Today’s Topics

• Comments and/or Questions?
• comparing data
• other useful String class methods
• Logical operators
• Start loops
Comparing data

• Comparing primitive type variable values is different than comparing variables of non-primitive types.
• Recall the primitive types: char, byte, short, int, long, float, double, boolean.
• String, Integer, Double, and many others which we have yet to see are classes which can be types for variables. These kinds of variables are not of the primitive types. These kids of variables are also called objects or object references.
Comparing data

• Example chars
  ```
  char some_char = 'a', first_letter = 'E', a_digit = '7';
  // note: a char literal is always in single quotes.
  ```

• Examples of how to compare chars for equality
  ```
  (some_char == 'a')
  (some_char == 'b')
  (some_char == a_digit)
  ```

• Also could compare less than, greater than, etc. for chars

• What would that mean?
Comparing data

• Example ints (same for byte, long, short)
  ```java
  int height = 65, length = 4, width = 6;
  ```

• Examples of how to compare ints
  ```java
  (height < 70)
  (width == length)
  ```
Comparing data

• Example booleans
  
  ```java
  boolean done_yet = false, old_enough = true;
  ```

• Examples of how to compare booleans
  
  (done_yet == true)
  
  (done_yet)
  
  (old_enough)
  
  (!old_enough)
  
  (done_yet == old_enough)
Comparing data

• Example doubles (same for floats)
  double weight = 5.6;
  // note: a decimal number literal is assumed by Java to be
  // a double (not a float) 5.6f is how to create a float literal

• Examples of how to compare doubles
  (weight > 5.1)
  (weight == 5.6) // problem because doubles (and floats)
  // are not stored exactly (depending on the value)
  // so it is dangerous to check for equality – don't do it.
Comparing data

• Example Strings
  
  String name = “Mike”;
  
  // note: a String literal is enclosed in double quotes

• Example of how **not** to compare Strings
  
  (name == “Mike”) /* sometimes works sometimes doesn't because since name is an object (not a variable of a primitive type), when compared using the comparison operators, Java compares the memory locations of name and “Mike”, not the values. */
Comparing data

• Example Strings
  String name = “Mike”;

• Example of how to compare Strings (use method(s) in the String class for this purpose)
  (name.equals(“Mike”)) // or
  (name.equalsIgnoreCase(“mike”))

• Here we are calling methods in the String class. If you had to guess, what is the type of the value that is returned by these methods?
Comparing data

• Another way to compare Strings is to use the compareTo method of the String class.
• The calling String (the one to the left of the . ) is compared lexicographically to the String that is passed in as a parameter. If the calling String is less than the parameter String the result is a negative number. If they are equal, the result is 0. Otherwise the result is a positive number.
• What is the type of the value that is returned by this method?
other String class methods

- boolean equals(String s)
- boolean equalsIgnoreCase(String s)
- char charAt(int index)
- int length()
- String toUpperCase()
- String toLowerCase()

All of these methods are called by using a String object followed by the . (dot operator) then the name of the method (and any necessary parameters within the parens).
calling methods in String

• Notice how we call methods in the String class:
  – first we need something of type String (an object/variable OR a String literal)
  – then we use the . (dot operator)
  – then we specify the name of the method
  – then in parentheses we put whatever parameters are required
  – we also pay attention to what type is returned by the method

• examples:
  String name = “Eckmann”;
  int len, len2;
  len = name.length();
  len2 = “CS106”.length();
other String class methods

- Examples (let's put some of this code in a program):
  - String name = “Joe”, name2, lastname;
  - char a_char;
  - if (name.length() > 5)
  - a_char = name.charAt(2);
  - name2 = name.toUpperCase();

if (lastname.compareTo(“Jones”) < 0)
  System.out.println(lastname + “ comes before Jones”);
else
  if (lastname.compareTo(“Jones”) > 0)
    System.out.println(“Jones comes before ” + lastname);
  else
    System.out.println(“Jones is the same as ” + lastname);
Logical operators

- Not      !
- And      & or &&
- Or       | or ||

- The double ones (&& and ||) are “short circuit” operators --- if left operand is sufficient to determine truth or falsity, then the right operand is not evaluated.
- The single ones (& and |) always evaluate both operands.
- This really only matters if you have side-effects,
- e.g. Like a ++ or --
Logical operators

- Precedence first !, then && then ||.
- Parentheses are still performed first. Should be used to force the order that the programmer desires.

- Not operator, !, takes one operand to its right.
- And operator, &&, works on two operands
- Or operator, ||, works on two operands
Logical operators

NOT:
!true = false, !false = true

AND:
true && true = true
true && false = false
false && true = false
false && false = false

OR:
true || true = true
true || false = true
false || true = true
false || false = false
Logical operators

• Examples:

boolean end_of_file = false;
// ...
if (! end_of_file)
    // ...

if (!(age >=21))
    // ...

Michael Eckmann - Skidmore
College - CS 106 - Fall 2015
Logical operators

if (!(age >= 21))
   // ...

