Control Flow for Selection Statements

1. Integer value 1 generated.
2. First if-else evaluated.
3. Only code in the main if-clause is executed.
4. Interior if-else is evaluated.
5. Only code in first inner else-clause will execute.

Differences between if, else, and else-if

- A set of instructions inside an if-clause block may or may not execute.
- When we add an else-block, then exactly one set of instructions must execute.

Strings:
- String s = button.getInput();
- s = s.toLowerCase();
- int flip = (int)(Math.random() * 2 + 1);
- if (flip == 1) {
  win.add(heads, 0);
  if (s.equals("heads")) {
    l2.setText("You win!");
  } else if (s.equals("tails")) {
    l2.setText("You lose!");
  } else {
    l2.setText("Be serious...");
  }
} else {
  win.add(tails, 0);
  if (s.equals("tails")) {
    l2.setText("You win!");
  } else if (s.equals("heads")) {
    l2.setText("You lose!");
  } else {
    l2.setText("Be serious...");
  }
}

```
if (i % 2) == 0 )

int i = (int)(Math.random() * 6) + 1;
if ( (i % 2) == 0 )
    System.out.println( "Even" );
else
    System.out.println( "Odd" );
```

```
17 February 2020
Software Design I (CS 120)
```

Programming Assignments

- Pay attention to filenames given in the writeup
- Comment your code. A lot.
- Block comment at top of each .java file you write or edit:

```
/**
 * Description of the problem and your program
 * 
 * Your name
 * CS120 Section [your section]
 * Spring 2020
 * 
 * Programming Assignment [number goes here]
 * [due date]
 */
```

```
17 February 2020
Software Design I (CS 120)
```

```
```
Differences between *if*, *else*, and *else-if*

- When we add an *else* to an *if*-clause block, we now have multiple conditions, each of which may or may not execute.

```java
int i = (int)(Math.random() * 6) + 1;
if (i <= 2)
    System.out.println("Low");
else if (i <= 4)
    System.out.println("Medium");
else
    System.out.println("High");
```

Now this code prints something for any legal value of `i` (1 to 6).

- Again, adding an *else*-block, means exactly one set of instructions must execute.

```java
int i = (int)(Math.random() * 6) + 1;
if (i <= 2)
    System.out.println("Low");
else if (i <= 4)
    System.out.println("Medium");
else
    System.out.println("High");
```

If `i` is greater than 4, nothing happens.

The boolean Primitive Type

- Java boolean variables/expressions are used for true or false values
- Constant expressions: `true, false`
- Binary infix operators: `&&` (and), `||` (or)
- Unary prefix operator: `!` (not)

<table>
<thead>
<tr>
<th>Operator Precedence (highest to lowest)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>++</code>, <code>--</code></td>
</tr>
<tr>
<td><code>!</code> (unary negation)</td>
</tr>
<tr>
<td><code>+</code>, <code>/</code>, <code>%</code></td>
</tr>
<tr>
<td><code>+ -</code> (subtraction)</td>
</tr>
<tr>
<td><code>&lt; &lt;=</code> <code>&gt;</code> <code>&gt;=</code></td>
</tr>
<tr>
<td><code>==</code>, <code>!=</code></td>
</tr>
<tr>
<td><code>&amp;&amp;</code>, `</td>
</tr>
<tr>
<td><code>=</code></td>
</tr>
</tbody>
</table>

As usual, to combine these with relational and mathematical operators, we must remember *precedence order*.

For a set of operations this complex, use of parentheses to mark precedence is often highly recommended!

Using boolean Expressions

- When we use relational operators (==, !=, etc.) properly, we produce an expression that is either true or false.

```java
int i = 3;
t
int j = 5;
if (i <= j) {
    System.out.println(i);
}
```

Such an expression can be used anywhere boolean values could be used:

- condition for an if-clause
- instantiation of some boolean variable
- Anywhere else...

```java
int i = 3;
t
int j = 5;
boolean b = (i <= j);
if (b) {
    System.out.println(i);
}
```

Evaluating boolean Expressions

- The simple numeric comparison operators and equality relations are easy to evaluate.

```
<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P &amp; &amp; Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
```

