



L1: Course Overview

1.13.25

Programming Paradigm: Programming art style

e.g. OOP, functional, imperative, ...
* ones covered in slides *

Imperative Programming: explicitly
write out control-flow with gotos
* used in assembly

Structured Programming: imperative programming
where control-flow is defined by
nested loops, conditionals, subroutines

Object-Oriented Programming: organizing
code abstractly in classes and objects

Declarative Programming: implicit
control flow - state what we
want and not how to obtain it

Functional Programming: declarative pro.
expressed by combining function calls

* can use any paradigm in any language

Pure language: language that implements
a paradigm 100% - rare

Date. / /

L2: JavaScript - 1

1.15.25

* On Exam: Mantra of this class

- Ask questions!
- Ask early!
- Ask often!

JavaScript:

created: 1995

creator: Brendan Eich

recognized for:

- first-class functions
- weak typing
- prototypes
- automatic semicolon comp

uses:

- web applications
- asynchronous servers

"the assembly language of the web"

Paradigms

- imperative
- OOP
- procedural
- declarative
- functional

* graph of these
in slides

note, OOP not through classes,
instead w/ OOLA model

Compiled Languages :

source code → compilation process → executable code

↪ very complicated process - covered in compilers & also in slides

Interpreted Languages :

source code → read 1st instruction → execute

↪ read 2nd instruction → execute

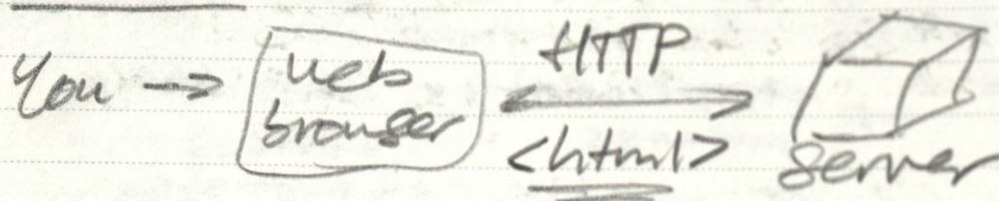
↪ read nth instruction → execute → end

Javascript : interpreted

- need a Javascript engine to run
→ a part of all modern web browsers
- programs are called scripts

Date. / /

Web Dev :



<html>

<head>

→ metadata

</head>

<body>

→ content

</body>

</html>

to attach JS to the web...

<script>

// code

</script>

inside of <head>

alert("message"); // sends message
to web-browser

note.

Scripts in the <head> tag are loaded first on startup of a web page
• slows down loading time

Can place less crucial scripts in <body> to delay loading until they are needed

Can also link JS files instead of writing all JS directly in HTML =

```
<script src="file.js"></script>
```

JS files do not need to be stored on local machine — can get them from HTTP

```
<script src="http://cdn...">
```

Can also write JS straight to the console:

• need an interpreter: will use

node.js {
 → REPL (read-evaluate-print loop)
 → Run standalone JS code

REPL:

```
$ node   

> console.log('Hello world')
```

Standalone JS:

```
$ node file.js
```

note, can access global objects on JS on web browsers but cannot w/ standalone JS

ex. alert() only works on web browser

node.js: instead has modules

```
require('module')
```

fs: filesystem

readline: I/O

Date. / /

Concept: Syntax vs Semantics

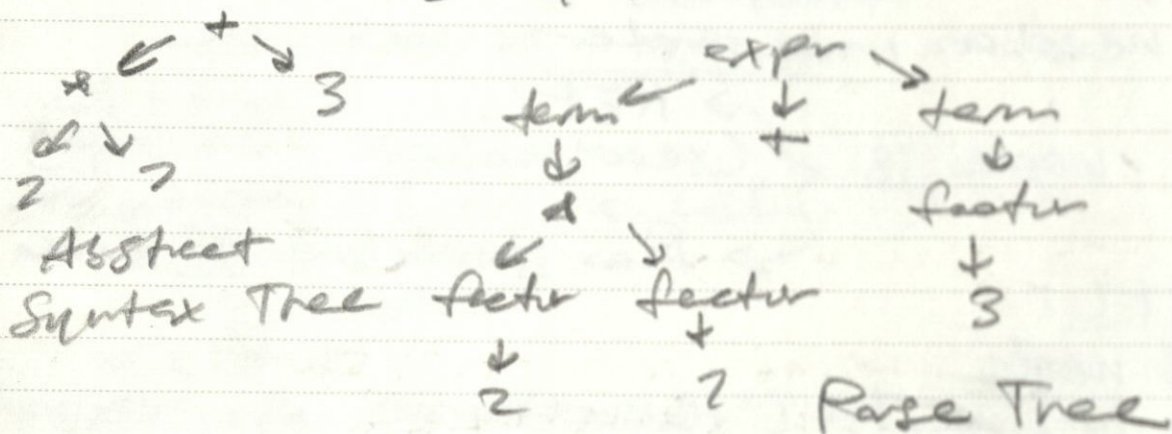
Syntax: structure of program

✓ "A red car"
✗ "A car red"

grammar of a programming language defines the valid set of instructions and how to put them together

parser: Source code $\rightarrow$ AST / Parse tree

$2 + 7 + 3$



Semantics: behavior

language with no meaning

"colorless green ideas

sleep furiously" ???

Lecture Notes:

JavaScript: semicolons indicate different statements

Statement Blocks

```
{  
  ...  
}
```

Code comments:

// line comment

/* block

*/ comment

* Very similar syntax to C

Declaring Variables: you don't set data type at declaration

```
const user1 = "Joe";
```

```
var $user2 = "Joe";
```

```
let user-3 = "John";
```

good coding practice: camel Case

ex. myVar

note, vars are case-sensitive

Date. / /

Variable Scope :

Global Scope : variables declared outside a function

Local Scope : variables declared inside a function and only visible inside that function

Block Scope : variables declared inside a block eg. if, for, while

Declarations :

redeclared? reassigned? scope?

var	✓	✓	global function
-----	---	---	-----------------

const	X	X	all
-------	---	---	-----

let	X	✓	all
-----	---	---	-----

var a;	// declaration
a = 100	// assignment

Reference error : attempting to access a var outside of its scope

Variable hoisting:

declaration $\neq$ initialization

declarations can be moved up scope

```

x = 5;
console.log(x);
var x;
  
```

← this works!

ex.

```

x = 5;
console.log(x);
console.log(y);
var x;
var y = 7;
  
```

← var x;
← var y;

prints

x

undefined

} declarations are
hoisted but
initializations are not!

Variable shadowing: if a variable declared in an inner scope has the same name as a variable in an outer scope:

inner scope will shadow outer scope and take precedence

ReferenceError vs Undefined:

```

let a1;
console.log(a1) // undefined
console.log(a2) // ReferenceError
  
```


Lecture Notes:

10.27.25

JS { Primitives (immutable)
Objects

Primitives:Boolean type

let var = true; // can be true or false
let isGreater = 4 > 1; // assigning boolean val

Number type: has a min/max range

let x = 135; // integer
let y = 12.345 // float

let a = 1/0; // infinity
let b = -1/0; // -infinity
let c = "string"/2; // NaN
"not a number"

BigInt:

const x = 12345678912345n;

↑
"n" indicates BigInt

Strings:

let str = "Hello";
let str2 = 'Mark';

let phrase1 = `can embed another \${str3}`;
let phrase2 = 'Hello, \${1+23}';

Null type: (absence of a value)

let age = null; // value unknown / empty

Undefined type: (absence of an object)

let name; // value not assigned

Symbol type: (used to create unique property keys)

```
const sym1 = Symbol("id");
const sym2 = Symbol("id");
```

note. sym 2 is different than sym 1

Objects: (mutable)

```
let car = {
  make: "Toyota",
  model: "Corolla",
  color: "white"
};
car.color = "green";
```

key (value pairs) ←

let a = {} // still an object?

note. JSON = JavaScript Object Notation

Checking Data Types:

typeof Unary Expression

↪ operator — not a function

Ex. `typeof 0` // "number"
`typeof 10n` // "bigint"
`typeof []` // "object"
`typeof console.log` // "function"

note, `typeof console.log(1)`

→ prints 1 in terminal

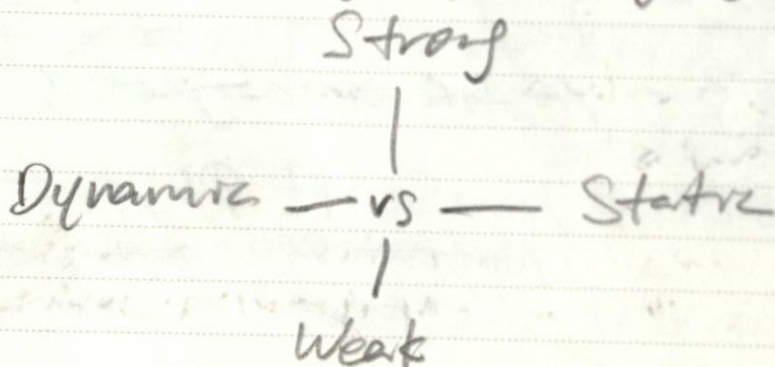
→ returns an undefined type

→ `typeof undefined` is "undefined"

`typeof null` // "object"

`typeof NaN` // "number"

Weakly vs Strongly typed languages and
Dynamic vs Strong typed languages



Conversions b/w unrelated types:

implicit: weakly typed

explicit: strongly typed

note. this is in reference to attempting operations b/w different languagesWhen is type deducing done?

compile time: static typing

runtime: dynamic typing

JS: weak and dynamic

C: strong and static

Python: strong and dynamic

JS exs. "Hw" + 1 = "Hw1"

"3" - 1 = 2

"a" - 1 = NaN

 $3 - ("2" + 1) = 3 - "21" = -18$ * table of values as numbers
in slides *notablySymbol('dog') → throws Type Error
any object x → result of x.valueOf()

Logical Operators:

OR || AND && NOT !

note. these can be applied to values of any type - not just booleanthe result can also be of any type"Truthy / Falsy" values:

non-boolean values implicitly converted into a boolean value

false

0 (number)

"" or "" (empty string)

null

undefined

NaN

all falsy

all other values are truthy

exs. $1 || 0 = 1$ $null || 1 = 1$ $null || 0 || 1 = 1$

first truthy value

 $undefined || null || 0 = 0$ ←

last falsy value

a || b:

1. first truthy value
2. last falsy value

a && b:

1. first falsy value
2. last truthy value

!a: either True or False! $\emptyset$ = True

!! "string" = True

!! null = False

double negation turns arbitrary type
to boolean value

Null Coalescing Operator (??):a ?? b:

- if a not null and not undefined, then a
- if a null or undefined, then b

ex. let user;

```
console.log(user ?? "Anonymous");
// "Anonymous"
```


Conditional Structures : same syntax as C

- switch
- if / else

note. (expression) ? (expr-if-true) :
(expr-if-false);

ex. let canDrink = age >= 21 ? "Y" : "N";
ie. ternary operator

note. all expressions can be boolean or
will be truthy / falsy

Loops : same syntax as C

Breaking out of Loops:

break; breaks out of a loop like in C

can also break out of a target:

```
outer : for (; i <= 3; i++) {  
    for (j = 0; j <= 5; j++) {  
        // code  
        break outer;  
    }  
}
```

3

breaks the entire section of code

Lecture Notes:

1.29.25

JavaScript is a weakly typed language:
 == vs ===

let a = 5 == "5"; // true

↗
 compares values of primitives
 converts second value to first

let b = false == 0; // true

let c = null == undefined; // true

let d = true == 1; // true

note, converts to "truthy" value

note, this is called a "loose" equality

let e = 5 === "5"; // false

↗
 "strict" equality
 does not do type conversion
 for you

special case: follows IEEE 754

let i = NaN === NaN; // false

let j = NaN == NaN; // false

not a quantifiable entity,
 just a flag - cannot be compared

Objects : comparison done by reference - different behavior

let $o = [] == []$; // false
mem addr: 0x1 0x2

* objects must be in the same location in memory to be true

note. empty objects are treated as true

let $k = \{name: 'Mark'\} = \{name: 'Mark'\}$

↑ equates to false

↓ let $j = \lfloor j \rfloor = \{ j \}$,

Javascript Functions:

first-class functions: functions are treated like any other variable

can be:

- passed as argument
- returned by another function
- assigned as a value to a var

JS has first-class functions

Exs.

