Java Summary Cheat Sheet - 1

https://www.ccbp.in/

Data Types in Java

In programming languages, every value or data has an associated type known as data type. Java supports various data types. These data types are categorized into,

1. Primitive Data Types
2. Non-Primitive Data Types

Primitive Data Types Primitive data types are those that are predefined by the programming language (Java).
boolean: In general, anything that can take one of two possible values is considered a `boolean`. In Java, `true` and `false` are considered `boolean` values.

```
boolean canLearn = true;
```

byte: The `byte` data type is used to store integers without any fractional part whose values are in the range -128 to 127.

```
byte points = 100;
```

short : The `short` data type is used to store integers without any fractional part whose values are in the range -32,768 to 32,767.

```
short number = 28745;
```

int: The `int` data type is used to store integers without any fractional part whose values are in the range -2^{31} to $2^{31}-1$.

```
int distance = 2036372;
```

long: The `long` data type is used to store integers without any fractional part whose values are in the range -2^{63} to $2^{63}-1$. The long values should contain the suffix 'l' or 'L'.

```
long area = 2036372549999L;
```

float: The `float` data type is used to store any number with a decimal point. The float data type stores a value up to 7 point precision (ex: 12.1234567). The float values should contain the suffix 'f' or 'F'.

```
float height = 5.10f;
```

double: The `double` data type is used to store any number with a decimal point. The double data type stores a value up to 16 point precision. The double values may contain the suffix 'd' or 'D'.

```
double breadth = 9.2345D;
```

Non-Primitive Data Types These data types are used to store multiple values. Non-primitive data types are defined by the programmer. In Java programming, all non-primitive data types are simply called objects. Some commonly used non-primitive data types are,

- String
- Array
- Class

String: The `String` data type is simply a sequence of characters. In Java, double quotes (") are used to represent a string.

```
String name = "Rahul";
```

Array: In Java, an `array` is an object used to store similar data type values. In Java, the number of elements that an array can store is always fixed.

```
int[] arr = {60, 25, 20, 15, 30};
```

Conditional Statements

Conditional Statement: Conditional Statement allows us to execute a block of code only when a specific condition is true.

If...Else Statements: When an `if...else` conditional statement is used, the `if` block of code executes when the condition is true, otherwise the `else` block of code is executed.

```
int token = 20;
if (token == 20)
    System.out.println("Collect Water Bottle");
else
    System.out.println("Invalid Token");
// Output is:
Collect Water Bottle
```

Else if Statement: Java provides an `else if` statement to have multiple conditional statements between `if` and `else`. The `else if` statement is optional. We can add any number of `else if` statements after `if` conditional block.

```
int token = 20;
if (token == 10)
    System.out.println("Collect Chips");
else if(token == 20)
    System.out.println("Collect Soft Drink");
else
    System.out.println("Invalid Token");
// Output is:
Collect Soft Drink
```

Switch: A `switch` block can have multiple `case` or `default` labels. The `switch` statement allows us to execute a block of code among many cases.

```
switch (100 / 10) {
    case 10:
        System.out.println("Ten");
        break;
    case 20:
        System.out.println("Twenty");
        break;
    default:
        System.out.println("Other Number");
        break;
}
// Output is:
Ten
```

Nested Conditions: The conditional statements inside another conditional statement is called a nested conditional statement.

```
int token = 30;
char softDrink = 'S';
if (token == 30) {
    if (softDrink == 'S')
        System.out.println("Collect Sprite");
    else if (softDrink == 'F')
        System.out.println("Collect Fanta");
    else
        System.out.println("Invalid Option");
```

```
}
else
    System.out.println("Invalid Token");
// Output is:
 Collect Sprite
```

String - working with strings

Creating Strings:

- **Using String Literals**: A `string` literal is a value which is enclosed in double quotes (" ").

```
String str = "ABC";
```

- **Using new Keyword**: A string can be created by initializing the `String` class with the `new` keyword.

```
String str1 = new String(str);
```

String Concatenation: The `concat()` method appends one string to the end of another string. Using this method we can concatenate only two strings.

```
String concatenatedStr = "Hello ".concat("World");
System.out.println(concatenatedStr);  // Hello World
System.out.println("Hello " + "World");  // Hello World
```

Length of String: The string object has a `length()` method that returns the number of characters in a given string.

```
String name = "Rahul";
int strLength = name.length();
System.out.println(strLength); // 5
```

String Indexing: We can access an individual character in a string using their positions. These positions are also called indexes. The `charAt()` method is used to get the character at a specified index in a string.

```
String name = "Rahul";
char firstLetter = name.charAt(0);
System.out.println(firstLetter); // R
```

String Slicing: Obtaining a part of a string is called string slicing. The `substring()` method can be used to access a part of the string.

```
String message = "Welcome to Java";
```

```
String part = message.substring(0, 5);
System.out.println(part); // Welco
```

Slicing to end

```
String message = "Welcome to Java";
String part = message.substring(11);
System.out.println(part); // Java
```

String Repetition: The `repeat()` method returns a string whose value is the concatenation of the given string repeated N times. If the string is empty or the count is zero then the empty string is returned.

```
String str = "Rahul";
System.out.println(str.repeat(2)); // RahulRahul
```

Calculations in Java

Addition: Addition is denoted by `+` sign. It gives the sum of two numbers.

```
System.out.println(2 + 5); // 7
System.out.println(1 + 1.5); // 2.5
```

Subtraction: Subtraction is denoted by `-` sign. It gives the difference between the two numbers.

```
System.out.println(5 - 2); // 3
```

Multiplication: Multiplication is denoted by `*` sign. It gives the product of two numbers.

```
System.out.println(5 * 2); // 10
System.out.println(5 * 0.5); // 2.5
```

Division: Division is denoted by `/` sign. It returns the quotient as a result.

```
System.out.println(5 / 2); // 2
System.out.println(4 / 2); // 2
```

Modulo operation: Modulo is denoted by `%` sign. It returns the modulus (remainder) as a result.

```
System.out.println(5 % 2); // 1
System.out.println(4 % 2); // 0
```

Input and Output Basics

Take Input From User: The `Scanner` class in the package `java.util` is used to read user input.

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

```
class Main {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        String username = input.nextLine();
        System.out.println(username);
        input.close();
    }
}
```

Following are the methods, used to read different types of inputs from the user:

Method	Description
nextInt()	Reads an int value
nextLong()	Reads a long value
nextFloat()	Reads a float value
nextBoolean()	Reads a boolean value
next()	Reads a String value only until a space(" ") is encountered
nextLine()	Reads a String value till the end of line
nextDouble()	Reads a double value
nextShort()	Reads a short value
nextByte()	Reads a byte value

Printing the Output In Java, we have different methods available to print the output to the console.
The `print()` accepts a value as a parameter and prints text on the console. It prints the result in the same line.

```
System.out.print("Hello ");
System.out.print("Rahul");
// Output is:
Hello Rahul
```

The `println()` accepts a value as a parameter and prints text on the console. It prints the result in the new line.

```
System.out.println("Hello");
System.out.println("Rahul");
// Output is:
Hello
Rahul
```

Comments Single-line comments start with two forward slashes `//`

```
// This is a comment
```

Multi-line comments start with `/*` and end with `*/`. In between these, we can write any number of statements.

```
/* This is a
 Multi-line Comment */
```

String Methods

Method	Syntax	Usage
trim()	str.trim();	removes all the leading and trailing spaces of the given string and returns a new string.
toLowerCase()	str.toLowerCase();	converts each character of the given string to lowercase and returns a new string.
toUpperCase()	str.toUpperCase();	converts each character of the given string to uppercase and returns a new string.
startsWith()	str.startsWith(value);	returns true if the given string starts with the specified value. Otherwise, false is returned.
endsWith()	str.endsWith(value);	returns true if the given string ends with the specified value. Otherwise, false is returned.
replace()	str.replace(old, latest);	replaces all the occurrences of the old character/substring with the latest character/substring and returns a new string.
replaceFirst()	str.replaceFirst(old, latest);	returns a new string after replacing the first occurrence of the old substring with the latest substring.
split()	str.split(separator);	used to split the string at the specified separator. It returns an array of substrings.
join()	String.join(delimiter, str1, str2, ...);	joins the given elements with the specified delimiter and returns a new string.
equals()	str1.equals(str2);	It returns true if the given strings are equal. Otherwise false is returned.
equalsIgnoreCase()	str1.equalsIgnoreCase(str2);	It works similar to equals(), but ignores the case difference between the strings.

Method	Syntax	Usage
compareTo()	str1.compareTo(str2);	used to compare two strings based on the Unicode values.
compareToIgnoreCase()	str1.compareToIgnoreCase(str2);	It works similar to the compareTo() but ignores the case difference between the strings.

