Review of Control Flow

Control Flow: Controls the order in which statements are executed in your program

if and if-else: Conditionally execute a block of statements based on a boolean conditional expression. Example:

```java
if ( option == 1 )
    System.out.println( "Read image" );
else if (option == 2 )
    System.out.println( "Double" );
else if ( option == 9 )
    System.out.println( "Quit" );
else
    System.out.println( "Sorry, invalid" );
```

while and do-while loops: Repeatedly execute some block of statements as long as a condition holds.

The Switch Statement

Switch Statement: is a convenient (and often more efficient) way to perform a multi-way conditional based on a single control value.

Example:

```java
switch ( option ) {
    case 1:
        System.out.println( "Read image" );
        break;
    case 2:
        System.out.println( "Double" );
        break;
    case 9:
        System.out.println( "Quit" );
        break;
    default:
        System.out.println( "Sorry, invalid" );
        break;
}
```
The Switch Statement

General form:

```java
switch ( (control-expression) ) {
    case (case-label-1) :
        (statement-sequence-1)
        break;
    case (case-label-2) :
        (statement-sequence-2)
        break;
    ...
    case (case-label-n) :
        (statement-sequence-n)
        break;
    default :
        (default-statement-sequence)
        break;
}
```

The control expression can be of one of the following types:
- `char`, `int`, `short`, `byte`.
- not `float` or `double`,
- not `boolean` or `long`
- not an object (Too bad! Strings would have been nice.)

The "break" statement jumps out of the switch statement. Otherwise control flow just "falls through" into the next case.

```java
int option = 2;
switch ( option ) {
    case 1:
        System.out.println( "Read image" );
    case 2:
        System.out.println( "Double" );
    case 9:
        System.out.println( "Quit" );
    default:
        System.out.println( "Sorry, invalid" );
}
```
The Switch Statement

The falling through behavior is handy, because it allows you to combine cases. **Example:** Allowing either upper-case or lower-case for characters:

```java
char command = 'D';
switch ( command ) {
    case 'i':
    case 'I':
        MyUtility.insert();
        numberOfItems++;
        break;
    case 'd':
    case 'D':
        MyUtility.delete();
        numberOfItems--;
        break;
    ...
}
```

The Switch Statement

The "default" case is optional. If it is not included, and no case matches, then the switch statement does nothing.

It is considered good practice to always include a default case, if only to print an error message of an illegal choice.

Cases are not required to be in order. The following is legal, but is confusing for the reader.

```java
switch ( option ) {
    case 2:
    ...
    case 9:
    ...
    default:
    ...
    case 1:
}
```

**Recommended:** List cases in increasing order, and put the default last.
The For Loop

Common loop structure:

```plaintext
sum = 0.0;
〈initialization〉
while ( (boolean-test) ) {
  〈loop-body〉
  〈update〉
}
```

The for loop provides a shorthand for expressing this type of loop:

```plaintext
for ( (〈initialization〉); (〈boolean-test〉); (〈update〉) ) {
  〈loop-body〉
}
```

The above loop is equivalent to:

```plaintext
sum = 0.0;
for ( n = 1; n <= 3; n++ ) {
  sum += n;
}
```

The For Loop: Examples

Loop that counts from 100 down to 0: (100, 99, 98, ..., 1, 0)

```plaintext
for ( count = 100; count >= 0; count-- )
  System.out.println( count + " bottles of beer on the wall" );
```

Loop that counts by twos from 0 up to max+10: (0, 2, 4, ..., max+10)

```plaintext
for ( m = 0; m <= max+10; m += 2 ) {
  // … something exciting …
}
```

It is very convenient to declare the loop-control variable within the initialization. The scope of the variable is limited to the for loop:

```plaintext
for ( int i = 0; i < 20; i++ ) {
  sum = sum + i;
}
System.out.println ( i );  // this is a compiler error: i only accessible inside the loop
```
The For Loop: Further Elements

Multiple Initialization/Increment: Sometimes it is useful to have multiple initializations and multiple increments. This can be achieved by separating the operations by commas.

Example:
```java
for ( m = 0, n = 100; m < n; m++, n -= 2 ) {
    // … something senseless …
}
```
Equivalent to:
```java
m = 0;
n = 100;
while ( m < n ) {
    // … something senseless …
    m++;
    n -= 2;
}
```

The For Loop: Common Errors

Semicolon after the increment:
```java
for ( int j = 0; j < 100; j++; ) System.out.println( j );
```

Semicolon after closing parenthesis:
```java
for ( k = 0; k < 10; k++ ) ;
    System.out.println( k );
```
The println is executed only once, after the loop exits (prints 10).

Infinite loop due to careless loop condition:

Example: The following intended to count from low up to high-1.
```java
int low = …
int high = …
for ( z = low; z != high; z ++ ) {
    …
        // what if high < low?
}
```
The "break" Statement

We saw that the break statement exits from a switch statement. It can also be used to exit immediately from any loop:
- while
- do-while
- for

Example: Generate up to 500 random numbers, but exit the loop as soon as a value less than 0.01 is generated.

```java
int count;
for (count = 1; count <= 500; count++) {
    if (Math.random() < 0.01) break;
}
System.out.println("count = " + count);
```

The "break" Statement

Warning: A break "violates" the loop's natural structure, and can be hard on the reader (particularly if the loop is large). It is best to avoid them, unless it is needed to keep the code simple.

Example: We can easily avoid the break in the above example, by creating a boolean variable.

```java
boolean foundIt = false;
for (count = 1; (count <= 500 && !foundIt); count++) {
    if (Math.random() < 0.01) foundIt = true;
}
System.out.println("count = " + count);
```
The "continue" Statement

The **break** statement exits entirely from a loop.
The **continue** statement is similar, but jumps immediately to the test portion of the loop, ready to start a new iteration.

**Example:**

```java
    count = 1;
    while ( count < 20 ) {  // continue jumps here
        sum += count;
        if ( ... ) break;
        if ( ... ) continue;
    }  // continue jumps here
System.out.println( "Done" );  // break jumps here
```

**Warning:** We only mention **continue** for completeness.

**break:** Is usually bad practice. Use it very sparingly.

**continue:** Is considered bad practice. Avoid it altogether.