Lecture 20:
Selection, Part II

Sierra College
CSCI-12
Spring 2015
Mon 04/13/15

Announcements

- **General**
  - <u>**All**</u> assignments are now graded: please check on your grades in Canvas
  - UC Davis Picnic Day on Sat 4/18: campus-wide open house (all day)
    - Department exhibits, demos, tours, lectures, live music, athletic events, …
- **Schedule**
  - Spring withdraw deadline is Thurs 4/16
    - Final off-ramp: after that point, you <u>will</u> receive a letter grade for this class
    - Please check your grades in Canvas, and assess where you stand ("gut check")
      - Let's talk if any concerns…
- **Current assignments**
  - PRGM19: Age Utils (due Sunday 04/19 @ 11pm) lab time this wk
    - We went over this in lab last week
    - Any questions so far??
    - **Expectations**: follow ALL good software conventions (header, braces/indentations, commenting, naming conventions, etc.)
    - I will publish the test program I'll use to grade <u>your</u> **UtilsFL.getAge()**
      - Your program's methods MUST run cleanly against this program!
      - Use this to CHECK your algorithm before submitting!

Lecture Topics

- **Last time**:
  - Conditions, *if* and *if-else* logic
- **Today**:
  - Finish up *if-else if* logic
  - The *switch* statement
  - Nested program logic
  - Testing your logic

