A. B2.3

Lesson flow you can use

1. Introduce the idea of **program control flow**
   • Sequence of instructions
   • How the computer executes statements one by one

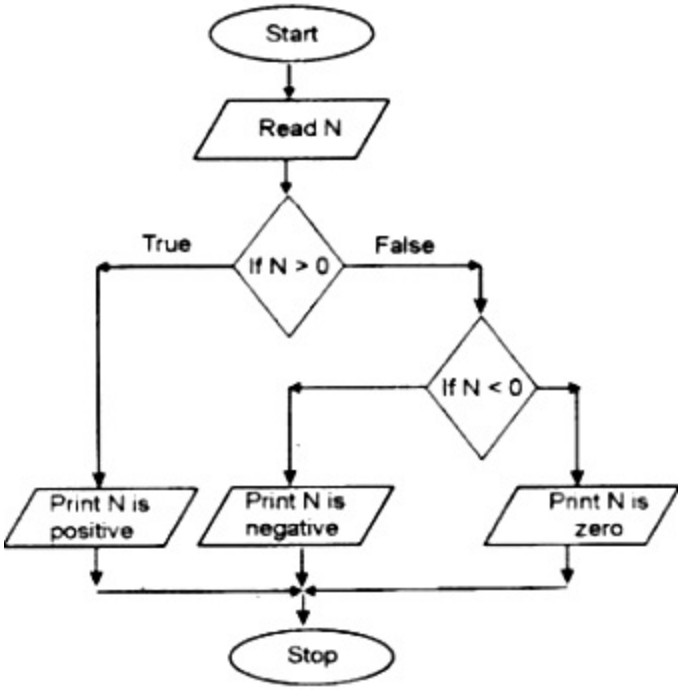
2. B2.3.1
   • Show simple programs where **order of instructions** changes the behaviour
   • Discuss problems such as infinite loops, deadlock in simple terms, and incorrect output
   • Use trace tables to follow a program step by step

3. B2.3.2
   • Explain **selection** and branching
   • Introduce `if`, `else if`, `else` in Java
   • Explain relational operators and Boolean operators
   • Show example programs with branching and trace tables

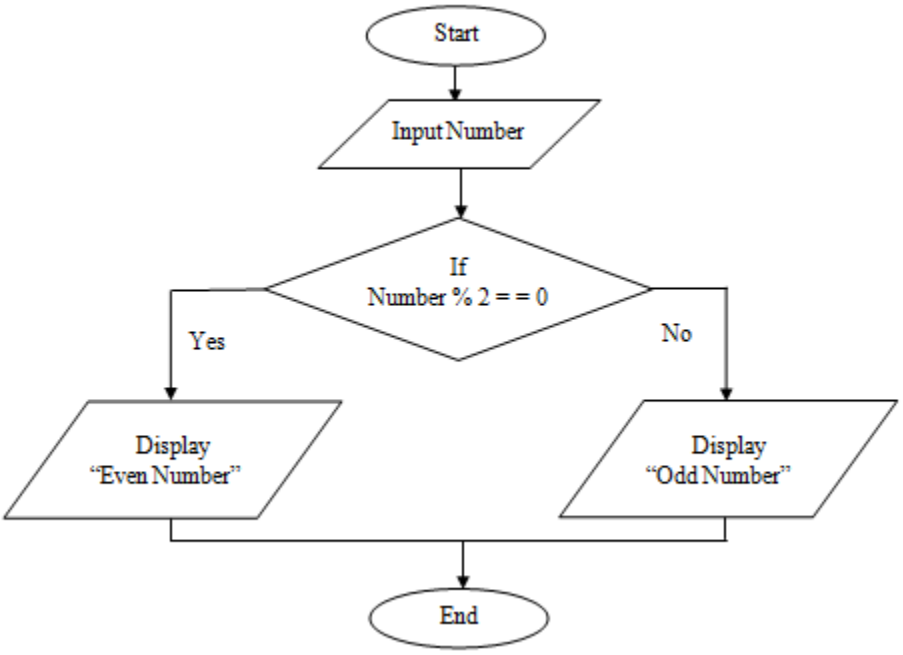
4. Wrap up
   • Why sequence and selection are essential for correct algorithms
   • IB style questions using command terms like construct, explain, describe, trace etc

