

## Java Lessons

## Your First Java Program

- Java class file - same name as class
- Compile
  - `javac classfilename.java`
- Run
  - `java classfilename` (*no extension*)

```
public static void main( String[] args)
```

## Shorthand Assignment Statements

Example:	Equivalent To:
<code>count += 2;</code>	<code>count = count + 2;</code>
<code>sum -= discount;</code>	<code>sum = sum - discount;</code>
<code>bonus *= 2;</code>	<code>bonus = bonus * 2;</code>
<code>time /= rushFactor;</code>	<code>time = time / rushFactor;</code>
<code>change %= 100;</code>	<code>change = change % 100;</code>
<code>amount *= count1 + count2;</code>	<code>amount = amount * (count1 + count2);</code>

## String Indexes

### Display 1-5 String Indexes

The 12 characters in the string "Java is fun." have indexes 0 through 11.

0	1	2	3	4	5	6	7	8	9	10	11
J	a	v	a		i	s		f	u	n	.

*Notice that the blanks and the period count as characters in the string.*

## Comments and a Named Constant

Display 1.8 Comments and a Named Constant

```

1  /**
2  Program to show interest on a sample account balance.
3  Author: Jane Q. Programmer.
4  E-mail Address: janeq@nomemachine.etc.etc.
5  Last Changed: September 21, 2004.
6  */
7  public class ShowInterest
8  {
9      public static final double INTEREST_RATE = 2.5;
10
11     public static void main(String[] args)
12     {
13         double balance = 100;
14         double interest; //as a percent
15
16         interest = balance * (INTEREST_RATE/100.0);
17         System.out.println("On a balance of $" + balance);
18         System.out.println("you will earn interest of $"
19             + interest);
20         System.out.println("All in just one short year.");
21     }
22 }

```

### SAMPLE DIALOGUE

```

On a balance of $100.0
you will earn interest of $2.5
All in just one short year.

```

## Communicating with Users

- Output
  - println, print, printf
- Input
  - Scanner class



<http://www.ces.utexas.edu/~and-daniel/winter/>

## System.out.printf(format, output)

Display 2.1 Format Specifiers for System.out.printf

CONVERSION CHARACTER	TYPE OF OUTPUT	EXAMPLES
d	Decimal (ordinary) integer	%5d %d
f	Fixed-point (everyday notation) floating point	%6.2f %f
e	E-notation floating point	%8.3e %e
g	General floating point (java decides whether to use E-notation or not)	%8.3g %g
s	String	%12s %s
c	Character	%2c %c

## DecimalFormat Class

- java.text.DecimalFormat
- 00.000 is 2 before and three after
- #0.00# is 1 or 2 before and 2 or 3 after
- % at end multiplies by 100 and appends percent sign
- E gives e-notation 00.###E0

## The DecimalFormat Class (Part 1 of 3)

Display 2.5 The DecimalFormat Class

```

1 import java.text.DecimalFormat;
2
3 public class DecimalFormatDemo
4 {
5     public static void main(String[] args)
6     {
7         DecimalFormat pattern0dot000 = new DecimalFormat("00.000");
8         DecimalFormat pattern0dot00 = new DecimalFormat("0.00");
9
10        double d = 12.3456789;
11        System.out.println("Pattern 00.000");
12        System.out.println(pattern0dot000.format(d));
13        System.out.println("Pattern 0.00");
14        System.out.println(pattern0dot00.format(d));
15
16        double money = 19.8;
17        System.out.println("Pattern 00.00");
18        System.out.println("$" + pattern0dot00.format(money));
19
20        DecimalFormat percent = new DecimalFormat("0.00%");

```

(continued)

