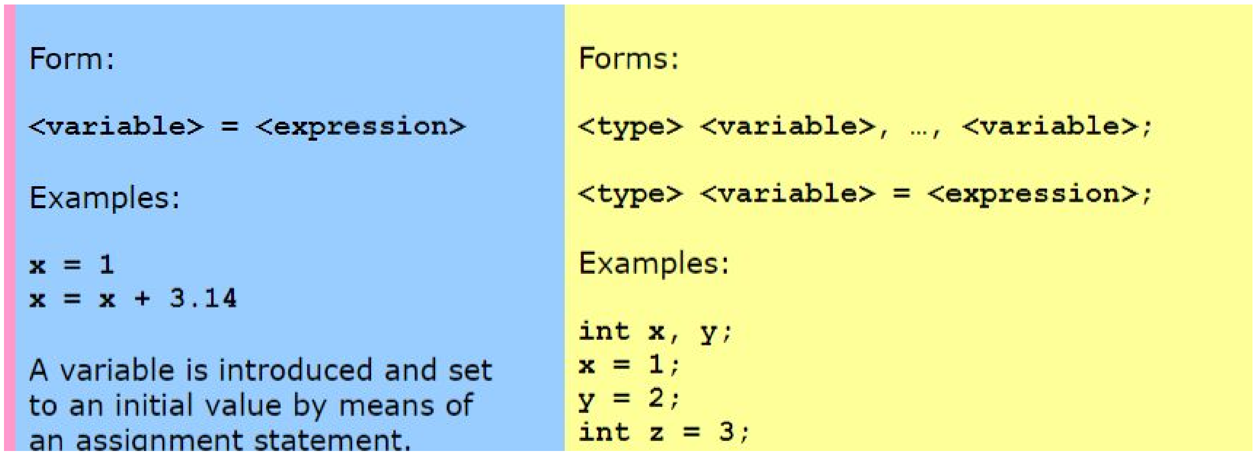
Reading 1: Static Checking

(lots of java, checks before running) vs Dynamic Checking (lots of Python, checks during running)

- java checks types and syntax before running.
- both python and java might run into list errors during running. python runs into type errors during running.

Python vs Java

- Java app defines at least one class that defines a main method
- both have literals, identifiers, reserved words
- java has primitive (double, int, char, bool) and reference types (classes, strings, arrays, etc). python, all data types are reference types, as everything is an object
- casting to string in java: “” + val
- 

Form:

```
<variable> = <expression>
```

Examples:

```
x = 1
x = x + 3.14
```

A variable is introduced and set to an initial value by means of an assignment statement.

Forms:

```
<type> <variable>, …, <variable>;

<type> <variable> = <expression>;
```

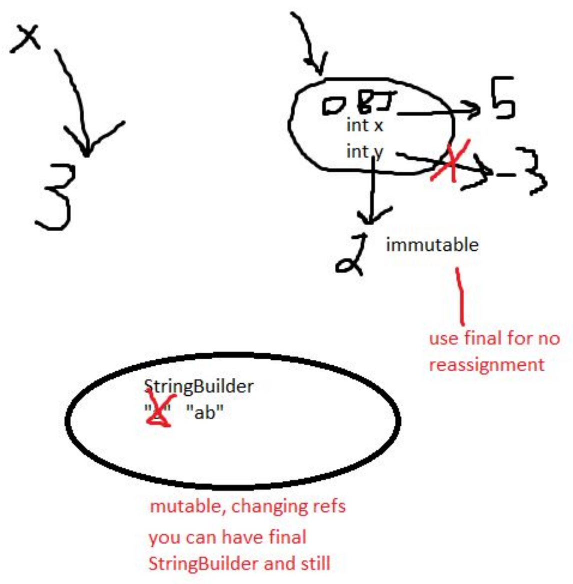
Examples:

```
int x, y;
x = 1;
y = 2;
int z = 3;
```

- input - use scanner
- java interface: specifies the set of methods that an implementing class must include.
  - interface can extend another interface. like List extends Collection
  - ArrayList implements List interface (ie it has all the defined methods that List would require, like add, sort, etc)

Snapshot Diagrams

- some things are immutable. so you don't have shallow copies with this (and you can indicate this w a double circle)
- two variables pointing to the same object is aliasing


x
3
OBJ
int x
int y
5
3
2
immutable
use final for no reassignment
StringBuilder
"ab"
mutable, changing refs
you can have final StringBuilder and still

Collections

- Arrays - reference type!
  - int[] varName = new int[size];
  - initialize default to 0, or null for strings, or any ref type
- List<String> - implementing classes include ArrayList and LinkedList
- Sets - implementing classes include HashSet and TreeSet
- Map - TreeMap, SortedMap, HashMap
- Iterator - all collections implement the Iterable interface.
  - one method - .iterator()
  - this can next(), hasNext(), remove(), etc

