Java Class 14

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 floating point values for equality
- Comparing characters
- Comparing strings (alphabetical order)
- Comparing object vs. comparing object references

Comparing Floating Point Values

You should rarely (never) use the equality operator (==) when comparing two floating point values (`float` or `double`) because two floating point values are equal only if their underlying binary representations match exactly

To determine the equality of two floating point numbers, use the following technique:

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

If the difference between the two floating point values is less than the tolerance, they are considered to be equal

The tolerance could be set to any appropriate level, such as 0.000001

Comparing Characters

As we've discussed, Java character data is based on the Unicode character set

Unicode establishes a particular numeric value for each character, and therefore an ordering

We can use relational operators on character data based on this ordering

For example, the character `'+'` is less than the character `'J'` because it comes before it in the Unicode character set

Appendix C provides an overview of Unicode

Comparing Characters

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

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

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

Comparing Strings

Remember that in Java a character String is an object

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 because they are not primitives

The `String` class contains the `compareTo` method for determining 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

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

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

Lexicographic Ordering

Lexicographic ordering is not strictly alphabetical when uppercase and lowercase characters are mixed

For example, the string `"Great"` comes before the string `"fantastic"` because all of the uppercase letters come before all of the lowercase letters in Unicode

Also, short strings come before longer strings with the same prefix (lexicographically)

Therefore `"book"` comes before `"bookcase"`

Comparing Object References

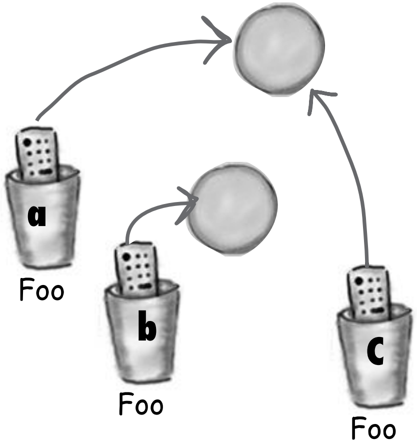
To see if two object references are the same (which means they refer to the same object on the heap), use the equality operator (==)

```
Foo a = new Foo();
Foo b = new Foo();
Foo c = a;

if (a == b)      false

if (a == c)      true

if (b == c)      false
```



Repetition Statements

Repetition statements allow us to execute a statement multiple times

Often they are referred to as ***loops***

Like conditional statements, they are controlled by boolean expressions

Java has three kinds of repetition statements: `while`, `do`, and `for` loops

The while Statement

A ***while statement*** has the following syntax:

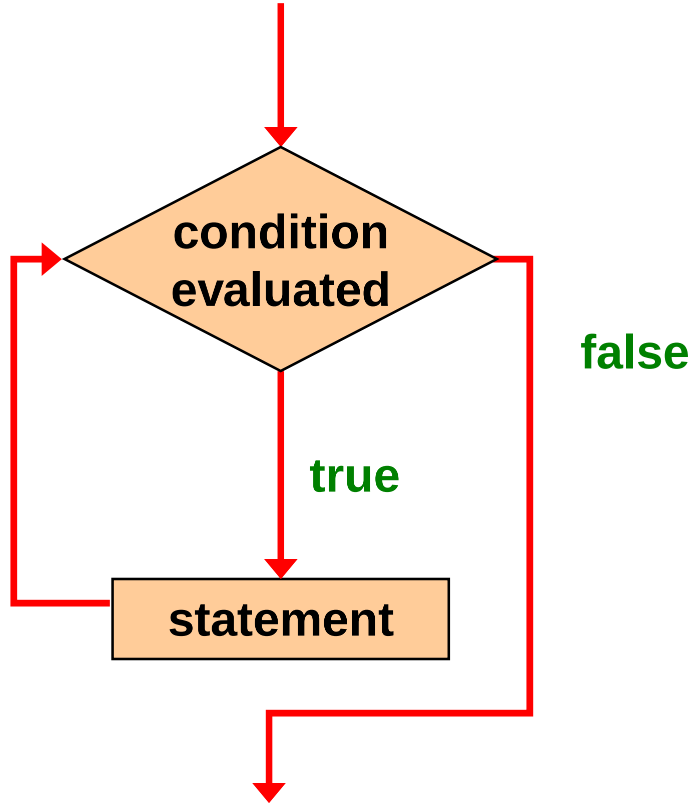
```
while ( condition )
   statement;
```

