Control Structures

1 Structured Programming

In reasoning about the flow of control to and from a statement, consider control flow issues:

- Multiple vs. single entry ("How did we get here?")
- Multiple vs. single exit ("Where do we go from here?")
- `goto` considered harmful (`goto` makes it hard to answer questions above)

Structured programming: block structure, single entry, single exit, no `goto`. All algorithms expressed by:

- Sequence - one statement after another
- Selection - conditional execution (not conditional jumping)
- Iteration - loops

2 Boolean Values

In Java, boolean values have the `boolean` type. Four kinds of boolean expressions:

- `boolean` literals: `true` and `false`
- `boolean` variables
- expressions formed by combining non-`boolean` expressions with comparison operators
- expressions formed by combining `boolean` expressions with logical operators

3 Boolean Expressions Formed From Comparisons

Simple boolean expressions formed with comparison operators:

- Equal to: ==, like $=$ in math
  - Remember, = is assignment operator, == is comparison operator!
- Not equal to: `!=`, like $\neq$ in math
- Greater than: >, like $>$ in math
- Greater than or equal to: >=, like $\geq$ in math
- ...

Examples:

```
1 == 1 // true
1 != 1 // false
1 >= 1 // true
1 > 1  // false
```

4 Boolean Expressions Formed From Logical Operators

Simple boolean expressions can be combined to form larger expressions using:

- And: `&&`, like $\wedge$ in math
- Or: `||`, like $\vee$ in math

Examples:

```
(1 == 1) &&  (1 != 1) // false
(1 == 1) ||  (1 != 1) // true
```

Also, unary negation operator `!`:

```
!true     // false
!(1 == 2) // true
```

5 The `if-else` Statement

Conditional execution:

```
if (booleanExpression)
    // a single statement executed when booleanExpression is true
else
    // a single statement executed when booleanExpression is false
```

- *booleanExpression* must be enclosed in parentheses
- `else` not required

Example:

```
if ((num % 2) == 0)
    System.out.printf("I like %d.%n", num);
else
    System.out.printf("I'm ambivalent about %d.%n", num);
```

6 Ternary If-Else Expression

The ordinary `if-else` control structure is a statement, leading to conditional assignment code like this:

```
String dinner = null;
if (temp > 60) {
    dinner = "grilled";
} else {
    dinner = "baked";
}
```

The ternary operator combines the above into one expression (recall that expressions have values):

```
String dinner = (temp > 60) ? "grilled" : "baked"
```

7 Blocks

Java is block-structured. You can enclose any number of statements in curly braces ({ ... }) to create a block. Blocks are like single statements (not expressions - they don't have values).

```
if ((num % 2) == 0) {
    System.out.printf("%d is even.%n", num);
    System.out.println("I like even numbers.");
} else {
    System.out.printf("%d is odd.%n", num);
    System.out.println("I'm ambivalent about odd numbers.");
}
```

The Java conventions recommend using braces always, even for single statements. A very common error is adding statements to an if-branch and forgetting to add braces.

8 Multi-way `if-else` Statements

This is hard to follow:

```
if (color.toUpperCase().equals("RED")) {
    System.out.println("Redrum!");
} else {
    if (color.toLowerCase().equals("yellow")) {
        System.out.println("Submarine");
    } else {
        System.out.println("A Lack of Color");
    }
}
```

This multi-way `if-else` is equivalent, and clearer:

```
if (color.toUpperCase().equals("RED")) {
    System.out.println("Redrum!");
} else if (color.toLowerCase().equals("yellow")) {
    System.out.println("Submarine");
} else {
    System.out.println("A Lack of Color");
}
```

9 Short-Circuit Evaluation

Here's a common idiom for testing an operand before using it:

```
if ((kids !=0) && ((pieces / kids) >= 2))
    System.out.println("Each kid may have two pieces.");
```

In this example Java uses short-circuit evaluation. If

```
kids !=0
```

evaluates to `false`, then the second sub-expression is not evaluated, thus avoiding a divide-by-zero error.

Note: You can force a complete evaluation by using `&` or `|`, for example if you have side effects you want to ensure happen in the second expression. We mention this fact for completeness but implore you not to write such code.

See Conditionals.java for examples.

10 The `switch` Statement

Java provides `switch` statement for multi-way branching.

```
switch (expr) {
case 1:
    // executed only when case 1 holds
    break;
case 2:
    // executed only when case 2 holds
case 3:
    // executed whenever case 2 or 3 hold
    break;
default:
    // executed only when other cases don't hold
}
```

- Execution jumps to the first matching case and continues until a `break`, `default`, or `switch` statement's closing curly brace is reached
- Type of `expr` can be `char`, `int`, `short`, `byte`, or `String`

- In example above, what is type of `expr`?

11 Avoid the `switch` Statement

The `switch` statement is error-prone.