String Formatting As Java is a statically typed language, it requires the data type of value to be specified that is going to be formatted. This is done with the help of format specifiers.
The `format specifiers` define the `type` of data that is going to be used.
Below are the commonly used format specifiers:

Specifier	Description
%s, %S	Used for string values
%b, %B	Used for boolean values
%c, %C	Used for characters
%d	Used for integers i.e., int, long etc.
%f	Used for floating-point values like float and double
%e, %E	Used for a scientific notation for floating-point values

String.format() Method: In Java, string formatting can be done using the `format()` method of the `String` class. This method returns the formatted string.

```
String name = "James";
int age = 20;
String formattedStr;
formattedStr = String.format("%s is %d years old", name, age);
System.out.println(formattedStr);
// Output is:
James is 20 years old
```

printf() Method: The `System.out.printf()` method can be used to directly format and print the string.

```
String name = "James";
int age = 20;
System.out.printf("%s is %d years old", name, age);
// Output is:
James is 20 years old
```

Formatting Decimal Numbers: For floating-point numbers, we can specify the number of digits after the decimal point needs to be formatted.

```
float runRate = 10.2564f;
System.out.printf("Actual Run rate: %f\n", runRate);
System.out.printf("Run rate rounded to 2 decimals: %.2f", runRate);
// Output is:
Actual Run rate: 10.256400
Run rate rounded to 2 decimals: 10.26
```

DecimalFormat Class: The `DecimalFormat` class from the package `java.text` used to format decimal numbers. The `format()` method takes a `double` or `long` value and returns a `String` value. If `float` or `int` are data types are passed, they are implicitly converted to `double` and `long` respectively.

```
double doubleNum = 1.666;
DecimalFormat df = new DecimalFormat("#.##");
System.out.println(df.format(doubleNum)); // 1.67
```

Numbering Format Specifiers: We can provide an argument index for the format specifiers. It indicates which argument to be inserted at its position.

```
String name = "James";
int age = 20;
System.out.printf("%2$s is %1$d years old", age, name);
// Output is:
James is 20 years old
```

Character Methods

Method	Syntax	Usage
isLetter()	Character.isLetter(value);	checks whether the given character is an alphabet and returns a boolean value.
isDigit()	Character.isDigit(value);	checks whether the given character is a number/digit and returns a boolean value.
isWhiteSpace()	Character.isWhitespace(value);	checks whether the given character is a whitespace and returns a boolean value.
isUpperCase()	Character.isUpperCase(value);	checks whether the given character is an uppercase letter and returns a boolean value.
isLowerCase()	Character.isLowerCase(value);	checks whether the given character is a lowercase letter and returns a boolean value.
toUpperCase()	Character.toUpperCase(value);	converts the given lowercase letter to uppercase and returns the new character.

Method	Syntax	Usage
toLowerCase()	Character.toLowerCase(value);	converts the uppercase letter to lowercase and returns the new character.
toString()	Character.toString(value);	converts the character into the respective string.

Relational & Logical Operators

Relational Operators are used to compare values. It returns `true` or `false` as the result of a comparison.

Operator	Name	Example	Output
>	Is greater than	2 > 1	true
<	Is less than	5 < 10	true
==	Is equal to	3 == 4	false
<=	Is less than or equal to	2 <= 1	false
>=	Is greater than or equal to	2 >= 1	true
!=	Is not equal to	2 != 1	true

Logical operators are used to perform logical operations on `boolean` values. These will also produce a `true` or `false` as a result.

Name	Example	Output
&&	(5 < 10) && (1 < 2)	true
||	(5 < 10) || (2 < 2)	true
!	!(2 < 3)	false

Logical Operators Truth Table:

A	B	A && B
true	true	true
true	false	false
false	false	false
false	true	false

A	B	A || B
true	true	true
true	false	true
false	false	false
false	true	true

A	!A
true	false
false	true

Ternary Operator

Ternary Operator: `Ternary Operator` is a conditional operator which works similar to `if...else` statements.

```
int num1 = 345
int num2 = 689
int largest = num1 > num2 ? num1 : num2 ;
System.out.println(largest); // 689
```

Nested Ternary Operator: A `Ternary operator` can be used inside another ternary operator. It is called the nested ternary operator.

```
int a = 345
int b = 689
int c = 986
int largest = (a >= b) ? ((a >= c) ? a : c) : ((b >= c) ? b : c);
System.out.println(largest); // 986
```

More Arithmetic Operators

Compound Assignment Operators: `Compound assignment operators` provide a shorter way to assign an expression to a variable. There are different `compound assignment operators` available in Java: `+=`, `-=`, `*=`, `/=`, `%=`, etc.

```
int a = 10;
int b = 11;
a -= 2;
b %= 3;
System.out.println(a); // 8
System.out.println(b); // 2
```

Unary Operators: The `unary operators` are those that operate upon a single operand and produce a new value. We have learned some of the unary operators like the `logical NOT` (!) operator. A few other `unary operators` are,

- Increment Operator
- Decrement Operator

Increment Operator: The `Increment Operator` (++) is an operator which is used to increment the value of a variable by 1, on which it is applied. The `increment operator` can be used in two ways:
Prefix (++x): If an `Increment operator` is used in front of an operand, then it is called a `Prefix`.

```
int a = 10;
```

```
++a;
int number = ++a;
System.out.println("a = " + a);
System.out.println("number = " + number);
// Output is:
a = 12
number = 12
```

Postfix (x++): If an `Increment operator` is used after an operand, then is called a `Postfix`.

```
int a = 10;
a++;
int number = a++;
System.out.println("a = " + a);
System.out.println("number = " + number);
// Output is:
a = 12
number = 11
```

Decrement Operator: The `Decrement operator` is an operator which is used to decrease the value of the variable by 1, on which it is applied. The `decrement operator` can also be used in two ways:
Prefix (--x): `Prefix decrement` is similar to `prefix increment`, except that the variable value is decremented by one instead of being incremented.

```
int a = 10;
--a;
int number = --a;
System.out.println("a = " + a);
System.out.println("number = " + number);
// Output is:
a = 8
number = 8
```

Postfix (x--): `Postfix decrement` is similar to `postfix increment`, except that the variable value is decremented by one instead of being incremented.

```
int a = 10;
a--;
int number = a--;
```

```
System.out.println("a = " + a);
System.out.println("number = " + number);
// Output is:
a = 8
number = 9
```

Escape Sequences: A character with a backslash `\` just before a character is called an `escape character` or `escape sequence`. Most commonly used `escape sequences` include:

- `\n` □ New Line
- `\t` □ Tab Space
- `\\` □ Backslash
- `\'` □ Single Quote
- `\"` □ Double Quote

Java Summary Cheat Sheet - 2

https://www.ccbp.in/

Loops

Loops: It allow us to execute a block of code several times.

while loop: It allows us to execute a block of code several times as long as the condition evaluates to true

```
int counter = 0;
while (counter < 2) {
    System.out.println(counter);
    counter = counter + 1;
}
// Output is:
0
```

```
1
```

do...while loop: It is similar to the `while` loop. The only difference is that in the `do...while` loop, the check is performed after the `do...while` block of code has been executed.

```
do {
    System.out.println("Line Executed");
} while (3 > 10);
// Output is:
Line Executed
```

for loop: It is used to execute a block of code a certain number of times. It is generally used where the number of iterations is known.

```
for (int i = 0; i < 2; i++ ) {
    System.out.println(i);
}
// Output is:
0
1
```

for-each loop: It is used to iterate over arrays and various collections.

```
String[] players = {"Bryant", "Wade"};
for(String name: players) {
    System.out.println(name);
// Output is:
Bryant
Wade
```

Arrays
Array: In Java, an `array` is an object used to store similar data type values. In Java, the number of elements that an `array` can store is always fixed.
Accessing Array Elements: We can access the elements of an `array` using these index values.

```
int[] arr = {12, 4, 5, 2, 5};
System.out.println(arr[0]); //12
```

Creating an Array using a new Keyword: We can also create the `array` using `new` keyword. We can create an `array` of required length and later assign the values.

