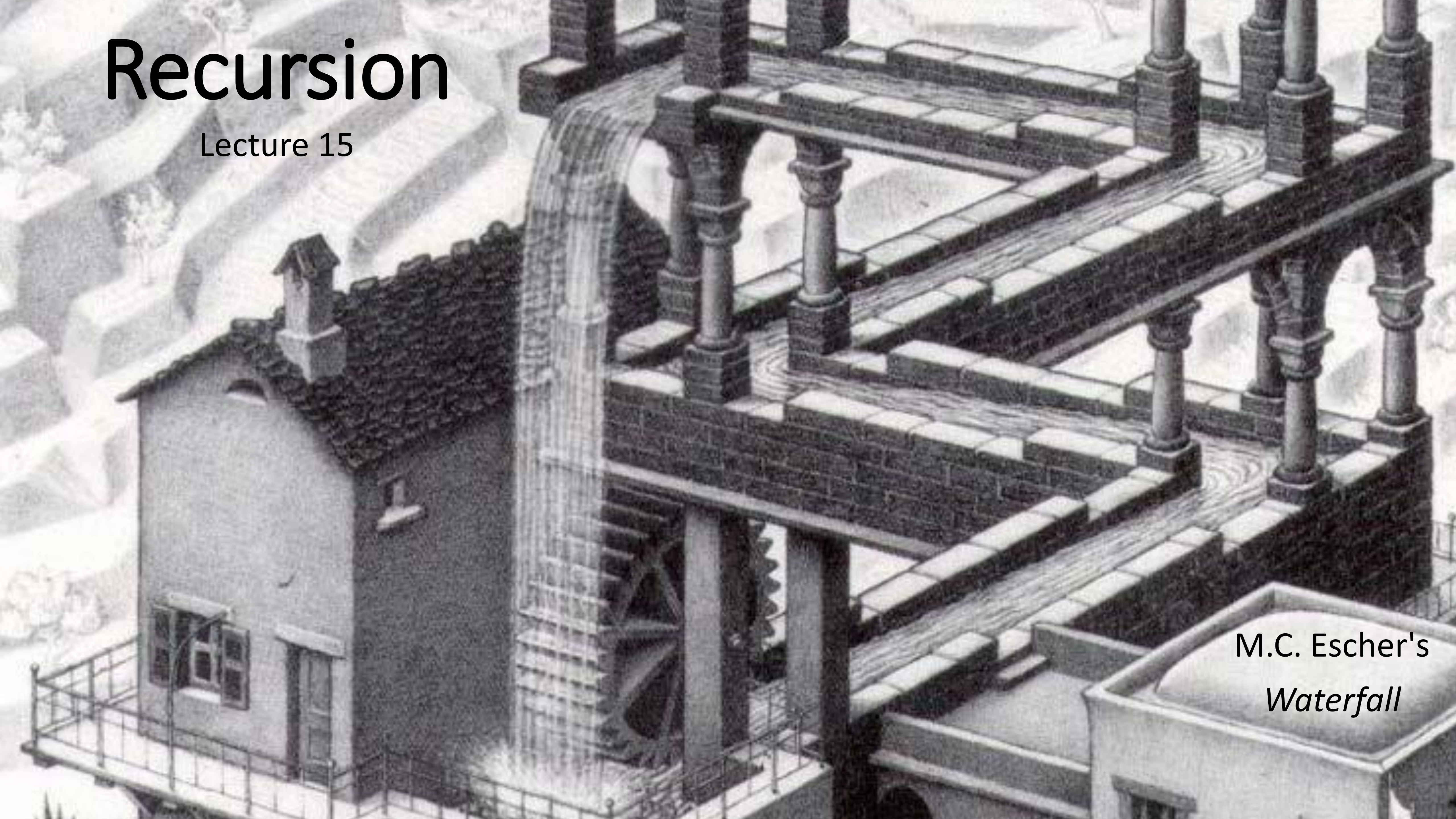


Recursion

Lecture 15

M.C. Escher's
Waterfall

Hack-110.com - RSVP for Hack110

- Starts Friday, April 12th at 7pm
- Design your own final project and work with a friend
- Talks on the transition to 401, web dev, game dev, app dev
- Free 110 T-Shirts for those hacking past midnight
- Free food, fun activities, helpful UTAS, a ton of fun!
- **Learn More and RSVP at Hack-110.com!**

Panel Discussion on Experiences in CS - Tomorrow!

- Everyone interested in learning more about what it's like to study computer science and succeed in industry or academia should attend!

