Java Basic Program

Module 1
1. Introduction part, Languages Programing, Java history, JDK, “hello World”
2. Project, package, Class, method
3. Variables. Keyboard input
4. Variables and data types
5. Consultation

Module 2
1. TEST #1 and second part Compilation and constructor
2. Methods and Random ways
3. Practice and examples with methods
4. boolean. Boolean expressions
5. Consultation

Module 3
1. if-else-if
2. Switch, ternary operator
3. Loops, for
4. Loops, while, do while
5. Consultation

Module 4
1. Arrays in Java
2. Arrays search and sort
3. String, StringBuilder, StringBuffer, practice
4. TEST #2 and second part Class and Object
5. Consultation

Module 5 (Optional)
1. Method main() for the test and introduction to JUnit testing
2. Practice, repetitions, console Lottery game
3. Practice, implementation of the distribution of cards in Poker
4. Summarizing and Introduction to the professional course program and the profession Back-end developer in Java
5. Consultation (questions)

INTRODUCTION TO

JAVA

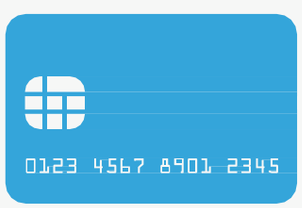
OBJECT ORIENTED PROGRAMMING IN DEPTH

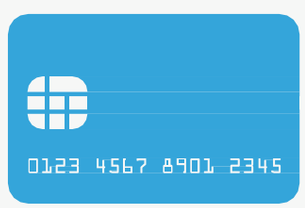
PRIMARY CONCEPTS: CLASS AND OBJECT

- Class describes template (blueprint) of something with state and behaviour
- Object is concrete instance of that class with set state

EXAMPLE: BANK CARD (STATE)

Class		Object
A. Bank Name	→	A. Citadele Banka
B. Payments Processor	→	B. Master Card
C. Name on Card	→	C. John Doe
D. Card Number	→	D. 5224 9989 7556 2871
E. Expiration Date	→	E. 12/2022
F. Security Code	→	F. 218





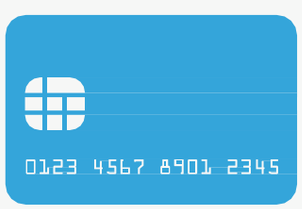
EXAMPLE: BANK CARD (BEHAVIOUR)

Class

A. Get balance

B. Deposit funds

C. Withdraw funds

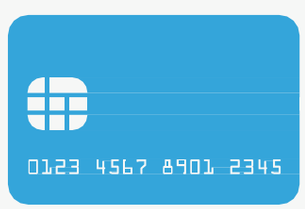


Object

A. Get balance

B. Deposit funds

C. Withdraw funds



CLASS DECLARATION IN JAVA: SYNTAX

```
class ClassName {

    type variable1;
    type variable2;
    ...
    type variableN;

    method1() {}
    method2() {}
    ...
    methodN() {}

}
```

CLASS DECLARATION IN JAVA: EXAMPLE BREAKDOWN

Special keyword

Class name

Class variables (fields) defining state

Class methods defining behaviour

```
class Dog {

    String name;
    int age;
    boolean hungry;

    void bark() {

    }

    void sleep() {

    }

    void eat() {

    }

}
```

OBJECT INSTANTIATION IN JAVA: SYNTAX

- Object instantiation without assignment

```
new Class();
```

- Object instantiation with assignment

```
Class var = new Class();
```

OBJECT INSTANTIATION IN JAVA: SYNTAX

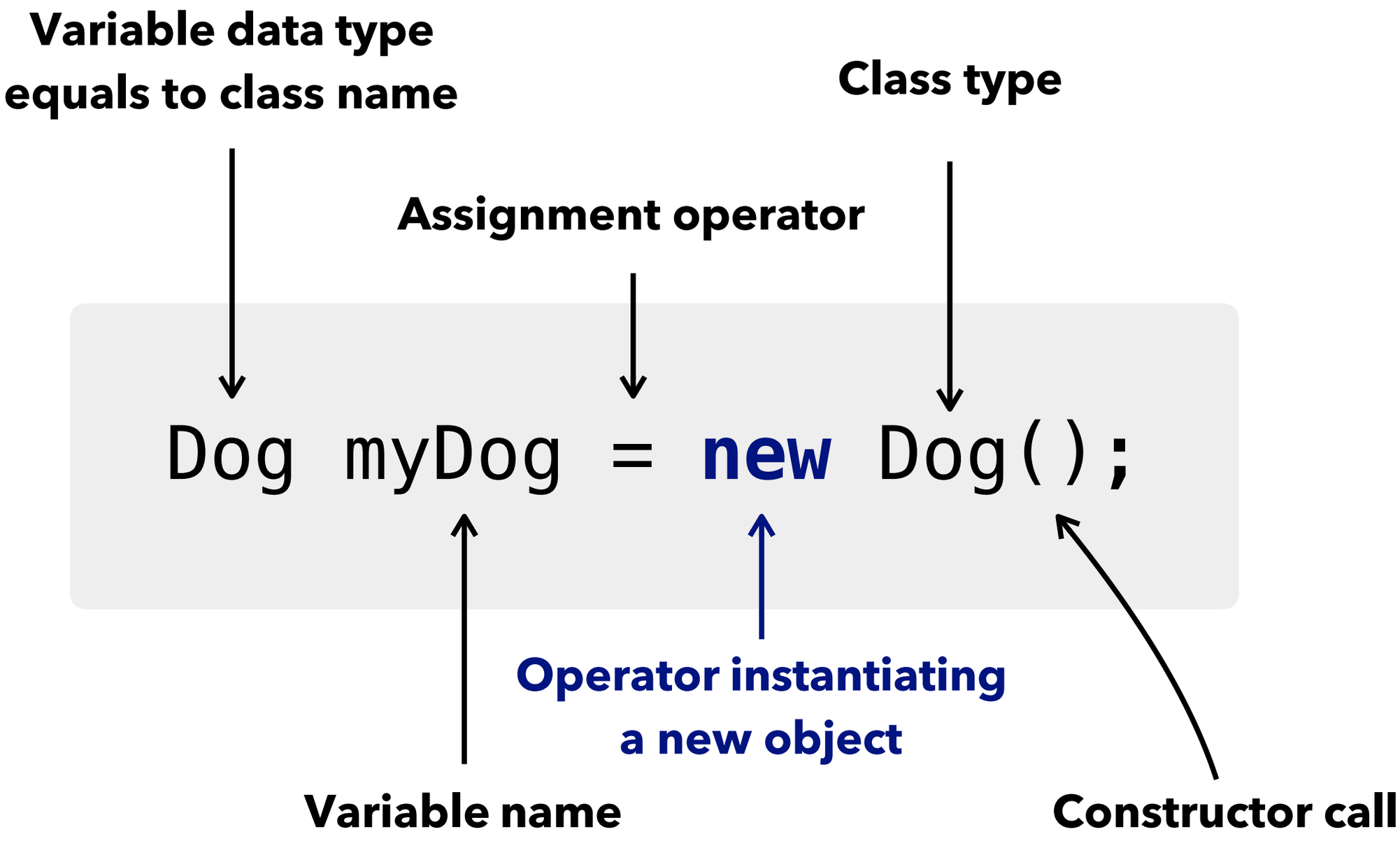
- Object instantiation without assignment

```
2 new Dog();
```

- Object instantiation with assignment

```
Dog myDog = new Dog();
```

OBJECT INSTANTIATION BREAKDOWN



THREE-STEP PROCESS OF OBJECT CREATION

1. Declaration - object variable declaration of a class type
2. Instantiation - the process of creating an object with new operator
3. Initialisation - the process of object construction by setting its initial state

CONSTRUCTORS

- Every class has a constructor
- If explicit constructor(s) is not specified in code, Java Compiler will generate default constructor implicitly
- Each time a new object is created, at least one constructor will be invoked
- Each defined constructor must have unique signature (i.e. ordered number and type of arguments)

