Eric Roberts
CS 106A

Handout #62
June 1, 2012

Looking Ahead

Looking Ahead

Eric Roberts
CS 106A
May 30, 2012

Looking Ahead

- Chapter 14 offers a brief introduction to four topics that are sometimes included an introductory computer science course.
- In today's class, I'll offer a brief introduction to three of those topics:
  1. *Recursion*
  2. *Concurrency*
  3. *Programming patterns*
- The first topic is a central theme of CS 106B. The others are covered in more advanced classes.

Recursion

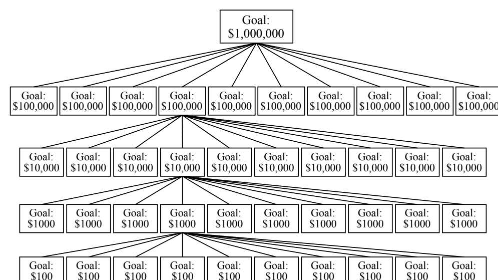
- ***Recursion*** is an extremely powerful programming strategy in which you solve a problem by dividing it into smaller subproblems *of the same form*.
- The italicized phrase represents an essential characteristic of recursion; without it, all you have is a description of stepwise refinement.
- The fact that recursive decomposition generates subproblems that have the same form as the original problem means that recursive programs will use the same method to solve subproblems at different levels of the solution. In terms of the structure of the code, the defining characteristic of recursion is having methods that call themselves, directly or indirectly, as the decomposition process proceeds.

A Simple Illustration of Recursion

- Suppose that you are the national fundraising director for a charitable organization and need to raise $1,000,000.
- One possible approach is to find a wealthy donor and ask for a single $1,000,000 contribution. The problem with that strategy is that individuals with the necessary combination of means and generosity are difficult to find. Donors are much more likely to make contributions in the $10 range.
- Another strategy would be to ask 100,000 friends for $10 each. Unfortunately, most of us don't have 100,000 friends.
- There are, however, more promising strategies. You could, for example, find ten regional coordinators and charge each one with raising $100,000. Those regional coordinators could in turn delegate the task to local coordinators, each with a goal of $10,000, continuing the process reached a manageable contribution level.

A Simple Illustration of Recursion

The following diagram illustrates the recursive strategy for raising $1,000,000 described on the previous slide:



A Pseudocode Fundraising Strategy

If you were to implement the fundraising strategy in the form of a Java method, it would look something like this:

```
private void collectContributions(int n) {
   if (n <= 100) {
      Collect the money from a single donor.
   } else {
      Find 10 volunteers.
      Get each volunteer to collect n/10 dollars.
      Combine the money raised by the volunteers.
   }
}
```

What makes this strategy recursive is that the line

Get each volunteer to collect n/10 dollars.

will be implemented using the following recursive call:

```
collectContributions(n / 10);
```

Recursive Functions

- The easiest examples of recursion to understand are functions in which the recursion is clear from the definition. As an example, consider the factorial function from Chapter 5, which can be defined in either of the following ways:

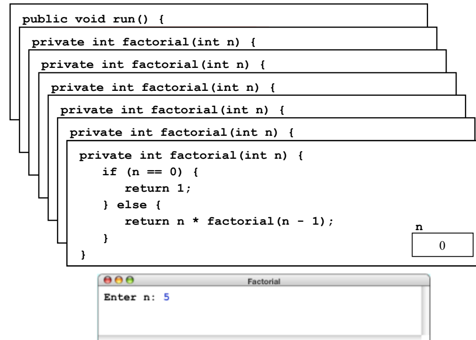
$$n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$$

$$n! = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

- The second definition leads directly to the following code, which is shown in simulated execution on the next slide:

```
private int factorial(int n) {
   if (n == 0) {
      return 1;
   } else {
      return n * factorial(n - 1);
   }
}
```

Simulating the `factorial` Method



The Recursive Paradigm

- Most recursive methods you encounter in an introductory course have bodies that fit the following general pattern:

```
if (test for a simple case) {
   Compute and return the simple solution without using recursion.
} else {
   Divide the problem into one or more subproblems that have the same form.
   Solve each of the problems by calling this method recursively.
   Return the solution from the results of the various subproblems.
}
```

- Finding a recursive solution is mostly a matter of figuring out how to break it down so that it fits the paradigm. When you do so, you must do two things:
  1. Identify ***simple cases*** that can be solved without recursion.
  2. Find a ***recursive decomposition*** that breaks each instance of the problem into simpler subproblems of the same type, which you can then solve by applying the method recursively.

