Quiz 05 Review Session

4/3/19

Welcome!

- recursion
  - base case (how do we come up with one?)
- recursive data structures
  - rest, cons, first

🔥 hot date 🔥 & check-in on course.care

- generics
  - classes and functions

To access the google slides version of this slide deck, follow this link: http://bit.ly/2HS0zu8

RECURSION RECURSION RECURSION RECURSION RECUR

What is recursion?

Recursion is a solution to a problem that relies on smaller instances of *the same problem!*

When do we use recursion?

We use recursion when we we want to repeat some functionality over and over.

RECURSION RECURSION RECURSION RECURSION RECUR

What are important components of recursion?

1. Base Case!
    a. How do we find a base case?
        i. Think about the intended result. When do you want to stop making recursive calls?
        ii. Think about how a base case for a linked list is different than a base case for some computation
2. A recursive call!
    a. A recursive call is simply a call to the function that we are currently in.

The best way to get good at recursive solutions is to practice coming up with them!

RECURSION RECURSION RECURSION RECURSION RECUR

Take the following recursive solutions and figure out the base cases for each.

```
let multNums = (n: number): number => {
   if (?) {
      ?;
   }
   return n * sumNums(n - 1);
};
```

Find product of numbers from 1 to n.

```
let sumNums = (n: number): number => {
   if (?) {
      ?;
   }
   return n + sumNums(n - 1);
};
```

Find sum of numbers from 1 to n.

RECURSION RECURSION RECURSION RECURSION RECUR

Take the following recursive solutions and figure out the base cases for each.

```
let multNums = (n: number): number => {
   if (n === 0) {
      return 1;
   }
   return n * sumNums(n - 1);
};
```

Find product of numbers from 1 to n.

```
let sumNums = (n: number): number => {
   if (n === 0) {
      return 0;
   }
   return n + sumNums(n - 1);
};
```

Find sum of numbers from 1 to n.

Stepping through recursion...

```
let main = async () => {
     print(countChars(["aa","ggg", "aiioo"], 0));
};

let countChars = (arr: string[], index: number): number =>
{
   if (index >= arr.length) {
       return 0;
   }
   return arr[index].length + countChars(arr, index + 1);
};
main();
```

Stepping through recursion...

```
let main = async () => {
     print(countChars(["aa","ggg", "aiioo"], 0));
};

let countChars = (arr: string[], index: number): number =>
{
   if (index >= arr.length) {
       return 0;
   }
   return arr[index].length + countChars(arr, index + 1);
};
main();
```

1: countChars(arr, 0) ->

arr[0].length +

Stepping through recursion...

```
let main = async () => {
     print(countChars(["aa","ggg", "aiioo"], 0));
};

let countChars = (arr: string[], index: number): number =>
{
   if (index >= arr.length) {
       return 0;
   }
   return arr[index].length + countChars(arr, index + 1);
};
main();
```

1: countChars(arr, 0) ->

arr[0].length +

2: countChars(arr, 1) ->
arr[1].length +

Stepping through recursion...

```
let main = async () => {
     print(countChars(["aa","ggg", "aiioo"], 0));
};

let countChars = (arr: string[], index: number): number =>
{
   if (index >= arr.length) {
       return 0;
   }
   return arr[index].length + countChars(arr, index + 1);
};
main();
```

1: countChars(arr, 0) ->
arr[0].length +

2: countChars(arr, 1) ->
arr[1].length +

3: countChars(arr, 2) ->
arr[2].length +

Stepping through recursion...

```
let main = async () => {
     print(countChars(["aa","ggg", "aiioo"], 0));
};

let countChars = (arr: string[], index: number): number =>
{
   if (index >= arr.length) {
       return 0;
   }
   return arr[index].length + countChars(arr, index + 1);
};
main();
```

1: countChars(arr, 0) ->
arr[0].length +

2: countChars(arr, 1) ->
arr[1].length +

3: countChars(arr, 2) ->
arr[2].length +

4: countChars(arr, 3) ->

Stepping through recursion...

```
let main = async () => {
     print(countChars(["aa","ggg", "aiioo"], 0));
};

let countChars = (arr: string[], index: number): number =>
{
   if (index >= arr.length) {
       return 0;
   }
   return arr[index].length + countChars(arr, index + 1);
};
main();
```