Flowchart Examples

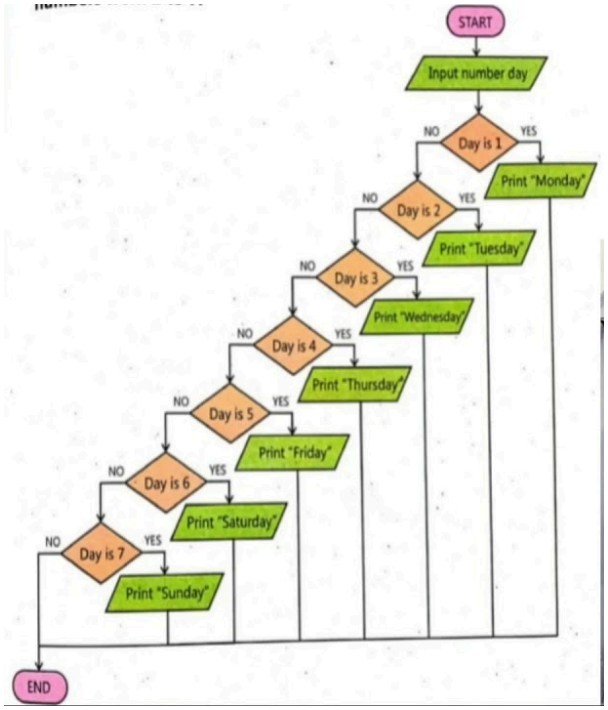
Positive Negative



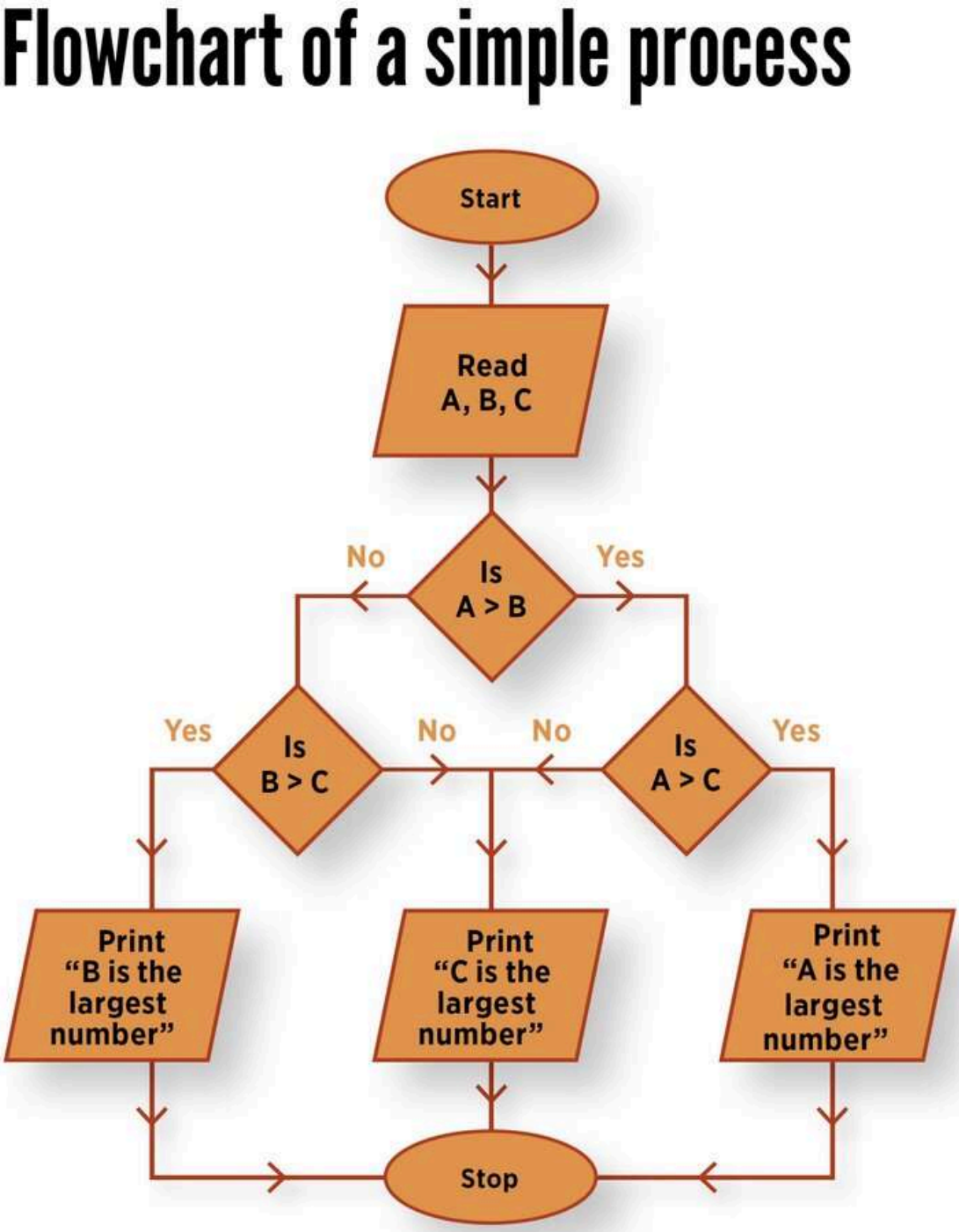
Even Odd



Day Name


START
Input number day
Day is 1
NO
YES
Print "Monday"
Day is 2
NO
YES
Print "Tuesday"
Day is 3
NO
YES
Print "Wednesday"
Day is 4
NO
YES
Print "Thursday"
Day is 5
NO
YES
Print "Friday"
Day is 6
NO
YES
Print "Saturday"
Day is 7
NO
YES
Print "Sunday"
END

Largest Number:



B. B2.3.1 Sequence of instructions

1. What is sequence in programming

In Java, if you do not tell the program to branch or loop, the computer executes one statement after another, from top to bottom inside a method.

This is the **sequence** construct.

Example

```
public class SequenceExample {

    public static void main(String[] args) {

        int a = 5;

        int b = 2;

        int sum = a + b;

        System.out.println("Sum is " + sum);

    }

}
```

The computer does

1. Create an integer `a` and set it to 5
2. Create an integer `b` and set it to 2
3. Create an integer `sum` and store `a + b`
4. Print the message

The steps always happen in this order.

2. Impact of instruction order

Even with the same statements, changing the order can change the result.

Example 1 correct order

```
public class OrderExample1 {

    public static void main(String[] args) {

        int x = 10;

        int y = 3;

        int ratio = x / y;

        System.out.println("Ratio is " + ratio);

    }

}
```

Example 2 incorrect order

```
public class OrderExample2 {

    public static void main(String[] args) {

        int ratio;

        System.out.println("Ratio is " + ratio);  // uses ratio before
giving it a value

        ratio = 10 / 3;

    }

}
```

