CSE 331

Object-Oriented Programming

Kevin Zatloukal

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

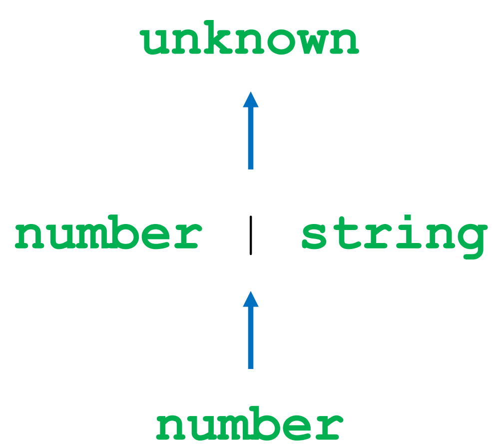
Subtypes of Concrete Types

- 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

- 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

- 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

```
{name: string}
       ↑
{name: string, completed: boolean}
```

Subtyping Used by TypeScript

- **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 <u>passed</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

- TypeScript uses subtyping in function calls

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

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

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

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

- Similar rules in Java

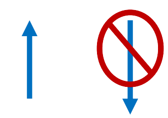
Subtyping Used by TypeScript

- 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

`{options: string[], index: number}`



`OptionState`

- `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

Subtypes Are Substitutable

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

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

Subtypes of 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…**

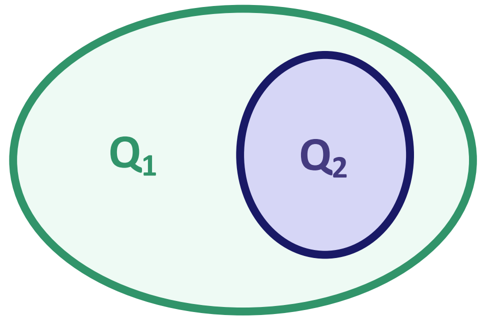
   must accept all the inputs that A’s does

   must also promise everything in A’s postcondition

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

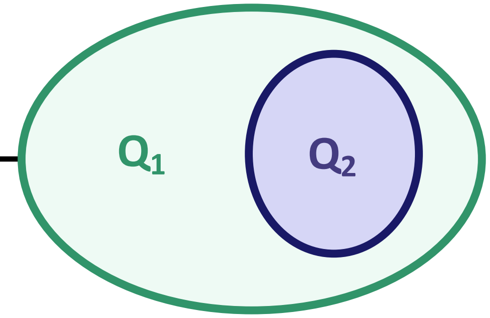
Review: Stronger Assertions vs Specifications

- **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

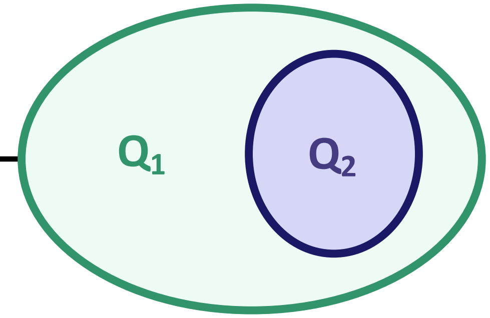



```
interface A {
  f: (x: number) => number

  // @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



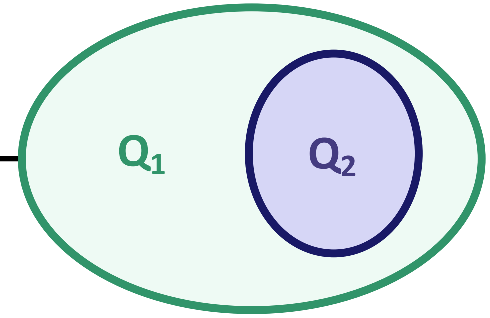

```
interface A {
  f: (x: number) => number

  // @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




```
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

```
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

- 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

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

- **Subclassing does not guaranty 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

- 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

- 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

```
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

```
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**

Subclassing Creates Tight Coupling

- 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

```
// 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

```
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

Example 3: Tight Coupling

- 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 no *order* of method class
  - 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

Equality

Equity 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!

