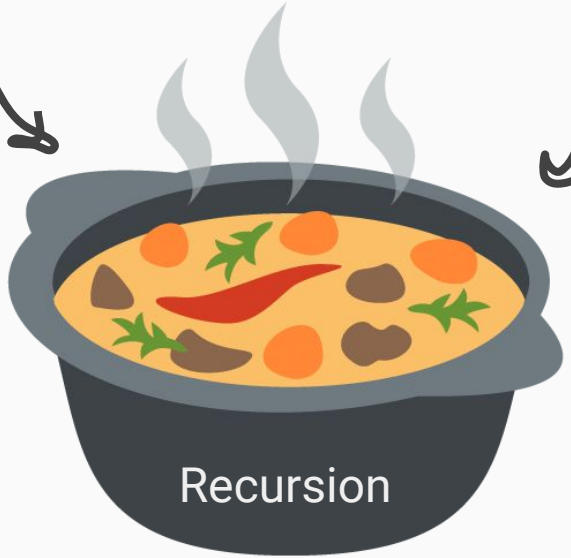
Quiz 4 Review Session

Outline

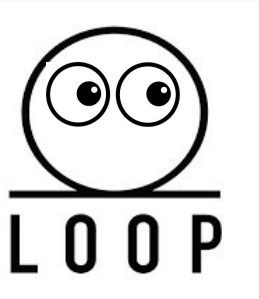
- Recursion
  - Essentials & Examples
  - Environment diagrams
- List functions
  - first
  - rest
  - cons
- Generics

Recursion Soup!

Recursive call
(function call to itself)



Base case
(recursion done, wrap up each recursive call)



Example 1

```
export let main = async () => {
    let x = howFunWasIt(5);
    print(x);
};

let howFunWasIt = (n: number): string => {
    if (n <= 0) {
        return "fun!";
    }
    return "so " + howFunWasIt(n - 1);
};

main();
```

"so" + howFunWasIt(4);

"so" + howFunWasIt(3);

"so" + howFunWasIt(2);

"so" + howFunWasIt(1);

"so" + howFunWasIt(0);

"fun!"

Base case

Recursive call

Example 1

```
export let main = async () => {
    let x = howFunWasIt(5);
    print(x);
};

let howFunWasIt = (n: number): string => {
    [redacted]
    return "so " + howFunWasIt(n - 1);
};

main();
```

THIS WOULD PRINT:

so so so so so fun!

Base case

Recursive call

yikes!

Example 1

```
export let main = async () => {
    let x = howFunWasIt(5);
    print(x);
};

let howFunWasIt = (n: number): string => {
    [redacted]
    return "so " + howFunWasIt(n - 1);
};

main();
```

Recursive call

THIS WOULD PRINT:

Sike! This will give you a stack overflow error. You'll never get done with howFunWasIt so you'll never get to that print statement in main.

Throwing some more business in there

```
export let main = async () => {
    let x = howCoolWasIt(4);
    print(x);
};

let howCoolWasIt = (n: number): string => {
    print("hey");
    if (n <= 0) {
        print("Dude!");
        return "cool,";
    }
    if (n === 4) {
        return "It was " + howCoolWasIt(n - 1);
    } else {
        return "very " + howCoolWasIt(n - 1) + " dude";
    }
};

main();
```

