CS100M

Introduction to Computer Programming

Spring 2004
Lectures 19-20
OOP

Announcements

- reminder: you are expected to be reading the textbook and practicing with small examples.... very important!
- A5 focus on objects, posted this Weds (waiting for labs to finish OOP), due 4/14
- T3: Tue 4/20
- final exam reminder (it's been posted all semester): 5/18

Overview

- OOP:
  - what is it?
  - what's an object? what's a class?
  - what are all these mysterious keywords?
- Goals:
  - learn how to define and use customized types
  - use these types to reduce more redundancy
  - create virtual objects that help design really cool programs
- First:
  - why OOP?
  - what happens if you use only primitive types....
  - example) program that organizes and searches for information in a database (students and GPAs)

Motivation

```
public class NonOOP {
    public static void main(String[] args) {

        // initialize information for 3 people:
        String name1 = "Dimmu";
        double gpa1 = 3.7;
        String name2 = "Dani";
        double gpa2 = 3.6;
        String name3 = "Shagrath";
        double gpa3 = 3.9;

        // Report most accomplished student:
        System.out.println( "Max Student is " +
            maxGPA(name1,gpa1, name2,gpa2, name3,gpa3) );
    }

    // Search for student with highest gpa and return name:
    public static String maxGPA(String s1, double g1,
                    String s2, double g2, String s3, double g3) {

        double max = g3;  // assume max gpa is 3rd gpa
        String s = s3; // name of student with max gpa
        if (g2 > max) { max = g2; s = s2; }
        if (g1 > max) { max = g1; s = s1; }
        return s+": "+max;
    }

}
```

Problems/Improvements

- What about more than 3 students?
  - could use something called ***data structures*** (collection of information, like arrays)
  - arrays are objects in Java, so we need to wait on those...)
- Variables?
  - Seems redundant to have repeated variable names (`name1`, `name2`, etc) (use arrays?)
  - Seems dangerous to use two variables to represent each student (a student has ___ attributes)
  - What about adding more attributes?
- max GPA method:
  - better way to write header?
  - better way to write return?

OOP Gist

- So...the previous example does work, but is crap
- OOP approach:
  - brainstorm and "research" (find **nouns**, **verbs**)
  - **verbs** become either
    - operators
    - methods
  - **nouns** become either
    - classes (code used to make objects)
    - fields (variables in classes)
    - parameters (variable in methods)
    - local variables (in methods)
    - constants (known values like numbers, chars, booleans)
- Quick overview of what's to come....

Vastly Improved Solution

- Revisit problem of finding "max student"
- Main Class:
  - create structure (array `s`) to hold student data
  - create each student and put in `s`
  - search for max student
- Code with all kinds of things (quick view!):

```
public class OOP1 {
    public static void main(String[] args) {

        Student[] s = new Student[3];      // what is this?
        s[0] = new Student("Dimmu",3.7);   // put object where?
        s[1] = new Student("Dani",3.6);
        s[2] = new Student("Shagrath",3.9);
        // even better idea: read info from file!

        System.out.println("Max Student is "+Student.maxGPA(s));
    }
}
```


example continued

- Syntax Reminder (from Java intro):

```
class name {
   fields (data)
   methods (actions)
}
```

- `Student` Class:
  - data:
    - each `Student` has its own name and GPA
  - actions/behaviors:
    - need a way to create each `Student`
    - need a way to "stringify" a `Student`
    - need a way to compare two `Student`s
    - need a way to find a max `Student`
- For now, *skim* rest of code (see `OOP1.java` on-line)

the code...

```
class Student implements Comparable {
    private String name;
    private double gpa;

    public Student(String n,double g) {
        name = n;
        gpa = g;
    }

    public String toString() {
        return name+": "+gpa;
    }

    public int compareTo(Object s) {
        double diff =  gpa - ((Student) s).gpa;
        if (diff < 0) return -1;
        else if (diff > 0) return 1;
        else return 0;
    }

    public static Student maxGPA(Student[] s) {
        Student max = s[0];
        for (int c=1; c<s.length; c++)
            if (s[c].compareTo(max) > 0) max = s[c];
        return max;
    }
}
```


