



Chapter 3: **Decision and Repetition Statements**

- Types of Statements in Java
- A **statement specifies an action in a Java program**, such as assigning the sum of x and y to z, printing a message to the standard output, writing data to a file, etc.
- Statements in Java can be broadly classified in to three categories:
 1. Declaration Statement
 2. Expression Statement
 3. Control flow Statement



Declaration Statement

- A declaration statement is used to declare a variable. For example
 - `int num;`
 - `int num2=100;`
 - `String str;`
 - `float area;`



Expression Statement

- An expression with a semicolon at the end is called an expression statement. For example,
 - `num++;` // increment expression
 - `num=100;` // assignment expression
 - **`someMethod(par1, par2);`** // Method invocation expressions



Control Flow Statement

- By default, all statements in a Java program are executed **in the order they appear** in the program.
- Sometimes you may want to execute a set of statements repeatedly for a number of times or as long as a particular condition is true.
- All of these are possible in Java using **control flow statements**. If block, while loop, and for loop statements are examples of control flow statements.



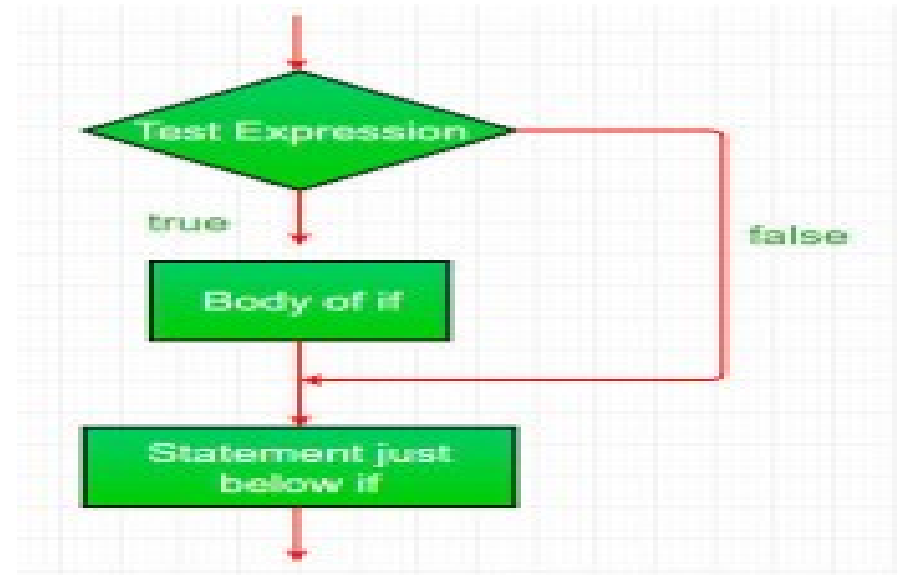
Decision Making Statements

- There are two types of decision making statements in Java. They are:
 - If statements
 - Switch statements
- **The IF Statement**
 - An if statement consists of a **Boolean expression** followed by one or more statements.



Syntax:

```
if(Boolean_expression)
{
    // Statements will
    // execute if the Boolean
    // expression is true
}
```





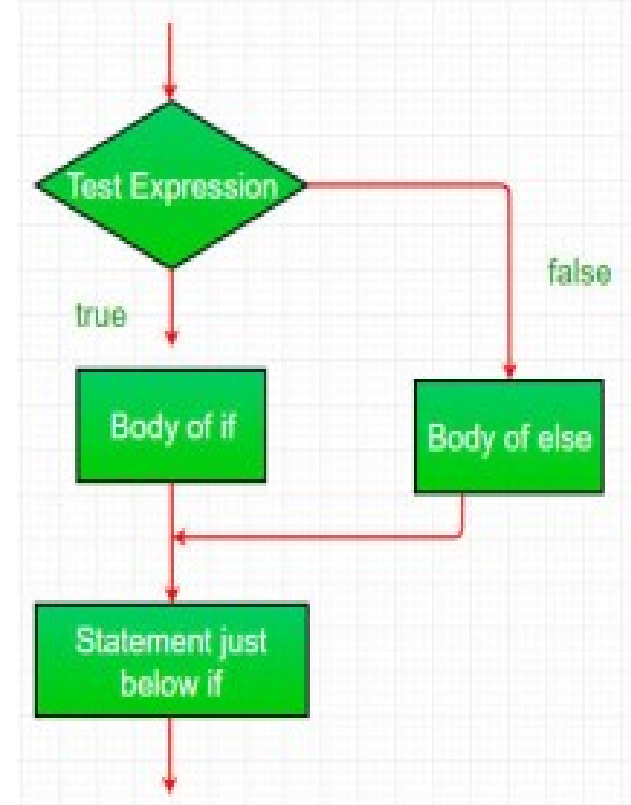
statement example

```
public class IfTest {  
  
    public static void main(String args[]) {  
        int x = 10;  
  
        if( x < 20 ) {  
            System.out.print("This is if statement");  
        }  
    }  
}
```

The if----else Statement

- An if statement can be followed by an optional *else* statement , which executes when the Boolean expression is false.
- Syntax

```
if(Boolean_expression){  
    //Executes when the Boolean expression is true  
}else{  
    //Executes when the Boolean expression is false  
}
```





If---else example

```
public class IfElseTest {  
  
    public static void main(String args[]) {  
        int x = 30;  
  
        if( x < 20 ){  
            System.out.print("This is if statement");  
        }else{  
            System.out.print("This is else statement");  
        }  
    }  
}
```



