CS 5004

LECTURE 4

COMPOSITION VS. INHERITANCE

KEITH BAGLEY

SPRING 2022

Northeastern University
Khoury College of Computer Sciences

440 Huntington Avenue ▪ 202 West Village H ▪ Boston, MA 02115 ▪ **T** 617.373.2462 ▪ **khoury.northeastern.edu**

AGENDA

- Review: Interfaces & Abstract Classes
- Triangle Breakout
- Composition vs. Inheritance
- Envelope & Letter
- Composition Design Pattern: Decorator
- Shapes Refactor for Composition: Point
- Q & A

REVIEW: INTERFACES & ABSTRACT CLASSES

- A Java **interface** is a "protocol" (in other languages) it describes the external behavior expected by anything that implements it. Effectively forms the "contract" for external behavior.
  - Basis for specifying our User Defined Types
- An **Abstract Class** is a class designed to be inherited from, but NOT instantiated itself. Abstract classes can be conceptual, but most often placeholder for common elements/operations the rest of the inheritance hierarchy can reuse.
- We implement interfaces (no inheritance chain required)
- We extend abstract (and concrete) classes (inheritance chain accepted)
- We can do both: implement interfaces and extend from a single direct superclass

ABSTRACT CLASSES & ABSTRACT METHODS

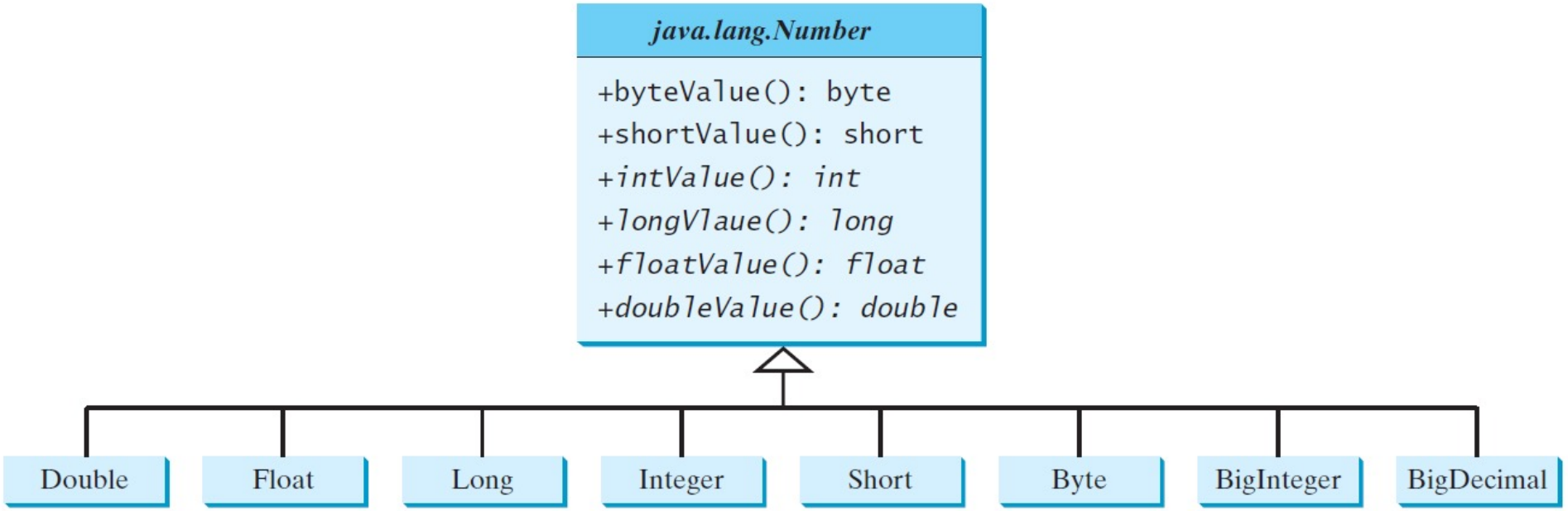
- Abstract classes cannot be instantiated
- Abstract classes can specify actual methods for reuse in subclasses. Or they may specify ”placeholder” methods that do not have any implementation
  - These “placeholder methods” are called **abstract methods**
- Abstract methods have no implementation. They provide the behavioral frame for which all concrete subclasses must adhere to
  - Any subclass that is designed to be concrete must override all abstract methods. In other words, an abstract method cannot be contained in a concrete (non-abstract) class

ABSTRACT CAN BE ANYWHERE IN HIERARCHY

- Abstract classes can be defined anywhere in the hierarchy
  - In Java, the abstract keyword can be used to create abstract classes whose superclass is a concrete class
  - Note 👉🏽 This is more of a programming-language feature than a “general design” approach!
    - Typically for design, abstract classes are higher in the hierarchy, and do not have concrete classes interspersed in-between