Comparison of Approaches

- Without OOP:
  - all the data and actions "smushed" into Main Class
  - lots of redundancy
  - very algorithmic design, but we still needed to identify all data and actions to make variables and methods
  - need to rewrite whole program if definition of Student changes
- With OOP:
  - Main Class (the driver) becomes rather short, almost like English (easy to write, we *abstracted* away the details)
  - Dirty work moved into design (done earlier!) to decide on data and actions of other classes
  - can reuse classes for other programs! if **`Student`** changes, Main Class doesn't "care"
  - Write generic code! Saves time! Saves $$$!
- So, how do you write a class? do OOP?

Abstract Data Type

- Think about non-OOP types and values:

```
int x;
x = 1;
```

- Classes as types, objects as values....
  - Want to be able to create your own types for variables
  - examples of types NOT provide by Java:

```
Student s;
Person p;
Polynomial p;
Complex c;
```

  - Variables will eventually hold composite values and provide access to data and actions via ***objects***
- Examples:

```
Complex c1 = new Complex(1,3); // 1+3i
Complex c2 = new Complex(2,-1);
Complex c3 = c1.add(c2); // 3 + 2i
```


Classes/Objects

- ***Class***:
  - your defined type (***abstract data type***) (ADT)
  - holds data and actions related to that type
  - a blueprint → code that you use to create ***objects***
- ***Object***:
  - represents an actual thing
  - portion of memory that holds values and provides access to ADT's actions
  - effectively, the "value" you would assign to a variable declared with a class type
- Example of OOP types (classes) and values (objects):

```
Complex c1,c2;
c1 = new Complex(1,3);
c2 = new Complex(0,-5);
```

Classes/Objects continued

- To create your own types, write a class:
  - `class Complex { ... }` is a class we write
  - write fields and methods inside `Complex`'s body
  - need to write a bit more though...
- To create objects:
  - in class, write special methods called ***constructors***
    - "explain" to language what to do when creating an object
    - usually set fields and call methods in class
    - Syntax:
      `public classname(params) block`
  - to *call* a class's constructor (and create object), use syntax
    `new classname(arguments)`
  - eg: `new Complex(1,3)` creates an object in memory
  - object is created in memory when Java *runs* this code

Development Snapshot

- Class syntax

```
modifiers class name stuff {
   fields
   constructors
   methods
   morestuff
}
```

- Accessing object's fields and methods?

```
ClassType var;
var = new Type(args);

ClassType var = new ClassType(args);

var.method(args);
var.field;

new ClassType(args).member
```

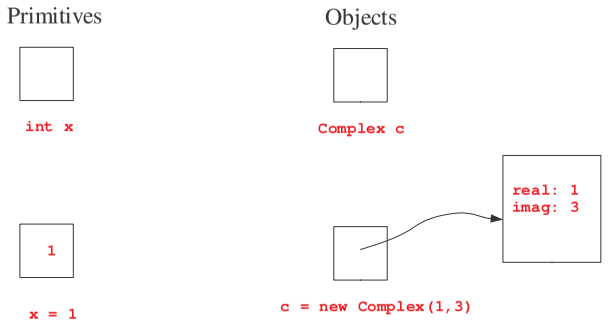

Brief Example

```
class Complex {

    // fields
    private double real; // real component
    private double imag; // imaginary component

    // constructor
    public Complex(double r, double i) { real = r; imag = i; }

    // methods
    public Complex add(Complex other) {
        return new Complex(real+other.real, imag+other.imag);
    }

    public String toString() {
        return real+"+"+imag+"i"; // what if imag is neg?
    }
}

public class TestComplex {
    public static void main(String[] args) {
        Complex c1 = new Complex(1,3);
        Complex c2 = new Complex(2,-1);
        System.out.println(c1.add(c2));
    }
}
```


Introduction to Object Address

- How does object reside in memory? (Savitch4.3):
- Think about primitive values: `int x; x=1;`
  - portion of memory reserved to store a value of type `int`
  - value `1` is created and stored in `x`'s memory location
  - if `x` is alive and visible, whenever you use `x`, you get `1`
- Objects: to create your own "values" (objects):
  - `new name(...)` creates an object and returns it's address
  - just like method that does actions and returns a value
  - you can store the return value (the address) in a variable that has the same type as the object you created
  - eg) `Complex c1 = new Complex(1,3);`

