Data Comparisons and Switch

- Data Comparisons
- Switch
- Reading for this class: L&L 5.3, 6.1-6.2

Comparing Data

- When comparing data using boolean expressions, it's important to understand the nuances of certain data types
- Let's examine some key situations:
  - Comparing double/float values for equality
  - Comparing characters
  - Comparing strings (alphabetical order)

Comparing Float Values

- You should rarely use the equality operator (`==`) when comparing two floating point values (`float` or `double`)
- Two floating point values are equal only if their underlying binary representations match exactly
- Computations often result in slight differences that may be irrelevant
- In many situations, you might consider two floating point numbers to be "close enough" even if they aren't exactly equal

Comparing Float Values

- Your tolerance for equality could be set as follows:

```
final double TOLERANCE = 0.000001;
```

- To determine the equality of two doubles or floats, use the following technique:

```
if (Math.abs(f1 - f2) < TOLERANCE)
   System.out.println ("Essentially equal");
```

- If the absolute value of the difference between the two double/float values is less than the tolerance, they are considered to be equal, the if condition is true, and the print statement will execute

Comparing Characters

- As we've discussed, Java character data is based on the Unicode character set
- Unicode assigns a particular numeric value to each character and this creates an ordering of characters
- We can use relational operators on character data based on this ordering
- For example, the character `'A'` is less than the character `'J'` because it comes before it in the Unicode character set
- L&L Appendix C provides an overview of Unicode

Comparing Characters

- In Unicode, the digit characters (0-9) are contiguous and in order of their numerical value

- Likewise, the uppercase letters (A-Z) and lowercase letters (a-z) are contiguous and in alphabetical order

Characters	Unicode Values
0 – 9	48 through 57
A – Z	65 through 90
a – z	97 through 122

Comparing Characters

- Therefore, if we want to base a decision in our program on whether a character is a digit or not, we can use the following code:

```
if (character >= ‘0’ && character <= ‘9’)
   System.out.println (“Yes, it’s a digit!”);
```

- We can also check for a valid upper case alphabetic character as follows:

```
if (character >= ‘A’ && character <= ‘Z’)
   System.out.println (“It’s a capital letter!”);
```

Comparing Strings

- Remember that in Java a string is an object
- We cannot use the == operator to determine if the values of two strings are identical (character by character)
- The `equals` method can be called with strings to determine if two strings contain exactly the same characters in the same order
- The `equals` method returns a boolean result

```
if (name1.equals(name2))
   System.out.println ("Same name");
```

Comparing Strings

- We cannot use the relational operators to compare strings
- The `String` class contains a method called `compareTo` to determine if one string comes before another
- A call to `name1.compareTo(name2)`
  - returns zero if `name1` and `name2` are equal (contain the same characters)
  - returns a negative value if `name1` is less than `name2`
  - returns a positive value if `name1` is greater than `name2`

Comparing Strings

```
if (name1.compareTo(name2) < 0)
   System.out.println (name1 + "comes first");
else
   if (name1.compareTo(name2) == 0)
      System.out.println ("Same name");
   else
      System.out.println (name2 + "comes first");
```

- Because comparing characters and strings is based on a character set, it is called a *lexicographic ordering*

Lexicographic Ordering

- Lexicographic ordering is not strictly alphabetical with mixed uppercase and lowercase characters
- For example, the string `"Great"` comes before the string `"fantastic"` because in Unicode the uppercase letters have lower values than the lowercase letters. Therefore, ‘G’ is less than ‘f’
- Also, short strings come before longer strings with the same prefix (lexicographically)
- Therefore `"book"` comes before `"bookcase"`

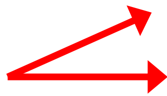
The switch Statement

- The *switch statement* provides another way to decide which statement to execute next
- The `switch` statement evaluates an integral expression (int or char only), then attempts to match the result to one of several possible *cases*
- Each case contains a value and a statement list
- The flow of control transfers to the statement list associated with the first case value that matches

The switch Statement

- The general syntax of a `switch` statement is:

switch and **case** are reserved words



```
switch ( expression )
{
   case value1 :
      statement-list1
   case value2 :
      statement-list2
   case value3 :
      statement-list3
   case   ...

}
```

If *expression* matches *value2*, control jumps to here

The switch Statement

- Often a *break statement* is used as the last statement in each case's statement list
- A `break` statement causes control to transfer to the end of the `switch` statement
- If a `break` statement is not used, the flow of control will continue into the next case
- Sometimes this may be appropriate, but often we only want to execute the statements associated with one case

The switch Statement

- An example of a switch statement:

```
switch (option)
{
   case 'A':
      aCount++;
      break;
   case 'B':
      bCount++;
      break;
   case 'C':
      cCount++;
      break;
}
```

The switch Statement

- A `switch` statement can have an optional *default case*
- The default case has no associated value and simply uses the reserved word `default`
- If there is a default case and no other value matches, control will transfer to the default statement list
- If there is no default case and no other value matches, control falls through to the statement after the switch without executing any statements

The switch Statement

- An example of a switch statement using default:

```
switch (option)
{
   case 'A':
      aCount++;
      break;
   case 'B':
      bCount++;
      break;
   default:
      errorCount++;
      break;
}
```