If the **condition** is true, the **statement** is executed

Then the condition is evaluated again, and if it is still true, the statement is executed again

The statement is executed repeatedly until the condition becomes false

Logic of a while Loop



The while Statement

An example of a `while` statement:

```
int count = 1;
while (count <= 5)
{
   System.out.println (count);
   count++;
}
```

If the condition of a `while` loop is `false` initially, the statement is never executed

Therefore, the body of a `while` loop will execute zero or more times

Sentinel Values

Let's look at some examples of loop processing

A loop can be used to maintain a *running sum*

A ***sentinel value*** is a special input value that represents the end of input

```java
//********************************************************************
//  Average.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a while loop, a sentinel value, and a
//  running sum.
//********************************************************************

import java.text.DecimalFormat;
import java.util.Scanner;

public class Average
{
   //-----------------------------------------------------------------
   //  Computes the average of a set of values entered by the user.
   //  The running sum is printed as the numbers are entered.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      int sum = 0, value, count = 0;
      double average;

      Scanner scan = new Scanner (System.in);

      System.out.print ("Enter an integer (0 to quit): ");
      value = scan.nextInt();
```

continue

continue

```
   while (value != 0)  // sentinel value of 0 to terminate loop
   {
      count++;

      sum += value;
      System.out.println ("The sum so far is " + sum);

      System.out.print ("Enter an integer (0 to quit): ");
      value = scan.nextInt();
   }
```

continue

continue

```
      System.out.println ();

      if (count == 0)
         System.out.println ("No values were entered.");
      else
      {
         average = (double)sum / count;

         DecimalFormat fmt = new DecimalFormat ("0.###");
         System.out.println ("The average is " + fmt.format(average));
      }
   }
}
```

continue

```
      System.ou

      if (count
         System
      else
      {
         average

         Decima
         System                                        at(average));
      }
   }
}
```

Sample Run

```
Enter an integer (0 to quit): 25
The sum so far is 25
Enter an integer (0 to quit): 164
The sum so far is 189
Enter an integer (0 to quit): -14
The sum so far is 175
Enter an integer (0 to quit): 84
The sum so far is 259
Enter an integer (0 to quit): 12
The sum so far is 271
Enter an integer (0 to quit): -35
The sum so far is 236
Enter an integer (0 to quit): 0

The average is 39.333
```

Input Validation

A loop can also be used for *input validation*, making a program more *robust*

It's generally a good idea to verify that input is valid (in whatever sense) when possible

```java
//********************************************************************
//  WinPercentage.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a while loop for input validation.
//********************************************************************

import java.text.NumberFormat;
import java.util.Scanner;

public class WinPercentage
{
   //-----------------------------------------------------------------
   //  Computes the percentage of games won by a team.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int NUM_GAMES = 12;
      int won;
      double ratio;

      Scanner scan = new Scanner (System.in);

      System.out.print ("Enter the number of games won (0 to "
                        + NUM_GAMES + "): ");
      won = scan.nextInt();
```

continue

continue

```
      while (won < 0 || won > NUM_GAMES)
      {
         System.out.print ("Invalid input. Please reenter: ");
         won = scan.nextInt();
      }

      ratio = (double)won / NUM_GAMES;

      NumberFormat fmt = NumberFormat.getPercentInstance();

      System.out.println ();
      System.out.println ("Winning percentage: " + fmt.format(ratio));
   }
}
```

continue

```
      whil
      {
         Sy
         wo
      }

      ratio = (double)won / NUM_GAMES;

      NumberFormat fmt = NumberFormat.getPercentInstance();

      System.out.println ();
      System.out.println ("Winning percentage: " + fmt.format(ratio));
   }
}
```

