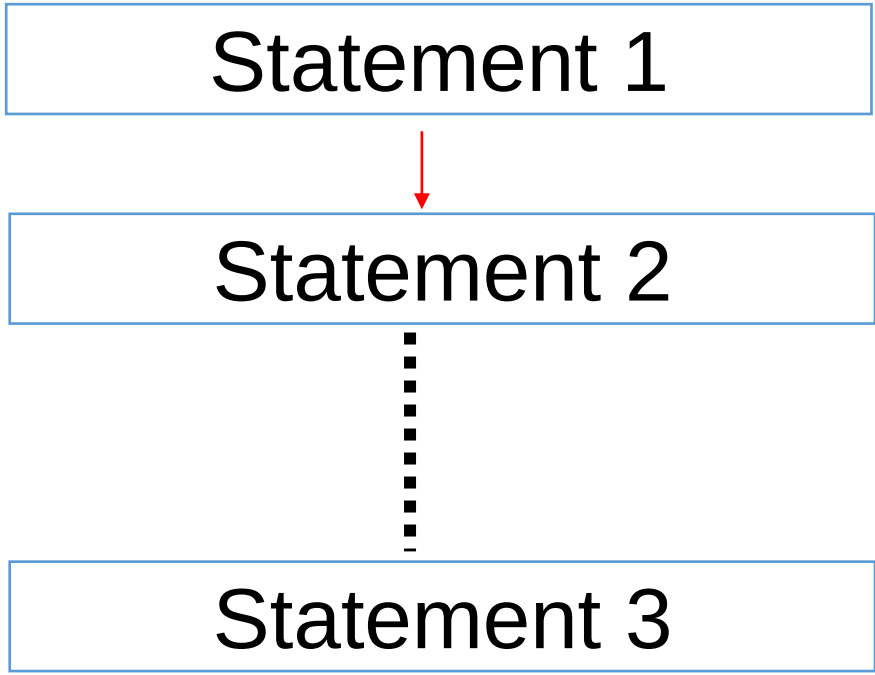
Control Statements

(Selection, Iteration & Jump)

Control Statements

- The default flow of control in a program is top to bottom, i.e. all the statements of a program are executed one by one in the sequence from top to bottom.
- This execution order can be altered with the help of control instructions.
- Java supports three types of control instructions
  - ✓Selection
  - ✓Iteration
  - ✓Jump

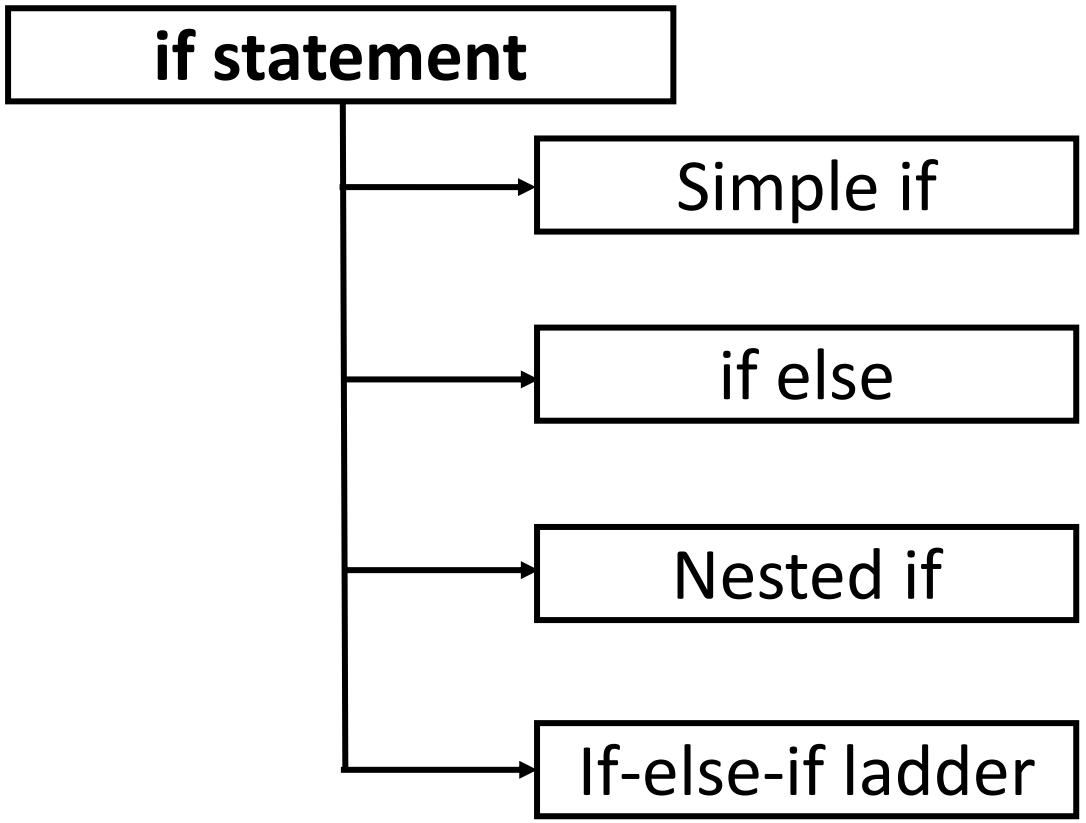


Selection

- Java supports two selection statements:
  - ✓if
  - ✓switch

if Statements

- The if statements can be categories into four subcategories.