ABSTRACT CAN BE ANYWHERE IN HIERARCHY

- Java example of abstract class: Number & numeric wrappers



intValue(), etc. are abstract methods

Image from Liang, © 2019

INTERFACES – JAVA PARTICULARS

- We use Java interfaces as **protocols** for type specifications
  - Remember: interfaces are language specific implementations of the general protocol concept
- Java interfaces diverge in a couple of ways. Here are some Java-isms:
  - Java allows for static fields to be defined in interfaces
  - Java allows for default implementations to be specified for interfaces
    - This is of limited use, since interfaces cannot have instance variables

INTERFACES – JAVA PARTICULARS

- All data fields in interfaces are public final static and all methods are public abstract in an interface. For this reason, these modifiers can be omitted, as shown below:

```
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T1 {
  int K = 1;

  void p();
}
```

Image and verbiage from Liang, 2020

INTERFACES – JAVA PARTICULARS

- As of Java 8, default implementations can be provided for interfaces. This is of limited value since no instance variables are allowed so detailed implementation information remains unknown until a class implements the interface.
- Default methods are specified with the default keyword
- Java 9 also allows for private methods to be specified in the interface

```
public interface JavaInterface {
  public default void doSomething(){
      System.out.println("Did it!");
  }
}
```

INTERFACES – JAVA PARTICULARS

- As the previous slides show, Java interfaces diverge a bit from the classical (and cross-language) concept of protocols.
- For our purposes (and this class) we'll stick with the multi-language protocol concept and use Java interfaces as protocols

REVIEW: SUBTYPING VS SUBCLASSING

- Remember: protocols/contracts (Java interfaces) are TYPE specifications
  - The protocol tells us what the type does, NOT how the type does it, or what kind of internal representation it uses
  - In CS5001, we learned about Abstract Data Types (like Queues and Stacks). Recall we chose a Python List for one implementation, but that was NOT the only alternative
- Remember: classes are ONE style of implementation. In Java it is THE style of implementation. In other languages (like C) there is no such thing as a class, so there is no such thing as “subclassing”.
  - Types and subtyping is a concept that is consistent across all languages, and even hardware

REVIEW: SUBTYPING VS SUBCLASSING

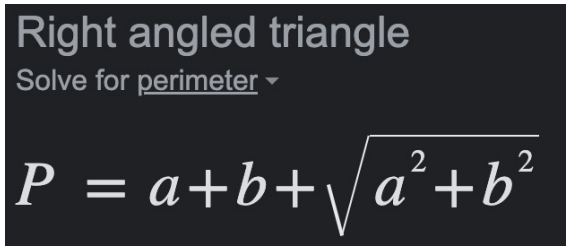
- Consider the Type: Lock
  - Simple protocol:

    `lock()`
    `unlock()`

Subtypes: Hardware cylinder lock, Keycard lock (hardware/software), Cyber security lock
Implementation is different. No “subclassing”. Classes are a OO software mechanism

YOU TRY!

- Given the Shape Interface (and abstract shape class), create a concrete (right) Triangle class that implements the Shape protocol. Feel free to use an abstract class for code reuse if you wish.
  - ½ base * height
  - Perimeter:



DESIGN: INHERITANCE VS. COMPOSITION

- Structural concerns: is-a vs. has-a

REVIEW: INHERITANCE

- Implementation mechanism via Java's extends keyword
  - Design choice that requires you to accept certain structural aspects of what you're extending from
  - Usually an is-a or is-a-kind-of relationship (tracking with supertype-subtype) BUT NOT MANDATORY TO DO SO!
  - Sometimes causes issues – especially if you do not control the code
  - Code reuse

INHERITANCE IS CODE REUSE

Keith Bagley 29 days ago Here's another example to distinguish the two. Assume we have a List class

```
public class List {
  data = new int [100] ;

  int size() {
    // return the number of elements
  }

  add(int anInteger) {
    // etc.
  }
}
```

Subclassing is not necessarily sub-typing!

