Object-Oriented Programming & Design Patterns

CIS 1962 (Fall 2025)
October 6th, 2025

Penn Engineering

Lesson Plan

Object-Oriented Programming

5	Introduction to OOP
16	Pre-ES6: Prototypes
26	Post ES6: Classes

Design Patterns

40	Singleton Pattern
44	Observer Pattern
47	Factory Pattern
50	MVC/MVVM

Weekly Logistics

2nd Half Lecture Content Survey!

- This is a **GRADED** survey about your interest in JavaScript and web dev topics- please do it!
- We want to know what YOU are interested in learning about JavaScript- we can't cover everything!
- You will have a choice to choose a framework that you're interested in, which we will cover broadly in 2 lectures
- There are a variety of topics that may cover a full or half of a lecture for you to choose from

PollEverywhere!

We will use PollEv to take attendance and do polls during lecture. Scan this QR Code or use the link to join for attendance! (Please add your name for identification)

PollEv.com/voravichs673

Object-Oriented Programming

Introduction to Object-Oriented Programming

How do we create well-maintained and scalable code in JavaScript by leveraging OOP concepts?

Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming style that uses **classes** and **objects** to model data and behavior.

JavaScript makes extensive use of objects, but only recently introduced classes with ES6.

OOP promotes code reuse and scalability through its various features:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Abstraction/Encapsulation

Abstraction focuses on exposing only essential features and hiding unnecessary details from the user.

Encapsulation is the practice of bundling data and methods together and restricting direct access to some of the object's components.

JavaScript, while it doesn't have dedicated abstract classes or interfaces, achieves these ideas through:

- Functions
- Objects/Methods
- Scope/Closures
- Classes

Abstraction Examples

```
function calculateArea(radius) {
     return
          Math.PI * radius * radius;
}

console.log(calculateArea(5));
```

Functions

```
const bankAccount = {
     balance: 0,
     deposit(amount) {
          this.balance += amount;
     },
     getBalance() {
          return this.balance;
     }
};

bankAccount.deposit(100);
console.log(bankAccount.getBalance());
```

Objects/Methods

Encapsulation Examples

```
function makeCounter() {
  let count = 0; // not directly accessible
  return { // Creates closures
    increment() {
      count++;
    },
    getCount() { // access count variable
      return count;
    }
  };
}

const counter = makeCounter();
counter.increment();
console.log(counter.getCount()); // 1
console.log(counter.count); // undefined
```

Scope/Closures

```
class Rectangle {
  #width; // private, hidden to user
  #height; // private, hidden to user
  constructor(width, height) {
    this.#width = width;
    this.#height = height;
  }
  getArea() {
    return this.#width * this.#height;
  }
}

const rect = new Rectangle(10, 5);
console.log(rect.getArea());
```

Classes

Inheritance

Inheritance lets one class (a subclass) acquire the properties and methods of another class (a superclass).

Before classes in ES6, JavaScript used to use prototypal inheritance, but now it has proper class syntax and the `extends` keyword for inheritance.

```
class Animal { // Superclass
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(`${this.name} makes a
sound.`);
  }
}

class RedPanda extends Animal { // Subclass
  speak() {
    console.log(`${this.name} says wah!`);
  }
}

const klein = new RedPanda("Klein");
klein.speak(); // Klein says wah!
```

Polymorphism

Polymorphism is the practice of allowing objects to share and override behaviors. It allows the same method to function differently based on what object they act on.

JavaScript handles polymorphism in 2 ways:

- Method Overriding
- Method Overloading (not natively)

Method Overriding

Overriding lets a subclass provide a different implementation of a method in a superclass.

You need no extra syntax, just use the same name for the method within a class that extends a superclass.

