CS61B Summer 2006
Instructor: Erin Korber
Lecture 5, 3 July
Reading for tomorrow: Chs. 7 and 8

1 Comparing Objects

- Every class has an `equals` method, whether you write one or not.
  - If you don't, it will return the same thing as `==`
  - But this may not be what you want...
- Several different kinds of equality to consider:
  - Reference equality, ==.
    - Two objects are == only if they are the same object.
    - Could also describe this by saying, two references are == if they point to the same object.
  - Shallow structural equality
    - Two objects are "equals" if all of their fields (instance variables) are ==.
  - Deep structural equality
    - Two objects are "equals" if all of their fields (instance variables) are "equals".
  - Logical equality
    - Precisely what this means will depend on what you are dealing with.
    - Example: Two "Set" objects are "equals" if they contain the same elements, even if they are stored in a different order.
    - Example: You would probably want to say that the fractions 1/2 and 2/4 are "equals", even though their numerators and denominators are different, since they represent the same value.
- You will want to decide which kind of equality to test for, depending on what seems most appropriate for your particular application.

2 The “static” keyword

2.1 static fields

A static field is a single variable shared by a whole class of objects; its value does not vary from object to object. For example, if ”numberOfHumans” is the number of Human objects that have been constructed, it is not appropriate for each object to have its own copy of this number; every time a new Human is created, we would have to update every Human.

If we declare a field ”static”, there is just one field for the whole class. You may remember these from CS61A - they were called <u>class variables</u>.

Example of a static field:

```
class Human {
  public static int numberOfHumans;

  public int age;
  public String name;

  public Human() {
    numberOfHumans++;    // The constructor increments the number by one.
  }
}
```

If we want to look at the variable numberOfHumans from another class, we write it in the usual notation, but we prefix it with the class name rather than the name of a specific object.

```
  int k = Human.numberOfHumans;  // Good.
  int k = john.numberOfHumans;   // This works too, but has nothing to
                                 // do with john specifically.  Don't
                                 // do this; it's bad (confusing) style.
```

`System.out` and `Math.PI` are other examples of static fields.

2.2 static methods

Methods can be static too. A static method runs *without any instance of the class.* THERE IS NO ”this”, since there is no object. Any attempt to reference ”this” will cause a compile-time error.

Static methods never use instance variable values - of course they can’t, since there is no instance to get them from!

```
class Human {
  ...
  public static void printHumans() {
    System.out.println(numberOfHumans);
  }
}
```

Now, we can call "Human.printHumans()" from another class. We could also call "john.printHumans()", but it's bad style.

The main() method is always static, because when we run a program, we are not passing an object in, so there are no instances around on which to call any non-static methods.

3 final

Java's `final` keyword is used to declare something that can never be changed.

For example:

```
  public final static int FEBRUARY = 2;
```

If you ever after try to assign a value to FEBRUARY, you'll have a compiler error.

If you find yourself repeatedly using a numerical value with some "meaning" in your code, you should probably turn it into a `final` constant.

BAD: if (month == 2)

GOOD: if (month == FEBRUARY)

Why? Because if you ever need to change the numerical value assigned to February, you'll only have to change one line of code, rather than hundreds.

It is a long-standing stylistic convention to put the names of constants in all caps - this was inherited from C.

For any array x, "x.length" is a "final" field. (Why do you think this is?)

`final` isn't just for variables - methods and classes can also be declared `final`. We'll talk about what exactly this does later, when we talk about inheritance.

4 Special loop constructs

4.1 break

A ”break” statement immediately exits the innermost loop statement enclosing the ”break”, and continues execution at the code following the loop.

Recall the `takeDog` method from our Kennel example last week:

```
void takeDog(Dog d) {
  boolean has_home = false;

  for(Doghouse h : houses) {
     if (!has_home && h.canFitDog(d)) {
         h.resident = d;
         has_home = true;
        //****
     }
  }
  if (!has_home) {
     System.out.println("Dog won’t fit!");
  }
}
```

You may have noticed that the for loop will keep repeating until the end of the array, even if we find a home for the dog in the beginning! We could prevent this from happening by using a `break` statement at the starred point in the code - this will cause the flow of control to exit the loop immediately at that point.

WARNING: It’s easy to forget exactly where ”break” will jump to. For this reason, Java (unlike C) allows you to attach labels to any kind of enclosing statement, including ”if” statements and any group of statements placed in braces . Then, ”break” can jump to the end of any labeled enclosure that encloses the ”break” statement - just give it the name of the label for the enclosure that you want to jump to the end of.

Example: finding an item in a two-dimensional array of ints with labelled break:

```
public void findInt(int[] a, int searchfor){
    search:
        for (int i=0; i<a.length; i++;) {
            for (j = 0; j < arrayOfInts[i].length; j++) {
                if (arrayOfInts[i][j] == searchfor) {
                    foundIt = true;
                    break search;
                }
            }
        }
        if (foundIt) {
            System.out.println("Found "+searchfor+" at "+i+", "+j);
        } else {
            System.out.println(searchfor+ "not in the array");
        }
}
```

Note that the break statement terminates the labeled statement; it does not transfer the flow of control to the label. The flow of control is transferred to the statement immediately following the labeled (terminated) statement (since it's the labelled enclosure that is being "broken").

4.2 continue

The `continue` statement is used to skip the current iteration of a loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop, basically skipping the remainder of this iteration of the loop.

Note that `continue` does NOT exit the loop - another iteration may commence, if the loop condition is satisfied.

What's the difference between the following two loops?

```
  int i = 0;                  | for (int i = 0; i < 10; i++) {
  while (i < 10) {            |   if (condition(i)) {
    if (condition(i)) {       |     continue;
      continue;               |   }
    }                         |   call(i);
    call(i);                  | }
    i++;                      |
  }                           |
```

Answer: when "continue" is called in the "while" loop, "i++" is not executed. In the "for" loop, however, i is incremented with every iteration.

Use `break` and `continue` sparingly. They make it more difficult for both you and others to read, understand, and debug your code. If you find a lot of them appearing in your code, you may want to rethink how you've planned out control flow.

4.3 switch

Some long chains of if-then-else clauses can be simplified by using a "switch" statement. "switch" is appropriate only if every condition tests whether a variable x is equal to some constant.

```
switch (month) {     |  if (month == 2) {
case 2:              |    days = 28;
  days = 28;         |  } else if ((month == 4) || (month == 6) ||
  break;             |             (month == 9) || (month == 11)){
case 4:              |    days = 30;
case 6:              |  } else {
case 9:              |    days = 31;
case 11:             |  }
  days = 30;         |
  break;
default:
  days = 31;
  break;
}              //  These two code fragments do exactly the same thing.
```

IMPORTANT: "break" jumps to the end of the "switch" statement. If you forget a break statement, the flow of execution will continue right through past the next "case" clause. If month == 12 in the following example, both Strings are printed:

```
switch (month) {
case 12:
  output("It's December.");
  // Just keep moving right on through.
case 11:  case 1:  case 2:
  output("It's cold.");
default:
}
```

However, this style of coding is hard to read and understand. If there's any chance that other people will need to read or modify your code (which is very common when you program for a business), don't code it like this. Use break statements in the switch, and use subroutines to reuse code or clarify the control flow.

4.4 return

You've already used `return` as a means for a function to return a value. It can have another purpose: to affect the flow of control of a program. It causes a method to end immediately, so that control returns to the calling method.

A small example:

```
public void myMethod(int x) {
  if (x == 10) {
    return;
  }
  System.out.println("Thank goodness x isn't 10.");
}
```