CS193J: Programming in Java
Summer Quarter 2003

Lecture 2
OOP/Java

Manu Kumar
sneaker@stanford.edu

Handouts

- 3 Handouts for today!
  - #5: Java 3
  - #6: OOP Design
  - #7: HW1: Pencil Me In
- Continue handout #4 from lecture
- Logistics
  - July 3rd class show of hands

Recap

- Last Time
  - Course Introduction
  - Student Introductions
  - Introduction to Java
  - OOP concepts
- To Dos
  - Write a HelloWorld program in Java, compile it and run it on Leland machines.
  - SCPD students: email introductions

Q&A and Updates

- Link to HTML tutorials on course web page
- Link to OOP presentation on course web page
- Link to slides on course web page
- Link to lecture archives on course web page
- Pointer to HW submission instructions included in HW handout
- Smallest Java Virtual Machine
  - K VM from Sun
    - http://java.sun.com/products/cldc/ds/
  - 50-80KB in its smallest configuration

Today

- OOP in Java (Student Example)
- Explore more Java features
  - Primitives
  - Arrays
  - Multi-Dimensional Arrays
  - String Class
  - StringBuffer Class
  - Static keyword
- OOP Design
  - Encapsulation
    - Interface vs. Implementation
  - Client Oriented Design
- HW1: Pencil Me In
  - Due before midnight Wednesday July 9th, 2003

OOP in Java (Handout #4)

- Java is fundamentally Object-Oriented
  - Every line of code you write in Java must be inside a Class (not counting import directives)
- Clear use of
  - Variables
  - Methods
- Re-use through “packages”
- Modularity, Encapsulation, Inheritance

Student Java Example

- Complete code and explanation provided in handout
- First some designations we will use for this section
  - The person who writes the inner implementation of the class is the “programmer”
  - The person who “uses” the class is the “client”
    - The client cares about the interface exported by the Class/Object
- Analogy
  - Implementing an ATM machine vs. using an ATM machine

Implementation vs. Interface

- Implementation
  - Data structures and code that implement the features (variables and methods)
  - Usually more involved and may have complex inner workings
  - The guts of the black box
- Interface
  - The controls exposed to the “client” by the implementation
  - The knobs on the block box

Student Client Side

- Plan
  - Allocate objects with "new" -- calls constructor
  - Objects are always accessed through pointers
    - shallow, pointer semantics
  - Send messages
    - methods execute against the receiver
  - Can access public, but not private/protected from client side

Object Pointers

- The declaration:

  Student bart;

  Declares "bart" as a pointer to an object of class Student. It does not allocate the object
- Object is allocated by calling "new"

  bart = new Student();
- Object pointers are "shallow"
  - Using = (assignment) on a pointer, copies the value so that two pointers may be pointing to the same object
  - Using == (equals) on a pointer simply compares pointers and does not check if the objects are the same internally

Constructor

- Every class has a default “method” called a Constructor
  - Invoked when the object is to be “created” / “allocated” by using “new”
- A class may have multiple constructors
  - Distinguished at compile time by having different arguments
  - The default constructor takes no arguments and is implicit when no other constructors are specified

Invoking methods (sending messages)

- bart.getUnits();
- bart.getStress();

- Fairly straightforward by design

- Objective is for client code to be very simple, i.e. the client can use the object easily.

