CS193J: Programming in JAVA

January 13, 1999

Lecture #3

Admin

- Need to register with Webclass by today!
- This Friday's section will be usurped for lecture
  - To avoid slow start from MLK holiday
  - Normal section resumes next Friday

From last time: OOP

- Many small improvements C to Java, but also a big paradigm shift
- Program organized around data
- Object = data + behavior
- Composition and inheritance allow reuse of objects
- OO program is a set of cooperating objects, messaging each other

Advantages of OOP

- Objects tend to map to real world entities
- Improved reliability

  Fewer project interdependencies
- Supports teamwork

  Clear interface/implementation boundaries
- More opportunity for code re-use
- Easier code maintenance
- Enables object-based testing

Basic OO idea

- Look at problem, decompose into objects
- Try to use off-the-shelf components where possible
- Arrange classes in tree for maximum code reuse
- Wire up objects to communicate with one another

OO vocabulary

- Class

  A data type which describes the pattern of data and behavior

- Object or instance

  A variable of a particular class

- Messaging

  We say that you "send a message" to ask object to perform an action

- Method

  "function" for a class, it is the code body that implements a message

Defining a class

- Time class is defined in Time.java file
  - Class name must match filename, case matters
  - Class definition cannot be split among multiple files
  - Only one public class defined per file
- Class file is both interface and implementation
  - No separate header files, unlike C
- Instance variables
  - Fields or data in the "object struct"
- Instance methods
  - Object operations, functions that are strongly tied to class
- Order of definition doesn't matter
  - Can refer to variables/methods that appear lexically afterwards

Class example

```
public class Time {

   private int hour, minute;

   public int getHour() {  // note: no void arg list
      return hour;
   }
   public void setHour(int newValue) {
      hour = newValue;
   }
   public int getMinute() {
      return minute;
   }
   public void setMinute(int newValue) {
      minute = newValue;
   }
}
```

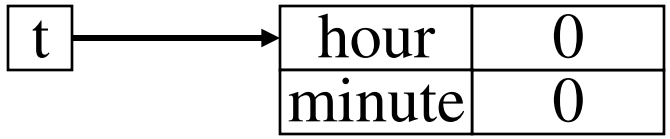
Access control

- Each variable and method has an access specifier
- public
  - Accessible to everyone, both designer and users of the class
  - Public instance variables frowned upon
- private
  - Accessible only within class implementation, only class designer
- protected
  - Similar to private, but also visible in subclasses (classes that are designed based on this one)
- default when you list no keyword is "friendly"
  - Public within this package, but private outside it

Storage for objects

- A Time object has two instance variables
- Object about same size as comparable structure
- Creating a Time object allocates the space

```
Time t = new Time();
```



- Each Time object has its own distinct copies of the instance variables
- When messaging an object or accessing fields, you are referring to a particular object's copy of the fields

Sending a message

- Public members accessed using ".", similar to accessing a field within a C struct
- Using a public variable

```
Time t = new Time();
t.hour = 3;    // allowable if hour is public
```

- Sending a public message

```
Time t = new Time();
t.setHour(3);
System.out.println("The hour is " + t.getHour());
```

  - Object you are messaging is called the "receiver"
- Accessing non-public variable/method outside class will raise a compiler error

Implementing a method

- Methods are implemented as part of class definition
- Within a method, all variables/methods are visible

```
public void shiftBy(int deltaHr, int deltaMin) {
    hour += deltaHr;
    minute += deltaMin;
}
```

- Can send receiving object messages

```
public void shiftBy(int deltaHr, int deltaMin) {
     setHour(hour + deltaHr);
     setMinute(minute + deltaMin);
}
```

- Access receiving object using hidden variable "this"

```
public void clear() {
    this.hour = 0;
    this.setMinute(0);
}
```

Maintaining object consistency

- Setters can constrain to correct range

```
public void setHour(int newValue) {
      if (newValue < 0) hour = 0; // constrain
      else if (newValue > 23) hour = 23
      else hour = newValue;
}

public void setMinute(int newValue) {
      minute = newValue % 60; // wrap into range
}
```

- What if the instance variables were public?
- What is the advantage of making all accesses, even inside implementation, go through setters?
- Should the setters themselves be public?

Object allocation

- Newly declared variable starts as a null reference

```
Time t;                    //t has value null
```

- Use new to create new object

```
t = new Time();            //t now refers to a new Time instance
```

  Fields in a new object are zeroed by default

- Objects are implemented as references

  Like arrays, these are "pointers in disguise"

- Assignment between objects doesn't make a "deep copy", it makes another reference to same object

```
Time u = t;
u.setHour(10);
System.out.print("The hour is = " + t.getHour());
```

Constructors

- Special method to initialize newly created object
- Part of new operator, so allocation and initialization are always done together
- Constructor must have exact same name as class
- Has no declared return type and can't return anything
- Access control applies as for other methods

  What would happen if constructor were declared private?
- Constructors can have parameters and can be overloaded

Constructor example

- Add a constructor to Time class

```
public Time(int hr, int min)
{
    hour = hr;
    minute = min;
}
```

- Can message receiving object, even in constructor

```
public Time(int hr, int min)
{
    setHour(hr);
    setMinute(min);
}
```

