CODE 301

Intermediate Software Development

AGENDA

- Assignment review
- Functional Programming

FUNCTIONAL PROGRAMMING

SIMPLE != EASY

- Rich Hickey

Making things simple is not the same as doing things the easy way. It can take **effort** and **design** to make your program simple.

Simplicity is an **overall goal** of functional programming.

“Simplicity is a prerequisite for Reliability” - Edsger Dijkstra

Why is that? Why are functions simple?

Review slides here for more concepts: http://www.slideshare.net/evandrix/simple-made-easy

Complex is the opposite of simple. Complexity is our enemy.

“Simplicity is the ultimate sophistication” - Leonardo da Vinci

"SOMETIMES, THE ELEGANT IMPLEMENTATION IS JUST A FUNCTION. NOT A METHOD. NOT A CLASS. NOT A FRAMEWORK. JUST A FUNCTION."

- John Carmack

You don't always need an object hierarchy or to attach all functions to objects

"OBJECT-ORIENTED PROGRAMMING IS AN EXCEPTIONALLY BAD IDEA WHICH COULD ONLY HAVE ORIGINATED IN CALIFORNIA."

– Edsger Dijkstra

Not everybody likes OO. Edsger Dijkstra is a famous computer scientist

I don't agree with OO being bad, necessarily, but it can be overdone.

> “THE PROBLEM WITH OBJECT-ORIENTED LANGUAGES IS THEY’VE GOT ALL THIS IMPLICIT ENVIRONMENT THAT THEY CARRY AROUND WITH THEM. YOU WANTED A BANANA BUT WHAT YOU GOT WAS A GORILLA HOLDING THE BANANA AND THE ENTIRE JUNGLE.”
>
> — *Joe Armstrong*

Known as the gorilla banana problem

FUNCTIONAL PROGRAMMING

- Why?
  - Long tradition going back to Lisp (vs Fortran)
  - Has been primarily in academia but strongly resurgent in industry
  - For many hiring managers, a signal that you know what you're doing
  - Cleaner code - easier to reason about
  - Scalable and Performant on multi-core systems, large volumes of data

Lambda calculus vs turning machine

FUNCTIONAL PROGRAMMING

- What is it? No one “standard” for functional but includes:
  - Declarative vs. Imperative code
  - Stateless (pure) functions
  - Immutability
  - First-class Functions and Currying
- JavaScript was almost Scheme. It’s actually a combo of Scheme and Self
  - in 1994 Brendan Eich wanted a functional language for the browser but under competitive pressure from Java introduced OO features into the language

We almost got a FP language in the browser. Instead, because of competition with Java, we got a hybrid.

Many people start using JS as an OO language and then discover the secret hidden identity as a versatile FP tool.

We will explain all of these concepts over the next few slides

DECLARATIVE VS IMPERATIVE

- Describe **WHAT** you want

vs

- **HOW**: The steps to get it done

Declarative: language that describes relationships between variables in terms of functions or inference rules, and the language executor (interpreter or compiler) applies some fixed algorithm to these relations to produce a result.

Contrast with **imperative** languages which specify explicit manipulation of the computer's internal state; or **procedural** languages which specify an explicit sequence of steps to follow.

source: http://codenugget.co/2015/03/05/declarative-vs-imperative-programming-web.html

for example (next slide)

IMPERATIVE EXAMPLE

$$s = \sum_{x=1}^{N} x^2 = 1^2 + 2^2 + 3^3 + \ldots + N^2$$

```
function sumOfSquares(nums) {
  var i, sum = 0, squares = [];
  for (i = 0; i < nums.length; i++) {
    squares.push(nums[i]*nums[i]);
  }

  for (i = 0; i < squares.length; i++) {
    sum += squares[i];
  }

  return sum;
}

console.log(sumOfSquares([1, 2, 3, 4, 5]));
```

Looks familiar, right?
But what potential problems are there?

out of bounds errors
scope of i, other variables

DECLARATIVE EXAMPLE

$$s = \sum_{x=1}^{N} x^2 = 1^2 + 2^2 + 3^3 + \ldots + N^2$$

```
function sumOfSquares2(nums) {
  return nums
    .map(function(num) { return num * num; })
    .reduce(function(start, num) { return start + num; }, 0)
    ;
}

console.log(sumOfSquares2([1, 2, 3, 4, 5]));
```

explain how map and reduce work - demo each by itself in Chrome console

no mutable variables
no extra code keep track of
shorter = easier to debug (reason about)

FUNCTIONAL PROGRAMMING

- Functional features built in to JavaScript (ECMA 5 standard)
  - Array
    - .forEach
    - .some and .every
    - **.concat**
    - **.filter**
    - **.map**
    - **.reduce**

I've highlighted the most important ones to understand for functional programming

Demo Array.prototype.concat and .filter in Chrome console

https://github.com/codefellows/sea-301d1/blob/master/class-07-functional-programming/examples/array.concat.js

https://github.com/codefellows/sea-301d1/blob/master/class-07-functional-programming/examples/array.filter.js

PURE (STATELESS) FUNCTIONS

```
// pure (stateless)

function square(x) {
  return x * x;
}

function squareAll(items) {
  return items.map(square);
}

//  impure (stateful)

function square(x) {
  updateXinDatabase(x);
  return x * x;
}

function squareAll(items) {
  var i;
  for (i = 0; i < items.length; i++) {
    items[i] = square( items[i] );
  }
}
```

Pure functions (*ƒ*) give you predictable guarantees. They have no side effects.

