CSE 331
Software Design & Implementation

Autumn 2023

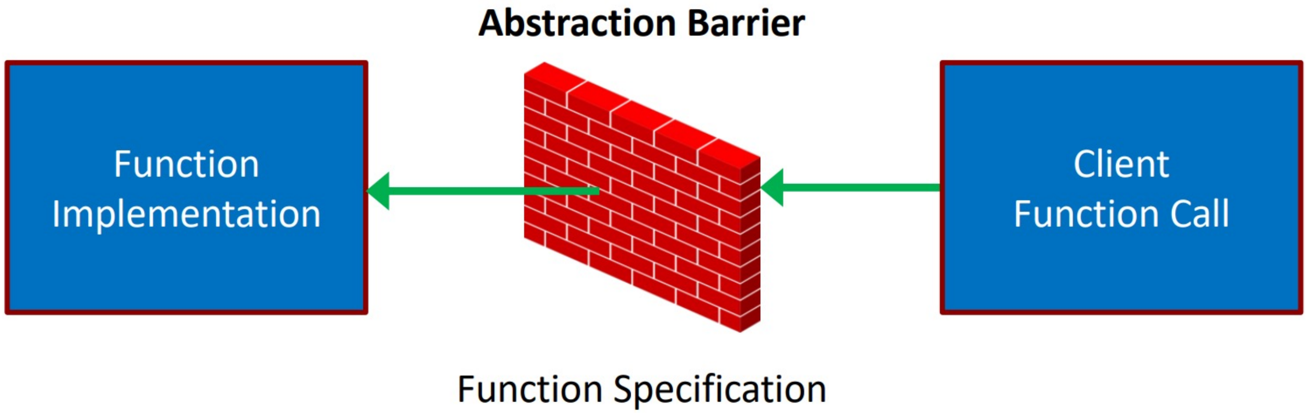
Section 5 – Functional Programming III

Administrivia

- HW5 released later today
  - Due Wednesday (11/1) @ 11:00pm
  - Remember to check that the autograder passes! Helps make sure you turned in the right files, pass the linter, etc.

- Can resubmit as many times as you'd like until the deadline.
  - Use the autograder as a tool if you're not sure if your code/tests have bugs

Abstraction Barrier – Review



- Specifications acts as the “barrier” between each side
  – improves understandability, changeability, and modularity
- Clients can only depend on the spec
- Implementer can write any code that satisfies the spec

Specifications for ADTs – Review

- New Terminology for specifying ADTs:
  - **Concrete State / Representation (Code)**
    - Actual fields of the record and the data stored
    - Ex: `{ list: List, last: number | undefined }`
  - **Abstract State / Representation (Math)**
    - How clients should understand the object
    - Ex: List (nil or cons)

Specifications for ADTs – Review

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
export interface FastList {
  …
  /**
   * Returns the object as a regular list of items.
   * @returns obj
   */
 toList: () => List<number>;
```

obj is the abstract state

- Talk about functions in terms of the abstract state
- Hide the representation details (i.e. real fields) from the client

Documenting ADTs – Review

Abstract Function (AF) – defines what abstract state the field values represent

- Maps field values → the object they represent
- Output is math, this is a mathematical function

Representation Invariants (RI) – facts about the field values that must always be true

- Constructor must always make sure RI is true at runtime
- Can assume RI is true when reasoning about methods
- AF only needs to make sense when RI holds
- Must ensure that RI *always* holds

Documenting ADTs – Review

```
class FastLastList implements FastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  …
  // @returns last(obj)
  getLast = (): number | undefined => {
    return this.last;
  };
}
```

Prove correctness of $\text{last}(\text{obj}) = \text{this.last}$ using both

$$\begin{aligned} \text{Last}(\text{obj}) &= \text{last}(\text{this.list}) && \textbf{by AF} \\ &= \text{this.last} && \textbf{by RI} \end{aligned}$$

Defining Interfaces

Typescript

```
interface FastList {
  getLast: () => number|undefined;
  toList: () => List<number>;
}
```

=

Java

```
interface FastList {
  int getLast() throws EmptyList;
  List<Integer> toList();
}
```

Readonly – Typescript

- The prefix **`readonly`** is used to make a property read-only
  - Value cannot be changed
  - Protects variables from unwanted mutations
  - Should be our default

Ex:

```
class FastLastListImpl implements FastList {
        readonly last: number | undefined;
        readonly list: List<number>;
}
```

Abstract Data Class – Example