CONSTRUCTOR DECLARATION IN JAVA: EXAMPLE BREAKDOWN

Explicit default constructor without arguments →

```
public class Dog {

    private String name;

    public Dog() {
    }

    public Dog(String name) {
        this.name = name;
    }

}
```

← **Explicit constructor with argument and initialisation**

MEMORY
OVERVIEW

MEMORY TYPES

- Java Heap Memory
  - Created objects are stored in the heap space
  - Lives from the start till the end of application execution
  - Objects stored in heap are globally accessible
- Java Stack Memory
  - Contains local primitive variables and reference variables to objects in heap space
  - Lives only within method execution, short-lived
  - Bound to the current execution thread

METHODS

OVERVIEW

METHOD DEFINITION

- Java method is a collection of statements that are grouped together to perform an operation
  - Invoking *System.out.println()* method actually executes several statements in order to display a message on the console
- Describes behaviour of class or actions that object can perform
- Method either produces output or not

METHOD DECLARATION IN JAVA: SYNTAX

Defines the access type of the method

Defines method name

```
modifier returnType methodName (arg1, arg2, ..., argN) {

    //body

}
```

Specifies return type for methods producing output

List of parameters, it is type, order and number

METHOD DECLARATION IN JAVA: WITH RETURN EXAMPLE

Can be accessed from everywhere

Result type

Method name

Method accepts two integer parameters

```
public int sum(int a, int b) {
    int result = a + b;
    return result;
}
```

Operation with parameters

Keyword with variable that returns value back to the invoker

METHOD DECLARATION IN JAVA: WITHOUT RETURN EXAMPLE

Method name

Keyword meaning no return value

Method accepts two integer parameters

```
public void printSum(int a, int b) {
    int result = a + b;
    System.out.println("Sum of numbers is" + result);
}
```

Operations with parameters

A BIT MORE ABOUT RETURNING RESULT

- After completion method returns to the code that invoked it
- Whether method returns value or not is declared in method signature
- When type is void - return statement is unnecessary, however can be stated
- Other type - return statement is necessary

ACCESSING AND CHANGING OBJECT STATE: GETTERS & SETTERS

- In OOP another party should not be able to access object state directly
- To keep things safe, one can
  - Retrieve object state via get methods (getters)
  - Change object state via set methods (setters)

GETTERS & SETTERS DECLARATION

```java
public class Person {

    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Getters

Setters

GETTERS & SETTERS USAGE

```
public class PersonTest {

    public static void main(String[] args) {

        Person person = new Person();
        person.setName("JohnDoe");
        person.setAge(32);

        String personName = person.getName();
        int personAge = person.getAge();

        System.out.println("Hisname  is " + personName);
        System.out.println("He is " +personAge  + " years old");

    }

}
```

CLEAN CODE PRACTICES

BAD CODE AND GOOD CODE

Bad

```
public class Cat {
    privateString n;
    public String getN() {
        return n;
    }

    public void setN(String n) {
        this.n = n;
    }

    public void v() {
        System.out.println("Meow");
    }

}
```

Good

```
public class Cat {
    privateString name;
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void voice() {
        System.out.println("Meow");
    }

}
```

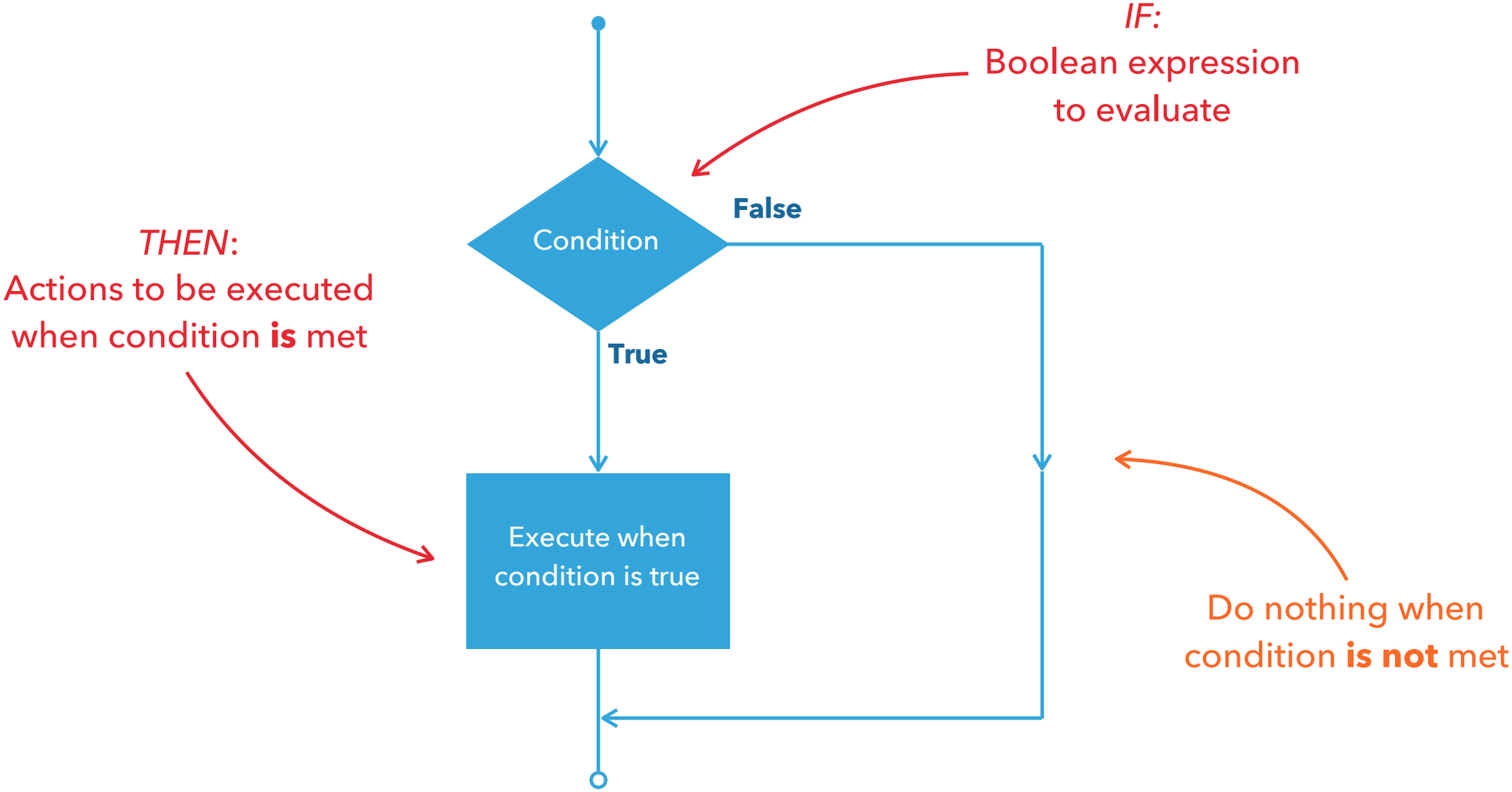
CONDITIONAL

FLOW CONTROL

CONDITIONAL STATEMENTS

- Control code execution by specifying certain conditions
  - When conditional statement is **met** (equals to 'true')
  - When conditional statement is **not met** (equals to 'false')
- There are two main conditional statements:
  - **If** statement
  - **Switch** statement

DECISION MAKING FLOWCHART: IF



IF STATEMENT: SYNTAX

Keyword specifying conditional statement

Variable or expression with boolean result

```
if (statement) {
    //Code to execute
    //When statement is true
}
```

IF STATEMENT: EXAMPLE

Boolean variable expression

