Conditionals and Loops

- Objectives - when we have completed this set of notes, you should be familiar with:
  - flow of control: sequence, selection, iteration
  - boolean expressions
  - equality, relational, and logical operators
  - selection: if statement (with optional else or else if)
  - iteration: while statement (a.k.a. while loop)
  - the ArrayList class
  - comparing data
  - indentation and block statements revisited
  - more details on if statements and while loops
  - file input using the File and Scanner classes
  - iterators

Flow of Control

- The order of statement execution in a method is called the ***flow of control***
  - **Sequence** - Unless specified otherwise, the order of statement execution in a method is sequential; i.e., one statement after another
  - **Selection** - statements that allow us to decide whether or not to execute a particular statement (or block of statements); i.e., select among alternatives
    Examples: *if*, *if-else*, *switch* statements
  - **Iteration** (repetition) - statements that allow us to execute a statement (or block of statements) iteratively or repeatedly, as long as some condition is true; i.e., loop through the statement or block of statements
    Examples: *while*, *do-while*, *for* statements (a.k.a. loops)
- ***Boolean expressions*** (which evaluate to true or false) are used by Selection and Iteration statements (except *switch and for each*) to determine whether a statement (or block of statements) is executed

Flow of Control

Q1

- When we read source code, the sequence, selection, and iteration is relative to the method we are reading
  - Example: In the `main` method, we may have sequence, selection, and iteration. If one of the statements invokes/calls a method, then we may jump (or step-in in debug mode) to that method where we will again encounter sequence and possibly selection and/or iteration while the flow of control is in this method
- You can use the debugger to follow the detailed flow of control (see examples in later slides)

Boolean Expressions

- A boolean expression is an expression that evaluates to true or false.
  - Example: (where num1 and num2 are **int** values)

```
num1 > num2 + 5
```

- When a boolean expression is evaluated, the result can be assigned to a boolean variable
  - Example: (where email references a **String** object)

```
boolean validEmail = email.contains("@");
```

boolean expression

Boolean Expressions

- An `if` statement uses a boolean expression as its condition (if and if-else statements with simple boolean expressions were introduced in the class notes on Data and Expressions)
  Example: if `temp` is greater than 80 then print "Stay indoors."

boolean expression

```
if (temp > 80) {
    System.out.println("Stay indoors.");
}
```

- Now we'll consider more complex boolean expressions (equality, relational, and logical operators)

Operators

- Equality and Relational Operators (review):
  - Evaluate to ***true*** or ***false*** and must have compatible operands (only numeric types, including char, can be used with relational operators: <, >, <=, >=)
  - Have lower precedence than arithmetic operators

Operator	Meaning
==	equal to
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Q2 Q3

Operators

- Logical operators - have boolean operands; evaluate to a boolean result (***true*** or ***false)***; have lower precedence than the equality and relational operators

Operator	Meaning
!	Logical NOT (applied before &&, ||)
&&	Logical AND (applied before ||)
||	Logical OR

  - Example**:** A String *mail* is a valid email address if it contains an @ symobol **and** its length **is greater than or equal to** 5 **and** it does **not** contain a space.

```
if (mail.contains('@') && mail.length() >= 5
      && !mail.contains(' ')) {
   System.out.println("Valid e-mail address!");
}
```

Operators

(Java Lang. Spec. - Conditional Operators)

- The && and || operators are "short circuited"
  - If the first operand of the && is false, then the other operand is not evaluated
  - If the first operand of the || is true, the other operand is not evaluated
- Suppose that strIn is a String. Which of the two *if* clauses will cause a run-time error if strIn is equal to null?

```
if (strIn != null && strIn.length() > 0)
```

or…

```
if (strIn.length() > 0 && strIn != null)
```

if Statement
(using logical operator)

- Example: "If the temperature (temp) is greater than 80 and humidity is greater than or equal to 60, then tell the user to stay indoors"

```
if (temp > 80 && humidity >= 60) {
    System.out.println("Hot: Stay indoors.");

}
```

if-else Statement
(i.e., if with optional *else)*

- Suppose you wanted to add "… otherwise, tell the user that the weather is good."

```
if (temp > 80 && humidity >= 60) {
   System.out.println("Hot: Stay indoors.");
}
else {
   System.out.println("Weather is good.");
}
```

if, else if, else Statement
(*i.e., if* with optional *else if* and *else*)

