Question 1

```
public class Demo {
    void main() {
        System.out.println("JAVA");
    }

    static void main(String args) {
        System.out.println("Spring");
    }

    public static void main(String[] args) {
        System.out.println("Hibernate");
    }

    void main(Object[] args) {
        System.out.println("Apache Camel");
    }
}
```

Output:

```
Hibernate
```

✅ Explanation:

- JVM **always starts execution** from `public static void main(String[] args)`.
- Other methods named `main` are just **overloaded methods** and will not be called unless explicitly invoked.
- So only `"Hibernate"` will print.

Answer: 1. Hibernate

Question 2

```
class Employee {
    public final void show() {
        System.out.println("show() inside Employee");
    }
}

final class Unit extends Employee {
    public void show1() {
        final int x = 100;
        System.out.println("show() inside Unit");
        System.out.println(x);
    }
}

public class Demo11 {
    public static void main(String[] args) {
        Employee employee = new Unit();
        new Unit().show1();
    }
}
```

Output:

```
show() inside Unit
100
```

Explanation:

- `Employee employee = new Unit();` → creates object of `Unit`, but only `Employee` reference.
- Then `new Unit().show1();` → executes `show1()` of `Unit`.
- Prints:
    - `"show() inside Unit"`
    - `"100"`

Answer: A. Show() inside Unit + 100

Question 3 (Bank Example)

```
public class Bank {
    static class Customer {
        public void go() {
            System.out.println("Inside Customer");
        }
    }

    static class Account extends Customer {
        public void go() {
            System.out.println("Inside Account");
        }
    }

    static class Branch extends Customer {
        @Override
        public void go() {
            System.out.println("Inside Branch");
        }
    }

    public static void main(String[] args) {
        Customer customer = new Account();
        Branch branch = (Branch) customer; // Line 1
        branch.go();
    }
}
```

Explanation:

- `customer` is an `Account` object.
- Casting `Account` → `Branch` is **not valid** since `Account` is not a subclass of `Branch`.
- So, runtime throws `ClassCastException`.

Answer: 5. An exception is thrown at runtime

Question 4 (Game Example)

```
public class Game {
   public static void main(String[] args) {
      displayRegistration("Hockey");              // Line 1
      displayRegistration("Kho-Kho", 132, 102, 36); // Line 2
   }

   public static void displayRegistration(String gameName, int... id) {
      System.out.println("Registration for " + gameName + ":");
      for (int i = 0; i < id.length; i++) {
         System.out.print(id[i] + " ");
      }
      System.out.println();
   }
}
```

Output:

```
Registration for Hockey:
Registration for Kho-Kho:
132 102 36
```

Answer: 3. Registration for Hockey: (blank) and Registration for Kho-Kho: 132 102 36

Key points

1. **Varargs syntax (`int... id`)**
   - `int... id` means the method can take **0 or more integers**.
   - It's a shorthand for creating a method that can accept multiple integers without explicitly defining an array every time.
2. **Method call examples**
   - `displayRegistration("Hockey")` → no IDs passed, so `id.length == 0`.
   - `displayRegistration("Kho-Kho", 132, 102, 36)` → three IDs passed, so `id` behaves like an array `[132, 102, 36]`.

3. **For loop prints all IDs**
    - `for (int i = 0; i < id.length; i++)` iterates over the variable-length integer arguments and prints them.

Question 5 (Interface Example)

```
public interface InterfaceDemo {
    // Line 1
}
```

Options:

1. `void display(int x);` ✅ valid abstract method
2. `void display(int x) { }` ❌ not allowed in interface (before Java 8)
3. `public static void display(int x) { }` ❌ static methods allowed only with body (Java 8+), must have `default` or `static` properly
4. `default void display(int x) { }` ✅ valid since Java 8
5. `public interface Demo { }` ✅ allowed (nested interface)

Answer: 1, 4, 5

Question 6 (Employee/Main Example)

```
class Employee {
    void disp(char c) {
        System.out.print("Employee name starts with : " + c + ". ");
        System.out.print("His experience is : 11 years. ");
    }
}

class Main extends Employee {
    void disp(char c) {
        super.disp(c);
        System.out.print("Another employee name also starts with : " + c + ". ");
        new Employee().disp('D');
        disp(7);
    }

    String disp(int c) {
        System.out.print("His experience is :" + c);
        return "Bye";
    }
}

public class Demo11 {
    public static void main(String[] a) {
        Employee emp = new Main();
        emp.disp('S');
    }
}
```

