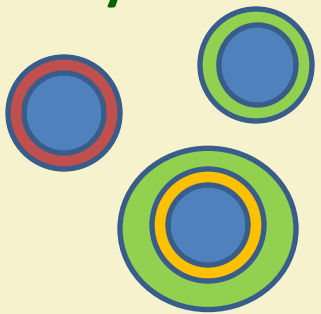
Decorator Pattern

Decorator Pattern: Introduction

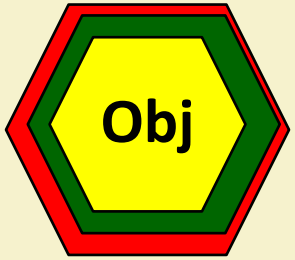
- **Intent:**
  - **Attach additional responsibilities to objects dynamically.**
  - Provides a flexible alternative to subclassing.

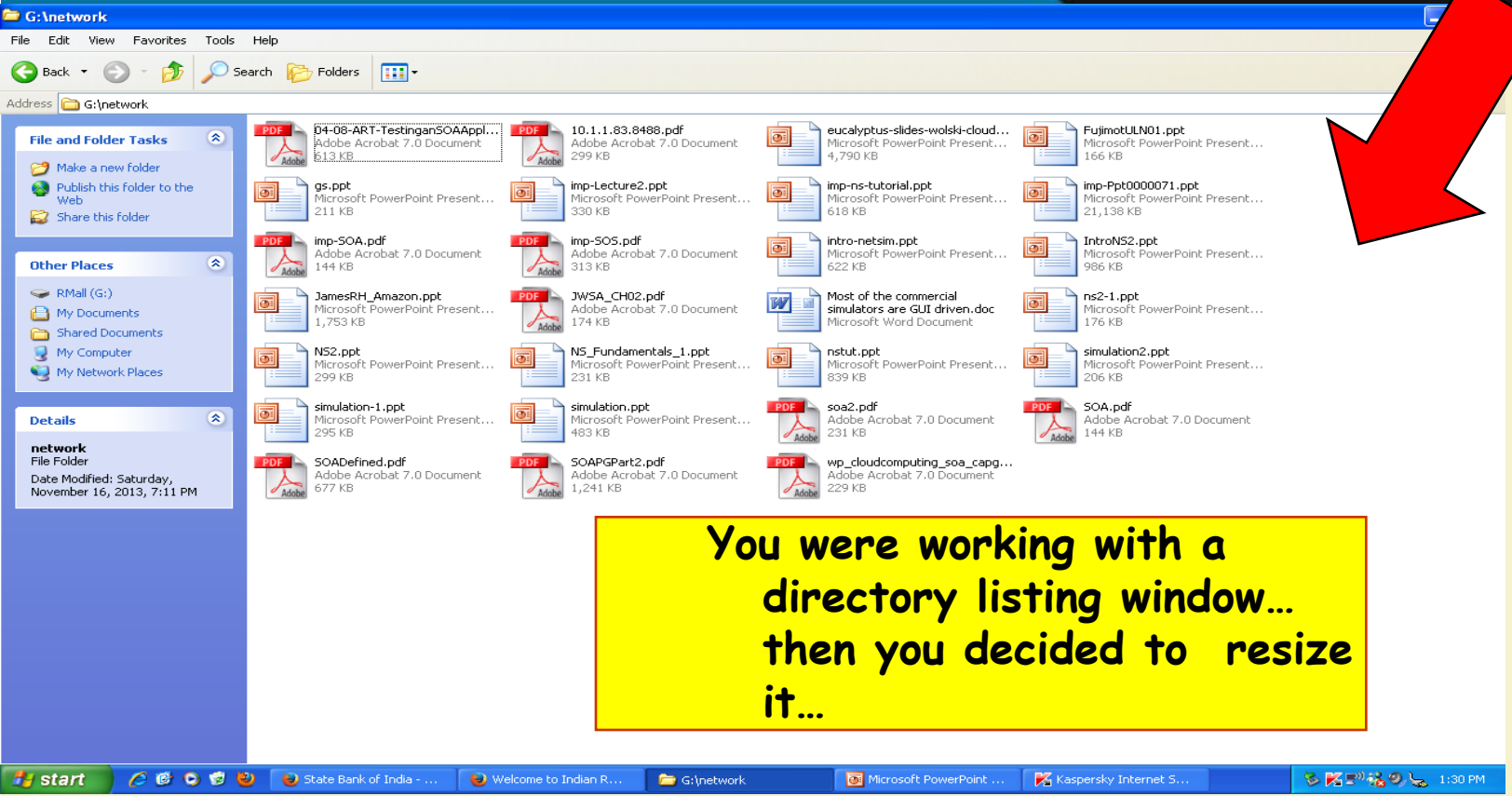


- A **wrapper pattern!**
- **Motivation:**
  - **Add responsibilities to individual objects as and when required and not to an entire class**
  - Should conform to the interface of the object.

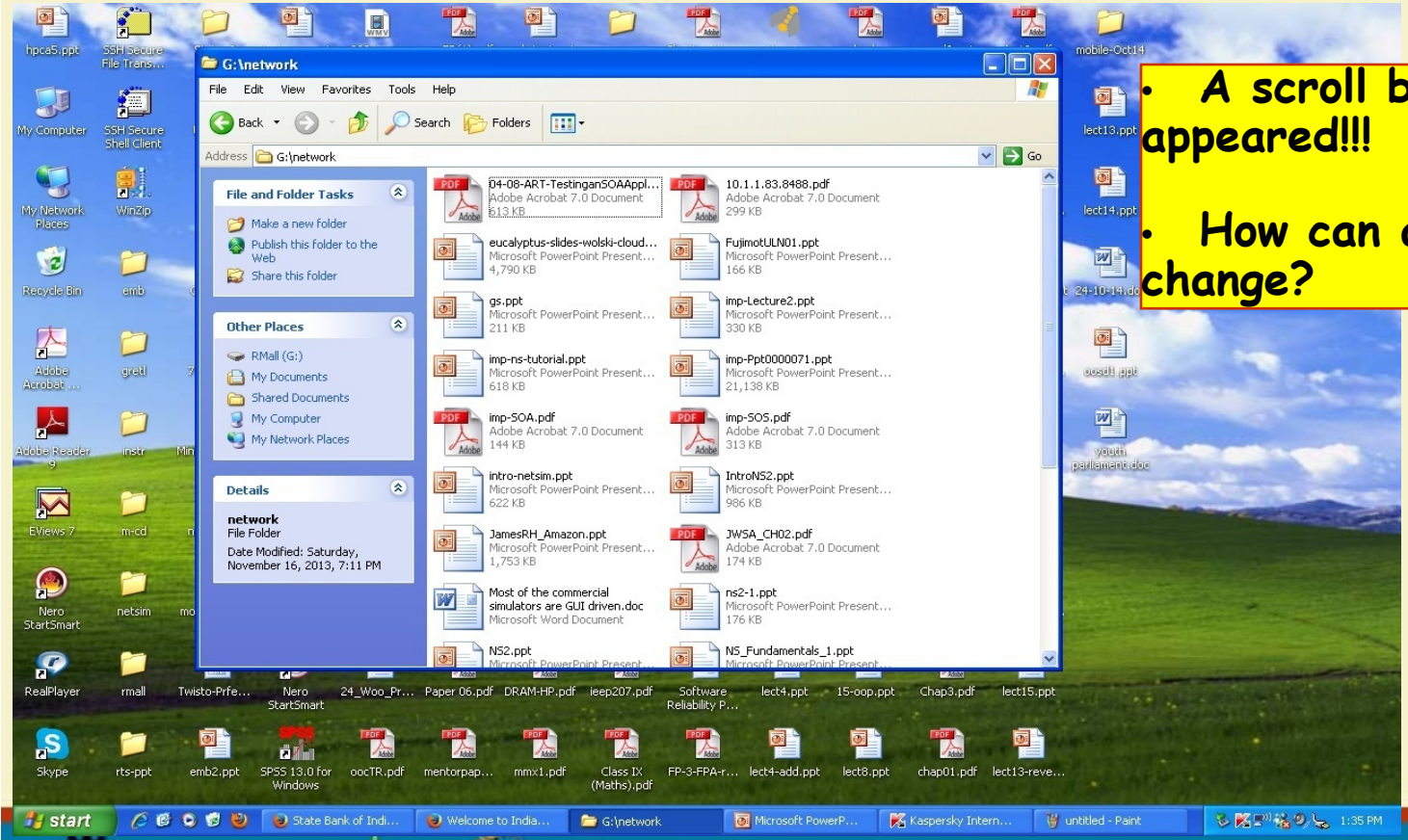
Decorator: In Simple Words

- **You have an object:**
  - You wrap it with another object.
  - They both support the same interface.
  - Later possibly wrap it with more objects..
  - **The ones on outside are "decorators"**
- **A Client uses the outermost decorator.**
  - A decorator passes on a method call to the object inside it, either as it is or with some enhancement.





You were working with a directory listing window… then you decided to resize it…



- A scroll bar has appeared!!!
- How can a window change?

