EECS 368
Programming Language Paradigms

David O. Johnson

Fall 2022

Reminders

- Assignment 1 due: 11:59 PM, Wednesday, September 7
- Assignment 2 due: 11:59 PM, Monday, September 19

Any Questions?

In-Class Problem Solution

- 3-(8-29) In-Class Problem Solution.pptx

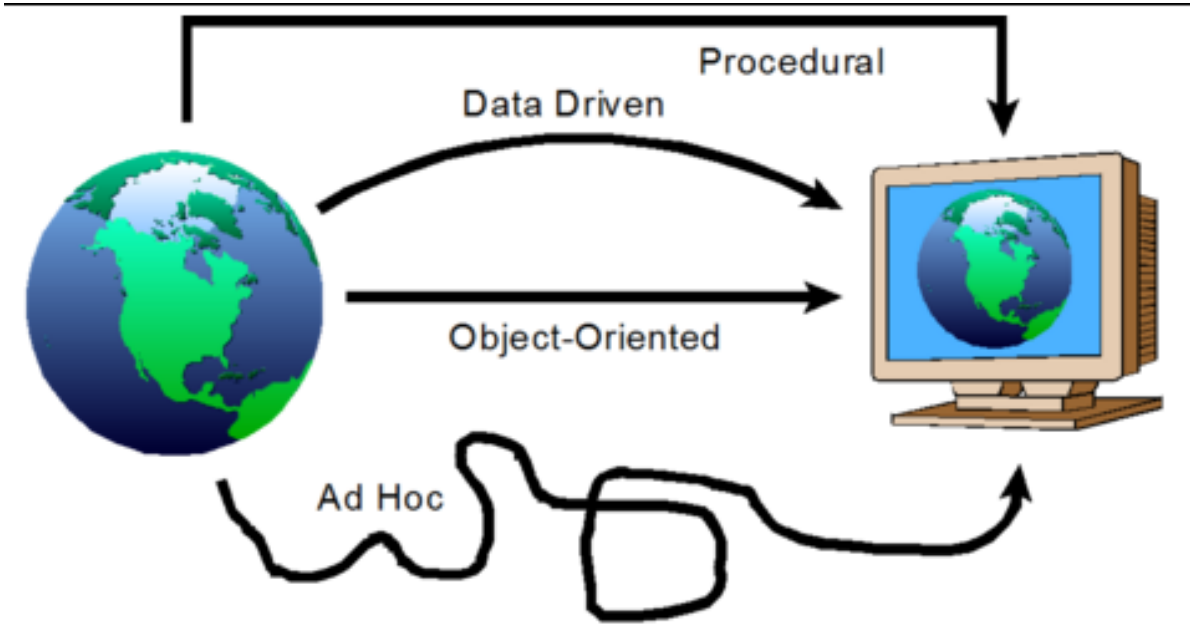
Any Questions?

Function-Oriented Design Paradigm

- Before we dive into JavaScript functions, we need to back away and see why functions are so important in JavaScript.
- They are important in JavaScript because the language was designed to support the function-oriented design paradigm.

What Is a Design Paradigm?

- A paradigm is an example or pattern that that can be copied.
- Paradigms consist of "a set of assumptions, concepts, values, and practices that constitutes a way of viewing reality for the community that shares them, especially in an intellectual discipline."
- When applied to software development, a paradigm guides the way that developers view a given problem and organize the solution.
- For example, here are 4 software development paradigms illustrated as paths between the real world and a working program:

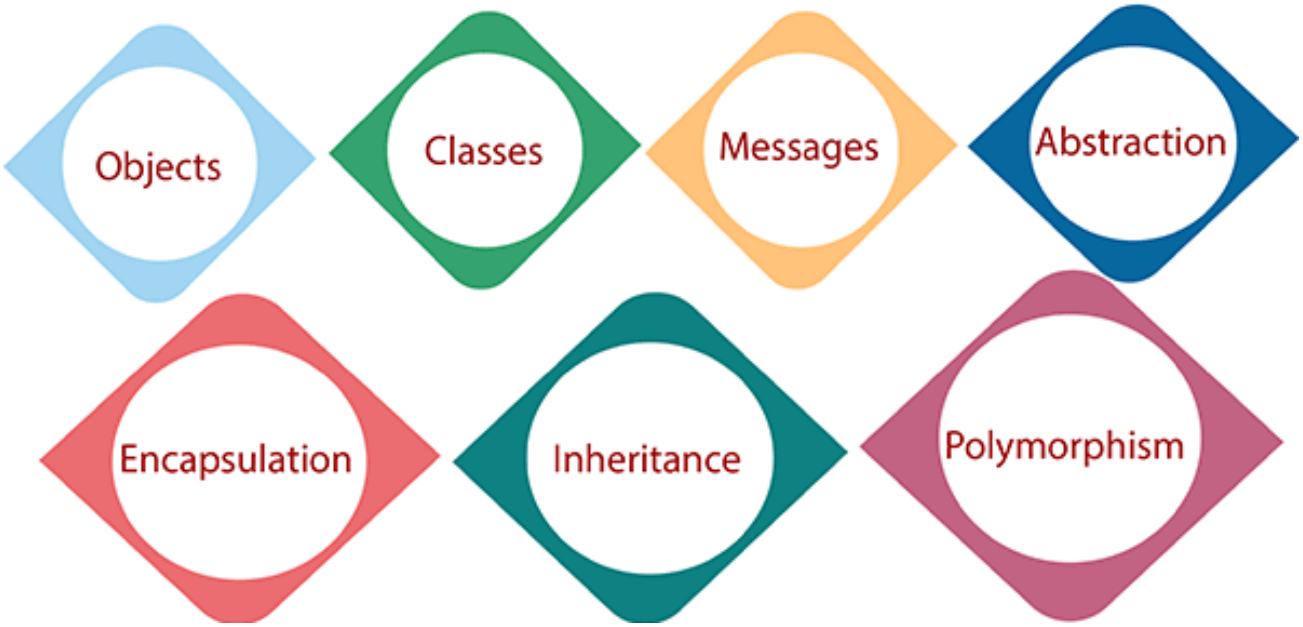


Some Design Paradigms

- Object-oriented design (EECS 168 & 268)
- Function-oriented design (EECS 368)
- Aspect-oriented design (EECS 448)
- Data-structured centered design (EECS 448)

Object-Oriented Design

- For object-oriented languages (e.g., C++ and Java)
- Main steps:
  - Create class diagrams
  - Formats of attributes are determined
  - Methods are assigned to relevant classes



Function-Oriented vs. Object-Oriented Design

Function-Oriented Design:

- The system is designed from a functional viewpoint.
- Function oriented comes from math:

$$y = f(x,y) = x \cdot x + y \cdot y \text{ (sum of squares)}$$

- JavaScript and C are examples of Function-Oriented programming languages.

