Interfaces In Java

Prof. Chris Jermaine
cmj4@cs.rice.edu

Now On To Interfaces in Java

- Java gives the ability to declare an “interface”
- Like a class, except:
  - — Can’t declare any member variables (well, you can, but don’t do it)
  - — All functions are implicitly abstract, public
  - — So no implementations for anything!
- Example: “Iterator” in Java standard class library:

```
interface Iterator <E> { // note: this is a generic interface
  boolean hasNext ();    // parameterized on type E
  E next ();
  void remove ();
}
```

- So why does Java have these?

Interfaces vs. Abstract Classes

- Like an abstract class
- Very similar, but are a few key differences
- A class “implements” an interface, vs. “extends” another class.
- Ex:

```
class IntArrayIter implements Iterator <Integer> { }
```

- Or, if “Iterator” hadn’t been generic:

```
class IntArrayIter implements Iterator { }
```

Interfaces vs. Abstract Classes

- Like an abstract class
- Very similar, but are a few key differences
- A class “implements” an interface, vs. “extends” another class.
- Ex:

```
class IntArrayIter implements Iterator <Integer> { }
```

- Or, if “Iterator” hadn’t been generic:

```
class IntArrayIter implements Iterator { }
```

- Another key difference: multiple inheritance is allowed:

```
class IntArrayIter implements Iterator <.>, IResettable { }
// “IResettable” might look like:
interface IResettable {
  void startOver ();
}
```

Interfaces vs. Abstract Classes

- Very similar, but are a few key differences
- A class “implements” an interface, vs. “extends” another class.
- Ex:

```
class IntArrayIter implements Iterator <Integer> { }
```

- Or, if “Iterator” hadn’t been generic:

```
class IntArrayIter implements Iterator { }
```

- Another key difference: multiple inheritance is allowed:

```
class IntArrayIter implements Iterator <.>, IResettable { }
```

Is this a good idea here?

```
// “IResettable” might look like:
interface IResettable {
  void startOver ();
}
```

Don’t Abuse Multiple Inheritance!

- In this example, would have been much better to extend “Iterator”:

```
interface IResettable <T> extends Iterator <T> {
  void startOver ();
}

class IntArrayIter implements IResettable <Integer> { }
```

- Why is this better?

Don’t Abuse Multiple Inheritance!

- In this example, would have been much better to extend “Iterator”:

```
interface IResettable <T> extends Iterator <T> {
  void startOver ();
}

class IntArrayIter implements IResettable <Integer> { }
```

- Why is this better?

  — We really are adding to the interface... “IResettable” does not make much sense outside the context of “Iterator”

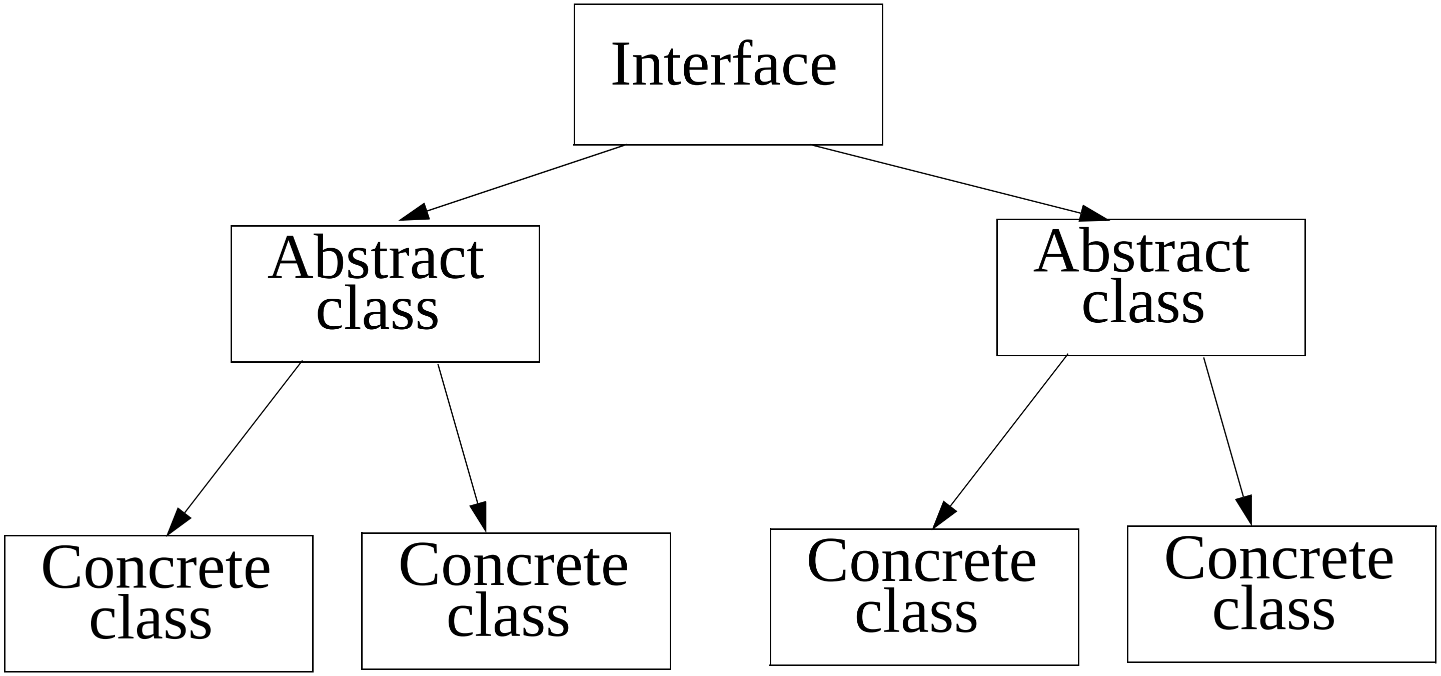
  — Use multiple inheritance only if a class really provides many, totally separate types of functionality