- What if there were other specific conditions that require a different action?
  - If the temperature > 80 and humidity >= 60, tell user it's hot. `if`
  - Otherwise, if the temperature < 40, tell the user it's cold. `else if`
  - For any other condition, tell the user that the weather is good. `else`

if, else if, else Statement
(using logical operator)

```
if (temp > 80 && humidity >= 60) {
  System.out.println("Hot: Stay indoors.");
}
else if (temp < 40) {
  System.out.println("Cold: Stay indoors.");
}
else {
  System.out.println("Weather is good.");
}
```

if, else if, . . ., else Statement

- An *if* statement can have any number of *else if* blocks (*else* block containing an *if* statement)

```
if (condition1) {

}
else
   if (condition2) {

   }
   // . . .
   else
      if (conditionN) {

     }
      else {

      }
```

usually written like this to improve readibilty

```
if (condition1) {

}
else if (condition2) {

}
// . . .
else if (conditionN) {

}
else {

}
```

- The *else* (or *else if*) clause is optional

- Examples: If_Else_Example.java Triangle.java

while Statement

- A ***while*** statement (or ***while*** loop) will continually execute a statement or block of statements as long as its condition (boolean expression) is true; i.e., it repeats the statement (or block) until the condition is false

```
while (/* boolean expression */) {
  /* code performed on each iteration */
}
```

- Any variables in the boolean expression need to initialized before the loop; code in the loop body should alter values so that the condition is eventually false and the loop terminates

while Statement

- Example: print all numbers from 1 to 10

```
int count = 1;
while (count <= 10) {
    System.out.println(count);
    count++;
}
```

- The debugger is a useful tool for seeing the control flow in loops

Count1.java

while Statement

- Consider a `NumbersSet` class which has **`int`** fields `low` and `high`, intended to hold positive integers
- The `NumbersSet` class includes two methods:
  - `findEvensBetween` - returns a `String` that includes all even numbers between values of the `low` and `high` fields (inclusive) of a `NumbersSet` object
  - `findCommonDivisors` - returns a `String` that includes the positive common divisors of the `low` and `high` fields in a `NumbersSet` object
- Each method requires a **`while`** loop to perform its work; let's look at the details of these methods

while Statement

Q4 Q5

- The `NumbersSet` class – method details
  - `findEvensBetween` - returns a `String` that includes all even numbers between values of the `low` and `high` fields (inclusive) of a `NumbersSet` object
    - Begin with a `candidate` number equal to `low`
    - While `candidate` is less than or equal to `high`
      - If `candidate` is divisible by 2, concatenate `candidate` to the end of the `String` to be returned
      - Increment `candidate`
    - Return the result `String`
  - `findCommonDivisors` - see source code for details
- NumbersSet.java NumbersSetDriver.java

java.util.ArrayList

- The `ArrayList` class defines an object that can hold a list of other objects called elements
- Includes methods to add an element, remove an element, get an element, find the index of an element, determine if the list is empty, and determine the size (number of items in the list)
- Your class should include an import statement

```
import java.util.ArrayList;
```

- Then in a method (e.g., main), you can declare an `ArrayList` instance that can hold objects

```
ArrayList names = new ArrayList();
```

ArrayList

- You can (and should) use a generic type to specify the type of objects the list can hold
- Examples:
  - To ensure the `ArrayList` names can hold only objects of type String, the generic type `<String>` (i.e., class name in angle brackets) is used in the declaration and constructor

```
ArrayList<String> names = new ArrayList<String>();
```

  - To ensure the `ArrayList` titles can only hold objects of type `Book`:

```
ArrayList<Book> titles = new ArrayList<Book>();
```

ArrayList

- See the Java API for a list of `ArrayList` methods. Commonly used methods are:
  - `add`: adds an object to the list
  - `remove`: removes an object or the object at a specified index
  - `get`: returns the object at the specified index
  - `indexOf`: returns the index of the specified object (indexed from 0)
  - `size`: returns the number of objects in the list
- See TriangleList.java (also see examples in text)

Comparing Data

- When comparing two items, consider the following:
  - Numeric types (including their corresponding wrapper classes) and type `char` can be compared using the equality and relational operators (==, !=, <, >, <=, >=)
  - Numeric types `double` and `float` should use == or != with care due to possible rounding; best to check that the absolute value of the difference between two items is less a specified tolerance
  - Object types (other than the numeric wrapper classes) can be compared using the equality operators (==, !=), but in most cases the object's `equals` or `compareTo` method should be used

Comparing *char* Values

- Recall a character (type `char`) is represented by a 16-bit numeric value (Unicode)
  - Letters A through Z: numeric values 65 to 90
  - Letters a through z: numeric values 97 to 122
  - What happens if you add 32 to an upper-case char value?

```
char value = 'G' + 32;
```

