Module 1: Introduction to Java Fundamentals

Premanand S

Assistant Professor
School of Electronics Engineering
Vellore Institute of Technology
Chennai Campus

premanand.s@vit.ac.in

January 12, 2026

Module 1: Introduction to Java Fundamentals

- OOP Paradigm and Features of Java
- JVM, Bytecode, Java Program Structure
- Data Types, Variables, Naming Conventions
- Operators, Control and Looping Constructs
- One- and Multi-dimensional Arrays
- Enhanced for-loop
- Strings, StringBuffer, StringBuilder, Math Class
- Wrapper Classes

Looping Constructs — Introduction

A **loop** is used to execute a block of code **repeatedly** until a condition becomes false.

Looping Constructs — Introduction

A **loop** is used to execute a block of code **repeatedly** until a condition becomes false.

Why loops?

- Avoid writing the same code again and again
- Reduce program length
- Make programs efficient and clean

Looping Constructs — Introduction

A **loop** is used to execute a block of code **repeatedly** until a condition becomes false.

Why loops?

- Avoid writing the same code again and again
- Reduce program length
- Make programs efficient and clean

Types of loops in Java:

- `for`
- `while`
- `do--while`

for Loop — Explanation

The `for` **loop** is used when:

- Number of iterations is **known**
- We want controlled repetition

`for` Loop — Explanation

The `for` **loop** is used when:

- Number of iterations is **known**
- We want controlled repetition

In simple words:

- Start from a value
- Repeat till a condition is true
- Change the value each time

for Loop — Syntax

```
for (initialization; condition; update) {
 // statements
}
```

for Loop — Syntax

```
for (initialization; condition; update) {
 // statements
}
```

Parts explained:

- **Initialization** → starting point
- **Condition** → loop runs while true
- **Update** → changes value each round

for Loop — Python vs Java

Python

```
for i in range(1, 6):
print(i)
```

- Uses range()
- No data type declaration
- Indentation defines block
- Simple and readable

Java

```
for (int i = 1; i <= 5; i++) {
 System.out.println(i);
}
```

- Uses initialization, condition, update
- Data type must be declared
- Braces define block
- More control, more structure

Printing Numbers from 1 to 10 — Java `for` Loop

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

  // for loop to print numbers from 1 to 10
  for (int i = 1; i <= 10; i++) {
   System.out.println(i);
  }

