

# Object Oriented Programming

- Even though we started using object oriented concepts in unit testing, we will go back to basics and define those concepts
- We will then talk about a small but real statistical analysis problem
- We will show how the best solution uses the object-oriented concepts that we discuss today

# What is an object?

- An object is a software bundle of related state and behavior
  - An object associates data (variables) with the methods used to manipulate that data
- Objects are used to model real world behavior
  - Example: A bicycle

# Bicycle object

- State
  - How fast it's going
  - Number of gears
  - The gear it's in
  - How hard the break is being applied
  - How quickly it's being pedaled
  - Which way the front wheel is pointing
- Behavior
  - Change from current gear to another gear
  - Break more / less
  - Turn front wheel left / right
  - Pedal faster / slower

# What is a class?

- A class is a blueprint from which individual objects are created
- For example, on the way here, you may have observed a bicycle chained to a pole as well as one being quickly pedaled by a messenger
- Both of these bicycles are instances of the class bicycle, but each has its own state
- Therefore, we say that an object is an instance of a class

# Three benefits of Object Oriented Programming

- Encapsulation
  - The ability to hide complexity inside an object while making the object's connections to the outside world simple and intuitive
- Inheritance
  - The ability to re-use code by inheriting functionality from a parent class
- Polymorphism
  - The ability to create objects that, to the outside world, look like they have the same behavior, but internally may behave differently

# What is inheritance?

- There are many different types of bicycles
  - Examples: Recumbent, racing bike, off-road-bike, hybrid, folding bike, tandem bike, etc.
- We call them all bicycles because they all have some essential features in common
- However, they each have certain features that separate them from their parent class
- Object oriented programming allows for classes to inherit some functionality from other classes while also implementing their own

# What is polymorphism?

- In computer science, polymorphism is a programming language feature that allows objects of different classes to be handled using a uniform interface.
- One way to implement polymorphism is through interfaces, the way we've implemented the StatsQInterface in our (upcoming) example
  - The actual q type is unknown, but the interface is tells us how to work with a class that implements StatsQInterface

# What is polymorphism? (2)

- Another way to implement polymorphism is through virtual methods, which we will cover in another lecture
- A real world example may be the pricing of an Instrument, which may be an Option, a Bond, etc.
- All of these are priced differently, but if our code knows it is pricing an object of a class derived from Instrument, it can call the getPrice method of the Instrument object and get the behavior of the derived Instrument – Option, Bond, Stock, etc.



# Define our problem

- Often, we want to perform an analysis on some rolling queue of data
- For example, we may have price relatives (returns) for the last fifteen minutes, and may want to know the sample standard deviation of those returns
- As new returns come in, we want to remove the effect of the oldest returns from our analysis
- We want to do this as efficiently as possible
- In the following example, we will consider how to write a Stats class that performs this calculation. Several versions will be written to demonstrate object-oriented design decisions
- We will also consider what is the best test for these classes

# Calculations

- Return

$$r_{t+1} = \frac{p_{t+1} - p_t}{p_t}, \quad \text{where } p \text{ is price}$$

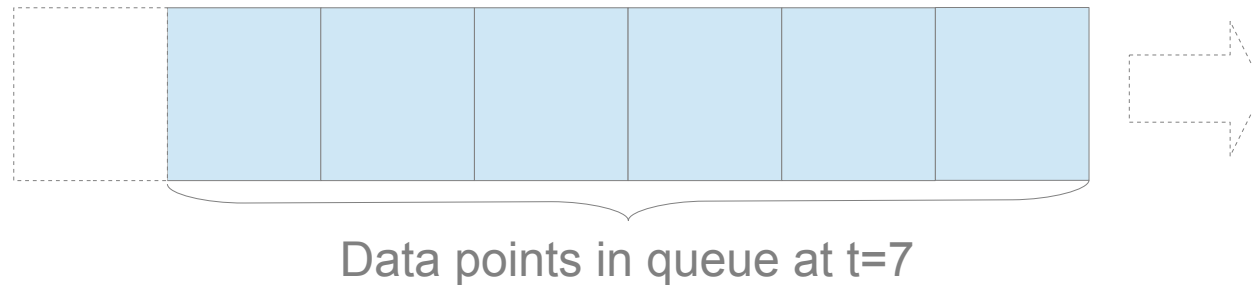
- Sample standard deviation (version 1)

$$s = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2}, \quad \text{where } \bar{x} = \frac{1}{N} \sum_{i=1}^N x_i$$

