Lesson:

Operator, Array, String In Java

PW SKILLS

List of Concepts Involved:

- Working with Operators
- Working with flow control statements
- Working with Arrays
- Working with String,StringBuilder,StringBuffer

Operators in Java

Operators in Java can be classified into 6 types:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Unary Operators
6. Bitwise Operators

Java Arithmetic operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:
Assume integer variable num1 holds 20 and variable num2 holds 30, then:

Operator	Description and Example
+ Addition	Adds values on either side of the operator. Example: num1 + num2 will give 50
- Subtraction	Subtracts right hand operand from left hand operand Example: num1 - num2 will give -10
* Multiplication	Multiplies values on either side of the operator Example: num1 * num2 will give 600
/ Division	Divides left hand operand by right hand operand Example: num1 / num2 will give 0.67
% Modulus	Divides left hand operand by right operand and returns remainder Example: num2 % num1 will give 10
++ Increment	Increases the value of operand by 1 Example: num2++ gives 31
-- Decrement	Decreases the value of operand by 1 Example: num1-- gives 19

Conditional operators

They are used when a condition comprises more than one boolean expression. For instance, if we want to print a number only if it is greater than 2 and less than 5, then we will use conditional operators to combine the 2 expressions. We have 3 types of conditional operators - logical-and, logical-or and ternary operator.

Logical-and operator (&&)

It is used when we want the condition to be true iff both the expressions are true.

Syntax

```
if(condition - 1 && condition - 2) {
      statement;
}
```

Example

Print the number if the input value is greater than 5 and less than 10.

Code

```
if (val > 5 && val < 10) {
      System.out.print(val);
}
```

Case - 1: val = 3
Output- No output
Explanation- The input value is less than 10 but it is not greater than 5.

Case - 2: val = 7
Output- 7
Explanation- The input value is both less than 10 and greater than 5.

Case - 3: val = 13
Output- No output
Explanation- The input value is greater than 5 but it is not less than 10.

Common misconception: '&' v/s '&&'

& -> compares bitwise
&& -> logical and

The first one parses through all the conditions, despite their evaluation as true or false. However the latter traverses through the second condition only if the first condition is evaluated as true, otherwise it negates the entire condition. For instance,

```
System.out.println(false & 5/0==2);
```

It gives us a runtime error as 5 is being divided by 0, which isn't a valid operation. However if we write-

```
System.out.println(false && 5/0==2);
```

We get "false" as our output. This is so because our first condition is false, which is enough to make the entire condition false in case of logical and.

Try this

Write a program to print the value of input if it is even and divisible by 3.

Logical-or operator (||)

This operator is used when we are satisfied as long as any one of the boolean expressions is evaluated as true.

Syntax

```
if(condition - 1 || condition - 2) {
 statement;
}
```

Example
Print the number if the input value is greater than 10 or less than 5.

Code

```
if (val < 5 || val > 10) {
 System.out.print(val);
}
```

Case - 1: val = 3
Output- 3
Explanation- The input value is less than 5. It is enough to satisfy the condition so the second condition won't be tested and the val will be printed.

Case - 2: val = 7
Output- No output
Explanation- Both the conditions are evaluated as false.

Case - 3: val = 13
Output- 13
Explanation- The input value is not smaller than 5 but it is greater than 10. So overall it will be evaluated as true.

Try this

Write a program to print the value of input if it is divisible by 3 or 5.

Common misconception: '|' v/s '||'

| -> compares bitwise
|| -> logical or

The first one parses through all the conditions, despite their evaluation as true or false. However the latter traverses through the second condition only if the first condition is evaluated as false, otherwise it validates the entire condition. For instance,

```
System.out.println(true | 5/0==2);
```

It gives us a runtime error as 5 is being divided by 0, which isn't a valid operation. However if we write-

```
System.out.println(true || 5/0==2);
```

We get "true" as our output. This is so because our first condition is true, which is enough to make the entire condition true in case of logical or.

