Abstraction

COMP110 - Spring 2018 - Lecture 25

This Week

- Normal office hours except on LDOC
  - LDOC we close at 5pm
- Last day to submit problem sets late is Sunday 4/29 at 11:59pm
- Deliverables
  - WS5 – Due Tomorrow
  - PS6 – Due Friday
  - HACK110 Projects – You'll get an e-mail tomorrow!
- COMP110 UTA Application Deadline is Friday at 11:59pm

Finals Prep

- Final Exam Schedule
  - Section 1 – Monday, May 7th, at 12pm
  - Section 2 – Tuesday, May 1st, at 4pm
- Q&A Based Review Session this Wednesday at 5pm in SN14
- Tutoring open Friday from 12-4pm
- Final Review Session
  - In-class Thursday
  - Each Sunday preceding each exam (4/29 and 5/6) with times TBA via e-mail
- Study guide practice problems and concept inventory posted on COMP110.com
- 3 exams in 24 hours? Be sure to get your pink slip! There will be a form next class.

Final Exam Concept Inventory

Computer Science 110: Introduction to Programming

- Types
  - ☐ Primitives (`number`, `string`, `boolean`)
  - ☐ Lists (`cons`, `first`, `rest`)
  - ☐ Arrays
  - ☐ Classes
  - ☐ Functions (Functional Interfaces)
  - ☐ Inference
- Expressions
  - ☐ `boolean` operators: `&&`, `||`, `!`
  - ☐ `number` operators
    - ☐ arithmetic: `+`, `-`, `*`, `/`, `%`
    - ☐ relational: `===`, `!==`, `>`, `>=`, `<`, `<=`
  - ☐ `string` concatenation: `+`
  - ☐ Function Calls
  - ☐ Method Calls
- Variables
  - ☐ Assignment: `=`
  - ☐ Reassignment: `++`, `--`, `+=`, `-=`, `*=`, `/=`
  - ☐ Values (primitive types)
  - ☐ References (objects and arrays)
  - ☐ Scope: block vs. global
- Functions
  - ☐ Defining
  - ☐ Parameters vs. Arguments
  - ☐ `return` statements
  - ☐ `void` functions
  - ☐ Higher-order Functions
  - ☐ Generic Functions
  - ☐ Function Literals
- Recursion
  - ☐ Base Case
  - ☐ List Parameter Recursion
  - ☐ Number Parameter Recursion
  - ☐ Recursive List Generation
- Conditional Statements
  - ☐ `if` statements
  - ☐ `else` statements
  - ☐ Nested Conditional Logic
  - ☐ `else-if` syntax
  - ☐ Unreachable Code
- Loop Statements
  - ☐ `while` loops
  - ☐ `for` loops
  - ☐ Infinite loops and their prevention
  - ☐ `return` statements inside loops
- Arrays
  - ☐ 0-based indexing
  - ☐ Accessing and assigning to indices
  - ☐ `length` property
  - ☐ Looping through an array
  - ☐ `filter`, `map`, & `reduce` methods
  - ☐ 2 dimensional arrays
- Algorithmic Concepts
  - ☐ Swapping two variables
  - ☐ Linear Search
  - ☐ Binary Search
  - ☐ Sorting with a `Comparator`
  - ☐ `filter`
  - ☐ `map`
  - ☐ `reduce`
- Object-oriented Fundamentals
  - ☐ Class Definitions
  - ☐ Properties
  - ☐ Constructors and the `new` Keyword
  - ☐ Method Definitions and Calling
  - ☐ `this` keyword

Warm-up: Supposing **a**'s value is 1 and **b**'s value is 2, what is printed when the below code runs?

```
print(first(rest(cons(a, cons(b, null)))));

print(rest(rest(cons(a, cons(b, null)))));
```

Computer Science is the study of Abstraction

Process Abstraction

- Bundling of an algorithm into a reusable block of functionality.
- Simplifies the construction of more complex processes.
- Examples: includes, filter, map, reduce, sort

Data Abstraction

- Bundling of data into a reusable block of representation.
- Simplifies the construction of more complex models.
- Examples: List, Set, classes, and data structures, etc.

