Three Topics in One Lecture!!!

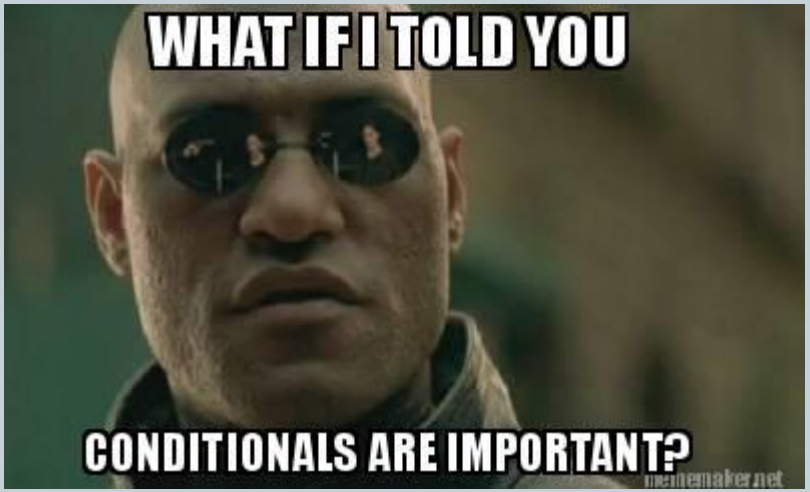
- Conditionals Revisited
- Basic Input and Output
- Programming Style

Conditionals Revisited

Today's Zits comic is one for conditionals afficionados:

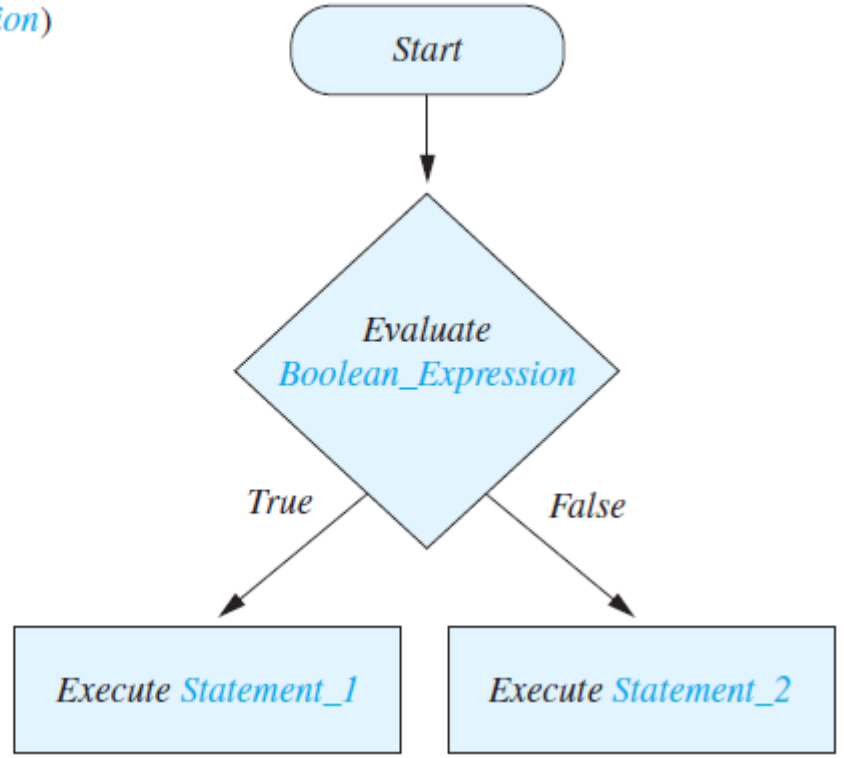
Outline: Conditionals Revisited

- if Statement
- Boolean Expressions
- switch Statement



Meaning of the if Statement

```
if (Boolean_Expression)
    Statement_1
else
    Statement_2
```



Compound Statements

- To include multiple statements in a branch, enclose the statements in braces.

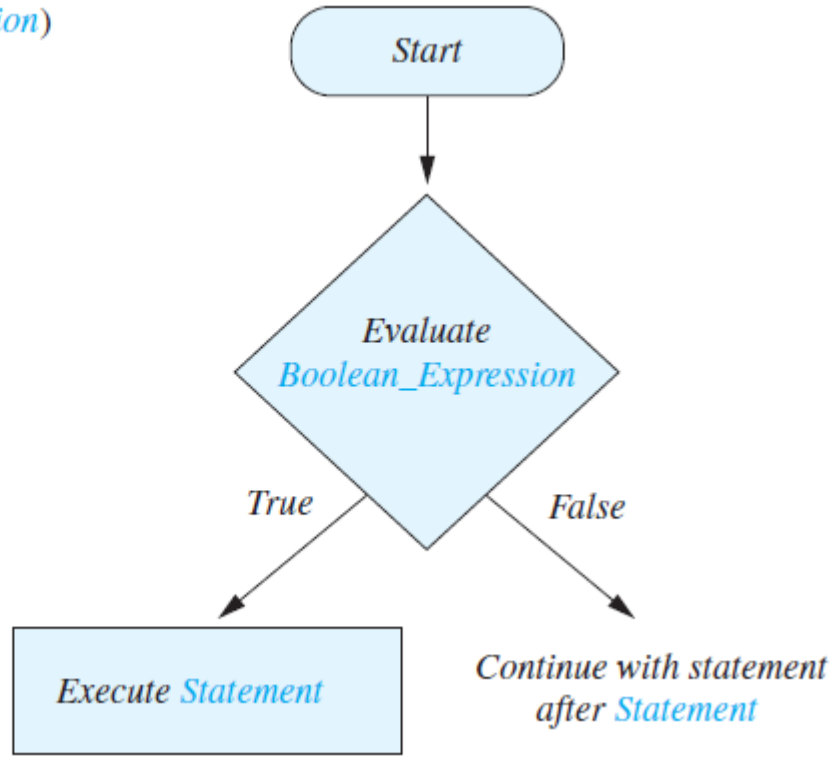
```
if (count < 3)
    {
        total = 0;
        count = 0;
    }
```