In Example 2, Java does not allow the use of `ratio` before it is assigned. This shows that **order matters** for correctness.

3. Sequence example with trace table

Program

```
public class SumExample {

    public static void main(String[] args) {

        int sum = 0;

        int i = 1;

        sum = sum + i;    // line A

        i = i + 1;        // line B

        sum = sum + i;    // line C

        System.out.println("Sum is " + sum);

    }

}
```

Trace table for this program

Step	Executed line	i before	sum before	i after	sum after	Output
1	int sum = 0	-	-	-	0	-
2	int i = 1	-	0	1	0	-
3	line A	1	0	1	1	-
4	line B	1	1	2	1	-
5	line C	2	1	2	3	-
6	print	2	3	2	3	"Sum is 3"

Students see clearly how values change in sequence.

---

4. Avoiding infinite loops

An **infinite loop** is a loop that never ends.

Simple example of an infinite loop in Java

```
while (true) {

    System.out.println("Hello");

}
```

This loop does not have any condition that can become false, so it keeps printing forever until you stop the program.

More subtle infinite loop

```
int i = 1;

while (i < 5) {

    System.out.println(i);

    // missing update of i

}
```

Here, `i` is never changed inside the loop, so `i < 5` is always true and the loop never ends.

Ways to avoid infinite loops

- Make sure the loop variable is updated inside the loop
- Make sure the condition will eventually become false for some input
- Test with small values and use trace tables for the loop

5. Deadlock at this level

In full computer science, deadlock often appears with threads and shared resources. At DP level, you can introduce a simple idea:

Deadlock is a situation where two parts of a program are waiting for each other and so nothing moves forward.

