Java's program control statements can be put into the following categories:

1. Selection
2. Iteration
3. Jump

- **Selection statement** allows your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
- **Iteration statements** enable program execution to repeat one or more statements. (Iteration statements mean all loops).
- **Jump statements** allow your program to execute in a nonlinear fashion.

(1) Java's selection statements:

Java supports two selection statements: **if** and **switch. These** statements allow you to control the flow of your program's execution based upon conditions known only during at run time.

❖ If statement:

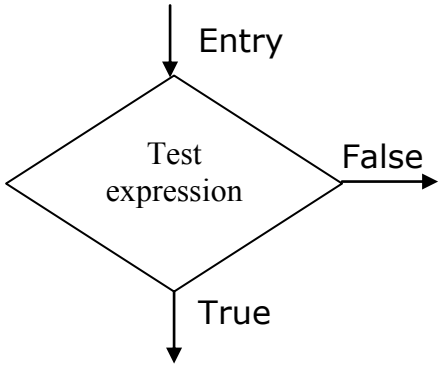
If statement is Java's conditional branch statement. It is powerful decision making statement and is used to control the flow of execution of statements. It is basically two-way statement and is used in conjunction with an expression.

Here the general form of the **if** statement:

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

Here each statement may be a single statement or a compound statement enclosed in curly braces (that is a block). The **condition** is any expression that returns **boolean** value. The **else** clause is optional.

If works like this: If the condition is **true**, then **statement1** is executed. Otherwise **statement2** (if it exists) is executed. This point of program has two paths to follow, one for the **true** condition and the other for the **false** condition as shown in **following figure**.



Two-Way Branching

Consider the **following program:**

```
int a,b;
// …
if(a<b)
        a=0;
else
        b=0;
```

Here, if **a** is less than **b**, then **a** is set to zero. Otherwise, **b** is set to zero. In no case are they both set to zero.

Most often, the expression used to control the **if** will involve the relational operators. However, this is not technically necessary. It is possible to control the **if** using a single **boolean** variable, as shown in this code fragment.

```
boolean dataavailable;
// …
        if (dataavailable)
                processdata();
        else
                waitformoredata();
```

Remember, only one statement can appear directly after the **if** or the **else**. If you want to include more statements, you will need to create a block, as in this fragment.

```
int bytesavailable;
        // …
        if (byteavailable>0)
        {
                processdata();
                bytesavailable-=n;
        }
        else
                waitformoredata();
```

Here, both statements within the if block will execute if **bytesavailable** is greater than zero.

- **Nested ifs:** A nested **if** is an **if statement** that is the target of another **if or else**. **Nested ifs** are very common in programming. When you nest ifs, the main thing to remember is that an **else statement** always refers to the nearest **if statement** that is within the same block as the else and that is not already associated with an else. Here is an example.

```
        If(i==10)
        {
              if(j<20)
                    a=b;
              if(k>100) c=d;
              else
                    a=c;
        }
else    a=d;
```

As the comments indicate, the final **else** is not associated with **if(j<20)**, because it is not in the same block (even though it is the nearest **if** without an **else**). Rather, the final **else** is associated with **if(i==10)**. The inner **else** refers to **if(k>100)**, because it is the closest **if** within the same block.

- **The if-else-if ladder:**

A common programming construct that is based upon a sequence of nested **ifs** is the **if-else-if** ladder. It looks like this:

```
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; that is, if all other conditional tests fail, then the last **else** statement is performed.

If there is no final **else** and all other conditions are **false**, then no action will take place.

Consider the following program that uses an **if-else-if** ladder to determine which season a particular month is in.

```
// Demonstrate if-else-if statements.
class Ifelse {
  public static void main(String args[]) {
    int month = 4; // April
    String season;

    if(month == 12 || month == 1 || month == 2)
      season = "Winter";
    else if(month == 3 || month == 4 || month == 5)
      season = "Spring";
    else if(month == 6 || month == 7 || month == 8)
      season = "Summer";
    else if(month == 9 || month == 10 || month == 11)
      season = "Autumn";
    else
      season = "Bogus Month";

    System.out.println("April is in the " + season + ".");
  }
}
```

(2) Switch Statement:

The switch statement is Java's multiway branch statement. It 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 if-else-if statements. Here is the general form of a switch statement.

