McGill

COMP 303

Lecture 16

Inheritance III

Winter 2025

slides by Jonathan Campbell

Announcements

- Midterm tomorrow (Wednesday)
    - No class on Thursday
- Project
    - Final proposal grades out + review grades
    - Quiz 2 coming out soon for three-person groups
    - Link to be sent to schedule meeting with TA (milestone 2)
- https://www.twitch.tv/claudeplayspokemon

Today

- Inheritance
  - TEMPLATE METHOD pattern
  - Polymorphic copying with inheritance
  - Inheritance vs. composition
  - Decorator pattern with inheritance
  - Proper use of inheritance

Recap

Overloading methods

- Overloading is when multiple versions of a method are specified, **each with a different signature** (i.e., different explicit parameters).
    - Note: A method's signature only comprises its name and parameter types, not return type.

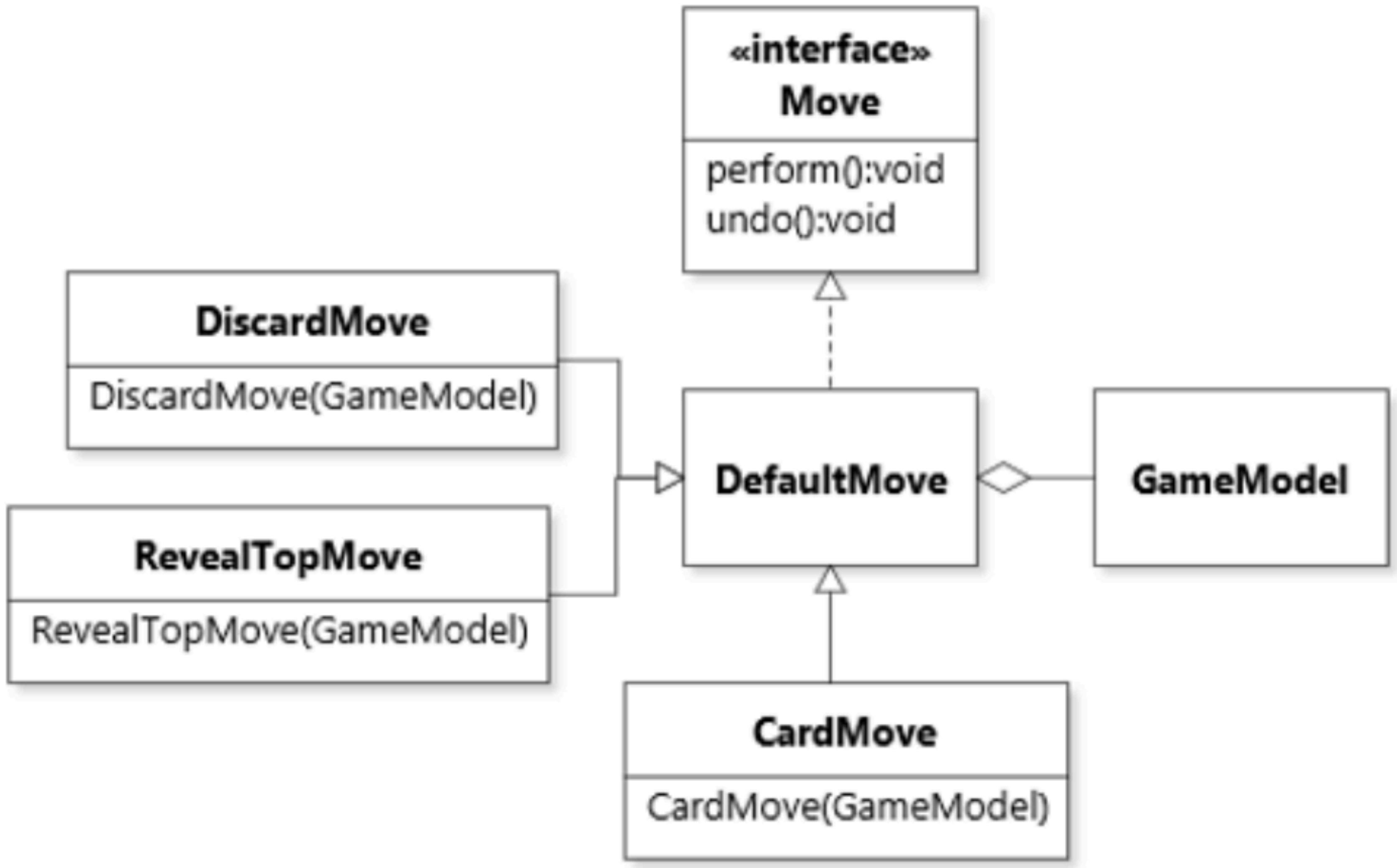
Overloading methods

```
public class MemorizingDeck extends Deck {
  private CardStack aDrawnCards = new CardStack();
  public MemorizingDeck() {
    /* c1: Does nothing besides initialization */
  }
  public MemorizingDeck(CardSource pSource) {
    /* c2: Copies all cards of pSource into this object */
  }
  public MemorizingDeck(MemorizingDeck pSource) {
    /* c3: Copies all cards and drawn cards of pSource
     * into this object */
  }
}
```

