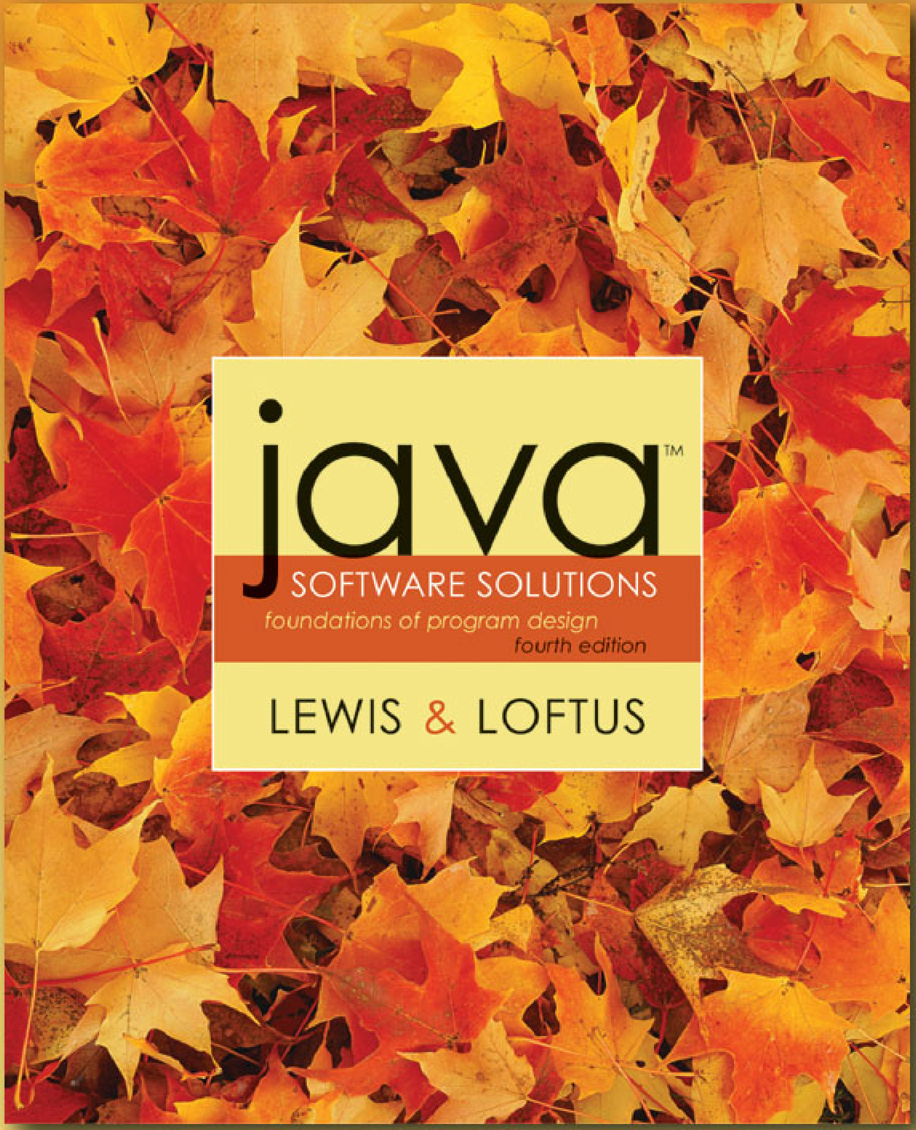
Lecture 6

Object-Oriented Design



Object-Oriented Design

- Now we can extend our discussion of the design of classes and objects

- Lecture 6 focuses on:
  - software development activities
  - determining the classes and objects that are needed for a program
  - the relationships that can exist among classes
  - the static modifier
  - writing interfaces
  - the design of enumerated type classes
  - method design and method overloading

Outline

- Software Development Activities
- Identifying Classes and Objects
- Static Variables and Methods
- Class Relationships
- Interfaces
- Enumerated Types Revisited
- Method Design
- Testing

Program Development

- The creation of software involves four basic activities:
  - establishing the requirements
  - creating a design
  - implementing the code
  - testing the implementation
- These activities are not strictly linear – they overlap and interact