```
switch(expression)
{
        case value 1:
                //statement sequence
                break;
        case value 2:
                //statement sequence
                break;
        .
        .
        .
        case valueN:
                //statement sequence
                break;

        default:
                //default statement sequence
}
```

The expression must be of type **byte, short, int or char**; each of the values specified in the **case** statements must be of a type compatible with the expression.

Each **case** value must be a unique literal (that is, it must be a constant, not a variable). Duplicate **case** values are not allowed.

The switch statement works like this: The value of the expression is compared with each of the literal values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed.

If none of the constants matches the value of the expression, then the default statement is executed. However, the default statement is an 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. When a **break** statement is encountered, execution branches to the first line of code that follows the entire switch statement.

Here is a simple example that uses a switch statement.

```
// A simple example of the switch.
class SampleSwitch {
  public static void main(String args[]) {
    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.");
      }
  }
}
```

Output:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

- The **break** statement is an optional. If you omit the break, execution will continue on into the next case. It is sometimes desirable to have multiple **cases** without break statements between them. For example, consider the following program.

```
// In a switch, break statements are optional.
class MissingBreak {
  public static void main(String args[]) {
    for(int i=0; i<12; i++)
      switch(i) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
          System.out.println("i is less than 5");
          break;
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
          System.out.println("i is less than 10");
          break;
        default:
          System.out.println("i is 10 or more.");
      }
  }
}
```

- The previous example is, of course unnatural for the sake of illustration, omitting the **break** statement has many practical applications in real programs. Consider the following rewrite of the season example shown earlier.

// An improved version of the season program.

```
class Switch {
   public static void main(String args[]) {
      int month = 4;
      String season;
      switch (month) {
        case 12:
        case 1:
        case 2:
          season = "Winter";
          break;
        case 3:
        case 4:
        case 5:
          season = "Spring";
          break;
```

```
        case 6:
        case 7:
        case 8:
          season = "Summer";
          break;
        case 9:
        case 10:
        case 11:
          season = "Autumn";
          break;
        default:
          season = "Bogus Month";
      }
      System.out.println("April is in the " + season + ".");
   }
}
```

❖ **Nested Switch Statements:**

You can use a **switch** as part of the statement sequence of an outer **switch**. This is called a nested **switch**. For example consider the following fragment is perfectly valid:

```
switch(count)
{
      case 1:
            switch(target)
            {
                  case 0:
                        System.out.println("target is zero");
                        break;
                  case 1:
                        System.out.println("target is one");
                        Break;
            }
      break;
      case 2:     //      …..
}
```

Here, the **case 1:** statement in the **inner switch** does not conflict with the **case 1:** statement in the outer switch. The **count** variable is only compared with the list of cases at the outer level. If **count is 1**, then target is compared with the inner list cases.

There are three important features of the **switch** statement to note:

1. The **switch** differs from the **if** in that **switch** can only test equality, whereas **if** can evaluate any type of **Boolean expression**. That is, the

switch looks only for a match between the value of the expression and one of its **case** constants.

2. No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement enclosed by an outer **switch** can have **case** constants in common.

3. A **switch** statement is usually more efficient than a set of nested **ifs**.

❖ **How Java compiler compiles switch statement?**

Or

❖ **When switch statement is better than if-else ?**

When Java compiler compiles a **switch** statement, the Java compiler will inspect each of the **case constants** and create a **"jump table"** that it will use for selecting the path of execution depending on the value of expression.

Therefore, if you need to select among a large group of values, a switch statement will run much faster than the equivalent logic coded using a sequence of **if-elses**. The compiler can do this because it knows that the **case** constants are all the same type and simply must be compared for equality with the **switch** expression. The compiler has no such knowledge of a long list of **if** expressions.

(3) Iteration Statements:

Java's iteration statements are **for**, **while** and **do-while.** These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met.

❖ **While Loop:** The while loop is Java's most fundamental looping statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
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.

Here is a while loop that counts down from 10, printing exactly ten lines of value.

```
// Demonstrate the while loop.
class While {
  public static void main(String args[]) {
    int n = 10;

    while(n > 0) {
      System.out.println("tick " + n);
      n--;
    }
  }
}
```

Since the **while** loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with.

The body of the while (or any other of Java's loops) can be empty. This is because a null statement (one that consists only of a semicolon) is syntactically valid in Java. For example, consider the following program:

```
class nobody
{
        public static void main(String arg[])
        {
                int i,j;
                i=100;
                j=200;

                //find midpoint between i and j
                while(++I < --j) ;        //no body in this loop

                System.out.println("Midpoint is" +i);
        }
}
```

Output: Midpoint is 150

❖ do-while loop:

We know that, if the conditional expression controlling a **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 **while loop** at least once, even if the conditional expression is **false** to begin with.

In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that:

The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is:

```
do
{
        //body of the 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. As with all of Java's loops, condition must be a Boolean expression.