Non-software example

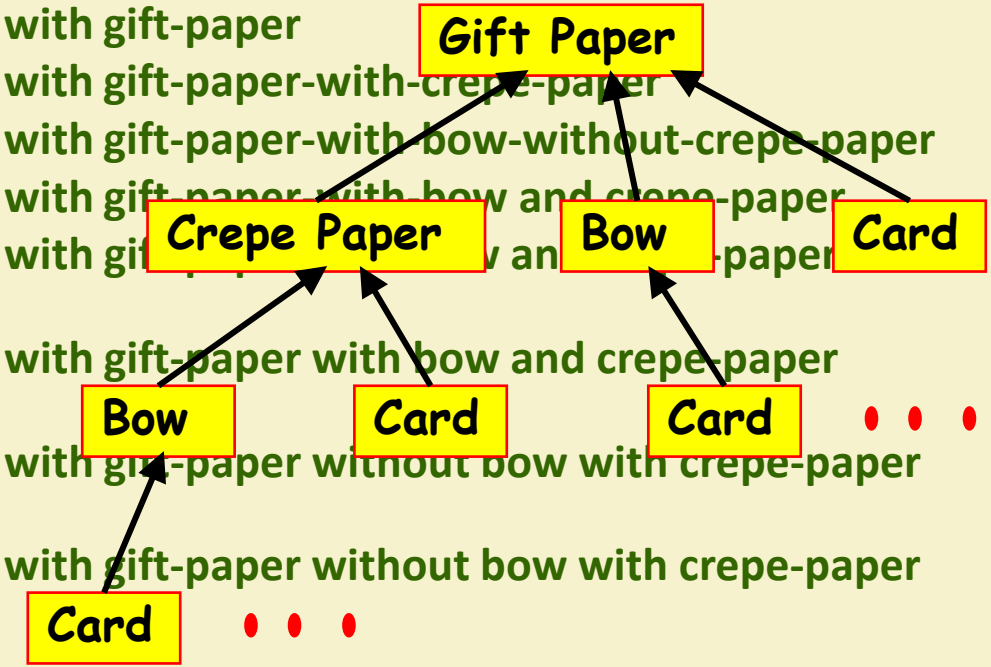
- Suppose you buy a gift:
  - First you select the gift…
  - Next you ask to wrap the gift appropriately…

- **A gift can be wrapped in several ways…**

Giftwrap Possibilities

They could provide various options for wrapping gift:

- box wrapped with gift-paper
- box wrapped with gift-paper-with-crepe-paper
- box wrapped with gift-paper-with-bow-without-crepe-paper
- box wrapped with gift-paper-with-bow and crepe-paper
- box wrapped with gift-paper-with-bow and crepe-paper card
- box wrapped with gift-paper with bow and crepe-paper without card
- box wrapped with gift-paper without bow with crepe-paper and card
- box wrapped with gift-paper without bow with crepe-paper without card
- Etc…



Solution

- In order to circumvent the problem, manufacturers sell the following materials:
  - **Boxes**
  - **Gift Paper**
  - **Cards**
  - **Bows**
  - **Crepe-paper**

You can select the ones you need and the order you want…

Decorator Pattern: Some Examples…

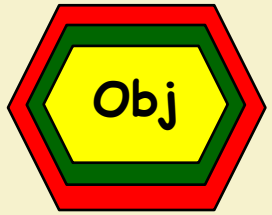
- Add borders or scrollbars to a GUI component
- Add headers and footers to an advertisement
- Add functionality to an input/output stream :
  - Such as reading inputs line by line or compressing a file before sending it over.

Decorator: General Concepts

- A Decorator adds responsibilities to individual objects (not to all objects of a class!) dynamically:
  - **In situations where a large number of independent extensions required…**
  - **An explosion of subclasses would occur if every combination to be supported.**
  - **Difficult to understand, remember and apply…**

Decorator: Review of the main Ideas

- A Decorator is an object that has an interface identical to an object that it contains.

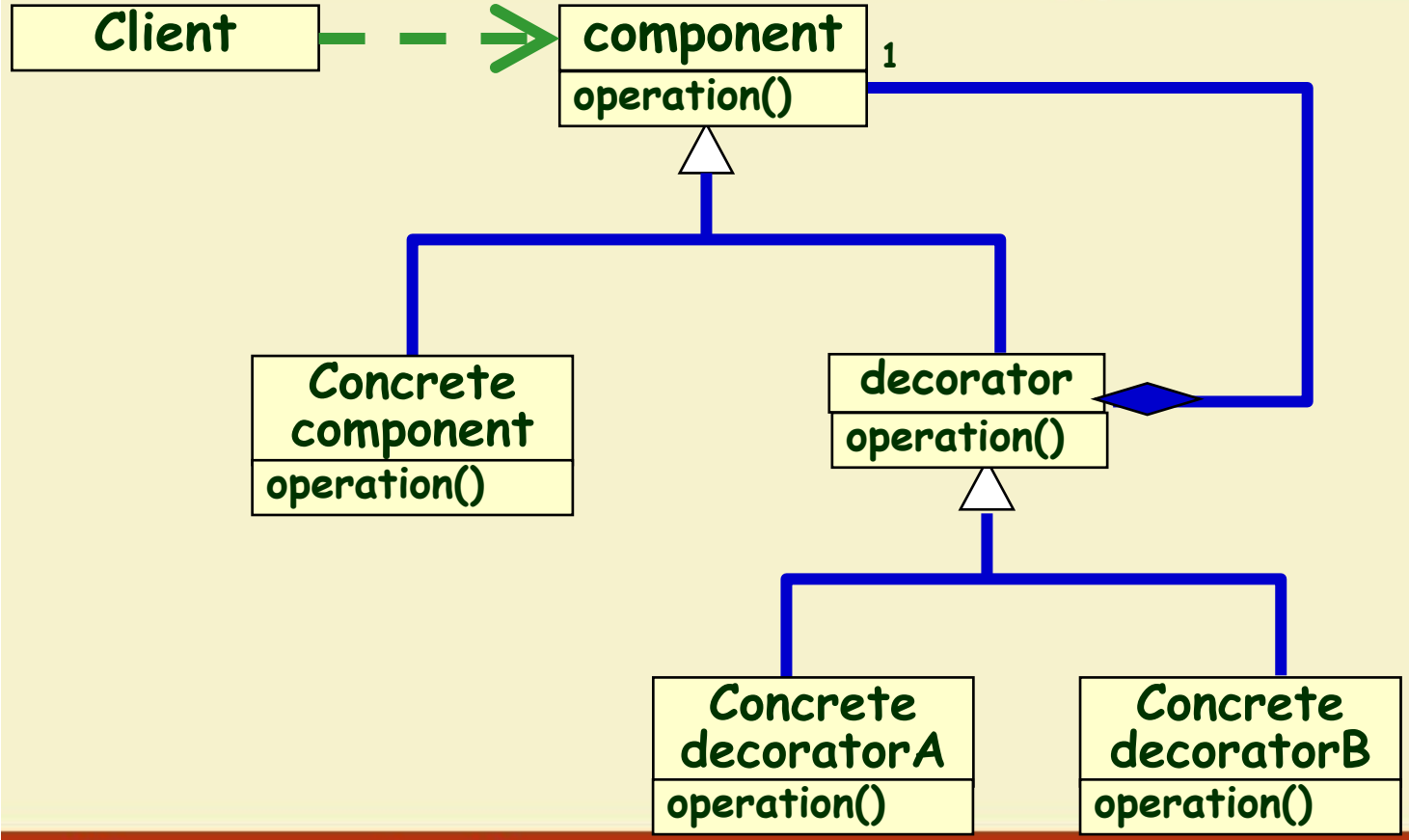


  - Used for adding additional functionality to a particular object at run time as opposed to a class of objects.
  - **Any call that a decorator gets, it relays to the object that it contains, and adds its own functionality along the way, either before or after the call.**

Decorator: Some Comments

- The Decorator pattern gives the designer flexibility:
  - Can change decorator at runtime, as opposed to a static change determined at compile time by subclassing.
- Since a Decorator has the same interface as the object it contains:
  - **The Decorator is indistinguishable from the object that it contains and from any other concrete instances, including other decorated objects.**
- This is a powerful technique and used to great advantage:
  - **Designers can recursively nest decorators without any other objects being able to tell the difference, allowing significant customization.**

Decorator Structure



An Example Application

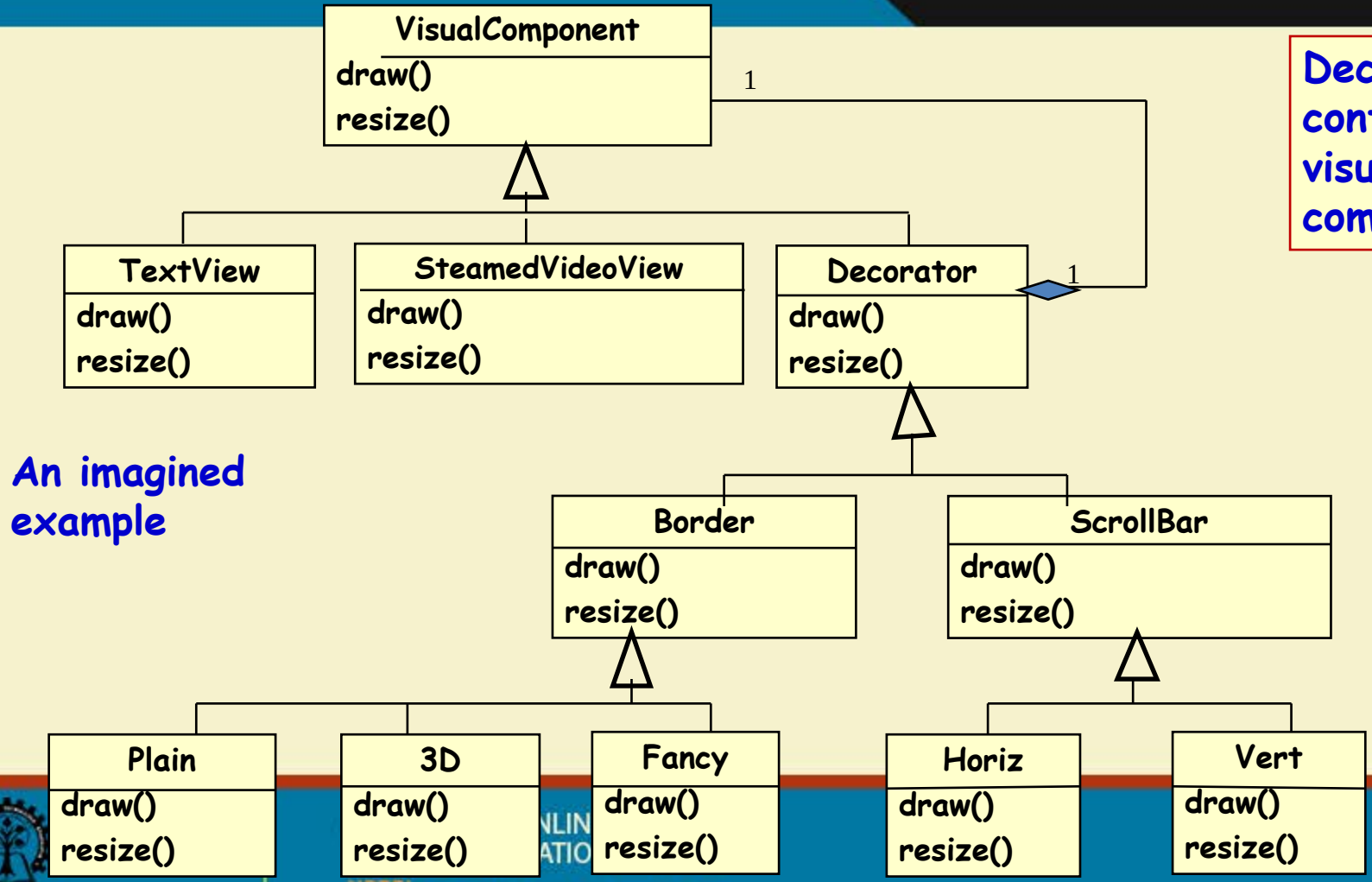
- Consider a TextView GUI component:
  - You want to add different kinds of borders and/or scrollbars to it.
- You can add 3 types of borders:
  - Plain, 3D, or Fancy
- and , 1, or 2 two scrollbars
  - Horizontal or Vertical
- **An inheritance solution would require 15 classes.**

1.TextView_Plain
2.TextView_Fancy
3.TextView_3D
4.TextView_Horizontal
5.TextView_Vertical
6.TextView_Horizontal_Vertical
7.TextView_Plain_Horizontal
8.TextView_Plain_Vertical
9.TextView_Plain_Horizontal_Vertical
10.TextView_3D_Horizontal
11.TextView_3D_Vertical
12.TextView_3D_Horizontal_Vertical
13.TextView_Fancy_Horizontal
14.TextView_Fancy_Vertical
15.TextView_Fancy_Horizontal_Vertical

Rather Large Number of classes!

