7B. 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
  - types of testing

Interfaces

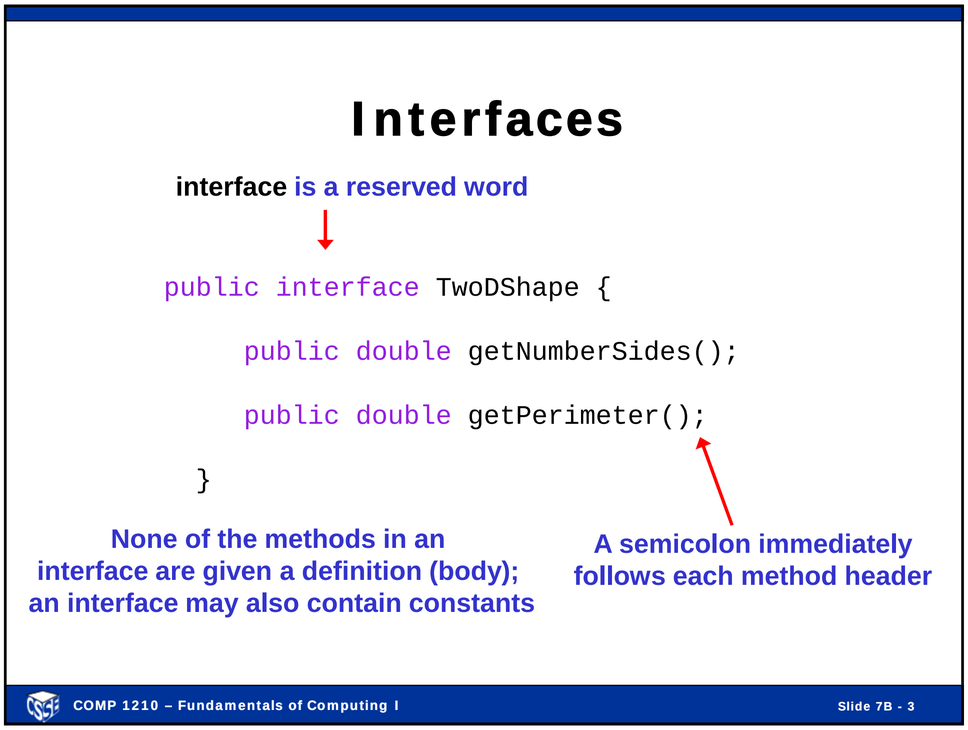
- A Java *interface* consists of abstract methods and constants
  - An *abstract method* is a method header without a method body:

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

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

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

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



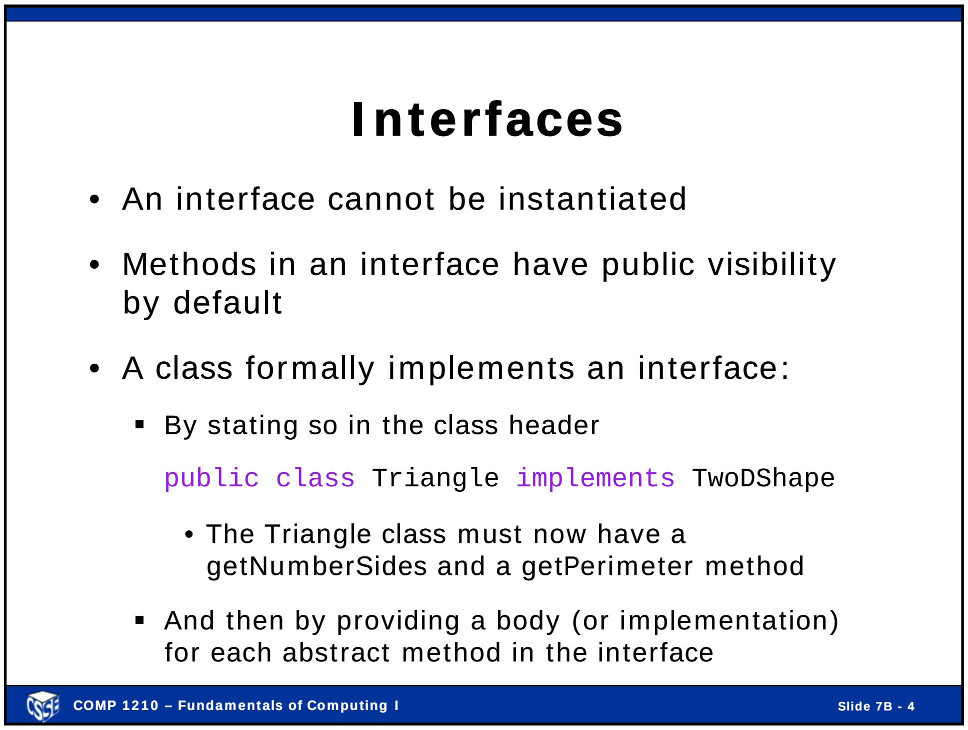
Interfaces

interface is a reserved word

```
public interface TwoDShape {

    public double getNumberSides();

    public double getPerimeter();

}
```

None of the methods in an interface are 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
- 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

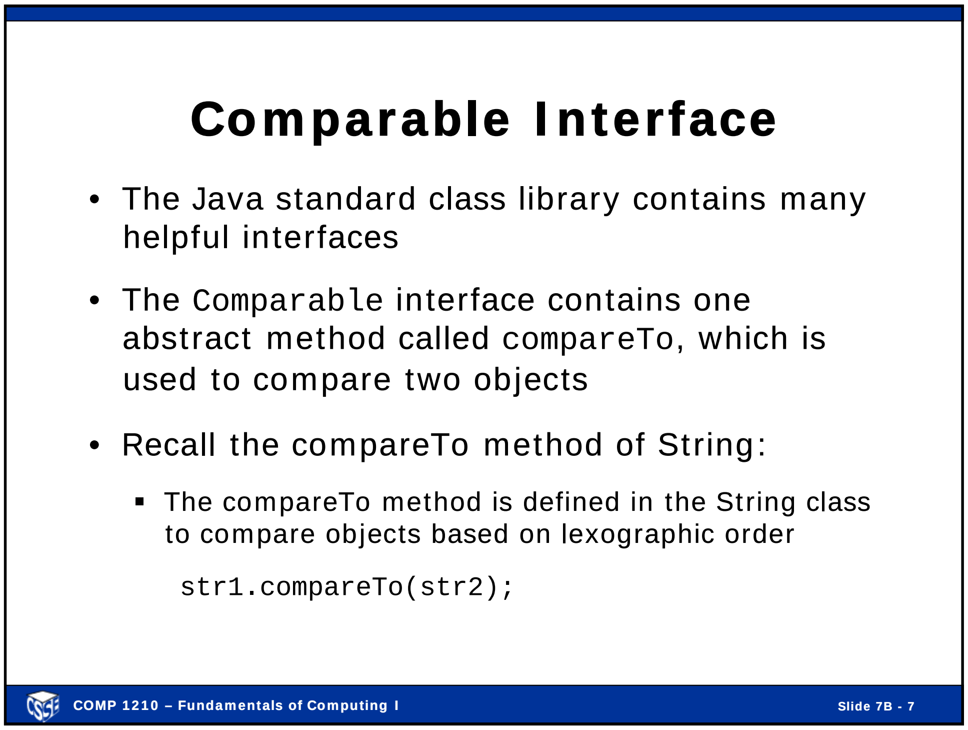
Interfaces

- 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 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 (see Rectangle.java)

