McGill

COMP 303

Lecture 20

Functional design

Winter 2025

slides by Jonathan Campbell

Announcements

- Project
  - Team survey coming soon...
  - Your building image
  - Final demos: April 3-8; schedule to be sent later tonight/tomorrow.
  - Uploading code to remote server.

Today

- More about Visitor pattern
- Functional design

Recap

Events

- Events are typically defined by the component library.
    - E.g., TextField defines an event that occurs when the user types the [enter] key.
- After we instantiate a component, we must create and register an **event handler**: the code that will execute when this event occurs.

Event handlers

```
public class IntegerPanel extends Parent implements Observer {
  private TextField aText = new TextField();
  private Model aModel;
  public IntegerPanel(Model pModel) {
    aModel = pModel;
    aModel.addObserver(this);
    aText.setText(new Integer(aModel.getNumber()).toString());
    getChildren().add(aText);
    aText.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent pEvent) {
        int number;
        try {
          number = Integer.parseInt(aText.getText());
        } catch(NumberFormatException pException ) {
          number = 1;
        }
        aModel.setNumber(number);
      }
    });
  }
}
```

setOnAction: registering a new event handler

defining an anonymous class, subtype of EventHandler

handle method will be called when the event occurs.

Event handlers

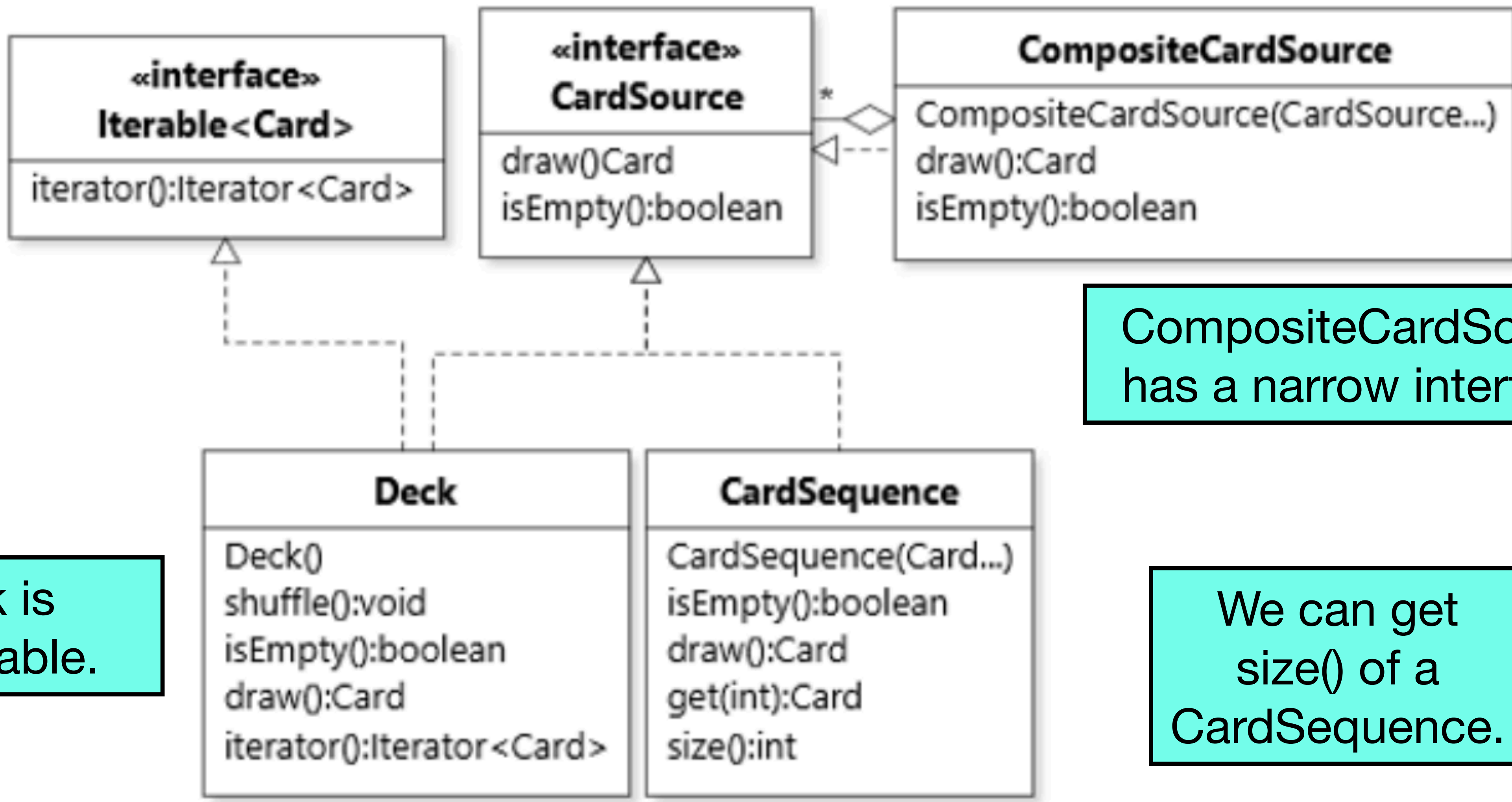
```
class IntegerPanel(tk.Frame):
  def __init__(self, parent, model):
    super().__init__(parent)
    self.__aModel = model
    self.__aModel.add_observer(self)

    self.__aText = tk.Entry(self)
    self.__aText.insert(0, str(self.__aModel.get_number()))
    self.__aText.pack()

    self.__aText.bind("<Return>", self.on_enter)

  def on_enter(self, event):
    try:
      number = int(self.__aText.get())
    except ValueError:
      number = 1
    self.__aModel.set_number(number)
```