Here is the reworked version of the previous program that demonstrates the **do-while** loop.

```
class dowhile
{
        public static void main(String arg[])
        {
                int n=10;
                        do
                        {
                                System.out.println("n=" +n);
                                n--;
                        }while(n>0);
        }
}
```

The above loop can be written more efficiently as follows:

```
do
{
        System.out.println("n=" +n);
}while(--n>0);
```

The **do-while** loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once.

Consider the following program which implements a very simple help system for Java's selection and iteration statements:

// Using a do-while to process a menu selection -- a simple help // system.

```
class Menu {
  public static void main(String args[])
    throws java.io.IOException {
    char choice;

    do {
      System.out.println("Help on:");
      System.out.println("  1. if");
      System.out.println("  2. switch");
      System.out.println("  3. while");
      System.out.println("  4. do-while");
      System.out.println("  5. for\n");
      System.out.println("Choose one:");
      choice = (char) System.in.read();
    } while( choice < '1' || choice > '5');

    System.out.println("\n");
     switch(choice) {
      case '1':
        System.out.println("The if:\n");
        System.out.println("if(condition) statement;");
        System.out.println("else statement;");
        break;
      case '2':
        System.out.println("The switch:\n");
        System.out.println("switch(expression) {");
        System.out.println("  case constant:");
        System.out.println("    statement sequence");
        System.out.println("  break;");
        System.out.println("  // ...");
        System.out.println("}");
        break;
      case '3':
        System.out.println("The while:\n");
        System.out.println("while(condition) statement;");
        break;
      case '4':
```

```
        System.out.println("The do-while:\n");
        System.out.println("do {");
        System.out.println("  statement;");
        System.out.println("} while (condition);");
        break;
      case '5':
        System.out.println("The for:\n");
        System.out.print("for(init; condition; iteration)");
        System.out.println(" statement;");
        break;
    }
  }
}
```

Notice that characters are read from the keyboard by calling **System.in.read().** This is one of Java's console input functions. **System.in.read()** is used here to obtain the user's choice. It reads characters from standard input (returned as integers, which is why the return value was cast to **char**).

By default, standard input is line buffered, so you must press **ENTER** before any characters that you type will be sent your program.

Because **System.in.read()** is being used, the program must specify the throws java.io.IOException clause. This line is necessary to handle input errors. It is a part of Java's exception handling features.

❖ **For loop:**

The general form of the **for** statement:

```
for(initialization;condition;iteration)
{
        //body of the for loop
}
```

If only one statement is being repeated, there is no need for the curly braces.

The for loop operates as follows:

1. 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**, which acts as a counter that controls the loop. So, initialization expression is only executed once.

2. 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.

3. Next the **iteration** portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. 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.

// Demonstrate the for loop.

```
class ForTick {
        public static void main(String args[]) {
          int n;

          for(n=10; n>0; n--)
          System.out.println("tick " + n);
        }
}
```

When the loop control variable will not be needed elsewhere, most Java programmers declare it inside the **for**. **For example**, here is a simple program that tests for prime numbers.

// Test for primes.

```
class findprime {
  public static void main(String args[]) {
    int num;
    boolean isPrime = true;

    num = 14;
    for(int i=2; i <= num/2; i++) {
      if((num % i) == 0) {
        isPrime = false;
        break;
      }
    }
    if(isPrime) System.out.println("Prime");
    else System.out.println("Not Prime");
  }
}
```

❖ Using the Comma:

There will be times when you will want to include more than one statement in the initialization and iteration portion of the for loop. Java allow two or more variables to control a **for** loop, Java permits you to include multiple statements in both the initialization and iteration portions of the **for**. Each statement is separated from the next by the comma.

// Using the comma.

```
class Comma {
  public static void main(String args[]) {
    int a, b;

    for(a=1, b=4; a<b; a++, b--) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
    }
  }
}
```

❖ Some for loop variations:

The **for** loop supports a number of variations that increase its power and applicability. The reason it is so flexible is that its three parts, the initialization, the conditional test, and the iteration, do not need to be used for only those purposes.

One of the most common variations involves the conditional expression. Specifically, this expression does not need to test the loop control variable against some target value. In fact, the condition controlling the **for** can be any valid **Boolean** expression.

For example,

```
boolean done=false;

        for(int i=1;!done;i++)
        {
                //…..
                if(interrupted())
                done=true;
        }
```

Here is another interesting **for** loop variation. Either the initialization or the iteration expression or both may be absent as in the next program.

```
// Parts of the for loop can be empty.
class ForVar {
  public static void main(String args[]) {
    int i;
    boolean done = false;

    i = 0;
    for( ; !done; ) {
      System.out.println("i is " + i);
      if(i == 10) done = true;
      i++;
    }
  }
}
```

Here is one more **for** loop variation. You can intentionally create an infinite loop (a loop that never terminates) if you leave all three parts of the **for** empty. For example,

```
for(;;)
{
        //…..
}
```

This loop will run forever, because there is no condition under which it will terminate. Although there are some programs, such as operating system command processor that require a **nonfinite loop**, most "infinite loops" are really just loops with special termination requirements.

