) returns the remainder r, as can be seen here :



```
class Modular
{
 public static void main (String[] args)
 {
 int a = 20;
 int b = 7;
 System.out.println(a%b); // Prints 6.
 }
}
```

**Output :**



6

## 1.2. Modular Addition

The rule for modular addition is:

$$(a + b) \text{ mod } m = ((a \text{ mod } m) + (b \text{ mod } m)) \text{ mod } m$$

$$\text{In other words, } (a+b) \% m = ((a \% m) + (b \% m)) \% m$$

Note that it's simply not enough to apply the mod operator to both the numbers a and b, but we need to apply the operator once again after we have added the expressions (a mod m) and (b mod m). This could be a rookie mistake to make so one should keep that in mind.

**For Example:**



$$\begin{aligned}
 & (18 + 19) \% 7 \\
 &= ((18 \% 7) + (19 \% 7)) \% 7 \\
 &= (4 + 5) \% 7 \\
 &= 9 \% 7 \\
 &= 2
 \end{aligned}$$

The same rule is for modular subtraction. We don't require much modular subtraction but it can also be done in the same way.

### 1.3. Modular Multiplication

The rule for modular multiplication is:

$$(a * b) \bmod m = ((a \bmod m) * (b \bmod m)) \bmod m$$

$$\text{In other words, } (a * b) \% m = ((a \% m) * (b \% m)) \% m$$

Again, note how we need to apply the modular operator once again at the end.

**For Example:**



$$\begin{aligned}
 & (10 * 15) \% 6 \\
 &= ((10 \% 6) * (15 \% 6)) \% 6 \\
 &= (4 * 3) \% 6 \\
 &= 12 \% 6 \\
 &= 0
 \end{aligned}$$

Before we study some more concepts from Modular Arithmetic, it is necessary to cover a few other important concepts like GCD.

## 2. Greatest Common Divisor

The Greatest Common Divisor (GCD, also called the Highest Common Factor - HCF) of two or more integers is the largest integer that divides each of the integers such that their remainder is zero. For example, see the image :

## GCD or HCF of two numbers in C

$$24 - 2 \times 2 \times 2 \times 3$$

$$30 - 2 \times 3 \times 5$$

**GCD = Multiplication of common factors**

$$= 2 \times 3$$

$$= 6$$

Here, the GCD of 24 and 30 is 6 because it is the largest number which perfectly divides both 24 and 30.

Similarly, GCD of 42, 120, 285 = 3 (3 is the largest number which divides 42, 120 and 285 with remainder as 0).

### 2.1 Using Euclidean Algorithm

Now how do we find the GCD of two given numbers? One very basic approach could be to find all the factors of the two numbers, and then find the common ones for both the numbers and then choose the greatest of them. However, An efficient solution \*\*\*\*is to use **Euclidean Algorithm** which is the primary algorithm used for this purpose.

The idea here is that, *GCD of two numbers doesn't change if a smaller number is subtracted from a bigger number.*

For example, think over what is the GCD of 10 and 35? It is 5. And what is the GCD of 25 (i.e. 35-10) and 10? 5 it is. And the GCD of 15 and 10 is 5 too. So is the GCD of 5 and 10.

You can convince yourself by looking at more such examples.

Given below is the Java implementation of the algorithm which makes use of this property to calculate the GCD of two numbers :



```
class AlmaBetter {
```

```
 static int gcd(int a, int b)
```

```
{
```

```

if (a == 0)
 return b;
if (b == 0)
 return a;

// base case
if (a == b)
 return a;

// a is greater
if (a > b)
 return gcd(a-b, b);
return gcd(a, b-a);

}

public static void main(String[] args)
{
 int a = 44, b = 10;
 System.out.println("GCD of " + a + " and " + b + " is " + gcd(a, b));
}

```

**Output :**



GCD of 44 and 10 is 2

**Time Complexity:** O(min(a,b))

Now, instead of using Euclidean Algorithm by subtraction, a better approach using the Modulo operator exists. We don't perform subtraction here. we continuously divide the bigger number by the smaller number.

Basically, we take the two given numbers, say a and b, and then recursively call the function calculating the GCD as follows :



```
class AlmaBetter
{
 static int gcd(int a, int b)
 {
 if (b == 0)
 return a;
 return gcd(b, a % b);
 }

 public static void main(String[] args)
 {
 int a = 66, b = 420;
 System.out.println("GCD of " + a + " and " + b + " is " + gcd(a, b));
 }
}
```

**Output :**



GCD of 66 and 420 is 6

**Time Complexity:**  $O(\log(\min(a,b)))$

## 2.2 Extended Euclidean Algorithm

We have seen how the Euclidean Algorithm calculates the GCD of two given numbers using the modulo operator.

