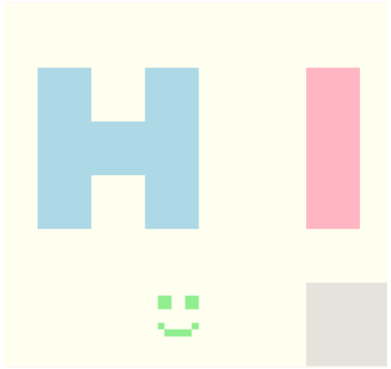
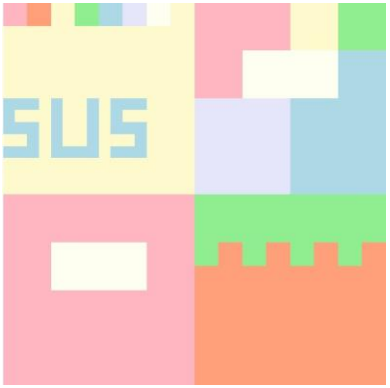
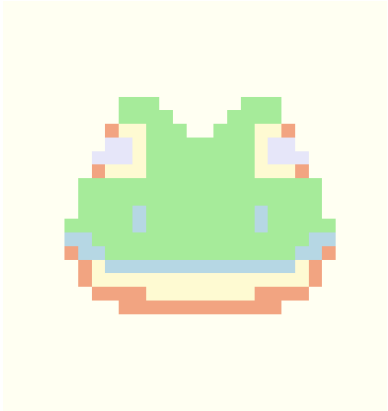
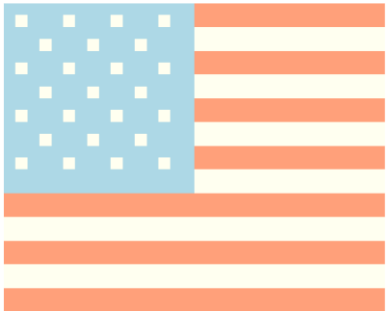
CSE 331
Spring 2025

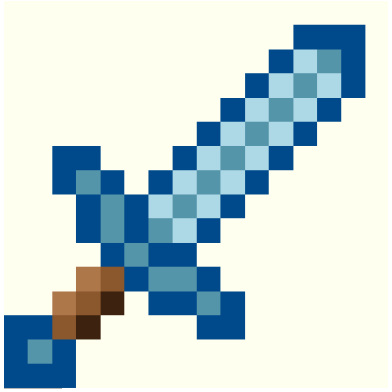
Mutable ADTs











Matt Wang

& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong

Administrivia (05/30)

- **HW9 is out!**
  - **due Friday, June 6th! (2 extra days!!)**
    thus, late due date is Saturday, June 7th
    we are not shifting office hour times – please plan around this!
    advice: still try to finish on Wed, use Thu section for exam prep
  - **"capstone" programming & math homework**
    builds significantly off of section & HW8
    intentionally *no* array proofs (see Matt's guarantee on Wed)
- **Bob Bandes Nominations are out!**
  - **nominate your GOAT TAs :)**
  - **fun fact: 331 TA was a winner last year!**

Final Exam: Logistics

- Tuesday June 10th from 12:30 – 2:20 in KNE 130
- paper exam, closed book & closed notes
  - we will provide you with a small reference sheet*
  - you should bring your Husky card + writing tool
- focuses of the exam
  - proving correctness of code
  - implement (small) TS functions according to a spec
  - writing tests for code (using testing heuristics)
  - broader conceptual questions on all course topics (incl. debugging, client–server programming, OOP…)
- must let me know by Tuesday if there is a conflict

Final Exam: Studying

- **we intend for the exam to:**
  - **mirror the homework as much as possible***

    exception: some broader conceptual questions on coding material & topic 10
  - **thus, should not require significant explicit studying***

    two key differences: **closed book/notes** and **timed**
- **Matt's advice™**
  - **do HW9! this will help you for the exam!**
  - **start by making sure you're solid on course content**

    not just reviewing slides/lectures – can you do the section problems & HW?

    take notes on what's hard. Get help on this!
  - **once you're solid, *then* try a practice exam**

    simulate the exam environment (e.g. closed book & notes, time yourself)

    after you do the exam – go back to the previous step!

Final Exam: Other Resources

- existing 23sp practice midterm & final on website
- next week's quiz section is exam review
  - don't skip to work on HW9!
- also dropping next week:
  - another practice final & solutions (written by Matt)
  - some async videos going through final solutions
- finals week:
  - no office hours
  - TA-led exam review session (likely Monday)
  - still will be active on Ed!

Specifying & Using Mutable ADTs

Recall: Immutable Map ADT

- An "association list" also called a "map"

observer

observer

producer

```
// List of (key, value) pairs
interface Map<K, V> {

  // @returns contains-key(x, obj)
  containsKey(x: K): boolean;

  // @requires contains-key(x, obj)
  // @returns get-value(x, obj)
  getValue(x: K): V;

  // @returns set-value(x, v, obj)
  setValue(x: K, v: V): Map<K, V>;
}
```

Using the Immutable Map ADT

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  const M1 = M.setValue("one", 2);
  const r = M1.getValue("one");
  return r;
};
```

- Let's check that this code is correct...

Proving Correct Use of Immutable Map (1/4)

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  const M1 = M.setValue("one", 2);
  const r = M1.getValue("one");
  {{ Post: r > 0 }}
  return r;
};
```

```
// @requires contains-key(x, obj)
// @returns get-value(x, obj)
getValue(x: K): V;
```

Proving Correct Use of Immutable Map (2/4)

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  const M1 = M.setValue("one", 2);
  {{ get-value("one", M1) > 0 }}
  const r = M1.getValue("one");
  {{ Post: r > 0 }}
  return r;
};
```

```
// @returns set-value(x, v, obj)
setValue(x: K, v: V): Map<K, V>;
```

Proving Correct Use of Immutable Map (3/4)

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  {{ get-value("one", set-value("one", 2, M)) > 0 }}
  const M1 = M.setValue("one", 2);
  {{ get-value("one", M1) > 0 }}
  const r = M1.getValue("one");
  {{ Post: r > 0 }}
  return r;
};
```

Proving Correct Use of Immutable Map (4/4)

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  {{ get-value("one", set-value("one", 2, M)) > 0 }}
  const M1 = M.setValue("one", 2);
  {{ get-value("one", M1) > 0 }}
  const r = M1.getValue("one");
  {{ Post: r > 0 }}
  return r;
};
```

get-value("one", set-value("one", 2, M))
= get-value("one", ("one", 2) :: M) **def of** set-value
= 2 **def of** get-value
> 0

set-value(x, v, L) := (x, v) :: L
get-value(x, (y, v) :: M) := v if x = y
get-value(x, (y, v) :: M) := get-value(x, M) if x ≠ y

Recall: Mutable Map

- An "association list" also called a "map"

```
// List of (key, value) pairs
interface Map<K, V> {

  // @returns contains-key(x, obj)
  containsKey(x: K): boolean;

  // @requires contains-key(x, obj)
  // @returns get-value(x, obj)
  getValue(x: K): V;

  // @modifies obj
  // @effects obj = set-value(x, v, obj)
  setValue(x: K, v: V): void;
}
```

observer

observer

mutator

We still need the immutable math functions (e.g., set-value) to *define* a mutable ADT

Using the Mutable Map

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  M.setValue("one", 2);
  const r = M.getValue("one");
  return r;
};
```

- Let's check that this code is correct...
  - try this forward this time...

Proving Correct Use of Mutable Map (1/4)

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  M.setValue("one", 2);
  const r = M.getValue("one");
  return r;
};
```

```
// @modifies obj
// @effects obj = set-value(x, v, obj)
setValue(x: K, v: V): void;
```

Proving Correct Use of Mutable Map (2/4)

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  M.setValue("one", 2);
```

{{ M = set-value("one", 2, M_0) }}

```
  const r = M.getValue("one");
  return r;
};
```

Notice that two versions (M_0 vs M_1)
show up in the *reasoning*
even though our code has one version!

Proving Correct Use of Mutable Map (3/4)

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  M.setValue("one", 2);
  {{ M = set-value("one", 2, M0) }}
  const r = M.getValue("one");
  {{ M = set-value("one", 2, M0) and r = get-value("one", M) }}
  return r;
};
```

Proving Correct Use of Mutable Map (4/4)

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  M.setValue("one", 2);
  const r = M.getValue("one");
  {{ M = set-value("one", 2, M0) and r = get-value("one", M) }}
  {{ Post: r > 0 }}
  return r;
};
```

r = get-value("one", M)
  = get-value("one", set-value("one", 2, M_0)) **since** M = set-value("one", 2, M_0)
  = get-value("one", ("one", 2) :: M_0) **def of** set-value
  = 2 **def of** get-value
  > 0

set-value(x, v, L) := (x, v) :: L

get-value(x, (y, v) :: M) := v if x = y
get-value(x, (y, v) :: M) := get-value(x, M) if x ≠ y

Implementing Mutable ADTs

ADTs in Topic 7

- **Main place we have heap state is in an ADT**
- **Previously:**
  - **state was immutable**
  - **set in the constructor and then never changed**

    only need to confirm RI holds at the end of the constructor

    if RI holds there, then it holds forever
- **Now:**
  - **allow state to be changed by methods**

ADTs With Mutability

- **Main place we have heap state is in an ADT**
- **New Power:**
  - **allow state to be changed by methods**
- **New Responsibilities:**
  - **more complex specifications**

    add `@effects` and `@modifies`
  - **must check the RI holds after any method that mutates**

    often a good idea to write code to check this at runtime
  - **more responsibilities we will meet later…**

Recall: List ADT with a Fast `getLast`

```
// Represents an (immutable) list of numbers.
interface FastList {

  // @returns x :: obj
  cons: (x: bigint) => FastList;

  // @returns last(obj)
  getLast: () => bigint|undefined;

  // @returns obj
  toList: () => List<bigint>;
};

const makeFastList = (): FastList => {
  return new FastListImpl(nil);
};
```

producer method

Mutable List ADT with a Fast `getLast`

```
// Represents a mutable list of numbers.
interface MutableFastList {

  // @modifies obj
  // @effects obj = x :: obj_0
  cons: (x: bigint) => void;
  …
```

mutator method

- **Method `cons` changes the list, putting `x` in front**
  - **now returns `void`**
  - **mutation explained in `@modifies` and `@effects`**

    abstract state is the old abstract state with x put in front

Recall: One Concrete Rep for `FastList`

```
class FastListImpl implements FastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  readonly last: bigint | undefined;
  readonly list: List<bigint>;

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

- **We can use the same rep for a mutable version**

Mutable List ADT Implementation

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: bigint | undefined;
  list: List<bigint>;

  // @modifies obj
  // @effects obj = x :: obj_0
  cons = (x: bigint): void => {
    this.list = cons(x, this.list);
  };
```

- Let's check correctness...

Proving `MutableFastListImpl` Correct (1/9)

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: bigint | undefined;
  list: List<bigint>;

  // @modifies obj
  // @effects obj = x :: obj_0
  cons = (x: bigint): void => {
    this.list = cons(x, this.list);
    {{ this.list = x :: this.list_0 }}
    {{ Post: obj = x :: obj_0 }}
  };
```

Proving `MutableFastListImpl` Correct (2/9)

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: bigint | undefined;
  list: List<bigint>;

  // @modifies obj
  // @effects obj = x :: obj_0
  cons = (x: bigint): void => {
    this.list = cons(x, this.list);
    {{ this.list = x :: this.list_0 }}
    {{ Post: obj = x :: obj_0 }}
  };
```

What is missing?