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(`${this.name} makes a
sound.`);
  }
}

class RedPanda extends Animal {
  speak() { // Overrides Animal's speak()
    console.log(`${this.name} says wah!`);
  }
}

const klein = new RedPanda("Klein");
klein.speak(); // Klein says wah!
```

Method Overloading

```
class Calculator {
     // Check arguments, then do something
different depending on existing arguments
     add(a, b) {
          if (b) {
               return a + b;
          }
          return a + a;
     }
}

const calc = new Calculator();
console.log(calc.add(1)) // 2
console.log(calc.add(1,2)) // 3
```

Method overloading allows you to have multiple methods with the same name but different arguments.

JavaScript **does not natively support method overloading**, however we can simulate it through checking arguments.

POLL QUESTION

```
function createCounters() {
  let count = 0;
  return [
    function() {
      count += 1;
      return count;
    },
    function() {
      count += 2;
      return count;
    }
  ];
}

const [inc1, inc2] = createCounters();

console.log(inc1());
console.log(inc2());
console.log(inc1());
console.log(inc2());
```

What is printed to the console?

1, 3, 4, 6

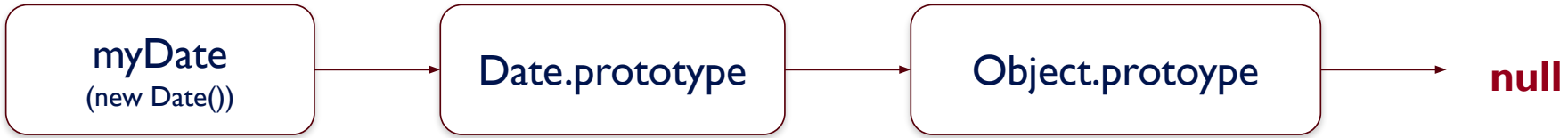
Pre-ES6: Prototypes

How did JavaScript handle OOP without using classes like other programming languages do?

JavaScript and Prototypes

All objects in JS have an internal link to another object called a **prototype**, which define shared properties and methods that other objects will inherit.

If a property or method is not found in an object itself, JavaScript will look for it in the prototypes, up the **prototype chain**, until the end of the chain (null)



Native Prototypes: Date

Let's walk up the prototype chain of Date and see the properties within:

```
const myDate = new Date();
do {
  myDate = Object.getPrototypeOf(myDate);
  console.log(
     Object.getOwnPropertyNames(myDate));
} while (myDate);
```

What happens when we call myDate.propertyIsEnumerable?

Date

```
[
  'constructor',       'toString',          'toDateString',
  'toTimeString',      'toISOString',       'toUTCString',
  'toGMTString',       'getDate',           'setDate',
  'getDay',            'getFullYear',       'setFullYear',
  'getHours',          'setHours',          'getMilliseconds',
  'setMilliseconds',   'getMinutes',        'setMinutes',
  'getMonth',          'setMonth',          'getSeconds',
  'setSeconds',        'getTime',           'setTime',
  'getTimezoneOffset', 'getUTCDate',        'setUTCDate',
  'getUTCDay',         'getUTCFullYear',    'setUTCFullYear',
  'getUTCHours',       'setUTCHours',       'getUTCMilliseconds',
  'setUTCMilliseconds', 'getUTCMinutes',    'setUTCMinutes',
  'getUTCMonth',       'setUTCMonth',       'getUTCSeconds',
  'setUTCSeconds',     'valueOf',           'getYear',
  'setYear',           'toJSON',            'toLocaleString',
  'toLocaleDateString', 'toLocaleTimeString'
]
```

Object

```
[
  'constructor',
  '__defineGetter__',
  '__defineSetter__',
  'hasOwnProperty',
  '__lookupGetter__',
  '__lookupSetter__',
  'isPrototypeOf',
  'propertyIsEnumerable',
  'toString',
  'valueOf',
  '__proto__',
  'toLocaleString'
]
```

Properties, Fields, and this

We use the term **property** to define the key-value pairs of an object.

We use the term **field** to describe the properties of classes of JavaScript.

To refer to the current instance of the class from within the object itself, you can use the keyword `this`.