Extended Euclidean algorithm also finds integer coefficients  $x$  and  $y$  such that:



$$ax + by = \text{GCD}(a, b).$$

**For Example:**



Input:  $a = 30$ ,  $b = 20$

Output:  $\text{gcd} = 10$ ,  $x = 1$ ,  $y = -1$

(As  $30*1 + 20*(-1) = 10$ )

Input:  $a = 49$ ,  $b = 25$

Output:  $\text{gcd} = 1$ ,  $x = -1$ ,  $y = 2$

(Note that  $49*(-1) + 25*(2) = 1$ )

The integers  $x$  and  $y$  are called **Bézout coefficients** for  $(a, b)$ . And note that they are not unique. A pair of Bézout coefficients can be computed by the extended Euclidean algorithm which is exactly what we want to learn here.

### The Algorithm and It's Implementation

The extended Euclidean algorithm updates the results of  $\text{GCD}(a, b)$  using the results calculated by the recursive call  $\text{GCD}(b\%a, a)$ . Let values of  $x$  and  $y$  calculated by the recursive call be  $x_1$  and  $y_1$ . Then  $x$  and  $y$  are updated using the below expressions :



$$ax + by = \text{gcd}(a, b)$$

$$\text{gcd}(a, b) = \text{gcd}(b\%a, a)$$

$$\text{gcd}(b\%a, a) = (b\%a)x_1 + ay_1$$

$$ax + by = (b\%a)x_1 + ay_1$$

$$ax + by = (b - [b/a] * a)x_1 + ay_1$$

$$ax + by = a(y_1 - [b/a] * x_1) + bx_1$$

Comparing LHS and RHS,

$$x = y_1 - [b/a] * x_1$$

$$y = x_1$$

The Java implementation for the above approach is shown here :



```
class AlmaBetter {
 // extended Euclidean Algorithm

 public static int gcdExtended(int a, int b, int x, int y)
 {
 // Base Case
 if (a == 0) {
 x = 0;
 y = 1;
 return b;
 }

 int x1 = 1, y1 = 1; // To store results of recursive call

 int gcd = gcdExtended(b % a, a, x1, y1);

 // Update x and y using results of recursive
 // call
 x = y1 - (b / a) * x1;
 y = x1;

 return gcd;
 }

 public static void main(String[] args)
 {
 int x = 1, y = 1;
 int a = 35, b = 15;
 int g = gcdExtended(a, b, x, y);
 System.out.print("gcd(" + a + " , " + b + ") = " + g);
 }
}
```

}

**Output :**



$\text{gcd}(35, 15) = 5$

You might be wondering what do we need this pair  $(x,y)$  for. The answer will become clear to you after we finally come back to some more concepts from involving the Modulo Operator.

### 3. Modular Multiplicative Inverse

The modular multiplicative inverse for a number **a** under modulo **b** is an integer **x** such that:



$$a * x \equiv 1 \pmod{b}$$

Note: The value of **x** should be in the range  $\{1, 2, \dots, b-1\}$ , i.e., in the range of integer modulo **b**.

(Also note that **x** cannot be  $0$  as  $a0 \pmod{b}$  will never be 1).



Input:  $a = 9, M = 16$

Output: 9

Explanation: Since  $(9*9) \pmod{16} = 1$ , 9 is modulo inverse of 9 (under 16).

Note here that 25 also might seem as a valid output as  $(25*9) \pmod{16} = 1$ ,

but 25 is outside the range  $\{1, 2, \dots, 15\}$ , so it is not valid.

Input:  $A = 30, M = 7$

Output: 4

Explanation: Since  $(30*4) \pmod{7} = 1$ , 4 is modulo inverse of 30 (under 7).

#### Does the Modular Multiplicative Inverse always exist?

No. It does not.

The multiplicative inverse of “ $a$  modulo  $b$ ” exists if and only if  $a$  and  $b$  are relatively prime, that is they don’t have any factor in common other than 1, or we can also say, if  $\text{GCD}(a, b) = 1$ .

Why is this so? The proof for this is out of scope of the content, but you can have an insight to convince yourself that this has to be the case always. For example, for given integers  $a = 10$  and  $b = 15$ , (clearly  $a$  and  $b$  are not relatively prime as  $\text{gcd}(10, 15) = 5$ ), do you think we can find an integer  $x$ , such that :



$$10*x \bmod 15 = 1$$

No matter what value we choose for x, we can't find such an integer.

### 3.1 Finding the Modular Multiplicative Inverse

Recall the extended Euclidean algorithm which we learnt in the previous section. This algorithm is particularly useful when a and b are co-prime (or GCD is 1) to calculate the multiplicative inverse.

In the given equation,



$$a*x + b*y = \text{GCD}(a,b)$$

When  $\text{GCD}(a,b) = 1$ , then x is the modular multiplicative inverse of "a under modulo b", and y is the modular multiplicative inverse of "b under modulo a".