Simpler Solution

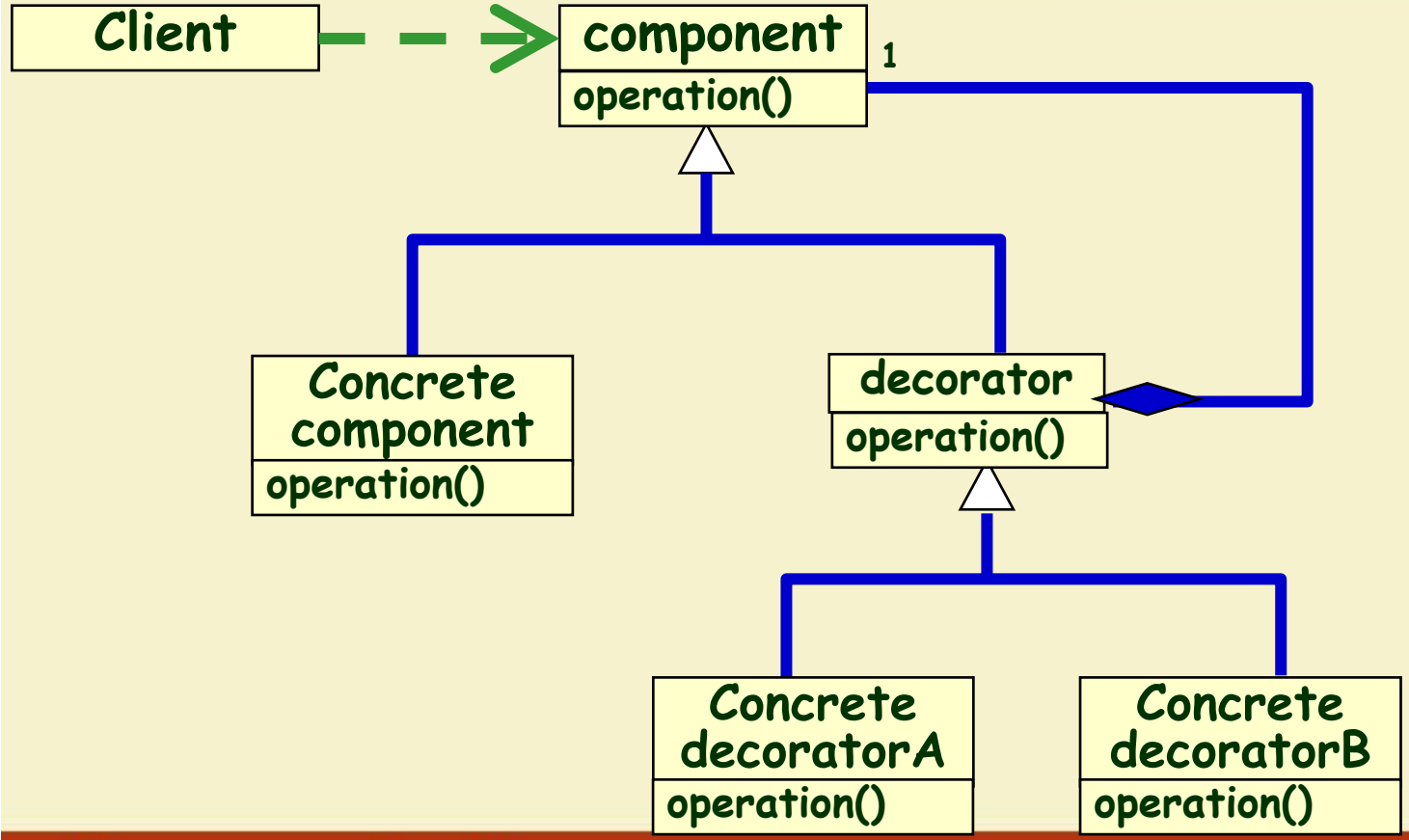
- **The component is contained in another object (decorator) that adds the border.**
- The decorator conforms to the interface of the component:
  - **So its presence is transparent to clients**
- The decorator forwards requests to the component:
  - May perform additional actions before or after forwarding.



Decorator contains a visual component

An imagined example

Decorator Structure



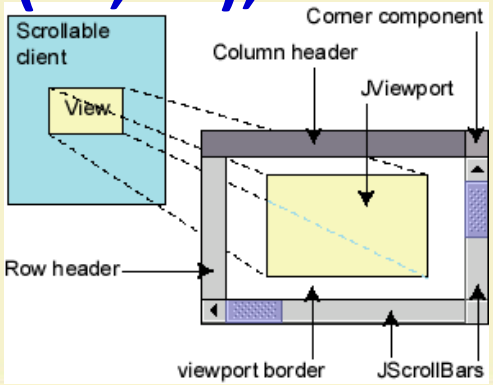
Decorator: Some Issues

- **Consequences:**
  - + Responsibilities can be added/removed at run-time
  - + Avoids subclass explosion
  - + Recursive nesting allows multiple responsibilities
- **Implementation Issues:**
  - – Interface conformance
  - – Should use a lightweight class as a Decorator
  - – Heavyweight classes make Strategy more attractive

Decorator example: GUI

- Normal Java GUI components don't have scroll bars
- JScrollPane is a container with scroll bars to which you can add any component to make it scrollable

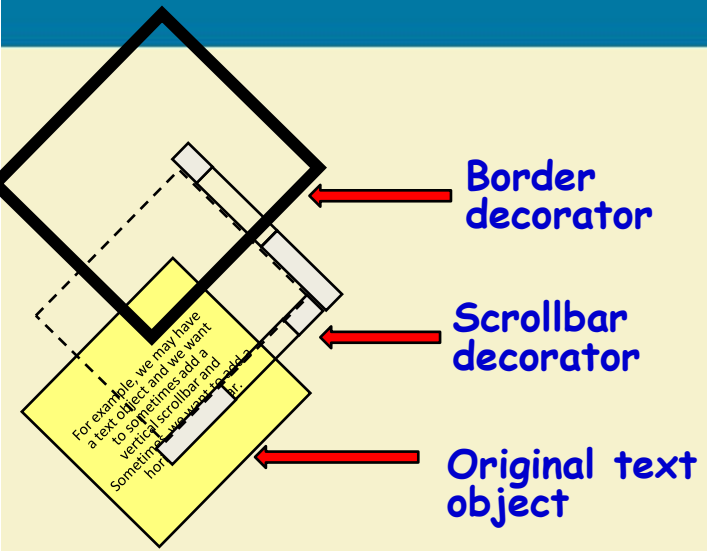
```
// JScrollPane decorates GUI components
JTextArea area = new JTextArea(20, 30);
JScrollPane scrollPane =
  new JScrollPane(area);
contentPane.add(scrollPane);
```

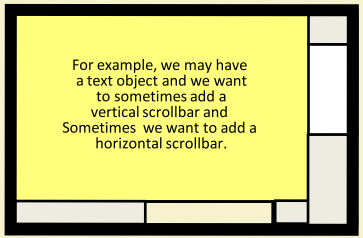


Java Borders

- Any JComponent can have 1 or more borders
- Borders are useful objects that, while not themselves components:
  - Know how to draw the edges of Java Swing components
- **Borders are useful not only for drawing lines and fancy edges:**
  - **But also for providing titles and empty space around components and also some basic behavior.**

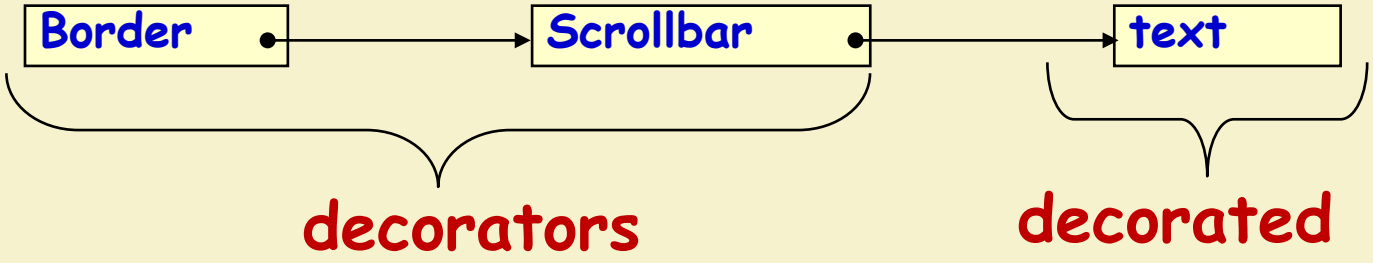
Decorator: Programming Example





- We have a text object and …
  - We want to add a border
  - At times we also want to add a scrollbar…

Decorator Terminology



- The objects refer each other like a linked list or chain of objects.

- The last in the list is the decorated object.