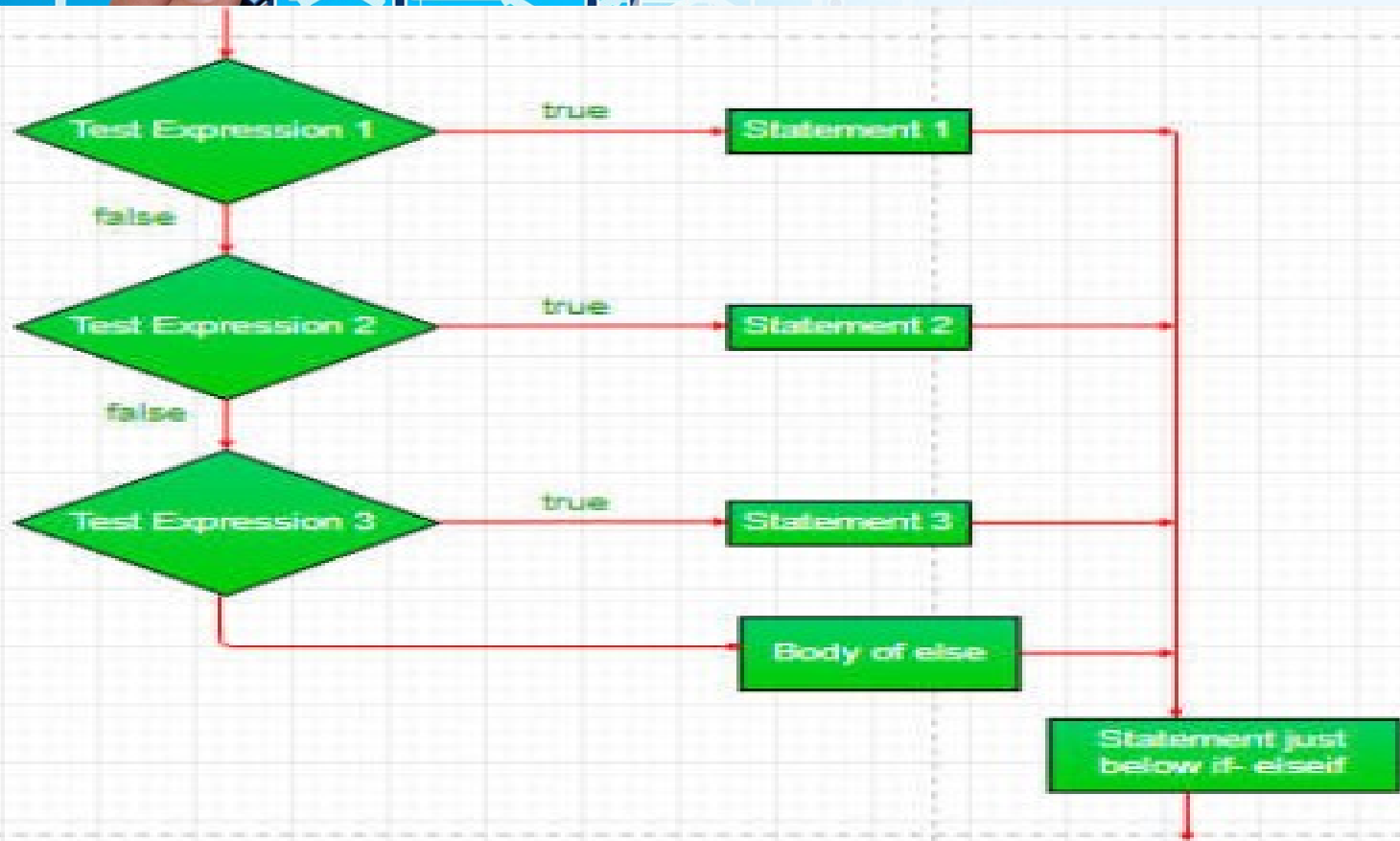
The if...else if...else Statement

- An if statement can be followed by an optional *else if...else* statement, which is very useful to test various conditions using single **if...else if** statement.
- When using if , else if , else statements there are few points to keep in mind.
 - An if can have zero or one else's and it must come after any else if's.
 - An if can have zero to many **else if's** and they must come before the else.
 - Once an else if succeeds, none of the remaining else if's or else's will be tested.



- Syntax :

```
if(Boolean_expression 1){  
    //Executes when the Boolean expression 1 is true  
}else if(Boolean_expression 2){  
    //Executes when the Boolean expression 2 is true  
}else if(Boolean_expression 3){  
    //Executes when the Boolean expression 3 is true  
}else {  
    //Executes when the none of the above condition is true.  
}
```





if---else if---else example

```
public class IfElseIfElseTest {  
  
    public static void main(String args[]){  
        int x = 30;  
  
        if( x == 10 ){  
            System.out.print("Value of X is 10");  
        }else if( x == 20 ){  
            System.out.print("Value of X is 20");  
        }else if( x == 30 ){  
            System.out.print("Value of X is 30");  
        }else{  
            System.out.print("This is else statement");  
        }  
    }  
}
```