Consider the following Java implementation to find the modular inverse :



```
// Iterative Java program to find modular
// inverse using extended Euclidean algorithm
```

```
class AlmaBetter {
```

```
// Returns modulo inverse of a under modulo m using extended Euclidean Algorithm
// Here we have assumed that : a and m are coprimes, i.e, gcd(a, m) = 1
```

```
static int modInverse(int A, int M)
```

```
{
```

```
 int m0 = M;
```

```
 int y = 0, x = 1;
```

```
 if (M == 1)
```

```
 return 0;
```

```

while (A > 1) {
 // q is quotient
 int q = A / M;
 int t = M;

 // m is remainder now, the process is same as Euclidean's algorithm
 M = A % M;
 A = t;
 t = y;

 // Update x and y
 y = x - q * y;
 x = t;
}

// Make x positive
if (x < 0)
 x += m0;

return x;
}

public static void main(String args[])
{
 int A = 18, M = 25;

 // Function call
 System.out.println("Modular multiplicative Inverse is " + modInverse(A, M));
}

```

```
 }
}
```

**Output :**



Modular multiplicative Inverse is 7

Since  $18 * 7 = 126$ , and  $126 \% 25 = 1$ , we can say that 7 is the modular multiplicative inverse of 18 under modulo 25.

### 3.2 Modular Division

The modular division is totally different from modular addition, subtraction and multiplication which we have studied earlier. It also does not exist always.

First of all, you should remember that :



$(a / b) \text{ mod } m$  is not equal to  $((a \text{ mod } m) / (b \text{ mod } m)) \text{ mod } m$ .

Modular division is defined only when modular inverse of the divisor exists. We have already discussed when does the modular inverse exist for a number.

#### How to find modular division?

The task is to compute  $a/b$  under modulo  $m$  ( $a/b \% m$ ) -

1. First we check if inverse of  $b$  under modulo  $m$  exists or not, i.e. if  $\text{GCD}(b,m) = 1$  or not.
2. If the inverse doesn't exist, we say that the division is not defined.
3. Else we calculate the following



$$d = ((\text{inverse of } b \text{ under modulo } m) * (a)) \% m,$$

which is exactly what we want.

The following Java code calculates the Modular Division. First, it checks if the modular inverse of  $b$  under modulo  $m$  exists or not. If it does, it makes use of the method defined in the previous section to calculate that inverse. Then it returns the division as defined in the above mentioned equation.



```
class AlmaBetter {
```

```

static int gcd(int a,int b){

 if (b == 0){

 return a;

 }

 return gcd(b, a % b);

}

// Function to find modulo inverse of b. It returns
// -1 when inverse doesn't exist.

static int modInverse(int b,int m){

 int g = gcd(b, m) ;

 if (g != 1)

 return -1;

 else

 {

 int m0 = m;

 int y = 0, x = 1;

 if (m == 1)

 return 0;

 while (b > 1) {

 // q is quotient

 int q = b / m;

 int t = m;

 // m is remainder now, the process is same as Euclidean's algorithm

 m = b % m;

```

```

 b = t;
 t = y;

 // Update x and y
 y = x - q * y;
 x = t;
 }

 // Make x positive
 if (x < 0)
 x += m0;

 return x;
}

// Function to compute a/b under modulo m
static void modDivide(int a,int b,int m){
 a = a % m;
 int inv = modInverse(b,m);
 if(inv == -1){
 System.out.println("Division not defined");
 }
 else{
 System.out.println("Result of Division is " + ((inv*a) % m));
 }
}

```

```
public static void main(String[] args) {
 int a = 2;
 int b = 9;
 int m = 16;
 modDivide(a, b, m);
}
}
```

**Output :**



Result of Division is 1

#### **4. Modular Exponentiation**

Given three numbers a, b and c, we need to find  $a^b \% c$ .

Now why do we often need to have “% c” after exponentiation? This is because  $a^b$  could become really large even for relatively small values of a, b and that is often a problem because the data type of the language that we try to code the problem, will most probably not let us store such a large number.

**Examples:**



Input : a = 234 b = 511 c = 100

Output : 84

(Of course,

Input : a = 8 b = 7 c = 40

Output : 32

The idea to calculate the modular exponentiation is based on below properties.

#### **Property 1: Modular Multiplication**

$$(m * n) \% p = ((m \% p) * (n \% p)) \% p$$

#### **Property 2:**

If b is even:

$$(a \wedge b) \% c = ((a \wedge b/2) * (a \wedge b/2)) \% c$$

If b is odd:

$$(a \wedge b) \% c = (a * (a \wedge (b-1))) \% c$$

### Property 3:

If we have to return the mod of a negative number x whose absolute value is less than y:

then  $(x + y) \% y$  will do the trick

### Note:

Also as the product of  $(a \wedge b/2) * (a \wedge b/2)$  and  $a * (a \wedge (b-1))$  may cause overflow, hence we must be careful about those scenarios.

