Index of /~cs61b/lectures/12

Name	Last modified	Size	Description
Parent Directory	24-Sep-98 21:22	-	
Lecture12.java	21-Sep-98 09:48	3k	

```
                     CS61b: Lecture 12
                    Monday, September 21
INHERITANCE REVIEW
------------------

class Worker {
  public Worker() { ... }
  public void work () { ... }
  public void eat () { ... }
  public void live() { work(); eat(); }

  protected int hunger;
  private static int workDone = 0;
}

class TA extends Worker {
  public TA(String p) { ... }
  public void work() { hunger++; System.out.println("let me help"); }
  public void add(String student) { ... }
  private String prof;
  private Vector classList;
}

Use:
    static void foo (Worker w) { ...cannot use "add"... }
    static void bar (TA t) { ...may use "add"... }

    Worker w1 = new Worker();
    w1.live();                   // prints: whistle, yum
    foo(w1);
    // bar(w1);    not legal

    TA t1 = new TA("Yelick");
    t1.live();                   // prints: let me help, yum
    foo(t1);
    bar(t1);

Inside foo(t1), the w variable has static type Worker, dynamic types TA.

ABSTRACT CLASSES
----------------
Sometimes you define a class that you know will be extended.
E.g., you'd like to have every Worker print the names of people they work with,
but this code depends on what kind of worker they are.

abstract class Worker {
```

```
  ...
  abstract public void printWorksWith(); // Prints the names of people one
                        // works with
}

class TA extends Worker {
  ...
  public void printWorksWith() {
      System.out.println("Professor of course: " + prof);
      System.out.println("Students in section: " + classList);
  }
}
```

Use:

```
   Worker w1 = new Worker();     // not legal, some methods not defined.
   TA t1 = new TA("Yelick");     // OK
   Worker w2 = new TA("Yelick"); // OK
```

Any (non-abstract) class that extends Worker must define printWorksWith().
Cannot create objects of an abstract class, e.g., new Worker(..) not legal.

INHERITANCE

So far, we have seen 2 kinds of classes:

1) (concrete) classes -- defines a new type that you can use to build objects
2) abstract classes -- a class with 1 or more "holes" called abstract methods.

There are many ways of building new classes from old.

    class extends class
    class extends abstract classes
    abstract class extends abstract class (but eventually you need a class)
    abstract class extends class

Sometimes it makes sense to have a class in which ALL of the methods are "undefined" because this acts as a specification for a set of classes.

INTERFACES

```
    interface Negatable {
      void negate();
    }

    interface Comparable {
      boolean lessThan (Comparable c);
      boolean greaterThan (Comparable c);
    }

    class IntBox implements Negatable, Comparable {
      public void negate() { value = -value; }
      public boolean lessThan (Comparable c) { return value < ((IntBox) c).value; }
      public boolean greaterThan (Comparable c) { return value > ((IntBox) c).value;
      ...
      private int value;
    }
```

Or we could group these two interfaces into a third:

```
    interface NegatableAndComparable extends Negatable, Comparable { }
```

Now we could write a method like "sort" that requires something be

"Comparable," but it would allow any type that "implements" Comparable.

Interfaces in General:
  * An interface is like an abstract class, except
     * Only has methods signatures and constant, static fields.
     * All members are implicitly public.
  * A class may implement any number of interfaces (unlike abstract classes!!).
  * Cannot apply new to an interface; never any pointers to 'bare' interfaces.
  * An interface may extend any number of interfaces (superinter-faces).

WHY INTERFACES? WHY CAN'T WE JUST USE ABSTRACT CLASSES?

Two general kinds of inheritance (independent of the programming language):
  1) Implementation inheritance: classes inheriting from classes (may override
     some or all of the methods)
  2) Interface inheritance: classes inheriting the method names and
     signatures (i.e., numbers and types of arguments)

Type 2 works find with multiple inheritance (> 1 parent), but 1 gets complicated
if both parents define the same method or data field.