Lecture 2: More Recursion than OOP

Recursion Exercises

1. What does the function p below output when you call p(3)?

   ```java
   static void p(int n)
   {
     if (n == 0) return;
     System.out.printf("Before: %d\n", n);
     p(n-1);
     System.out.printf("After: %d\n", n);
   }
   ```

2. Write a recursive implementation of factorial. (*) What will your implementation do for a negative value?

3. High/Low: Your friend picks a number between 1 and n, inclusive. You keep guessing the number. For each guess he gives one of the three following answers: correct, too high, too low.

   (a) How many guesses will you need (at most) when n = 3, 7, 15?

   (b) In Θ-notation, how many guesses will you need for arbitrary n?

   (c) Suppose you have a sorted array of n ints. How long will it take to find whether a value v is in the array?

4. (*** ) What does the function h below compute? Also gives its runtime and memory usage.

   ```java
   static int hHelper(int[][] arr, int left, int right)
   {
     if (left == right) return arr[left];
     int mid = (left + right)/2;
     int L = hHelper(arr,left,mid), R = hHelper(arr,mid+1,right);
     return L + R;
   }
   static int h(int[][] arr)
   {
     return hHelper(arr,0,arr.length-1);
   }
   ```
Solutions

1. We obtain the output
   Before: 3
   Before: 2
   Before: 1
   After: 1
   After: 2
   After: 3

2. Code is listed below:

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

   We could also use a tertiary condition and write

   ```java
   static int factorial(int n) { return n == 0 ? 1 : n * factorial(n - 1); }
   ```

   If we attempt to call factorial(−1) then the program will keep recursing until we run out of stack space and the program exits (called a stack overflow).

3. (a) 2, 3, 4, respectively by always choosing the middle value. More generally we can do \( n = 2^{k+1} - 1 \) in \( k \) steps.

   (b) Each time we roughly halve the number of values we must consider. At worst we must reduce the number of possible values to a single number. This will give our \( \Theta(\lg n) \) runtime.

   (c) \( \Theta(\lg n) \). Play the High/Low game with the indices of the array. This is called binary search. Here is a recursive and an iterative implementation:

   ```java
   public class BinarySearch {
     public static int binarySearch(int[] arr, int L, int R, int v) {
       if (L > R) return -1;
       int M = (L+R)/2;
       if (v == arr[M]) return M;
       if (v < arr[M]) return binarySearch(arr,L,M-1,v);
       else return binarySearch(arr,M+1,R,v);
     }
   }
   ```
public static int binarySearch(int[] arr, int v) {
    return binarySearch(arr, 0, arr.length - 1, v);
}

public static int binarySearchIter(int[] arr, int v) {
    int L = 0, R = arr.length - 1;
    while (L <= R) {
        int M = (L + R) / 2;
        if (arr[M] == v) return M;
        if (v < arr[M]) R = M - 1;
        else L = M + 1;
    }
    return -1;
}

4. The function $h$ computes the sum of the given array. More precisely, $h$Helper computes the sum of the entries of $arr$ between the indices left and right, inclusive. To do this it splits the array in half, and recurses on each half. The memory usage is $\Theta(\lg n)$ where $n$ is the length of $arr$. To see why, note that each recursive call halves the number of indices we are summing. It takes $\lg n$ halvings to reduce the interval to size 1, the base case. There are a few ways to see the runtime is $\Theta(n)$. One is to see that $L + R$ will occur $n - 1$ times: one for each plus in the following expression

$\text{arr}[0] + \text{arr}[1] + \text{arr}[2] + \ldots + \text{arr}[n-2] + \text{arr}[n-1]$

Here we will give some intuition behind the running time and the memory usage for those that know what a binary tree is. We will come back to this later in the semester. Consider a complete binary tree of height $k$. It has

$$1 + 2 + 4 + 8 + \cdots + 2^k = \sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

nodes and $2^k$ leaves. The two things to notice are that there are roughly half as many leaves as nodes, and the height is the logarithm of the number of leaves. The logarithm that shows up here is essentially the source of all logarithms we will encounter in this class. In this particular example we have an implicit tree formed by breaking each interval into its left and right halves. There will be $n$ leaves so the height will be roughly $\log n$ ($n$ may not be a power of 2). Furthermore, the total number of nodes will be less than double the number of leaves, so the runtime will be $\Theta(n)$. 