- A compound statement can be used wherever a statement can be used.

```
if (total > 10)
{
    sum = sum + total;
    total = 0;
}
```

Omitting the `else` Part

```
if (Boolean_Expression)
    Statement
```



Nested if Statements

- Syntax

```
if (Boolean_Expression_1)
    if (Boolean_Expression_2)
        Statement_1)
    else
        Statement_2)
else
   if (Boolean_Expression_3)
        Statement_3)
    else
        Statement_4);
```

Nested if Statements

- Each `else` is paired with the nearest unmatched `if`.
- **If used properly**, indentation communicates which `if` goes with which `else`.
- Curly braces can be used like parentheses to group statements.

Nested Statements

- Subtly different forms

First Form

```
if (a > b)
{
    if (c > d)
        e = f
}
    else
        g = h;
```

Second Form

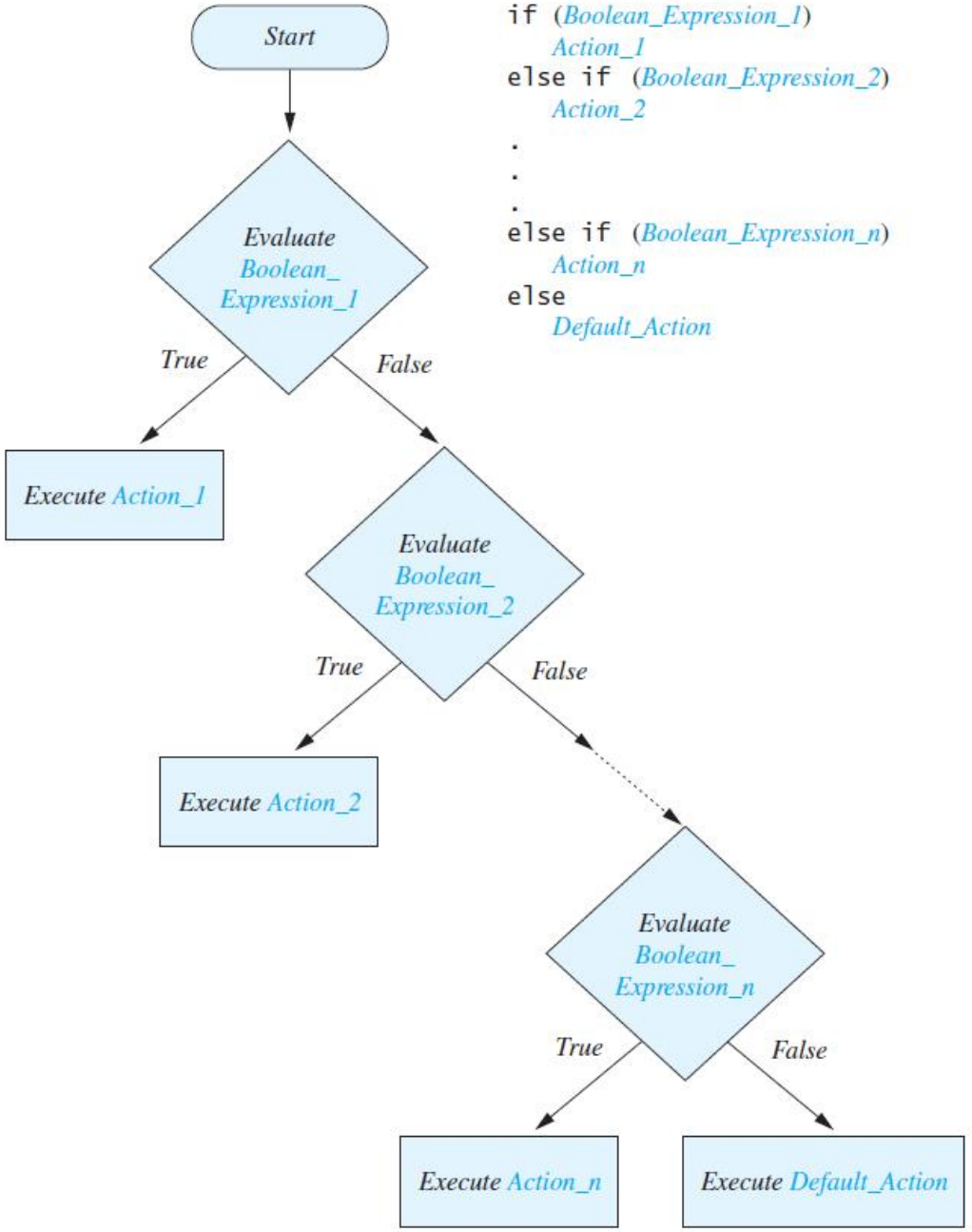
```
if (a > b)
   if (c > d)
       e = f
else
   g = h;

// oops
```

Multibranch `if-else` Statements

```
if (Boolean_Expression_1)
    Statement_1
else if (Boolean_Expression_2)
    Statement_2
else if (Boolean_Expression_3)
    Statement_3
else if …
else
    // Default_Statement
```

Multibranch `if-else` Statements



Multibranch `if-else` Statements

```
if (score >= 90)
    grade = 'A';
else if ((score >= 80) && (score < 90))
    grade = 'B';
else if ((score >= 70) && (score < 80))
    grade = 'C';
else if ((score >= 60) && (score < 70))
    grade = 'D';
else
    grade = 'F';
```