- Tomorrow (Wednesday 3/27) at 6pm in SN014

- Panel Includes
  - Gabi Stein - Senior UTA on the 110 Team - Joining Microsoft Fulltime
  - Diane Pozefsky - IBM Technical Fellow and Director of Undergrad Studies in CS
  - Whitney Jenkins - UNC Grad, Program Manager at Microsoft
  - Kipp Williams - Sophomore CS Major - Executive Director of Queer Hack
  - Junior Oliva - Professor - PhD at Carnegie Melon in Machine Learning

Warm-up Question #0: When the **main** function runs, what is printed?

```
let main = (): void => {
   f(2);
};

let f = (n: number): void => {
    print(n);
    if (n <= 0) {
        print("!");
    } else {
        f(n - 1);
    }
    print(n);
};
```

Hands-on: A Recursive Function

- Open lec15 / 00-recursion-demo-app.ts
- Under the print statement that prints "Recur!", still inside of the else block, try calling:

```
f(n - 1);
```

- Now try changing the call to **f** in **main** to use **2** as an argument instead of 0.
- Check-in on PollEv.com/compunc once you've tried f(2)
  - Done? Try other numbers, as well...

Recursion

- A recursive function is **a function that calls itself**
- It's a *beautiful*, *powerful* concept!
- Anything you can do with a loop, you can do with recursion.
    - Many algorithms in computer science are simpler when expressed recursively.

Recursion and the Call Stack

- Recursion works because every time a function call is encountered a new frame is added to the call stack
- A frame's local variable values are independent of other frames'
- We are able to leave a bookmark in the middle of a function, add a new frame, and jump into the same function!
- You've been doing this *between* functions and methods all semester. The only difference is with recursion we're calling the *same* function.

What happens when the following code runs?

```
let main = async (): void => {
    print("Enter main()");
    f(10);
    print("Leave main()");
};

let f = (n: number): void => {
    print("Enter f(" + n + ")");
    f(n - 1);
    print("Leave f(" + n + ")");
};
```

- Open 01-stack-overflow-app
- Run it.
- Scroll wayyy down...

What's a "**Stack Overflow Error**"?

- Remember, every function call gets its own **frame** on the **call stack** for storing its parameters and local variables
- Here we're continuing to infinitely call the same function using recursion
- Each frame on the stack requires memory, though!
- When the stack grows too tall, we run out of memory and crash.

How do we *prevent* Stack Overflows?

- The same way we prevent *infinite loops*.
- We need a *test condition* at which point we *stop recurring.*
- This is called a **base case**.
- Additionally, we need something to *change* as we recur.
  Typically, this is the *argument* we pass to successive recursive calls.

Hands-on #2: Adding a Base Case

1. When you run this file, you will see another Stack Overflow error. Let's fix it!

2. In the function **f**, add an if statement that tests if n is less than or equal to 0
    1. If so, print out **"Base case!"**
    2. Otherwise, call the **f** function again *recursively* with an argument of n – 1

3. Try running again! Try changing the argument in **main** to test.

4. Check-in when you're complete!

Base Case

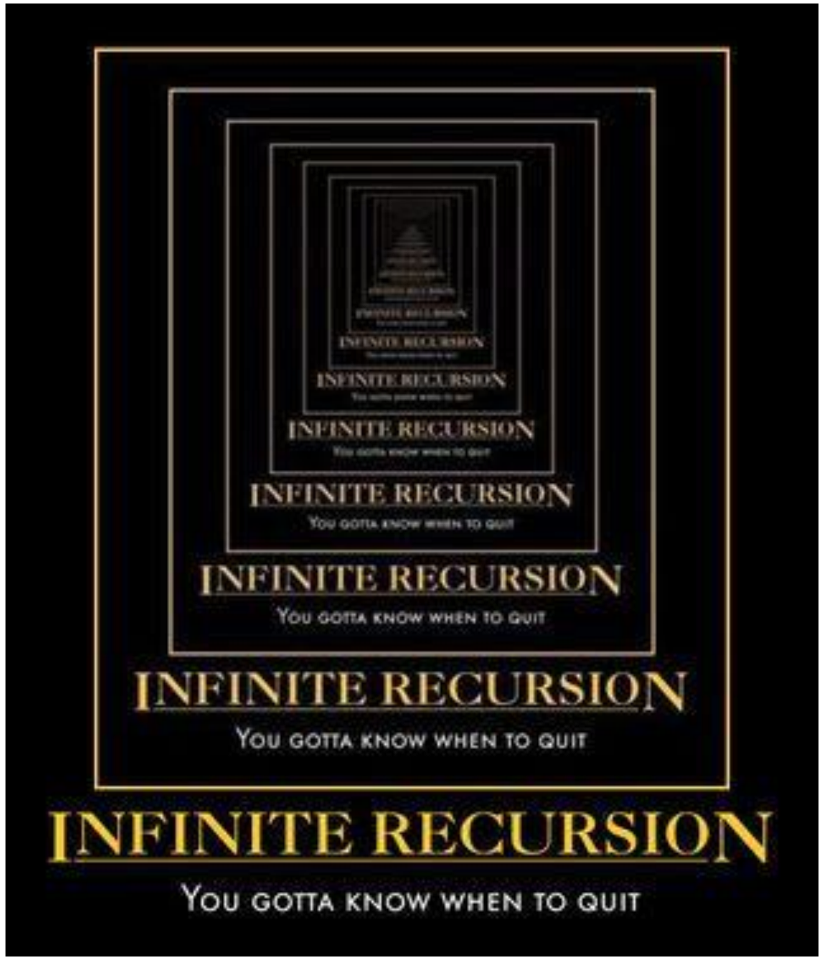
Recursive Case

```
function f(n: number): void {
   print("Enter f(" + n + ")");
   if (n <= 0) {
      print("Base Case!");
   } else {
      f(n - 1);
   }
   print("Leave f(" + n + ")");
}
```

