CS 241 — Introduction to Problem Solving and Programming

Object-Oriented Programming

Interfaces

Mar 21, 2005

Static

So what is static?

If a member of a class is `static`, then it belongs to a class, rather than an instance; all instances of the class share the same one.

```
public class C {

     int value;                     // Each instance has its own
     static int accumulator         // Single variable shared by all instances

}
```

Static

A static method cannot refer to the class's instance variables (including `this`) or invoke a non-static method (except using dot notation on an instance of the class).

```
public class D {

     int x;
     static int y;

     int xTimes5() { return x * 5; }

     static int crazyMethod() {
          int z = x - 10;                // error
          return z + y / xTimes5();    // error
     }
}
```

Static

Call static methods from outside the class by using dot notation on the class name (not on an instance).

```
public class C
    static int mult5(int x)  return x * 5;



...

    int y = C.mult5(12);
```

Static

Some statics we've seen. . .

- Any `main` method.
- `System.out.println()`, where `out` is a static field of the `System` class (it is an instance of class `PrintStream` which has a method `println()`).
- `DocsIO.readint()` and friends, where `readint()` is a static method of the class `DocsIO`.

Static

Example use of `static`: giving every instance a uniqe id #

```
public class Book {

     private static int currentId;
     private int uniqueId;

     Book() {
          uniqueId = currentId++;
     }
```

Static

The `Math` class has these `static` methods:

```
double Math.pow(double, double)          int Math.abs(double)
int Math.round(double)                   double Math.sqrt(double)
int Math.floor(double)                   int Math.ceil(double)
```

See pg 280.

Back to Friday

Recall the set of measur(e)ments example. . .

Classes have class invariants specifying what conditions should be preserved by all methods of the class (except the constructors, which set those conditions).

On the method level, we have pre- and post-conditions.

Postcondition

```
/**
 * Add a new measurement to the set.
 * Create a new array of larger size; copy all the old elements into
 * the new one; place also the new measurement into the new
 * array; finally, set measurements to refer to the new array.
 * POSTCONDITION: The array measurements has been replaced
 * by one of size one greater, with the old elements and the new one
 * @param measurement the measurement to add to the collection
 */
public void add(double measurement) {
    // To hold the new array
    double[] newMeasurements = new double[measurements.length + 1];
    for (int i = 0; i < measurements.length; i++)
        newMeasurements[i] = measurements[i];
    newMeasurements[measurements.length] = measurement;
    measurements = newMeasurements;
}
```

Correction

Mistake in the handout:

```
public void remove(int position) {
    // To hold the new array
    double[] newMeasurements = new double[measurements.length - 1];
    for (int i = 0; i < position; i++)
        newMeasurements[i] = measurements[i];
    for (int i = position + 1; i < measurements.length; i++)
        newMeasurements[i - 1] = measurements[i];
    measurements = newMeasurements;
    recalculateStats();
}
```

Measurements

We have also had several decisions to make about the implementation.

```
public Measurements1(double[] initials) {
    measurements = new double[initials.length];
    for (int i = 0; i < initials.length; i++)
        measurements[i] = initials[i];
}

public Measurements2(double[] initials) {
    measurements = initials;
    recalculateStats();
}
```

Measurements

Keeping values like max, min, and average instead of recalculating added to our class invariant:

```
public class Measurements2 implements Measurements {
    double[] measurements;
    double max;
    double min;
    double average;
```

This also adds to our vulnerability. . .

Sermon on Encapsulation

We've seen the use of modularity/encapsulation for reusability

It is also important for correctness.

To determine that our variables are used correctly, it helps to isolate where the variables are used.

Accessibility

Members of a class can be given an accessibility level: `public` or `private`.

- An instance variable or method that is designated `public` can be accessed (read or written to; for instance variables) or invoked (for methods) by code in another class.

- An instance variable or method that is designated `private` can be be accessed (for instance variables) or invoked (for methods) only by code in the same class.

Accessibility

```
public class Measurements2 implements Measurements {
    private double[] measurements;
    private double max;
    private double min;
    private double average;
```

We now can guarntee the instance variables cannot be read from or written to. Any attempt would generate a compiler error:

```
MeasurementsDriver.java:4: average has private access in Measurements2
        mm.average = 5;
```

Accessibility

Why is there a `public` access modifier? Aren't members public if they are not declared `private`?

No, there is a subtle difference. The default access modifier is package-scoped. For our purposes, we won't notice a difference, but it's good style to make everything that can be used elsewhere `public`.