Simple if

Syntax :

```
if (condition)
{
statement1;
}
```

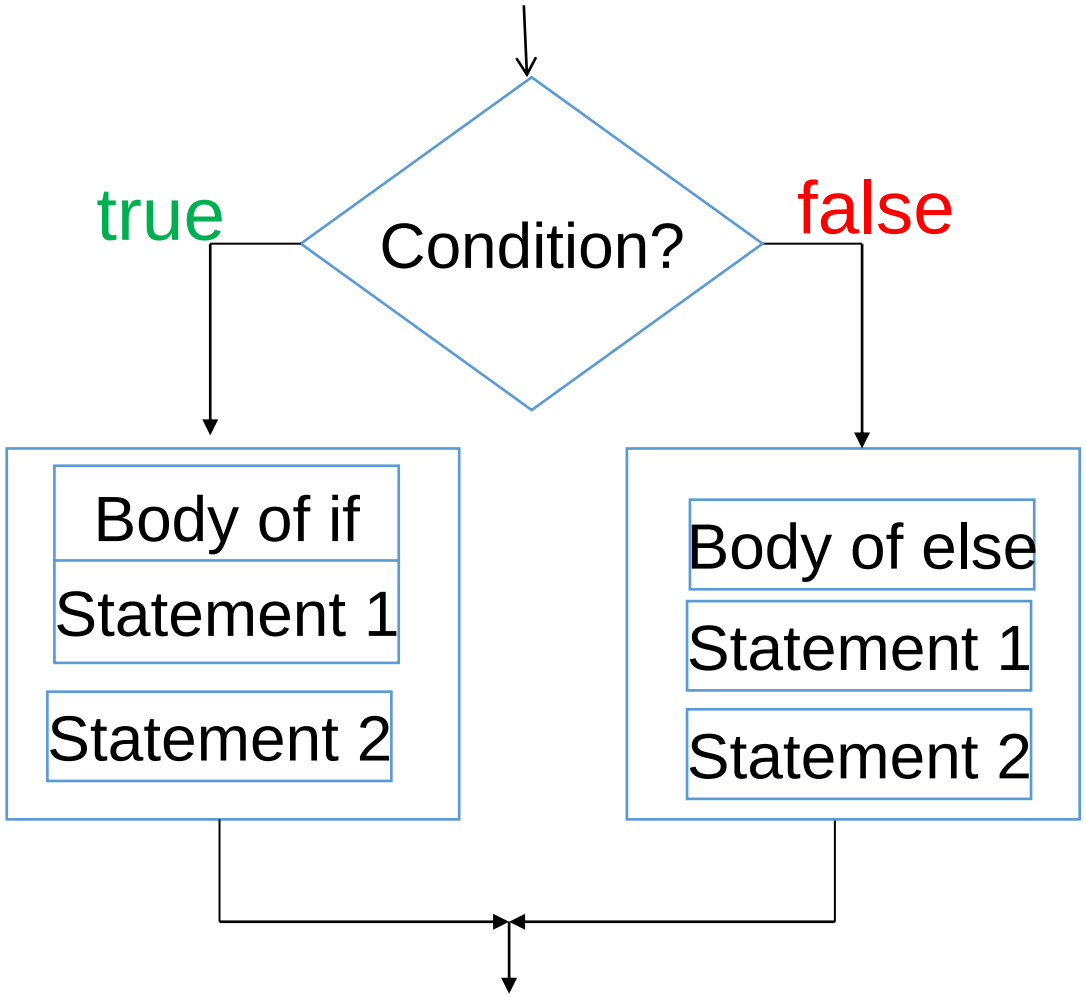
Purpose: The statements will be evaluated if the value of the condition is true

If else

- The if statement is Java's conditional branch statement.
- It can be used to route program execution through two different paths.

```
if (condition)
        statement1;
else
        statement2;
```

- Each statement may be a single statement or a compound statement enclosed in curly braces.
- The condition is any expression that returns a boolean value.
- The else clause is optional.



If Statements

- The conditional expression may be a simple expression or a compound expression.
- Each statement block may have a single or multiple statements to be executed.
- In case there is a single statement to be executed then it is not mandatory to enclose it in curly braces ({}) but if there are multiple statements then they must be enclosed in curly braces ({})
- The else clause is optional and needs to be included only when some action is to be taken if the test condition evaluates to false.