bind: registering a new event handler

defining a method taking an event parameter

method will be called when the event occurs.

CardSource



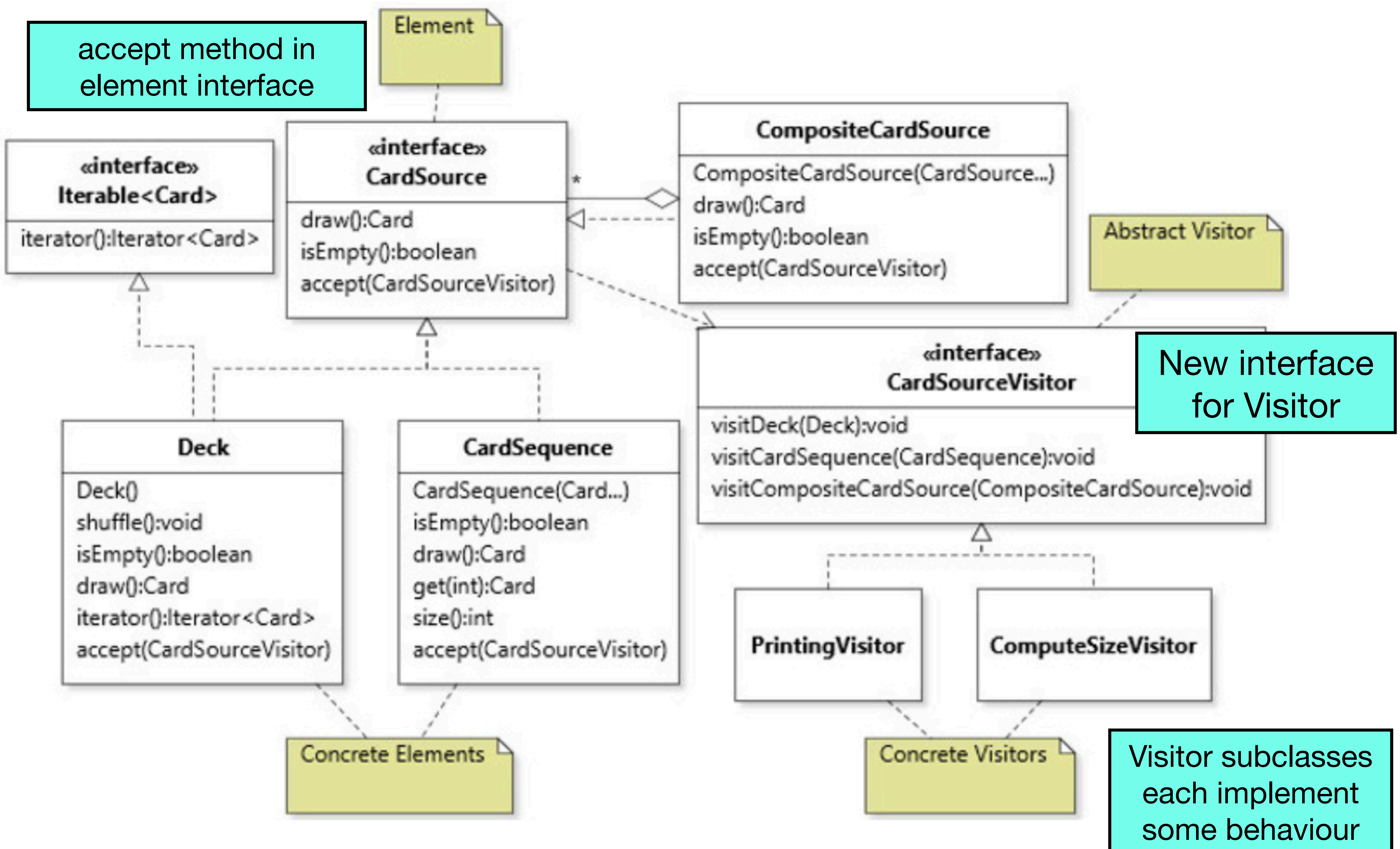
CompositeCardSource has a narrow interface.

