TYPESCRIPT 2

AND YOU...

MINDFUL EXERCISE

THE MOCK QUORA RESPONSE: “HOW DO I MASTER PROGRAMMING?”

- Learn as much as you can about as much as you can
- Always try to see the big picture - the high-level why, not just how
- Stuck on a nagging problem? Leave it and revisit it later - you might be surprised at the outcome.
- There are no shortcuts - being good at anything takes time. The more time you have, the better you will become
- Seek areas to improve your craft and never assume there is one “right” way to solve a problem
- Find solutions that are scalable on many levels - meaning the solution will effectively adapt to an increased demands placed on it and will resist complexity over time.
- Accept that you will never truly master programming and work it into your learning strategy

THE MOCK QUORA RESPONSE: “HOW DO I MASTER PROGRAMMING?” (CONT’D)

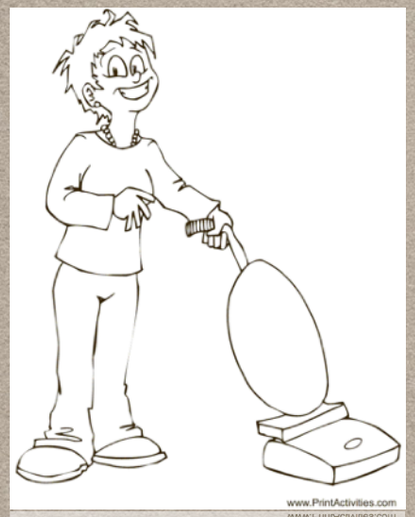
- Most importantly learn effective ways to make being lazy scale.

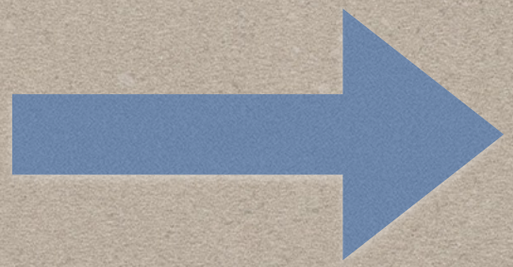
SCALABLE LAZINESS

- **Definition**: Doing **more things** with the same or less amount of effort required. Ideally, this means never having to do the same thing twice…

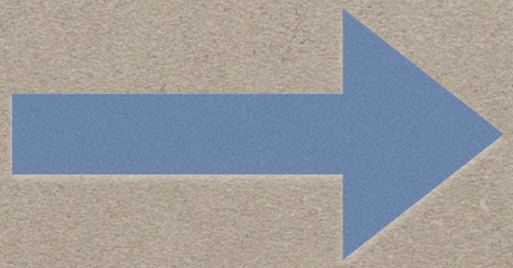
EXAMPLES OF SCALEABLE LAZINESS

- Vacuuming (Don't like it so we scale…)





- Dishwashing (Who really likes this?!?)