Also, need the RI to hold!

obj = this.list — by AF

= x :: this.list$_0$ — since this.list = cons(x, this.list$_0$)

= x :: obj$_0$ — by AF

Proving `MutableFastListImpl` Correct (3/9)

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: bigint | undefined;
  list: List<bigint>;

  // @modifies obj
  // @effects obj = x :: obj_0
  cons = (x: bigint): void => {
    this.list = cons(x, this.list);
    {{ this.list = x :: this.list_0 }}
    {{ Post: obj = x :: obj_0 and
             this.last = last(this.list) }}
  };
```

Also, need the RI to hold!

Does it? No!

- **Postcondition is `@returns`, `@effects`, and RI**

Proving `MutableFastListImpl` Correct (4/9)

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: bigint | undefined;
  list: List<bigint>;

  // @modifies obj
  // @effects obj = x :: obj_0
  cons = (x: bigint): void => {
    this.list = cons(x, this.list);
    this.last = last(this.list);
    {{ this.list = x :: this.list_0 and this.last = last(this.list) }}
    {{ Post: obj = x :: obj_0 and this.last = last(this.list) }}
  };
```

Rep Invariant now holds

Proving `MutableFastListImpl` Correct (5/9)

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: bigint | undefined;
  list: List<bigint>;

  // @modifies obj
  // @effects obj = x :: obj_0
  cons = (x: bigint): void => {
    this.last = last(this.list);
    {{ this.last = last(this.list) }}
    this.list = cons(x, this.list);
    {{ this.list = x :: this.list_0 and this.last = last(this.list_0) }}
    {{ Post: obj = x :: obj_0 and this.last = last(this.list) }}
  };
```

Rep Invariant would not hold if we switched the order

Proving `MutableFastListImpl` Correct (6/9)

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: bigint | undefined;
  list: List<bigint>;

  // @modifies obj
  // @effects obj = x :: obj_0
  cons = (x: bigint): void => {
    this.list = cons(x, this.list);
    this.last = last(this.list);
    {{ this.list = x :: this.list_0 and this.last = last(this.list) }}
    {{ Post: obj = x :: obj_0 and this.last = last(this.list) }}
  };
```

This version is obviously correct, but O(n).

Can we do it faster?

Proving `MutableFastListImpl` Correct (7/9)

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: bigint | undefined;
  list: List<bigint>;

  // @modifies obj
  // @effects obj = x :: obj_0
  cons = (x: bigint): void => {
    if (this.list === nil)
      this.last = x;
    this.list = cons(x, this.list);
    {{ ____________________________________________ }}
    {{ Post: obj = x :: obj_0 and this.last = last(this.list) }}
  };
```

O(1) version, but more complex reasoning (two branches)

Proving `MutableFastListImpl` Correct (8/9)

```
class MutableFastListImpl implements MutableFastList {

  cons = (x: bigint): void => {
    if (this.list === nil)
      this.last = x;
    this.list = cons(x, this.list);
    {{ this.list = x :: this.list_0 and this.list_0 = nil and this.last = x }}
    {{ Post: obj = x :: obj_0 and this.last = last(this.list) }}
  };
```

Case "then":

last(this.list)	= last(x :: this.list$_0$)	**since** this.list = x :: this.list$_0$
	= last(x :: nil)	**since** this.list$_0$ = nil
	= x	**def of** last
	= this.last	**since** x = this.last

last(x :: nil) := x

last(x :: y :: L) := last(y :: L)

Proving `MutableFastListImpl` Correct (9/9)

```
class MutableFastListImpl implements MutableFastList {

  cons = (x: bigint): void => {
    if (this.list === nil)
      this.last = x;
    this.list = cons(x, this.list);
    {{ this.list = x :: this.list0 and this.list0 ≠ nil and
       this.last = this.last0 and this.last0 = last(this.last0) }}
    {{ Post: obj = x :: obj0 and this.last = last(this.list) }}
```

from the RI
(will need this)

Case "else":

last(this.list)	= last(x :: this.list$_0$)	
	= last(this.list$_0$)	**since** this.list = x :: this.list$_0$
	= this.last$_0$	**since** this.list$_0$ ≠ nil
	= this.last	**since** this.last$_0$ = last(this.list$_0$)
		since this.last = this.last$_0$

last(x :: nil) := x

last(x :: y :: L) := last(y :: L)

Adding `getFirst` to `MutableFastList`

```
// Represents a mutable list of numbers.
interface MutableFastList {

  // @modifies obj
  // @effects obj = x :: obj_0
  cons: (x: bigint) => void;

  // @returns first(obj), where
  //     first(nil)    := 0
  //     first(x :: L) := x
  getFirst: () => bigint|undefined;

  // @returns last(obj), where …
  getLast: () => bigint|undefined;

};
```

More Complex Uses of Mutable Map: `extend`

```
// @requires L /= nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//    where m = first(L)
const extend = (L: MutableFastList, k: bigint,
                R: MutableFastList): void => {
  let i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    const m = L.getFirst();
    R.cons(m + i);
    i++;
  }
};
```

Proving `extend` Correct (1/5)

```
// @requires L /= nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//     where m = first(L)
const extend = (L: MutableFastList, k: bigint,
                R: MutableFastList): void => {
  let i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 }}
    const m = L.getFirst();
    R.cons(m + i);
    i++;
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 }}
  }
```

Proving `extend` Correct (2/5)

```
// @requires L /= nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//     where m = first(L)
const extend = (L: MutableFastList, k: bigint,
                R: MutableFastList): void => {
  let i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 }}
    const m = L.getFirst();
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 and m = first(L) }}
    R.cons(m + i);
    i++;
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 }}
  }
```

Proving `extend` Correct (3/5)

```
// @requires L /= nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//    where m = first(L)
const extend = (L: MutableFastList, k: bigint,
                R: MutableFastList): void => {
  let i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    const m = L.getFirst();
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 and m = first(L) }}
    R.cons(m + i);
    {{ R = (m+i) :: R1 and R1 = (m+i-1) :: ... :: (m+1) :: R0 and m = first(L) }}
    i++;
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 }}
  }
```

Proving `extend` Correct (4/5)

```
// @requires L /= nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//     where m = first(L)
const extend = (L: MutableFastList, k: bigint,
                R: MutableFastList): void => {
  let i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    const m = L.getFirst();
    R.cons(m + i);
    {{ R = (m+i) :: R_1 and R_1 = (m+i-1) :: ... :: (m+1) :: R_0 and m = first(L) }}
    {{ R = (m+i) :: ... :: (m+1) :: R_0 }}
    i++;
    {{ R = (m+i-1) :: ... :: (m+1) :: R_0 }}
  }
```

Proving `extend` Correct (5/5)

```
// @requires L /= nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//     where m = first(L)
const extend = (L: MutableFastList, k: bigint,
                R: MutableFastList): void => {
  let i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    const m = L.getFirst();
    R.cons(m + i);
    {{ R = (m+i) :: R_1 and R_1 = (m+i-1) :: ... :: (m+1) :: R_0 and m = first(L) }}
    {{ R = (m+i) :: ... :: (m+1) :: R_0 }}
    i++;
  }
};
```

$R = (m+i) :: R_1$

$= (m+i) :: (m+i-1) :: \ldots :: (m+1) :: R_0$ since $R_1 = \ldots$

WWDKS? (What Would Don Knuth Say?)

```
const extend = (L: MutableFastList, k: bigint,
                R: MutableFastList): void
```

- We have proven this code correct, but...

"Beware of bugs in the above code;
I have only proved it correct, not tried it."

Donald Knuth, 1977

- We should also try it...

Trying `extend` (1/2)

```
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//     where m = first(L)
const extend = (L: MutableFastList, k: bigint,
                R: MutableFastList): void => {
```

- **Try out the code:**

```
… // L = 2 :: 1
… // R = 2 :: 1
extend(L, 3, R)
console.log(R);
```

- **What list should this print?**

5 :: 4 :: 3 :: 2 :: 1 :: nil

Trying `extend` (2/2)

```
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//     where m = first(L)
const extend = (L: MutableFastList, k: bigint,
                R: MutableFastList): void => {
```

- **Try out the code:**

```
… // L = 2 :: 1
… // R = 2 :: 1
extend(L, 3, R)
console.log(R);
```

- **Instead, it prints** $8 :: 5 :: 3 :: 2 :: 1 :: \text{nil}$ **! How?!?**

L and R are aliases to the same MutableFastList

Reasoning with Aliases

- Aliasing **breaks** reasoning!
  - there was nothing wrong with our math
  - our math did not correctly describing the program

    modeling programs with aliasing is basically impossible

- Isn't this just a weird, special case?
  - just double check that $\mathrm{L} \neq \mathrm{R}$

- How about a more practical example?

Demo: New HW8 Features

Reasoning with (all-too-common) Aliasing!

- Aliasing breaks reasoning!
  - there was nothing wrong with our math
  - our math did not correctly model the program

    modeling programs with aliasing is basically impossible

- Aliasing is rampant in applications!
  - **no way** to easily check that there are no aliases
  - cannot reason about or debug individual functions

- Only option is to prevent unexpected aliasing…

Aliasing and Mutation Don't Mix

- Aliasing breaks reasoning!
  - Root cause: mutating aliased data
  - Fix: allow mutation XOR aliasing (i.e., not both)
- Option 1: data is immutable
  - program can't tell if data is aliased
- Option 2: data is not aliased
  - local reasoning principles work great
  - new responsibility: no aliases of *my* data
- See also: Rust enforces this rule with type checker

Implementing *More* Mutable ADTs

Recall: Immutable Queue ADT

- A queue is a list that can *only* be changed two ways:
  - add elements to the front
  - remove elements from the back

```
// List that only supports adding to the front and
// removing from the end
interface NumberQueue {

  // @returns len(obj)
  size: () => bigint;

  // @returns [x] ++ obj
  enqueue: (x: bigint) => NumberQueue;

  // @requires len(obj) > 0
  // @returns (x, Q) with obj = Q ++ [x]
  dequeue: ()=> [bigint, NumberQueue];
}
```

observer

producer

producer

Mutable Queue ADT

- Mutable versions has mutators instead of producers

```
// Mutable array that only supports adding to the front
// and removing from the end.
interface MutableNumberQueue {

  // @returns obj
  elements(): bigint[];

  // @modifies obj
  // @effects obj = [x] ++ obj_0
  enqueue(x: bigint): void;

  // @requires len(obj) > 0
  // @modifies obj
  // @effects obj_0 = obj ++ [x]
  // @returns x
  dequeue(): bigint;
}
```

observer

mutator

mutator

Recall: Implementing a Queue with Two Lists

```
// Implements a queue using two lists.
class ListPairQueue implements NumberQueue {

  // AF: obj = this.front ++ rev(this.back)
  // RI: if this.back = nil, then this.front = nil
  readonly front: List;
  readonly back: List;

  // makes obj = concat(front, rev(back))
  constructor(front: List, back: List) {
    …
  }
```

- Queue was in two parts, front and back
  - back stored in reverse order
  - full list was this.front ⧺ rev(this.back)

Implementing Mutable Queue with Two Arrays

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  front: bigint[];
  back: bigint[];

  // makes obj = vals
  constructor(vals: bigint[]) {
    this.front = [];
    this.back = vals;
  }
```

We should check this...

Proving `ArrayPairQueue` Correct (1/2)

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  front: bigint[];
  back: bigint[];

  // makes obj = vals
  constructor(vals: bigint[]) {
    this.front = [];
    this.back = vals;
    {{ this.front = [] and this.back = vals }}
    {{ Post: obj = vals }}
  }
```

Proving `ArrayPairQueue` Correct (2/2)

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  front: bigint[];
  back: bigint[];

  // makes obj = vals
  constructor(vals: bigint[]) {
    this.front = [];
    this.back = vals;
    {{ this.front = [] and this.back = vals }}
    {{ Post: obj = vals }}
  }
```

Is this really correct?