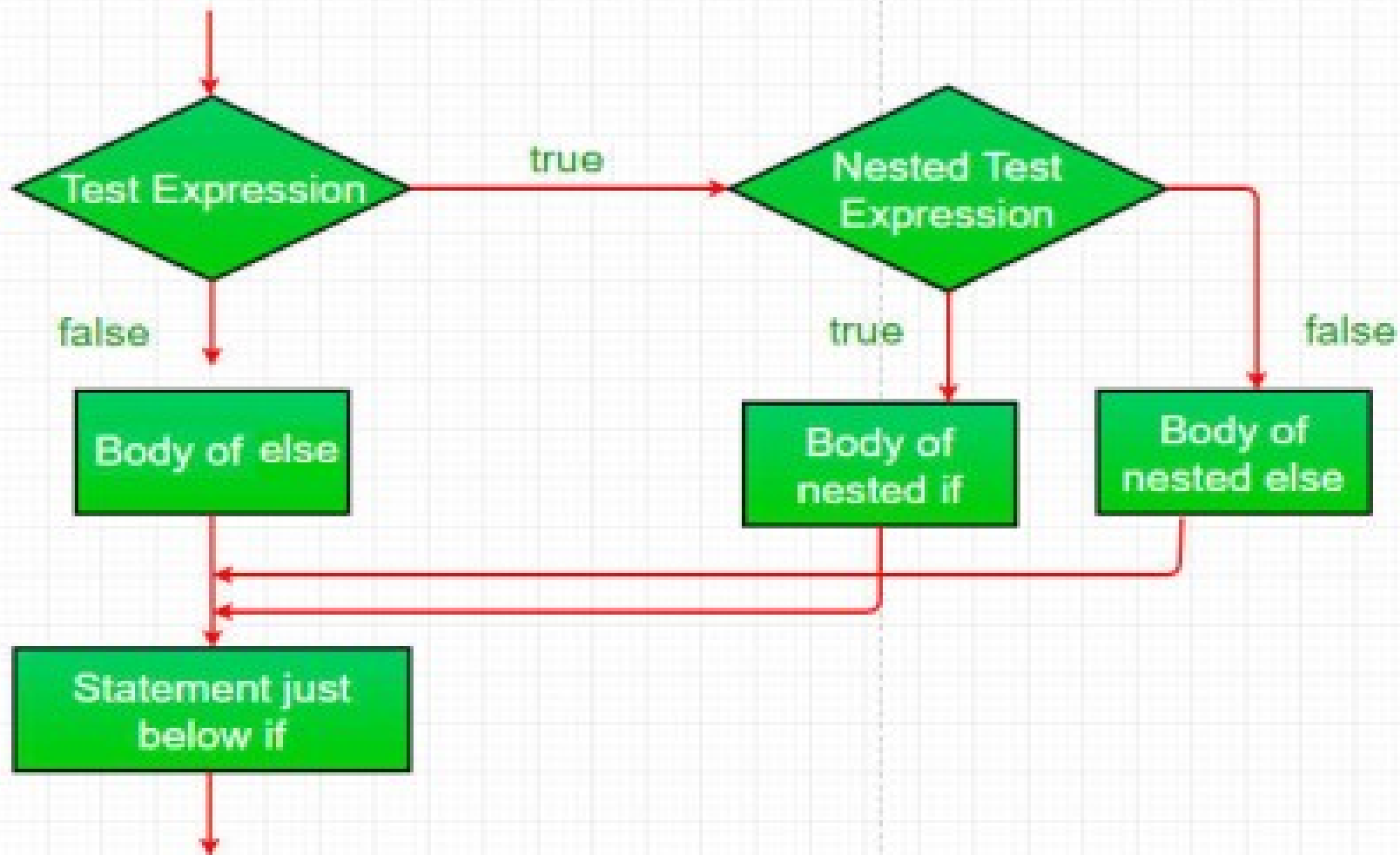


Nested if...else Statement:

- It is always legal to **nest if-else statements** which means you can use one if or else if statement inside another if or else if statement

- Syntax

```
if(Boolean_expression 1){  
    //Executes when the Boolean expression 1 is true  
    if(Boolean_expression 2){  
        //Executes when the Boolean expression 2 is true  
    }  
}
```





Nested If statement example

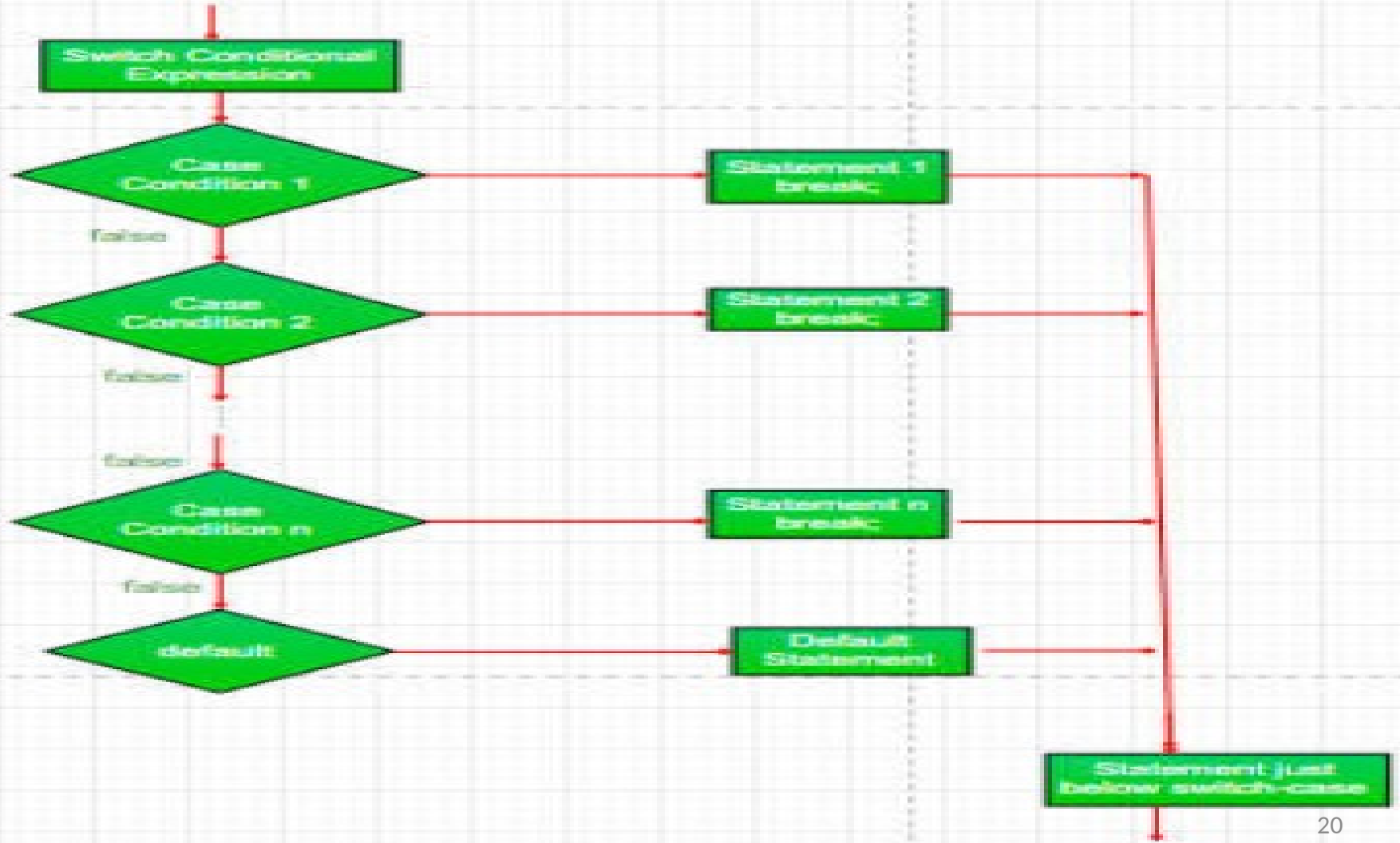
```
public class NestedIfTest {  
  
    public static void main(String args[]) {  
        int x = 30;  
        int y = 10;  
  
        if( x == 30 ) {  
            if( y == 10 ) {  
                System.out.print("X = 30 and Y = 10");  
            }  
        }  
    }  
}
```

- A *switch* statement allows a variable to be tested for equality against a list of values.
- Each value is called a case, and the variable being switched on is checked for each case.
- Syntax:

```
switch(expression){  
    case value :  
        //Statements  
        break; //optional  
    case value :  
        //Statements  
        break; //optional  
    //You can have any number of case statements.  
    default : //Optional  
        //Statements  
}
```



- The following rules apply to a switch statement:
 - The variable used in a switch statement can only be a **byte, short, int, or char**.
 - You can have **any number of case statements within a switch**. Each case is followed by the value to be compared to and a colon.
 - The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
 - When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
 - When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.



```

import java.util.Scanner;
public class SwitchDemo {

    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);
        System.out.println("Please Enter Number 1-12:");

        int month = input.nextInt();

        String monthString;
        switch (month){
            case 1: monthString = "January"; break;
            case 2: monthString = "February"; break;
            case 3: monthString = "March"; break;
            case 4: monthString = "April"; break;
            case 5: monthString = "May"; break;
            case 6: monthString = "June"; break;
            case 7: monthString = "July"; break;
            case 8: monthString = "August"; break;
            case 9: monthString = "September"; break;
            case 10: monthString = "October"; break;
            case 11: monthString = "November"; break;
            case 12: monthString = "December"; break;
            default: monthString = "Invalid month"; break;
        }
        System.out.println(monthString);
    }
}