Brief Depiction



c stores the addres of the object in memory (object too large to "fit" in **c**'s location)

Class Syntax

- To make objects, we have to spend some time talking about classes
- Syntax

```
modifiers class name stuff {
   fields
   constructors
   methods
   morestuff
}
```

- Writing:
  - ideally, one class per file (make it **public**)
  - to put more than one class in a file, modify only *one* class as **public**
  - put all files in same directory for now
  - example) see `OOP1.java` from before

Example of Class Design

```
// Define stubs for classes and methods

// Driver class: A calculator for Complex numbers
public class ComplexCalc {

    public static void main(String[] args) {
        // examples
    }

    // other methods for calculator?

}

// Complex number has real and imag values:
class Complex {

    // fields
    // constructors
    // methods

}
```


How to "fill in" class?

- Need to write
  - fields
  - methods
  - constructors
- What becomes a class?
  - see next two panels
  - need to be a bit conceptual to help with design
- Example: want to be able to write something like:

```
Complex c1 = new Complex(1,3);
Complex c2 = new Complex(2,4);
System.out.println(c1.add(c2));
```

- Note:
  - dot operator (**.**)
  - used to access an object's member when member is visible

Has-A Relationships

- During brainstorming, design your classes by looking for
  - composite nouns/things in the problem brainstorm (become classes used to create objects)
  - attributes/data of the objects
  - actions/behaviors by/of the objects
- ***Has-A relationship***:
  - excellent rule-of-thumb for knowing when to write a class
  - if "thing" has parts/components/behaviors, you should likely model it with a class
- Examples:
  - a complex number has real and imaginary components and special arithmetic operations
  - a student has a name, age, ID, courses, grades, and a way to compute a GPA
  - a conveyor belt has items on it, workers adding/removing those items, a speed, and a way to start/top.

Encapsulation

- Class provide ***abstraction***:
  - see Non-OOP/OOP analysis
  - OOP puts details about objects in blocks of code that can be named very easily
  - provides higher-level algorithms for many methods
- ***Information hiding***:
  - explicit mechanism to hide details of code from other code
  - use modifiers, like public and private
- ***Encapsulation***:
  - effectively, the formal term for "has-a relationship"
  - process of deciding what elements belong to a class
  - provides abstraction and information hiding

Fields

- Field:
  - attributes/parts of an object
    - eg, **`Person`** has a name, ID, age, weight, gender, ...)
    - eg, **`Complex`** number has real and imaginary components
  - data shared by all methods in the class (scope again!)
- Syntax:

```
modifiers type var ;
modifiers type var = expression ;
```

- So,
  - fields resemble local variables in a method
  - difference is that the class scope makes the fields visible to every method in the *same* class

More Fields

- Visibility to outside class? Usually **`private`**:
  - prevents outside class from changing (very good style for fields)
  - if **`public`**, object allows it's fields to be changed (dangerous)
- Default values of fields:
  - unlike local variables and params (unknown defaults), fields get values of "zero"
  - eg) integers: 0, doubles: 0.0, boolean...?, objects...?
- Fields are given defaults and then assigned *top-down* as before code inside constructor executes:

```
public class Blah {
    int x;
    int y=x+1;
    public Blah() {
        x = 3;
        System.out.println(y);
    }
}
// what outputs if you create an object Blah b=new Blah()?
```

Development continued

```
// fields needed in main class? not for now
// Driver class:
public class ComplexCalc {
    public static void main(String[] args) {
        // can't really use Complex yet
    }
}

// fields for Complex class?
// Complex number has real and imag values:
class Complex {

    private double real; // real component
    private double imag; // imaginary component

    // constructors
    // methods

}
```