3
As we have seen, recursive implementations can be more costly than iterative (loop based) implementations. There are some languages that have optimizations to improve the space and performance of certain recursive calls (such as tail-call optimization; you can look this up if you are interested). Later in the course we will use recursion when dealing with trees.

### OOP Segue Exercise

1. You are given as input (through standard input, i.e., System.in) earthquake data. The first line contains a single integer \( n \), the number of earthquakes. Each of the \( n \) following lines has the format:
   
   City,Date,Magnitude

   City and Date are just strings you can store. Magnitude will be a decimal value. An example of the input will be:

   2
   Valdivia,5/22/1960,9.5
   Kamchatka,10/17/1737,8.5

   Write a program that reads the input and outputs (in the format below) the city, date, and magnitude of the largest magnitude earthquake given. If there is a tie, output any of the largest ones. On the above example you should output:

   City = Valdivia
   Date = 5/22/1960
   Magnitude = 9.5

   It’s ok if you don’t remember how to do the input in Java. Focus on how the program would be structured.

### Solutions

1. In this problem we review I/O and start to talk about objects. Listed below is a solution.

   Earthquakes.java

   ```java
   import java.util.Scanner;
   public class Earthquakes {
     public static int findMax(double[] arr) {
       int pos = 0;
       for (int i = 1; i < arr.length; ++i) {
         if (arr[i] > arr[pos]) pos = i;
     ```
public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    int n = in.nextInt();
    in.nextLine(); // Why?
    String[][] cities = new String[n];
    String[][] dates = new String[n];
    double[] magnitudes = new double[n];
    for (int i = 0; i < n; ++i) {
        String line = in.nextLine();
        String[] parts = line.split(",");
        cities[i] = parts[0];
        dates[i] = parts[1];
        magnitudes[i] = Double.parseDouble(parts[2]);
    }
    int pos = findMax(magnitudes);
    System.out.printf("City = %s\n",cities[pos]);
    System.out.printf("Date = %s\n",dates[pos]);
    System.out.printf("Magnitude = %f\n",magnitudes[pos]);
}

We decided to split out findMax into its own function for cleanliness. Note that on line 17 we need to call in.nextLine(). This is because the initial call to nextInt on line 16 reads the integer n but doesn’t read the newline after it on the first line. The call to nextLine on line 17 finishes reading the first line (it will return ",", an empty string, which we discard). On line 24 we use the split method of String. It accepts a pattern (called a regular expression) and uses it to match delimeters. It returns an array of the substrings that have been separated by the given delimeter we pattern. We won’t have time to delve into regular expressions, but as an example of their usefulness we could have written the following code which splits a String using commas surrounded by optional whitespace:

    String[] parts = line.split("\s*,\s*");

Objects

Most of the material here is also discussed in Liang’s book on Java Programming (chapters 9-13 in the 10th edition).

In the Earthquakes program above we had separate arrays for dates, cities, and magnitudes. Instead we can create a class called Earthquake to represent our data.
Earthquake.java

```java
public class Earthquake {
    public String city;
    public String date;
    public double magnitude;
    public Earthquake(String c, String d, double m) {
        city = c;
        date = d;
        magnitude = m;
    }
}
```

EarthquakesObj.java

