14-0: **Function (method) calls**

- 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**

```
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**

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

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

```
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 probelm *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)

14-5: **Recursion**

- Example: Factorial
  - $n! = n * (n-1) * (n-2) * ... * 3 * 2 * 1$
  - $5! = 5 * 4 * 3 * 2 * 1 = 120$

- $8! = 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 40320$

- 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
  - $n! = n * (n-1) * (n-2) * ... * 3 * 2 * 1$
- What is a small, easy version of the problem that we can solve immediately?
  - 1! == 1.

14-7: **Recursion – Factorial**

- How do we make the problem smaller?
  - What's a smaller problem than $n!$?
  - (only a *liitle* bit smaller)

14-8: **Recursion – Factorial**

- 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 $n!$?

14-9: **Recursion – Factorial**

- 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 $n!$?
  - $n! = (n-1)! * n$

14-10: **Recursion – Factorial**

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

14-11: **Recursion – Factorial**

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

14-12: **Recursion – Factorial**

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

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

14-13: **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

14-14: **Recursion – ToH**

- 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 intially seems a little tricky
- Becomes very easy with recursion!

```
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?

14-17: **Recursion – ToH**

- Base case:

- What is a small version of the problem that we could solve immediately?
- Moving a single disk

```
void towers(int nDisks, char startTower, char endTower, char tmpTower)
{
  if (nDisks == 1)
  {
      doMove(startTower, endTower);
  }
  ...
}
```

14-18: **Recursion – ToH**

- 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 intial 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

14-20: **Recursion – ToH**

```
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);
  }
}
```

14-21: **Recursion – ToH**

- Trace through Towers of Hanoi

14-22: **Recursion – Tips**

- 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

14-23: **Recursion – Fibonacci**

- 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?

14-24: **Recursion – Fibonacci**

```
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 efficientcy?
- Can we do it faster?

14-26: **Iterative Fibonacci**

```
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];
}
```

14-27: **Iterative Fibonacci**

```
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;
}
```

14-28: **Recursion – Fibonacci**

```
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);
   }
}
```

14-29: **Recursion – Reversing Digits**

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

```
void printReversed(int n)
{
 ...
}
```

14-30: **Recursion – Reversing Digits**

- What's a easy number to print reversed?

```
void printReversed(int n)
{
 ...
}
```

14-31: **Recursion – Reversing Digits**

- What's a easy number to print reversed?

```
void printReversed(int n)
{
   if (n < 10)
   {
      System.out.println(n);
   }
   ...
}
```

14-32: **Recursion – Reversing Digits**

- 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?

14-34: **Recursion – Reversing Digits**

```
void printReversed(int n)
{
   if (n < 10)
   {
      System.out.println(n);
   }
   else
   {
      System.out.print(n % 10);
      printReversed(n / 10);
   }
}
```

14-35: **Recursion – Hands on**

- 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?

14-36: **Recursion – Reverse**

- 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