Deck is shuffleable.

We can get size() of a CardSequence.

Three different types of CardSources, each having slightly different methods.

Solution: VISITOR pattern



VISITOR pattern

- Solution: **VISITOR** pattern.
- Idea: Define functionality (like `contains(Card)`) in its own class.
- Three parts:
  - Abstract Visitor interface
  - Concrete Visitors (one for each behaviour)
  - accept methods in element subtypes

Abstract visitor

```java
public interface CardSourceVisitor {
  void visitCompositeCardSource(CompositeCardSource pSource);
  void visitDeck(Deck pDeck);
  void visitCardSequence(CardSequence pCardSequence);
  // ...
}
```

A visit method for every different element subclass.

Concrete visitor

```
public class PrintingVisitor implements CardSourceVisitor {
  public void visitCompositeCardSource(CompositeCardSource pSource)
  {}

  public void visitDeck(Deck pDeck) {
    for (Card card : pDeck) {
      System.out.println(card);
    }
  }

  public void visitCardSequence(CardSequence pCardSequence) {
    for (int i = 0; i < pCardSequence.size(); i++) {
      System.out.println(pCardSequence.get(i));
    }
  }
}
```

A concrete visitor for every different behaviour.

Integrating the visitors

```
public interface CardSource {
  Card draw();
  boolean isEmpty();
  void accept(CardSourceVisitor pVisitor);
}
```

Integrating the visitors

- The accept method will simply call the relevant visit method on the visitor.

```java
public class Deck {
  public void accept(CardSourceVisitor pVisitor) {
    pVisitor.visitDeck(this);
  }
}
public class CardSequence {
  public void accept(CardSourceVisitor pVisitor) {
    pVisitor.visitCardSequence(this);
  }
}
```

Invoking the behaviour

```
// in client code
Deck deck = new Deck();

PrintingVisitor visitor = new PrintingVisitor();
deck.accept(visitor);
```

Visitor pattern pt. 2

Question

A visit method for every different element subclass.

```
public interface CardSourceVisitor {
  void visitCompositeCardSource(CompositeCardSource pSource);
  void visitDeck(Deck pDeck);
  void visitCardSequence(CardSequence pCardSequence);
  // ...
}
```

Question

A visit method for every different element subclass.

```
public interface CardSourceVisitor {
  void visit(CompositeCardSource pSource);
  void visit(Deck pDeck);
  void visit(CardSequence pCardSequence);
  // ...
}
```

Why can't we write this?