```
boolean flag = true;

if (flag) {
    System.out.print("True");
}
```

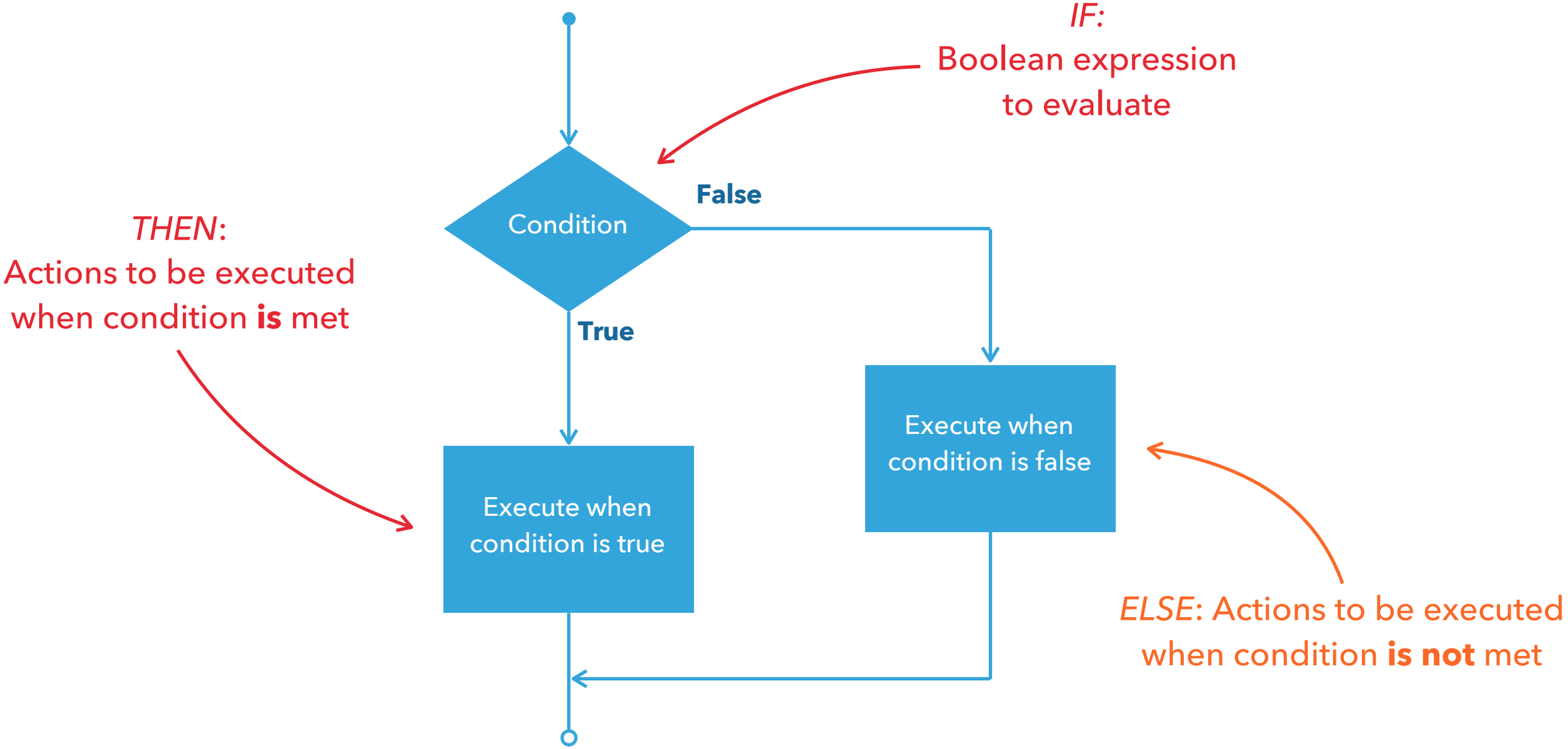
Inline expression

```
int x = 5;

if (x > 10) {
    System.out.print("x > 10");
}
```

IF STATEMENT RULES RECAP

- Consists of a **boolean expression** followed by **one or more** statements
- Boolean expression can be **composed** of multiple subexpressions

DECISION MAKING FLOWCHART: IF – ELSE



IF – ELSE STATEMENT: SYNTAX

Keyword specifying conditional statement

Variable or expression with boolean result

```
if (statement) {
    //Code to execute
    //When statement is true
} else {
    //Code to execute
    //When statement is false
}
```

Keyword specifying alternative code block

IF – ELSE STATEMENT: EXAMPLE

Boolean variable expression

```java
boolean flag = false;

if (flag) {
    System.out.print("True");
} else {
    System.out.print("False");
}
```

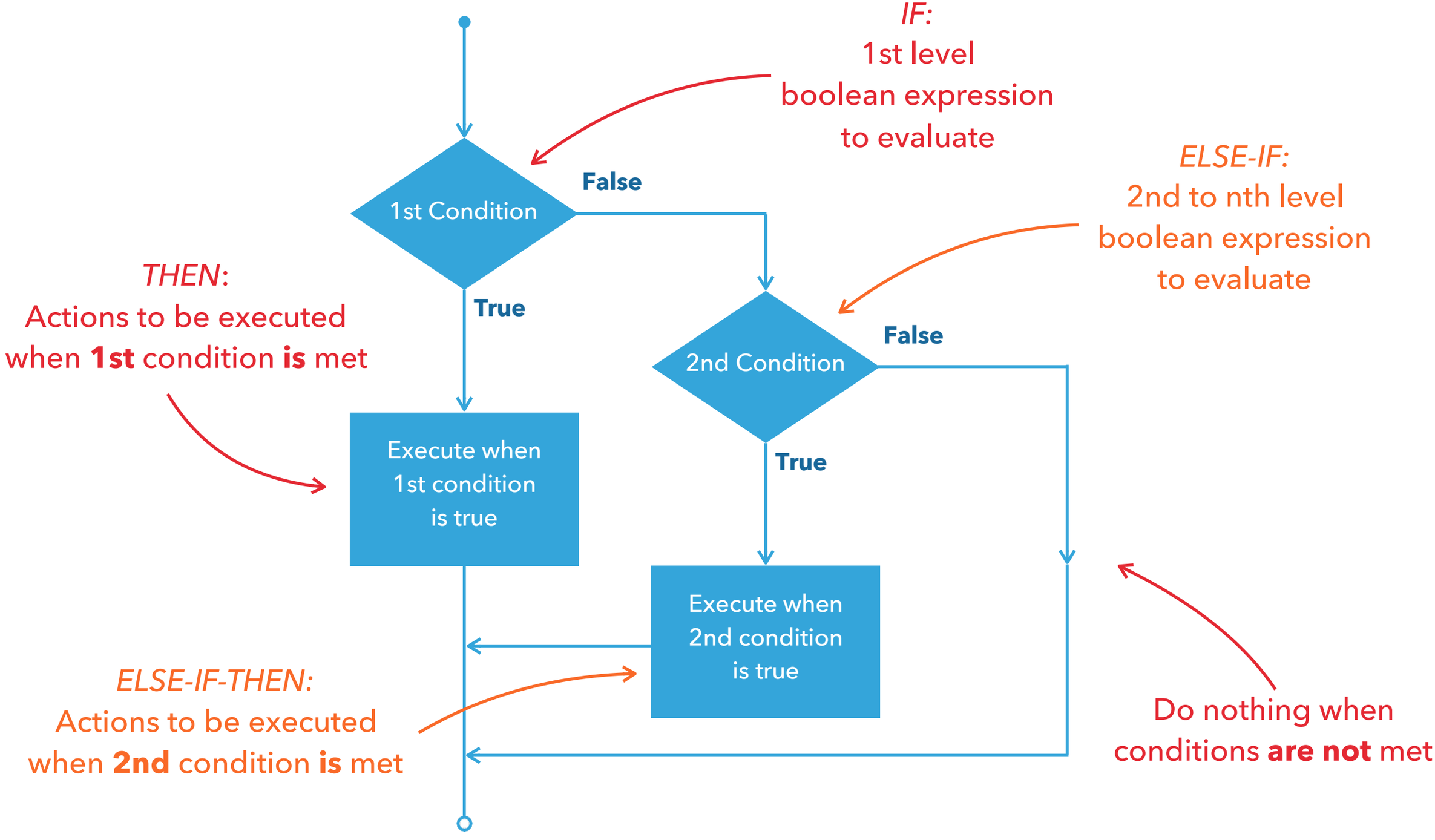
Inline expression