1: countChars(arr, 0) ->

arr[0].length +

2: countChars(arr, 1) ->

arr[1].length +

3: countChars(arr, 2) ->

arr[2].length +

4: countChars(arr, 3) ->

BASE CASE

Stepping through recursion...

```
let main = async () => {
     print(countChars(["aa","ggg", "aiioo"], 0));
};

let countChars = (arr: string[], index: number): number =>
{
   if (index >= arr.length) {
       return 0;
   }
   return arr[index].length + countChars(arr, index + 1);
};
main();
```

1: countChars(arr, 0) ->

arr[0].length +

2: countChars(arr, 1) ->
arr[1].length +

3: countChars(arr, 2) ->
arr[2].length + 0

4: returned 0

BASE CASE

Stepping through recursion...

```
let main = async () => {
     print(countChars(["aa","ggg", "aiioo"], 0));
};

let countChars = (arr: string[], index: number): number =>
{
   if (index >= arr.length) {
       return 0;
   }
   return arr[index].length + countChars(arr, index + 1);
};
main();
```

1: countChars(arr, 0) ->

arr[0].length +

2: countChars(arr, 1) ->
arr[1].length +

3: countChars(arr, 2) ->
5 + 0

4: returned 0

BASE CASE

Stepping through recursion...

```
let main = async () => {
     print(countChars(["aa","ggg", "aiioo"], 0));
};

let countChars = (arr: string[], index: number): number =>
{
   if (index >= arr.length) {
       return 0;
   }
   return arr[index].length + countChars(arr, index + 1);
};
main();
```

1: countChars(arr, 0) ->

arr[0].length +

2: countChars(arr, 1) ->
arr[1].length + 5

3: returned 5

4: returned 0

BASE CASE

Stepping through recursion...

```
let main = async () => {
     print(countChars(["aa","ggg", "aiioo"], 0));
};

let countChars = (arr: string[], index: number): number =>
{
   if (index >= arr.length) {
       return 0;
   }
   return arr[index].length + countChars(arr, index + 1);
};
main();
```

1: countChars(arr, 0) ->

arr[0].length +

2: countChars(arr, 1) ->
3 + 5

3: returned 5

4: returned 0

BASE CASE

Stepping through recursion...

```
let main = async () => {
     print(countChars(["aa","ggg", "aiioo"], 0));
};

let countChars = (arr: string[], index: number): number =>
{
   if (index >= arr.length) {
       return 0;
   }
   return arr[index].length + countChars(arr, index + 1);
};
main();
```

1: countChars(arr, 0) ->

arr[0].length + 8

2: returned 8

3: returned 5

4: returned 0

BASE CASE

Stepping through recursion...

```
let main = async () => {
     print(countChars(["aa","ggg", "aiioo"], 0));
};

let countChars = (arr: string[], index: number): number =>
{
   if (index >= arr.length) {
       return 0;
   }
   return arr[index].length + countChars(arr, index + 1);
};
main();
```

1: countChars(arr, 0) ->

2 + 8

2: returned 8

3: returned 5

4: returned 0

BASE CASE

Stepping through recursion...

```
let main = async () => {
     print(countChars(["aa","ggg", "aiioo"], 0));
};

let countChars = (arr: string[], index: number): number =>
{
   if (index >= arr.length) {
       return 0;
   }
   return arr[index].length + countChars(arr, index + 1);
};
main();
```

1: returned 10

2: returned 8

3: returned 5

4: returned 0

BASE CASE

1: countChars(arr, 0) ->
arr[0].length + 2: countChars(arr, 1) ->
arr[1].length + 3: countChars(arr, 2) ->
arr[2].length + 4: countChars(arr, 3) ->
BASE CASE: return 0

```
let main = async () => {
    print(countChars(["aa","ggg",
"aiioo", "a"], 0));
};

let countChars = (arr: string[], index:
number): number => {
   if (index >= arr.length) {
       return 0;
   }
   return arr[index].length +
countChars(arr, index + 1);
```

JUST ANOTHER VISUAL
OF RECURSION

CHALLENGE Practice with RECURSION RECURSION RECURSION RECURSION RECUR

Write a recursive solution to find the greatest common factor between two numbers. Your function should take in two numbers as arguments. You can assume the first number will always be greater than the second.