 }
}
```

for Loop — How It Really Works

A for **loop** runs in three phases:

`for` Loop — How It Really Works

A `for` **loop** runs in three phases:

1. **Initialization** — runs once `int i = 1;`

for Loop — How It Really Works

A `for` **loop** runs in three phases:

1. **Initialization** — runs once `int i = 1;`
2. **Condition** — checked before every iteration `i <= 10`

`for` Loop — How It Really Works

A `for` **loop** runs in three phases:

1. **Initialization** — runs once `int i = 1;`
2. **Condition** — checked before every iteration `i <= 10`
3. **Update** — runs after every iteration `i++`

for Loop — How It Really Works

A `for` **loop** runs in three phases:

1. **Initialization** — runs once `int i = 1;`
2. **Condition** — checked before every iteration `i <= 10`
3. **Update** — runs after every iteration `i++`

Execution Flow:

Start → Check → Execute → Update → Repeat

for Loop — Boundary Conditions

Most bugs in loops come from **wrong limits**.

`for` Loop — Boundary Conditions

Most bugs in loops come from **wrong limits**.

Examples:

- `i < 10` → prints 1 to 9

`for` Loop — Boundary Conditions

Most bugs in loops come from **wrong limits**.
Examples:

- `i < 10` → prints 1 to 9
- `i <= 10` → prints 1 to 10

`for` Loop — Boundary Conditions

Most bugs in loops come from **wrong limits**.
Examples:

- `i < 10` → prints 1 to 9
- `i <= 10` → prints 1 to 10
- `i <= 0` → loop never runs

`for` Loop — Boundary Conditions

Most bugs in loops come from **wrong limits**.

Examples:

- `i < 10` → prints 1 to 9
- `i <= 10` → prints 1 to 10
- `i <= 0` → loop never runs

Key Lesson: Always double-check **start** and **end** values.

for Loop — Variable Scope

Block scope rule:

for Loop — Variable Scope

Block scope rule:

- Variable declared inside `for` exists only inside the loop block.

`for` Loop — Variable Scope

Block scope rule:

- Variable declared inside `for` exists only inside the loop block.

Example:

- `for (int i=1; ...)` → `i` not visible outside
- `int i;` before loop → `i` visible after loop

`for` Loop — Variable Scope

Block scope rule:

- Variable declared inside `for` exists only inside the loop block.

Example:

- `for (int i=1; ...)` → `i` not visible outside
- `int i;` before loop → `i` visible after loop

Why this matters: Prevents accidental misuse of loop counters.

for Loop — Pre vs Post Increment

In loop update part:

for Loop — Pre vs Post Increment

In loop update part:

- i++ and ++i behave the same

`for` Loop — Pre vs Post Increment

In loop update part:

- `i++` and `++i` behave the same

In conditions:

- `++i > 5` → increment first, then compare

`for` Loop — Pre vs Post Increment

In loop update part:

- `i++` and `++i` behave the same

In conditions:

- `++i > 5` → increment first, then compare
- `i++ > 5` → compare first, then increment

`for` Loop — Pre vs Post Increment

In loop update part:

- `i++` and `++i` behave the same

In conditions:

- `++i > 5` → increment first, then compare
- `i++ > 5` → compare first, then increment

Interview Tip: This is a classic placement trap.

`for` Loop — Infinite Loop Danger

A loop becomes **infinite** when:

for Loop — Infinite Loop Danger

A loop becomes **infinite** when:

- Condition is always true
- Update moves in the wrong direction

`for` Loop — Infinite Loop Danger

A loop becomes **infinite** when:

- Condition is always true
- Update moves in the wrong direction

Example:

- `for(int i=1; i<=10; i--)` → never stops

`for` Loop — Infinite Loop Danger

A loop becomes **infinite** when:

- Condition is always true
- Update moves in the wrong direction

Example:

- `for(int i=1; i<=10; i--)` → never stops

Rule: Update must move the variable **towards termination**.

for Loop — Semicolon Trap

A misplaced semicolon can break your loop logic.

for Loop — Semicolon Trap

A misplaced semicolon can break your loop logic.

Dangerous pattern:

- `for(int i=1; i<=10; i++);`

for Loop — Semicolon Trap

A misplaced semicolon can break your loop logic.

Dangerous pattern:

- `for(int i=1; i<=10; i++);`

What happens?

- Loop has an empty body
- Block after it runs only once

for Loop — Semicolon Trap

A misplaced semicolon can break your loop logic.

Dangerous pattern:

- `for(int i=1; i<=10; i++);`

What happens?

- Loop has an empty body
- Block after it runs only once

Key Lesson: Never put a semicolon after `for(...)`.

for Loop — Professional Insight

When you write:

```
for(int i=1; i<=10; i++)
```

`for` Loop — Professional Insight

When you write:

```
for(int i=1; i<=10; i++)
```

You are showing:

- Control over iteration

for Loop — Professional Insight

When you write:

```
for(int i=1; i<=10; i++)
```

You are showing:

- Control over iteration
- Awareness of boundaries

for Loop — Professional Insight

When you write:

```
for(int i=1; i<=10; i++)
```

You are showing:

- Control over iteration
- Awareness of boundaries
- Understanding of scope

for Loop — Professional Insight

When you write:

```
for(int i=1; i<=10; i++)
```

You are showing:

- Control over iteration
- Awareness of boundaries
- Understanding of scope
- Ability to avoid infinite loops

`for` Loop — Professional Insight

When you write:

```
for(int i=1; i<=10; i++)
```

You are showing:

- Control over iteration
- Awareness of boundaries
- Understanding of scope
- Ability to avoid infinite loops

In interviews: This simple loop reflects your coding maturity.

Tricky for Loop — Q1 (Semicolon Trap)

Predict what happens:

```
for (int i = 1; i <= 5; i++);
{
 System.out.println("Hello");
}
```

Tricky for Loop — Q1 (Semicolon Trap)

Predict what happens:

```
for (int i = 1; i <= 5; i++);
{
 System.out.println("Hello");
}
```

Think about:

- Is the loop really looping?
- How many times does "Hello" print?

Tricky for Loop — Q2 (Boundary Condition)

Predict the output:

```
for (int i = 1; i < 10; i++) {
 System.out.print(i + " ");
}
```

Tricky for Loop — Q2 (Boundary Condition)

Predict the output:

```
for (int i = 1; i < 10; i++) {
 System.out.print(i + " ");
}
```

Think about:

- Does this print till 10?
- Why or why not?

Tricky for Loop — Q3 (Pre/Post Increment)

Predict what happens:

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

Tricky `for` Loop — Q3 (Pre/Post Increment)

Predict what happens:

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

Think about:

- How does `i++` differ from `++i` here?
- Will this loop terminate?

Tricky for Loop — Q4 (Infinite Loop)

Predict what happens:

```
for (int i = 10; i > 0; i++) {
 System.out.println(i);
}
```

Tricky `for` Loop — Q4 (Infinite Loop)

Predict what happens:

```
for (int i = 10; i > 0; i++) {
 System.out.println(i);
}
```

Think about:

- Does this loop ever stop?
- Why or why not?

Tricky for Loop — Q5 (Scope Confusion)

Will this program compile?