No way to say!

obj = rev(this.front) ++ this.back — **by AF**

= rev([]) ++ this.back — **since** this.front = []

= [] ++ this.back — **def of** rev

= this.back = vals — **since** this.back = vals

Defensive Programming: Copy-on-Read

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  front: bigint[];
  back: bigint[];

  // makes obj = vals
  constructor(vals: bigint[]) {
    this.front = [];
    this.back = vals.slice(0, vals.length);
  }
```

- **Must make a copy of the array!**
  - then, we have the only reference to it (no aliases)

Faster `elements` (in Mutable Queue)

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  front: bigint[];
  back: bigint[];

  // @returns obj
  elements = (): bigint[] => {
    let revFront: bigint[] =
      this.front.slice(0, this.front.length);
    revFront.reverse();
    return revFront.concat(this.back);
  };
```

This is O(n)...

We can optimize it if front = [].

$$\text{rev}([]) \# \text{this.back} = [] \# \text{this.back} = \text{this.back}$$

`elements` Reasoning

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  front: bigint[];
  back: bigint[];

  // @returns obj
  elements = (): bigint[] => {
    if (this.front.length === 0) {
      return this.back;   // O(1) when this.front = []
    } else {
      let revFront: bigint[] =
        this.front.slice(0, this.front.length);
      revFront.reverse();
      return revFront.concat(this.back);
    }
  };
```

Is this correct?

No way to say!

Defensive Programming: Copy-on-Write

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  front: bigint[];
  back: bigint[];

  // @returns obj
  elements = (): bigint[] => {
    let revFront: bigint[] = this.front.slice(0);
    revFront.reverse();
    return revFront.concat(this.back);
  };
```

- **Cannot return an alias to `this.back`**
  - **must make a copy in all cases**

Moral of the Story for Mutable Heap State

- More mutation gave us better efficiency
  - saved memory
  - immutable version could be just as fast
- More mutation means more complex reasoning
  - more facts to keep track of
  - more ways to make mistakes
  - more work to make sure we did it right
- New possibilities for exciting bugs!
  - must avoid aliasing of anything mutable

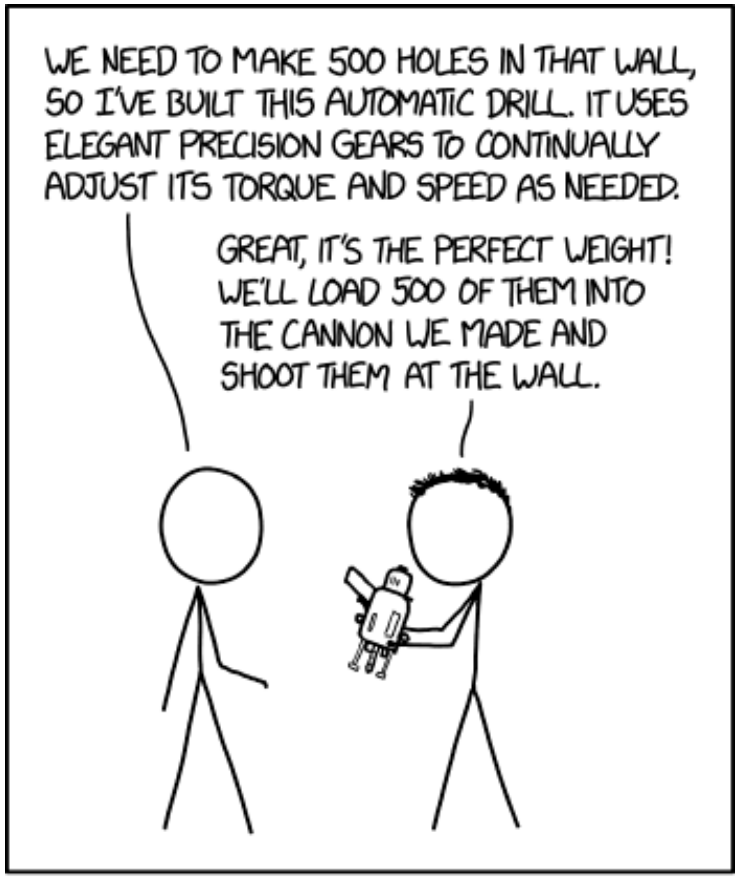
    we call this "**representation exposure**"

Need for Mutable Heap State

- Saw that mutable heap state is complex
  - better to avoid when possible
- Cannot be avoided in some cases
  1. server-side data storage
  2. client-side UI
- In both cases, we try to constrain its use
  - including coding conventions to keep ourselves sane

CSE 331
Spring 2025

Subtyping



xkcd #2021

Matt Wang

& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong

Administrivia (06/02)

- fixed typo in sample final solution, Task 4b proof
- reminder: Bob Bandes Nominations are out!

Object-Oriented Programming

- We haven't done any OO this quarter
  - this week, we will see some reasons why!

- Plan for this week:
  - focus on topics that are good to know but not needed for HW

    usually, mistakes you want to avoid
  - every lecture will include one related to OO

Subtypes

Subtypes of Concrete Types (in math)

- We initially defined types as sets

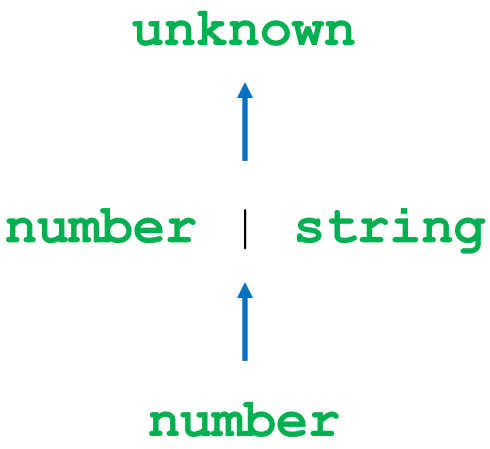
- In math, a **subtype** can be thought of as a **subset**
  - e.g., the even integers are a subtype of $\mathbb{Z}$
  - e.g., the numbers {1, 2, 3, 4, 5, 6} are a subtype of $\mathbb{Z}$
  - likewise, a superset would be a **supertype**

- Any even integer "is an" integer
  - "is a" is often (but not always) good intuition for subtypes

Subtypes of Concrete Types (in TypeScript)

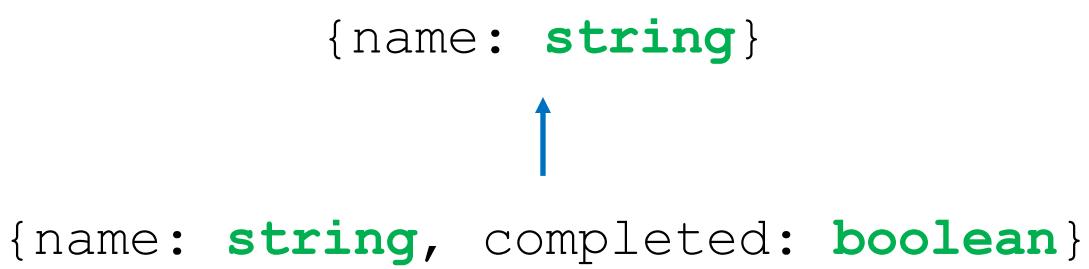
- We initially defined types as sets

- In TypeScript, some subtypes are also subsets
  - `number` has a set of allowed values
  - it is a subtype of types that allow those values + more



Subtypes of Concrete Types (for records)

- We initially defined types as sets
- In TypeScript, some subtypes are also subsets
  - record types require certain fields but allow more
  - record type with a superset of the fields is a subtype



Subtyping Used by TypeScript: Parameters

- TypeScript uses subtyping in function calls

```
const f = (s: number | string): number => { … };

const x: number = 3;
… f(x) …
```

  - types are not the same (`number` vs `number | string`)
  - subtype can be passed where super-type is expected

    any element of the subtype "is an" element of the super-type

- Similar rules in Java

Subtyping Used by TypeScript: Returns

- TypeScript uses subtyping in function calls

```
const g = (n: number): number => { … };

const x: number | string = g(3);
```

  - types are not the same (`number` vs `number | string`)
  - subtype can be <u>returned</u> where super-type is expected

    any element of the subtype "is an" element of the super-type

- Similar rules in Java

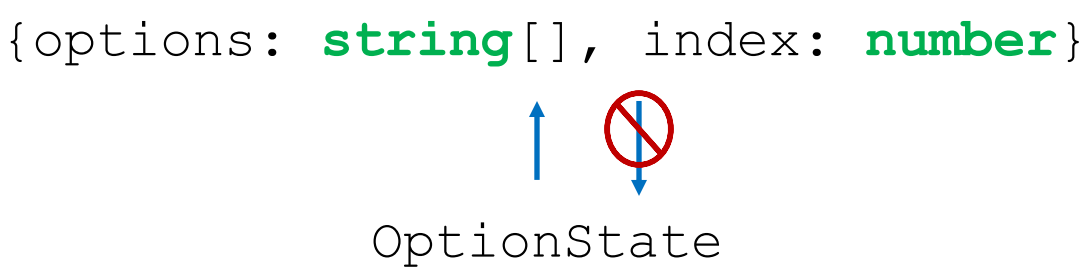
Subtyping Used by TypeScript: Invariants (1/2)

- TypeScript only sees the declared types
  - any other behavior is left to reasoning

- Example: invariants

```
// RI: 0 <= index < options.length
type OptionState = {
  options: string[],
  index: number
}
```

Subtyping Used by TypeScript: Invariants (2/2)



- `OptionState` **is a subtype of the bare record type**
  - **it is a record with those fields**
  - **but reverse is not true**

- **TypeScript will see these as the same**
  - **will let you pass the top where the bottom is expected**

    up to us to make sure this doesn't happen

Subtypes of Abstract Types

- Recall: ADTs are collections of functions
  - hide the concrete representation
  - pass functions that operate on the data

    create, observe, mutate

- "Subtypes are subsets" does not work well here
  - set of all possible functions with … yuck

- Would be nice to find a cleaner approach

Subtypes Are Substitutable

- If B is a subtype of A, can send B where A is expected:

```
const f = (s: A): void => { … }
const g = (): B => { … }

const x: B = …;
f(x);                    // okay

const y: A = g();   // okay
```

  - okay to “substitute” a B where an A is expected

Liskov Substitution Principle

- Subtypes are **substitutable** for supertype
  - this is the "Liskov substitution principle"
  - due to Barbara Liskov

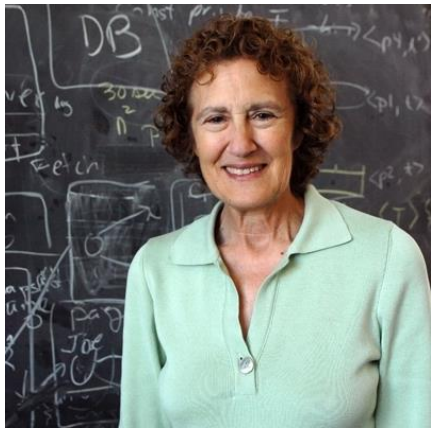


photo courtesy MIT

- For ADTs, we use this as our definition of subtypes
  - (for concrete types, subsets are usually easier)

Liskov's Linked Lists

Def: If B is a subtype of A, can send B where A is expected. In Java, `String` **is a subtype of** `Object`**.**

What is the subtyping relationship between `List<String>` **and** `List<Object>`**?**

Hint: consider these methods:

sli.do #cse331

```
void foo(List<Object> a) {
  a.add(new Object());
}
```

```
String foo(List<String> b) {
  return b.get(0);
}
```

1. `List<String>` is a subtype of `List<Object>`
2. `List<Object>` is a subtype of `List<String>`
3. **both are subtypes of each other**
4. **neither are subtypes of each other**

Defining Substitutable Abstract Types

- When is ADT B substitutable for A?