Overloading methods

- Java will select which method to run based on the **number and static (i.e., compile-time) types of the parameters** given to it.
    - It will choose the **most specific** out of the **compatible** options.

Abstract classes



Abstract classes

- An abstract class cannot be instantiated directly. Its constructor can only be called from subclass constructors.
- It does not need to specify an implementation for all methods. But any concrete (i.e., non-abstract) class inheriting from it will need to do so.
- We can also declare new abstract methods in an stract class (more on this later).

Template method pattern

TEMPLATE METHOD pattern

- Continuing with our example of card game moves, with a Move interface that supplies a perform method, assume that perform, for any kind of Move, should do the following:

  1. Add the move to an undo stack, so that it could be undone later.
  2. Perform the actual move.
  3. Log the move by printing out or writing somewhere about it.

Template Method pattern

```
public void perform() {
  aModel.pushMove(this); // step 1

  // step 2
  /* Code here to actually perform the move */

  log(); // step 3
}
```

TEMPLATE METHOD pattern

- If the move pushing/logging (step 1 & 3) is the same for all moves, then we should really put this in our AbstractMove class, so that the subclass Moves can inherit it.
    - Then, we avoid duplicated code (anti-pattern).
    - Also, we avoid the issue of a Move forgetting to push or log.

TEMPLATE METHOD pattern

- But, perform depends in some way on the code for the actual move (step 2). So we cannot completely implement perform in the AbstractMove class.
    - Solution: put "**hooks**" in AbstractMove#perform to allow subclasses to provide specialized functionality where needed.
    - Called the **TEMPLATE METHOD** pattern.

Template Method pattern

```
public abstract class AbstractMove implements Move {
  protected final GameModel aModel;
  protected AbstractMove(GameModel pModel) {
    aModel = pModel;
  }
  public final void perform() {
    aModel.pushMove(this);
    execute(); // subclasses will override
    log();
  }
  protected abstract void execute();
  private void log() {
    System.out.println(getClass().getName());
  }
}
```

Defined as final, so that it cannot be overridden by subclasses.

execute is defined as protected, so that subclasses can override.

Also defined as abstract, because there is no behaviour for an abstract move.

TEMPLATE METHOD pattern

```
from abc import ABC, abstractmethod
from typing import final

class AbstractMove(ABC):
    def __init__(self, model):
        self._model = model

    @final
    def perform(self):
        self._model.push_move(self)
        self._execute()
        self._log()

    @abstractmethod
    def _execute(self):
        pass

    def _log(self):
        print(self.__class__.__name__)
```

@final in Python: not enforced at runtime; IDE will check

TEMPLATE METHOD pattern

- The pattern is so called because the common method in the superclass is a **template**, which gets instantiated differently for each subclass.
  - The template method will call the concrete and abstract **step methods**.
  - The step methods must have a different signature from the template method (which should be declared final so that it cannot be overridden).

final keyword in Java

- Final has a different meaning for fields, methods, classes:
    - A final field cannot be assigned a value more than once.
    - A final method cannot be overridden.
    - A final class cannot be inherited.

Polymorphic copying w/ inheritance

copy() with subclasses

```java
public Deck copy() { // in Deck
  Deck deck = new Deck();
  deck.aCards = new CardStack(aCards);
  return deck;
}

Deck deck = new MemorizingDeck();
Deck copy = deck.copy(); // Error
```

- If we don't implement copy() in MemorizingDeck, then it will use the copy() implemented in Deck. But that won't actually give us a copy of our MemorizingDeck.

