

# 5. 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
  - selection: if and if-else statements
  - iteration: while statement (a.k.a. while loop)
  - equality, relational, and logical operators
  - block statements
  - comparing objects
  - nested while loops
  - the ArrayList class
  - 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 through 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); that is, 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

- When we read source code, the sequence, selection, and iteration is relative to the method we're reading
  - Example: In the main method, we may have sequence, selection, and iteration. If one of the statements invokes/calls a method, then we 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 this method
- You can use the debugger to follow the detailed flow of control (see examples in later slides)

# Boolean Expressions

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

`num1 > num2 + 5`

- A boolean variable can be assigned the result of a boolean expression:
  - 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 (recall if and if-else statements with simple boolean expressions were introduced in 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 let's look at more complex boolean expressions

# Operators

- Equality and Relational Operators (review):
  - Must have compatible operands (only numeric types can be used with ordering operators:  $<$ ,  $>$ ,  $<=$ ,  $>=$ ); evaluate to **true** or **false**
  - Have lower precedence than arithmetic operators

Operator	Meaning
<code>==</code>	equal to
<code>!=</code>	not equal to
<code>&lt;</code>	less than
<code>&gt;</code>	greater than
<code>&lt;=</code>	less than or equal to
<code>&gt;=</code>	greater than or equal to

# Operators

- Logical operators - have boolean operands, evaluate to a boolean result (**true** or **false**), logical operators (except unary !) have lower precedence than 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 @ symbol **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 true, the other operand is not evaluated
  - If the first operand of the && is false, then 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 / if-else* Statements

- 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 / if-else* Statements

- 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 / if-else* Statements

- 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 / if-else* Statements

```
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* / *if-else* Statements

- An *if* statement can be 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  
readability

```
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*** Loop

- Suppose we want a block of statements to execute as long as a certain condition is true
- A ***while*** statement will repeat a block of code until its condition is no longer true
- The debugger is a useful tool when using loops
- Example: print all numbers from 1 to 10

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

[Count1.java](#)

# *while* Loop

- Example: [NumbersSet.java](#)
  - Return a String that includes all even numbers between two specified values inclusive.
    - Begin with count set to the lower number.
    - While count is less than or equal to the higher number...
      - Add count to the output string if it is divisible by 2
      - Increment count
    - Return the output string
  - Return a String that includes all divisors between two specified values.

# java.util.ArrayList

- The ArrayList class holds a set of objects of type Object (so any reference type is okay)
- Includes operations to add and remove elements, find the index of an element, determine if the list is empty, and determine the number of items in the list
- Your class should include an import statement

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

- Then in a method declare an instance

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



# ArrayList

- You can (and should) specify what type of objects the list will hold using a generic type:
- The ArrayList names only holds objects of type String:

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

- The ArrayList titles only holds 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

- Recall that characters (`char`) correspond to numbers
  - Letters A through Z: numerical values 65 to 90
  - Letters a through z: numerical 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 Data

- 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 Data

- Instead, use the ***equals*** method to compare objects.
  - Returns a boolean representing whether the objects are equal as defined in the class.
  - You can find out how the equals method works by consulting the Java API for the class.
- 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 Data

- The ***compareTo*** method is also available to some classes, but returns an int value

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

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

# Comparing Data

- The ***compareTo*** method compares Strings based on the value of its characters.
- 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 Data

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

```
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 Data

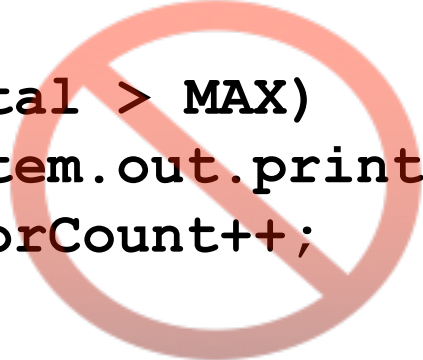
- The String class has the ***equalsIgnoreCase*** and ***compareToIgnoreCase*** methods

```
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 the Java language 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, the increment will occur whether the condition is true or not

# ***Block* Statements**

- Several statements can be grouped together into a ***block*** statement delimited by braces
- 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

- The block of statements executed as a result of an *if* statement or *else* clause could contain other *if* and/or *if-else* statements
- These are called *nested if statements*
- An *else* clause is matched to the last unmatched *if* (no matter what the indentation implies)
- Braces can be used to specify the *if* statement to which an *else* clause belongs
- See [Taxes.java](#)

# Infinite Loops

- A statement in the body of a *while* loop eventually must make the loop condition false
- Otherwise, we have an *infinite loop*, which will execute until the program is interrupted from outside the loop (usually by the user)
- 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 ("Control-C" in DOS window or "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
- That is, the body of a loop can contain another loop
- For each iteration of the outer loop, the inner 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”.
- 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 move to 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
- In chapter 6, you will see how to use a break statement when writing a switch statement

[BreakWhileExample.java](#)

[ContinueWhileExample.java](#)

# Reading from a File

- Example: Read in a lines of text from a file; print the lines in reverse order; query the user to do again.
  - Strategy:
    - Use an outer loop to read file name from user
    - Use an inner loop to read lines from file and store 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?
- [ReverseLinesReadFromFile.java](#)  
Also, see [ReverseWordsFromFile.java](#)  
[ReadItemsFromFile.java](#)

# Reading from a File

## Points to Remember

- Import statements:

```
import java.io.File;  
import java.io.IOException;
```

- Include throws clause with main

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

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

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

- Read the input using: `scanFile.nextLine()`
- Close Scanner object: `scanFile.close();`

[TriangleListApp.java](#)

# Reading from a File

- Example: Read in file name from user; read in name for Triangle List from the file; read in triangle data from a file and ; create print the lines in reverse order; query the user to do again.
- Strategy:
  - Read file name from user; read list name from file
  - Use a loop to read triangle data, create 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

[TriangleListApp.java](#)

# 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, which is discussed further in Chapter 6

# Iterators

- Several classes in the Java standard class library are iterators
- The `Scanner` class is an iterator
  - the `hasNext` method returns true if there is more data to be scanned
  - the `next` method returns the next scanned token as a string
- The `Scanner` class also has variations on the `hasNext` method for specific data types (such as `hasNextInt`)

# Iterators

- The fact that a `Scanner` is an iterator is particularly helpful when reading input from a file
- Suppose we wanted to read and process a list of URLs stored in a file
- One scanner can be set up to read each line of the input until the end of the file is encountered
- Another scanner can be set up for each URL to process each part of the path
- See page 242-243 in the book on your own



# Summary

## Conditionals and Loops

- You should now be familiar with:
  - flow of control: sequence, selection, iteration
  - boolean expressions
  - selection: if and if-else statements
  - iteration: while loops
  - equality, relational, and logical operators
  - block statements
  - comparing objects
  - nested while loops
  - the ArrayList class
  - file input using the File and Scanner classes
  - Iterators