Java Comparison Operators

Math Notation	Name	Java Notation	Java Examples
=	Equal to	==	`balance == 0` `answer == 'y'`
≠	Not equal to	!=	`income != tax` `answer != 'y'`
>	Greater than	>	`expenses > income`
≥	Greater than or equal to	>=	`points >= 60`
<	Less than	<	`pressure < max`
≤	Less than or equal to	<=	`expenses <= income`

Compound Boolean Expressions: And

- Boolean expressions can be combined using the "and" **(&&)** operator.

```
if ((score > 0) && (score <= 100))
```

- Not allowed – Don't do this!!

```
if (0 < score <= 100)
```

- Syntax:

```
(Expression_1) && (Expression_2)
```

- Parentheses often are used to enhance readability.
- The larger expression is true only when both of the smaller expressions are true.

Compound Boolean Expressions: Or

- Boolean expressions can be combined using the "or" `(||)` operator.

```
if ((quantity > 5) || (cost < 10))
```

- The larger expression is true
  - When either of the smaller expressions is true
  - When both of the smaller expressions are true.
- The Java version of "or" is the *inclusive or* which allows either or both to be true.
- The *exclusive or* allows one or the other, but not both to be true.

Negating a Boolean Expression

- A boolean expression can be negated using the "not" (`!`) operator.
- Syntax

  ```
  !(Boolean_Expression)
  ```

- Example

  ```
  (a || b) && !(a && b)
  ```

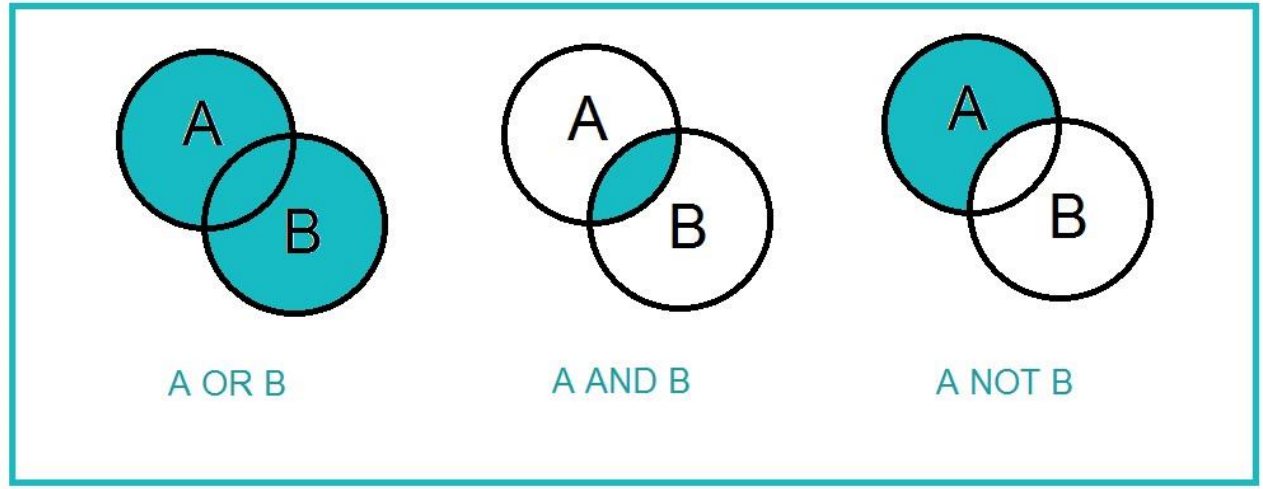
  which is the *exclusive or*

Negating a Boolean Expression

! (*A Op B*) Is Equivalent to (*A Op B*)	
<	>=
<=	>
>	<=
>=	<
==	!=
!=	==

Java Logical Operators

Name	Java Notation	Java Examples
Logical *and*	&&	`(sum > min) && (sum < max)`
Logical *or*	||	`(answer == 'y') || (answer == 'Y')`
Logical *not*	!	`!(number < 0)`



Using ==

- == is appropriate for determining if two integers or characters have the same value.

```
if (a == 3)
```

where a is an integer type

- == is **not** appropriate for determining if two floating points values are equal. Use < and some appropriate tolerance instead.

```
if (abs(b - c) < epsilon)
```

where b, c, and epsilon are floating point types, and epsilon is a small number

Using ==

- == is not appropriate for determining if two objects have the same value.
  - `if (s1 == s2),` where `s1` and `s2` refer to strings, determines only if s1 and s2 refer the a common memory location.
  - If `s1` and `s2` refer to strings with identical sequences of characters, but stored in different memory locations, `(s1 == s2)` is false.

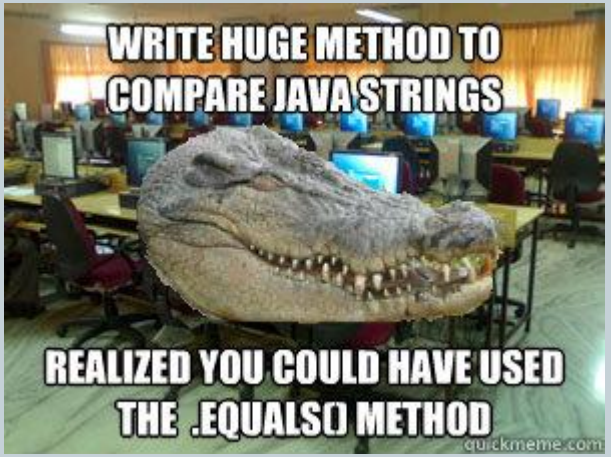
Using ==

- To test the equality of objects of class String, use method **`equals`**.