Example:-

```
int a=10, b=5;
if(a > b)
        System.out.println("a is greater than b");
else
        System.out.println("b is greater than a");
```

Ternary Operator or Conditional operator

- Ternary operator returns the statement depends upon the given expression result.

```
Test_expression ? statement1: statement2
```

- If the given test condition is true, then conditional operator will return statement1. If the condition is false, then statement2 is returned.

Example:-

```
int a=10, b=5, largest;
largest = (a> b)? a: b;
        System.out.println("The Largest Number = " + largest);
```

Nested if statement

- A nested if is an if statement that is the target of another if or else.

- Example:- Find the largest number among three.

```
int n1 = 5, n2 = 10, n3 = 1;
 if(n1 > n2)
{
        if(n1 > n3)
           System.out.println( "n1 is the largest number ”+ n1);
        else
           System.out.println("n3  is the largest number ” +n3);
}
else
{
        if(n2 > n3)
           System.out.println("n2 is the largest number ” + n2);
        else
           System.out.println ("n3  is the largest number ” +n3);
}
```

If-else-if Ladder

- A nested if is an if statement that is the target of another if or else.

```
if(condition)
        statement;
else if(condition)
        statement;
else if(condition)
        statement;
...
else
        statement;
```

- The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed.
- If none of the conditions is true, then the final else statement will be executed. The final else acts as a default condition;

If-else-if Ladder

- A nested if is an if statement that is the target of another if or else.

- Example:- Find the largest number among three.

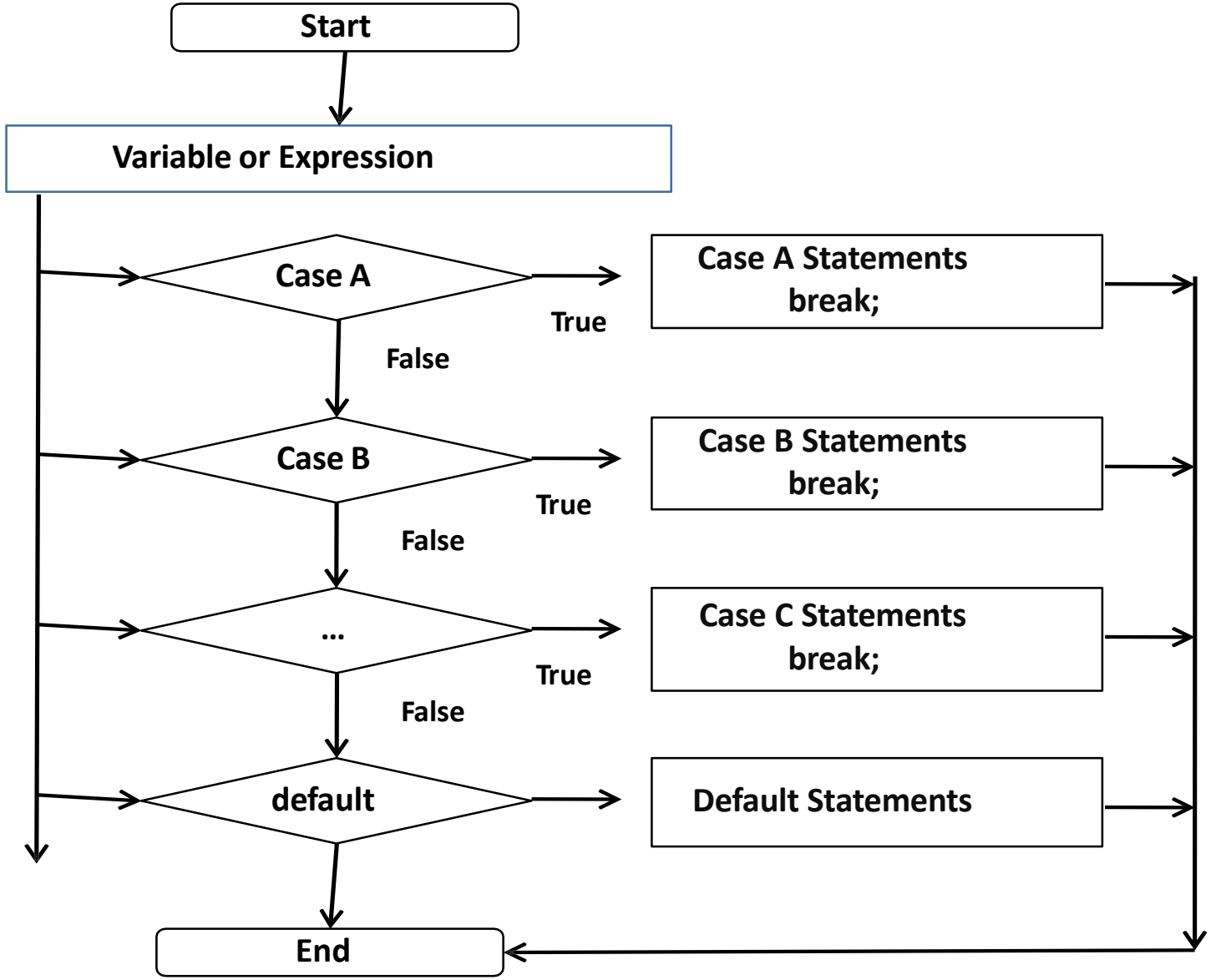
```
int n1 = 5, n2 = 10, n3 = 1;
if(n1 > n2 && n1 > n3)
        System.out.println( "n1 is the largest number ”+ n1);
else if(n2>n1 && n2>n3)
        System.out.println("n2 is the largest number ” + n2);
else
        System.out.println ("n3  is the largest number ” +n3);
```