Requirements

- ***Software requirements*** **specify the tasks that a program must accomplish**
  - **<u>what</u> to do, not how to do it**
- **Often an initial set of requirements is provided, but they should be critiqued and expanded**
- **It is difficult to establish detailed, unambiguous, and complete requirements**
- **Careful attention to the requirements can save significant time and expense in the overall project**

Design

- **A *software design* specifies <u>how</u> a program will accomplish its requirements**
- **That is, a software design determines:**
  - **how the solution can be broken down into manageable pieces**
  - **what each piece will do**
- **An object-oriented design determines which classes and objects are needed, and specifies how they will interact**
- **Low level design details include how individual methods will accomplish their tasks**

Implementation

- *Implementation* is the process of translating a design into source code
- Novice programmers often think that writing code is the heart of software development, but actually it should be the least creative step
- Almost all important decisions are made during requirements and design stages
- Implementation should focus on coding details, including style guidelines and documentation

Testing

- *Testing* attempts to ensure that the program will solve the intended problem under all the constraints specified in the requirements
- A program should be thoroughly tested with the goal of finding errors
- *Debugging* is the process of determining the cause of a problem and fixing it
- We revisit the details of the testing process later in this chapter

Outline

Software Development Activities

Identifying Classes and Objects

Static Variables and Methods

Class Relationships

Interfaces

Enumerated Types Revisited

Method Design

Testing

Identifying Classes and Objects

- The core activity of object-oriented design is determining the classes and objects that will make up the solution
- The classes may be part of a class library, reused from a previous project, or newly written
- One way to identify potential classes is to identify the objects discussed in the requirements
- Objects are generally nouns, and the services that an object provides are generally verbs

Identifying Classes and Objects

- **A partial requirements document:**

> The user must be allowed to specify each product by its primary characteristics, including its name and product number. If the bar code does not match the product, then an error should be generated to the message window and entered into the error log. The summary report of all transactions must be structured as specified in section 7.A.

Of course, not all nouns will correspond to a class or object in the final solution

Identifying Classes and Objects

- Remember that a class represents a group (classification) of objects with the same behaviors
- Generally, classes that represent objects should be given names that are singular nouns
- Examples: `Coin`, `Student`, `Message`
- A class represents the concept of one such object
- We are free to instantiate as many of each object as needed

Identifying Classes and Objects

- Sometimes it is challenging to decide whether something should be represented as a class
- For example, should an employee's address be represented as a set of instance variables or as an `Address` object
- The more you examine the problem and its details the more clear these issues become
- When a class becomes too complex, it often should be decomposed into multiple smaller classes to distribute the responsibilities

Identifying Classes and Objects

- We want to define classes with the proper amount of detail
- For example, it may be unnecessary to create separate classes for each type of appliance in a house
- It may be sufficient to define a more general `Appliance` class with appropriate instance data
- It all depends on the details of the problem being solved

Identifying Classes and Objects

- **Part of identifying the classes we need is the process of *assigning responsibilities* to each class**
- **Every activity that a program must accomplish must be represented by one or more methods in one or more classes**
- **We generally use verbs for the names of methods**
- **In early stages it is not necessary to determine every method of every class – begin with primary responsibilities and evolve the design**

Outline

Software Development Activities

Identifying Classes and Objects

Static Variables and Methods

Class Relationships

Interfaces

Enumerated Types Revisited

Method Design

Testing

Static Class Members

- Recall that a static method is one that can be invoked through its class name
- For example, the methods of the `Math` class are static:

```
result = Math.sqrt(25)
```

- Variables can be static as well
- Determining if a method or variable should be static is an important design decision

The static Modifier

- We declare static methods and variables using the `static` modifier
- It associates the method or variable with the class rather than with an object of that class
- Static methods are sometimes called *class methods* and static variables are sometimes called *class variables*
- Let's carefully consider the implications of each

Static Variables

- Normally, each object has its own data space, but if a variable is declared as static, only one copy of the variable exists

```
private static float price;
```

- Memory space for a static variable is created when the class is first referenced
- All objects instantiated from the class share its static variables
- Changing the value of a static variable in one object changes it for all others

Static Methods

```
class Helper
{
   public static int cube (int num)
   {
      return num * num * num;
   }
}
```

Because it is declared as static, the method can be invoked as

```
value = Helper.cube(5);
```