Hint: Figure out the base case! What is the simplest gcd that you can calculate?

1. Is the first num evenly divisible by the second?
   a. Yes → return the second number
   b. No → recursion

2. What state needs to change with each recursive call? Lets try gcf of 20 and 10.
   a. Do we change 20?
   b. Do we change 10?
   c. Do we change both?

Solution(s):

```
let gcf = (a: number, b: number):
number  => {
if (a % b === 0) {
        return b;
    }
    return gcf(a, a % b);
};
```

```
let gcf = (a: number, b: number):
number  => {
if (b === 0) {
        return a;
    }
    return gcf(b, a % b);
};
```

Recursive Data Structures!

A data type that has property which is an element of that same data type

ex. a class that has a property which is an object of that same class!

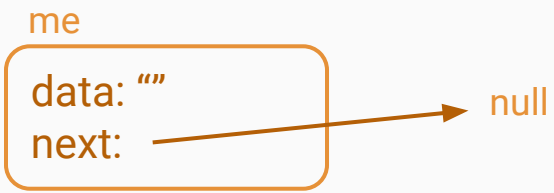
```
class Node {
    data: string = "";
    next: Node = null;
}
```

Linked Lists

```
class Node {
   data: string = "";
   next: Node = null;
}
```

We can link node objects together via that property to create a list

```
let me: Node = new Node();
```

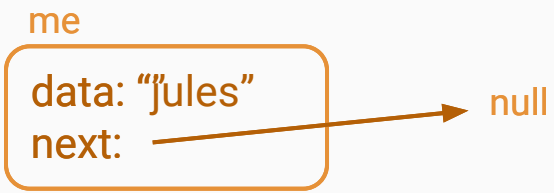


Linked Lists

```
class Node {
   data: string = "";
   next: Node = null;
}
```

We can link node objects together via that property to create a list

```
let me: Node = new Node();
me.data = "jules";
```

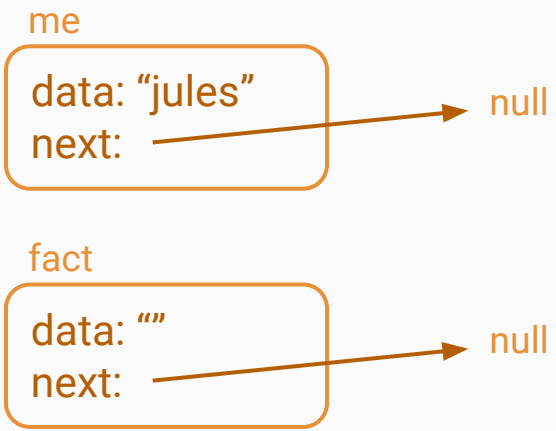


Linked Lists

```
class Node {
   data: string = "";
   next: Node = null;
}
```

We can link node objects together via that property to create a list

```
let me: Node = new Node();
me.data = "jules";
let fact: Node = new Node();
```

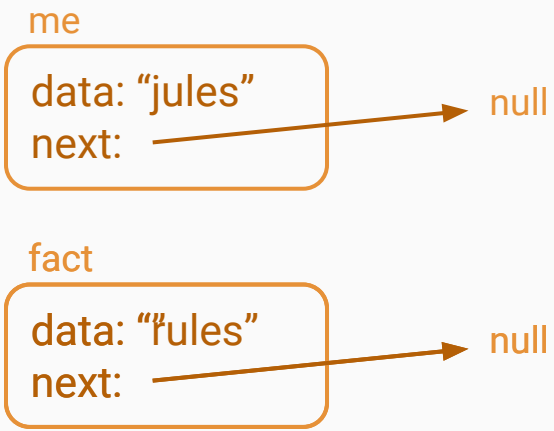


Linked Lists

```
class Node {
   data: string = "";
   next: Node = null;
}
```

We can link node objects together via that property to create a list

```
let me: Node = new Node();
me.data = "jules";
let fact: Node = new Node();
fact.data = "rules";
```