- Sample standard deviation (version 2)

$$s = \sqrt{\left(\frac{N}{N-1}\right) \left( (\bar{x^2}) - (\bar{x})^2 \right)}$$

# Adding and dropping returns



Discarded data point

Newly added data point

# First attempt at writing Stats class, *Stats1*

Data is passed into the constructor as an array

```
public Stats1( double[] values ) throws Exception {  
    if( values == null || values.length < 2 ) {  
        throw new Exception(  
            "Need at least two values to compute sample standard deviation"  
        );  
    }  
    _count = values.length;  
    _sum = 0;  
    for( int i = 0; i < _count; i++ ) {  
        _sum += values[ i ];  
    }  
    _mean = _sum / values.length;  
    double diffSqrSum = 0;  
    for( int i = 0; i < _count; i++ ) {  
        double diff = values[ i ] - _mean;  
        diffSqrSum += ( diff * diff );  
    }  
    _sampleStd = Math.sqrt( diffSqrSum / ( _count - 1 ) );  
}  
  
public double getSampleStd() {  
    return _sampleStd;  
}
```

An exception will be thrown if the number of data points is less than 2

The entire calculation takes place in the constructor, so all of the data must be passed to the constructor each time a point is added to or dropped from our analysis. This is not very efficient.

# Testing the Stats1 class

Test the three boundary conditions that may cause an exception to be thrown

```
public void testNullValues() {
    double[] values = null;
    Exception expectedError = null;
    try {
        new Stats1( values );
    } catch (Exception e) {
        expectedError = e;
    }
    assertTrue( expectedError != null );
}

public void testLessThan2Values() {
    double[] values = { 1 };
    Exception expectedError = null;
    try {
        new Stats1( values );
    } catch (Exception e) {
        expectedError = e;
    }
    assertTrue( expectedError != null );
}

public void test2Values() {
    double[] values = { 1, 2 };
    Exception expectedError = null;
    try {
        new Stats1( values );
    } catch (Exception e) {
        expectedError = e;
    }
    assertTrue( expectedError == null );
}
```

# Testing the Stats1 class (contd.)

```
public void test30Values() {
    double[] values = new double[ 30 ];
    for( int i = 0; i < 30; i++ ) {
        values[ i ] = i;
    }
    boolean noError = true;
    Stats1 stats1 = null;
    try {
        stats1 = new Stats1( values );
    } catch (Exception e) {
        noError = false;
    }
    junit.framework.TestCase.assertTrue( noError );
    double s = stats1.getSampleStd();
    /* Note - This is R code that was used to check the calculation
       > sd(1:30)
       [1] 8.803408
       >
    */
    double tolerance = 0.0001;
    double valueObtainedFromRCalculation = 8.803408;
    junit.framework.TestCase.assertEquals( s, valueObtainedFromRCalculation, tolerance );
}
```



Test the calculation itself

# Why is Stats1 bad?

- It is inefficient to perform the whole calculation from scratch every time we add a data point and drop a data point
- Instead, we can use the second version of the calculation for sample standard deviation

$$s = \sqrt{\left(\frac{N}{N-1}\right) \left( \sum x_i^2 - \frac{(\sum x_i)^2}{N} \right)}$$

- The above tells us is that if we maintain the following three values, we can incrementally add and remove data points from our analysis

$$\sum_{i=1}^j x_i^2, \quad \sum_{i=1}^j x_i, \quad j$$

Suppose we want to drop data point  $z$  and add data point  $a$ . We update our analysis as follows.

- $\text{sumXX} -= z * z$
- $\text{sumX} -= z$
- $\text{count}--$
- $\text{sumXX} += a * a$
- $\text{sumX} += a$
- $\text{count}++$

The sample standard deviation is then computed on the fly using the previously shown formula:

```
double diffOfSums = ( _sumXX / _count ) - ( ( _sumX / _count ) * ( _sumX / _count ) );  
return Math.sqrt( diffOfSums * ( (double)_count / ( _count - 1 ) ) );
```



We will now write a second class, Stats2 as an example of the principle of encapsulation

- Its interface to the outside world will be simple
- Its internal state will not be visible to the outside world
- Though there will be some complexity in managing its internal state, the outside world will not have to be concerned with that complexity

# Stats2 – Constructor and *add* method

```
protected double _sumX;  
protected double _sumXX;  
protected int    _count;
```