1.

```
let add = function(a, b) {  
  return a + b;  
}  
console.log(add(1, 2));
```
2.

```
function subtract(a, b) {  
  return a - b;  
}  
console.log('I subtracted(2, 1)');
```
3.

```
let isEven = (n) => n % 2 === 0;
```

args
↑
arrow
expression
↑
output

```
console.log(isEven(4));  
  
let isEven = (n) => {  
  return n % 2 === 0;  
};
```
4. Callback functions: function that takes another function as an input

```
function calculate(operation, a, b) {  
  return operation(a, b);  
}  
console.log(calculate(add, 2, 1));
```

uses the add() function
for operation()

Function Hoisting:

a();
b();
c();

function a() {
...
}

function
declaration

let b = function () {
...
}

function
expression

var c = () => ...;

arrow
expression

note, ONLY function declarations
have function hoisting

note, c2 and c3 will give different
errors

- All var declarations are hoisted
So var c₂ is hoisted but
not defined
- let b; is never hoisted at all

Imperative Code: explicitly spell each step of how you want something done

Declarative Code: you only say what it is you want done

seen in `live-array.js` in `github` repo

`Array.every(callback, thisArg*)`

returns `Boolean` value if every element in an array passes a test in the callback function

`Array.filter(callback, thisArg*)`

creates a new array with all elements that pass the test in the callback function

`Array.map(callback, thisArg*)`

creates a new array populated with the result of the callback function for each element in an array

* optional argument

Date. / /

Lecture Notes: imperative vs. declarative programming 2.3.25

Ex. let people = [
 { id: 20, name: 'Peter' },
 ;
];

```
function extractIds_map(array) {  
  let callback = (a) => a.id;  
  return array.map(callback);  
}
```

let output = extractIds_map(people)

* more examples of imperative vs declarative challenges in live array.js &

note, when using + between a number and a string, the string takes precedence

Recall

Imperative

Procedural

Global Data



F1 F2 F3

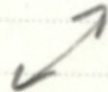
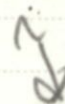
Local Data

Object Oriented

Object 1 Object 2

Data ↔

methods



Object 3

Class-Based OOP: classes are blueprints to make objects

ie. classes as a container to fill object info in

Prototype-Based OOP: create new objects based on existing objects (prototype objects)

ie. build objects on top of objects

JavaScript: uses Prototype-Based OOP!

Objects: let user = { name: "Mark", ... };
ie. key: value pairs

```
console.log(user.name);  
console.log(user["name"]);
```

// can dynamically add properties
user.profession = "Engineer";

// object constructor
let user1 = new Object();

```
let user2 = {  
  name: "Amy",  
  birth_year: 1981,  
  age: function () {  
    return new Date().getFullYear() - this.birth_year;  
  }  
};
```

} properties
} method

Date. / /

this: special expression referring to receiver object

→ same as self for python

ex. console.log(user.age());

receiver
object

method

ie, this is a placeholder for receiver object

shorter syntax:

let user4 = {

age() {

...

}

no need for:

age: function () {

...

}

Lecture Notes:

2.5.25

Object.create(p) in JS

- creates a new object using an existing object p as the prototype

let o1 = {x: 1}

↖ let o2 = Object.create(o1)

o2.y = 2;

↖ let o3 = Object.create(o2)

o1 has x → x is inherited
 o2 has x and y → y is its own
 o3 has x and y

console.log(o2.x) = 1

note: changes in var propagates
 down but not up

accessing attributes that do not
 exist returns undefined

Prototypal Instantiation :

note. every function in JS is an object
and has two attributes

— length

— prototype

constructor

ex. function Snger (n, birth-year) {

 this.name = n;

 this.birth-year = birth-year;

}

let S = new Snger ("Name", year);

adding new methods :

Snger.prototype.age = function () {

};

length = # of arguments

Prototype Chain :

Snger

function
length : 2
prototype

super1 → { name : " " ,
birth-year : }

constructor
age : { }

function()

Alternative Syntax:Prototypical Instantiation:

```
function Singer(name, birth-year) {
  this.name = name;
  this.birth-year = birth-year;
}
```

```
Singer.prototype.age = function() {
  return new Date().getFullYear() - this.birth-year;
};
```

Syntactic Sugar: class keyword

```
class Singer {
  constructor(name, birth-year) {
    this.name = name;
    this.birth-year = birth-year;
  }
}
```

```
  age() {
    ...
  }
}
```

- these are both just prototypes

- to create objects from these:

```
object = new Singer(...);
```

note. Classes start with uppercase char
objects start with lowercase

Data Structures in JS:

Arrays: good random access

Queues: good ordered access (FIFO)

Stacks: good ordered access (LIFO)

Sets: good to check for element existence

Maps: good for retrieval based on keys

Sets:

```
const s = new Set()
```

```
s.add(1)
```

```
s.has(1) // true or false
```

```
s.delete(1)
```

```
console.log(s.size)
```

```
s.forEach(x => console.log(x))
```

// iterates through s

note, can do these operations on any data type or object

note, has method uses === under the hood, remember objects are compared based on their location in memory

Maps: const map1 = new Map();

```
map1.set('key', value); // add or update
```

```
let val = map1.get('key');
```

```
map1.delete('key');
```

```
console.log(map1.size);
```

Queues and Stacks:

```
const queue = []    const stack = []
```

OR
new Array()

note, same architecture here

```
queue.push()
```

```
stack.push()
```

```
let element = queue.shift() // pops FIFO
```

```
let element = stack.pop() // pops LIFO
```

```
console.log(queue.length) // or stack.length
```

note, same w/ stacks

```
console.log(queue.length === 0);
```

checks if empty

```
queue.length = 0 // removes all elements
```

```
queue.length = 1 // adds elements
```

```
// [ < 1 empty item > ]
```

```
let item = queue.shift() // undefined
```

see ex graph-methods.js

Lecture Notes:

2.10.25

Parameters: the 'slots' / placeholders for inputs a function should receive

Arguments: specific values we give to parameters when we call a function

Arguments in JS:

- fully evaluated before f'n call and values assigned to parameters
- extra arguments are ignored
- too few arguments are
 - undefined
 - refers to f'n defaults

Pack / Unpack: using "..." on the last parameter of a function

```
const args = [1, 2]
```

```
console.log(f(...args))
```

```
function f(x, y) { return [x, y] }
```

→ outputs [1, 2]

```
function g(x, ...y) { return [x, y] }
```

```
console.log(g(1, 2, 3)) // [1, [2, 3]]
```

```
console.log(g(1, 2)) // [1, [2]]
```

Default Values: Similar to python

function h(x, y=0) { return [x, y] }

Parameter Destructuring:

function z({x, y} = {}) { return [x, y] }

console.log(z({x: 70, y: 175}));

- receives attributes of objects or uses an empty object

Ex. function f(a, b, c) {

a = 300;

b[1] = 400;

c += 2;

}

function g() {

const x = [1, 2, 3];

const y = [4, 5, 6];

const z = 10;

f(x, y, z);

console.log(x); // [1, 2, 3]

console.log(y); // [4, 400, 6]

console.log(z); // 10

→ only value modified by reference

- all values passed by value
- arrays are only passed by reference when modified w/ indexing

Free Variables:

```
function second() {
  console.log(x);
}
```

• var used but not declared anywhere
a function as free var

Dynamically Scoped: languages that use
caller's values for free vars

Statically Scoped: languages that look
outward to textually enclosing regions

& JS is Statically Scoped &

Closure: combination of a function and
the references to the f's surrounding
state (i.e. lexical environment)

i.e. looks for free var working up in scope
block → function → global

Shallow Binding: binds free vars after passing
them to f'n

Deep Binding: binds free vars where
the function is declared

& JS is deep binding &

dynamic scope $\leftrightarrow$ shallow binding

static scope $\leftrightarrow$ deep binding

* more exs in L8 slides *

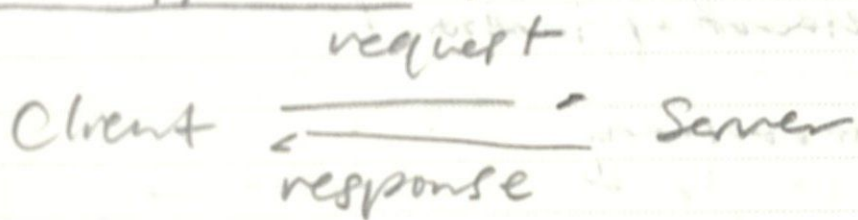
(slides 137-140)

* ex of closure in live_closure.js *

note, assigning generator to vars assign
distinct instances of the generator

Front-End Web Development:

Web Application:



works over a protocol (HTTP)

GET / HTTP/1.1

Host: www.example.com

User-Agent: AppleTV11, ...

Accept: text/html

Internet: medium to facilitate two
interactions

Date. / /

Front-End: how things look

- images, content, structure
- HTML, CSS, JS, TypeScript

Back-End: how things work

- logic, data
- Ruby, Python, PHP, Java, JS, ...

FE Challenges: UX, responsive design

BE Challenges: Performance, Scalability, Availability, Optimization, Security

HTML, CSS, JS:

HTML: Structure of application (content)

CSS: Styling / themes

JS: user interactivity

Lecture Notes:

2.12.25

see FrontEnd/example_600 for
website demo

open index.html opens the website

```

<html>
  <head>
    <title> Demo </title>
  </head>
  <body>
    <h1> Header </h1>
    <p> Paragraph >
      <a href = "link" target = "blank">
        None
      </a>
    </p>
    <img src = "location" width = "100px"
    ... alt = "description" >
  
```

→ can be link or file

Common HTML elements:

- p
- div
- a
- img

Common HTML attributes:

- id
- class

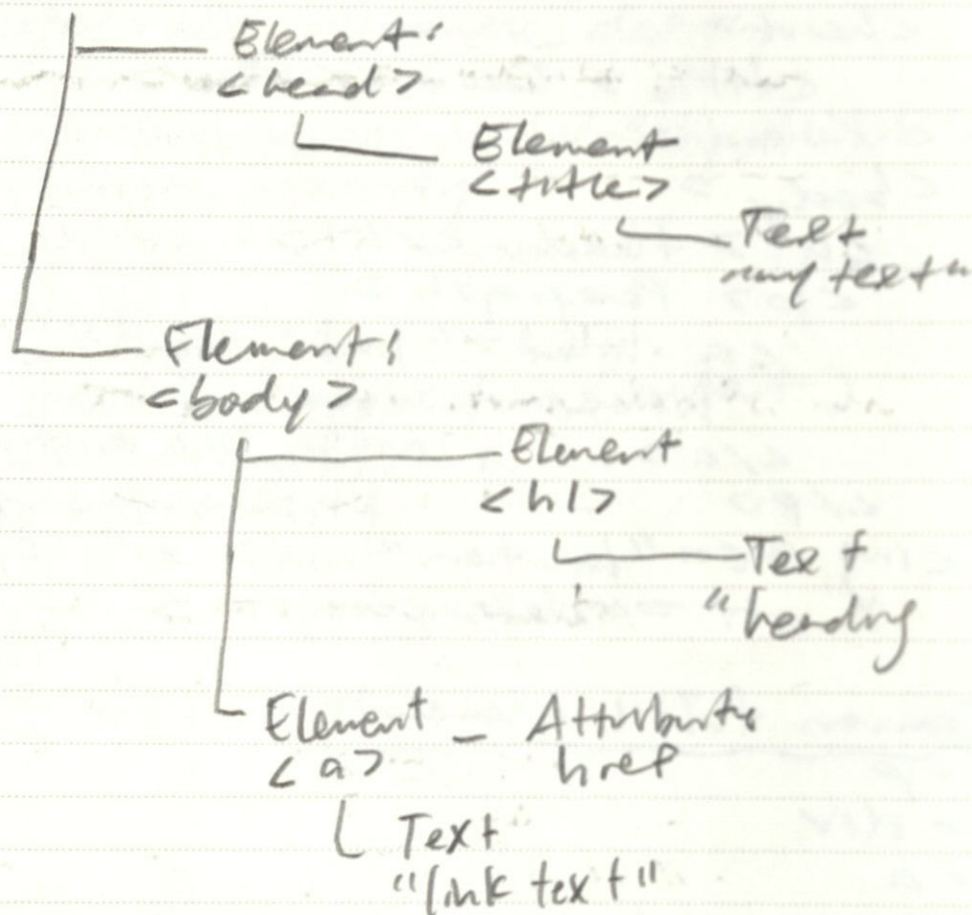
Date. / /

JavaScript runs in the background to add interactivity to your website

DOM: Document Object Model
represented as a tree:

document

↳ Root element
<html>



DOM API:

document.getElementsByTagName("h1")
document.getElementsByClassName("class-name")
document.getElementById("myId").innerHTML="content"

& more in API documentation &

Ex of JS in FrontEnd / Example use A

head >

```
<script type="text/javascript">
  alert("hello world");
</script>
```

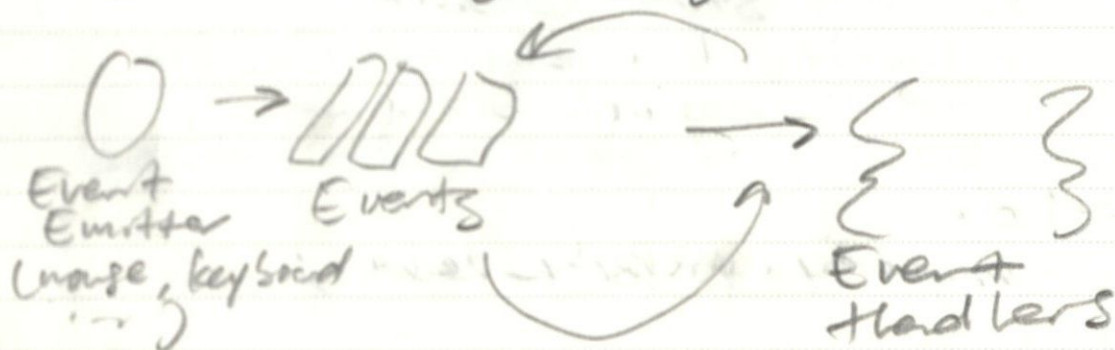
note, html is parsed and read from
top to bottom

so move scripts to bottom of file

document.getElementsByTagName()
 returns a collection of elements

- must iterate over all of them to edit every element
- if only 1 element must access `tag[0]`