Switch statement

- The switch statement provides an easy way to dispatch execution to different parts of your code based on the value of an expression.
- It often provides a better alternative than a large series of else-if ladder statements.

```
switch (expression) {
        case value1:
                // statement sequence
                break;
        case value2:
                // statement sequence
                break;
        ...
         case valueN :
                // statement sequence
                break;
        default:
                // default statement sequence
}
```

Switch statement



Switch statement

- The expression must be of type byte, short, int, char, or an enumeration.
- In JDK 7, expression can also be of type String.
- Each value specified in the case statements must be a unique constant expression (such as a literal value).
- Duplicate case values are not allowed.
- However, the **default** statement is **optional**. If no case matches and no default is present, then no further action is taken.
- The **break** statement is used inside the switch to terminate a statement sequence. **break** statements are optional.

Switch statement

- Example:-

```
for(int i=0; i<6; i++)
    switch(i) {
        case 0:
            System.out.println("i is zero.");
            break;
        case 1:
            System.out.println("i is one.");
            break;
        case 2:
            System.out.println("i is two.");
            break;
        case 3:
            System.out.println("i is three.");
            break;
        default:
            System.out.println("i is greater than 3.");
    }
```

Switch statement

- Example use of **break**:-

```
for(int i=0; i<10; i++)
    switch(i) {
        case 0:
        case 1:
        case 2:
            System.out.println("i is less than 3");
            Break;
        case 3:
        case 4:
            System.out.println("i is less than 5");
            break;
        default:
            System.out.println("i is 5 or more");
    }
```

