Scope

❖ Every variable and parameter that is declared has a scope which is the area of a program where the variable can be used

❖ A variable's scope begins at its declaration and continues until the closing curly brace "`}`" of the scope to which it was declared. (The closing curly brace could be for a method, block of code, etc.)

❖ Local Scope:
- Variables declared in methods are "local" variables
- Parameters are variables to a method, and therefore are local to the method
- Local variables cannot be accessed outside of the method to which they are declared

❖ Nested Scope:
- Defined: One scope WITHIN another scope
- All method's scopes are nested inside a class' scope.
- Compound statements (e.g., if, while, else blocks) form new scopes that are nested inside a method's scope (and possibly further nested inside other compound statements' scopes)
- Variables declared inside a scope (whether nested or not) are NOT accessible outside that scope
- Variables declared in an outer scope are accessible in any inner scopes

- class data members belong to the class' scope and are available to all inner scopes (i.e., methods). However, if a data member is non-static, it is NOT accessible by static methods of the class.

❖ Example #1:

```
class Parameters {
  public static void main (String[] args) {
      int x = 0;
      System.out.println (“x in main: ” + x);

      whatHappensToX (x);
      System.out.println (“x in main: ” + x);
  }

  public static void whatHappensToX (int x) {
      x = 10;
      System.out.println (“x in whatHappensToX: ” + x);
  }
}
```

❖ Example #2:

```
class Scope {
  private static int foo = 3;
  private static int goo = 5;

  public static void main (String[] args) {
      int foo = 5;
      int goo = 3;

      for (int i = 0; i < 4; i++) {
          foo = foo(foo);
          print (i, foo, goo);
          goo = goo(goo);
          print (i, foo, goo);
      }
  }

  private static int foo (int goo) {
      goo = goo + foo;
      foo = foo – goo;
      return goo;
  }

  private static int goo (int foo) {
      foo = foo + goo;
      goo = goo – foo;
      return foo;
  }

  private static void print (int i, int f, int g) {
      System.out.println  (“i = ” + i + “, f = ” + f +
                           “, g = ” + g + “, foo = ” +
                           foo + “, goo = ” + goo);
  }
```

Character Primitive Type

❖ Like arithmetic primitive data types, except the value is a single character

❖ Character Literals (aka Character Constants): symbols enclosed in single quotes

e.g.: **‘a’ ‘1’ ‘$’ ‘+’ ‘Z’**

❖ Syntax for Character Variables:

Declaration: **char <variable name>;**

Assignment: **<variable name> = ‘<character>’;**

- **char** is a Java reserved word
- **<character>** is any unicode character
- **‘<character>’** can be substituted for any expression that evaluates to a **char**.
- The open quote **‘** and the close quote **’** must be on the same line

❖ Examples:

```
char c;
c = ‘a’

char charVal = ‘#’;

char menuChoice = getMenuChoice ();
/* getMenuChoice() returns a char */

char c1, c2, c3, c4 = ‘8’
```

❖ Unicode: The international standard coding scheme for characters (Contains the ASCII coding scheme)

Character Primitive Type...continued

❖ Characters have a numerical value.

- Can do simple arithmetic on characters

```
char c = ‘a’ + 1;    // the value of c is ‘b’
```

- Can use logic on characters

```
if (‘a’ < ‘b’) {...}
```

- Can cast between characters and integers

```
print (“The ASCII code of c is “ + (int)‘c’);

print (“The char with ASCII code of 35 is “ +
          (char) 35);
```

❖ Write a code fragment that prints to standard output (System.out) the alphabet in either all lowercase or all uppercase letters.

```
for (char c = ‘a’; c <= ‘z’; c++) {
    if (c != ‘z’) {
        System.out.print (c + “, ”);
    }
    else {
        System.out.println (c);
    }
}
```

Strings

❖ String Literals

- Any set of characters within double quotation marks
- Can have any number of characters within a string
- The open quote **“** and the close quote **”** must be on the same line
- The string literal begins with the first character after the open quote and ends with the last character before the end quote
- Examples:

  A typical String: **“I love Java!”**

  A zero character String: **“”**

- Certain characters need an escape sequence in order to be a part of the string. An escape sequence is a backslash followed by a character.