Polymorphic copying w/ subclasses

- To support polymorphic copying, we must then override copy() in all leaf classes.

```
public MemorizingDeck copy() {
  MemorizingDeck deck = new MemorizingDeck();
  deck.aCards = new CardStack(aCards); // error
  deck.aDrawnCards = new CardStack(aDrawnCards);
  return deck;
}
```

- Problem: We can't correctly update the aCards field in our new object, because it is private to Deck.

Solution: implement Cloneable

- To fix this, we will make all classes in the class hierarchy (i.e., everything implementing CardSource, including subclasses) implement the Cloneable interface.
    - The interface defines a single method, clone(), which we must override.

clone

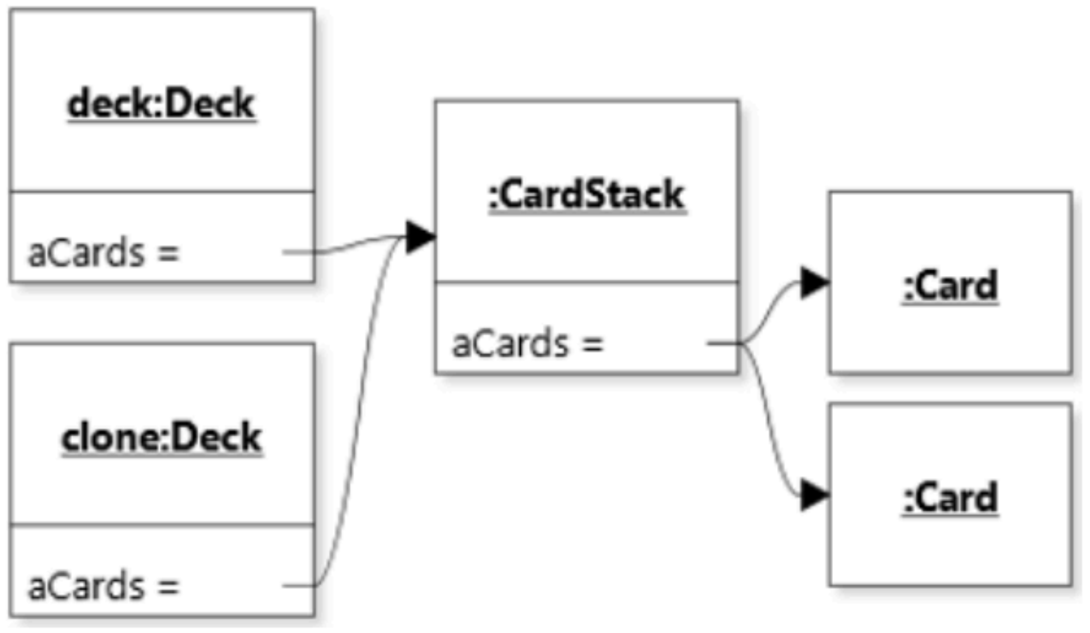
- The first thing in our clone method should be a call to the superclass clone method (which could be Object#clone).

```
public Deck clone() {
  Deck clone = (Deck) super.clone();
  // ...
  return clone;
}
```

- Using metaprogramming, Object#clone will make a field-by-field **shallow** copy of the current object and return it.

clone

- If any of our fields are mutable, simply using Object#clone won't be sufficient, since we want to make a deep copy.
- In this case, a shallow Object#clone copy would copy a reference to the same CardStack, leading to issues later on.



clone

```
public Deck clone() {
  Deck clone = (Deck) super.clone();
  clone.aCards = new CardStack(aCards);
  return clone;
}
```

- Here, after calling to super.clone, we make our own copy of any mutable fields before returning.

clone

```
public MemorizingDeck clone() {
  MemorizingDeck clone = (MemorizingDeck) super.clone();
  clone.aDrawnCards = new CardStack(aDrawnCards);
  return clone;
}
```

Inheritance vs. composition

Inheritance vs. composition

- Inheritance and composition can often be used to accomplish the same task.
- We will see an example of this, then discuss the pros and cons of using each approach.

MemorizingDeck w/ inheritance

```
public class MemorizingDeck extends Deck {
  private final CardStack aDrawCards = new CardStack();
  public void shuffle() {
    super.shuffle();
    aDrawnCards.clear();
  }
  public Card draw() {
    Card card = super.draw();
    aDrawnCard.push(card);
    return card;
  }
}
```

MemorizingDeck w/ composition

```
public class MemorizingDeck implements CardSource {
  private final CardStack aDrawCards = new CardStack();
  private final Deck aDeck = new Deck();
  public boolean isEmpty() {
    return aDeck.isEmpty();
  }
  public void shuffle() {
    aDeck.shuffle();
    aDrawnCards.clear();
  }
  public Card draw() {
    Card card = aDeck.draw();
    aDrawnCard.push(card);
    return card;
  }
}
```

Differences

- Using composition, the MemorizingDeck aggregates a **separate** Deck object, and called out to it when needed.
- With inheritance, we don't have a separate Deck object; we have access to all the fields and methods on our current object, and call out to the super method when needed.
  - Since we inherit all methods, we don't have to re-specify any implementation that we may not want to override (e.g., isEmpty), unlike when using composition.