Event Driven Programming : Paradigm



flow of program determined by events

Lecture Notes:

2.17.25

Closure is a function combined with a lexical environment — references a variable

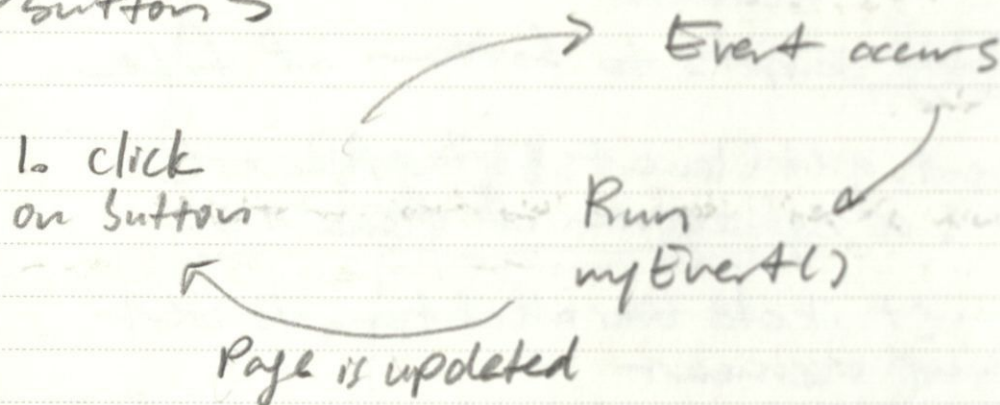
note, var inside blocks have global scope

Event-Driven Programming:

```
<button onclick = "myEvent()">
```

Click Me!

```
</button>
```



* ex of this in /FrontEnd/example_weather

CSS: Cascading Style Sheets

- used to modify how elements are displayed in the web browser

```

selector {
  property: property-value;
}
  
```

declaration

How to use CSS:

Option 1: inline

```
<h1 style="color: blue;" > ... </h1>
```

Option 2: internal

```
<head>
```

```
  <style>
```

```
    h1 { color: blue; }
```

```
  </style>
```

```
</head>
```

note web pages fail silently ... things
will just not work with no errors shown

Option 3: external

```
<link rel="stylesheet" href="a-path-to  
file">
```

add this to head

and then use properties in this css file

Date. / /

Responsive Design: adapts webpage to fit devices of all sizes

use media queries:

```
@media screen and (min-width: 1024px) {  
  // add properties here  
}
```

Bootstrap: common styling library

- makes responsive design much easier

ex in example_src/index_with_bootstrap.html

use getbootstrap.com and go to Docs/Quick Start

2.24.25

Lecture Notes:Python Supports:

- procedural
- imperative
- OOP
- declarative
- functional

Python is interpreted:

converts into byte code instead of machine code

- .pyc .pyo
- interpreted by python virtual machine
- stored in -- bytecode --

Can run in 2 ways: REPL or standalone script

Source Code Encoding:

- default is UTF-8

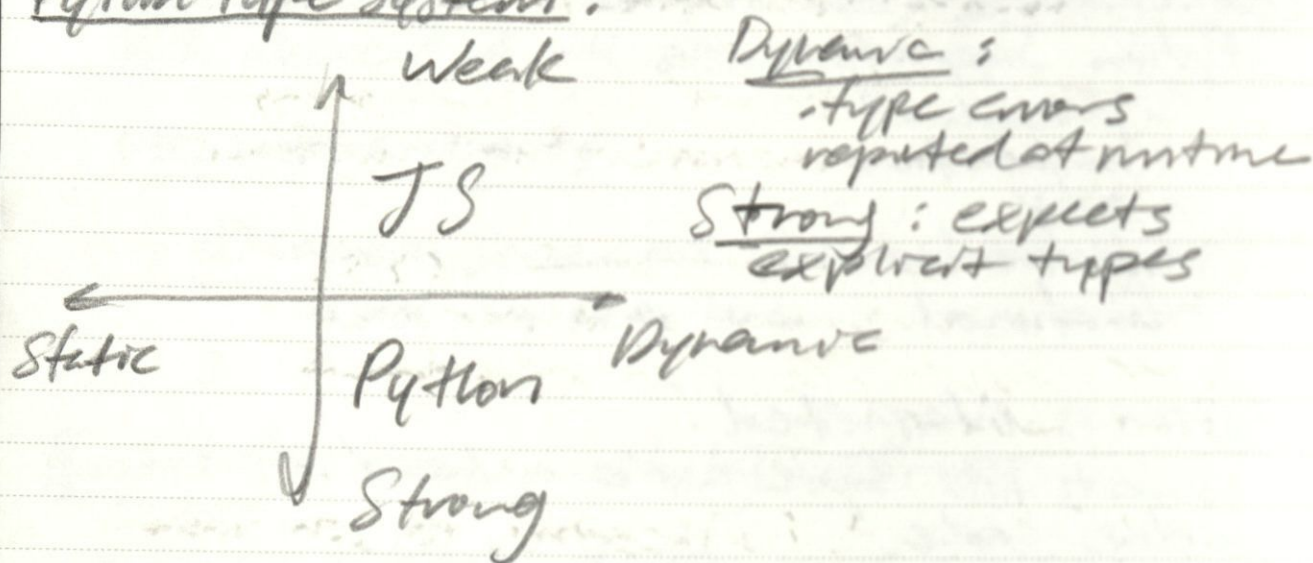
cardage with:

-*- coding: utf-8 -*-

↳ or ascii etc.

Python Basics:

1. all types are objects
2. can create your own objects
3. no parentheses!
4. has subtypes and supertypes

Python Type System:Type Annotations:

```
def fn (name: str) -> str:
    return
```

↑
↑

can specify expected data types

Types: all types are subtypes of the object type

Scope: Python only has

- global scope
- function scope

although scopes can be nested!

global keyword: directive to the parser

allocates var to global scope during "compilation" instead of at runtime

Shadowing: vars inside inner scopes with the same name as a var in an outer scope will take precedence of its values

nonlocal keyword: directive to parser

allows inner functions to access and modify vars defined in their enclosing functions

Name Error: using a global var before it is first assigned

Unbound Local Error: using a local var before it is first written

* Logically the same thing but about different scopes

Arguments: need to pass as many as specified

- not like JS
- unless defaults are specified, then can have less

Date. / /

Positional vs. Keyword args: positional parameters must come before keyword parameters

can call function w/ parameters positionally
`func(1, 2, 3)`

or by keyword:

`func(x=1, y=3, z=2)`

Parameter Association: (*)

`def f(x, y=1000, *z)`

packs any additional arguments into a tuple

or it used like this, it unpacks the collection:

`f(*[i for i in range(6)])`

`# x=0, y=1, z=(2, 3, 4, 5)`

Parameter Overview:

- Positional
- Arbitrary args (*args $\rightarrow$ packed into tuple)
- Keyword args (**kwargs $\rightarrow$ packed into dict)

A many of these in python directory

`def f(var, *var2, **var3)`
0 1 2 stars

Pass by value or by reference?

let's see! pass by sharing

i.e. passes arguments to parameters
by passing each arguments id

id $\rightarrow$ unique identifier of object

i.e. has a copy of the "address" of var (id)
 $\rightarrow$ immutable types create new copies
 $\rightarrow$ mutable types change original var

Lecture Notes:

2.26.25

Pass by Sharing:

immutable types - abstraction/references
of pass by value

mutable types - abstraction/references
of pass by reference

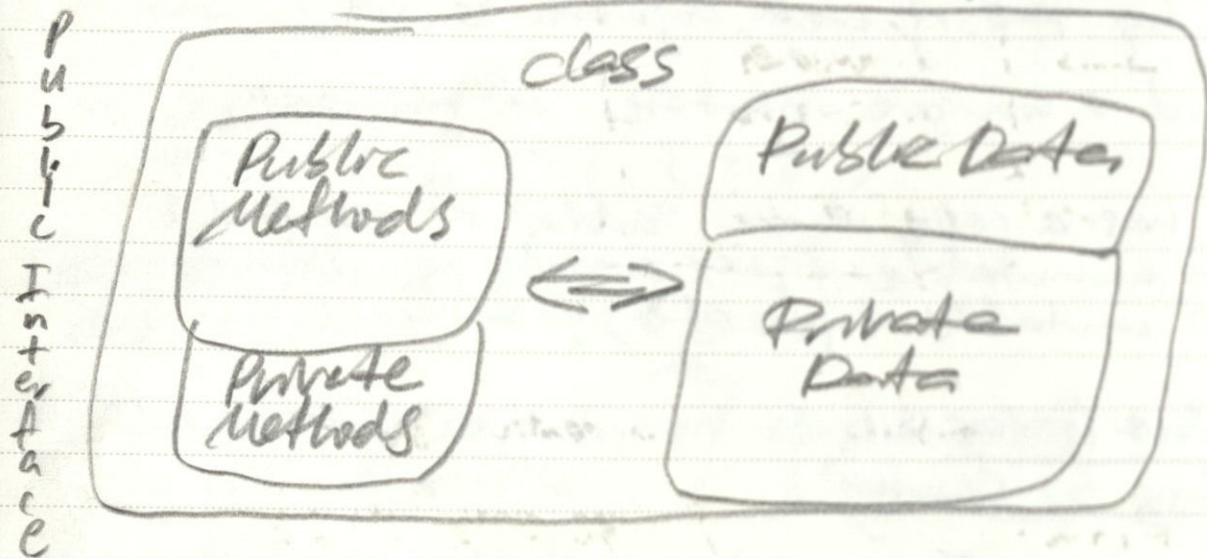
note, python default args are evaluated
once when function is defined

see ex in L12-15 slide 32

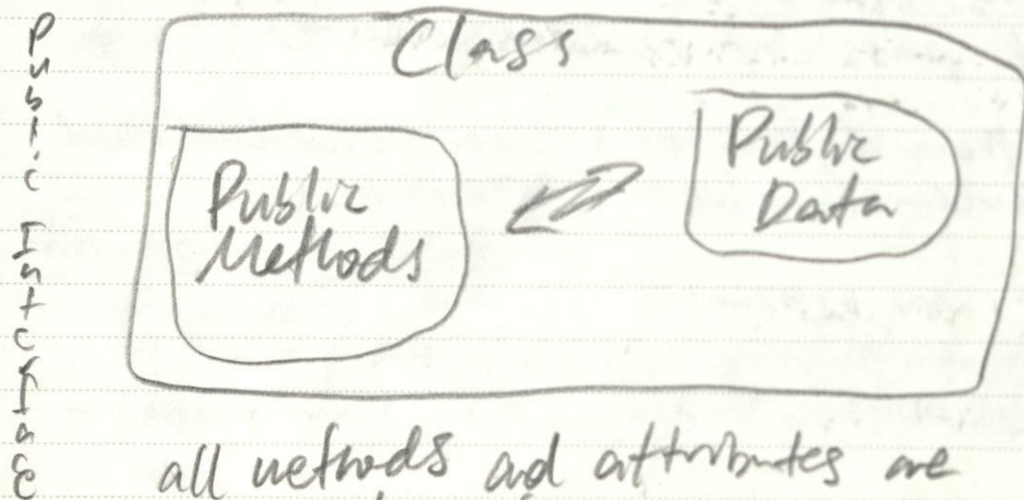
Date. / /

JS - prototype-based OOP
Python - Class Based OOP!

Class-Based OOP: blueprint of
properties and behaviors of objects



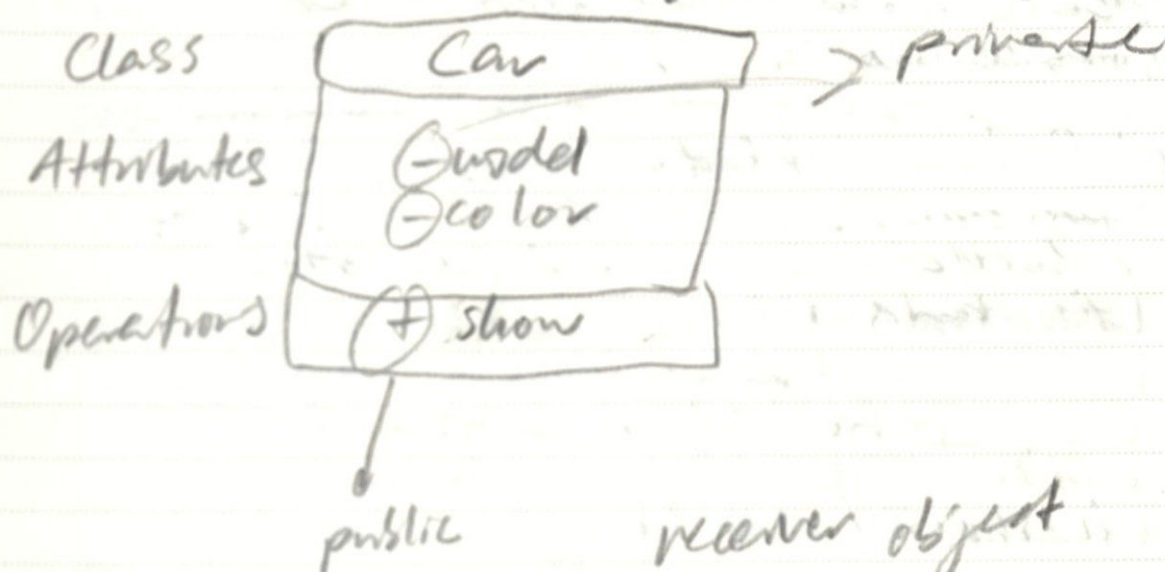
in Python:



all methods and attributes are
public!