It's good progamming practice to make all instance variables `private`.

Getter methods

Notice all these methods in Measurements2 do is read the instance variables.

```
/**
 * Compute the average measurement.
 * Getter method for instance variable average.
 * @return the mean measurement
 */
public double average() { return average;   }

 public double max() {  return max; }

 public double min() { return  min; }
```

Such methods are called getter methods or accessor methods.

Setter methods

I want client code to be able change the value of an instance variable. How can I do that without making it public?

Use setter methods (or mutator methods).

```
public class X {
      int y;
      void setY(int y) {
            this.y = y;
      }
}
```

Getters and Setters

I've defined a getter method and a setter method for an instance variable. Doesn't that in practice make it public? Why should I still declare it `private`?

Getters and Setters

I've defined a getter method and a setter method for an instance variable. Doesn't that in practice make it public? Why should I still declare it `private`?

- Debugging
- Preparing for changes you might make later (client code should depend on the methods of the class, not instance variables.

More on reason #2 later . . .

Accessibility

Methods, too, can be private, and sometimes should be.

```
private void recalculateStats() {
    double sum = max = min = measurements[0];
    for (int i = 1; i < measurements.length; i++) {
        sum += measurements[i];
        if (measurements[i] < min)
            min = measurements[i];
        if (measurements[i] > max)
            max = measurements[i];
    }
    average = sum / measurements.length;
}
```

Encapsulation sermon, part II

Class invariants and private instance variables help us make guarantees about the correctness of a module or component (classes, methods, . . .).

We can reason about the correctness of interaction between models by thinking in terms of contracts.

A method signature and documentation describe the contract.

Programming by contract

A general description of the transaction.

```
/**
 * Add a new measurement to the set.
 * Create a new array of larger size; copy all the old elements into
 * the new one; place also the new measurement into the new
 * array; finally, set measurements to refer to the new array.
 * POSTCONDITION: The array measurements has been replaced
 * by one of size one greater, with the old elements and the new one
 * @param measurement the measurement to add to the collection
 */
public void add(double measurement) {
```

Programming by contract

The client's side of the agreement.

```
/**
 * Add a new measurement to the set.
 * Create a new array of larger size; copy all the old elements into
 * the new one; place also the new measurement into the new
 * array; finally, set measurements to refer to the new array.
 * POSTCONDITION: The array measurements has been replaced
 * by one of size one greater, with the old elements and the new one
 * @param measurement the measurement to add to the collection
 */
public void add(double measurement) {
```

Programming by contract

The method's side of the agreement.

```
/**
 * Add a new measurement to the set.
 * Create a new array of larger size; copy all the old elements into
 * the new one; place also the new measurement into the new
 * array; finally, set measurements to refer to the new array.
 * POSTCONDITION: The array measurements has been replaced
 * by one of size one greater, with the old elements and the new one
 * @param measurement the measurement to add to the collection
 */
public void add(double measurement) {
```

Programming by contract

Not part of the contract.

```
/**
 * Add a new measurement to the set.
 * Create a new array of larger size; copy all the old elements into
 * the new one; place also the new measurement into the new
 * array; finally, set measurements to refer to the new array.
 * POSTCONDITION: The array measurements has been replaced
 * by one of size one greater, with the old elements and the new one
 * @param measurement the measurement to add to the collection
 */
public void add(double measurement) {
```

Two versions

What do `Measurements1` and `Measurements2` have (completely) in common?

Two versions

What do `Measurements1` and `Measurements2` have (completely) in common?

Not instance variables. . .

Not algorithms. . .

Not even private methods. . .

Two versions

`Measurements1` and `Measurements2` share a common set of public methods and contract with client code.

We call this the interface of these classes.

Interfaces

Java has a construct for declaring an interface for classes to implement.

```
public interface Measurements {
    public int size();
    public void add(double measurement);
    public void remove(int position);
    public double average();
    public double max();
    public double min();
}
```

Interfaces

Document the contract, not the algorithm.

```
public interface Measurements {

    /**
     * Compute the size of this set of measurements
     * @return the number of measurements contained
     */
    public int size();
```

Interfaces

Classes then implement the interface.

```
public class Measurements1 implements Measurements {
```

Interface

Then clients can use the classes interchangeably.

```
Measurements meas;
if (DocsIO.readint("Use version 1 or 2?") == 1)
     meas = new Measurements1();
else
     meas = new Measurements2();
meas.add(DocsIO.readdouble("Next reading-->"));
```