CS 4: Lecture 8
Monday, February 13, 2006

TYPES OF VARIABLES

Java has three kinds of variables: local variables, instance variables, and class variables. Here's how Java tells them apart.
- _Local_variables_ are declared inside a method.
- _Instance_variables_, also known as _fields_, are declared inside a class, but NOT inside any method.
- _Class_variables_, also known as _static_fields_, are declared inside a class with the "static" keyword, but NOT inside any method.

Local Variables

- Local variables are created when a method is called, and vanish forever when the method ends (returns).
- Each method's local variables can be used inside the method only.
- Formal parameters are local variables.
- Local variables cannot be used before they are initialized.

For example, if I try to compile this method:

```
  void add(long i) {
    long j;

    j = j + i;
  }
```

the Java compiler prints an error message, "Variable 'j' may not have been initialized." You can't add something to j until you've assigned j a value. The following code compiles without errors.

```
  void add(long i) {
    long j;

    j = 1;
    j = j + i;
  }
```

When Java executes this method, it creates spaces in memory to store the local variables "i" and "j". When Java reaches the end of the method, it erases both variables and frees their memory so it can be reused.

A method that calls add() can also have a local variable named "j". Even though it has the same name, it is NOT the same variable.

```
  void callAdd() {
    int j = 20;

    add(5);
    System.out.println(j);
  }
```

While this method is calling add(), there are two variables named "j". But only add() knows about its "j", and only "callAdd" knows about its "j". There is no way to get the two mixed up. At the end of "callAdd", Java prints "20", not "6".

Class Variables

A _class_variable_, also known as a _static_field_, is a single variable shared by a whole class of objects. Its value does not vary from object to object.

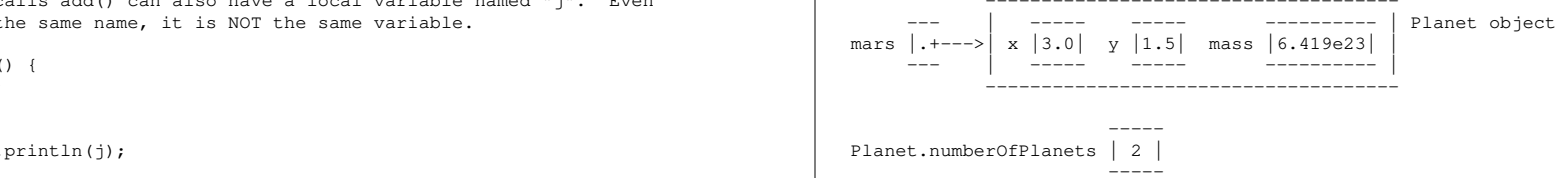
For example, if "numberOfPlanets" is the number of Planet objects that have been constructed, it is not appropriate for each Planet object to have its own copy of this number; every time a new Planet is created, we would have to update _every_ Planet. This is impractical, because we might have constructed millions of Planet objects. We don't need to waste the time nor memory to maintain a million copies of the same number.

```
public class Planet {
  public static long numberOfPlanets;        // class variable, shared by all

  private double x, y;                       // instance variables:
  private double mass;                       //   position, mass, nickname
  public String nickname;

  public Planet(newX, newY, newMass) {       // constructor
    x = newX;                                // x and y are instance variables,
    y = newY;                                //   but newX and newY are local
    mass = newMass;
    numberOfPlanets++;           // the constructor increments the count by one
  }

  private nuke() {
    mass = 0.0;
  }
}
```

(We'll talk later about what the "private" stuff is all about.)

If we want to look at the variable numberOfPlanets 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. Here's code in another class:

```
  void helloBye() {
    Planet mars = new Planet(3.0, 1.5, 6.419e23);
    mars.nickname = "God of War";              // Changes the object mars only
    Planet.numberOfPlanets++;                  // Changes EVERY Planet object
  }
```



System.out is another example of a class variable (in the System class).

Lifetimes of Variables

Not all variables survive through an entire run of your program.

- A local variable (declared in a method) is gone forever as soon as the method in which it's declared finishes executing. (If it references an object, the object might continue to exist, though.)
- An instance variable (non-static field) lasts as long as the object exists. An object lasts as long as there's a reference to it.
- A class variable (static field) lasts as long as the program runs.

Consider the method helloBye() above. When Java reaches the end of the method, the local variable "mars" vanishes in a puff of smoke. Then, there are no references left that point at the object, so the object vanishes in a puff of smoke, too, and so do the instance variables "x", "y", and "mass" inside that object. But Planet.numberOfPlanets, the class variable, survives.

THE "public" AND "private" KEYWORDS
==========

Thus far, we've usually declared fields and methods using the "public" keyword. However, we can also declare a field or method "private". A private method or field is invisible and inaccessible to other classes, and can be used only within the class in which the field or method is declared.

Why would we want to make a field or method private?
(1) To prevent data within an object from being corrupted by other classes.
(2) To ensure that you can improve the implementation of a class without causing other classes that depend on it to fail.

In the following example, EvilTamperer tries to corrupt the internals of a Planet object.

```
public class EvilTamperer {
  public void tamper(Planet earth) {
    earth.mass = -10.0;          // Foiled!!
    earth.nuke();                // Foiled again!!
  }
}
```

However, EvilTamperer won't compile, because the Planet class has declared mass and nuke() "private". The Planet constructor is public, so tamper() could call it, but that only means tamper() can create a new planet; tamper() can't change the earth.

Scope

The _scope_ of a variable is the portion of the program that can access the variable. When you try to access a variable that's not in scope, you usually get a compiler error. Here are some of Java's scope rules.

(1) Class variables (static fields) are in scope everywhere in the class.
(2) If a class variable is declared "public", it is in scope in _all_ classes if it is _fully_qualified_, meaning it has the class name as a prefix, like "Planet.numberOfPlanets" or "System.out" .
(3) Instance variables (non-static fields) like "mass" are in scope in _non-static_ methods called on an appropriate object.
(4) Fully qualified instance variables like "mars.mass" are in scope everywhere in the class. If they're public, they're in scope in _all_ classes.
(5) Local variables and parameters are in scope only inside the method that declares them.

To illustrate rules (3) and (4), compare the following methods inside the Planet class.

```
  void m1() {
    double d1 = mass;                  // mass of "p" if called with "p.m1()"
  }

  static void m2() {
    double d2 = mass;                  // Compiler error!
  }

  static void m3(Planet planet) {
    double d3 = planet.mass;           // mass of "planet"
  }
```

Method m1 is okay, because m1 is not static, so "mass" is in scope. Method m2 causes a compiler error, because "mass" is not in scope in a static method. But method m3 is okay, because "planet" is a local variable in scope in m3, and "planet.mass" is a fully qualified instance variable.