



---

# Abstract Data Types

Wellesley College CS230  
Lecture 08  
Monday, February 26  
Handout #16

*Note: This lecture includes slides from Lectures  
06 & 07 that were not covered in those lectures*

Problem Sets: PS2 due Monday, February 26  
PS3 due Wednesday, March 7

08 1



---

## What is an Abstract Data Type (ADT)?

An Abstract Data Type (ADT) is a contract that specifies the behavior of methods that operate on a data structure without exposing the representation of the data structure.

In general, there are many different implementations of a given ADT.

In this lecture:

- We explore ways of specifying and implementing ADTs in Java.
- We study some sample ADTs:
  - Immutable Points
  - Comparables
  - Iterators
  - Stacks

08 2



## Immutable Point Contract

---

// Constructor Method

```
public IPoint (int x, int y); /* Create an immutable point with
                               coordinates x and y */
```

// Instance Methods

```
public int getX(); // Return the x-coordinate of this point.
public int getY(); // Return the y-coordinate of this point
public String toString(); // Return string of form (x,y)
```

08 3



## Two Implementations of Immutable Points

---

```
public class IPointVars { // Give classes (and constructors) different names
    private int x, y; // Represent coords with two int variables.
    public IPointVars (int x, int y) {this.x = x; this.y = y;}
    public int getX() { return x; }
    public int getY() { return y; }
    public String toString() { return "(" + getX() + "," + getY() + ")"; }
}
```

```
public class IPointArray {
    private int[] ints = new int[2]; // Represent coords with two-slot array.
    public IPointArray (int x, int y) {ints[0] = x; ints[1] = y;}
    public int getX() { return ints[0]; }
    public int getY() { return ints[1]; }
    public String toString() { return "(" + getX() + "," + getY() + ")"; }
}
```

08 4



## Manipulating Immutable Points

```
public static int sumCoordsVars (IPointVars p) {  
    return p.getX() + p.getY();  
}
```

```
public static int sumCoordsArray (IPointArray p) {  
    return p.getX() + p.getY();  
}
```

The sumCoord methods are the same modulo the point type.  
Can we define a single sumCoords() method that works on instances of both classes?

Yes! By using (1) an interface or (2) an abstract class.

08 5



## An IPoint Interface

```
public interface IPoint {  
    public int getX();  
    public int getY();  
    public String toString();  
}
```

An interface specifies the types of some instance method contracts.

Gives new information to Java type checker

```
public class IPPointVars implements IPoint { ... same impl. as before ...}  
public class IPPointArray implements IPoint { ... same impl. as before ...}
```

```
public static int sumCoords (IPoint p) { return p.getX() + p.getY(); }
```

```
sumCoords(new IPPointVars(1,2)) → 3  
sumCoords(new IPPointArray(3,4)) → 7
```

08 6



## An IPoint Abstract Class

A class must be declared **abstract** if any of its methods are declared **abstract**.

```
public abstract class IPoint {  
    public abstract int getX();  
    public abstract int getY();  
    public String toString() { return "(" + getX() + "," + getY() + ")"; }  
}
```

A method without a body is declared abstract.

An abstract class may have concrete methods; these can use abstract methods.

Use **extends** rather than **implements** with abstract class

```
public class IPointVars extends IPoint  
    { ... same impl. as before except no toString() method ...}  
public class IPointArray extends IPoint  
    { ... same impl. as before except no toString () method ...}  
  
public static int sumCoords (IPoint p) { return p.getX() + p.getY(); }  
  
sumCoords(new IPointVars(1,2)) → 3  
sumCoords(new IPointArray(3,4)) → 7
```

08 7



## Interfaces vs. Abstract Classes

Interfaces are used solely to specify a contract (API/ADT)

- All methods in an interface must be **public instance methods** (no class methods or constructor methods)
- All variables in an interface must be **public static final variables** with initializers. E.g.  
public static final CLUBS = 0; public static final DIAMONDS = 1; ...
- Interfaces may have **subinterfaces**:  
public interface ColoredIPoint extends IPoint { ... }
- Classes can extend one class but **implement any number of interfaces**.  
public class Frobnitz extends Frob  
 implements IPoint, Cloneable, Comparable<Frobnitz>
- **In CS230, we will use interfaces to specify ADTs**

Abstract Types are used to specify a partially implemented class; they can also be used for an API.

- Unlike interfaces, abstract classes may **include methods with code**.
- Abstract classes may **include (non-abstract) static methods**.
- **In CS230, we will use abstract classes to specify methods shared among different implementations of the same interface.**

08 8



## Interfaces and Abstract Classes in Java

---

Some standard Java Interfaces:

- Cloneable: `public Object clone();`
- Comparable<T>: `public int compareTo (T x);`
- java.util.Iterator<T>: `public boolean hasNext();`  
`public T next();`  
`public void remove(); (optional)`
- java.util.Iterable<T>: `public Iterator<T> iterator();`
- java.util.Collection<T>: ... *discussed in a later lecture ...*
- java.awt.event.ActionListener: ... *discussed in GUI lectures ...*

Some CS230 Java Interfaces: Stack<T>, Queue<T>, Set<T>, Bag<T>, ...