Widget and Stream Examples

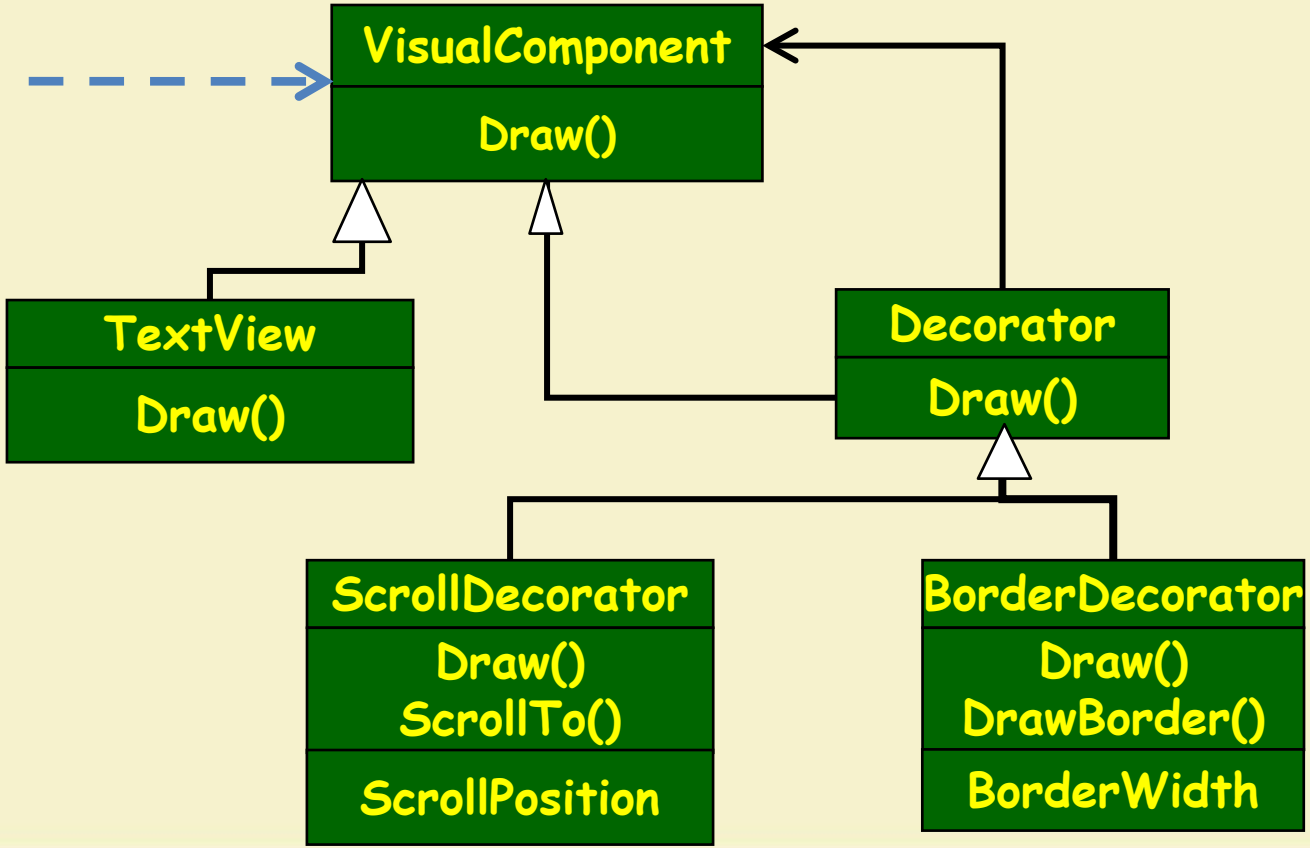
- Suppose you have designed a user interface toolkit and you wish to provide a choice of border or scrolling.
- **Widget aWidget = new BorderDecorator(new ScrollDecorator(new TextView()));**
  aWidget.draw();
- **Stream Example:**

Draw class diagram…

–cascade responsibilities to an output stream

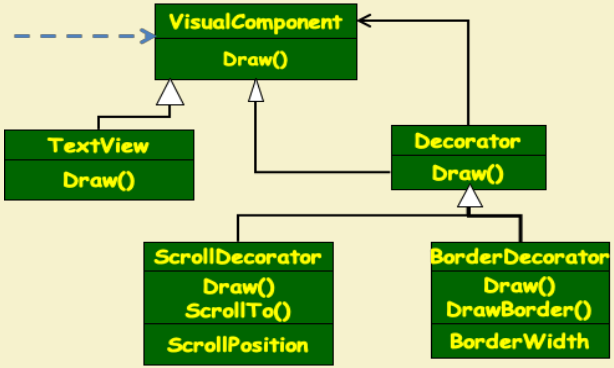
```
Stream aStream = new CompressingStream(
    new ASCII7Stream(new FileStream( "fileName.dat" )));
aStream.putString( "Hello world" );
```

Decorator Example Solution



Disadvantages of Inheritance

- Use of inheritance leads to an explosion of classes
- With another type of border added:
  - Many more classes would be needed
  - If another view were added such as StreamedVideoView:
  - It would double the number of Borders/Scrollbar classes
- **Use the Decorator Pattern instead!**



Decorator: Some Disadvantages

- When tempted to add many decorators:
  - A package may become hard to understand…
  - **Like Java I/O streams!!!**
- Solutions become complex:
  - A factory class may help

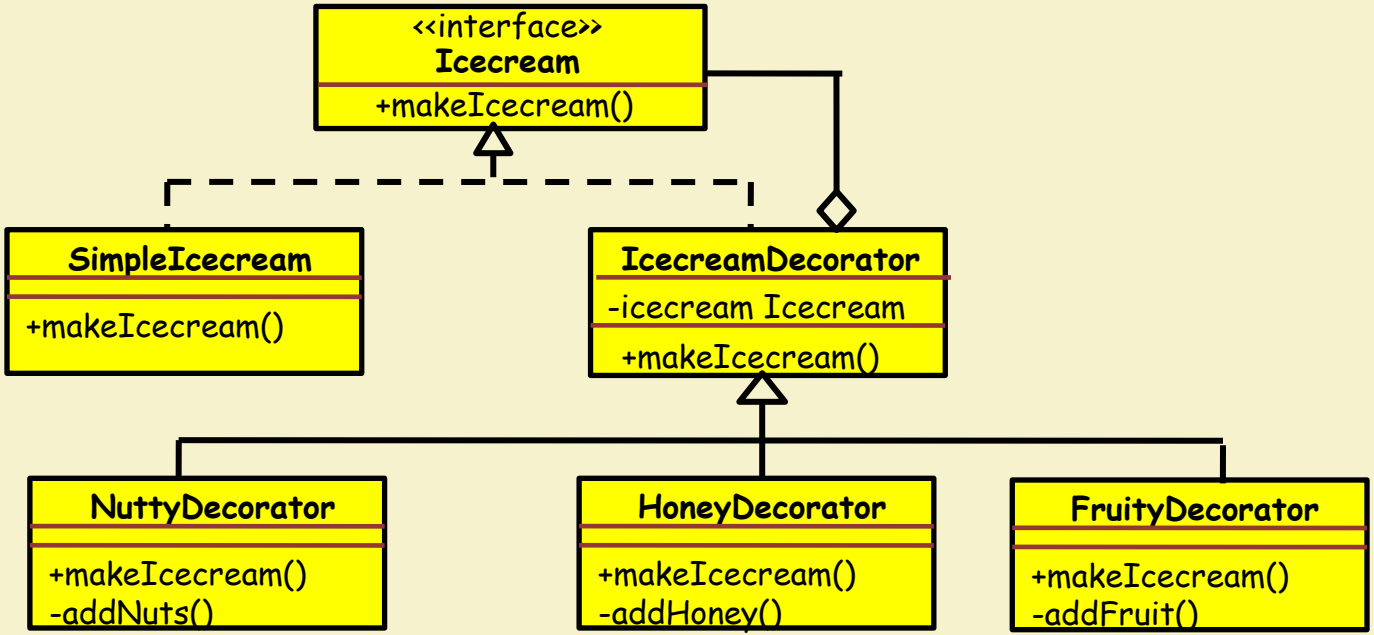
Quiz

- An ice cream can be made with the following types of toppings in any combination and order:
  - Nutty
  - Honey
  - Fruity
  - Chocolate
  - Vanilla

- **Apply decorator pattern**
- **Draw class diagram**
- **Write Java code**

1. **Draw class diagram**
2. **Write Java Code**

Quiz: Solution

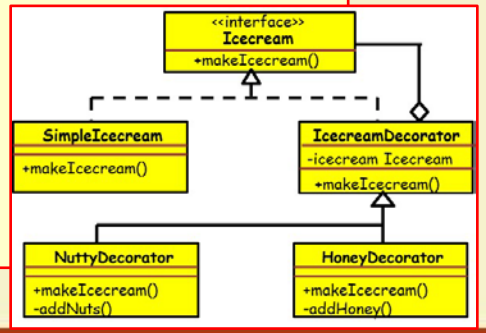


Quiz: Java Code

```
public interface Icecream {
  public String makeIcecream();
}
```

```
public class SimpleIcecream implements Icecream
{
public String makeIcecream() {
    return "Base Icecream";
  }
```

```
abstract class IcecreamDecorator implements Icecream {
  protected Icecream specialIcecream;
  public IcecreamDecorator(Icecream specialIcecream) {
    this.specialIcecream = specialIcecream;}
  public String makeIcecream() {
    return specialIcecream.makeIcecream();}
}
```



```
public class NuttyDecorator extends IcecreamDecorator {
  public NuttyDecorator(Icecream specialIcecream) {
    super(specialIcecream);  }
  public String makeIcecream() {
    return specialIcecream.makeIcecream() + addNuts();  }
  private String addNuts() {
    return " + cruncy nuts";}
}
```

```
public class HoneyDecorator extends IcecreamDecorator {
            public HoneyDecorator(Icecream specialIcecream) {
                        super(specialIcecream);  }
            public String makeIcecream() {
                        return specialIcecream.makeIcecream() + addHoney();  }
            private String addHoney() {
                        return " + sweet honey";}
  }
```

Decorator: Pros

- **More flexible than static inheritance**:
  - Responsibilities can be added and removed at run-time
  - Decorators also make it easy to add a property twice. For example, to give a TextView a double border, simply attach two BorderDecorators. Inheriting from a Border class twice is error.
- **Avoids inheriting from feature-laden classes.**
  - An application needn't pay for features it doesn't use.
  - It's also easy to define new kinds of Decorators independently from the classes of objects they extend.