Student Client Code

```
// Make two students
Student a = new Student(12);  // new 12 unit student
Student b = new Student();    // new 15 unit student (default ctor)

// They respond to getUnits() and getStress()
System.out.println("a units:" + a.getUnits() +
    " stress:" + a.getStress());
System.out.println("b units:" + b.getUnits() +
    " stress:" + b.getStress());

a.dropClass(3);               // a drops a class

System.out.println("a units:" + a.getUnits() +
    " stress:" + a.getStress());
```

Student Client Code

```
// Now "b" points to the same object as "a" (pointer copy)
b = a;
b.setUnits(10);

// So the "a" units have been changed
System.out.println("a units:" + a.getUnits() +
    " stress:" + a.getStress());

// NOTE: public vs. private
// A statement like "b.units = 10;" will not compile in a client
// of the Student class when units is declared protected or private
```

Student Example Output

```
/*
    OUTPUT...
        a units:12 stress:120
        b units:15 stress:150
        a units:9 stress:90
        a units:10 stress:100
*/
```

Student Implementation

- Class Definition

  public class Student extends Object {

    ... <definition of the Student ivars and methods> ....

  }

- All classes are derived from the special class "Object"
  - We could have omitted extends Object here
- The class is defined in a file with the same name and a .java extension
  - In this case: Student.java

public / protected / private

- Public
  - Accessible anywhere by anyone
- Protected
  - Accessible only to the class itself and to it's subclasses or other classes in the same "package"
- Private
  - Only accessible within this class

Student Implementation Code

```
// Student.java
/*
 Demonstrates the most basic features of a class.
 A student is defined by their current number of units.
 There are standard get/set accessors for units.
  The student responds to getStress() to report
 their current stress level which is a function
 of their units.
  NOTE A well documented class should include an introductory
 comment like this. Don't get into all the details -- just
 introduce the landscape.
*/
public class Student extends Object {
     // NOTE this is an "instance variable" named "units"
     // Every Student object will have its own units variable.
     // "protected" and "private" mean that clients do not get access
     protected int units;
```

Student Implementation Code

```
/* NOTE
    "public static final" declares a public readable constant that
    is associated with the class -- it's full name is Student.MAX_UNITS.
    It's a convention to put constants like that in upper case.
*/
    public static final int MAX_UNITS = 20;
    public static final int DEFAULT_UNITS = 15;

    // Constructor for a new student
    public Student(int initUnits) {
        units = initUnits;
        // NOTE this is example of "Receiver Relative" coding --
        // "units" refers to the ivar of the receiver.
        // OOP code is written relative to an implicitly present receiver.
    }

    // Constructor that that uses a default value of 15 units
    // instead of taking an argument.
    public Student() {
        units = DEFAULT_UNITS;
    }
```

Student Implementation Code

```
// Standard accessors for units
    public int getUnits() {
            return(units);
    }
    public void setUnits(int units) {
            if ((units < 0) || (units > MAX_UNITS)) {
                        return;
                        // Could use a number of strategies here: throw an
                        // exception, print to stderr, return false
            }
            this.units = units;
            // NOTE: "this" trick to allow param and ivar to use same name
    }
    /*
     Stress is units *10.
      NOTE another example of "Receiver Relative" coding
    */
    public int getStress() {
            return(units*10);
    }
```

Student Implementation Code

```
/*
 Tries to drop the given number of units.
 Does not drop if would go below 9 units.
 Returns true if the drop succeeds.
*/
public boolean dropClass(int drop) {
        if (units-drop >= 9) {
                setUnits(units - drop);         // NOTE send self a message
                return(true);
        }
        return(false);
}
```

An idiom explained

- You will see the following line of code often:
  - public static void main(String args[]) { …}
- About main()
  - Invoked when you try to run an Application
  - Since the runtime must know which method to start at, it is made static (more later on this) so there is only one method per class
  - The Client code we saw earlier can be inside this main method.
    - See handout for details.

Java Features (Handout #5)

- Inheritance
  - A way of defining more specific versions of a class
    - Shape
      - Rectangle, Circle, Line
- We will cover inheritance in more detail later
  - For now just remember that all Java classes inherently inherit from a special class called Object (extends Object)

Primitives

- Very similar to C
  - Common across all platforms (JVM to the rescue!)
  - No unsigned variants
- Java Primitives

boolean	**true/false**	**long**	**8 bytes**
byte	**1 byte**	**float**	**4 bytes**
char	**2 bytes (unicode)**	**double**	**8 bytes**
int	**4 bytes**		

  - Generally used as local variables, parameters and instance variables (property of an object)

Primitives (cont)

- Note the lowercase letter for primitives!
- Primitives can be stored in arrays
- You cannot get a pointer to a primitive
  - To do that you need an Object
- There are Object “wrappers” for all primitives
  - The Object wrappers use upper case names!
    - Boolean, Integer, Float, Double
  - Hold a single primitive value
  - “Immutable!”