Using these properties, we calculate the Modular Exponentiation  $a^{b \% c}$  as shown below :



```
class AlmaBetter
{
 static int exponentMod(int A, int B, int C)
 {
 // Base cases
 if (A == 0)
 return 0;
 if (B == 0)
 return 1;

 // If B is even
 long y;
 if (B % 2 == 0)
 {
 y = exponentMod(A, B / 2, C);
 y = (y * y) % C;
 }
 }
}
```

```

// If B is odd
else
{
 y = A % C;
 y = (y * exponentMod(A, B - 1, C) % C) % C;
}

return (int)((y + C) % C);
}

```

```

public static void main(String args[])
{
 int A = 234, B = 511, C = 100;
 System.out.println("Power is " + exponentMod(A, B, C));
}
}

```

**Output :**



Power is 84

## 5. Primality Testing

We know what a prime number is. It is a natural number greater than 1 which cannot be divided by any number (except 1 and itself). Examples of the first few prime numbers are {2, 3, 5, 7, 11, ...}

$\sum Kx$   $\sin x p \times \sin x$   $\Delta S \alpha = 1$   $\sin x$   
 $P(x=k) = \binom{n}{k} p^k q^{n-k}$   $(t = \cos x)$   $\sqrt{3} x = \cos i + \tan x \cdot \frac{1}{4} q$   $\Delta x$   
 $\frac{1}{12} \int_{\pi}^{2\pi} dx$   $S = x^2$   $(n+1)$   $x = 5$   $y = x + D$   
 $E(x) = \sum_{x=0}^{\infty} n e^x - p(x^2 - p)(x = k^2)$   $x = 1$   $\sum_{Y=0}^L np \times 1 = \alpha$   
 $\sum_{x=0}^{\infty} \sin^2 x = 3\pi$   $\lim_{x \rightarrow 0} \frac{dx}{\cos^2 x}$   $y < \sqrt{2x + \sqrt{x}}$   $y = \cos x^2$   $\Delta S$   
 $A^2 x q^2 + B^2$   $\lim_{x \rightarrow 0} \alpha$   $\frac{\sqrt{6}}{2} + \frac{3}{2} \sqrt{\frac{6}{5}} \left( \frac{\pi}{10} - \frac{1}{3} \right) = 1$   
 $\lim_{x \rightarrow 2} \frac{\sin x - \sqrt{2x - 4}}{x^3 - (3+1)x^1 - 2^3}$   $\lim_{x \rightarrow 2} \sqrt{2x}$   $\Delta S = (S + C^3) \frac{2+3}{2+3} \alpha$   
 $\left(\frac{1}{2}\right)^{-x} = 1$   $\lim_{x \rightarrow R} \frac{\cos x}{\sin x} > B^2 \frac{\pi}{2\pi} \alpha$   $C = \lim_{\alpha \rightarrow 0} -q_0 + q_1$   
 $12\pi^3 = \sin x$   $\lim_{x \rightarrow 0} \frac{\sin x}{x} = 1$   $R = \frac{S_0 R_0^3}{R_0 + P_0} + \frac{3 S_0 R_0 + H}{R_0 + P_0 n^2} \alpha^0$   
 $\sqrt{c} = \frac{5\pi^2}{2\pi^2} \tan x$   $\alpha^2 = x^2$   $K = \sqrt{q R^2} = \sqrt{q} p^3$   
 $\log \frac{x}{y} = \log z$   $\alpha^3 = x^3$   $e = VR^2$   $S = 3$   
 $(\cos x) = \cos(z)$   $\lim_{x \rightarrow 0} \frac{\sin x}{x} = 1$   $P = 6$   
 $1 = \sin dy^3$   $\sum_{m=0}^{\infty} K x^m \left[ \begin{matrix} p \\ q \end{matrix} \right] \sin^{p+q} x$   $N p \times 1 = N p$   
 $O_2 = \int \cos x dx$   $\sum_{m=0}^{\infty} \frac{dt}{x^m} - \alpha x \sin x$   $\frac{1}{3} q \sum_{p=0}^{\infty} \left( c = T q^2 \right)$   
 $2 - \sin^2 x$   $\frac{1}{1+2x} \frac{1}{1} \frac{C^{2-2p}}{2}$   $\lim_{K=0} \frac{K}{K+1} \frac{du^3}{dx}$   
 $= np \sum_{i=0}^{\infty} \left[ \begin{matrix} x = 1 \\ \lim \end{matrix} \right] c^2 + x (-1) = \cos p x^2$   $\int \tan^2 x dx$   
 $B^2 S = 3$   $\alpha = PK PG$   $- \int \sin \frac{1}{2} dt [3^{\frac{1}{2}}] c x$

**Task :** Given a positive integer, check if the number is prime or not.

A simple solution is to iterate through all numbers from 2 to  $(n-1)$  and for every number check if it divides  $n$ . If we find any number that divides, we return false.

However, we can do better. Think for a moment whether we need to check all the way upto  $(n-1)$ . We actually don't! We can check till  $\sqrt{n}$  because a larger factor of  $n$  must be a multiple of a smaller factor that has been already checked. The implementation of this method is as follows:



```
class AlmaBetter {
```

```
static boolean isPrime(int n)
```

{

// Corner case

if ( $n \leq 1$ )

```

 return false;

 // Check from 2 to square root of n
 for (int i = 2; i * i <= n; i++)
 if (n % i == 0)
 return false;

 return true;
}

public static void main(String args[])
{
 if (isPrime(6))
 System.out.println(" true");
 else
 System.out.println(" false");
 if (isPrime(19))
 System.out.println(" true");
 else
 System.out.println(" false");
}
}

```

**Output**



false

true

**Time Complexity:**  $\sqrt{n}$

**Auxiliary Space:** O(1)

## 6. Sieve of Eratosthenes

Sieve of Eratosthenes is a simple and ancient algorithm used to find the prime numbers up to any given limit. It is one of the most efficient ways to find small prime numbers.