  - You can thus use relational and equality operators on `char` values as well. Suppose that letterValue is of type char…

```
if (letterValue >= 65 && letterValue <= 90) {
    System.out.println("Capital letter");
}
```

Comparing *double* Values

- When calculations are done on **`double`** (or **`float`**) values, there can be rounding due to the binary arithmetic and underlying 64-bit IEEE floating point representation (sign, exponent, and mantissa)
- Rather than using ==, it is best to check that the absolute value of the difference between two items is less that some specified tolerance
  - For example, assume d1 and d2 are variables of type **`double`** and the constant `TOLERANCE = 0.000001`, then

```
if (Math.abs(d1 - d2) < TOLERANCE) {
    System.out.println("consider equal");
}
```

Comparing Objects

- You can also use equality operators (== and !=) on objects, but remember that reference variables hold memory addresses. The results may not be what you expect!

- Try the following in interactions:

```
String s1 = new String("Red Sox");
String s2 = new String("Red Sox");
s1 == s2
false
```

Comparing Objects

- Instead of ==, <u>use the **`equals`** method to compare objects</u>
  - Returns a **`boolean`** indicating whether the objects are equal as defined in the class
  - You can find out how the `equals` method works for a particular Java class by consulting the Java API
- For the `String` class, objects are compared based on the characters that they contain

```
String s1 = new String("Red Sox");
String s2 = new String("Red Sox");
s1.equals(s2)
true
```

Comparing Objects

- Some classes provide a **`compareTo`** method, which returns an **`int`** (rather than boolean)

```
int comparison = obj1.compareTo(obj2);
```

- Interpreting the return value:
  - Less than 0 indicates obj1 < obj2
  - Equal to 0 indicates obj1 is equal to obj2
  - Greater than 0 indicates obj1 > obj2
- Class-specific so check the Java API for each class to see how objects are compared

Comparing Objects

- The **`compareTo`** method compares two `String` objects by comparing their individual characters left to right until <, ==, or > can be determined

- What does the following code print?

```
String food1 = "Apple", food2 = "Banana";
if (food1.compareTo(food2) < 0) {
   System.out.println(food1 + " before " + food2);
}
else if (food1.compareTo(food2) > 0) {
   System.out.println(food2 + " before " + food1);
}
else {
   System.out.println(food2 + " and " + food1
                      + " are the same");
}
```

Prints: `Apple before Banana`

Comparing Objects

- Remember that any upper case character will have a lower numeric value than any lower case character

```
String food1 = "apple", food2 = "Carrot";
if (food1.compareTo(food2) < 0) {
   System.out.println(food1 + " before " + food2);
}
else if (food1.compareTo(food2) > 0) {
   System.out.println(food2 + " before " + food1);
}
else {
   System.out.println(food2 + " and " + food1
                      + " are the same");
}
```

Prints: `Carrot before apple`

Comparing Objects

- The `String` class has **`equalsIgnoreCase`** and **`compareToIgnoreCase`** methods

```
String food1 = "apple", food2 = "Carrot";
if (food1.compareToIgnoreCase(food2) < 0) {
   System.out.println(food1 + " before " + food2);
}
else if (food1.compareToIgnoreCase(food2) > 0) {
   System.out.println(food2 + " before " + food1);
}
else {
   System.out.println(food2 + " and " + food1
                      + " are the same");
}
```

Prints: `apple before Carrot`

Indentation Revisited

- Remember that indentation in Java is for the human reader and is ignored by the computer

```
if (total > MAX)
   System.out.println ("Error!!");
   errorCount++;
```

Despite what is implied by the indentation above, the increment will occur whether the condition is true or not (the correct indentation below shows the error more clearly)

```
if (total > MAX)
   System.out.println ("Error!!");
errorCount++;
```

Block Statements

- Several statements can be grouped together delimited by braces into a ***block*** statement
- A block statement can be used wherever a statement is called for in the Java syntax rules

```
if (total > MAX) {
   System.out.println ("Error!!");
   errorCount++;
}
```

- Our coding standard (supported by Checkstyle) requires blocks in `if` statements

Nested *if* Statements

- When an `if` or `if-else` contains other `if` or `if-else` statements, we have *nested `if` statements*
- An `else` clause is matched to the last unmatched `if` (no matter what the indentation implies, unless braces are used)
- Braces can be used to specify the `if` statement to which an `else` clause belongs
- **Always use braces!**

Taxes.java TaxesIfWithoutBraces.java

Infinite Loops

- When a `while` loop is executing, eventually its boolean expression should become false
- Otherwise, we have an *infinite loop*, which will execute until the program is interrupted (e.g., the user manually ending the program)
- Common logical error
- Double check the logic of a program to ensure that your loops will terminate normally