```

Switch Statement Demo

- It takes an integer user input
- It returns the string equivalent of the given numeric month



- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A *switch* statement can have an **optional default case**, which **must appear at the end of the switch**.
- The default case can be used for performing a task when none of the cases is true.
- **No break is needed in the default case**



Loop Control

- There may be a situation when we need to execute a block of code several number of times, and is often referred to as a loop.
- Java has very flexible three looping mechanisms. You can use one of the following three loops:
 1. while Loop
 2. do...while Loop
 3. for Loop



The while Loop

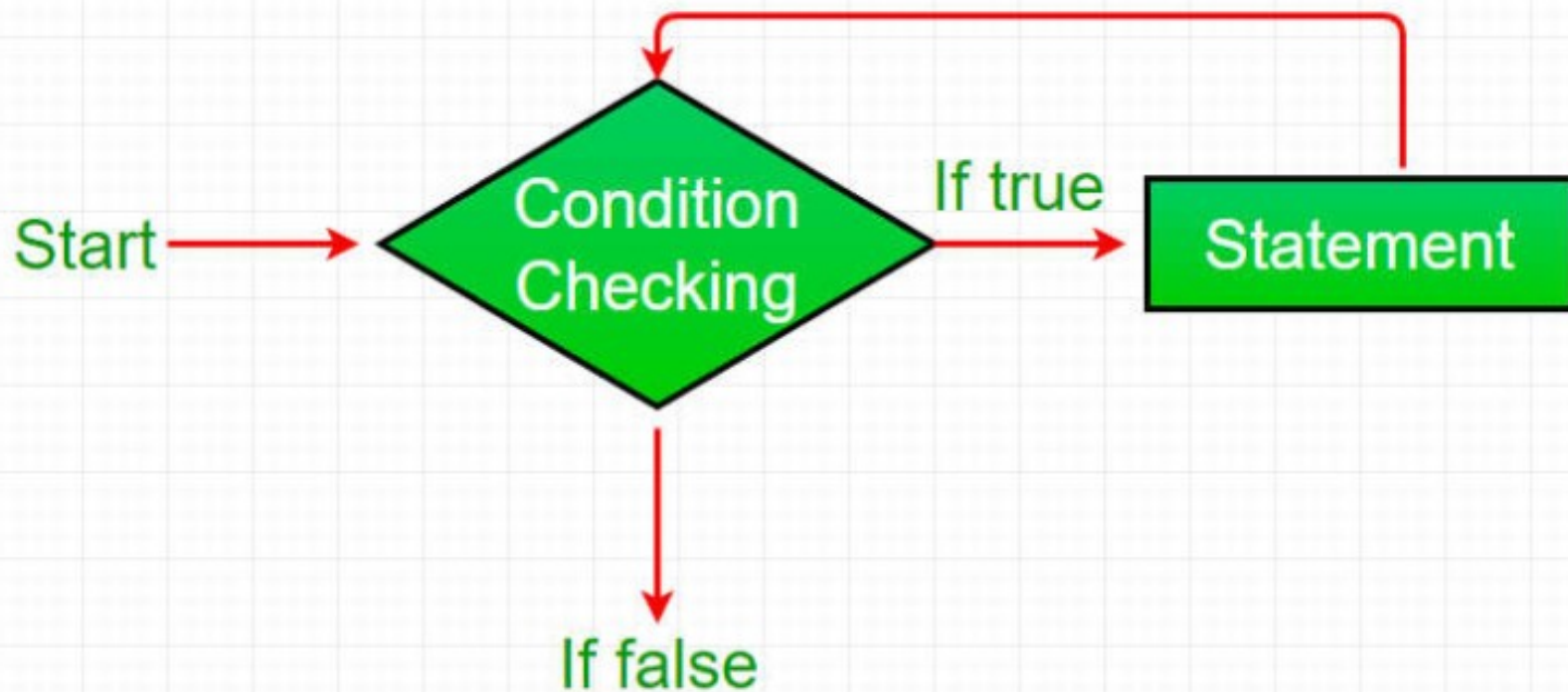
- A while loop is a control structure that allows you to repeat a task a certain number of times.

- Syntax

```
while(Boolean_expression)
{
    //Statements
}
```

- When executing, if the *boolean_expression* result is true, then the actions inside the loop will be executed.
- This will continue as long as the expression result is true.

Flowchart:





- key point of the *while* loop is that the loop might not ever run.
- When the expression is tested and the result is false, the **loop body will be skipped** and the first statement after the while loop will be executed.



While loop example

```
public class WhileLoop {  
  
    public static void main(String args[]) {  
        int x = 10;  
  
        while( x < 20 ) {  
            System.out.print("value of x : " + x );  
            x++;  
            System.out.print("\n");  
        }  
    }  
}
```