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

Printing an Array: The `Arrays` class provides a method `toString()`, which can be used to print the `array` in Java.

```
int[] arr = {12, 4, 5, 2, 5};
System.out.println(Arrays.toString(arr));
// Output is:
[12, 4, 5, 2, 5]
```

Iterating Over an Array: We can use loops to iterate over an array.
Using for Loop

```
int[] arr = {12, 4, 5};
for (int i = 0; i < arr.length; i++)
    System.out.println(arr[i]);
// Output is:
12
4
5
```

Using for-each Loop

```
int[] arr = {12, 4, 5};
for(int element: arr)
    System.out.println(element);
// Output is:
12
4
5
```

Length of an Array: In Java, we can find the array length by using the attribute `length`.

```
int[] arr = {12, 4, 5, 2, 5, 7};
System.out.println(arr.length); // 6
```

Array Concatenation: Joining two arrays is called Array Concatenation. In Java, the System class contains a method named `arraycopy()` to concatenate two arrays.

```
int[] arr1 = {12, 4, 5, 2, 5};
int[] arr2 = {6, 10,11,6};
```

```
int arr1Len = arr1.length;
int arr2Len = arr2.length;
int[] concatenatedArr  = new int[arr1Len + arr2Len];
System.arraycopy(arr1, 0, concatenatedArr,  0, arr1Len);
System.arraycopy(arr2, 0, concatenatedArr,  arr1Len, arr2Len);
System.out.println(Arrays.toString(concatenatedArr));
// Output is:
[12, 4, 5, 2, 5, 6, 10, 11, 6]
```

Array Slicing: It is a method of obtaining a subarray of an array. We can get the subarray from an array using `Arrays.copyOfRange()` method.

```
int[] originalArr = { 1, 2, 3, 4, 5 };
int startIndex = 2;
int endIndex = 4;
int[] slicedArr = Arrays.copyOfRange(originalArr, startIndex, endIndex);
System.out.println(Arrays.toString(slicedArr));
// Output is:
[3, 4]
```

Multi-dimensional Array: It consists of an array of arrays. A two-dimensional array is a collection of one-dimensional arrays.

```
int[][] arr = {{12, 4, 5}, {16, 18, 20}};
System.out.println(arr[1][2]); // 20
```

Reversing Arrays: The `Collections.reverse()` method is used for reversing the elements present in the array passed as an argument to this method.

```
Integer[] arr = {3, 30, 8, 24};
Collections.reverse(Arrays.asList(arr));
System.out.println(Arrays.toString(arr)); // [24, 8, 30, 3]
```

Sorting Arrays: The `Arrays.sort()` method can be used to sort an Array. The sorting can be done in two different ways:

- **Ascending Order**

```
int[] arr = {3, 1, 2};
Arrays.sort(arr);
System.out.println(Arrays.toString(arr)); // [1, 2, 3]
```

- **Descending Order**: The reverse sorting is done by passing `Collections.reverseOrder()` as an argument to the `Arrays.sort()` method.

```
Integer[] arr = {3, 1, 2};
Arrays.sort(arr, Collections.reverseOrder());
System.out.println(Arrays.toString(arr)); // [3, 2, 1]
```

Methods

Methods: Java doesn't have independent functions because every Java function belongs to a class and is called a Method.

Method Declaration: A Method must be declared before it is used anywhere in the program.

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

Calling a Method: The block of code in the methods is executed only when the method is called.

```
static void greet() {
    System.out.println("Hello, I am in the greet method");
}
public static void main(String[] args){
    greet();
}
// Output is:
Hello, I am in the greet method
```

Returning a Value: We can pass information from a method using the `return` keyword. In Java, `return` is a reserved keyword.

```
static String greet() {
    return "Hello, I am in the greet method";
}
public static void main(String[] args){
    System.out.print(greet());
}
// Output is:
Hello, I am in the greet method
```

Method with Parameters: The information can be passed to methods as arguments in Java.

```
static void greet(String username){
    System.out.println("Hello "+ username);
}
public static void main(String[] args){
    String name = "Rahul";
    greet(name);
}
// Output is:
Hello Rahul
```

Method Overloading: If a class has multiple methods having same name but different in parameters, it is known as Method Overloading.

```
static int addition(int a, int b) {
    return a + b;
}
static double addition(double a, double b) {
    return a + b;
}
public static void main(String[] args) {
    System.out.println(addition(20, 34)); // 54
    System.out.println(addition(45.6, 32.3)); // 77.9
}
```

Passing Mutable Objects: when mutable objects are passed as method arguments, the changes done to the object inside the method will affect the original object.

```
static void twice(int[] arr2 ) {
    for(int i = 0; i < arr2.length ; i++)
        arr2[i] = 2 * arr2[i];
}
public static void main(String[] args) {
    int[] arr1 = {36,43,10,112,66,18};
    twice(arr1);
    System.out.println(Arrays.toString(arr1));
}
```

```
// Output is:
[72, 86, 20, 224, 132, 36]
```

Passing Immutable Objects: Immutable objects are passed as method arguments, the changes done to the object inside the method will not affect the original object.

```
static void fullName(String str2) {
    str2 = "William Smith";
    System.out.println("Inside fullName() method: " + str2);
}
public static void main(String[] args) {
    String str1 = "William";
    fullName(str1);
    System.out.println("Inside main() method: " + str1);
}
// Output is:
Inside fullName() method: William Smith
Inside main() method: William
```

Recursion: Recursion is a process in which a method calls itself in the process of its execution. A recursive method terminates when a condition is met. This condition is also called the Base Case.

```
static int factorial(int num) {
    if (num == 1)
        return 1;
    return num * factorial(num - 1);
}
public static void main(String[] args) {
    System.out.println(factorial(5)); // 120
}
```

Nested Loops

Nested Loops: If a loop exists inside the body of another loop, it is called a nested loop. The inner loop will be executed one time for each iteration of the outer loop.
Example 1:

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

```
    for (int j = 0; j < 2; j = j + 1)
        System.out.println("  Inner: " + j);
}
System.out.println("After nested for loops");
// Output is:
Outer: 0
  Inner: 0
  Inner: 1
Outer: 1
  Inner: 0
  Inner: 1
After nested for loops
```

Example 2:

```
for (int i = 0; i < 2; i = i + 1) {
    System.out.println("Outer For Loop: " + i);
    int counter = 0;
    while (counter < 2) {
        System.out.println("  Inner While Loop: " + counter);
        counter = counter + 1;
    }
}
//Output is:
Outer For Loop: 0
  Inner While Loop: 0
  Inner While Loop: 1
Outer For Loop: 1
  Inner While Loop: 0
  Inner While Loop:1
```

Loop Control Statements: The statement which alters the flow of control of a loop is called a Loop Control Statement.

Name	Usage
Break	break keyword is used to stop the execution of the loop.
Continue	continue keyword is used to skip the current execution of the loop.
Break (in nested loops)	break keyword in the inner loop stops the execution of the inner loop.

Big Integers

BigInteger: The `BigInteger` is the class used for mathematical operations which involve very big integer calculations that are outside the limit of all available primitive data types.

```
String str = "51090942171709440000";
BigInteger bigNum = new BigInteger(str);
System.out.println(bigNum);
// Output is:
51090942171709440000
```

BigInteger Methods

Method	Syntax	Usage
add()	bigNum1.add(bigNum2);	performs the addition for the given two BigIntegers and returns the sum.
subtract()	bigNum1.subtract(bigNum2);	performs the subtraction for the given two BigIntegers and returns the difference.
multiply()	bigNum1.multiply(bigNum2);	performs the multiplication for the given two BigIntegers and returns the product.
divide()	bigNum1.divide(bigNum2);	performs the division for the given two BigIntegers and returns the quotient.
pow()	bigNum.pow(exponent);	performs the exponentiation operation and returns the result.
abs()	bigNum.abs();	returns a value that is equal to the absolute value of the given BigInteger.

Converting Integers to BigInteger: The `valueOf()` method can be used to convert integers values to a `BigInteger`.

```
int num = 2023;
long longNum = 435468567463L;
BigInteger bigNum1 = BigInteger.valueOf(num);
BigInteger bigNum2 = BigInteger.valueOf(longNum);
System.out.println(bigNum1); // 2023
System.out.println(bigNum2); // 435468567463
```

Converting BigInteger to Integers or String: We can convert BigInteger to int, long, and String data types.

```
String str = "45900";
BigInteger bigNum1 = new BigInteger(str);
str = bigNum1.toString();
long longNum = bigNum1.longValue();
float floatNum = bigNum1.floatValue();
```

```
double doubleNum = bigNum1.doubleValue();
System.out.println(str); // 45900
System.out.println(longNum); // 45900
System.out.println(floatNum); // 45900.0
System.out.println(doubleNum); // 45900.0
```

BigInteger Constants: The BigInteger class defines some constants for the ease of initialization.

- BigInteger.ZERO: The BigInteger constant for 0
- BigInteger.ONE: The BigInteger constant for 1
- BigInteger.TWO: The BigInteger constant for 2
- BigInteger.TEN: The BigInteger constant for 10

Java Summary Cheat Sheet - 3

https://www.ccbp.in/

ArrayList

ArrayList: The `ArrayList`s adjusts its size automatically when an element is added or removed. Hence, it is also known as a `Dynamic Array`.

```
ArrayList<Type> arrList = new ArrayList<>();
```

Adding Primitive Data Types: `ArrayList` can only store objects. To use primitive data types, we have to convert them to objects. In Java, Wrapper Classes are can be used to convert primitive types (int, char, float, etc) into corresponding objects.

Autoboxing: The conversion of primitive types into their corresponding wrapper class objects is called Autoboxing. **Unboxing**: The conversion of wrapper class objects into their corresponding primitive types is called Unboxing.

Method	Syntax	Usage
add()	arrList.add(index, element);	used to add a single element to the ArrayList.
get()	arrList.get(index);	used to access an element from an ArrayList.
set()	arrList.set(index,	used to replace or modify an element in the ArrayList.