+

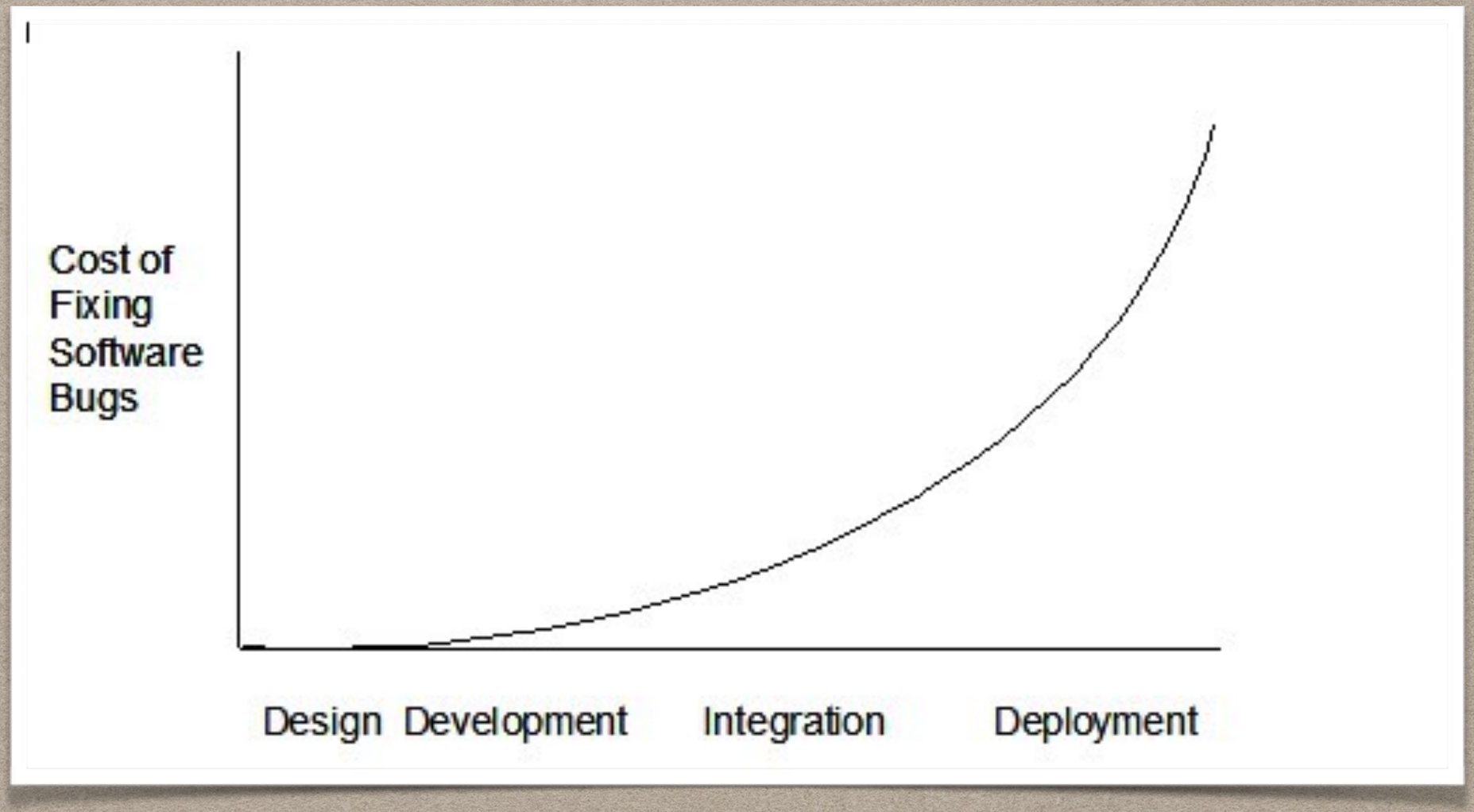
SCALEABLE LAZINESS IN SOFTWARE DEVELOPMENT IN A NUTSHELL

- The greatest enemy of scaleable laziness is complexity
- Increasing complexity gives rise to issues of maintainability of extensibility of the application
- Ideally we want complexity to remain relatively constant as an application's feature set grows
- Employing more Developers to a solve a complex problem can make things worse (**Ref**: Mythical Man Month)

SCALEABLE LAZINESS IN SOFTWARE DEVELOPMENT IN MORE DETAIL

- Having a clear understanding of the problem you are trying to solve
- Leveraging the expertise (technical and non-technical) of others to adopt best practices so we focus less on the tech and more on our users/mission
- Breaking down components into consumable layers, applying “proper” abstraction and encapsulation of data and logic into a role of single responsibility. Easier said than done of course…
- Refactor when things start to get too complex, or weird (leading back to the first point)
- Favour readability over vague ingenuity, except when writing GL shaders (all bets are off there…)
- Adopt appropriate tools used promote code quality (i.e. code linting, IDE intellisense, automated testing, etc.) and rapid development. Utilize easy to learn (hard to master) technologies like JavaScript/ECMAScript.

DON'T JUST TAKE MY WORD FOR IT: THE (NOT-SO) HIDDEN COST OF COMPLEXITY



NIST. *The Economic Impacts of Inadequate Infrastructure for Software Testing* (Planning Report 02-3). Gaithersburg, MD: National Institute of Standards and Technology, 2002.

```
   import this
   """The Zen of Python, by Tim Peters. (poster by Joachim Jablon)"""

 1 Beautiful is better than ugly.
 2 Explicit is better than impl..
 3 Simple is better than cσмplεx.
 4 Cσмplεx is better than cOmp1|c@ted.
 5 Flat is better than nested.
 6 S p a r s e  is better than dense.
 7 Readability counts.
 8 Special cases aren't special enough to break the rules.
 9 Although practicality beats purity.
10 raise PythonicError("Errors should never pass silently.")
11 # Unless explicitly silenced.
12 In the face of ambiguity, refuse the temptation to guess.
13 There should be one-- and preferably only one --obvious way to do it.
14 # Although that way may not be obvious at first unless you're Dutch.
15 Now is better than ...                    never.
16 Although never is often better thanrightnow.
17 If the implementation is hard to explain, it's a bad idea.
18 If the implementation is easy to explain, it may be a good idea.
19 Namespaces are one honking great idea -- let's do more of those!
```

MANAGING COMPLEXITY

(HINT: NOT JUST USEFUL FOR DEVELOPMENT)

JAVASCRIPT: SOME OF THE GOOD PARTS

- The Language of the Web
- Functions
- Loose Typing*
- Dynamic Objects*
- Literal Object Notation

JAVASCRIPT: SOME OF THE BAD PARTS

- Global Object Namespace
- Functional scoping
- == vs === coercion can and will haunt you
- Semicolon insertion i.e. return with a value on the following line).
- **The good news**: Many of the “Bad Parts” (many not mentioned here) addressed in ES6/7

IMPROVING ON JAVASCRIPT WITH TYPESCRIPT

- Provides **optional static type analysis**. Great for catching weird bugs especially when the application scales.
- TypeScript (TS) is an open-source language superset of ES3-ES7+ meaning that you can use functionality from current and future proposed language features now.
- Developed by C# creator Anders Hejlsberg at Microsoft
- See more at https://www.typescriptlang.org/

TYPESCRIPT: THE GOOD PARTS

- Transpiles to your favourite ES**X** language that can then be used in conjunction with Webpack and Babel. Supports ES6, SystemJS and CommonJS modules out of the box.

