Object-Oriented Design II

- Objectives - when we have completed this set of notes, you should be familiar with:
  - writing interfaces
  - using interfaces in the Java API including Comparable and Iterator
  - method and constructor overloading
  - method design
  - Passing Objects to Methods

Interfaces

- A Java *interface*, in one of its common forms, consists of abstract methods and/or constants
  - An *abstract method* is a method header without a method body:

    ```
    public abstract double getPerimeter();
    ```

  - The abstract reserved word can be left off because instance methods in an interface are assumed to be abstract:

    ```
    public double getPerimeter();
    ```

- An interface can be used to establish a set of methods that a class will implement

Interfaces

interface is a reserved word

```
public interface TwoDShape {

     public double getNumberSides();

     public double getPerimeter();

}
```

The abstract methods in an interface are not given a definition (body); an interface may also contain constants

A semicolon immediately follows each method header

Interfaces

- An interface cannot be instantiated
- Methods in an interface have public visibility by default so the *public* modifier is optional
- A class formally implements an interface:
  - By stating so in the class header

    ```
    public class Triangle implements TwoDShape
    ```

    - The Triangle class must now have a getNumberSides and a getPerimeter method
  - And then by providing a body (or implementation) for each abstract method in the interface

Interfaces

- A class that implements an interface can implement other methods as well
  - See Triangle.java and Rectangle.java, which both implement the TwoDShape interface
- In addition to (or instead of) abstract methods, an interface can contain constants
- When a class implements an interface, it gains access to all of its constants

Multiple Interfaces

- A class can implement multiple interfaces
- The interfaces are listed in the `implements` clause
- The class must implement all methods in all interfaces listed in the header

```
class ManyThings implements Interface1, Interface2
{
   // all methods of both interfaces
}
```

Comparable Interface

- The Java standard class library contains many helpful interfaces

- The `Comparable` interface contains one abstract method called `compareTo`, which is used to compare two objects

- Recall the compareTo method of String:

  - The compareTo method is defined in the String class to compare objects based on lexographic order

```
str1.compareTo(str2);
```

The Comparable Interface

- Any class can implement the `Comparable` interface to define the natural ordering of its objects, making the following method call possible:

```
obj1.compareTo(obj2); // return type is int
```

- The `int` value returned by`compareTo` should be:
    - negative if `obj1` is less than `obj2`
      (think: if `obj1` comes before `obj2`)
    - 0 if they are equal
    - positive if `obj1` is greater than `obj2`
      (think: if `obj1` comes after `obj2`)

The Comparable Interface

- The customer/designer/programmer decides what constitutes the natural ordering for the objects of a class (what the makes one object less than, greater than, or equal to another)
- For example, you may define the `compareTo` method of an `Employee` class to order employees by name (alphabetically) or by employee number, smallest to largest
- When Rectangle.java implements the Comparable interface, the compareTo method is based on area, smallest to largest

Interfaces

- You could implement the `compareTo` method without implementing the `Comparable` interface, but you would limit the functionality
  - For example, Collections.sort relies on objects being Comparable (i.e., the class of the objects to be sorted implements the Comparable interface)
  - If you try to use Collections.sort on an ArrayList of Rectangles, it will generate a compile error **if the Comparable interface is not implemented** (even if you have defined compareTo and it compiled okay)
  - Run RectangleArrayListSorter.java and then again after commenting out: implements Comparable<Rectangle> in Rectangle.java

The Iterator Interface

- An iterator is an object that provides a means of processing a collection of objects one at a time
- An iterator is created formally by implementing the `Iterator` interface, which contains three methods
  - The `hasNext` method returns a boolean result – true if there are items left to process
  - The `next` method returns the next object in the iteration
  - The `remove` method (optional) removes the object most recently returned by the `next` method

The Iterator Interface

- An example of a class that implements Iterator:
  - Scanner: iterates through “tokens” based on a delimiter (default delimiter is whitespace)
- Although we will not implement the Iterator interface in our own classes, we do call its methods when we use classes that implement Iterator interface
- When you take the data structures course, you will likely implement the Iterator interface in classes representing data structures such as lists and trees