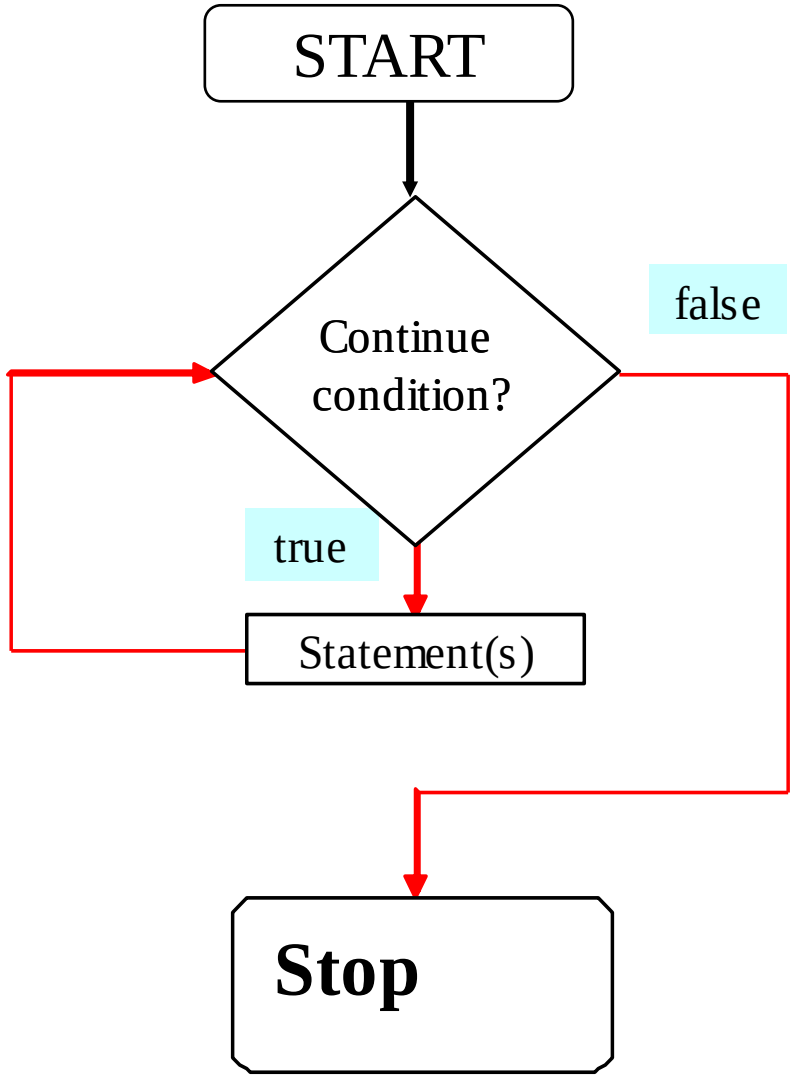
Iteration or loop Statements

- Java supports three iteration statements:
  - ✓While
  - ✓do-while
  - ✓For

- A loop repeatedly executes the same set of instructions until a termination condition is met.

While

```
while(condition) {
// body of loop
}
```



While

- Java supports three iteration statements:

```
while(condition) {
// body of loop
}
```

- The condition can be any Boolean expression.
- The body of the loop will be executed as long as the conditional expression is true.
- When condition becomes false, control passes to the next line of code immediately following the loop.
- The curly braces are unnecessary if only a single statement is being repeated

While

- Example: Print the numbers from 10 to 1.

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

While

- Example: Print the numbers from 10 to 1.

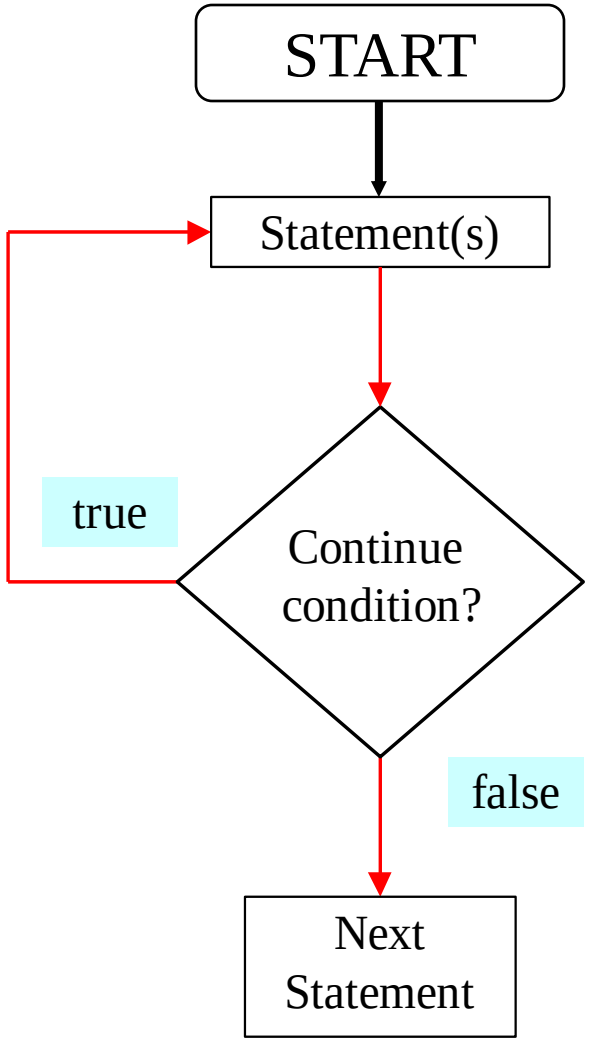
```
int i = 10;
while(i > 0)
{
        System.out.println(i);
        i--;
}
```

