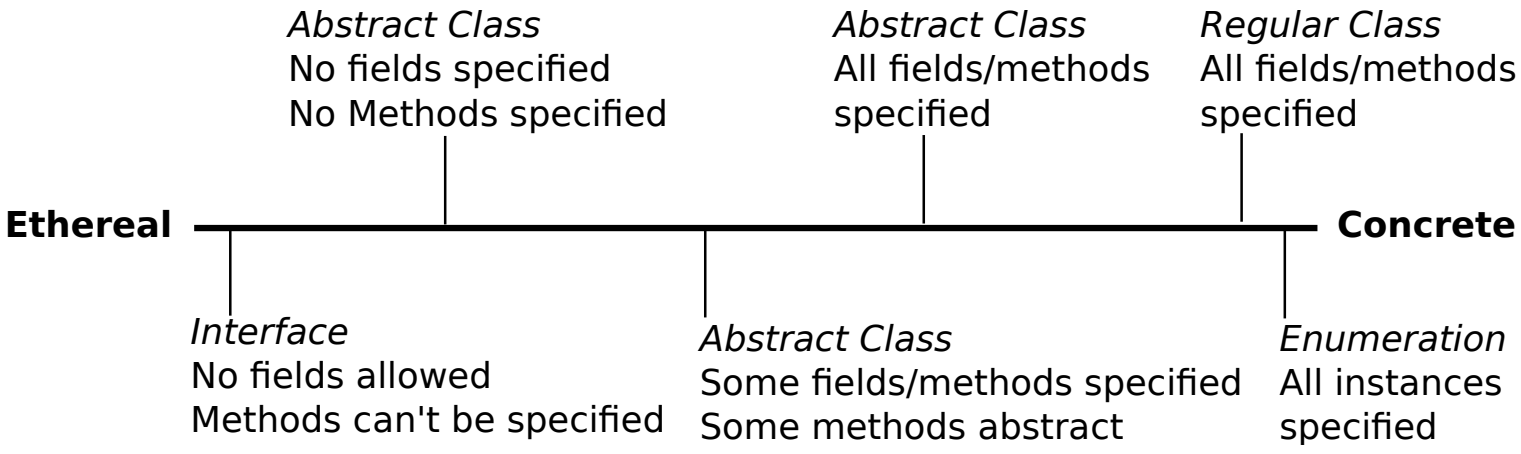
CS 211: Interfaces

Chris Kauffman

Week 8

The Continuum of Java's Top-Level Entities



- Regular classes are more concrete
- Abstract classes are more ethereal
- Enumerations are as concrete as possible
- Interfaces are as ethereal as possible

Quick Input Calculation

Consider simple data file `scores.dat` of name / score

```
Adama    17.0
Baltar   18.0
Thrace   16.0
Tye      10.0
Rosslyn  15.0
```

Write some quick code that computes

- Mean score: average of all scores
- Max score with name of max earner

Sample output

```
> cat scores.dat
Adama    17.0
Baltar   18.0
Thrace   16.0
Tye      10.0
Rosslyn  15.0
> javac SimpleScores.java
> java SimpleScores
Mean: 15.20  Max: 18.00 by Baltar
```

Generalizations

More Columns

First	Last	HW1	HW2	Exam1
Lee	Adama	17.0	12.0	34.0
Gaius	Baltar	18.0	13.0	38.0
Kara	Thrace	16.0	10.0	24.5
Saul	Tye	10.0	13.5	34.0
Laura	Rosslyn	15.0	12.0	36.0

More Statistics

Mean, max(name), min(name), total, standard deviation, mode, median...

- Which of these are easy/hard?
- What is the general pattern of a *statistic*?
- Can we generalize?

A Possible Pattern

A statistic

- Has an initial value (may be `NaN`)
- Can be *updated* with new input value
- Can report its current *value*
- Can be stringified

Is there a default implementation of these that fits for several statistics?

- Mean, max(name), min(name), total, standard deviation, mode, median...

Which of the 4 top-level Kinds are appropriate?