Equality

- 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…

Equality

- **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. $\text{equal}(a, a) = T$ for any $a : A$ — reflexive
  2. $\text{equal}(a, b) = \text{equal}(b, a)$ for any $a, b : A$ — symmetric
  3. if $\text{equal}(a, b)$ and $\text{equal}(b, c)$, then $\text{equal}(a, c)$ for any … — transitive

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

Example: Duration

- **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

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

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

$$\begin{aligned} &\text{equal}(\{\text{min: m, sec: s}\}, \{\text{min: m, sec: s}\}) \\ &\quad = (m = m) \text{ and } (s = s) && \textbf{def of } \text{equal} \\ &\quad = T \text{ and } T \\ &\quad = T \end{aligned}$$

proof by calculation
that it holds for any record

  - symmetric

$$\begin{aligned} &\text{equal}(\{\text{min: m, sec: s}\}, \{\text{min: n, sec: t}\}) \\ &\quad = (m = n) \text{ and } (s = t) && \textbf{def of } \text{equal} \\ &\quad = (n = m) \text{ and } (t = s) \\ &\quad = \text{equal}(\{\text{min: n, sec: t}\}, \{\text{min: m, sec: s}\}) && \textbf{def of } \text{equal} \end{aligned}$$

Example: Duration

equal({min: m, sec: s}, {min: n, sec: t}) := (m = n) 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**: `"" != " "`

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

Example: List Equality

- **Can define equality on List type this way:**

equal(nil, nil)	:=	T	
equal(nil, b :: R)	:=	F	
equal(a :: L, nil)	:=	F	
equal(a :: L, b :: R)	:=	F	if $a \neq b$
equal(a :: L, b :: R)	:=	equal(L, R)	if $a = b$

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

$$\text{equal}(\boxed{1} \rightarrow \boxed{2}\,,\ \boxed{1} \rightarrow \boxed{2}) = \text{equal}(\boxed{2}, \boxed{2})$$

$$= \text{equal}(\text{nil}, \text{nil})$$

$$= \text{T}$$

Example: List Equality

- **Can define equality on** List **type this way:**

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

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

$$
\begin{aligned}
\text{equal}(\boxed{1} \rightarrow \boxed{2}\,,\ \boxed{1} \rightarrow \boxed{3}) &= \text{equal}(\boxed{2}\,,\ \boxed{3}) \\
&= \text{F}
\end{aligned}
$$

Example: List Equality

- **Can define equality on List type this way:**

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

- **Has all three required properties**
  - **how would we prove $\text{equal}(\text{L}, \text{L})$ 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 Again

```
// 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

```
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

```
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

```
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

```
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**

Example: NanoDuration Again

- 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

Recall: Tight Coupling (Example 3)

```
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 is not true in method call!

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 <u>can</u> 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
  }
}
```

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

- 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

- 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

- 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
```

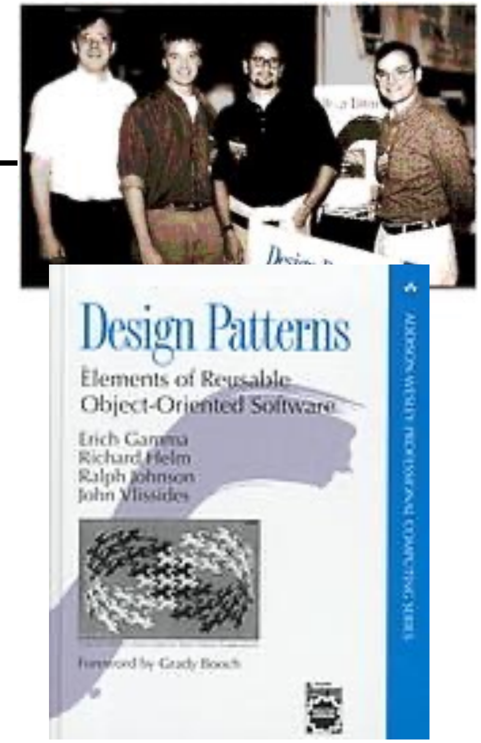
  - takes one argument, not two
  - client writes "`makeFilledArray({len: 3, value: 0})`"

    much easier in JS than Java

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

More Design Patterns

Recall: Design Patterns



- Introduced in the book of that name
  - written by the “Gang of Four”

    Gamma, Helm, Johnson, Vlissides
  - worked in C++ and SmallTalk

- Found that they independently developed many of the same solutions to recurring problems
  - wrote a book about them
  - required at least three real-world uses to be included

- Many are solutions to problems with OO languages
  - authors worked in C++ and SmallTalk

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
  - I remember when C++ introduced iterators
  - huge improvement over code we were writing before

Categories of Design 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, ...
  - we will not cover all, just some highlights

Categories of Design 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 = mentioned already

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*

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

- 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

Creational Pattern: 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

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());
```

Categories of Design 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 = mentioned already

Structural Pattern: Adapter

- 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`

Structural Pattern: Adapter

- 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

Categories of Design 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 = mentioned already