Ternary operator (?:)

It is a smaller version for the if-else statement. If the condition is true then the statement - 1 is executed else the statement - 2 is executed.

Syntax

```
condition ? statement - 1 : statement - 2;
```

Example
Without ternary operator

```
if (val % 2 == 1) {
 System.out.println("Value entered is odd");
} else {
 System.out. println("Value entered is even");
}
```

With ternary operator

```
val % 2 == 1 ? System.out.println("Value entered is odd) :
System.out.println("Value entered is even);
```

Case - 1: val = 1
Output **(without ternary operator)** - Value entered is odd
Output **(without ternary operator)** - Value entered is odd

Case - 2: val = 2
Output **(without ternary operator)** - Value entered is even
Output **(without ternary operator)** - Value entered is even

Try this

1. **Write a short program that gives the following as output -**
   For each multiple of 3, print "Fizz" instead of the number.
   For each multiple of 5, print "Buzz" instead of the number.
   For numbers which are multiples of both 3 and 5, print "FizzBuzz" instead of the number.
   Otherwise print the number itself.

2. **Write a short program that prints each number from 1 to 100 on a new line, except if the number is a multiple of 5 or 7.**

If statement

In real life we often encounter situations where our actions are governed by some sort of conditions. For instance, if the weather is rainy, we carry an umbrella. Here carrying an umbrella is an action which is

performed only when the condition of the weather being rainy is fulfilled.

An if statement is based on the same principle. It executes a statement based upon if some condition is true.

Syntax

```
if (condition) {
 statement;
}
```

Example:-

```
if(marks > 33) {
 System.out.print("Pass");
}
```

Case - 1: marks = 85
Output - Pass
Explanation - Since the marks are greater than 80 i.e. the condition inside 'if' parentheses is true, we get "Pass" printed in our output.

Case - 2: marks = 70
Output - No output
Explanation - Since the marks are not greater than 80 i.e. the condition inside if parentheses is false, the statement in it's block is skipped i.e. we get no output.

If-else statement

Sometimes we encounter situations where we have 2 types of actions that we can perform. For example, if it's tuesday i'll eat veg burger otherwise i'll eat non-veg burger.
An if else statement is designed to give us this functionality in our code. It executes statements based upon if some condition is true or false. If the condition is true, the if statement is executed, otherwise the else statement is executed.

Syntax

```
if (condition) {
statement - 1
} else {
statement - 2
}
```

Example -

To illustrate the if-else statement, we can create a grading system. We'll assume that the score ranges between 0 to 100 inclusive. A score above 33 gets a "Pass" verdict, otherwise it's "Fail".

To solve this problem, we can use an if else statement to execute different actions for failing and passing grades:

```
if (score > 33) {
 System.out.println("Pass");
} else {
```

```
 System.out. println("Fail");
}
```

Case - 1: grade = 60
Output - Pass
Explanation - Since the score is more than 33 i.e. the condition inside 'if' parentheses is true, we get "Pass" printed.

Case - 2: grade = 20
Output - Fail
Explanation - Since the score is not more than 33 i.e. the condition inside 'if' parentheses is false, we get "Fail" printed.

Try this

1. **Find if the input value is odd or even. If it's odd print "Odd", otherwise print "Even".**
   Note: Input value will be between 1 and 10^6.
2. **Find if the input character is 'a' or not.**
   Note: Input characters will be lowercase alphabets.

If-else if statement
It works on the same principle as if-else statements, except here we can have multiple conditions. If the condition inside the if block is true, then a code/ statement is executed, but if it is false, it moves to the else-if block and will check if it is true and will execute the statement in the else-if block. But when the condition in this block is false, it will execute the statement in the final else block.

We can have multiple else-if blocks too.