\” prints “ within the string
**** prints \ within the string
\’ prints ‘ within the string
\n prints a carriage return (aka newline) within the string
\t prints a tab space within the string

Example:

```
print (“This is a \”quote\”\nand this” +
       “is a backslash\t\\”);
```

output:

```
This is a “quote”
and this is a backslash   \
```

Strings...continued

❖ Concatenation Operator: **+**

One can concatenate (i.e., merge) the following using the concatenation operator:

- two Strings: **"Java is " + "Fun!";**
- a String and a Primitive Type: **"The sum of 1+2 is " + (1+2);**
- a String and another Object **"This is a Fish: " + fish1;**

❖ **toString()** Method

ALL objects inherit a **toString()** method from the Object class. This method "converts" an object into a string. Whenever an object is concatenated with a String (or used as an argument to such methods as **printLine()**), the object's **toString()** method is called.

When creating an instantiable class, it is always a good idea to define a **toString()** method that will override the inherited **toString()** method.

Example of a toString() method:

```
public String toString () {
    return ("Name:  " + fishName + "; Food: " +
    foodPoints + " ; Position: " + position);
}
```

Strings...continued

- String Objects

Strings (including String Literals) are all instances (i.e., objects) of the String class defined in the **`java.lang`** package. Therefore, they are declared and created in the same way as other objects:

Declaration: **`String <object name>;`**

Creation: **`<object name> = new String (<String Literal>);`**

Because String objects are used so frequently, a shortcut syntax for declaring and creating is included in Java syntax:

`String <object name> = <String Literal>;`

Examples:

```
String s;
s = new String (“Hello World!”);

String name = new String (“David Schneider”);

String joke = “There once was a man from
Nantucket...”;

String input = inBox.getString (“Enter a string: “);
```

Strings...continued

- The individual characters of a String are indexed from 0 to length of the String –1. For example:

```
String s = "I Love Java";
```

0	1	2	3	4	5	6	7	8	9	10
I		**L**	**o**	**v**	**e**		**J**	**a**	**v**	**a**

- String objects are "immutable". That is, you cannot change the value of a String object once it has been created. If you do attempt to change the value, a brand new String object is made with the new value.

```
String s1 = "Hello!";
String s2 = s1;
s2 = "Java is fun!";
```

- The String class contains many methods for manipulating String objects.

```
char charAt (int index) { ... }

boolean equals (Object obj) { ... }

boolean equalsIgnoreCase (Object obj) { ... }

int compartTo ( String str ) { ... }

int length () { ... }

String substring (int beginIdx, int endIdx) { ... };
```

(See the Javadocs for all of the methods defined for the String class.)

Write a method that returns a given string s in reverse. For example, if the method was given the string “I love java”, it would return the string “avaj evol I”.

```
public String reverseString (String origString) {
    String newString = “”;
    for (int i = origString.length()-1; i >= 0; i--) {
        newString = newString + origString.charAt(i);
    }
    return newString;
}
```

Pig Latin Example

General Rules:

I. If word begins with one or more consonants:
- Remove the consonant(s) and append them to the end of the word
- Append the suffix "ay" to the end of the word

II. If word begins with a vowel:
- Append the suffix "ay" to the end of the word

III. Hyphenated words count as one word

Examples:

Original Word	Word new to Pig Latin
subway	ubwaysay
chicken	ickenchay
elephant	elephantay

Write a method that takes as a parameter a String and converts that String to Pig Latin. The method should both print the original and new String to standard output (i.e., **`System.out.print()`**) AND return the new String. Assume that the String that is passed in as a parameter is only ONE word.