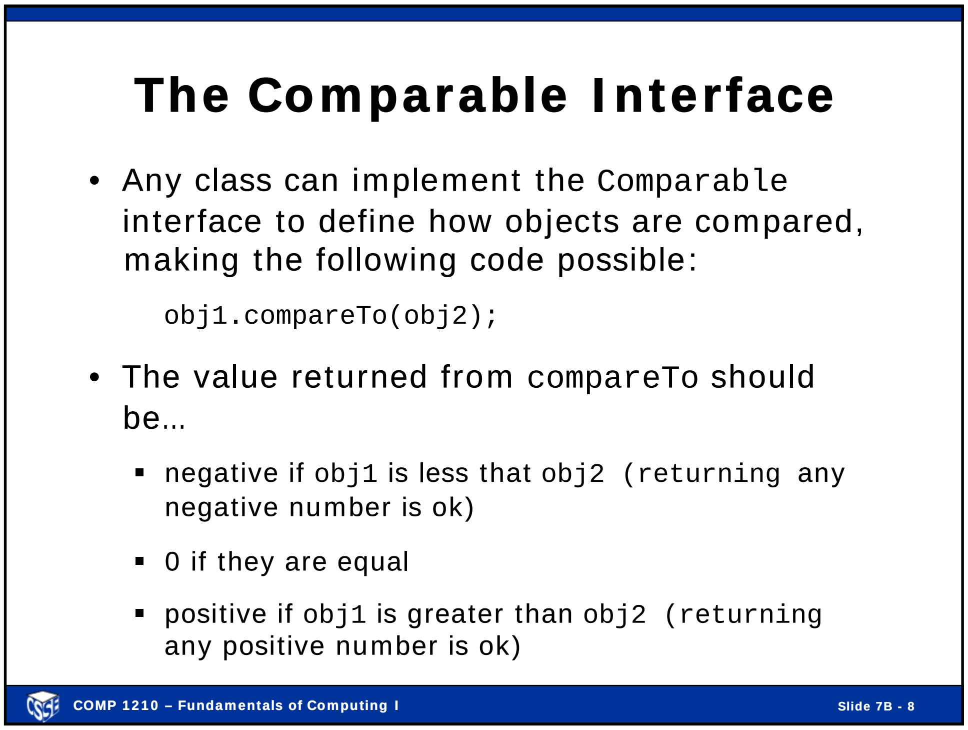
```
class ManyThings implements Interface1, Interface2
{
   // all methods of both interfaces
}
```



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 how objects are compared, making the following code possible:

```
obj1.compareTo(obj2);
```

- The value returned from compareTo should be…
  - negative if `obj1` is less that `obj2` (returning any negative number is ok)
  - 0 if they are equal
  - positive if `obj1` is greater than `obj2` (returning any positive number is ok)

The Comparable Interface

- The programmer decides what makes one object less than another
- For example, you may define the `compareTo` method of an `Employee` class to order employees by name (alphabetically) or by employee number
- The compareTo method for Rectangle.java is based on area