- Must satisfy two conditions:

1. B must provide all the methods of A

   If A has a method “f”, then B must have a method called “f”

2. B’s corresponding method must…

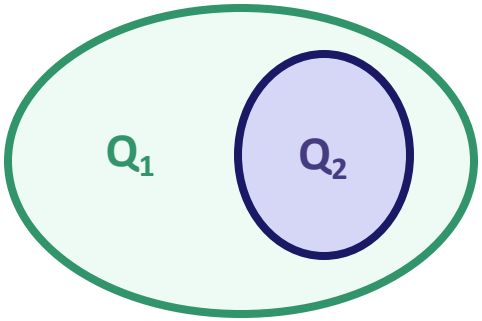
   must accept all the inputs that A’s does

   must also promise everything in A’s postcondition

   I.e., B must have the same or a "**stronger**" spec

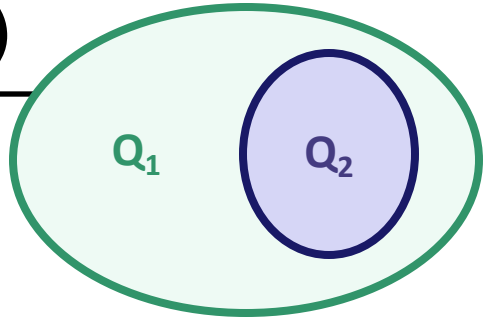
Review: Stronger Assertions vs Specifications

- **Assertion** is stronger iff it holds in a subset of states



- Stronger assertion implies the weaker one
  - stronger is a synonym for "implies"
  - weaker is a synonym for "is implied by"

Strengthening a Specification (1/3)

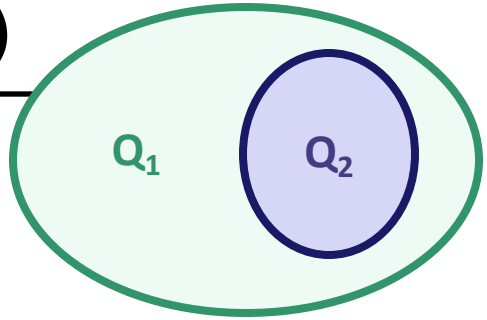



```
interface A {
  f: (x: number) => number

  // @requires x >= 0
  g: (x: number) => number
}
```

- Stronger specs promise more (or same) outputs
  - more specific return type (or thrown type)

```
interface D extends A {
  f: (x: number) => 0 | 1 | 2 | 3
}
```

Strengthening a Specification (2/3)



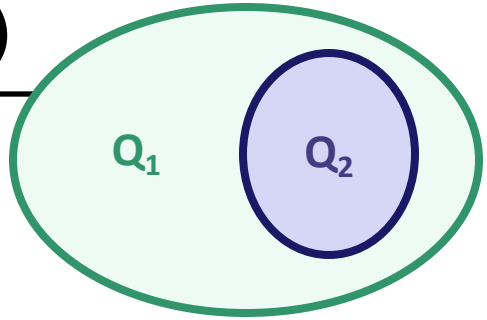

```
interface A {
  f: (x: number) => number

  // @requires x >= 0
  g: (x: number) => number
}
```

- Stronger specs promise more (or same) outputs
  - more specific return type (or thrown type)
  - more facts included in @returns and @effects

```
interface E extends A {
  // @requires x >= 0
  // @returns an even integer
  g: (x: number) => number
}
```

  - fewer objects listed in @modifies

Strengthening a Specification (3/3)




```
interface A {
  f: (x: number) => number

  // @requires x >= 0
  g: (x: number) => number
}
```

- Stronger specs allow more (or same) inputs
  - allowed argument types are supersets

```
interface B extends A {
  f: (x: number | string) => number
}
```

  - fewer requirements on arguments

```
interface C extends A {
  g: (x: number) => number    // x can be negative
}
```

Example: Rectangle and Square

- Is Square a subtype of Rectangle?
  - math intuition says yes
  - a square “is a” rectangle

- Let’s check this with substitutability…

Example: Immutable Rectangle and Square

```
interface Rectangle {
  getWidth: () => number,
  getHeight: () => number
}

// A rectangle with width = height
interface Square extends Rectangle {
  getSideLength: () => number
}
```

extra invariant
on abstract state
(an "abstract invariant")

Yes

- **Is `Square` substitutable for `Rectangle`?**
  - **allows the same inputs (none)**
  - **makes the same promises about outputs (numbers)**
  - **adds another promise: both methods return same number**

Example: Mutable Rectangle and Square (1/2)

```
interface Rectangle {
  getWidth: () => number,
  getHeight: () => number
  resize: (width: number, height: number) => void
}

// A rectangle with width = height
interface Square extends Rectangle {
  // @requires width = height
  resize: (width: number, height: number) => void
}
```

- **Is `Square` substitutable for `Rectangle`?** No!
  - **allows fewer inputs to resize!**

Example: Mutable Rectangle and Square (2/2)

- None of these work:

weaker spec

```
// @requires width = height
resize: (width: number, height: number) => void

// @throws Error if width != height
resize: (width: number, height: number) => void
```

incomparable specs

```
// Sets height = width also
resize: (width: number , height: number) => void
```

- Mutation sometimes makes subtyping impossible
  - yet another reason to avoid it

Subclasses & Subtyping

Review: Subclasses

- **Subclassing is a means of sharing code**
  - **subclass gets parent fields & methods (unless overridden)**

```
class Product {
  private String name;
  private int price;
  public String getName() {return name; }
  public int getPrice() { return price; }
}

class SaleProduct extends Product {
  private float discount;
  public int getPrice() {
    return (1 - discount) * super.getPrice();
  }
}
```

Subclasses are not always Subtypes

- **Subclassing does not guarantee subtyping relationship**

```
class Product {
  public int getPrice() { ... }

  // @returns true iff obj's price < p's price
  public boolean isCheaperThan(Product p) {
    return getPrice() < p.getPrice();
  }
}

class WackyProduct extends Product {
  // @returns some boolean value
  public boolean isCheaperThan(Product p) {
    return false;
  }
}
```

Legal Java, but not a subtype

Subclasses in Java (and other OOP languages)

- Java subclassing is a means of sharing code
  - subclass gets parent fields & methods (unless overridden)
- Does not guarantee subtyping
  - up to you to check that method specs are stronger
- Java treats it as a subtype
  - will let you pass subclasses where superclass is expected
- Subclassing is a surprisingly dangerous feature
  - that's not the only reason…

Subclasses & Coupling

- Subclassing is a surprisingly dangerous feature
- Subclassing tends to break modularity
  - creates tight coupling between super- and sub-class
  - often see the "fragile base class" problem

    changes to super class often break subclasses
- Let's see some Java examples…

Example 1: Tight Coupling

```
class Product {
  private int price;
  public int getPrice() { return price; }

  // @returns true iff obj's price < p's price
  public boolean isCheaperThan(Product p) {
    return getPrice() < p.getPrice();
  }
}

class SaleProduct extends Product {
  public int getPrice() {
    return (1 - discount) * super.getPrice();
  }
}
```

– looks okay so far...

Example 1: Tight Coupling Gone Wrong!

```
class Product {
  private int price;
  public int getPrice() { return price; }

  // @returns true iff obj's price < p's price
  public boolean isCheaperThan(Product p) {
    return this.price < p.price;
  }
}

class SaleProduct extends Product {
  public int getPrice() {
    return (1 - discount) * super.getPrice();
  }
}
```

Made it "faster" by eliminating a method call!

What's wrong?

Oops! Broke the subclass

Example 2: Tight Coupling

```
class InstrumentedHashSet extends HashSet<Integer> {
  private static int count = 0;

  public boolean add(Integer e) {
    count += 1;
    return super.add(e);
  }

  public boolean addAll(Collection<Integer> c) {
    count += c.size();
    return super.addAll(c);
  }

  public int getCount() { return count; }
}
```

– what could possibly go wrong?

Example 2: Tight Coupling Gone Wrong!

```
InstrumentedHashSet S = new InstrumentedHashSet();
System.out.println(S.getCount());  // 0
S.addAll(Arrays.asList(1, 2));
System.out.println(S.getCount());  // 4?!?
```

- what does this print?

- What is printed depends on `HashSet`'s `addAll`:
  - if it calls `add`, then this prints 4
  - if it does not call `add`, then this prints 2

- Also possible to be dependent on *order* of calls

Generalizing Examples 1 & 2

- Creates tight coupling between super- and sub-class
- Example 1: super-class needs to know about subclass
  - direct field access in parent breaks subclass
- Example 2: subclass needs to know about super-class
  - subclass dependent on which methods call each other
- But wait… There's more!

Example 3: Tight Coupling

```
class WorkList {
  // RI: len(names) = len(times) and total = sum(times)
  protected ArrayList<String> names;
  protected ArrayList<Integer> times;
  protected int total;

  public addWork(Job job) {
    addToLists(job.getName(), job.getTime());
    total += job.getTime();
  }

  protected addToLists(String name, int time) {
    names.add(name);
    times.add(time);
  }
}
```

Example 3: Tight Coupling … Okay So Far …

```
// Makes sure no task is too large compared to rest
class BalancedWorkList extends WorkList {
  protected addToLists(String name, int time) {
    if (times.size() <= 3 || 2*time < total)
      super.addToLists(name, time);  // okay
    } else {
      throw new ImbalancedWorkException(name, time);
    }
  }
}
```

- prevents item from being added if too big
- (also: this subclass is not a subtype!)

Example 3: Tight Coupling Gone Wrong!

```
class WorkList {
  // RI: len(names) = len(times) and total = sum(times)
  protected ArrayList<String> names;
  protected ArrayList<Integer> times;
  protected int total;

  public addWork(Job job) {
    int time = job.getTime();  // just one call
    total += time;
    addToLists(job.getName(), time);
  }
}
```

RI not true in method call

- reordering the updates breaks the subclass!
- subclass is using `total` that includes the new job

Generalizing Example 3

- RI can be false in calls to non-public methods
  - only needs to hold at end of the public method
- Requires extra care to get it right
  - method is tightly coupled with the ones that call it
  - needs to know what is true in those methods

    not enough to just know the RI
- Hard for multiple people to communicate this clearly
  - can be okay when it's all your code
  - very error prone when methods are written by others

Subclassing Creates Tight Coupling

- Creates tight coupling between super- and sub-class
  - direct field access can break subclass
  - subclass dependent on which methods call each other
  - subclass dependent on *order* of method calls
  - subclass can be called when RI is false
- Often see the "fragile base class" problem
- Subclassing is a surprisingly dangerous feature!
  - up to you to verify subclass method specs are stronger
  - up to you to prevent tight coupling