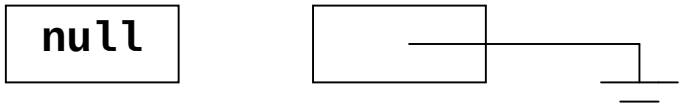
HINT: The **`charAt()`** and **`substring()`** methods of the String class should probably be used

```
public String convertToPigLatin (String str)
{
    int i = 0;

    while (s.charAt(i) != ‘a’ || s.charAt(i) != ‘A’
        || s.charAt(i) != ‘e’ || s.charAt(i) != ‘E’
        || s.charAt(i) != ‘i’ || s.charAt(i) != ‘I’
        || s.charAt(i) != ‘o’ || s.charAt(i) != ‘O’
        || s.charAt(i) != ‘u’ || s.charAt(i) != ‘U’)
    {
        i++;
    }

    String newStr;

    if (i == 0)
    {
        newStr = str + “ay”;
    }

    else
    {
        newStr = str.substring (i, str.length() – 1)
                 + str.substring (0, i – 1)
                 + “ay”;
    }

    System.out.println (“Original String: “ + str);
    System.out.println (“In Pig Latin: “ + newStr);

    return newStr;

}
```

null

The word **`null`** is a Java Reserved word that is used to mean that an object has been declared, but has not been created (i.e., it does not refer to any object in memory)

Aquarium aquarium;



class StringBuffer

Because Strings are immutable, minor changes to the contents of a String can be expensive. For example, appending the letter 'A' to the end of a String object repeatedly is very expensive because a new String object is formed after each 'A' is appended.

```
String result = "Many A's: ";
for (int i = 0; i < n; i++) {
    result = result + 'A';
}
```

Rather, a StringBuffer object can be created that represents the same sequence of characters as a String, but allows the manipulation of its value directly (without having to create an entire new object).

```
String result = "Many A's: ";
StringBuffer tempResult = new StringBuffer(result);

for (int i = 0; i < n; i++)
{
    tempResult.append ('A');
}
```

❖ The StringBuffer class contains many methods for manipulating StringBuffer objects.

```
StringBuffer append ( X )

StringBuffer delete (int start, int end)

StringBuffer deleteCharAt (int index)

StringBuffer insert (int offset, X )

StringBuffer replace (int start, int end, String str)

StringBuffer reverse ()

StringBuffer setCharAt (int index, char ch)

String toString ()
```

(See the Javadocs for all of the methods defined for the StringBuffer class.)

Parameter Passing

The contents of the argument are copied into the parameter
(i.e., “pass-by-value”)

For primitive data types: the primitive value is copied

For reference data types: the address is copied

Implication for primitive data types

- changing the parameter does NOT affect the argument

Implications for reference data types

- the object is NOT copied; rather, it is shared
  (the argument and the parameter are “aliases”)
- changing the parameter’s address does NOT affect the argument’s address
- changing the parameter’s object DOES change the argument’s object

Parameter Passing Example

```
public static void main (String[] args)
{
    int i1 = 5;
    Student s1 = new Student(4.0);
    function (i1, s1);
}

public static void function (int i2, Student s2)
{
  i2 = 8;
  s2.setGPA(3.2);
  s2 = new Student(3.2);
}
```

Returning Values

The value of the expression in the return statement is returned to the message (i.e., “pass-by-value”)

For primitive data types: the primitive value returned

For reference data types: the address is returned

NOTE: local variables, including parameters, are destroyed when the method finishes execution

Implication

- A locally created object can survive if it is returned OR if it is stored in an object as a data member.

Returning Values Example

```
public static void main (String[] args) {
    int i1 = getInteger();
    String s1 = getString();
}

public static int getInteger() {
    int i2 = 6;
    return i2;
}

public static String getString() {
    String s2 = “Java”;
    return s2;
}
```