UML class diagrams:

UML = unified modeling language

in Python:receiver object
like "this" in JS

```

class Car:
    # constructor
    def __init__(self, model, color):
        self.model = model
        self.color = color
  
```

operations

```

def show(self):
    print("Model is", self.model)
    print("Color is", self.color)
  
```

```

andi = Car("audi a4", "blue")
andi.show()
  
```


Date. / /

object OML:

```
audi: Car
model = "Audi A4"
color = "blue"
```

The self keyword: for every instance method call, python sends instance as a hidden argument to the 'self' parameter

ie. syntactic sugar

audi.show() $\Rightarrow$ Car.show(audi)

Default Constructor vs. Parameterized Constructor

class Student:

default

```
def __init__(self):
    self.name = "Aron"
```

parameterized

```
def __init__(self, name):
    self.name = name
```

obj 1 = Student()

obj 2 = Student("Alice")

obj 1.print()
 # "Aron"

obj 2.print()
 # "Alice"

Multiple Constructors:

passes syntax "compilation" but will fail at runtime

constructor furthest down in class (last) will take precedence

see slide 43

Special Methods: hidden methods that get called w/ syntactic sugars

$1 + 2 \rightarrow \text{-- add --}()$

Nice primary method

```
def __str__(self):
    return f' ... '
```

note: methods/methods will return None if they don't have a return statement

* table of special methods in L12-L15 slide 48

see ex vector.py in class repo

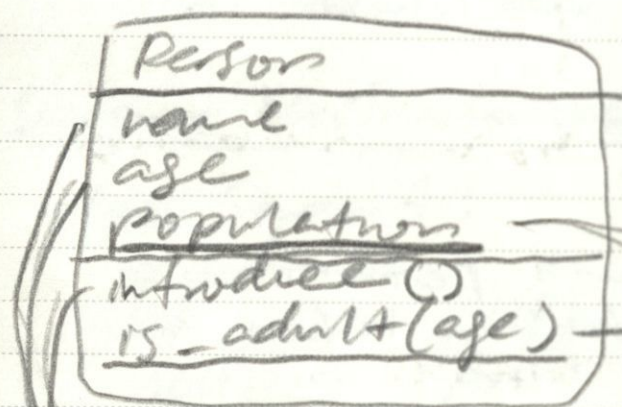
Static vs. Instance Methods

Static Method: defined within a class, but does not access instance fields.

→ used to implement operations that are independent of class instances

Instance Method: operates on an instance of the class (object)

→ access object attributes with 'self'



in UML underline 'static' methods and fields

instance

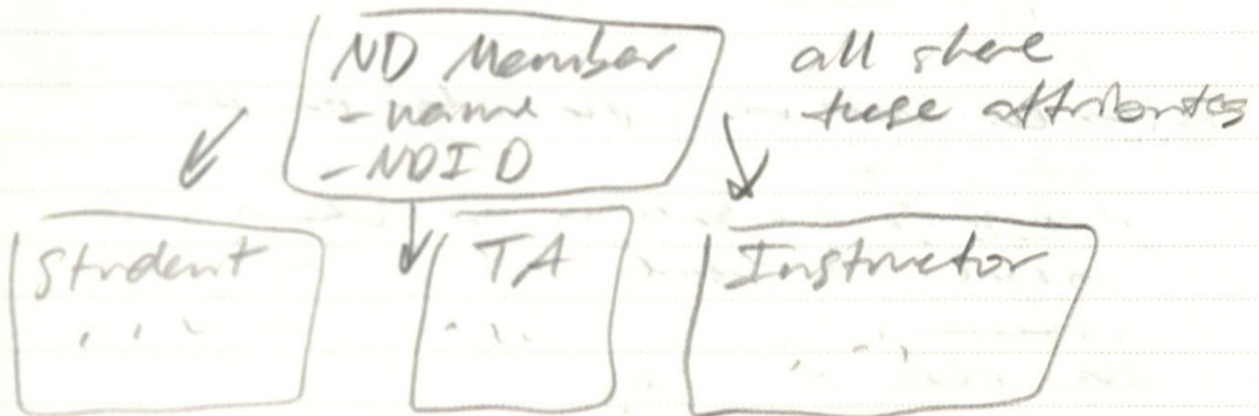
ex of this in oop/person.py

Lecture Notes:

3.3.25

Class Inheritance:

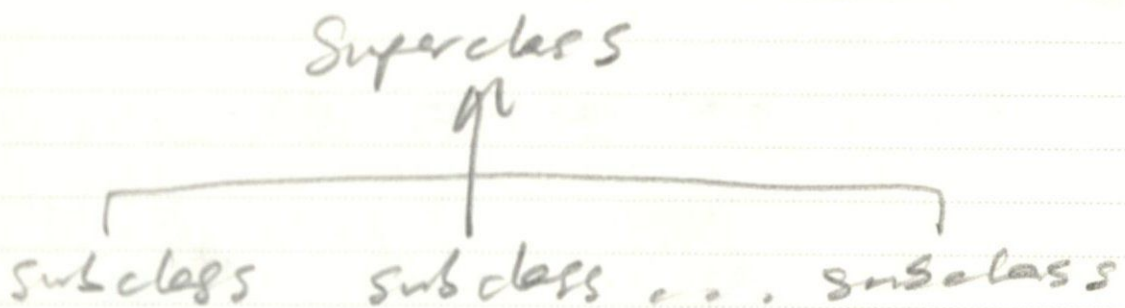
Enables reusability: derived class (subclass) inherits data and operations from its superclass



but have extra specific attributes

in UML:

subclass → superclass



in Python:

```
class Superclass:
```

```
    ;
```

```
class Subclass(Superclass):
```

```
    ;
```

note, can have multiple interfaces

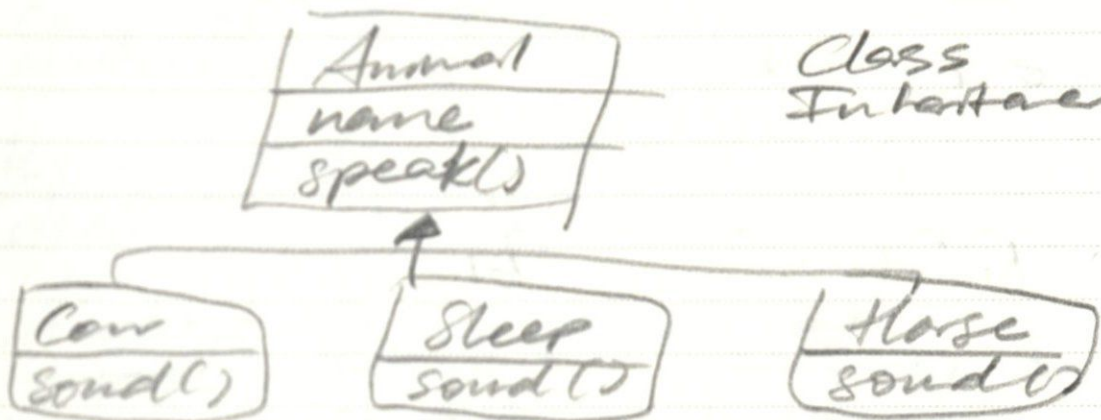
Polymorphism: ability to use same method signature and have different execution based on receiver object

I will be asked an example

exs of multiple interface in slides

Learn Notes.

3.6.25

Class
Interface

class Animal

Python

class Cow(Animal):

```

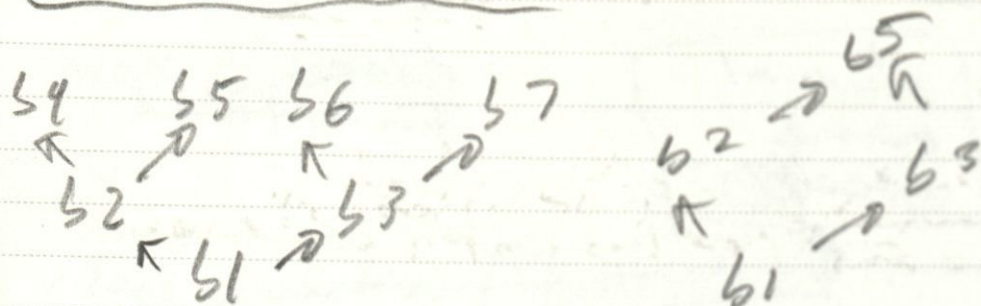
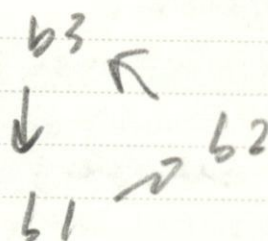
animals = [Horse('J'), Cow('Bessie'), ...]
for a in animals:
    print(a.sound())
  
```

↳ prints sound for each animal

can have Multiple Inheritance:

MRO → Method Resolution Order
 • resolves method conflicts

print Class.mro() to see this
 * on exam, which method gets invoked

How to handle conflicts?Multiple-Interface treeDiamond Problem

most oop languages do not allow multiple inheritances for this reason

CycleOOP Good Practices:

Interface is meant to express FS relationship

Program to an interface and not an implementation

But python only has Abstract Classes

Abstract Classes: classes that contains one or more abstract methods

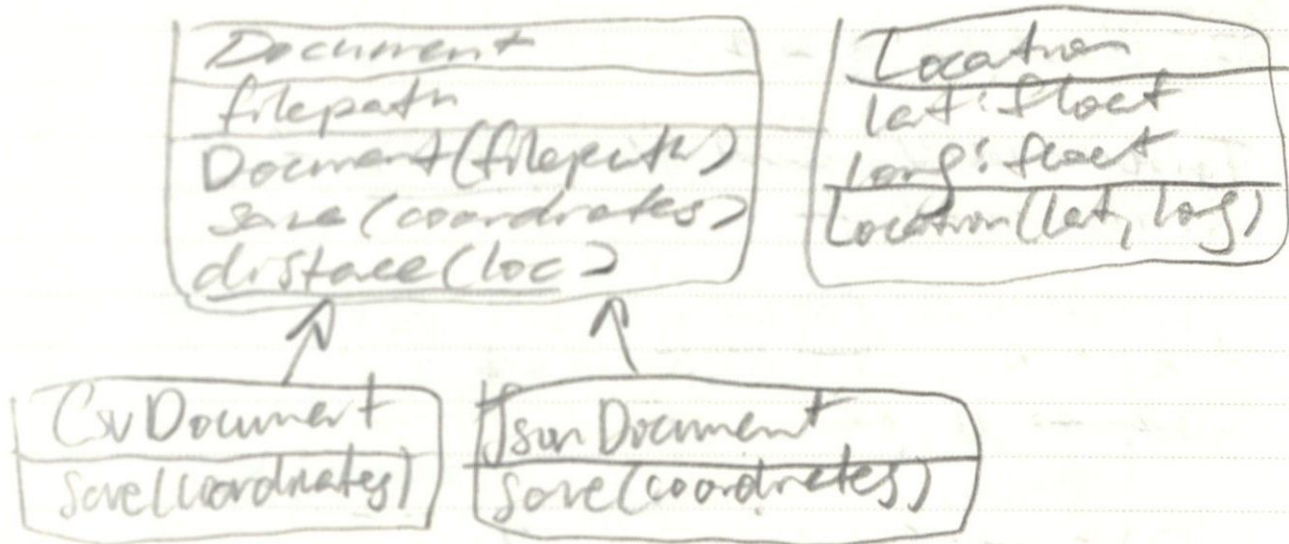
Abstract Methods: method that is declared, but has no implementation

A Abstract classes cannot be instantiated

→ require subclasses to provide implementation for the abstract method

In UML, titles of abstract classes are in italics

Ex. Save locations in documents



Date. / /

Abstract Classes in Python:

from abc import ABC, abstractmethod

class Abstract(ABC):

attributes
 methods

→ indicates
abstract
class
(no instantiation!)