1. Notice when a function call reaches the **base case**, it ***does not* recur**

2. The **recursive case** is recurring and changing the argument it recurs with
   - Notice this argument is bringing the value n *towards* to the base case. What would happen if it were *`f(n+1)`*?

Every Recursive Function *Needs* a Base Case



- The base case marks the end of a series of recursive function calls.
- Until the computer reaches the base case, each recursive call is adding another frame to the stack.
- Once the base case call returns, control returns to the return address inside its originating call, and so on.

Compound Data Type Properties

- So far we've focused on classes with value-type properties, such as:
  - string
  - number
  - boolean

- Properties can also be reference types, like:
  - arrays
  - objects

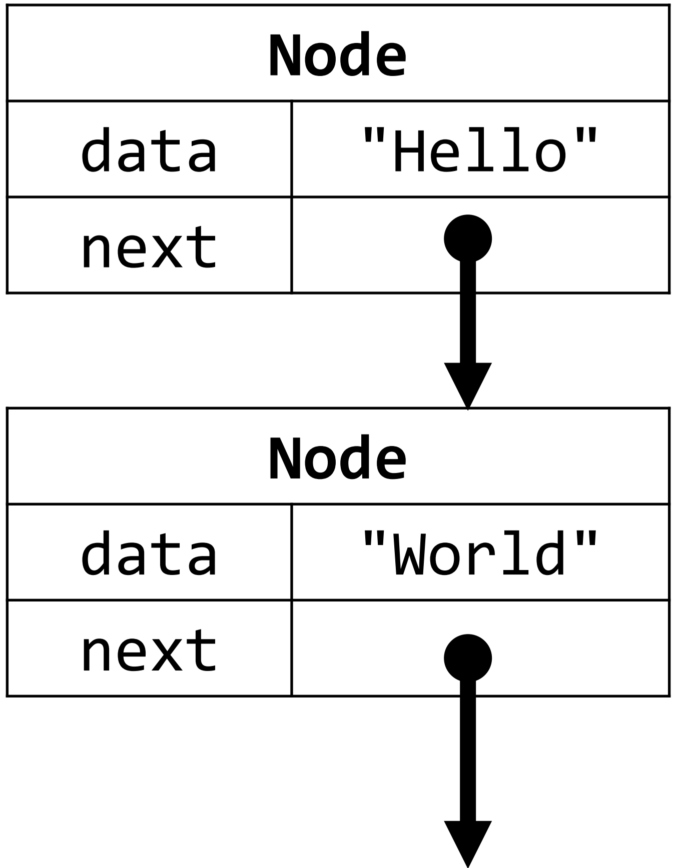
```
class Person {
    name: string = "";
    pets: Dog[] = [];
}

class Dog {
    name: string = "";
    breed: string = "";
}
```

Recursive Data Types

- A property *can* refer to another object of the *same type*
- Notice the class **Node**. It has a property named **next** and its value must be... *another Node*.
- This is a recursive data type!
- We'll discuss how to initialize a recursive property to avoid infinite recursion shortly...