```java
import java.util.Scanner;
public class EarthquakesObj {
    public static int findMax(Earthquake[] arr) {
        int pos = 0;
        for (int i = 1; i < arr.length; ++i) {
            if (arr[i].magnitude > arr[pos].magnitude) pos = i;
        }
        return pos;
    }
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = Integer.parseInt(in.nextLine().trim());
        Earthquake[] earthquakes = new Earthquake[n];
        for (int i = 0; i < n; ++i) {
            String line = in.nextLine();
            String[] parts = line.split(",");
            Earthquake e = new Earthquake(parts[0],parts[1],Double.parseDouble(parts[2]));
            earthquakes[i] = e;
        }
        int pos = findMax(earthquakes);
        Earthquake big = earthquakes[pos];
        System.out.printf("City = %s\n",big.city);
        System.out.printf("Date = %s\n",big.date);
        System.out.printf("Magnitude = %f\n",big.magnitude);
    }
}
```
The above code is cleaner than our original implementation since we only have a single array earthquakes. You can see that if an earthquake had 10 properties, storing 10 arrays could become very annoying. Secondly, we now have the concept of an Earthquake in our program. One of the virtues of OOP (object-oriented programming) is that some of the important concepts in our mental view of a problem can be represented as objects in a program.

Here we have used the notion of a compound data structure or record. Our objects are just a way of taking multiple pieces of data and aggregating them in one location. OOP extends this usage of objects by allowing us to mix behavior with data, and provides tools to enable this method of programming/design.

As another example, here is a Student class.

```java
public class Student {
    public static int numStudents;
    public String name;
    public String email;
    public int credits;

    public Student(String n, String e) {
        name = n;
        email = e;
        numStudents++;
    }

    public String getName() { return name; }
    public String getEmail() { return email; }
    public int getCredits() { return credits; }
    public void addCredits(int creds) { credits += creds; }
    public static int getNumStudents() { return numStudents; }
    public String toString() {
        return String.format("Student: %s, %s, %d", name, email, credits);
    }
}
```

As with the Earthquake class, the Student class defines a data type. Each object of type Student will have 3 fields: name, email, credits. We also have a class variable called numStudents and a class method named getNumStudents. The class has 6 methods. The first method above is called a constructor, and is used when you create objects of type Student. The last method (toString) is what is used by Java to turn your object into a String when needed. This is what allows us to use System.out.println(s) below. Here are some code snippets that use this class.
Both of these snippets show how objects of type Student can be used. Let’s first go into detail on what is occurring in Snippet1. On line 1 we create a variable of type Student. All types we create by defining classes are reference types: a variable of a reference type is a reference or pointer to the value as opposed to holding the value itself (in contrast with a primitive data type like int or double). The variable s points at an object of type Student which we create using its constructor. Note that credits is not given a value in the constructor. Any field we do not initialize is given the value 0 (or null if its a reference type). The constructor also increments the static variable numStudents. Since numStudents is static, it belongs to the class and not each instance. This allows numStudents to accurately count the number of students that have been created.

Each object of type Student will have 3 fields. Note that the 3 fields are declared in the Student class without using the keyword static. This means that each object of type Student will have its own variables named email, name, and credits. If we had used the keyword static we would have declared a class variable. On line 2 we call the addCredits method with argument 150. Note that when we defined addCredits in class Student, we didn’t use the keyword static. This means that addCredits is used on objects (also called instances) of type Student as is shown in the snippet.

We can imagine that Snippet2 exists somewhere in another file and also uses the Student class. Snippet2 and Snippet1 seem similar in what they do. Is there a reason to prefer one over the other? To see why Snippet1 is preferrable, suppose we have to change the Student class as follows: No student is allowed to have more than 130 credits; if credits are added to violate this, the student is capped at 130 credits. To implement this we modify addCredits:

```java
public void addCredits(int creds) {
    credits += creds;
    if (credits > 130) credits = 130;
}
```

After the change Snippet1 continues to work as desired but Snippet2 is broken. If you imagine we have a huge program with hundreds of lines of code that use Student in the manner that Snippet2 does, then implementing this change would require a lot of work. As another example, imagine we wanted to change to using first and last name instead of just
name. Then code using getName would be fine since we can fix the method, but any code
directly accessing the name field would all break. To prevent Snippet2 from happening we
introduce access modifiers.