Map<String, Int> newMap = new HashMap<>();

Enums are apparently a thing outside google

public enum Month { JANUARY, FEBRUARY.....DECEMBER;}

Reading 3: Testing

- Validation
  - Verification - formal reasoning about a program, like a real proof
  - Code review - oof
  - Testing - running program on inputs and checking results
- Test First Programming (iterative)
  - write specification first - describes input and output of function, types of params, return val
- how to choose good test cases

    - divide input space into subdomains
    - EX BigInteger.multiply: a x b = a.multiply(b) ==> a,b could be pos or neg, or 0, +/-1 (special cases), really big numbers
    - cover boundaries
- unit tests tests individual modules @Test, assertEquals, assertTrue
    - assertEquals(expected, actual)
    - Details testing strategy, which tests covers which partitions
- Integration tests
    - tests a combo of modules
    - mock objects but apparently they aren't used in 031! smfh
- black box testing - testing according to specs
- glass box testing - testing according to knowing how the function works
- coverage
    - statement coverage- is every statement run by some test
    - branch coverage - for every if/while, are both T and F direction taken
    - path coverage - is every combo of branches taken by a test?
- regression testing - running tests after every change
    - every time you find a bug, make a new test!

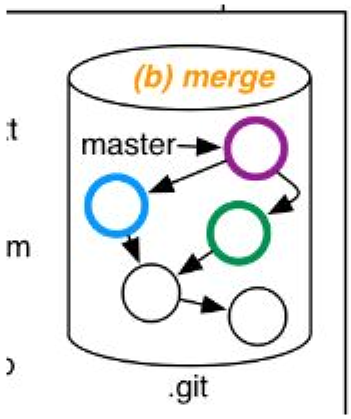
reading 4: CODE REVIEW

- Improve the code and the programmer
- style standards
- **commenting**
    - javadoc (spec) - document *assumptions*
    - cite sources
- **Fail fast!** Static is faster than dynamic is faster than no error.
- **avoid magic numbers** - anything that's not 0 1 or 2
- **DON'T REPEAT YOURSELF**
- **keep one purpose per variable**
    - it's a good idea to use final for method parameters that are unchanged
- **better names**
    - camelcase
    - uppercase, underscores for constants
    - packages are lowercase
- always use spaces
- **global variables are bad** (public static) - ie ones that can be accessed anywhere, changed anywhere
- **global constants (public static final) are good** - ones that can be accessed anywhere, not changed anywhere
    - final means variable is not reassignable, but an object can still be mutable!
- **variables**

    - **local - appear when method starts and then disappears**
    - **instance - appear with an instance, disappears when object goes away**
    - **static - program starts and exist forever with class (like its a class level variable)**
- System.out.println() should only happen in the highest level parts of a program

reading 5: Version Control

- **diff can find differences between things of text**
- things that you should be able to do
    - reverting to a past version
    - comparing two versions
    - pushing a version to another location
    - merging two versions
    - log information (see who did stuff) blame
- multiple branches
- Distributed vs centralized
    - Centralized VC systems - one collab graph
        - share to and from the master repo
    - Distributed (like Git)
        - you can experiment w different versions of code, and then merge together
- terms
    - repository: a local or remote store of the versions in our project
    - working copy: a local, editable copy of our project that we can work on
    - file: a single file in our project
    - version or revision: a record of the content of our project at a specific point in time
    - change: difference between two versions
    - head: current version
- features
    - reliable, multiple files, meaningful versions, revert, compare, history
    - multiple people to merge, track responsibility, work in parallel, work in progress
- parts: working directory -> staging area -> .git directory (local commits), and the Git object graph
    - the github.mit one is another copy of this graph
- commands
    - `git clone` copies the graph
        - git clone into ps0/.git
        - check out from master (ps0/.git)
    - `git log [file_name]` - shows commits, messages
    - `git lol` shows the tree, and commit hashes
    - `git diff [commit_hash1] [commit_hash2]`

            - `git diff` by itself just shows differences between local and last commit
    - `git log --name-status` can depict whether files were added/modified/deleted in each commit
    - `git show` [commit_hash] [:]
        - whos details of a commit
        - : shows the files in the repo
    - git diff shows differences between working copy and staging area
    - git show [commit hash] [:file]
        - :file means you can see what's inside the file at a given commit
    - git checkout SHA -- file: reverting a single file to a specific commit
    - `git checkout -b [branch_name]` creates a new branch
    - `git checkout [branch_name]` switches branches to this existing branch
    - `git add/rm` adds or removes files from the commit stage
- commit graph - a snapshot of our project, which is a tree node. multiple commits can share the same copy of a file, if it hasn't been modified a lot.
    - HEAD POINTS DOWN
    - commit points to a tree, which points to each file in the commit (snapshot of repo at that time), and points to the last commit
- push/pull
    - origin is the remote repo
    - git commit adds commits to the **local history** on the **master branch**
    - git push origin master
    - git pull
- merging
    - when you git pull, you pull nodes from the tree into your own object graph
    - then it attempts to merge
    - 

