Control Statements

Java's Selection Statements

- **If**

  if (condition) statement1;

  else statement2;

- **Nested ifs**

```
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 == 10)
```

The if-else-if Ladder

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

else
    statement;

switch

- The **switch statement is Java's multiway branch statement**

```
switch (expression) {
        case value1:
                // statement sequence
                break;
        case value2:
                // 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 **break** statement is used inside the switch to terminate a statement sequence
- This has the effect of **"jumping out"** of the switch.

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

Nested switch Statements

```
switch(count) {
case 1:
    switch(target)
 { // nested switch
    case 0:
          System.out.println("target is zero");
          break;
    case 1: // no conflicts with outer switch
          System.out.println("target is one");
          break;
}
break;
case 2: // ...
```

In summary, there are three important features of the **switch statement to note:**

- **switch can only test for equality, whereas if can evaluate any type of Boolean expression.**
- **No two case constants in the same switch can have identical values.**
- **more efficient than a set of nested ifs.**

Iteration Statements

while

- The **while loop is Java's most fundamental loop statement. It repeats a statement or block** while its controlling expression is true.

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

```
// The target of a loop can be empty.
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);
  }
}
```

- **do-while**

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

- **for**

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

- **Declaring Loop Control Variables Inside the for Loop**

```
for(int i=2; i <= num/i; i++)
```

- **Using the Comma**

```
for(a=1, b=4; a<b; a++, b--)
```

If you are familiar with C/C++, then you know that in those languages the comma is an operator that can be used in any valid expression. However, this is not the case with Java. ***In Java, the comma is a separator***

- **Some for Loop Variations**

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

```
// Parts of the for loop can be empty.
for( ; !done; )

for( ; ; ) {
// ...
}
```

The For-Each Version of the for Loop

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

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

For(int i=0;i<10;i++)
{
Sum+=num[i];
}
```

```
// Use break with a for-each style for.
class ForEach2 {
public static void main(String args[]) {
int sum = 0;
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
// use for to display and sum the values

for(int x : nums) {
System.out.println("Value is: " + x);
sum += x;
if(x == 5) break;
}

System.out.println("Summation of first 5 elements:
" + sum);
} }
```

```
// The for-each loop is essentially read-only.
class NoChange {
public static void main(String args[]) {
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

for(int x : nums) {
System.out.print(x + " ");
x = x * 10;
}

System.out.println();

for(int x : nums)
System.out.print(x + " ");

System.out.println();
} }
```

Iterating Over Multidimensional Arrays

```
// Use for-each style for on a two-dimensional
array.
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.println("Value  is: " + y);
      sum + = y;
    }
  }
  System.out.println("Summation:  " + sum);
 }
}
```

It is a reference to a one-dimensional array of integers. This is necessary because each iteration of the for obtains the next *array in nums, beginning with* the array specified by nums[0]. The inner for loop then cycles through each of these arrays, displaying the values of each element.

Applying the Enhanced for

```
// Search an array using for-each style for.
class Search {
  public static void main(String args[]) {
    int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
    int val = 5;
    boolean found = false;
     // use for-each style for to search nums for val
    for(int x : nums) {
      if(x == val) {
        found = true;
        break;
      }
    }

    if(found)
      System.out.println("Value found!");
  } }
```

- **Nested Loops**

Jump Statements

- Java supports three jump statements:
- **break, continue, and return.**
- **These statements transfer** control to another part of your program.
- **Using break**
- In Java, the **break statement has three uses.**
- ***it terminates a statement*** *sequence in a* ***switch statement.***
- ***it can be used to exit a loop.***
- ***it can be used as*** *a "civilized" form of goto.*

```
// Using break as a civilized form of goto.
class Break {
  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.");
    }  }}
```

```
// 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;
System.out.print(j + " ");
}
System.out.println("This will not print");
}

System.out.println("Loops complete.");
}
}
```

Using Continue

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

```
// 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();
    }
}
```

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

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.