Primitives (cont)

- Object wrappers also contain some useful methods!
- Some common idioms to remember
  - Integer.parseInt(String) parses a String into an int primitive
  - Integer.toString(int) makes a String out of an int primitive
- The above idioms use static methods
  - We will cover static methods in a bit

Arrays

- Built in to Java
  - Not faked using pointers like in C
- Arrays are typed
  - Student[] students – will hold objects of type Student
  - int[] numbers – will hold int primitives
- Allocated using new – similar to allocating a new Object
- Arrays can be any size, but cannot change their size once allocated
  - No realloc() call like in C

Arrays (cont)

- Declaring Arrays
  - Preferred syntax: Student[] students;
  - Syntax for C refugees: Student students[];
- Allocating Arrays
  - students = new Student[100];
  - int[] numbers = new int[2*i + 100];
- Accessing Array elements
  - Same as C
- Java array extras
  - Arrays know their length (array.length)
  - Perform runtime checking on size

Array examples

Int Array Code

```
// Here is some typical looking int array code -- allocate an array and fill it with
    square numbers: 1, 4, 9, ...
// (also, notice that the "int i" can be declared right in the for loop -- cute.)
{
    int[] squares;
    squares = new int[100];                   // allocate the array in the heap
    for (int i=0; i<squares.length; i++) {            // iterate over the array
        squares[i] = (i+1) * (i+1);
    }
}
```

Student Array Code

```
// Here's some typical looking code that allocates an array of 100 Student objects
{
    Student[] students;
    students = new Student[100];        // 1. allocate the array
    // 2. allocate 100 students, and store their pointers in the array
    for (int i=0; i<students.length; i++) {
        students[i] = new Student();
    }
}
```

Array Literals and Anonymous Arrays

- Array Literal/Constant
  - Contents declared at declaration time
    - String[] words = { "hello", "foo", "bar" };
    - int[] squares = { 1, 4, 9, 16 };
    - Student[] students = { new Student(12), new Student(15) };
- Anonymous arrays
  - No variable defined to point to the array
    - new String[] { "foo", "bar", "baz"}

Array Utilities

- Java provides utilities for working on Arrays
  - System.arraycopy(sourceArray, sourceIndex, destArray, destIndex, length)
    - Will copy from one array to the other
    - Similar to memcpy in C
  - Arrays Class
    - Convenience methods for filling, searching, sorting
- Good time to visit the Java Docs!
  - API docs are your friend. USE THEM!!

Multidimensional Arrays

- Similar to C
  - int[][] big = new int[100][100]; // allocate a 100x100 array
  - big[0][1] = 10;// refer to (0,1) element
- Caveat
  - Unlike C, a 2-d java array is not allocated as a single block of memory. Instead, it is implemented as a 1-d array of pointers to 1-d arrays.

Strings

- Java has *great* support for Strings
  - String is an object, not a point to an array of chars
  - Strings (and char) both use 2-byte characters to support Internationalization (Kanji, Russian)
  - Strings are “Immutable”
    - String state doesn’t change
    - No append() or reverse() that changes the state of the object
    - To change a String, a *new* String is created!
    - This is done to allow sharing of objects

Strings (cont)

- String constants
  - Use double quotes
    - “Hello World!”
  - Builds a string and returns a pointer to it
- String concatenation
  - Official way String.concat
  - BUT for ease of use “This” + “That” will work!
    - String a = "foo";
    - String b = a + "bar"; // b is now "foobar“
- toString()
  - Most classes support a toString which will give a String representation of an Object!