Syntax

```
if (condition - 1) {
statement - 1
} else if (condition - 2) {
statement - 2
} else {
 statement - 3
}
```

Example
To understand this, we can further expand our grading system. Apart from pass and fail now it will give grades based on the score.

Grade	Score
A	80 - 100
B	60 - 80
C	40 - 60
D	< 40

To solve this problem, we can use an if - else if - else statement to execute different actions depending upon the score:

```
if(score > 80) {
 System.out.println(“A”);
} else if (score > 60) {
 System.out.println(“B”);
} else if (score > 40) {
 System.out.println(“C”);
} else {
 System.out.println(“D”);
}
```

Case - 1: grade = 81
Output - A
Explanation - Explanation - Since the score is more than 80 i.e. the condition inside if parentheses is true, “A” gets printed.

Case - 2: grade = 61
Output - B
Explanation - Since the score is less than 80, the first block is skipped and since it is more than 60 i.e. the condition inside else-if parentheses is true, “B” gets printed.

Case - 3: grade = 41
Output - C
Explanation - Since the score is 41, the first 2 blocks are skipped and then the condition for the third block is checked. It turns out that it is true, so “C” gets printed and the rest is skipped.

Case - 4: grade = 21
Output - D
Explanation - Since all the conditions are falsified by the input, the else-block is run and we get “D” as output.

Try this

1. **Write a program to identify people as “Child” (age < 12), “Teenager” (12 <= age < 18) or “Adult” (age >= 18).**
2. **Print the maximum of 3 numbers a, b, c taken as input.**

Nested if-else
It is simply an if-else statement inside another if-else statement.

Syntax

```
if (condition - 1) {
    if (condition - 2) {
         statement - 1
    } else {
         statement - 2
    }
```

```
} else {
     statement - 3
}
```

Example:-

```
if (score > 33) {
     if(marks > 80) {
          System.out.print(“Gracefully ”);
     }
     System.out.println(“Pass”);
} else {
     System.out. println(“Fail”);
}
```

Note - Watch Your Curly Braces

Switch statement

Let’s say we have a variable. Now, we want to do multiple operations on it based upon what value it is storing. In such cases the switch statement comes into play.

It is like an if-else ladder with multiple conditions, where we check for equality of a variable with various values.

It works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use **strings** in the switch statement.

Syntax

```
switch (expression) {
case x:
       // code
       break;
case y:
       // code
       break;
.
.
.
default:
       // code
}
```

Note: The case value must be literal or constant, and must be unique.

Example

Write a program using switch statements to check if the input lowercase character is vowel or consonant.

Code

```
switch (ch) {
```

```
case 'a':
      System.out.println("Vowel");
      break;
case 'e':
      System.out.println("Vowel");
      break;
case 'i':
      System.out.println("Vowel");
      break;
case 'o':
      System.out.println("Vowel");
      break;
case 'u':
      System.out.println("Vowel");
      break;
default:
      System.out.println("Consonant");
}
```

Case - 1: ch = 'e'
Output - Vowel

Case - 2: ch = 'w'
Output - Consonant

Try this

Write a program to print the day name based upon the day number.
1 - Monday, 2 - Tuesday, etc.

Why Array?

If we use a traditional approach, then to store 5 values we need to create 5 variables.
Similarly to store 100 values we need to create 100 variables.
The drawback in the traditional approach is that remembering the variables names is complex, so to avoid this problem we need to use "Arrays".

What is Array?

It refers to index collection of fixed no of homogeneous data elements.
Single variable holding multiple values which improves readability of the program.

How to create an Array?

Array declarations

1. Single Dimension Array

Declaration of array

- int[] a;//recommended to use as variable is separated from type.
- int a[];
- int []a;
- int[6] a; // compile time error. we cannot specify the size.

Array Construction

Every array in java is an object hence we create using a new operator.

Example

```
int[]  a;
a=new int[5];
  or
int[] a =new int[5];
```

- For every type corresponding classes are available but these classes are part of java language but not applicable at the programmer level.

```
int[]    [I
 float[]  [F
 double[] [D
```

Rule1

At the time of Array construction compulsorily we should specify the size.

