1. (a) Call by Value vs. Call by Reference & Swap Program

Two Basic Differences:

Feature	Call by Value	Call by Reference
Parameter Passing	A copy of the actual value is passed to the function.	The memory address (reference) of the actual variable is passed to the function.
Effect on Original	Changes made to the parameter inside the function **do not affect** the original variable.	Changes made to the parameter inside the function **directly affect** the original variable.

C++ Program to Swap Two Integers using Call by Reference:

```cpp
#include <iostream>
using namespace std;

// Function to swap values using call by reference
void swap(int &x, int &y) {
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 10, b = 20;

    cout << "Before swap: a = " << a << ", b = " << b << endl;

    // Calling the swap function
    swap(a, b);

    cout << "After swap: a = " << a << ", b = " << b << endl;

    return 0;
}
```

Output:

```text
```

```
Before swap: a = 10, b = 20
After swap: a = 20, b = 10
```

Explanation: The `&` in the function parameter `int &x` denotes a reference. When `a` and `b` are passed to `swap`, `x` and `y` become aliases for `a` and `b` respectively. Therefore, swapping `x` and `y` directly swaps the values of `a` and `b` in the `main` function.

1. (b) Code Snippet Output

Code:

```
c

int a = 5, b = 18, c = 27;
a = b++ + ++c;
b = a++ - c--;
c = a++ + b--;
```

Step-by-Step Evaluation:

1. **Initial Values:** `a = 5`, `b = 18`, `c = 27`
2. **Statement 1:** `a = b++ + ++c;`
   - `b++` is post-increment: value used is 18, then `b` becomes 19.
   - `++c` is pre-increment: `c` becomes 28 first, then value used is 28.
   - So, `a = 18 + 28 = 46`.
   - **Current Values:** `a = 46`, `b = 19`, `c = 28`
3. **Statement 2:** `b = a++ - c--;`
   - `a++` is post-increment: value used is 46, then `a` becomes 47.
   - `c--` is post-decrement: value used is 28, then `c` becomes 27.
   - So, `b = 46 - 28 = 18`.
   - **Current Values:** `a = 47`, `b = 18`, `c = 27`
4. **Statement 3:** `c = a++ + b--;`
   - `a++` is post-increment: value used is 47, then `a` becomes 48.
   - `b--` is post-decrement: value used is 18, then `b` becomes 17.
   - So, `c = 47 + 18 = 65`.
   - **Final Values:** `a = 48`, `b = 17`, `c = 65`

Final Output:

```
text

a = 48, b = 17, c = 65
```

1. (c) Polymorphism in Java

What is Polymorphism?
Polymorphism is a fundamental Object-Oriented Programming (OOP) concept that allows an object to take on many forms. It lets us perform a single action in different ways. The word itself means "many forms." In Java, it is primarily achieved through:

1. **Compile-time Polymorphism (Static):** Achieved by **Method Overloading**. The decision about which method to call is made by the compiler at compile time.
2. **Run-time Polymorphism (Dynamic):** Achieved by **Method Overriding**. The decision about which method to call is made by the JVM at runtime based on the actual object type.

Java Program Demonstrating Both:

```
java

// Parent class
class Animal {
    // Method Overloading (Compile-time Polymorphism)
    public void sound() {
        System.out.println("Animal makes a sound");
    }
    public void sound(String type) { // Same method name, different parameter
        System.out.println("This " + type + " animal makes a sound");
    }
}

// Child class
class Dog extends Animal {
    // Method Overriding (Run-time Polymorphism)
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}

public class PolymorphismDemo {
    public static void main(String[] args) {
        // --- Compile-time Polymorphism Demo ---
```

```
        Animal myAnimal = new Animal();
        myAnimal.sound();        // Calls sound()
        myAnimal.sound("wild");  // Calls sound(String type)

        System.out.println(); // Blank line for clarity

        // --- Run-time Polymorphism Demo ---
        Animal myDog = new Dog(); // Parent reference, Child object
        myDog.sound(); // JVM decides at runtime to call Dog's sound()
    }
}
```

◣◣

Output:

```
text

Animal makes a sound
This wild animal makes a sound

Dog barks
```

◣◣

Explanation:

- **Compile-time:** The compiler sees the different parameters in `sound()` and `sound(String type)` and binds the call at compile time.
- **Run-time:** The reference variable `myDog` is of type `Animal`, but it points to a `Dog` object. At runtime, the JVM identifies the object type as `Dog` and executes the overridden `sound()` method in the `Dog` class.

1. (d) Just In Time (JIT) Compiler

What is a JIT Compiler?
A Just-In-Time (JIT) compiler is a component of the Java Runtime Environment (JRE) that improves the performance of Java applications by compiling bytecode into native machine code at **run time**.

How it supports Machine Independent Programming:

1. **The Foundation: Write Once, Run Anywhere (WORA):** Java achieves platform independence by compiling source code (`.java`) into an intermediate, platform-neutral format called **bytecode** (`.class`). This bytecode is not specific to any physical machine; it's for the conceptual **Java Virtual Machine (JVM)**.
2. **The Performance Problem:** An **interpreter** inside the JVM can execute this bytecode line-by-line, which is slow compared to code compiled directly to native machine code (like C++).

3. **The JIT Solution:**
    - The JIT compiler works alongside the JVM. As the Java program runs, the JVM initially uses the interpreter to execute the bytecode.
    - The JIT compiler monitors the application's execution (this is part of the Java HotSpot Technology). It identifies "hot spots" – sections of code that are executed frequently (e.g., loops, commonly called methods).
    - When a method is identified as a hot spot, the JIT compiler **dynamically compiles** the entire bytecode for that method into the native machine code of the host system.
    - This newly compiled native code is then cached. Subsequent calls to that method execute the highly efficient native code directly, bypassing the slower interpretation step.

The Connection to Machine Independence:
The JIT compiler perfectly complements the "Write Once, Run Anywhere" model. A developer writes code once, and it is compiled to bytecode once. This same bytecode can run on any system with a JVM.

- On a Windows machine, the JIT compiles it to x86 machine code.
- On a Mac, the JIT compiles it to x64 machine code.
- On a different architecture, the JIT compiles it to that specific native code.

The JIT compiler **bridges the gap** between platform-independent bytecode and the high performance of native execution, providing machine independence without sacrificing speed.