code BEFORE recursive call

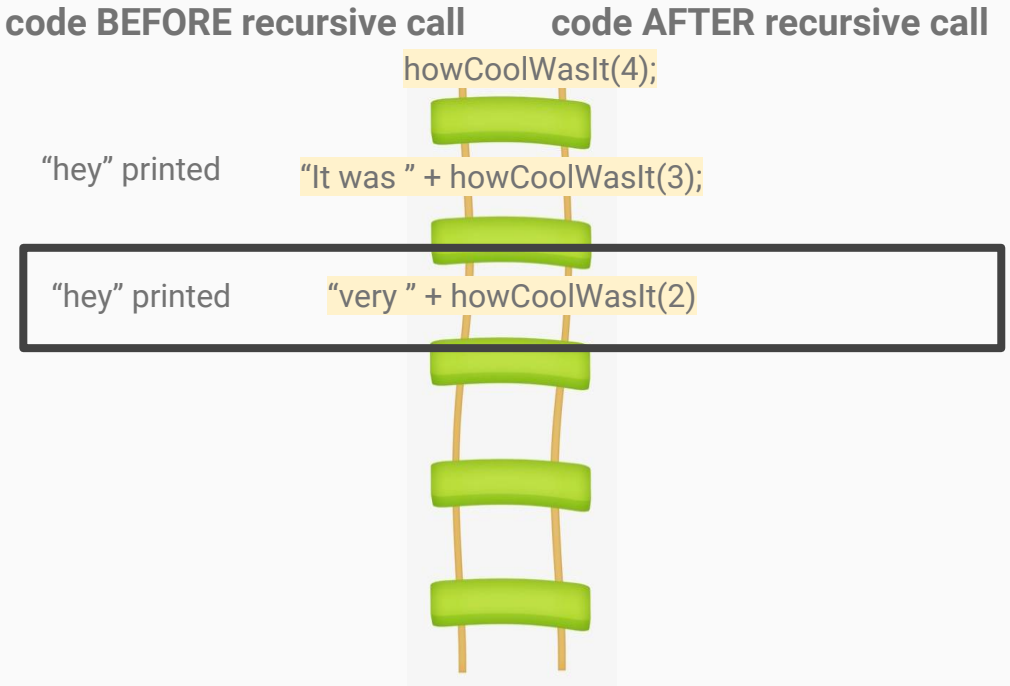
code AFTER recursive call

Throwing some more business in there

```
export let main = async () => {
    let x = howCoolWasIt(4);
    print(x);
};

let howCoolWasIt = (n: number): string => {
    print("hey");
    if (n <= 0) {
        print("Dude!");
        return "cool,";
    }
    if (n === 4) {
        return "It was " + howCoolWasIt(n - 1);
    } else {
        return "very " + howCoolWasIt(n - 1) + " dude";
    }
};

main();
```

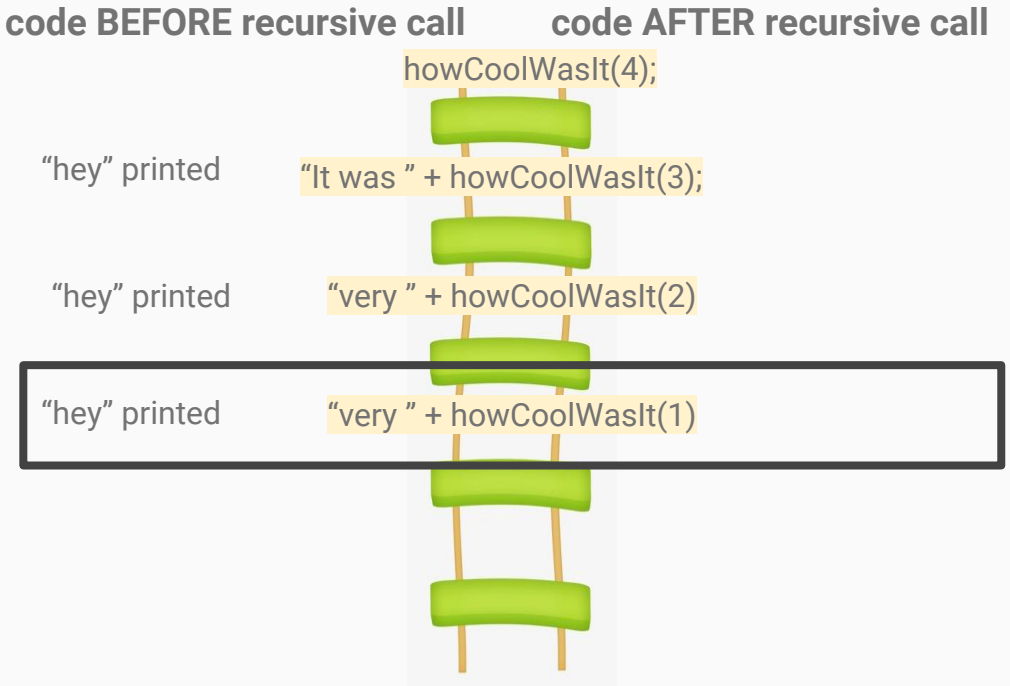
Throwing some more business in there

```
export let main = async () => {
    let x = howCoolWasIt(4);
    print(x);
};

let howCoolWasIt = (n: number): string => {
    print("hey");
    if (n <= 0) {
        print("Dude!");
        return "cool,";
    }
    if (n === 4) {
        return "It was " + howCoolWasIt(n - 1);
    } else {
        return "very " + howCoolWasIt(n - 1) + " dude";
    }
};

main();
```