    - now you can git push

reading 6: Specifications

- **good because it lets implementer change the implementation without having to tell clients (it's a firewall between client and implementer)**
    - **decoupling -** allows the code of the unit and the code of a client to be changed independently
- **pros**
    - good specs can make code faster bc some states in which a method is called might not happen
- Are two pieces of code the same or diff?
    - **behavioral equivalence -** whether we could replace one implementation with the other (in the eye of the client)
        - as long as they both fit the specs, it's good
- what a spec consists of
    - **precondition: "requires"**
        - obligation of the caller of the method
    - **postcondition "effects"**
        - obligation of the implementer
    - **if precondition, holds, then postcondition must hold**
    - **if precondition fails, then client has a bug**
- java convention - javadocs
    - @param - preconditions
    - @return - post conditions
    - other tags
        - @author
        - @version
        - @param
        - @return
        - @throws
        - @see
        - @since
        - @deprecated
- null
    - primitives cannot be null
    - other vars can (like strings or array)
    - **null's are often disallowed in parameters and return values**
        - **there exists @NonNull**
        - null isn't clear
- spec should NOT talk about
    - private fields
    - local variables

- testing and specs
    - black box tests are chosen w spec in mind
    - glass box tests STILL gotta follow the spec
    - unit test
        - even if one method fails to satisfy its spec, other unit tests should still run
    - integration test
        - diff methods have compatible specs
- we can also assume that **mutation is disallowed unless stated otherwise** (no mutation to inputs)
- types and stuff like that are already implicitly included in the function def. so you don't need to explicitly say it again in the pre/postcondition
- try catch and make it easier to find exceptions and fail fast and then do something in response
- checked vs unchecked exceptions
    - checked ones throw a checked exception, where the possibility must be declared in the signature.
    - this is a possibility.
    - unchecked ones signal bugs
        - not expected to be handled by the code
        - like nPE, arrayindexoutofbounds, or outofmemory
- throwable hierarchy
    - throwable is class of objects that can be thrown or caught
    - in this there are errors and exceptions
        - error
            - generally not caught
            - indicates a bug
        - exception
    - runtimeException, Error and its subclasses are unchecked
    - all others are checked
- when defining own exceptions, subclass runTimeException (for unchecked) or Exception (to make it checked)
- when documenting an expected exception, use @throws and declaring throws
- unchecked exception (NOT UNEXPECTED FAILURES)- omit throws clause, but maybe define it in the (postcondition)

reading 7: Designing specs

- How deterministic is it? Can the implementer choose legal outputs?
    - deterministic - one return value, and only one final state is possible
    - not deterministic (underdetermined) - there may be more than one legal output
- How declarative it is? Does it say how to compute the output?

- How strong is it? Does the spec have a small set of legal implementations?

Declarative vs Operational Specs

- operational - tell you specific steps on how the method performs, like pseudocode
- declarative - dont give details on intermediate steps
    - this one is pref

Stronger vs weaker specs

- spec S2 is stronger than or equal to (>=) spec 1 if
    - S2 precondition <= S1
    - && S2 >= S1
    - stronger means generally more limitations on output, or more requirements on the inputs the implementation must accommodate
- the stronger, the smaller region of implementations it can use, so the find_OneOrMoreAnyIndex is weaker than find_OneOrMoreFirstIndex

Writing good specs

- make it coherent. make it about one thing
- make spec strong enough, but also weak enough
- make sure it uses abstract types (like List or Map) when possible
- precondition or no precondition? do we check to make sure it's been met before proceeding?
- you can add exceptions thrown to the postcondition

reading 8: Mutability and Immutability

- mutable - value of object can be changed (like the thing inside the circle can be changed)
    - Strings are immutable - you can reassign a string
    - StringBuilder is mutable.
- watch out for two variables pointing to the same object
- immutable types are safe from bugs
- Problems
    - passing mutable values
        - lists are mutable. So when you pass it (a mutable object) into a function u can change it. it’s a bad idea
    - returning mutable values is bad.
- instead,
    - do defensive copying
    - we assume mutations to input are disallowed
- iterators -