String Class methods!

- Extensive list of methods available in the API documentation!
  - int length() -- number of chars
  - char charAt(int index)-- char at given 0-based index
  - int indexOf(char c) -- first occurrence of char, or -1
  - int indexOf(String s)
  - boolean equals(Object) -- test if two strings have the same characters
  - boolean equalsIgnoreCase(Object) -- as above, but ignoring case
  - String toLowerCase()-- return a new String, lowercase
  - String substring(int begin, int end) -- return a new String made of the begin..end-1 substring from the original

String example

```
String a = "hello";        // allocate 2 String objects
String b = "there";
String c = a;       // point to same String as a – fine

int len = a.length();       // 5
String d = a + " " + b;             // "hello there"

int find = d.indexOf("there");  // find: 6
String sub = d.substring(6, 11);        // extract: "there“

sub == b;            // false (== compares pointers)
sub.equals(b);    // true (a "deep" comparison)
```

StringBuffer

- Similar to String but mutable
  - Difference due to performance
- StringBuffer Example

```
StringBuffer buff = new StringBuffer();
for (int i=0; i<100; i++) {
        buff.append(<some thing>);
        // efficient append
}
String result = buff.toString();
// make a String once done with appending
```

System.out

- System class
  - Out represents the screen
- System.out.println()
  - Prints the string followed by an end of line
  - Forces a flush
- System.out.print()
  - Does not print the end of line
  - Does not force a flush
- System.out.flush()
  - Force a flush

== vs. equals()

- Remember
  - everything is a pointer (except primitives)
- ==
  - Compares pointers only! (shallow comparison)
  - Does *not* compare what is pointed to by the pointers
- equals() method
  - Default implementation same as ==
  - String class overrides to do a deep compare

String == and equals() example

```
String a = new String("hello");
// in reality, just write this as "hello“
// i.e. String a = “hello”;

String a2 = new String("hello");

a == a2    // false
a.equals(a2)    // true
```

Garbage Collector

- Example
  - String a = new String("a");
  - String b = new String("b");
  - a = a + b;   // a now points to "ab"
- Where did the original String a go?
  - Still sitting in the heap (memory) but it is "unreferenced"
    - It is unreachable by the program
  - But the Garbage collector knows it is there and can come clean it up!

Static

- Can have *static*
  - Instance variables
  - Methods
- Static variables and methods
  - Are associated with the class itself!!
  - Not associated with the object
- Therefore Statics can be accessed without instantiating an object!