Method	Syntax	Usage
	element);	
remove(index)	arrList.remove(index);	removes the element at the specified position, i.e index, in the ArrayList.
remove(object)	arrayList.remove(obj);	removes the first occurrence of the specified element from the ArrayList if it is present. Remains unchanged if not present
clear()	arrList.clear()	It completely removes all of the elements from the ArrayList.
size()	arrList.size()	used to find the size of an ArrayList.
indexOf()	arrList.indexOf(obj);	returns the index of the first occurrence of the specified element in the ArrayList. Returns -1 if not present

Iterating over an ArrayList: Similar to iterating over an `array`, we can use the loops to iterate over an `ArrayList`.

```
ArrayList<String> players = new ArrayList<>();
players.add(0, "Bryant");
players.add("Wade");
for (String name : players)
    System.out.println(name);
}
// Output is:
Bryant
Wade
```

ArrayList Concatenation: The `addAll()` method is used to concatenate two `ArrayLists`. This method appends the second `ArrayList` to the end of the first `ArrayList`.

```
ArrayList<Integer> arrList1 = new ArrayList<>();
arrList1.add(5);
arrList1.add(10);
ArrayList<Integer> arrList2 = new ArrayList<>();
arrList2.add(25);
arrList2.add(30);
arrList1.addAll(arrList2);
System.out.println(arrList1);
// Output is:
[5, 10, 25, 30]
```

ArrayList Slicing: The `subList()` method is used for slicing of `ArrayLists`. It works similar to the `copyOfRange()` method in `Arrays`.

```
ArrayList<Integer> arrList = new ArrayList<>();
arrList.add(5);
arrList.add(10);
arrList.add(15);
arrList.add(20);
ArrayList<Integer> subArrList = new ArrayList<>(arrList.subList(1, 3));
System.out.println(subArrList);
// Output is:
[10, 15]
```

Conversion between Arrays and ArrayLists: Arrays.asList() method is used to convert Array to ArrayList

```
Arrays.asList(arr);
```

toArray() method of ArrayList is used to convert ArrayList into an Array

```
arrList.toArray(arr);
```

Frequency of an Element: The Collections.frequency() method is used to find the frequency with which an element occurs in the given ArrayList.

```
Integer[] arr = {3, 6, 2, 1, 2};
ArrayList<Integer> arrList= new ArrayList<>(Arrays.asList(arr));
int frequency = Collections.frequency(arrList, (Integer)2);
System.out.println(frequency); // 2
```

Reversing ArrayLists: We can reverse an ArrayList by using the Collections.reverse() method.

```
Interger[] arr = {1, 2, 3, 4};
ArrayList<Integer> arrList = new ArrayList<>(Arrays.asList(arr));
Collections.reverse(arrList);
System.out.println(arrList); // [4, 3, 2, 1]
```

Sorting an ArrayList: The `Collections.sort()` method can be used to sort the given `ArrayList` in two different ways

- **Ascending order**

```
Integer[] arr = {3, 6, 2, 1};
```

```
ArrayList<Integer> arrList= new ArrayList<>(Arrays.asList(arr));
Collections.sort(arrList);
System.out.println(arrList); // [1, 2, 3, 6]
```

- **Descending order**: An ArrayList can be sorted in descending order by passing the argument Collection.reverseOrder() to the Collections.sort() method.

```
Integer[] arr = {3, 6, 2, 1};
ArrayList<Integer> arrList= new ArrayList<>(Arrays.asList(arr));
Collections.sort(arrList, Collections.reverseOrder());
System.out.println(arrList); // [6, 3, 2, 1]
```

HashSet

HashSet: The `HashSet` is also an unordered collection of elements. The `HashSet` stores only unique elements and duplicate elements are not allowed.

```
HashSet<Type> hset = new HashSet<>();
```

Method	Syntax	Usage
add()	hset.add(element);	to add a single element to the HashSet.
remove()	hset.remove(element);	removes an element from the HashSet.
clear()	hset.clear()	removes all the elements from a HashSet.
contains()	hset.contains(element);	checks if an element is present in a given HashSet.
size()	hset.size()	used to find the size of a HashSet.

Iterating Over a HashSet: Similar to iterating over an `Array` or `ArrayList`, we can use the `for-each` loop to iterate over a `HashSet`.

```
HashSet<String> players = new HashSet<>();
players.add("Rahul");
players.add("Rohit");
players.add("Virat");
players.add("Sachin");
for (String name : players)
    System.out.println(name);
// Output is:
Rohit
Rahul
Virat
Sachin
```

HashSet Operations

Union: The `addAll()` method can be used to perform the union of two sets.

Syntax: hset1.addAll(hset2);

```
HashSet<Integer> hset1 = new HashSet<>();
HashSet<Integer> hset2 = new HashSet<>();
hset1.add(3);
hset1.add(32);
hset1.add(8);
hset2.add(8);
hset2.add(32);
hset2.add(30);
hset1.addAll(hset2);
System.out.println(hset1);
// Output is:
on:  [32, 3, 8, 30]
```

Intersection: The retainAll() method can be used to perform the intersection of two sets. **Syntax**: hset1.retainAll(hset2);

```
HashSet<Integer> hset1 = new HashSet<>();
HashSet<Integer> hset2 = new HashSet<>();
hset1.add(3);
hset1.add(32);
hset1.add(8);
hset2.add(8);
hset2.add(32);
hset2.add(30);
hset1.retainAll(hset2);
System.out.println(hset1);
// Output is:
[32, 8]
```

Difference: The removeAll() method can be used to find the difference between two sets. **Syntax**: hset1.removeAll(hset2);

```
HashSet<Integer> hset1 = new HashSet<>();
HashSet<Integer> hset2 = new HashSet<>();
hset1.add(3);
```

```
hset1.add(32);
hset1.add(8);
hset2.add(8);
hset2.add(24);
hset2.add(30);
hset1.removeAll(hset2);
System.out.println(hset1);
//Output is:
[32, 3]
```

SuperSet: A superset of any given set is defined as the set which contains all the elements present in the given set. The `containsAll()` can be used to check if the given set is the superset of any other set.

```
hset1.containsAll(hset2);
```

Here, `containsAll()` checks if all the elements in `hset2` are present in `hset1`, i.e, it checks if `hset1` is a superset of `hset2`.
Subset: A subset of any given set is defined as the set which contains atleast one element present in the given set. The `containsAll()` can also be used to check if the given set is the subset of any other set.

```
hset2.containsAll(hset1);
```

Here, `containsAll()` checks if all the elements in `hset1` are present in `hset2`, i.e, it checks if `hset1` is a subset of `hset2`.
Converting to ArrayList: Conversion of `HashSet` to an `ArrayList` is done by passing the `HashSet` as an argument to the constructor of `ArrayList`.

```
ArrayList<Type> arrList = new ArrayList<>(hset);
```

HashMap

HashMap: The `HashMap` is also an unordered collection of elements. `HashMap` stores the data in key/value pairs. Here, `keys` should be unique and a `value` is mapped with the `key`. `HashMap` can have duplicate `values`.

```
HashMap<KeyType, ValueType> hmap = new HashMap<>();
```

Method	Syntax	Usage
put()	hmap.put(key, value);	used to add/update an element to the HashMap.
get()	hmap.get(key);	used to access the value mapped with a specified key in a HashMap.
replace()	hmap.replace(key, newValue);	replaces the old value of the specified key with the new value.

Method	Syntax	Usage
remove()	hmap.remove(key);	used to remove an element from a HashMap.
clear()	hmap.clear();	to remove all the elements from a HashMap.
keySet()	hmap.keySet();	returns a HashSet of all the keys of a HashMap.
values()	hmap.values();	to get all the values mapped to the keys in a HashMap.
entrySet()	hmap.entrySet();	used to get the elements of a HashMap.
size()	hmap.size();	used to find the size of a HashMap.
containsKey()	hmap.containsKey(key);	It returns true if the HashMap contains the specified key. Otherwise false is returned.
containsValue()	hmap.containsValue(value);	It returns true if the HashMap contains the specified value. Otherwise false is returned.
putAll()	hmap2.putAll(hmap1);	used to copy all the elements from a HashMap to another HashMap.