Question

A visit method for every different element subclass.

```
public interface CardSourceVisitor {
  void visit(CompositeCardSource pSource);
  void visit(Deck pDeck);
  void visit(CardSequence pCardSequence);
  // ...
}
```

Why can't we write this?

Answer: We can, because when we called an overloaded method, Java will select the most specific method based on the parameter compile-time types.

Question

A visit method for every different element subclass.

```
public interface CardSourceVisitor {
  void visit(CompositeCardSource pSource);
  void visit(Deck pDeck);
  void visit(CardSequence pCardSequence);
  // ...
}
```

Why can't we write this?

Answer: We can, because when we called an overloaded method, Java will select the most specific method based on the parameter compile-time types.

But: method overloading is frowned upon, so we prefer the other approach.

Main idea

- If you have some functionality that you want to add to existing classes (maybe in different parts of the class hierarchy) **without modifying those classes directly**.
    - The classes only need to be able to **accept** a Visitor, which could be any behaviour.

CompositeCardSource

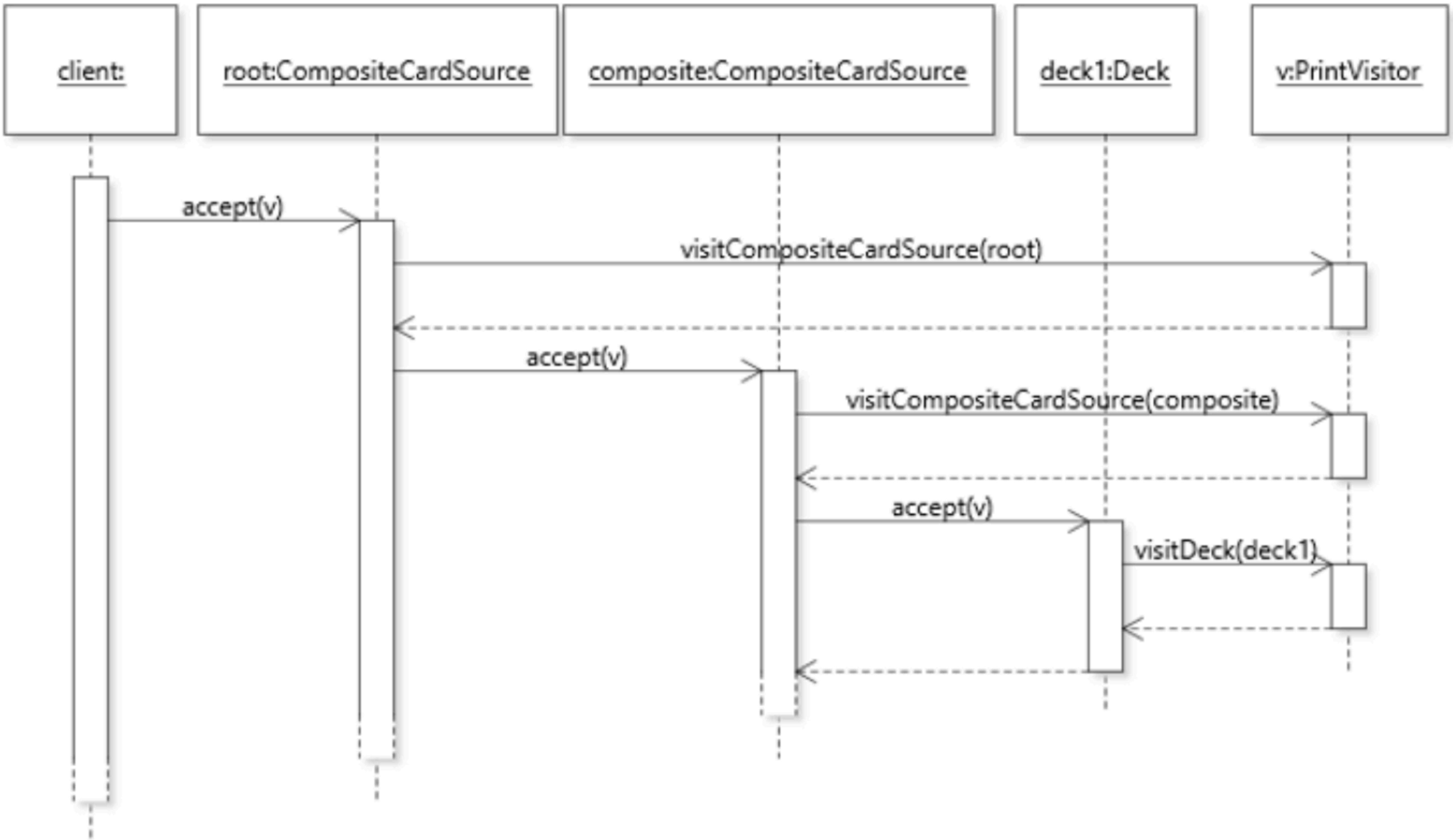
- We have to do some extra work to make the Visitor pattern work with CompositeCardSource.