Infinite Loops

- An example of an infinite loop:

```
int count = 1;
while (count <= 25)
{
    System.out.println (count);
    count = count - 1;
}
```

- This loop will continue executing until interrupted (e.g., entering Ctrl-C in DOS window or clicking "End" on jGRASP Run I/O tab) or until an underflow error occurs

CountInfinite.java

Nested Loops

- Similar to nested `if` statements, loops can be nested as well
- For example, the body of a `while` loop can contain another `while` loop
- For each iteration of the outer `while` loop, the inner `while` loop iterates completely

Nested Loops

- How many times will the string "Here" be printed?

```
count1 = 1;
while (count1 <= 10)
{
    count2 = 1;
    while (count2 <= 20)
    {
        System.out.println ("Here");
        count2++;
    }
    count1++;
}
```

10 * 20 = 200

Nested Loops

- Example: Read in a line of text from the user; print the words in reverse order; query the user to do again; repeat if y or Y is entered.
- Strategy:
  - Use an outer loop to read lines of text
  - Use an inner loop to store words in an `ArrayList`
  - Print the `ArrayList`
  - Print the elements of the `ArrayList` in order (using a loop)
  - Print the elements of the `ArrayList` in reverse order (using a loop)
  - Repeat?

ReverseWords.java

break and *continue*

- A `break` statement in a loop will skip the rest of the code in that iteration and exit the loop
- The `continue` statement will skip the rest of the code in that iteration and attempt the next iteration of the loop
- The `break` and `continue` statements for loops are generally used in conjunction with an `if` statement inside a loop

BreakWhileExample.java ContinueWhileExample.java

Reading from a File

- The `Scanner` can be used to read from a text file just as we read text from `System.in`
- Required import statements:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
```

- Include **`throws`** clause with main

```
public static void main(String[] args)
                    throws FileNotFoundException
```

  - Indicates that a `FileNotFoundException` may be thrown when a `Scanner` is created on a `File` and that your program is going to ignore it
  - If a `FileNotFoundException` does occur the program will end immediately

Reading from a File

- Create a Scanner object on a new File object where `fileName` is `String`

```
Scanner scanFile =
        new Scanner(new File(fileName));
```

- Use a loop to read in the file data, e.g.,

```
while (scanFile.hasNext()) {. . .}
```

  The data is read in using the `Scanner` methods we have used to read from `System.in` (`scanFile.nextLine()`, `scanFile.next()`, `scanFile.nextInt()`, etc.); the data is usually stored and/or processed in the loop

- After the loop, close Scanner object:

```
scanFile.close();
```

Reading from a File

- Example: ReverseLinesReadFromFile.java
  Read in a lines of text from a file; print lines in reverse order
- Strategy:
  - Read file name from user
  - Use loop to read lines from file and store in an `ArrayList`
  - Print the `ArrayList`
  - Use loop to print elements of the `ArrayList` in order
  - Use loop to print the elements of the `ArrayList` in reverse order

Also, see ReverseWordsFromFile.java
ReadItemsFromFile.java

Reading from a File

- Example: TriangleListApp.java
  Create TriangleList object from file data then print out the TriangleList object followed by a summary of its data.
- Strategy:
  - Read file name from user; read list name from file
  - Use a loop to read each set of Triangle data; create a Triangle object; add the Triangle to the ArrayList<Triangle>
  - Create TriangleList object with parameters name and ArrayList<Triangle>
  - Print the TriangleList
  - Print summary for TriangleList

Iterators

- An *iterator* is an object that allows you to process a collection of items one at a time (lets you step through each item in turn and process it as needed)
- An iterator object has a `hasNext` method that returns true if there is at least one more item to process
- The `next` method returns the next item
- Iterator objects are defined using the `Iterator` interface (see text for details)

Iterators

- Several classes in the Java standard class library are iterators
- The `Scanner` class is an iterator which facilitates scanning/reading from `System.in`, a file, or a `String`
  - the `hasNext` method returns true if there is more data to be scanned (`Scanner` class also has variations for specific data types, such as `hasNextInt, hasNextDouble, ...`)
  - the `next` method returns the next scanned token as a string (also variations for specific data types, such as `nextInt, nextDouble, ...`)

Summary
Conditionals and Loops

- You should now be familiar with:
  - flow of control: sequence, selection, iteration
  - boolean expressions
  - equality, relational, and logical operators
  - selection: if statement (with optional else or else if)
  - iteration: while statement (a.k.a. while loop)
  - the ArrayList class
  - comparing data
  - indentation and block statements revisited
  - more details on if statements and while loops
  - file input using the File and Scanner classes
  - iterators