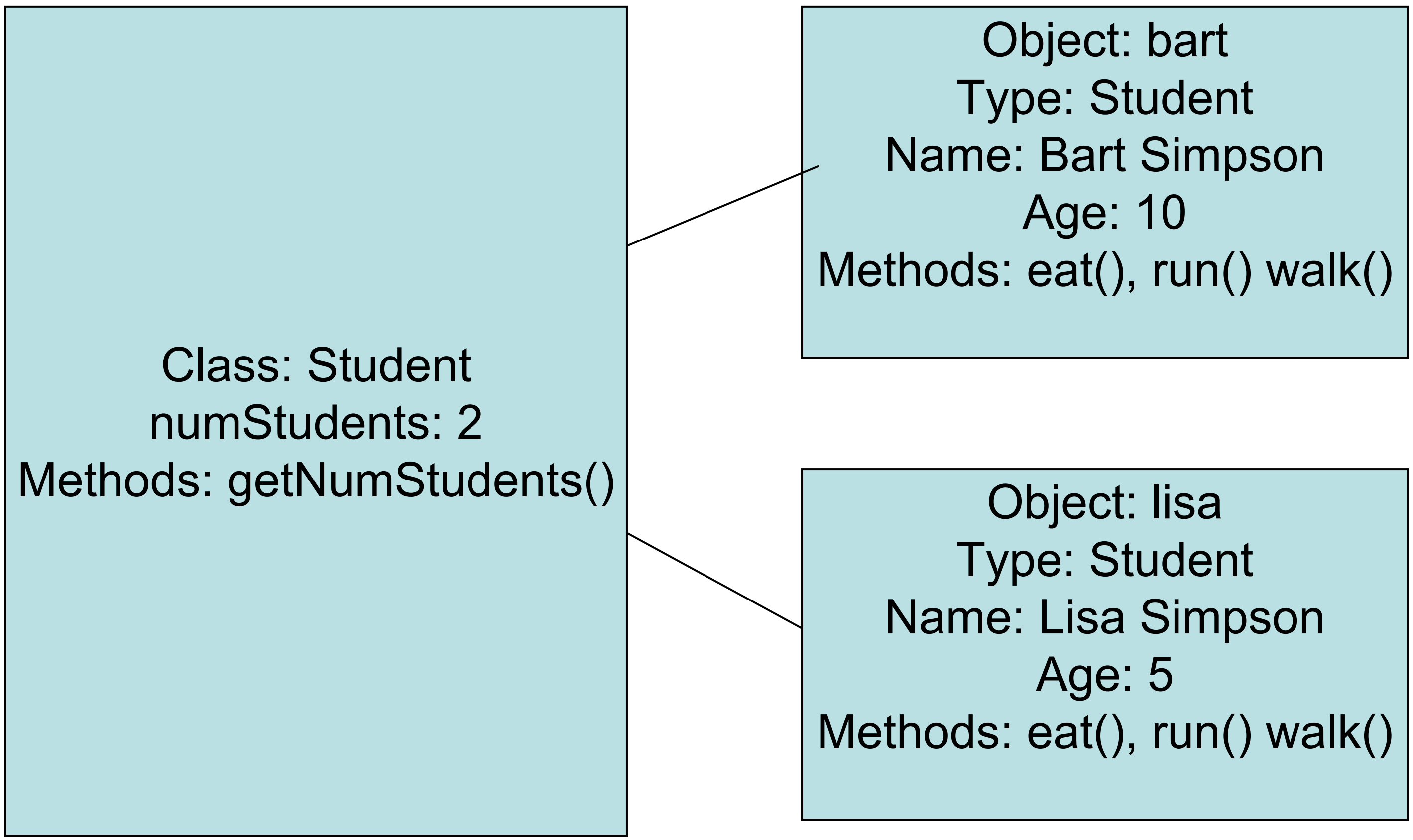
Static Variable

- Like a global variable
  - But on a class by class basis
  - Stored in the class
- Static variable occurs as a single copy in the class
  - Instance variables occur as multiple copies – one in each instance (object)
- Example
  - System.out is a static variable!

Static Methods

- Like a “global function”
  - Again on a class by class basis
- No Receiver!
  - Since the static method is associated with the class, there is no object that is associated with it and therefore, no “receiver”
  - You can think of it as the class being the receiver.
- Example
  - System.arrayCopy() is a static method

Static Fun


Class: Student
numStudents: 2
Methods: getNumStudents()
Object: bart
Type: Student
Name: Bart Simpson
Age: 10
Methods: eat(), run() walk()
Object: lisa
Type: Student
Name: Lisa Simpson
Age: 5
Methods: eat(), run() walk()

Static Example

```
public class Student {
    private int units;
    // Define a static int counter
    private static int count = 0;
    public Student(int init_units) {
        units = init_units;
        // Increment the counter
        count++;
    }
    public static int getCount() {
        // Clients invoke this method as Student.getCount();
        // Does not execute against a receiver, so
        // there is no "units" to refer to here
        return(count);
    }
    // rest of the Student class
    ...
}
```

Static Gotcha!