```java
public class Student {
    private static int numStudents;
    private String name;
    private String email;
    private int credits;

    public Student(String n, String e) {
        name = n;
        email = e;
        numStudents++;
    }

    public String getName() { return name; }
    public String getEmail() { return email; }
    public int getCredits() { return credits; }
    public void addCredits(int creds) { credits += creds; }
    public static int getNumStudents() { return numStudents; }
    public String toString() {
        return String.format("Student: %s, %s, %d", name, email, credits);
    }
}
```

Now Snippet2 will not compile since the private modifier means that those fields can only
be accessed from code within class Student. Modifiers can be put on fields, static variables,
methods and static methods. The other two modifiers are package and protected which we
will see later. Classes can also carry access modifiers, but we won't discuss this now.

Access modifiers can be thought of as compiler error generators. You may be at the
point in your programming career where you hate compiler errors and generating more of
them sounds horrible. Part of becoming better as a programmer is learning to love compiler
errors. The hope is that we generate more compiler errors and wind up with fewer runtime
errors.

By making the fields of an object private we are restricting the way in which users of
our class can access it. This is part of what is called encapsulation. Except in a few rare
cases (such as academic examples) we will make all of our fields private (or maybe protected;
see later). To a user, all that matters is which methods are public and what their stated
behaviour is.

**Object Exercises**

1. Assume you have the following Stock class already written:

```java
public class Stock
```
private String name;
private double price;

public Stock(String n, double p) { name = n; price = p; }

public String getName() { return name; }
public double getPrice() { return price; }

public void setPrice(double p) { price = p; }

public String toString() {
    return String.format("%s $%f", name, price);
}

We left out comments in the code for brevity. Design a class Position that has a Stock
stock and an int quantity as members. Give a constructor, and accessor methods (i.e.,
getters) for the fields. Also give a method getValue that returns a double indicating
the value of the position (quantity times price). Then create a class called TestPosition
that has a main method for creating a sample Position and outputs the value of each
method call.

2. In the following code, say what effect each of the 4 swap methods has.

IntValue.java

public class IntValue {
    public int val;
    public IntValue(int v) { val = v; }
}

SwapExample.java

import java.util.Arrays;

public class SwapExample {
    public static void swap(int a, int b) {
        int tmp = a;
        a = b;
        b = tmp;
    }

    public static void swap(int[] a, int[] b) {
        int[] tmp = a;
        a = b;
        b = tmp;
    }

    public static void swap(int[] arr, int i, int j) {
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }
}
public static void swap(IntValue v, IntValue w) {
    int tmp = v.val;
    v.val = w.val;
    w.val = tmp;
}

public static void main(String[] args) {
    int a = 4, b = 5;
    int[] arr = {0,1,2,3};
    int[] arr2 = {4,5,6,7};
    int[] arr3 = arr2;
    swap(a,b);
    swap(arr,arr2);
    swap(arr,0,1);
    arr3[0] = 9;
    IntValue x = new IntValue(1), y = new IntValue(2);
    swap(x,y);
    System.out.printf("a=%d,b=%d\n",a,b);
    System.out.printf("arr=%s\n",Arrays.toString(arr));
    System.out.printf("arr2=%s\n",Arrays.toString(arr2));
    System.out.printf("arr3=%s\n",Arrays.toString(arr3));
    System.out.printf("x.val=%d,y.val=%d\n",x.val,y.val);
}

Object Solutions

1. The code follows.

Position.java

public class Position {
    private Stock stock;
    private int quantity;
    public Position(Stock stock, int q) {
        this.stock = stock;
        quantity = q;
    }
    public int getQuantity() { return quantity; }
    public Stock getStock() { return stock; }
    public double getValue() { return quantity*stock.getPrice(); }
}

TestPosition.java

public class TestPosition {
    public static void main(String[] args) {
The code above has no methods that allow you to set the fields. In other words, once the objects are constructed, they will never change. These kinds of objects are called **immutable**. Another example of immutable objects are Java Strings.

2. The first two swap methods have no effect due to pass-by-value. The third swaps the values in positions \( i, j \) of the array referred to by \( \text{arr} \). The fourth swaps the values in the \( \text{val} \) fields of \( \text{v}, \text{w} \), respectively. To further illustrate this point, note that main will output the following.

\[
\begin{align*}
\text{a} &= 4, \text{b} = 5 \\
\text{arr} &= [1, 0, 2, 3] \\
\text{arr2} &= [9, 5, 6, 7] \\
\text{arr3} &= [9, 5, 6, 7] \\
\text{x.val} &= 2, \text{y.val} = 1
\end{align*}
\]

Also important to notice is that line 31 changed the arrays referenced by \( \text{arr2} \) and \( \text{arr3} \).

### Inheritance and Polymorphism

For our next topic today on OOP we look at inheritance. Earlier we had a Student class. Suppose we want to create a class called CIMSStudent that has all of the fields of Student, but adds on the two extra String fields cimsLogin and cimsEmail. When creating the new class CIMSStudent we could copy and paste all of the code in Student and just add a few more fields. More often than not, if you are programming and doing a lot of copying and pasting you are doing something wrong. Java provides a way to say one class is an extension (or a more specialized form) of another class:

```
CIMSSStudent.java
1 public class CIMSSStudent extends Student
2 {
3     private String cimsEmail;
4     private String cimsLogin;
5     public CIMSSStudent(String name, String email,
6                             String cEmail, String cLogin)
7     {
8         super(name, email);
9         cimsEmail = cEmail;
```
```java
cimsLogin = cLogin;
public String getCIMSEmail() { return cimsEmail; }
public String getCIMSLLogin() { return cimsLogin; }
public String toString()
{
    //String n = name; //Doesn't compile
    return String.format("CIMSStudent: %s, %s, %s, %d\n",getName(),
                        cimsEmail,cimsLogin, getCredits());
}
}
```

The first thing to notice is the extends keyword on line 1. We say the class CIMSStudents extends (or inherits from, or derived from, or is a subclass of, or is a child of) the class Student. Conversely we say Student is the superclass (or parent, or base class) of CIMSStudent. Here we are saying that every object of type CIMSStudent also has all the fields a Student object would have.

On line 8 we use the keyword super. The first thing any constructor does is call the constructor of its parent class. If we don’t provide the super keyword, it calls the no-argument constructor of the parent class if it exists (if it doesn’t exist, a compiler error occurs). By using the super keyword we explicitly state which constructor to call, and what arguments to pass in. Before we explain the rest, let’s give a code snippet using CIMSStudent:

```
Snippet
CIMSSStudent s = new CIMSSStudent("A","a@a.com",
                                    "a@cims.nyu.edu","a1234");
System.out.println(s.get CIMSEmail());
System.out.println(s.getName());
System.out.println(s);
```

As you can see in the snippet the method getName can be called on objects of type CIMSSStudent. In fact, all public fields or methods of any ancestor are accessible. This is what we had in mind when we extended Student. Next we come to line 16 of CIMSStudent. The reason that line will not compile is that private members of a class are not accessible in its children. Thus to access the name and credits fields of the parent class we use the accessor (get) methods we wrote earlier. Stated differently, a CIMSSStudent object does have the name, email, and credits field from its parent, but it cannot access/see them directly due to the private access modifier. If you want to create a field that your children can see, but people outside your inheritance hierarchy cannot you use the protected keyword. [This is almost true. Technically, protected means any descendent or anyone in your package has access.]

The final thing we must address is line 14. The method toString occurs in both the parent Student and child CIMSSStudent (with the same arguments, i.e., none). This is called method overriding and is a very important part of OOP. The following illustrates some of
the true power of inheritance: polymorphism.

Snippet