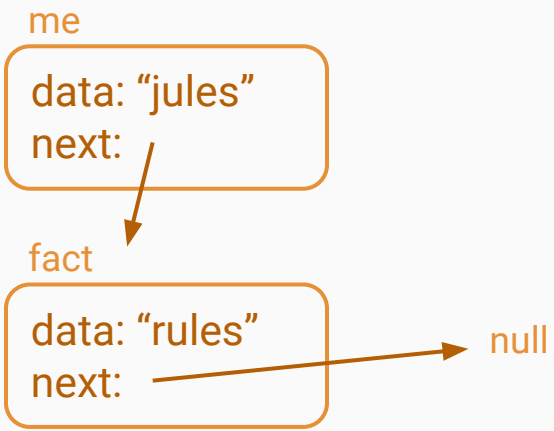


Linked Lists

```
class Node {
   data: string = "";
   next: Node = null;
}
```

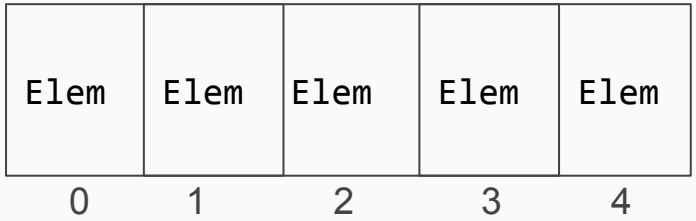
We can link node objects together via that property to create a list

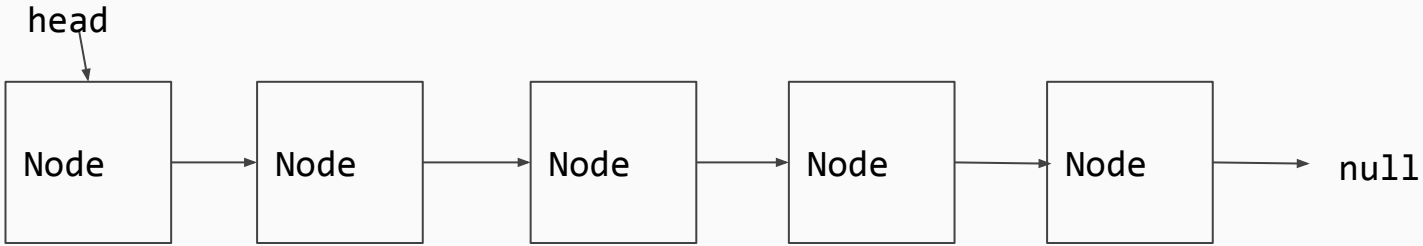
```
let me: Node = new Node();
me.data = "jules";
let fact: Node = new Node();
fact.data = "rules";
me.next = fact;
```



Linked Lists

- Serve similar purpose as arrays: store info
  - However: no indices! How do we access data?
  - Use the beginning node & recursively step through “next” properties
- Lists are null-terminated





A note on NULL

It’s the known absence of an object

Like a placeholder where an object could be, but isn’t

Can be written: null

Can use the ∅ symbol

Can draw an arrow pointing to the word null on the heap

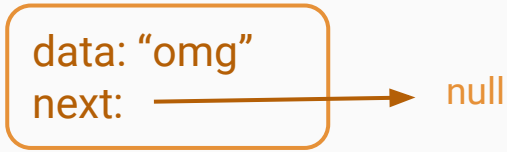
Working with Lists

You can’t see all of your list at once & you can’t use indices with it! Yike!

