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 peculiarities of certain data types
- Let's examine some key situations:
  - Comparing double/float values for equality
  - Comparing characters
  - Comparing strings (alphabetical order)

Comparing Decimals

- You should rarely use the equality operator (==) when comparing two decimals (`float` or `double`)
- They are equal **only** if their underlying binary representations match exactly
- Two decimals may be "close enough" even if they aren't exactly equal, yet computations often result in slight differences that may be irrelevant

How To Compare Decimals

1. Decide on a "maximum tolerable inequality":

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

1. To determine the equality of two decimals, use the following technique:

```
double d1, d2;
...
if (Math.abs(f1 - f2) < TOLERANCE){
   System.out.println ("Essentially equal");
}
```

1. If **the absolute value of the difference** is less than the tolerance, the *if-condition* will be true, and the print statement will execute. (The idea here is "equal enough")

Comparing Characters

- As we've discussed, Java character data is based on the Unicode character set (*See L&L Appendix C*)
- Each character has a particular numeric value, which creates an ordering of characters
- Thus, we can use relational operators on character data
- For example, `'A' < 'J' == true` because 'A' has the smaller numeric value in the Unicode set

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

- Notice that *uppercase precedes lowercase*!

Comparing Characters

- Therefore, we can determine whether a character is a digit, a letter, etc.

```
if (character >= '0' && character <= '9'){
  System.out.println ("Yes, it's a digit!");
}
else if ((character >= 'A' && character <= 'Z') ||
      (character >= 'a' && character <= 'z')){
  System.out.println ("It's a letter!");
}
else{
  System.out.println("Something else entirely!");
}
```

Code to Remember

- public class … { …. }
- public static void main (String[] args){…}
- System.out.println(…);
- System.out.print(…);
- Scanner scan = new Scanner(System.in); (*and "import java.util.Scanner;" at top*)
- Math.abs(…) Math.pow(…, …)
- (int) (Math.random() * MAX)
- Random random = new Random(); (*and "import java.util.Random;" at top*)
- random.nextInt() random.nextDouble();

