CS61B, Spring 2003 **Discussion #1** **Amir Kamil**
UC Berkeley **1/27/03**

Topics: Administrivia, Advice, Course Overview, OOP Review, Java Intro

1 Administrivia

- About me:
  - Amir Kamil
  - 3rd year EECS major
  - Email: `cs61b-tb@cory.eecs.berkeley.edu`
  - Webpage: `http://inst.eecs.berkeley.edu/~cs61b-tb`
  - Be warned that I have a twin brother who is a CS major, so if you get an unexpected reaction when you ask one of us for help, it's probably him.
- Course:
  - You can and should work together on labs.
  - Labs will be due at the end of the session for which they are assigned. I recommend that you get started before you come to lab.
  - Labs and homeworks will be graded.
  - If you are in a cancelled section or want to attend a different lab than the one you are in, please switch through Telebears. You may attend any discussion section, space permitting, and any TA's office hours.
- Section:
  - Notes will be available on my webpage, probably as PDFs. They will usually cover more material than we do in section.

2 Advice

- If you have questions, first check the newsgroup (`ucb.class.cs61b`) to see if it has already been answered. If not, post your question. Most questions you have will probably be answered already.
- Since you can work together on labs, do them in groups.
- I know I don't have to say this (`*ahem*`), but I will anyway. DON'T CHEAT! You will be caught, trust me. The software program we use catches all forms of cheating that I can think of. If you fall behind, ask the teaching staff for help.

3 Course Overview

- The goal of CS61B is to teach you how to program, not to teach you Java. If that's what you want, take CS9G. Java is a tool through which programming is taught.
- We will teach you data structures including arrays, lists, heaps, stacks, trees, graphs.
- We will also teach you some algorithms such as sorting, algorithmic analysis, and possibly other, related topics.
- There are three projects in addition to homeworks and labs. Start early, or you won't finish.

4 OOP Review

- OOP Vocabulary:
  - `object`: A repository of data with access to `methods` to manipulate that data.
  - `class`: A type of object. All objects of that type have the same, but independent, `variables` and share the same `methods`.
  - `method`: A procedure that operates on an object or class.
  - `variable`: Names for pieces of data. Variables that are contained within an object are also called `instance variables` or `fields`.
  - `inheritance`: The ability to extend a class with additional methods or fields without modifying the original class.
- An object of class `A` is called an `instance` of `A`.
- Multiple instances of a class can exist at the same time - they each have their own sets of variables. For example, if you have a class `Computer`, you can have as many computers as you want in your program. Say you want to represent 277 Soda in Java. Then you might have 30 or so computers to represent each computer in the lab. They might have variables to represent their state (on, off, crashed), who's using them, and the programs they're running. The beauty of OOP is that you only have to define a single `Computer` class, and you can have as many independent instances as you need.
- Inheritance means you can have `subclasses` of a class. For example, there are many different types of computers, e.g. laptops, desktops, Macs, PCs. They all share much of the same behavior: they can be on, off, or crashed, have different users, and run programs. But they also have differences in behavior. A laptop may be out of battery while a desktop can't be, a PC might be blue-screened while a Mac would just be unresponsive. Inheritance allows them to share many behaviors while differing in others. We will go into more detail on inheritance next time.
- Methods are the means through which objects interact with each other. I reluctantly remind you of message passing from 61A, but keep in mind that this is not how OOP is implemented in Java. Methods may require certain `parameters`, variables that are passed to the target object when one of its methods is called.

5 Java Introduction

- Class definitions:

```
public class Computer {
  <Variables and Methods>
}
```

access modifier
class name

- Method definitions:

```
public int intItem(int i) { // This line is the prototype.
  <Expressions>
}
```

access, other modifiers (e.g. static, final)

return type
function name
parameters

- Unlike in Scheme, methods in Java must explicitly return values, using the `return` keyword.
- Also unlike Scheme, Java is typed. Variables must be declared to be a certain type, and then can only be assigned values that are of that type or a subtype of that type.
- Variable declarations:

  ```
  int num = 3;
  Computer comp = new Computer("MSDOS 5.0");
  ```

  variable type
  variable name
  optional initialization

- Primitive types:
  Java includes some built-in types that are NOT objects. Examples include `int`, `float`, `boolean`, `char`, and others. Primitive types are initialized with values, while objects are initialized with `new`.
- Variable values:
  The value of a primitive variable is the actual number or boolean that it represents. The value of an object variable is actually a `reference` or `pointer` to the object.

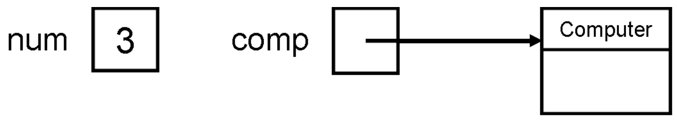

Figure 1: The values of a primitive and an object variable after initialization.

- Constructors:

  ```
  public Computer(String os) {
    <Expressions>
  }
  ```

  A constructor is like a method, but it is only called at the creation of an object and has no return value. The expression `new Computer(<String>)` creates a new instance of the `Computer` class, initializes it by calling the constructor, and returns a reference to the object.
- Comments:
  Anything between `//` and the end of the line is a comment, and will be ignored by the compiler. Anything between `/*` and `*/` is a comment, and will be ignored by the compiler.
- Example:

  ```
  /* A simple example of a Java class */
  /* Imports classes from external packages */
  import java.util.*;
  public class Computer {
  ```

```
  /* Fields */
  int state = 0; // Let 0 = off, 1 = on, -1 = crashed.
  String os; // The operating system on the computer.
  String user;
  // Initially an empty list; no programs running.
  LinkedList programs = new LinkedList();

  /* Turns on the computer, then boots it. */
  public void turnOn() {
    state = 1;
    boot();
    return;
  }

  /* Boots the computer */
  private void boot() {
    if (os.equals("Windows 95")) {
      state = -1; // crash
    } else {
      programs.add(os);
    }
  }

  /* Logs a user into the computer */
  public void login(String user) {
    this.user = user;
  }

  /* Constructor */
  public Computer(String os) {
    this.os = os;
    user = "no one";
  }

}
```

Notes:
`void` denotes no return value. In this case, a return statement is unnecessary, but allowed, at the end of the method. `this` refers to the current object, so `this.<name>` refers to the instance variable named `<name>`. It may be necessary to use `this` if variables names from an inner scope obscure instance variables of the same name.