- If we apply an operation (like print, size, etc.) to a CompositeCardSource, we really want the operation to be applied to all of its aggregated elements.

CompositeCardSource

```
public class CompositeCardSource implements CardSource {
  private final List<CardSource> aElements;

  public void accept(CardSourceVisitor pVisitor) {
    pVisitor.visitCompositeCardSource(this);
    for (CardSource source : aElements) {
      source.accept(pVisitor);
    }
  }
}
```

CompositeCardSource



CompositeCardSource

- We could have instead placed this same code inside the visitCompositeCardSource method, instead of accept:

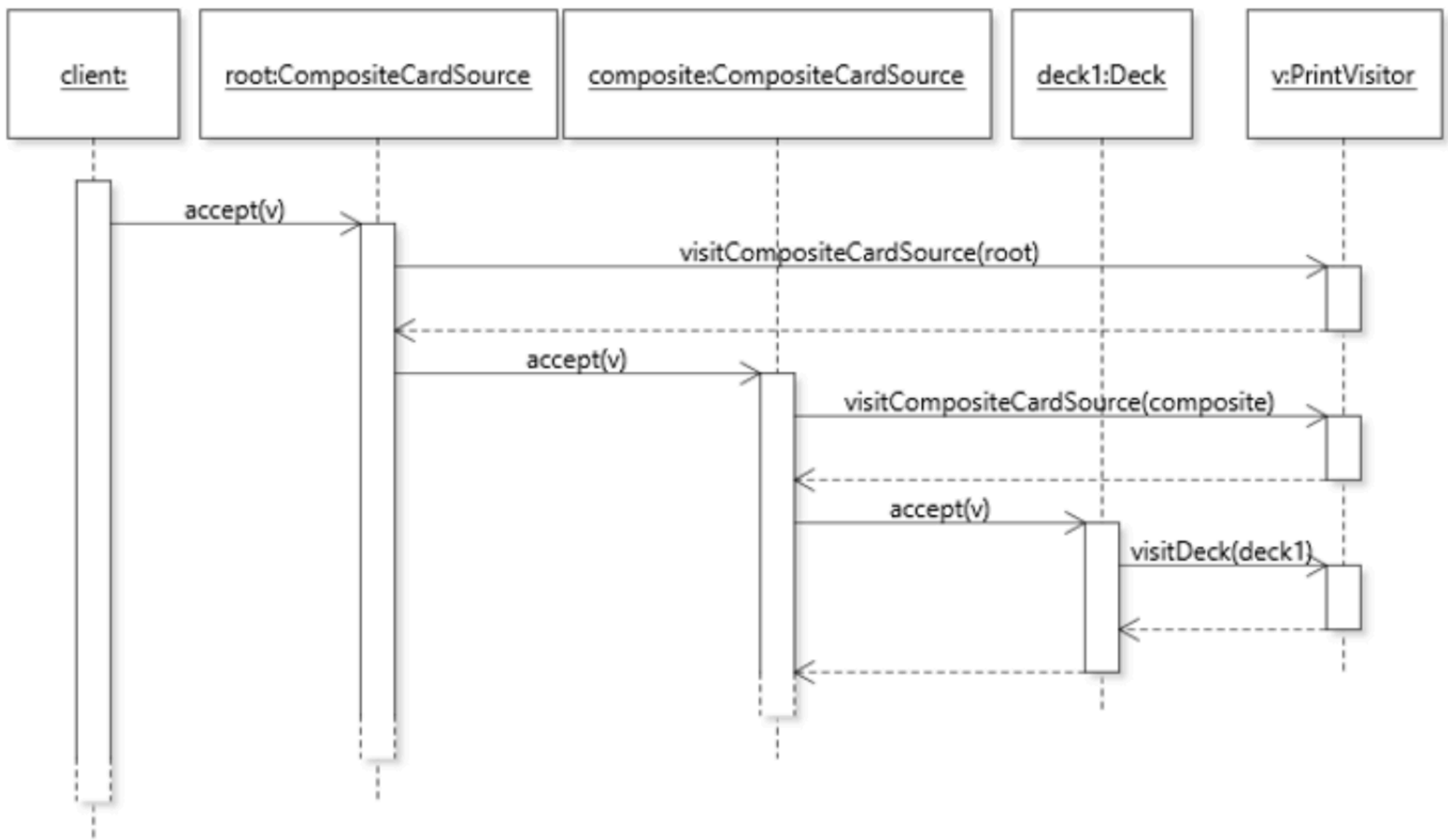
```
public class PrintVisitor implements CardSourceVisitor {
  public void visitCompositeCardSource(CompositeCardSource pCompCardSrc) {
    for (CardSource source : pCompositeCardSource) {
      source.accept(this);
    }
  }
  ...
}
```