Method Overloading

- *Method overloading* is the process of giving a single method name multiple definitions
- If a method is overloaded, the method name is not sufficient to determine which method is being called
- The *signature* of each overloaded method must be unique
- The signature includes the method's name and its parameters (number, type, and order), but it does not include the return type

Method Overloading

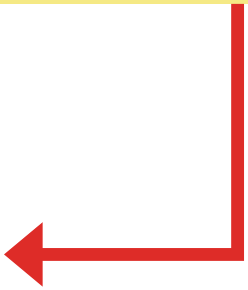
- The compiler determines which method is being invoked by analyzing the parameters

```
 double tryMe(int x)
 {
    return x + .375;
 }

double tryMe(int x, double y)
{
   return x*y;
}
```

Invocation

```
result = tryMe(25, 4.32)
```



Method Overloading

- The `println` method for the PrintStream out in the System class is overloaded:

```
println (String s)
println (int i)
println (double d)
```

and so on...

- The following lines invoke different versions of the `println` method:

```
System.out.println ("The total is:");
System.out.println (total);
```

Overloading Notes

- Remember, the return type of the method is not part of the signature; i.e., overloaded methods cannot differ only by their return type

- When you compile your program, the compiler must find the class and matching method signature for each method call in your program; otherwise, your program will not compile.
  - The class and matching method signature may be found in your program or in another class imported by your program (e.g., from the Java API)

Constructor Overloading

- **Constructors** can be overloaded as well; for example, if we had a class Book, we might have the following constructors:

```
Book()
Book(String titleIn)
Book(String titleIn, String authorIn)
```

- Many classes in the JDK API have multiple constructors. For the String class:

```
String(String original)
String(char[] value)
 . . .  plus 6 other constructors
```

Method Design

- An *algorithm* is a step-by-step process for solving a problem
- Non-programming examples of algorithms: a recipe, travel directions
- An algorithm may be expressed in *pseudocode*, a mixture of code statements and English that communicate the steps to take
- Every Java method implements an algorithm that determines how the method accomplishes its goals

Method Decomposition

- A method should be relatively small, so that it can be understood as a single entity
- A potentially large method should be decomposed into several smaller methods as needed for clarity
- A public service method of an object may call one or more private support methods to help it accomplish its goal
- Support methods might call other support methods if appropriate

Method Decomposition

- Let's look at an example that requires method decomposition – translating English into Pig Latin
- Pig Latin is a language in which each word is modified by moving the initial sound of the word to the end and adding "ay"
- Words that begin with vowels have the "yay" sound added on the end
- Examples

