JAVA - Loop's Statements

Loops in Java

Loops are fundamental control structures in Java that allow repetitive execution of a block of code as long as a specified condition is true. They reduce redundancy, enhance readability, and provide efficiency in programming.

Introduction to Loops

- A **loop** in Java is a construct used to execute a set of statements repeatedly until a condition is satisfied.
- Loops help in avoiding repetitive code and facilitate automation of repetitive tasks.
- All Java loops belong to the category of **control flow statements**.

General benefits of loops:

- Simplify code by avoiding duplication
- Make programs more dynamic and flexible
- Handle repetitive tasks effectively (e.g., traversing arrays, processing data collections)

Types of Loops in Java

Java provides three primary loop structures and one enhanced loop:

1. For Loop
2. While Loop
3. Do-While Loop
4. Enhanced For Loop (for-each loop)

---

1. For Loop

The **for loop** is an entry-controlled loop (condition checked before execution). It is best suited when the number of iterations is known in advance.

Syntax:

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

Flow:

1. Initialization (executed once at the beginning).
2. Condition evaluated before every iteration.
3. Loop body executes if condition is true.
4. Update statement runs after each iteration.
5. Process repeats until condition is false.

Example:

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

Use cases:

- Iterating a fixed number of times
- Array or collection traversal (standard way)

---

2. While Loop

The **while loop** is also an entry-controlled loop. It is useful when the number of iterations is not predetermined and depends on a condition.

Syntax:

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

Flow:

1. Condition checked at the start.
2. If true, loop executes the body.
3. After execution, condition is checked again.
4. Continues until condition becomes false.

Example:

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

```
    i++;
}
```

Use cases:

- Running loops until a condition changes dynamically
- Reading input until end-of-file or a specific trigger

---

3. Do-While Loop

The **do-while loop** is an exit-controlled loop. It guarantees at least one execution of the loop body before the condition is tested.

Syntax:

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

Flow:

1. Loop body executed first.
2. Condition checked after execution.
3. Continues if condition is true.

Example:

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

Use cases:

- Menu-driven programs
- Scenarios where loop must run at least once

---

4. Enhanced For Loop (For-Each)

Introduced in Java 5, the **enhanced for loop** simplifies iteration over arrays and collections.

Syntax:

```
for(dataType element : collection) {
    // statements
}
```

Example:

```
int[] numbers = {10, 20, 30, 40};
for(int num : numbers) {
    System.out.println(num);
}
```

Limitations:

- Cannot modify array values directly
- Not suitable for situations where index tracking is required

---

Control Statements in Loops

Java provides **loop control statements** to alter flow inside loops.

- **break**: Terminates the loop entirely.

  ```
  for(int i=1; i<=10; i++) {
      if(i==5) break;
      System.out.println(i);
  }
  ```

- **continue**: Skips the current iteration and moves to the next.

  ```
  for(int i=1; i<=5; i++) {
      if(i==3) continue;
      System.out.println(i);
  }
  ```

- **return**: Exits from the current method, and hence terminates the loop too.

```
for (int i = 1; i <= 5; i++) {
     if (i == 3) {
     return;                        // ends the program
 }
     System.out.println(i);
}
    System.out.println("This won't run");  // not executed
```

Comparison of Loops

Feature	For Loop	While Loop	Do-While Loop	Enhanced For Loop
Condition Check	Before execution	Before execution	After execution	Implicit (collection/array)
Execution Guarantee	0 or more times	0 or more times	At least 1 time	Based on collection size

Feature	For Loop	While Loop	Do-While Loop	Enhanced For Loop
Best For	Fixed iteration count	Unknown iteration count	Must execute once scenarios	Array/collection traversal

Pros and Cons of Loops

Advantages:

- Code reusability and efficiency
- Dynamic handling of repetitive tasks
- Simplifies operations on data structures

Disadvantages:

- Risk of **infinite loops** if condition never becomes false
- Overuse may reduce readability
- Improper logic can lead to performance issues

Key Takeaways

- **For loop**: Best for fixed, countable iterations.
- **While loop**: Used when the number of iterations depends on runtime conditions.
- **Do-While loop**: Guarantees at least one execution.
- **Enhanced For loop**: Convenient for collections and arrays, but limited functionality.
- Always ensure loop termination conditions are correct to avoid infinite loops.