Differences

- Using inheritance can introduce some bugs, though. For example, if our Deck constructor calls shuffle(), we will get a NullPointerException.
  - That's because the default MemorizingDeck constructor (which initializes aDrawnCards) will only be called after the Deck constructor (and thus the shuffle method). But the shuffle method, overridden in MemorizingDeck, tries to access aDrawnCards.

Differences

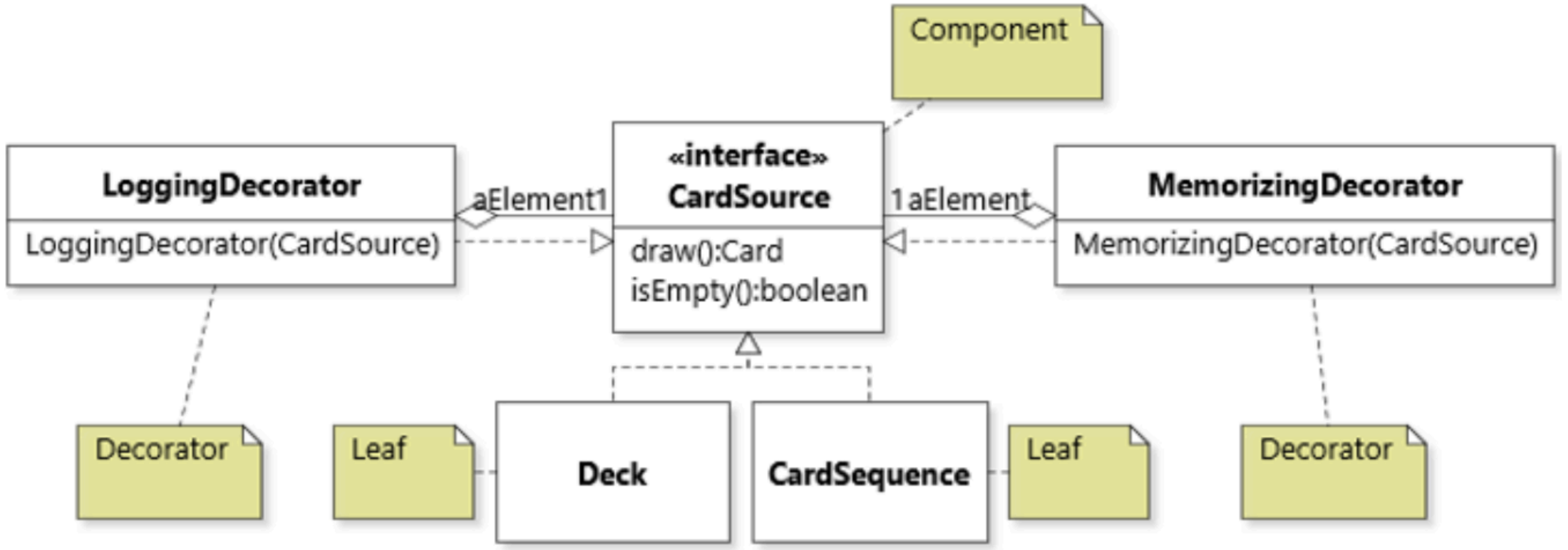
- Composition-based designs generally provides more runtime flexibility.
    - Because we can change the configuration of our composite object at runtime (we can switch the Deck being memorized to something else, if the field was of type CardSource).
    - But they provide fewer options for detailed access to the internal state of a well-encapsulated object (since it would be private).

Differences

- Inheritance-based designs tend to have more flexible compile-time configuration.
    - Since we can designate internal state to be protected instead of private, the subclass can access it.
    - They also support finer-grained polymorphism (e.g., different compile-time types for variables), while in composition, we typically work with a common interface or abstract type (e.g., CardSource).