Output:

```
Employee name starts with : S. His experience is : 11 years.
Another employee name also starts with : S.
Employee name starts with : D. His experience is : 11 years.
His experience is :7
```

Answer: 1

Question 7 (Dog–Cat–BullDog Example)

```
class Dog {
    void show() {
        System.out.print("Dog");
    }
}
```

```
class Cat {
    void show() {
        System.out.print("Cat");
    }
}

class BullDog extends Dog {
    void show() {
        System.out.print("BullDog");
    }
}

public class Test {
    public static void main(String[] args) {
        System.out.print("Implementing type Casting");
        Dog d = new Dog();
        BullDog bd = (BullDog) d; // invalid downcast
        bd.show();
    }
}
```

Explanation:

- `d` is a `Dog`, not a `BullDog`.
- Casting a superclass object to a subclass causes `ClassCastException`.

Answer: 3. Runtime Error

Question 8 (ExcepDemo)

```
public class ExcepDemo {
    public static void main(String[] args) {
        try {
            method();
            System.out.print("Inside try");
        } catch (RuntimeException ex) {
            System.out.print("Inside catch(RuntimeException)");
        } catch (Exception ex1) {
            System.out.print("Inside catch(Exception)");
        } finally {
            System.out.print("finally");
        }
        System.out.print("end");
    }

    public static void method() {
        // Line 26
        throw new RuntimeException();
    }
}
```

Output:

```
Inside catch(RuntimeException) finally end
```

Answer: 1. throw new RuntimeException();

Question 9 (ExceptionInClass)

```
public class ExceptionInClass {
    int data = 10;

    void calculate() throws Exception {
        try {
            data++;
            try {
                data++;
                // Line 12
                try {
                    data++;
                    throw new Exception();
                } catch (Exception ex) {
                    data++;
                    throw new Exception();
                }
            } catch (Exception ex) {
                data++;
            }
        } catch (Exception ex) {
            data++;
        }
    }

    void display() {
        System.out.println(data);
    }

    public static void main(String[] args) throws Exception {
        ExceptionInClass e = new ExceptionInClass();
        e.calculate();
        e.display();
    }
}
```

- Start: `data = 10`
- Outer `try`: `data = 11`
- Inner `try`: `data = 12`
- At Line12 fragment → increments and throws, caught → `data = 13`, rethrow → outer catch → `data = 14`
- Final catch again adds → `data = 15`

Answer: 1. The first option (nested try–catch with increments and rethrow).

Perfect 👍 Let's again **reformat each program properly** and then solve with **clear explanations**.

Question 10

```
import java.util.regex.Matcher;

import java.util.regex.Pattern;


public class Demo1 {

    public static void main(String[] args) {

        Pattern pattern = Pattern.compile("x*y");

        Matcher match = pattern.matcher("y");

        Boolean boolean1 = match.matches();

        System.out.println(boolean1);

    }

}
```

Explanation

- Regex `"x*y"` means **zero or more `x` followed by `y`**.
- `"y"` matches this rule (0 `x` + 1 `y`).
- So `boolean1 = true`.

Answer: a. True

Question 11

```
public class Operator {

    public static void main(String[] args) {
        float val1 = 5.3f;

        float val2 = 2.3f;

        float result = val1 % val2;

        System.out.println(result);

    }

}
```

✅ Explanation

- `%` (modulus) operator works with integers, floats, and doubles.
- `5.3 % 2.3 = 0.7000003` (approximation due to floating-point precision).
- Code compiles and runs fine.

Answer: a. Code compiles, runs and produces 0.7000003

Question 12

```
class Light {

    Boolean isOn;


    void turnOn() {

        isOn = true;

    }
    void turnOff() {
        isOn = false;
    }
}

class LightDemo {
    public static void main(String[] args) {
        Light light1 = new Light();
        Light light2 = new Light();

        light1.turnOn();
        System.out.println(light1.isOn);

        light1.turnOff();
        System.out.println(light1.isOn);

        System.out.println(light2.isOn);
    }
}
```

✅ Explanation

- `light1.turnOn()` → `isOn = true`

- Print → `true`
- `light1.turnOff()` → `isOn = false`
- Print → `false`
- `light2` never initialized → default `null`

Answer: True, False, null

Question 13

```
public class ABC {
    public static void main(String[] args) {
        Boolean flag = false;


        if (flag = true) { // assignment, not comparison
            System.out.println("true");
        }
        if (flag = false) { // assignment again
            System.out.println("false");
        }
    }
}
```

Explanation

- `flag = true` → assigns `true`, condition true → prints `"true"`.
- `flag = false` → assigns `false`, condition false → no print.