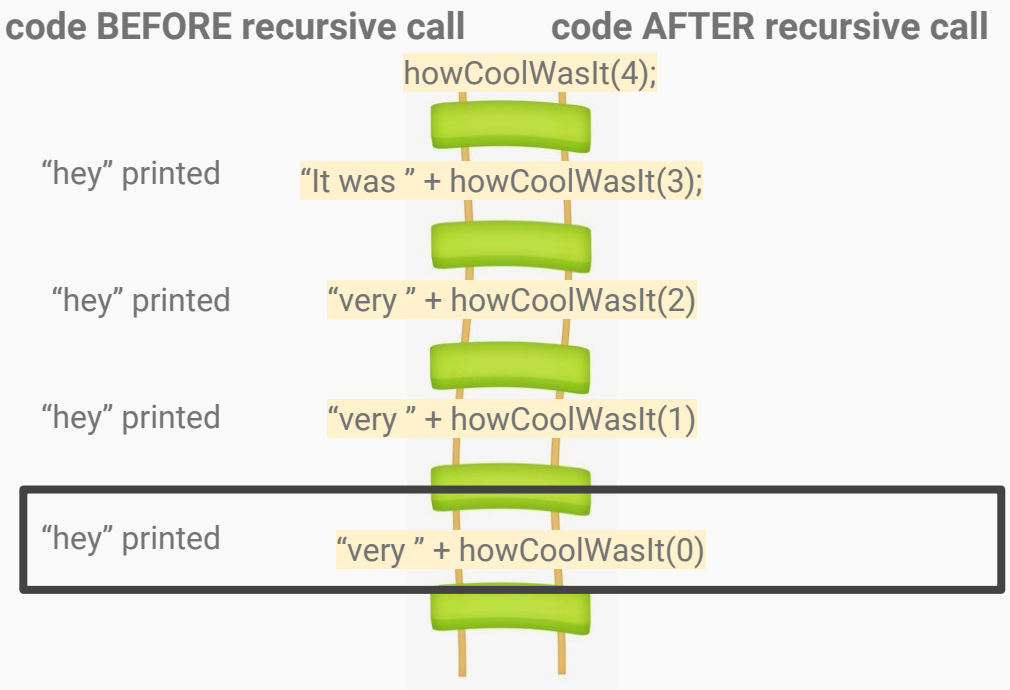
Throwing some more business in there

```
export let main = async () => {
    let x = howCoolWasIt(4);
    print(x);
};

let howCoolWasIt = (n: number): string => {
    print("hey");
    if (n <= 0) {
        print("Dude!");
        return "cool,";
    }
    if (n === 4) {
        return "It was " + howCoolWasIt(n - 1);
    } else {
        return "very " + howCoolWasIt(n - 1) + " dude";
    }
};

main();
```



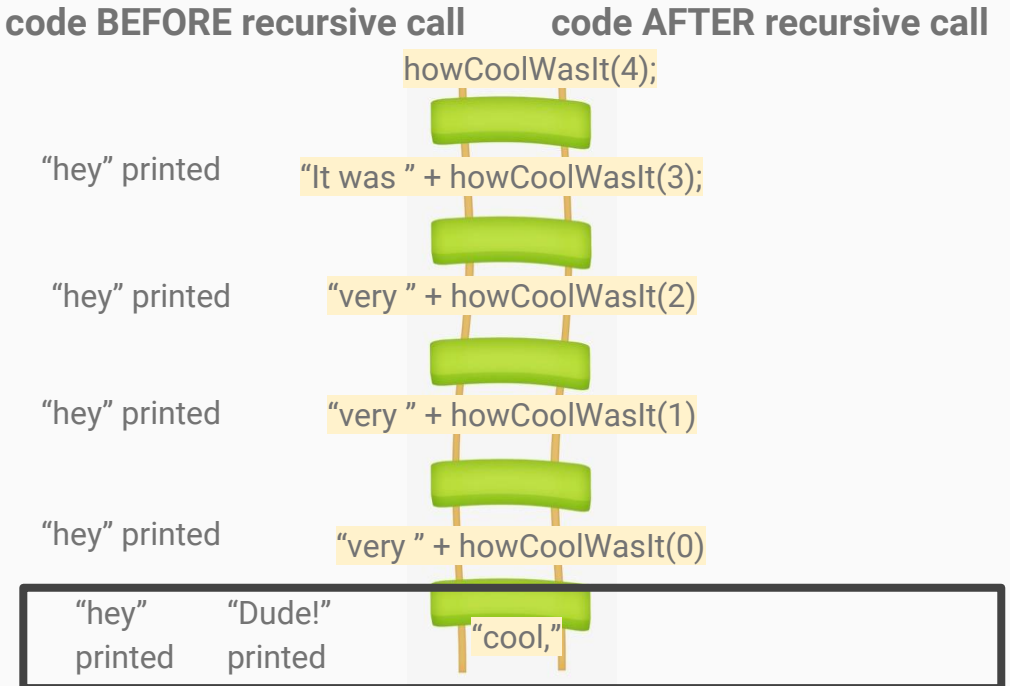
Throwing some more business in there

```
export let main = async () => {
    let x = howCoolWasIt(4);
    print(x);
};

let howCoolWasIt = (n: number): string => {
    print("hey");
    if (n <= 0) {
        print("Dude!");
        return "cool,";
    }
    if (n === 4) {
        return "It was " + howCoolWasIt(n - 1);
    } else {
        return "very " + howCoolWasIt(n - 1) + " dude";
    }
};

main();
```