For a given upper limit ' $n$ ' the algorithm works by iteratively marking the multiples of primes as composite, starting from 2. Once all multiples of 2 have been marked composite, the multiples of next prime, i.e. 3 are marked composite. This process continues until  $p \leq \sqrt{n}$  where  $p$  is a prime number.

If we follow this process for  $n=100$ , we would be left with all the prime numbers less than 100, as can be seen in the following image :

|    |    |    |    |    |    |    |    |    |     |
|----|----|----|----|----|----|----|----|----|-----|
|    | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10  |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20  |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30  |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40  |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50  |
| 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60  |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70  |
| 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80  |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90  |
| 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 |

### Implementation in Detail

In the following algorithm, the number 0 represents a composite number.

1. To find out all primes under  $n$ , generate a list of all integers from 2 to  $n$ . (**Note:** 1 is not prime)
2. Start with the smallest prime number, i.e.  $p=2$ .
3. Mark all the multiples of  $p$  which are less than  $n$  as composite. To do this, mark the value of the numbers (multiples of  $p$ ) in the generated list as 0. **Do not** mark  $p$  itself as composite.
4. Assign the value of  $p$  to the next prime. The next prime is the next non-zero number in the list which is greater than  $p$ .
5. Repeat the process until  $p \leq \sqrt{n}$  .

We are done!

Now all **non-zero** numbers in the list represent primes, while the numbers which are 0 in the list represent composite numbers.

The Java implementation is as follows :



```
class SieveOfEratosthenes
{
 void sieveOfEratosthenes(int n)
 {
 // Create a boolean array "prime[0..n]" and initialize
 // all entries in it as true. A value in prime[i] will
 // finally be false if i is Not a prime, else true.

 boolean prime[] = new boolean[n+1];
 for(int i=0;i<=n;i++)
 prime[i] = true;

 for(int p = 2; p*p <=n; p++)
 {
 // If prime[p] is not changed, then it is a prime
 if(prime[p] == true)
 {
 // Update all multiples of p
 for(int i = p*p; i <= n; i += p)
 prime[i] = false;
 }
 }

 // Print all prime numbers
 for(int i = 2; i <= n; i++)
 {
 if(prime[i] == true)
```

```

 System.out.print(i + " ");
 }

}

public static void main(String args[])
{
 int n = 50;
 System.out.print("Following are the prime numbers ");
 System.out.println("smaller than or equal to " + n);
 SieveOfEratosthenes g = new SieveOfEratosthenes();
 g.sieveOfEratosthenes(n);
}

```

**Output :**



Following are the prime numbers smaller than or equal to 50

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

**Interview Questions**

**1. How do you check if a number is prime or not?**

A simple solution is to iterate through all numbers from 2 to n-1 and for every number check if it divides n. If we find any number that divides, we return false.

In a more efficient approach, instead of checking till n, we can check till  $\sqrt{n}$  because a larger factor of n must be a multiple of a smaller factor that has been already checked.

**2. Why does the Euclidean algorithm for calculating the GCD of given numbers work?**

The Euclidean algorithm for calculation of GCD is based on the following properties -

- If we subtract a smaller number from a larger one (we reduce a larger number), GCD doesn't change. So if we keep subtracting repeatedly the larger of two, we end up with GCD.
  - Now instead of subtraction, if we divide the smaller number, the algorithm stops when we find the remainder 0.
- 3. Explain the Modular Multiplicative Inverse.**

The modular multiplicative inverse for a number **a** under modulo **b** is an integer **x** such that  $a * x \equiv 1 \pmod{b}$ . It does not always exist for all possible pairs (a,b). If a and b are co-prime, only then the Modular Multiplicative Inverse of a under modulo b can be calculated.

### **Additional Resources**

1. Relation between LCM and GCD for two numbers :

[<https://demonstrations.wolfram.com/LeastCommonMultipleAndGreatestCommonDivisor/>]

2. Find numbers from 1 to N with exactly 3 divisors :

[<https://www.geeksforgeeks.org/numbers-exactly-3-divisors/>]