So, Why Do We Like Interfaces?

- It’s very useful to have the ability to write code whose only purpose is to describe functionality
  - No implementaton needed or wanted
  - Abstraction in its purest form

Question: When Should You Use Abstract Classes, Interfaces, “Regular Classes”

- Typically, you will use all three:



What Do You Put In An Interface?

- All of the functionality that is so abstract...
- That it has nothing to do with an implementation
  - Examples:
  - A stack has “push” and “pop”
  - A queue has “enqueue” and “dequeue”
  - A map has “put” and “get”

What Do You Put In An Abstract Class?

- Here you’ll imp functionality that many/all imps will share
- Often, it will be interface functions that can be written in terms of other interface functions
- Try hard to pull functionality into the abstract
  - — Avoids repeating work!
- Out of the following methods:

```
void push (int);
int pop ();
int size ();
void reverse ();
```

  - — Which one would most likely go in the an abstract class?

What Do You Put In An Abstract Class?

- Might put “reverse” in there:

```
abstract class AIntStack implements IInstStack {

  public abstract void push (int i);
  public abstract int pop ();
  public abstract int size ();
  public void reverse () {
    ????
  }

}
```

Abstract Ops Not In Interface

- Why? Can implement reverse using just push and pop
- How?

Abstract Ops Not In Interface

- Why? Can implement reverse using just push and pop
- How?

```
abstract class AIntStack implements IInstStack {
  public void reverse () {
    if (size () != 0) {
      int i = pop ();
      reverse ();
      addAtBottom (i);
    }
  }

  private void addAtBottom (int i) {
    if (size () == 0)
      push (i);
    else {
      int j = pop ();
      addAtBottom (i);
      push (j);
    }
  }
}
```

Abstract Ops Not In Interface

- Note that this requires an “addAtBottom” routine
- Pretty slow, since linear time in terms of push and pop
  - Instead, could just require the concrete to implement it
  - Since many implementations are going to be constant time
- Example:

```
abstract class AIntStack implements IInstStack {
  public abstract void push (int i);
  public abstract int pop ();
  public abstract int size ();
  protected abstract void addAtBottom (int i);
  public void reverse () {...
  }
}
```

Abstract Ops Not In Interface

- Note that this requires an “addAtBottom” routine
- Pretty slow, since linear time in terms of push and pop
  - Instead, could just require the concrete to implement it
  - Since many implementations are going to be constant time
- Example:

```
abstract class AIntStack implements IInstStack {
  public abstract void push (int i);
  public abstract int pop ();
  public abstract int size ();
  protected abstract void addAtBottom (int i);
  public void reverse () {...
  }
}
```

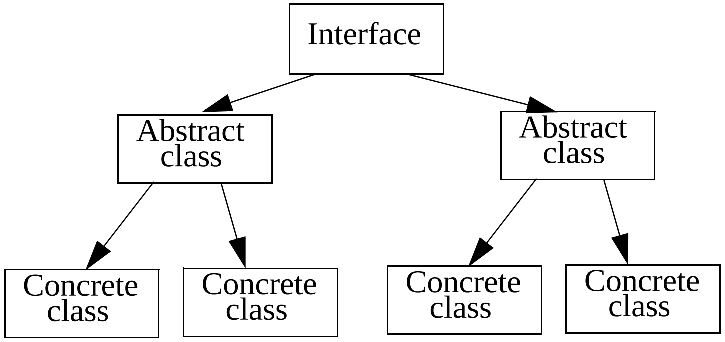
- Note: if happy with linear, then leave imp in abstract
  - Concrete can over-ride if desired

What Do You Put In The Concrete Class?

