Why flip recitations?

Usual way. 50-minute lecture, then study on your own. One hour? Total of, say, 2 hours.

Disadvantages:
- Hard to listen attentively for 50 minutes. Many people tune out, look at internet, videos, whatever
- Much time wasted here and there
- You don’t always know just how to study. No problem sets, and if there are, no easy way to check answers.
- Study may consist of reading, not doing. Doesn’t help.

Flipped way. Watch short, usually 3-5 minute, videos on a topic. Then come to recitation and participate in solving problems.

Disadvantage: If you don’t study the videos carefully, you are wasting your time.

Advantages
- Break up watching videos into shorter time periods.
- Watch parts of one several times.
- In recitation, you get to DO something, not just read, and you get to discuss with a partner and neighbors, ask TA questions, etc.

One student’s reaction, in an email

… I really enjoyed today’s activity and found it extremely effective in gaining a strong understanding of the material. The act of discussing problems with fellow classmates made me aware of what topics I was not as strong in and gave me the opportunity to address those areas immediately. What I enjoyed most about it, however, was working collaboratively with my peers.

I wanted to give you my feedback because I can see these interactive lessons becoming very effective if implemented again in the future.

== versus equals

Use `p1 == p2` or `p1 != p2` to determine whether p1 and p2 point to the same object (or are both null).

NEVER use `p1.equals(p2)` for this purpose, because it doesn’t always tell whether they point to the same object! It depends on how equals is defined.

```
p1 == p2   true
p3 == p1   false
p4 == p1   false
```
Sum the digits in a non-negative integer

```java
/** = sum of digits in n.
 * Precondition: n >= 0 */
public static int sum(int n) {
    // { n has at least two digits }
    // return first digit + sum of rest
    return n%10 + sum(n/10);
}
```

E.g. sum(7) = 7
E.g. sum(8703) = 3 + sum(870);

Two issues with recursion

```java
/** return sum of digits in n.
 * Precondition: n >= 0 */
public static int sum(int n) {
    // { n has at least two digits }
    // return first digit + sum of rest
    return n%10 + sum(n/10);
}
```

1. Why does it work? How does execution work?
2. How do we understand a given recursive method or how do we write/develop a recursive method?

Stacks and Queues

Stack: list with (at least) two basic ops:
- Push an element onto its top
- Pop (remove) top element
Last-In-First-Out (LIFO)
Like a stack of trays in a cafeteria

Queue: list with (at least) two basic ops:
- Append an element
- Remove first element
First-In-First-Out (FIFO)

Stack Frame

A “frame” contains information about a method call:
At runtime Java maintains a stack that contains frames for all method calls that are being executed but have not completed.

Method call: push a frame for call on stack. Assign argument values to parameters. Execute method body. Use the frame for the call to reference local variables and parameters.

End of method call: pop its frame from the stack; if it is a function leave the return value on top of stack.

Frames for methods sum main method in the system

```java
public static int sum(int n) {
    if (n < 10) return n;
    return n%10 + sum(n/10);
}
```

```java
public static void main(String[] args) {
    int r= sum(824);
    System.out.println(r);
}
```

Example: Sum the digits in a non-negative integer

```java
public static int sum(int n) {
    if (n < 10) return n;
    return n%10 + sum(n/10);
}
```

```java
public static void main(String[] args) {
    int r= sum(824);
    System.out.println(r);
}
```

Frame for method in the system that calls method main

Frame for method in the system that calls method main: main is then called
Example: Sum the digits in a non-negative integer

```java
public static int sum(int n) {
    if (n < 10) return n;
    return n%10 + sum(n/10);
}

public static void main(String[] args) {
    int r = sum(824);
    System.out.println(r);
}
```

Method main calls sum:

```
8  n 824  return info
2  n 82  return info
1  n 8  return info
```

Using return value 8 stack computes
\(2 + 8 = 10\) pops frame from stack puts return value 10 on stack

Example: Sum the digits in a non-negative integer

```
public static int sum(int n) {
    if (n < 10) return n;
    return n%10 + sum(n/10);
}

public static void main(String[] args) {
    int r = sum(824);
    System.out.println(r);
}
```

```
82  n 82 return info
24  n 824 return info
8  n 82 return info
```

Using return value 8 stack computes
\(2 + 8 = 10\) pops frame from stack puts return value 10 on stack
Example: Sum the digits in a non-negative integer

```java
public static int sum(int n) {
    if (n < 10) return n;
    return sum(n/10) + n%10;
}

public static void main(String[] args) {
    int r = sum(824);
    System.out.println(r);
}
```

Using return value 14 main stores 14 in r and removes 14 from stack.

Questions about local variables

```java
public static void m(...) {
    ...
    while (...) {
        int d= 5;
        ...
    }
}
```

In a call m(...) when is local variable d created and when is it destroyed? Which version of procedure m do you like better? Why?