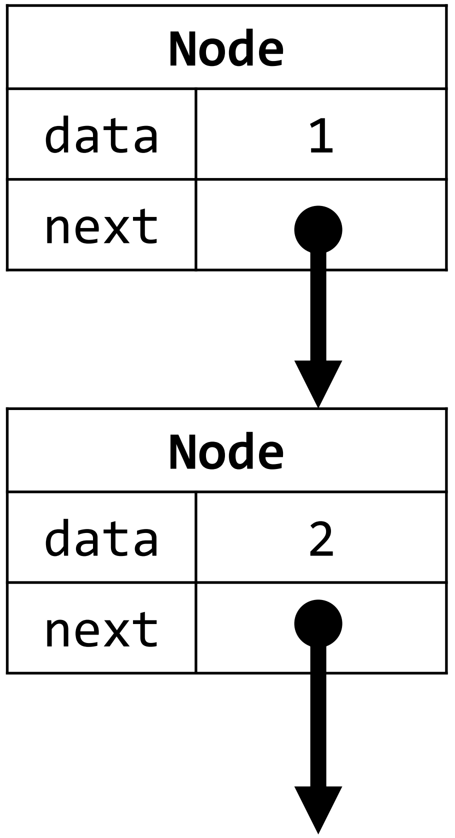
```
class Node {
   data: string = "";
   next: Node = ?;
}
```



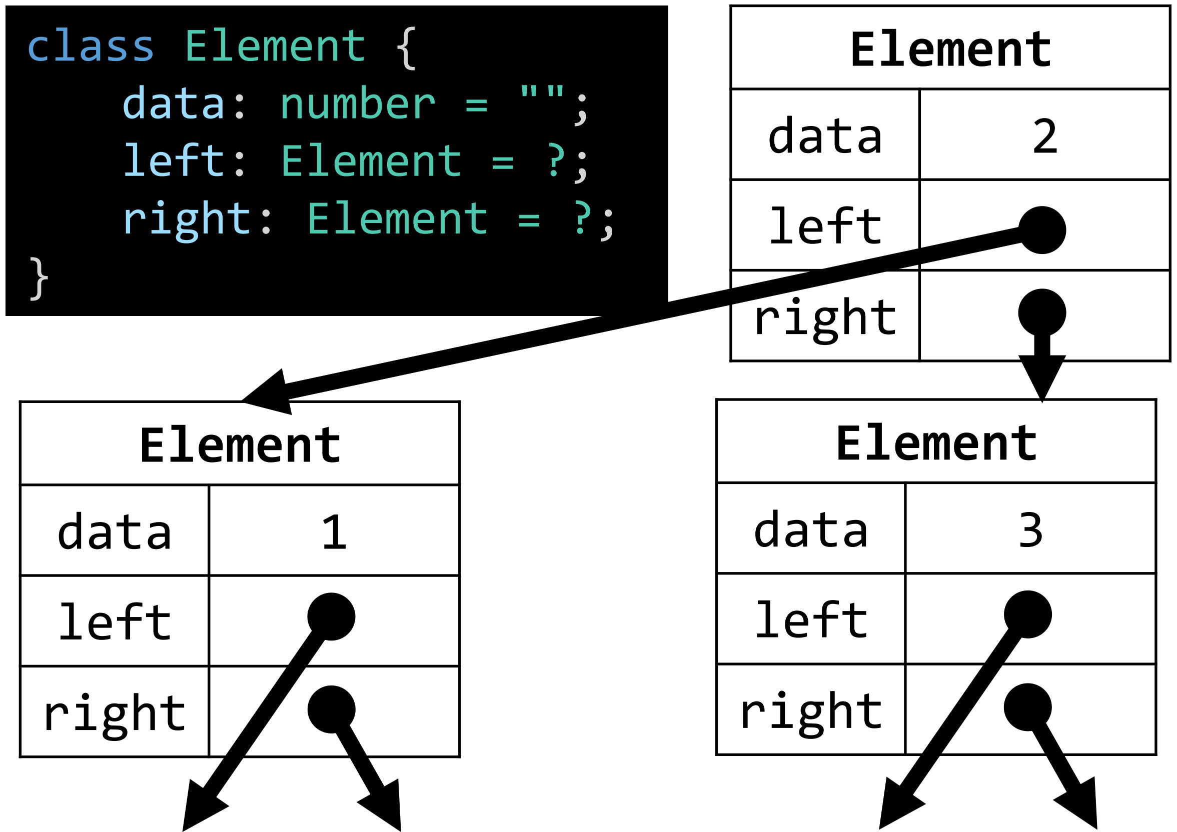
Data Structures

- You can use this ability to form **data structures** with different properties and uses.
- In COMP110, you'll explore the Linked List (left)
- In COMP410, you'll explore other data structures like Trees (right) and Graphs

```
class Node {
    data: number = "";
    next: Node = ?;
}
```