@abstractmethod

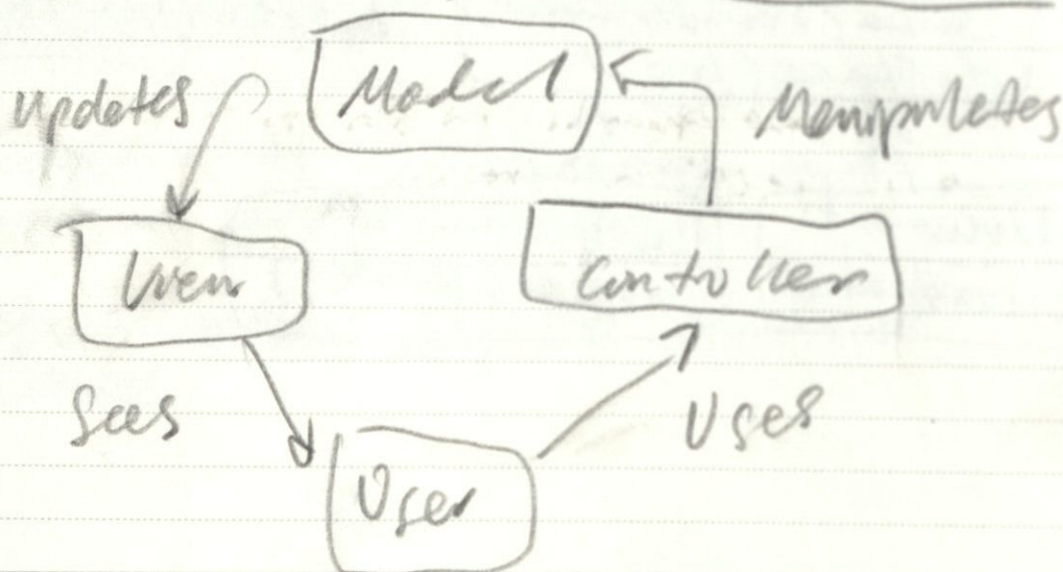
def abstractmethod(self, ...):

:

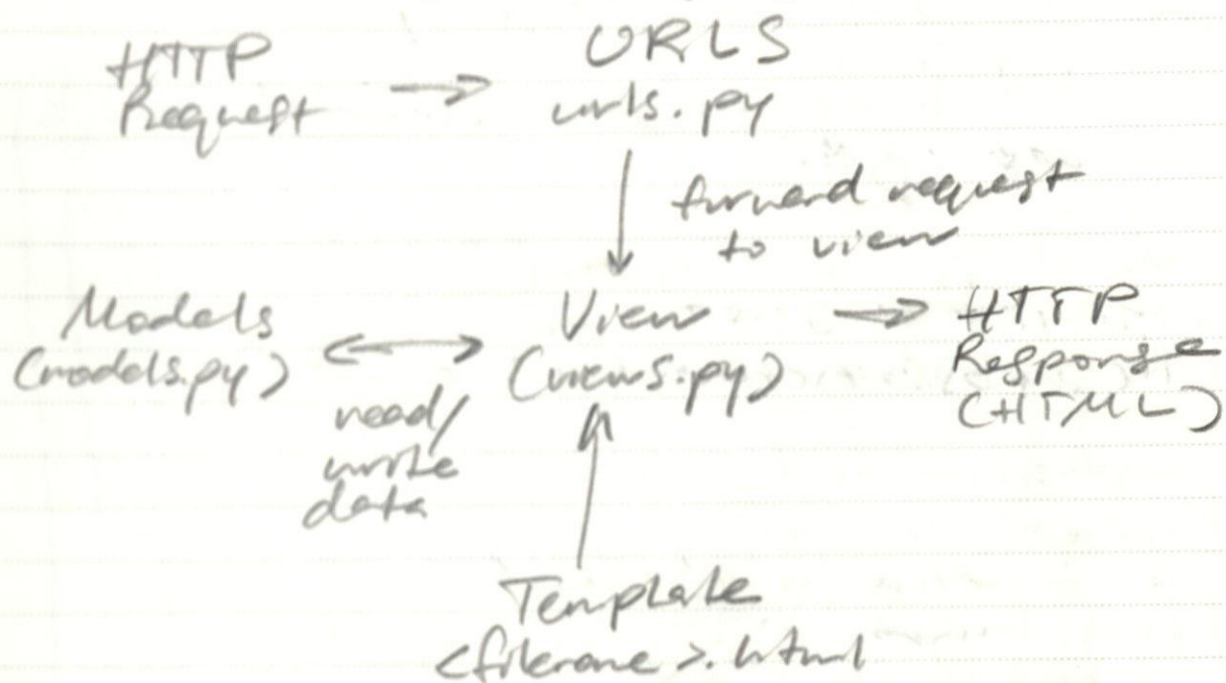
→ indicates abstract
method

Django: high-level Python web
framework

Follows MVS: Model View Controller



3.12.25

Lecture Notes:MVC Paradigm: "Django way"

View: "controller" component of MVC

Template: "View" component of MVC

* step-by-step guide to starting
django project in slides
+ live demo

Lecture Notes

3.19.25

Recap of live demoie how to start a django project

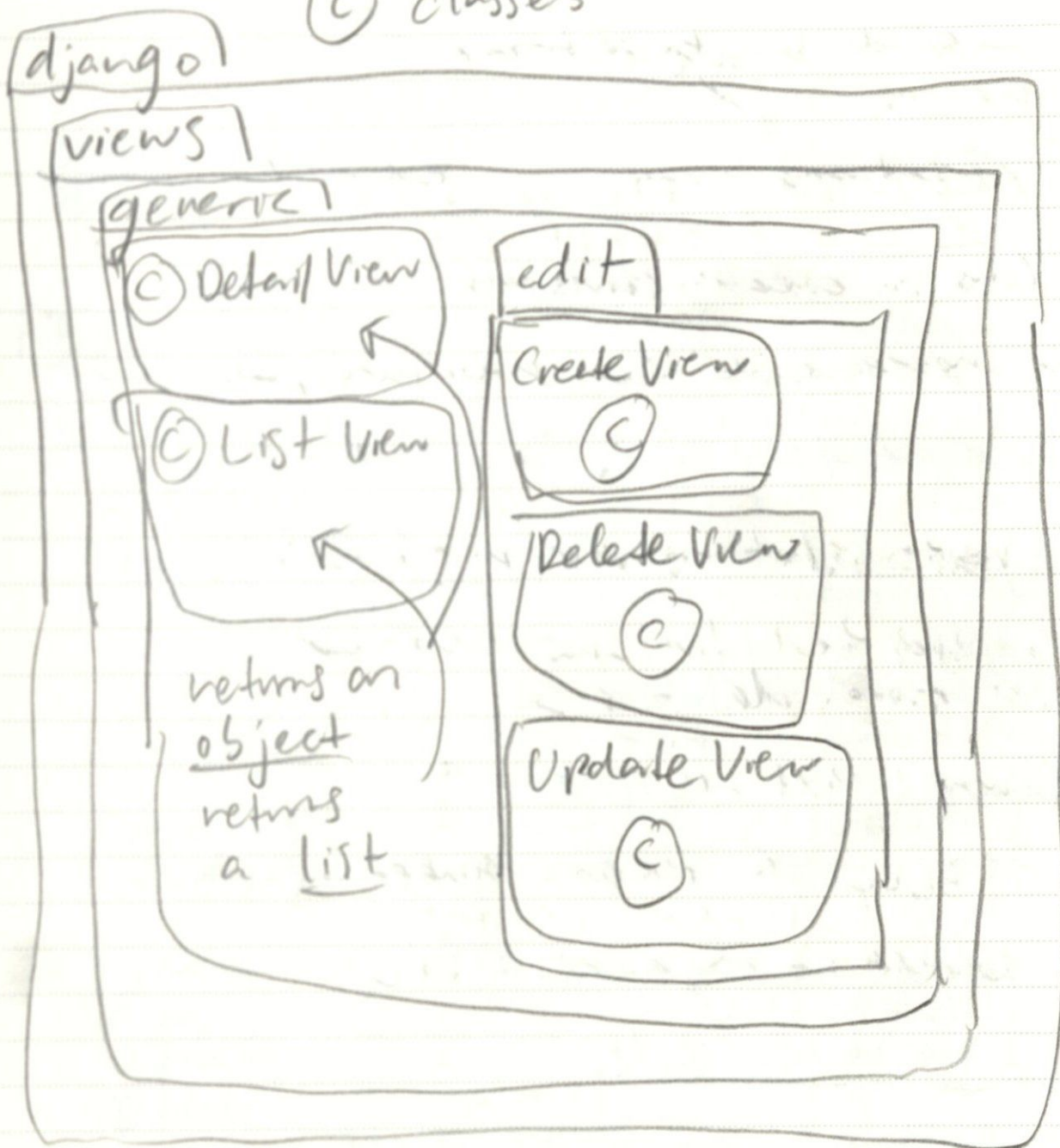
1. created Django project (live_website) and app (polls)
2. added app 'polls.apps.PollConfig' to the list of INSTALLED_APPS (live_website/settings.py)
3. implemented the model classes (live_website/polls/models.py)
4. ran migrations (commands makemigrations and migrate)
5. Modified live_website/urls.py to delegate any request to a URL that starts w/ /polls/ to polls/urls.py
6. Setup the admin interface (command: createSuperuser) and added the model class to admin interface

& more details on features in slides

CRUD in Django :

Create, Read, Update, Delete

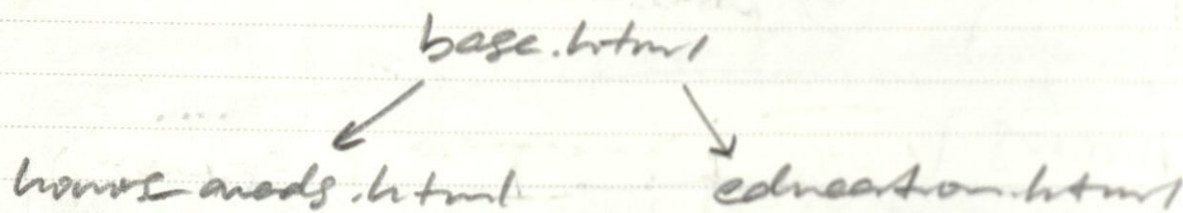
(C) classes



* more features in slides

Lecture Notes :

3.24.25

Template Inheritance : allows for shared html

allows to create skeleton webpages
or reuse code for headers (footers)
\$ ex of this in slides! \$

Object-Relational Mapping : ORM

bridges the gap b/w OOP and
relational databases

Query : get all questions

SELECT * FROM question

Question.objects.all()

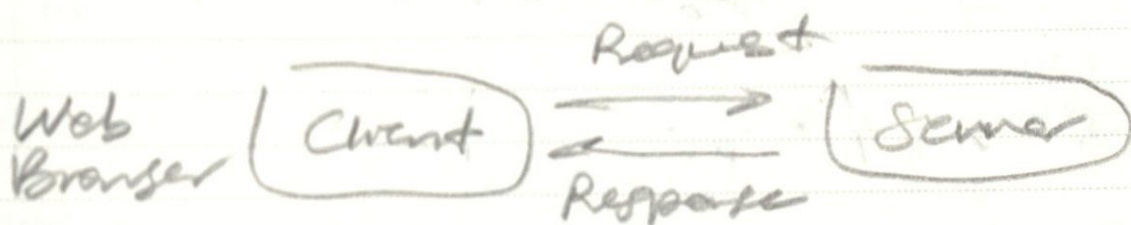
Explore in the shell :

python3 manage.py shell

Lecture Notes

3.26.25

Recall. Internet is a medium to interact with the application server data per request

REST: REpresentational State Transfer

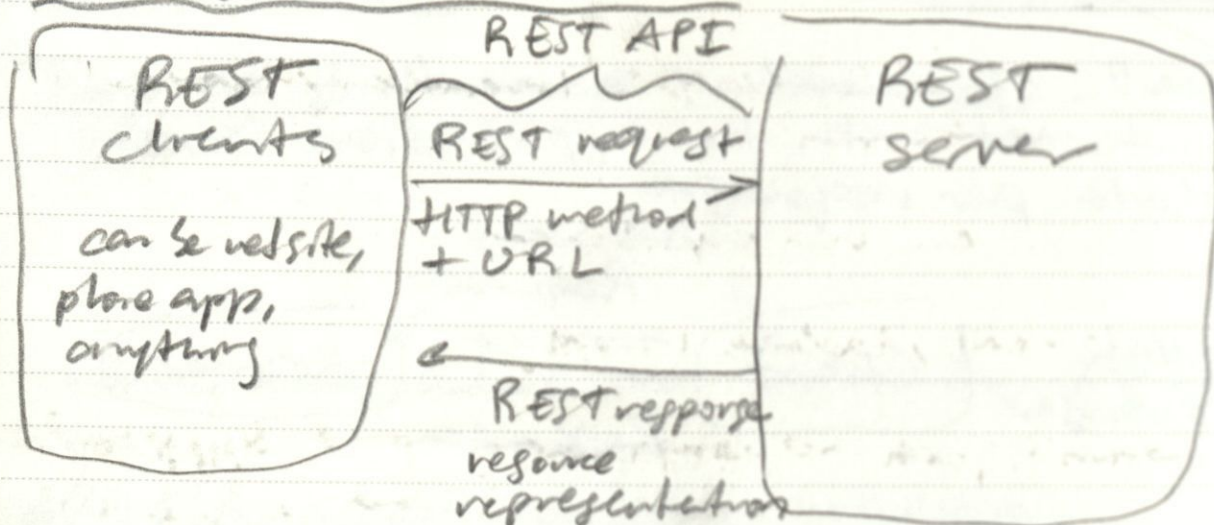
an architecture style:

- still use HTTP to exchange messages
 between server and client
- everything is a resource
- A resource has a representation
 - XML, JSON, TXT
- A resource has a unique identifier
 - URI (uniform resource identifier)
- resources can be handled using HTTP methods

