What happens when a function / method is called?
- Create space on the stack to store parameters / local variables of the method (including implicit this parameter for methods)
- Copy values of parameters onto the stack
- Execute the body of the method / function
14-1: Function Call Example

```java
static int plus(int a, int b)
{
    return a + b;
}

static void main(String args[])
{
    int x, y;
    x = plus(3,5);
    y = plus(plus(1,2), plus(3,4));
}
```
14-2: Function Call Example

```java
static int add(int a, int b) {
    while (b > 0) {
        a++;
        b--;
    }
    return a;
}

static int multiply(int a, int b) {
    int result = 0;
    while (b > 0) {
        result = result + a;
        b--;
    }
    return result;
}

static void main(String args[]) {
    int x, y;
    x = multiply(4, 2);
    y = multiply(add(3, 2), add(1, 1));
}
```
14-3: Recursion

- The way function calls work give us a fantastic tool for solving problems
  - Make the problem slightly smaller
  - Solve the smaller problem *using the very function that we are writing*
  - Use the solution to the smaller problem to solve the original problem
14-4: Recursion

- What is a really easy (small!) version of the problem, that I could solve immediately? (Base case)
- How can I make the problem smaller?
- Assuming that I could magically solve the smaller problem, how could I use that solution to solve the original problem (Recursive Case)
Example: Factorial

- \( n! = n \times (n - 1) \times (n - 2) \times \ldots \times 3 \times 2 \times 1 \)
- \( 5! = 5 \times 4 \times 3 \times 2 \times 1 = 120 \)
- \( 8! = 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 40320 \)

What is the base case? That is, a small, easy version of the problem that we can solve immediately?
14-6: Recursion – Factorial

- Example: Factorial
  - \( n! = n \times (n - 1) \times (n - 2) \times \ldots \times 3 \times 2 \times 1 \)
- What is a small, easy version of the problem that we can solve immediately?
  - \( 1! == 1 \).
How do we make the problem smaller?
- What’s a smaller problem than $n!$?
- (only a *little* bit smaller)
How do we make the problem smaller?
  • What’s a smaller problem than $n!$?
  • $(n - 1)!$

If we could solve $(n - 1)!$, how could we use this to solve $n!$?
• How do we make the problem smaller?
  • What’s a smaller problem than $n!$?
  • $(n - 1)!$

• If we could solve $(n - 1)!$, how could we use this to solve $n!$?
  • $n! = (n - 1)! \times n$
int factorial(int n)
{
    if (n == 1)
    {
        return 1;
    }
    else
    {
        return n * factorial(n - 1);
    }
}
Recursion – Factorial

- 0! is defined to be 1
- We can modify factorial to handle this case easily
0! is defined to be 1
We can modify factorial to handle this case easily

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

To solve a recursive problem:

- **Base Case:**
  - Version of the problem that can be solved immediately

- **Recursive Case**
  - Make the problem smaller
  - Call the function recursively to solve the smaller problem
  - Use solution to the smaller problem to solve the larger problem
• Towers of Hanoi
  • Move a sequence of disks from starting tower to ending tower, using a temporary
  • Move one disk at a time
  • Never place a larger disk on top of a smaller disk
14-15: Recursion – ToH

- Writing a program to solve Towers of Hanoi initially seems a little tricky
- Becomes very easy with recursion!

```java
void doMove(char startTower, char endTower)
{
    System.out.print("Move a single disk from tower ");
    System.out.println(startTower + "to tower " + endTower);
}

void towers(int nDisks, char startTower, char endTower, char tmpTower)
{
    ...
}
```
14-16: Recursion – ToH

• Base case:
  • What is a small version of the problem that we could solve immediately?
• Base case:
  • What is a small version of the problem that we could solve immediately?
  • Moving a single disk

```c
void towers(int nDisks, char startTower, char endTower, char tmpTower)
{
    if (nDisks == 1)
    {
        doMove(startTower, endTower);
    }
    ...
}
```
How can we move $n$ disks?

- We can assume that we can magically move $(n - 1)$ disk from any tower to any other tower.
- How can this help us?
14-19: Recursion – ToH

- How can we move $n$ disks?
  - If we could only move $n - 1$ disks from the initial disk to the final disk, we could solve the problem
  - Move the $n - 1$ disks to the temporary peg
  - Move the bottom disk to the final peg
  - Move the $n - 1$ disks from the temporary peg to the final peg
void towers(int nDisks, char startTower, char endTower, char tmpTower)
{
    if (nDisks == 1)
    {
        doMove(startTower, endTower);
    }
    else
    {
        towers(n - 1, startTower, tmpTower, endTower);
        doMove(startTower, endTower);
        towers(n - 1, tmpTower, endTower);
    }
}
Recursion – ToH

- Trace through Towers of Hanoi
When writing a recursive function
Don’t think about how the recursive function works all the way down
Instead, assume that the function just works for a smaller problem
  • Recursive Leap of Faith
Use the solution to the smaller problem to solve the larger problem
Fibonacci Sequence:
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... 

\( F(0) = 1, \ F(1) = 1, \ F(n) = F(n - 1) + F(n - 2) \)

Recursive solution?
int fib(int n)
{
    if (n <= 1)
    {
        return 1;
    }
    else
    {
        return fib(n - 1) + fib(n - 2);
    }
}
14-25: Recursion – Fibonacci

- Problems with this version of fib?
- What about efficiency?
- Can we do it faster?
int fib(int n)
{
    if (n <= 1)
    {
        return 1;
    }
    int fibValues = new int[n+1];
    fibValues[0] = 1;
    fibValues[1] = 1;
    for (int i = 2; i <= n; i++)
    {
        fibValues[i] = fibValues[i-1] + fibValues[i-2];
    }
    return fibValues[n];
}
int fib(int n)
{
    if (n <= 1)
    {
        return 1;
    }
    int next = 1;
    int prev = 1;
    for (int i = 2; i <= n; i++)
    {
        oldNext = next;
        next = next + prev;
        prev = next;
    }
    return next;
}
int fib(int n)
{
    return fib(n, 1, 1);
}

int fib(int n, int next, int prev)
{
    if (n <= 1)
    {
        return next;
    }
    else
    {
        return fib(next + prev, next);
    }
}
Recursion – Reversing Digits

- Function that takes as input an integer
- Writes out the digits in reverse order

```c
void printReversed(int n)
{
    ...
}
```
What’s a easy number to print reversed?

```c
void printReversed(int n)
{
    ...
}
```
Recursion – Reversing Digits

- What’s a easy number to print reversed?

```java
void printReversed(int n)
{
    if (n < 10)
    {
        System.out.println(n);
    }
    ...
}
```
How can we make the problem smaller

- We have to make the problem smaller such that a solution to the smaller problem helps us solve the original problem.
14-33: Recursion – Reversing Digits

- How can we make the problem smaller
  - Remove the last digit (dividing by 10)
  - How can this help?
void printReversed(int n)
{
    if (n < 10)
    {
        System.out.println(n);
    }
    else
    {
        System.out.print(n % 10);
        printReversed(n / 10);
    }
}
Write a method power
public static int power(int x, int n)
    • Return \( x^n \)

What is the base case?

How can we make the problem smaller?

How can we use the solution to the smaller problem to solve the original problem?
• Write a function to reverse a string
  • What is a string that is easy to reverse?
  • How do you make the string smaller
  • How do you use the solution to the smaller problem to solve the original problem?

• String Functions
  • s.substring(k) returns a substring starting from index k
  • s.charAt(k) returns the character at index k in the string