Abstract classes

CSE 142, Summer 2002

Computer Programming 1

http://www.cs.washington.edu/education/courses/142/02su/

Readings and References

- Reading
  - » Chapter 15, *An Introduction to Programming and Object Oriented Design using Java*, by Niño and Hosch
- Other References
  - » Sections *Object-Oriented Programming Concepts* and *Classes and Inheritance* of the Java tutorial
  - » http://java.sun.com/docs/books/tutorial/java/

Good design characteristics

- Design strategy
  - » Figure out the fundamental elements of a problem
  - » Design your solution to model those key elements
- A good design will be
  - » robust - it doesn't need major changes to adapt to small changes in the problem statement
  - » long-lived - it can be adapted easily over time, and so it lives beyond the initial problem itself

Abstraction is the key to good design

- What are the fundamental elements of a problem?
- Those aspects of the problem that appear over and over in different problem statements
  - » A scheduling problem
    - what are the fundamental things being scheduled?
    - what are the common state and behavior of scheduled things?
  - » An inventory problem
    - what are the elements actually being tracked?
    - what are the common state and behavior of inventoried items?

Inheritance

- Inheritance gives us a way to let a superclass (or base class) implement state and behavior that is common to a group of subclasses
- The subclasses differ in some way from the superclass and from each other, and yet they share some characteristics
- So we have the notion of common or shared characteristics and unique or non-shared characteristics

Interfaces

- An interface is a tool for defining the behavior that all implementing classes will have
  - » it names the methods that a class must have if the class claims to implement the interface
  - » the interface definition is a good tool for identifying what must be implemented
  - » the interface does not provide the programmer with any help in actually implementing the methods!

The Shape interface

- Here are the methods in the Shape interface

```
void addTo(GWindow gw)
Rectangle getBoundingBox()
int getCenterX()
int getCenterY()
java.awt.Color getColor()
int getHeight()
int getWidth()
int getX()
int getY()
```

```
InternalGWindow currentWindow()
boolean intersects(Shape other)
void moveBy(int deltaX, int deltaY)
void moveTo(int x, int y)
void paint(java.awt.Graphics g)
void recordWindow(InternalGWindow gw)
void removeFromWindow()
void rotateAround(int pX,int pY,double d)
void setColor(java.awt.Color c)
```

Do we have to start from scratch when we want to implement a new class like Triangle that implements Shape?

Recall the syntax of inheritance

- Specify inheritance relationship using **`extends`**

```
public class Triangle extends PolyShape {
```

```
public abstract class PolyShape extends ShapeImpl  {
        private int npoints;
```

```
public abstract class ShapeImpl implements Shape {
        protected Rectangle boundingBox;
        …
        public int getX() {
                return boundingBox.getX();
        }
}
```

Constructor Summary

Triangle()
Create a new blue, filled triangle with default position and size.

Triangle(int x1, int y1, int x2, int y2, int x3, int y3)
Create a new black, unfilled triangle between the given three vertices

Triangle(int x1, int y1, int x2, int y2, int x3, int y3, java.awt.Color c, boolean fill)
Create a new triangle of the given color and filledness between the given three vertices

Method Summary

java.lang.String	**toString**() Answer the printed representation of this shape.

Methods inherited from class uwcse.graphics.PolyShape

addPoint, moveTo, paint, resize, rotateAround

Methods inherited from class uwcse.graphics.ShapeImpl

addTo, currentWindow, getBoundingBox, getCenterX, getCenterY, getColor, getHeight, getWidth, getX, getY, intersects, moveBy, recordWindow, removeFromWindow, setColor

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, wait, wait, wait

The List interface

```
void add(int index, Object element)
boolean add(Object o)
boolean addAll(Collection c)
boolean addAll(int index, Collection c)
void clear()
boolean contains(Object o)
boolean containsAll(Collection c)
boolean equals(Object o)
Object get(int index)
int hashCode()
int indexOf(Object o)
boolean isEmpty()
Iterator iterator()
int lastIndexOf(Object o)
ListIterator listIterator()
ListIterator listIterator(int index)
Object remove(int index)
boolean remove(Object o)
boolean removeAll(Collection c)
boolean retainAll(Collection c)
Object set(int index, Object element)
int size()
List subList(int fromIndex, int toIndex)
Object[] toArray()
Object[] toArray(Object[] a)
```

Did Josh Bloch have to start from scratch when he wanted to implement the class ArrayList, which implements the List interface?