Luckily, you have some handy dandy helper functions to work with

Cons allows you to construct a list

```
cons("omg", null);
```

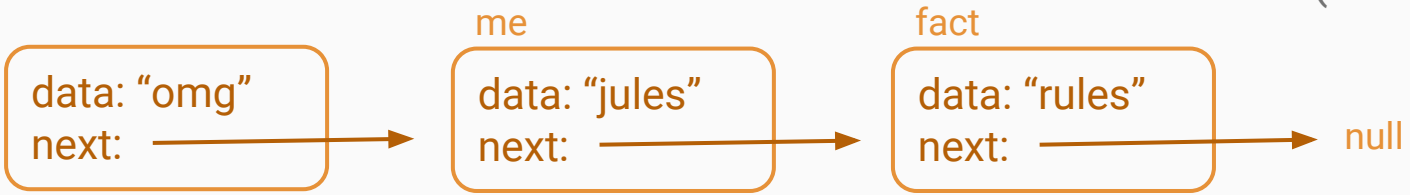


The data or value you’d like to preprend (aka add to the front of) your list

The list you’re adding on to… this becomes the “rest” property of your node

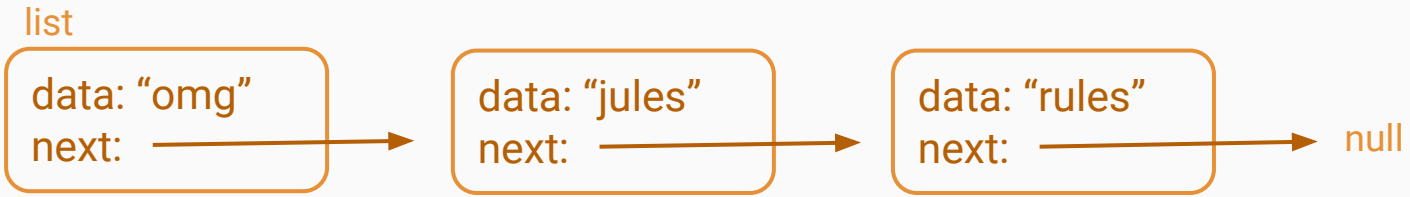
Working with Lists: cons

(from before)




```
cons("omg", me);
```

Working with Lists: cons




```
cons("omg", me);
```

...the same thing can be achieved by:

```
let list = cons("omg", cons("jules", cons("rules", null)));
```

Notice which nodes now have their own names/are accessible by name!

Working with Lists: first

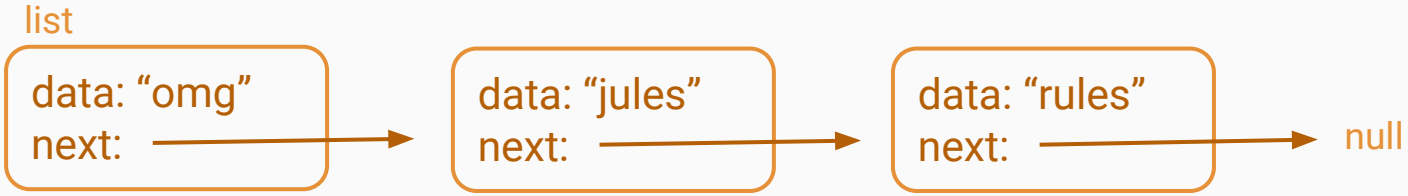
first allows you to retrieve the data stored in the node you give the function

```
let x = first(list);
print(x);
```

What is x's type?

What is x?

omg
string



Working with Lists: rest

rest gives you the node stored in the “next” property; the REST of your list

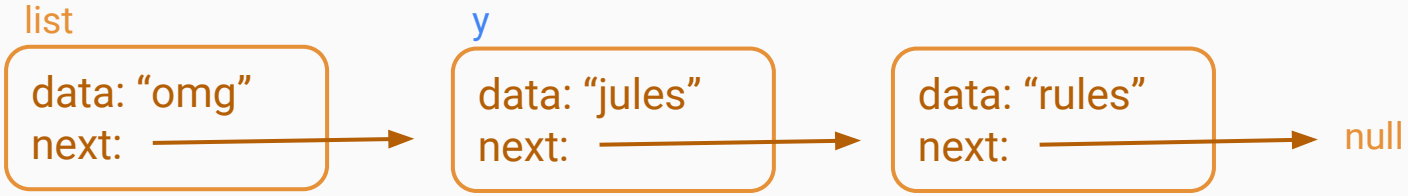
```
let y = rest(list);
print(y);
```

What is y’s type?

What is y?

jules
rules
null
List<string>

*the list isn’t changed, just which node you’re treating as the head does



Working with Lists

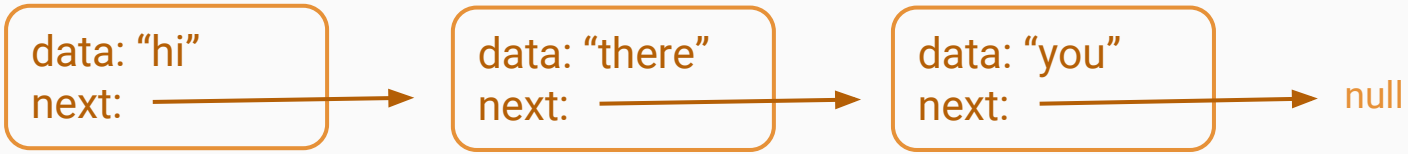
Tip: work from the inside out

You can use first, cons, & rest in conjunction with each other

```
let z = first(rest(rest(cons("hi", cons("there", cons("you", null))))));
print(z);
```

What is z's type? What is z?

you
string



Working with Lists

Lists are a recursive data type, so it's best to process them recursively.