`class`

- Run of the mill concrete objects
- Child classes `extend`

`enum`

- Like a class (fields methods) except...
- All instances declared up front, automatically `static final`
- Good for modeling fixed collections
- Cannot `extend`

`abstract class`

- Can't instantiate but good for *single* inheritance hierarchies,
- Child classes `extend`

`interface`

- Can't instantiate
- Good for capabilities cutting across class hierarchies: savable, accessible, observable, comparable
- Child classes `implement`

Abstract Statistic

```
// What can statistics "do"?
public abstract class Statistic {
  public abstract double value();    // Current value
  public abstract void update(String s, double x); // Update
  public abstract String toString(); // Pretty print
}
```

Establish A hierarchy rooted at `Statistic`

```
Statistic mean = new Mean();
Statistic stdev = new StandardDev();
```

Benefit `abstract` methods don't need to be written: no body present, just prototype

Cost Can't actually create a plain `Statistic`

```
Statistic s = new Statistic();
// Error: Statistic is abstract;
//        cannot be instantiated
```

Alternative: `interface`

If `Statistic` isn't going to have any fields nor any concrete methods, why make it a class at all.

Abstract Parent

```
public abstract class Statistic {
  public abstract double value();
  public abstract
    void update(String s, double x);
  public abstract String toString();
}
```

Interface

```
public interface Statistic {
  public double value();
  public void update(String s,
                     double x);
  public String toString();
}
```

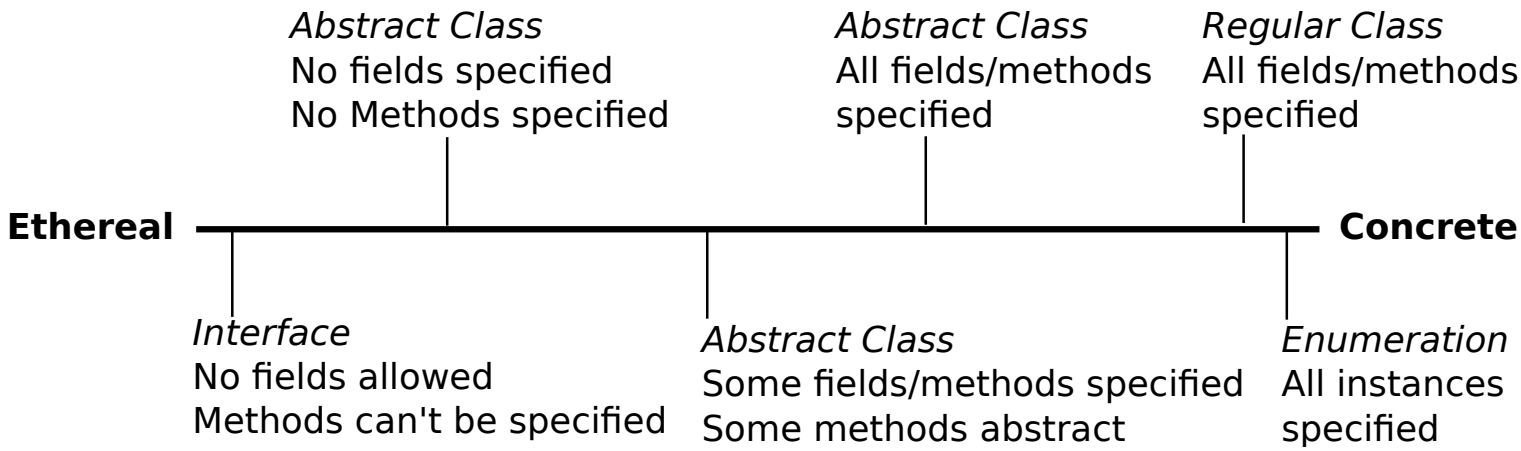
All methods automatically `abstract`, can't specify a body for any of them

Walk-Through: Implementations

Want substitutable behaviors

- Substitute: hard-coded $\rightarrow$ parameterized
- Components have internal state and behavior
- `Statistic` is an interface
- `Mean`, `Max`, `Total` implement `Statistic`
- In `SimpleScores.java` walk through a transformation to use interchangeable parts

Abstract vs Interface



`interface` $\approx$ `abstract` class with all `abstract` methods

Similar Except

- Classes can only descend from one parent, abstract or not
- Classes can implement many `interfaces`
- Methods from both places referred to as "inherited from. . . "

Example: `JButton` (javadoc)

`extends` how many parents? vs `implements` how many interfaces?