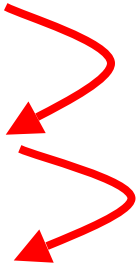
Subclasses inherit implementation

- Specify inheritance relationship using **`extends`**

```
public class ArrayList extends AbstractList {
```

```
public abstract class AbstractList
        extends AbstractCollection implements List {
```

```
public abstract class AbstractCollection
        implements Collection {
```



- AbstractCollection implements some methods of the Collection interface, but not all of them.
  - » it is declared to be an **`abstract`** class

java.util

Class ArrayList

```
java.lang.Object
  |
  +--java.util.AbstractCollection
        |
        +--java.util.AbstractList
              |
              +--java.util.ArrayList
```

All Implemented Interfaces:

Cloneable, Collection, List, RandomAccess, Serializable

---

public class **ArrayList**
extends AbstractList
implements List, RandomAccess, Cloneable, Serializable

Methods inherited from class java.util.AbstractList
equals, hashCode, iterator, listIterator, listIterator, subList

Methods inherited from class java.util.AbstractCollection
containsAll, remove, removeAll, retainAll, toString

Methods inherited from class java.lang.Object
finalize, getClass, notify, notifyAll, wait, wait, wait

Method Summary

void	**add**(int index, Object element) Inserts the specified element at the specified position in this list.
boolean	**add**(Object o) Appends the specified element to the end of this list.
boolean	**addAll**(Collection c) Appends all of the elements in the specified Collection to the end of this list, in the order that they are returned by the specified Collection's Iterator.
boolean	**addAll**(int index, Collection c) Inserts all of the elements in the specified Collection into this list, starting at the specified position.
void	**clear**() Removes all of the elements from this list.
Object	**clone**() Returns a shallow copy of this `ArrayList` instance.
boolean	**contains**(Object elem) Returns `true` if this list contains the specified element.
void	**ensureCapacity**(int minCapacity) Increases the capacity of this `ArrayList` instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
Object	**get**(int index) Returns the element at the specified position in this list.
int	**indexOf**(Object elem) Searches for the first occurence of the given argument, testing for equality using the `equals` method.
boolean	**isEmpty**() Tests if this list has no elements.
int	**lastIndexOf**(Object elem) Returns the index of the last occurrence of the specified object in this list.
Object	**remove**(int index) Removes the element at the specified position in this list.
protected void	**removeRange**(int fromIndex, int toIndex) Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
Object	**set**(int index, Object element) Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.
Object[]	**toArray**() Returns an array containing all of the elements in this list in the correct order.
Object[]	**toArray**(Object[] a) Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.
void	**trimToSize**() Trims the capacity of this `ArrayList` instance to be the list's current size.

Abstract classes

- Recall that we can think of a class as a blueprint for making objects
- An abstract class is a blueprint that is missing some details that must be filled in later
  - » the abstract class can identify some methods that must be implemented by any subclass

    "there must be a garage, but it's not specified here"
  - » the abstract class can claim to implement an interface, but leave the details to the subclasses

    "this building design is approved for occupancy, but the fire escapes must be added in the final design before use"

AbstractCollection

- Implements

```
boolean add(Object o)
boolean addAll(Collection c)
void clear()
boolean contains(Object o)
boolean containsAll(Collection c)
boolean isEmpty()
boolean remove(Object o)
boolean removeAll(Collection c)
boolean retainAll(Collection c)
Object[] toArray()
Object[] toArray(Object[] a)
String toString()
```