We build *layers* of abstraction.

Higher-level concepts are built on the foundation of simpler conceptual layers.

PS3 - Set Utilities

`includes` `isSet` `isSubset` `intersect`

`isSuperset` `isSetEqual` `union` `subtract`

List Primitives

`cons` `first` `rest`

Well designed layers of abstraction can support *other layers* built above them.

Layers can be formed made of *both* process *and* data abstractions.

One way we could model a product search on a website is to have a "Set" for each possible criteria of a product and allow the user to choose which they are interested in.

Then, we could intersect all of their interested sets to find products that match all criteria.

Product Search	`redShoes`	`affordableShoes`		`nikeBrand`
		`matchAllCriteria`		
PS3 - Set Utilities	`includes`	`isSet`	`isSubset`	`intersect`
	`isSuperset`	`isSetEqual`	`union`	`subtract`
List Primitives	`cons`		`first`	`rest`

PS3 - Set Utilities

`includes` `isSet` `isSubset` `intersect`

`isSuperset` `isSetEqual` `union` `subtract`

List Primitives

`cons` `first` `rest`

Are *cons, first,* and *rest* really primitive functions?

No! They're abstractions, too.

Notice how far we have come this semester *without* you needing to know *how* they are implemented. **This is the power of abstraction.**

What did **`cons`**, **`first`**, and **`rest`** provide?

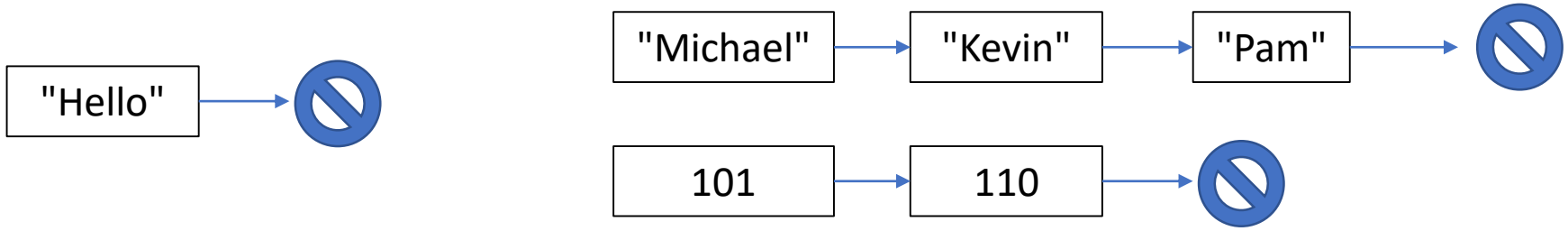
- Each provided a guarantee about what to expect when it was used.
- These guarantees can be asserted in term of one another.

```
a === first(cons(a, null))

null === rest(cons(a, null))

b === first(rest(cons(a, cons(b, null))))

null === rest(rest(cons(a, cons(b, null))))
```

What is a **`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`**

How do we represent the concept of a Node?

- With a class!

- We'd like each Node to generically hold any type of value, so we'll declare it generically for type T.
  - We'll think through and add the properties of a Node shortly.

- In list.ts:

```
export class Node<T> {

}
```

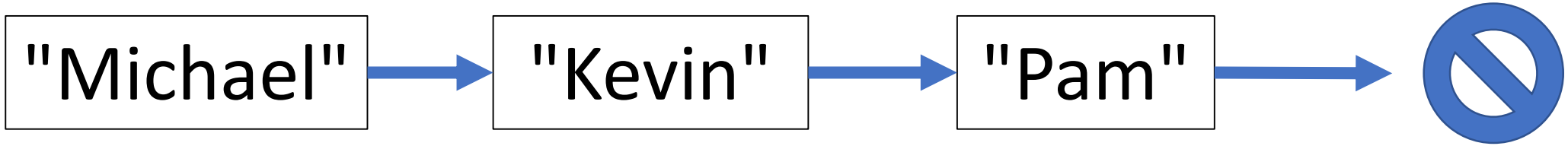
How can we express the idea of List being either a **`Node`** *or* empty (**`null`**)?