```
class FastLastListImpl implements FastList {
        readonly last: number | undefined;
        readonly list: List<number>;

        constructor(list: List<number>) {
                this.last = last(list);
                this.list = list;
        }

        getLast = () => { return this.last; }
        toList = () => { return this.list; }
}
```

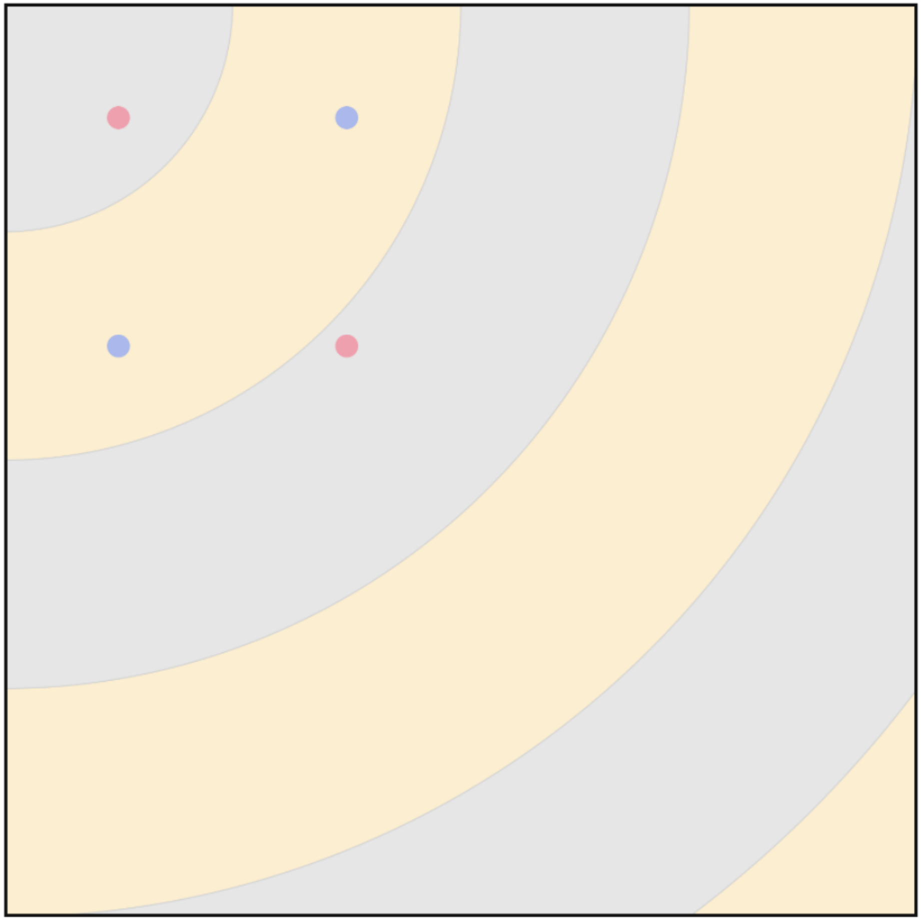
```
interface FastList {
  getLast: () => number|undefined;
  toList: () => List<number>;
}
```

Can create new record using "new":

```
new FastLastListImpl(list);
```

Question 1 & 2 – Coding

Run **`npm run start`** in `sec-highlight` to check it out!



input the points:

```
100 100
100 300
300 100
300 300
```

Questions 1 & 2 – Recap

- From concrete implementation → ADT, writing specs shouldn't be too hard
  - the specs already exist
  - just need to adjust what objects they're operating on: parameters → '`obj`'
  - and add appropriate AF and RI

- Only did 1 in this example, but we're able to have multiple classes implement the same interface, all with the same spec
  - Implementation can be switched out as needed, but expected inputs and behavior (spec) will be consistent

Question 4

Prove by structural induction that, for any left-leaning tree T, we have: $\boxed{\text{size}(T) \leq 2^{\text{height}(T)+1} - 1}$

Hints:

1) Define the tree in your IH according to the definition of tree `node(x, S, T)` so you can access the left and right trees
2) Remember the exponent rule: $x^y \times x = x^{y+1}$

$$
\begin{aligned}
&\textbf{func}\ \text{size}(\text{empty}) &&:= 0 \\
&\quad\ \text{size}(\text{node}(x, S, T)) &&:= 1 + \text{size}(S) + \text{size}(T) \\
&\textbf{func}\ \text{height}(\text{empty}) &&:= -1 \\
&\quad\ \text{height}(\text{node}(x, S, T)) &&:= 1 + \text{height}(S) \qquad \text{for any } x : \mathbb{Z} \text{ and } S, T : \text{Tree}
\end{aligned}
$$

Question 3 – Preface

sep takes a list L and a value x, and returns two lists, A containing all values $\leq x$ and B containing all values $> x$.

$$
\begin{aligned}
\textbf{func }\ \mathsf{sep}(\mathsf{nil}, x) &:= (\mathsf{nil}, \mathsf{nil}) \\
\mathsf{sep}(\mathsf{cons}(y, L), x) &:= (\mathsf{cons}(y, A), B) && \text{if } y \leq x \\
\mathsf{sep}(\mathsf{cons}(y, L), x) &:= (A, \mathsf{cons}(y, B)) && \text{if } x < y \\
&\quad\ \text{where } (A, B) := \mathsf{sep}(L, x)
\end{aligned}
$$

Note: in the recursive case, you:

- make a call to $\mathbf{sep}(\boldsymbol{L}, \boldsymbol{x})$
- take the return value of that call $(\boldsymbol{A}, \boldsymbol{B})$
- $\mathbf{cons}(\boldsymbol{y}$ on to A or B and returns $(A, \mathrm{cons}(y, B))$ or $(cons(y, A), B)$

- Making an additional step to make our recursive result cleaner and avoid multiple recursive calls