The do...while Loop

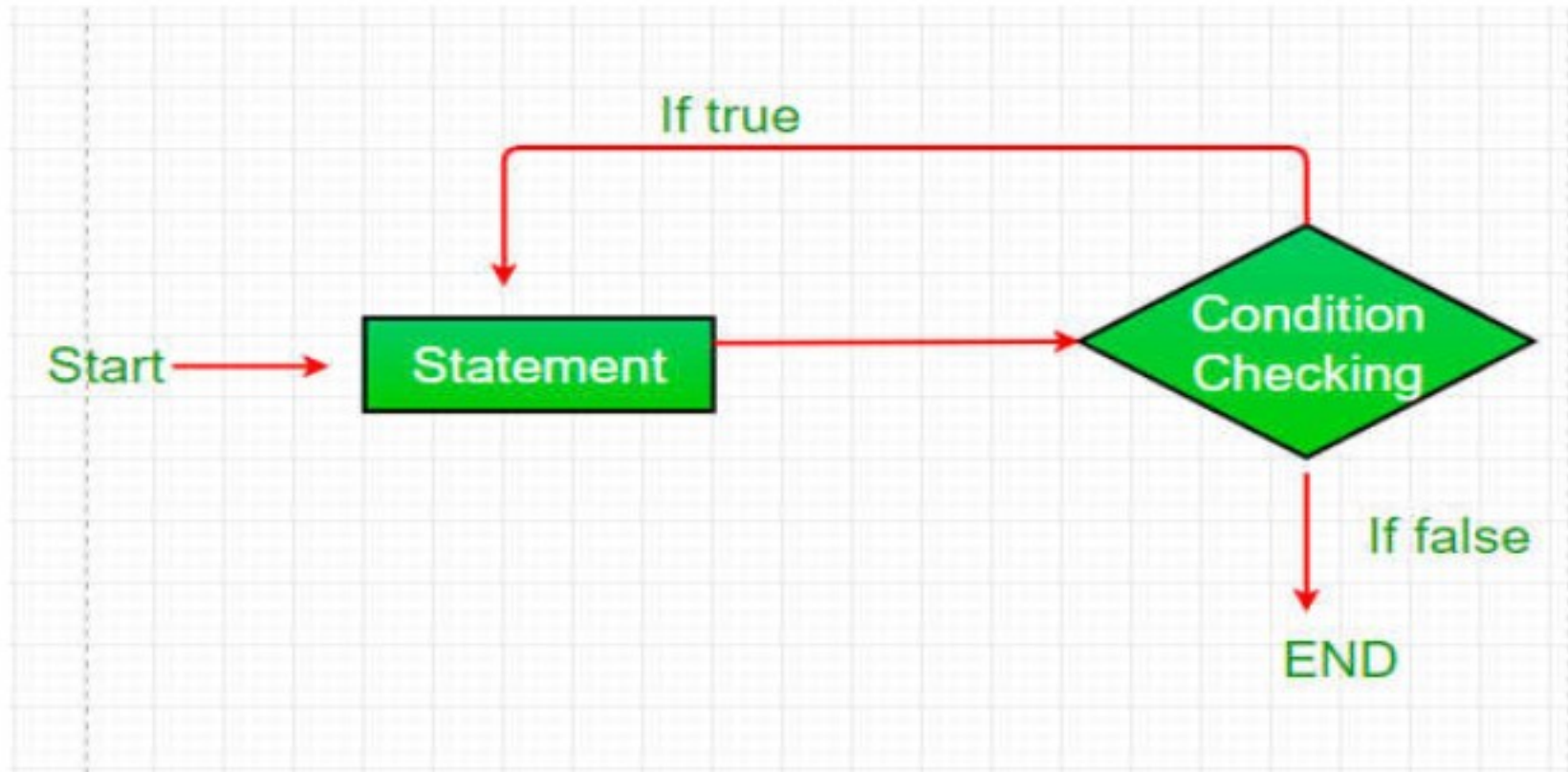
- A **do...while** loop is similar to a **while** loop, except that a **do...while** loop is guaranteed to execute at least one time.

- Syntax:

```
do
{
    //Statements
}while(Boolean_expression);
```

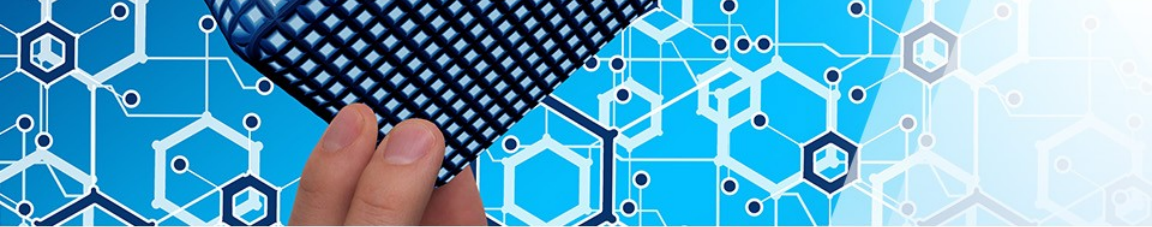
- Notice that the **Boolean expression appears at the end of the loop**, so the statements in the loop execute once before the Boolean is tested.

Flowchart:





```
//DoWhileLoop.java  
public class DoWhileLoop {  
  
    public static void main(String args[]) {  
        int x = 10;  
  
        do {  
            System.out.print("value of x : " + x );  
            x++;  
            System.out.print("\n");  
        } while ( x < 20 );  
    }  
}
```