## The DecimalFormat Class (Part 2 of 3)

Display 2.5 The DecimalFormat Class

```

18 System.out.println("Pattern 0.00%");
19 System.out.println(percent.format(0.308));
20
21 DecimalFormat eNotation1 =
22     new DecimalFormat("#0.##E0");//1 or 2 digits before point
23 DecimalFormat eNotation2 =
24     new DecimalFormat("#00.##E0");//2 digits before point
25
26 System.out.println("Pattern #0.##E0");
27 System.out.println(eNotation1.format(123.456));
28 System.out.println("Pattern 00.##E0");
29 System.out.println(eNotation2.format(123.456));
30
31 double smallNumber = 0.0000123456;
32 System.out.println("Pattern #0.##E0");
33 System.out.println(eNotation1.format(smallNumber));
34 System.out.println("Pattern 00.##E0");
35 System.out.println(eNotation2.format(smallNumber));
36 }

```

(continued)

## The DecimalFormat Class (Part 3 of 3)

Display 2.5 The DecimalFormat Class

SAMPLE DIALOGUE

```

Pattern 00.000
12.346
Pattern 0.00
12.35
Pattern 0.00
$19.80
Pattern 0.00%
30.80%
Pattern #0.##E0
1.2346E2
Pattern 00.##E0
12.346E1
Pattern #0.##E0
12.346E-6
Pattern 00.##E0
12.346E-6

```

The number is always given, even if this requires violating the format pattern.

## Keyboard Input Demonstration (Part 1 of 2)

Display 2.6 Keyboard Input Demonstration

```

1 import java.util.Scanner;
2
3 public class ScannerDemo
4 {
5     public static void main(String[] args)
6     {
7         Scanner keyboard = new Scanner(System.in);
8
9         System.out.println("Enter the number of pods followed by");
10        System.out.println("the number of peas in a pod:");
11        int numberOfPods = keyboard.nextInt();
12        int peasPerPod = keyboard.nextInt();
13
14        int totalNumberOfPeas = numberOfPods*peasPerPod;
15
16        System.out.print(numberOfPods + " pods and ");
17        System.out.println(peasPerPod + " peas per pod.");
18        System.out.println("The total number of peas = "
19            + totalNumberOfPeas);
20    }
21 }

```

(continued)

## Keyboard Input Demonstration (Part 2 of 2)

Display 2.6 Keyboard Input Demonstration

### SAMPLE DIALOGUE 1

Enter the number of pods followed by  
the number of peas in a pod:  
22 10  
22 pods and 10 peas per pod.  
The total number of peas = 220

The numbers that are  
input must be  
separated by  
whitespace, such as  
one or more blanks.

### SAMPLE DIALOGUE 2

Enter the number of pods followed by  
the number of peas in a pod:  
22  
10  
22 pods and 10 peas per pod.  
The total number of peas = 220

A line break is also  
considered whitespace and  
can be used to separate the  
numbers typed in at the  
keyboard.

## Another Keyboard Input Demonstration (Part 1 of 3)

Display 2.7 Another Keyboard Input Demonstration

```
1 import java.util.Scanner;
2 public class ScannerDemo2
3 {
4     public static void main(String[] args)
5     {
6         int n1, n2;
7         Scanner scannerObject = new Scanner(System.in);
8
9         System.out.println("Enter two whole numbers");
10        System.out.println("seperated by one or more spaces:");
11
12        n1 = scannerObject.nextInt();
13        n2 = scannerObject.nextInt();
14        System.out.println("You entered " + n1 + " and " + n2);
15
16        System.out.println("Next enter two numbers.");
17        System.out.println("Decimal points are allowed.");
18    }
19 }
```

Creates an object of  
the class Scanner  
and names the object  
scannerObject.

Reads one int from the  
keyboard.

(continued)

## Another Keyboard Input Demonstration (Part 2 of 3)

Display 2.7 Another Keyboard Input Demonstration

```
15 double d1, d2;
16 d1 = scannerObject.nextDouble();
17 d2 = scannerObject.nextDouble();
18 System.out.println("You entered " + d1 + " and " + d2);
19 System.out.println("Next enter two words:");
20 String word1 = scannerObject.next();
21 String word2 = scannerObject.next();
22 System.out.println("You entered '" +
23     word1 + "' and '" + word2 + "'");
24 String junk = scannerObject.nextLine(); //To get rid of '\n'
25 System.out.println("Next enter a line of text:");
26 String line = scannerObject.nextLine();
27 System.out.println("You entered: '" + line + "'");
28 }
29 }
```

Reads one double from  
the keyboard.