Exercise: A Recursive `gcd` Function

In the discussion of algorithmic methods in Chapter 5, one of the primary examples was Euclid's algorithm for computing the greatest common divisor of two integers, x and y. Euclid's algorithm can be implemented using the following code:

```
public int gcd(int x, int y) {
   int r = x % y;
   while (r != 0) {
      x = y;
      y = r;
      r = x % y;
   }
   return y;
}
```

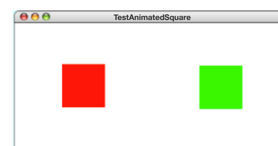
Rewrite this method so that it uses recursion instead of iteration, taking advantage of Euclid's insight that the greatest common divisor of x and y is also the greatest common divisor of the y and the remainder of x divided by y.

Concurrency

- One of the initial design criteria for Java was that it be able to support ***concurrency***, which is simply the ability to carry on several activities in parallel, even on computers that have only one processor.
- The classical approach to supporting concurrency is called ***multiprogramming***, in which a computer with a single processor runs multiple programs by sharing that processor among each of the individual ***processes***. The computer runs one process for a short period and then switches over to one of the other processes, cycling around the different tasks to ensure that they all get a fair share of processor time.
- Java supports concurrency at a lower level by allowing users to create new ***threads***, which are independent activities that coexist when the same program and share access to the same memory space. As in multiprogramming, a multithreaded system shares the processor among the active threads.

A Simple Concurrent Application

- The `TestAnimatedSquare` program on the next slide offers a simple illustration of multithreaded programming.
- Each of the squares shown in the sample run below is an instance of the `AnimatedSquare` class, which implements the `Runnable` interface which allows it to support a thread.
- The `run` method for each square causes it to move in a random direction, which changes every 50 steps.



The **`TestAnimatedSquare`** Program

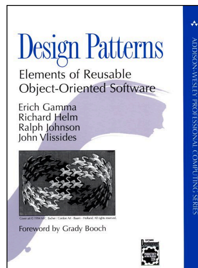
```
/**
 * This program tests the AnimatedSquare class by putting two squares
 * on the screen and having them move independently.
 */
public class TestAnimatedSquare extends GraphicsProgram {

   public void run() {
      double x1 = getWidth() / 3 - SQUARE_SIZE / 2;
      double x2 = 2 * getWidth() / 3 - SQUARE_SIZE / 2;
      double y = (getHeight() - SQUARE_SIZE) / 2;
      AnimatedSquare redSquare = new AnimatedSquare(SQUARE_SIZE);
      redSquare.setFilled(true);
      redSquare.setColor(Color.RED);
      add(redSquare, x1, y);
      AnimatedSquare greenSquare = new AnimatedSquare(SQUARE_SIZE);
      greenSquare.setFilled(true);
      greenSquare.setColor(Color.GREEN);
      add(greenSquare, x2, y);
      Thread redSquareThread = new Thread(redSquare);
      Thread greenSquareThread = new Thread(greenSquare);
      waitForClick();
      redSquareThread.start();
      greenSquareThread.start();
   }

/* Private constants */
   private static final double SQUARE_SIZE = 75;

}
```

The **`AnimatedSquare`** Class

```
/**
 * This class creates an animated square that has its own thread of control.
 * Once started, the square moves in a random direction every time step.
 * After CHANGE_TIME time steps, the square picks a new random direction.
 */
public class AnimatedSquare extends GRect implements Runnable {

/* Creates a new AnimatedSquare of the specified size */
   public AnimatedSquare(double size) {
      super(size, size);
   }

/* Runs when this object is started to animate the square */
   public void run() {
      for (int t = 0; true; t++) {
         if (t % CHANGE_TIME == 0) {
            direction = rgen.nextDouble(0, 360);
         }
         movePolar(DELTA, direction);
         pause(PAUSE_TIME);
      }
   }

/* Private constants */
   private static final double DELTA = 2;     /* Pixels to move each cycle       */
   private static final int PAUSE_TIME = 20;  /* Length of time step             */
   private static final int CHANGE_TIME = 50; /* Steps before changing direction */

/* Private instance variables */
   private RandomGenerator rgen = RandomGenerator.getInstance();
   private double direction;
}
```

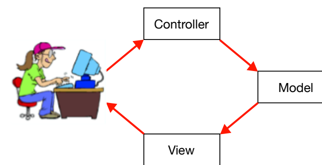
Programming Patterns

- In 1994, Addison-Wesley published *Design Patterns,* a groundbreaking book that called attention to the fact that many programming problems are most easily solved by applying certain common software patterns.
- Although the patterns described in the book are typically implemented using classes and methods, each pattern tends to represent more of a general solution strategy for some class of problems and not a ready-made solution.