Iterating a HashMap

```
HashMap<String, Integer> playersScore = new HashMap<>();
playersScore.put("Robert", 145);
playersScore.put("James", 121);
playersScore.put("Antony", 136);
playersScore.put("John", 78);
for (Map.Entry<String, Integer> entry : playersScore.entrySet())
    System.out.printf("%s:%d\n", entry.getKey(), entry.getValue());
// Output is:
James:121
Robert:145
John:78
Antony:136
```

Math Methods

Methods	Description
pow()	It calculates the exponents and returns the result.
round()	It rounds the specified value to the closest int or long value and returns it.
min()	Returns the numerically lesser number between the two given numbers.
max()	Returns the numerically greater number between the two given numbers.
abs()	Returns the absolute value of the given number

Type Conversions

Type Conversion: Type conversion is a process of converting the value of one data type (int, char, float, etc.) to another data type. Java provides two types of type conversion :

1. Implicit Type Conversion
2. Explicit Type Conversion (Type Casting)

Implicit Type Conversion: Java compiler automatically converts one data type to another data type. This process is called Implicit type conversion.

```
int value1 = 10;
float value2 = value1;
System.out.println(value1); // 10
System.out.println(value2); // 10.0
```

Explicit Type Conversion: In Explicit Type Conversion, programmers change the data type of a value to the desired data type. This type of conversion is also called `Type Casting` since the programmer changes the data type.

```
float x = 10.0f;
System.out.println((int)x); // 10
```

Type Conversion using Methods
Converting any Primitive Type to String: Any data type can be converted to `String` using the string method `valueOf()`.

```
int num = 2;
float floatNum = 2.34f;
char ch = 'a';
System.out.println(String.valueOf(num)); // 2
System.out.println(String.valueOf(floatNum)); // 2.34
System.out.println(String.valueOf(ch)); // a
```

We may also use the `toString()` method of the corresponding wrapper class to convert primitive data types to `String`. Example 1:

```
int a = 10;
String str = Integer.toString(a);
System.out.println(str); // 10
```

Example 2:

```
char a = 'A';
String str = Character.toString(a);
```

```
System.out.println(str); // A
```

Converting String to any Primitive Type: We can convert `String` to `int` in Java using `Integer.parseInt()` method.

```
String str = "21";
int num = Integer.parseInt(str);
System.out.println(num); // 21
```

Similarly, for other primitive data types as given the table below,

Primitive Data Type	Syntax
byte	Byte.parseByte()
short	Short.parseShort()
int	Integer.parseInt()
long	Long.parseLong()
float	Float.parseFloat()
double	Double.parseDouble()
boolean	Boolean.parseBoolean()

Converting char to int: We can convert `char` to `int` in Java using `Character.getNumericValue()` method.

```
char ch = '3';
int num = Character.getNumericValue(ch);
System.out.println(num); // 3
```

Converting int to char: We can convert `int` to `char` in Java using `Character.forDigit()` method.

```
int num = 3;
char ch = Character.forDigit(num, 10);
System.out.println(ch); // 3
```

Getting Unicode Value of a Character: Using Explicit Type Conversion, we convert `char` to unicode value of type `int`.

```
char ch = 'A';
System.out.println((int)ch); // 65
```

Getting Character Representation of a Unicode Value: we have to explicitly typecast the int value to char.

```
int unicodeValue = 65;
char ch = (char)unicodeValue;
```

```
System.out.println(ch); // A
```

Java Summary Cheat Sheet - 4

https://www.ccbp.in/

Classes

Classes: `Class` can be used to bundle related properties and actions.

```
class ClassName {
    // attributes
    // methods
}
```

this keyword: Java consists of `this` keyword to access the instance attributes and methods. In Java, however, we can access them without using `this` keyword.

Attributes & Methods: In Java, instance or class attributes/methods are distinguished with the `static` keyword.

Instance Attributes: In Java, instance attributes are also called non-static attributes.

```
class Mobile {
    String model;
    String camera;
    // methods
}
```

Instance Methods: In Java, instance methods are also called non-static methods.

```
class Mobile {
    // attributes
    void makeCall() {
        System.out.println("calling...");
    }
}
```

Constructor: Unlike Java methods, a constructor should be named the same as the class and should not return a value.

```
class Mobile {
    String model;
    String camera;
    Mobile(String modelSpecs, String cameraSpecs) {
        model = modelSpecs;
        camera = cameraSpecs;
    }
}
```

Instance of Class: An instance of a class is called an `object`. An `object` is simply a collection of attributes and methods that act on those data.

```
class Mobile {
    String model;
    String camera;
    Mobile(String modelSpecs, String cameraSpecs) {
        model = modelSpecs;
        camera = cameraSpecs;
    }
}
class Base {
    public static void main(String[] args) {
        Mobile mobile1 = new Mobile("Samsung Galaxy S22", "108 MP");
    }
}
```

Accessing Attributes & Methods with Objects: we can use the dot (`.`) notation to access the attributes and methods with `objects` of a `class`.

```
class Mobile {
    String model;
    String camera;
    Mobile(String modelSpecs, String cameraSpecs) {
        model = modelSpecs;
        camera = cameraSpecs;
    }
```

```
    void makeCall(long phnNum) {
        System.out.printf("calling...%d", phnNum);
    }
}
class Base {
    public static void main(String[] args) {
        Mobile mobile1 = new Mobile("Samsung Galaxy S22", "108 MP");
        System.out.println(mobile1.model);
        System.out.println(mobile1.camera);
        mobile1.makeCall(9876543210L);
    }
}
// Output is:
iPhone 12 Pro
12 MP
calling...9876543210
```

Updating Attributes: We can update the attributes of the `objects`.

```
class Mobile {
    String model;
    String camera;
    Mobile(String modelSpecs, String cameraSpecs) {
        model = modelSpecs;
        camera = cameraSpecs;
    }
}
class Base {
    public static void main(String[] args) {
        Mobile mobile = new Mobile("iPhone 12 Pro", "12 MP");
        mobile.model = "Samsung Galaxy S22";
        mobile.camera = "108 MP";
        System.out.println(mobile.model);
        System.out.println(mobile.camera);
    }
}
// Output is:
```

```
Samsung Galaxy S22
108 MP
```

Class Attributes: Attributes whose values stay common for all the `objects` are modelled as class Attributes. The `static` keyword is used to create the class attributes.

```
class Cart {
    static int flatDiscount = 0;
    static int minBill = 100;
}
```

Accessing Class Attributes: The class attributes can also be accessed using the dot (`.`) notation. We can access the class attributes directly using the class name.

```
class Cart {
    static int flatDiscount = 0;
    static int minBill = 100;
}
class Base {
    public static void main(String[] args) {
        System.out.println(Cart.flatDiscount);
        System.out.println(Cart.minBill);
    }
}
// Output is:
0
100
```

Class Methods: In Java, class methods are also called static methods. The `static` keyword is used to create the class methods.

```
class Cart {
    static int flatDiscount = 0;
    static int minBill = 100;
    static void updateFlatDiscount(int newFlatDiscount) {
        flatDiscount = newFlatDiscount;
    }
}
```

Accessing Class Methods: The class methods can also be accessed using the dot (.) notation. We can access the class methods directly using the class name.

```
class Cart {
    static int flatDiscount = 0;
    static int minBill = 100;
    static void updateFlatDiscount(int newFlatDiscount) {
        flatDiscount = newFlatDiscount;
    }
}
class Base {
    public static void main(String[] args) {
        Cart.updateFlatDiscount(50);
        System.out.println(Cart.flatDiscount); // 50
    }
}
```

OOPS

OOPS: Object-Oriented Programming System (OOPS) is a way of approaching, designing, developing software that is easy to change.

Encapsulation: It is a process of wrapping related code and data together into a single unit.

```
class Student {
    private int age;
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
class Main {
    public static void main(String[] args) {
        Student student = new Student();
        student.setAge(20);
        System.out.println(student.getAge()); // 20
    }
```

```
}
```

Inheritance: It is the mechanism by which one class is allowed to inherit the features(fields and methods) of another class.

```
class Mammal {
  // attributes and methods
}
class Horse extends Mammal {
  // attributes and methods of Horse
}
```

The class from which the subclass is derived is called a `superclass`.

```
class Mammal {
    String name;
    Mammal(String name) {
        this.name = name;
    }
    void eat() {
        System.out.println("I am eating");
    }
}
```

The class that is derived from another class is called a `subclass`.

```
class Horse extends Mammal {
    void displayName() {
        System.out.println("My name is " + name);
    }
}
```

Method Overriding: It allows a subclass to provide a specific implementation of a method that its superclass already provides.

```
class Mammal {
    void eat() {
        System.out.println("Mammal is eating");
    }
}
class Horse extends Mammal {
    void eat() {
```

```
        System.out.println("Horse is eating");
    }
}
class Base {
    public static void main(String[] args) {
        Horse horse = new Horse();
        horse.eat();
    }
}
// Output is:
Horse is eating
```

Rules of Method Overriding

- Constructors cannot be overridden
- The overriding method should have the same return type and parameters
- A final method cannot be overridden
- The private methods cannot be overridden
- The access level cannot be more restrictive than the overridden method's access level

Composition: It describes a class whose non-static attributes refer to one or more objects of other classes.

```
import java.util.*;
class Book {
    String title;
    String author;
    Book(String title, String author) {
        this.title = title;
        this.author = author;
    }
}
class Library {
    ArrayList<Book> books;
    Library(ArrayList<Book> books) {
        this.books = books;
    }
```

```
    void displayBooks() {
        for (Book book : books) {
            System.out.printf("Title: %s, Author: %s\n", book.title,
book.author);
        }
    }
}
class Base {
    public static void main(String[] args) {
        Book book1 = new Book("Head First Design Patterns", "Eric
Freeman");
        Book book2 = new Book("Clean Code", "Robert C. Martin");
        ArrayList<Book> booksList = new ArrayList<>();
        booksList.add(book1);
        booksList.add(book2);
        Library library = new Library(booksList);
        library.displayBooks();
    }
}
// Output is:
Title: Head First Design Patterns, Author: Eric Freeman
Title: Clean Code, Author: Robert C. Martin
```