Decorator: Cons

- **Lots of little objects:**
  - The objects differ only in the way they are interconnected, not in their class or in the value of their variables.
  - Although these are easy to customize by those who understand them, **they can be hard to learn and debug.**

Composition vs. Inheritance

- **Both are ways to re-use functionality**
- **Inheritance:**
  - Re-use functionality of parent class
  - Statically decided
  - Weakens encapsulation
- **Composition:**
  - Re-use functionality of objects at run-time
  - Invoked through the interface
  - Dynamic: multiple types with same interface
  - Black-box re-use

Composition vs. inheritance

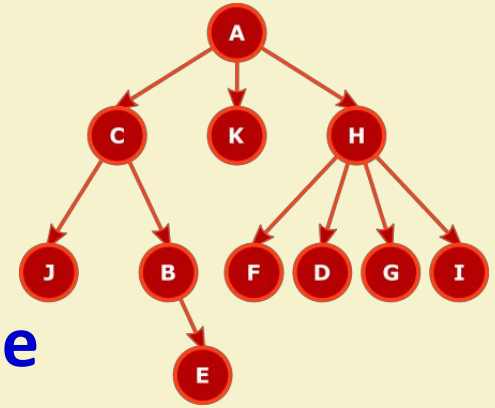
- Inheritance is a quick way to design new components:
  - That are variants of existing ones
- Over-use of inheritance creates bloated hierarchies
  - Code is more difficult to maintain
  - **Unnecessary baggage for many classes**
- **Composition drawback:** it is harder to understand the behavior of a program by looking only at its source code
  - **Semantics of interaction are decided at run-time**

Iterator Pattern

Background

- Traversing through elements in an aggregation is a very common problem in programming:
  - Iterators provide a uniform way of doing this …
- Using an iterator, we don't need to know:
  - **How the collection is implemented!**
  - **Concurrent access and synchronization are supported**
  - **Other class methods may be accessing the iterator**

Iterator Pattern: Intent



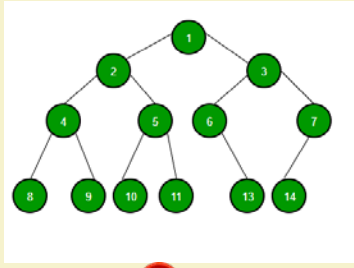
- Provide a way to access the elements of an aggregate object sequentially:
  - **Without exposing the programmer to the complexity of underlying representation.**
- Move the responsibility for access and traversal:
  - **From the aggregate object to an iterator object.**

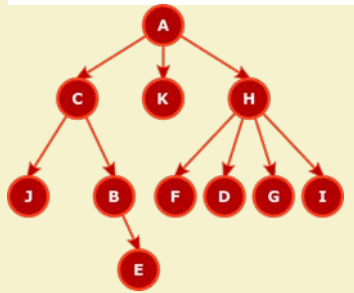
Iterator Pattern

- Helps access an aggregate's contents sequentially:
  - In a manner that is independent of the implementation of the aggregate

- **Supports the same interface for all aggregates.**
- **Supports multiple types of traversals of aggregates**
- **Supports multiple iterators**

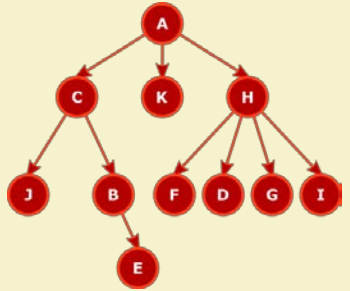
Iterator: Essential Idea





- The elements of a collection:
  - **Accessed in some sequential order that may be independent of the specific collection.**
- For example:
  - Level order: Labelling each object of a tree from left to right
  - Inorder, post order, preorder, etc

The Iterator Pattern: Context



- It might be necessary to have more than one type of traversal on the same aggregate object.
  - **Also, not all types of traversals can be anticipated *a priori*.**
- **One should not bloat the interface of the aggregate object with all possible traversals.**

Iterator: Basic Methods

- **reset:**
  - Move to the beginning of the range of elements
- **next:**
  - Returns the next element.
- **hasNext:**
  - Interrogate it to see if it is at the end of the range

Additional Methods

- **remove:** Current element is removed, as in Java List.
- **add:** Add an element at current position
- **count:** Return the number of elements in the aggregate.
- **first:** Reset

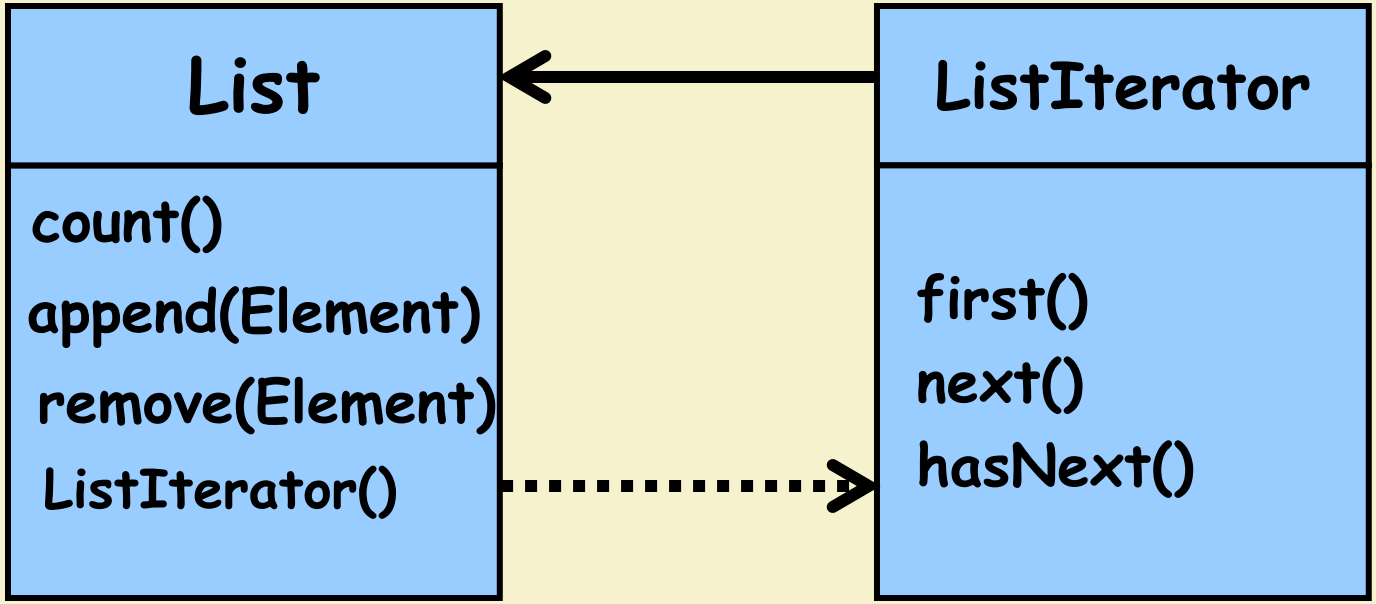
Iterator Object Methods

- Two methods most common:
  - **public boolean hasNext()**
    - Returns true if the aggregate has more elements (objects) that have not yet been returned by next()
  - **public Object next()**
    - Returns the next element (object).

next() in Java: Some Issues

- Theoretically, when an iterator returns an element of the collection:
  - **It might return a clone of the element, or it might return a reference to the element**
- Iterators for Java collection classes **ArrayList<T> and HashSet<T>**:
  - **Actually return references**
- Therefore, modifying the returned value will modify the element in the collection

Iterator Pattern: Main Idea

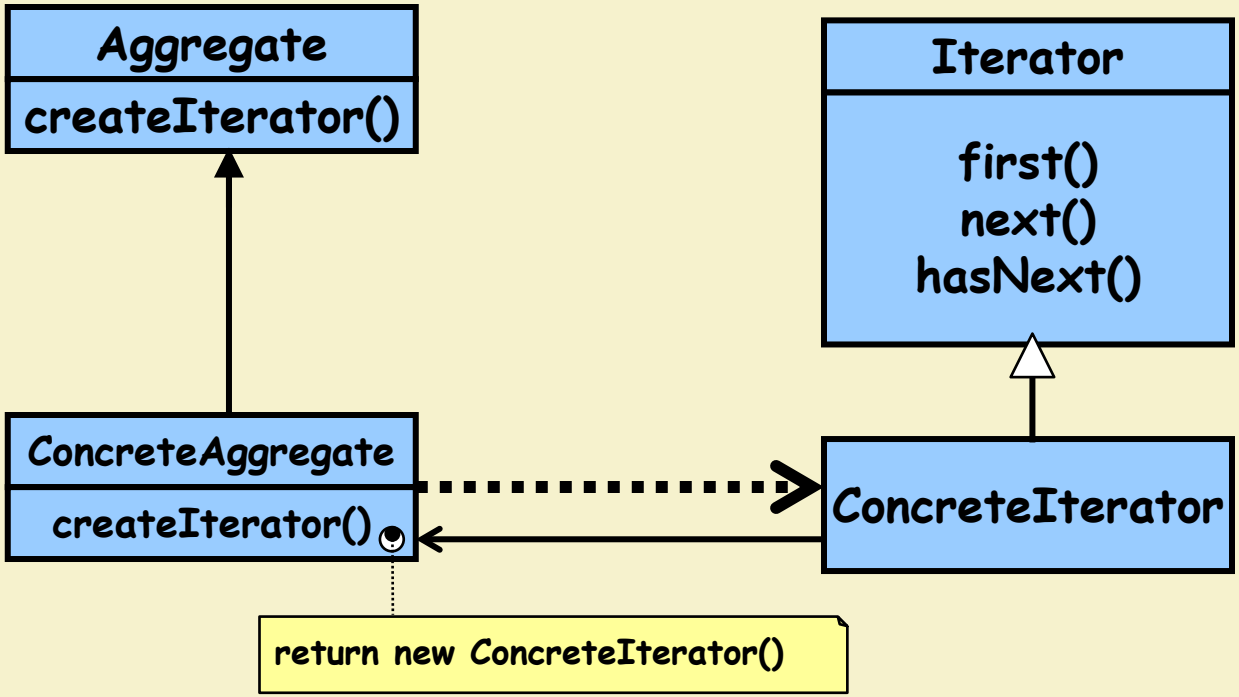


...

Iterator Pattern: Solution

- The responsibility for accessing and traversing an aggregation object:
  - **Placed in a separate iterator object and not in the aggregate object**
- An iterator object should:
  - **Provide the same interface for accessing and traversing elements, regardless of the class of the aggregate object and the kind of traversal performed…**

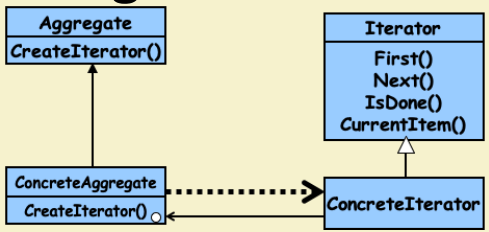
Structure of Iterator Pattern



A ConcreteIterator object holds a reference to the ConcreteAggregate object that created it.