```java
int x = 5;

if (x > 10) {
    System.out.print("x > 10");
} else {
    System.out.print("x =< 10");
}
```

IF – ELSE STATEMENT RULES RECAP

- If statement can be followed by an **optional** else statement, which executes when the boolean expression is **false**

DECISION MAKING FLOWCHART: IF – ELSE IF



IF – ELSE IF STATEMENT: SYNTAX

Keyword specifying conditional statement

Variable or expression with boolean result

```
if (statement1) {
    //Code to execute
    //When statement1 is true
} else if (statement2) {
    //Code to execute
    //When statement2 is true
}
```

Keyword specifying alternative conditional code block

IF – ELSE IF STATEMENT: EXAMPLE

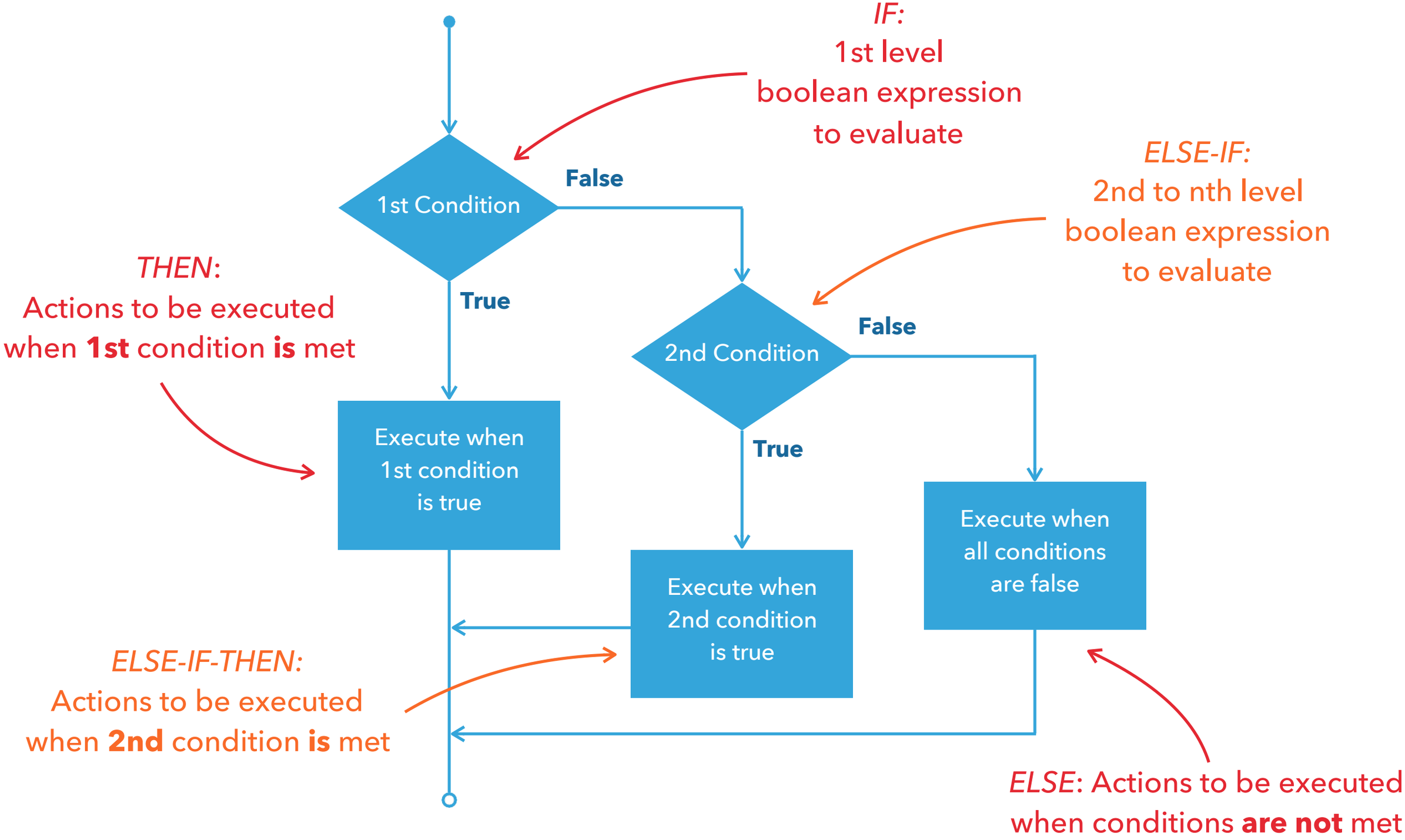
Boolean variable expression

```
boolean flag1 = false;
boolean flag2 = true;

if (flag1) {
    System.out.print("flag1");
} else if (flag2) {
    System.out.print("flag2");
}
```

Inline expression

```
int x = 7;

if (x == 3) {
    System.out.print("x == 3");
} else if (x == 7) {
    System.out.print("x == 7");
}
```

DECISION MAKING FLOWCHART: IF – ELSE IF – ELSE



IF – ELSE IF – ELSE STATEMENT: SYNTAX

Keyword specifying conditional statement

Variable or expression with boolean result

```
if (statement1) {
    //Code to execute
    //When statement1 is true
} else if (statement2) {
    //Code to execute
    //When statement2 is true
} else {
    //Code to execute
    //When all statements are false
}
```

Keyword specifying alternative conditional code block

Keyword specifying alternative code block

IF – ELSE IF – ELSE STATEMENT: EXAMPLE

Boolean variable expression

```
boolean flag1 = false;
boolean flag2 = false;

if (flag1) {
    System.out.print("flag1");
} else if (flag2) {
    System.out.print("flag2");
} else {
    System.out.println("none");
}
```

Inline expression

```
int x = 7;

if (x == 3) {
    System.out.print("x == 3");
} else if (x == 7) {
    System.out.print("x == 7");
} else {
    System.out.print("NOTA");
}
```

IF – ELSE IF – ELSE STATEMENT RULES RECAP

- An if can have **zero** or **one** else's and its must come after any else if's
- An if can have **zero** to **many** else if's and they must come **before** else
- Once an else if **succeeds**, **none** of the **remaining** else if's or else's will be tested

SWITCH STATEMENT OVERVIEW

- Provides an **effective** way to deal with a section of code that could branch in **multiple directions** based on **single variable**
- **Doesn't** support the conditional operators that the **if statement** does
- **Can't** handle **multiple** variables

SWITCH STATEMENT: SYNTAX

Keyword specifying switch statement

Single variable to test against

```
switch (singleVariable) {
    case value1:
        //Code to execute
        //When value1 matches
        break;

    case value2:
        //Code to execute
        //When value2 matches
        //Without break it
        //Falls through cases
        //Until first break

    default:
        //Code to execute
        //When none of the above
        //Values match
}
```

Variable must match one of these (else - if)

Keyword telling to stop switch execution right here

Fallback case (else)

SWITCH STATEMENT: EXAMPLE

```
String drink = "coffee";

switch (drink) {
    case "coffee":
        System.out.println("I would go for Java!");
        break;

    case "tea":
        System.out.println("Everything but Lipton");
        break;

    default:
        System.out.println("Ugh.. What?");
}
```

BUILDING BOOLEAN EXPRESSIONS

THE EQUALITY AND RELATIONAL OPERATORS

Operator	Operation
==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

CONDITIONAL OPERATORS

Operator	Operation
&&	Conditional AND
||	Conditional OR
!	Conditional NOT

COMPLEX BOOLEAN STATEMENT EXAMPLE

Make sure that BOTH statements are true

Check if x is greater than 5

Check if x is lesser than 15