Interface

Like a "capabilities badge" that classes can wear

Interfaces

I'm `interface Savable`. You must implement `void save(String fn)` to wear my badge.

```
public interface Savable{
  void save(String fname);
}
```

I'm `interface Describable`. You must implement both versions of the `describe()` method to wear my badge.

```
public interface Describable{
  void describe(AudioStream o);
  void describe(PrintStream o);
}
```

Implementing Classes

I'm `class C` and I can be saved because I `implements Savable`

```
public class C implements Savable {
  public void myMeth(){...}
  public void save(String fn){...}
}
```

I'm `class D`, my parent is `X` and I'm both `Savable` and `Describable`

```
public class D extends X
implements Savable, Describable {
  public void save(String fn){...}
  public void
    describe(AudioStream o){...}
  public void
    describe(PrintStream o){...}
  public String dooDad(){...}
}
```

Interface Particulars

Methods are automatically `public abstract`

```
public interface Savable{
  void save(String fname);
}
```

IDENTICAL TO

```
public interface Savable{
  public abstract
  void save(String fname);
}
```

Can form a Hierarchy (infrequent)

```
public interface Savable{
  void save(String fname);
}
```

```
public interface ReadWritable
extends Savable {
  void load(String fname);
}
```

```
public class C
implements ReadWritable {
  public void save(String f){...}
  public void load(String f){...}
}
```

Interfaces Set Up Interchangeable Parts

- Any class that `implements Savable` can be in an array of `Savable` objects
- Dynamic dispatch to the specific object's version of `save(f)`

```
class D extends X
implements Savable, Describable
{..}

public class C
implements Savable
{..}

public interface Savable{
  void save(String fname);
}
```

```
public class X {
  public static void
  saveAll(Savable [] arr,
          String [] fnames)
  {
    for(int i=0; i<arr.length; i++){
      Savable s = arr[i];
      String f = fnames[i];
      s.save(f);
    }
  }

  public static
  void main(String args[]){
    Savable [] sa = {new C(),
                     new D(),
                     new Y()};
    X.saveall(sa);
  }
}
```

Interface Examples

Code pack has versions of `Statistic` as both

- Abstract class hierarchy
- Interface implemented by classes

Both look similar

Implements vs Extends

Recall `Statistic`: formerly `abstract class`, now `interface`

Interface Definition

```
public interface Statistic {
  public double value();
  public void
   update(String s, double x);
  public String toString();
}
```

Implementing Class

```
public class Total implements Statistic
{
  protected double sum;
  public Total(){
    this.sum=0.0;
  }
  public void update(String s,
                     double d){
    this.sum += d;
  }
  public double value(){
    return sum;
  }
  public String toString(){
    return
      String.format("total %5.2f",
                    this.sum);
  }
}
```

Exercise: IntOp Interface

```
interface IntOp{
  public int transform(int i);  // Transform arg
  public int opsPerformed();    // Track # calls to transform(..)
  public String toString();     // Show string repr
}

public class OpDemo{
  public static void main(String args[]) {
    IntOp [] ops = {
      new DoubleIt(), new IncrIt(),
      new TwoPowIt(),
    };
    int arg = 10;
    printf("Transforms of %d\n",arg);
    for(IntOp op : ops){
      int ans = op.transform(arg);
      printf("%s (%d) : %d\n",op,arg,ans);
    }
    ops[0].transform(5);  // One more for 0th
    printf("\nNumber of transforms\n");
    for(IntOp op : ops){
      int opsPerformed = op.opsPerformed();
      printf("%s : %d\n",op,opsPerformed);
    }
  }
}
```

Provide implementations for `DoubleIt`, `IncrIt`, `TwoPowIt` so that the following output is produced

```
> javac OpDemo.java
> java OpDemo
Transforms of 10
DoubleIt (10) : 20
IncrIt (10) : 11
TwoPowIt (10) : 1024

Number of transforms
DoubleIt : 2
IncrIt : 1
TwoPowIt : 1
```

Common Interfaces Discussed Later

`interface Comparable<T>`

- Objects compare to each other
- Class has function `int compareTo(T y)`

```
if(x.compareTo(y) < 0){...
```

- `x.compareTo(y)`: Returns "x minus y"
  - Negative for `x` before `y`
  - 0 for equal
  - Positive for `x` after `y`

Things that are Comparable

From the `Comparable` Java Doc

`Integer, String, Double, File, IntBuffer, BigDecimal, Calendar, ...`

- Little shared functionality: bad candidate for class hierarchy enforcing `compare(..)` method
- Only shared feature: `compare(..)` method

Will discuss further later along with the related `Comparator` interface in context of searching and sorting.

implements vs extends

James Gosling

Why have both?

During the memorable Q&A session, someone asked [James Gosling]: "If you could do Java over again, what would you change?" "I'd leave out classes," he replied. After the laughter died down, he explained that the real problem wasn't classes per se, but rather implementation inheritance (the extends relationship). Interface inheritance (the implements relationship) is preferable. You should avoid implementation inheritance whenever possible.

– Allen Holub, Why extends is evil (2003)

Allen Holub