+

Elementary Java Programming & Selection Statements

Introduction to Computer Science CSCI-UA.0101

Lecture 3

+

Agenda Day 3

- More Input / Output: Dialog Boxes
- More on variables and assignment
- Selection statements

+
Console Input/Output Examples:

- Circle.java
- PayrollConsole.java

+

More Input/Output: Dialog Boxes

+
The Java Library

- Java comes pre-installed with a huge library of pre-written classes that you can use in your programs.
- The full library – along with documentation – is available on Oracle's website:
  - http://docs.oracle.com/javase/7/docs/api/
- Most Java programmers are fluent with a core set of classes in the Java library.
- It's more important to be able to know how to work with the library rather than to know how to use every single class – you can always look up a particular class when you encounter the need to use it!

+

Displaying Text in a Message Dialog Box

- Java has an extensive library for constructing Graphical User Interfaces (GUIs)
- This library, called “swing”, can be imported into your program in order to give you access to Java’s GUI functionality
- The “swing” library is housed in the following “package” (note that some packages are nested):

```
javax.swing
```

- The “swing” library contains a number of classes – let’s check them out:

http://docs.oracle.com/javase/7/docs/api/

+

Importing a class from the library

- Before you can use a class in the Java library you must first import it into your program
- You can do this by using the import statement
- The import statement is placed above your class definition and looks like the following.

```
import package.className
```

+

More about the import statement

- The import statement is designed to make it easy to utilize classes outside of your program
- There are two types of import statements
  - Specific import statements: You specify the exact class you wish to import into your program. Example:

    ```
    import javax.swing.JOptionPane;
    ```

  - Wildcard import statements: You specify that you want to import ALL classes inside a specific package into your program. Example:

    ```
    import javax.swing.*;
    ```

Displaying Text in a Message Dialog Box

- The JOptionPane class in the “swing” library has the ability to create a pop-up window that can be used to display strings of text to the user
- To use the JOptionPane class you must first import it into your class as follows:

```
import javax.swing.JOptionPane;

public class Welcome
{
      // class code to go here
}
```

+ Displaying Text in a Message Dialog Box

- We can then use the JOptionPane to display text by calling the "showMessageDialog" method inside the JOptionPane class.
- Methods on classes can be called using "dot" syntax. To call the "showMessageDialog" method on the JOptionPane class you can do the following:

```
JOptionPane.showMessageDialog(null, "Hello!");
```

+ Displaying Text in a Message Dialog Box

Welcome.java

```
import javax.swing.JOptionPane;

public class Welcome
{
    public static void main(String[] args)
    {
        // display a GUI pop-up box with some text
        JOptionPane.showMessageDialog(null, "Hello, World!");
    }
}
```

So what about `System.out.println()`?

- As you remember from a few slides ago, System.out.println() lets you print text to the console
- However, you never imported a package for this method call, so how does Java know what to do when you call it?
- If you look at the Java library you will see that the System class is organized in the “java.lang” package. This package is automatically imported into all Java classes for you.

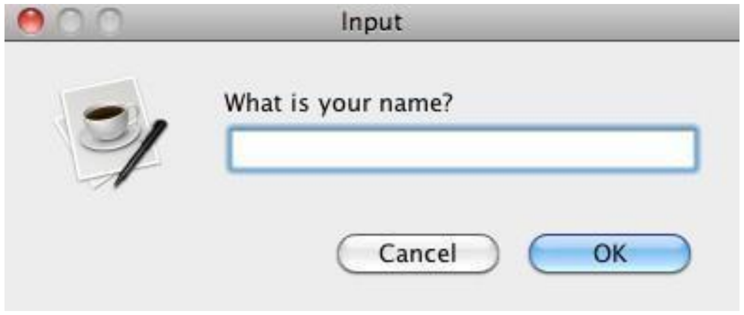
+

Getting user input using a GUI

- During our previous lecture we discussed how we can obtain console input from the user using an instance of the Scanner class
- You can also obtain user input using a GUI component from the Java library as follows:

```
String s = JOptionPane.showInputDialog("What is your  name?");
```

- Which will yield a text box that looks like the one below.