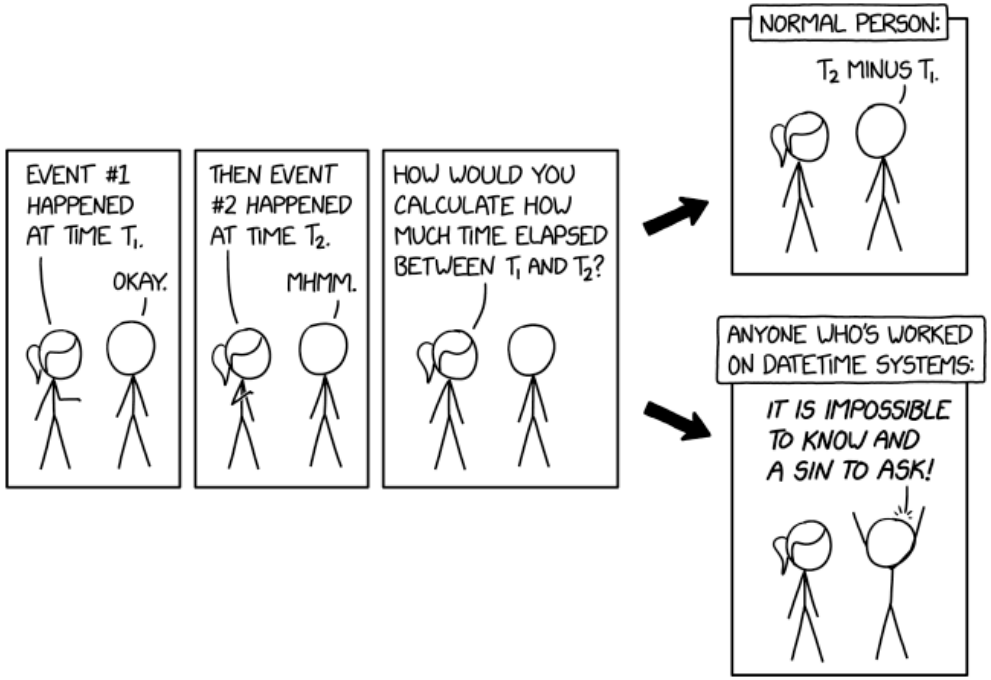
Subclassing is Best Avoided

- Java advice: either design for subclassing or prohibit it
  - from Josh Bloch, author of (much of) the Java libraries
- We haven't used subclassing in TypeScript
  - didn't even describe how to do it!

    we've just used classes as a quick way to create records
  - these problems are the main reason why we avoided it
- Subclassing is not necessary anyway
  - we have other ways to share code

CSE 331
Spring 2025

Equality & Constructors



xkcd #2867

Matt Wang

& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong

Administrivia (06/04)

- exam review session: Monday, June 9th, 5-6:30 PM
  - in CSE 3rd/4th/5th floor breakouts + one extra TBD CSE room
  - each location focused on a topic, otherwise office hours-style
  - completely optional
  - more details on Ed. Bring your questions!
- Matt is behind on extra sample final – sorry!
  - will make an Ed post when available
  - no video accompanying final, but will have solutions
- course evals are out!
  - please give us (actionable) feedback!
  - will dedicate ~ 10 min in lecture on Friday to do them

Equality

Equality of User-Defined Types

- For any type, useful to know which are “the same”
- TypeScript “===” is not useful on records:

```
{a: 1} === {a: 1}  // false!
```

  - as in Java, this is “reference equality”
  - tells you if they refer to the same object in memory

- `deepStrictEquals` would work here
  - checks that the records have the same fields and values
  - but that also is not perfect…

Recall: Queue With Two Lists

```
// Implements a queue using two lists.
class ListPairQueue implements NumberQueue {

  // AF: obj = this.front ++ rev(this.back)
  readonly front: List<number>;
  readonly back: List<number>;
```

– three ways of representing the same abstract state:

front	back	front ⧺ rev(back)
[1, 2]	[]	[1, 2]
[1]	[2]	[1, 2]
[]	[2, 1]	[1, 2]

– these should be considered equal!

Defining Equality Methods

- Often useful / necessary to define your own `equal`
  - check if references point to records that are "the same"

- Very important to get definitions correct
  - reasoning uses definitions, so
    if our definitions are wrong, our reasoning will be wrong
  - only tools for checking definitions: simplicity & testing

- Sometimes we can also sanity check them
  - saw this in Topic 8, e.g., $\text{get-value}(x, \text{set-value}(x, v, L)) = v$
  - can do something similar here...

Properties of Equality Functions

- **Often useful / necessary to define your own `equal`**
  - **check if references point to records that are "the same"**

- **Sensible definition should act like "=" in math:**

1. equal(a, a) = T for any a : A — reflexive
2. equal(a, b) = equal(b, a) for any a, b : A — symmetric
3. if equal(a, b) and equal(b, c), then equal(a, c) for any … — transitive

  - **(311 alert: this is an "equivalence relation")**
  - **Java has two more rules for `Object.equal` (see Java docs)**

Equality in Java

equals

```
public boolean equals(Object obj)
```

Indicates whether some other object is "equal to" this one.

The `equals` method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

An equivalence relation partitions the elements it operates on into *equivalence classes*; all the members of an equivalence class are equal to each other. Members of an equivalence class are substitutable for each other, at least for some purposes.

Example: Duration & Equality

- **Define** Duration **to be an amount of time in seconds**

$$\textbf{type}\ \text{Duration} = \{\text{min} : \mathbb{Z}, \text{sec} : \mathbb{Z}\}\ \text{with}\ 0 \leq \text{sec} < 60$$

  - **second part is a rep invariant**

- **Can define equality on** Duration **this way:**

$$\text{equal}(\{\text{min: m, sec: s}\}, \{\text{min: n, sec: t}\}) \ := \ (\text{m} = \text{n})\ \text{and}\ (\text{s} = \text{t})$$

  - **true iff these are the same amount of time**
    (wouldn't be true without the invariant)

Example: Duration & Checking Equality (1/2)

equal({min: m, sec: s}, {min: n, sec: t}) := (m = n) and (s = t)

- **Does this have the required properties?**
  - reflexive

equal({min: m, sec: s}, {min: m, sec: s})
= (m = m) and (s = s) **def of** equal
= T and T
= T

proof by calculation
that it holds for any record

  - symmetric

equal({min: m, sec: s}, {min: n, sec: t})
= (m = n) and (s = t) **def of** equal
= (n = m) and (t = s)
= equal({min: n, sec: t}, {min: m, sec: s}) **def of** equal

Example: Duration & Checking Equality (2/2)

$$\text{equal}(\{\text{min: m, sec: s}\}, \{\text{min: n, sec: t}\}) \;:=\; (m = n) \text{ and } (s = t)$$

- **Does this have the required properties?**
  - **reflexive** yes
  - **symmetric** yes
  - **transitive** also yes (but a little long for a slide)

- **Good evidence that this is a reasonable definition**

Non-Example: "==" in JavaScript

```
0 == "0"        true
0 == ""         true
0 == " "        true
```

- **Which property fails?**
  - **transitivity**: `"" != " "` (and `"0" != " "`)

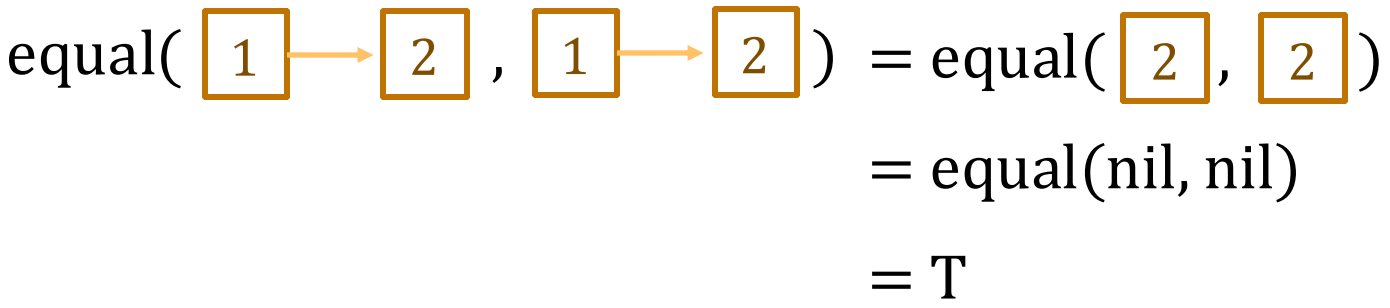
- **Good evidence that this is <u>not</u> a reasonable definition**

Example: List Equality (1/3)

- Can define equality on List type this way:

$$
\begin{array}{lcll}
\text{equal(nil, nil)} & := & \text{T} & \\
\text{equal(nil, b :: R)} & := & \text{F} & \\
\text{equal(a :: L, nil)} & := & \text{F} & \\
\text{equal(a :: L, b :: R)} & := & \text{F} & \text{if } a \neq b \\
\text{equal(a :: L, b :: R)} & := & \text{equal(L, R)} & \text{if } a = b
\end{array}
$$

- Checks that the values in the list are all the same
  - this is a definition, so we can only check it on examples…

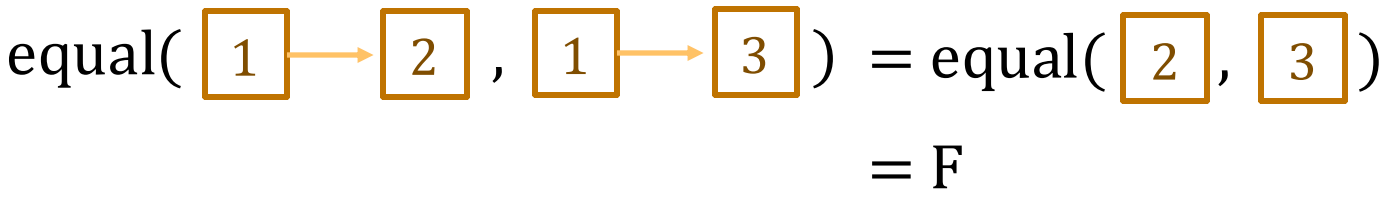


Example: List Equality (2/3)

- Can define equality on List type this way:

$$
\begin{array}{lcll}
\text{equal(nil, nil)} & := & \text{T} & \\
\text{equal(nil, b :: R)} & := & \text{F} & \\
\text{equal(a :: L, nil)} & := & \text{F} & \\
\text{equal(a :: L, b :: R)} & := & \text{F} & \text{if } a \neq b \\
\text{equal(a :: L, b :: R)} & := & \text{equal(L, R)} & \text{if } a = b
\end{array}
$$

- Checks that the values in the list are all the same
  - this is a definition, so we can only check it on examples…



Example: List Equality (3/3)

- Can define equality on List type this way:

$$
\begin{array}{lcll}
\mathrm{equal}(\mathrm{nil}, \mathrm{nil}) & := & \mathrm{T} & \\
\mathrm{equal}(\mathrm{nil}, \mathrm{b} :: \mathrm{R}) & := & \mathrm{F} & \\
\mathrm{equal}(\mathrm{a} :: \mathrm{L}, \mathrm{nil}) & := & \mathrm{F} & \\
\mathrm{equal}(\mathrm{a} :: \mathrm{L}, \mathrm{b} :: \mathrm{R}) & := & \mathrm{F} & \text{if } \mathrm{a} \neq \mathrm{b} \\
\mathrm{equal}(\mathrm{a} :: \mathrm{L}, \mathrm{b} :: \mathrm{R}) & := & \mathrm{equal}(\mathrm{L}, \mathrm{R}) & \text{if } \mathrm{a} = \mathrm{b}
\end{array}
$$

- Has all three required properties
  - how would we prove $\mathrm{equal}(\mathrm{L}, \mathrm{L})$ holds for any list L?

induction

Recall: Abstract Data Types (ADTs)

- **Abstraction over data**
  - **hide the details of the data representation**
  - **only give users a set of operations (the interface)**

    data abstraction via procedural abstraction

- **Can define Duration as an ADT instead…**
  - **hide the representation as two fields**

Example: Duration as an ADT

```
// Represents an amount of time measured in seconds
class Duration {

  // RI: 0 <= sec < 60
  // AF: obj = 60 * this.min + this.sec
  readonly min: number;
  readonly sec: number;

  equal = (d: Duration): boolean => {
    return this.min === d.min && this.sec === d.sec;
  };

  …
```

- **defines `Duration` as an ADT**
  - `getTime` method not shown
  - `equal` still makes sense, just as before

Recall: Subtypes vs Subclasses

- Subclasses are **code sharing**
  - everything from the parent is copied into the subclass
  - subclass can also replace (override) with its own versions

- Subtypes must be **substitutable** for supertype
  - this is the “Liskov substitution principle”
  - due to Barbra Liskov

- Not all subclasses are subtypes!
  - it's dangerous whenever that happens

Example: NanoDuration

- **Suppose a subclass also measures nanoseconds**

```
class NanoDuration extends Duration {

  // min: number (inherited)
  // sec: number (inherited)
  readonly nano: number;

  …
```

- **How should we define `equal`?**
  - **remember that it takes an argument of type `Duration`**

    we cannot accept fewer arguments

Example: NanoDuration & Equality

```
class NanoDuration extends Duration {

  // min: number (inherited)
  // sec: number (inherited)
  readonly nano: number;

  equal = (d: Duration): boolean => {
    if (d instanceof NanoDuration) {
      return this.min === d.min &&
             this.sec === d.sec &&
             this.nano === d.nano;
    } else {
      return false;
    }
  };
```

Must take `Duration` argument to be a subtype

symmetry

– which property does this lack?

