SUMMARY
In this assignment you will develop a class to work with solos in jMusic.

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

DESCRIPTION
A LITTLE MUSIC MATH
Music is very mathematical when you break it down. Don’t worry if you don’t have a music background. This section should bring you up to speed on what you need to know for the next part.

The key of a song describes which pitches fit into a song, and which do not. A key can be described by a root and a scale. The root is an integer value pitch that determines the start point of the scale. The scale is an array of ints that determines which offset pitches are “in-key.”

Let’s start with the major scale. The int array for the major scale is
\{0, 2, 4, 5, 7, 9, 11\}.

This is also given by JMC.MAJOR_SCALE.

Now, given any root \(N\), figuring out which pitches are in the key is just a simple math problem. The following pitches are in the key:
\(N, N+2, N+4, N+5, N+7, N+9, N+11\)
Which means, of course that the following pitches are not in the key:
\(N+1, N+3, N+6, N+8, N+10\)

Take the root of C for example. Using C4 = 60 as an example, the following pitches are in the key of C Major:
60, 62, 64, 65, 67, 69, 71
Also, C5 = 72, so the following pitches are also in the key of C Major:
72, 74, 76, 77, 79, 81, 83

In fact, adding or subtracting 12 to any root gives the same note letter in a different octave, so we can use this formula to determine if any pitch \(P\) (0 ≤ P ≤ 127) is in the key or not.
Furthermore, this can be done with any root \(R\) (0 ≤ R ≤ 127), and any scale given as an array of ints \(S = \{S_1, S_2, ..., S_n\}\) where 0 ≤ \(S_i\) ≤ 11 for 0 ≤ i ≤ n.
Consider the following two data structures:

n integers stored in an array (of length n)

n integers stored in a linked list (with n nodes)

Which requires more space in memory?

A. The array.
B. The linked list.
C. They both require the same.

### SIZE COMPARISON: N INTS

- The array requires
  - Reference... 4 bytes
  - (int... 4 bytes) * n
  - = 4n + 4
- The linked list requires
  - Reference... 4 bytes
  - (int... 4 bytes + Reference... 4 bytes) * n
  - = 8n + 4
- Of course, this overhead becomes relatively smaller with larger data types.

### SIZE COMPARISON: N OBJECTS

- Say we have an object of size 64 bytes
- The array requires
  - Reference... 4 bytes
  - (Object... 64 bytes) * n
  - = 64n + 4
- The linked list requires
  - Reference... 4 bytes
  - (Object... 64 bytes + Reference... 4 bytes) * n
  - = 68n + 4
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

- 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?

FACTORIAL

$n! = n \times (n-1)!$

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

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

- 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 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 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);
  }
  ```

**FIBONACCI**

- In ML:
  ```ml
  fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib n-1 + fib n-2
  ```
• In Scheme:

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

FIBONACCI: BOX TRACE

```
<table>
<thead>
<tr>
<th>n: 4</th>
<th>ret: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>n: 3</td>
<td>ret: 2</td>
</tr>
<tr>
<td>n: 2</td>
<td>ret: 1</td>
</tr>
<tr>
<td>n: 2</td>
<td>ret: 1</td>
</tr>
<tr>
<td>n: 1</td>
<td>ret: 1</td>
</tr>
<tr>
<td>n: 1</td>
<td>ret: 0</td>
</tr>
<tr>
<td>n: 0</td>
<td>ret: 0</td>
</tr>
</tbody>
</table>
```

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

// What is current here? . . . null

The last node in the list
// Of course we have to be sure
// the list has at least one element.

TRAVERSING A LINKED LIST: ITERATION

Node current = head; // Start Point
while (current.next != 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

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 int 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 array is a field of the object).
  - However, in order to recursively traverse the array 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.

TRAVERSAING A LINKED LIST: RECURSION

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

```
public static void printPitches(Node node) {
    if (node == null) {
        return;
    } else {
        System.out.print(node.note);
        printPitches(node.next);
    }
}
```
IClicker Question

Copying only the reference to an object is called

A. Reference Copy
B. Deep Copy
C. Shallow Copy
D. Object Copy
E. Copy Copy