Object Oriented Design and UML

- Class Relationships
  - Dependency
  - Aggregation
  - Interfaces
  - Inheritance
- Interfaces
- Reading for this Lecture: L&L 6.4 – 6.5

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 (as in B is an integral part of A)
  - Interface: A is B (adjective) or A is-a B (noun)
  - Inheritance: A *is-a* B
- We cover the first three now
- We cover inheritance later

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 previous examples and in Projects 1 and 2
- 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 `equals` method of the `String` class takes as a parameter another `String` object

```
boolean b = str1.equals(str2);
```

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

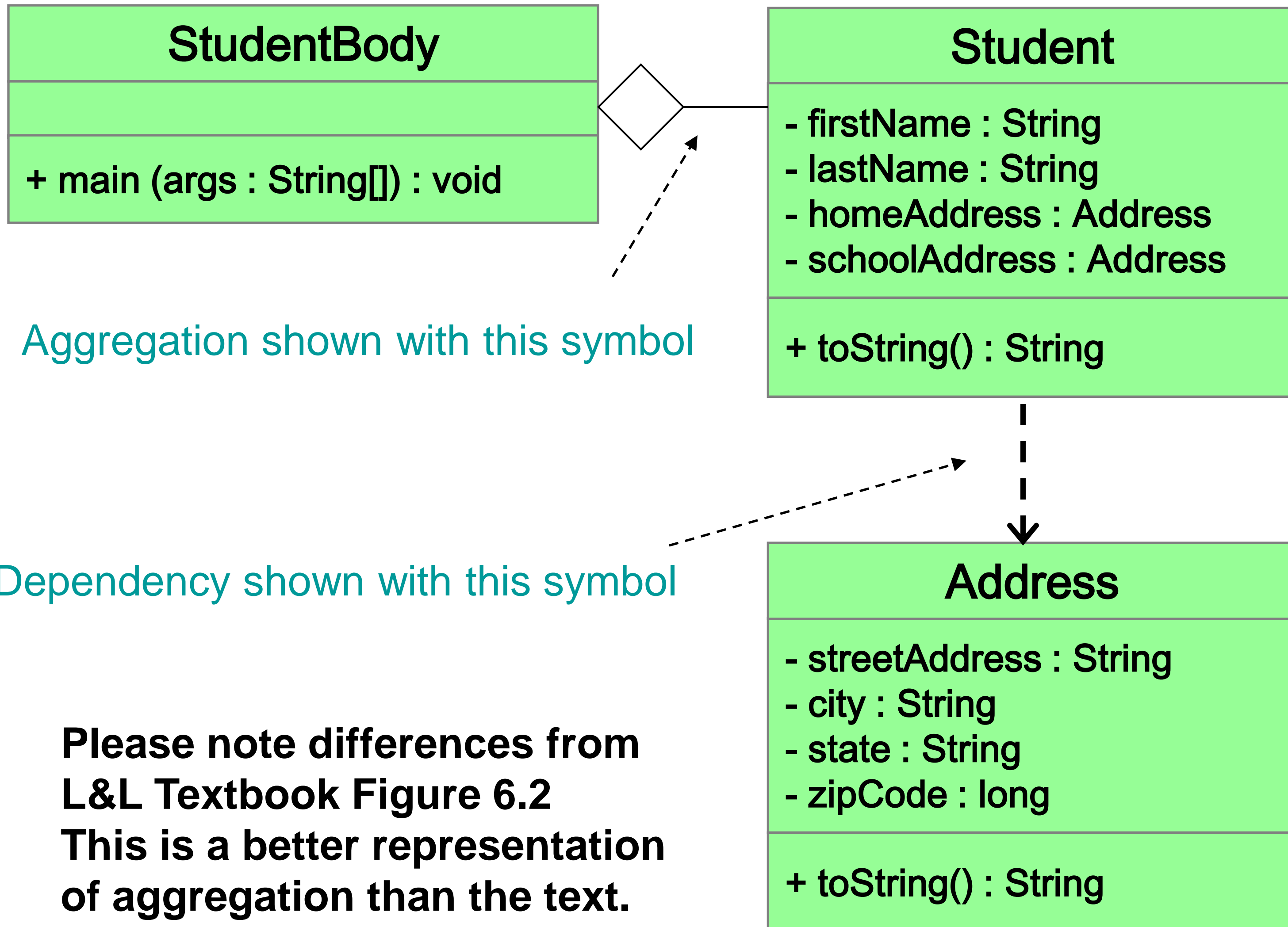
Aggregation

- An *aggregate* is an object that is made up of other objects
- Therefore aggregation is a *has-a* relationship
  - A Car *has a* Chassis and *has an* Engine
  - A StudentBody has (a) Student object(s)
- In code, an aggregate object contains references to its component objects as instance data
- The aggregate object itself is defined in part by the objects that make it up
- This is a special kind of dependency – the aggregate usually relies for its existence on the component objects

Aggregation

- In the following example, a `StudentBody` object is composed of integral `Student` objects which then depend on `Address` objects
- A `StudentBody` has one or more `Student`(s)
- See StudentBody.java (page 312)
- See Student.java (page 313)
- See Address.java (page 314)
- An aggregation association is shown in a UML class diagram using an open diamond at the aggregate end (Note difference from text diagram)

Dependency/Aggregation in UML



Please note differences from L&L Textbook Figure 6.2 This is a better representation of aggregation than the text.

Aggregation

- There are two ways to include the component objects in an object that is an aggregation