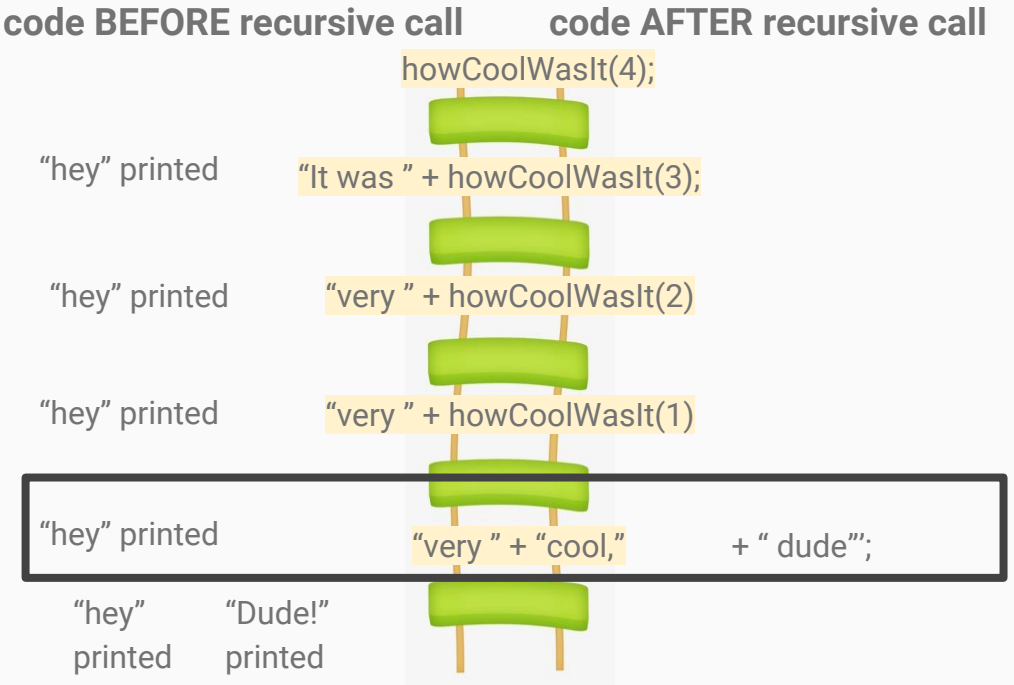
Throwing some more business in there

```
export let main = async () => {
    let x = howCoolWasIt(4);
    print(x);
};

let howCoolWasIt = (n: number): string => {
    print("hey");
    if (n <= 0) {
        print("Dude!");
        return "cool,";
    }
    if (n === 4) {
        return "It was " + howCoolWasIt(n - 1);
    } else {
        return "very " + howCoolWasIt(n - 1) + " dude";
    }
};

main();
```