- Now give arguments to new which calls constructor

```
Time t = new Time(9, 15);
```

Cleaning up

- Java doesn't quite have analog to C++ destructors
- Garbage collector takes care of deallocating storage
- What if you have more cleanup to do?
- finalize method called when storage reclaimed

```
protected void finalize(){
    System.out.println("I'm outta here");
}
```

- Not guaranteed to be called in timely manner

  Not guaranteed to be called at all, actually!
- If you need guaranteed cleanup, add own method and call explicitly yourself

Overloading and constructors

- Can overload ctor, like any other method
- Constructor can chain to other constructor

```
public Time(int hr) {
    this(hr, 0);   // syntax a little weird
}
```

- Resolves best/closest match to how call is made

```
t = new Time(3, 15);        // 2-arg version
t = new Time(5);            // 1-arg version
t = new Time();             // doesn't compile!
t = new Time("3:00");       // nope either!
```

- Default constructor, only if no ctors defined at all

  Default zero-arg constructor that just clears (zeros) all instance variables (or sets to specified default values)

Field initialization, default values

- Can assign default values when defining ivars

```
public class Time {
    private int hour = 12, minute = 30;
```

- Doesn't have to be constant expression

```
private int hour = Math.random()*12;
```

- This initialization takes place before constructor

  Replaces initialization to zero for default constructor

- Assignment happens in order of declaration
- Any advantage over assigning in constructor?

The final modifier for variables

- final makes a variable "read-only"
  - Variable is assigned once and cannot change value after that
  - Used to define constants
  - Could add final variables to the Time class for constrains

```
public class Time {
  private final int MinHour = 0, MaxHour = 23;
  private int hour, minute;
  ...
```

- This means each Time object now has 4 variables
  - Two for its own hour and min, and two more for final variables
  - It would be better to share these last two values for all Time objects instead of making individual copies for each

Static final variables

- static applied to a variable makes it "class-wide"
  Only one piece of storage for all instances , shared by all
- Class constants done with static and final together

```
public class Time {
  private static final int MinHr = 0, MaxHr = 23;
  private int hour, minute;
```

- This is more efficient way to organize the data
  Each Time object has only 2 variables, class variables shared
- Also used to build enum-like entries

```
public class Time {
   public static final int AM = 0, PM =1;
```

  Accessed outside like this: **`Time.AM`**
- Is it safe/good for anything final to be public?

The static modifier for variables

- Static is one-per-class instead of one-per-instance
- Access specifiers still apply

  public static variables accessible to all, private only inside class
- Not all static variables must be final

  Example: class variable tracks count of objects created

```
public class Time {
    public static int numObjectsCreated = 0;
    public Time(int hour, int minute) {
        numObjectsCreated++;  //increment count in constructor
        ...
```

- Access field of class itself, instead of instance

```
System.out.print("Num " + Time.numObjectsCreated);
```

The static modifier for methods

- static keyword makes a method class-wide
- Example: create accessor for private static data

```
public class Time {
    private static int numObjectsCreated = 0;
    static public int numCreated() {
      return numObjectsCreated;
    }
```

- Send message to class, not to instance

```
System.out.print("Num " + Time.numCreated());
```

- Static methods used as substitute for functions

  Math.random(), Math.abs(), Character.isLowerCase(), etc.

More on static methods

- Static methods have no "this"

  Within static method, cannot access instance variables or call instance methods

- Instance methods can access static, though

  Within instance method, can call static methods, access static vars

  Can't declare instance and class variable/method of same name

  When field accessed without explicit receiver, assumed to mean "this" or "thisClass"

```
Time t = new Time();
int count = t.numCreated(); // ok
int hour = Time.getHour();  // not ok
```

Singleton design pattern

- Only want one instance ever created, shared by all

```
public class Clipboard {
    private Clipboard() {      // private ctor
          ....
     }
    private static Clipboard sharedInstance;
    public static Clipboard getSharedInstance()
    {
        if (sharedInstance == null)
            sharedInstance = new Clipboard();
        return sharedInstance;
    }
}
```

- Clients use static method to get shared instance

```
Clipboard clip = Clipboard.getSharedInstance();
```

Another static final example

- Want each instance assigned a unique id, start from 1

```
public class Time {
    private static int nextID = 1;
    private final int myID = nextID++;
```

could also have assigned myID in ctor, no big difference

- When object created

Assigns its own id number (final so cannot be changed)

Increments class variable for next one

Compile-time type checking

- Java is considered a strongly-typed language
  - All variables, parameters must declare type (int, String, Time, etc.)
- Can use variable only in ways that respect type
  - e.g. can't divide Time by float or assign String into int
- Can't message an object with wrong message
  - String has length method, Time does not

```
Time t = new Time();
System.out.println(t.length());
```

  - This produces **compile-time** error
- All object expressions type-checked
  - Both accessing a variable or sending a message
  - Variable or method must exist, be accessible, and match type usage

Reminders

- Must register with Webclass by today!
- Reading on common Java built-ins

  Ch 13-15 sections on String, primitive wrapper classes, Vector, Hashtable, Date/Calendar