The Iterator Interface

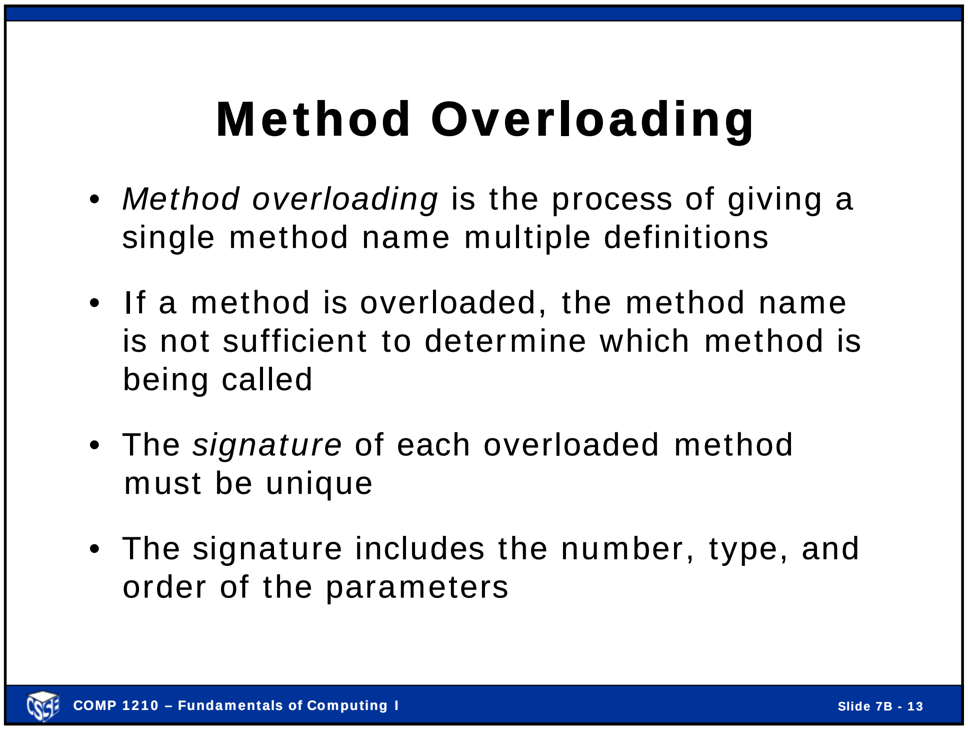
- An iterator is an object that provides a means of processing a collection of objects one at a time
- An iterator is created formally by implementing the `Iterator` interface, which contains three methods
  - The `hasNext` method returns a boolean result – true if there are items left to process
  - The `next` method returns the next object in the iteration
  - The `remove` method 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 one or more spaces)
- You'll use this in COMP 2210 when you start building data structures like lists
- The for-each version of the `for` loop can be used to process the items in classes that implement the Iterable Interface (e.g., ArrayList).

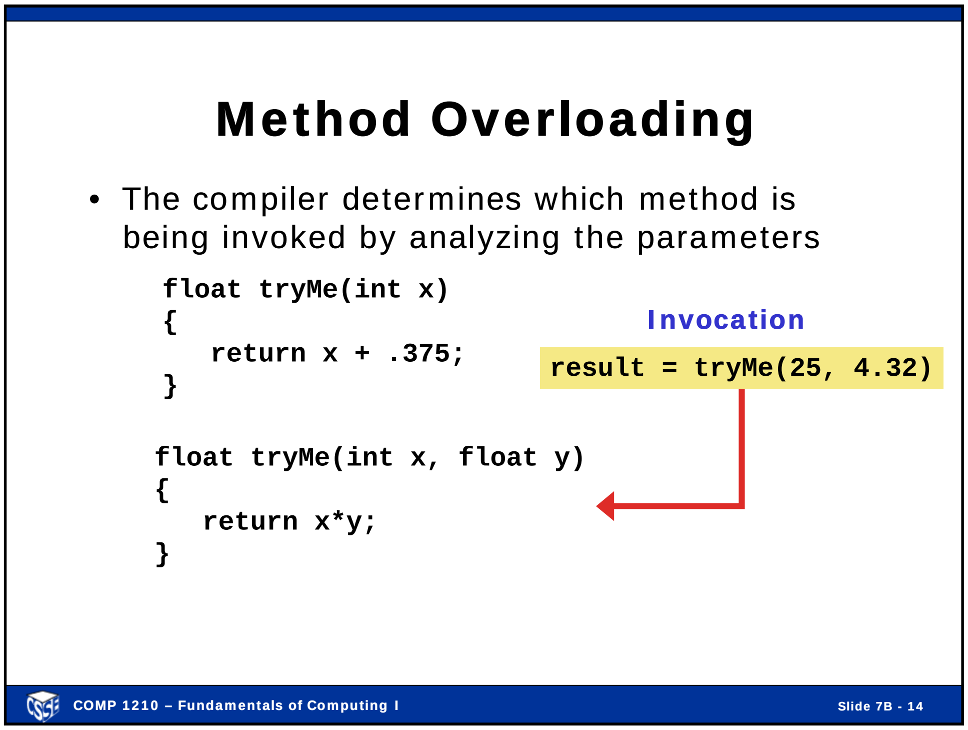
Interfaces

- You could implement `compareTo` without implementing the interface `Comparable`, but you would limit the functionality
  - For example, Arrays.sort relies on compareTo.
  - If you try to use Arrays.sort on an array of Rectangles, it will generate a run-time error **unless Comparable is implemented** (even if you have defined compareTo)
  - Try commenting out implements Comparable<Rectangle> in Rectangle.java and running RectangleSorter.java



