**DAY 1 AND 2 ASSIGNMENT**

**1. What are the different primitive data types available in Java?**

The eight primitive data types that Java offers are byte, short, int, long, float, double, char, and boolean.

byte: signed 8-bit integer with a range of -128 to 127

short: 16-bit signed integer with a range of -32,768 to 32,767

int: 32-bit signed integer with a range of -2,147,483,648 to 2,147,483,647

long: 64-bit signed integer, with a range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

float: 32-bit IEEE 754 floating-point, frequently used for fractional numbers (approximately 6-7 decimal places of precision).

double: IEEE 754 floating-point, 64-bit, usually used for decimal values (approximately 15–16 decimal digits).

char: a 16-bit Unicode character that represents a single character and has a range of 0 to 65,535

boolean: denotes a single bit of data with only two possible values: true or false (the size depends on the JVM)

2. **Explain the difference between primitive and non-primitive data types in Java.**

Primitive Data Types:

* Built into the Java language (e.g., int, boolean, double).
* Store actual values directly in memory.
* Always have a value—cannot be null.
* Named with lowercase letters.
* Fixed size, faster operations, stored on the stack.
* Examples: int, byte, short, long, float, double, char, boolean.

Non-Primitive Data Types:

* Also called reference types; store references (memory addresses) to objects in heap memory.
* Can be assigned null (do not always have a value).
* Typically start with an uppercase letter (e.g., String, Array, Class).
* Size can change, and operations are slower compared to primitives.
* Created by users except for String.
* Examples: String, Array, Class, Interface, Object

**4. What is type casting? Provide an example of implicit and explicit casting in Java**

Type casting in Java is the process of converting a variable from one data type to another.

**Implicit Type Casting (Widening)**

* **Definition**: Automatically performed by Java when converting a smaller data type to a larger data type.
* **Why**: No risk of data loss; the target type can always accommodate the value.
* **Example**:

java

**int** myInt = 10;

**double** myDouble = myInt;

System.out.println(myInt);

System.out.println(myDouble);

**Explicit Type Casting (Narrowing)**

* **Definition**: Manually performed by the programmer, converting a larger data type to a smaller type. May lead to loss of data.
* **Why**: The target type cannot always represent the source value, so explicit instruction is needed.
* **Example**:

java

**double** myDouble = 10.99;

**int** myInt = (**int**) myDouble;

System.out.println(myDouble);

System.out.println(myInt);

**5. What is the default value of each primitive data type in Java?**

* Default value of float is 0.0f
* Default value of integer is 0L
* Default value of char is \u0000
* Default value of Boolean is false

**6. What are control statements in Java? List the types with examples.**

Java control statements are used to regulate a program's execution flow. They make it possible to repeat code blocks, make decisions, and change the order in which things are executed depending on specific circumstances. For applications to be dynamic and adaptable, control statements are necessary. In Java, control statements come in three primary varieties:

1. Statements of Decision-Making

If a condition is true, the if statement runs code.

if (age >= 18)

{

System.out.println("Adult");

}

If the condition is true, the if-else statement executes one block; if it is false, it executes another.

if (score >= 50) {

System.out.println("Pass");

} else

{

System.out.println("Fail");

}

ladder of if-else-if: tests several conditions one after the other.

if (marks >= 90)

{ System.out.println("Grade A");

}

else if (marks >= 80) {

System.out.println("Grade B");

}

else {

System.out.println("Grade C");

}

A switch statement compares a variable to several precise values.

switch(day) {

case 1:

System.out.println("Monday");

break;

case 2:

System.out.println("Tuesday");

break;

default:

System.out.println("Invalid day");

}

2. Statements that Loop (Iterate)

These enable a block of code to be repeated while a condition is true.

A block is repeated a predetermined number of times using a for loop.

for (int i = 0; i < 5; i++)

{

System.out.println("Iteration " + i);

}

while loop: repeats as long as a condition is met.

int i = 0;

while(i < 5) {

System.out.println("Iteration " + i);

i++;

}

Do-while loop: repeats while the condition is true after executing the block at least once.

int i = 0;

do { System.out.println("Iteration " + i);

i++;

} while(i < 5);

Statements of Jump   
These jump to a different section of the code, changing the flow.  
  