+

Getting user input using a GUI

- Note that you need to import the JOptionPane class from the javax.swing package in order for this to work:

```
import  javax.swing.JOptionPane;
```

- Also note that there’s no guarantee that the user will actually type something into the box that’s provided. If that’s the case then then String that gets created will be empty.

+

Converting Strings to Numeric Data Types

- The JOptionPane.showInputDialog() method will always return a String (much like how the input() function in Python always returns a String)
- If you wish to use the input data in a numeric expression you first need to convert it into the appropriate data type.
- You can use the following static methods which are pre-imported into your class to do this. Note that the original String is unaffected by the data type conversion.

```
// define a String
String num = "5";

// create an int version of the String
int numInt =  Integer.parseInt(num);

// create a double version of the String
double numDouble =  Double.parseDouble(num);
```

+
Converting Numeric Data to String Types

- The JOptionPane.showMessageDialog() method will automatically convert some numeric to string. The output in a Dialog box is always String.

- If you wish to do the conversion yourself, you can use another method available in the Integer and Double classes. We will need to do this later, when working with various methods.

- You can use the following static methods which are pre-imported into your class to do this.

```
// define an Integer
int num =  5 ;

// create a  String version of the Integer
String  snum = Integer.toString(num);


// create a  String version of the Double
Double num = 4.2;
String  snum = Double.toString(num);
```

Dialog-Boxes Example:

- PayrollDialogBoxes.java

+ Escape Characters

- What if you wanted to store the following line of text inside of a String variable?

```
Hello, welcome to “CS101” !!!
```

- The internal quotes around “CS101” would cause problems with the String delimiter (which is also a quotation mark)

- To solve the problem Java, like Python, supports escape characters. The following is a valid Java statement that allows you to store quotation marks inside of a String:

```
String m = “Hello, welcome to \”CS101\” !!!”;
```

Special Escape Characters

Escape Character	Function
\b	Backspace
\t	Tab
\n	Linefeed
\r	Carriage Return
\\	Backslash
\″	Quote

+

More on Variable Mechanics

From Beginning of Chapter 4

+
Exponent Operations

- Java does not have a built-in operator for exponents
- With that said, you can use the Math class in the Java library to perform a number of math operations not available in the language itself
- The Math class is a static class so it doesn't need to be instantiated like the Scanner class, which requires the "new" keyword – you can just use it right out of the box.
- In addition, it exists inside the java.lang package, so you don't need to import anything in order to use it.
- Example:

```
// print out 2 raised to the 3rd power
System.out.println( Math.pow(2, 3) );
```

+

Self Referential Assignment Operations

- Often you will need to change a variable based on its existing value. For example:

```
int numItems = 5;

// let the user buy 2 more items
numItems = numItems + 2;

// numItems is now 7
```

- We call these “self referential” assignment statements since the assignment statement literally refers to the variable being changed.

+

Augmented Assignment Operators

- The core math operators (+, -, /, * and %) can be combined with the assignment operator (=) to form an augmented assignment operator
- Augmented assignment operators also allow you to perform operations on variables that refer back to themselves. For example:

```
int x = 5;
x += 1;  //same as x = x + 1;
System.out.println(x); // 6
```

+

Augmented Assignment Operators

Operator	Name	Example	Equivalent
`+=`	`Addition`	`i += 8`	`i = i + 8`
`-=`	`Subtraction`	`i -= 8`	`i = i - 8`
`*=`	`Multiplication`	`i *= 8`	`i = i * 8`
`/=`	`Division`	`i /= 8`	`i = i / 8`
`%=`	`Remainder`	`i %= 8`	`i = i % 8`

Increment and Decrement Operators

- Java contains two special augmented assignment operators that allow you to add or subtract 1 from an item using a special shorthand notation. For example:

```
int a = 5;

// increase a by 1
a++;

System.out.println(a); //  6

// decrease a by 1

a--;

System.out.println(a); //  5
```

+

Numeric Type Conversions

- Java will automatically perform some data type conversions for you. Take the following for example:

```
double a = 5.0 * 2;
```