Static Class Members

- The order of the modifiers can be interchanged, but by convention visibility modifiers come first
- Recall that the `main` method is static – it is invoked by the Java interpreter without creating an object
- Static methods cannot reference instance variables because instance variables don't exist until an object exists
- However, a static method can reference static variables or local variables

Static Class Members

- Static methods and static variables often work together

- The following example keeps track of how many `Slogan` objects have been created using a static variable, and makes that information available using a static method

- See SloganCounter.java (page 294)
- See Slogan.java (page 295)

Outline

Software Development Activities

Identifying Classes and Objects

Static Variables and Methods

Class Relationships

Interfaces

Enumerated Types Revisited

Method Design

Testing

Class Relationships

- Classes in a software system can have various types of relationships to each other
- Three of the most common relationships:
  - Dependency: A *uses* B
  - Aggregation: A *has-a* B
  - Inheritance: A *is-a* B
- Let's discuss dependency and aggregation further
- Inheritance is discussed in detail in Chapter 8

Dependency

- A *dependency* exists when one class relies on another in some way, usually by invoking the methods of the other
- We've seen dependencies in many previous examples
- We don't want numerous or complex dependencies among classes
- Nor do we want complex classes that don't depend on others
- A good design strikes the right balance

Dependency

- **Some dependencies occur between objects of the same class**
- **A method of the class may accept an object of the same class as a parameter**
- **For example, the `concat` method of the `String` class takes as a parameter another `String` object**

```
str3 = str1.concat(str2);
```

- **This drives home the idea that the service is being requested from a particular object**

Dependency

- The following example defines a class called `Rational` to represent a rational number

- A rational number is a value that can be represented as the ratio of two integers

- Some methods of the `Rational` class accept another `Rational` object as a parameter

- See RationalTester.java (page 297)
- See RationalNumber.java (page 299)