Simple conceptual example (no actual thread code)

• Method A waits for a signal from Method B
• Method B waits for a signal from Method A

If both are waiting, neither will continue.

For DP classes you can keep it conceptual and stress that **poor control flow design** can make a program get stuck.

6. Incorrect output due to wrong order

Even if the program runs and stops, incorrect order can give wrong results.

Example

```
public class AverageExampleWrong {

    public static void main(String[] args) {

        int total = 0;

        int count = 3;

        int average = total / count;   // line X

        total = total + 5;

        total = total + 7;

        total = total + 9;

        System.out.println("Average is " + average);

    }

}
```

Here, `average` is calculated before adding the numbers. The program runs but prints zero.

Correct version

```
public class AverageExampleCorrect {

    public static void main(String[] args) {

        int total = 0;

        int count = 3;

        total = total + 5;

        total = total + 7;

        total = total + 9;

        int average = total / count;   // moved after updates

        System.out.println("Average is " + average);

    }

}
```

Same lines, different order, completely different meaning.

C. B2.3.2 Selection structures in Java

1. What is selection

Selection is the ability of a program to choose between different paths based on a condition.

In Java, selection uses these constructs

- `if`
- `if` followed by `else`
- `if` followed by `else if` and possibly `else`

These allow branching of control flow.

---

2. Relational operators in Java

Relational operators compare two values and produce a Boolean result.

- `<` less than
- `<=` less than or equal
- `>` greater than
- `>=` greater than or equal
- `==` equal
- `!=` not equal

Example

```
int age = 16;

boolean isAdult = age >= 18;   // false

boolean isTeen = age >= 13 && age <= 19;  // true
```

The comparisons produce true or false, which selection constructs use.

---

3. Logical operators in Java

Logical operators combine or negate Boolean values.

- `&&` logical AND
- `||` logical OR
- `!` logical NOT

Idea

- `A && B` is true only if both A and B are true
- `A || B` is true if at least one of A or B is true
- `!A` is true if A is false

Example

```
boolean hasTicket = true;

boolean hasID = false;


boolean canEnter = hasTicket && hasID;   // false

boolean canStandOutside = hasTicket || hasID;  // true

boolean noTicket = !hasTicket;           // false
```

---

4. Simple `if` selection

```
public class IfExample {

    public static void main(String[] args) {

        int score = 85;


        if (score >= 80) {

            System.out.println("You passed");

        }


        System.out.println("End of program");

    }
```

```
}
```

If `score` is 85, the condition is true and both lines print.

If `score` is 70, the condition is false and only `"End of program"` prints.

---

5. `if` with `else`

```
public class IfElseExample {

    public static void main(String[] args) {

        int score = 55;


        if (score >= 50) {

            System.out.println("Pass");

        } else {

            System.out.println("Fail");

        }

    }

}
```

Only one of the branches will execute.

Trace table for `score = 55`

```
Step   score   Condition score >= 50   Executed branch   Output

1      55      true                    if                "Pass"
```

Trace table for `score = 40`

Step	score	Condition score >= 50	Executed branch	Output
1	40	false	else	"Fail"

---

6. `if`, `else if`, `else`

This structure allows multiple conditions in order.

```
public class GradeExample {

    public static void main(String[] args) {

        int score = 72;


        if (score >= 90) {

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

        } else if (score >= 80) {

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

        } else if (score >= 70) {

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

        } else {

            System.out.println("Grade D or below");

        }

    }

}
```

Important points

• Conditions are checked from top to bottom
• As soon as one condition is true, its block runs and the rest are skipped

Trace table for `score = 72`

```
Step   score   Check score >= 90   Check score >= 80   Check score >=
70   Branch taken   Output

1      72      false               (not checked)       (not yet)
-              -

2      72      -                   false               (not yet)
-              -

3      72      -                   -                   true
else if 3     "Grade C"
```

Only the third condition is true, so `"Grade C"` is printed.

---

7. Selection with relational and logical operators

Selection becomes more powerful when you combine tests.