```java
Student s = new CIMSSStudent("A","a@a.com",
    "a@cims.nyu.edu","a1234");
//System.out.println(s.getCIMSEmail()); //Doesn't compile
System.out.println(s.getName());
String t = s.toString(); //Calls the CIMSSStudent version
System.out.println(t);
```

On line 1 we create a CIMSSStudent object and our Student variable points at it. Java allows a reference of type T to point at objects whose type is any direct or indirect subclass of T (i.e., you can point at children, grandchildren, etc.). Here we say the reference s has type Student (or the static type is Student) and the object it points at has type CIMSSStudent (or the dynamic type is CIMSSStudent). On line 2 we would get a compiler error since Students don’t have a method getCIMSEmail. Thus, the name and arguments of the methods we can call must match the available methods on the type of the variable (and not the type of the object it points at). On line 4 we call the getName method which calls the getName method of Student. On line 5 we call the CIMSSStudent version of toString. This gives the following:

1. The type of the variable (i.e., reference) is used to determine (at compile time) what method calls are allowed, **BUT**

2. The version of the method is determined by the object. The one chosen is either in the object’s type, or if it doesn’t exist, in its closest ancestor.

This behavior where the runtime type of the object determines which method is used is called polymorphism. It is extremely powerful.

**Inheritance and Polymorphism Exercises**

1. Consider the following code.

```java
class Grandparent {
    public Grandparent() {
        System.out.println("Constructing Grandparent");
    }
    public int getValue() { return -1; }
}
class Parent extends Grandparent {
    protected int pi;
    public Parent(int i) {
        System.out.println("Constructing Parent "+i);
        pi = i;
    }
```
```java
public void printName() {
    System.out.printf("P %d %d\n", pi, getValue());
}

public class Child extends Parent {
    private int ci;
    public Child(int i, int pi) {
        super(pi);
        System.out.println("Constructing Child "+i);
        ci = i;
    }
    public int getValue() { return ci; }
    public void printName() {
        System.out.printf("C %d %d %d\n", pi, ci, getValue());
    }
    public static void main(String[] args) {
        Parent p = new Parent(4);
        Parent c = new Child(6, 7);
        Grandparent c2 = c;
        p.printName();
        c.printName();
        //c2.printName(); //Compiler error
        System.out.printf("%d %d %d\n", p.getValue(),
                         c.getValue(), c2.getValue());
        System.out.println(c);
    }
}
```

(a) What does it mean when you don’t have public in front of a class?

(b) What does it output when we run Child?

2. Why does System.out.println(v) work for any object v?

Inheritance and Polymorphism Solutions

1. (a) It means the class has package access. That is, can be accessed from other classes in the same package. At this point we haven’t created any explicit packages so all of our classes are in the default package which has no name. Two classes are in the same unnamed package if they share a folder on the file system (technical: not always true, but true in nearly every case; see Java Language Spec 8 section 7.4.2).

(b) The output is as follows:

    Constructing Grandparent
Line 29 generates lines 1,2 of the output. Line 30 generates lines 3-5 of the output. Lines 6,7 are the two printName calls. Line 8 shows that Parent objects use the Grandparent getValue and Child object’s use the Child’s getValue (independent of the reference type). Line 9 shows what happens when you don’t override toString.

A few other things to note here:

- Child’s printName can access pi since it is protected.
- All calls to getValue use the corresponding object’s version.
- getValue cannot be called using a reference of type Grandparent.

2. System.out is a type of PrintStream (more precisely, out is a public static field of type PrintStream of the class System). All PrintStreams have the following method:

```java
In java.io.PrintStream

public void print(Object obj) {
    write(String.valueOf(obj));
}
```

which calls the static String method:

```java
In java.lang.String

public static String valueOf(Object obj) {
    return (obj == null) ? "null" : obj.toString();
}
```

The write method of PrintStream does the actual writing and is omitted. The key here is that print (and println too) have an overloaded version that takes an Object. A few points on this:

- All reference types (including arrays) descend from Object. When you don’t explicitly extend a class you end up extending Object instead.
- Object has various useful methods such as toString, hashCode, equals and others. These have default implementations that are frequently overridden.
- PrintStreams have a overloaded print methods that take Objects and Strings. If you call System.out.print("Hello") it knows to call the String version since it will always choose the closest matching version of the function. Here closest means
prefering your type over parents, parents over grandparents, etc (i.e., closest ance-cestor in the type hierarchy).