```
int x = 10;

if ((x > 5) && (x < 15)) {
    System.out.print("Within bounds!");
}
```

BASIC CODE TESTING APPROACH

TASK OBJECTIVES

1. Write class that returns **max number** from two given numbers
2. Write test scenarios to **verify** method works as **expected**
3. **Run** test scenarios

1. WRITE CLASS SOLVING GIVEN PROBLEM

```
public class QuickMaths {

    public int max(int a, int b) {
        if (a > b) {
            return a;
        } else {
            return b;
        }
    }

}
```

2.A. WRITE TEST CLASS WITH VERIFICATION SCENARIOS

```
public class QuickMathsTest {

    public void test1() {

        QuickMaths victim = new QuickMaths();

        int a = 3;
        int b = 5;

        int expectedResult = 5;

        int actualResult = victim.max(3, 5);

        check(actualResult, expectedResult, "test1");

    }

    public void check(int actualResult, int expectedResult, String testName) {
        if (actualResult == expectedResult) {
            System.out.println(testName + " has passed!");
        } else {
            System.out.println(testName + " has failed!");
            System.out.println("Expected " + expectedResult + " but was " + actualResult);
        }
    }

}
```

2.B. INSTANTIATE TEST CLASS AND CALL VERIFICATION METHODS

```
public class QuickMathsTest {

    public static void main(String[] args) {
        QuickMathsTest testRunner = new QuickMathsTest();
        testRunner.test1();

    }

    ...

}
```

3. RUN AND CHECK RESULTS

```
test1 has passed!

Process finished with exit code 0
```

LOOPING STATEMENTS

OVERVIEW

- There may be situation when you need to execute a block of code several number of times
- A loop statement allows us to execute a statement or group of statements multiple times
- Looping statements available:
  1. while
  2. for
  3. do...while

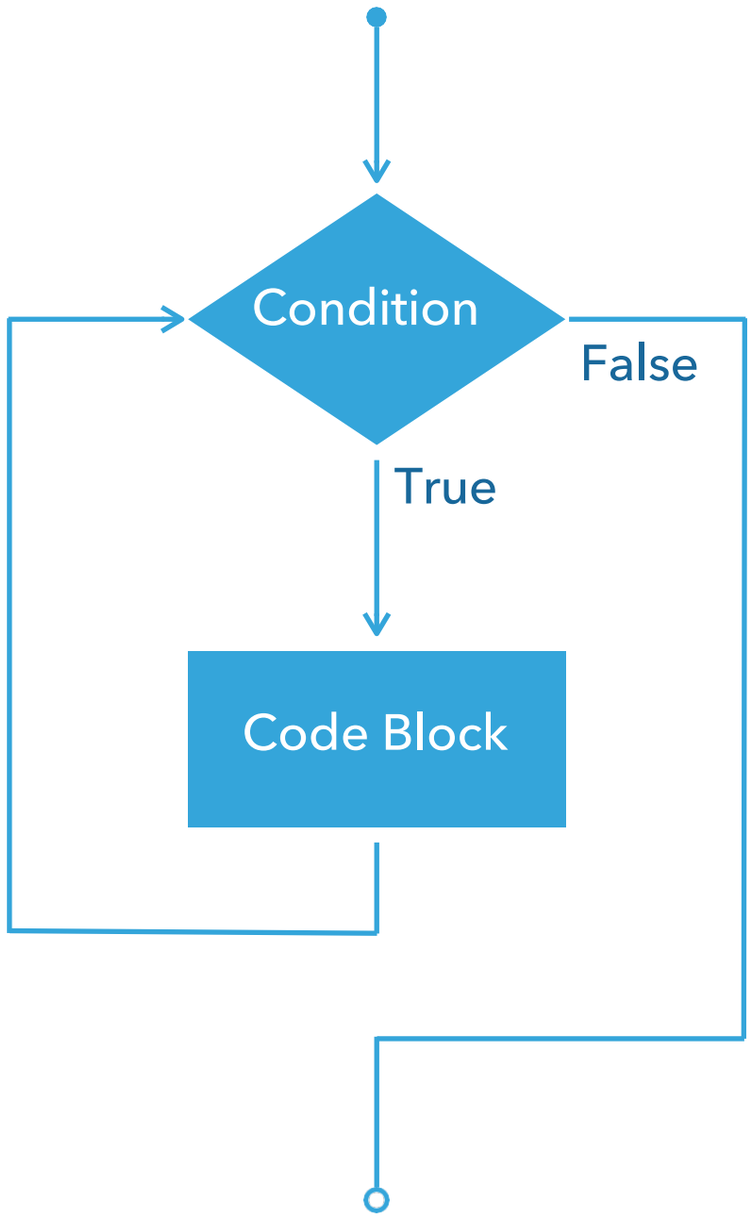
COMMON LOOPS STRUCTURE

- There is a control variable, called the loop counter
- Loop variable must be initialized
- The increment or decrement of the control variable, which is modified each time the iteration of the loop occurs
- The loop condition that determines if the looping should continue or the program should break from it

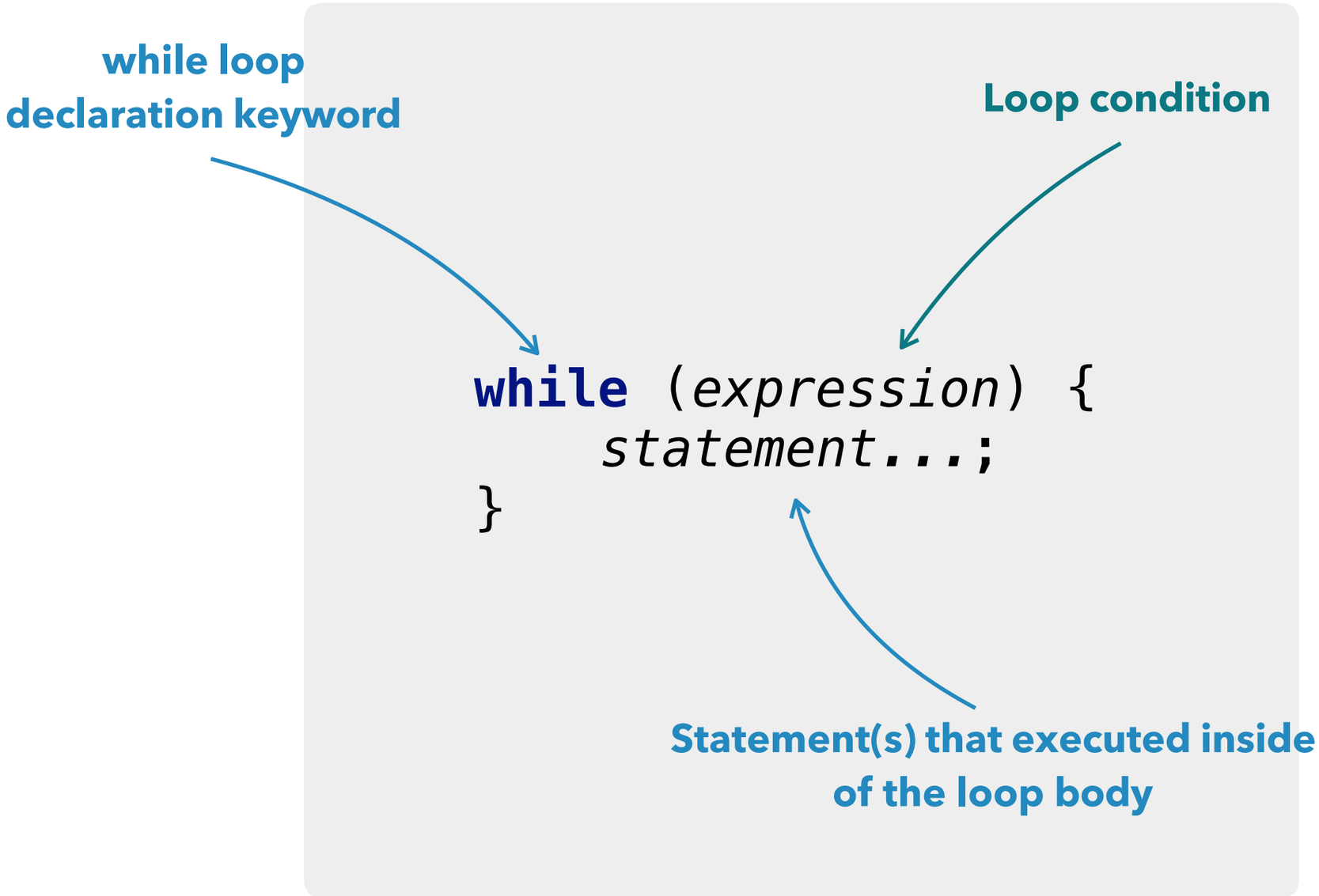
WHILE LOOP: SUMMARY

- Repeats a statement or block of statements while its controlling boolean expression is true
- Boolean expression is evaluated before the first iteration of the loop, hence executed zero or many times
- Usually used when number of iterations depends

WHILE LOOP: FLOWCHART



WHILE LOOP: SYNTAX



WHILE LOOP: CODE EXAMPLE

Code