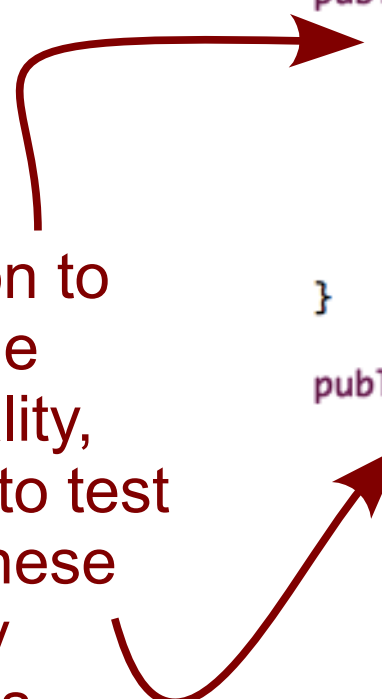
```
public Stats2() {  
    _sumX = 0;  
    _sumXX = 0;  
    _count = 0;  
}
```

```
public void add( double value ) {  
    _sumXX += value * value;  
    _sumX += value;  
    _count++;  
}
```

This is simple to write and simple to test because it can not throw an exception

# *getMean* and *getSampleStd* methods

In addition to testing the functionality, we want to test around these boundary conditions

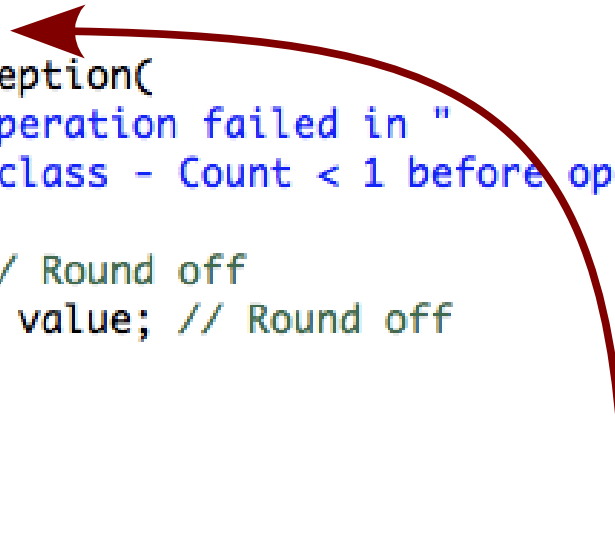


```
public double getMean() throws Exception {
    if( _count < 1 )
        throw new Exception(
            "Invalid count in stats mean "
            + "calculation = (" + _count + ")"
        );
    return _sumX / _count;
}

public double getSampleStd() throws Exception {
    if( _count < 2 )
        throw new Exception(
            "Invalid count in stats sample "
            + "std calculation = (" + _count + ")"
        );
    double diffOfSums =
        ( _sumXX / _count )
        - ( ( _sumX / _count ) * ( _sumX / _count ) );
    return Math.sqrt(
        diffOfSums
        * ( (double)_count / ( _count - 1 ) )
    );
}
```

# *remove* method

```
public void remove( double value ) throws Exception {  
    if( _count < 1 )  
        throw new Exception(  
            "Remove operation failed in "  
            + "stats class - Count < 1 before operation"  
        );  
    _sumX -= value; // Round off  
    _sumXX -= value * value; // Round off  
    _count--;  
}
```



Obviously, we want to test for count = 0, 1, and 2. But what other conditions should we test? Hint: We have a subtle, untested condition that, if it ever occurs, should throw an exception.

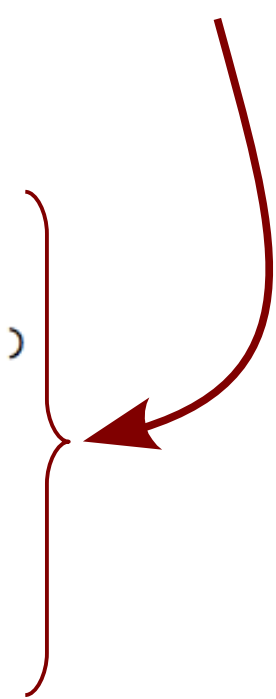
# How do we know that we are removing only what was added?