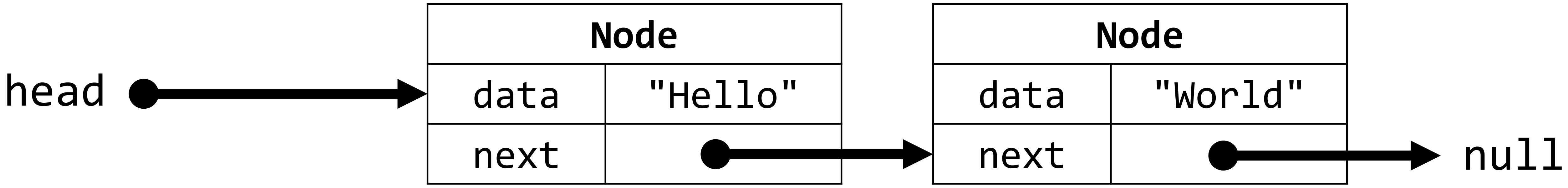


```
class Element {
    data: number = "";
    left: Element = ?;
    right: Element = ?;
}
```



Linked List

- A classic, simple data structure in Computer Science
- Formed by chaining together a sequence of objects
  - The first node is conventionally called the **head**
- Linked lists are more cumbersome to work with than arrays
  - However, they're important for understanding and exploring fundamentals including:
  - **`null`** values
  - References
  - Recursive algorithms

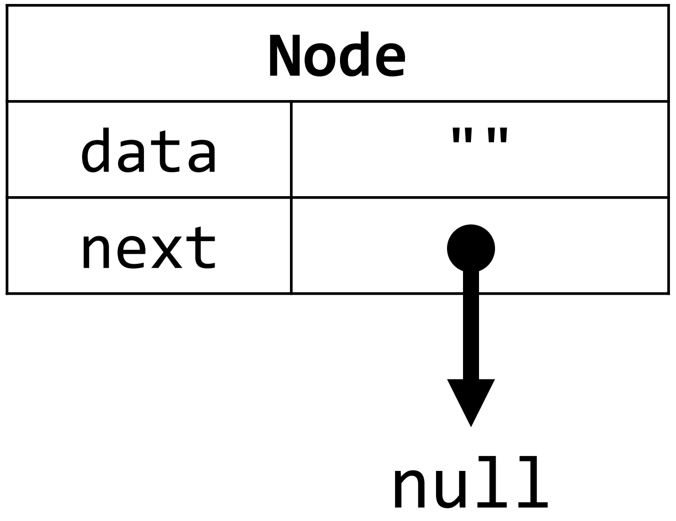


What is a recursive property's "base case"?

- If a Node refers to a `next` Node, and the `next` Node refers to another `next` Node, then *when does it end?*

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

- Recursive properties are terminated with a special value called **`null`**.
    - It is a "reference to nowhere" that you can read as "this property refers to nothing."
- Our linked lists are "**null** terminated".



Hands-on: Constructing a Linked List Node-by-Node

- Open 00-node-app.ts

1. Before you begin, understand the two Nodes already constructed. Try diagramming these out on paper.

2. At TODO #1 - Construct a new Node object with the properties described in the comments. Assign it to head.

3. At TODO #2 - Print the value of the last Node in the list that should now contain the data "C".

- Check-in when you've got UNC printing out!

Costs of not abstracting away details...

- The last hands-on suffers from two important problems:

1. It's cumbersome and error prone to construct a linked list by manipulating the properties of its nodes directly.

2. Our program now depends on the details of the Node class' properties. Changing the implementation of Node would permeate all aspects of our program that relied upon a Node.

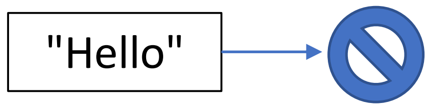
- These are the costs of programming at the wrong level of abstraction.

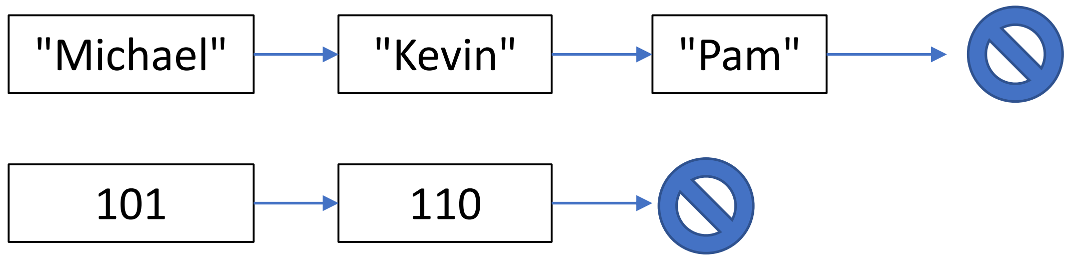
- Let's try working with the abstract concept of a "list" instead...

What is a Linked List?

1. A **List** may be empty

2. A List may be a sequence of one or more values of the same type





3. Each item in a List is called a **Node**

4. The end of a List is marked by a special value called **`null`**

What can you *do* with a list?