```
for (int i = 1; i <= 3; i++) {
 System.out.println(i);
}
System.out.println(i);
```

Tricky for Loop — Q5 (Scope Confusion)

Will this program compile?

```
for (int i = 1; i <= 3; i++) {
 System.out.println(i);
}
System.out.println(i);
```

Think about:

- Where does variable i exist?
- How would you fix this?

while Loop — What is it?

The while **loop** is used to repeat a block of code **as long as a condition is true**.

`while` Loop — What is it?

The `while` **loop** is used to repeat a block of code **as long as a condition is true**.

In simple words:

- Check condition first
- If true → execute
- Repeat until condition becomes false

`while` Loop — What is it?

The `while` **loop** is used to repeat a block of code **as long as a condition is true**.

In simple words:

- Check condition first
- If true → execute
- Repeat until condition becomes false

Best used when:

- Number of iterations is **not known**
- Loop depends on a condition (input, sensor, status)

while Loop — Syntax

```
while (condition) {
 // statements
}
```

while Loop — Syntax

```
while (condition) {
 // statements
}
```

Key Rule:

- Condition is checked **before** every iteration
- If condition is false initially → loop never runs

for vs while

- Use for when:
    - Number of repetitions is known

`for` vs `while`

- Use `for` when:
    - Number of repetitions is known
- Use `while` when:
    - Repetition depends on a condition
    - Input-driven loops

while Loop — Python vs Java

Python

```
i = 1
while i <= 5:
print(i)
i += 1
```

- No data type declaration
- Indentation-based blocks

Java

```
int i = 1;
while (i <= 5) {
 System.out.println(i);
 i++;
}
```

- Type declaration required
- Braces define blocks

while Loop — Question

Write a Java program to print numbers from 1 to 10 using a while loop.

while Loop — Solution 1

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

  int i = 1;

  while (i <= 10) {
   System.out.println(i);
   i++;
  }
 }
}
```

while Loop — Question 1

Write a Java program to print numbers from 1 to 10 using a while loop.

while Loop — Solution 1

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

  int i = 1;