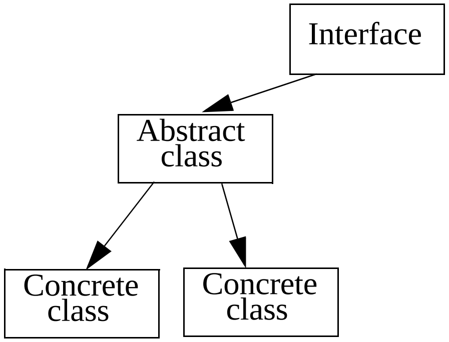
- The actual implementation!
- Ex: can implement “AIntStack” using “ArrayList <Integer>”:
  - — “push (i)” uses “list.add (i)” (adds integer at end of list)
  - — “pop ()” uses “list.remove (list.length () - 1)”
  - — “isEmpty ()” uses “list.length () == 0”
  - — addAtBottom adds to front of the list (still linear time!)

Now Time for Some Navel Gazing

- Will you actually ever have this?

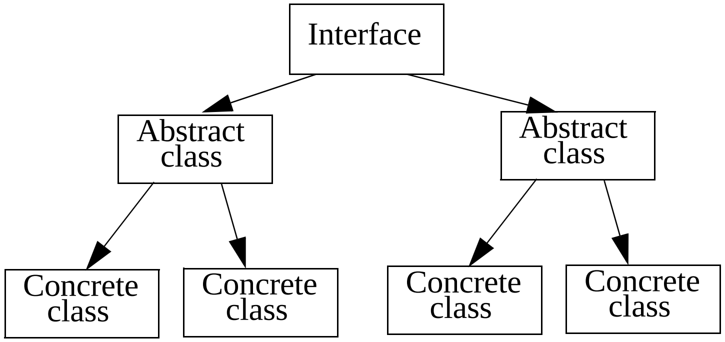


Or is it always this:



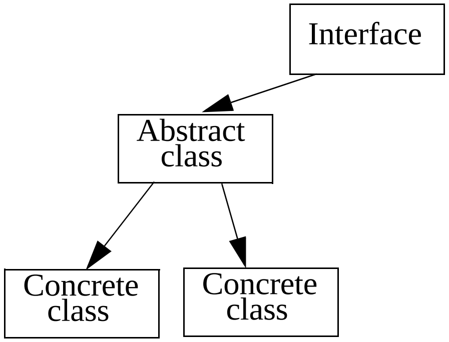
Now Time for Some Navel Gazing

- Will you actually ever have this?



This one should be pretty rare!

Or is it always this:



Why Avoid Multiple Abstract Imps?

- Often this is the knee-jerk reaction to multiple implementations of interface ops that can be written in terms of other interface ops
- But this can be problematic
  - — Imagine multiple stack implementations
  - — Along with multiple implementations of “reverse”
  - — Where all are interchangeable
  - — What happens if use “multiple abstract class” solution, put each in diff abstract?

Why Avoid Multiple Abstract Imps?

- Often this is the knee-jerk reaction to multiple implementations of interface ops that can be written in terms of other interface ops
- But this can be problematic
  - Imagine multiple stack implementations
  - Along with multiple implementations of “Reverse”
  - Where all are interchangeable
  - What happens if use “multiple abstract class” solution, put each in diff abstract?

    End up repeating concrete implementations with each abstract class!