```
int i = 0;
while (i < 5) {
    System.out.print("i = " + i + "; ");
    i++;
}
```

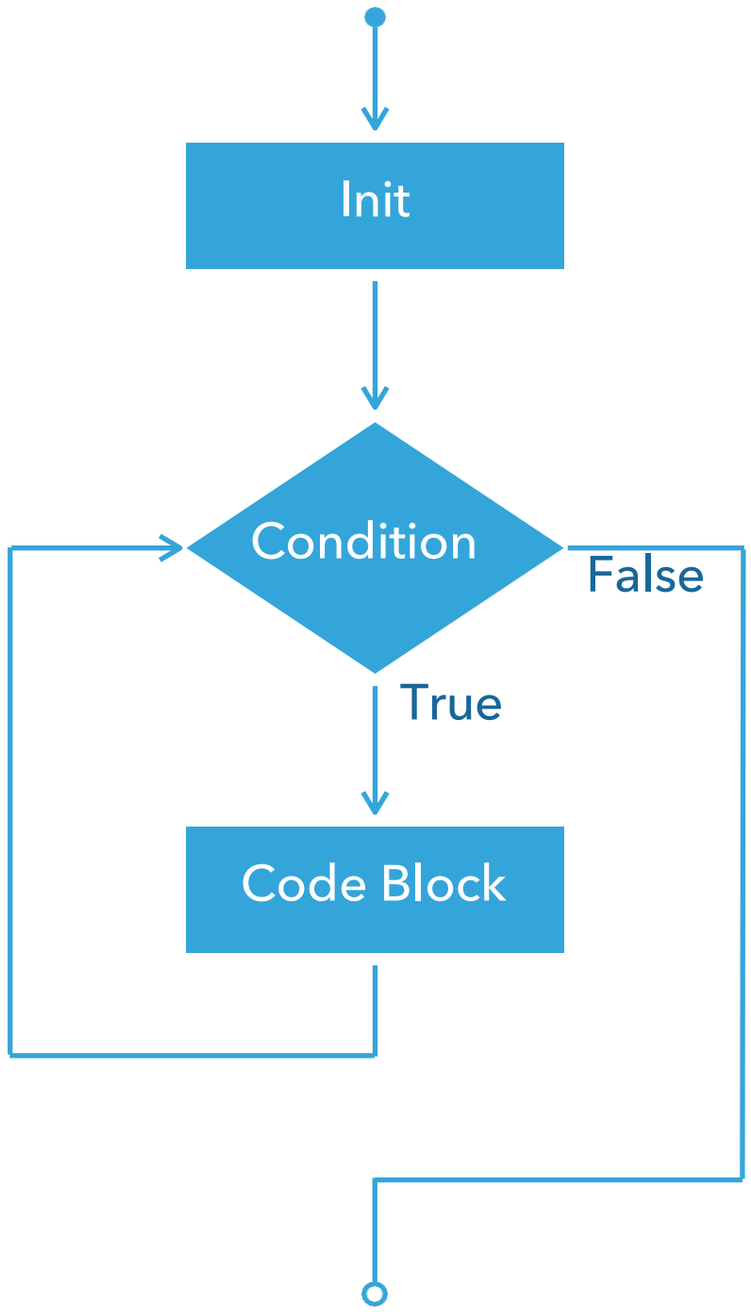
Console output

```
i = 0; i = 1; i = 2; i = 3; i = 4;

Process finished with exit code 0
```

FOR LOOP: SUMMARY

- Control structure that allows us to repeat certain operations by incrementing or decrementing and evaluating a loop counter
- Boolean expression is evaluated before the first iteration of the loop, hence executed zero or many times
- Usually used when number of iterations are known in advance

FOR LOOP: FLOWCHART



FOR LOOP: SYNTAX

for loop declaration keyword

Initialize couter variable(s)

Loop condition

Increment / Decrement operations on counter variable

```
for (initialization; expression; modification) {
    statement;
}
```

Statement(s) that executed inside of the loop body

FOR LOOP: CODE EXAMPLE

Code

```
for (int i = 0; i < 5; i++) {
    System.out.print("i=" + i + ";    ");
}
```

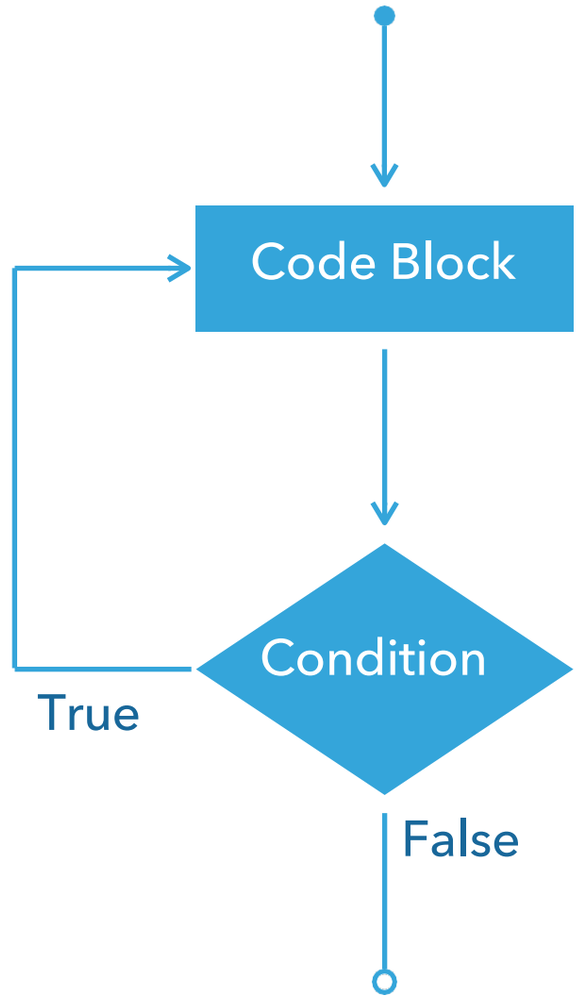
Console output

```
i = 0; i = 1; i = 2; i = 3; i = 4;

