**Operators and Statements**

* Three flavors of operators are available in Java: unary, binary, and ternary.
* Java operators are not necessarily evaluated from left-to-right order.
* If two operators have the same level of precedence, then Java guarantees left-to-right evaluation
* Table for Order of operator precedence

**Operator Symbols and examples**

Post-unary operators expression**++**, expression**--**

Pre-unary operators **++**expression, **--**expression

Other unary operators +, -, !

Multiplication/Division/Modulus \*, /, %

Addition/Subtraction +, -

Shift operators <<, >>, >>>

Relational operators <, >, <=, >=, instanceof

Equal to/not equal to ==, !=

Logical operators &, ^, |

Short-circuit logical operators &&, ||

Ternary operators boolean expression ? expression1 : expression2

Assignment operators =, +=, -=, \*=, /=, %=, &=, ^=, !=, <<=, >>=, >>>=

* Operators Detail
  + **>> (Signed right shift)**
  + **<< (Signed left shift)**
  + **>>> (UnSigned right shift) 🡪**Number is stored using 32 bit 2's complement form before shift, For ex. Binary representation of -1 is all 1s (111..1).
* **Arithmetic Operators**
  + **Numeric Promotion**

Primitive Datatype of Result = max(int, TypeOF val1, TypeOF val2,…)

* + Unary Operators

**Unary operator Description**

+ Indicates a number is positive, although numbers are assumed to be positive in Java unless accompanied by a negative unary operator

- Indicates a literal number is negative or negates an expression

++ Increments a value by 1

-- Decrements a value by 1

! Inverts a Boolean’s logical value

* + Example

int x = 3;

int y = ++x \* 5 / x-- + --x;

Ans: x=2, y=7

* + **Compound Assignment Operators (** += and -=)
  + **Relational Operators( <, <=, >, >=)**
  + Relational instanceof operator

**a instanceof b** True if the reference that a points to is an instance of a class, subclass, or class that implements a particular interface, as named in b.

* The *logical operators*, (&), (|), and (^), may be applied to both numeric and boolean data types(For Numeric they are referred to as *logical operators whereas for Boolean* referred to as *bitwise operators*.)
* conditional operators, && and ||, which are often referred to as short-circuit operators identical to the logical operators,& and | **( except that the right-hand side of the expression may never be evaluated if the final result can be determined by the left-hand side of the expression )**
* A more common example of where short-circuit operators are used is checking for null objects before performing an operation,

if(x != null && x.getValue() < 5) {

// Do something}

* Equality operators (== and !=) are used in one of three scenarios:
  + Comparing two numeric primitive types.
  + Comparing two boolean values.
  + Comparing two objects, including null and String values.

========================XXX================================================

**Methods and Encapsulation**

**Access Modifiers**

* Java offers four choices of access modifier:
  + ***public*** The method can be called from any class.
  + ***private*** The method can only be called from within the same class.
  + ***protected*** The method can only be called from classes in the same package or subclasses.
  + ***Default (Package Private) Access*** The method can only be called from classes in the same package. This one is tricky because there is no keyword for default access. You simply omit the access modifier.

Working with Varargs

* A vararg parameter must be the last element in a method’s parameter list.
* you are only allowed to have one vararg parameter per method.
* Java will create an empty array if no parameters are passed for a vararg.
* It is still possible to pass null explicitly to varargs, Java treats it as an array reference that happens to be null.

**Designing Static Methods and Fields**

* main() can be called just like any other static method.
* Regular imports are for importing classes. Static imports are for importing static members of classes, you can use a wildcard or import a specific member.

Overloading Methods

* *Method overloading* occurs whenthere are different method signatures with the same name but different type parameters and different no of parameters.
* public void fly(int[] lengths) { } and public void fly(int... lengths) { } are treated same by compiler so can not be compile for method overloading.
* We can call either method by passing an array: fly(new int[] { 1, 2, 3 });
* But can only call the varargs version with stand-alone parameters:fly(1, 2, 3);

Creating Constructors

* A constructor is typically used to initialize instance variables.

**Order of Initialization**

1. If there is a superclass, initialize it first .
2. Static variable declarations and static initializers in the order they appear in the file.
3. Instance variable declarations and instance initializers in the order they appear in the file.
4. The constructor.

Encapsulating Data

* Encapsulation means we set up the class so only methods in the class with the variables can refer to the instance variables.

**Creating Immutable Classes**

**public class ImmutableSwan**

**{**

**private int numberEggs;**

**public ImmutableSwan(int numberEggs) {**

**this.numberEggs = numberEggs;**

**}**

**public int getNumberEggs() {**

**return numberEggs;**

**}**

**}**

* Defensive copy of mutable class

**public Mutable(StringBuilder b) {**

**builder = new StringBuilder(b);**

**}**

**public StringBuilder getBuilder() {**

**return new StringBuilder(builder);**

**}**

* Another approach for the getter is to return an immutable object:

**public String getValue() {**

**return builder.toString();**

**}**

Writing Simple Lambdas

* A *lambda expression* is a block of code that gets passed around. You can think of a lambda expression as an *continued* anonymous method.
* In other words, a lambda expression is like a method that you can pass as if it were a variable.
* Syntax of Lambda expression

****

Or



* The parentheses can only be omitted if there is a single parameter and its type is not explicitly stated.
* Examples