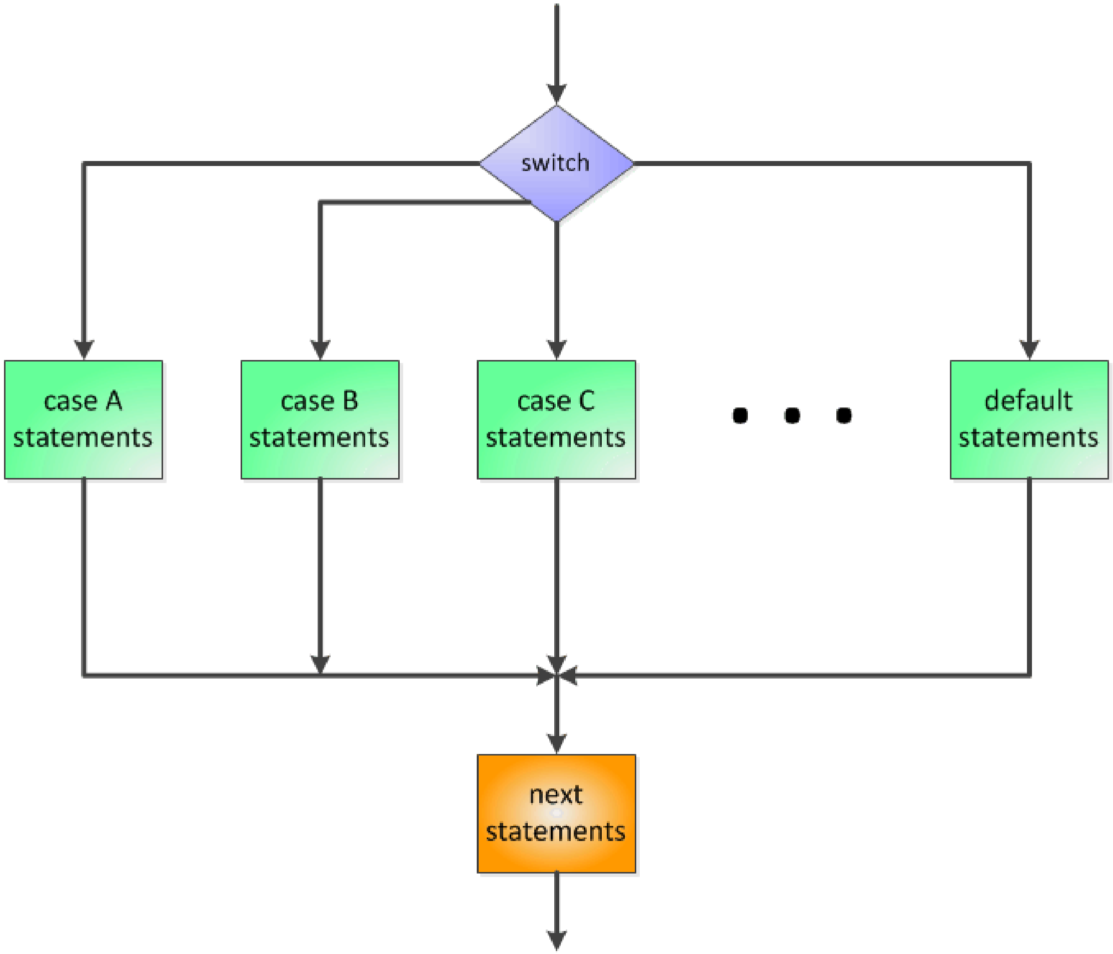
For Next Time

- **Lecture Prep**
  - Text readings and lecture notes
- **Program**
  - Continue on the next assignment
  - Suggestion: write your client class first, using the *existing* version of the starter UtilsFL class
  - Work out how to express the needed “age” logic

The *switch* Structure

- The ***switch*** structure is an alternative to using *if-else if* structures, for certain types of comparisons
  - Both cases represent choices between **multiple mutually exclusive options**
- Each *switch* **case** represents the comparison of a variable or expression to certain types of known constant values
- We can perform a ***switch*** on:
  - **integer** values (*byte*, *short*, or *int* only – no *long*)
  - **character** values (*char* – since these ultimately represent Unicode values)
  - ***String** values* (NEW in Java 7)
- The *switch* structure forks directly to the applicable case
  - No multiple condition evaluation is needed
  - No overhead of checking each and every *if-else if* case
- Alternate names:
  - *switch* statement, *case* statement

The *switch* Flowchart



Comparison of ***switch*** vs. ***if-else if***

```
int value;  // or char or String
// set value somehow…

switch (value) {
    case 1:
        // actions for value=1 here
        break;
    case 2:
        // actions for value=2 here
        break;
    case 3:
        // actions for value=3 here
        break;
    default;
        // actions for all other values
        break;
}
```

```
int value;  // or char or String
// set value somehow…

if (value == 1) {
    // actions for value=1 here
}
else if (value == 2) {
    // actions for value=2 here
}
else if (value == 3) {
    // actions for value=3 here
}
else {
    // actions for all other values
}
```

See ***SelectionSwitchComparison.java*** in **Example Source Code**

switch General Form & Operation

General form:

```
switch ( expression )  {
    case constant1:
        // statement(s);
        break;              // optional
    case constant2:
        // statement(s);
        break;              // optional

     ...
    default:                // optional
        // statement(s);
        break;              // optional
}
```

Operation:

- *expression* is first evaluated
- Its value is compared to the *case* constants in order
- When a match is found, those *case* statements are executed
  - If a break is encountered first, the switch block execution is complete
  - If another case statement is encountered first, those statements are ALSO executed
- Optional elements:
  - The *break* statements
  - The *default* case
  - Statements within any one *case*
- *case* statements may be “stacked”
  - Identical code may be “shared” amongst cases