  `s1.equals(s2)`

  or

  `s2.equals(s1)`



- To test for equality ignoring case, use method **`equalsIgnoreCase`**.

  `("Hello".equalsIgnoreCase("hello"))`

Lazy Evaluation

- Sometimes only part of a boolean expression needs to be evaluated to determine the value of the entire expression.
  - If the first operand associated with an `||` is `true`, the expression is `true`.
  - If the first operand associated with an `&&` is `false`, the expression is `false`.
- This is called *short-circuit* or *lazy* evaluation.

Lazy Evaluation

- Lazy evaluation is not only efficient, sometimes it is essential!
- A run-time error can result, for example, from an attempt to divide by zero.

```
if ((number != 0) && (sum/number > 5))
```

- *Complete evaluation* can be achieved by substituting `&` for `&&` or `|` for `||`.

The `switch` Statement

- The `switch` statement is a multiway branch that makes a decision based on an *integral* (integer or character) expression.
  - Java 7 allows String expressions
- The `switch` statement begins with the keyword `switch` followed by an integral expression in parentheses and called the *controlling expression*.

The `switch` Statement

- A list of cases follows, enclosed in braces.
- Each case consists of the keyword `case` followed by
  - A constant called the *case label*
  - A colon
  - A list of statements.
- The list is searched for a case label matching the controlling expression.

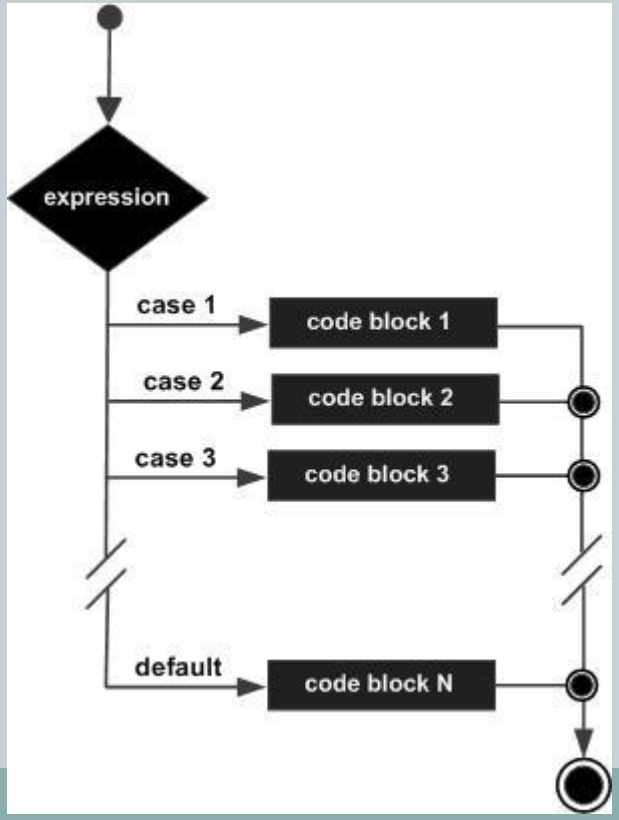
The `switch` Statement

- The action associated with a matching case label is executed.
- If no match is found, the case labeled `default` is executed.
  - The `default` case is optional, but recommended, even if it simply prints a message.
- Repeated case labels are not allowed.

The `switch` Statement

- Syntax

```
switch (Controlling_Expression)
{
    case Case_Label:
        Statement(s);
        break;
    case Case_Label:
    …
    default:
    …
}
```



Conditional Action from a Set

- Do something depending on a value value
  - if-else if-else if… statements can get tedious

```
if (day == 1)
   monthStr = "Monday";
else if (day == 2)
   monthStr = "Tuesday";
else if (day == 3)
   monthStr = "Wednesday";
else if (day == 4)
   monthStr = "Thursday";
else if (day == 5)
   monthStr = "Friday";
else if (day == 6)
   monthStr = "Saturday";
else if (day == 7)
   monthStr = "Sunday";
else
   monthStr = "Invalid day!";
```

Set a String variable `monthStr` to a string according to the integer value in the `day` variable.

Conditional Action from a Set

- `switch` statement
  - Works with: `byte`, `short`, `char`, `int`, `enumerations`
  - Java 1.7: `String`

```
switch (day)
{
   case 1:  monthStr = "Monday";       break;
   case 2:  monthStr = "Tuesday";      break;
   case 3:  monthStr = "Wednesday";    break;
   case 4:  monthStr = "Thursday";     break;
   case 5:  monthStr = "Friday";       break;
   case 6:  monthStr = "Saturday";     break;
   case 7:  monthStr = "Sunday";       break;
   default: monthStr = "Invalid day!"; break;
}
```

case block normally ends with a break

default block is optional, but if present executes if no other case matched. Like the else in an if-else if-else statement.

The switch Statement

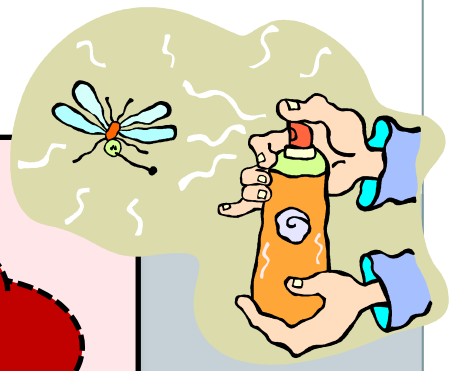
```
final int NORTH = 0;
final int SOUTH = 1;
final int EAST  = 2;
final int WEST  = 3;

int direction   = 0;

switch (direction)
{
   case NORTH:
      y--;
      System.out.println("Walking north");
      break;
   case SOUTH:
      y++;
      System.out.println("Walking south");
      break;
   case EAST:
      x++;
      System.out.println("Walking east");
      break;
   case WEST:
      x--;
      System.out.println("Walking west");
   break;
}
```

You can have as many statements as you want between case and break.