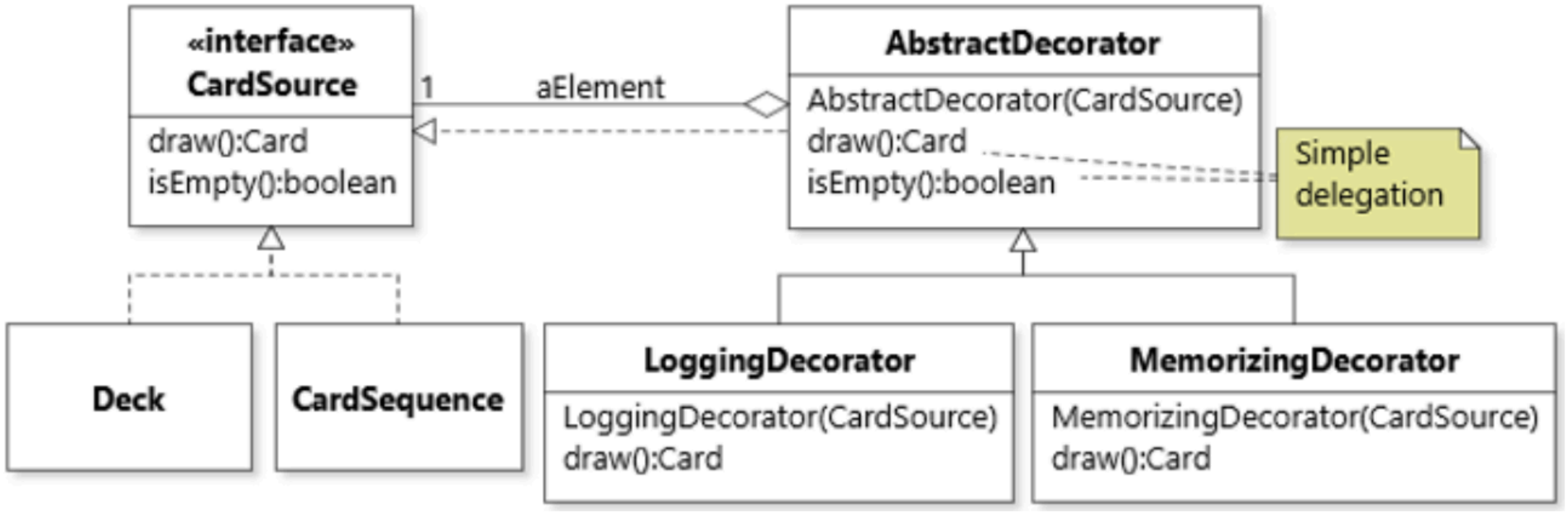
Decorator w/ inheritance

Revisiting DECORATOR



Each Decorator class must aggregate an object of the component interface (aElement). This duplicate code can be reduced using inheritance.

Revisiting DECORATOR



Now the object to be decorated (aElement) is defined in the AbstractDecorator class, and default delegation is implemented there also.

AbstractDecorator

```
public abstract class AbstractDecorator implements CardSource {
  private final CardSource aElement;
  protected AbstractDecorator(CardSource pElement) {
    aElement = pElement;
  }
  public Card draw() {
    return aElement.draw();
  }
  public boolean isEmpty() {
    return aElement.isEmpty();
  }
}
```

Decorated element is private.

Methods implementing the interface just delegate to the decorated element, with no special behaviour.

AbstractDecorator

```
public class LoggingDecorator extends AbstractDecorator {
  public LoggingDecorator(CardSource pElement) {
    super(pElement);
  }
  public Card draw() {
    Card card = super.draw();
    System.out.println(card);
    return card;
  }
}
```

Call AbstractDecorator#draw (the default behaviour).

Then add the special behaviour.

isEmpty is not implemented; we inherit the AbstractDecorator version.

When not to use inheritance

When not to use inheritance

- **Liskov Substitution Principle**: Subclasses should not restrict what clients of the superclass can do with an instance.

```
public class UnshufflableDeck extends Deck {
  public void shuffle() {
    /* Do nothing */
    // or: throw new OperationNotSupportedException();
  }
}
```

When not to use inheritance

- If we pass an UnshufflableDeck to the following method, the code won't work as expected, or raise an exception.

```
private Optional<Card> shuffleAndDraw(Deck pDeck) {
  pDeck.shuffle();
  if (!pDeck.isEmpty()) {
    return Optional.of(pDeck.draw());
  }
  else {
    return Optional.empty();
  }
}
```

Liskov Substitution Principle

- Per LSP, methods of a subclass:
  - cannot have stricter preconditions;
  - cannot have less strict postconditions;
  - cannot take more specific types as parameters;
  - cannot make the method less accessible (e.g., public -> protected);
  - cannot throw more checked exceptions; and
  - cannot have a less specific return type.
- The last four are automatically checked by the compiler.