Some Notes On *switch*

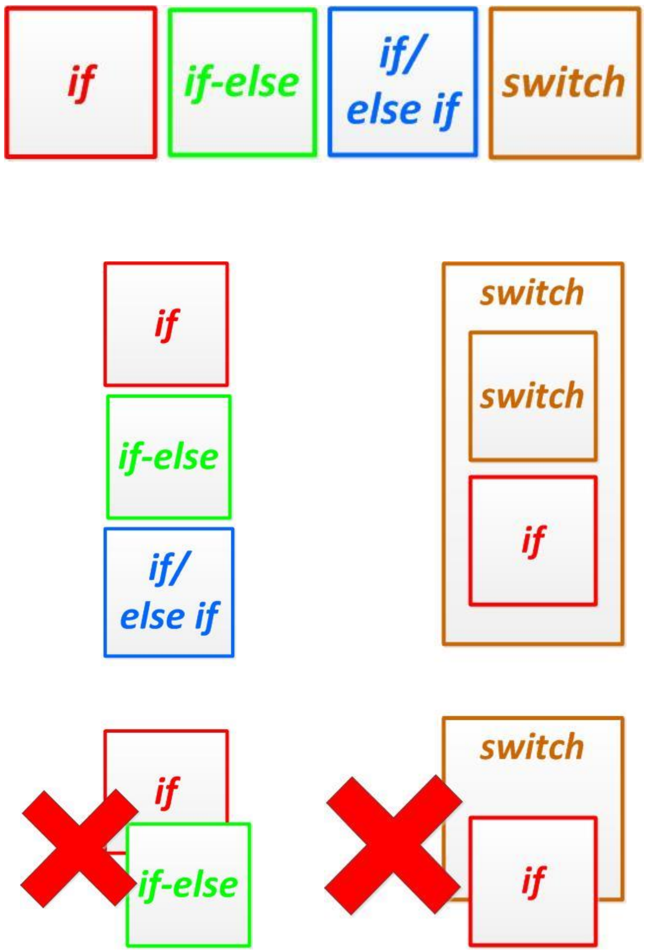
- One of the *if* structures or *switch*: which one to use?
  - If the situation involves comparison between values, or some sort of detailed logic: <u>probably use an *if* construct</u>
  - If the situation calls for decision between multiple discrete, <u>known</u> integer/*char*/*String* options: <u>can use *switch*</u>
- A common "gotcha" using *switch*:
  - Forgetting to include a *break* statement at the end of a *case*
  - This allows code to "fall thru" to the next option
  - Unwanted case logic may be executed also
- A common usage for *switch*:
  - Keyboard input handling
  - Cases correspond to various keystrokes (handle case sensitivity??)