Constructors

- Primary rules (modified later for inheritance):
  - every class must have at least one constructor
  - syntax reminder:
    `public classname(params) block`
  - if you do *not* write a constructor, Java automatically provides the ***empty constructor***:
    `public classname() { }`
  - if you *do* provide a constructor, Java does *not* provide the empty constructor
  - constructors return the address of a newly created object, but you do *not* explicitly write a return type
  - because you can have multiple constructors, constructors follow rules of overloading
  - how to call a constructor from another constructor:
    `this(params);`

Development Continued

```
// Driver class:
public class ComplexCalc {
    public static void main(String[] args) {
        Complex c1 = new Complex();
        Complex c2 = new Complex(1,3);
    }
}

// fields for Complex class?
// Complex number has real and imag values:
class Complex {

    private double real; // real component
    private double imag; // imaginary component

    // Create a default complex number (0 components):
    public Complex() { }

    // Create a complex number with R and I components:
    public Complex(double r, double i) {
        real = r;
        imag = i;
    }

    // methods

}
```


Methods

- Reminder:
  - all methods go inside a class
  - see rules of methods for "seeing" each other
  - see other rules fom Methods lecture
- Now trying to avoid **`static`**... why?
  - **`static`** members provide non-OOP programming (can access information without objects...use class name to access)
  - **`static`** usually for
    - field is shared by all objects (**`Math.PI`** the same for everyone)
    - method should affect all objects (entire company gets a 3% raise)
  - more about **`static`** later

Special Methods

- OOP and inheritance:
  - as usual, we have abstracted some details from you
  - one cool detail is inheritance
  - classes can use code from other classes without having to literally rewrite that code
- package **`java.lang`**
  - all Java code automatically "knows" about java.lang
  - see class **`Object`** in **`java.lang`** (Java API link online)
  - **`Object`** has several methods that all classes automatically define
  - you may redefine the meaning of any of these methods in your class

Primary Example

- method **`toString`**:
  - all classes have a **`toString`** method
  - syntax:
    ```
    public String toString() block
    ```
  - returns a stringified description of the object
- how used: promote to a **`String`**:
  ```
  Complex c = new Complex(1,2);
  System.out.println("my num: "+c);
  System.out.println(c);
  ```
- problem
  - by default, **`toString`** returns the address of the object in memory (not particularly useful)
  - to be useful, you write your own version!

Development continued

```
public class ComplexCalc {
    public static void main(String[] args) {
        // examples
    }
}

class Complex {
    private double real; // real component
    private double imag; // imaginary component
    public Complex() { }
    public Complex(double r, double i) { real = r; imag = i; }

    // Add two complex numbers together (one approach):
    public Complex add(Complex other) {
        return new Complex(real+other.real, imag+other.imag);
    }

    // Alternative approach:
    public static Complex add(Complex c1, Complex c2) {
        return new Complex(c1.real+c2.real,c2.real+c2.imag);
    }

    // Stringify complex number:
    public String toString() {
        return real+"+"+imag+"i"; // what if imag is neg?
    }
}
```


Did you notice?

- operator overloading
- using an ADT not only for assigning variables in another class, but using it as part of itself
  - passing addresses of objects to methods
  - returning addresses of newly created objects
  - more to be discussed!
- dot operator used to access object's fields (dot operator also for accessing object's methods)
- objects can see each other's members when objects are created from same class

Driver

- Main Class:
  - all classes may contain a **`main`** method
  - from command-line, you decide which class's **`main`** method you will call...example)
    ```
    > java Blah
    ```
  - so, the class **`Blah`** must have been compiled and it must contain a **`main`** method with the proper syntax
- Main Class serves as driver class:
  - set up program
  - high-level abstraction
  - ***driver***: calls all the methods and classes that do the "dirty work"
  - often very short

Development continued

```
public class ComplexCalc {
    public static void main(String[] args) {
        Complex c1 = new Complex();
        Complex c2 = new Complex(1,3);
        Complex c3 = c1.add(c2);
        System.out.println(c2.add(c1));
        System.out.println(Complex.add(c1,c3));

        // Can you do this?
        // c1.real = 4;

    }
}
```


More To Come

- Notion of reference
  - when you create an object, the value of the address is returned
  - you can never use that value specifically
  - reference does allow you to access an object's members
  - use of **`this`** quite interesting
  - passing references to methods
- **`static`**
- arrays
- inheritance