Example::

```
int[] a=new int[5];
int[] a =new int[];//ce:: array dimension is missing.
```

Rule2

It is legal to have an array with size zero.

Example::

```
int[] a =new int[0];
System.out.println(a.length);// 0
```

Rule3

If we declare an array with negative size it would result in a Negative Array size exception.

Example

```
int[] a=new int[-5]; //NegativeArraySizeException.
```

Rule4::

The allowed datatypes to specify the size are byte,short,int,char.

Example

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

byte b=10;
int[] a =new int[b];//valid
```

```
short s=25;
int[] a =new int[s];//valid

char c='A';
int[] a=new int[c];//valid

int[] a=new int[10L];//CE
int[] a=new int[3.5f];//CE
```

Rule5

The maximum allowed array size in java is the maximum value of int size.

```
int[] a=new int[2147483647]; //but valid:: OutOfMemoryError
int[] a=new int[2147483648]; //CE
```

ArrayInitialisation

Since arrays are treated as objects,internally based on the type of data we keep inside array JVM will keep default values.

Example

```
int[] a =new int[5];
        System.out.println(a);//[I@....
        System.out.println(a[0]);//0
```

Example

```
int[] a=new int[4];
        a[0]=10; a[1]=20; a[2]=30;
        System.out.println(a[3]); //0
        System.out.println(a[4]); //ArrayIndexOutOfBoundsException.
        System.out.println(a[-4]);//ArrayIndexOutOfBoundsException.
```

Shortcut of way declaration,construction,initialisation in single line

```
int[]a = {10,20,30,40};
char[] a= {'a','e','i','o','u'};
String[] a= {"sachin","ramesh","tendulkar","IND"};
```

2-D Array

2D-Array =1D-Array + 1D-Array
(ref) (data)

Declaration(All are valid)

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

ArrayConstruction

```
int[][] a =new int[3][2];
      or
int[][] a= new int[3][];
  a[0]=new int[5];
  a[1]=new int[3];
  a[2]=new int[1];
```

ArrayInitalisation

```
a[0][0] = 10;
a[2][3] = 5;
```

Tricky Question

- int[] a,b; // a-> 1D,b->1D
- int[][] a,b; //a->2D,b->1D
- int[] a[],b; //a->2D,b->1D
- int[] a[],[]b; //CE
- int[] []a,b; //a->2D,b->1D

Rule

If we want to specify the dimension, we need to specify that before the first variable,but from the second variable onwards rule is not applicable.

shortcut of working with 2-D array

```
int[][] a= {{10,20},{100,200,300},{1000}};
```

String Introduction

String it refers to an Object in java present in package called java.lang.String
String refers to collection of characters

Example

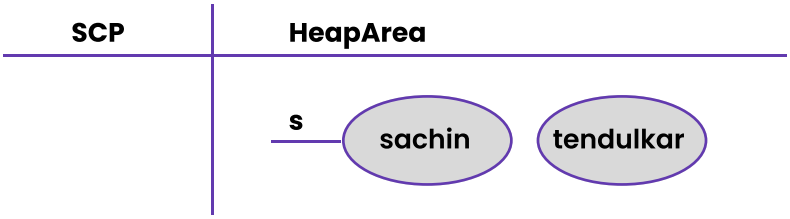
```
String s= "sachin";
System.out.println(s);//sachin