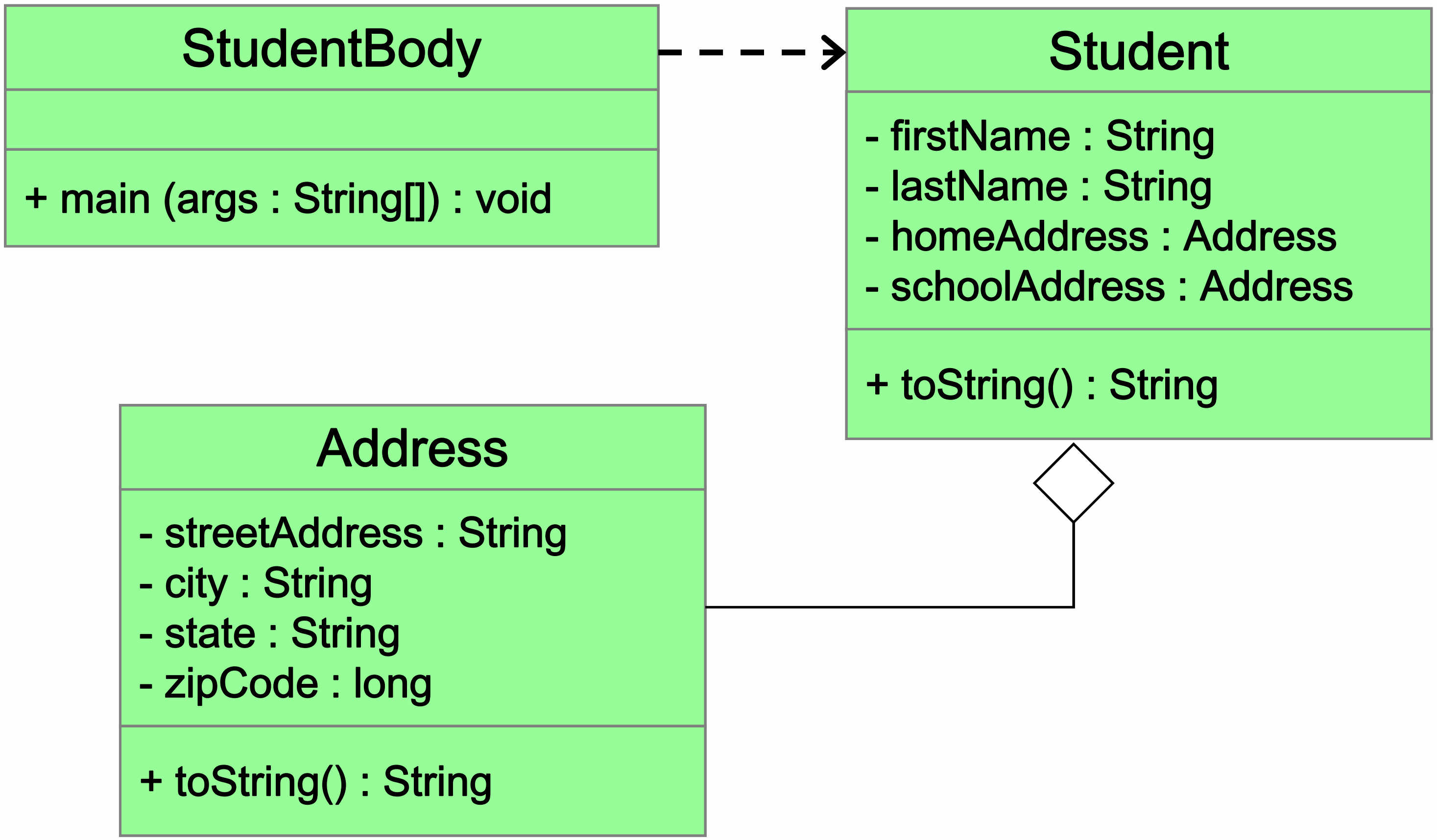
Aggregation

- **An *aggregate* is an object that is made up of other objects**
- **Therefore aggregation is a *has-a* relationship**
  - **A car *has a* chassis**
- **In software, an aggregate object contains references to other objects as instance data**
- **The aggregate object is defined in part by the objects that make it up**
- **This is a special kind of dependency – the aggregate usually relies on the objects that compose it**

Aggregation

- In the following example, a `Student` object is composed, in part, of `Address` objects
- A student has an address (in fact each student has two addresses)
- See StudentBody.java (page 304)
- See Student.java (page 306)
- See Address.java (page 307)
- An aggregation association is shown in a UML class diagram using an open diamond at the aggregate end

Aggregation in UML



The this Reference

- The `this` reference allows an object to refer to itself
- That is, the `this` reference, used inside a method, refers to the object through which the method is being executed
- Suppose the `this` reference is used in a method called `tryMe`, which is invoked as follows:

```
obj1.tryMe();
obj2.tryMe();
```

- In the first invocation, the `this` reference refers to `obj1`; in the second it refers to `obj2`

The this reference

- The `this` reference can be used to distinguish the instance variables of a class from corresponding method parameters with the same names
- The constructor of the `Account` class (from Chapter 4) could have been written as follows:

```
public Account (Sring name, long acctNumber,
                double balance)
{
   this.name = name;
   this.acctNumber = acctNumber;
   this.balance = balance;
}
```

Outline

Software Development Activities

Identifying Classes and Objects

Static Variables and Methods

Class Relationships

Interfaces

Enumerated Types Revisited

Method Design

Testing

Interfaces

- **A Java *interface* is a collection of abstract methods and constants**
- **An *abstract method* is a method header without a method body**
- **An abstract method can be declared using the modifier `abstract`, but because all methods in an interface are abstract, usually it is left off**
- **An interface is used to establish a set of methods that a class will implement**

Interfaces

interface is a reserved word

None of the methods in an interface are given a definition (body)

```
public interface Doable
{
   public void doThis();
   public int doThat();
   public void doThis2 (float value, char ch);
   public boolean doTheOther (int num);
}
```

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**
  - **providing implementations for each abstract method in the interface**
- **If a class asserts that it implements an interface, it must define all methods in the interface**

Interfaces

```
public class CanDo implements Doable
{
   public void doThis ()
   {
      // whatever
   }

   public int doThat ()
   {
      // whatever
   }

   // etc.
}
```

implements is a reserved word

Each method listed in **Doable** is given a definition

Interfaces

- A class that implements an interface can implement other methods as well

- See Complexity.java (page 310)
- See Question.java (page 311)
- See MiniQuiz.java (page 313)