When to use Inheritance & Composition? **Inheritance**: Prefer modeling with inheritance when the classes have an IS-A relationship. **Composition**: Prefer modeling with composition when the classes have the HAS-A relationship. **Polymorphism**: It refers to an object's capacity to take several forms. Polymorphism allows us to perform the same action in multiple ways in Java. Polymorphism is of two types:

1. Compile-time polymorphism
2. Runtime polymorphism

Compile-time Polymorphism: A polymorphism that occurs during the compilation stage is known as a Compile-time polymorphism. Method overloading is an example of compile-time polymorphism.

```
class Shapes {
    public void area(double base, double height) {
```

```
        System.out.println("Area of Triangle = " + 0.5 * base * height);
    }
    public void area(int length, int breadth) {
        System.out.println("Area of Rectangle = " + length * breadth);
    }
}
class Base {
    public static void main(String[] args) {
        Shapes triangle = new Shapes();
        Shapes rectangle = new Shapes();
        triangle.area(8.5, 10.23);
        rectangle.area(5, 3);
    }
}
// Output is:
Area of Triangle = 43.4775
Area of Rectangle = 15
```

Runtime Polymorphism: A polymorphism that occurs during the execution stage is called Runtime polymorphism. Method overriding is an example of Runtime polymorphism.

```
class Mammal {
    void eat() {
        System.out.println("Mammal is eating");
    }
}
class Dog extends Mammal {
    void eat() {
        System.out.println("Dog is eating");
    }
}
class Base {
    public static void main(String args[]) {
        Mammal mammal = new Dog();
        mammal.eat();
    }
```

```
}
// Output is:
Dog is eating
```

Abstraction: It is the process of hiding certain details and showing only essential information to the user. Abstraction can be achieved with,

1. Abstract classes
2. Interfaces

Abstract Classes and Methods **Abstract Classes**: The `abstract` keyword is a non-access modifier, applied to classes and methods in Java. A class which is declared with the `abstract` keyword is known as an abstract class.

```
abstract class ClassName {
    // attributes and methods
}
```

Abstract Methods: A method which is declared with the `abstract` keyword is known as an abstract method.

```
abstract returnType methodName();
```

Interface: It is similar to an abstract class. It cannot be instantiated and consists of abstract methods.

```
interface InterfaceName {
    // body of the interface
}
```

The `implements` keyword is used to inherit interfaces from a class.

```
interface CricketPlayer {
    void run();
}
class Person implements CricketPlayer {
    public void run() {
        System.out.println("Running");
    };
}
class Base {
    public static void main(String[] args) {
        Person person = new Person();
```

```
        person.run();
    }
}
// Output is:
Running
```

Default Methods in Interfaces: We can write the implementation of a method inside the interface. These methods are called `default methods`. The `default` keyword is used to declare a method in the interface as `default method`.

```
accessModifier default returnType methodName() {
    // block of code
}
```

Inheritance among Interfaces: An interface can inherit another interface. We use the `extends` keyword to inherit an interface from another interface.

```
interface CricketPlayer {
    void run();
}
interface Wicketkeeper extends CricketPlayer {
    void wicketkeeping();
}
class Person implements Wicketkeeper {
    public void wicketkeeping() {
        System.out.println("Wicketkeeping");
    }
    public void run() {
        System.out.println("Running");
    };
}
class Base {
    public static void main(String[] args) {
        Person person = new Person();
        person.wicketkeeping();
        person.run();
    }
}
// Output is:
```

```
Wicketkeeping
Running
```

Errors & Exceptions

Errors & Exceptions **Errors**: In Java occur due to syntactical errors, infinite recursion, and many other reasons. The most common are syntactical errors that occur when a programmer violates the rules of Java programming language.
Exceptions: Even when our code is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called Exceptions.
Handling Exceptions: In Java, we have `try...catch` block to handle the exceptions. we can specify multiple `catch` blocks to handle different types of exceptions. `finally` block which is always executed whether an exception occurs inside the try block or not.

```
try {
    int result = 5 / 0;
    System.out.println(result);
    } catch (ArithmeticException e) {
        System.out.println("Denominator can't be 0");
    } catch (ArithmeticException e) {
        System.out.println("Invalid value");
    } finally {
        System.out.println("Execution completed");
    }
}
// Output is:
Denominator can't be 0
Execution completed
```

Raising Exceptions: In Java, we can throw an exception explicitly using `throw` keyword.

```
try {
    throw new ArithmeticException("Denominator can't be 0");
} catch (ArithmeticException e) {
    System.out.println(e.getMessage());
}
// Output is:
```

```
Denominator can`t be 0
```

throws Keyword: In Java, `throws` keyword is used to specify the type of exception that might be thrown by a method.

```
class Main {
    static void divideByZero() throws ArithmeticException {
        throw new ArithmeticException("Division with zero");
    }
    public static void main(String[] args) {
        try {
            divideByZero();
        } catch (ArithmeticException e) {
            System.out.println(e);
        }
    }
}
// Output is:
java.lang.ArithmeticException: Division with zero
```

Working With Dates & Times

Date and Time Java has a built-in java.time package which provides various classes to work with date and time.

Working with LocalDate class Java `LocalDate` class allows us to create a `date object` and represent a valid date (year, month and day). The `of()` method of `LocalDate` class is used to create an instance of `LocalDate` from the given year, month and day.

```
LocalDate dateObj = LocalDate.of(2019, 4, 13);
System.out.println(dateObj); // 2019-04-13
```

Today's Date: The `LocalDate` class provides `now()` method which returns the date object with today's date.

```
LocalDate dateObj = LocalDate.now();
System.out.println(dateObj); //2023-2-22
```

Working with LocalTime class Java `LocalTime` class allows us to create a `time object` and represent a valid time (hours, minutes and seconds).

```
LocalTime timeObj = LocalTime.of(11, 34, 56);
System.out.println(timeObj); // 11:34:56
```

Working with LocalDateTime class Java `LocalDateTime` class allows us to create a `date-time object` and represent a valid date and time together.

```
LocalDateTime dateTimeObj = LocalDateTime.of(2018, 11, 28, 10, 15, 26);
System.out.println(dateTimeObj.getYear()); // 2018
System.out.println(dateTimeObj.getMonthValue()); // 11
System.out.println(dateTimeObj.getHour()); // 10
System.out.println(dateTimeObj.getMinute()); // 15
```

Working with DateTimeFormatter class The `DateTimeFormatter` class have a `ofPattern()` method that creates an instance of the `DateTimeFormatter` for the given pattern of the date, time and date-time like,

- mm/dd/yyyy
- hh/mm/ss

```
LocalDate now = LocalDate.now();
DateTimeFormatter format1 = DateTimeFormatter.ofPattern("dd MMMM yyyy");
String formattedDate = now.format(format1);
System.out.println(formattedDate); // 13 September 2022
```

Parsing Date and Time The `DateTimeFormatter` class have a `parse()` method which creates the respective object (date, time, or date-time) from the give string.

```
String dateStr = "28 November 2018";
DateTimeFormatter format1 = DateTimeFormatter.ofPattern("dd MMMM yyyy");
LocalDate date = LocalDate.parse(dateStr, format1);
System.out.println(date); // 2018-11-28
```

Difference Between Dates & Times: In Java, we have a `Period` class to find the difference between two dates in terms of years, months and days.

```
LocalDate startDate = LocalDate.of(2020, 2, 20);
LocalDate endDate = LocalDate.of(2021, 10, 21);
Period period = Period.between(startDate, endDate);
System.out.println(period.getYears()); // 1
System.out.println(period.getMonths()); // 8
System.out.println(period.getDays()); // 1
```

In Java, we have a `Duration` class to find the difference between two times in seconds.

```
LocalTime startTime = LocalTime.of(10, 30, 30);
LocalTime endTime = LocalTime.of(10, 31, 30);
Duration duration = Duration.between(startTime, endTime);
System.out.println(duration.getSeconds()); // 60
```

Types of Inheritance

Single Inheritance: Single inheritance involves extending a single superclass from a single subclass.

```
class Mammal {
    String type;
}
class Horse extends Mammal {
    String breed;
}
```

Multilevel Inheritance: In multilevel inheritance, a subclass extends from a superclass and then the same subclass acts as a superclass for another class.

```
class Mammal {
    String type;
}
class Horse extends Mammal {
    String breed;
}
class MustangHorse extends Horse {
    String name;
}
```

Hierarchical Inheritance: In hierarchical inheritance, multiple subclasses extend from a single superclass.

```
class Mammal {
    String type;
}
class Horse extends Mammal {
    String breed;
}
class Dog extends Mammal {
    String breed;
```

```
}
```

Multiple Inheritance: In multiple inheritance, a single class/interface can inherit multiple interfaces.

```
interface InswingBowler {
    void inswing();
}
interface OutswingBowler {
    void outswing();
}
class BowlerA implements InswingBowler, OutswingBowler {
    public void inswing() {
        System.out.println("Inswing bowling");
    }
    public void outswing() {
        System.out.println("Outswing bowling");
    }
}
```