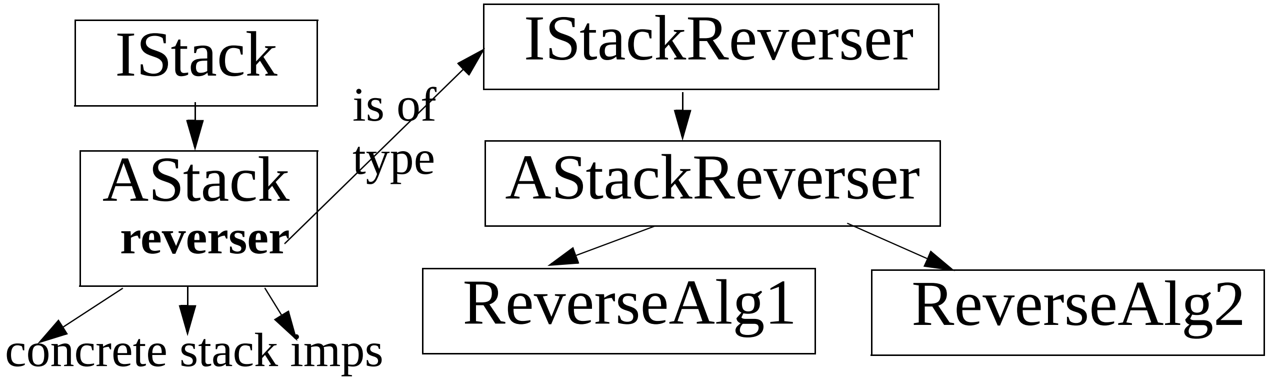
Why Avoid Multiple Abstract Imps?

- Often this is the knee-jerk reaction to multiple implementations of interface ops that can be written in terms of other interface ops
- But this can be problematic
  - — Imagine multiple stack implementations
  - — Along with multiple implementations of “Reverse”
  - — Where all are interchangeable
  - — What happens if use “multiple abstract class” solution, put each in diff abstract?

    End up repeating concrete implementations with each abstract class!
  - — What should we have done?

Why Avoid Multiple Abstract Imps?

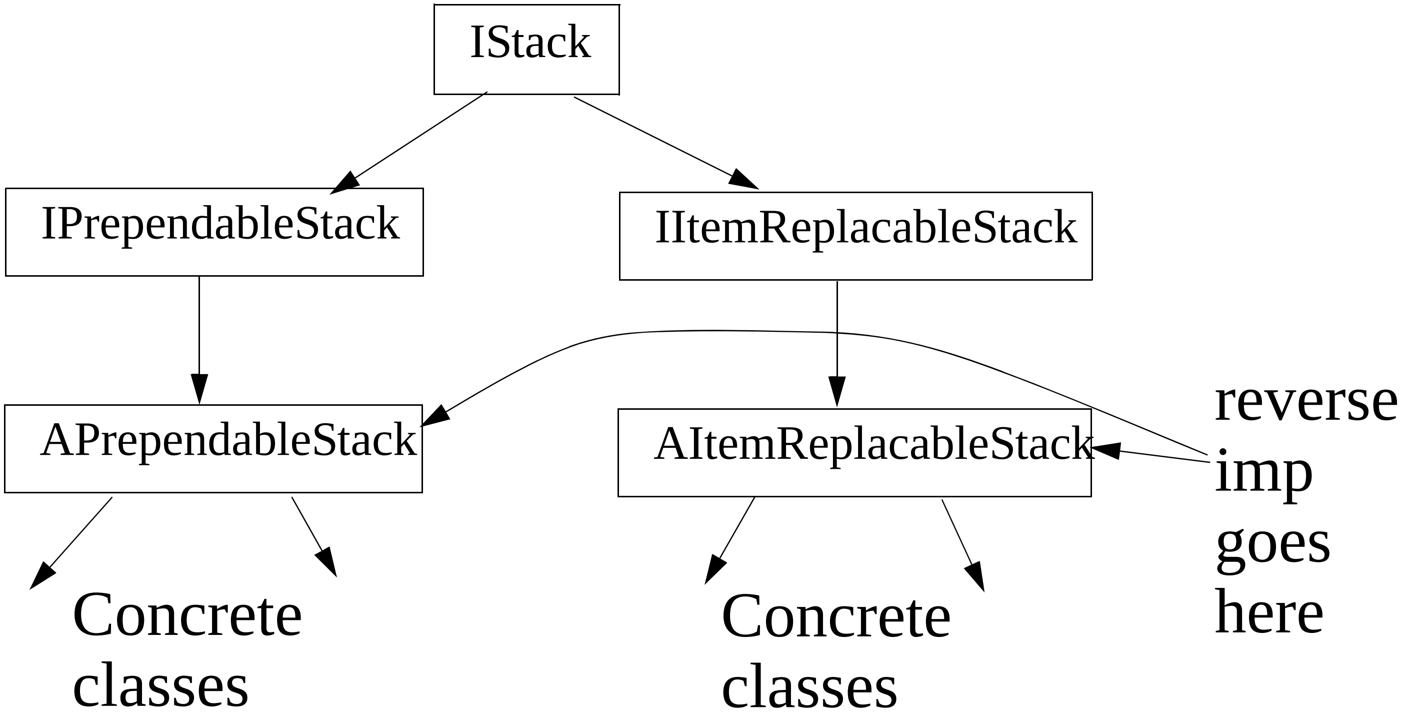
- Often this is the knee-jerk reaction to multiple implementations of interface ops that can be written in terms of other interface ops
- But this can be problematic
  - — Imagine multiple stack implementations
  - — Along with multiple implementations of “Reverse”
  - — Where all are interchangeable
  - — What happens if we use the “multiple abstract class” solution?
  - — What should we have done?