    - next() - returns the next element in the collection
    - hasNext() - tests whether the iterator has reached the end of collection
- class
    - fields (instance variables)
    - constructor - makes a new object instance and initializes the instance variables
    - this.var refers to instance object
- Useful immutable types
    - List.of, Set.of, etc
    - primitive types
    - immutablelist
    - you can also do List.copyOf for unmodifiable shallow copies
    - Collections.emptyList
- wrapped collections (even if they're immutable) can still be modified if you modify the thing inside it

reading 9: Avoiding Debugging

- static checking and dynamic checking, immutability, immutable references
- fail fast
    - checking preconditions is an example of defensive programming, by throwing unchecked exceptions, you prevent errors from propagating
    - assert (booleanexpression): message
        - method argument requirements, like preconditions
        - method return value requirements (like a self check)
        - write these while you write the code
        - don't let asserted expressions have side effects
- incremental development
    - unit testing
    - regression testing
- modularity and encapsulation
    - modularity - dividing a system into components, which can each be designed and implemented and tested separately
    - encapsulation - access control
        - variable scope

reading 10: Abstract variable types, access control

- private fields can be used by any code in the same class, even if its different instance of same class

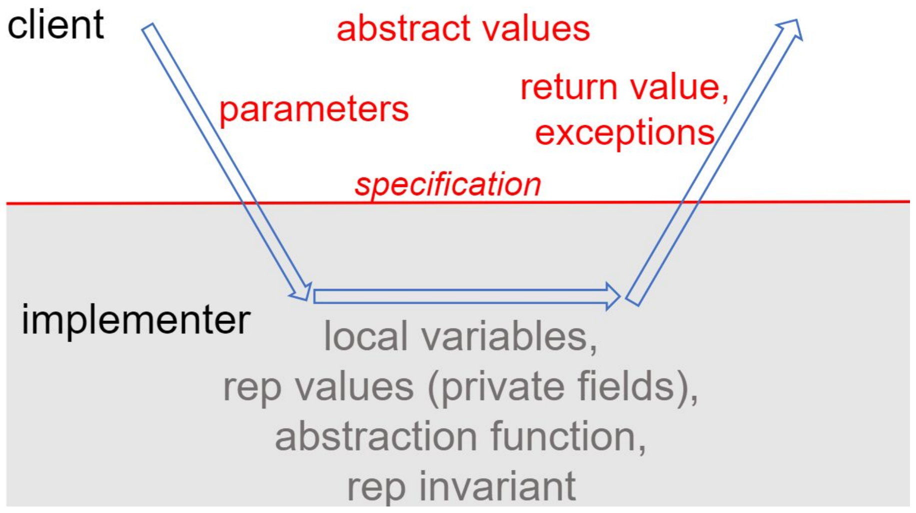
- if you're in another class and you're trying to access a private field of another object, oh no
- ideas
    - Abstraction - client only has to know pre and post conditions
    - Modularity - unit tests and specs make it a module
    - Encapsulation - local variables - only method itself can use it
    - Information hiding - some freedom for the implementer
    - Separation of concerns - if its responsible for just one concern
- abstract data types
    - **defined by operations you can perform on it**
    - determine mutability
    - **operations**
        - Creators create new objects of the type
            - constructors
                - **make sure that you're not passing mutable objects into the constructor when the user has a reference to it. consider copying the object in the constructor to get a clean copy**
            - or static method like List.of()
                - factory method
        - Producers create new objects from old objects of the type
        - Observers take objects of the abstract type and return another type of object
        - mutators change objects
            - signalled by void return
        - creator : t* → T
        - producer : T+, t* → T
        - observer : T+, t* → t
        - mutator : T+, t* → void | t | T
- designing abstract types
    - few simple operations that are coherent and adequate
    - representation independent (use is independent of implementation)
- spec - includes name of class, javadoc, specs of public methods/fields
- representation - fields (including private ones), any assumptions/reqs of fields
- implementation - method implementations

reading 11: Abstraction Functions & Rep Invariants

- invariants
    - ADT must uphold its own invariants. doesn't depend on clients
- immutability - threats to being able to change stuff within an object

    - **representation exposure**
        - like when you can just do obj.field = “iafjlesajf”; bc field wasn’t private
        - solution: just make it private final omg
    - reference leaking
        - like if you have an outside reference to something that's supposed to be private, uh hm that’s BAD
        - solution: just return a copy of the object or set the field as a copy of the object passed in, **defensive copying**
        - if any types are mutable, make sure impl doesn’t return direct references to its representation
- immutable wrappers
    - Collections.unmodiafiableList() returns a list w/o mutators
        - no compile-time checking
- some theory
    - rep values - values of the actual impl entities
    - abstract values - values that the type is designed to support
    - if this is supposed to represent a set of characters, we can use a number of different impls. Here we use a string.