Two views of recursive methods

- How are calls on recursive methods executed?
  We saw that. Use this only to gain understanding / assurance that recursion works.
- How do we understand a recursive method — know that it satisfies its specification? How do we write a recursive method?
  This requires a totally different approach. Thinking about how the method gets executed will confuse you completely! We now introduce this approach.

How to understand what a call does

Make a copy of the method spec, replacing the parameters of the method by the arguments

```java
/** = sum of the digits of n.
* Precondition:  n >= 0 */
public static int sum(int n) {
    if (n < 10) return n;
    // n has at least two digits
    return n%10 + sum(n/10);
}
```

Recursion is used extensively in math

Math definition of n factorial

\[ 0! = 1 \]
\[ n! = n \times (n-1)! \] for \( n > 0 \)

E.g. \( 3! = 3 \times 2 \times 1 = 6 \)

Easy to make math definition into a Java function!

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

Math definition of \( b^c \) for \( c \geq 0 \)

\[ b^0 = 1 \]
\[ b^c = b \times b^{c-1} \] for \( c > 0 \)

Lots of things defined recursively: expression grammars trees ....
We will see such things later.

Memorize method call execution!

A frame for a call contains parameters, local variables, and other information needed to properly execute a method call.

To execute a method call:
1. push a frame for the call on the stack,
2. assign argument values to parameters,
3. execute method body,
4. pop frame for call from stack, and (for a function) push returned value on stack

When executing method body look in frame for call for parameters and local variables.

Two views of recursive methods

- How are calls on recursive methods executed?
  We saw that. Use this only to gain understanding / assurance that recursion works.
- How do we understand a recursive method — know that it satisfies its specification? How do we write a recursive method?
  This requires a totally different approach. Thinking about how the method gets executed will confuse you completely! We now introduce this approach.

How to understand what a call does

Make a copy of the method spec, replacing the parameters of the method by the arguments

```java
/** = sum of the digits of n.
* Precondition:  n >= 0 */
public static int sum(int n) {
    if (n < 10) return n;
    // n has at least two digits
    return n%10 + sum(n/10);
}
```
Step 1. Have a precise spec!

Step 2. Check that the method works in the base case(s): That is, Cases where the parameter is small enough that the result can be computed simply and without recursive calls.

If \( n < 10 \) then \( n \) consists of a single digit. Looking at the spec we see that that digit is the required sum.

Step 3. Look at the recursive case(s). In your mind replace each recursive call by what it does according to the method spec and verify that the correct result is then obtained.

Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the parameters of the method.

** = sum of digits of \( n \).
* Precondition: \( n >= 0 \)

```java
public static int sum(int n) {
    if (n < 10) return n; // n has at least two digits
    return n%10 + sum(n/10); // e.g. n = 843
}
```

Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the parameters of the method.

\( n/10 \ < \ n \)

For the rest of the class we demo writing recursive functions using the approach outlined below. The java file we develop will be placed on the course webpage some time after the lecture.

Step 1. Have a precise spec!

Step 2. Write the base case(s): Cases in which no recursive calls are needed. Generally for “small” values of the parameters.

Step 3. Look at all other cases. See how to define these cases in terms of smaller problems of the same kind. Then implement those definitions using recursive calls for those smaller problems of the same kind. Done suitably, point 4 (about termination) is automatically satisfied.

Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the parameters of the method.
The Fibonacci Function

Mathematical definition:
\[
fib(0) = 0 \\
fib(1) = 1 \\
fib(n) = fib(n - 1) + fib(n - 2) \quad n \geq 2
\]

Fibonacci sequence: 0 1 1 2 3 5 8 13 ...

Fibonacci (Leonardo Pisano) 1170-1240?
Statue in Pisa Italy
Giovanni Paganucci 1863

Check palindrome-hood

A String palindrome is a String that reads the same backward and forward.
A String with at least two characters is a palindrome if
- (0) its first and last characters are equal and
- (1) chars between first & last form a palindrome:

have to be the same

e.g. AMANAPLANACANALPANAMA
have to be a palindrome

A recursive definition!

Example: Is a string a palindrome?

```java
/** = "s is a palindrome" */
public static boolean isPal(String s) {
    if (s.length() <= 1) return true;
    // { s has at least 2 chars }
    int n = s.length() - 1;
    return s.charAt(0) == s.charAt(n) && isPal(s.substring(1, n));
}
```

isPal("racecar") returns true
isPal("pumpkin") returns false

Example: Count the e’s in a string

```java
/** = number of times c occurs in s */
public static int countEm(char c, String s) {
    if (s.length() == 0) return 0;
    // { s has at least 1 character }
    if (s.charAt(0) != c)
        return countEm(c, s.substring(1));
    // { first character of s is c}
    return 1 + countEm(c, s.substring(1));
}
```

countEm(‘e’, “it is easy to see that this has many e’s”) = 4
countEm(‘e’, “Mississippi”) = 0