Final

Final keyword: The `final` keyword is used for variables, classes and methods, which makes them non-changeable (impossible to inherit or override).
Final variable: When the `final` keyword is used with a variable, it indicates that the variable is constant and the value of it cannot be reassigned.

```
final int SPEED_LIMIT = 60;
SPEED_LIMIT = 90;
System.out.println(SPEED_LIMIT);
// Output is:
file.java:4: error: cannot assign a value to final variable SPEED_LIMIT
        SPEED_LIMIT = 90;
        ^
```

Final method: A method declared with `final` is called a `final method`, it restrict the unwanted and improper use of method definition while overriding the method.

```
class Mammal {
    final void eat() {
        System.out.println("Mammal is eating");
```

```
    }
}
class Horse extends Mammal {
    void eat() {
        System.out.println("Horse is eating");
    }
}
class Base {
    public static void main(String[] args) {
        Horse horse = new Horse();
        horse.eat();
    }
}
// Output is:
file.java:10: error: eat() in Horse cannot override eat() in Mammal
    void eat() {
         ^
overridden method is final
```

Final class: A class declared with `final` is called a `final class`, it cannot be inherited. All the wrapper classes are `final classes`. Hence, we cannot inherit the wrapper classes.

```
final class Product {
    String name;
    int price;
}
class ElectronicItem extends Product {
    int warrantyInMonths;
}
class Base {
    public static void main(String[] args) {
        ElectronicItem obj = new ElectronicItem();
    }
}
// Output is:
Main.java:5: error: cannot inherit from final Product
```

```
class ElectronicItem extends Product {
                              ^
```

Access Modifiers

Access Modifiers: `Access modifiers` are the keywords that set access levels when used with the classes, methods, constructors, attributes, etc. In Java, we have four access modifiers to set the accessibility. They are,

1. **Private**: The access level of a private modifier is only within the declared class. It is not accessible outside of the class.
2. **Default**: The access level of a default modifier is up to the class/subclass of the same package. It cannot be accessed from outside the package.
3. **Protected**: The access level of a protected modifier is up to the class/subclass of the same package and also to a different package through the subclass. A subclass is required to access it from a different package.
4. **Public**: The access level of a public modifier is everywhere in the program. It means that it is accessible from the class/subclass of the same/different package.

Accessibility of Access Modifiers

Access Modifier	Same Class	Same package subclass	Same package other classes	Different package subclass	Different package other classes
private	Yes	No	No	No	No
default	Yes	Yes	Yes	No	No
protected	Yes	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes	Yes

All the `access modifiers` work the same with the variables, methods and constructors.

Access Modifiers with Classes: `Classes` in Java can only have `Public` or `Default` as access modifiers.

- **Public**: When a class is declared `public`, it is accessible from the class/subclass of the same/different package.
- **Default**: When no access modifier is specified to the classes, then they can be called `default classes`, it is accessible only to the classes or subclasses of the same package.

Upcasting

Upcasting: In Java, a superclass reference variable can be used to refer to its subclass object i.e., we can specify a superclass as a type while creating an object of its subclass.
Invoking Methods: While upcasting, we can access all the members of the superclass but can only access a few members like overriding methods of the subclass.

```
class Mammal {
    void eat() {
        System.out.println("Mammal is eating");
    }
}
class Horse extends Mammal {
    void eat() {
        System.out.println("Horse is eating");
    }
}
class Base {
    public static void main(String[] args) {
        Mammal animal = new Horse();
        animal.eat();
    }
}
// Output is:
Horse is eating
```

Invoking a method specific to subclass,

```
class Mammal { }
class Horse extends Mammal {
    void eat() {
        System.out.println("Horse is eating");
    }
}
class Base {
    public static void main(String[] args) {
        Mammal animal = new Horse();
        animal.eat();
```

```
    }
}
// Output is:
Main.java:10: error: cannot find symbol
        animal.eat();
```

Invoking Methods: We cannot access the attributes of the subclass.

```
class Mammal { }
class Horse extends Mammal {
    String breed = "Shire";
}
class Base {
    public static void main(String[] args) {
        Mammal animal = new Horse();
        System.out.println(animal.breed);
    }
}
// Output is:
file.java:8: error: cannot find symbol
        System.out.println(animal.breed);
```

Super Keyword

super: `super` is the keyword is used to access the superclass members like attributes, methods and also the constructors inside the subclass.

Accessing Attributes and Methods using super Keyword

```
class Mammal {
    String type = "animal";
    void eat() {
        System.out.println("Eating");
    }
}
class Horse extends Mammal {
    String type = "mammal";
    void display() {
        System.out.println("Horse is an " + super.type);
    }
    void eat() {
```

```
        super.eat();
    }
}
class Base {
    public static void main(String[] args) {
        Horse horse = new Horse();
        horse.display();
        horse.eat();
    }
}
// Output is:
Horse is an animal
Eating
```

Invoking Constructors using super()

- Invoking non-parameterized superclass constructor
- Invoking parameterized superclass constructor

Invoking non-parameterized superclass constructor

```
class Mammal {
    String name;
    Mammal() {
        System.out.println("Superclass constructor called");
    }
}
class Horse extends Mammal {
    String breed;
    Horse(String breed) {
        this.breed = breed;
    }
}
class Base {
    public static void main(String[] args) {
        Horse horse = new Horse("Shire");
    }
}
```

```
// Output is:
Superclass constructor called
```

Invoking parameterized superclass constructor

```
class Mammal {
    String name;
    Mammal(String name) {
        this.name = name;
    }
}
class Horse extends Mammal {
    String breed;
    Horse(String name, String breed) {
        super(name);
        this.breed = breed;
    }
}
class Base {
    public static void main(String[] args) {
        Horse horse = new Horse("Alex", "Shire");
        System.out.println(horse.name);
        System.out.println(horse.breed);
    }
}
// Output is:
Alex
Shire
```

Java Summary Cheat Sheet - 5

https://www.ccbp.in/

Lambda Expressions

Functional Interface: A functional interface is an `interface` that has a single `abstract` method (SAM).

```
interface Runner {
    void run();
}
```

Lambda Expressions: They provide a way to represent a function as an `object`. It is an anonymous function that can be passed around as a value.

```
interface Runner {
    void run();
}
class BaseballPlayer {
    public static void main(String[] args) {
        Runner ref = () -> System.out.println("Running");
        ref.run();
    }
}
// Output is:
Running
```

Lambda Expression with Parameters: The `lambda` expressions can have parameters similar to methods.

```
interface Sum {
    int add(int a, int b);
}
class Main {
    public static void main(String[] args) {
        Sum ref = (a, b) -> a + b;
        System.out.println(ref.add(2, 3)); // 5
    }
}
```

Lambda body for a block of code: The lambda expression consisting a block of code is surrounded by curly braces.

```
interface Sum {
    int add(int a, int b);
}
class Main {
    public static void main(String[] args) {
        Sum ref = (x, y) -> {
            int numSum = x + y;
            if (numSum > 0)
              return numSum;
            return 0;
        };
        System.out.println(ref.add(2, 3)); // 5
    }
}
```

Streams

Stream: A `stream` is a sequence of elements that supports various operations for processing the elements. We can perform operations on a stream without modifying the original data.

Creating an Empty Stream: The `Stream` interface consists of a `static` and default method `empty()` that can be used to create an empty stream.

```
Stream<String> stream = Stream.empty();
```

Creating a Stream with Collections: All the classes that implement `Collection` interface consists of a `stream()` method that can be used for creating a `stream` of its corresponding type.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
Stream<Integer> numStream = numbers.stream();
```

We can also create a stream directly using the `of()` method of the `Stream` interface.

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
```

Chaining Streams: We can combine multiple `stream` operations together to form a single expression.

```
streamSource.intermediateOperation().terminalOperation();
```

Intermediate Operations: An intermediate operation is a stream operation that produces a new stream as output after performing the specified operations. A stream can have zero or multiple intermediate operation.

Method	Syntax	Usage
filter()	stream.filter(Predicate)	used to filter the elements of a stream based on a condition.
map()	stream.map(Function)	used to perform an operation on each element of the stream.
sorted()	stream.sorted(Comparator)	used to sort the elements of the stream.
distinct()	stream.distinct()	used to remove duplicate elements from the stream.

Terminal Operations: A terminal operation is a stream operation that consumes the elements of a stream and produces a result. After a terminal operation, no other operation can be done.

Method	Syntax	Usage
forEach()	stream.forEach(Consumer);	used to iterate and perform the specified operation on each element of the stream.
count()	stream.count();	used to count the number of elements in a stream.
collect()	stream.collect(Collector);	commonly used to perform a reduction operation on the elements of a stream and produce a result.
anyMatch()	stream.anyMatch(Predicate);	used to check if any of the elements in the stream has matched a specified condition.
allMatch()	stream.allMatch(Predicate);	used to check if all the elements in the stream match a specified condition.
noneMatch()	stream.noneMatch(Predicate);	used to check if no element in the stream matches a specified condition.
findFirst()	stream.findFirst();	used to find the first element in a stream.
findAny()	stream.findAny();	used to find any element in a stream.
reduce()	stream.reduce(BinaryOperator);	used to combine a stream of elements into a single result, using a specified operation.
min()	stream.min(Comparator);	used to find the minimum element of a stream.