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 whether this is the case.
- 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");
   }
}
```

- The comparison is based on characters' **numeric** values, so it is called a *lexicographic ordering*

Lexicographic Ordering

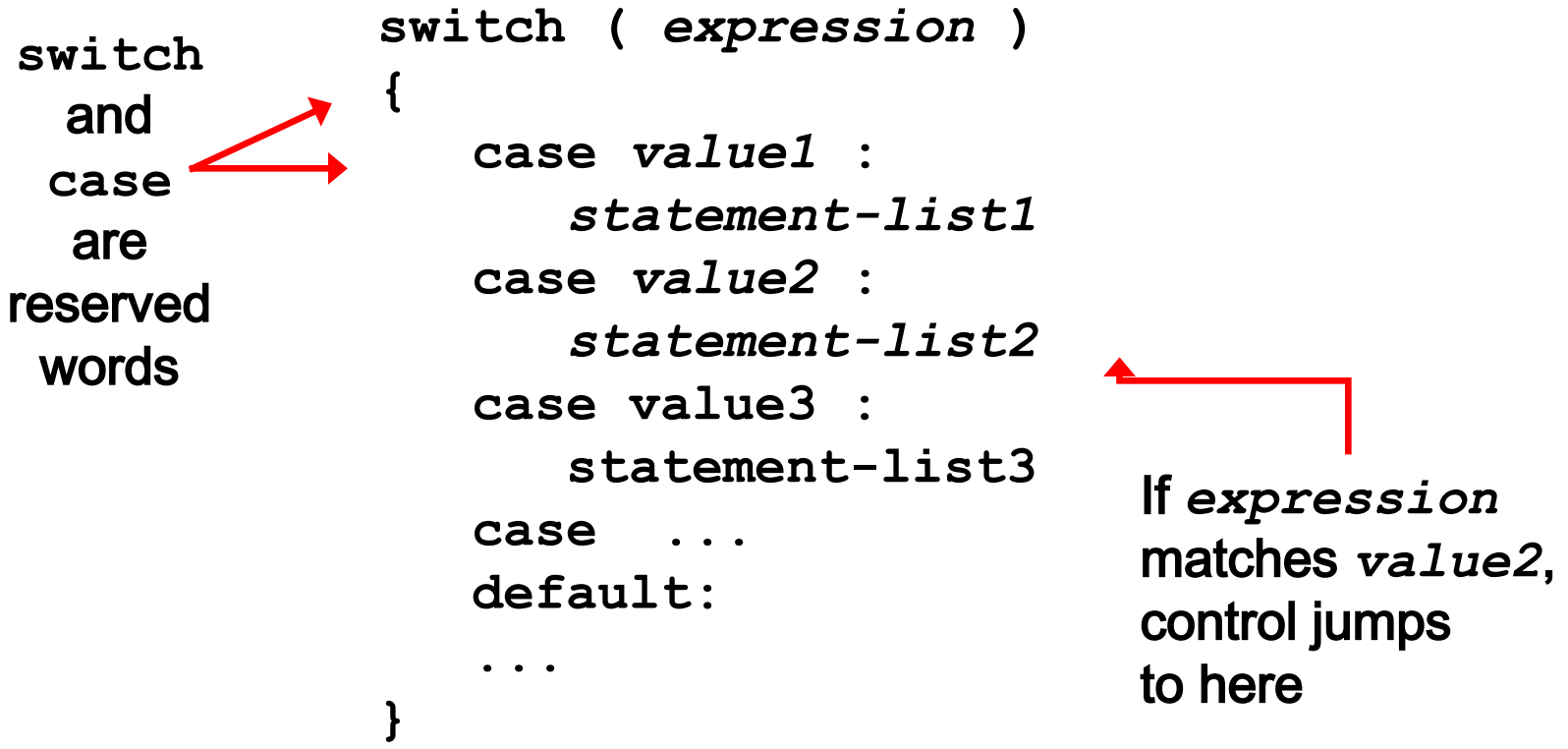
- Lexicographic ordering is not strictly alphabetical
- For example, the string `"Great"` comes before the string `"fantastic"`. In Unicode, the uppercase letters have lower values than lowercase, so ‘G’ is technically less than ‘f’
- Also, short strings come before longer strings with the same prefix
- `"book"` comes before `"bookcase"`, but `"Bookcase"` comes before **both**!

The switch Statement

- The *switch statement* matches program statements to specific `int` or `char` values
- The `switch` statement evaluates an integral value, 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 first case value that matches. We "switch" on a particular value

The switch Statement

- The general syntax of a `switch` statement:



An example switch statement

```
System.out.print("You will belong to ");
switch (studentId % 4)
{
 case 0:
   System.out.println("Gryffindor...but don't let " +
     "it go to your head!");
   break;
 case 1:
   System.out.println("Ravenclaw...nerd!");
   break;
 case 2:
   System.out.println("Hufflepuff...nah, too easy!");
   break;
 default:
   System.out.println("Slytherin...NOW we're talking!");
   break;
}
```

The switch Statement

- The `break` statement causes us to leave the `switch` statement. Otherwise, the flow of control would continue into the next case
- Sometimes this may be appropriate, but we usually only want to go to one case
- The `default` case is where we go when no other case matches the switch value. If there is no `default`, then we just exit the `switch` statement without executing anything.
- Whether you need a `default` case depends on what your program is doing at that time.

An example without breaks

```
switch ((int)(age / 10.0))
{
 case 0:
 case 1:
   System.out.println("Gather ye rosebuds while ye may");
   break;
 case 2:
   System.out.println("Enjoy the bloom of youth");
   break;
 case 3:
 case 4:
 case 5:
   System.out.println("Ahh, the wisdom of age!");
   break;
 default:
   System.out.println("So...any stories about Fortran?");
   break;
}
```