Do We Ever Actually Need Mult Abstracts?

- What if diff implementations of reverse depend on different ops...
  - — And those ops are not gonna be common to all imps?
  - — Imagine multiple stack implementations
  - — Easy to implement “addAtBottom” for some imps
  - — Easy to implement “replaceValAtPosI (val, pos)” for other imps
  - — (An aside: what does “reverse” look like using “replaceValAtPosI”?)
- It’s clear we can’t use prior design strategy here
- In this case, should you have multiple abstract classes?
  - — Not clear...
  - — What might we have done instead?

A Better Design?



Sooo... is this better? I'd say: it depends

Last Bit of Navel Gazing

- Should you **always** have a complete hierarchy?
  - — Interface, abstract base, (multiple?) concrete
- Some will argue emphatically “**YES!**”
  - — Perhaps even in later classes you’ll be told this?
- I’ll be a little more permissive
  - — Either have one level (only a single concrete base)
  - — Or three or more (that is, never start with an abstract base)
- Why?

Last Bit of Navel Gazing

- Should you **always** have a complete hierarchy?
  - — Interface, abstract base, (multiple?) concrete
- Some will argue emphatically “**YES!**”
  - — Perhaps even in later classes you’ll be told this?
- I’ll be a little more permissive
  - — Either have one level (only a single concrete base)
  - — Or three or more (that is, never start with an abstract base)
- Why?
  - — Sometimes you’ll have a class where you **know** you’ll never need mult. imps
  - — Just make sure to switch over at first sign of trouble!
  - — But if you’ve got an abstract base, just define the interface

This Sort of Does It For the Formal Intro To OODesign

- Talked about how one uses inheritance, polymorphism, and the proper role of interfaces
  - — You'll get this much more rigorously in COMP 310
  - — Will distill many of the ideas we've discussed here into "design patterns"
- Closing throughts
  - — I know many of you are skpetical. But keep in mind:

This Sort of Does It For the Formal Intro To OODesign

- Talked about how one uses inheritance, polymorphism, and the proper role of interfaces
  - You’ll get this much more rigorously in COMP 310
  - Will distill many of the ideas we’ve discussed here into “design patterns”
- Closing throughts
  - I know many of you are skpetical. But keep in mind:
  - I’ll grant you that the best programmers can write 50K SLOC in a year...
  - ...and never think explicitly about design...
  - ...and the code just works perfectly
  - HOWEVER, most people (inluding, probably, you!) just are not that good
  - And even if you are, heaven forbid you ever move on...

We Finish Up With Some Notes on A2

- Goal is to implement the “IDoubleVector” interface
  - Having a vector of doubles is a key abstraction to implementing our ML algorithm
- We’ll need two actual implementations
  - A dense one (built on top of array of doubles)
  - A sparse one (built on top of ISparseArray generic, which you’ll imp next)
  - What’s the diff between a dense array and a sparse array?

Need to Have a Notion of a “Backing Value”

- What’s that?
  - — It’s the default value for entry in an IDoubleVector
  - — Every non-empty slot is actually a “delta” or diff from the backing value
  - — Allows you to add same number to every entry in constant time
  - — Vital for sparse vector, might as well use for dense one, too

Most Ops Are Self-Explanatory

- In case you haven’t seen it, the “L1” norm of a vector...
  - — Is the sum of the absolute values of the entries in the vector
- Noramlization...
  - — Means you scale all of the entries to the L1 norm is one
  - — Keeping all ratios constant

A Super-Quick Note on Java Exceptions

- An “exception” tells caller that there was a problem in a method
  - — Caller is forced to handle this using the “try-catch” framework
  - — Look at test code to see this
- Many of the IDoubleVector ops throw “OutOfBoundsException”
  - — This means that whenever someone does something out of range...
  - — You should execute the line:

```
throw new OutOfBoundsException ();
```

- Much more on exceptions later on...