book	→	ookbay	table	→	abletay
item	→	itemyay	chair	→	airchay

Method Decomposition

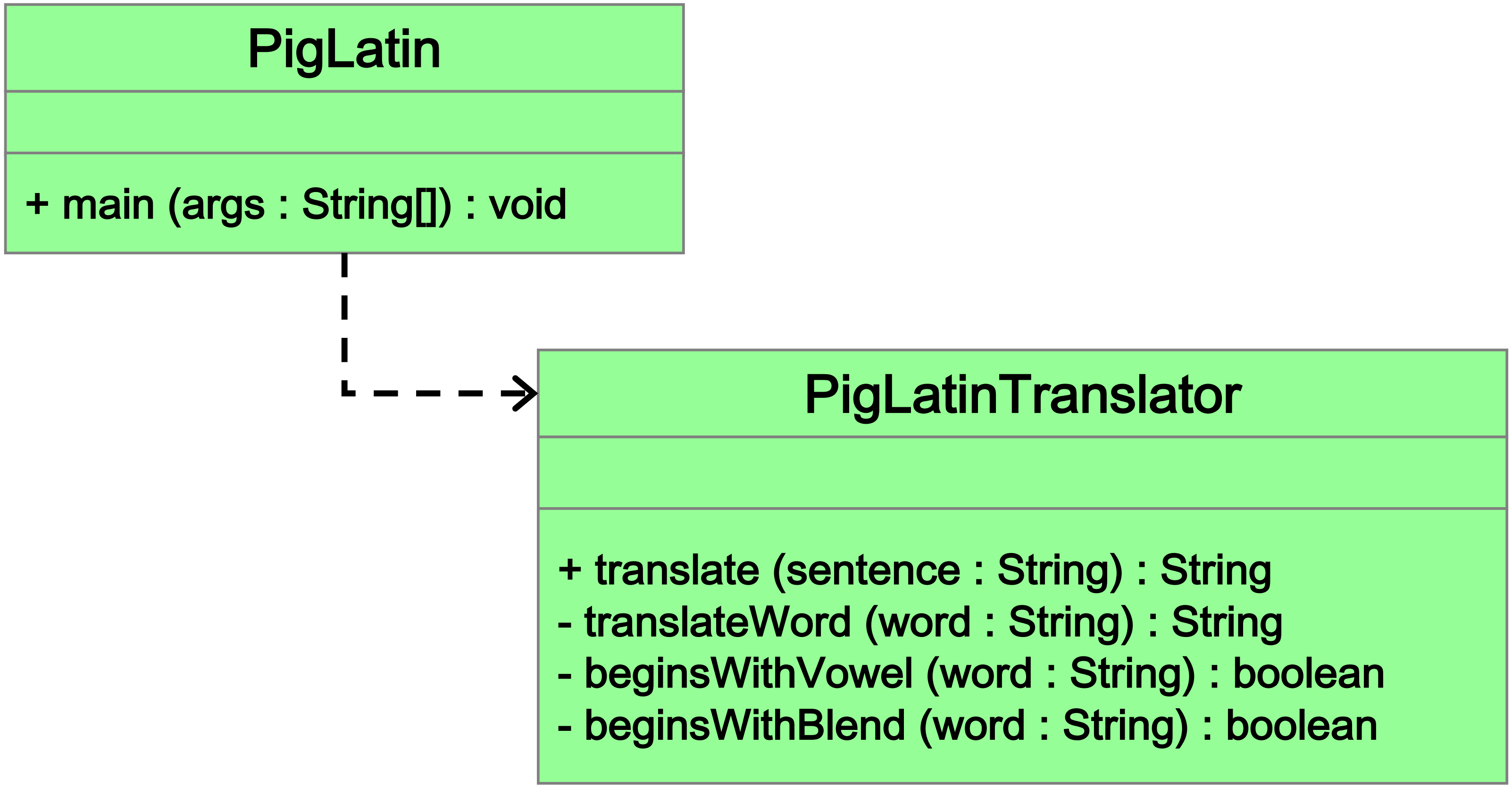
- The primary objective (translating a sentence) is too complicated for one method to accomplish
- Therefore we look for natural ways to decompose the problem/solution
- Translating a sentence can be decomposed into the process of translating each word
- The process of translating a word can be separated into translating words that:
  - begin with vowels
  - begin with consonant blends (sh, cr, th, etc.)
  - begin with single consonants

Method Decomposition

- See PigLatin.java
- See PigLatinTranslator.java

- In a detailed UML class diagram, the accessibility of a field or method can be shown using special characters
  - Public class members are preceded by a plus sign
  - Private class members are preceded by a minus sign
- In the UML class diagram generated by jGRASP, the details are shown in the Info tab when a class is selected
  - Access modifier follows the field or method name

Class Diagram for Pig Latin


PigLatin
+ main (args : String[]) : void
PigLatinTranslator
+ translate (sentence : String) : String
- translateWord (word : String) : String
- beginsWithVowel (word : String) : boolean
- beginsWithBlend (word : String) : boolean

Objects as Parameters

- Another important issue related to method design involves parameter passing
- Parameters in a Java method are *passed by value*
- A copy of the actual parameter (the value passed in) is stored into the formal parameter (in the method header)
- Therefore passing parameters is similar to an assignment statement
- When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other

Passing Objects to Methods

- What a method does with an object parameter may or may not have an effect on the object outside the method

- See ParameterTester.java
- See ParameterModifier.java
- See Num.java

- Note the difference between changing the internal state of an object versus changing which object a reference variable points to