Control Structures

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

All algorithms expressed by:

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

No `goto`

Boolean Values

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

Comparison Expressions

- 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

Comparison Examples

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

Logical Combinations

- 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
```

if-else Statement

```
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

if-else Example

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

Conditional Assignment

`if-else` is a statement, so conditional assignment like this:

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

Ternary If-Else Expression

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

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

Blocks

Enclose any number of statements in curly braces ({ . . . }) to create a block, which is like a single statement.

```
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.");
}
```

Always use curly braces in control structures.

Nested if-else

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");

}
```

Multi-way `if-else`

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");
}
```

Short-Circuit Evaluation

Common idiom for testing an operand before using it:

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

If `kids !=0` evaluates to `false`, then the second sub-expression is not evaluated, thus avoiding a divide-by-zero error.
See Conditionals.java for examples.

switch

```
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`

Avoid `switch`

The `switch` statement is error-prone.

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

You can do without the `switch`. 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.

Repeated Operations

<iframe width="560" height="315" src="https://www.youtube.com/embed/mXPeLctgvQI" frameborder="0" allowfullscreen></iframe>

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 Loop/Iteration Structures

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

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
}
```

do-while

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.

for Statements

The general for statement syntax is:

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

- intializer is a statement
- condition is a boolean expression – when false loop exits
- update is a statement

for vs. while

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.

Simple Repetition

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!");
```

Iterating With Indexes

From `CharCount.java`. We use the `for` loop's loop variable to index each character in a `String`

```
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++;
}
```

Multiple Loop Variables

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.

Loop Gotchas

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
```

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)
```

Forever

Infinite means "as long as the program is running."
With `for`:

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

and with `while`:

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

See `Loops.java` for loop examples.

break and continue

Non-structured ways to alter loop control flow:

- break exit the loop, possibly to a labeled location in the program
- continue skip the remainder of a loop body and continue 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 . . .

‘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);
    }
}
```

Reasoning About Imperative Programs

What will this code print?

```java
public class ShortCircuit {

    private static int counter = 0;

    private static boolean inc() {
        counter++;
        return true;
    }
    public static void main(String args[]) {
        boolean a = false;
        if (a || inc()) {
            System.out.println("Meow");
        }
        System.out.println("counter: " + counter);
        if (a && inc()) {
            System.out.println("Woof");
        }
        System.out.println("counter: " + counter);
    }
}
```

Reasoning About Imperative Programs

Substitute values, trace code, track ‘counter‘ and output:

```
Code                                        counter  Output

boolean a = false;                          0
if (a || inc()) {                           1
    System.out.println("Meow");             1        Meow
}                                           1
System.out.println("counter: " + counter); 1        counter: 1
if (a && inc()) {                           1
    System.out.println("Woof");             1
}                                           1
System.out.println("counter: " + counter); 1        counter: 1
```

Key points:

- ‘inc()‘ always returns ‘true‘
- Due to short-curcuit evaluation, ‘inc()‘ not always evaluated