  
Break statement: Closes the closest switch or loop.   
  
 for (int i = 0; i < 10; i++)

{   if (i == 5)

break;

  System.out.println(i);

}   
statement to continue: moves on to the following loop iteration, skipping the current one.   
  
 for (int i = 0; i < 10; i++)

{

if (i == 5) continue;

System.out.println(i);

}   
statement of return: returns a value if desired and exits the current method.   
  
public int add(int a, int b) {

return a + b;

}

**7. What is the difference between break and continue statements?**

* Break Statement: The break statement is used to immediately terminate the entire loop or switch statement in which it appears. When a break is encountered, the control exits the loop completely, and the program continues with the first statement after the loop.
* Continue Statement: The continue statement is used to skip the current iteration of the loop and immediately jump to the next iteration. It does not terminate the loop but skips the remaining code inside the loop body for that particular iteration.

**8. What are keywords in Java? List 10 commonly used keywords.**

 10 commonly used Java keywords:

1. class – Declares a class.
2. public – An access specifier indicating that a member is accessible from any other class.
3. static – Indicates that a method or variable belongs to the class rather than instances of the class.
4. void – Specifies that a method does not return any value.
5. if – Used to perform conditional branching.
6. else – Defines an alternative branch in an if statement.
7. for – Used to create a for loop for iteration.
8. while – Used to create a while loop for iteration.
9. return – Exits from a method and optionally returns a value.
10. break – Terminates the innermost loop or switch statement.

**9. Explain the purpose of the following keywords: static, final, this, super**.

**static**

The keyword **static** is used to define class-level members (variables or methods) that belong to the class itself rather than any specific instance. Static members can be accessed without creating an object of the class.

**final**

The keyword **final** is used to declare constants or prevent modification. A final variable’s value cannot be changed once initialized, a final method cannot be overridden by subclasses, and a final class cannot be subclassed

**this**

The keyword **this** is a reference to the current object instance within a class. It is commonly used to differentiate between instance variables and parameters with the same name or to invoke other constructors in the same class.

**super**

The keyword **super** refers to the parent class of the current object. It allows accessing superclass methods, constructors, and variables, especially when overridden or hidden by the subclass.

**10. What are the types of operators in Java?**

1. **Arithmetic Operators** – Used for basic mathematical operations such as addition (+), subtraction (-), multiplication (\*), division (/), and modulus (%).
2. **Unary Operators** – Operate on a single operand, examples include increment (++), decrement (--), unary plus (+), unary minus (-), and logical complement (!).
3. **Assignment Operators** – Used to assign values to variables; examples include =, +=, -=, \*=, /=, and %=.
4. **Relational Operators** – Used to compare two values and return a boolean result, such as equal to (==), not equal to (!=), greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=).
5. **Logical Operators** – Used to combine multiple boolean expressions, including AND (&&), OR (||), and NOT (!).
6. **Ternary Operator** – A shorthand for if-else statements, represented as ? :.
7. **Bitwise Operators** – Perform operations on individual bits within integral data types, including AND (&), OR (|), XOR (^), complement (~).
8. **Shift Operators** – Shift bits left or right, such as left shift (<<), right shift (>>), and unsigned right shift (>>>).
9. **Instanceof Operator** – Checks whether an object is an instance of a specific class or implements an interface.

**11. What is operator precedence? How does it affect the outcome of expressions?**

Operator precedence in Java refers to the rules that determine the order in which different operators in an expression are evaluated. When an expression contains multiple operators, the operators with higher precedence are evaluated before those with lower precedence.

**12. How does Java handle overflow and underflow with numeric types?**

For **integer types** (byte, short, int, long), overflow happens when a calculation exceeds the maximum value allowed by the data type, causing the value to wrap around to the minimum value.

Similarly, underflow occurs when a value goes below the minimum limit and wraps back to the maximum.

For **floating-point types** (float, double), overflow results in special values like positive or negative infinity, while underflow can lead to the value being rounded to zero. When numbers exceed the floating-point range, they are represented as infinity, while very small numbers approaching zero may be rounded down to zero without errors.