Some standard Java Abstract Classes:

- InputStream
- java.awt.Component
- java.awt.AWTEvent

08 9



## Cloneable & Comparable Example

---

```
public class FancyIPointVars extends IPointVars
    implements Cloneable, Comparable<IPoint> {

    public FancyIPointVars (int x, int y) { super(x,y); }

    public Object clone() { return new FancyIPointVars(getX(), getY()); }

    // Compare lexicographically, first by x position, then by y position
    public int compareTo (IPoint p) {
        if (getX() < p.getX()) {
            return -1;
        } else if (getX() > p.getX()) {
            return 1;
        } else {
            return getY() - p.getY();
        }
    }
}
```

08 10



## Cloneable & Comparable Example Continued

---

```
FancyIPointVars p = new FancyIPointVars (3,4);
FancyIPointVars p2 = (FancyIPointVars) p.clone();

p.compareTo(p2) →
p.compareTo(new IPointVars(2,3)) →
p.compareTo(new IPointVars(3,0)) →
p.compareTo(new IPointVars(3,10)) →
p.compareTo(new IPointVars(4,0)) →
(new IPointVars(4,0)). compareTo(p) →
```

08 11



## Iterator Example

---

```
public static <T> void displayIterator (Iterator<T> elts) {
    while elts.hasNext() {
        System.out.println(elts.next().toString());
    }
}
```

Suppose *v* is a `Vector<String>` with the following string elements:  
"aardvark", "beetle", "cat", "dog", "elephant"

Then `displayIterator(v.iterator())` displays the following on the console:

```
aardvark
beetle
cat
dog
elephant
```

08 12



## Iterator Removal Example

---

```
public static void removeShort (Iterator<String> elts) {  
    while elts.hasNext() {  
        if (elts.next().length() <= 3) {  
            elts.remove();  
        }  
    }  
}
```

Then `removeShort(v); displayIterator(v.iterator())` displays the following:

```
aardvark  
beetle  
elephant
```

08 13



## Vector Filtering Gotcha!

---

How could we remove short strings from a vector without an iterator?

```
Public static void removeShort (Vector<String> v) {
```

```
}
```

08 14



## Stack Interface

```
import java.util.*; // imports Iterator and Iterable

/** Interface to mutable stacks with elements of type T.
 * A stack is a last-in-first-out (LIFO) collection with duplicates.
 * WARNING: This CS230 Stack is an interface, and is *different*
 * from the class named java.util.Stack. */
public interface Stack<T> extends Iterable<T> {

    public void push(T elt); // Push elt on top of the stack.
    public T pop(); // Remove and return top element from stack.
    // Throw RuntimeException if stack is empty.
    public T top(); // Return top element of stack without removing it.
    // Throw RuntimeException if stack is empty.
    public boolean isEmpty(); // Return true if this stack is empty; else false
    public int size(); // Return the number of elements in this stack
    public void clear(); // Remove all elements from this stack
    public Stack<T> copy(); // Return a shallow copy of this stack.
    public Iterator<T> iterator(); // Return an iterator that yields this stack's
    // elements from top to bottom.

}
```

08 15



## Stack Examples 1

```
Stack<String> s = new Stack<String>;

s.isEmpty() →      s.size() →      s.top() →

s.push("B");      s.isEmpty() →      s.size() →      s.top() →

s.push("C");      s.isEmpty() →      s.size() →      s.top() →

s.push("B");      s.isEmpty() →      s.size() →      s.top() →

s.push("A");      s.isEmpty() →      s.size() →      s.top() →

s.pop() →         s.isEmpty() →      s.size() →      s.top() →

s.pop() →         s.isEmpty() →      s.size() →      s.top() →

s.pop() →         s.isEmpty() →      s.size() →      s.top() →
```

08 16



## Stack Examples 2

---

```
s.push("D");      s.isEmpty() →      s.size() →      s.top() →
s.push("A");      s.isEmpty() →      s.size() →      s.top() →
s.push("C");      s.isEmpty() →      s.size() →      s.top() →
Stack<String> s2 = s.copy();
                  s.isEmpty() →      s.size() →      s.top() →
                  s2.isEmpty() →     s2.size() →     s2.top() →
s.clear();        s.isEmpty() →      s.size() →      s.top() →
                  s2.isEmpty() →     s2.size() →     s2.top() →
displayIterator(s2.iterator());
displayIterator(s.iterator());
```

08 17



## Better Printout of Stack Contents

---

```
/** prints the contents of a Stack from top to bottom */
public static <T> void printStack (Stack<T> stk) {
    Stack<T> tempStack = new Stack<T>();
    //tempStack to hold stk contents
    String S;
    System.out.print("Contents of Stack from top to bottom: ");
    while (!stk.empty()) {
        System.out.print(stk.pop().toString + " ");
        tempStack.push(S);
    }
    System.out.println();
    // restore contents of stk
    while(!tempStack.isEmpty())
        stk.push(tempStack.pop());
}
```

08 18