Reads one word from  
the keyboard.

This line is  
explained in the  
PIfall section  
"Dealing with the  
Line Terminator,  
'\n'"

(continued)

## Another Keyboard Input Demonstration (Part 3 of 3)

Display 2.7 Another Keyboard Input Demonstration

### SAMPLE DIALOGUE

Enter two whole numbers  
separated by one or more spaces:  
42 43  
You entered 42 and 43  
Next enter two numbers.  
A decimal point is OK.  
9.99 57  
You entered 9.99 and 57.0  
Next enter two words:  
jelly beans  
You entered "jelly" and "beans"  
Next enter a line of text:  
Java flavored jelly beans are my favorite.  
You entered "Java flavored jelly beans are my favorite."

### Pitfall: Dealing with the Line Terminator, '\n'



- The method `nextLine` of the class `Scanner` reads the remainder of a line of text starting wherever the last keyboard reading left off
- This can cause problems when combining it with different methods for reading from the keyboard such as `nextInt`

### Pitfall: Dealing with the Line Terminator, '\n'



- Given the code,
  - `Scanner keyboard = new Scanner(System.in);`
  - `int n = keyboard.nextInt();`
  - `String s1 = keyboard.nextLine();`
  - `String s2 = keyboard.nextLine();`
- and the input,
  - 2
  - Heads are better than
  - 1 head.
- What are the values of `n`, `s1`, and `s2`?

## Let's Build Some Programs



### Building the OO program



- Create classes that are single purpose
- Create instances of those classes where needed to do the 'work' of the program
- Pass messages between instances

## Pig Latin translator

```
Word
Word(String)
boolean beginsWithVowel()
boolean beginsWithBlend()
String firstLetterAtEnd()
String firstTwoAtEnd()
```

Assume word begins with a single consonant



## One Solution

```
import java.util.Scanner;

public class PigLatinTranslator {

    public static void main(String[] args)
    {
        String ending = "ay";
        Scanner keyboard = new Scanner(System.in);
        String theWord = keyboard.next();
        Word toBeTranslated = new Word(theWord);
        String pigLatinWord =
        toBeTranslated.firstLetterAtEnd() + ending;

    }
}
```



## Flow of Control

- Flow of control refers to branching and looping mechanisms
- Java has several branching mechanisms: `if-else`, `if`, and `switch`
- Java has three types of loop statements: the `while`, `do-while`, and `for` statements
- Most branching and looping statements are controlled by *Boolean* expressions
  - A Boolean expression evaluates to either true or false
  - The primitive type `boolean` may only take the values true or false



## Branching with an `if-else` Statement

- An `if-else` statement chooses between two alternative statements based on the value of a Boolean expression

```
if (Boolean_Expression)
    Yes_Statement
else
    No_Statement
```
- *Compound Statement*: A branch statement that is made up of a list of statements
  - A compound statement must always be enclosed in a pair of braces `{ }`
  - A compound statement can be used anywhere that a single statement can be used



## Compound Statements

```
if (myScore > your Score)
{
    System.out.println("I win!");
    wager = wager + 100;
}
else
{
    System.out.println
        ("I wish these were golf
        scores.");
    wager = 0;
}
```

## Multiway if-else Statement

- The multiway if-else statement is simply a normal if-else statement that nests another if-else statement at every else branch.