    -
      ```
      public class CharSet {
          private String s;
          // Rep invariant:
          //    s contains no repeated characters
          // Abstraction function:
          //    AF(s) = {s[i] | 0 <= i < s.length()}
          ...
      }
      ```
        - note here that “Abc” and “Bac” both map to {a,b,c}
    - **abstraction function maps rep values->abstract values**
    - **rep invariant maps rep values -> boolean (**rep values in which AF is defined)
- main point - **choosing spaces for both rep and abstract. Also deciding which rep values are legal and how to interpret them as abstract values.**
- clients don’t really need to know about the specific representation.
- you can trace rep invariants to assertions that you run for every creator or mutator or producer
- you should still check implicitly or explicitly to make sure fields aren’t null
- **beneficent mutation**
    - u can change the rep or impl, so long as it retains the same rep values hashing to the same abstract values
- **Safety from rep exposure:**
    - Are fields private?
    - Are fields immutable?

    - If they aren't, how do we use defensive copying? Are mutable objects ever passed or returned from an operation?
- **specs for ADT**
    - only talk about things visible to client
    - 
client
abstract values
parameters
return value, exceptions
specification
implementer
local variables, rep values (private fields), abstraction function, rep invariant
    - these also help us reduce down the list of preconditions for each method spec
- **Rep invariant rules**
    - **established by creators, producers**
    - **preserved by mutators and observers**
    - **no rep exposure happens**

reading 12: Interfaces/Enums

- interface provides method signatures, but no method bodies, and then class implements interface
    - cant have constructors
    - circular references between classes and interfaces are legal in Java
    - make sure it's representation-independent
- subtype
    - If B is a subtype of A, then every B satisfies the spec for A
    - if B's spec is at least as strong as A
- introducing subtypes without breaking abstraction barrier: using a static method in the interface and returning a new subType(obj)
- generic Set<E> where E is any sort of type
    - when implementing factory methods, make sure you include an extra <E> at the beginning
- Enums
    - public enum EnumVar {}
    - var.equals(ENUMCONSTANT)

  - you can also have a constructor and methods
- @Override in 6.031 implies that we're keeping the same spec as the one in the interface
  - static factory creator in interface (must have the code body here) to return specific reps in the interface
  - implementations can be stronger than the interface

reading 13: Debugging

- reproduce the bug
- if you have a super large annoying input, then look at it at a smaller scale
- process
  - study the data
  - hypothesize
  - experiment
  - repeat
- slicing is a good technique for following all paths for where a bad variable could've been changed
  - hints could be immutability and unreassignablility, and helpful things are scope minimization
- delta debugging - defines the cause of a bug as a difference b/w successful execution and failing execution

reading 14: Recursive function - base cases + recursive steps

- some include helper methods for recursion, which satisfies a diff spec from the original recursive function.
- recursion is implementation specific. don't expose it.
- mutually recursive - A and B call each other
- Don't pass through mutable types! Make aDTs immutable by deleting mutators and replacing them w producers
  - so like returning the Sudoku object is a better solution, and then throwing an UnsolvableException if unsolvable

