Announcements

SUMMARY
In this assignment you will write a program to design stage lighting.

DEADLINE
This assignment is due on Wednesday, May 30 at 11:00 pm.

SPECIFICATION
You will create two classes to simulate a stage backdrop with colored lighting fixtures. The program will allow a user to see the beams from the light fixtures and the color-mixing that occurs. Create a package called `stagedesigner` with the following two public classes:

**Light**
The class `Light` will represent a single lighting fixture.

**Constructors:**
- `public Light(int x, int y, double direction, double angle, Color color)`
  - `x`... the coordinate of the light (origin of the beam)
  - `y`... the coordinate of the light (origin of the beam)
  - `direction`... the direction of the beam in radians
  - `angle`... the width of the beam in radians
  - `color`... the color of the beam

**Methods:**
- `public Color getColor()`
  - Return the color of the light fixture
- `public boolean inBeam(int px, int py)`
  - Return `true` if the coordinates (`px`, `py`) are in the beam, `false` otherwise.

Recursion
Recursion

- So far we have been using iteration to accomplish large tasks.
- How do we do something many times?
  - Use a loop.
- There is another way to repeat an action many times.
- Can we define the problem in terms of one (or more) subproblems of the same type?
  - Then we can use recursion.

Factorial

\[ n! = n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times 3 \times 2 \times 1 \]

\[ n \quad \times \quad (n-1)! \]

Factorial

- What is the solution to 10!?
  - \[ 10! = 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 \]
  - \[ 10! = 3,628,800 \]
- What about n!?
  - \[ n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 \]
  - But Java doesn’t understand ...
  - Can we define n! in terms of a subproblem of the same type?

\[ n! = n \times (n-1)! \]

\[ (n-1)! = (n-1) \times (n-2)! \]

\[ (n-2)! = (n-2) \times (n-3)! \]

- But this can’t go on forever.
  - What is \((-1)!\)?
- We need a base case
  - \[ 1! = 1 \]
Factorial

- So, in terms of Java:

```java
public static int fact(int n)
{
    if(n == 1)
        return 1;
    else
        return n * fact(n-1);
}
```

Time Complexity: \(O(n)\)

No Loops

- Notice our method to compute the factorial contains no loops.
- Recursive functions will (usually) not contain any loops.
- Instead they achieve the repeated application of some action through multiple successive recursive calls.

Key Parts

- Each recursive method must have the following key parts:
  - Base Case
    - The base case represents the simplest version of the problem.
    - e.g. \(1! = 1\)
    - Without the base case, the recursion will never stop.
  - Recursive Case
    - The recursive case includes a call to the same method we're currently in.
    - The recursive call must make the problem smaller.
    - Otherwise the base case will never be reached.
- There can be more than one of each
  - e.g. 2 base cases, 2 recursive cases, etc.
Factorial: Box Trace

n: 4
return: 24
n: 3
return: 6
n: 2
return: 2
n: 1
return: 1

Fibonacci

• In Java:

```java
public static int fib(int n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
    else return fib(n-1) + fib(n-2);
}
```

• In ML:

```ml
fun fib 0 = 0 |
    fib 1 = 1 |
    fib n = fib n-1 + fib n-2
```

Fibonacci Numbers

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 ...

• $f_0 = 0$
• $f_1 = 1$
• $f_n = f_{n-1} + f_{n-2}$

• This already looks recursive
  • How many base cases?
    • 2
  • How many recursive cases?
    • 1, with 2 recursive calls
Fibonacci

- In Scheme:

```scheme
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
)
```

Fibonacci: Box Trace

```
| n:   | 4
| ret: | 3
| n:   | 3
| ret: | 2
| n:   | 2
| ret: | 1
| n:   | 1
| ret: | 1
| n:   | 0
| ret: | 0
| n:   | 1
| ret: | 1
| n:   | 0
| ret: | 0
```

Time Complexity: $O(2^n)$
Traversing a Linked List: Iteration

Node current = head; // Start Point
while(current != null)
{
    // Do Something
    // ...
    current = current.next; // Advance
}
// What is current here? . . .  null

Traversing a linked list: Recursion

- Lets do a recursive traversal of a linked list.
- We’ll use Node in NoteLinkedList as an example and do something simple, like print out the pitch of each note.
- What does our method need to have?
  - Base Case:
    - List is empty
      - current == null
  - Recursive Case:
    - List is not empty, print and proceed
      - current.next

Traversing a Linked List: Iteration

Node current = head; // Start Point
while(current.next != null)
{
    // Do Something
    // ...
    current = current.next; // Advance
}
// What is current here? . . . The last node in the list
// Of course we have to be sure the list has at least one element.

Traversing a Linked List: Recursion

public void printPitches(Node node)
{
    if(node == null) {
        return;
    } else {
        System.out.println(node.note.getPitch());
        printPitches(node.next);
    }
}
Traversing a linked list: Recursion

Now lets do something a little more interesting, like sum the rhythm values.

What does our method need to have?

- Base Case:
  - List is empty
    - `current == null`
    - `return 0.0;`

- Recursive Case:
  - List is not empty, add current num to the sum of the rest
    - `current.note.getRhythmValue() + sumRhythmValues(current.next)`

```
public double sumRhythmValues(Node node)
{
    if (node == null) {
        return 0.0;
    } else {
        return node.note.getRhythmValue() + sumRhythmValues(node.next);
    }
}
```

Recursion: A Leap of Faith

- This brings up an important point about writing recursive methods.
  - Believe your method works.
  - Always assume that the recursive call will work.
  - This is hard to do, but essential to creating a recursive method.
  - Relates to the principle of mathematical induction.

Recursive Helpers

- Sometimes a method description doesn’t contain all the parameters we need for recursion.
- We can define another method with the necessary parameters to do the recursion.
  - Sometimes called a recursive helper method.
  - Can have the same name (method overloading)
    - Parameter list will be different.
  - Can be private
    - Won’t be called directly
  - The original method simply calls the helper method with the additional parameters.
Recursive Helpers

• For example, let’s add a recursive public method to `NoteLinkedList` to sum the rhythm values that has no parameters.
  
  • `public double sumRhythmValues();`
  
  • Method has no parameters (the list is a field of the object).
  
  However, in order to recursively traverse the list we need to track our current position.
  
  • So we need a helper method:
    
    • `private double sumRhythmValues(Node node);`
  
  • The helper method does the real work.
  
  • The original method just calls it.

Recursion

• Work can be done on the way out or the way back.
  
  • Before or after the recursive call.
  
  • What if I wanted to print out the list of elements in reverse?

---

Traversing a Linked List: Recursion

```java
public double sumRhythmValues()
{
    return sumRhythmValues(head);
}
private double sumRhythmValues(Node node)
{
    if (node == null) {
        return 0.0;
    } else {
        return node.note.getRhythmValue() +
               sumRhythmValues(node.next);
    }
}
```

Traversing a Linked List: Recursion

```java
public static void printPitches(Node node)
{
    if (node == null) {
        return;
    } else {
        System.out.println(node.note.getPitch());
        printPitches(node.next);
    }
}
```
Traversing a Linked List: Recursion

```java
public static void printPitches(Node node) {
    if (node == null) {
        return;
    } else {
        printPitches(node.next);
        System.out.println(node.note.getPitch());
    }
}
```