Output:

```
10
9
8
7
6
5
4
3
2
1
```

Do-while

```
do {
// body of loop
} while (condition);
```



Do-while

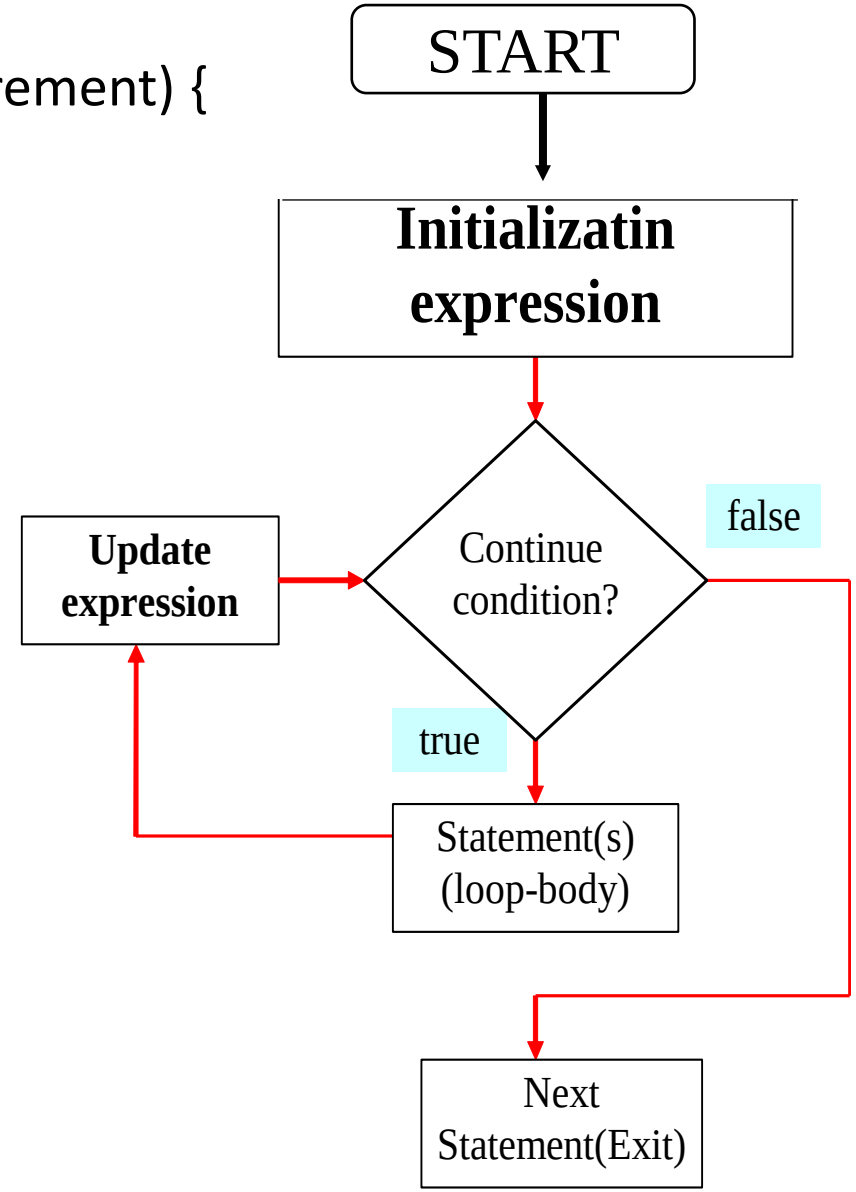
- While loop is initially false, then the body of the loop will not be executed at all.
- However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with.
- The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop.

```
do {
// body of loop
} while (condition);
```

- Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression.
- If this expression is true, the loop will repeat. Otherwise, the loop terminates.

For

```
for(initialization; condition; increment/decrement) {
        // body
        }
```



For

```
for(initialization; condition; increment/decrement)
{
      // body
}
```

- When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable
- The initialization expression is executed only once.
- Next, condition is evaluated. This must be a Boolean expression.
- It usually tests the loop control variable against a target value.
- If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.
- Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable.

For

```
for(initialization; condition; increment/decrement)
{
    // body
}
```

- The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.
- We can include more than one statement in the initialization and iteration portions of the for loop using comma.

```
for(a=1, b=10; a<b; a++, b--)
```

- **for** loop variation: we can leave all three parts of the for empty.

For-each

- In JDK 5, a second form of **for** was defined that implements a "**for-each**" style loop.
- A for-each style loop is designed to cycle through a **collection of objects**, such as an **array**, in **strictly sequential fashion**, from **start to finish**.
- The advantage of this approach is that **no new keyword** is required, and no preexisting code is broken

```
for(type itr-var : collection)
statement-block
```

- Here, type specifies the **type** and **itr-var** specifies the name of an iteration variable that will receive the elements from a collection (eg. array), one at a time, from beginning to end.
- With each iteration of the loop, the next element in the collection is retrieved and stored in itr-var. The loop repeats until all elements in the collection have been obtained.