...and then for convenience/speed/laziness I create a Set class that makes use of our List (NB: I'm not ad example of subclassing being legal even when we're not subtyping):

```
public class Set extends List {
  add(int anInteger) {
    if (/* data is not already in the collection, add it */)) {
      super.add(anInteger);
    }
  }

// etc.
}
```

COMPOSITION

- Has-a relationship
  - Does not require or tie you to an inheritance hierarchy
  - More flexible than inheritance in many cases
    - Can have more than one delegate to provide services
    - Change change the composition at runtime
  - Forms the basis of many design patterns
  - Code reuse

COMPOSITION & DELEGATION

- Delegation is a mechanism which allows one class to provide an interface to clients for a number of services, while allowing the actual performance of those services to be accomplished by other objects
- Delegation de-couples the service interface from the service implementation, on an object/class level

DELEGATION

- Using data abstraction and information hiding, the physical structure of an object does not have to match the abstract structure

- Physical behavior must match the abstract model However, any implementation that matches the model may be used. Underlying implementations may be changed as long as they conform to the abstract model

ENVELOPE/LETTER

- Structurally, delegation entails a “mini pattern” often called “envelope/letter”.
- The envelope/letter pattern composes two (or more) classes to provide a service
- The outer class performing memory management and the inner class performing the expected service

ENVELOPE/LETTER EXAMPLE

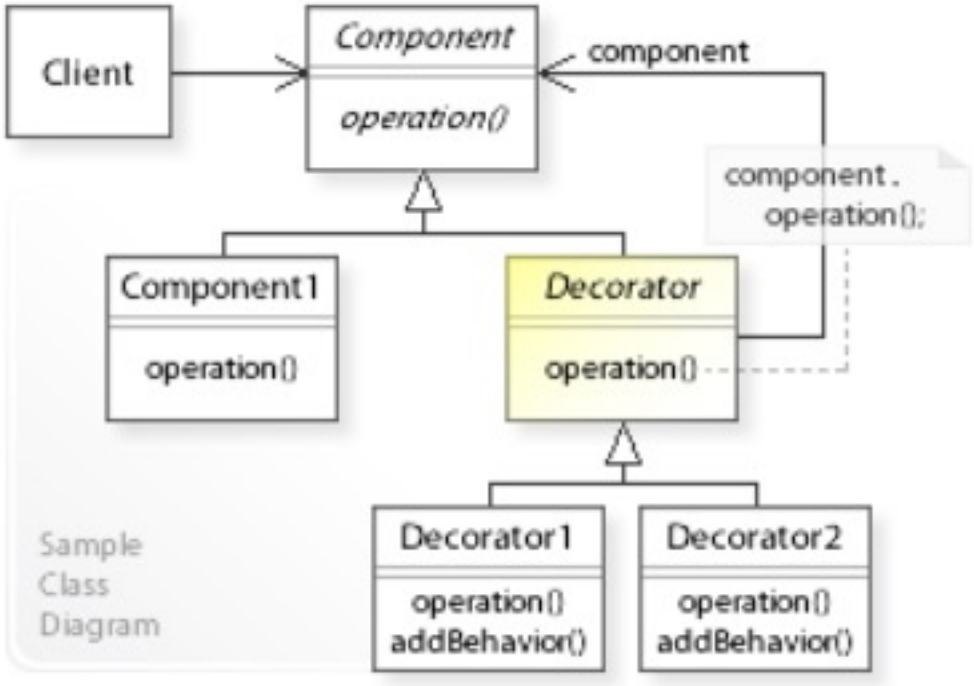
```
public interface Worker {/* etc. */ }

class SummerCoOpStudent implements Worker{
  public void doRealWork();
  // etc...
}

class Politician implements Worker {
  public void pretendToWork(){ /* no-op */ }
  public void doWork(){ delegate.doRealWork();}
  // etc..
  private SummerCoOpStudent delegate;
}
```

ENVELOPE/LETTER

- Java uses the envelope/letter for its numeric wrapper classes:
  - Double, Integer, etc.
  - Primitive types do not work with collection classes (only classes/objects do) so the envelope/letter paradigm wraps the primitive service in an object encasing
  - We'll discuss this more when we cover Generics and Collections

FOR EXAMPLE: DECORATOR PATTERN

- The Decorator pattern allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class
- It provides a flexible way for adding functionality at runtime, without subclassing or being tied to some compile-time decisions
- Decorators use composition



DECORATOR EXAMPLE

- Let's examine a decorator implementation so we can see composition and delegation in action

COMPOSITION & DELEGATION: SHAPES

- Let's refactor our Shapes and replace discrete x,y values with Point and delegate to that class
- We'll also talk through "well-defined interfaces" for both external clients and subclasses. Why might we still want to have private data (instead of protected) when subclasses need access to that information?

```
3  public abstract class AbstractShape implements IShape {
4    protected int x;
5    protected int y;
6
7    public AbstractShape(int x, int y) {
```

Make private?