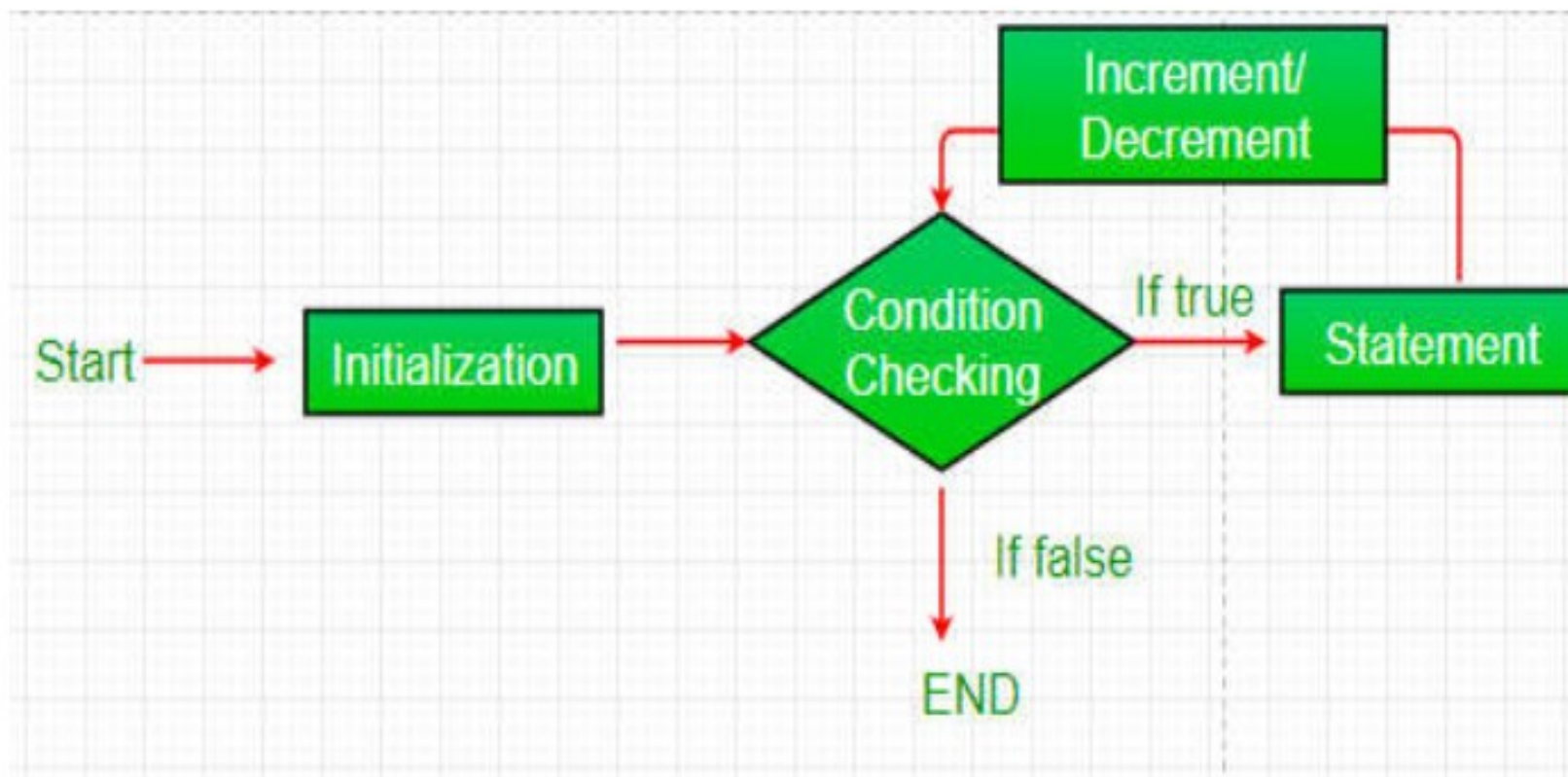


The for Loop

- A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.
- A for loop is useful when you know how many times a task is to be repeated.
- Syntax

```
for(initialization; Boolean_expression; update)
{
    //Statements
}
```

Flowchart:





- **flow of control in a for loop**

- ✓ The initialization step is executed first, and only once.
- ✓ Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.
- ✓ After the body of the for loop executes, the flow of control jumps back up to the update statement.
- ✓ This statement allows you to update any loop control variables.
- ✓ The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.



For loop example

```
public class ForLoop {  
    public static void main(String args[]) {  
        for(int x = 10; x < 20; x = x+1) {  
            System.out.print("value of x : " + x );  
            System.out.print("\n");  
        }  
    }  
}
```



Enhanced for loop in Java

- As of Java 5, the enhanced for loop was introduced. This is mainly used for Arrays

- Syntax

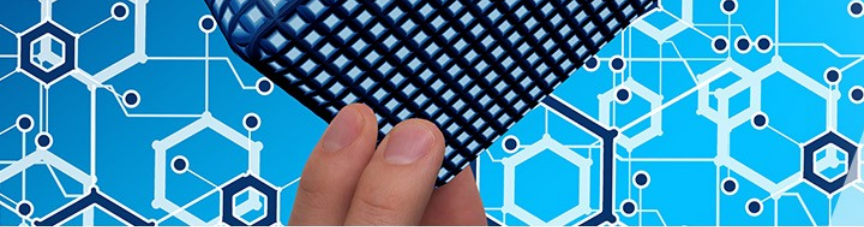
```
for(declaration : expression)
{
    //Statements
}
```

- **Declaration:** The newly declared block variable, which is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
- **Expression:** This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an



Enhanced For Loop example

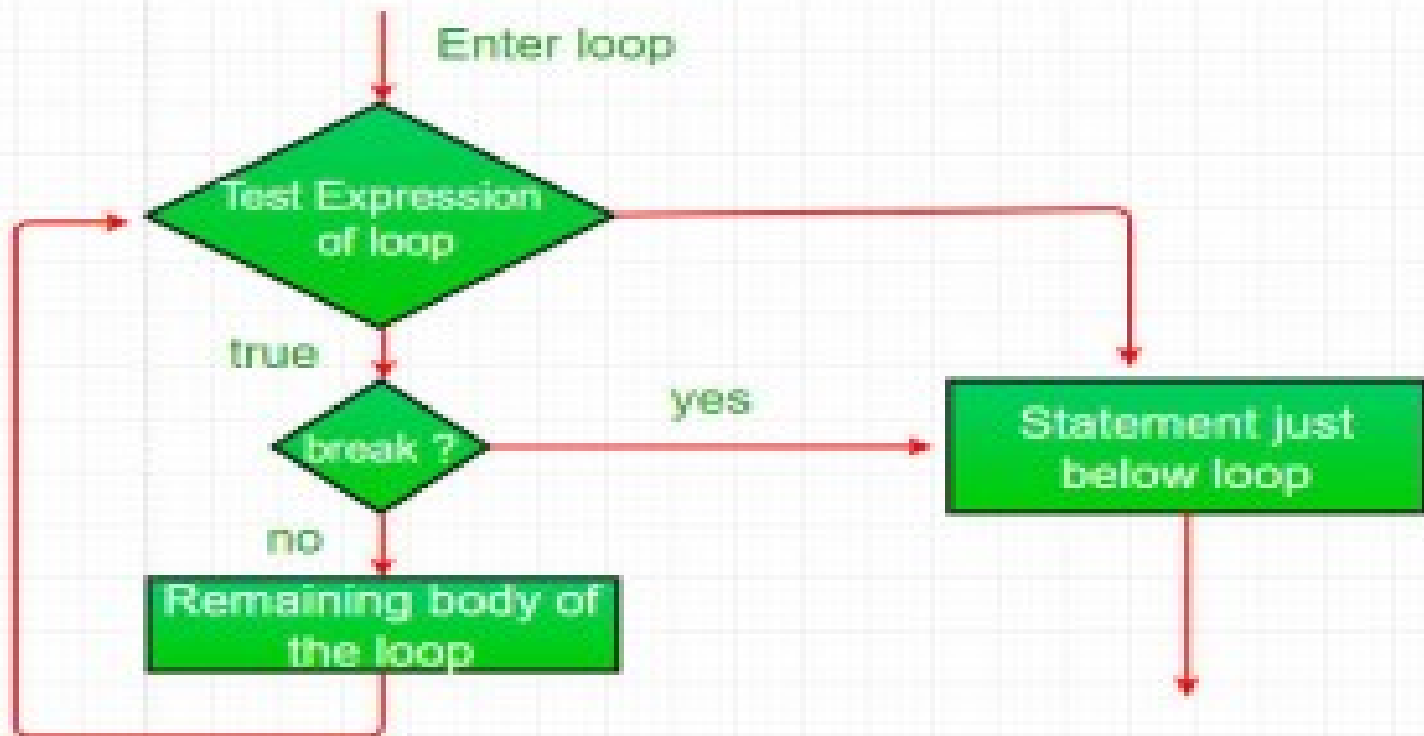
```
public class EnhancedFor {  
  
    public static void main(String args[]){  
        int [] numbers = {10, 20, 30, 40, 50};  
  
        for(int x : numbers ){  
            System.out.print( x );  
            System.out.print(",");  
        }  
        System.out.print("\n");  
        String [] names ={"James", "Larry", "Tom", "Lacy"};  
        for( String name : names ) {  
            System.out.print( name );  
            System.out.print(",");  
        }  
    }  
}
```