Example check if a year is a leap year using a simplified rule

```
public class LeapYearExample {

    public static void main(String[] args) {

        int year = 2024;


        if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)) {

            System.out.println(year + " is a leap year");

        } else {
```

```
            System.out.println(year + " is not a leap year");
        }
    }
}
```

Here you see

• relational operators `==` and `!=`
• logical operators `&&` and `||`
• brackets to control the order of evaluation

D. Why branching and logical operators are important

Without selection, a program would always do the same thing every time it runs. It could not react to user input or data.

Branching allows the program to

• make decisions
• handle different cases
• respond to errors

Logical operators allow complex conditions, for example

• user is logged in and has admin role
• temperature is below freezing or above a high limit
• input is not empty and length is within a range

Together, selection and logical operators are key to building correct algorithms.

E. Combined example with sequence and selection

Program to compute a ticket price with a discount for students and seniors.

```
import java.util.Scanner;
```

```
public class TicketExample {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);


        System.out.print("Enter age: ");

        int age = sc.nextInt();


        System.out.print("Are you a student (true or false): ");

        boolean isStudent = sc.nextBoolean();


        double price = 10.0;


        if (age >= 65 || isStudent) {

            price = price * 0.5;

        }


        System.out.println("Ticket price is " + price);

        sc.close();

    }

}
```

Explanation

1. Sequence of input operations
2. Initial price set to 10
3. Selection uses logical OR to check if age is at least 65 or the person is a student
4. If the condition is true, the price is reduced to half
5. Program outputs the final price

Trace table for age 70, isStudent false

```
Step   age   isStudent   price before   Condition (age >= 65 ||
isStudent)   price after   Output

1      70    false       10.0           true
5.0           "Ticket price is 5.0"
```

Trace table for age 20, isStudent true

```
Step   age   isStudent   price before   Condition (age >= 65 ||
isStudent)   price after   Output

1      20    true        10.0           true
5.0           "Ticket price is 5.0"
```

Trace table for age 30, isStudent false

```
Step   age   isStudent   price before   Condition (age >= 65 ||
isStudent)   price after   Output

1      30    false       10.0           false
10.0          "Ticket price is 10.0"
```

F. IB style questions for B2.3

Use these for exercises, homework, or assessments.

Sequence and instruction order B2.3.1

1. **Define** the term sequence as used in programming. [2]

2. **Describe** how the order of instructions in a Java program can affect its output, using a simple numerical example. [3]

The following Java code compiles and runs but prints an incorrect result.

```
public static void main(String[] args) {

    int total = 0;

    int count = 4;

    int average = total / count;

    total = total + 10;

    total = total + 8;

    total = total + 6;

    total = total + 4;

    System.out.println("Average is " + average);

}
```

3. a. **Identify** the logical error in the sequence of statements. [1]
   b. **Explain** how changing the order of the instructions would correct the program. [4]
   c. **Construct** a corrected version of the code. [4]

4. **Explain** two ways that a programmer can reduce the risk of infinite loops when writing repetition structures. [4]

5. **Outline** what is meant by deadlock in the context of program control flow. [3]

Selection and branching B2.3.2

6. **State** the purpose of a selection statement in a Java program. [2]

7. **Construct** a Java method `getCategory` that receives an integer `age` and returns a string according to the rules

   - `"Child"` if age is less than 13
   - `"Teenager"` if age is between 13 and 19 inclusive
   - `"Adult"` if age is 20 or more

   Use `if`, `else if`, and `else`. [6]

8. **Explain** the difference between `&&` and `||` in Java, using one example for each operator. [4]

A programmer writes the following code to decide if a user can access a system.

```
boolean loggedIn = true;

boolean isAdmin = false;


if (loggedIn && isAdmin) {

    System.out.println("Access granted");

} else {

    System.out.println("Access denied");

}
```

9. a. **Trace** the code for the given values of `loggedIn` and `isAdmin`. State the output. [3]
   b. **Describe** how the program output would change if the condition were `loggedIn || isAdmin`. [3]

10. **Construct** a Java program that reads an integer mark from the user and prints `"Pass"` if the mark is at least 50 and `"Fail"` otherwise. Use an appropriate selection structure. [5]

11. **Discuss** how complex selection structures that combine relational and logical operators can improve the clarity of a Java program, but can also introduce new types of errors if

not designed carefully. [6]