Programming Assignments

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

```java
/**
 * Description of the problem and your program
 * 
 * Your name
 * CS120 Section [your section]
 * Fall 2019
 * 
 * Programming Assignment [number goes here]
 * [due date]
 */
```

```
String s = button.getInput();
int flip = (int)(Math.random() * 2 + 1);
if (flip == 1) {
    win.add(heads, 0);
    if (s.equals("heads")) {
        l2.setText(" You win!");
    }
} 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...");
    }
}
```

Control Flow for Selection Statements

1. Integer value \(i\) 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.

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

- 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.

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

Using boolean Expressions

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

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

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

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

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)`

  ```java
  Operator Precedence (highest to lowest)
  ++ --
  ! - (unary negation)
  * / %
  + - (subtraction)
  < <= > >=
  == !=
  &&
  ||
  =
  ```

- 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!

Evaluating boolean Expressions

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

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

  ```java
  P | Q | P && Q
  T | T | T
  T | F | F
  F | T | F
  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>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>T</td>
<td>!P</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>!(P &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 | T |
| F | F | T |

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

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 | F | F |
| T | T | F | T | F | F |
| T | F | T | T | F | F |
| F | T | T | T | F | F |
| F | F | T | T | F | F |
| F | F | F | T | T | T |
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 are able to be compared meaningfully

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

int i is widened to a double. Expression then evaluates to be true

Relations between Reference Types

- Do not assume comparison operators work for non-primitive types
- For any reference type (Class), basic comparison is not reliable

```java
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.

Here, no matter what the user types, this will be false, since the two String objects are created in two different ways in two different locations.

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!" );
}
```

The equals() method for the String class actually checks whether or not they are identical, character-by-character.

If a class has such a method, then we can compare it reliably. When writing our own classes, if we want to be able to do this, then we must write our own method to do so.

Here, as long as the user types in the same word exactly, this will be true.

Comparison between Reference Types

- Every primitive type can be compared using basic operators like <, <=, etc.
- This is not true for reference (Class) types
- If 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 is:
     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

```java
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 );
}
```

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

- **Program 02**: due 11:59 PM Tonight

- Reading Assignment 3: due Thursday, 26 September at Noon

- **Midterm 1**: Monday, 07 October

- **Office Hours**: Wing 212
  - Monday/Friday: 2:15 PM–3:15 PM
  - Tuesday: 1:30 PM–2:30 PM
  - Wednesday: 12:05 PM–1:00 PM

- **Lab and Tutor Hours**: On my website