The break Keyword

- The ***break*** keyword is used to stop the entire loop.
- The **break** keyword must be used inside any **loop** or a **switch** statement.
- The break keyword will stop the execution of the innermost loop and start executing the next line of code.
- Syntax:

```
break;
```





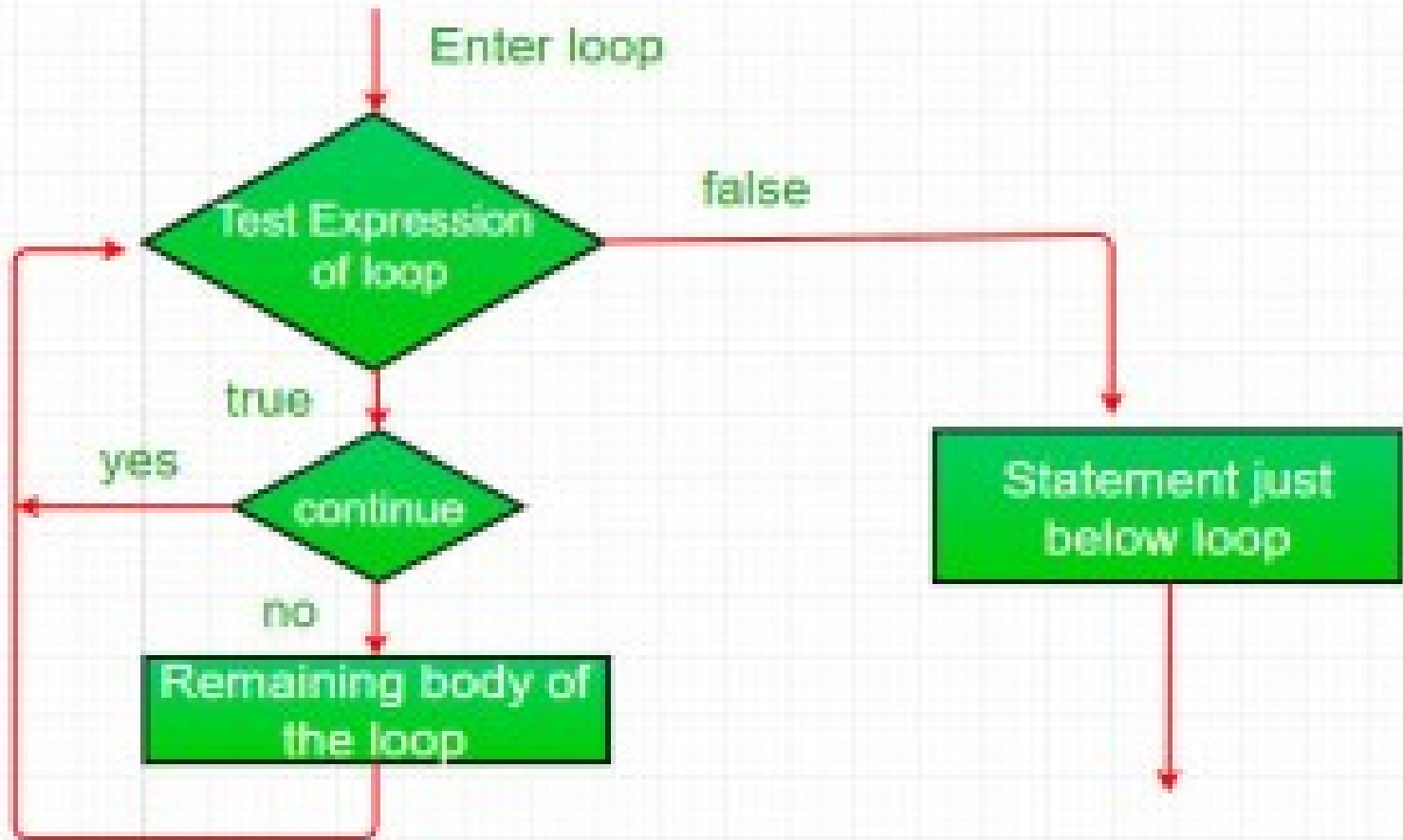
Break Test

```
public class BreakTest {  
  
    public static void main(String args[]) {  
        int [] numbers = {10, 20, 30, 40, 50};  
  
        for(int x : numbers ) {  
            if( x == 30 ) {  
                break;  
            }  
            System.out.print( x );  
            System.out.print("\n");  
        }  
    }  
}
```



The `continue` Keyword

- The *continue* keyword can be used in any of the loop control structures. It causes the loop to immediately **jump to the next iteration** of the loop.
- ✓ In a **for loop**, the `continue` keyword causes flow of control to immediately **jump to the update statement**.
- ✓ In a `while` loop, the `continue` keyword causes flow of control to immediately **jump to the Boolean expression**.





Continue

```
public class ContinueTest {  
  
    public static void main(String args[]) {  
        int [] numbers = {10, 20, 30, 40, 50};  
  
        for(int x : numbers ) {  
            if( x == 30 ) {  
                continue;  
            }  
            System.out.print( x );  
            System.out.print("\n");  
        }  
    }  
}
```



Java array

Array and Array List

Java Array

- An array is a **collection of similar type of elements** that have a contiguous memory location.
- Java array is an object which contains elements of a similar data type . It is a data structure where we store similar elements .
- We can store only a fixed set of elements in a Java array.
- Array in java is index-base





- **Advantages**

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** we can get any data located at an index position.

- **Disadvantage**

- **Size Limit:** we can store a fixed size of elements in an array. It doesn't grow its size at runtime.



