6.031 EXAM #1

- static type - like String, int
- static checking / dynamic

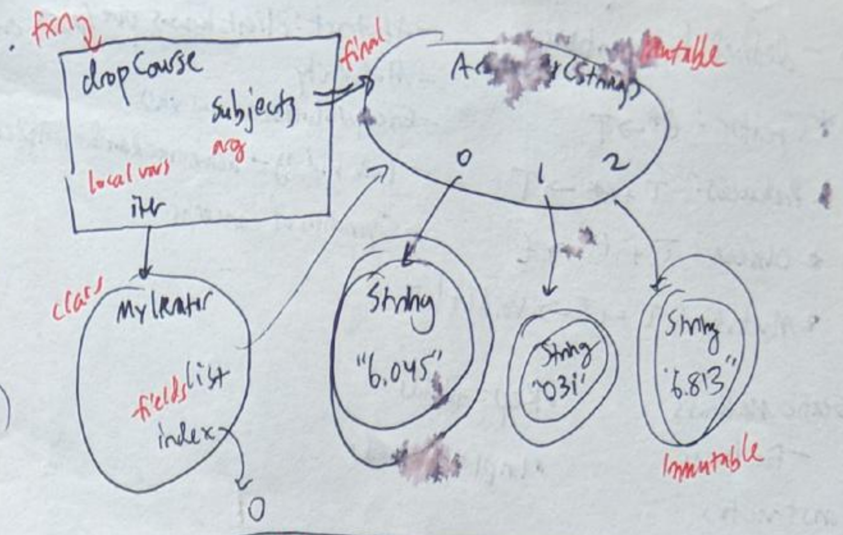
  ↓ wrong Types, #args, names

  ↓ - illegal args (Out of Range) - Null

- primitive, object/ref types (classes, strings, arrays)

  ↳ operations + methods

  ↓ some are overloaded



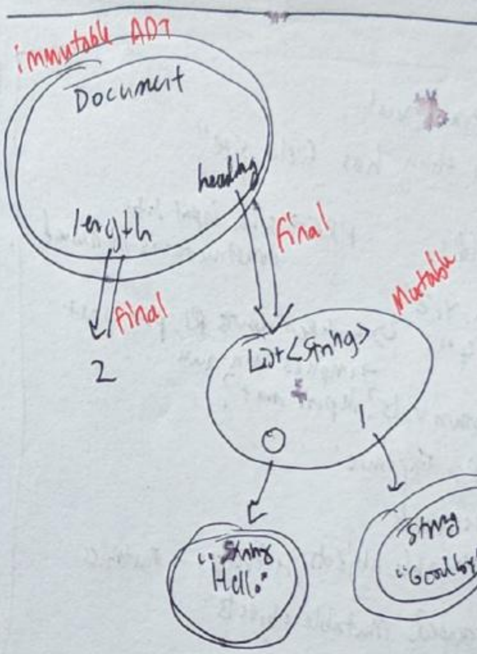
Testing - spec, test, code

- Black box testing - acc to specs.
- Glass box testing - knowing how fxn works

Ex partitions:

text.length = 0, 1, >1

location of bound is beginning, middle, end



CODE REVIEW

→ Good var names

→ DRY → Avoid Magic #s → Return, not print

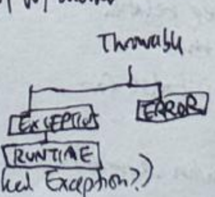
→ Fail Fast → One Purpose/var → Global (public static) are bad

→ Global constants (public static final) are good

SPEC

→ DON'T INCLUDE STUFF FROM REP (like fields)

- behavioral eq: whether you could replace one impl w/ another for client



PRECONDITION

- constraints on inputs
- type, args implicit

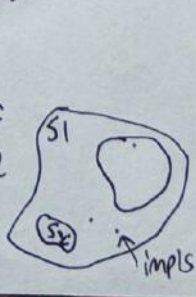
POSTCONDITION

- return val
- exceptions (throws Checked Exception?)
- modifying objs (usu not)

- Deterministic: one return value for an input nondeterministic: random
- underdetermined: might be ~~the~~ >1 output
- operational - steps
- declarative - no specific impl steps

