

13-0: == vs. .equals()

```
String s1, s2, s3;

s1 = "hello";
s2 = "he";
s3 = "llo";
s4 = s2 + s3;
s5 = s1;

System.out.println(s1 == s4);
System.out.println(s1 == s5);
System.out.println(s1.equals(s4));
System.out.println(s1.equals(s5));
```

13-1: == vs. .equals()

```
String s1, s2, s3;

s1 = "hello";
s2 = "he";
s3 = "llo";
s4 = s2 + s3;
s5 = s1;

System.out.println(s1 == s4); // False!
System.out.println(s1 == s5); // True
System.out.println(s1.equals(s4)); // True
System.out.println(s1.equals(s5)); // False
```

13-2: == vs. .equals()

```
String s1, s2;

s1 = "hello";
s2 = "hello";
s3 = s1;

System.out.println(s1 == s2);
System.out.println(s1 == s3);
System.out.println(s1.equals(s2));
System.out.println(s1.equals(s3));
```

13-3: String are Immutable

- Once we create a string, we cannot change the contents of the string
 - Examples
- We can, however, change what a string variable points to
 - Examples
- The compiler only creates one copy of “hello”, has both strings point to this copy
 - Safe because strings are immutable

13-4: == vs. .equals()

```
String s1, s2;

s1 = "hello";
s2 = "hello";
s3 = s1;

System.out.println(s1 == s2); // Depends on the compiler!
System.out.println(s1 == s3); // Always true
System.out.println(s1.equals(s2)); // Always true
System.out.println(s1.equals(s3)); // Always true
```

13-5: == vs. .equals()

```
String s1, s2;

s1 = "hello";
s2 = "he" + "llo";
s3 = s1;

System.out.println(s1 == s2);
System.out.println(s1 == s3);
System.out.println(s1.equals(s2));
System.out.println(s1.equals(s3));
```

13-6: == vs. .equals()

```
String s1, s2;

s1 = "hello";
s2 = "he" + "llo";
s3 = s1;

System.out.println(s1 == s2);      // Depends on the compiler!
System.out.println(s1 == s3);      // Always true
System.out.println(s1.equals(s2)); // Always true
System.out.println(s1.equals(s3)); // Always true
```

13-7: == vs. .equals()

```
String s1, s2;

s1 = "hello";
s2 = "he";
s2 = s2 + "llo";
s3 = s1;

System.out.println(s1 == s2);
System.out.println(s1 == s3);
System.out.println(s1.equals(s2));
System.out.println(s1.equals(s3));
```

13-8: == vs. .equals()

```
String s1, s2;

s1 = "hello";
s2 = "he";
s2 = s2 + "llo";
s3 = s1;

System.out.println(s1 == s2);      // False
System.out.println(s1 == s3);      // Always true
System.out.println(s1.equals(s2)); // Always true
System.out.println(s1.equals(s3)); // Always true
```

13-9: 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)

13-10: Recursion

- Example: Factorial
 - $n! = n * (n - 1) * (n - 2) * \dots * 3 * 2 * 1$
 - $5! = 5 * 4 * 3 * 2 * 1 = 120$
 - $8! = 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 40320$
- What is the base case? That is, a small, easy version of the problem that we can solve immediately?

13-11: Recursion – Factorial

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

13-12: Recursion – Factorial

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

13-13: 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 $n!$?

13-14: 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 $n!$?
 - $n! = (n - 1)! * n$

13-15: Recursion – Factorial

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

13-16: Recursion – Factorial

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

13-17: 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);
    }
}
```

13-18: 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

13-19: 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

13-20: Recursion – ToH

- Writing a program to solve Towers of Hanoi initially 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)
{
    ...
}
```

13-21: Recursion – ToH

- Base case:
 - What is a small version of the problem that we could solve immediately?

13-22: 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);
    }
    ...
}
```

13-23: 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?

13-24: 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

13-25: 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);
    }
}
```

13-26: Recursion – ToH

- Trace through Towers of Hanoi

13-27: 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

13-28: 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?

13-29: Recursion – Fibonacci

```
int fib(int n)
{
    if (n <= 1)
    {
        return 1;
    }
    else
    {
        return fib(n - 1) + fib(n - 2);
    }
}
```

13-30: Recursion – Fibonacci

- Problems with this version of fib?
- What about efficiency?
- Can we do it faster?

13-31: 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];
}
```

13-32: 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 = oldNext;
    }
    return next;
}
```

13-33: 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);
    }
}
```

13-34: Recursion – Reversing Digits

- Function that takes as input an integer

- Writes out the digits in reverse order

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

13-35: Recursion – Reversing Digits

- What's a easy number to print reversed?

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

13-36: Recursion – Reversing Digits

- What's a easy number to print reversed?

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

13-37: 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

13-38: Recursion – Reversing Digits

- How can we make the problem smaller
 - Remove the last digit (dividing by 10)
 - How can this help?

13-39: Recursion – Reversing Digits

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

13-40: 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?