Examples Using *switch*

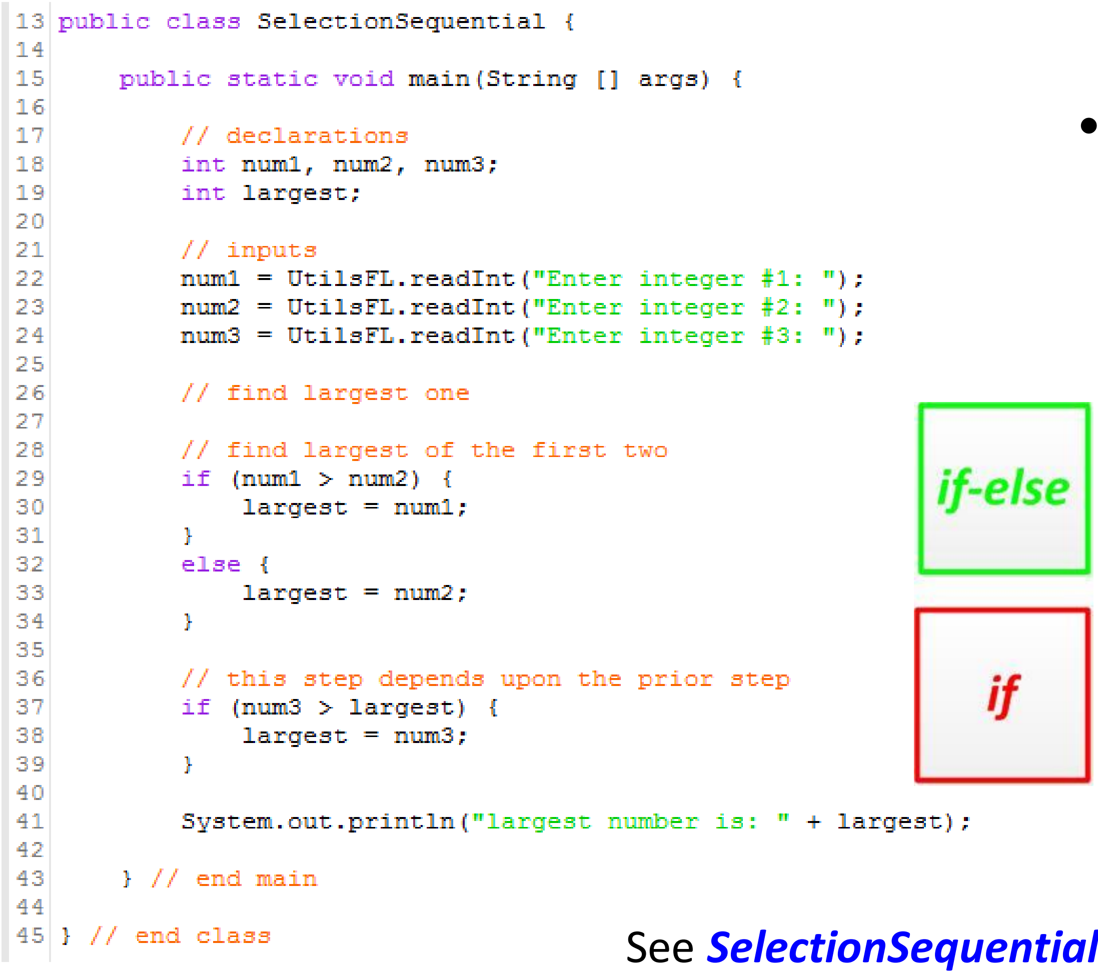
- **Numbers**
  - In Example Source Code: ***SelectionSwitchNumbers.java***
    - Demonstrates: Numeric *switch* usage, fall-thru, using an expression in a case
- **Chars**
  - In Example Source Code: ***SelectionSwitchChars.java***
    - Demonstrates: Char *switch* usage, handling keyboard input options, case-insensitivity, *case* calls to outside methods
- **Strings**
  - In Example Source Code: ***SelectionSwitchStrings.java***
    - Demonstrates: String *switch* usage, fall-thru, case-insensitivity, case calls to various object methods

Hybrid Selection Structures

- The basic selection structure “blocks” may be combined in any hybrid manner to form more complex program logic
  - **Sequential**: When the result of a prior selection block is needed to proceed with a later selection block
  - **Nested**: When the logic of an inner selection block is conditionally executed only if a containing outer selection block is true
- The only restriction is that there can be no “overlap” among blocks
  - One block must “terminate” (close) before another one can begin

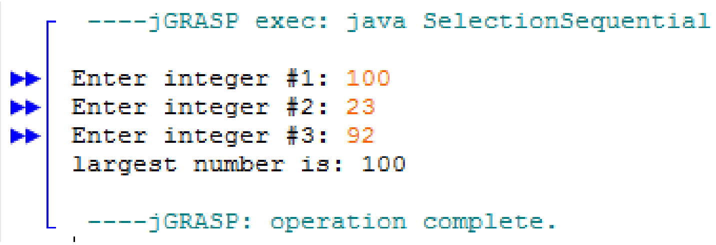


Example: Sequential Selection



```java
public class SelectionSequential {

    public static void main(String [] args) {

        // declarations
        int num1, num2, num3;
        int largest;

        // inputs
        num1 = UtilsFL.readInt("Enter integer #1: ");
        num2 = UtilsFL.readInt("Enter integer #2: ");
        num3 = UtilsFL.readInt("Enter integer #3: ");

        // find largest one

        // find largest of the first two
        if (num1 > num2) {
            largest = num1;
        }
        else {
            largest = num2;
        }

        // this step depends upon the prior step
        if (num3 > largest) {
            largest = num3;
        }

        System.out.println("largest number is: " + largest);

    } // end main

} // end class
```

- The 2nd if block depends upon the outcome of the 1st if-else block



```
 ----jGRASP exec: java SelectionSequential

Enter integer #1: 100
Enter integer #2: 23
Enter integer #3: 92
largest number is: 100

 ----jGRASP: operation complete.
```