3. Mod of negative numbers in Java :

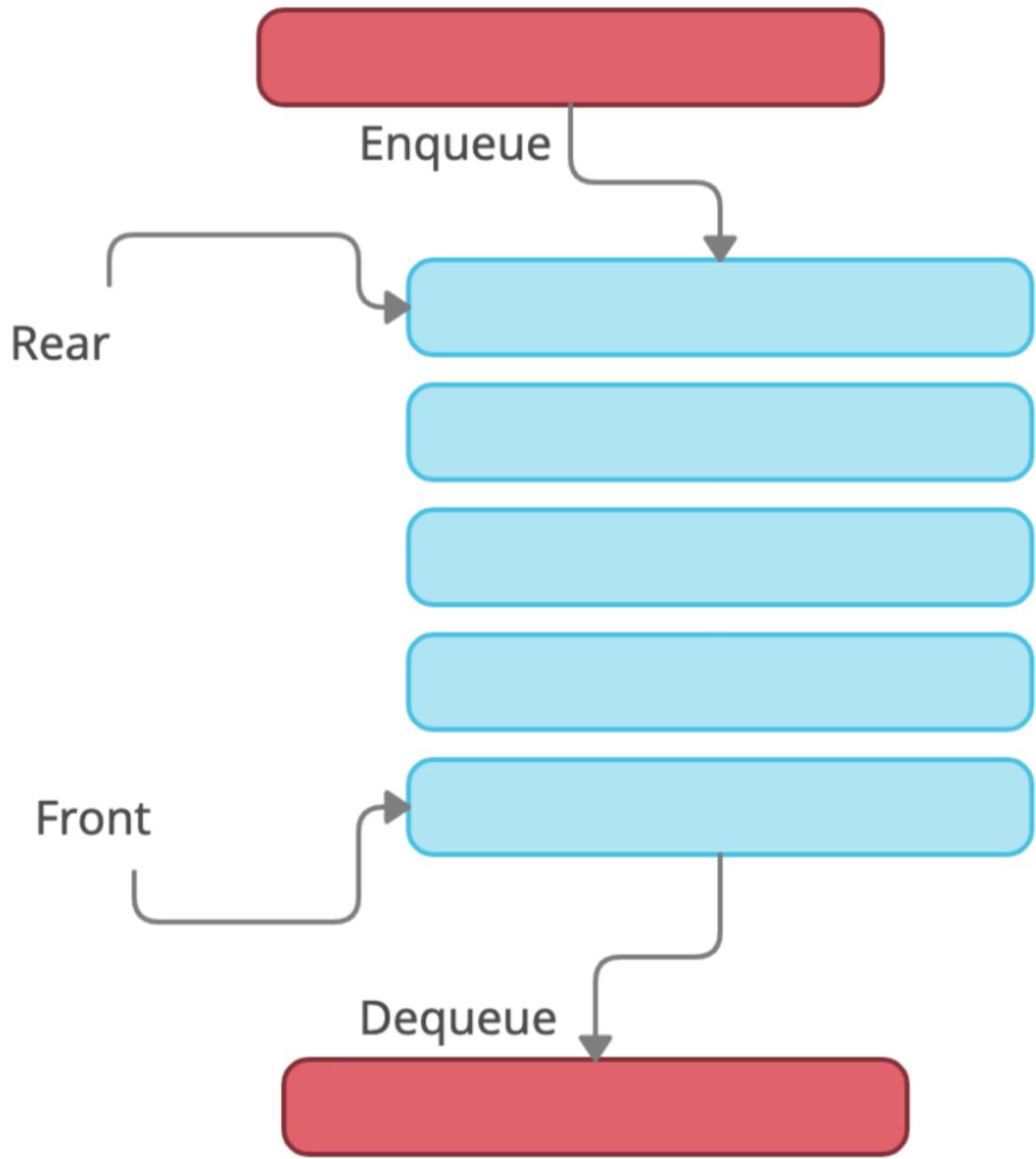
[<https://www.delftstack.com/howto/java/mod-of-negative-numbers-in-java/>]

## **Agenda**

- Intro to Queue
- Application of Queue
- Algorithms required to implement queue using Array
- Queue implementation using Array

## **Introduction**

A queue is a linear data structure that follows an order in which the elements can be accessed. It is very similar to stacks, but the only difference is that a queue is open on both ends. One end is used to add elements, and the other end is used to remove elements. The technical term for adding and removing elements is called ***Enqueue*** and ***Dequeue***, respectively.



The exciting part of queues is that we cannot operate upon each element present in the queue. Only the front and the rear of the queue can be accessed or operated upon.

A queue data structure is a First In First Out (FIFO) data structure. Items in the queue are retrieved in the order in which they are inserted.

#### **Basic features of Queue:**

1. Like stack, queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO( First in First Out ) structure.

3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. `peek()` function is often used to return the value of first element without dequeuing it.

### Applications of Queue

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served
4. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
5. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
6. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
7. Queues are used in operating systems for handling interrupts.

### Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

- `enqueue()` – add (store) an item to the queue.
- `dequeue()` – remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are –

- `peek()` – Gets the element at the front of the queue without removing it.
- `isfull()` – Checks if the queue is full.
- `isempty()` – Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue –

#### `peek()`

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows –



```
begin procedure peek
 return queue[front]
end procedure
isfull()
```

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function –



```
begin procedure isfull

if rear equals to MAXSIZE
 return true
else
 return false
endif
```

end procedure

**isempty()**

Algorithm of isempty() function –



```
begin procedure isempty

if front is less than MIN OR front is greater than rear
 return true
else
```

```
 return false
endif
```

```
end procedure
```