For-each

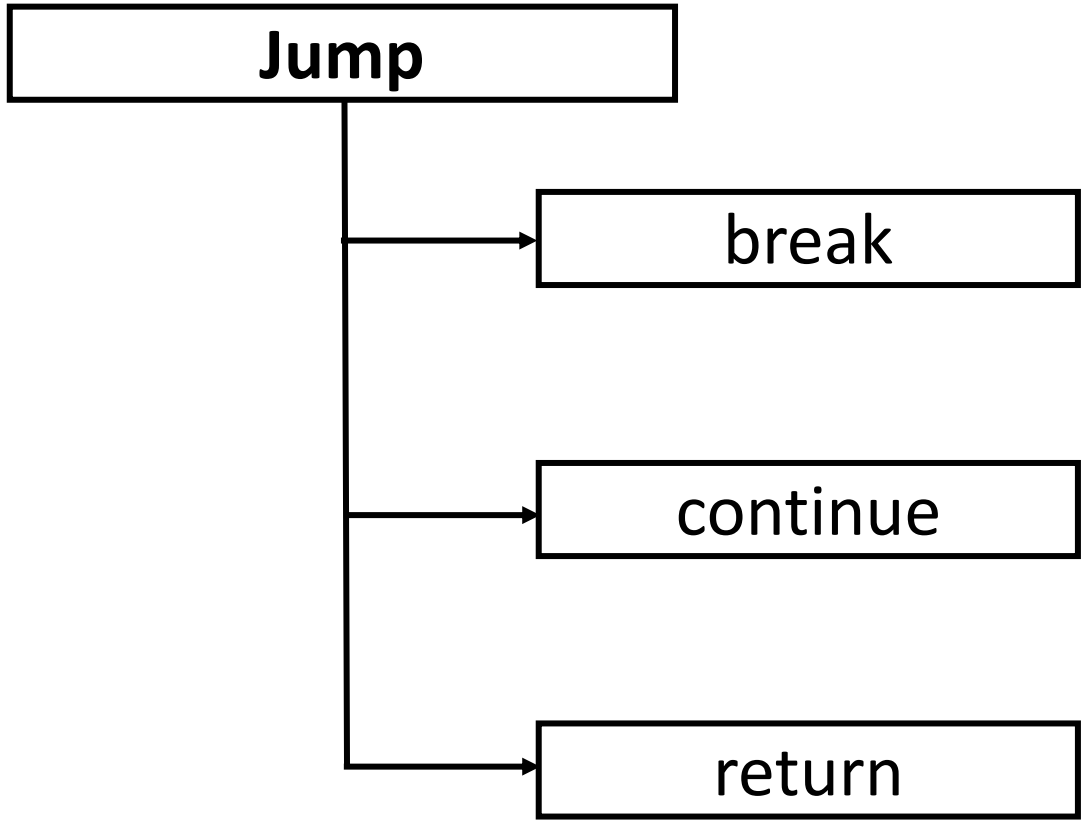
- The iteration variable receives values from the collection, type must be the same as (or compatible with) the elements stored in the collection

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0; i < 10; i++)
sum += nums[i];
```

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums)
sum += x;
```

Jump statements

- Java supports three jump statements:



- These statements transfer control to another part of your program.

Break

- In Java, the break statement has three uses.
- First, as you have seen, it terminates a statement sequence in a **switch** statement.
- Second, it can be used to **exit a loop**.
- Third, it can be used as a **form** of **goto**.

break to Exit a Loop

- We can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.
- When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

```
int i = 0;
while(i < 100) {
        if(i == 10)
                break;
        System.out.println("i: " + i);
        i++;
}
System.out.println("Loop complete.");
```

break to Exit a Loop

- We can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.
- When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

```
int i = 0;
while(i < 100) {
        if(i == 10)
                break; // terminate loop if i is 10
        System.out.println("i: " + i);
        i++;
}
System.out.println("Loop complete.");
```

- **Output**:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

break to Exit a Loop

- When used inside a set of nested loops, the break statement will only break out of the innermost loop.

```
for(int i=0; i<3; i++) {
        System.out.print("Pass " + i + ": ");
        for(int j=0; j<100; j++) {
                if(j == 10)
                        break;
                System.out.print(j + " ");
        }
        System.out.println();
}
System.out.println("Loops complete.");
```

break to Exit a Loop

- When used inside a set of nested loops, the break statement will only break out of the innermost loop.

```
for(int i=0; i<3; i++) {
        System.out.print("Pass " + i + ": ");
        for(int j=0; j<100; j++) {
                if(j == 10)
                        break;          // terminate loop if j is 10
                System.out.print(j + " ");
        }
        System.out.println();
}
System.out.println("Loops complete.");
```

- **output**:

  Pass 0: 0 1 2 3 4 5 6 7 8 9
  Pass 1: 0 1 2 3 4 5 6 7 8 9
  Pass 2: 0 1 2 3 4 5 6 7 8 9
  Loops complete.

break to Exit a Loop

- More than one **break** statement may appear in a loop
- **break** was not designed to provide the normal means by which a loop is terminated. The loop's conditional expression serves this purpose.
- The break statement should be used to cancel a loop only when some sort of special situation occurs.

break as a Form of Goto

- Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner.
- This usually makes goto-ridden code hard to understand and hard to maintain.
- It also prohibits certain compiler optimizations.
- However, a few places where the **goto** is a valuable.
- For example, **goto** can be useful when you are exiting from a deeply nested set of loops.
- Java defines an expanded form of the break statement.
- For example, break out of one or more blocks of code. These blocks need not be part of a loop or a switch. They can be any block.
- This form of break works with a **label**. So, break gives you the benefits of a goto without its problems.

break as a Form of Goto

- The general form of the labeled break statement is shown here:

  break label;