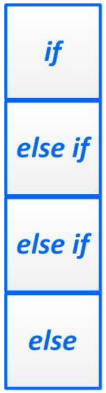
See ***SelectionSequential.java*** in **Example Source Code**

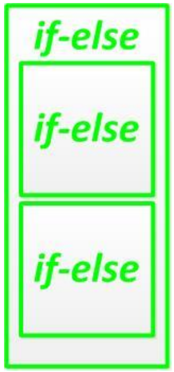
Example: Nested Selection

- Objective: determine a suggested activity, given:
  - Wind speed
  - Temperature

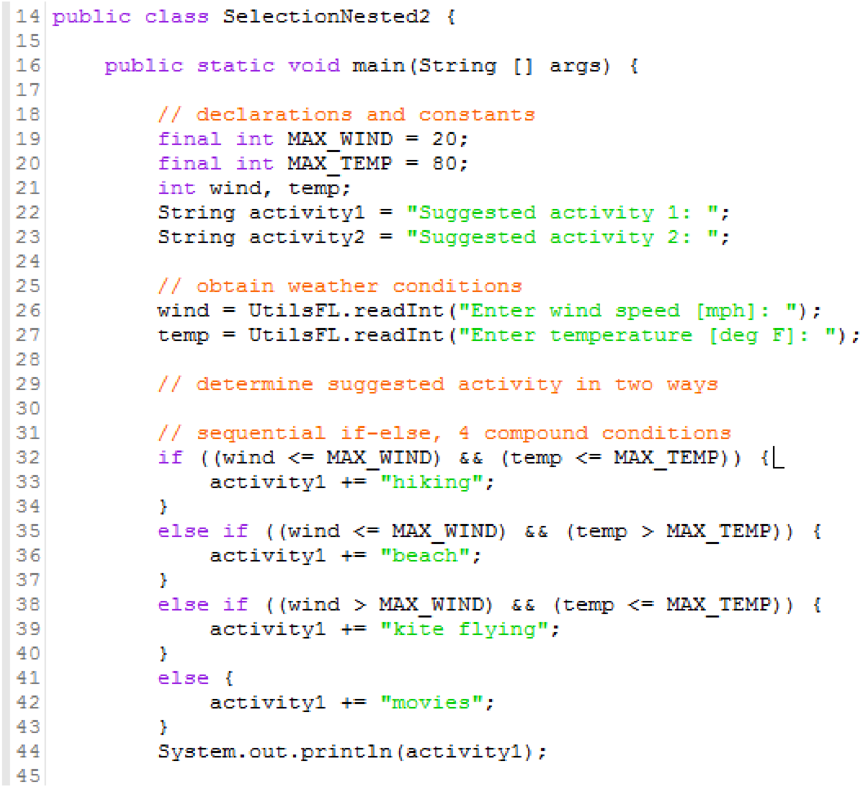
	wind <= 20	wind > 20
temp <= 80	Hiking	Kite flying
temp > 80	Beach	Movies

- This can be implemented in two ways:
  - Sequential if-else
    - 4 individual cases
      - Cell-by-cell
    - Compound conditions
  - Nested if-else
    - 2 nested levels
      - First columns, then rows (or vice versa)
    - Single conditions