- Does not implement

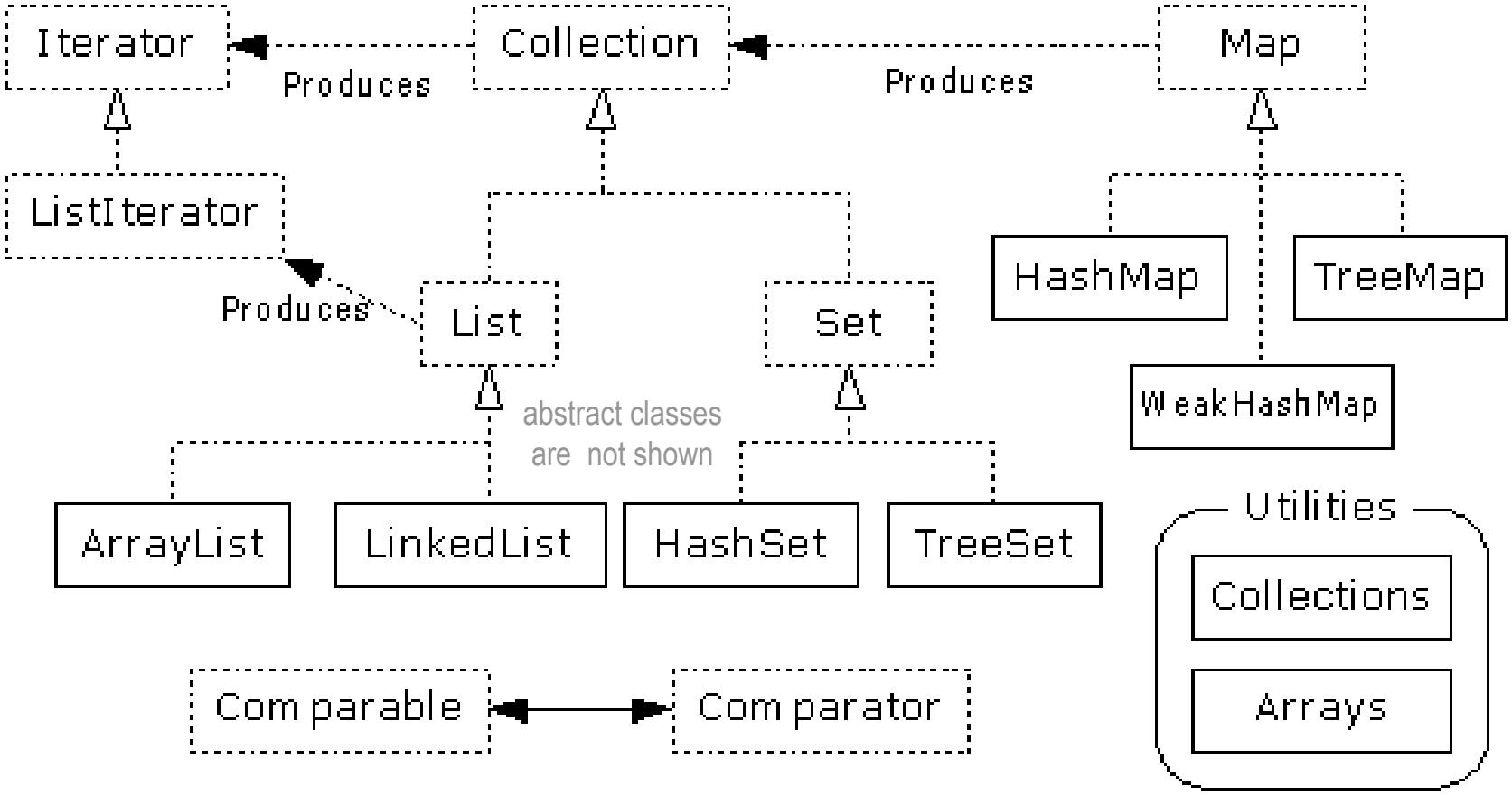
```
abstract Iterator iterator()
abstract int size()
```

These are not necessarily the fastest implementations, because the specific collection might have special features that could be used to speed them up, but at least there is something to get started with.

Design Pattern

- The pattern shown for ArrayList is a good design pattern
- Interface
  - » defines the capabilities that must be present
- Abstract Base Classes
  - » basic implementation of some or all methods
- Concrete classes
  - » complete and well designed implementations

Collections Framework Diagram



Interfaces, Implementations, and Algorithms
From Thinking in Java, page 462