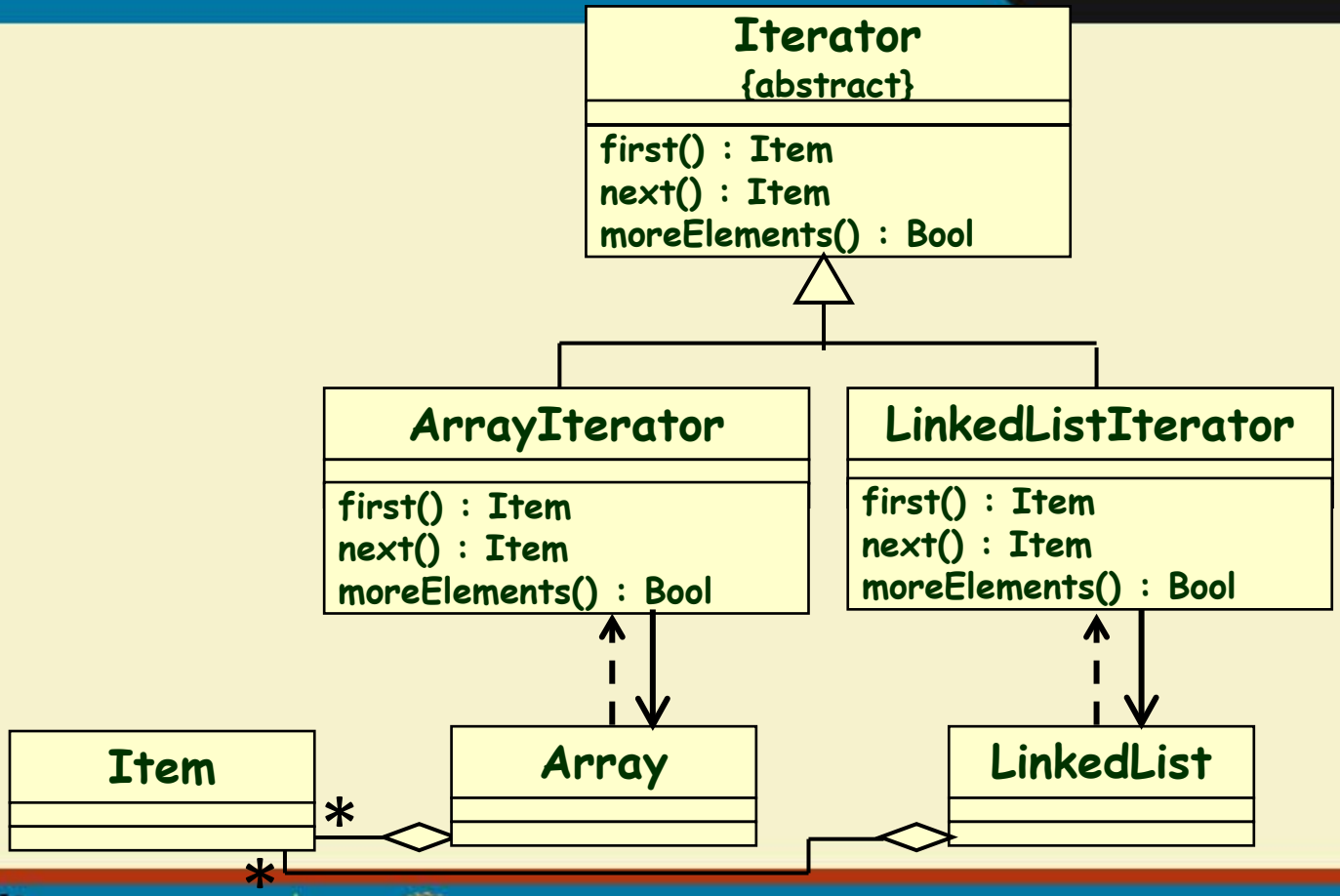
Iterator Participants

- **Iterator**
  - Defines an interface for accessing and traversing elements
- **ConcreteIterator**
  - Implements the Iterator interface.
  - Keeps track of the current position in traversal of the aggregate



- **Aggregate**
  - Defines an interface for creating an Iterator object
- **ConcreteAggregate**
  - Creates and returns an instance of ConcreteIterator

Iterator: Example 1



Iterator: Example 2

```
public class ArrayIterator
    implements Iterator {

    private String[] data;

    private int index;

    public ArrayIterator
    (String[] data) {

            this.data = data;

            this.index = 0;

    }
```

```
    public String first() {
            index = 0;
            return data[0];
    }
    public String next() {
            return data[index++];
    }
    public boolean moreElements() {
            if (index >= data.length)
                    return false;
            return true;
    }
}
```

Iterator: Example 1

```
public class LinkedListIterator
 implements Iterator {
  private LinkedListElement
  first;
  private LinkedListElement
  current;
  public LinkedListIterator
  (LinkedListElement first){
        this.first = first;
        this.current = first;
  }
  public String first() {
        return first.value();
  }
  public String next() {
        current = current.getNext();
        return current.value();
  }
  public boolean moreElements() {
        if (current.getNext() == null)
                return false;
        return true;
  }
}
```

Iterating Through A Vector: without Iterator

```
import java.util.*;

public class IterationExample1{

  public static void main(String args[])   {

     Vector v = new Vector();

     // Put some Complex number objects in the vector.

     v.addElement(new Complex(3.2, 1.7));
```

Iterating Through A Vector

```
    v.addElement(new Complex(7.6));
    v.addElement(new Complex());
    v.addElement(new Complex(-4.9, 8.3));
    for (int i = 0; i < v.size(); i++) {
      Complex c = (Complex)v.elementAt(i);
      System.out.println(c);}
  }
}
```

Iterating Through A Vector Without Iterator

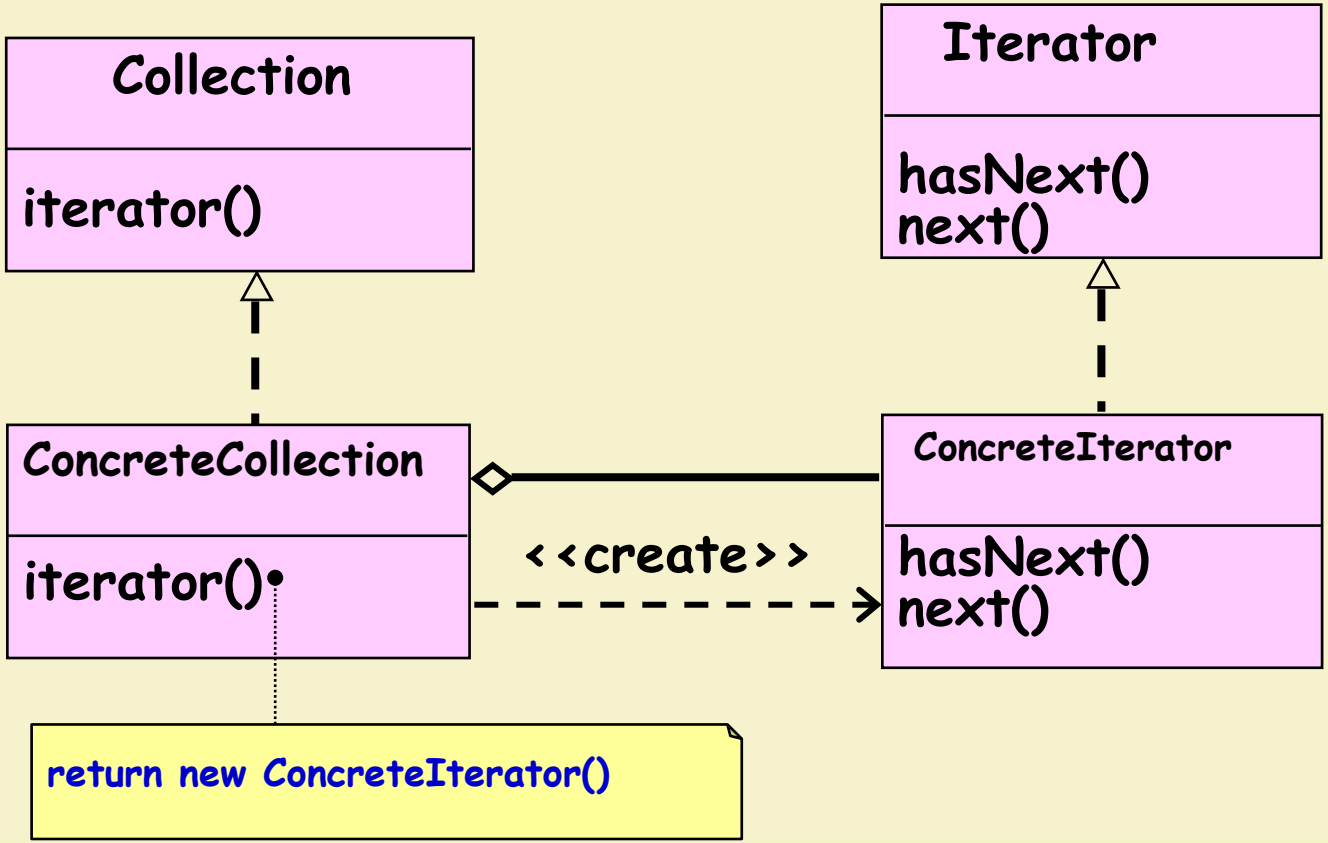
- For vectors this approach does not appear too bad:
  - Because the elements of a vector can be retrieved by specifying their position:

v.elementAt(i);

Iterating Through a Vector: Using Iterator

```
Iterator  i = v.iterator();
 while (i.hasNext()) {
     Complex c= (Complex)i.next();
     System.out.println(c);
 }
```