String s =new String("sachin");
System.out.println(s);//sachin
```

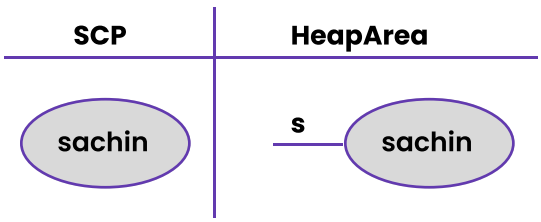
In java String object is by default immutable,meaning once the object is created we cannot change the value of the object, if we try to change then those changes will be reflected on the new object not on the existing object.

Example

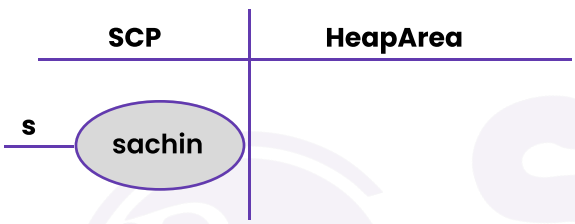
```
String s= "sachin";
s.concat("tendulkar");//(new object got created with modification so immutable)
System.out.println(s);// sachin
```



Note:
String s1 =new String("sachin");
In this case 2 objects will be created one in the heap and the other one in the String Constant Pool, the reference will always point to Heap.



String s1 ="sachin";
In this case only one object will be created in the SCP and it will be referred by our reference.



Note

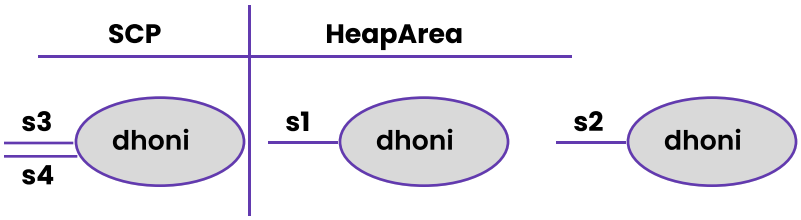
- Object creation in SCP is always optional,1st JVM will check if any object already created with required content or not.
- If it is already available then it will reuse the existing object instead of creating the new Object.
- If it is not available only then a new object will be created, so we say in SCP there is no chance of existing 2 objects with the same content.
- In SCP duplicates are not permitted.
- Garbage Collector cannot access SCP Area, Even though Object does not have any reference still object is not eligible for GC.
- All SCP objects will be destroyed only at the time of JVM ShutDown.

Example

```
String s1=new String("dhoni");
String s2=new String("dhoni");
String s3="dhoni";
String s4="dhoni";
```

Output

Two objects are created in the heap with data as "dhoni" with reference as S1,S2
One object is created in SCP with the reference as S3,S4.



Note:
Importance of SCP

- In our program if any String object is required to use repeatedly then it is not recommended to create multiple objects with same content iit reduces performance of the system and affects memory utilisation.
- We can create only one copy and we can reuse the same object for every requirement. This approach improves performance and memory utilisation. We can achieve this by using "scp".
- In SCP several references pointing to the same object the main disadvantage in this approach is by using one reference if we are performing any change the remaining references will be impacted. To overcome this problem sun people implemented immutability concept for String objects.
- According to this once we create a String object we can't perform any changes in the existing object if we are trying to perform any changes with those changes a new String object will be created hence immutability is the main disadvantage of scp.

Types of String

In java Strings are classified into 2 types

1. Mutable String
2. Immutable String

Mutable String

Once if we create a String, on that String if we try to perform any operation and if those changes get reflected in the same object then such strings are called “Mutable String”.

Example: StringBuffer, StringBuilder

Immutable String

Once if we create a String, on that String if we try to perform any operation then those changes won’t be reflected in the same object, rather a new object will be created.Such type of String is called as “Immutable String”.

Example: String

Working with String,StringBuilder,StringBuffer

Parameter	String	StringBuilder	String Buffer
mutability	immutable	mutable	mutable
Storage	String constant pool	heap	heap
Thread Safety	Not used in the threaded environment as it is immutable	Not thread-safe so it is used in a single-threaded environment	Thread-safe so it is used in a multi-threaded environment
Speed	Comparably slowest	Comparably fastest	Faster than String butu Slower than StringBuilder