- `switch` considered harmful. 97% of fall-throughs unwanted[1]
- Anachronism from "structured assembly language", a.k.a. C (a `switch` is just a jump table)

You can do without the switch statement. See

- CharCountSwitch.java for a `switch` example,
- CharCountIf.java for the same program using an `if` statement in place of the `switch` statement, and
- CharCount.java for the same program using standard library utility methods.

12 Loops and Iteration

Algorithms often call for repeated action or iteration, e.g. :

- "repeat ... while (or until) some condition is true" (looping) or
- "for each element of this array/list/etc. ..." (iteration)

Java provides three iteration structures:

- `while` loop
- `do-while` loop
- `for` iteration statement

13 `while` and `do-while`

`while` loops are pre-test loops: the loop condition is tested before the loop body is executed

```
while (condition) { // condition is any boolean expression
      // loop body executes as long as condition is true
}
```

[1] Peter van der Linden, *Deep C Secrets*

`do-while` loops are post-test loops: the loop condition is tested after the loop body is executed

```
do {
      // loop body executes as long as condition is true
} while (condition)
.
```

The body of a `do-while` loop will always execute at least once.

14 `for` Statements

The general `for` statement syntax is:

```
for(initializer; condition; update) {
     // body executed as long as condition is true
}
```

- *intializer* is a statement
  - Use this statement to initialize value of the loop variable(s)
- *condition* is tested before executing the loop body to determine whether the loop body should be executed. When false, the loop exits just like a while loop
- *update* is a statement
  - Use this statement to update the value of the loop variable(s) so that the *condition* converges to `false`

15 `for` Statement vs. `while` Statement

The `for` statement:

```
for(int i = 0; i < 10; i++) {
     // body executed as long as condition is true
}
```

is equivalent to:

```
int i = 0
while (i < 10) {
  // body
  i++;
}
```

`for` is Java's primary iteration structure. In the future we'll see generalized versions, but for now `for` statements are used primarily to iterate through the indexes of data structures and to repeat code a particular number of times.

16 `for` Statement Examples

Here's an example from CharCount.java. We use the `for` loop's index variable to visit each character in a `String` and count the digits and letters:

```
int digitCount = 0, letterCount = 0;
for (int i = 0; i < input.length(); ++i) {
    char c = input.charAt(i);
    if (Character.isDigit(c)) digitCount++;
    if (Character.isAlphabetic(c)) letterCount++;
}
```

And here's a simple example of repeating an action a fixed number of times:

```
for (int i = 0; i < 10; ++i)
        System.out.println("Meow!");
```

17 `for` Statement Subtleties

Better to declare loop index in `for` to limit it's scope. Prefer:

```
for (int i = 0; i < 10; ++i)
```

to:

```
int i; // Bad.  Looop index variable visible outside loop.
for (i = 0; i < 10; ++i)
```

You can have multiple loop indexes separated by commas:

```
String mystery = "mnerigpaba", solved = ""; int len = mystery.length();
for (int i = 0, j = len - 1; i < len/2; ++i, --j) {
    solved = solved + mystery.charAt(i) + mystery.charAt(j);
}
```

Note that the loop above is one loop, not nested loops.
Beware of common "extra semicolon" syntax error:

```
for (int i = 0; i < 10; ++i); // oops!  semicolon ends the statement
    print(meow);  // this will only execute once, not 10 times
```

18 Forever

Note that in the context of programming, infinite means "as long as the program is running." Here are two idioms for infinite loops. First with `for`:

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

and with `while`:

```
while (true) {
    // forever
}
```

Infinite loops are useful for things like game loops and operating system routines that poll input buffers or wait for incoming network connections. In both of these cases the loop is inteded to run for the duration of the program.

See Loops.java for loop examples.

19 `break` and `continue`

Java provides two non-structured-programming ways to alter loop control flow:

- `break` exits the loop, possibly to a labeled location in the program
- `continue` skips the remainder of a loop body and continues with the next iteration

Consider the following while loop:

```
boolean shouldContinue = true;
while (shouldContinue) {
    System.out.println("Enter some input (exit to quit):");
    String input = System.console().readLine();
    doSomethingWithInput(input); // We do something with "exit" too.
    shouldContinue = (input.equalsIgnoreCase("exit")) ? false : true;
}
```

We don't test for the termination sentinal, "exit," until after we do something with it. Situations like these often tempt us to use `break` ...

20 `break`ing out of a `while` Loop

We could test for the sentinal and `break` before processing:

```
boolean shouldContinue = true;
while (shouldContinue) {
    System.out.println("Enter some input (exit to quit):");
    String input = System.console().readLine();
    if (input.equalsIgnoreCase("exit")) break;
    doSomethingWithInput(input);
}
```

But it's better to use structured programming:

```
boolean shouldContinue = true;
while (shouldContinue) {
    System.out.println("Enter some input (exit to quit):");
    String input = System.console().readLine();
    if (input.equalsIgnoreCase("exit")) {
        shouldContinue = false;
    } else {
        doSomethingWithInput(input);
    }
}
```