Buggy switch Statement



```
final int NORTH = 0;
final int SOUTH = 1;
final int EAST  = 2;
final int WEST  = 3;
int direction = 0;

switch (direction)
{
   case NORTH:
      y--;
      System.out.println("Walking north");
   case SOUTH:
      y++;
      System.out.println("Walking south");
   case EAST:
      x++;
      System.out.println("Walking east");
   case WEST:
      x--;
      System.out.println("Walking west");
}
```

case block will fall through to next block if no break!

```
Output:

Walking north
Walking south
Walking east
Walking west
```

Falling Through Cases

```
int direction = 0;

switch (direction)
{
   case NORTHWEST:
   case NORTHEAST:
   case NORTH:
      System.out.println("Heading northbound!");
      break;
   case SOUTHWEST:
   case SOUTHEAST:
   case SOUTH:
      System.out.println("Walking southbound!");
      break;
}
```

Sometimes falling through to next case block is what you want.

Easy way to do same thing for a set of discrete values.

```
Output:

Heading southbound
```

Enumerations

- Consider a need to restrict contents of a variable to certain values
- An enumeration lists the values a variable can have
- Example

```
enum MovieRating {E, A, B}
MovieRating rating;
rating = MovieRating.A;
```

Enumerations

- Now possible to use in a **`switch`** statement

```
switch (rating)
{
   case E: //Excellent
      System.out.println("You must see this movie!");
      break;
   case A: //Average
      System.out.println("This movie is OK, but not great.");
      break;
   case B: // Bad
      System.out.println("Skip it!");
      break;
   default:
      System.out.println("Something is wrong.");
}
```

Enumerations

- An even better choice of descriptive identifiers for the constants

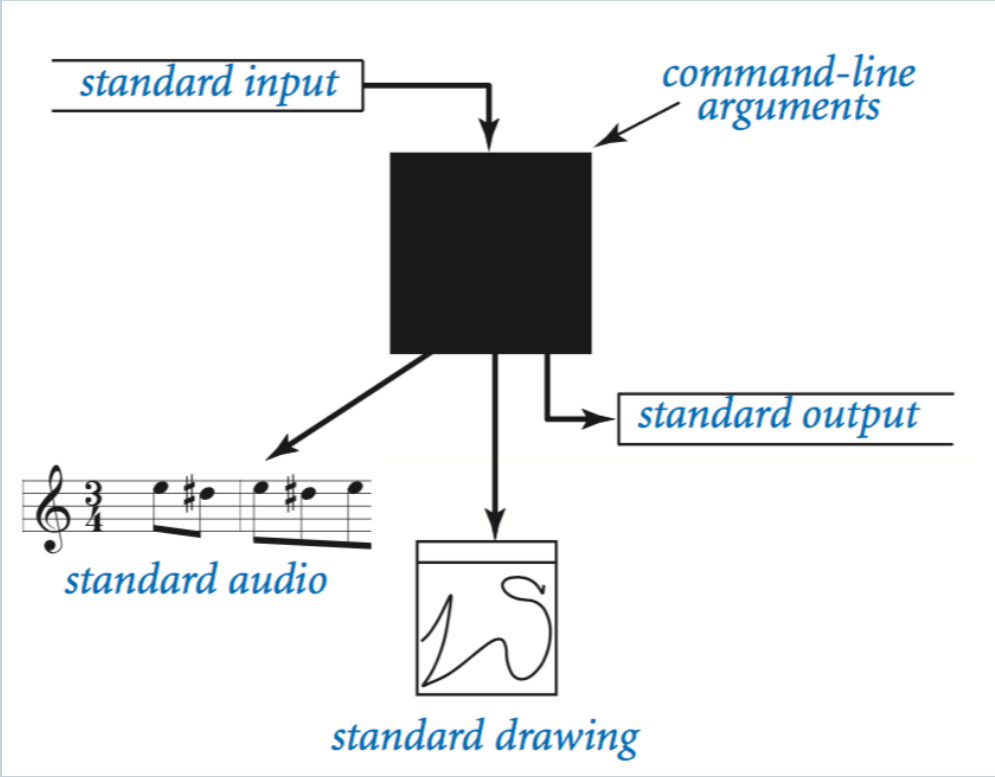
```
enum MovieRating
        {EXCELLENT, AVERAGE, BAD}
rating = MovieRating.AVERAGE;

case EXCELLENT:   ...
```

Summary: Conditionals Revisited

- if Statement
- Boolean Expressions
- switch Statement

Basic Input/Output



Outline: Basic Input/Output

- Screen Output
- Keyboard Input

Simple Screen Output

```
System.out.println("The count is " + count);
```

- Outputs the sting literal `"the count is "`
- Followed by the current value of the variable `count`.

- We've seen several examples of screen output already.
  - `System.out` is an object that is part of Java.
  - `println()` is one of the methods available to the `System.out` object.

Screen Output

- The concatenation operator (+) is useful when everything does not fit on one line.

```
System.out.println("Lucky number = " + 13 +
  "Secret number = " + number);
```

- Do not break the line except before or after the concatenation operator (+).

