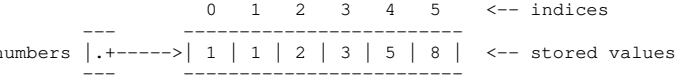
CS 4: Lecture 9
Wednesday, February 15, 2006

ARRAYS
======
In Lecture 7, we discussed a way to compute the position of a car whose velocity is a changing function of time. Suppose you perform a million timesteps, but you need to _remember_ where the car was at each point in time--perhaps so you can use the car position data to analyze its gas mileage at various speeds, or determine exactly when it entered Nebraska.

You could declare a million different variables, one for each point in time, but you'll be busy writing the code for a good long time. Or you could use an _array_.

An array is an object consisting of a numbered list of variables, each of the same type. The variables could be ints, doubles, or any other primitive type; or they could be references to objects, all of the same class. The variables in an array are always numbered from zero, in increments of one. For example, here is an array of ints.



Because arrays are objects, you access them through references like "numbers" above. You declare a reference to an array like this.

```
int[] numbers;      // Reference to an array (of any length) of ints.
```

We can construct an array of six ints, numbered 0 through 5 (as illustrated above), like this.

```
numbers = new int[6];
```

We now have an array object that contains six _elements_ of type int. Each element has a different _index_, from 0 to 5. An array element is just like an instance variable, except that you access it by using its index and the following notation.

```
numbers[5] = 8;     // Store an "8" at index "5".
```

If we try to address any index outside the range 0...5, Java will complain with a run-time error called an ArrayIndexOutOfBoundsException.

```
numbers[6] = 0;     // Program halts during run-time with an error message.
```

A _run-time_error_ is an error that doesn't show up when you compile the code, but does show up later when you run the program and Java tries to access the out-of-range index.

Once "numbers" references an array, you can find out its length by looking at the field "numbers.length". You can never assign a value to the "length" field, though. Java will give you an error at compile-time if you try.

What makes arrays great is that you can set up a loop whose loop variable indexes the array. The following loop computes the Fibonacci sequence.

```
numbers[0] = 1;
numbers[1] = 1;
int i = 2;
while (i < numbers.length) {
  numbers[i] = numbers[i - 1] + numbers[i - 2];
  i++;
}
```

Now the array looks just like the picture near the beginning of this lecture. If you want a longer Fibonacci sequence, just construct a longer array and use the same code. Observe that during the last iteration of the loop, i == numbers.length - 1. The loop does not try to store anything in numbers[number.length]; if it did, it would cause a run-time error.

SIMULATIONS WITH ACCELERATION
==========

In Lecture 7, we saw how to discretize the motion of a car whose velocity is given as a function of time. But suppose the acceleration a(t), instead of the velocity, is given as a function of time. Now what do we do?

Just as velocity is the derivative of distance with respect to time, acceleration is the derivative of velocity with respect to time.

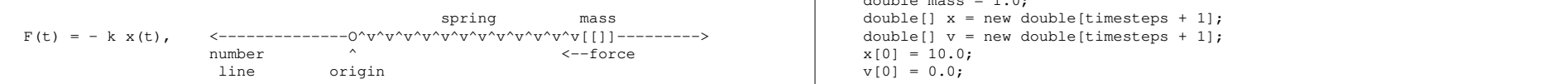
$$\frac{dx(t)}{dt} = v(t), \quad \frac{dv(t)}{dt} = a(t).$$

This is called a _coupled_differential_equation_. It's a differential equation with two unknown functions--x(t) and v(t)--that are related to each other somehow. In this case, x(t) depends on v(t), but v(t) does not depend on x(t), so we can discretize it using the trapezoid rule.

$$v(t + T) \approx v(t) + \frac{a(t) + a(t + T)}{2} T, \quad x(t + T) \approx x(t) + \frac{v(t) + v(t + T)}{2} T.$$

First we compute v(t + T), then x(t + T), which depends on v(t + T).

But let's change the problem a bit. Instead of a car, let's model a mass that is attached to the origin by a spring, which pulls the mass toward the origin. The spring exerts a force on the mass described by Hooke's law,



where k is the _spring_constant_, which quantifies how strong the spring is. By Newton's law, F = ma, so

$$a(t) = \frac{F(t)}{m} = -\frac{k}{m} x(t).$$

But now our discretization has a chicken-and-egg problem: a(t + T) depends on x(t + T), which depends on v(t + T), which depends on a(t + T). We can't calculate any of them without calculating the others first.

The solution is to use a slightly different discretization, called _Heun's_ _method_. Heun's idea is to break the cycle by using Euler's method to guess the value of x(t + T). Of course, Euler's method isn't as accurate as the trapezoid rule, but we'll still compute the position accurately, because we're going to calculate x(t + T) a _second_ time, using the trapezoid rule. One can mathematically prove that this gives quite good accuracy. So Heun's method works like this:

$x^*(t + T) \approx x(t) + v(t)\, T$, (compute a less accurate x* with Euler's method)

$$v(t + T) \approx v(t) - \frac{k}{m} \frac{x(t) + x^*(t + T)}{2} T, \quad \text{(trapezoid rule)}$$

$$x(t + T) \approx x(t) + \frac{v(t) + v(t + T)}{2} T. \quad \text{(trapezoid rule)}$$

Here's the inner loop of the code for performing the simulation.

```
    while (step < timesteps) {
      double xEuler = x[step] + v[step] * steptime;
      v[step + 1] = v[step] - k / mass * (x[step] + xEuler) / 2.0 * steptime;
      x[step + 1] = x[step] + (v[step] + v[step + 1]) / 2.0 * steptime;
      step++;
    }
```

Another example where Heun's method is useful is a simulation of planets orbiting each other, because the gravitational forces that planets exert on each other depend on the positions of the planets.

Postscript

Here's the entire program. I didn't write the whole thing down in class, but you might find it helpful to see how the variables are set up before the loop. It's also interesting to run the program and try playing with the numbers (number of timesteps, spring constant, etc.) to see how it affects the simulation.

```
public class Spring {
  public static void main(String[] args) {
    double duration = 90.0;                            // Duration of simulation
    int timesteps = 100;                               // Number of timesteps
    double steptime = duration / (double) timesteps;   // Time for one timestep
    double k = 0.1;
    double mass = 1.0;
    double[] x = new double[timesteps + 1];
    double[] v = new double[timesteps + 1];
    x[0] = 10.0;
    v[0] = 0.0;

    double distance = 0.0;                             // Distance traveled
    int step = 0;                                      // Current step number
    while (step < timesteps) {
      System.out.println("Position:  " + x[step] + ".");
      double time = steptime * (double) step;
      double xEuler = x[step] + v[step] * steptime;
      v[step + 1] = v[step] - k / mass * (x[step] + xEuler) / 2.0 * steptime;
      x[step + 1] = x[step] + (v[step] + v[step + 1]) / 2.0 * steptime;
      step++;
    }
    System.out.println("Position:  " + x[step] + ".");
  }
}
```

Observe that x and v reference arrays of length timesteps + 1. Why did we have to add one?