```
// username and password are properties
function Account(username, password) {
  this.username = username;
  this.password = password;
}

// username and password are fields
class Account {
  constructor(username, password) {
    this.username = username;
    this.password = password;
  }
}
```

Function Constructors

```
function Account(username, password) {
  this.username = username;
  this.password = password;
}

// Creation
const goose = new Account('goose123',
'secret123');
const randy = new Account('randy456',
'mypassword');

// Access properties
console.log(goose.username); // "goose123"
console.log(randy.password);   // "mypassword"
```

Constructors are ways to create an initialize objects, often with initial properties/fields.

Before JS had classes, it could use function constructors to build objects with properties.

Prototypal Inheritance

Because JavaScript has prototype and the prototype chain, we can use these properties to perform **prototypal inheritance**.

Prototypal inheritance makes use of the fact that we can look to other objects down the prototype chain for inherited properties and methods.

Prototypal Inheritance Examples

```
function Account(username, password) {
  this.username = username;
  this.password = password;
}
```

```
Account.prototype.describe = function() {
  console.log(`Username: ${this.username}`);
};

const goose = new Account('goose123', 'secret123');
goose.describe();
```

Adding prototype methods

Prototypal Inheritance Examples

```
function VIPAccount(username, password, level) {
  Account.call(this, username, password); // Inherit properties
  this.level = level; // add new properties
}

VIPAccount.prototype = Object.create(Account.prototype);
VIPAccount.prototype.constructor = VIPAccount;

VIPAccount.prototype.vipHello = function() {
  console.log("VIP access for " + this.username);
};


const randy = new VIPAccount('randy456', 'mypassword', 'gold');
randy.vipHello() // from VIPAccount
randy.describe() // from Account
```

Adding Inheritance steps

Limitations of Prototypes

Complex Syntax: Prototypal inheritance is verbose to setup and uses specific syntax for prototype chaining (`Object.create`, `.constructor` property)

No access modifiers: All properties and methods are public

Debugging and Readability: Developers used to proper OOP syntax from other languages (like Java) may find prototypal inheritance confusing and harder to debug.

POLL QUESTION

```
function Animal(name) {
  this.name = name;
}
Animal.prototype.speak = function() {
  return `${this.name} makes a noise.`;
};

function Mammal(name) {
  Animal.call(this, name);
}
Mammal.prototype = Object.create(Animal.prototype);
Mammal.prototype.constructor = Mammal;
Mammal.prototype.speak = function() {
  return `${this.name} growls.`;
};

function Dog(name) {
  Mammal.call(this, name);
}
Dog.prototype = Object.create(Mammal.prototype);
Dog.prototype.constructor = Dog;

const dog = new Dog("Fido");
console.log(dog.speak());
console.log(dog.bark());
```

What is printed to the console?

"Fido growls." and a TypeError

Post-ES6: Classes

How did ES6's classes simplify OOP for JavaScript developers?

Classes

ES6 finally introduced classes, allowing more readable and familiar OOP code.

Classes are blueprints for creating objects, complete with proper syntax for fields, access modifiers, constructors, and inheritance.

Class Syntax

```
class User {
  static species = 'Homo sapiens';      // Public field
  #password;                            // Private field

  constructor(name, pwd) {
    this.name = name;
    this.#password = pwd;
  }

  greet() {                             // Instance method
    return `Hello, ${this.name}`;
  }

  get maskedPassword() {                // Getter
    return '*'.repeat(this.#password.length);
  }

  set password(newPwd) {                // Setter
    this.#password = newPwd;
  }

  static describe() {                   // Static method
    return 'All users are humans.';
  }
}
```

Classes feature:

- Class constructors
- Inheritance with `extends` and `super`
- Getters/Setters
- Public and private fields (ES2022+)
- Instance, Static, and Private (ES2022+) methods

Class Constructor

Classes provide a direct way to interface with the **constructor** method in order to perform initialization tasks.

Only 1 constructor can be defined.

Often, you will use super in subclasses to call the superclass's constructor.

```
class Account {
  constructor(name, pwd) {
    this.name = name;
    this.#password = pwd;
  }
}

class VIPAccount extends Account {
  constructor(name, pwd, level) {
    super(name, pwd);
    this.level = level;
  }
}
```

Inheritance: extends and super

The keyword `extends` allows you to define a subclass that inherits from a superclass.

Additionally, you can use the `super` keyword to call the constructor of the superclass to say, set initial fields.

This greatly simplifies the syntax of inheritance over prototypal inheritance.