  - For one component (or a small constant number of components), use parameters in the constructor

```
public Car(Chassis c, Engine e)
{   ...   }
```

  - For a large or indefinite number of components, define an add method to add them one at a time

```
public void add(Student aStudent)
{   ...   }
```

Interfaces

- A Java *interface* is a collection of constants and *abstract methods* with a name that looks like a class name, i.e. the first letter is capitalized
- An interface is used to identify a set of methods that a class will implement
- An *abstract method* is a method header with a ; and without a method body, i.e. No `{ . . . }`
- An abstract method can be declared using the modifier `abstract`, but because all methods in an interface are abstract, it is usually left off
- Methods in an interface have public visibility by default

Interfaces

interface is a reserved word

None of the methods in an interface are given a definition {body}

```
public interface Doable
{
   // Doable constants
   public static final boolean DONE = true;
   public static final boolean NOT_DONE = false;

   // Doable required methods (signatures only)
   public void doThis();
   public int doThat();
}
```

A semicolon immediately follows each method header

Interfaces

- An interface name can be either an adjective (like …able) or a noun (like a class name)
- An interface cannot be instantiated by itself
- A class implements an interface by:
  - using the Java reserved word implements
  - providing an implementation for each abstract method that is defined in the interface
- Classes that implement an interface can also implement their own methods and they usually do

Interfaces

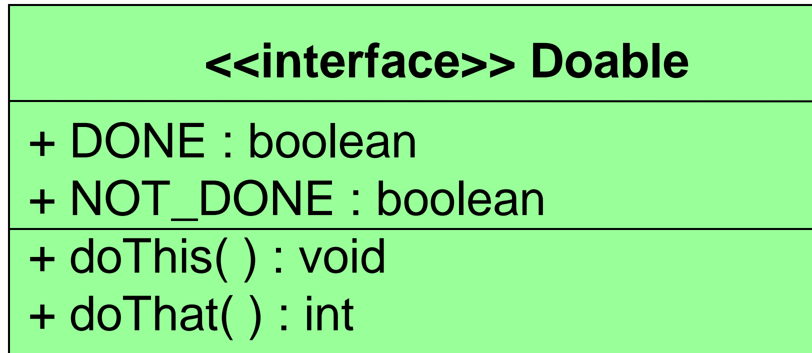
```
public class CanDo implements Doable
{

    public void doThis ()
    {
        // whatever
    }

    public int doThat ()
    {
        // whatever
    }

    // etc.
}
```