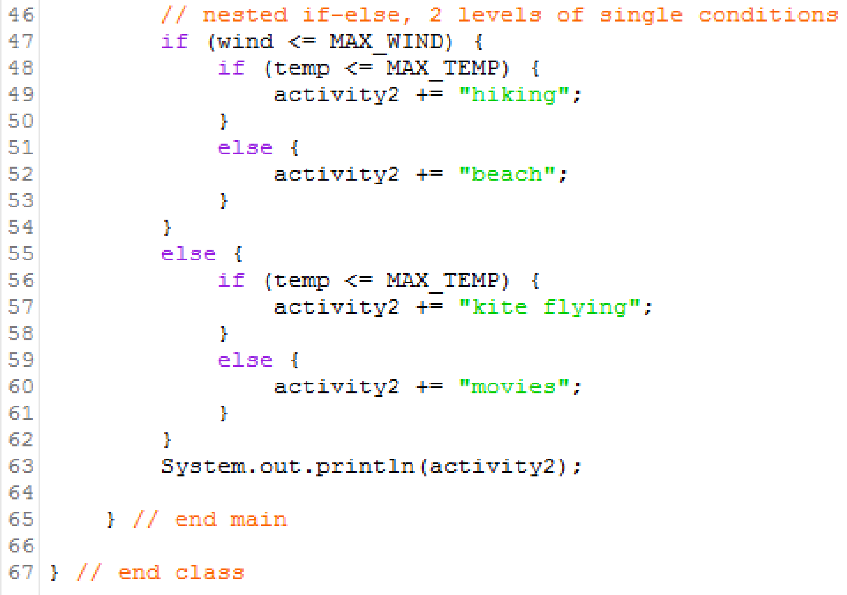




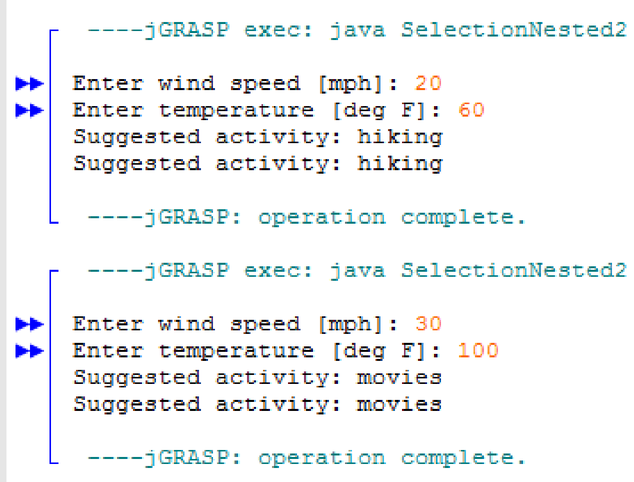
Example: Nested Selection (cont.)



```java
public class SelectionNested2 {

    public static void main(String [] args) {

        // declarations and constants
        final int MAX_WIND = 20;
        final int MAX_TEMP = 80;
        int wind, temp;
        String activity1 = "Suggested activity 1: ";
        String activity2 = "Suggested activity 2: ";

        // obtain weather conditions
        wind = UtilsFL.readInt("Enter wind speed [mph]: ");
        temp = UtilsFL.readInt("Enter temperature [deg F]: ");

        // determine suggested activity in two ways

        // sequential if-else, 4 compound conditions
        if ((wind <= MAX_WIND) && (temp <= MAX_TEMP)) {
            activity1 += "hiking";
        }
        else if ((wind <= MAX_WIND) && (temp > MAX_TEMP)) {
            activity1 += "beach";
        }
        else if ((wind > MAX_WIND) && (temp <= MAX_TEMP)) {
            activity1 += "kite flying";
        }
        else {
            activity1 += "movies";
        }
        System.out.println(activity1);

```



```java
        // nested if-else, 2 levels of single conditions
        if (wind <= MAX_WIND) {
            if (temp <= MAX_TEMP) {
                activity2 += "hiking";
            }
            else {
                activity2 += "beach";
            }
        }
        else {
            if (temp <= MAX_TEMP) {
                activity2 += "kite flying";
            }
            else {
                activity2 += "movies";
            }
        }
        System.out.println(activity2);

    } // end main

} // end class
```



```
 ----jGRASP exec: java SelectionNested2

Enter wind speed [mph]: 20
Enter temperature [deg F]: 60
Suggested activity: hiking
Suggested activity: hiking

 ----jGRASP: operation complete.

 ----jGRASP exec: java SelectionNested2

Enter wind speed [mph]: 30
Enter temperature [deg F]: 100
Suggested activity: movies
Suggested activity: movies

 ----jGRASP: operation complete.
```

Tradeoffs:

- **Simpler logic + more complex conditions**
- **Complex (nested) logic + simpler conditions**
- **In either case, 4 logic conditions to handle**

See ***SelectionNested2.java*** in **Example Source Code**

“Gotcha” Example: Default Evaluation

```
if ( x == 2 )
   if ( y == x )
      System.out.println( "x and y equal 2" );
   else
      System.out.println( "x equals 2, but y does not" );
```

The *else* clause is paired with the second *if* , that is:

```
   if ( y == x )
```

(However: for this course, DON’T turn in this code without some explicit braces added!)