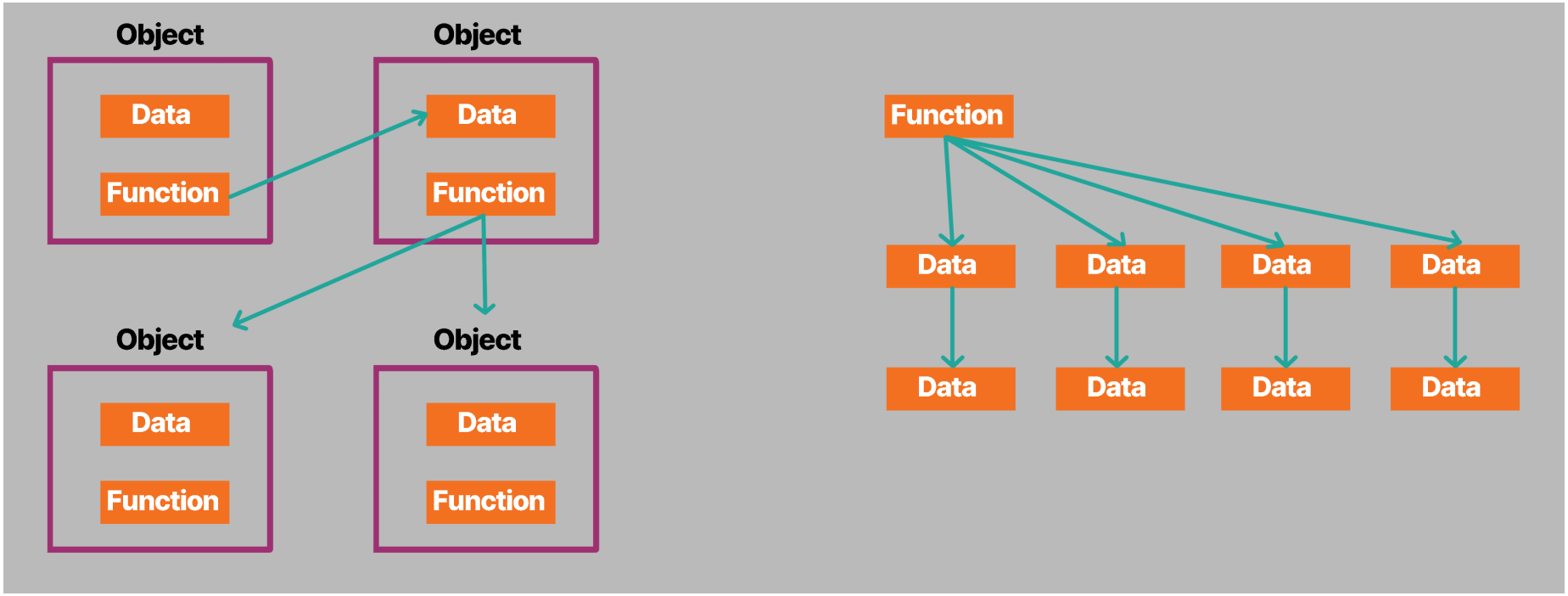
Object-Oriented Design:

- Begins with an examination of the real-world objects.
- Focuses on data that are to be manipulated by the object
- Not on the function performed by the object
- Orthogonal to Function-Oriented Design
- C++ and Java are examples of Object-Oriented programming languages

Object-Oriented vs. Function-Oriented Design

Object-Oriented Design

Function-Oriented Design



Function-Oriented Design

- Oldest design paradigm – been around since first computer programs written
  - OK – yes that is how I learned to program (and still do for the most part)
  - But, I know how to do object-oriented programming and design
    - My PhD research was all done in Java
    - I have taught C++
- Thousands of systems have been developed using this approach
- Supported by most programming languages
  - Biggest exception: Java

How to Do Function Oriented Design

Ways for functions to be introduced into programs:

1. You find yourself writing similar code multiple times.
   - You'd prefer not to do that.
   - Having more code means more space for mistakes to hide and more material to read for people trying to understand the program.
   - So you take the repeated functionality, find a good name for it, and put it into a function.
2. You find you need some functionality that you haven't written yet and that sounds like it deserves its own function.
   - You'll start by naming the function, and then you'll write its body.
   - You might even start writing code that uses the function before you actually define the function itself.

Any Questions?