1. You can *construct* a new node at the front of another list
   - via the ***`cons`*** function

2. You can ask a list for its first value
   - via the ***`first`*** function

3. You can ask a list for a sub-list of itself, excluding the first value
   - via the ***`rest`*** function

- That's it! By default, these are the only operations you can do with a list!
  - These are all the capabilities you *need.*
  - Using these simple operations, you will write more advanced functions, or abstractions, to perform more sophisticated tasks with lists.

cons-tructing a list, value-by-value (1/2)

- The function **cons** is short for "construct List".

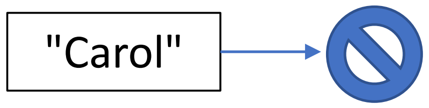
- The **cons** function requires 2 parameters:
  1. The value you are adding on to the front of the `List`
  2. The `List` you are adding the value onto

- The **cons** function *returns* a new `List` with the value added to the front.

The **cons** function **usage** (2/2)

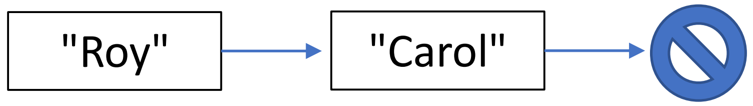
- Construct a list with a single Node in it

```
let names = cons("Carol", null);
```



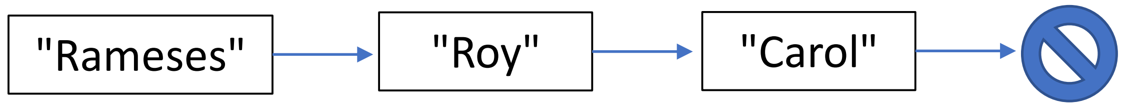
- Construct a list with two values in it

```
names = cons("Roy", cons("Carol", null));
```



- Modify a list by adding onto itself

```
names = cons("Rameses", names);
```



Follow-along: 01-list-abstraction-app.ts

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

// TODO: cons U onto list
list = cons("U", list);
print(toString(list));
```

The **`first`** function returns the first value of a list

- The List function **`first`** returns the first value in a non-empty list
  - Warning: the `first` function *will* error if given an empty List

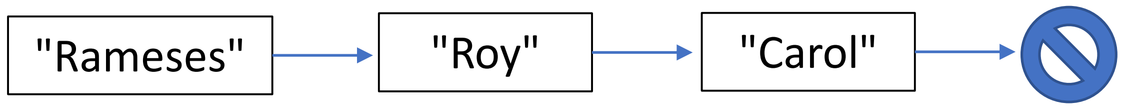
- The **`first`** function requires one parameter: a *non-empty* list

- Usage:

```
let series = cons(10, cons(20, cons(30, null)));
print(first(series));
```

$\longrightarrow$ 10

`first` Examples Visualized

Function Call	**Return Value**
`first(`  `)`	"Rameses"
`first( null 🚫 )`	ERROR

The **rest** function returns a sub-List, w/o first value

- The function **rest** returns a list with every value except the first
  - Warning: the `rest` function *will* error if given an empty `List`

- The **rest** function requires one parameter: a non-empty List.

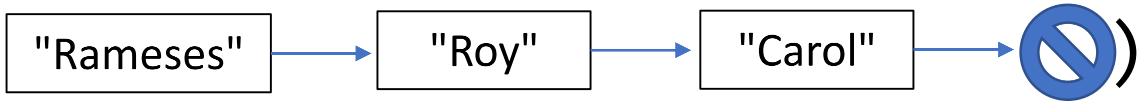
- Usage:

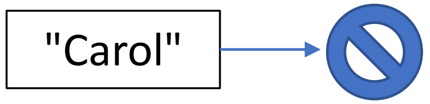
```
let series = cons(10, cons(20, cons(30, null)));
print(toString(rest(series)));
```

```
20 → 30 → null
```

rest Examples Visualized

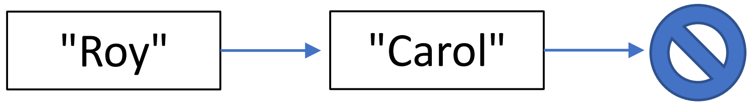
Function Call

`rest(`  `)`

`rest(`  `)`

`rest( null` 🚫 `)`

Return Value



🚫

ERROR

Follow-Along: Print the 2nd & 3rd Entries in the List