S2 is stronger than or equal to >= S1 if

- S2 precondition ≤ S1 and
- S2 ≥ S1



ADTS - defined by operations

$* = 0+$

- * Creator: $t^* \to T$
- * Produces: $T + t^* \to T$
- * Observer: $T + t^* \to t$
- * Mutator: $T + t^* \to$ void | t | T

- Abstract: client knows pre/post only
- Modularity
- Encapsulation - local vars
- Info hiding - some freedom for implementer
- Separation of concerns

- Static methods
  - Factories
- Constructors
- Instance Methods

- Rep = fields
- Impl = methods

AF(fields) = abstract val

Ex. "the abstract thing has field size"

RI → constraints on fields

"field1 is x, y, z"

"$0 \leq$ field2 ≤ 4"

PRE → when input into constructor is mentioned

→ checkRep asserts RI, pre, post

→ implied non-null

EXP → private? final? return vals? input mut?

ways to screw up Rep Exposure:

- Not private fields
- returning Mutable objects, references to Mutables
- returning wrapped mutable objects

Interfaces: method signatures, no bodies

can have statics

- B subtype of A, then every B satisfies spec for A
- B spec $\geq$ A spec

Why? Documentation, performance tradeoffs w/ impls

- impls can be stronger

EQUALITY - Object Contract: equals() defines eq relation

equals() is consistent

X.equals(null) is false for nonnull

hashCode = for when .equals()

default = referential eq

immutable types: observational = behavioral equality

observers + producers

all, inc mutators

- impl both hashCode, equals

Mutable types: compare refs, behavioral equality

- don't override equals hashcode

ALL → (a = b) <=> AF(a) = AF(b)

→ eq relation

6.031 EXAM #2

Code Review

→DRY →Avoid Magic #s

→Fail Fast → One purpose/var

→Return not print → Good var names

→Global (public static) are bad

→Global constants (public static final) are good

SPEC

→no rep stuff

→behavioral eq - whether you can replace one impl w/ another for client

Precond - constraints on input, type, args implicitly

Postcond - return val, exceptions, modifying objs

- Deterministic - one return value for an input · Nondeterm: random
- Underdetermined: might be >1 output
- operational - steps
- declarative - no specific impls
- S2 is stronger than or equal to ≥ S1 if
  - S2 pre ≤ S1 &
  - S2 ≥ S1

ADTS -

Creator: t* → T

Producers: T + t* → T

Observers: T + t* → t

Mutator: T + t* → void | t | T

- Abstraction - client only knows pre/post
- Modularity
- Encapsulation - local vars
- Info hiding - some freedom for implementer
- Separation for concerns

Rep - fields

Impl - methods

AF(fields) = abstract val

PRE → when input into constructor is mentioned

RI → constraints on fields

checkRep → RI, pre, post, implied nonnull

EXP → private? final? return vals? input mutable?

ways to screw it up:

- Not private fields
- returning Mutable objects, references to mutables
- returning wrapped Mutable objects

Equality

→Object Contract: equals() defines eq relation

x.equals(null) is false for nonnull

hashcode = for when .equals()

Immutable types: observational = behavioral equality

↳ observers + producers

↳ all, inc mutators

- impl both hashcode/equals

mutable types: compare refs, behavioral equality

don't override equals/hashcode

ALL (a=b) <=> AF(a) = AF(b)

eq relation

Interfaces: method signatures, no bodies, can have statics

B subtype of A, then every B satisfies A's spec

B spec ≥ A spec

Recursive DTs

Tree<E> = Empty + Node(e: E, left: Tree<E>, right: Tree<E>)