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

else if (Boolean_Expression_n)
    Statement_n
else
    Statement_For_All_Other_Possibilities
```

## Pig Latin translator - 2

```
Word
Word(String)
boolean beginsWithVowel()
boolean beginsWithBlend()
String firstLetterAtEnd()
String firstTwoAtEnd()
```

Three possibilities: word begins with a vowel, a consonant or a blend

## A Solution

```
...//after the last one
String pigLatinWord = null;
if (toBeTranslated.beginsWithBlend())
{
    pigLatinWord = toBeTranslated.firstTwoAtEnd() + ending;
}
if (toBeTranslated.beginsWithVowel())
{
    pigLatinWord = toBeTranslated + ending;
}
else
{
    pigLatinWord = toBeTranslated.firstLetterAtEnd() +
    ending;
}
```

## The switch Statement



- The `switch` statement is the only other kind of Java statement that implements multiway branching
  - When a `switch` statement is evaluated, one of a number of different branches is executed
  - The choice of which branch to execute is determined by a controlling expression enclosed in parentheses after the keyword `switch`
    - The controlling expression must evaluate to a `char`, `int`, `short`, or `byte`

## The switch Statement



- Each branch statement in a `switch` statement starts with the reserved word `case`, followed by a constant called a `case` label, followed by a colon, and then a sequence of statements
  - Each `case` label must be of the same type as the controlling expression
  - `Case` labels need not be listed in order or span a complete interval, but each one may appear only once
  - Each sequence of statements may be followed by a `break` statement (`break;`)

## The switch Statement



- There can also be a section labeled `default`:
  - The `default` section is optional, and is usually last
  - Even if the `case` labels cover all possible outcomes in a given `switch` statement, it is still a good practice to include a `default` section
    - It can be used to output an error message, for example
- When the controlling expression is evaluated, the code for the `case` label whose value matches the controlling expression is executed
  - If no `case` label matches, then the only statements executed are those following the `default` label (if there is one)

## The switch Statement



- The `switch` statement ends when it executes a `break` statement, or when the end of the `switch` statement is reached
  - When the computer executes the statements after a `case` label, it continues until a `break` statement is reached
  - If the `break` statement is omitted, then after executing the code for one `case`, the computer will go on to execute the code for the next `case`
  - If the `break` statement is omitted inadvertently, the compiler will not issue an error message

## The switch Statement

```
switch (Controlling_Expression)
{
    case Case_Label_1:
        Statement_Sequence_1
        break;
    case Case_Label_2:
        Statement_Sequence_2
        break;
        :
    case Case_Label_n:
        Statement_Sequence_n
        break;
    default:
        Default_Statement_Sequence
        break;
}
```



## Pig Latin translator - 3

```
Word
Word(String)
boolean beginsWithVowel()
boolean beginsWithBlend()
String firstLetterAtEnd()
String firstTwoAtEnd()
char startsWith() //returns v, c or b
```

Three possibilities: word begins with a vowel, a consonant or a blend



## A Solution

```
switch (toBeTranslated.startsWith())
{
    case 'v':
        pigLatinWord = toBeTranslated + ending;
        break;
    case 'b':
        pigLatinWord = toBeTranslated.firstTwoAtEnd() +
            ending;
        break;
    case 'c':
        pigLatinWord = toBeTranslated.firstLetterAtEnd() +
            ending;
        break;
    default:
        pigLatinWord = ending;
        break;
}
```



## The Conditional Operator

- The conditional operator is a notational variant on certain forms of the if-else statement
  - Also called the ternary operator or arithmetic if
  - The following examples are equivalent:

```
if (n1 > n2)    max = n1;
else           max = n2;
```

vs.

```
max = (n1 > n2) ? n1 : n2;
```
  - The expression to the right of the assignment operator is a conditional operator expression
  - If the Boolean expression is true, then the expression evaluates to the value of the first expression (n1), otherwise it evaluates to the value of the second expression (n2)



## Pitfall: Using == with Strings



- The equality comparison operator (==) can correctly test two values of a primitive type
- However, when applied to two objects such as objects of the `String` class, == tests to see if they are stored in the same memory location, not whether or not they have the same value
- In order to test two strings to see if they have equal values, use the method `equals`, or `equalsIgnoreCase`