```
// TODO: Print the 2nd and 3rd Values
print(first(rest(list)));
print(first(rest(rest(list))));
```

How are `cons`, `first`, and `rest` implemented?

- They're defined in list.ts
- These are very simple functions!
- What's the big deal?
- We've addressed the two shortcomings of our initial example! How?

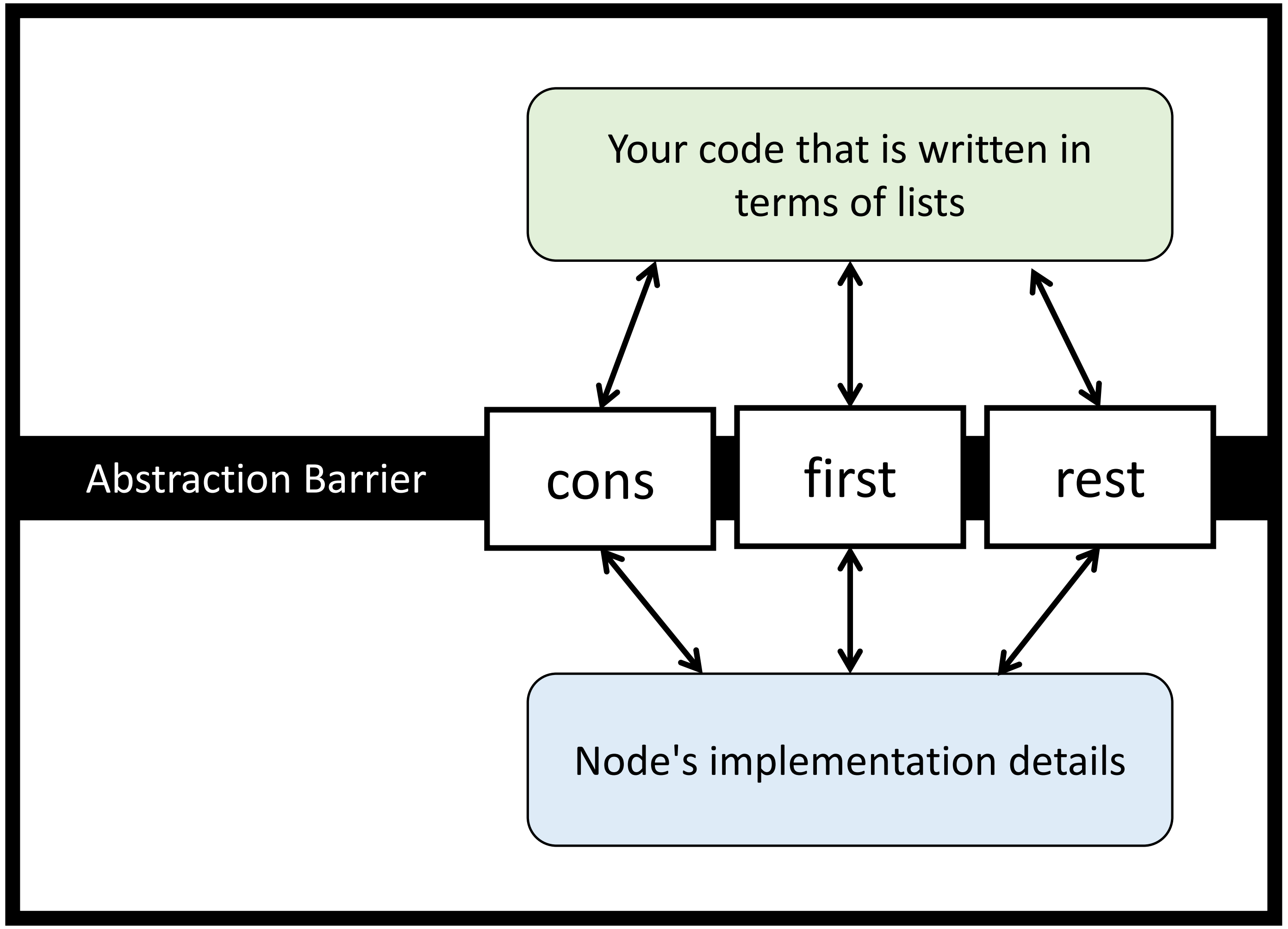
```
/* Constructor */
export let cons = (data: string, next: Node): Node => {
    let n = new Node();
    n.data = data;
    n.next = next;
    return n;
};