Process finished with exit code 0
```

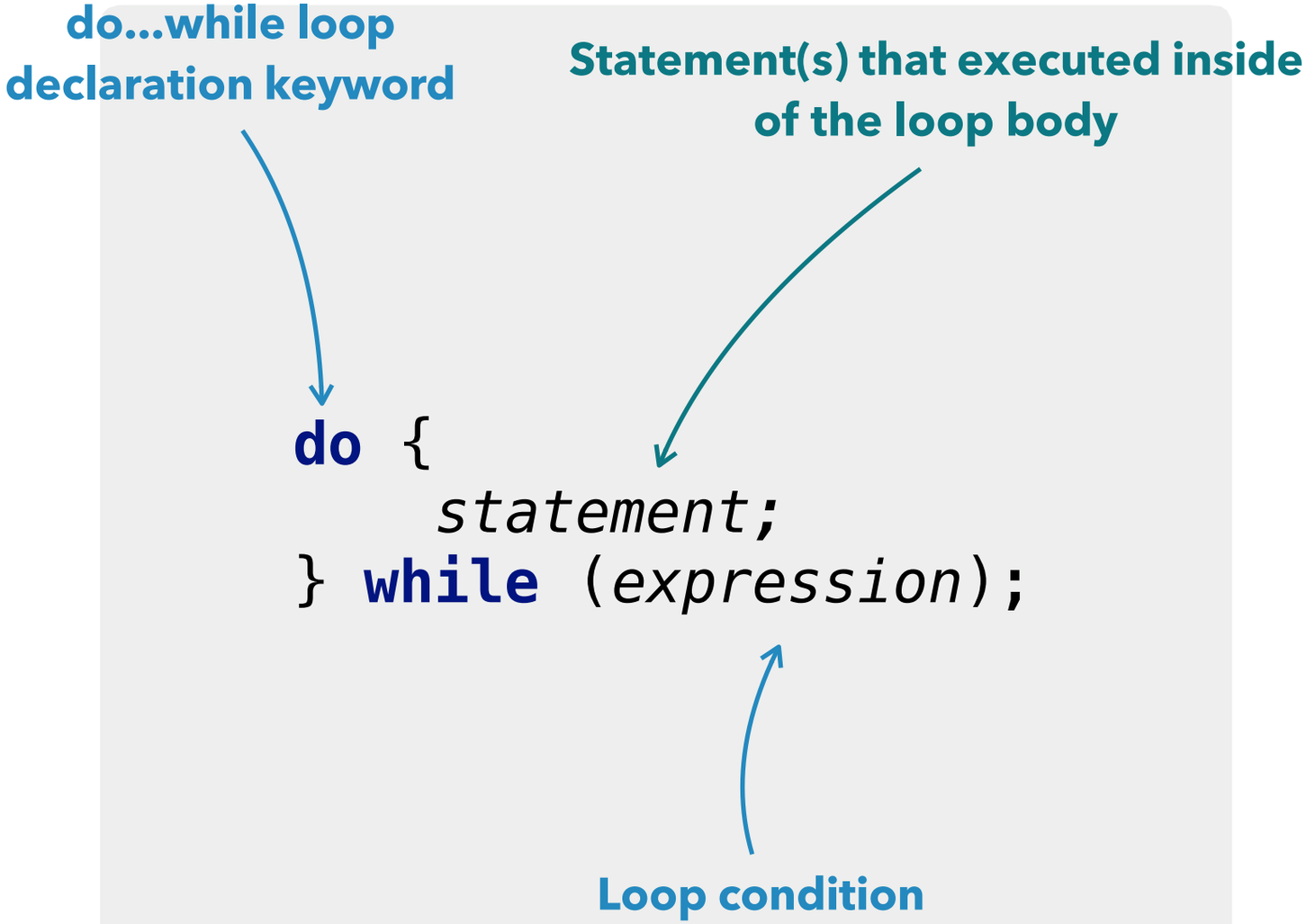
DO WHILE LOOP: SUMMARY

- Repeats a statement or block of statements while its controlling boolean expression is true
- Boolean expression is evaluated after the first iteration of the loop, hence executed one or many times
- Usually used when number of iterations are known in advance

DO WHILE LOOP: FLOWCHART



DO WHILE LOOP: SYNTAX



DO WHILE LOOP: CODE EXAMPLE

Code

```
int i = 0;
do {
    System.out.print("i = " + i + "; ");
    i++;
} while (i < 5);
```

Console output

```
i = 0; i = 1; i = 2; i = 3; i = 4;

Process finished with exit code 0
```

BRANCHING STATEMENTS IN LOOPS

BRANCHING STATEMENTS IN LOOPS

- Branching statements are used to change normal flow of execution based on some condition
- Branching statements available in loops:

1. break
2. continue

BREAK STATEMENT: OVERVIEW

- Terminates the innermost for, while, do...while statement
- When the break statement encountered, the loop is immediately terminated and the program control resumes at the next statement following the loop

BREAK STATEMENT: EXAMPLE

Code

```
for (int i = 0; i < 10; i++) {
    if (i == 3) {
        break;
    }
    System.out.print("i = " + i + "; ");
}
```

Console output

```
i = 0; i = 1; i = 2;

Process finished with exit code 0
```

CONTINUE STATEMENT: OVERVIEW

- In a for loop, the continue keyword causes control to immediately jump to the modification statement
- In a while or do...while loop, causes control to immediately jump to the boolean expression

CONTINUE STATEMENT: EXAMPLE

Code

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
        continue;
    }
    System.out.print("i = " + i + "; ");
}
```

Console output

```
i = 1; i = 3; i = 5; i = 7; i = 9;

Process finished with exit code 0
```

REFERENCES

- https://www.tutorialspoint.com/java/java_loop_control.htm
- https://www.baeldung.com/java-loops
- https://www.developer.com/java/data/using-different-types-of-java-loops-looping-in-java.html
- https://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html