Screen Output

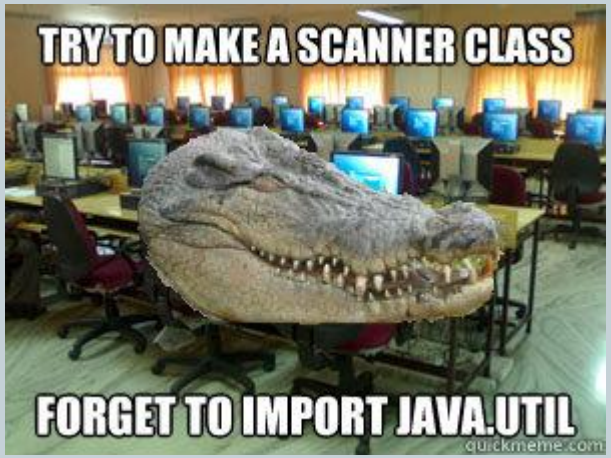
- Alternatively, use `print()`

```
System.out.print("One, two,");
System.out.print(" buckle my shoe.");
System.out.println(" Three, four,");
System.out.println(" shut the door.");
```

ending with a `println()`.

Keyboard Input

- Java has reasonable facilities for handling keyboard input.
- These facilities are provided by the `Scanner` class in the `java.util` package.
  - A *package* is a library of classes.



Simple Input

- Sometimes the data needed for a computation are obtained from the user at run time.
- Keyboard input requires

  ```
  import java.util.Scanner
  ```

  at the beginning of the file.

Simple Input

- Data can be entered from the keyboard using

```
Scanner keyboard =
        new Scanner(System.in);
```

  followed, for example, by

```
eggsPerBasket = keyboard.nextInt();
```

  which reads one `int` value from the keyboard and assigns it to `eggsPerBasket`.

Using the Scanner Class

- Near the beginning of your program, insert

```
import java.util.Scanner;
```

- Create an object of the `Scanner` class

```
Scanner keyboard =
    new Scanner (System.in)
```

- Read data (an `int` or a `double`, for example)

```
int n1 = keyboard.nextInt();
double d1 = keyboard,nextDouble();
```

Some `Scanner` Class Methods

Scannner_Object_Name`.next()`
Returns the `String` value consisting of the next keyboard characters up to, but not including, the first delimiter character. The default delimiters are whitespace characters.

Scannner_Object_Name`.nextLine()`
Reads the rest of the current keyboard input line and returns the characters read as a value of type `String`. Note that the line terminator `'\n'` is read and discarded; it is not included in the string returned.

Scannner_Object_Name`.nextInt()`
Returns the next keyboard input as a value of type `int`.

Scannner_Object_Name`.nextDouble()`
Returns the next keyboard input as a value of type `double`.

Scannner_Object_Name`.nextFloat()`
Returns the next keyboard input as a value of type `float`.

Some `Scanner` Class Methods

- Figure 2.7b

Scannner_Object_Name`.nextLong()`
Returns the next keyboard input as a value of type `long`.

Scannner_Object_Name`.nextByte()`
Returns the next keyboard input as a value of type `byte`.

Scannner_Object_Name`.nextShort()`
Returns the next keyboard input as a value of type `short`.

Scannner_Object_Name`.nextBoolean()`
Returns the next keyboard input as a value of type `boolean`. The values of `true` and `false` are entered as the words *true* and *false*. Any combination of uppercase and lowercase letters is allowed in spelling *true* and *false*.

Scannner_Object_Name`.useDelimiter(`*Delimiter_Word*`);`
Makes the string *Delimiter_Word* the only delimiter used to separate input. Only the exact word will be a delimiter. In particular, blanks, line breaks, and other whitespace will no longer be delimiters unless they are a part of *Delimiter_Word*.

This is a simple case of the use of the `useDelimiter` method. There are many ways to set the delimiters to various combinations of characters and words, but we will not go into them in this book.

`nextLine()` Method Caution

- The `nextLine()` method reads
  - The remainder of the current line,
  - Even if it is empty.
- Example – given following declaration.

```
int n;
String s1, s2;
n = keyboard.nextInt();
s1 = keyboard.nextLine();
s2 = keyboard.nextLine();
```

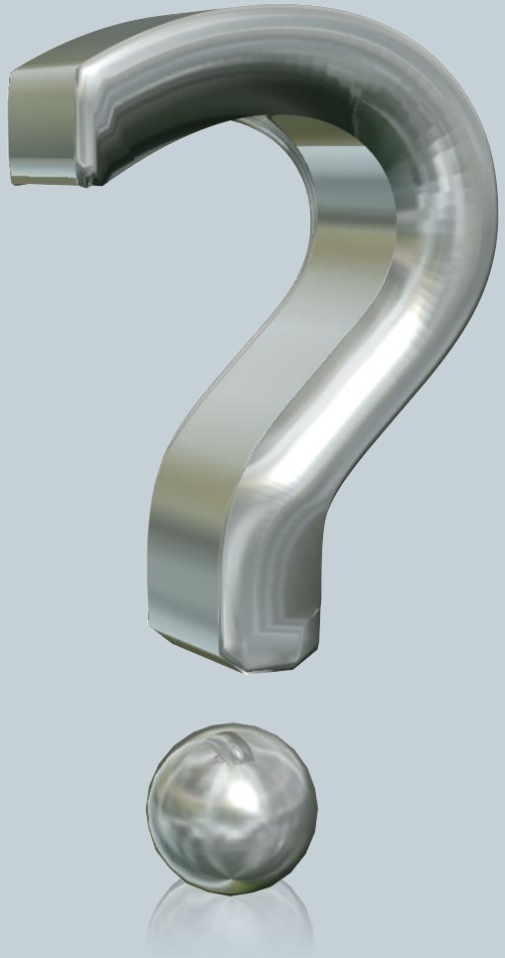
- Assume input shown

`n` is set to `42`
but `s1` is set to the empty string.