- Cannot refer to a non-static instance variable in a static method
  - There is no receiver (no object)
  - So the instance variable doesn't exist!
- Example

```
public static int getCount() {
    units = units + 1;   // error
}
```

OOP Design (Handout #6)

- Principles of OO Design
  - Encapsulation
    - Modularity
    - Inheritance (later)
  - Client Oriented Design
    - Implementation vs. Interface
    - User-centered design
- Good design and planning will go a long way in building software with fewer bugs!

Encapsulation

- "Don't expose internal data structures!"
- Objects hold data and code
  - Neither is exposed to the end user
- Objects expose an interface
  - Anthropomorphic nature of objects
    - Think of objects and people who have specialized roles!
      - Lawyer, Mechanic, Doctor
- Complexity is hidden inside the object
  - More modular approach
  - Less error prone

Public Interface Design

- Not adequate to simply provide getters and setters
  - Also known as accessors and mutators
- The interface exported by a class should mirror how that object is to be used.
  - example: ATM machine
- "Think about what the client wants to accomplish, not the details and mechanism of doing the computation"

Example: Bad Design #1

```
// client side code
private int computeSum(Binky binky) {
   int sum = 0;
   for (int i=0; i<binky.length; i++) {   // BAD
         sum += binky.data[i];   // BAD
   }
   return sum;
}
```

Exmaple: Bad Design #2

```
// client side code
private int computeSum(Binky binky) {
    int sum = 0;
    for (int i=0; i<binky.getLength(); i++) {  // BAD
        sum += binky.getData(i);               // BAD
    }
    return sum;
}
```

- External entity is doing too much work, the object should know how to do this itself!
  - Give the man a fish or teach a man to fish…

Example: Good Design

```
// Give Binky the capability
// (this is a method in the Binky class)
public int computeSum() {
    int sum = 0;
    for (int i=0; i<length; i++) {
        sum += data[i];
    }
    return sum;
}
// Now on the client side we just ask the object to
    perform the operation
// on itself which is the way it should be!
int sum = binky.computeSum();
```

Advantages of Encapsulation

- Clean Code!
  - Client code is cleaner and easier to understand
- Modularity
  - Easier debugging, less complexity
- Separate testing
  - Unit testing is possible
- Re-Use
- Team Programming
  - Easier to break down work amongst group members

OO Encapsulation Summary

- Separate abstraction from implementation
  - in OOP, expressed as messages (interface) vs. methods (implementation).
- "Expose" an interface that makes sense to the clients.
  - Ideally, the interface is simple and useful to the client, and the implementation complexity is hidden inside the object.
- Objects are responsible for their own state
  - Move the code to the data it operates on.

Client Oriented Design

- Based on what the user wants to accomplish
  - Not on how you implemented the functionality
- Intuitive and well documented
  - Java libraries are in general a good example of this
- Principle of least surprise
- Common-case convenience methods