Create	→	POST	} HTTP Methods
Read	→	GET	
Update	→	PUT	
Delete	→	DELETE	

Date. / /

Client-Server Architecture :



* ex of REST API design in slides
you can make REST APIs in ANY language

Implementation in Python :

Step 1.

pip3 install requests ←
need this module

Step 2.

use HTTP request methods

`requests.get(url ...)`

`requests.post(url ...)`

`requests.put(url, ...)`

`requests.delete(url ...)`

↑
other info
with defaults (optional)

* live demo for this in repo

4.2.25

Lecture Notes

Java . known as classes, threads, native platform

"Wrote see, run anywhere

Java Virtual Machine (JVM):

[program].java → compiler → Bytecode
 javac [program].java [program].class

write over, run anywhere

JVM
Linux

Jv
Mae

JrM
Windows

i.e. java is interpreted and compiled to
bytecode like python

javac HelloWorld.java } run java program
java HelloWorld

Syntax Basics: similar to C/C++

- blocks denoted w/ $\{ \}$
- variables have explicit datatypes
- comments have same syntax

main method is programs entry point

Signature must be:

public static void

. receives an array of Strings

Up to one public class per file (must match filename)

. but can have multiple (non-public) classes

Python

class Ex:

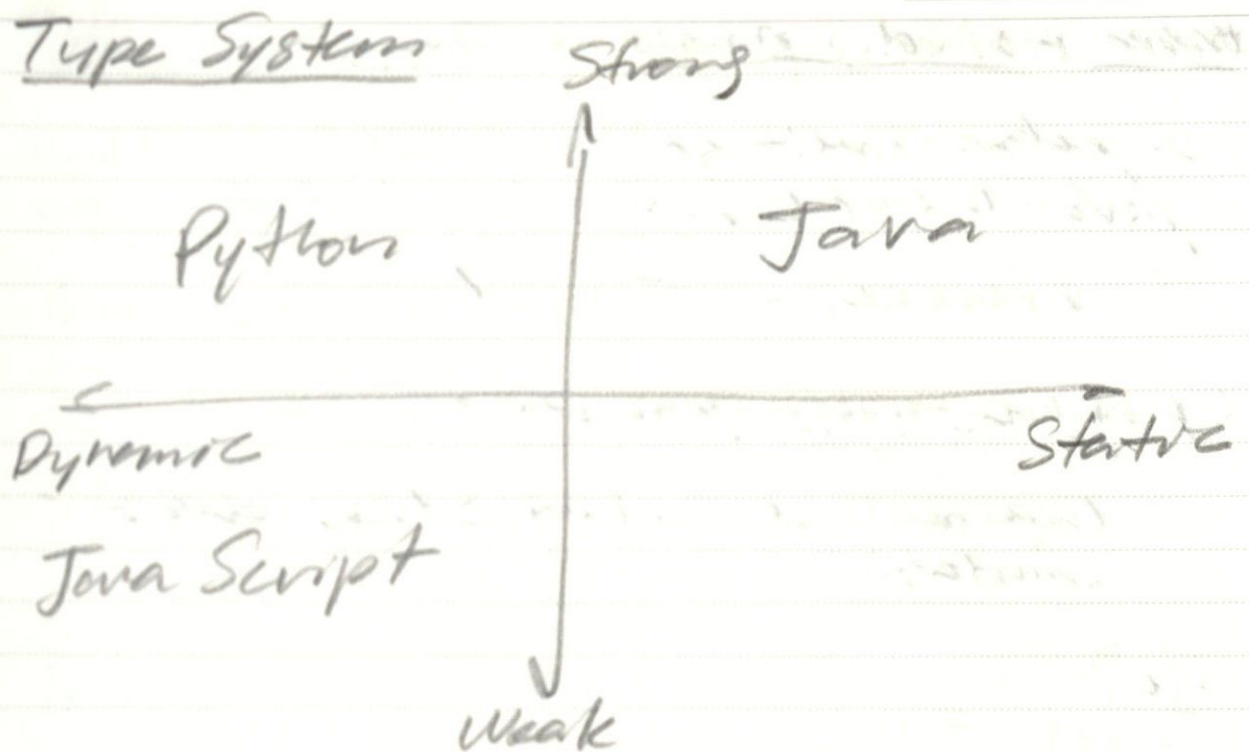
def myMethod(self, x, y):
return x + y

Java:

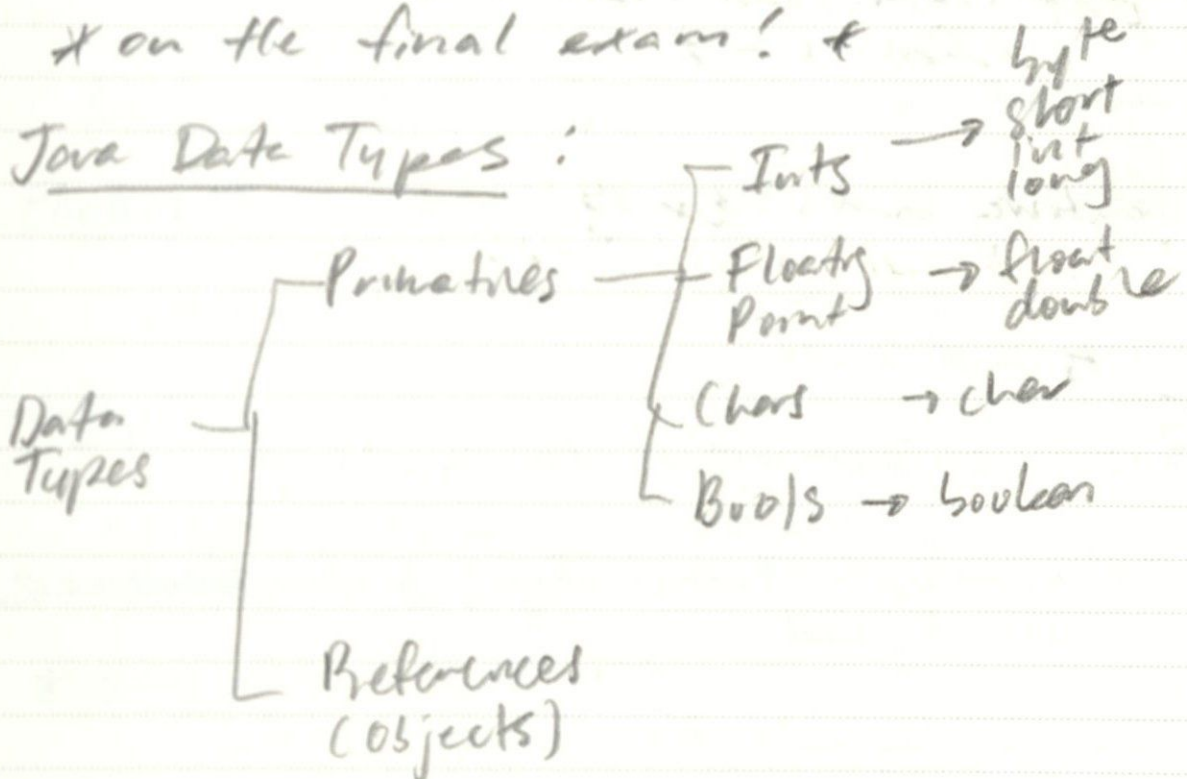
public class Ex {

int myMethod(int x, int y) {
return x + y;
}

}

Type System

* on the final exam! *

Java Data Types :

* table of primitive specifies in slides

Date. / /

References Objects:

- objects are instances of classes
- `java.lang.Object` is root of a program's class hierarchy

Strings are not primitives:

- instances of `java.lang.String` class
- immutable!

Note.

primitive name	Class name
↙ lowercase	↘ uppercase

ex. `String s = "string";`

Objects and null:

null : object refers to 'nothing'

NullPointerException : trying to dereference a null object

similar to Segmentation Fault in C

Type Conversions:

widening: allowed

narrowing: not allowed

long a = 1;

int b = 2;

long c = b;

int d = a;

✓

X no narrowing!

String s1 = "Hi";

Object o = s1;

String s2 = o;

✓

X no narrowing!

in objects can go subclass → superclass
 but not superclass → subclass

Packages: group a set of related classes

similar to Python libraries

Why?

- avoids naming conflicts
- keeps things organized

* example in github

Java does OOP the "classical" way:

- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

It all explained in slides

Lecture Notes:

4.7.25

Abstraction: hiding complex implementations details and only exposing essential features to user

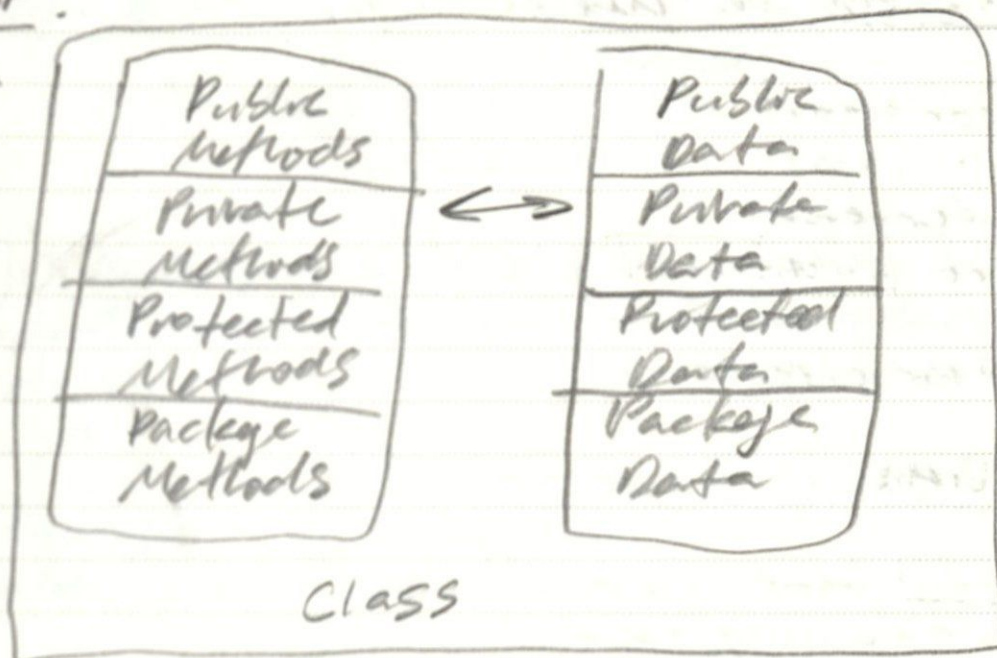
Encapsulation: Bundles data (attributes) and the methods that operate on that data into a single unit called a class

Inheritance: allowing us to create a new class based on an existing class

Polymorphism: enables objects of different classes to be treated uniformly through a common interface

OOP.

INTERFACE



Java

unlike Python only, has public methods and public data!

ex of public vs private vs package protected methods/data in

Java / paradigms / StudentMain.java

Java / paradigms / classes / Student.java

Java Inheritance

- only has single inheritance (one superclass)
- has interfaces
- has abstract classes

Abstract Classes in Java:

- no direct instances
- cannot create objects using this class

Java Interfaces:

- Set of public methods of a class
- indicates what the class can do
- has no state

Abstract Classes vs Interfaces:Design #1.

⊙ Rectangle

- double: base
- double height
- calculateArea()

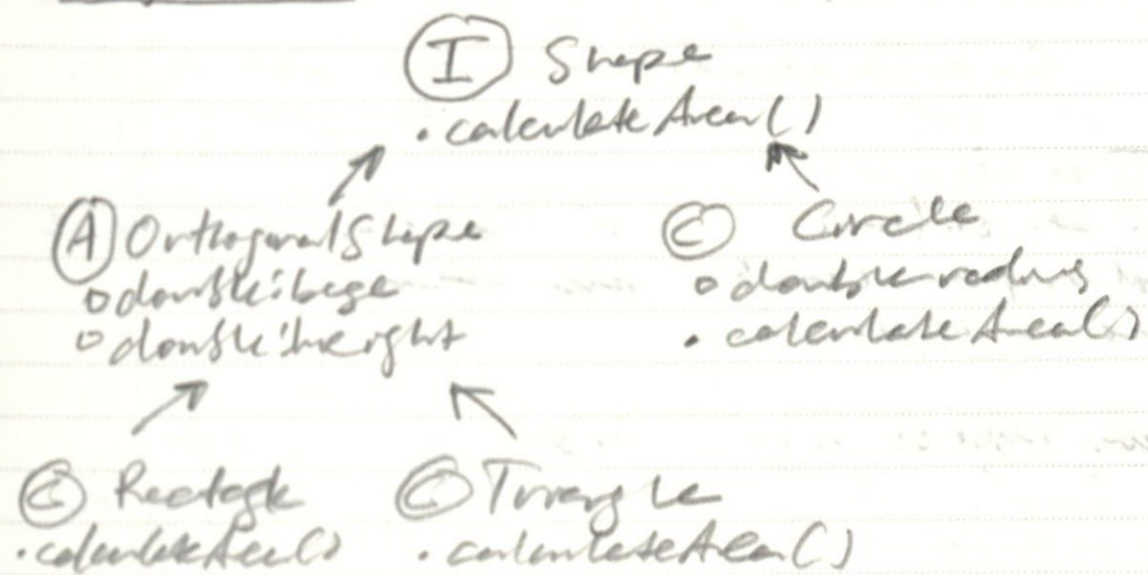
⊙ Triangle

- double base
- double height
- calculateArea()

⊙ Circle

- double radius
- calculateArea()

All "shapes" where we can calculate an area!