UML BEHAVIOR

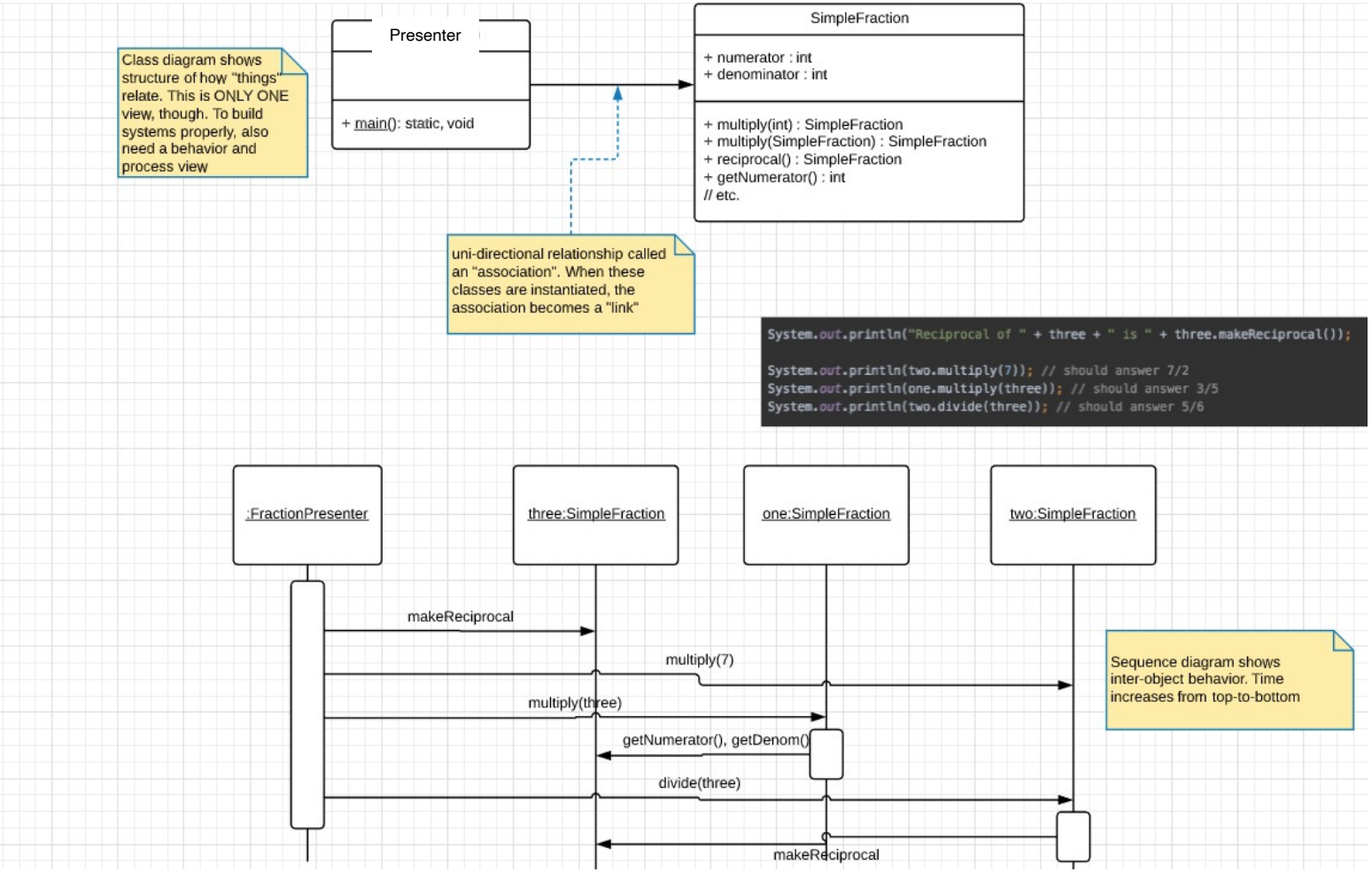
- Now that we're creating composite objects and delegating functionality to them, we need to consider inter-object communication in our designs
- Thus far, our class diagrams have given us a structural/static view of the world we're building
- We need to describe the behaviors as time progresses and objects collaborate with each other AND the behavior internal to the object as it changes state
- In UML, we do this through behavioral diagrams:
  - Sequence diagrams
  - Object Interaction diagrams
  - State diagrams

UML SEQUENCE DIAGRAMS

- Sequence diagrams describe the object interactions and lifelines of the instances participating in a particular scenario
- Unlike class diagrams, sequence diagrams
  - Focus on behavior allowed by the structure described in class diagrams
  - Show instances (objects) not classes
  - Identify communication paths that must exist via associations/links

https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/

UML: SEQUENCE DIAGRAMS



Q & A

THANKS!

- Stay safe, be encouraged, & see you next week!