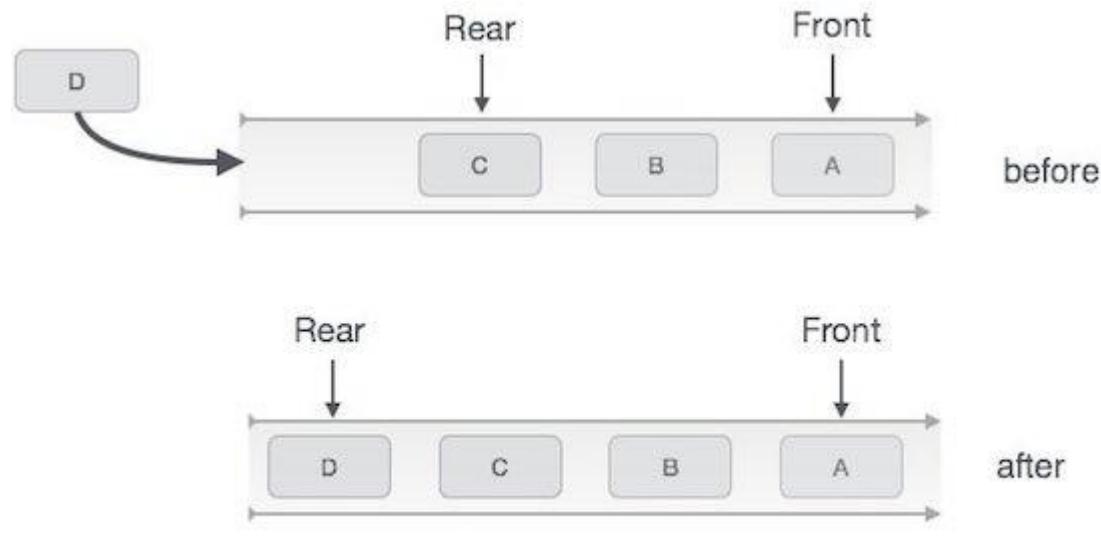
If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

### Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue –

- **Step 1** – Check if the queue is full.
- **Step 2** – If the queue is full, produce overflow error and exit.
- **Step 3** – If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** – Add data element to the queue location, where the rear is pointing.
- **Step 5** – return success.



## Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

### Algorithm for enqueue operation



```
procedure enqueue(data)
```

```
if queue is full
 return overflow
endif
```

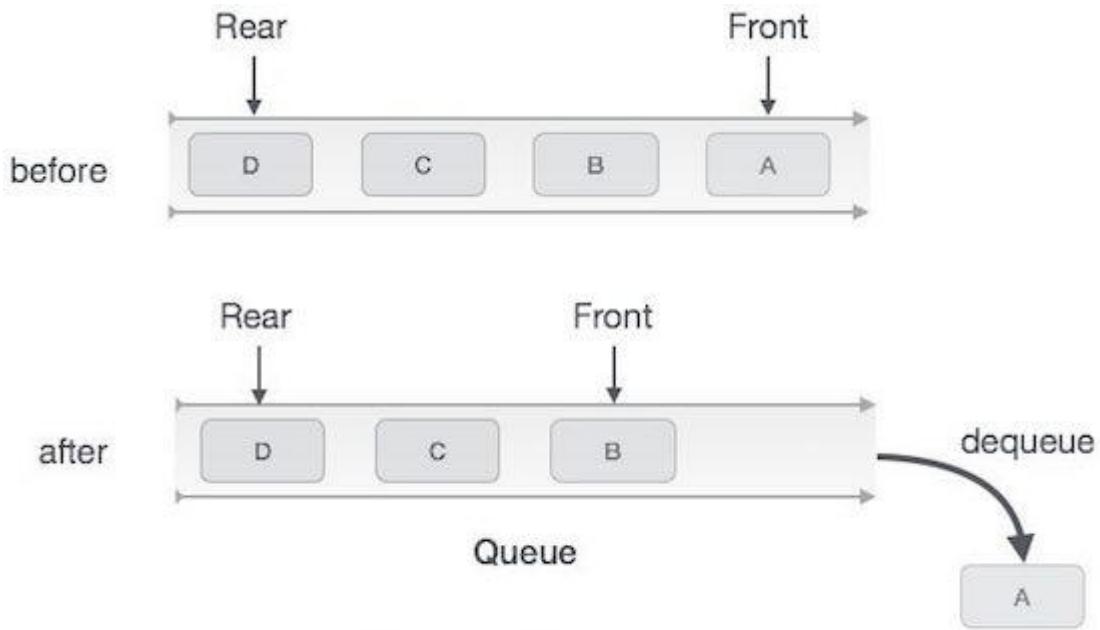
```
rear ← rear + 1
queue[rear] ← data
return true
```

```
end procedure
```

### Dequeue Operation

Accessing data from the queue is a process of two tasks – access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation –

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.
- **Step 5** – Return success.