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 number, type, and order of the parameters



Method Overloading

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

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

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

Invocation

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

Method Overloading

- The `println` method 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 Methods

- The return type of the method is not part of the signature
- That is, overloaded methods cannot differ only by their return type
- 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)

Method Design

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

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

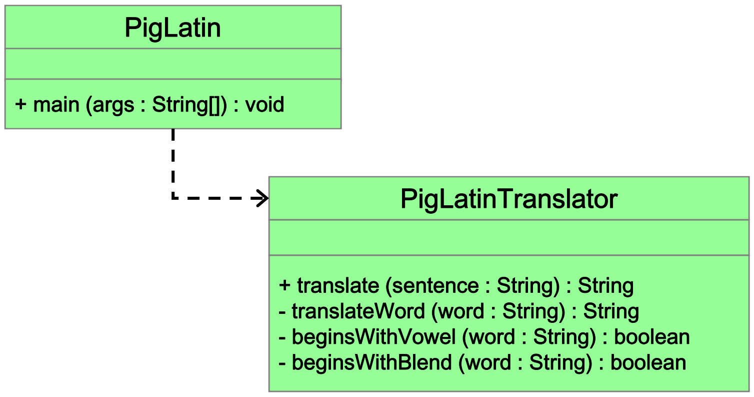
Method Decomposition

- The primary objective (translating a sentence) is too complicated for one method to accomplish
- Therefore we look for natural ways to decompose the solution into pieces
- 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 UML class diagram, the visibility of a variable or method can be shown using special characters
- Public members are preceded by a plus sign
- Private members are preceded by a minus sign

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 a parameter may or may not have a permanent effect (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 points to

Testing

- Testing can mean many different things
- It certainly includes running a completed program with various inputs
- It also includes any evaluation performed by human or computer to assess quality
- Some evaluations should occur before coding even begins
- The earlier we find an problem, the easier and cheaper it is to fix

Testing

- The goal of testing is to find defects (via failures)
- As we find and fix defects, we raise our confidence that a program will perform as intended
- We can never really be sure that all defects have been eliminated
- So when do we stop testing?
  - Conceptual answer: Never
  - Snide answer: When we run out of time
  - Better answer: When we are willing to risk that an undiscovered defects still exists

Test Cases

- A *test case* is a set of input and/or user actions, coupled with the expected results
- Often test cases are organized formally into *test suites* which are stored and reused as needed
- For medium and large systems, testing must be a carefully managed process
- Many organizations have a separate Quality Assurance (QA) department to lead testing efforts

Defect and Regression Testing

- *Defect testing* is the execution of test cases to uncover defects/errors
- The act of fixing a defect/error may introduce new defects
- After fixing a set of defects/errors we should perform *regression testing* – running previous test suites to ensure new errors haven't been introduced
- It is not possible to create test cases for all possible input and user actions
- Therefore we should design tests to maximize their ability to find problems

Black-Box Testing

- In *black-box testing*, test cases are developed without considering the internal logic
- They are based on the input and expected output
- Input can be organized into *equivalence categories*
- Two input values in the same equivalence category would produce similar results
- Therefore a good test suite will cover all equivalence categories and focus on the boundaries between categories

White-Box Testing

- *White-box testing* focuses on the internal structure of the code
- The goal is to ensure that every <u>independent</u> path through the code is tested
- Paths through the code are determined by conditional or looping statements in a program
- A good testing effort will include both black-box and white-box tests