Design # 2

Implementation of Design #2 in slides!

instanceof operator:

True if object is an instance of the specified type (class or subclass or interface)

```
Shape v = new Rectangle(2, 3);
boolean a = v instanceof Rectangle;
```


Abstract Classes and Interfaces in Java:

Abstract Class

A class can extend one and only one class (abstract or concrete)

Can have instance and static fields w/ any access modifiers (public/private/protected/package)

Methods can be public/private/protected/package

Interface

A class can implement more than one interface

Can have only public static fields

All methods public by default

Lecture Notes :

4.9.25

toString() : override this so your object can have a meaningful string representation

like `--str--()` in python

@ Override

```
public String toString() {
    return " "
}
```

No argument constructor :

can invoke a default constructor defined by the compiler if no constructor defined

* Default vals def in slides

Object Equality :

boolean equals(Object obj) method
obj1.equals(obj2)

- tests whether obj1 and obj2 reference 2 objects with equal contents.

obj1 == obj2

- tests whether obj1 and obj2 are two references to the same object

Equals expected properties:Reflexive: $a.equals(a)$ Non-null: $a.equals(null) = false$ Symmetric: $a.equals(b) \rightarrow b.equals(a)$ Transitive: $a.equals(b) \text{ and } b.equals(c) \rightarrow a.equals(c)$

An ex in Java/Point.java

hashCode(): returns a hash code value for an objectif a class overrides `equals()`, it must also override `hashCode()`Expectations

- if $a.equals(b)$, then $a.hashCode() == b.hashCode()$
- generates consistent hash codes for the same object over multiple invocations
- `equals()` and `hashCode()` must use the same set of fields
- `java.util.Objects.hash()` used to compute hash code

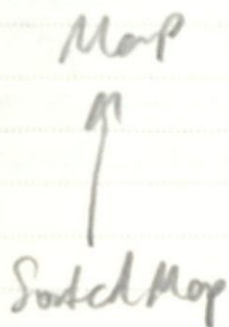
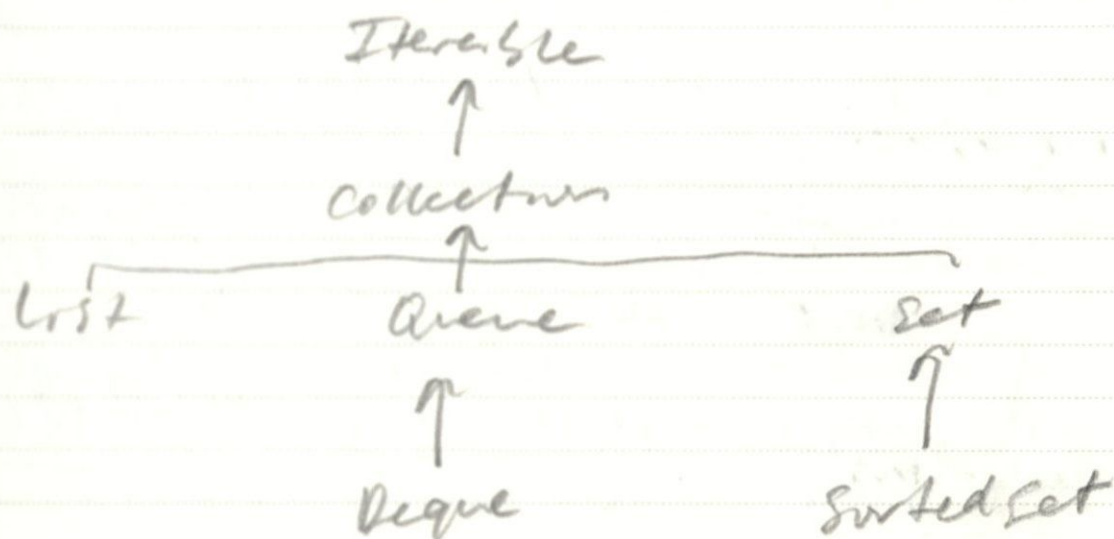
• Override hashCode() ex in `Tara/Point.java`

• on exam

Data Structures in Java:

Queues: regardless of implementation, use interfaces

java.util package: all interfaces



• has a set of classes to implement these interface in various ways

→ in slides

Wrapper Classes

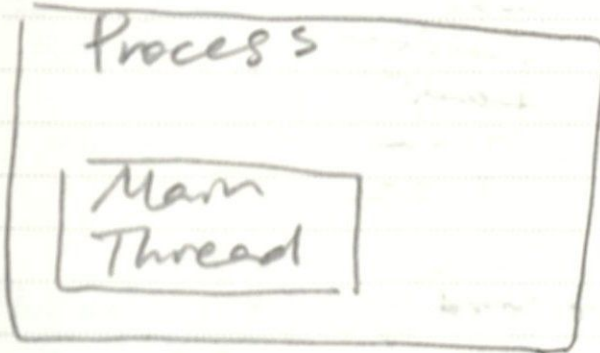
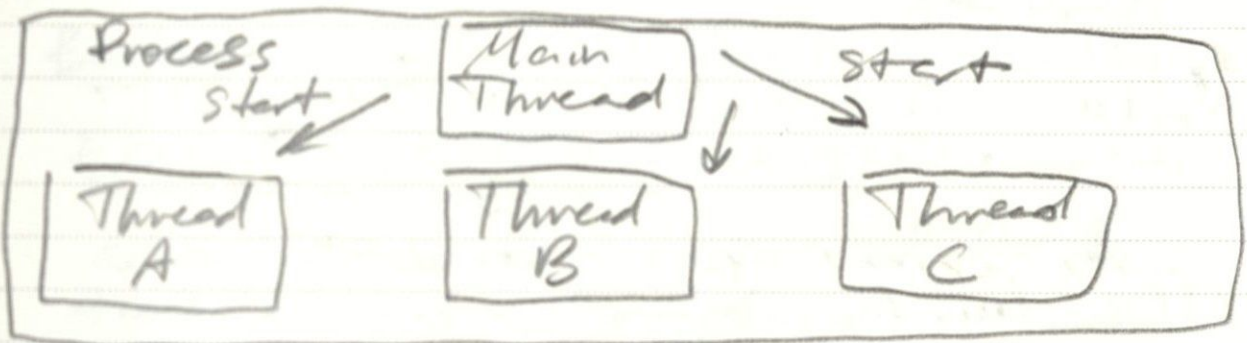
Boolean		Boolean
char	Autoboxing →	Character
byte		Byte
short		Short
int		Integer
long	Unboxing ←	Long
float		Float
double		Double

These collections can only take objects
 need to use corresponding class for
 primitives

eg, `List<int> myList = new ArrayList<>();`
 NO!

`List<Integer> myList = new ArrayList<>();`
 YES!

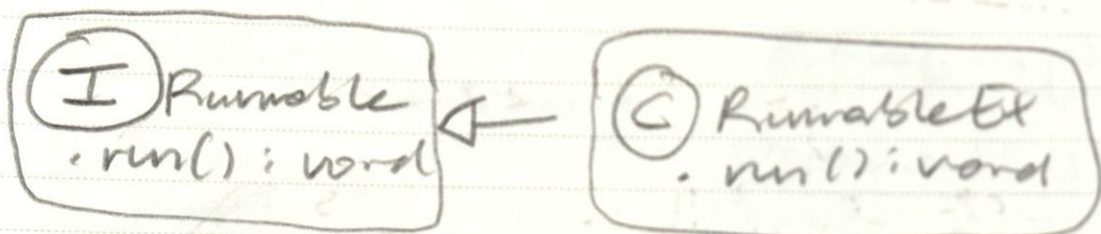
ex represent directed unweighted graph
 using adjacency list in slides

Lecture Notes:Threading in Java:Single Threaded:Multithreaded:

• threads may switch & exchange data

Concurrency: modelling / coordinating activities that may overlap

Multithreading in Java:



- create a class that implements the Runnable interface
- Pass that instance to the constructor Thread(Runnable target)

```

public class RunnableEx implements Runnable {
    @Override
    public void run() {
    }
}
  
```

```

public class Main {
  
```

```

    public static void main(String[] args) {
        Runnable r = new RunnableEx();
        Thread t = new Thread(r);
        t.start();
    }
}
  
```

Exec of multithreaded file finder
in slides

• use of .join() method

note. must try {} catch {}
certain errors in Java

→ Python try - except
equivalent

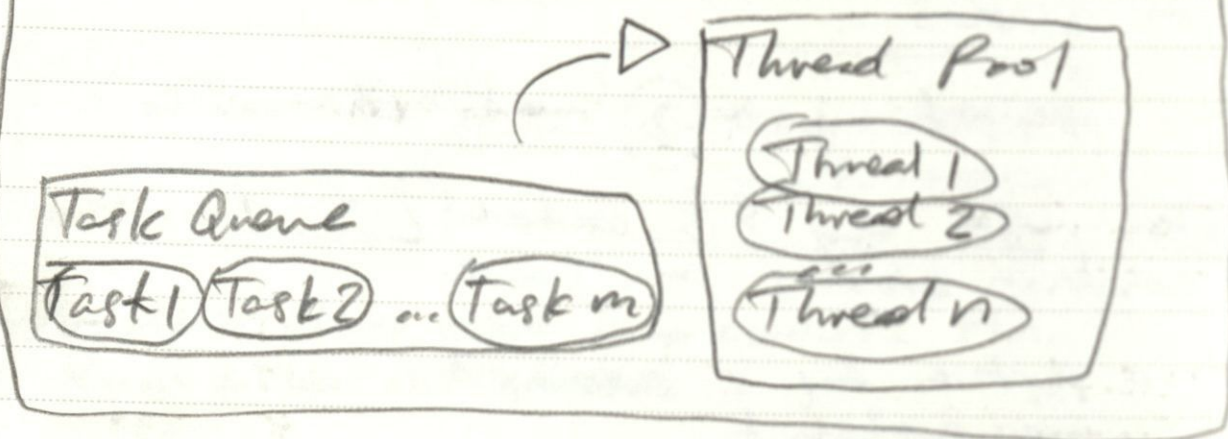
Thread.join(): makes the current
thread wait until another thread
is done executing (for dependencies)

Better...

Executor Service: manages creation,
execution, and termination of tasks
that run on a thread

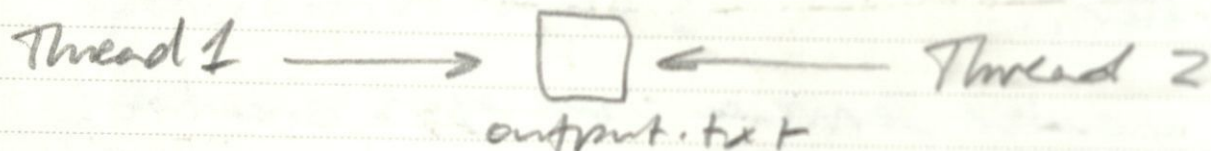
Thread Pools: allows you to multiplex
a large # of tasks on a smaller
of threads.

Executor Service



* changing prev ex to use this model in slides

Sharing data across threads?



appending text to this file at the same time is risky!