- In this case we are multiplying an integer and a double, and storing the result in a double. Java will automatically convert the integer in this expression into a double for you in order to perform the calculation. This means that the statement above is equivalent to the statement below:

```
double b = 5.0 * 2.0;
```

+

Numeric Type Conversions (casting)

- Java will always convert data from data types that support smaller ranges into ones that support larger ranges, if necessary.
- Java won't, however, go the other way. Consider the following:

```
System.out.println(1.0 / 2);   // 0.5
```

- In this case Java turned the integer 2 into 2.0 and performed a division operation. What if we wanted to treat 1.0 as an integer and perform integer division instead? We can use a technique called "casting" to tell Java to force a value to behave in a certain way. For example:

```
System.out.println( (int) 1.0 / 2);   // 0
```

+

Numeric Type Conversions (casting)

- When you cast a variable from a smaller type into a larger type (i.e. int to double) you are “widening” the type. Java will do this for you automatically as necessary.

- When you cast a variable from a larger type into a smaller type (i.e. double to int) you are “narrowing” the type. You must do this explicitly using casting syntax as follows:

```
System.out.println( (int) 10.75 * 2); //  20
```

- When you narrow a type you lose some information in the process. For example, in the statement above the double value 10.75 will lose its fractional value and will be truncated to a whole number.

+

NumericType Conversions (casting)

- Note: casting will not change the variable being cast. In the following example, “b” will not change from a double.

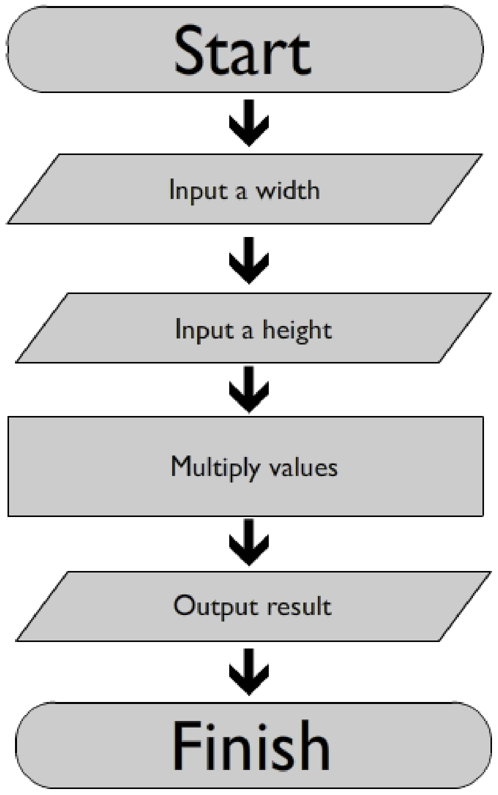
```
int a = 10;
double b = 5.5;

a += (int) b;

System.out.println(a + “ - “ + b);

// 15 - 5.5
```

+

Selection Statements

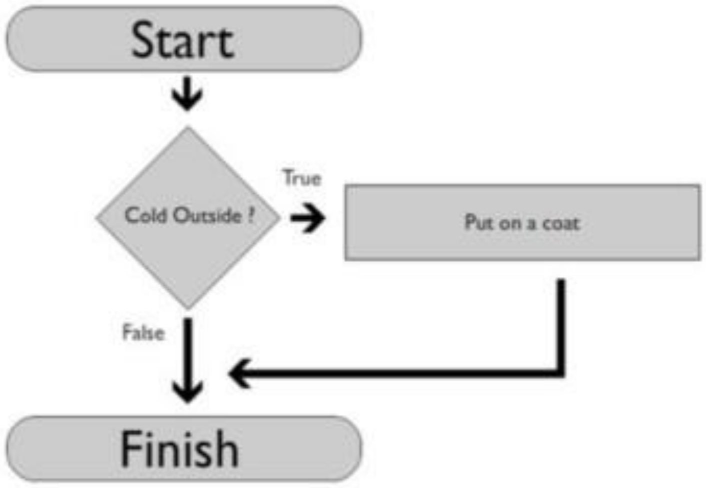
Sequence Structures

- So far we have been programming "sequence structures" in our main method
- Sequence structures are sets of statements that execute in the order in which they appear
- Obviously not all programs can be written this way, as there are certain times when we need to deviate from a linear structure and adapt our program based on information provided.