```
string1.equals(string2)
string1.equalsIgnoreCase(string2)
```

## Lexicographic and Alphabetical Order



- Lexicographic ordering is the same as ASCII ordering, and includes letters, numbers, and other characters
  - All uppercase letters are in alphabetic order, and all lowercase letters are in alphabetic order, but all uppercase letters come before lowercase letters
  - If `s1` and `s2` are two variables of type `String` that have been given `String` values, then `s1.compareTo(s2)` returns a negative number if `s1` comes before `s2` in lexicographic ordering, returns zero if the two strings are equal, and returns a positive number if `s2` comes before `s1`
- When performing an alphabetic comparison of strings (rather than a lexicographic comparison) that consist of a mix of lowercase and uppercase letters, use the `compareToIgnoreCase` method instead

## Building Boolean Expressions



- When two Boolean expressions are combined using the "and" (&&) operator, the entire expression is true provided both expressions are true
  - Otherwise the expression is false
- When two Boolean expressions are combined using the "or" (||) operator, the entire expression is true as long as one of the expressions is true
  - The expression is false only if both expressions are false

## Building Boolean Expressions



- Any Boolean expression can be negated using the `!` operator
  - Place the expression in parentheses and place the `!` operator in front of it
- Unlike mathematical notation, strings of inequalities must be joined by `&&`
  - Use `(min < result) && (result < max)` rather than `min < result < max`

## Short-Circuit and Complete Evaluation



- Java can take a shortcut when the evaluation of the first part of a Boolean expression produces a result that evaluation of the second part cannot change
- This is called short-circuit evaluation or lazy evaluation
  - For example, when evaluating two Boolean subexpressions joined by `&&`, if the first subexpression evaluates to false, then the entire expression will evaluate to false, no matter the value of the second subexpression
  - In like manner, when evaluating two Boolean subexpressions joined by `||`, if the first subexpression evaluates to true, then the entire expression will evaluate to true

## Short-Circuit and Complete Evaluation



- There are times when using short-circuit evaluation can prevent a runtime error
    - In the following example, if the number of `kids` is equal to zero, then the second subexpression will not be evaluated, thus preventing a divide by zero error
    - Note that reversing the order of the subexpressions will **not** prevent this
- ```
if ((kids !=0) && ((toys/kids) >=2)) . . .
```
- Sometimes it is preferable to always evaluate both expressions, *i.e.*, request complete evaluation
    - In this case, use the `&` and `|` operators instead of `&&` and `||` (learn the Precedence and Associativity Rules)

## Loops



- Loops in Java are similar to those in other high-level languages
- Java has three types of loop statements: the `while`, the `do-while`, and the `for` statements
  - The code that is repeated in a loop is called the **body** of the loop
  - Each repetition of the loop body is called an **iteration** of the loop

## while statement



- A `while` statement is used to repeat a portion of code (*i.e.*, the loop **body**) based on the evaluation of a Boolean expression
  - The Boolean expression is checked before the loop **body** is executed
    - When false, the loop **body** is not executed at all
  - Before the execution of each following **iteration** of the loop **body**, the Boolean expression is checked again
    - If true, the loop **body** is executed again
    - If false, the loop statement ends
  - The loop **body** can consist of a single statement, or multiple statements enclosed in a pair of braces ( `{ }` )

## while Syntax

```
while (Boolean_Expression)
    Statement
```

OR

```
while (Boolean_Expression)
{
    Statement_1
    Statement_2
    :
    Statement_Last
}
```

## do-while Statement

- A do-while statement is used to execute a portion of code (i.e., the loop **body**), and then repeat it based on the evaluation of a Boolean expression
  - The loop **body** is executed **at least once**
    - The Boolean expression is checked **after** the loop **body** is executed
  - The Boolean expression is checked after each **iteration** of the loop **body**
    - If true, the loop **body** is executed again
    - If false, the loop statement ends
    - Don't forget to put a semicolon after the Boolean expression
  - Like the while statement, the loop **body** can consist of a single statement, or multiple statements enclosed in a pair of braces

## do-while Syntax

```
do
    Statement
while (Boolean_Expression);
```

OR

```
do
{
    Statement_1
    Statement_2
    :
    Statement_Last
} while (Boolean_Expression);
```