“Gotcha” Fix: Explicit Evaluation

```
if ( x == 2 ) {
   if ( y == x )  {
      System.out.println( "x and y equal 2" );
   }
}
else {
   System.out.println( "x does not equal 2" );
}
```

With curly braces added, the *else* clause is paired with the first *if,* that is: if (x == 2)

Coding standard for this course: always add the other braces for explicit clarity around the single-line *if* statements

“Gotcha” Example: The "Dangling *else*"

A **dangling *else*** occurs when an *else* clause cannot be paired with an *if* condition

```
if ( x == 2 )
  if ( y == x )
      System.out.println( "x and y equal 2" );
  else // paired with ( y == x )
      System.out.println( "y does not equal 2" );
else  // paired with ( x == 2 )
  System.out.println( "x does not equal 2" );
else  // no matching if!
  System.out.println( "x and y are not equal" );
```

This generates the **compiler error**: 'else' without 'if '

Anti-Bugging Suggestions

- Always use parentheses for conditions
- Don’t use a semicolon after conditions
  - Less of an issue if the opening brace comes right after
- Always indent the true/false blocks for clarity
  - This becomes increasingly important as logic becomes more deeply nested
  - Very helpful for visual layout debugging
- Line up the closing brace with its opening *if*
  - Again, helpful for visual layout and debugging
- Technically, the braces are not required for single-statement ifs, but good practice to use them always!

Anti-Bugging Suggestions

- Suggestion:
  - Start with the logical framework of your code first, then add the conditions and code
  - Comment closing braces if needed, to “match them up”
- Suggestion:
  - Indent and nest your braces and conditions consistently (3-4 spaces minimum)
  - Use braces even for single-line if/else, even though not required

```
if ( ( ) && ( ) ) {
   if ( ) {
   }
   else {
   }
}
else if ( ( ) || ( ) ) {
   if (  ( ) || ( ) ) {
   }
   if ( ) {
   }
}
else {
}
```

Anti-Bugging Suggestions

- Suggestions:
  - Always lay out the logical framework of your switch structure first
  - Make sure to indent all case logic
  - Make sure to always include a *default* case
  - Make sure to explicitly add *breaks* to each case
    - You can always group identical cases later

```
switch () {
  case 1:
                         ← Add code here
    break;

  case 2:
                         ← Add code here
    break;

  case 3:
                         ← Add code here
    break;

  default:
                         ← Add code here
    break;
}  // end switch
```

Testing Types

- **White box testing**: when we know the internal details of the code (*as in this class*)
  - **Develop a test plan with data input sets that will exercise every possible (known) logic branch**
  - Check the results against the program specs

- **Black box testing**: when we treat the code as a black box, and know nothing of its details
  - Develop a test plan with data input sets based upon the program specs (which is all we'd probably know)
  - Again, check the results against the program specs

Testing Methods

- Once you have eliminated any **compiler errors** in your code, you then must closely examine the correctness of its execution
  - Find and fix any **logic errors**
  - The program does what you TOLD it do, not what you WANT it to do!
- To locate any logic errors, there are two main approaches:
  - Print statements
    - Liberally include *println*() statements in your code
    - Before and after calculations, inside the various logic branches, etc.
    - Make sure to disable (comment out) or remove any residual diagnostic outputs, before submitting
  - Debugger
    - Step thru the execution path of your code with various inputs
    - Confirm you took the logical forks in the road you expected to!