

Software design principles

Modeling: identify concepts, properties, behaviors, and their relationships

Choose the right architecture

Create/Use appropriate data structures

Use design patterns when appropriate

Refactor (change the design) as necessary

Android

AndroidManifest.xml contains all activities
extends AppCompatActivity
onCreate(Bundle savedInstanceState){
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
<Button android:id="@+id/title"
android:onClick="onClearButtonClick" />
//onClearButtonClick(View v) in the class
Button b = (Button) findViewById(R.id.title);
Intent i = new Intent(this, AnotherActivity.class) // args = context, class
i.putExtra("key", "value"); //getStringExtra("key") in other class
startActivityForResult(i, AnotherActivity_ID);
onFinishButtonClick(View v){
Intent i = new Intent();
i.putExtra("NUM_CLICKS", num_clicks);
setResult(RESULT_OK, i);
finish();
}
//in the returning activity...
onActivityResult(int requestCode, int resultCode, Intent intent){
switch (requestCode) {
case AnotherActivity_ID:
}
Toast.makeText(context, text, duration).show();
//getApplicationContext() for context
//Toast.LENGTH_LONG for duration

What's wrong with Monolith?

analyzability: main method can be fairly long and hard to read/understand

changeability: hard to find the code you want to change

stability: hard to know whether a change in one place will affect other places

testability: if something goes wrong, hard to find the bug

hard to reuse any of the code

Javascript

push, unshift (front); pop, shift (front)
if a is array of size 3
a[-5] = 'elephant' adds key value mapping to array
a[5] = 'elephant' adds elephant at index 5
with element at index 4 = undefined
function print(n) {
console.log(n)}
nums.forEach(print);
function square(n){
return n*n;}
var squares = nums.map(square);
// returns the array of squares
var myDog = {
name: 'Cooper',
type: 'dog'}
pet['status'] = 'good boy';
var add = function (a,b){...}
add(3,5);
var teacher = {
name: 'name',
greet: function () {...}
}
teacher.greet();
//this is called a prototype
function Teacher (name, subject){
this.name = name;
this.subject = subject;
this.greet = function() {...}
}

Javascript (cont)

```
var t1 = new Teacher('Kate', 'math');
var t2 = new Teacher('Mike','science');
set.has(...);
set.delete/add("fox");
for (let v of set.values()) //iterate
map.set("dog", "rover"); //has " instead of '
map.get(...)
for (let [key, val] of map.entries())
let square = function (n) //same as
let square = n => {}
```

MongoDB

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/myDatabase');
var Schema = mongoose.Schema;
var personSchema = new Schema({
name: {type: String, required: true, unique: true},
age: Number } );
module.exports = mongoose.model('Person', personSchema);
var Person = require('./Person.js');
app.use('/create', (req, res) => {
var newPerson = new Person ({
name: req.body.name,
age: req.body.age,});
newPerson.save( (err) => {
if (err) {
res.type('html').status(500);
res.send('Error: ' + err);
}else{
res.render('created', {person: newPerson });}
}); });
all within app.use
Person.find( (err, allPeople) => {
if (err) {
res.type('html').status(500);
res.send('Error: ' + err);
}else{
```

MongoDB (cont)

```
res.render('showAll', { people: allPeople });
});
Person.findOne( {name: searchName }, (err,
person) => {
if (err) {...}
else if(!person) {
res.type('html').status(200);
res.send('No person named ' + search-
Name);
else{
res.render('personInfo', {person:person});}
});
to update the database, combine both
findOne and save
```

ES6

ES6 adds classes (to replace prototypes), sets (distinct and ordered), maps (set of keys and values), and arrow functions among other things.

Three-Tier Architecture

User Interface: get input, show output

Processor/Controller: process data, searching and filtering

Data Management: storing/retrieving data

Express

```
var express = require('express');
var app = express();
app.use('/', (req, res) => {
res.send('Hello World!');
res.json('Hello World!');});
//res.json is identical to res.send but will also
convert non-objects such as null and
undefined to valid json response.
res.status(404).send('Not found!');
//invoke the function when request for '/' is
received
//res represents the HTTP response
app.listen(3000, () => {
console.log('Listening on port 3000');});
//listens for incoming HTTP requests
```

Express (cont)

```
app.use('/public', express.static('files'));
//express.static middleware for serving
static files locally stored
?name=Lydia&returning=true in the URL
var query = req.query;
var name = query.name;
body parser middleware
var bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({extended:
true}));
<form action = "/handleForm" method = "-
post">
app.user('/handleForm', (req, res) => {
var whatever = req.body.inputNameInH-
TMLForm;
var name = req.body.username;
var animals = [].concat(req.body.animal);
//don't have to worry about null
res.type('html').status(200);
//start constructing response
res.write('Hello');
//write but don't send yet
res.write("<ul>");
animals.forEach( (animal) => {
res.write('<li>' + animal + '</li>');});
res.write('</ul>');
res.end();
//send response
}
app.set('view engine', 'ejs');
app.get('/', (req, res) => {
res.render('welcome', {username: 'Olivi-
a'});});
//render the welcome.ejs file in views/
subdirectory
//EJS will execute any JavaScript that
appears between <% and %> when
generating the HTML page on the server.
//<%= username %>
//allows you to not have to do a lot of
res.writes with html in your javascript
```

Singleton

```
public class MyClass{
private MyClass(){...}
private static final MyClass instance = new
MyClass();
public static MyClass getInstance(){
return instance;}
public void smthng(){...}
```

Factory Method

```
public abstract class Proc{
protected Reader r;
public Proc(){ r = createReader}
protected abstract Reader createReader();
}
public class fileProc extends Proc{
protected Reader createReader(){
return new fileReader();}
}
```