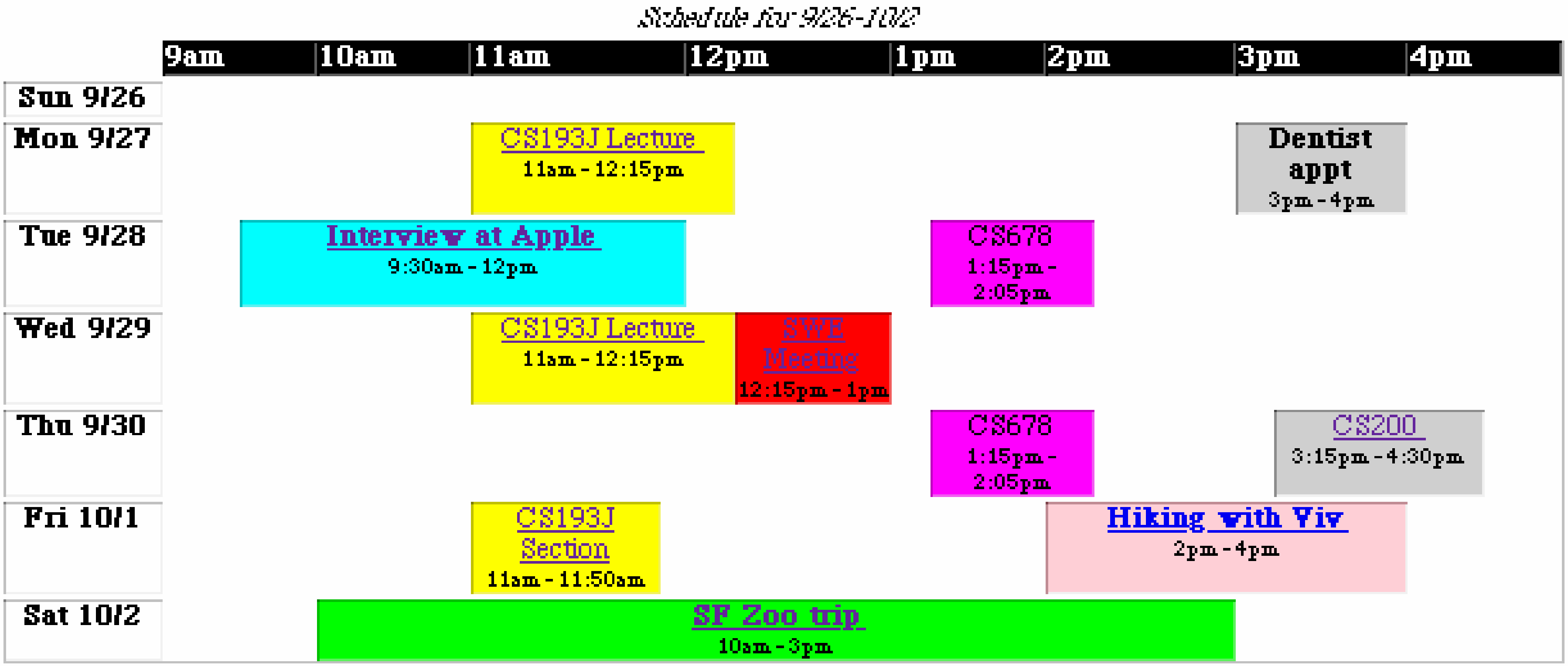
HW #1: Pencil Me In! (Handout #7)

- Basic Idea:
  - Input a text file description of schedule
    - Using one time events
    - Recurring events
  - Output listing of appointments for the week
    - List format
    - Table format
- Handout
  - Lots of detail and design ideas – READ WELL!
  - Start early!

HW #1: Pencil Me In!

- Sun 9/26
- Mon 9/27
  - *11am - 12:15pm* CS193J Lecture
  - *3pm - 4pm* Dentist appt
- Tue 9/28
  - *9:30am - 12pm* **Interview at Apple**
  - *1:15pm - 2:05pm* CS678
- Wed 9/29
  - *11am - 12:15pm* CS193J Lecture
  - *12:15pm - 1pm* SWE Meeting
- Thu 9/30
  - *1:15pm - 2:05pm* CS678
  - *3:15pm - 4:30pm* CS200
- Fri 10/1
  - *11am - 11:50am* CS193J Section
  - *2pm - 4pm* **Hiking with Viv**
- Sat 10/2
  - *10am - 3pm* **SF Zoo trip**

HW #1: Pencil Me In! (Handout #7)



Summary

- Today
  - OOP/Java
    - Student Example
  - Java Features
    - arrays, strings, static etc
  - OOP Design
    - encapsulation, client-oriented design
- Assigned Work:
  - HW #1: Pencil me In
    - Due before midnight Wednesday, July 9^{th}, 2003
  - Skim the Sun Java Tutorial
    - http://java.sun.com/docs/books/tutorial/