The Continuum of Java’s Top-Level Entities

- **Ethereal**
  - **Interface**: No fields allowed, Methods can't be specified
  - **Abstract Class**: No fields specified, No Methods specified

- **Concrete**
  - **Regular Class**: All fields/methods specified
  - **Abstract Class**: Some fields/methods specified, Some methods abstract
  - **Enumeration**: All instances specified

- **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

<table>
<thead>
<tr>
<th>Name</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adama</td>
<td>17.0</td>
</tr>
<tr>
<td>Baltar</td>
<td>18.0</td>
</tr>
<tr>
<td>Thrace</td>
<td>16.0</td>
</tr>
<tr>
<td>Tye</td>
<td>10.0</td>
</tr>
<tr>
<td>Rosslyn</td>
<td>15.0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>First</th>
<th>Last</th>
<th>HW1</th>
<th>HW2</th>
<th>Exam1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lee</td>
<td>Adama</td>
<td>17.0</td>
<td>12.0</td>
<td>34.0</td>
</tr>
<tr>
<td>Gaius</td>
<td>Baltar</td>
<td>18.0</td>
<td>13.0</td>
<td>38.0</td>
</tr>
<tr>
<td>Kara</td>
<td>Thrace</td>
<td>16.0</td>
<td>10.0</td>
<td>24.5</td>
</tr>
<tr>
<td>Saul</td>
<td>Tye</td>
<td>10.0</td>
<td>13.5</td>
<td>34.0</td>
</tr>
<tr>
<td>Laura</td>
<td>Rosslyn</td>
<td>15.0</td>
<td>12.0</td>
<td>36.0</td>
</tr>
</tbody>
</table>

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

### Abstract Parent

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

### Interface

```java
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

<table>
<thead>
<tr>
<th>Ethereal</th>
<th></th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interface</strong></td>
<td>☐ No fields allowed</td>
<td>☐ All fields/methods specified</td>
</tr>
<tr>
<td>☐ Methods can't be specified</td>
<td>☐ Some fields/methods specified</td>
<td>☐ Some methods abstract</td>
</tr>
<tr>
<td>☐ No Methods specified</td>
<td>☐ All fields/methods specified</td>
<td>☐ All instances specified</td>
</tr>
</tbody>
</table>

interface ≈ 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.

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

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

```java
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

```java
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

```java
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

```java
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){...}
}
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

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

Implementing Class

```java
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 \%.2f", this.sum);
    }
}
```
Exercise: IntOp Interface

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

```java
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

\nNumber of transforms
DoubleIt : 2
IncrIt : 1
TwoPowIt : 1
```
interface Comparable<T>

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

```java
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.
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)