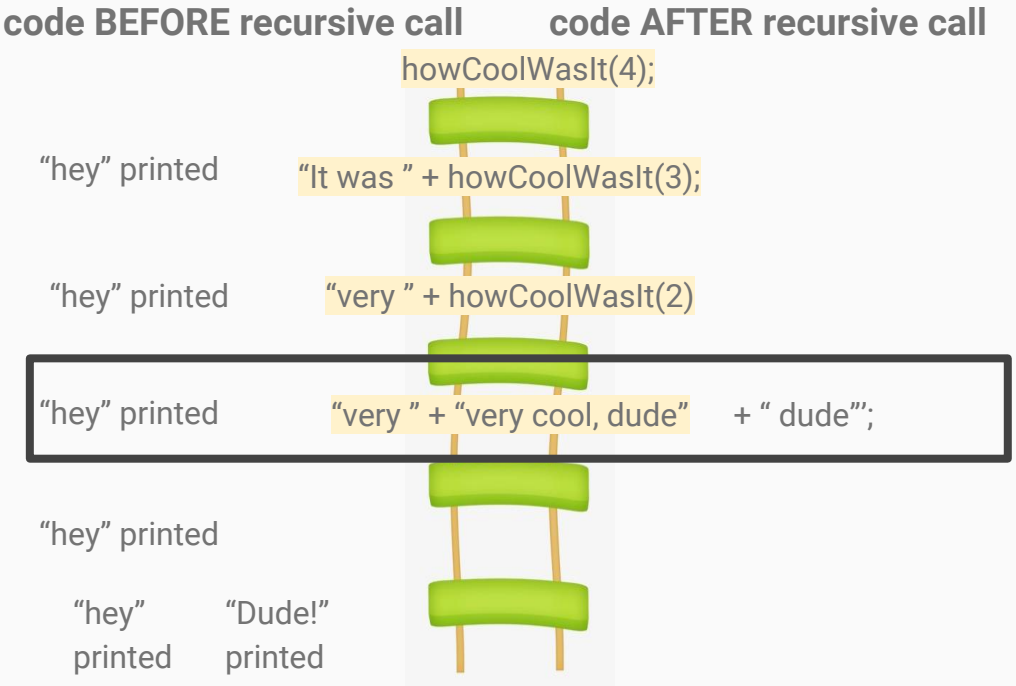
Throwing some more business in there

```
export let main = async () => {
    let x = howCoolWasIt(4);
    print(x);
};

let howCoolWasIt = (n: number): string => {
    print("hey");
    if (n <= 0) {
        print("Dude!");
        return "cool,";
    }
    if (n === 4) {
        return "It was " + howCoolWasIt(n - 1);
    } else {
        return "very " + howCoolWasIt(n - 1) + " dude";
    }
};

main();
```



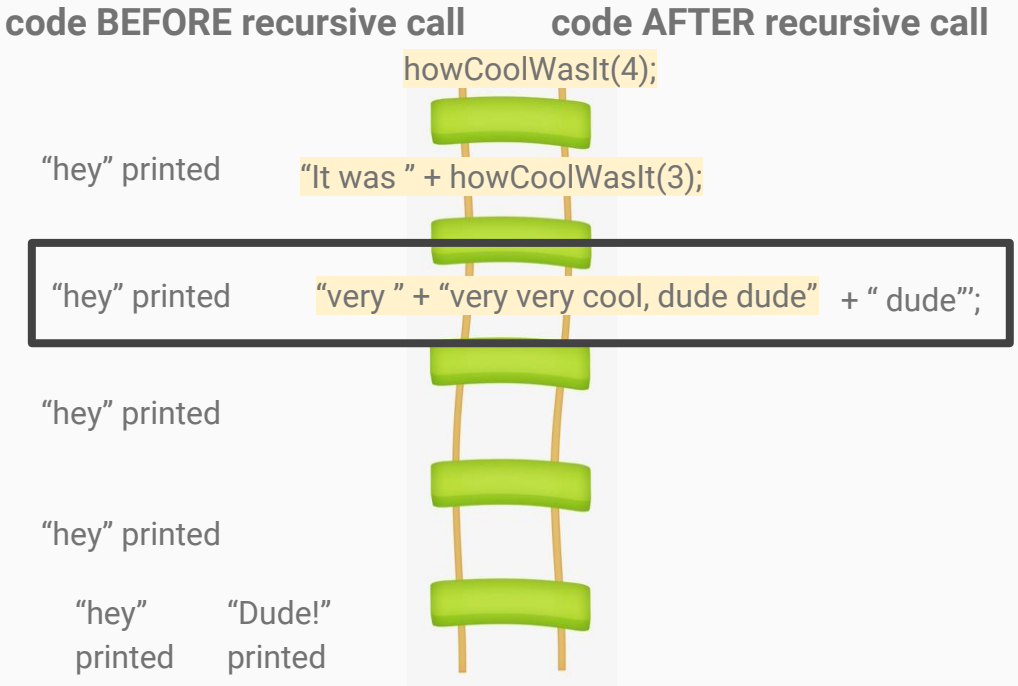
Throwing some more business in there

```
export let main = async () => {
    let x = howCoolWasIt(4);
    print(x);
};

let howCoolWasIt = (n: number): string => {
    print("hey");
    if (n <= 0) {
        print("Dude!");
        return "cool,";
    }
    if (n === 4) {
        return "It was " + howCoolWasIt(n - 1);
    } else {
        return "very " + howCoolWasIt(n - 1) + " dude";
    }
};

main();
```