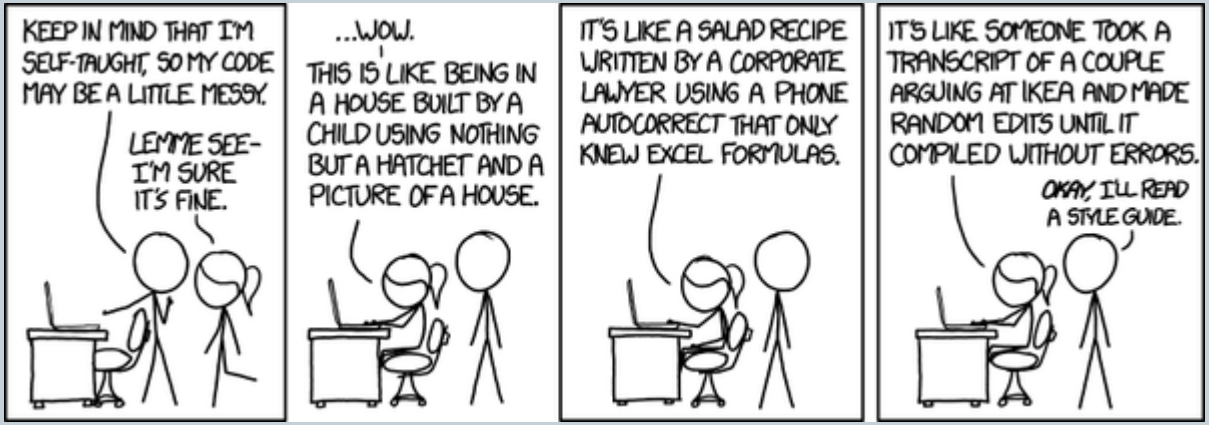
```
42
and don't you
forget it.
```

Outline: Basic Input/Output

- Screen Output
- Keyboard Input



Programming Style



Documentation and Style: Outline

- Meaningful Names
- Comments
- Indentation
- Named Constants
- Whitespace
- Compound Statements

Documentation and Style

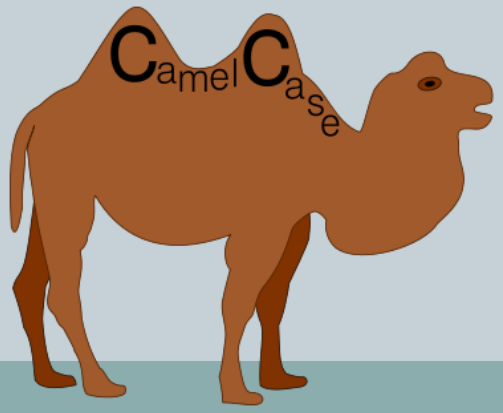
- Most programs are modified over time to respond to new requirements.
- Programs which are easy to read and understand are easy to modify.
- Even if it will be used only once, you have to read it in order to debug it .
- And when we talk about javadoc, if your comments are meaningful, your code will write its own documentation!!

Meaningful Variable Names

- A variable's name should suggest its use
  - e.g. `taxRate`
- Boolean variables should suggest a true/false value
  - Choose names such as `isPositive` or `systemsAreOk`.
  - Avoid names such as `numberSign` or `systemStatus`.

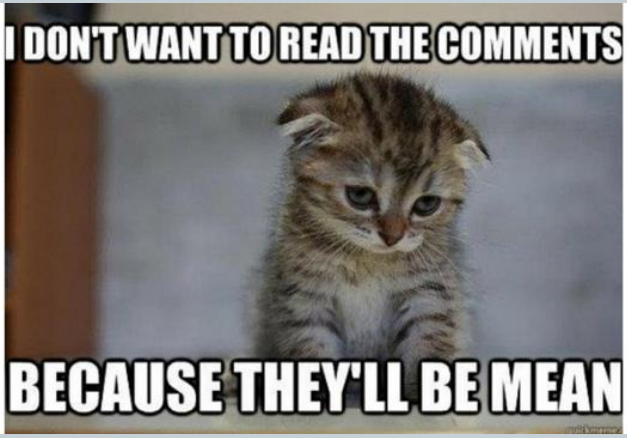
Style: Naming Things

- Variable names
  - Begin with lowercase, uppercase each new word
  - `int totalWidgets;`
- Class names
  - Begin uppercase, then lowercase except for new words
  - `public class InventoryTracker`
  - Name exactly as in assignment description
- Constants
  - All upper case, use _ between words
  - `double SPEED_LIGHT = 3.0e8;`



Comments

- The best programs are self-documenting.
  - Clean style
  - Well-chosen names
- Comments are written into a program as needed to explain the program.
  - They are useful to the programmer, but they are ignored by the compiler.



Style: Comments

- Comments help reader/grader understand your program
  - Good comments explain why something is done
  - Write comments before coding tricky bits
    - Helps you formulate a plan
  - Don't comment the obvious:
    - `i++;  // Increment i by one`

Comments

- A comment can begin with //.
- Everything after these symbols and to the end of the line is treated as a comment and is ignored by the compiler.


```
double radius; //in centimeters
```

Comments

- A comment can begin with `/*` and end with `*/`
- Everything between these symbols is treated as a comment and is ignored by the compiler.

```
/**
This program should only
be used on alternate Thursdays,
except during leap years, when it should
only be used on alternate Tuesdays.
*/
```

Comments

- A *javadoc* comment, begins with `/**` and ends with `*/`.
- It can be extracted automatically from Java software.

```
/** method change requires the number of coins
  to be nonnegative */
```

We will talk about javadoc later in the semester.

When to Use Comments

- Begin each program file with an explanatory comment
  - What the program does
  - The name of the author
  - Contact information for the author
  - Date of the last modification.
- Provide only those comments which the expected reader of the program file will need in order to understand it.