Stricter precondition example

```
public class Deck implements CardSource {
  protected final CardStack aCards = new CardStack();
  public Card draw() { return aCards.pop(); }
  public boolean isEmpty() { return aCards.isEmpty(); }
}
public class DrawBestDeck extends Deck {
  public Card draw() {
    Card card1 = aCards.pop();
    Card card2 = aCards.pop();
    Card high = // get highest card betw. card1 & card2
    Card low = // get lowest card betw. card1 & card2
    aCards.push(low);
    return high;
  }
}
```

DrawBestDeck#draw assumes that there are at least two cards in the Deck. We need to write a check for this.

Stricter precondition example

```
public Card draw() {
  Card card1 = aCards.pop();
  if (isEmpty()) {
    return card1;
  }
  Card card2 = aCards.pop();
  ...
}
```

Not as elegant, more time to test, etc.

Stricter precondition example

```
public class DrawBestDeck extends Deck {
  public int size() {
    return aCards.size();
  }
  public Card draw() {
    assert size() >= 2;
    ...
  }
}
```

Now DrawBestDeck#draw has a stricter precondition than Deck#draw: violation of the LSP.

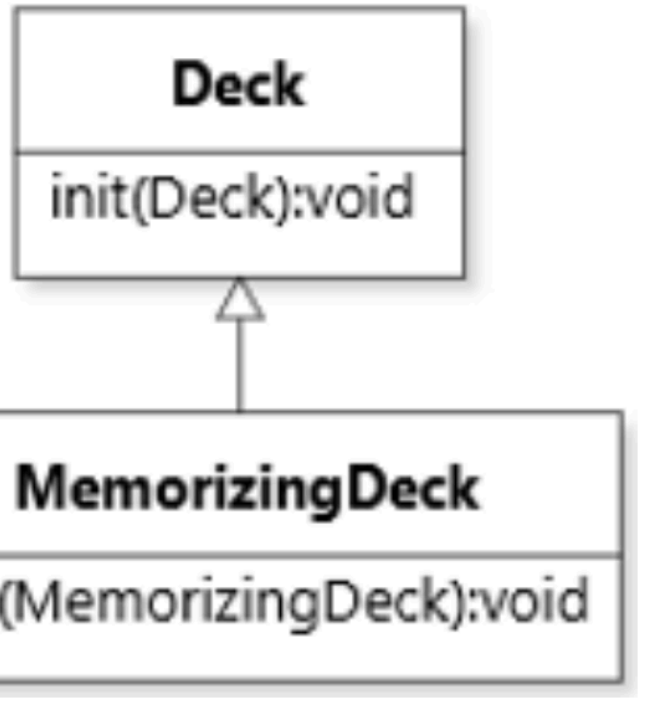
That's bad, because code that depends on the Deck#draw precondition will fail:

```
if (deck.size() >= 1) {
  return deck.draw();
}
```

Another problem: Client code cannot check size() before calling draw() on a CardSource, because size is not part of the CardSource interface.

More specific parameters

Deck#init will initialize the cards in the deck to be same as the input Deck.



MemorizingDeck#init will overload Deck#init, and take a MemorizingDeck instead of Deck as input, so that it can copy both the aCards and aDrawnCards.

More specific parameters

- Doing so would lead to strange behaviours. (Recall what happens when we call an overloaded method - it depends on the compile-time type of the object.)

```
MemorizingDeck deck = new MemorizingDeck();
MemorizingDeck memorizingDeck = new MemorizingDeck();
Deck mDeck = memorizingDeck;
deck.init(memorizingDeck); // Calls MemorizingDeck.init
deck.init(mDeck); // Calls Deck.init
```

When not to use inheritance

- A subclass must require both of the following:
  - reuse of the class member declarations of the base class, and
  - subtype-supertype relation ("is-a") between the subclass and superclass.
- If we only inherit for reuse, but not for the second purpose, it is considered an abuse of inheritance.
  - In such case, composition should be used instead of inheritance.

References

- Robillard ch. 7.6-7.7, 7-9-7.11, p.171-177, 181-193.
    - Exercises #4-10: https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter7.md

Coming up

- Next lecture:
  - New topic!