The Selection Statement

- Simplest form - perform an action only if a certain condition exists
- For example, in the program to the right we begin by asking a question - “is it cold outside?”
- If the answer to this question is yes (“True”) then we can execute an alternate set of commands
- Otherwise we can continue with the program as-is



+ Constructing a Selection Statement

- In order to construct a selection statement we must first identify the question or condition that needs to be evaluated
- We can then construct a relational statement that can be tested in order to determine if that particular condition exists. For example:
  - Is variable a greater than variable b?
  - Is variable c equal to variable d?
  - Is variable e less than or equal to variable f?
- The answers to all of the questions above can be answered as either True or False.
- We call these "Boolean Conditions" or "Boolean Expressions" after the English mathematician George Boole

+ Relational Operators

Relational Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Boolean Expressions

- You can use relational operators to construct relational expressions that test the relationship between various entities
- Boolean expressions always evaluate to "true" or "false"
- Example:

```
int a = 5;
int b = 10;

System.out.println(a < b);

// will print "true" to the console
```

The boolean data type

- In Java, the boolean data type lets you represent true and false values. Example:

```
boolean var1 = true;
boolean var2 = false;
```

- Note the capitalization - ‘true’ and ‘false’ are both lowercase, unlike in Python.

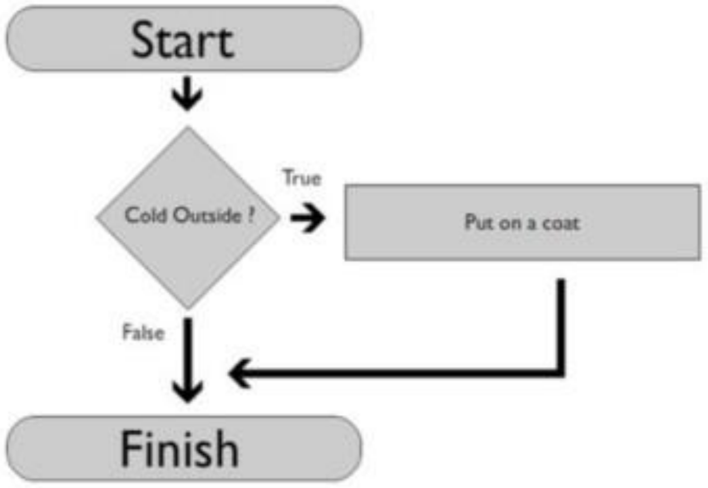
- ‘true’ and ‘false’ are considered literals, just like numbers. They are reserved words so you can’t use them as identifier names.

- You can store the result of a Boolean expression in a boolean variable. For example:

```
int a = 5;
int b = 10;
boolean test = a < b;
```

“if” statements

- An “if” statement can be used to execute a series of statements if a certain condition evaluates to true
- The simplest form of an “if” statement is the “one way” if statement. In this statement we pose a question - if the answer is true, we execute a series of statements. If it is false we simply continue with the program.



“if” statements

- An “if” statement can be constructed in Java using the following syntax:

```
if (boolean-expression)
{
      statement(s);
}
```

- Parenthesis are required around your Boolean expressions
- Unlike in Python, indentation does not matter, though you should follow standard Java formatting rules.
- An “if” statement does not require a semicolon.
- Having a block of curly braces is required for “if” statements that block contains more than 1 statement. It’s usually a good idea to always put curly braces in anyway.

Two-way “if-else” statements

- A one-way “if” statement only allows you to respond to a condition if the Boolean expression in question evaluates to true
- If you want to respond to when the condition evaluates to false you can use a two-way “if-else” statement using the following syntax. Note that in the example below one of the two sets of statements is guaranteed to execute - if the original boolean expression evaluates to false then the else clause automatically is invoked.

```
if (boolean_expression)
{
      statements(s);
}
else
{
      statements(s);
}
```

+ Nested "if" statements

- "if" statements can be placed inside of one another to form "nested" if statements using the following syntax:

```
if (boolean_condition)
{
     statement(s);

     if (boolean_condition_2)
     {
            statement(s);
     }
     else
     {
            statements(s);
     }
}
```

Programming Challenge

- Write a grade determination program that accepts a double and reports its associated letter grade. Write this program using nested "if" statements.