In summary, Java handles numeric overflow and underflow by wrapping integer values within their fixed range and representing floating-point overflow as infinity and underflow as zero, without throwing exceptions.

**13. What is the difference between char and String in Java**

Char is character represented in single quotes i.e ‘ ‘’

String is represented in double quotes i.e “ “

**14. Explain wrapper classes and their use in Java.**

Wrapper classes in Java are special classes that encapsulate (or "wrap") primitive data types in an object. Each primitive data type such as int, char, float, etc., has a corresponding wrapper class like Integer, Character, Float, and so on. These wrapper classes allow primitive values to be treated as objects.

The main uses of wrapper classes are:

* **Representing primitives as objects:** Since Java's collections (like ArrayList, HashMap) can only store objects, wrapper classes enable storing primitive data types in these collections by converting them into objects.
* **Autoboxing and Unboxing:** Java can automatically convert primitives to their wrapper objects (autoboxing) and vice versa (unboxing), simplifying code that mixes primitive and object types.
* **Nullability:** Unlike primitives, wrapper objects can be assigned null, which is useful for representing the absence of a value.
* **Utility Methods:** Wrapper classes provide useful methods for converting between types (e.g., parsing strings to numbers), comparing values, and other operations.
* **Immutability and Thread Safety:** Wrapper objects are immutable, meaning their value cannot be changed once created, which is useful for safe concurrent programming.

**15. How do you exit from nested loops in Java**

In Java, to exit from nested loops, you can use a labeled break statement.

1. Define a label before the outer loop followed by a colon (labelName:).
2. Use the break statement with the label name inside the inner loop to exit the labeled outer loop.

**16. Explain the difference between `==` and `.equals()` in Java.**

* The == operator compares reference equality for objects. It checks whether two reference variables point to the exact same object in memory. For primitive types (like int, char), it compares the actual values.
* The .equals() method compares content equality. It checks whether two objects are logically "equal" based on the contents or state defined by the class. Many classes (like String, Integer) override .equals() to compare the actual data rather than memory addresses.

**17. What is the use of `this` and `super` in method overriding?**

* this keyword: In the context of method overriding, this refers to the current object — the instance of the subclass where the method is being executed. It is used to call other methods or access fields within the same class, including the overridden methods.
* super keyword: The super keyword is used to refer explicitly to the superclass of the current class. In method overriding, super allows you to invoke the superclass version of an overridden method.

**18**. **Explain bitwise operators with examples.**

Bitwise operators in Java perform operations directly on the individual bits of integer data types (byte, short, int, long). These operators are useful for low-level programming and bit manipulation tasks. Here are the main bitwise operators with examples:

1. **Bitwise AND (&)**: Performs a logical AND on each pair of corresponding bits of two numbers. The result bit is 1 only if both bits are 1; otherwise, it's 0.

**int** a = 5;

**int** b = 7;

**int** c = a & b;

1. **Bitwise OR (|)**: Performs a logical OR on each pair of bits. The result bit is 1 if at least one bit is 1.

**int** a = 5;

**int** b = 7;

**int** c = a | b;

1. **Bitwise XOR (^)**: Performs a logical exclusive OR on each pair of bits. The result bit is 1 if the bits are different, 0 if they are the same.

**int** a = 5;

**int** b = 7;

**int** c = a ^ b;

1. **Bitwise Complement (~)**: A unary operator that inverts all bits of the number (0 to 1, and 1 to 0). The result is the one's complement.

**int** a = 5;

**int** c = ~a;

1. **Left Shift (<<)**: Shifts bits to the left by a specified number of positions, filling with zeros on the right.

**int** a = 5;

**int** c = a << 2;

1. **Signed Right Shift (>>)**: Shifts bits to the right by a specified number of positions, preserving the sign bit (leftmost bit).

**int** a = -20;

**int** c = a >> 2;

1. **Unsigned Right Shift (>>>)**: Shifts bits to the right, filling leftmost bits with zeros regardless of the sign.

**int** a = -20; **int** c = a >>> 2;