Sample Run

```
Enter the number of games won (0 to 12): -5
Invalid input. Please reenter: 13
Invalid input. Please reenter: 7

Winning percentage: 58%
```

Infinite Loops

The body of a `while` loop eventually must make the condition false

If not, it is called an ***infinite loop***, which will execute until the user interrupts the program

This is a common logical error

You should always 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) or until an underflow error occurs

Quick Check

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 * 19 = 190

The ArrayList Class

An `ArrayList` object stores a list of objects, and is often processed using a loop

The `ArrayList` class is part of the `java.util` package

You can reference each object in the list using a numeric index

An `ArrayList` object grows and shrinks as needed, adjusting its capacity as necessary

The ArrayList Class

Index values of an `ArrayList` begin at 0 (not 1):

0	"Bashful"
1	"Sleepy"
2	"Happy"
3	"Dopey"
4	"Doc"

Elements can be inserted and removed

The indexes of the elements adjust accordingly

ArrayList Methods

Some `ArrayList` methods:

```
boolean add(E obj)

void add(int index, E obj)

Object remove(int index)

Object get(int index)

boolean isEmpty()

int size()
```

The ArrayList Class

The type of object stored in the list is established when the `ArrayList` object is created:

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

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

This makes use of Java *generics*, which provide additional type checking at compile time

An `ArrayList` object cannot store primitive types, but that's what wrapper classes are for

```java
//********************************************************************
//  Beatles.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a ArrayList object.
//********************************************************************

import java.util.ArrayList;

public class Beatles
{
   //-----------------------------------------------------------------
   //  Stores and modifies a list of band members.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      ArrayList<String> band = new ArrayList<String>();

      band.add("Paul");
      band.add("Pete");
      band.add("John");
      band.add("George");
```

continue

continue

```
      System.out.println(band);
      int location = band.indexOf("Pete");
      band.remove(location);

      System.out.println(band);
      System.out.println("At index 1: " + band.get(1));
      band.add(2, "Ringo");

      System.out.println("Size of the band: " + band.size());
      int index = 0;
      while (index < band.size())
      {
         System.out.println(band.get(index));
         index++;
      }
   }
}
```

continue

```
      System.out.p
      int locatio
      band.remove(

      System.out.p
      System.out.p                                         1));
      band.add(2,

      System.out.p                                         d.size());
      int index =
      while (index < band.size())
      {
         System.out.println(band.get(index));
         index++;
      }
   }
}
```

Output

```
[Paul, Pete, John, George]
[Paul, John, George]
At index 1: John
Size of the band: 4
Paul
John
Ringo
George
```