3: print(() -> true); // 0 parameters

4: print(a -> a.startsWith("test")); // 1 parameter

5: print((String a) -> a.startsWith("test")); // 1 parameter

6: print((a, b) -> a.startsWith("test")); // 2 parameters

7: print((String a, String b) -> a.startsWith("test")); // 2 parameters

* Java 8 even integrated the Predicate interface into some existing classes. There is only one you need to know for the exam. ArrayList declares a removeIf() method that takes a Predicate.
* Example

List<String> bunnies = new ArrayList<>();

4: bunnies.add("long ear");

5: bunnies.add("floppy");

6: bunnies.add("hoppy");

7: System.out.println(bunnies); // [long ear, floppy, hoppy]

8: bunnies.removeIf(s -> s.charAt(0) != 'h');

9: System.out.println(bunnies); // [hoppy]

============XXX==================

**3. Core Java APIs**

Creating and Manipulating Strings

* Creating String

String name = "Fluffy";

String name = new String("Fluffy");

* **Concatenation**
  + + operator can be used in two ways within the same line of code

1. If both operands are numeric, + means numeric addition.
2. If either operand is a String, + means concatenation.
3. The expression is evaluated left to right.

* **Immutability**
  + Example

final class Immutable {

private String s = "name";

public String getS() { return s; }

}

* **The String Pool**
  + The *string pool*, also known as the intern pool, is a location in the Java virtual machine (JVM) that collects all these strings.
  + Two imp scenario

String name = "Fluffy"; 🡪Use the string pool normally

String name = new String("Fluffy"); 🡪 “No, JVM. I really don’t want you to use the string pool. Please create a new object or me even though it is less efficient”.

* **Important String Methods**
  + string is a sequence of characters and Java counts from 0 when indexed.

1. ***length() 🡪*** returns the number of characters in the String.
2. ***charAt()🡪****find* out what character is at a specific index.
3. ***indexOf()🡪*** looks at the characters in the string and fi nds the fi rst index that matches the desired value. indexOf can work with an individual character or a whole String as input.

doesn’t throw an exception if it can’t find a match, instead it returns -1.

Examples:-

System.out.println(string.indexOf('a')); // 0

System.out.println(string.indexOf("al")); // 4

System.out.println(string.indexOf('a', 4)); // 4 🡪start looking at char from index 4.

System.out.println(string.indexOf("al", 5)); // -1

1. ***substring()🡪***looks for characters in a string.

int substring(int beginIndex)

int substring(int beginIndex(including), int endIndex(excluding))

1. ***toLowerCase()* and *toUpperCase()***
2. ***equals()* and *equalsIgnoreCase()***
3. ***startsWith()* and *endsWith()***
4. ***contains()***
5. ***replace()***

String replace(char oldChar, char newChar)

String replace(CharSequence oldChar, CharSequence newChar)

1. ***trim() : public String trim()***

* **Method Chaining**

Example:

String result = "AniMaL ".trim().toLowerCase().replace('a', 'A');//Animal

Using the *StringBuilder* Class

* The StringBuilder class creates a String without storing all those interim String values
* Mutability and Chaining: When we chained String method calls, the result was a new String with the answer. Instead, the StringBuilder changes its own state and returns a reference to itself!
* Important example:

4: StringBuilder a = new StringBuilder("abc");

5: StringBuilder b = a.append("de");

6: b = b.append("f").append("g");

7: System.out.println("a=" + a); //abcdefg

8: System.out.println("b=" + b);//abcdefg

* **Creating a *StringBuilder***

StringBuilder sb1 = new StringBuilder();

StringBuilder sb2 = new StringBuilder("animal");

StringBuilder sb3 = new StringBuilder(10); //Default capacity is 16

* **Size vs. Capacity 🡪** Size is the number of characters currently in the sequence, and capacity is the number of characters the sequence can currently hold.
* Java automatically increase the capacity of StringBuilder object when it is required(Size exceeding Capacity value).
* **Important *StringBuilder* Methods**
  + ***charAt()*, *indexOf()*, *length()*, and *substring()***

Imp Example:

StringBuilder sb = new StringBuilder("animals");

String sub = sb.substring(sb.indexOf("a"), sb.indexOf("al"));

int len = sb.length();

char ch = sb.charAt(6);

System.out.println(sub + " " + len + " " + ch);// anim 7 s

* + ***append() : StringBuilder append(String str)***
  + ***insert():StringBuilder insert(int offset, String str)***
  + ***delete()* and *deleteCharAt() :*** ***StringBuilder delete(int start, int end)***

***StringBuilder deleteCharAt(int index)***

* + ***reverse():StringBuilder reverse()***
  + ***toString() : String toString()***
* Understanding Equality
* Imp Example

String x = "Hello World";

String z = " Hello World".trim();

System.out.println(x == z); // false

Understanding Java Arrays

* An array is an area of memory on the heap with space for a designated number of elements.
* Creating Array

int[] numbers1 = new int[3];

int[] numbers2 = new int[] {42, 55, 99};

int[] numbers2 = {42, 55, 99}; 🡪 This approach is called an anonymous array.

* Valid Array Declaration

int[] numAnimals;

int [] numAnimals2;

int numAnimals3[];

int numAnimals4 [];

* **Multiple “Arrays” in Declarations**

int[] ids, types; 🡪 Both var of type int[] .

int ids[], types; 🡪 one variable of type int[] and one variable of type int.

* **Creating an Array with Reference Variables**