Example: NanoDuration & Equality, Gone Wrong

```
const d = new Duration(2, 10);
const n = new NanoDuration(2, 10, 300);

console.log(n.equal(d));  // false
console.log(d.equal(n));  // true!
```

- `NanoDuration` **is only equal to other** `NanoDuration`**s**

- `Duration` **can be equal to a** `NanoDuration`
  if they have the same minutes and seconds

Example: NanoDuration & Equality, Round 2

```
class NanoDuration extends Duration {

  // min (inherited)
  // sec (inherited)
  readonly nano: number;

  equal = (d: Duration): boolean => {
    if (d instanceof NanoDuration) {
      return this.min === d.min &&
             this.sec === d.sec &&
             this.nano === d.nano;
    } else {
      return this.min == d.min && this.sec == d.sec;
    }
  };
```

No! It lacks transitivity

– fixes symmetry! all good now?

Example: NanoDuration & Equality, Still Wrong

```
const n1 = new NanoDuration(2, 10, 300);
const d = new Duration(2, 10);
const n2 = new NanoDuration(2, 10, 400);

console.log(n1.equal(d));   // true
console.log(d.equal(n2));   // true
console.log(n1.equal(n2));  // false!
```

– **transitivity requires** `n1` **to equal** `n2` **(but it doesn't)**

Subclasses and Equals Don't Always Mix

- **No good solution to this problem!**
  - **inherent tension between subtyping and equality**

    subtyping wants subclasses to behave the same

    equality wants to treat them differently (using extra information)

- **This is a general problem for "binary operations"**
  - **equality is just one example**

- **Real issue is that `NanoDuration` isn't a subtype…**
  - **would have seen this if we documented the ADT carefully**

NanoDuration *isn't* a Duration?

- Suppose a subclass also measures nanoseconds

```
// Represents an amount of time in nanoseconds
class NanoDuration extends Duration {

  // RI: 0 <= sec < 60 and 0 <= nano < 10000
  // AF: obj = 60,000,000 * this.min +
  //           1,000,000 * this.sec +
  //           this.nano
  readonly nano: number;

}
```

- Abstract states of the two types are different
  - time in seconds vs nanoseconds
  - abstract states of subtypes would need to be subtypes

Constructors

Public Constructors

- Most Java classes have public constructors
  - e.g., create an `ArrayList` with "`new ArrayList<String>()`"
- For our ADTs, we didn't do this
  - class was hidden (not exported)
  - we exported a "factory function" that used the constructor
    e.g., `makeSortedNumberSet`
  - this was not accidental…
- Constructors have undesirable properties
  - surprisingly error-prone
  - several important limitations

Method Calls from Constructors

- Any method call from a constructor is dangerous!

- Almost always calling with RI false
  - usually, the RI does not hold until all fields are assigned
    typically, that is the last line of the constructor
  - hence, any methods are called with the RI still false

- Asking for trouble!
  - method needs to know that some parts of RI may be false
  - eventually, someone changing code will mess this up
  - better to avoid method calls in the constructor

Limitations of Constructors

- Constructor is called *after* the object is created
  - can't decide, in the constructor, not to create it

- Limitations of constructors

  1. Cannot return an existing object

  2. Cannot return a different class

  3. Does not have a name!

Singleton

- Factory functions can return an existing object
- Common case: there is only one instance!
  - factory function can avoid creating new objects each time
  - called the "singleton" design pattern
- Example from before…

Example Singleton

```
interface FastList {
  cons(x: bigint): FastList;
  getLast(): bigint|undefined;
  toList(): List<bigint>;
};

const nilList: FastList = new FastBackList(nil);

const makeFastList = (): FastList => {
  return nilList;
};
```

Note: only allowed because `FastList` is **immutable**

- No need to create a new object using "`new`" *every time*
  - can reuse the same instance
  - example of the "singleton" **design pattern**

Returning a Subtype

- Factory functions can return a subtype
  - declared to return `A` but returns subtype `B` instead
  - allowed since every `B` is an `A`

- Example:

```
// @returns an empty NumberSet that can be used to
//     store numbers between min and max (inclusive)
const makeNumberSet = (min: number, max: number): NumberSet => {
  if (0 <= min && max <= 100) {
    return makeArrayNumberSet();  // only supports small sets
  } else {
    return makeSortedNumberSet(); // use a tree instead
  }
}
```

Recall: Multiple Constructors

- Java classes allow multiple constructors

```
class HashMap {
  public HashMap() { … }  // initial capacity of 16
  public HashMap(int initialCapacity) { … }
}
```

- TypeScript classes do not, but you can fake it with *optional* arguments

```
class HashMap {
  constructor(initialCapacity?: number) { ... }
}
```

Constructors Have No Name

- Do not get to name constructors
  - in Java, same name as the class
  - in TypeScript, called “constructor”

- Names are useful!

  1. Let you distinguish between different cases
     - use names to distinguish cases that otherwise look the same

  2. Let you explain what it does
     - the only thing you know the client will read!

Example: Distinguishing Constructors (1/3)

- JavaScript's Array has multiple constructors

```
new Array()              // creates []

new Array(a1, …, aN)     // creates [a1, …, aN]

new Array(2)             // creates [undefined, undefined]
```

  - what does "`new Array(a1)`" return when `a1` is a number?
  - how to make a 1-element array containing just `a1`

```
const A = new Array(1);
A[0] = a1;
```

  - don't have a name to distinguish these cases!

Example: Distinguishing Constructors (2/3)

- Factory functions have names
  - allow us to distinguish these cases

```
// @returns []
const makeEmptyArray = (): Array => { … };

// @returns A with A.length = len and
//     A[j] = undefined for any 0 <= j < len
const makeArray = (len: number): Array => { … };

// @returns [args[0], …, args[N-1]]
const makeArrayContaining = (...): Array => { … };
```

  - function name is also the one thing you know clients read!

    best chance to tell them how to use it correctly

Example: Distinguishing Constructors (3/3)

- Factory functions have names
  - allow us to distinguish these cases

```
// @returns []
const makeEmptyArray = (): Array => { … };

// @returns A with A.length = len and
//     A[j] = undefined for any 0 <= j < len
const makeArray = (len: number): Array => { … };

// @returns A with A.length = len and
//     A[j] = val for any 0 <= j < len
const makeFilledArray =
    (len: number, val: number): Array => { … };
```

Be very, very careful…

Argument Order Bugs

```
// @returns A with A.length = len and
//     A[j] = val for any 0 <= j < len
const makeFilledArray =
    (len: number, val: number): Array => { … };
```

Be very, very careful…
Type checker won't notice if client mixes these up!

- Some famous bugs due to mixing up argument order!
- If you program long enough, you will see this one

Use Records to Force Call-By-Name

- Can use a record to make clients type names

```
// @returns A with A.length = len and
//     A[j] = val for any 0 <= j < len
const makeFilledArray =
    (desc: {len: number, value: number}): Array
```

  - takes one argument, not two
  - client writes "`makeFilledArray({len: 3, value: 0})`"

    much easier in JS than Java

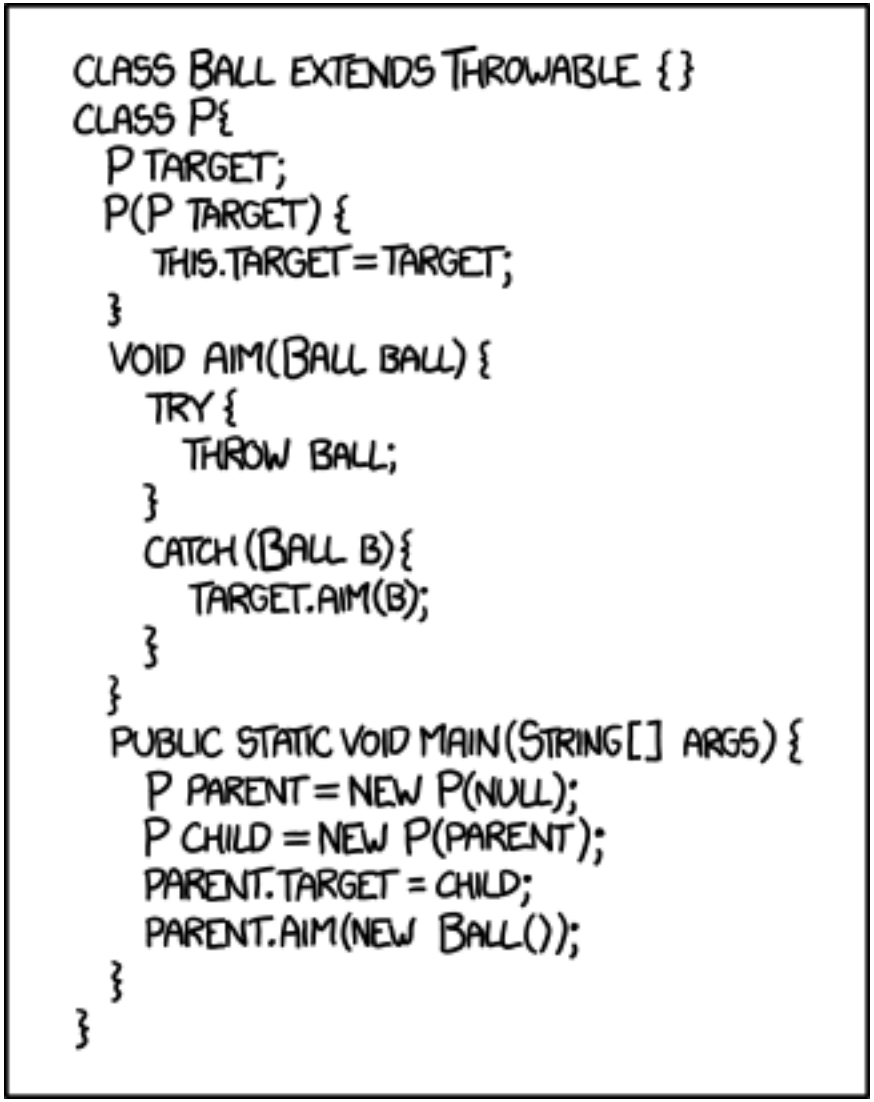
- Think about mistakes clients might make
  - be paranoid when debugging will be painful

CSE 331
Spring 2025

Design Patterns

Matt Wang

& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong



xkcd #1188

Administrivia (06/06)

- final exam Tue, June 10th, 12:30-2:20 in KNE 130
  - bring just your Husky Card (ID) and writing utensil

- exam resources since last lecture…
  - exam review session details finalized (see Ed post)
  - 23sp sample final videos published!
  - Matt's sample final + solutions out tonight

- Bob Bandes Nominations are out!

