

Topic 2: Statements

Reading: JBD Chapter 3

Statements

- A *statement* is one step in a Java program.
- Statements are terminated by semicolons (`;`).
- By default, statements are executed one after another, in the order written.
- Some kinds of statements alter the flow of control
 - Branches (`if`, `switch`)
 - Loops (`for`, `while`, `do`)
 - Jumps (`break`, `continue`, `return`)

- The simplest kind of statement is an empty statement

;

- A variable declaration is a statement

```
boolean married;
```

```
double annualYield = 0.055;
```

- Any side-effecting expression can be a statement

```
married = false;
```

```
annualYield *= 1.5;
```

```
yearsOfService++ ;
```

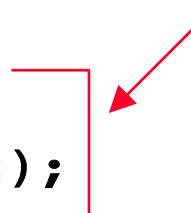
- A call to a method can be a statement

```
System.out.println("Hello");
```

Blocks

- A sequence of statements enclosed in { } behave like a single statement, called a *block*.

```
{  
    x = 5;  
    y = 7;  
    int z = x + y;  
    System.out.println(z);  
}
```



If a block contains a declaration, the scope of the variable is from the declaration to the end of the block.

General form of an **if** statement

```
if ( test )  
    statement1  
else  
    statement2
```

test can be any boolean expression

statement1 is executed if *test* is true (the "true branch")

statement2 is executed if *test* is false (the "false branch")

- The **else** part can be omitted
- Don't forget parentheses around the *test*
- Example:

```
if (x < y)  
    System.out.println("x is less than y");  
else  
    System.out.println("x is not less than y");
```

No semicolon here!

- The branches of an `if` statement are often blocks

```
if (stateOfResidence.equals("CA")) {
    fees = 8000;
    housing = 12000;
    transportation = 1000;
}
else {
    fees = 28000;
    housing = 15000;
    transportation = 2000;
}
```

- Note conventions for indentation and placement of `{ }`
- Indentation does not affect program execution

```
if (5 > 7)
    System.out.println("A");
    System.out.println("B");
System.out.println("C");
```

- The test can be a complex boolean expression

```
if (state.equals("CA") && income < 10000
    || state.equals("AK") && income < 12000)
    eligible = true;
```

- Does this work?

```
if (state == "CA" && income < 10000)
    eligible = true;
```

- Does this work?

```
if (state.equals("CA" || "NY" || "TX"))
    eligible = true;
```

- `if` statements can be nested


```
if (age < 25)
    if (sex == 'M')
        premium = 535;
    else
        premium = 350;
else
    if (sex == 'M')
        premium = 400;
    else
        premium = 300;
```

- The *orthogonality principle*:
Any kind of statement may be used where a statement is expected.

- *CAUTION:* `else` binds to the nearest unmatched `if`

```
if (state.equals("CA"))
    if (taxable)
        charge = listPrice * 1.08;
else
    charge = listPrice;
```

WARNING:
Misleading
indentation!



- What is the value of `charge` if state is "NV"?

- *CAUTION:* `else` binds to the nearest unmatched `if`

```
if (state.equals("CA"))
```


```
    if (taxable)
```

```
        charge = listPrice * 1.08;
```

```
    else
```

```
        charge = listPrice;
```

More
accurate
indentation



- What is the value of `charge` if state is "NV"?

- One fix for the misplaced **else**: use **&&** in the test

```
if (state.equals("CA") && taxable)
    charge = listPrice * 1.08;
else
    charge = listPrice;
```