put in interface

Regex

: regular grammar - substituting things in

[x]* = x 0 or more times

x+ = x 1 or more times

yz = y followed by z

y|z = either y or z

y? = 0 or 1 y

[xyz] = x|y|z

[^a-c] = everything except a, b, c

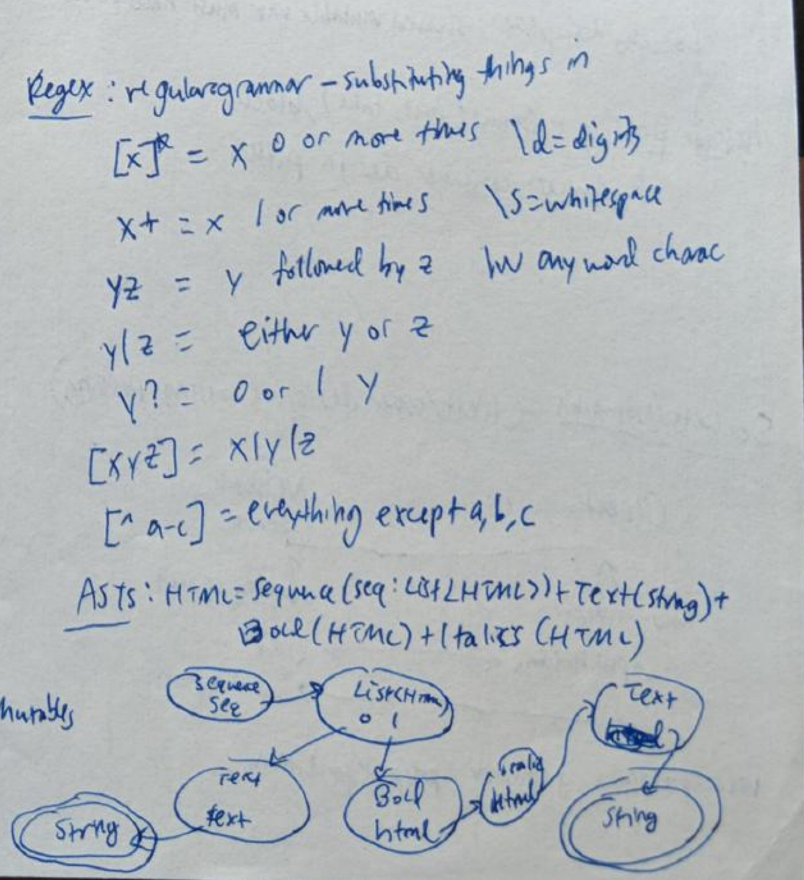
\d = digits

\s = whitespace

\w any word charac

ASTs

: HTML = Sequence(seq: List<HTML>) + Text(String) + Bold(HTML) + Italics(HTML)



Concurrency

process - instance of running program
isolated from other processes

shared memory - r/w w/shared objects
message passing - concurrent modules send
info back/forth

thread - many threads in process
- time-slicing

```
new Thread(new Runnable(){    anonymous
                              -reduces scope
    public void run() {
        do Something
    }
}).start()
```

Thread Safety

Confinement - don't share vars b/w threads → can't reassign vars in Runnable if final → fields are ou
→ local vars are confined

Immutability - make shared vars unreassignable or immutable
→ no beneficent mut.

Threadsafe data type - put shared data in. → atomic ops.

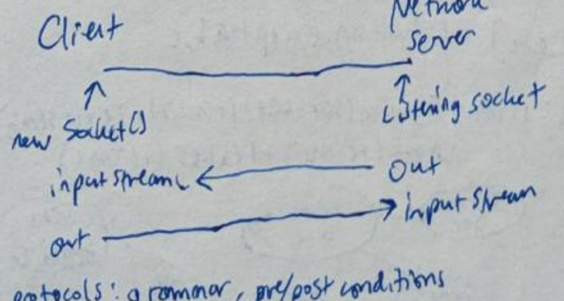
Synchronization - keep threads from accessing shared vars/data at same time

locks - monitor pattern. reassigning vars releases the lock!

Locking discipline - shared mutable var must have lock

Message passing - Queues (put, take), blocks.
Producer-consumer design pattern

Sockets/Networks - Client/server design (message passing)



wire protocols: grammar, pre/post conditions

Callbacks

Attach listener - Listener Pattern
- event source publishes events
- Listeners subscribe, provide fxn to be called

First class values are things that can be passed thru to fxns or returned

lambda (args) → {do something}

STREAMS - Stream <E>

Map (E→F) → Stream <F>
Filter (E→bool) → Stream<E>
Reduce (F, F×E→F, F×F→F) → F

Stream.forEach(fxn)

```
input.filter(item → !item.equals("yucky"))
    .map(item→item.substring(6));
List.of(1,2,3).stream().reduce("", (String s, int n)→ s+n,
                                   (String s, String t) → s+t);
```

Interpreter vs Visitor Pattern

Formula Interface

```
Not →
@Override ~~accpt_func~~ public accept(visitor)
    ~~return [illegible]~~
    return visitor.on(this)
```

```
Visitor class in Formula impls Formula.Vis.
    @Override on
    (Not not){
        return not.formula
        accept(this)
```