- (Since this class can't access the private aElements field of the composite, we'd have to make the composite iterable.)

CompositeCardSource

```
public class CompositeCardSource implements CardSource,
                                            Iterable<CardSource> {
  private final List<CardSource> aElements;

  public Iterator<CardSource> iterator() {
    return aElements.iterator();
  }
  ...
}
```

CompositeCardSource



CompositeCardSource

- The advantage to placing the traversal code in the visit method is that, depending on the behaviour, we can change the order of traversal, if we wanted.

- The downside is that we have to make the composite class iterable, possibly making its encapsulation weaker.

  - Another downside is that the traversal code would be repeated in every concrete visitor (DUPLICATED CODE).

Visitor with inheritance

```java
public abstract class AbstractCardSourceVisitor implements CardSourceVisitor {
  public void visitCompositeCardSource(CompositeCardSource pCompositeCardSrc) {
    for (CardSource source : pCompositeCardSource) {
      source.accept(this);
    }
  }

  public void visitDeck(Deck pDeck) {}
  public void visitCardSequence(CardSequence pCardSequence) {}
}
```

Avoids duplicated code problem.

Visitor with data flow

- All of our visit methods have been void so far.
- But we may want to return information from them. E.g., a size visitor should return the size.
  - But, all visit methods must return void, or else they wouldn't implement the abstract visitor interface.
  - Instead, we will store the computed data into the visitor object.

Visitor with data flow

```java
public class CountingVisitor extends AbstractCardSourceVisitor {
  private int aCount = 0;

  public void visitDeck(Deck pDeck) {
    for (Card card : pDeck) {
      aCount++;
    }
  }

  public void visitCardSequence(CardSequence pCardSequence) {
    aCount += pCardSequence.size();
  }

  public int getCount() {
    return aCount;
  }
}
```

Visitor with data flow

```
// in client code
CountingVisitor visitor = new CountingVisitor();

root.accept(visitor);
int result = visitor.getCount();
```

Functional design

Functional design

- **Higher-order function**: a function that takes another function as an argument.
  - Only certain programming languages (including Java and Python) support this. In particular, those that support first-class functions.