- Another fix: enclose each branch in { }

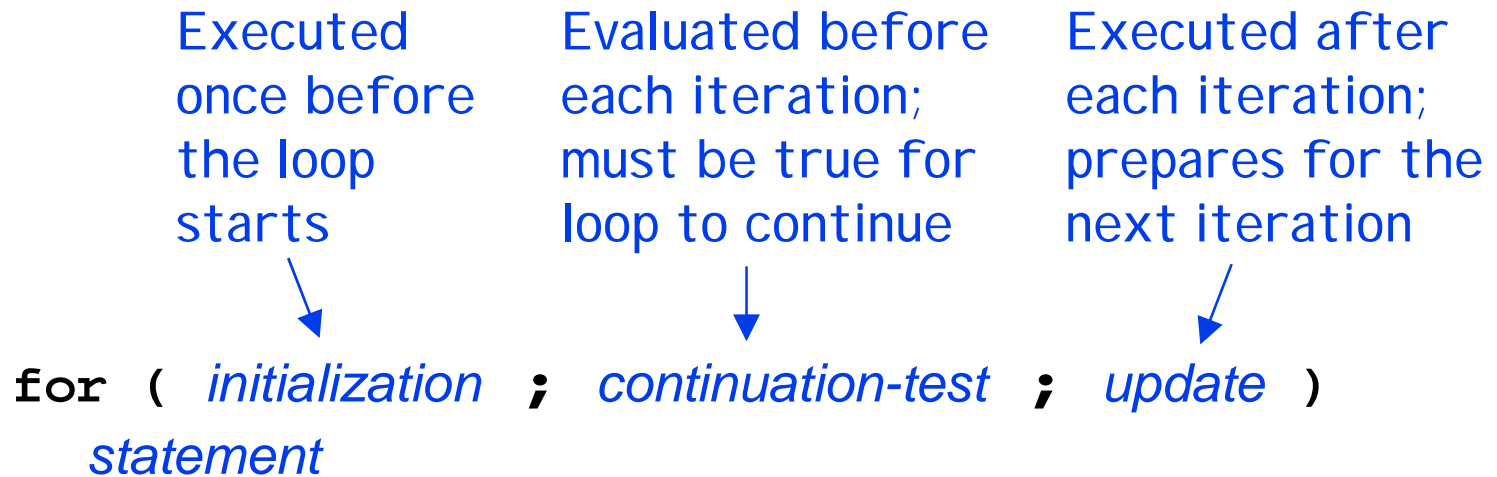
```
if (state.equals("CA")) {
    if (taxable) {
        charge = listPrice * 1.08;
    }
}
else {
    charge = listPrice;
}
```

- Enclosing every branch in { } is a safe practice.

- A "stack" of if-else statements:

```
if (shape.equals("circle")) {
    area = Math.PI * radius * radius;
    perimeter = 2.0 * Math.PI * radius;
}
else if (shape.equals("rectangle")) {
    area = length * width;
    perimeter = 2.0 * (length + width);
}
else if (shape.equals("equilateral triangle")) {
    area = side * side * Math.sqrt(3.0) / 4.0;
    perimeter = 3.0 * side;
}
```

General form of a **for** statement



- The body of a for-loop is often a block

```
for (i = 1; i <= 10; i++) {  
    square = i * i;  
    System.out.println  
        ("Square of " + i + " is " + square);  
}
```

- The initialization and body of a for-loop can declare local variables

```
for (int i = 1; i <= 10; i++) {  
    int square = i * i;  
    System.out.println  
        ("Square of " + i + " is " + square);  
}
```

- How often will the body of this loop execute?

```
for (int i = 0; i < 5; i++);  
{  
    System.out.println(i);  
}
```

Moral: Be careful where you put your semicolons.

Another kind of loop: **while**

- General form of a **while** statement:

Evaluated before each iteration;
must be true for loop to continue



while (*continuation-test*) *statement*

- The body of a while-loop is often a block
- Initialize the test variables outside the block

```
int i = 1;
while (i <= 10) {
    int square = i * i;
    System.out.println
        ("Square of " + i + " is " + square);
    i++;
}
```

Yet another loop statement: **do**

- General form of a **do** statement:

```
do statement
while ( continuation-test );
```



Evaluated after each iteration;
must be true for loop to continue

- The body of a do-loop is often a block

```
int i = 1;
do {
    int square = i * i;
    System.out.println
        ("Square of " + i + " is " + square);
    i++;
} while (i <= 10);
```

- The body is always executed at least once

Jumping out of loops: **break** and **continue**

- A **break** statement terminates a loop
- A **continue** statement terminates one iteration of a loop and begins the next iteration

```
Scanner scan = new Scanner(System.in);
while (true) { // seemingly runs forever
    System.out.print("Enter a positive integer");
    System.out.print(" or 0 to exit");
    int n = scan.nextInt();
    if (n == 0) break;
    if (n < 0) continue;
    System.out.println("Square root of " + n
        + " is " + Math.sqrt(n));
    // continue lands here
}
// break lands here
```

Keeping things honest: the **assert** statement

- Useful for debugging and catching errors.
- General form:

A boolean expression
that you believe to be
true

A message that you want
to appear if the boolean
expression is not true



```
assert expression : message ;
```

- Java will halt with an error message if your assertion is not true
- Example:

```
assert x > y : "Awk! x <= y!";
```

- Run your program with assertions enabled:

```
java -ea MyProgram
```