General formula:

1) Check if you've hit the **base case** (often, if the node you're on is null)
2) Process the data of the current node with **first**
3) Make your **recursive call** (often, you'll pass in the **rest** of your list as an argument)

Working with Lists

```
export let excitedPrinter = (list: Node): void => {
    if (list === null) {
        print("The End.");
    } else {
        print(first(list) + "!");
        excitedPrinter(rest(list));
    }
};
```

COMP!
string

110!
string

is great!
string

The End.
string

```
let printMe = cons("COMP", cons("110", cons("is great", null)));
excitedPrinter(printMe);
```

RECURSION RECURSION RECURSION RECURSION RECUR

Overview of introcs functions!

cons - remember, the last node in a list must point to null!

```
let welcome = cons("hello", cons("world", null));
```

first - return the value of your first node!

```
let hello: string = first(welcome);
```

rest - returns a list of nodes, the "next" property of the node you give it

```
let world = rest(welcome);
```

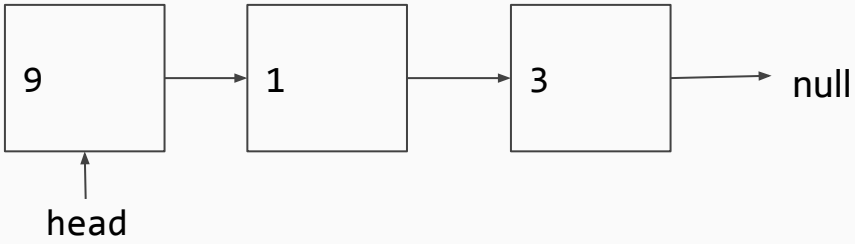
listify - produces a list with the given arguments

```
let myList = listify("goodbye", "moon");
```

RECURSION RECURSION RECURSION RECURSION RECUR

Using listify, cons, first, and rest, create a function that, given a list of numbers, returns a list of only odd numbers.

Ex: sumOdd(4, 9, 1, 3)




```
let sumOdd = (list: Node<number>): Node<number> =>
{
    if (list === null) {
        return null;
    }
    if (first(list) % 2 === 0) {
        return sumOdd(rest(list));
    }
    return cons(first(list), sumOdd(rest(list)));
};
```

Check-in & Hot Date!

A small break! Check in on course.care with this code: **02B3C**

Talk to your neighbor about the weather or something lol

GENERIC TYPES

(This hamster is being generic by not using your names)

Generalizing functions and classes can be useful for performing the same function/algorithm on different data types. This allows us to reduce repetitive code.

You can generalize a property's type in a class, or parameters in a function definition.

Functions to Return Equivalence for numbers, strings, and booleans

Numbers:

```
let cool = (thing1: number, thing2: number): boolean => {
    if (thing1 === thing2) {
        return true;
    }
    return false;
};
```

Strings:

```
let heyWait = (thing1: string, thing2: string): boolean => {
    if (thing1 === thing2) {
        return true;
    }
    return false;
};
```

Booleans:

```
let iveSeenThisBefore = (thing1: boolean, thing2: boolean): boolean => {
    if (thing1 === thing2) {
        return true;
    }
    return false;
};
```

Functions to Return Equivalence for numbers, strings, and booleans

Numbers:

```
let cool = (thing1: number, thing2: number): boolean => {
    if (thing1 === thing2) {
        return true;
    }
    return false;
};
```

Strings:

```
let heyWait = (thing1: string, thing2: string): boolean => {
    if (thing1 === thing2) {
        return true;
    }
    return false;
};
```

Booleans:

```
let iveSeenThisBefore = (thing1: boolean, thing2: boolean): boolean => {
    if (thing1 === thing2) {
        return true;
    }
    return false;
};
```

How to generalize a function though?