/* Selectors */
export let first = (n: Node): string => {
    return n.data;
};

export let rest = (n: Node): Node => {
    return n.next;
};
```

Barriers of Abstraction (1 / 2)

- To build complex systems you must manage complexity by abstracting away or "hiding" implementation details
- **Abstraction barriers** are common across all engineering disciplines
    - Fundamental to organizational management, too!
- Code on one side of an abstraction barrier knows nothing about what's on the other side.

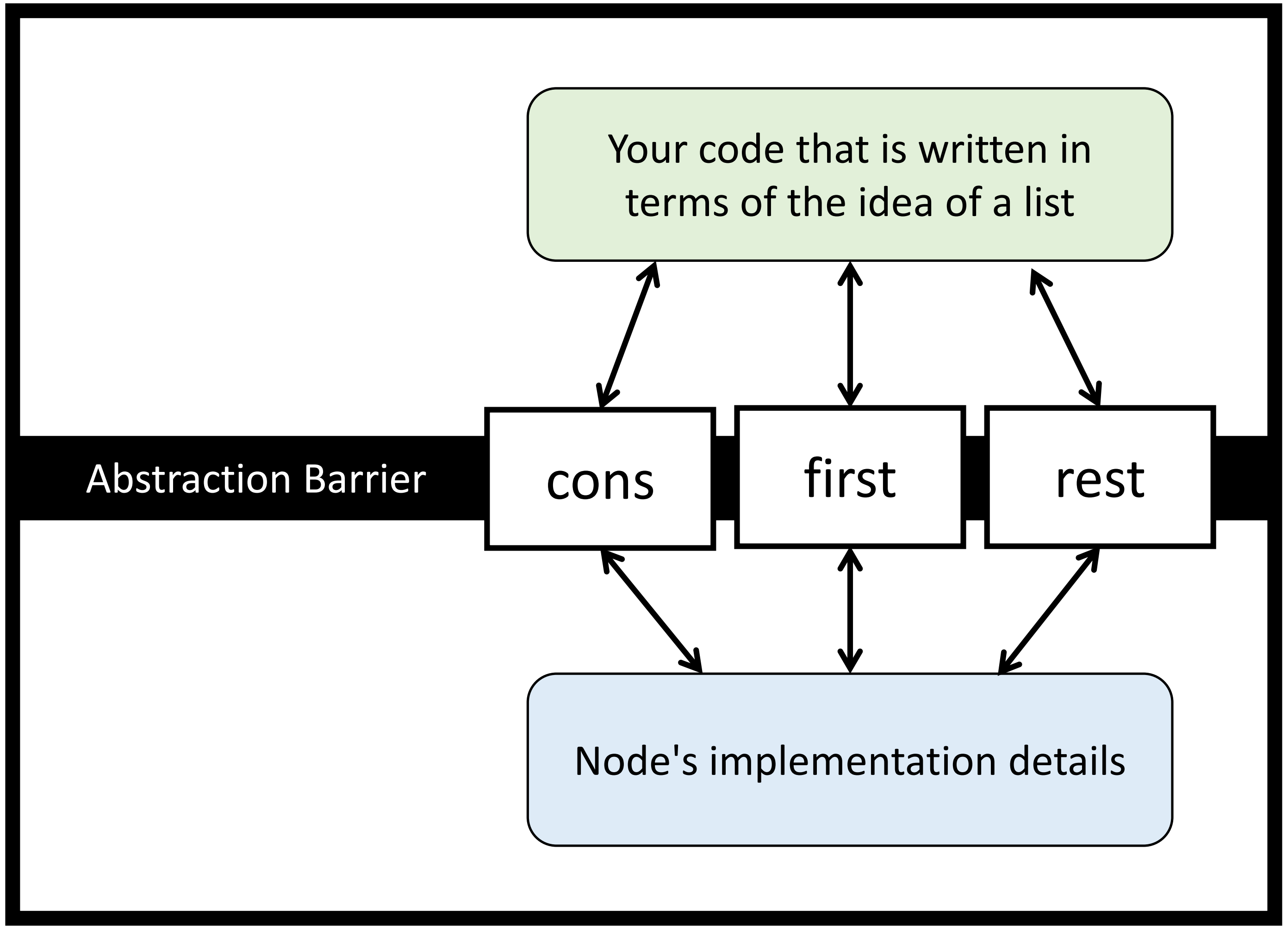

Your code that is written in terms of lists
Abstraction Barrier
cons
first
rest
Node's implementation details

Barriers of Abstraction (2 / 2)

- In our example, the **abstraction barrier** are the `cons`, `first`, `rest`.
- By writing code that *only* depends on these three functions we:

1. Simplify our code and avoid having to do all the bookkeeping.
2. Make it possible to change or improve code on either side of the abstraction barrier *independent of the other side*.

- Thus, we've solved our two problems with the opening example! In large scale software projects you will find many *layers of abstraction* each separated by a barrier.


Your code that is written in terms of the idea of a list
Abstraction Barrier
cons
first
rest
Node's implementation details

Graded PollEv Questions Thursday

- Bring 1 page of notes on this slide deck
- Questions will be re: recursion and recursive data types