Answer: a. True

Question 14

```
class Employee {

    int employeeId;


    Double getEmployeeId() {

        System.out.println("Employee Id");

        return employeeId; // int returned where Double expected

    }

}
```

Explanation

- `employeeId` is `int`.
- Method return type is `Double`. Implicit conversion from `int` to `Double` not allowed.
- Compile-time error.

Answer: b. Return type should be int, not Double

Question 15

```
public class Test {
    public void method() {
        for (int i = 0; i < 3; i++) {
            System.out.print(i);
        }
    }

    public static void main(String[] args) {
        method();   // invalid, method() is not static
        print(i);   // invalid, i is out of scope
    }
}
```

Explanation

- `method()` is non-static → cannot call directly inside static context.
- `i` not visible in `main`.
- Compilation error.

Answer: c. Compilation fails

Question 16

```
enum Customer {

    private CUSTID,     // ❌ enums cannot have private/protected/public variables directly

    public CUSTNAME,

    protected ADDRESS;

}
```

Explanation

- Enum constants must be identifiers (not fields with access modifiers directly).
- Compile-time error.

Answer: d. Compilation fails

Question 17

```
public class Test {

    public void method() {

        for (int i = 0; i < 3; i++) {

            System.out.print(i);

        }

        System.out.print(i); // i is out of scope here

    }

}
```

Explanation

- Variable `i` is declared inside `for` loop.
- Not accessible outside loop.
- Compile-time error.

Answer: c. Compilation fails

Question 18

```
public class Main {
    static int[] x;


    static {
        x[0] = 102; // x is null here
    }


    public static void main(String[] args) {
        System.out.println(x);
    }
}
```

Explanation

- `x` declared but not initialized. Default = `null`.
- `x[0] = 102;` → NullPointerException during **class initialization**.
- JVM throws `ExceptionInInitializerError`.

Answer: c. java.lang.ExceptionInInitializerError

Question 19

```
class VarArgsDemo {
    static void func(int... x) {
        System.out.println("Number of arguments: " + x.length);
        for (int i : x)
            System.out.print(i + " ");
        System.out.println();
    }

    void func(int a) { // Line1
        System.out.println("one");
    }

    public static void main(String[] args) {
        new VarArgsDemo().func(150);  // calls func(int a)
        func(11, 12, 13, 14);         // calls varargs
        func();                         // calls varargs with 0

    }

}
```

Output

```
one

Number of arguments: 4

11 12 13 14

Number of arguments: 0
```

Answer: b. One … Number of arguments: 4 … Number of arguments: 0

Question 20

```
class Employee {

    Double salary;


    public static void main(String[] args) {

        Employee employee1 = null;

        Employee employee2 = null;


        employee1 = new Employee();

        employee2 = new Employee();
        employee1 = new Employee();


        Employee employee3 = null;

        employee1 = employee3 = null; // Line8


        System.out.println("Hello World");

    }

}
```

Explanation

- Step 1: `employee1 = new Employee()` → object A
- Step 2: `employee2 = new Employee()` → object B
- Step 3: `employee1 = new Employee()` → object C (old A lost reference)
- Step 4: `employee1 = employee3 = null` → now C also lost reference

Objects eligible for GC: A, B, C → **3 objects**.

Answer: a. 3

Great set of questions 👍 Let's carefully go through **Q21 – Q40** one by one with reasoning and final answers.

---

21. Inheritance & composition

public class Developer {}

```
public class Employee {
    public String empName;
}
public class Tester extends Employee {
    public Developer developer;
}
public class Testing extends Tester {}
```

- `Testing` inherits from `Tester` → it **has** `empName` (from `Employee`)
- `Testing` inherits from `Tester` → it **has** a `Developer` field
- `Testing` is **not** a `Developer` (no `extends Developer`) ❌
- `Testing` is an `Employee` (indirectly via `Tester`)
- `Tester` is a `Testing` ❌ (reverse is not true).

Correct: **A, B, D**

....