Add a **<T>** before the list of parameters..

```
let better = <T> (thing1: string, thing2: string): boolean => {
```

Generalize our parameters that need to be changed...

```
let better = <T> (thing1: T, thing2: T): boolean => {
```

And change any variables that you may have used in your function. In this case, there were none!


```
let better = <T> (thing1: T, thing2: T): boolean => {
    if (thing1 === thing2) {
        return true;
    }
    return false;
};
```

Generic Class example: Node<T> (from Lecture 16 PowerPoint)

```
export class Node<T> {
   data: T;
   next: Node<T> = null;
}
```

Node<T> is a generic class that holds a “data” property that holds a value of a generic type. Now we don’t need different nodes for strings, numbers, or booleans.

```
// Explicit Typing
let a: Node<string> = new Node<string>();
a.data = "hello, world";

// Type Inference
let b = new Node<number>();
b.data = 110;
```

When we declare and use generic Nodes now though, we must tell the function what T will be.

For example, Node<string> means that the data property will be a string.

```
let listEater = <T> (head: Node<T>, n: number): Node<T> => {
    if (n === 0) {
        return head;
    }
    return listEater(rest(head), n - 1);
};
```

What would be the type of T, the type of the returned value, and the type of n for the following function calls?

```
listEater(listify(1, 2, 3, 4, 5), 3);
```

number, Node<number>, number

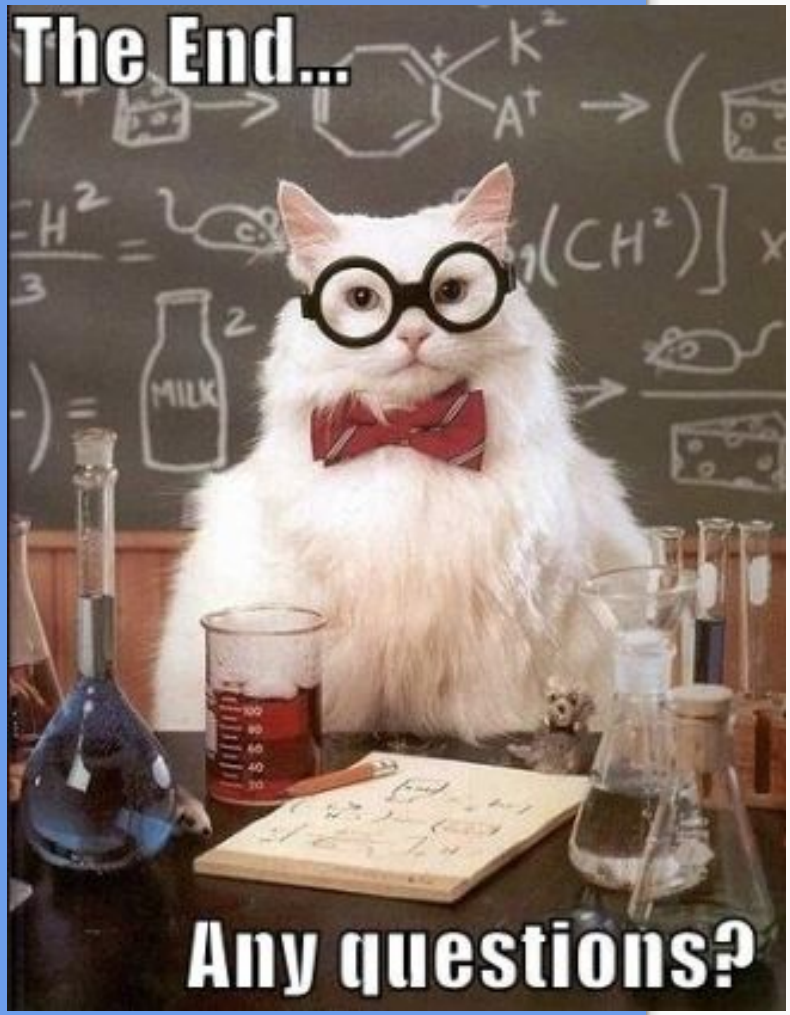
```
listEater(listify(true, true, false), 1);
```

boolean , Node<boolean>, number

An example with arrays!

```
let arrayPrinter = <T> (arr: T[]): void => {
    let output: T[] = [];
    for (let i = 0; i < arr.length; i++) {
        print(arr[i]);
    }
};
```

Now we don’t have to have three separate arrayPrinter functions for number, boolean, and string arrays!



Any questions?