   // how else might we write the above?

if ((age <= 12) || (age >= 65))
   // get a discount at the movies...

if (you_are_rich && you_are_good_looking)
   System.out.println("You got it made.");
   // what types must you_are_rich and you_are_good_looking, be?
Logical operators

if ((age >=21) && (++count > 10))
   // ...

here if age is not >= 21 then regardless of what's on the right hand side of the &&, the whole thing will be false (because false AND anything = false), because && is the “short circuit” format of the AND operator, (++count > 10) will not be evaluated.

if ((age >=21) & (++count > 10))
   // ...

here, even if age is not >= 21 the right hand side WILL be evaluated
What's the difference in the result?
Logical operators

&& (short circuit AND) vs. & (non-short circuit AND)

```c
if ((age >=21) && (++count > 10))
    // ...
if ((age >=21) & (++count > 10))
    // ...
```

The difference is, that 1 will be added to count (and compared to 10) regardless of whether or not age is >= 21 in the second if, but in the first if, 1 will be added to count (and compared to 10) only if age is indeed >=21.
Logical operators

if ((height >= 72) || (countPeople() <= 100))
    // ...

Here if height is >= 72, then the whole condition is guaranteed to be true (because true OR anything = true) and the || causes the right hand side NOT to be evaluated. Therefore, the method countPeople() would not be called.

if ((height >= 72) | (countPeople() <= 100))
    // ...

Here even if height is >=72, the righthand side WILL be evaluated because of the use of the non-short circuit OR |.
Logical operators

|| (short circuit OR) vs. | (non-short circuit OR)

```java
if ((height >= 72) || (countPeople() <= 100))
  // ...
if ((height >= 72) | (countPeople() <= 100))
  // ...
```

The difference is, that countPeople() method will ALWAYS be called in the second if but won't be called in the first if if the left hand side evaluates to true.
Logical operators

if ((height >= 72) || (countPeople() <= 100))
   // ...

Would countPeople() be called (and it's returned value compared to 100) if height happens to be < 72?
Why or why not?
Logical operators

if ((height >= 72) || (countPeople() <= 100))
    // ...

Would countPeople() be called (and it's returned value compared to 100) if height happens to be < 72?

Why or why not?

Yes it would because the LHS being false doesn't determine the outcome of the whole condition, even though we are using the || short-circuit version of OR.
Logical operators

if ((age >= 21) && (++count > 10))
   // ...

Would ++count > 10 occur if age is >= 21?
Why or why not?
Logical operators

if ((age >= 21) && (++count > 10))
    // ... 

Would ++count > 10 occur if age is >= 21?
Why or why not?

Yes it would because the LHS being true doesn't determine the outcome of the whole condition, even though we are using the && short-circuit version of AND.
while loops

- While loops are used to repeat lines of code until some condition is met.

e.g.

```java
while (condition) {
    // statements to do while condition is true.
}
```

- Condition is evaluated and if true the code within the curly braces executes. Then condition is tested again and if true the code within the curly braces executes. This continues until the condition is false (or a break; statement is hit.)

Michael Eckmann - Skidmore College - CS 106 - Fall 2015
while loops

• It's extremely important for you to know exactly what code is executed in what order for you to be able to follow what a program is doing. Let's follow the code below line by line in order. (assume that it is inside a main method of a class.)

```java
int x = 1;
System.out.println("Just before the loop");
while (x < 4)
{
    System.out.print("*");
    x++;
}
System.out.println("\nJust got out of the loop");
```
while loops

```java
int total = 0;
while (total < 100)
{
    total = total + 10;
    System.out.println("total = "+ total);
}
System.out.println("finished printing totals");
```

// What would this code do?
while loops

- Sentinel controlled loops vs. counter controlled loops.
- Counter controlled loops are those loops that usually execute a set number of times every time.
- A sentinel is a value that is used to terminate a loop. For instance, if you wanted a user to enter in some numbers (but you didn't know how many s/he was going to enter) to be averaged (and assuming that they are all supposed to be positive) we can ask the user to enter in -1 after s/he enters all the numbers so we know s/he's finished.
- Let's see an example.
while loops

```java
int input_num=0, count = 0;
double total = 0.0;
while (input_num != -1)
{
    input_str = JOptionPane("Enter a number to be averaged");
    input_num = Integer.parseInt(input_str);
    total += input_num;
    count++;
}
System.out.println("average = " + (total / count));
```

// any problems with this code???? Hint: yes there's a logic error.
while loops

• Sentinel controlled loops vs. counter controlled loops.
• Counter controlled loops are those loops that usually execute a set number of times every time.
• As a programmer what value makes sense to choose as the sentinel is specific to the situation.
• Could anyone characterize what would be a bad value to choose as the sentinel?
while loops

• Order of execution of a while loop.
  • First the condition is tested.
  – If it is true the statements within the curly braces execute
  – Then condition is tested again
  – If it is true the statements within the curly braces execute again
  – Then condition is tested again
  – ...
  – This continues until the condition is false when tested. At that time, program control continues with the code after the right curly brace.
• Note: It IS possible that even on the first test of the condition, that it may be false. If it is, the statements within the loop don't execute at all.
while loops

• What happens to loops that do not alter the value of a variable in the condition?
while loops

• What happens to loops that do not alter the value of variable(s) in the condition?
• If that condition is true the first time, it will stay true forever and we will have an infinite loop.
• e.g.
  
  int x = 2;
  while (x > 0)
  {
    System.out.println("x = " + x);
  
  }
while loops

• Any other ways to get an infinite loop?
while loops

• Would this be infinite?

```java
int x = 2;
while (x > 0)
{
    System.out.println("x = " + x);
    x++;
}
```