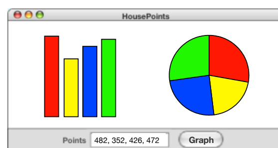
The Model/View/Controller Pattern

One of the most important patterns in terms of its applicability to user-interface design is the ***model/view/controller pattern,*** which is often abbreviated as **MVC**. The user interacts with a MVC-based user interface through the ***controller***, which is the part of the system capable of accepting user commands. It is typically a collection of Swing interactors. The controller never updates the displayed data directly. It instead sends requests to the ***model,*** which keeps track of the state of the system but is separate from the user interface. When changes occur in the model, it sends messages to one or more ***views***, which are responsible for updating the information that the user sees on the display.

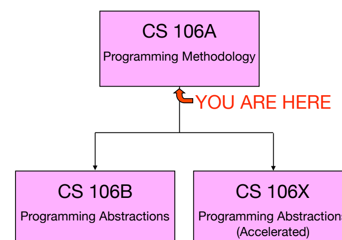


The **`HousePoints`** Program

- The **`HousePoints`** program described in section 14.4 uses the model/view/controller pattern to present two *views* of the house points assigned to the Hogwarts houses at the end of J. K. Rowling's *Harry Potter and the Philosopher's Stone*. The first appears as a bar graph, the second as a pie chart.
- In this example, the role of the controller is taken by the interactors at the bottom of the window. The controller sends messages to the model, which in turn updates both views.

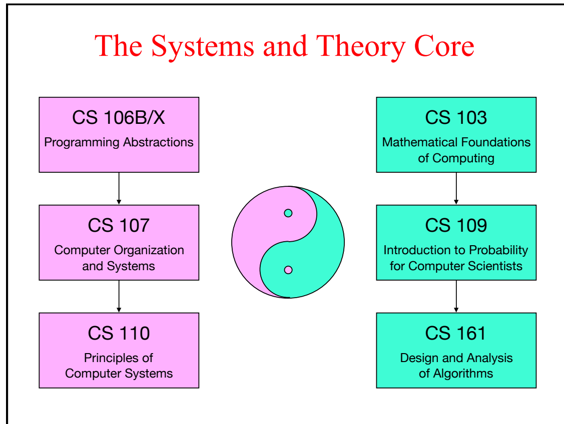


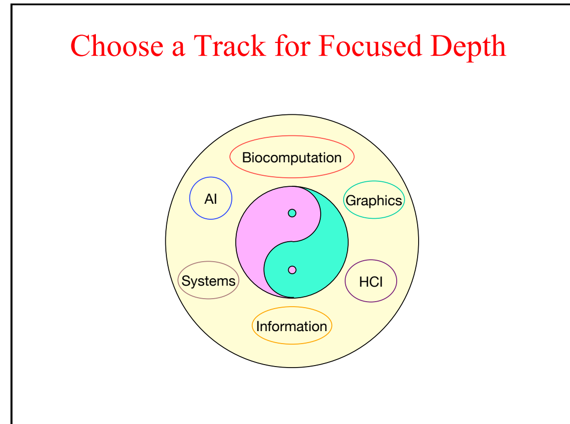
The Next Step

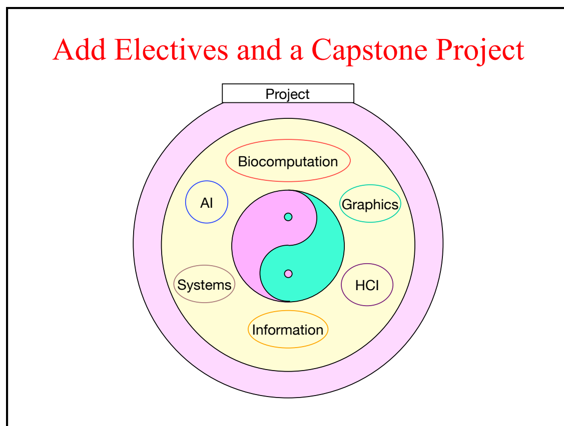


Continues from where we left off
Proceeds at much the same pace
Designed for a broad audience

Starts again from the beginning
Moves very fast
Attracts more "hot shots"


The Systems and Theory Core
CS 106B/X
Programming Abstractions
CS 107
Computer Organization and Systems
CS 110
Principles of Computer Systems
CS 103
Mathematical Foundations of Computing
CS 109
Introduction to Probability for Computer Scientists
CS 161
Design and Analysis of Algorithms


Choose a Track for Focused Depth
Biocomputation
AI
Graphics
Systems
HCI
Information


Add Electives and a Capstone Project
Project
Biocomputation
AI
Graphics
Systems
HCI
Information