Called Race Conditions!

have to lock / unlock the file for threads to prevent this

must use synchronized keyword here

Multithreading w/ Shared Objects:

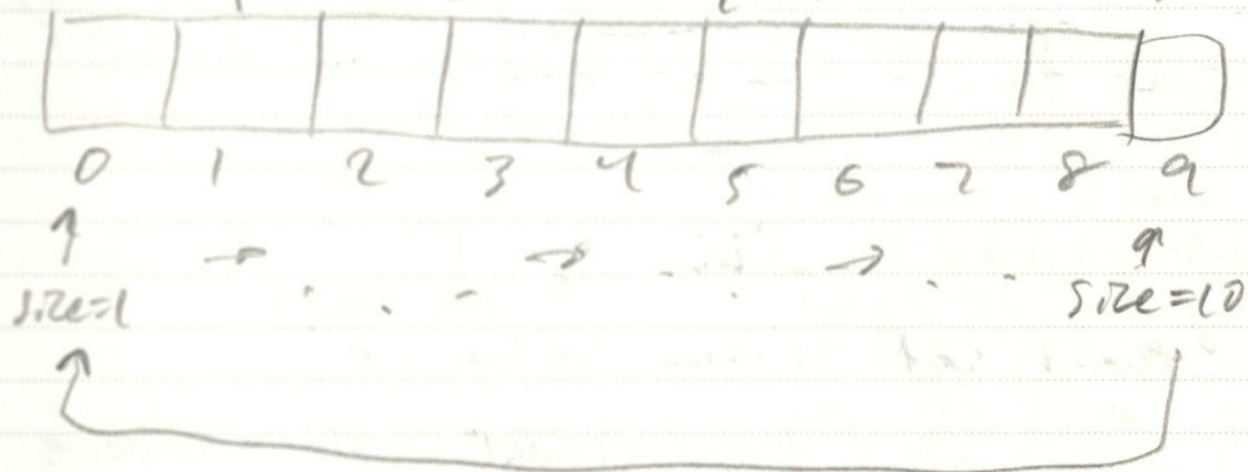
Thread 1 $\longrightarrow$  $\longleftarrow$ Thread 2

1. Generate value
2. Lock
3. Add to buffer
4. Unlock
5. Sleep

1. "
2. "
3. "
4. "
5. "

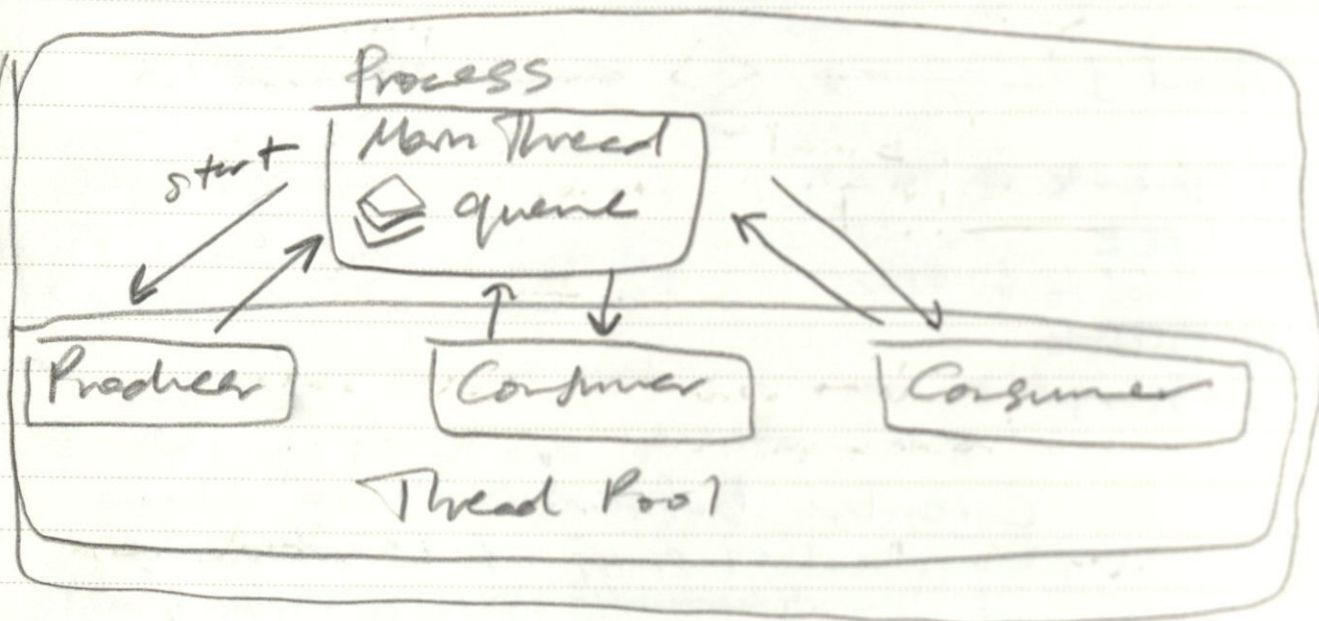
Circular Buffer

implemented as array of 10 elements



Architecture and implementation of this in slides

Producer / Consumer Problem:

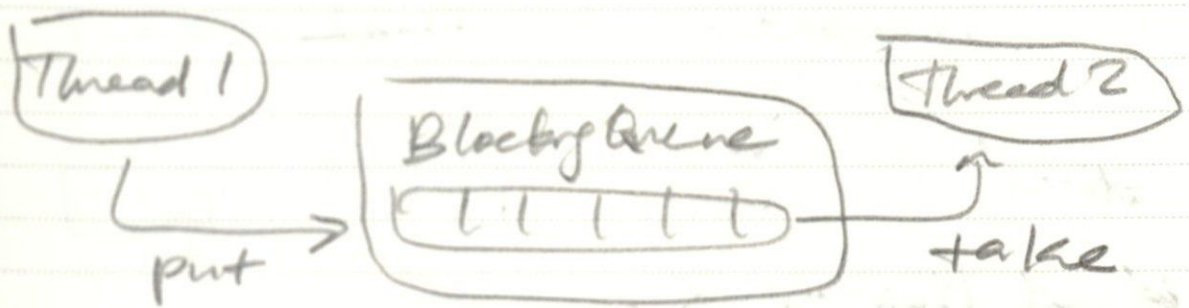


- Some threads produce data
- other threads consume that data

Data Structures: All data structures given in java.util.* are NOT thread safe (from test class)

Java has other thread-safe implementations in stools!

- slower but thread safe

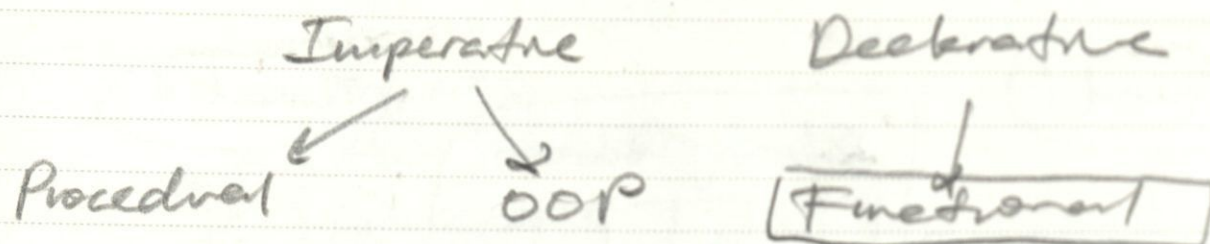
Ex. Thread-Safe Blocking Queue

concurrent data structure that solves
the producer-consumer problem

⌘ implementation of producer/consumer
in slides / repo

Lecture Notes

4.16.25

Functional Programming:

declarative what you want by
applying functions instead
of statements

- first-class f'ns → can assign f'ns to vars
- pure functions } f'ns output
- no side effects } consistent
- lazy evaluation
- statelessness → no knowledge of its past
- immutable data

• code is shorter, less error prone,
easier to prove correct

• more inherent parallelism

Java.Source
Codecharacters
↓

Compiler

→
bytecode

Effect

↑

JVM

ClojureSource
Codecharacters
↓

Reader

data structure
↓

Compiler

→
bytecode

Effect

↑

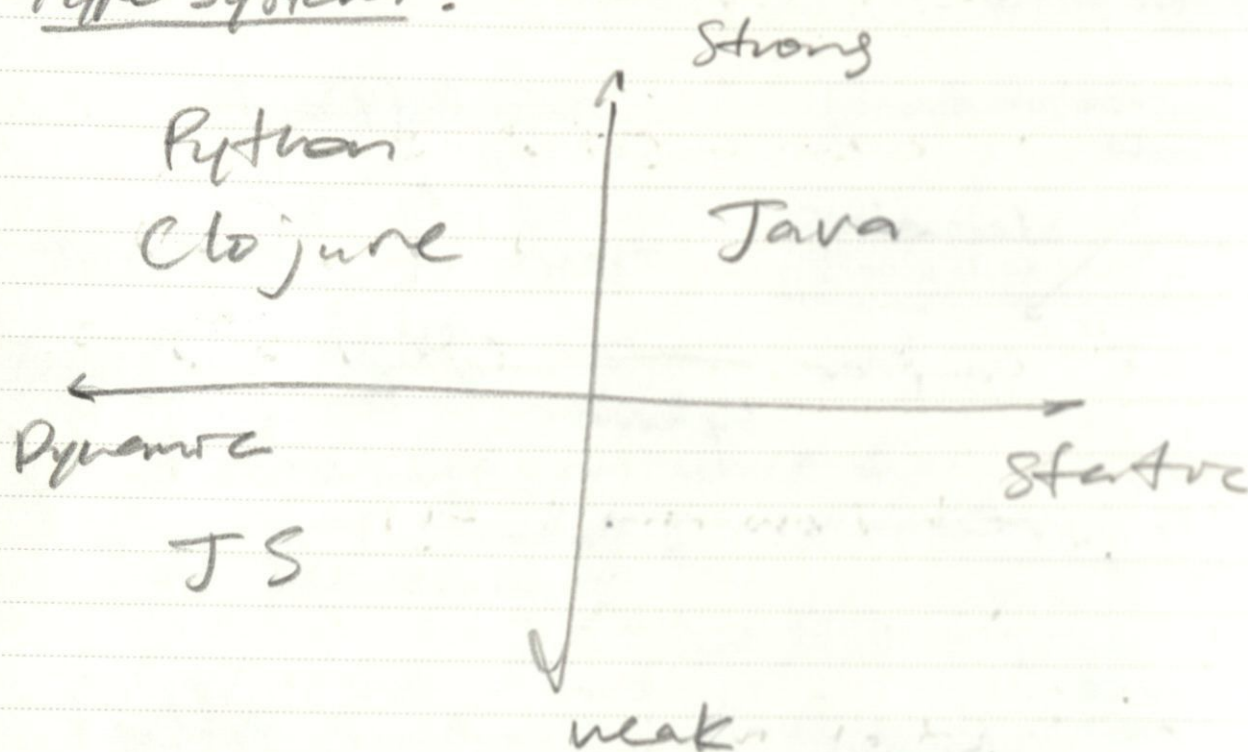
JVM

A compiled!

(clojure).cl;

Date. / /

Type System.



Expressions = smallest unit of granularity

$(f \ x \ y \ z)$

invokes function f with arguments x, y, z

evaluates to the value returned by the function

Ex. evaluate the expression
 $(7 + 3 * 4 + 5) / 10$

$(7 + (3 * 4) + 5) / 10$

$(1 / (+ 7 (* 3 4) 5)) 10)$

```
(def a (/ (+ 7 (* 3 4) 5) 10))
(print a)
```

\$ clj -M [program].clj

& ex of this in repo

```
(print (type a))
outputs a ratio
```

```
(print (float a)) converts to float
```

Declaring Functions:

```
(defn name [p1 p2 ... pn] (...))
      name parameters body
```

& exs of this in repo
 Live Functions.clj

- single "arity"
- multi-arity
- variadic functions

Date. / /

Single "arity"

```
(defn greet [name] (str "Hello, " name))  
(greet "students")  
; Hello, students
```

Multi-arity:

```
(defn messenger  
  ([] (messenger "Hello World!"))  
  ([msg] (println "Hello " msg)))  
  
(messenger) ; "Hello World!"  
(messenger "Bob") ; "Hello Bob"
```

Varadic Functions:

```
(defn hello [greeting & who]  
  (println greeting who))  
  
(hello "Hello" "world" "class")  
; Hello (world class)  
  
(hello "Hello" "world")  
; Hello (world)
```

& like packing/unpacking in Python

Data Types :

Numerics : integers, floaty point, ratios
 3 -1.5 2217

Character : string, character, regex
 "hello" \e #"(0-9)+"

Collections :

'(1 2 3)' ; list
 [1 2 3] ; vector
 # { 1 2 3 3 } ; set
 { :a 1, :b 2 } ; map

and Symbols in slides

Apply function : invokes a fun with
 0 or more fixed args and draws the
 rest from a sequence

let keyword :

let [name, value, ... name_n value_n]

bindings

(code that uses variables) ?

name_x used here

Scopes:

defn has function scope
let has "block" scope

Namespaces:

(ns library)

; every expression here is in library namespace

(ns app)

; now everything in new namespace

; access old namespace with
library / var
library / function

can switch with
(use 'library)

Lecture Notes

4.23.25

Collections in Clojure : immutable

- changing collection creates a NEW collection with the changes
- Equality implemented by "value"

* declarations and methods at
Vectors, Lists, Sets, and Maps
in slides!

Equality (=) : different from JS

Loosely 1 =

* ex of this in
Clojure / Live Temperatures. clj
• also reads in and parses
a CSV file

* more complicated ex in
Clojure / Live City Temperature. clj

Date. / /

Looping in Clojure done by Recursion:

Uses Tail Recursion: recursive call is the last thing executed by the function

Tail Recursion is optimized by Clojure

```
(defn factorial [n]
  (loop [count n acc 1]
    (if (zero? count)
        acc
        (recur (dec count) (* acc count))))
  ) ; end of if
  ) ; end of loop
  ) ; end of function declaration
```

Lecture Notes:

4.28.25

Persistent DS in Clojure

```
(def a (list (repeat 10 "dog")))
; 10 dog list
```

```
(def b (conj a "cat"))
; new list, cat at front
```

creates a new list w/ 11 elements
since Clojure lists are immutable

Compare Objects:

identical? tests if 2 arguments
are the same object

```
(identical? a (rest b)) ; true
both persistent lists!
```

Persistent DS: DS that preserves its
previous version when modified

• effectively immutable

& Clojure has efficient implementations
of persistent DS.

Date. / /

Persistent vs Transient DS:

Persistent : immutable

- good : thread safe, sharing
- bad : performance

Note, to improve performance

- construct object w/ transient structures
- convert to persistent before use

Transient Ex.

```
(defn vmap2 [n]
  (loop [i 0 v (transient [])]
    (if (< i n)
      (recur (inc i) (conj! v i))
      (persistent! v))
    )
  )
```

Date. / /