- What if, after we remove the effect of the last data point and the count is 0, sumX and sumXX are not zero?
- Possible causes
  - What we removed from our analysis was not properly added to our analysis
  - We have a round-off error
- Solutions
  - Simple but incomplete: Throw an exception
  - Not so simple but complete: Re-design the class to keep its own queue. This would absolutely prevent such errors but at the cost of a bit of data duplication

# Problem: How do we know that we are removing only what was added?

```
public void remove( double value ) throws Exception {  
    if( _count < 1 )  
        throw new Exception(  
            "Remove operation failed in "  
            + "stats class - Count < 1 before operation"  
        );  
    _sumX -= value; // Round off  
    _sumXX -= value * value; // Round off  
    _count--;  
    if( _count == 0 ) {  
        if(  
            ( ! DoubleComparator.equal( 0.0, _sumX, _tolerance ) )  
            || ( ! ( DoubleComparator.equal( 0.0, _sumXX, _tolerance ) ) )  
        ) {  
            throw new Exception(  
                "Empty stats object has non zero sums"  
            );  
        }  
        _sumX = 0.0;  
        _sumXX = 0.0;  
    }  
}
```

Check for round-off error  
or improper removal of  
data



There is no default value for tolerance, so we have to ask for it in the constructor

```
public Stats2( double tolerance ) {  
    _sumX = 0;  
    _sumXX = 0;  
    _count = 0;  
    _tolerance = tolerance;  
}
```

Blech! Things are already getting ugly. We need an extra variable just to alert users that there's an error that – if we had the right design – we could prevent.

# What we need is another class that combines a data queue with a Stats class

- This way, Stats will manage its own data and can prevent mistakes of erroneously removing data from the analysis
- We could incorporate that functionality into this version of the Stats class, but if we can envision a different use for Stats – one in which there won't be a queue of data – it's best to leave this class alone
- We'll build another Stats class, Stats3, that will inherit all of the functionality of Stats2, but will also incorporate a queue of data
- We want to define the functionality of this queue in the broadest terms, so we write an interface for it



First, we will write an interface for our StatsQ class that will illustrate the principle of polymorphism

- Polymorphic behavior is the idea that a class can work with the StatsQInterface without knowing which class it is really working with
- In other words, the underlying queue may be a linked list, a circular list, or a database connection. It doesn't matter. What matters is that this class implements the StatsQInterface, which makes it work with any other class that understands the StatsQInterface
- As previously mentioned, we can also implement polymorphism through virtual methods, but that's a subject for another day

This is the interface for our StatsQ class

```
public interface StatsQInterface {  
  
    public void    addLast( Double dataPoint );  
    public Double  removeFirst();  
    public boolean isReady();  
    public boolean hasElementsToRemove();  
  
}
```

# We will now write the StatsQ class as a means of demonstrating the principle of inheritance

```
public class StatsQ extends LinkedList<Double> implements StatsQInterface {  
  
    private static final long serialVersionUID = 1L;  
  
    @Override  
    public boolean isReady() {  
        return super.size() == 900;  
    }  
  
    @Override  
    public boolean hasElementsToRemove() {  
        return super.size() > 900;  
    }  
  
}
```

Inherit functionality

Facilitate polymorphism

There is almost nothing to write! We inherit all of the functionality from the Java collections library's LinkedList class. All we have to do is implement the two missing methods of our StatsQInterface. Clearly, in this case, inheritance works. Can we also apply inheritance to our Stats class?

We try to write *Stats3* using inheritance

See *Stat3.java*

..but there's a problem

# The *remove* method is the problem

- We have to implement it or the remove method from the parent class with the same signature will be visible to the outside world.
- However, all of the addition and removal from the queue is now performed inside a Stats3 object, where all of the calculations are updated with each addition or removal. We don't want the outside world to be able to mess up these calculations.
- So we write this method to throw an exception if it is called by the outside world, but this error won't show up until run time, not when the code is first written:

```
public void remove( double value ) throws Exception {  
    throw new Exception(  
        "Can't remove a value from this object"  
        + " - values are removed automatically" );  
}
```

**UGLY!**

# Conclusion of our inheritance experiment

- Using inheritance saved us a lot of time with StatsQ
- Using inheritance didn't work so well with Stats3
  - To begin with, Stats2 had very little functionality, so we're not saving a lot by deriving Stats3 from Stats2
  - On top of that, we don't want to expose the *remove* method of Stats2 so we had to build something ugly to hide it
- Now, we will try to solve the same problem without inheritance by building a Stats4 class. The strategy here is for a Stats4 object to contain a Stats2 object and a StatsQ object.

We try to write *Stats4* without inheritance

See *Stat4.java*

# *Stats4* is better than *Stats3* but...

- What if the queue that we pass into the Stats4 class constructor is modified outside of the object that is using it?
- We would get an error because data points would not be properly removed
- To get around this, we would need to have Stats4 create the queue internally, but that would mess up this beautiful idea that any queue that implements the StatsQInterface can be used with Stats4
- To be continued...