Back to Comparator

- We saw a while ago that, to compare two cards, we could implement the Comparator<T> interface:

```
List<Card> cards = ...;
Collections.sort(cards, new Comparator<Card>() {
  public int compare(Card pCard1, Card pCard2) {
    return pCard1.getRank().compareTo(pCard2.getRank());
  }
});
```

- (Here, we created an anonymous class to implement the interface.)

Back to Comparator

```
List<Card> cards = ...;
Collections.sort(cards, new Comparator<Card>() {
  public int compare(Card pCard1, Card pCard2) {
    return pCard1.getRank().compareTo(pCard2.getRank());
  }
});
```

- One problem with this, from a design point of view, is that we are passing an **object** (of an anonymous class) to Collections.sort.
  - But object implies a collection of data and methods to operate on the data. Here there is only a method, no data.

Back to Comparator

- Instead, we can pass a **method reference**.

- First, we'll define a new comparison method in Card:

```
public class Card {
  public static int compareByRank(Card pCard1, Card pCard2) {
    return pCard1.getRank().compareTo(pCard2.getRank());
  }
}
```

- Then, when we call sort, we will pass a reference.

```
Collections.sort(cards, Card::compareByRank);
```

Higher-order functions

- Collections.sort is an example of a higher-order function, because it can take a function as argument.
  - It then applies that function, in this case, to compare the cards and sort the list.
- Higher-order functions can lead to a larger design space to explore, and their use can help to realize and apply design patterns.

Higher-order functions

- We've seen that we can define an anonymous class that implements some interface, and pass that as an argument, or pass a method reference.

- We will now see how to define an **anonymous function** (called a lambda expression). To do so, we need to learn about functional interfaces first.

Functional interfaces

- Any interface with a single abstract method.
  (It could have default and/or static methods too.)

```
public interface Filter {
  boolean accept(Card pCard);
}
```

Functional interfaces

- Here's an anonymous class that implements Filter:

```
Filter blackCardFilter = new Filter() {
  public boolean accept(Card pCard) {
    return pCard.getSuit().getColor() == Suit.Color.BLACK;
  }
};
```

Functional interfaces

- Comparator<T>, which defines a single abstract method compare, can similarly be considered a functional interface.

Functional interfaces

- Java already defines some functional interfaces in java.util.function. One of them is called Predicate<T>, which defines an abstract method test that takes an argument of type T and returns a boolean; we can use this instead of our Filter interface:

```
Predicate<Card> blackCardFilter = new Predicate<Card>() {
  public boolean test(Card pCard) {
    return pCard.getSuit().getColor() == Suit.Color.BLACK;
  }
};
```

Lambda expressions

- To create an instance of a functional interface, we have defined a new instance of an anonymous class.

- But there's a much cleaner way to do it, by defining a **lambda expression**, which we can think of as an anonymous function (since we define it without a name).

Lambda expressions

functional interface type

name of the new instance
of our functional interface

```
Predicate<Card> blackCardFilter =
   (Card card) -> card.getSuit().getColor() == Suit.Color.BLACK;
```

parameters for the abstract
function declared in interface
(must be of compatible types)

body
(must evaluate to same type as specified in interface)

The same effect as defining an anonymous class to
implement our functional interface, but much cleaner.

Lambda expressions

- Three parts:
  - list of parameters; if none, put empty parentheses ()
  - the right arrow ->
  - body,
    - either a single expression, the result of which is automatically returned, or
    - a block of statements including an explicit return, inside {}

Lambda expressions

```
Predicate<Card> blackCards = (Card card) ->
    { return card.getSuit().getColor() == Suit.Color.BLACK; };
```

Defining a block makes the lambda expression more complicated, so we like to use single expressions whenever possible.