Iterator for Java Collection Class



Iterator Pattern in Java

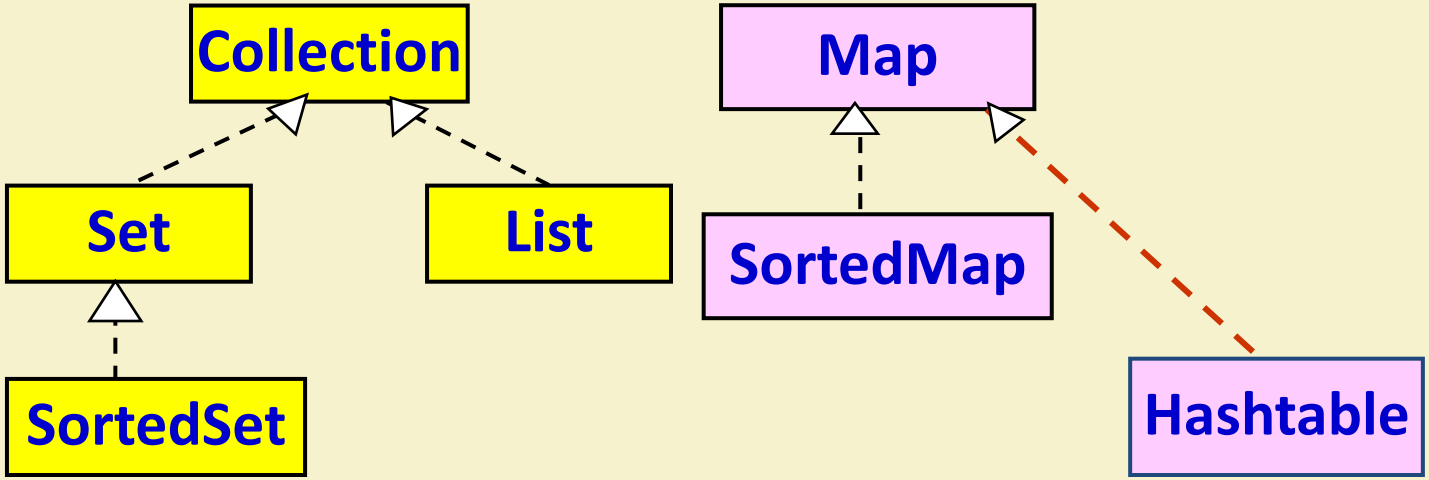
- Support for the iterator pattern provided in the Java SDK is based on Java interfaces:
  - Not abstract classes (why?)

```
public interface Iterator {
    public boolean hasNext();
    public Object next();
    public void remove();
}
```

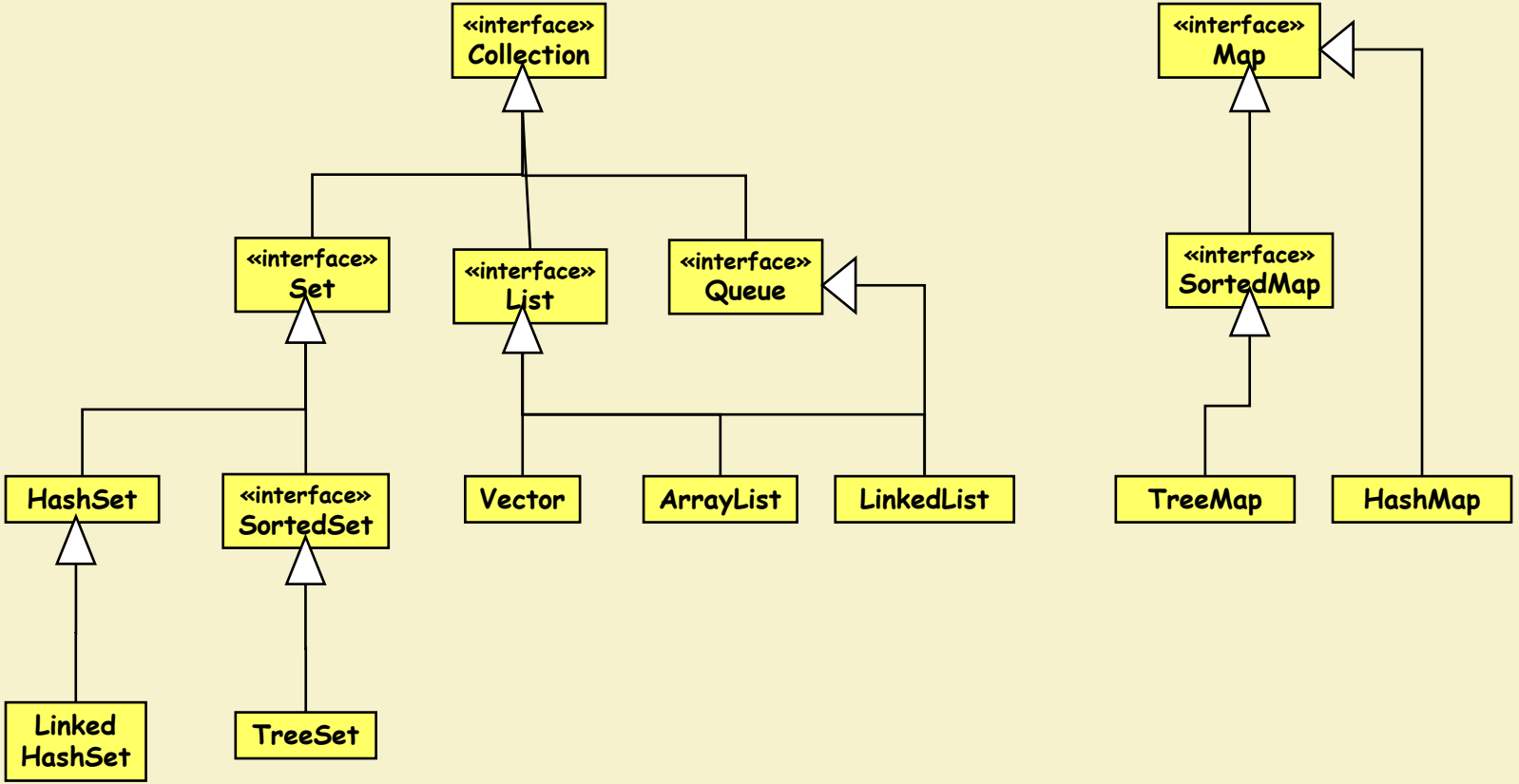
Java Iterator

- An Iterator object is an instance of a class that implements the Iterator interface:
  - **Java Iterator interface defines the hasNext(), next(), and remove() methods**

Java Collections



- Hashtable is an old (pre-Collections) class
- Hashtable has been retrofitted to implement the Map interface


«interface»
Collection
«interface»
Set
«interface»
List
«interface»
Queue
HashSet
«interface»
SortedSet
Vector
ArrayList
LinkedList
Linked
HashSet
TreeSet
«interface»
Map
«interface»
SortedMap
TreeMap
HashMap

Iterating Through a Hashtable

- **Hashtable does not implement Collection:**
  - Therefore does not have methods that return Iterator objects for the keys and values in the table
- Instead:
  - **entrySet()** returns a Set of the entries in the table
  - **keySet()** returns a Set of the keys in the table
  - **values()** returns a Collection of the values in the table
- iterator() messages are sent to these objects
- See the Java 1.2 API Specification for more information

Java Iterator

```
public interface java.util.Collection
  ...   // List, Set extend Collection
    public Iterator iterator();
}
public interface java.util.Map {
  ...
   public Set keySet();
   public Collection values();
}
```

```
public interface java.util.Iterator {
   public boolean hasNext();
   public Object next();
   public void remove();
}
```

Iterators in Java

- All Java collections have a method **iterator** that returns an iterator for the collection…

```
List list = new ArrayList();
... add some elements ...

for (Iterator itr = list.iterator(); itr.hasNext()) {
  BankAccount ba = (BankAccount)itr.next();
  System.out.println(ba);
}
```

Adding your own Iterators

- When implementing your own collections, it can be very convenient to use Iterators
  - **discouraged (has nonstandard interface):**

```
public class PlayerList {
    public int getNumPlayers() { ... }
    public boolean empty() { ... }
    public Player getPlayer(int n) { ... }
}
```

  - **preferred:**

```
public class PlayerList {
        public Iterator iterator() { ... }
        public int size() { ... }
        public boolean isEmpty() { ... }
}
```

Implementing an Iterator

- Three main approaches to create an Iterator for an aggregate class A:
  - **Use a separate class**
  - **Use an inner class within A**
  - **Use an anonymous inner class within A**

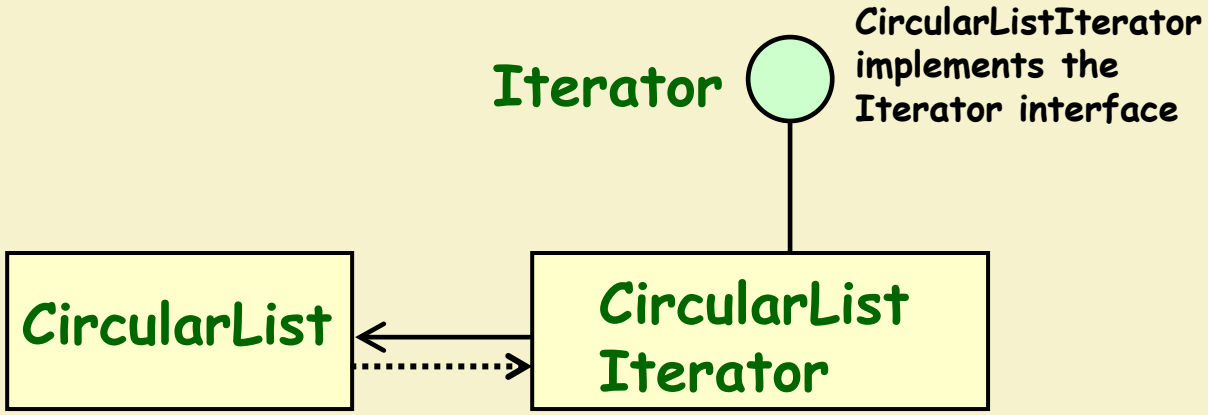
Exercise: Special Iterator for CircularList

```
public CircularList(); //constructor

// Count of number of elements in the list.
public int count();

// Return the element stored at the specified position in the CircularList.
public Object get(int index);

// Remove the element stored at the specified position in the CircularList.
public Object remove(int index);
```

```
CircularList l = new CircularList();
...
//store integers in the list
...
Iterator i = l.iterator();
while (i.hasNext()) {
    System.out.println(i.next());
}
```

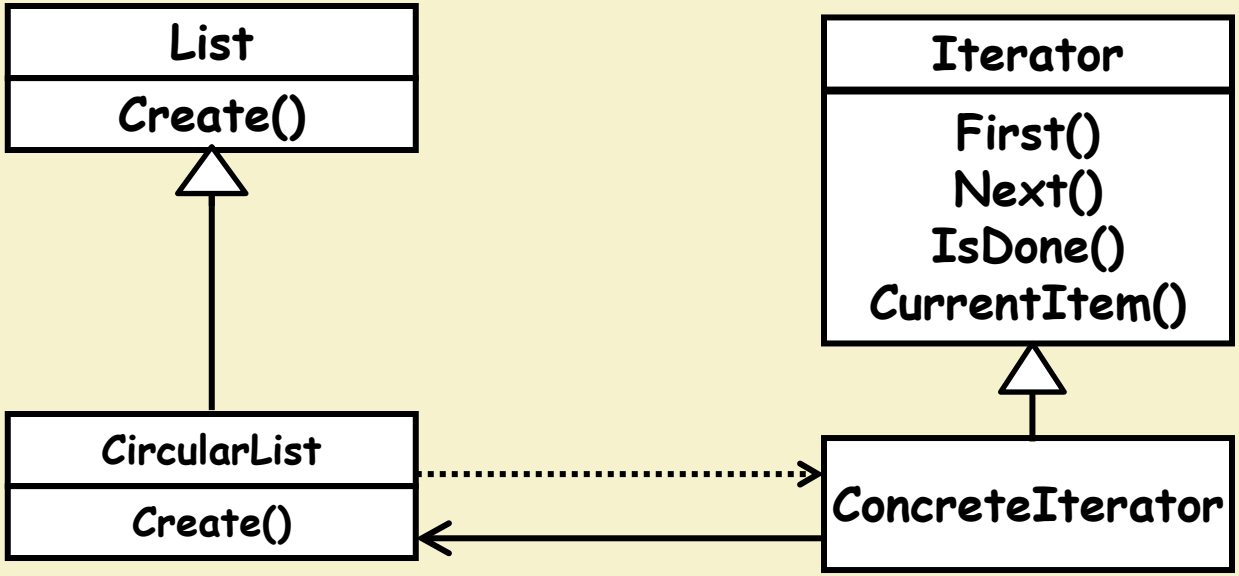
Exercise: Build an Iterator as a Separate Class…

Class Diagram: CircularList Iterator



A CircularListIterator object holds a reference to the CircularList object that created it.