- **GradeAverage.java**

Re-writing nested "if" statements using "else if"

- Sometimes it's easier to work with nested "if" statements if they are re-written using "else if" syntax.
- Each boolean expression is checked in succession. If one is found to be true we execute its associated block and do not test the additional expressions.
- If no expressions evaluate to true we can execute the "else" block at the end. Note that the presence of an "else" block is not required, and if it was omitted we could potentially never execute any statement in the nested "if" structure.

```
if  (boolean_expression)
{
        statement(s);
}

else if  (boolean_expression)
{
        statement(s);
}

else
{
        statement(s);
}
```

Programming Challenge

- Write a grade determination program that accepts a double and reports its associated letter grade. Write this program using nested "if" statements.
- Next, re-write the program using "else if" statements.
- **GradeAverage1.java**

Common Errors in Selection Statements

+ Don’t forget your braces

- Selection statements that contain more than one statement do not technically require braces. For example, the following code is valid:

```
if (x < 10)
    System.out.println(“Foo”);
```

- However, if a section statement contains more than 1 statement braces are required to define a block of execution. For example:

```
if (x < 10)
{
    System.out.println(“Foo  1”);
    System.out.println(“Foo 2”);
}
```

- Note that you can always include braces in a selection statement even if it only contains a single statement. This may help to minimize errors in your logic.

Semicolons are not required after your boolean condition

- A semicolon is only needed after a statement that is not immediately followed by a block of execution. The semicolon in the "if" statement below is incorrect and will actually result in a logic error:

```
int x = 5;

if (x == 1000);          // error!
{
     System.out.println("hi!");
}
```

Redundant Testing

- When testing to see if a boolean variable is true or false the use of a relational operator is not technically required. For example:

```
boolean flag = true;

// this boolean condition is the same as the one below
if (flag == true)
{
      System.out.println("True!");
}

// this boolean condition is the same as the one above
if (flag)
{
      System.out.println("Also True!");
}
```

Beware of Dangling Else clauses

- As you know, the “else” statement is syntactically optional in a selection statement
- Java will attach an else statement to the closest “if” statement that does not already have an else statement in place. For example, in the following the “else” statement in question actually matches the inner “if” statement and not the outer one:

```
int x = 1;
int y = 2;

if (x > 0)
     if (y < 0)
             System.out.println("1”);
else
     System.out.println("2");
```

+
Beware of Dangling Else clauses

- You can usually prevent this error from happening by making sure to consistently indent your code and use braces for all blocks of execution

```
int x = 1;
int y = 2;

if (x > 0)
{
     if (y < 0)
     {
            System.out.println("1”);
     }
     else
     {
            System.out.println("2");
     }
}
```

Logical Operators

+ Logical Operators

- A logical operator allows you to construct compound boolean expressions. For example:

```
If the temperature is higher than 50 degrees
and less than 70 degrees then tell the user
"It feels like autumn outside!"
```

- In the expression above two conditions are being tested - is temperature greater than 50 degrees? And is it also less than 70 degrees? If both of those conditions evaluate to true then we should present the user with the specified message.

+ Logical Operators

Operator	Description
`&&`	"and" - logical conjunction
`||`	"or" - logical disjunction
`!`	"not" - logical negation
`^`	"exclusive or" - logical exclusion

The “&&” operator

- The “&&” (and) operator allows you to test whether two conditions both evaluate to true. For example:

```
if (x > 5 && x < 10)
{
     System.out.println(“x is between 5 and 10”)
}
```

- In order for a compound boolean expression to evaluate to true using the “&&” operator all sub-expressions must also evaluate to true.