Throwing some more business in there

```
export let main = async () => {
    let x = howCoolWasIt(4);
    print(x);
};

let howCoolWasIt = (n: number): string => {
    print("hey");
    if (n <= 0) {
        print("Dude!");
        return "cool,";
    }
    if (n === 4) {
        return "It was " + howCoolWasIt(n - 1);
    } else {
        return "very " + howCoolWasIt(n - 1) + " dude";
    }
};

main();
```



Throwing some more business in there

```
export let main = async () => {
    let x = howCoolWasIt(4);
    print(x);
};

let howCoolWasIt = (n: number): string => {
    print("hey");
    if (n <= 0) {
        print("Dude!");
        return "cool,";
    }
    if (n === 4) {
        return "It was " + howCoolWasIt(n - 1);
    } else {
        return "very " + howCoolWasIt(n - 1) + " dude";
    }
};

main();
```

Throwing some more business in there

```
export let main = async () => {
    let x = howCoolWasIt(4);
    print(x);
};

let howCoolWasIt = (n: number): string => {
    print("hey");
    if (n <= 0) {
        print("Dude!");
        return "cool,";
    }
    if (n === 4) {
        return "It was " + howCoolWasIt(n - 1);
    } else {
        return "very " + howCoolWasIt(n - 1) + " dude";
    }
};

main();
```

code BEFORE recursive call **code AFTER recursive call**

"It was very very very cool, dude dude dude"

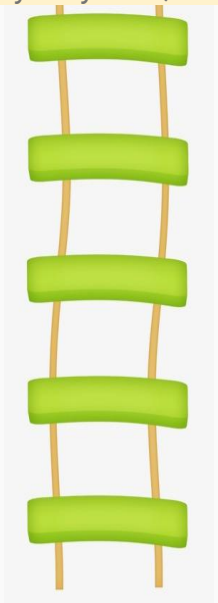
"hey" printed

"hey" printed

"hey" printed

"hey" printed

"hey" printed "Dude!" printed



Environment Diagrams

- Add a new frame on the stack **every time a function is called**
- Keep primitive values on the stack
  - “local”
- Keep reference type variables on the heap
  - same object regardless of frame you’re in

Environment Diagrams Example

```
export let main = async () => {
    let array = [1, 2, 3];
    let x = recur(array, array.length);
    print(x);
};

let recur = (a: number[], n: number): number => {
    if (n <= 0) {
        return 0;
    }
    n--;
    a[n]++;
    return a[n] + recur(a, n);
};

main();
```

Check-in and Hot Date

Check-in Code: **E022A**

Talk about your Halloween costumes for this year!

Recursive data type

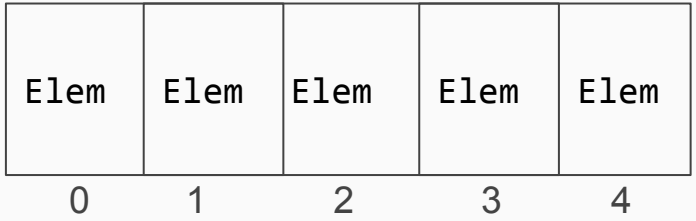
A type that has a property that refers to an object of it’s same type

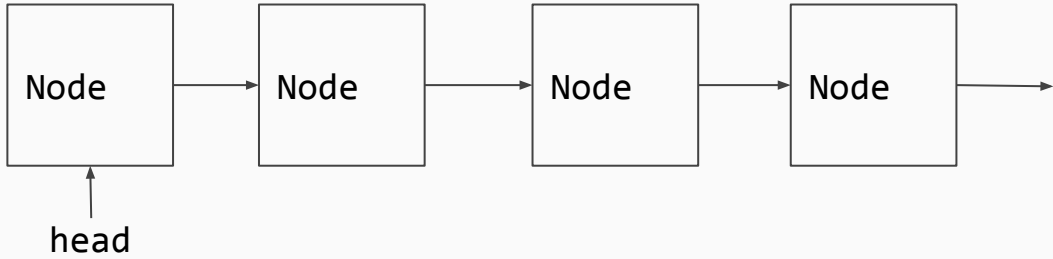
Like a Node!