Solution: Class Structure



A ConcreteIterator object holds a reference to the CircularList object that created it.

```
class CircularListIterator implements Iterator {

    public boolean hasNext()    {}

    public Object next()    {}

    public void remove()   {}

}
```

Solution: Class CircularListIterator

Class CircularListIterator: Code Details

```
class CircularListIterator implements Iterator{
    private CircularList myList;
    public CircularListIterator(CircularList theList) {
        myList = theList;  }
     public Object next() {}
     public boolean hasNext() {}
     public void remove() {}
}
```

```
class CircularListIterator implements Iterator {
   private CircularList myList;
 // constructor not shown
   private int pos = 0;
   public Object next()  {
      Object o = myList.get(pos);
      pos++;
      return o;
   }
}
```

Class CircularListIterator

Class CircularListIterator

```
public boolean hasNext() {
   return(pos < myList.count());   }

public void remove()   {
   if (pos > 0) {
      myList.remove(pos-1);
// This method should throw an
   // IllegalStateException if next() has not
// yet been invoked, or if remove() has
// already been invoked after the last
// invocation of next().
}
```

Modifying Class CircularList

- All that remains now:
  - Define a method in CircularList to create and return a CircularListIterator object
- In order to be consistent with other Java collection classes:
  - We'll call the method iterator()

Modifying Class CircularList

```
class CircularList{
                …   …   …
    public Iterator iterator()   {
            return new CircularListIterator(this);
    }
}
```

Inner Iterator Class

- Java allows one class to be nested inside another
- Inner class has full access to private data of main class
  - data is still kept as a private field

```
class LinkedList {
    private Node head;
    // put all the usual stuff here

    public class LLIterator implements
    Iterator {
        private Node nextObject;
        public LLIterator() {
            nextObject = head;
        }
    }
    public Iterator iterator() {
        return new  LLIterator();
    }
}
```

Interface Polymorphism

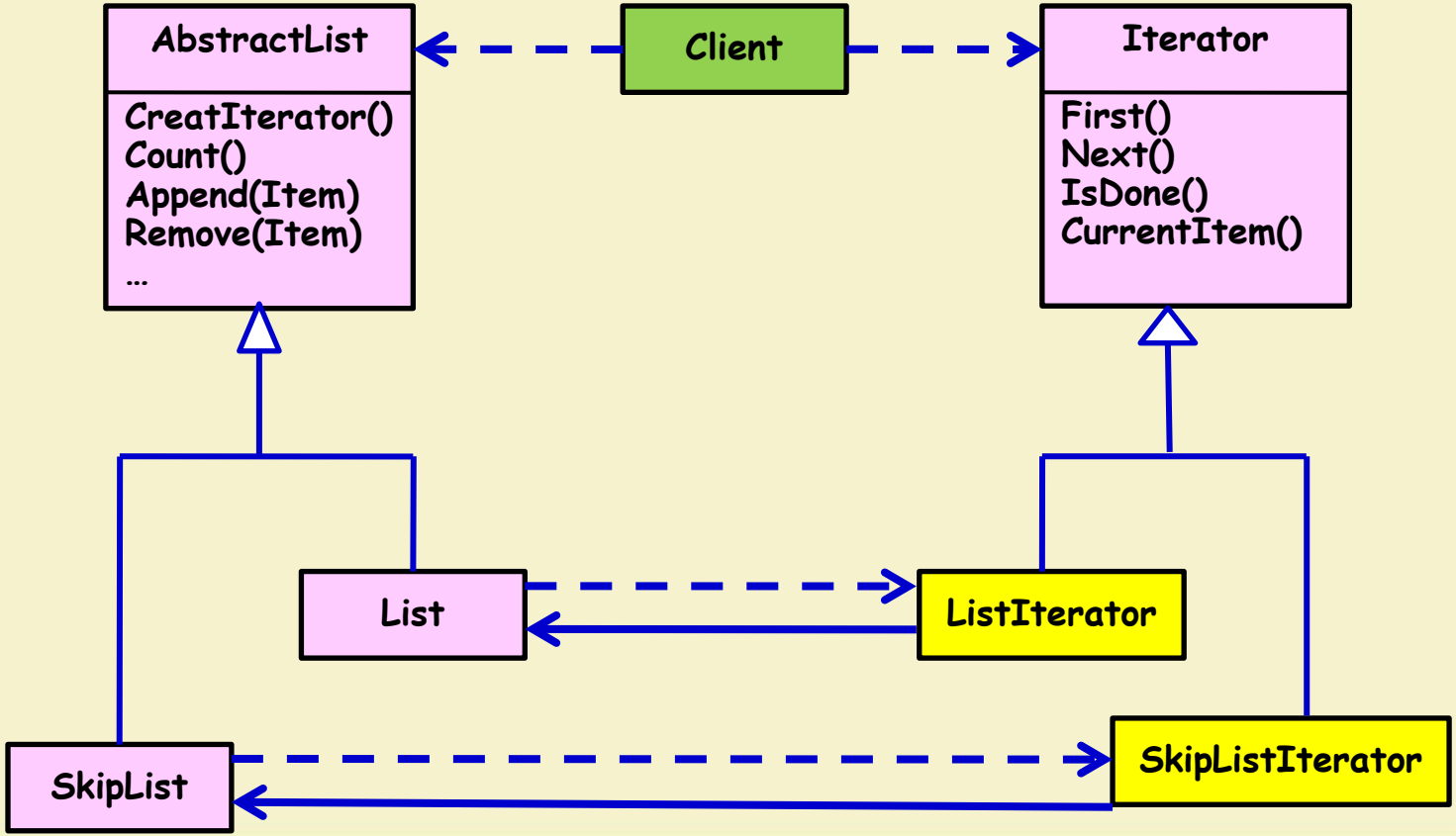
- Notice that CircularList's iterator() returns a reference to a CircularListIterator object:
  - but the method's return type is Iterator
  - Q: Why is this o.k?

```
class CircularList{
   public Iterator iterator()   {
   return new CircularListIterator(this);
   }
}
```

- Ans: CircularListIterator implements the Iterator interface
  - i.e., a CircularListIterator object is-a(n) Iterator object
- In other words, is-a relationships exist between
  - a subclass and a superclass
  - an interface and the class that implements the interface

Exercise: Polymorphic Iterators

- Assume that we have an AbstractList class.
- Based on this, we have defineds List and SkipList subclasses.
- **Design Iterators for these different implementations of the AbstractList.**


AbstractList
CreatIterator()
Count()
Append(Item)
Remove(Item)
...
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
List
ListIterator
SkipList
SkipListIterator

```
List list = new List();

SkipList skipList = new
    SkipList();

Iterator listIterator =
    list.CreateIterator();

Iterator skipListIterator =
    skipList.CreateIterator();

handleList(listIterator);

handleList(skipListIterator);

...
```

```
public void handleList(Iterator
    iterator) {
  iterator.First();
  while (!iterator.IsDone()) {
  Object item =
    iterator.CurrentItem();
// Code here to process item.
  iterator.Next();
  }
}
```

A Few Issues

- Who controls the iteration, e.g. while searching an item?
- **External:**
  - User explicitly calls all necessary operations.
- **Internal:**
  - Client asks the internal iterator an operation to perform, and the iterator applies that operation to every element.

Pattern Variations

- **Who defines the traversal algorithm?**
  - The iterator is not the only place where the traversal algorithm can be defined.
  - **The aggregate might define the traversal algorithm and use the iterator to store just the state of the iteration.**
  - **This is called the cursor approach**
- A client will invoke the Next operation on the aggregate with the cursor as an argument:
  - The Next operation will change the state of the cursor.

Multiple Traversal Algorithms

- Complex aggregates may be traversed in many ways.
  - For example, code generation and semantic checking involve traversing parse trees.
  - Code generation may traverse the parse tree inorder or preorder.
  - Iterators make it easy to change the traversal algorithm: **Just replace the iterator instance with a different one**

Traversal Issues (cont)

- If the iterator is responsible for the traversal algorithm:
  - **It becomes easy to use different iteration algorithms on the same aggregate**
  - **It is easy to reuse the same algorithm on different aggregates.**
- But, the traversal algorithm might need to access private data of the aggregate.:
  - Putting the traversal algorithm in the iterator violates the encapsulation of the aggregate.

Implementation Issues

- **How robust is the iterator?**
  - If elements are added or deleted, it can become dangerous to modify an aggregate while you're traversing it.
  - Why not just make a copy?
  - **A robust iterator ensures that insertions and removals won't interfere with traversal, and it does it without copying the aggregate.**

Implementation Issues

- There are many ways to implement robust iterators.
  - **Most rely on registering the iterator with the aggregate.**
- On insertion or removal:
  - **The aggregate either adjusts the internal state of iterators it has produced,**
  - Or it maintains information internally to ensure proper traversal.

Iterator: Final Analysis

- Programmer gets a consistent interface to traverse any collection
- It is easy to have multiple iterators traversing the same collection
- **Flexibility:** An iterator can give access to a filtered view of the items in a collection