+

The "&&" Operator

P	Q	P && Q
True	True	True
True	False	False
False	True	False
False	False	False

+ Using && to determine if a value falls within a range

- In mathematics the following expression is valid:

```
10 < x < 20
```

- However, in Java this would be considered a Syntax error. This is because Java will evaluate this expression from left to right which will cause a data type mismatch error. Consider the following:

```
x = 15

if (10 < x < 20)    // 10 < x, true (boolean result)
                    // true < 20, ERROR!   Type mismatch
{
        …
}
```

Using && to determine if a value falls within a range

- You can rewrite this expression using the && operator. For example:

```
x = 15

if ( (10 < x) && (x < 20) )
{
      …
}
```

Programming Challenge

- The honey badger is a notoriously hardy creature that can survive in a wide range of environments.
- Honey badgers thrive in environments that are consistently between 40 and 100 degrees Fahrenheit.
- Write a program that asks the user for a temperature value and reports back whether it is safe for a honey badger.
- **HoneyBadgerAND.java**

The “||” operator

- The “||” (or) operator allows you to test whether at least one condition evaluates to true. For example:

```
if (x == 5 || x == -5)
{
     System.out.println(“x is either +5 or -5”)
}
```

- In order for a compound boolean expression to evaluate to true using the “||” operator at least one sub-expressions must evaluate to true.

The “||” Operator

P	Q	P || Q
True	True	True
True	False	True
False	True	True
False	False	False

Programming Challenge

- Re-write the honey badger program to use the OR operator instead of the AND operator
- Honey badgers thrive in environments that are consistently between 40 and 100 degrees Fahrenheit.
- **HoneyBadgerOR.java**

“Short Circuit” Operators

- The “and” and “or” operators are considered “short circuit” operators
- This means that Java will not necessarily evaluate an entire expression in order to evaluate a given statement. For example, there’s no need to evaluate beyond the first ‘true’ and ‘false’ in the statements below given that there is no value that could appear after the operator that could cause the expression to change.

```
boolean s = true || false
boolean t = false && true
```

The “^” operator

- The “^” (exclusive or) operator allows you to test whether one condition evaluates to true, but not both. For example:

```
int num_cakes = 1;
int num_pies = 0;

if (num_cakes > 0 ^ num_pies > 0)
{
        System.out.println("You can have either cakes or pies!");
}
```

- In order for a compound boolean expression to evaluate to true using the “^” operator exactly one sub-expressions must evaluate to true, but not both.

The “^” Operator

P	Q	P ^ Q
True	True	False
True	False	True
False	True	True
False	False	False

+
The “!” operator

- The “!” (not) operator flips the value of a boolean variable or expression. For example:

```
int age = 16;

if ( ! (age >= 18) )
{
        System.out.println("You must be 18 years old to qualify.");
}
```

The "!" Operator

P	!P
True	False
False	True

+

Programming Challenge: Leap Years

- A leap year is any year that meets the following criteria:
  - (a) It is evenly divisible by 4 but not by 100
  - or (b) It is evenly divisible by 400

- Write a program that prompts the user to type in a year as an integer and determines if the year is a leap year

- **LeapYear.java**

+

The “switch” statement

+ The “switch” statement

- There are times when you are programming that you need to evaluate multiple conditions in succession. For example:

```
int status = 2;

if (status == 1)
{
    System.out.println(“Status 1 code ..”);
}
else if (status == 2)
{
    System.out.println(“Status 2 code ..”);
}
else
{
    System.out.println(“Invalid status!”);
}
```

+
The "switch" statement

- Java provides a "switch" statement to simplify these kinds of tasks. For example:

```
int status = 2;

switch (status)
{
      case 1: System.out.println("Status == 1"); break;
      case 2: System.out.println("Status == 2"); break;
      default: System.out.println("Invalid!"); break;
}
```

+ The “switch” statement

- You never “need” to use a switch statement. You can always write a program using “if”, “else” and “else if” statements instead of using “switch”.
- Switch statements always begin with the “switch ‘keyword and a variable enclosed in a set of parenthesis. This variable is the one that is being evaluated in the switch statement.
- Next comes a block of execution.

```
switch (status)

{


    …


}
```