- Historically TS has released proposed ES**X** features faster than other transpilers and it has great tooling in several IDE's (VisualStudio Code, WebStorm, etc.)

- Along with those features that currently exist in ES**X**. You can make use of traditional OOP features such as public private method and property accessors, overloaded methods, readonly properties, interfaces, custom types, decorators, generics, etc…

- Great 3rd party support for JS libraries and can be used simultaneously with existing libraries and frameworks i.e. Phaser, Angular 2, React, etc.

- Extremely useful for writing scaleable applications that add some syntax sugar to catch a greater subset of issues earlier than ESX alone.

TYPESCRIPT: THE BAD PARTS

- Is an abstraction of JavaScript and will probably never run natively in the browser except maybe Micro$oft Edge (not IE).
- There is a learning curve that is greater than that of JS (though not much worse in my opinion)
- Type annotations means extra boiler plate which promotes readability but makes at the cost of being more verbose
- Requires TypeScript Declaration Files to work effectively with native JS libraries.
- Many of the features of the language promote OOP-style which can promote extremely scaleable solutions. OOP, however can be inherently harder to design for and can lead to higher complexity if misused.

EXAMPLE TYPESCRIPT

```
// TypeScript
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}
let greeter = new Greeter("world");
let button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function() {
    alert(greeter.greet());
}
document.body.appendChild(button);
```

```
// TypeScript Transpiled to ES5
var Greeter = /** @class */ (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return "Hello, " + this.greeting;
    };
    return Greeter;
}());

var greeter = new Greeter("world");
var button = document.createElement('button');
button.textContent = "Say Hello”;

button.onclick = function () {
    alert(greeter.greet());
};

document.body.appendChild(button);
```

TYPESCRIPT: CORE TYPES

- string, string[]
- number, number[]
- boolean, boolean[]
- any, any[]
- (arg: type, arg_2: $type_2$...) => *function return type* (number, boolean,...etc.)
- null | undefined

TYPESCRIPT: VARIABLES

- var shouldSleepOnCouch: boolean = true ;
- let couch: Couch = new Couch(Couch.COLOUR.BLACK | Couch.TEXTURE.SMOOTH | Couch.MATERIAL.LEATHER);
- const gotHomeLate: (person: Person, arrivedAt: Date) => Couch | null = ….<function expression>…

TYPESCRIPT: FUNCTIONS

```
function sum(x: number, y: number) : number { return x + y; }

// overloading...below

function delegateChoreToWife(wife: Wife) : void;

function delegateChoreToWife(wife?: Person) : Couch;

function delegateChoreToWife(wife?: any) : any {

    if ( ! (wife instanceof Wife)) {

    return new Couch(Couch.COLOUR.BLACK | Couch.TEXTURE.SMOOTH | Couch.MATERIAL.LEATHER)

    }

    throw new SleepOnCouchError(`${wife.name} says: You know where to sleep tonight!`);

}
```

TYPESCRIPT: TYPES

- type ID = number;
- type FragShader = string | string[] | undefined;
- type ReturnsID = () => ID;
- type Excuses = “to_busy” | “feel_tired” | string[] | null;
- type ConvincibleExcuseFlag = { [E in Excuses]: boolean };

TYPESCRIPT: INTERFACES

```
interface AChore {
  readonly id: number;
  name: string;
  description?: string;
}
interface ABoringChore extends AChore {
  bestAvoidanceStrategy(): boolean;
}

interface DoChore {
  (person: Person, chore: Chore): boolean;
}

class Chore implements ABoringChore {}
```

TYPESCRIPT: CLASSES

```
class LifeAsOldMan extends  LifeAsYoungMan {
    public static DEFAULT_STUFF = Stuff(1, 'eating' , 0); // –> Stuff Contract: id, name and probability to remember stuff
    public chores: Chore[];
    private _happiness: boolean;
    private _oldMan: Male;
    private _partner: Partner;
    private _currentChoreCount: number = 0;
    //
    constructor(person: Male, chores: Chore[], partner?: Partner) {
        this._oldMan = person;
        this._partner = partner;
        this._chores = chores || ['dishes'];
    }
    public remembersStuff(stuff?: Stuff = LifeAsOldMan. DEFAULT_STUFF) : boolean {
        const doesRememberStuff: boolean =  Math.random() >= stuff.rememberProbability;
        return doesRemember
    }
    public currentChoreInProgress(): string | null {
        let c: Chore = this._chores[this._currentChoreCount];
        try { this._partner.delegateChore(c); }
        catch (e) { c = null; }
        return c.name;
    }
}
```

DEMO