First: Wrapping Up

What We Hope You Got From 331: The Core

- A toolkit for reasoning about code correctness
  - within 331: formalized "expert intuition" with math
  - requires slow, careful, and rigorous thinking
  - used *before* testing & debugging (~ complementary)

- Learning *when* to use this toolkit
  - not every problem requires it!
  - treat reasoning as a spectrum

    most experts reason informally for simple problems…

    … use diagrams for difficult problems …

    … and bring out pencil & paper for brutal problems!

    (or, "automated reasoning" tools, e.g. proof assistants)

What We Hope You Got From 331: Bonuses

- learning JavaScript & TypeScript
  - different approaches to types & OOP than Java
  - some issues are fundamental to both languages
- writing complex web applications
  - async code is tricky!
  - client-server interaction is *complicated*
  - debugging client-server apps is **hard**!
  - made some fun projects :)
- computer science is much more than writing code
  - fundamental focus on reasoning & abstraction
  - but also: many applications of “theoretical” CS & math

If you want more…

- reasoning, math, and programming languages
  - CSE 341 (PL), CSE 401 (Compilers), CSE 403 (SWE)
  - CSE 505 (Grad PL), CSE 507 (Automated Reasoning)
  - check out UW PLSE!
- interactive application development
  - some great courses in CSE

    CSE 340 (interaction programming, mobile dev), CSE 154 (web dev)*

    more broadly: CSE 440 (HCI), CSE 442 (viz), CSE 443 (accessibility)
  - but also, large culture of free self-paced resources

    now have most of the vocabulary to learn reactive programming

    largest barrier is time & practice, not "theory"
- matt's summer advice™: build your own side project!

Course Evals

- This iteration of 331 is still relatively new
  - some things probably (?) went well
  - some things could still be better!

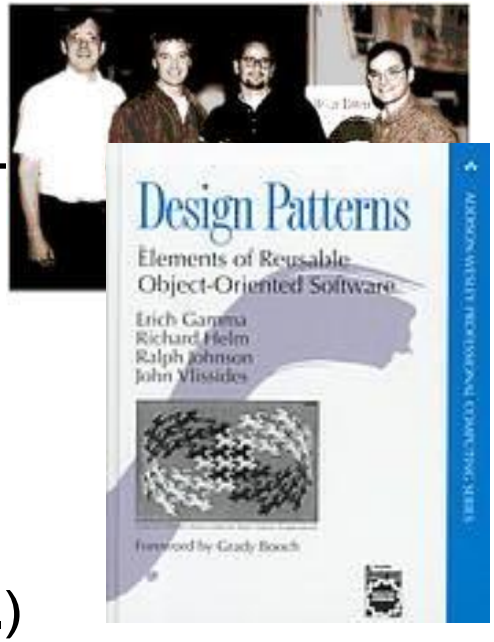
- Please give us feedback!
  - your perspective is valuable; we read *everything*!
  - one request: please be specific and actionable

    specificity helps us understand problems

    actionable suggestions scope out the solution space
  - CSE 331 A eval link, CSE 331 B eval link
  - also: your quiz section has an eval!

Design Patterns

Recall: Design Patterns



- Popularized in 1994 book of that name
  - written by the "Gang of Four"
    Gamma, Helm, Johnson, Vlissides
  - worked in C++ and SmallTalk
    (SmallTalk hugely influenced OOP in Java, etc.)

- Found that they independently developed many of the same solutions to recurring problems
  - wrote a book about them

- Many are problems with OO languages
  - authors worked in C++ and SmallTalk
  - some things are **not easy** to do in those languages

Parts of a Design Patterns

Each pattern in the book includes

- Problem to be solved
- Description of the solution
- Name of the pattern

Java Example: Iterator

- **Java Collections use the Iterator Design Pattern**
  - **enumerate a collection while hiding data structure details**
  - **return another ADT that outputs the items**

    that object knows how to walk through the data structure

    operations for retrieving the current item and moving on to the next one

- **Clever idea that is now used everywhere**
  - **Kevin remembers when C++ introduced iterators**
  - **huge improvement over code we were writing before**

Categories of Design Patterns (1/2)

The book has three categories of patterns

- **Creational**: factory function, factory object, builder, prototype, singleton, ...
- **Structural**: adapter, bridge, composite, decorator, façade, flyweight, proxy
- **Behavioral**: command, interpreter, iterator, mediator, observer, state, strategy, visitor, ...
  - we will not cover all, just some highlights

Categories of Design Patterns (2/2)

The book has three categories of patterns

- **Creational**: factory function, factory object, builder, prototype, singleton, ...
- **Structural**: adapter, bridge, composite, decorator, façade, flyweight, proxy
- **Behavioral**: command, interpreter, iterator, mediator, observer, state, strategy, visitor, ...

  – green and underlined = mentioned already

Creational Patterns

The book has three categories of patterns

- **Creational**: factory function, factory object, builder, prototype, singleton, ...
- **Structural**: adapter, bridge, composite, decorator, façade, flyweight, proxy
- **Behavioral**: command, interpreter, iterator, mediator, observer, state, strategy, visitor, ...
  - green and underlined = mentioned already

Why Creational Patterns?

- One third of the patterns deal with object creation
- We just saw why: constructors are terrible
  - surprisingly error-prone
  - several important limitations
    1. Cannot return an existing object
    2. Cannot return a different class
    3. Does not have a name!
- Already saw factory functions and singleton
  - yet we still need more!

Creational Pattern: Builder

- **Object that helps with creation of another object**
  - **constructor / factory requires you to give info all at once**
  - **builder lets you describe what you want bit by bit**

- **Java Example:** `StringBuilder`

```
StringBuilder buf = new StringBuilder();
buf.append("Total distance: ");
buf.append(distance);
buf.append(" meters.");
return buf.toString();
```

  - **each call adds more text / number to the final string**
  - **we can't do this with strings because strings are *immutable***

Builders and "Mutation XOR Aliasing"

- **Object that helps with creation of another object**
  - **constructor / factory requires you to give info all at once**
  - **builder lets you describe what you want bit by bit**

- **Good pairing: mutable Builder for an immutable type**
  - **must avoid aliasing with the mutable builder**

    e.g., never use it as a key in a BST or Map
  - **immutable object can be shared arbitrarily**

    no worries about aliasing

Writing a Builder

- **Builder is often written like this:**

```
class FooBuilder {
  …
  public FooBuilder setX(int x) {
    this.x = x;
    return this;
  }
  …
  public Foo build() { … }
}
```

  - **can then use them like this**

```
Foo f = new FooBuilder().setX(1).setY(2).build();
```

avoids worries about argument order

Recall: Argument Order Bugs

```
// @returns A with A.length = len and
//     A[j] = val for any 0 <= j < len
const makeFilledArray =
    (len: number, val: number): Array => { … };
```

Be very, very careful…
Type checker won't notice if client mixes these up!

- Some famous bugs due to mixing up argument order!
- If you program long enough, you will see this one
- Can fix with a record argument or a Builder
  - Java does not have record types, so we need the latter

Argument Builder

```
// Returns an array with length & value given in args.
public Integer[] makeFilledArray(args: Args) { … }

class Args {
  public int length;
  public int value;
}

Args args = new Args();
args.length = 10;
args.value = 5;
… = makeFilledArray(args);
```

- **code using the function is now more verbose…**

  can make this easier by giving them a Builder

Writing an Argument Builder

```
// Returns an array with length & value given in args.
public Integer[] makeFilledArray(args: Args) { … }

class ArgsBuilder {
  …
  public ArgsBuilder setLength(int length) {
    this.length = length;
    return this;
  }
  …
  public Args toArgs() { … }
}

… = makeFilledArray(new ArgsBuilder()
    .setLength(10).setValue(5).toArgs());
```

Structural Patterns

The book has three categories of patterns

- **Creational**: factory function, factory object, builder, prototype, singleton, ...
- **Structural**: adapter, bridge, composite, decorator, façade, flyweight, proxy
- **Behavioral**: command, interpreter, iterator, mediator, observer, state, strategy, visitor, ...

  – green and underlined = mentioned already

Recall: Java and Interoperability

- Mentioned this one in Topic 2...

- In Java, these two classes are not interoperable:

```
interface Duration {
  int getMinutes();
  int getSeconds();
}

interface AmountOfTime {
  int getMinutes();
  int getSeconds();
}
```

  - cannot pass one where the other is expected

Structural Pattern: Adapter

- **Mentioned this one in Topic 2...**

- **Get around this by creating an adapter**

```
class DurationAdapter implements AmountOfTime {
  private Duration d;

  public DurationAdapter(Duration d) {
    this.d = d;
  }

  int getMinutes() { return d.getMinutes(); }
  int getSeconds() { return d.getSeconds(); }
}
```

  - **makes a** `Duration` **into an** `AmountOfTime`

Adapters and Type Systems

- Adapters are often needed with nominal typing
  - design pattern working around a language issue

- With structural typing, these two interoperate:

```
type Duration = {min: number, sec: number};

type AmountOfTime = {min: number, sec: number};
```

  - can pass either where the other is expected
  - not an issue of concrete vs abstract

    still interoperable if we have `getMinutes` and `getSeconds` methods

Behavioral Patterns

The book has three categories of patterns

- **Creational**: factory function, factory object, builder, prototype, singleton, ...
- **Structural**: adapter, bridge, composite, decorator, façade, flyweight, proxy
- **Behavioral**: command, interpreter, iterator, mediator, observer, state, strategy, visitor, ...

  – green and underlined = mentioned already

Trees

- Trees are inductive data types
  - anything with a constructor that has 2+ recursive arguments

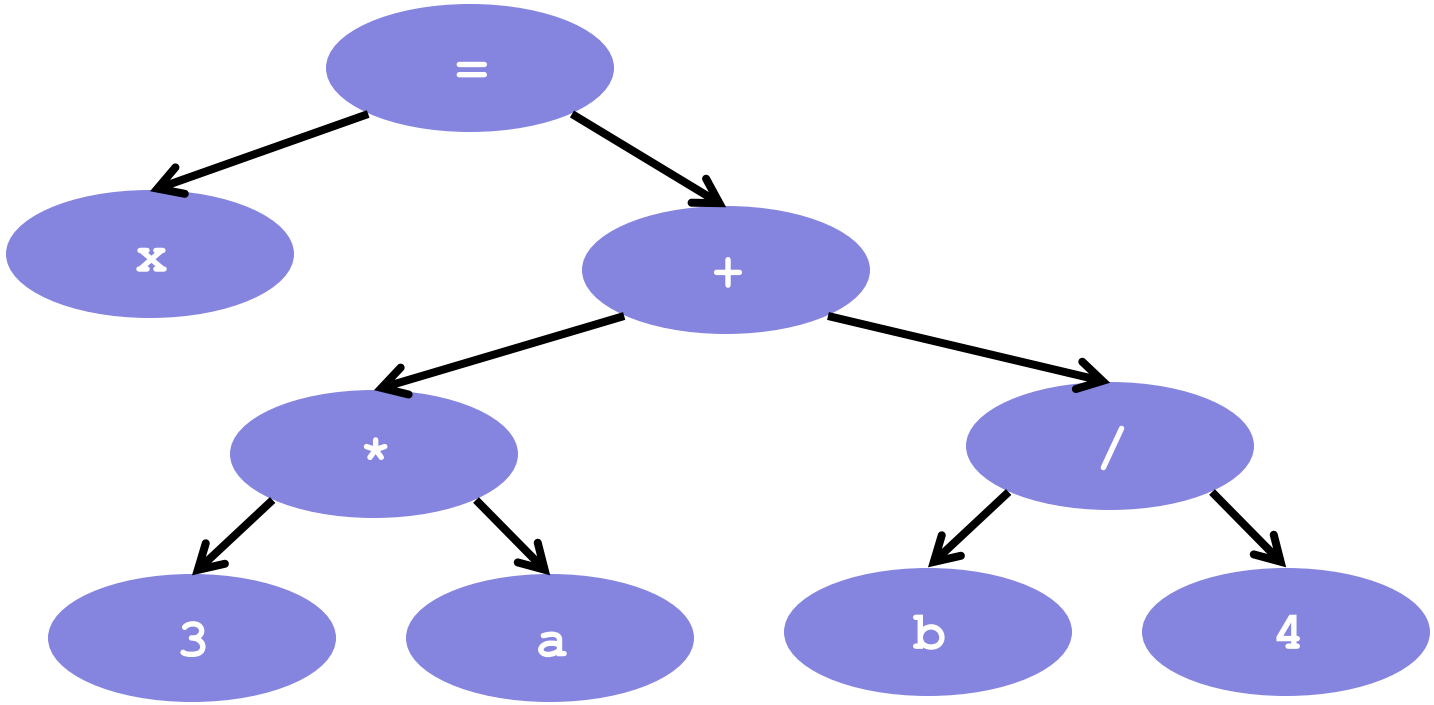
    HW8 tree (Square) has 4 recursive arguments