- 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

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
}
```

Interfaces

- **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**
- **We discussed the `compareTo` method of the `String` class in Chapter 5**
- **The `String` class implements `Comparable`, giving us the ability to put strings in lexicographic order**

The Comparable Interface

- Any class can implement `Comparable` to provide a mechanism for comparing objects of that type

```
if (obj1.compareTo(obj2) < 0)
   System.out.println ("obj1 is less than obj2");
```

- The value returned from `compareTo` should be negative is `obj1` is less that `obj2`, 0 if they are equal, and positive if `obj1` is greater than `obj2`
- When a programmer designs a class that implements the `Comparable` interface, it should follow this intent

The Comparable Interface

- It's up to the programmer to determine 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 implementation of the method can be as straightforward or as complex as needed for the situation

The Iterator Interface

- As we discussed in Chapter 5, 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

- By implementing the `Iterator` interface, a class formally establishes that objects of that type are iterators
- The programmer must decide how best to implement the iterator functions
- Once established, the for-each version of the `for` loop can be used to process the items in the iterator

Interfaces

- You could write a class that implements certain methods (such as `compareTo`) without formally implementing the interface (`Comparable`)
- However, formally establishing the relationship between a class and an interface allows Java to deal with an object in certain ways
- Interfaces are a key aspect of object-oriented design in Java
- We discuss this idea further in Chapter 9

Outline

Software Development Activities

Identifying Classes and Objects

Static Variables and Methods

Class Relationships

Interfaces

Enumerated Types Revisited

Method Design

Testing

Enumerated Types

- In Chapter 3 we introduced enumerated types, which define a new data type and list all possible values of that type

```
enum Season {winter, spring, summer, fall}
```

- Once established, the new type can be used to declare variables

```
Season time;
```

- The only values this variable can be assigned are the ones established in the enum definition

Enumerated Types

- An enumerated type definition is a special kind of class
- The values of the enumerated type are objects of that type
- For example, `fall` is an object of type Season
- That's why the following assignment is valid

