Control Statements

- A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program.
- Java's program control statements can be put into the following categories: selection, iteration, and jump.
- Selection statements allow 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 (that is, iteration statements form loops).
- Jump statements allow your program to execute in a nonlinear fashion. All of Java's control statements are
  examined here.

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 run time.

If

- The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths.
- Here is 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 a boolean value. The
  else clause is optional.
- 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.

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; // this if is
  else a = c;
  // associated with this else
  }
  else a = d;
  // this else refers to if(i == 1)
  ```
- 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.

```java
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 + "."); } }
```

Java

```
Output : April is in the Spring.
```

Java

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.
- Java has a loop to fit any programming need.

while

- The while loop is Java's most fundamental loop 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.

```java
class While { public static void main(String args[]) { int n = 10;
while(n > 0) { System.out.println("tick " + n); n--; } } }
```
Java

```java
Output : tick 10 tick 9 tick 8 tick 7 tick 6 tick 5 tick 4 tick 3
tick 2 tick 1
```
Java

- 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 a loop can be empty. Here is an example to describe that.

```java
class NoBody { public static void main(String args[]) { 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); } }
```
Java

```java
Output : Midpoint is 150.
```
Java

- Here is how this while loop works. The value of i is incremented, and the value of j is
  decremented. These values are then compared with one another. If the new value of i is still
  less than the new value of j, then the loop repeats.
- If i is equal to or greater than j, the loop stops. Upon exit from the loop, i will hold a value that is midway between the original values of i and j. (Of course, this procedure only works when i is less than j to begin with.)
- As you can see, there is no need for a loop body; all of the action occurs within the conditional expression, itself.

- In professionally written Java code, short loops are frequently coded without bodies when the controlling expression can handle all of the details itself.

for

- Beginning with JDK 5, there are two forms of the for loop. The first is the traditional form
  that has been in use since the original version of Java.
- The second is the newer “for-each” form. Both types of for loops are discussed here, beginning with the traditional form.
- Here is the general form of the traditional for statement:
  for(initialization; condition; iteration) {
  // body }
- If only one statement is being repeated, there is no need for the curly braces.
- The for loop operates as follows. 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.
- It is important to understand that 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. 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. Consider the following example.

```java
class ForTick { public static void main(String args[]) { int n;
for(n=10; n>0; n--) System.out.println("tick " + n); } }
```

Java

```
Output : tick 10 tick 9 tick 8 tick 7 tick 6 tick 5 tick 4 tick 3
tick 2 tick 1
```
Java

Declaring Loop Control Variables Inside the for Loop

- Often the variable that controls a for loop is needed only for the purposes of the loop and
  is not used elsewhere.
- When this is the case, it is possible to declare the variable inside the initialization portion of the for. Consider the following example.

```
class ForTick { public static void main(String args[]) { // here, n
is declared inside of the for loop for(int n=10; n>0; n--)
System.out.println("tick " + n); } }
```
Java

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

- When you declare a variable inside a for loop, there is one important point to remember:
  the scope of that variable ends when the for statement does. (That is, the scope of the variable
  is limited to the for loop.)
- Outside the for loop, the variable will cease to exist. If you need to use the loop control variable elsewhere in your program, you will not be able to declare it inside the for loop.

Using the Comma

- There will be times when you will want to include more than one statement in the
  initialization and iteration portions of the for loop.
- Sometimes the loop is governed by two variables, it would be useful if both could be included in the for statement, itself, instead of b being handled manually. Fortunately, Java provides a way to accomplish this.

- To 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 a comma. Consider the following example.

```java
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); } } }
```

Java

```java
Output : a = 1 b = 4 a = 2 b = 3
```

Java

- In this example, the initialization portion sets the values of both a and b. The two comma-
  separated statements in the iteration portion are executed each time the loop repeats.

The For-Each Version of the for Loop

- Java adds the for-each capability by enhancing the for statement. The advantage of this approach is that no new keyword is required, and no preexisting code is broken. The for-each style of for is also referred to as the enhanced for loop.
- The general form of the for-each version of the for is shown here:
- 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, one at a time, from beginning to end.
- The collection being cycled through is specified by collection.
- Because the iteration variable receives values from the collection, type must be the same as
  (or compatible with) the elements stored in the collection.
- Here is an entire program that demonstrates the for-each version described:

```java
class ForEach { public static void main(String args[]) { int nums[] =
{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; int sum = 0; // use for-each style
for to display and sum the values for(int x : nums) {
System.out.println("Value is: " + x); sum += x; }
System.out.println("Summation: " + sum); } }
```
Java

```
Output : Value is: 1 Value is: 2 Value is: 3 Value is: 4 Value is: 5
Value is: 6 Value is: 7 Value is: 8 Value is: 9 Value is: 10
Summation: 55
```
Java

- As this output shows, the for-each style for automatically cycles through an array in
  sequence from the lowest index to the highest.

Iterating Over Multidimensional Arrays

- The enhanced version of the for also works on multidimensional arrays. Remember,
  however, that in Java, multidimensional arrays consist of arrays of arrays. (For example,
  a two-dimensional array is an array of one-dimensional arrays.)
- This is important when iterating over a multidimensional array, because each iteration obtains the next array, not an individual element. Furthermore, the iteration variable in the for loop must be compatible with the type of array being obtained.
- Consider the following example.

```java
class ForEach3 { public static void main(String args[]) { int sum =
0; int nums[][] = new int[3][5]; // give nums some values for(int i =
0; i < 3; i++) for(int j=0; j < 5; j++) nums[i][j] = (i+1)*(j+1); //
use for-each for to display and sum the values for(int[] x : nums) {
for(int y : x) { System.out.print(y + " "); sum += y; }
System.out.println(); } System.out.println("Summation: " + sum); } }
```
Java

```
Output : 1 2 3 4 5 2 4 6 8 10 3 6 9 12 15 Summation: 90
```
Java

Nested Loops

- Like all other programming languages, Java allows loops to be nested. That is, one loop
  may be inside another. For example, here is a program that nests for loops:
- Consider the following example.

```java
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(); } } }
```

Java

```java
Output : .......... ......... ........ ....... ...... ..... .... ...
.. .
```

Java

Jump Statements

- Java supports three jump statements: break, continue, and return. These statements transfer
  control to another part of your program.
- In addition to the jump statements discussed here, Java supports one other way that you can change your program’s flow of execution: through exception handling.

Using 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 “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.

- Consider the following example.

```
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."); } }
```

Java

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

Java

- As you can see, although the for loop is designed to run from 0 to 99, the break statement
  causes it to terminate early, when i equals 10.
- The break statement can be used with any of Java’s loops, including intentionally infinite
  loops.
- Note that 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.

Using continue

- 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. 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. For all three loops, any intermediate code is bypassed.

- 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:

```java
// 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(""); } } }
```

Java

return

- 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.
- Consider the following example.

```java
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."); } }
```

Java

```
Output : Before the return.
```

Java

- As you can see, the final println( ) statement is not executed. As soon as return is executed,
  control passes back to the caller.
- One last point: 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.
- To prevent this error, the if statement is used here to trick the compiler for the sake of this demonstration