Types of array in Java

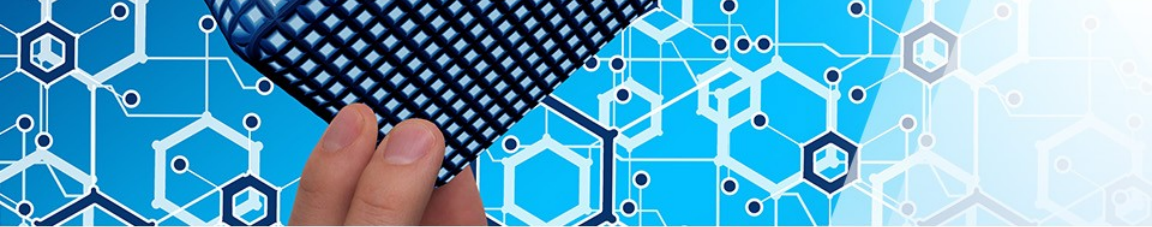
- One Dimensional Array
- Multidimensional Array
- One Dimensional Array in Java

Syntax to Declare an Array in Java

```
dataType[] arr; (or)  
dataType []arr; (or)  
dataType arr[];
```

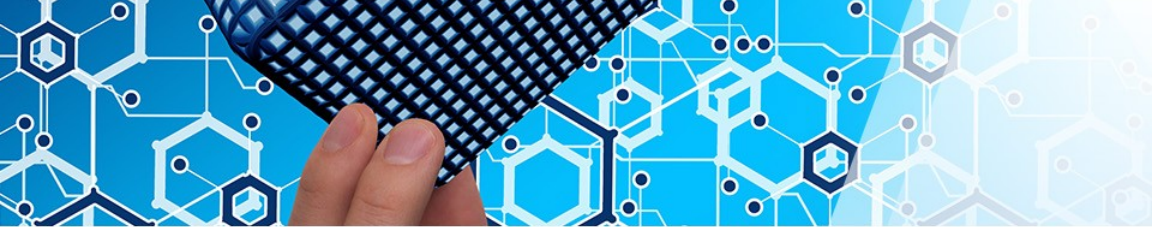
Instantiation of an Array in Java

```
arrayRefVar=new datatype[size];
```



Example

```
class ArrayTest {  
    public static void main(String args[]) {  
  
        int a[]=new int[5]; //declaration and intantiation  
        a[0]=10;  
        a[1]=20;  
        a[2]=70;  
        a[3]=44;  
        a[4]=64;  
  
        for(int i=0;i<a.length;i++){  
            System.out.println(a[i]);  
        }  
    }  
}
```



Example 2

```
class ArrayTest1 {  
    public static void main(String args[]) {  
  
        //declaration, instantiation and initialization  
        int a[]={10,20,30,40,55};  
  
        for(int i=0;i<a.length;i++) {  
            System.out.println(a[i]);  
        }  
    }  
}
```



Multidimensional Array

Syntax to Declare Multidimensional Array in Java

```
dataType[][] arrayRefVar; (or)  
dataType [][]arrayRefVar; (or)  
dataType arrayRefVar[][]; (or)  
dataType []arrayRefVar[];
```

Example to Instantiate Multidimensional Array in Java

```
int[][] arr=new int[3][3]; //3 row and 3 column
```



Multidimensional Array

```
class ArrayTest2 {  
    public static void main(String args[]){  
  
        //declaring and initializing 2D array  
        int a[][]={{10,20,30},{40,55,45},{3,5,6}};  
  
        //Printing 2D array  
  
        for(int i=0;i<3;i++){  
            for(int j=0;j<3;j++){  
                System.out.println(a[i][j]+"");  
            }  
            System.out.println();  
        }  
    }  
}
```

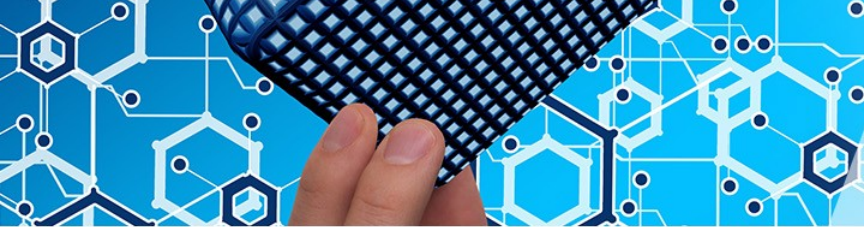


ArrayList

- **Dynamic sized arrays** in Java that implement List interface.
- It is part of **collection framework** in Java.
- ArrayList has a set of Methods to access elements and modify them.

```
ArrayList<Type> arrL = new ArrayList<Type>();
```

Here Type is the type of elements in ArrayList to be created



Package statement

- **import** java.util.List; // import just the List interface
- **import** java.util.ArrayList; // import just the ArrayList class
- Another option is to import everything at the same level in a package using import packageName.*.
- **import** java.util.*;
 - // import everything including List and ArrayList



common methods of class `ArrayList<T>`.

Method	Description
<code>add</code>	Adds an element to the end of the <code>ArrayList</code> .
<code>clear</code>	Removes all the elements from the <code>ArrayList</code> .
<code>contains</code>	Returns <code>true</code> if the <code>ArrayList</code> contains the specified element; otherwise, returns <code>false</code> .
<code>get</code>	Returns the element at the specified index.
<code>indexOf</code>	Returns the index of the first occurrence of the specified element in the <code>ArrayList</code> .
<code>remove</code>	Overloaded. Removes the first occurrence of the specified value or the element at the specified index.
<code>size</code>	Returns the number of elements stored in the <code>ArrayList</code> .
<code>trimToSize</code>	Trims the capacity of the <code>ArrayList</code> to current number of elements.



add() method of ArrayList

- **add(E e):** The **add(E e) method** is used to add the specified element **e** in the **ArrayList**.
- **add(int index, E e):** The **add(int index, E e) method** is used to add the specified element at a specific position in the **ArrayList**.
- **addAll(Collection c):** The **addAll(Collection c)** is used to add the collection of elements to the end of the **ArrayList**.
- **addAll(int index, Collection c):** The **addAll(int index, Collection c)** is used to add the collection of elements at specified position in the **ArrayList**.

```
import java.util.ArrayList;
import java.util.Arrays;
class ArrayListTest {
    public static void main(String args[]){
        // Normal Array
        int [] arr=new int[2];
        arr[0]=1;
        arr[1]=2;
        System.out.println(arr[0]);

        // Array List
        //create an arraylist with initial capacity 2
        ArrayList<Integer> arrL=new ArrayList<Integer>(2);

        //Add elems to arraylist
        arrL.add(1);
        arrL.add(2);

        System.out.println(arrL.get(0));

    }
}
```

ArrayListTest



```
import java.util.ArrayList;
import java.util.Arrays;
class ArrayListTest1 {
    public static void main(String args[]){
        // Normal Array
        //Need to Specify the size for array
        int [] arr=new int[3];
        arr[0]=1;
        arr[1]=2;
        arr[2]=3;

        // Array List
        //Need not to specify size
        ArrayList<Integer> arrL=new ArrayList<Integer>();

        //Add elems to arraylist
        arrL.add(1);
        arrL.add(2);
        arrL.add(3);
        arrL.add(4);
        // we can add morre elements to arrL
        System.out.println(arrL);
        System.out.println(Arrays.toString(arr));
    }
}
```



End of Chapter Three



CHAPTER FOUR