```
time = Season.fall;
```

Enumerated Types

- An enumerated type definition can be more interesting than a simple list of values
- Because they are like classes, we can add additional instance data and methods
- We can define an `enum` constructor as well
- Each value listed for the enumerated type calls the constructor
- See Season.java (page 318)
- See SeasonTester.java (page 319)

Enumerated Types

- Every enumerated type contains a static method called `values` that returns a list of all possible values for that type
- The list returned from `values` is an iterator, so a `for` loop can be used to process them easily
- An enumerated type cannot be instantiated outside of its own definition
- A carefully designed enumerated type provides a versatile and type-safe mechanism for managing data

Outline

Software Development Activities

Identifying Classes and Objects

Static Variables and Methods

Class Relationships

Interfaces

Enumerated Types Revisited

Method Design

Testing

Method Design

- **As we've discussed, high-level design issues include:**
  - **identifying primary classes and objects**
  - **assigning primary responsibilities**
- **After establishing high-level design issues, its important to address low-level issues such as the design of key methods**
- **For some methods, careful planning is needed to make sure they contribute to an efficient and elegant system design**

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

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

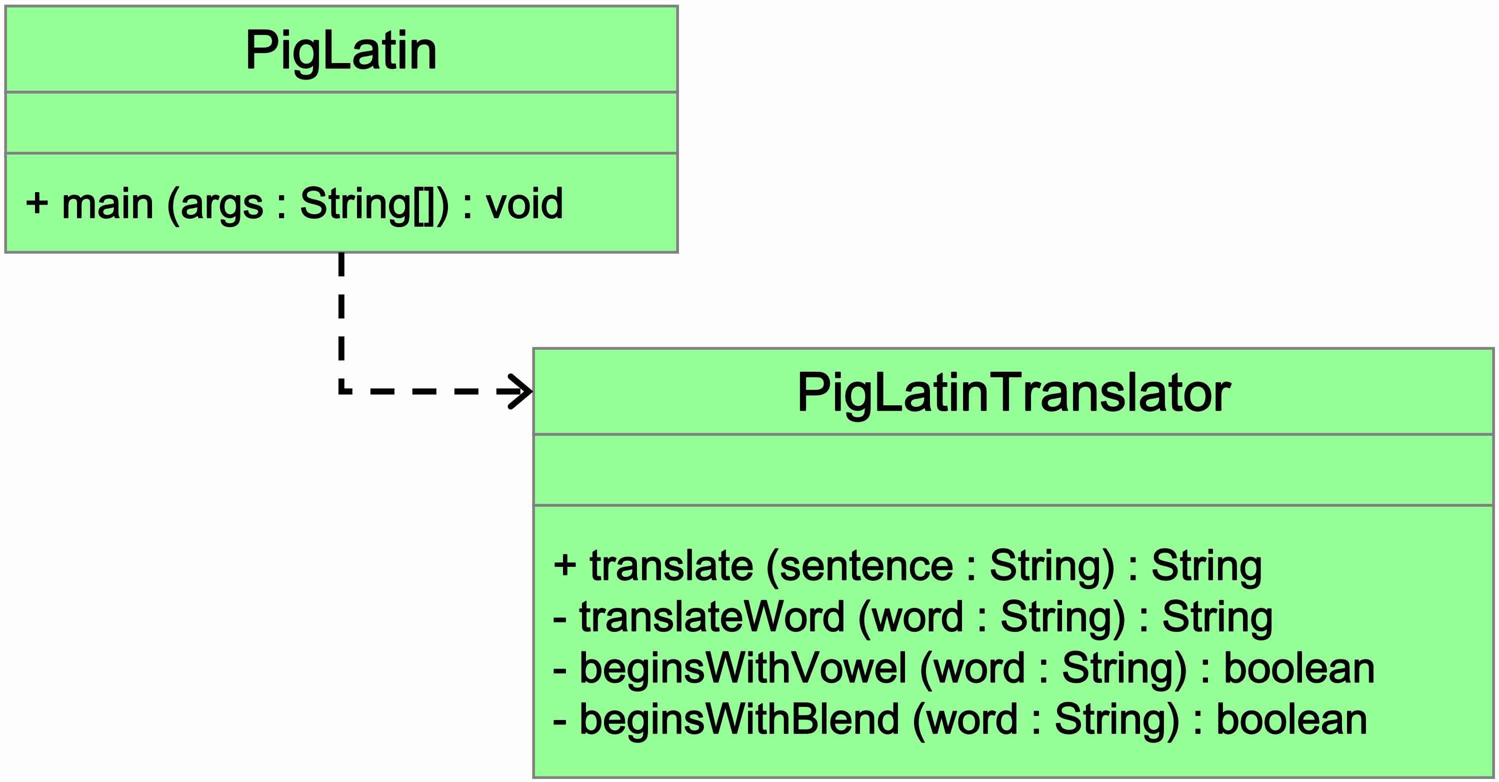
Method Decomposition

- 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 (page 320)
- See PigLatinTranslator.java (page 323)
- 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



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 (page 327)
- See ParameterModifier.java (page 329)
- See Num.java (page 330)

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

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;
}
```

Invocation

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

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

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 <u>not</u> part of the signature**
- **That is, overloaded methods cannot differ only by their return type**
- **Constructors can be overloaded**
- **Overloaded constructors provide multiple ways to initialize a new object**

Outline

Software Development Activities

Identifying Classes and Objects

Static Variables and Methods

Class Relationships

Interfaces

Enumerated Types Revisited

Method Design

Testing

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 errors
- As we find and fix errors, we raise our confidence that a program will perform as intended
- We can never really be sure that all errors 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 error still exists

Reviews

- A *review* is a meeting in which several people examine a design document or section of code
- It is a common and effective form of human-based testing
- Presenting a design or code to others:
  - makes us think more carefully about it
  - provides an outside perspective
- Reviews are sometimes called *inspections* or *walkthroughs*

Test Cases

- **A *test case* is a set of input and 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 errors
- The act of fixing an error may introduce new errors
- After fixing a set of 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 path through the code is tested**
- **Paths through the code are governed by any conditional or looping statements in a program**
- **A good testing effort will include both black-box and white-box tests**

Summary

- **Lecture 6 has focused on:**
  - **software development activities**
  - **determining the classes and objects that are needed for a program**
  - **the relationships that can exist among classes**
  - **the static modifier**
  - **writing interfaces**
  - **the design of enumerated type classes**
  - **method design and method overloading**