- label is the name of a label that identifies a block of code. This can be a standalone block of code.
- To name a block, put a label at the start of it.
- A label is any valid Java identifier followed by a colon.
- Once you have labeled a block, you can then use this label as the target of a break statement.

break as a Form of Goto

- Example:

```
boolean t = true;
first: {
        second: {
                third: {
                        System.out.println("Before the break.");
                        if(t)
                                break second;
                        System.out.println(“This is in third block ");
                }
        System.out.println(" This is in second block ");
        }
System.out.println("This is in first block.");
}
```

break as a Form of Goto

- Example:

```
boolean t = true;
first: {
    second: {
        third: {
            System.out.println("Before the break.");
            if(t)
                break second; // break out of second block
            System.out.println(“This is in third block ");
        }
    System.out.println(" This is in second block ");
    }
System.out.println("This is in first block.");
}
```

- **output**:

  Before the break.

  This is in first block.

break as a Form of Goto

- One of the most common uses for a labeled break statement is to exit from nested loops.

```
outer: for(int i=0; i<3; i++) {
                System.out.print("Pass " + i + ": ");
                for(int j=0; j<100; j++) {
                        if(j == 10)
                                break outer;
                        System.out.println(j + " ");
                }
        System.out.println("This is in outer loop");
        }
System.out.println("Loops complete.");
```

break as a Form of Goto

- One of the most common uses for a labeled break statement is to exit from nested loops.

```
outer: for(int i=0; i<3; i++) {
            System.out.print("Pass " + i + ": ");
            for(int j=0; j<100; j++) {
                  if(j == 10)
                        break outer; // exit both loops
                  System.out.println(j + " ");
            }
      System.out.println("This is in outer loop");
      }
System.out.println("Loops complete.");
```

- **output**:

```
Pass 0: 0
1
2
3
4
5
6
7
8
9
Loops complete.
```

break as a Form of Goto

- Keep in mind that you cannot break to any label which is not defined for an enclosing block.

```
one: for(int i=0; i<3; i++) {
                System.out.print("Pass " + i + ": ");
        }

for(int j=0; j<100; j++) {
        if(j == 10)
                break one;
        System.out.print(j + " ");
}
```

break as a Form of Goto

- Keep in mind that you cannot break to any label which is not defined for an enclosing block.

```
one: for(int i=0; i<3; i++) {
                System.out.print("Pass " + i + ": ");
     }

for(int j=0; j<100; j++) {
     if(j == 10)
                break one; // WRONG
     System.out.print(j + " ");
}
```

- **output**:

```
error: undefined label: one
                break one;
                ^
```

- Since the loop labeled one does not enclose the break statement, it is not possible to transfer control out of that block.

continue

- Sometimes, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration.
- The continue statement performs such an action.
- In while and do-while loops, a continue statement causes control to be transferred directly to the **conditional expression** that controls the loop.
- In a for loop, control goes first to the **iteration portion** of the for statement and then to the conditional expression.

continue

- Example:

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

continue

- Example:

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

- **output**:

  0 1
  2 3
  4 5
  6 7
  8 9

- This code uses the % operator to check if **i** is even. If it is, the loop continues without printing a newline.

continue

- **continue** may specify a label to describe which enclosing loop to continue.

```
outer: for (int i=0; i<10; i++) {
                for(int j=0; j<10; j++) {
                        if(j > i) {
                                System.out.println();
                                continue outer;
                        }
                        System.out.print(" " + (i * j));
                }
        }
System.out.print("\nNot in the loop");
```

continue

- **continue** may specify a label to describe which enclosing loop to continue.

```
outer: for (int i=0; i<10; i++) {
            for(int j=0; j<10; j++) {
                if(j > i) {
                    System.out.println();
                    continue outer;
                }
                System.out.print(" " + (i * j));
            }
        }
System.out.print("\nNot in the loop");
```

- **output**:

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
Not in the loop
```

- The continue statement terminates the loop counting j and continues with the next iteration of the loop counting i.

return

- The return statement is used to explicitly return from a method.
- It causes program control to transfer back to the caller of the method.
- At any time in a method, the return statement can be used to cause execution to branch back to the caller of the method.
- Thus, the return statement immediately terminates the method in which it is executed.

```
boolean t = true;
System.out.println("Before the return.");
if(t)
        return; // return to caller
System.out.println(“After the return.");
```

return

- The return statement is used to explicitly return from a method.
- It causes program control to transfer back to the caller of the method.
- At any time in a method, the return statement can be used to cause execution to branch back to the caller of the method.
- Thus, the return statement immediately terminates the method in which it is executed.

```
boolean t = true;
System.out.println("Before the return.");
if(t)
        return; // return to caller
System.out.println(“After the return.");
```

- **output**:

  Before the return.

- Here, return causes execution to return to the Java run-time system, since it is the run-time system that calls main( )