Method	Syntax	Usage
max()	stream.max(Comparator);	used to find the maximum element of a stream.

Example 1:

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "Dave");
names.stream()
    .filter(eachName -> eachName.length() > 4)
    .forEach(name -> System.out.println(name));
// Output is:
Alice
Charlie
```

Example 2:

```
List<Integer> numbers = Arrays.asList(2, 5);
numbers.stream()
    .map(eachNumber -> eachNumber * 2)
    .forEach(number -> System.out.println(number));
// Output is:
4
10
```

Optionals

Optional Class: The `Optional` class is designed to be used as a return type for methods that may or may not return a value. It provides several methods to help prevent the `NullPointerException`.

Creating Optionals: The Optional class provides different methods to create optionals.

1. empty()
2. of()
3. ofNullable()

empty(): The `empty()` is a static method that creates an instance of `Optional` with no values.

```
Optional<Integer> optionalInt = Optional.empty();
System.out.println(optionalInt);  // Optional.empty
```

of(): The `of()` is a static method that accepts a non-null value as an argument and returns an instance of the `Optional` with the specified value.

```
Optional<Integer> optionalInt = Optional.of(324);
System.out.println(optionalInt); // Optional[324]
```

ofNullable(): The `ofNullable()` is a static method that accepts a value and returns the instance of `Optional` with the specified value. If null is provided as a value, it returns an `empty` optional.

```
Optional<Integer> optionalInt = Optional.ofNullable(324);
System.out.println(optionalInt); // Optional[324]
Optional<Integer> optionalInt = Optional.ofNullable(null);
System.out.println(optionalInt); // Optional.empty
```

Optional Methods:

Method	Syntax	Usage
isPresent()	optional.isPresent();	used to check if an element is present in the optional.
get()	optional.get();	used to get the element in the optional. It throws an exception if no element is present.
orElse()	optional.orElse();	used to get a default value (constant value) if the optional is empty.
orElseGet()	optional.orElseGet(Supplier);	used to get a default value (dynamic value) if the optional is empty.
ifPresent()	optional.ifPresent(Consumer);	used to perform specified operation on the elements in the optional.
map()	optional.map(Function);	used to perform a specified operation on elements in an optional and return a new optional.
filter()	optional.filter(Predicate);	used to filter the elements in an optional based on a specified condition and return a new optional.

I/O Streams

Paths: In Java, the `Path` interface from `java.nio.file` package represents a path to a file or directory in the file system. The paths are of two types:

1. Relative path
2. Absolute path

Relative Path: A relative path is a path that specifies the location of a file or directory relative to the current working directory. Example: documents\file.txt

Absolute Path: An absolute path is a path that specifies the exact location of a file or directory in the file system, starting from the root directory. Example: /home/rahul/documents/file.txt
Finding Absolute Path using Java

```
File file = new File("file.txt");
System.out.println(file.getAbsolutePath());
// Output is:
/home/rahul/documents/file.txt
```

BufferedWriter: The `BufferedWriter` buffers characters to provide for the efficient writing of single characters, arrays, and strings. The `Writer` writes text to a character-output stream. **Writing Data to a File**

```
try {
    BufferedWriter buffer = new BufferedWriter(new
FileWriter("destination.txt"));
    buffer.write("Writing to a file using BufferedWriter");
    buffer.close();
} catch (IOException e) {
    System.out.println(e.getMessage());
}
// Text in the destination.txt file
Writing to a file using BufferedWriter
```

BufferedReader: The `BufferedReader` is a class from the `java.io` package extends the abstract class `Reader`. It uses the `Reader` object to read data from a file and it stores the data in an internal buffer. **Reading Data from a File**: The `BufferedReader` provides various methods to read data from a file:

1. read()
2. readLine()

read(): The `read()` method reads a single character from a file. It returns `int` data type.

```
// Text in the source.txt file
Hello World!
try {
    BufferedReader br = new BufferedReader(new
FileReader("source.txt"));
    char[] arr = new char[100];
```

```
    br.read(arr);
    System.out.println(arr); // Hello World!
    br.close();
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

readLine(): The `readLine()` method is used to read a single line of text in the file. It returns `String` data type.

```
try {
    BufferedReader br = new BufferedReader(new
FileReader("source.txt"));
    BufferedWriter bw = new BufferedWriter(new
FileWriter("destination.txt"));
    String line = br.readLine();
    bw.write(line);
    br.close();
    bw.close();
} catch (Exception e) {
    System.out.println(e.getMessage());
}
// Text in the source.txt file
Hello World!
This is the second line.
// Text in the destination.txt file
Hello World!
```

Generics

Type Parameters: In Java, `type parameter` names are typically written in uppercase letters to distinguish them from regular class or interface names. The most commonly used type parameter names are:

- E - Element
- K - Key
- N - Number
- T - Type
- V - Value

Generics: Generics allows us to write a single piece of code that can work with multiple types. Generics do not work with primitive types.
Generic Classes: In Java, a `generic class` is a class that can work with multiple data types. This allows for flexibility and reusability in programming.

```
class SampleClass<T> {
    private T data;
    public SampleClass(T data) {
        this.data = data;
    }
    public void printDataType() {
        System.out.println("Type: " +
this.data.getClass().getSimpleName());
    }
}
class Main {
    public static void main(String[] args) {
        SampleClass<Integer> intObj = new SampleClass<>(3);
        intObj.printDataType(); // Type: Integer
        SampleClass<String> stringObj = new SampleClass<>("Java");
        stringObj.printDataType(); // Type: String
    }
}
```

Generic Methods: We can create a method in Java that can be used with any type of data by using `generics`. This type of method is known as a `generics method`.

```
class SampleClass<T> {
    private T data;
    public SampleClass(T data) {
        this.data = data;
    }
    public T getData() {
        return this.data;
    }
}
class Main {
```

```
    public static void main(String[] args) {
        SampleClass<Integer> intObj = new SampleClass<>(3);
        System.out.println(intObj.getData()); // 3
        SampleClass<String> stringObj = new SampleClass<>("Java");
        System.out.println(stringObj.getData()); // Java
    }
}
```

Generic Interfaces: We can create `interfaces` in Java that can be used with any type of data by using `generics`. This type of method is known as a `generics interface`.

```
interface Processor<T> {
    void process(T t);
}
class Main<T> implements Processor<T> {
    public void process(T obj) {
        System.out.println("The process() method is called");
    }
    public static void main(String[] args) {
        Main<String> obj = new Main<>();
        obj.process("Java");
    }
}
// Output is:
The process() method is called
```

Bounded Types: We can use `bounded types` by specifying the upper bound type parameter with the `extends` keyword.

```
class Main<T extends Number> {
    T data;
    Main(T data) {
        this.data = data;
    }
    void display() {
        System.out.println(data);
    }
    public static void main(String[] args) {
```

```
        Main<Integer> intObj = new Main<>(3);
        intObj.display(); // 3
        Main<Double> doubleObj = new Main<>(3.14);
        doubleObj.display(); // 3.14
    }
}
```

Java Behind the Scenes

The `JRE` (Java Runtime Environment), `JVM` (Java Virtual Machine), and `JDK` (Java Development Kit) are the components of the `Java ecosystem`, and they are often used together when developing and running Java programs.

Java Development Kit (JDK): `JDK` provides the environment to develop and execute the Java program. It includes `JRE` and other development tools like compilers, debugger, etc.

Java Runtime Environment (JRE): `JRE` is an installation package that provides an environment to only run(not develop) the Java program onto our machine. It consists of many components like a Java class library, tools, and a separate `JVM`.

Java Virtual Machine (JVM): `JVM` is a software program that allows us to run Java programs on a computer. It is designed to work on any type of hardware or operating system, so we can run Java programs on any device as long as it has a `JVM` installed. This makes Java programs portable, meaning they can be run on any device with a `JVM`.

Execution of Java Programs: The execution of a Java code series of steps that convert the source code into a form that the computer can understand and execute.

1. **Source code**: All the Java codes we write consists of a `.java` as the file extension. These Java files act as the source of our Java codes.
2. **Compilation**: We use a command-line tool called `javac` provided by the JDK, to compile Java programs. The Java compiler takes the source file as input and produces a compiled file with the `.class` file extension. This file contains the `Java bytecode`.
3. **Interpretation**: To execute the bytecode, we use the Java interpreter, which is a command-line tool called `java`. The Java interpreter takes the `class` file as input and runs it on the `JVM`.
4. **Executing Machine Code**: `Machine code` is a low-level code that is understandable by the machine. It typically consists of 0's and 1's. The `machine code` is platform-dependent. The `JVM` is responsible for creating the `machine code` that is understandable for the specific processor or OS.

Java Archive (JAR): `JAR` is a file format used to package and share Java programs and libraries. It is similar to a ZIP file but it also includes metadata about the contents of the `JAR` file, such as the main class of a program, the version of the Java runtime required to run the program, and any dependencies on other libraries.
Classpath: The `classpath` specifies the locations where the `JVM` should look for class files when it is asked to load a class.