#### **Algorithm for dequeue operation**



procedure dequeue

if queue is empty

    return underflow

end if

data = queue[front]

front  $\leftarrow$  front + 1

return true

end procedure

#### **Implementation of Queue using Array**



```
public class DemoQueue {
```

```
/* Member variable declaration */

private int maxSize;

private int[] queueArray;

private int front;

private int rear;

private int currentSize;

/* Constructor */

public DemoQueue(int size) {

 this.maxSize = size;

 this.queueArray = new int[size];

 front = 0;

 rear = -1;

 currentSize = 0;

}

/* Queue:Insert Operation */

public void insert(int item) {

 /* Checks whether the queue is full or not */

 if (isQueueFull()) {

 System.out.println("Queue is full!");

 return;

 }

 if (rear == maxSize - 1) {

 rear = -1;

 }

 /* increment rear then insert element to queue */

 queueArray[++rear] = item;

 currentSize++;

 System.out.println("Item added to queue: " + item);

}
```

```
/* Queue>Delete Operation */
public int delete() {
 /* Checks whether the queue is empty or not */
 if (isQueueEmpty()) {
 throw new RuntimeException("Queue is empty");
 }
 /* retrieve queue element then increment */
 int temp = queueArray[front++];
 if (front == maxSize) {
 front = 0;
 }
 currentSize--;
 return temp;
}

/* Queue:Peek Operation */
public int peek() {
 return queueArray[front];
}

/* Queue:isFull Operation */
public boolean isQueueFull() {
 return (maxSize == currentSize);
}

/* Queue:isEmpty Operation */
public boolean isQueueEmpty() {
 return (currentSize == 0);
}

/* Driver Code */
public static void main(String[] args) {
 DemoQueue queue = new DemoQueue(10);
```

```

queue.insert(2);
queue.insert(3);
System.out.println("Item deleted from queue: " + queue.delete());
System.out.println("Item deleted from queue: " + queue.delete());
queue.insert(5);
System.out.println("Item deleted from queue: " + queue.delete());
}

}

/*

```

## OUTPUT

```

Item added to queue: 2
Item added to queue: 3
Item deleted from queue: 2
Item deleted from queue: 3
Item added to queue: 5
Item deleted from queue: 5

```

```

*/

```

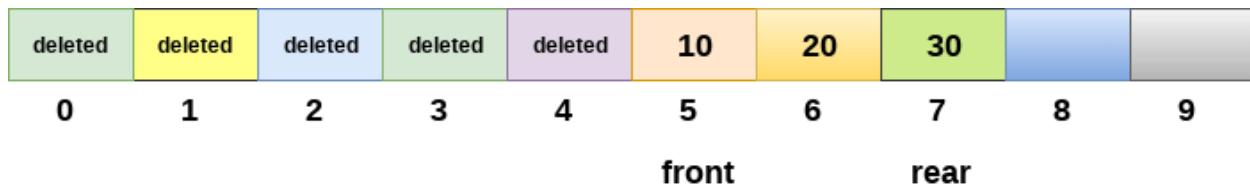
### Time Complexity

According to the queue data structure theory, all operations' time complexity should be  **$O(1)$** . For those who do not know what this means,  **$O(1)$**  is the most efficient time complexity a data structure or an algorithm can have.  **$O(1)$**  tells that, even if the number of elements in the queue increase, the time taken for that operation to complete will be constant.

### Drawbacks of Array Implementation

Although, the technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.

- **Memory wastage** : The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.



## limitation of array representation of queue

The above figure shows how the memory space is wasted in the array representation of queue. In the above figure, a queue of size 10 having 3 elements, is shown. The value of the front variable is 5, therefore, we can not reinsert the values in the place of already deleted element before the position of front. That much space of the array is wasted and can not be used in the future (for this queue).

- **Deciding the array size**

On of the most common problem with array implementation is the size of the array which requires to be declared in advance. Due to the fact that, the queue can be extended at runtime depending upon the problem, the extension in the array size is a time taking process and almost impossible to be performed at runtime since a lot of reallocations take place. Due to this reason, we can declare the array large enough so that we can store queue elements as enough as possible but the main problem with this declaration is that, most of the array slots (nearly half) can never be reused. It will again lead to memory wastage.

### Interview Questions

- **Difference between Stack and Queue Data Structure in Java?**

Stack is a LIFO data structure which means items are removed in the reverse order of their insertion, while a Queue is a FIFO data structure which means items are removed in the same order of their insertion.

- **Why don't we perform operations upon each element of a queue?**

If we can perform operations upon each element of a queue, then there is just no point of queues. Queues then just become an array. In an array, each element can be accessed directly and be operated upon. The application of queues are different compared to that of an array.

- **Why and when should I use Stack or Queue data structures instead of Arrays/Lists?**

Because they help manage your data in more a *particular* way than arrays and lists. It means that when you're debugging a problem, you won't have to wonder if someone randomly inserted an element into the middle of your list, messing up some invariants.

Arrays and lists are random access. They are very flexible and also easily *corruptible*. If you want to manage your data as FIFO or LIFO it's best to use those, already implemented, collections.

More practically you should:

- Use a queue when you want to get things out in the order that you put them in (FIFO)
- Use a stack when you want to get things out in the reverse order than you put them in (LIFO)
- Use a list when you want to get anything out, regardless of when you put them in (and when you don't want them to automatically be removed).

### **Additional Resources**

- Types of Queue : <https://www.programiz.com/dsa/types-of-queue>
- Circular Queue : <https://www.programiz.com/dsa/circular-queue>