- We'll use a type definition we haven't explored in 110, and you don't need to concern yourself with, but informs the compiler of this:

```
export type List<T> = Node<T> | null;
```

- This says: "the generic type List, for any type T, is either a Node of type T or null."

What *properties* does each `Node<T>` object need?



1. Each Node needs to store a ***value*** of type T

2. Each Node needs to store a *reference* to *the* ***next*** *Node* or *null*.
   - Since the List<T> type is a Node or null, our 2nd property is a List<T>

Let's add our properties…

- You'll notice that we didn't initialize the value property.
- How could we if we don't yet know what type T is?
- We'll initialize it in the constructor!

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

Hands-on: Initializing a Node

- In **list.ts**, add a **constructor** to the Node class

- It has two parameters:
  - `value` of type **T**
  - `next` of type `List<T>`

- Initialize the properties of the object being constructed with the parameter values

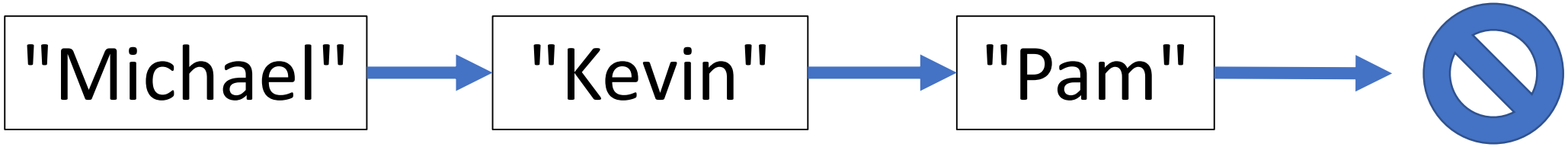
- In **00-list-app.ts**, uncomment the import Node line and in the main function. Declare a variable named aList and assign it a new Node object by calling the Node constructor you just wrote using any string you want and null as arguments.

- On another line, try printing out the Node's value. Check-in when complete!

```
export class Node<T> {
   value: T;
   next: List<T> = null;

   constructor(value: T, next: List<T>) {
      this.value = value;
      this.next = next;
   }
}
```

```
let aNode = new Node("Pam", null);
print(aNode.value);
```

Let's setup a chain of Nodes



- Notice this feels oddly similar to building a list with cons and accessing elements with first and rest…
- But it also feels *worse*, right?
  - There are more details we're worried with.
- Good abstractions hide unimportant details.

```
let names = new Node("Pam", null);
names = new Node("Kevin", names);
names = new Node("Michael", names);

if (names.next !== null) {
   print(names.next.value);
}
```

What could we *do* with a **`List`**?

1. We can add a value at the front of a List
   - via the *`cons`* function

2. We can ask the List for its first value
   - via the *`first`* function

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

- That's it! By default, this is all we can do with a **`List`**!
  - These are all the capabilities we *need.*
  - Using these simple operations, we will write our own more advanced functions, or abstractions, to perform more sophisticated tasks with Lists.

Follow Along: Implement a **cons** Function

- What does cons do? It ***cons**tructs* a new Node!

```
export let cons = <T> (value: T, next: List<T>): Node<T> => {
   return new Node(value, next);
};
```

- In our implementation today, *cons* is an alias for Node's constructor.
- Let's update our code back in main…

```
let names = cons("Pam", null);
names = cons("Kevin", names);
names = cons("Michael", names);
```

Follow Along: Implement a **`first`** Function

- What does first do? It returns a List's first value.

```
export let first = <T> (list: List<T>): T => {
      return list.value;
};
```

- This should work right?

- Not so fast: a List<T> is *either* a Node *or* it's *null.* We learned that the *first* function is undefined when the *list* is *null*, so let's throw an Error if so.

```
export let first = <T> (list: List<T>): T => {
    if (list === null) {
        throw new Error("Cannot call first on null.");
    } else {
        return list.value;
    }
};
```

Hands-on: Implement a **rest** Function

- In list.ts – try to implement a generic *rest* function for a List of type T that works as you would expect it to.