  while (i <= 10) {
   System.out.println(i);
   i++;
  }
 }
}
```

while — Condition Checked First

In a while loop:

while — Condition Checked First

In a while loop:

- Condition is checked **before** the loop body runs

while — Condition Checked First

In a while loop:

- Condition is checked **before** the loop body runs

Result:

- If condition is false initially → loop runs **zero times**

`while` — Condition Checked First

In a `while` loop:

- Condition is checked **before** the loop body runs

Result:

- If condition is false initially → loop runs **zero times**

Key Difference:

- Unlike `do--while`, `while` may not execute at all

`while` — Infinite Loop Risk

A `while` loop becomes infinite when:

`while` — Infinite Loop Risk

A `while` loop becomes infinite when:

- Condition always remains true
- Loop variable is never updated

`while` — Infinite Loop Risk

A `while` loop becomes infinite when:

- Condition always remains true
- Loop variable is never updated

Common Mistake:

- Forgetting to change the loop variable

while — Infinite Loop Risk

A while loop becomes infinite when:

- Condition always remains true
- Loop variable is never updated

Common Mistake:

- Forgetting to change the loop variable

Rule: Always ensure the condition will eventually become false.

`while` — Assignment vs Comparison

In Java:

`while` — Assignment vs Comparison

In Java:

- `=` → assignment
- `==` → comparison

`while` — Assignment vs Comparison

In Java:

- = → assignment
- == → comparison

Why this matters:

- `while(x = 1)` compile-time error
- `while(x == 1)`

while — Assignment vs Comparison

In Java:

- = → assignment
- == → comparison

Why this matters:

- `while(x = 1)` compile-time error
- `while(x == 1)`

Good News: Java prevents this dangerous mistake.

while — Semicolon Trap

A misplaced semicolon can break your loop.

`while` — Semicolon Trap

A misplaced semicolon can break your loop.

Dangerous pattern:

- `while(condition);`

`while` — Semicolon Trap

A misplaced semicolon can break your loop.

Dangerous pattern:

- `while(condition);`

What happens?

- Loop has an empty body
- Block after it runs only once

`while` — Semicolon Trap

A misplaced semicolon can break your loop.

Dangerous pattern:

- `while(condition);`

What happens?

- Loop has an empty body
- Block after it runs only once

Risk:

- Can lead to infinite loops

while — break and continue

break:

while — break and continue

break:

- Exits the loop immediately

while — break and continue

`break`:

- Exits the loop immediately

`continue`:

while — break and continue

`break`:

- Exits the loop immediately

`continue`:

- Skips the current iteration
- Moves to the next condition check

`while` — `break` and `continue`

`break`:

- Exits the loop immediately

`continue`:

- Skips the current iteration
- Moves to the next condition check

Key Understanding: Both change the natural flow of the loop.

while(true) — Safe Usage

An infinite loop can be useful when:

`while(true)` — Safe Usage

An infinite loop can be useful when:

- Program waits for events
- Server keeps running

`while(true)` — Safe Usage

An infinite loop can be useful when:

- Program waits for events
- Server keeps running

But:

- There must be a **break condition**

`while(true)` — Safe Usage

An infinite loop can be useful when:

- Program waits for events
- Server keeps running

But:

- There must be a **break condition**

Professional Rule: Never write `while(true)` without a safe exit.

`while` — Input-based Loop Danger

In input-driven loops:

while — Input-based Loop Danger

In input-driven loops:

- Always update input inside the loop

`while` — Input-based Loop Danger

In input-driven loops:

- Always update input inside the loop

Common Mistake:

- Reading input only once → infinite loop

`while` — Input-based Loop Danger

In input-driven loops:

- Always update input inside the loop

Common Mistake:

- Reading input only once → infinite loop

Rule: Every iteration must move closer to exit.

do--while Loop — What is it?

The do--while loop executes a block of code **at least once**, then repeats as long as the condition is true.

do--while Loop — What is it?

The do--while loop executes a block of code **at least once**, then repeats as long as the condition is true.

In simple words:

- Execute first
- Check condition later
- Repeat if condition is true

`do--while` Loop — What is it?

The `do--while` loop executes a block of code **at least once**, then repeats as long as the condition is true.

In simple words:

- Execute first
- Check condition later
- Repeat if condition is true

Best used when:

- The loop must run **at least once**
- Menus, user input, retries

do--while Loop — Syntax

```
do {
 // statements
} while (condition);
```

do--while Loop — Syntax

```
do {
 // statements
} while (condition);
```

Key Rule:

- Condition is checked **after** execution
- Semicolon after `while` is mandatory

while vs do--while

- while loop:
    - Condition checked first
    - Loop may not run at all

while vs do--while

- while loop:
  - Condition checked first
  - Loop may not run at all
- do--while loop:
  - Loop runs at least once
  - Condition checked later

do--while — Python vs Java

Python

```
while True:
print("Hello")
break
```

- Simulated using `while True`
- Break used to exit

Java

```
int i = 1;