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

Lists

- Looks like an array from the big picture
- Up close, it's a **recursive data type**
- Composed of objects of the **Node** type

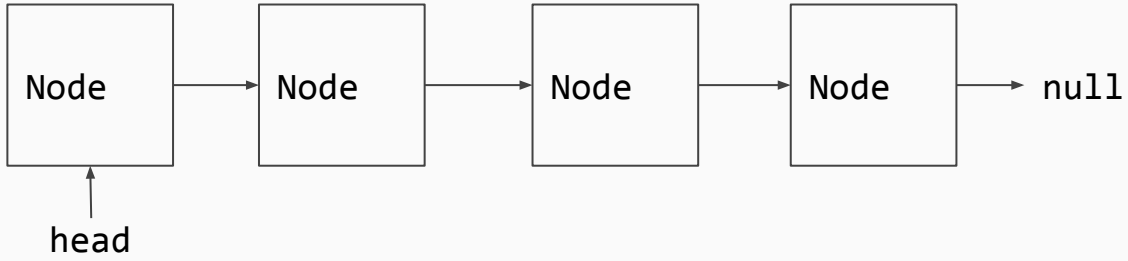




Lists cont.

- A list is a sequence of zero or more values of the same time
- Lists are **null terminated**, the node at the end of a list will have a **next** property of **null.**

Interacting with Lists



You can’t see all of it at once
You can’t index it
So…..?

cons

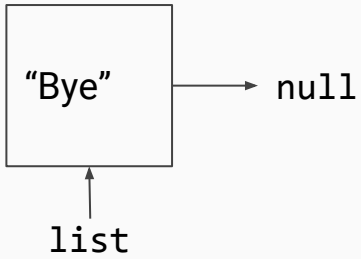
You can **cons**truct a list by using the **cons** function.

```
let cons = (value: string, list: Node) => {...}
```

2 parameters, one is the **value** you're adding onto a list, and the second is the **list** you're adding it onto.

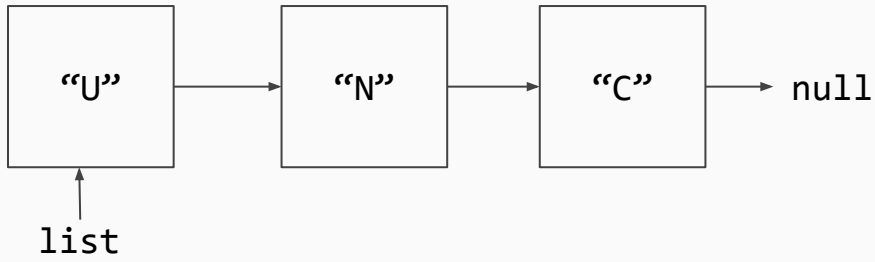
cons examples

```
let list = cons("Bye", null);
```



cons examples

```
let list = cons("U", cons("N", cons("C", null)));
```



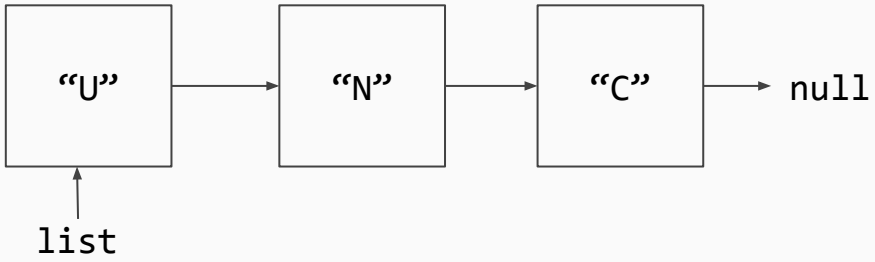
cons practice

What gets printed here?

```
let list = cons("O", null);
list = cons("L", list);
list = cons("L", list);
list = cons("E", list);
list = cons("H", list);
print(list);
```