JavaScript Functions

Recall:

- A function in JavaScript is a set of statements that performs a task or calculates a value.
- It should take some input and return an output where there is some obvious relationship between the input and the output.
- Note that we have three functions:
  - preload
  - create
  - update
- () – contains any variables or arguments passed to the function as input
- {} – contains code for functions
- There is no code yet

```
var config = {
    type: Phaser.AUTO,
    width: 800,
    height: 600,
    scene: {
        preload: preload,
        create: create,
        update: update
    }
};

var game = new Phaser.Game(config);

function preload ()
{
}

function create ()
{
}

function update ()
{
}
```

Defining a Function

- A function is created with an expression that starts with the keyword function.
- Functions have a set of parameters and a body, which contains the statements that are to be executed when the function is called.
- The function body must always be wrapped in braces, even when it consists of only a single statement.

```
const square = function(x) {
  return x * x;
};

console.log(square(12));
// → 144
```

Defining a Function

- A function can have multiple parameters or no parameters at all.
- Parameters to a function behave like regular variables, but their initial values are given by the caller of the function, not the code in the function itself.
- makeNoise does not list any parameter names, whereas power lists two.

```
const makeNoise = function() {
  console.log("Pling!");
};

makeNoise();
// → Pling!

const power = function(base, exponent) {
  let result = 1;
  for (let count = 0; count < exponent; count++)
{
    result *= base;
  }
  return result;
};

console.log(power(2, 10));
// → 1024
```

Defining a Function

- Some functions produce a value, such as power and square, and some don't, such as makeNoise, whose only result is a side effect.
- A return statement determines the value the function returns.
- When the JavaScript interpreter comes across a return, it immediately jumps out of the current function and gives the returned value to the code that called the function.
- A return keyword without an expression after it will cause the function to return undefined.
- Functions that don't have a return statement at all, such as makeNoise, similarly return undefined.

```
const makeNoise = function() {
  console.log("Pling!");
};

makeNoise();
// → Pling!

const power = function(base, exponent) {
  let result = 1;
  for (let count = 0; count < exponent; count++)
{
    result *= base;
  }
  return result;
};

console.log(power(2, 10));
// → 1024
```

Function Declaration

- A function declaration is a slightly shorter way to create a function binding.

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

- The statement defines the binding square and points it at the given function.
- It is slightly easier to write and doesn't require a semicolon after the function.

Arrow Functions

- There's a third notation for functions, which looks very different from the others.
- Instead of the function keyword, it uses an arrow (=>) made up of an equal sign and a greater-than character (not to be confused with the greater-than-or-equal operator, which is written >=).

```
const power = (base, exponent) => {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
};
```

- The arrow comes after the list of parameters and is followed by the function's body.
- It expresses something like "this input (the parameters) produces this result (the body)".

Arrow Functions

- When there is only one parameter name, you can omit the parentheses around the parameter list.
- If the body is a single expression, rather than a block in braces, that expression will be returned from the function.
- So, these two definitions of square do the same thing:

```
const square1 = (x) => { return x * x; };
const square2 = x => x * x;
```

- When an arrow function has no parameters at all, its parameter list is just an empty set of parentheses.

```
const horn = () => {
  console.log("Toot");
};
```

Variable Function Binding

- Typically a function binding usually simply acts as a name for a specific piece of the program.
- Such a binding is defined once and never changed, i.e., it is a constant, like a constant variable.
- However, in JavaScript a function binding that holds a function is still just a regular binding and can be assigned a new value, like so:

```
let launchMissiles = function() {
  missileSystem.launch("now");
};
if (safeMode) {
  launchMissiles = function() {/* do nothing */};
}
```

- This is a powerful feature of JavaScript!

Recursion

- JavaScript supports recursion:

```
function power(base, exponent) {
  if (exponent == 0) {
    return 1;
  } else {
    return base * power(base, exponent - 1);
  }
}

console.log(power(2, 3));
// → 8
```

- Recall that recursion allows you express a function closer to the way mathematicians define a function.
- Sometimes recursion describes a concept more clearly than the looping variant.
- But, in typical JavaScript implementations, it's about three times slower than the looping version.

Functions and Side Effects

- Functions can be roughly divided into those that are called:
  - for their side effects
  - for their return value
- Though it is definitely possible to both have side effects and return a value.