Doable is an adjective

implements is a reserved word

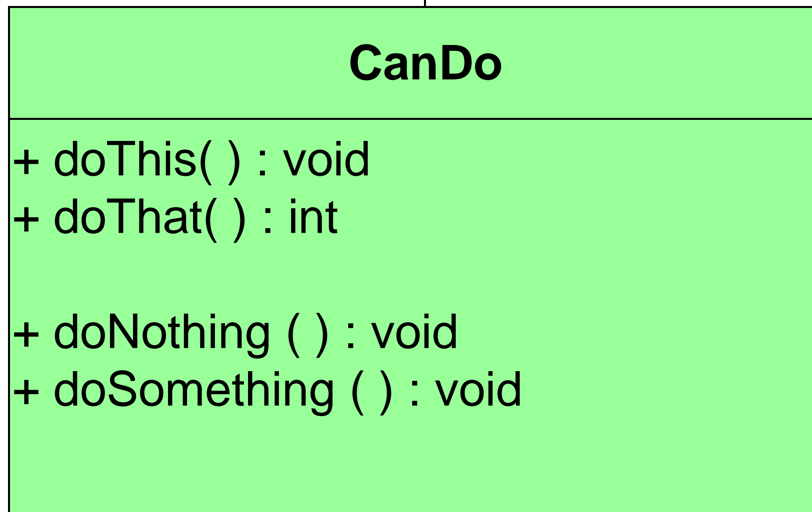
Each method listed in Doable must be given a definition

Interfaces In UML



Interface box looks like a class box with stereotype <<interface>>

A "Generalization" arrow is used for "implements" (and also for "extends" later)



Each method listed in **Doable** becomes a method of CanDo

CanDo can have other methods of its own

Interfaces

- 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 defined constants

Interfaces

- A class can implement multiple interfaces
- All interface names 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 all interfaces
}
```

Interfaces

- The Java standard class library contains many interface definitions that allow other classes to treat your new class as if it were that interface
- Note: Comparable is an adjective in this case
- The `Comparable` interface contains one abstract method called `compareTo`, which can compare an object with another object of the same type
- We discussed the `compareTo` method of the `String` class previously
- 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 by providing a `compareTo` method

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

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

The Comparable Interface

- It's up to you as 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), by salary, by employee number, or any other useful way
- The implementation of the method can be as straightforward or as complex as needed for the situation

Interfaces as "Reference Types"

- You could write a class that implements certain methods (such as `compareTo`) without formally implementing the interface (`Comparable`)
- But, formally establishing the relationship between your class and an predefined interface allows Java to deal with an object of your class as if it were an object of a class corresponding to the interface name

Interfaces as “Reference Types”

- You can cast using the interface name in ( )

```
CanDo iCanDo = new CanDo();
...
Doable iAmDoable = iCanDo;  // widening
```

- You can pass an object of CanDo class to a method as an object of Doable “class”.

```
doIt(iCanDo);
...
public void doIt(Doable isItReallyDoable)
{
     ...     // Yes, iCanDo is Doable!
}
```

Interfaces as “Reference Types”

- When you are using an object “cast as” one of the interfaces that it implements, you are treating this object as if it were an object of a class defined by the interface
- You can only access the subset of the object’s methods that are defined in the interface
- CanDo methods, such as `doNothing(),` are not accessible when a `CanDo` object is cast as a `Doable` object because they are not defined in the `Doable` interface

Interfaces as “Reference Types”

```
CanDo iCanDo = new CanDo();
iCanDo.doThis();                  // a Doable method
iCanDo.doNothing();               // a CanDo  method

// a widening conversion - no cast
Doable iAmDoable = new CanDo();
// all Doable methods are available
iAmDoable.doThis();

// CanDo method not accessible via Doable interface
// iAmDoable.doNothing(); // would be compiler error

// but it is really there - need a cast to call it
((CanDo)iAmDoable).doNothing();
```