reading 15: Equality

- rep equality as
  - equality relation
  - AF
    - do they map to the same abstract value?
  - observational/behavioral
    - also depends on above, so ultimately everything depends on each other
- equivalence is a relation that is
  - reflexive, symmetric, transitive
- .equals() should always be an equivalence relation in which
  - t==t
  - if t==u then u==t
  - if t==u and u==v then t==v
- for abstraction function -> a==b if and only if AF(a) == AF(b).
- for observational -> every operation you apply to two objects results the same object
  - sets of operations follow the obs function def of equality if you can
    - determine difference b/w two objects that should be diff
    - calcs to be same for two objects that are the same
- == vs .equals()
  - == is referential equality, if their arrows point to the same bubble
  - .equals() - object equality
  -
  -
    ```
    @Override
    public boolean equals(Object that) {
        return that instanceof Duration && this.sameValue((Duration)that);
    }
    ```
-
  - instanceof is disallowed anywhere except for implementing equals
    - **check the runtime type of Object that!!**
    - **use public observers when checking equality**
  - x.equals(null) returns false.
  - hashCode must produce same result for two objects deemed equal by the equals method
    - if equals, then hashcode must be same
    - **converse is not necessarily true - if hashcode is equal, then it doesn't mean equals**
  - always override hashCode when you override equals
  - mutable types inherit equals and hashCode from Object (compare references)

- for immutable types, equals should compare abstract values -- hashcode maps abstract value to an int

reading 16: Recursive Data Types

- Immutable List - this can be shared and its safe
    - Empty, Cons are implementations of ImList
- recursive data type - two ADT's that cooperate with each other.
    - **datatype definition: ImList<E> = Empty + Cons(elt:E, rest:ImList<E>)**
        - abstract datatype on the left defined by its representation on the right
        - variants combined by a union operator
        - variants are like constructors w/ 0+ arguments, written w name:type as args
        - `Tree<E> = Empty + Node(e:E, left:Tree<E>, right:Tree<E>)`
    - When you write a recursive datatype, document it as a comment in the interface
    - union means that you can have one or the other or both, so you have to include some base case here, otherwise you'll just keep recursing down. like it can be made from one of two ways
- implementing operations over recursive datatypes:
    - declaring the operation in the interface
    - implementing the operation recursively in each concrete variant
- declared type is like List, whereas actual type is like ArrayList
- Writing program w/ ADTs
    - Writing an ADT
        - write spec - name, arg types, return types, exceptions, pre/postcondition
            - specs for operations of datatype
        - Test
            - write for each operation
        - Implement
            - choose rep
            - assert rep invariant (checkRep())
                - checkrep includes things also like field != null
            - implement operations
    - writing a program
        - ADTs and procedures
        - Choose datatypes
        - choose procedures
        - spec
        - test
        - implement
        - optimize
    - thoughts

        - choose an identity - a value in the datatype that represents nothing
        - factory methods

reading 17: REGEX

- grammars can distinguish between legal and illegal sequences, and can parse sequences into data structures (often recursive data types)
    - ex. a grammar for URLs, which specifies set of strings that are legal URLs
    - terminals - like ‘http’ or ‘:’
- grammars are defined by a set of productions, where each production defines a nonterminal
    - production has form of nonterminal = expression of terminals, nonterminal and operators
    - one of these nonterminals is a root, the rest are nodes
- Grammar operators
    - x::=y* -- x matches 0 or more y
    - x ::== yz -- x matches y followed by z
    - x ::= y|z -- x matches either y or z
    - x ::= y? -- x is y or empty
    - x ::= y+ -- x is one or more y
    - character class: x:= [aeiou]
    - inverted character class x:=[^a-c] everything except a-c
- regular grammar - substituting every nonterminal w/ right hand side
- java.util.regex.Pattern.compile(regex)
- recursive grammar happens when you get like A ::= A | B or nested

reading 18: Parsers

- parser generator - takes a grammar as input, generates a parser
    - produces a parse tree. each node expands into one production of the grammar
    - ah yes. This tree is a recursive data type.
- ParserLib
    - rule consists of a name followed by ::= followed by definition, terminated by semicolon.
    - nonterminals are names basically. terminals are quoted strings or regex expressions
    - what about whitespace? we could do a nonterminal but thats so much
        - @skip whitespace{
            - blah blah
        - }

        - whitepsace ::=[\t\r\n]+;
- using parserlib
    - import it
    - define an Enum that contains all the nonterminals used by your grammar
    - Parser.compile => Parser<SomeGrammar> parser
    - parser.parse creates ParseTree<SomeGrammar>
        - you can visualize the tree
- ParseTree - four methods (3 observers), 1 producer

Abstract Syntax Tree
- captures important features only, not specific implementation (as in concrete)
- it's a recursive data type

grammar file might be bad - UnableToParseException

leftRecursion:
- `sum ::= number | sum '+' number;`

  `number ::= [0-9]+;`
- oh no sum has to evaluate first before number and then nothing works

forming a parse tree -
1) start at root, try to parse as all those options
2) so like if you have some long thing and then try to parse into x (‘+’ x)*, then the first thing is x, and the rest of your long thing goes into the (‘+’ x)*
3) like a DFS of trying to parse as all the different things
4) parse trees - expressions evaluated first should be at the leaves

grammar => Parser => parser.parse(expression) => ParseTree -> AST using makeAbstractSyntaxTree - AST is like Plus(Number....

make AST’s immutable always

Final Exam 1 Review

- snapshot diagrams - remember final vs immutable
    - object vs primitive
- aliasing - dangerous for mutable inputs/outputs
- AF/RI