Pure *ƒ* return value depends solely on the value of their arguments
Given the same input, you get the same output
Pure *ƒ* do not modify the values passed to them

Network, filesystem, or database calls are impure because they have side effects. ASK can you think of another system that is impure? (answer: modifying the DOM)

Another word for pure *ƒ* is stateless - they do not change the state of any object

demo Array.prototype.slice (pure)
demo Array.prototype.splice (impure - mutable)

Another big win for pure functions is that they are cacheable (we will get to memorization later)

IMMUTABILITY

- “Shared mutable state is the root of all evil.” - Pete hunt
- There are libraries for immutability in JS, but not required
  - ImmutableJS, Mori, Deep-freeze
- Object.freeze()
- Why?
  - Limiting the amount of things that change gives focus
  - Take away opportunities for things to be unintentionally modified
- Cons
  - Harder, (but simpler). Memory usage (maybe)

In some FP languages (Haskell, Scala, Erlang, Elm) all values are Immutable by default. They couldn’t change even if you wanted to. Why is this a good thing?

Again it comes down to predictable guarantees. Side effects in your code are problematic.

For Object.freeze() note that values that are objects can still be modified, unless they are also frozen. See the MDN page for deep-freeze polyfill or use the deepfreeze package

Demo Object.freeze()

FIRST CLASS FUNCTIONS

- Also called higher-order functions or λ
- In JS, all functions are objects
- You've already been using these in callbacks, etc.
- Enable Abstraction and Composability

```
// pure (stateless)

function square(x) {
  return x * x;
}

function squareAll(items) {
  return items.map(square);
}
```

Actually, lambda is an anonymous function.

square is a first class function, because it is being passed as a parameter to SquareAll

Abstraction and Composability are like super powered lego bricks. They allow you to think about the important parts of your code, and combine things together.

CALL AND APPLY FUNCTIONS

- Both invoke functions
- With **call**, you provide a **comma** separated list of arguments
- With **apply**, you provide an **array** (arguments)
- Mnemonic to remember
  - **C** is for **comma** - Function.prototype.**call**()
  - **A** is for **array** - Function.prototype.**apply**()

Demo call:
https://github.com/codefellows/sea-301d1/blob/master/class-07-functional-programming/examples/rot13.js

Demo apply:
Math.max (only accepts arguments by comma, use Math.max.apply)

errors:
Math.max([1,2,3,4])
works:
Math.max(1,2,3,4)
Math.max.apply(null,[1,2,3,4])

THE ARGUMENTS ARRAY-LIKE OBJECT

- how does **apply** work under the hood?
- JavaScript has a special array-like object called Arguments
- it's not an array - does not have all the array methods
- but it does kinda act like one

FUNCTION.PROTOTYPE.APPLY — AND MEMOIZATION

- How can we cache the result of pure functions?
- **memoization** is an optimization technique used to speed up functions by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

```
var memoize = function(f) {
  var cache = {};

  return function() {
    var arg_str = JSON.stringify(arguments);
    cache[arg_str] = cache[arg_str] || f.apply(f, arguments);
    return cache[arg_str];
  };
};
```

Remember we were talking about pure functions being cacheable?

CURRYING AND PARTIALLY APPLIED FUNCTIONS

- Arity = the number of arguments in a function signature
- Functions are Curried if they have an Arity of 1
- Curried functions are useful because they can be re-used
- Think of them as functions with some ‘pre-filled’ arguments

```
var add = function(x) {
  return function(y) {
    return x + y;
  };
};

var increment = add(1);
var addTen = add(10);

increment(2);
// 3

addTen(2);
// 12
```

USING .CALL AND .APPLY IN CURRY

```
var curry = function(uncurried) {
  var parameters = Array.prototype.slice.call(arguments, 1);
  return function() {
    return uncurried.apply(this, parameters.concat(
      Array.prototype.slice.call(arguments, 0)
    ));
  };
};
```

Demo .call and .apply to create a curry function

https://github.com/codefellows/sea-301d1/blob/master/class-07-functional-programming/examples/curried-functions.js

FUNCTIONAL BLOG DEMO

Demo adding features to blog

- statistics
https://github.com/codefellows/code-301/blob/class-07/blog/scripts/stats.js

(start from the bottom and work up explaining each function)

FUNCTIONAL PROGRAMMING

- There's much more to discover!
  - https://lodash.com
  - https://drboolean.gitbooks.io/mostly-adequate-guide/
  - http://reactivex.io/learnrx/
  - http://www.infoq.com/presentations/Simple-Made-Easy
  - Predicates, Optionals, Functors, oh my!

Mention and recommend lodash - show CDNJS.com usage

```
Array.prototype.flatMap = function(lambda) {
    return Array.prototype.concat.apply([], this.map(lambda));
};
```

RECAP

RECAP

- Functional programming will make your programs more understandable, maintainable, reliable, and performant.
- Speaking from experience, it can make a difference in an interview, too.