Lambda expressions

```
Predicate<Card> blackCardFilter =
   (card) -> card.getSuit().getColor() == Suit.Color.BLACK;
```

Since the parameter type(s) are specified in the interface, we can omit them when defining the lambda expression.

Lambda expressions

```
Predicate<Card> blackCardFilter =
    card -> card.getSuit().getColor() == Suit.Color.BLACK;
```

If there is just one parameter, we can even omit the parentheses.

Using lambda expressions

```
Predicate<Card> blackCardFilter =
   card -> card.getSuit().getColor() == Suit.Color.BLACK;

Deck deck = ...
int total = 0;
for (Card card : deck) {
  // calling our test method just defined above
  if (blackCardFilter.test(card)) {
    total ++;
  }
}
```

Using lambda expressions

- Many Java libraries define methods that accept functional interface types. For instance, ArrayList::removeIf takes a Predicate<T> to remove all objects that match some condition:

```
ArrayList<Card> cards = ...
cards.removeIf(card ->
     card.getSuit().getColor() == Suit.Color.BLACK);
```

- (The lambda expression is matched to the functional interface type of the removeIf method.)

Method references

- If we already have a method defined in a class, e.g.:

```
public final class Card {
  public boolean hasBlackSuit() {
    return aSuit.getColor() == Color.BLACK;
  }
}
```

- We could write a lambda that simply calls this method.

```
cards.removeIf(card -> card.hasBlackSuit());
```

Method references

- Or, we could pass a reference to the method directly, which reads almost like a regular (spoken language) sentence!

```
cards.removeIf(Card::hasBlackSuit);
```

- It is interpreted by the compiler as a shortcut to the full lambda expression:

```
cards.removeIf(card -> card.hasBlackSuit());
```

- (which is valid since removeIf takes a Predicate<T>, which has a single test method taking T type and returning bool, which is exactly what this lambda does.)

Method references

- In our example, we passed a reference to an instance method (of an arbitrary object).
- We can also pass a reference to a static method, or to an instance method of a particular object.

Reference to static method

```
public final class CardUtils {
  public static boolean hasBlackSuit(Card pCard) {
    return pCard.getSuit().getColor() != Color.BLACK;
  }
}

// passing lambda expression
cards.removeIf(card -> CardUtils.hasBlackSuit(card));

// passing reference to static method
cards.removeIf(CardUtils::hasBlackSuit);
```

Reference to instance method (2)

- Suppose we want to remove all cards in our List<Card> that have the same color (red/black) as the top card of a Deck (which has a method topSameColorAs).

```
Deck deck = new Deck();

// passing lambda expression
cards.removeIf(card -> deck.topSameColorAs(card));

// passing instance method of the deck object
cards.removeIf(deck::topSameColorAs);
```

Method references

- All the methods used as references seen in our examples (Card::hasBlackSuit, CardUtils::hasBlackSuit and deck::topSameColorAs) take a single input and return a boolean.

- Thus, they are compatible with the Predicate<T> functional interface, which is taken by removeIf.

- The lambda expression or method reference **must** be compatible with the parameter type of the method to which we are passing them.

Lambdas in Python

- Lambdas in python are defined using the lambda keyword:

```
x = lambda a: a + 10
print(x(5)) # prints 15
```

- Unlike in Java, we can't define a block of statements as the body. We can only use a single expression for the body.

Lambdas in Python

- A lambda is of type Callable, and we can specify the parameter and return types in its type annotation.

```
from typing import Callable

multiply: Callable[[int, int], int] = lambda x, y: x * y

result = multiply(5, 3)
print(result) # prints 15
```

Lambdas in Python

- One common use of a lambda expression in Python is to use it to sort a list, using the key parameter for sort.

```
vals = ["AA", "AD", "AZ", "AG"]

# sort according to character at index 1
vals.sort(key = lambda s: s[1])
```

References

- Robillard ch. 8.8, p.232-242
- Robillard ch. 9-9.2, 9.5, p.243-252, 261-264
  - Exercises #1-4: https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter9.md

Coming up

- Next lecture:
  - New topic!