    - RI refers to only the class fields, not like the inputs into the constructor (which is the precondition). so like if your class fields have certain restrictions
    - AF(rep1, rep2...) = abstract value

    - RI - don’t put types of your rep, don’t put things that are already checked statically, null is assumed
        - like sorted? same length? same keys? no repeats?
        - not abstract values
- interfaces/enums
    - interface:
        - a list of method signatures
        - allows for diff reps of ADT
        - static factories
    - subtype - classes that implement interfaces
        - B is a subtype of A -> RepMap is a subtype of IntervalSet
    - enums
        - small finite set of values
- Recursion
- equality **
    - equivalence relations - reflexive, symmetric, transitive
    - Object Contract - equals() must satisfy all equivalence relations
        - hashcode() must be equal for .equals() objects, no requirement ow
    - @Override hashCode and equals
    - Observational vs behavioral equality
        -
- ADT
    - Creators, Producers, Observers, mutators (have on cheatsheet)
    - rep independence - rep is private
        - don’t talk about the rep in the spec
- specs
    - deterministic (one input produces a single output) vs underdetermined
    - declarative (talking about only the output) vs operational (describing each operation)
    - coherent - not broken spec
    - stronger vs weaker spec
        - MORE people can use it
        - precondition is weaker
        - postcondition is stronger
        - if both are weaker/both are stronger, then its incomparable
- writing partitions
    - range of possible values
        - like 0 to Integer.MAX_VALUE

Reading 20: Concurrency - multiple things happening at once

- shared memory - read/write w shared objects. like two threads using same object
- message passing - concurrent modules send messages back and forth. like web browser and server
- Processes, threads, time-slicing
    - **process -** an instance of a running program that is isolated from other processes on the same machine
        - process abstraction = virtual computer
        - they don't share memory between them.
    - **thread -** locus of control inside a running program
        - thread abstraction - virtual processor
        - place in the program that's being run plus stack of method calls above it
        - they share memory w other threads in the process
        - main() is like one thread
        - if there's not that many processors, we do **time-slicing** where we split up the existing threads into the processors
- **Starting a thread:**

    -
      ```
      new Thread(new Runnable() {
          public void run() {
              System.out.println("Hello from a thread!");
          }
      }).start();
      ```
- Anonymous classes - declares an unnamed class that implements an interface and immediately creates the one and only instance of that class
    - reduces scope
- so above, we use an anonymous Runnable()
- Interleaving
    - race condition - correctness of program depends on relative timing of events in concurrent computations A and B (think 6.004)
- concurrency bugs are not 100% reproducible
    - printing takes so long that it leaves a lot of room for threads to safely run while another one is printing. That's why printing doesn't help debugging.

Reading 21: Thread Safety

- Confinement
    - don't share variables between threads!

- Immutability
  - make shared variables unreassignable or immutable!
- Threadsafe data type
  - put shared data in an existing threadsafe data type
- Synchronization
  - keep threads from accessing shared vars or data at the same time
- a data type or static method - threadsafe means when used from multiple threads, it behaves correctly (satisfying its spec and preserving its rep invariant)
- **confinement**
  - local variables are always thread confined
  - static variables are not automatically thread confined
  - functions that seem thread-safe might not be if they reference objects that aren't threadsafe (like Hashmap)
  - you can't reassign variables in the Runnable if they're final!!
- **Immutability**
  - final variables are safe to access from multiple threads because you can only read the variable, not write it. and make sure it's immutable.
  - stronger def of immutability
    - no mutator methods
    - all fields are private and final
    - no rep exposure
    - no mutation of mutable objects, not even beneficent mutation
- **using threadsafe data types**
  - ex. stringBuffer is threadsafe whereas StringBuilder isn't, but stringBuilder is slower
  - threadsafe collection wrappers
    - atomic operations
- **synchronization - preventing threads from accessing the shared data at the same time**
- Safety ARgument
  - immutability of data type
    - don't forget we might assume that maps are mutated somewhere else even if they're private final
- **serializability -** for any set of operations executed concurrently, result must be a result given by some sequential ordering of those operations (atomic operations)
- thread safety argument always depends on SRE argument
- make sure class fields are final too in addition to immutable

Reading 22: Locks and SYNCHRONIZATION

- locks - *acquire* (take ownership of a lock) and *release* (relinquish ownership of the lock)

- block - thread waits until an event occurs
- be careful of deadlock - when concurrent modules are stuck waiting for each other to do something (cycle of dependencies)
    - when you already own a lock, everything inside can acquire it too. (Reentrant lock)
- making locks
    - Object lock = new Object();
    - synchronized (lock) {
        - stuff u wanna do
    - }
    - be wary of just assuming this works - make sure you guard every access w/ the synchronized block
    - make sure you use the same lock!
- monitor pattern - monitor is a class whose methods are mutually exclusive, so only one thread can be inside an instance of the class at a time (wrap everything in synchronized (this) or public **synchronized** void delete()...
    - constructors are already expected to be confined to single thread
- locking discipline
    - every shared mutable variable must be guarded by some lock
    - an invariant that involves multiple shared mutable variables must be guarded by the same lock
- be careful that any runnables you return, which can access the enclosing object

Reading 23: Queues and Message - Passing