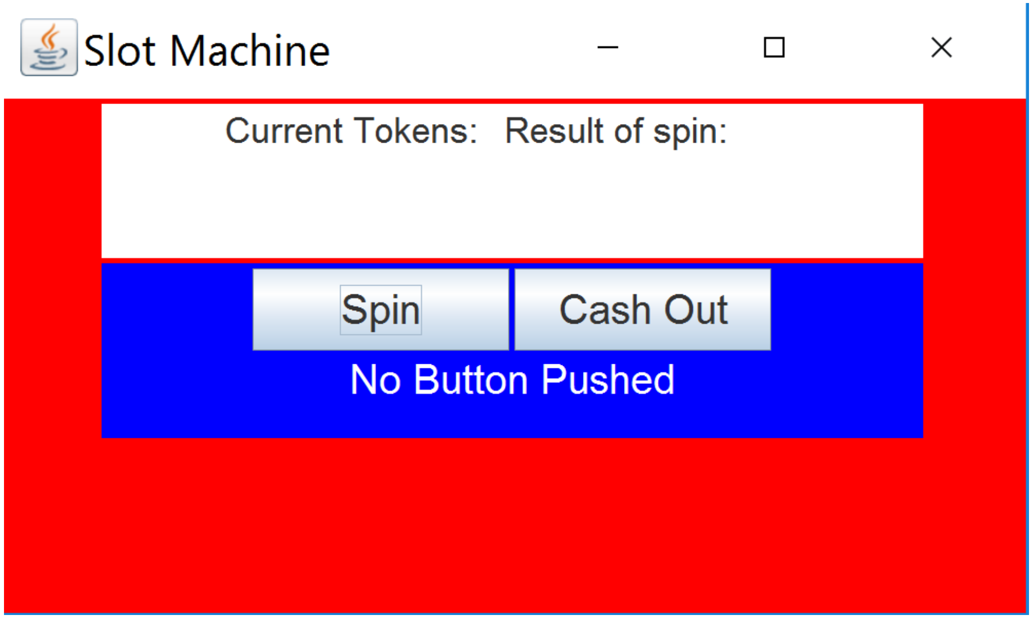
Individual Exercise

Note: this is an individual assignment!

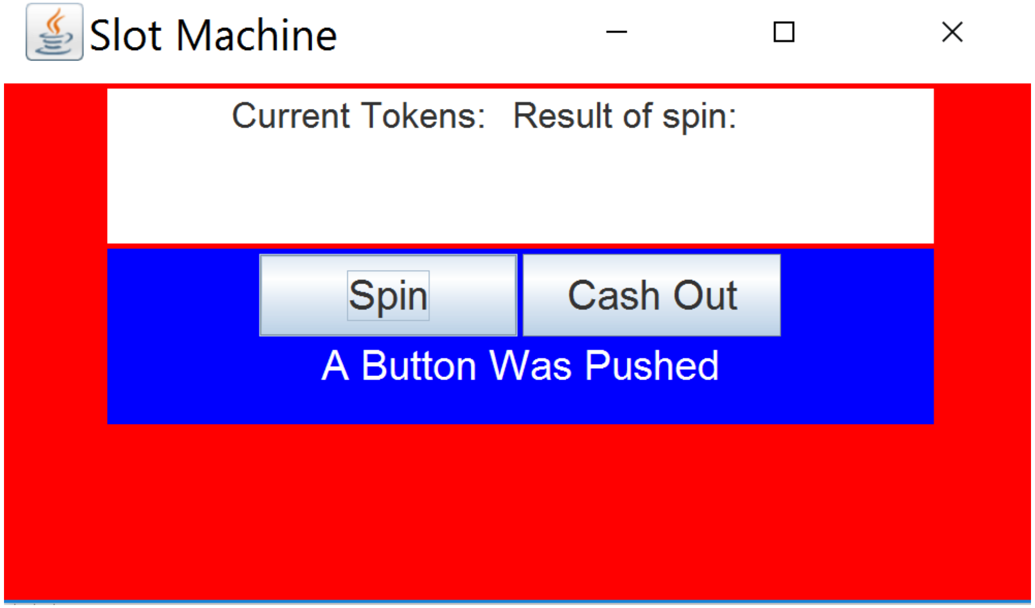
Continue with Programming Assignment #2

On your SlotMachinePane, add two labels to the upper pane for the current number of tokens and the spin result. Add two buttons to the lower pane labeled “Spin” and “Cash Out.” Create an event handler for the two buttons. As a first step, have the event handler print out a message that only indicates a button was pushed.

- *Add more instance variables for 3 labels and 2 buttons*
- *Create label and button objects*
- *Add 2 of the labels to slotsPane and 2 buttons and label to buttonPane*
- *Create the event handler*



No Button Pushed



A Button Was Pushed

Some Useful Code

```
spinLabel = new Label("Spin result:  ");
spinLabel.setFont(Font.font("Helvetcia", FontWeight.BOLD, 40));
spinLabel.setTextFill(Color.BLUE);
```

```
spinButton = new Button("Spin");
spinButton.setPrefSize(250, 80);
spinButton.setFont(Font.font("Arial", FontWeight.NORMAL, 40));
```

Group Exercises

Ex: 5.6

Ex: 5.7

Ex: 5.8

Ex: 5.9

Ex: 5.10

Assignment for Class 15

Review Average, WinPercentage, PalindroneTester, Beatles

Read Chapter 6.1, 6.2, 6.3, 6.4, 6.5