- They arise frequently in practice
  - HTML: used to describe UI
  - JSON: used for client/server communication
  - parse trees: represent code

Parse Tree Example

- **Output of parsing is a tree**
  - **encodes the order of operations**

- **Example: parse of "`x = a * 3 + b / 4`"**



Defining Parse Trees Inductively

- Output of parsing is a tree
  - records the order of operations

- Parse tree is an inductive data type

type Expression := variable(name: $\mathbb{S}^*$)
| constant(val : $\mathbb{Z}$)
| plus(left : Expr, right : Expr)
| times(left : Expr, right : Expr)
| divide(left : Expr, right : Expr)
| assign(name : $\mathbb{S}^*$, value : Expr)

  - parse of "`x = a * b + c / d`"

assign("x", plus(times(constant(3), variable("a")),
divide(variable("b"), constant(4)))

Operations on Parse Trees (1/2)

- Compilers perform various operations on expressions
  - type check
  - evaluate
  - code generation

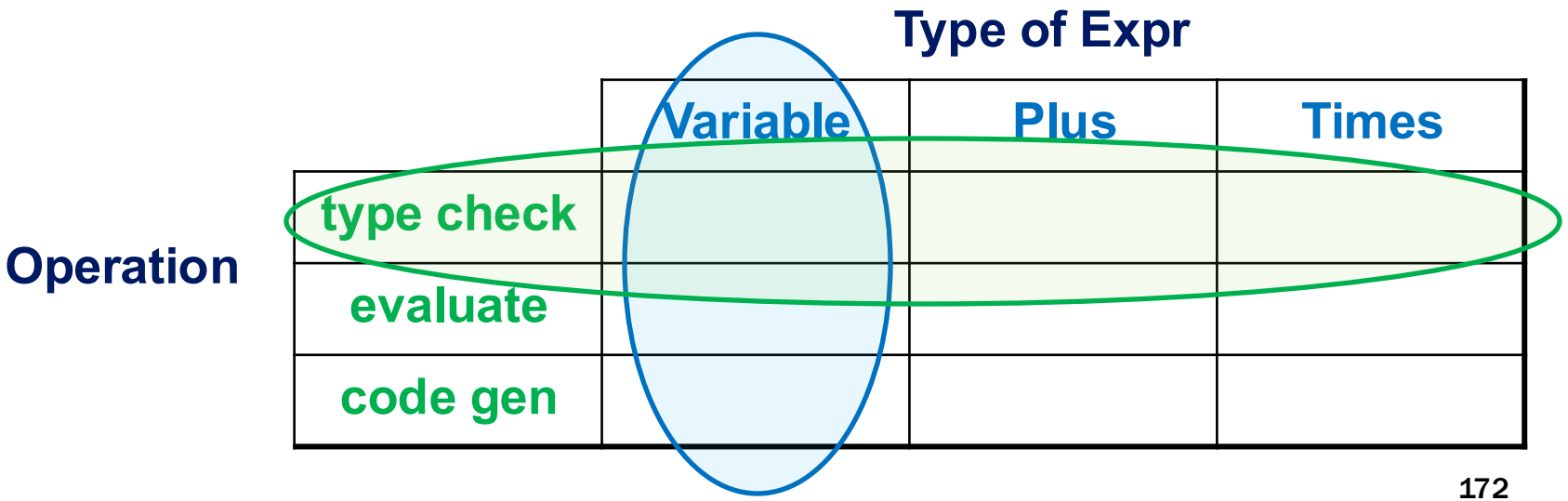
- Each operation defined for each type of expression

Operation \ Type of Expr	Variable	Plus	Times
type check			
evaluate			
code gen			

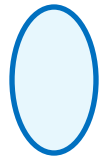
Operations on Parse Trees (2/2)

- Need to write code for each box
  - each case is slightly different

- Two reasonable ways to organize into files
  - file per expression type: Interpreter pattern
  - file per operation: Procedural pattern



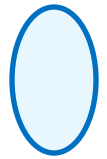
Interpreter Pattern Example



```
interface Expr {
  typeCheck = (c: Context) => Type,
  evaluate = (c: Context) => number | undefined,
  generate = (c: Context) => List<Instruction>
}

class Variable implements Expr {
  name: string;
  typeCheck = (c: Context): Type => {
    return c.get(this.name);
  }
  evaluate = (c: Context): number | undefined => {
    return undefined;
  }
  …
}
```

- Each type of expression is a class

Interpreter Pattern Tradeoffs



```
interface Expr {
  typeCheck = (c: Context) => Type,
  evaluate = (c: Context) => number | undefined,
  generate = (c: Context) => List<Instruction>
}
```

- Easy to add new types of expression
  - new subtype of `Expr`
  - goes into its own file
- Hard to add new operations
  - new method of `Expr`
  - changes *every* file

Procedural Pattern Example

```
interface Procedure<R> {
  processVar = (v: Variable, c: Context) => R,
  processConst = (n: Constant, c: Context) => R,
  …
}

class TypeChecker implements Procedure<boolean> {
  processVar = (v: Variable, c: Context): boolean => {
    return c.has(v.name);
  }
  processConst = (n: Constant, c: Context): boolean => {
    return true;
  }
  …
}
```

- Each type of procedure is a class
  - one method for each type of expression

Procedural Pattern Tradeoffs

```
interface Procedure<R> {
  processVar = (v: Variable, c: Context) => R,
  processConst = (n: Constant, c: Context) => R,
  …
}
```

- Easy to add new types of operations
  - new subtype of `Procedure`
  - goes into its own file
- Hard to add new expressions
  - new method of `Procedure`
  - changes *every* file

Interpreter vs Procedural Pattern

- **Both patterns are reasonable**
  - **best choice is problem-dependent**

    for a compiler, I prefer the procedural pattern

- **But there is a problem with Procedural in OO**
  - **suppose `e` is an `Expr` but we don't know which one**
  - **how do we call the right method?**

    could be `processVar`, `processConst`, `processPlus`, …

Problems with Procedural Pattern in OO

```
const process = (p: Procedure, e: Expr, c: Context) => {
  if (e instanceof Variable) {
    p.processVar(e, c);
  } else if (e instanceof Constant) {
    p.processConst(e, c);
  } else if (e instanceof Plus) {
    p.processPlus(e, c);
  } else …
}
```

- Not great, Bob!
  - code is slow
  - will call it enough times that this will matter
- There is a solution, but… buckle up!

Dynamic Dispatch (good case in Java)

```
interface Expr {
  boolean typeCheck(Context c);
}

class Variable implements Expr {
  public boolean typeCheck(Context c) { … }
}

class Constant implements Expr {
  public boolean typeCheck(Context c) { … }
}
```

- Java / TypeScript (or any OO) makes this case easy

```
Expr e = …
e.typeCheck(c);          // e could be any Expr
```

  - automatically “dispatches” to the right method

Dynamic Dispatch (bad case in Java)

```
interface Procedure<R> {
  R process(Variable v, Context c);
  R process(Constant n, Context c);
  …
}
```

overloading

```
class TypeChecker implements Procedure<Boolean> {
  Boolean process(Variable v, Context c) { … }
  Boolean process(Constant c, Context c) { … }
  …
}
```

- **This is impossible in Java:**

```
TypeChecker t = new TypeChecker();
Expr e = …
t.process(e, c);          // e could be any Expr
```

"Fixing" Impossible Dynamic Dispatch

- This is impossible in Java:

```
TypeChecker t = new TypeChecker();
Expr e = …
t.process(e, c);          // e could be any Expr
```

- Need to put "`e`" before " . " to get dynamic dispatch
  - here's how we do that… (gulp)

Implementing Double Dispatch

```
interface Procedure<R> {
  R process(Variable v, Context c);
  R process(Constant n, Context c);
  …
}

interface Expr {
  R perform(Procedure<R> p, Context c);
}

class Variable implements Expr {
  public R perform(Procedure<R> p, Context c) {
    p.process(this, c);
  }
}

class Constant implements Expr {
  public R perform(Procedure<R> p, Context c) {
    p.process(this, c);
  }
}
```

calls `process(Variable, Context)`

calls `process(Constant, Context)`

Using Double Dispatch

```
interface Procedure<R> {
  R process(Variable v, Context c);
  R process(Constant n, Context c);
  …
}
interface Expr {
  R perform(Procedure<R> p, Context c);
}
```

- **We can now do this**

```
Process p = new TypeChecker();
Expr e = …
e.perform(p, c);          // e could be any Expr
```

  - **calls `Expr.perform`, which calls `TypeChecker.process`**
  - **two function calls is still faster than all the "`if`"s**

Multiple Dispatch?

- This works, but… why so hard?
- Other languages just let you do this:

```
Process p = new TypeChecker();
Expr e = …
p.process(e, c);          // e could be any Expr
```

  - or even more general “multiple dispatch” cases
  - use a better language?

Traversing Trees

- **Same idea is used to traverse trees**

type Expression := variable(name: $\mathbb{S}^*$)
| constant(val : $\mathbb{Z}$)
| plus(left : Expr, right : Expr)
| times(left : Expr, right : Expr)
| divide(left : Expr, right : Expr)
| assign(name : $\mathbb{S}^*$, value : Expr)

  - **parse of "`x = 3 * a + b / 4`"**

assign("x", plus(times(constant(3), variable("a")),
divide(variable("b"), constant(4)))

  - **would like to process ("visit") each node in this tree**

Visitor Pattern

```
interface ExprVisitor {
  visitVariable = (v: Variable) => void,
  visitConstant = (n: Constant) => void,
  visitPlus = (p: Plus) => void,
  …
}

interface Expr {
  // Visits this node and all its children.
  accept = (v: ExprVisitor) => void
}

class Variable implements Expr {
  name: string;
  accept = (v: ExprVisitor): void => {
    v.visitVariable(this);
  }
}
```

…

Visitor Pattern (with child nodes)

- Combines double dispatch with tree traversal

```
class Plus implements Expr {
  left: Expr;
  right: Expr;

  accept = (v: ExprVisitor): void => {
    left.accept(v);
    right.accept(v);
    v.visitVariable(this);
  }
}
```

  - traverses children before visiting parent

Visitor Pattern (in steps)

```
p.accept(v)
  t.accept(v)
    h.accept(v)
      v.visitConstant(h)
    a.accept(v)
      v.visitVariable(a)
    v.visitTimes(t)
  d.accept(v)
    …
    v.visitDivide(f)
  v.visitPlus(p)
```