❖ **Nested loops:**

```
// Loops may be nested.
class Nested {
  public static void main(String args[]) {
    int i, j;

    for(i=0; i<10; i++) {
      for(j=i; j<10; j++)
        System.out.print(".");
      System.out.println();
    }
  }
}
```

(4) Jump statements:

Java supports three **jump** statements: **break, continue, and return**. These statements transfer control to another part of your program. Each is examined here.

❖ Using break:

In Java, the **break** statement has three uses.

1. It terminates a statement sequence in a **switch** statement.
2. It can be used to **exit loop**.
3. It can be used as a "civilized" form of **goto**.

❖ Using break to Exit a Loop:

By using break, you 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.

```
// Using break to exit a loop.
class breakloop {
  public static void main(String args[]) {
    for(int i=0; i<100; i++) {
      if(i == 10) break; // terminate loop if i is 10
      System.out.println("i: " + i);
    }
    System.out.println("Loop complete.");
  }
}
```

The **break statement** can be used with any of Java's loops, including intentionally infinite loops. **For example**, here is the preceding program coded by use of a **while** loop.

```
//  Using break to exit a while loop.

class breakloop2 {
  public static void main(String args[]) {
    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.");
  }
}
```

When used inside a set of nested loops, the break statement will only break out of the inner

```
// Using break with nested loops.
class breakloop3 {
  public static void main(String args[]) {
    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.");
  } }
```

❖ Using break as a Form of Goto:

The **break statement** can also be used by itself to provide a "elegant" form of the **goto** statement.

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 sometimes **goto** can be useful when you are exiting from a deeply nested set of loops. To handle such situations, Java defines an expanded form of the break statement. By using this form of **break,** you can break out of one or more block of code. These blocks need not be part of a loop or a **switch.** They can be any block.

The general form of the **labeled break** statement is shown here.

```
break label;
```

Here **label=**it is the name of label that identifies block of code.

When this form of break executes, control is transferred out of the named block of code. The labeled block of code must enclose the break statement, but it does not need to be the immediately enclosing block.

This means that you can use a labeled **break** statement to exit from set of nested blocks. But you cannot use break to transfer control to a block of code that does not enclose **break** statement.

For example, the following program shows three nested blocks, each with its own label. The break statement causes execution to jump forward, past the end of the block labeled secod, skipping the two println() statements.

// Using break as a civilized form of goto.

```
class breakblock {
  public static void main(String args[]) {
    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 won't execute");
        }
        System.out.println("This won't execute");
      }
      System.out.println("This is after second block.");
    }
  }
}
```

One of the most common uses for a **labeled break** statement is to exit from nested loops. For example, consider the following program, the outer loop executes only once.

// Using break to exit from nested loops

```
class breakloop4 {
  public static void main(String args[]) {
    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.print(j + " ");
      }
      System.out.println("This will not print");
    }
    System.out.println("Loops complete.");
  }
}
```

keep in mind that you can't break to any label which is not defined for an enclosing block. **For example** consider the following program that will not compile.

```
// This program contains an error.

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

    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 + " ");
    }}}
```

❖ **Using continue statement as jump:**

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop, but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end.
Here is an example program that uses **continue** to reason two numbers to be printed on each line:

```
// Demonstrate continue.
class Continue {
  public static void main(String args[]) {
    for(int i=0; i<10; i++) {
      System.out.print(i + " ");
      if (i%2 == 0) continue;System.out.println("");}}}
```

As with the **break** statement, **continue** may specify a label to describe which enclosing loop to continue.

Here is an **example program** that uses continue to print a triangular multiplication table for 0 through 9.

```
// Using continue with a label.

class continuelabel {
  public static void main(String args[]) {
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.println();
}}
```

❖ **Return statement:**

The last control statement is **return.** The return statement is used to explicitly return from a method. That is it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement.

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.

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

```
// Demonstrate return.
class Return {
  public static void main(String args[]) {
    boolean t = true;

    System.out.println("Before the return.");
    if(t) return; // return to caller
    System.out.println("This won't execute.");
  }}
```

In the **preceding program**, the if(t) statement is necessary. Without it the Java compiler would flag an "unreachable code" error, because the compiler would know that the last **println()** statement would never be executed.