The "switch" statement

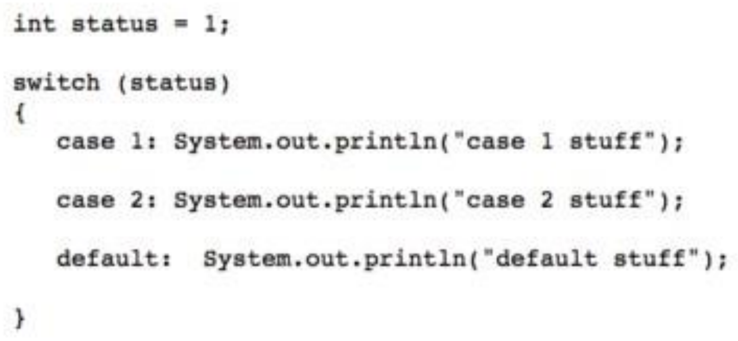
- Next comes a series of "case" statements.
- The case statements will be evaluated in order until a match can be found with the variable identified in the switch statement above. Case statements are terminated with a colon.
- Important Note! Case statements do not have blocks of execution associated with them.
- The break command inside of a case statement tells the "switch" statement to end.
- The "default" case is like the "else" block of an "if" statement. It executes if all of the blocks above evaluate to false.

```
switch (status)
{


   case 1: System.out.println("case 1"); break;
   case 2: System.out.println("case 2"); break;
   default: System.out.println("default"); break;


}
```

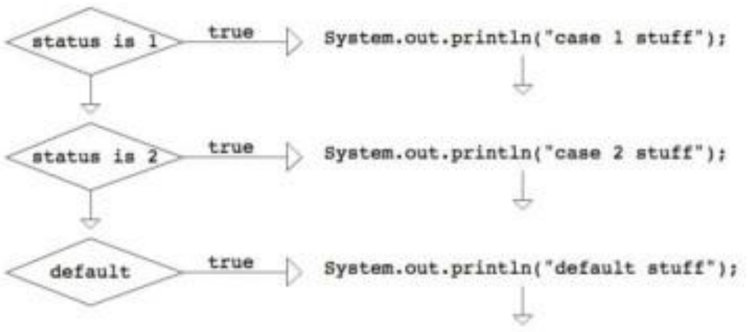
The "switch" statement

- Note that the "break" command is optional in a switch statement, but you almost always want to use it.
- If it you do not, the switch statement will continue and will treat all future cases as "matches".

Don’t forget the “break” statement



```
int status = 1;

switch (status)
{
   case 1: System.out.println("case 1 stuff");

   case 2: System.out.println("case 2 stuff");

   default:  System.out.println("default stuff");

}
```


status is 1
true
System.out.println("case 1 stuff");
status is 2
true
System.out.println("case 2 stuff");
default
true
System.out.println("default stuff");

+ Programming Challenge

- Write a program that asks the user to enter in a class year (1,2,3 or 4)
- Determine if the user is a Freshman, Sophomore, Junior or Senior based on this information. Use a switch statement.
- **ClassYearSwitch.java**

Operator Precedence

Operator Precedence

- Earlier we learned that Java supports the standard order of operations for arithmetic operations.
- The full operator precedence chart is as follows (items we have discussed already are highlighted)

Operator	**Description**
( )	**Parenthesis**
(type)	Casting
!	Not
***, /, %**	**Mult, Div, Mod**
+, -	**Add, Sub**
<, <=, >, >=	Comparison
==, !=	Equality
^	XOR
&&	AND
||	OR
=	Assignment

Operator Precedence

- Note that the logical operators do not occupy the same level of the precedence tree.
- This means that you should always group out your boolean expressions using parenthesis to ensure that they are evaluated in the correct order. For example:

```
true || true && false // true
(true || true) && false // false
```

Operator	Description
( )	**Parenthesis**
(type)	Casting
!	Not
***, /, %**	**Mult, Div, Mod**
+, -	**Add, Sub**
<, <=, >, >=	Comparison
==, !=	Equality
^	XOR
&&	AND
||	OR
=	Assignment