- Look to the *first* function for inspiration, they're very similar.
  The *rest* of the list is just the *next* property of a Node.

- What should the parameter be? The return type?

- Once you have it, try *importing it* in 00-list-app.ts and *using it* to test.

```
export let rest = <T> (list: List<T>): List<T> => {
   if (list === null) {
      throw new Error("Cannot call rest on null.");
   } else {
      return list.next;
   }
};
```

```
let names = cons("Pam", null);
names = cons("Kevin", names);
names = cons("Michael", names);
print(first(rest(rest(names))));
```

That's all there was to **`cons`**, **`first`**, and **`rest`**!

- In 30 lines of code we implemented a very powerful abstraction.
- We can now *use* these exactly as we did earlier in the semester.
- Let's try writing a recursive function to practice some recursion.

Hands-on: toString

- In 00-list-app.ts define a function named **`toString`** with the following signature:

```
<T> (list: List<T>): string
```

- It should return a string that is either *"null"* if *list* is null, or
  `<first> + " -> " + <natural recursion of toString>`

- Try printing a call of **`toString`** on `names` from main. You should get:
  "Michael -> Kevin -> Pam -> null"

- Check-in when complete!

```
let toString = <T> (list: List<T>): string => {
    if (list === null) {
        return "null";
    } else {
        return first(list) + " -> " + toString(rest(list));
    }
};
```

The Power of Properly Layered Abstractions

- The 00-list-app.ts program depends on a layer of abstraction defined in the concepts: **List, cons, first, rest**

- We could instead make it depend on "introcs/list" and it would work just the same!
  - It turns out these two implementations *are* the same, but they don't have to be!

- Let's look at an alternate implementation of this layer of abstraction in **`other-list.ts`**

Implementing List with arrays

- In other-list.ts you'll find an alternative implementation of this layer of abstraction, including `List, cons, first, rest`, and notice there are *no classes used* here. Only arrays!

```
export type List<T> = T[] | null;

export let cons = <T> (value: T, next: List<T>): T[] => {
    if (next === null) {
        return [value];
    } else {
        return [value].concat(next);
    }
};

// elided
```

Swapping out a layer of abstraction

- Open 01-abstraction-app.ts
  - Try importing List, cons, first, and rest from "./list" instead – your code!
  - Try importing the same from "./other-list" after that. It still works!

- By relying upon layers of abstraction you are agreeing that the details of how exactly something works is *someone else's concern.*

- This is a *wonderful* thing! We make progress by standing on the shoulders of giants.
  - Carpenters shouldn't forge nails and hammers.
  - Our ability to create larger, more capable programs requires us to depend on larger, more capable abstractions. COMP401 and 410 are concerned with this.

PS3 - Set Utilities	`includes`	`isSet`	`isSubset`	`intersect`
	`isSuperset`	`isSetEqual`	`union`	`subtract`
List "Primitives"	`cons`	`first`	`rest`	
Language Primitives	`classes`	`functions`	`(or arrays!)`	

What layer of abstraction do our List "primitives" depend on?

The programming language itself!

But what abstractions does a programming language depend on?

PS3 - Set Utilities	`includes`	`isSet`	`isSubset`	`intersect`
	`isSuperset`	`isSetEqual`	`union`	`subtract`
List "Primitives"	`cons`	`first`	`rest`	
Language Primitives	`classes`	`functions`	`(or arrays!)`	
…				
Physics	`electrons`	`current`	`voltage`	

"It's turtles all the way down."

Applications	**COMP401** **COMP426 – Web Programming**	**COMP433 – Mobile Computing** **COMP562 – Machine Learning**
	COMP110	
Systems	**COMP431 Networks**	**COMP530 Operating Systems**
	COMP521 Files & Databases	
Data Structures and Algorithms	**COMP410 Data Structures**	**COMP550 Algorithms**
Language Primitives	**COMP520 Compilers**	
Machine Primitives	**COMP411 Computer Architecture**	**COMP541 Digital Logic**