- message passing - concurrent modules send immutable messages to each other over a communication channel
- **blocking methods -** call to the method can block, waiting for some event to occur before returning
- **Blocking queue -** a queue that has operations where it waits for the queue to become non-empty when retrieving an element
    - put(e) blocks until it can add element e to the end of the queue (like some queue cap size)
    - take() blocks until it can remove and return the element at head, waiting until non-empty
- **producer-consumer design pattern -** producers put data or requests onto queue, consumers remove and process
- you can use Thread.interrupted()

Reading 24: Sockets and Networking

- Client/server design pattern
- IP address - network interface
- hostname - names that can be translated into IP addresses. one name can map to diff IPs. multiple names can map to the same IP.
- port numbers - server processes bind to particular ports. now the server listens on the port.
- sockets
    - listening socket - used by server process to wait for connections from remote clients
    - connected socket - send/receive messages to and from process on other end of connection
    - local -Socket(hostname u wanna connect to, port)
        - socket.getOutputStream and socket.getInputStream
        - while(true) {
            - String message = reader.readLine() [WAITS]
            - writeToServer.println(message) [WAITS FOR LOCAL BUFFER TO FILL]
            - **writeToServer.flush()**
            - String reply = freadFromServer.readline() [WAITS FOR SERVER]
            - if (reply==null) break; [if server closed cnxn]
        - close bufferReader/Writers, socket
        - poison pill - “quit”
    - ServerSocket(port) - listeninglistening
        - Socket socket = serverSocket.accept() [WAITS FOR A CLIENT TO COME]
- buffer - data arrives and puts in in here
- byte stream - data going in or out of a socket. InputStream or OutputStream
    - more often, Reader and Writers
    - exhibits blocking behavior
        - Is the incoming socket buffer empty? calls read until data is available
- Wire protocols
    - **protocol - set of messages that can be exchanged by two communicating parties**
    - **wire protocol - a set of messages rep’d as byte sequences**
    - designing a wire protocol
        - keep # of diff messages small
        - each should have coherent and well-defined behavior
        - adequate for clients to make requests they need

  - serialization - process of transforming data structures in memory to like JSON or something

Reading 25: Callbacks

- attach a listener (Listener pattern) - they do something when something changes
  - create an instance of an anonymous class
  - pattern - source *publishes* stream of events and listeners register (subscribe) to stream.
  - example - JButton is event source, events are button presses, listener is ActionListener, function called when event happens is actionPerformed
- **callback -** a function that a client provides to a module for the module to call
- HttpServer = HttpServer.create(new socket)
- server.createContext("/play/", new handler() {
  - public void handle(){}
  - }
  - handle here is a callback function that is called anytime an incoming request starts w/ "play"
- first class values are things that can be passed thru to functions or returned
  - like Runnable objects, ActionListener, HttpHandler, lambda expressions
  - functional objects - an object whose purpose is to represent a function
  - lambda expressions or method refs can create functional objects in Java
- lambda - (args) -> {do something;}

Reading 26: Map "Filter and Reduce

- big idea - functions as first-class data values, meaning that they can be stored in variables, passed as args to fxns and created dynamically
- Iterator abstracts away what kind of iterable you're dealing with
- stream - a sequence of elements E
  - .map(arg - something(arg))
  - .map(function as a method reference)
    - like Math::sqrt (class::method name)
    - (Math::sqrt).apply(number)
  - this is a function object
- or you can use .forEach
- .filter(Character::isLetter) needs a bool
  -
- .reduce(init, f)
  - like list.reduce(0,(x,y) -> x+y)

    - you can fold-left or fold right in python
    - we can reduce to another type
        - accumulator (growing result R adds in another E)
        - combiner (combines two partial results R and R)
    - be careful when init = 0, that is the first value in the computation.
- higher order functions - functions that take other functions or return other functions.
- you can do stream.parallel or collection.parallelStream

Reading 27: Little Languages I

- representing code as data
    - so that it does not evaluate immediately, it can exist and be modified as a **first-class value.**
    - like the Function<T,T> we can pass it around and it wont do the evaluate until we call f.apply(a,b)
    - functional objects w/ compact syntax Function f = lambda exp
- Composite pattern - music is an object , where its consisting of single objects and groups of objects (composites)
- emptiness - rest of duration 0

```
Music = Note(duration:double, pitch:Pitch, instrument:Instrument)
        + Rest(duration:double)
        + Concat(m1:Music, m2:Music)
```

Reading 28: Little Languages II

- Visitor Pattern
    - treating functions as first-class values and an alternative impl strategy for operations on recursive data types
- dynamic dispatch - for example, at declaration, something might be a formula, but at runtime it could be a Not. Actual type must always be a subtype.
- Interpreter pattern problem - complicated, the operation is distributed recursively thru all classes
    - adding a operation is hard
- so how to implement a function on a recursive type?
    - double dispatch - first method call uses dynamic dispatch to reach a concrete variant
        - variant then calls to the object representing recursive function

    - so for example, you implement a callFunction that takes in a FormulaFunction (which is an interface, and you can have things like VariablesInFormula as a class), which then has diff overrides for each of the concrete class types.
- this is the visitor pattern - create an instance of some Function Object, pass it around.

What's the reason for not being threadsafe???

- fields are not private and final
- fields are not the synchronized version of their data type
- methods can interleaved and aren’t synchronized (Monitor pattern)