do {
 System.out.println(i);
 i++;
} while (i <= 5);
```

- Native `do--while` support
- Guaranteed first execution

do--while — Runs At Least Once

The biggest difference of `do--while`:

`do--while` — Runs At Least Once

The biggest difference of `do--while`:

- Loop body executes **before** checking the condition

`do--while` — Runs At Least Once

The biggest difference of `do--while`:

- Loop body executes **before** checking the condition

Example:

- Even if condition is false, the loop runs once

`do--while` — Runs At Least Once

The biggest difference of `do--while`:

- Loop body executes **before** checking the condition

Example:

- Even if condition is false, the loop runs once

Key Insight:

Execution first — Decision later

`do--while` — Semicolon Rule

Unlike other loops, `do--while` requires:

do--while — Semicolon Rule

Unlike other loops, do--while requires:

- A semicolon after `while(condition);`

do--while — Semicolon Rule

Unlike other loops, `do--while` requires:

- A semicolon after `while(condition);`

Correct:

- `} while(condition);`

`do--while` — Semicolon Rule

Unlike other loops, `do--while` requires:

- A semicolon after `while(condition);`

Correct:

- `} while(condition);`

Common Mistake:

- Forgetting the semicolon → Compile-time error

do--while — Infinite Loop Risk

A do--while loop becomes infinite when:

do--while — Infinite Loop Risk

A do--while loop becomes infinite when:

- Condition is always true
- Loop variable is never updated

do--while — Infinite Loop Risk

A do--while loop becomes infinite when:

- Condition is always true
- Loop variable is never updated

Dangerous Example:

- `do { ... } while(true);`

do--while — Infinite Loop Risk

A do--while loop becomes infinite when:

- Condition is always true
- Loop variable is never updated

Dangerous Example:

- `do { ... } while(true);`

Professional Rule: Always ensure a clear exit condition.

do--while — Best Use Cases

do--while is perfect for:

do--while — Best Use Cases

do--while is perfect for:

- Menu-driven programs
- Login retries
- Input validation

do--while — Best Use Cases

do--while is perfect for:

- Menu-driven programs
- Login retries
- Input validation

Reason:

- These tasks must run at least once

while vs do--while

Same condition — different behavior:

while vs do--while

Same condition — different behavior:

- while: checks first → may not run
- do--while: runs first → always runs once

`while` vs `do--while`

Same condition — different behavior:

- `while`: checks first → may not run
- `do--while`: runs first → always runs once

Interview Favorite:

- `do { ... } while(false);`

`while` vs `do--while`

Same condition — different behavior:

- `while`: checks first → may not run
- `do--while`: runs first → always runs once

Interview Favorite:

- `do { ... } while(false);`

Result:

- Executes exactly once

do--while — Update Must Be Inside

Unlike `for`, `do--while` has:

do--while — Update Must Be Inside

Unlike for, do--while has:

- No separate update section

do--while — Update Must Be Inside

Unlike for, do--while has:

- No separate update section

So:

- Loop variable must be updated inside the block

do--while — Update Must Be Inside

Unlike for, do--while has:

- No separate update section

So:

- Loop variable must be updated inside the block

Mistake:

- Forgetting update → Infinite loop

do--while — break Behavior

break inside a do--while:

do--while — break Behavior

break inside a do--while:

- Exits the entire loop immediately

do--while — break Behavior

break inside a do--while:

- Exits the entire loop immediately

Not:

- Not just the current block

do--while — break Behavior

break inside a do--while:

- Exits the entire loop immediately

Not:

- Not just the current block

Key Learning:

- break always exits the loop structure

do--while — Interview Trap

Predict this:

do--while — Interview Trap

Predict this:

```
do { System.out.println("Test"); } while(false);
```

do--while — Interview Trap

Predict this:

```
do { System.out.println("Test"); } while(false);
```

Answer:

- Prints Test once

do--while — Interview Trap

Predict this:

```
do { System.out.println("Test"); } while(false);
```

Answer:

- Prints Test once

Why?

- Body runs before checking condition

Problem Statement — Password Validation

Write a Java program that:

Problem Statement — Password Validation

Write a Java program that:

- Asks the user to enter a password
- Correct password is: **java123**
- If wrong → print **“Incorrect password. Try again.”**
- If correct → print **“Access Granted!”** and stop

Problem Statement — Password Validation

Write a Java program that:

- Asks the user to enter a password
- Correct password is: **java123**
- If wrong → print **“Incorrect password. Try again.”**
- If correct → print **“Access Granted!”** and stop

Constraint:

- Use a `do--while` loop

Problem Statement — Password Validation

Write a Java program that:

- Asks the user to enter a password
- Correct password is: **java123**
- If wrong → print **“Incorrect password. Try again.”**
- If correct → print **“Access Granted!”** and stop

Constraint:

- Use a `do--while` loop

Why `do--while`**?**

- The program must run at least once

Solution — Password Validation using do--while

```
import java.util.Scanner;

public class PasswordValidation {
 public static void main(String[] args) {

  Scanner sc = new Scanner(System.in);
  String correctPassword = "java123";
  String userInput;

  do {
   System.out.print("Enter password: ");
   userInput = sc.nextLine();

   if (!userInput.equals(correctPassword)) {
    System.out.println("Incorrect password. Try again.");
   }

  } while (!userInput.equals(correctPassword));
```

Solution — Password Validation using do--while

```
  System.out.println("Access Granted!");
  sc.close();
 }
}
```

Thank You!

Stay Connected

Premanand S

Email:	premanand.s@vit.ac.in
Phone:	+91-7358679961
LinkedIn:	linkedin.com/in/premsanand
Instagram:	instagram.com/premsanand
WhatsApp Channel:	anandsDataX
Google Scholar:	Google Scholar Profile
GitHub:	github.com/anandprems