- For example, this one is called for its side effect:

  it prints a line

```
function printZeroPaddedWithLabel(number, label) {
  let numberString = String(number);
  while (numberString.length < 3) {
    numberString = "0" + numberString;
  }
  console.log(`${numberString} ${label}`);
}
```

Functions and Side Effects

- The second version, zeroPad, is called for its return value.
- It is no coincidence that the second is useful in more situations than the first.
- Functions that create values are easier to combine in new ways than functions that directly perform side effects.

```
function zeroPad(number, width) {
  let string = String(number);
  while (string.length < width) {
    string = "0" + string;
  }
  return string;
}
```

Any Questions?

Bindings and Scopes

- Each binding (e.g., variable or function) has a scope, which is the part of the program in which the binding is visible.
- Same as in C++ and almost all programming languages
- Scopes are determined by whether the binding is defined:
  - Inside a function
  - Inside a block of code delimited by curly braces {}
- Global Scope
  - For bindings defined outside of any function or block, the scope is the whole program—you can refer to such bindings wherever you want.
- Local Scope
  - Bindings created for function parameters or declared inside a function can be referenced only in that function.
  - Every time the function is called, new instances of these bindings are created.
  - This provides some isolation between functions—each function call acts in its own little world (its local environment) and can often be understood without knowing a lot about what's going on in the global environment.

Bindings and Scopes

- Bindings declared with let and const are in fact local to the block that they are declared in, so if you create one of those inside of a loop, the code before and after the loop cannot "see" it.
- Bindings declared with var are visible throughout the whole function (not block) that they appear in—or throughout the global scope, if they are not in a function.

```
let x = 10;
if (true) {
  let y = 20;
  var z = 30;
  console.log(x + y + z);
  // → 60
}
// y is not visible here
console.log(x + z);
// → 40
```

Nested Scope

- JavaScript distinguishes not just global and local bindings.
- Blocks and functions can be created inside other blocks and functions, producing multiple degrees of locality, or nested scopes.
- For example, the hummus function—which outputs the ingredients needed to make a batch of hummus—has another function inside it called ingredient.
- The code inside the ingredient function can see the **factor** binding from the outer function.
- But its local bindings, such as **unit** or **ingredientAmount**, are not visible in the outer function.

- Don't get worried – this works just like it does in C++ and Python … and pretty much all languages!

```
const hummus = function(factor) {
  const ingredient = function(amount, unit, name) {
    let ingredientAmount = amount * factor;
    if (ingredientAmount > 1) {
      unit += "s";
    }
    console.log(`${ingredientAmount} ${unit} ${name}`);
  };
  ingredient(1, "can", "chickpeas");
  ingredient(0.25, "cup", "tahini");
  ingredient(0.25, "cup", "lemon juice");
  ingredient(1, "clove", "garlic");
  ingredient(2, "tablespoon", "olive oil");
  ingredient(0.5, "teaspoon", "cumin");
};
```

Any Questions?

Summary

- Function-oriented design paradigm vs. object-oriented design paradigm
- Ways to define a function:

```
// Define f to hold a function value
const f = function(a) {
  console.log(a + 2);
};

// Declare g to be a function
function g(a, b) {
  return a * b * 3.5;
}

// A less verbose arrow function
let h = a => a % 3;
```

- Scope:
  - Each block creates a new scope.
  - Parameters and bindings declared in a given scope are local and not visible from the outside.
  - Bindings declared with var behave differently—they end up in the nearest function scope or the global scope.

Any Questions?

In-Class Problem

For this program, what is the console output for?

1. console.log(countEs("EECS"));
2. console.log(countChar("david", "d"));

3. Add a comment to each line of code to tell what it is doing.

If you have access to a web browser console, try to figure out the answer before checking it with the console.

```
function countChar(string, ch) {
  let counted = 0;
  for (let i = 0; i < string.length; i++) {
    if (string[i] == ch) {
      counted += 1;
    }
  }
  return counted;
}

function countEs(string) {
  return countChar(string, “E");
}
```