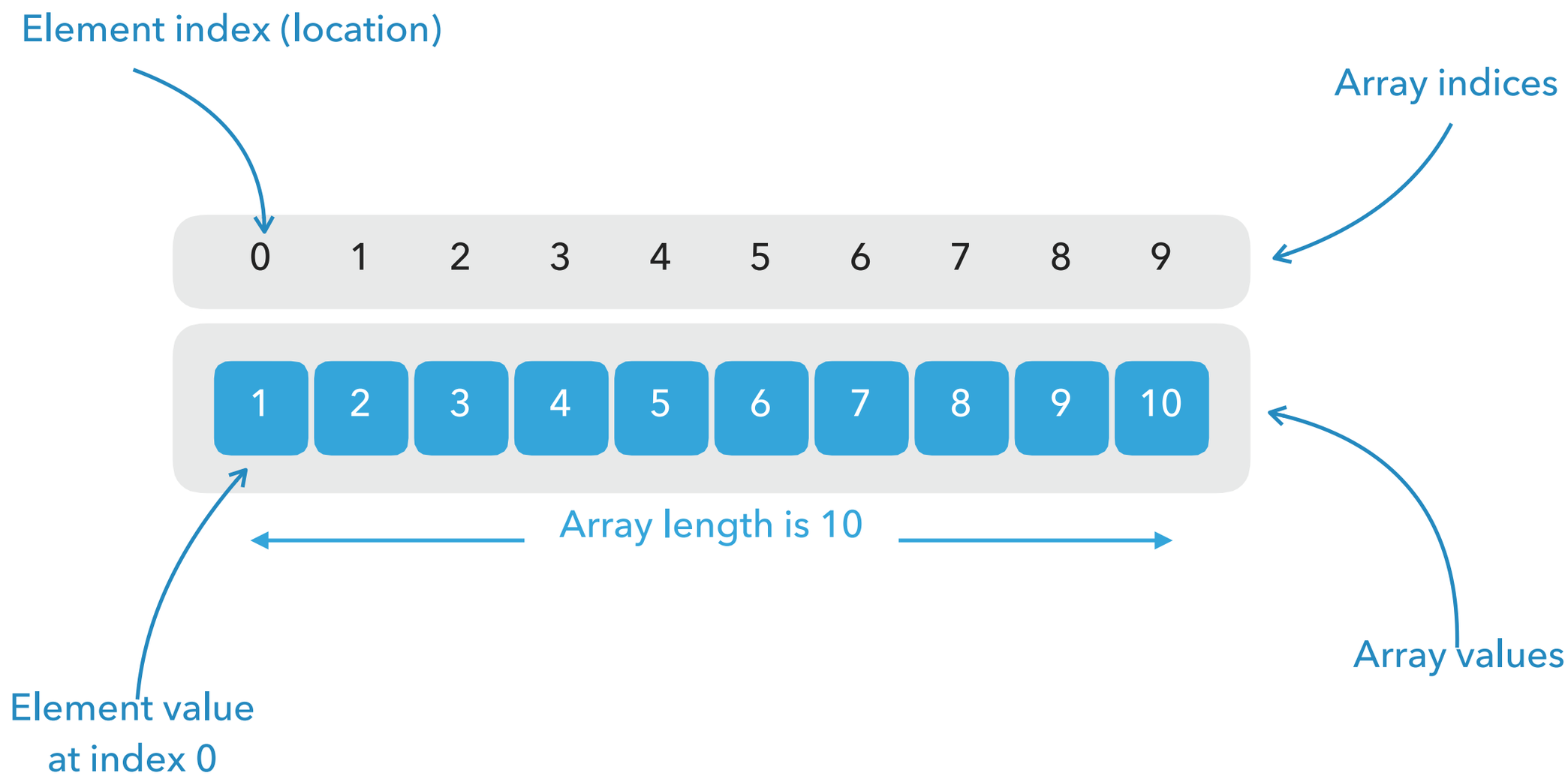
ARRAYS

OVERVIEW

DEFINITION

- An array is a container object that holds a fixed number of values of a single type
- The length of an array is established when the array is created
- After creation, its length is fixed

ARRAYS VISUALISATION



ARRAYS DECLARATION: SYNTAX

- Array declaration without instantiation

```
type[] name;
```

- Array declaration with instantiation

```
type[] name = new type[size];
```

- Array declaration with inline initialization

```
type[] name = {var1,.., varN};
```

ARRAY DECLARATION: INSTANTIATION CODE EXAMPLE

Code

```
int[] leapYears = new int[3];

leapYears[0] = 2020; leapYears[1]= 2016;  leapYears[2] = 2012;

System.out.println("Leap years = " + Arrays.toString(leapYears));
```

Console output

```
Leap years = [2020, 2016, 2012]

Process finished with exit code 0
```

ARRAY DECLARATION: INLINE INITIALIZATION CODE EXAMPLE

Code

```
int[] leapYears = {2020, 2016, 2012};

System.out.println("Leap years = " + Arrays.toString(leapYears));
```

Console output

```
Leap years = [2020, 2016, 2012]

Process finished with exit code 0
```

PROCESSING ARRAYS

WORKING WITH ARRAYS

- When working with arrays, loops are often used because of array iterable nature
- Array contains elements of the single type and size is fixed and known in advance

1. EXAMPLE: PRINTING ARRAY CONTENT

```
public class PrintingArrayDemo {
    public staticvoid main(String[]  args) {
        String[] alphabet = new String[5];

        alphabet[0] = "A";
        alphabet[1] = "B";
        alphabet[2] = "C";
        alphabet[3] = "D";
        alphabet[4] = "E";

        for (int i = 0; i < alphabet.length; i++){
            System.out.println("[" + i + "]: " + alphabet[i]);

        }

    }

}
```

2. EXAMPLE: SUM OF ARRAY ELEMENTS

```
public class SumOfArrayElementsDemo {

    public static void main(String[] args) {

        int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        int sum = 0;

        for (inti  = 0; i < numbers.length; i++) {
            sum += numbers[i];
        }

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

    }
}
```

3. EXAMPLE: FIND SMALLEST ELEMENT IN ARRAY

```
public class SmallestArrayElementDemo {

    public static void main(String[] args) {

        int[] numbers = {61, 97, 4, 37, 12};
        int min = numbers[0];

        for (int i = 0; i < numbers.length; i++) {
            if (numbers[i] < min) {
                min = numbers[i];
            }
        }

        System.out.println("min = " + min);

    }

}
```

ADVANCED ITERATION METHODS

FOR EACH (ENHANCED) LOOP: SUMMARY

- For each loop, also known as enhanced loop, is another way to traverse the array
- There is no use of the index or rather the counter variable
- Data type declared in the foreach must match the data type of the array that you are iterating
- Can access only current element
- Significantly reduces amount of code

FOR EACH (ENHANCED) LOOP: SYNTAX

for each loop declaration

```
type[] name = {var1,.., varN};

for (type item : name) {
    statements...
}
```

Iterator specification

Statement(s) that executed inside of the loop body

FOR EACH (ENHANCED) LOOP: CODE EXAMPLE

```
public class ForEachDemo {
    public staticvoid main(String[]  args) {
        String[] dogBreeds = {
                "Beagle",
                "Golden Retriever",
                "Pug",
                "Shiba Inu"
        };

        for (String breed : dogBreeds) {
            System.out.println(breed);
        }

    }

}
```

STATIC KEYWORD

OVERVIEW

STATIC KEYWORD OVERVIEW

- The keyword static indicates that the particular member belongs to a type itself, rather than to an instance of that type
- Only one instance of that static member is created which is shared across all instances of class
- Can be applied to the following elements:
  - Fields (variables)
  - Methods
  - Inner methods
  - Static code block

STATIC FIELDS

- Exactly a single copy of static field is created and shared among instances of that class
- No matter how many times class is initialized.. Always single copy of static field

1. STATIC FIELDS CODE EXAMPLE: MESSAGE CLASS

```
public class Message {

    public static int instancesCreated = 0;
    private String text;

    public Message(String text) {
        this.text = text;

        System.out.println("Creating message = '" + text + "'");
        instancesCreated++;
    }

}
```

2. STATIC FIELDS CODE EXAMPLE: MESSAGE CLASS

Code

```
System.out.println("Created = " + Message.instancesCreated);
Message greeting = new Message("Hi!");
Message question = new Message("How are you?");
Message farewell = new Message("Goodbye!");
System.out.println("Created = " + Message.instancesCreated);
```

Console output

```
Created = 0
Creating message = 'Hi!'
Creating message = 'How are you?'
Creating message = 'Goodbye!'
Created = 3
```

REASONS TO USE STATIC FIELDS

- When the value of variable is independent of objects
- When the value is supposed to be shared across all objects

KEY POINTS TO REMEMBER

- Since static fields belong to a class, they can be accessed directly using class name and don't need any object reference
- Static variables can only be declared at the class level
- Static fields can be accessed without object initialization
- Although static field can be accessed through reference, access via class name is preferred

STATIC METHODS

- Also belong to a class instead of the object
- Can be called without creating the object of the class in which they reside
- Generally used to perform an operation that is not dependent upon instance creation
- Widely used to create utility classes so that they can be obtained without creating a new object of these classes

1. STATIC METHODS CODE EXAMPLE: MATHS CLASS

```
public class QuickMaths {

    public static int min(int[] numbers) {
        if (numbers.length == 0) {
            return 0;
        }

        int min = numbers[0];

        for (int number : numbers) {
            if (number<  min) {
                min = number;
            }
        }

        return min;
    }

}
```

2. STATIC METHODS CODE EXAMPLE: MATHS CLASS

Code

```
int[] values = {44, 65, 61, 16, 89};

int result = QuickMaths.min(values);

System.out.println("result = " + result);
```

Console output

```
result = 16

Process finished with exit code 0
```

REASONS TO USE STATIC METHODS

- To access or manipulate static variables and other static members that don't depend upon objects
- Widely used in stateless utility classes

KEY POINTS TO REMEMBER

- Static methods cannot be overridden
- Instance methods can directly access both instance methods and instance variables
- Instance methods can directly access both static variables and static methods
- Static methods can access all static variables and other static methods
- Static methods cannot access instance variables and instance methods directly; only via object reference

REFERENCES

- https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html
- https://www.javatpoint.com/array-in-java
- https://www.baeldung.com/java-arrays-guide
- https://www.baeldung.com/java-static
- https://www.geeksforgeeks.org/static-keyword-java/