Indentation

- Indentation should communicate nesting clearly.
- A good choice is four spaces for each level of indentation.
- Indentation should be consistent.
- Indentation should be used for second and subsequent lines of statements which do not fit on a single line.
- Indentation does not change the behavior of the program.
- Proper indentation helps communicate to the human reader the nested structures of the program
- Eclipse will help you indent correctly
  - Eclipse can fix automatically, ctrl-a then ctrl-i

```
This is not indented.
    This is indented.
This is not indented.
```

Using Named Constants

- To avoid confusion, always name constants (and variables).

```
area = PI * radius * radius;
```

  is clearer than

```
area = 3.14159 * radius * radius;
```

- Place constants near the beginning of the program.
- Once the value of a constant is set (or changed by an editor), it can be used (or reflected) throughout the program.

```
public static final double INTEREST_RATE = 6.65;
```

- If a literal (such as 6.65) is used instead, every occurrence must be changed, with the risk than another literal with the same value might be changed unintentionally.

Declaring Constants

- Syntax

```
public static final Variable_Type
  Variable_Name = Constant_Value;
```

- Examples:

```
public static final double PI = 3.14159;
public static final String MOTTO = "The
  customer is always right.";
```

- By convention, uppercase letters are used for constants.

Style: Whitespace

```
public class StarTriangle
{
public static void main(String[] args)
  {int limit = Integer.parseInt(args[0]);
for (int i=0;i<limit;i++){
            for (int j = 0; j <= i; j++)
            System.out.print("*");System.out.println();
}}}
```

- Indent each level of conditionals/loops
  - Indent a fixed number of spaces (3-4)
- Use blank lines to separate logical sections
- Only one statement per line

Style: Whitespace

```
for (int i=0;i<limit;i++)
```

vs.

```
for (int i = 0; i < limit; i++)
```

```
a=b*c/d-(8.12*e);
```

vs.

```
a = b * c / d - (8.12 * e);
```

```
//this is a comment
//describing my code
```

vs.

```
// this is a comment
// describing my code
```

- Use spaces between
  - Statements in for loop
  - Operators in math expressions
  - After the // starting a comment

Style: Whitespace

```
Math . random ();
```
vs.
```
Math.random();
```

```
args [0];
```
vs.
```
args[0];
```

```
i = i + 1 ;
```
vs.
```
i = i + 1;
```

- Do NOT use spaces between
  - method class, dot, name, or ()'s
  - array name and []'s
  - statement and ending semicolon

"White space? You want more white space? Just look outside! "

Style: Whitespace

- Use spaces to align parallel code if it makes it more readable
  - Often helps to spot mistakes


```
int numPoints = Integer.parseInt(args[0]);
int startX = Integer.parseInt(args[0]);
int startY = Integer.parseInt(args[2]);
double velX = Integer.parseInt(args[3]);
double velY = Integer.parseInt(args[4]);
```

```
int    numPoints = Integer.parseInt(args[0]);
int    startX    = Integer.parseInt(args[0]);
int    startY    = Integer.parseInt(args[2]);
double velX      = Integer.parseInt(args[3]);
double velY      = Integer.parseInt(args[4]);
```

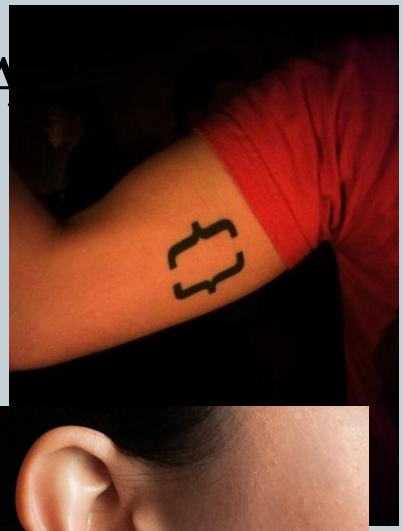
Style: Curly Bracing

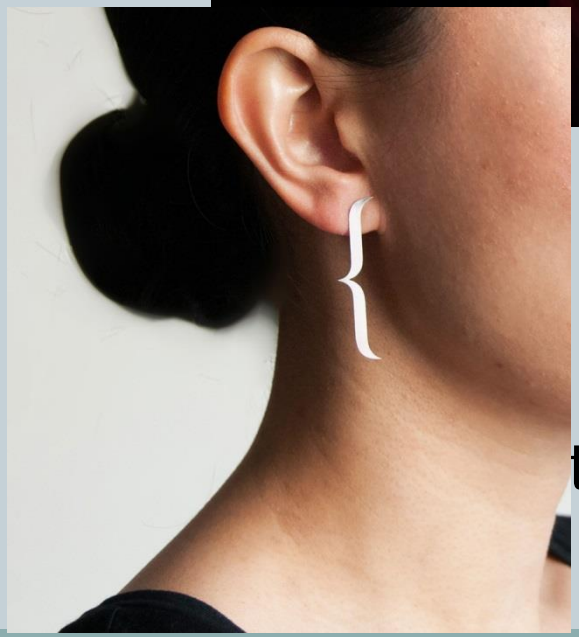
```
public class HelloWorld
{
   public static void main(String [] args)
   {
      System.out.println("Hello world!");
   }
}
```

```
public class HelloWorld {
   public static void main(String [] args) {
      System.out.println("Hello world!");
   }
}
```

```
public class HelloWorld {
   public static void main(String [] args)
   {
      System.out.println("Hello world!");
   }
}
```

BSD-A





t!

matching!

Summary

- Meaningful Names
- Comments
- Indentation
- Named Constants
- Whitespace
- Compound Statements