- To determine whether a more complex logical formula is true or false, we use truth tables for logical operators.

```
<table>
<thead>
<tr>
<th>P</th>
<th>P ! P</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>
```

```
| P | Q | P || Q |
|---|---|------|
| T | T | T    |
| T | F | T    |
| F | T | T    |
| F | F | F    |
```
Evaluating boolean Expressions

- Sometimes we need to negate a boolean expression that contains multiple variables

<table>
<thead>
<tr>
<th>P</th>
<th>!P</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>!(P &amp;&amp; Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

| P | Q | !(P || Q) |
|---|---|-----------|
| T | T | F         |
| T | F | F         |
| F | T | F         |
| F | F | F         |

| P | Q | P && Q | !(P & & Q) | P || Q |
|---|---|-------|-------------|-------|
| T | T | T     | F           | T     |
| T | F | F     | T           | T     |
| F | T | T     | F           | T     |
| F | F | F     | T           | F     |

Evaluating boolean Expressions

- Sometimes we need to negate a boolean expression that contains multiple variables

| P | Q | R | P || Q || R | !(P || Q || R) | !(P & & Q & & R) |
|---|---|---|-------|-------------|-----------------|------------------|
| T | T | T | T     | T           | F               | F                |
| T | T | F | T     | T           | F               | F                |
| T | F | T | T     | T           | F               | F                |
| F | T | T | T     | T           | F               | F                |
| F | T | F | T     | T           | F               | F                |
| F | F | T | T     | T           | F               | F                |
| F | F | F | T     | T           | F               | F                |

17 February 2020
Software Design I (CS 120)
Review: Relations between Primitives

- We can combine primitive types when we compare them.
- We can use any relational operator to produce a boolean value.
- Just like arithmetic, Java does an automatic widening of all types as needed so they can be compared meaningfully.

```java
int i = 3;
double j = 3.0;
if ( i == j ) {
    System.out.println( "Equal" );
}
```

Expression then evaluates to be true

```
Scanner scan = new Scanner( System.in );
String input = scan.next();
String check = "Exit";
if ( input == check ) {
    System.out.println( "Goodbye" );
}
```

The == operator checks that two values are exactly the same.

For reference types, this means that they refer to the same address in memory.

```
Scanner scan = new Scanner( System.in );
String input = scan.next();
String check = "Exit";
if ( input.equals( check ) ) {
    System.out.println( "Ciao!" );
}
```

The equals() method checks whether or not they are identical, character-by-character.

Equality between Reference Types

- For any reference (Class) type, to check whether the content of two objects is the same, we must use an equals() method, not the basic equality operator.

```java
Scanner scan = new Scanner( System.in );
String input = scan.next();
String check = "Exit";
if ( input.equals( check ) ) {
    System.out.println( "Ciao!" );
}
```

```
String s1 = "abacus";
String s2 = "barnacle";
if ( s1.compareTo( s2 ) < 0 ) {
    System.out.println( s1 + " before " + s2 );
}
else if ( s1.compareTo( s1 ) == 0 ) {
    System.out.println( s1 + " equals " + s1 );
}
else if ( s2.compareTo( s1 ) > 0 ) {
    System.out.println( s2 + " after " + s1 );
}
```

Comparison between Reference Types

- Every primitive type can be compared using basic operators like <, <=, etc.
- This is not true for reference (Class) types.
- If such a type is supposed to be comparable, then it must have a method, compareTo(), that:
  1. Takes another object of the same type as input.
  2. Returns an integer that:
     a) Zero (0) if the objects are equal
     b) Negative (less than 0) if the object calling the method comes before the input
     c) Positive (greater than 0) if the object calling the method comes after the input.

Here, all of the comparisons will be true, since the two String objects are compared using their proper method.
This Week & Next

- **Meetings this week:**
  - Wednesday: in the CS Lab (16 Wing)
  - Monday/Tuesday/Friday: regular classroom

- **Program 02:** due 11:59 PM, Wednesday 19 February

- **Reading:** Chapter 4 due Noon, Friday 21 February

- **Office Hours:** Wing 212
  - Monday/Wednesday/Friday: 11:00 AM–12:00 PM
  - Tuesday: 3:15 PM–4:15 PM

- **CS Lab & Tutor Hours:** Posted on my webpage