first

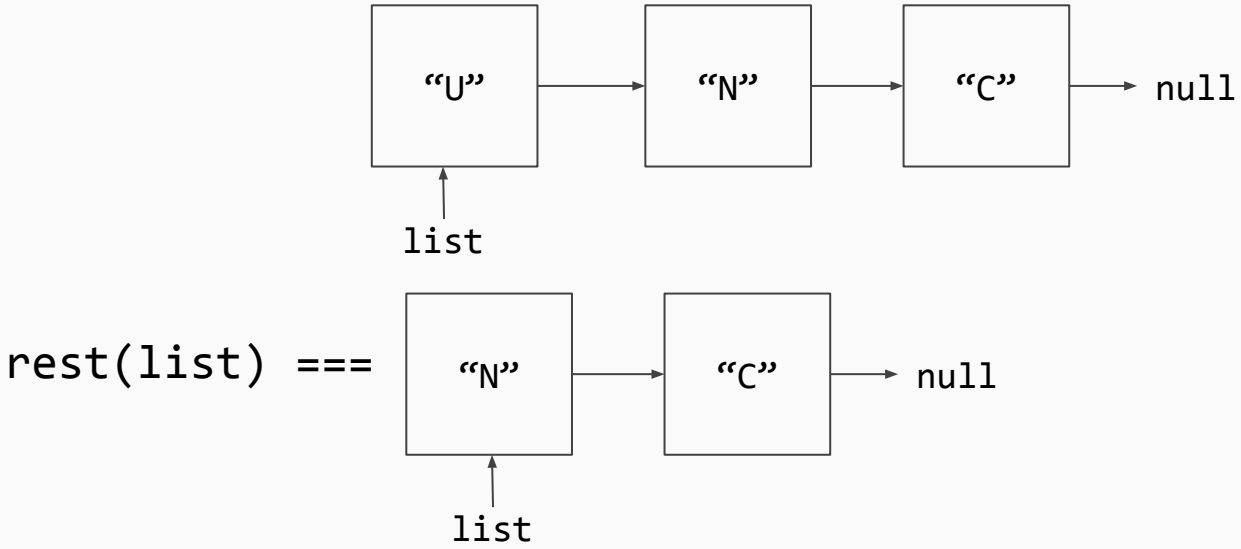
Grabs the first element of a list!




```
first(list) === “U”
```

rest

Gets the rest of a list!



Processing Lists

A list is a recursive data type, so we should process them recursively.

General Workflow:

1. Check if we've reached our base case.
2. Process **first** element of list.
3. Recursively repeat with **rest** of list.

Example: printerFn

```
export let printerFn = (list: Node<string>):void => {
  if (list === null) {
    print("That's your list folks");
  } else {
    print(first(list));
    printerFn(rest(list));
  }
};
```

```
let list = cons("U", cons("N", cons("C", null)));
printerFn(list);
```

U
string

N
string

C
string

That's your list folks
string

printerFn

```
export let printerFn = (list: Node<string>):void => {
  if (list === null) {
    print("That's your list folks");
  } else {
    print(first(list));
    printerFn(rest(list));
  }
};
```

printerFn2

```
export let printerFn2 = (list: Node<number>):void => {
  if (list === null) {
    print("That's your list folks");
  } else {
    print(first(list));
    printerFn2(rest(list));
  }
};
```

printerFn3

```
export let printerFn3 = (list: Node<boolean>):void => {
  if (list === null) {
    print("That's your list folks");
  } else {
    print(first(list));
    printerFn3(rest(list));
  }
};
```

Generic Types

Generalizing functions and algorithms to work as intended, no matter the type

- Useful when performing the same function/algorithm on collections of data of different types
- Parameterize the data types used at a class/function level

… WHY?

To reduce repetition in code

Rule of Thumb

2+ functions that differ ONLY in parameter types/return type?

Use a generic function!

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;
};
```

Syntax: Using the type T in a function

```
let ex = <T> (var1: T, var2: Node<T>): Node<T> => {
    return var2;
};
```

1. **Add a <T> (“diamond”) before the parameter list**
2. Identify all types that would change between your otherwise identical functions
3. Replace the types with the generic type T
    a. *Replace the necessary types of variables declared inside the body of your function with type T!*

Syntax: Using the type T in a function

```
let ex = <T> (var1: T, var2: Node<T>): Node<T> => {
    return var2;
};
```

1. Add a <T> (“diamond”) before the parameter list
2. **Identify all types that would change between your otherwise identical functions**
3. Replace the types with the generic type T
    a. *Replace the necessary types of variables declared inside the body of your function with type T!*

Syntax: Using the type T in a function

```
let ex = <T> (var1: T, var2: Node<T>): Node<T> => {
    return var2;
};
```

1. Add a <T> (“diamond”) before the parameter list
2. Identify all types that would change between your otherwise identical functions
3. **Replace the types with the generic type T**
    a. ***Replace the necessary types of variables declared inside the body of your function with type T!***

An Better Approach: Using Generics!

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

This approach is effective for functions that aren't dependent upon the types of their parameter(s) or return type!

Use generics whenever possible for concise + easily digestible code

Generic Class
Node<T>

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

Why is the Node class a generic class?

Generic Class Node<T>

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

Generic classes are useful when two separate classes would differ only by the types of their properties.

- Rather than having different Node classes for strings, numbers, booleans, etc, use T