```
class Account {
  constructor(name, pwd) {
    this.name = name;
    this.#password = pwd;
  }
}

class VIPAccount extends Account {
  constructor(name, pwd, level) {
    super(name, pwd);
    this.level = level;
  }
}

const vip = new VIPAccount("randy",
"mypassword", "gold")
```

Polymorphism with Classes

```
class Animal {
  speak() {
    console.log(`${this.name} makes a
sound.`);
  }
}

class RedPanda extends Animal {
  speak() { // Overrides Animal's speak()
    console.log(`${this.name} says wah!`);
  }
}

const klein = new RedPanda("Klein");
klein.speak(); // Klein says wah!
```

Classes provide easier overriding of methods by just allowing you to define a method with the same name in the subclass, no need for prototype assignment.

Getters & Setters

```
class User {
  #password;

  constructor(name, pwd) {
    this.name = name;
    this.#password = pwd;
  }

  get maskedPassword() {
    return '*'.repeat(this.#password.length);
  }

  set password(newPwd) {
    this.#password = newPwd;
  }
}

const user = new User("randy", "mypassword");
console.log(user.maskedPassword) // **********
user.password = "newpassword"
console.log(user.maskedPassword) // **********
```

Getters and **Setters** improve upon encapsulation by restricting access and updating to certain fields through specific `get` and `set` methods.

These is classic OOP syntax that easier maintenance of class and object properties.

Public & Private fields (ES2022)

All properties we've used before were public- which is problematic for encapsulation and data integrity

Access modifiers for fields were introduced in ES2022 to fix this issue

```
class User {
  static species = 'Homo sapiens';
  #password; // Private

  constructor(name, pwd) {
    this.name = name;
    this.#password = pwd; // Private
  }

  get maskedPassword() {
    return '*'.repeat(this.#password.length);
  }
}

const user = new User("randy", "mypassword");
console.log(user.password) // undefined
console.log(user.maskedPassword) // **********
```

Instance vs. Static

Instance Fields/Methods

Instance fields/methods belong to the objects create from classes themselves (instances). To access fields and methods, you call them on the instances themselves.

```
class User {
  species = 'Homo sapiens';

  describe() {
     console.log("This is a user")
  }
}

const user = new User();
console.log(user.species)
console.log(user.describe())
```

Static Fields/Methods

Static fields/methods belong to the class itself, not the instances. They can be accessed by calling upon the class directly.

```
class User {
  static species = 'Homo sapiens';

  static describe() {
     console.log("This is a user")
  }
}

console.log(User.species)
console.log(User.describe())
```

POLL QUESTION

```
class SecretKeeper {
  #secret = 'hidden';

  revealSecret() {
    return this.#secret;
  }

  static getSecret(instance) {
    return instance.#secret;
  }
}

class ChildKeeper extends SecretKeeper {
  revealParentSecret(instance) {
    return instance.#secret; // (3)
  }
}

const keeper = new SecretKeeper();
const child = new ChildKeeper();

console.log(keeper.revealSecret());                // (1)
console.log(SecretKeeper.getSecret(keeper));       // (2)
console.log(child.revealParentSecret(keeper));     // (3)
console.log(keeper.#secret);                       // (4)
```

Which lines will throw errors and why?

Lines (3) and (4) throw errors. Private fields are inaccessible from outside the class body, and subclasses cannot access parent class private fields.

POLL QUESTION

```
class Rectangle {
  constructor(width, height) {
    this.width = width;
    this.height = height;
  }

  area() {
    return this.width * this.height;
  }

  describe() {
    return `Rectangle with area ${this.area()}`;
  }
}

class ColoredRectangle extends Rectangle {
  constructor(width, height, color) {
    super(width, height);
    this.color = color;
  }

  describe() {
    return `${super.describe()} and color ${this.color}`;
  }

  doubleArea() {
    return super.area() * 2;
  }
}

const cr = new ColoredRectangle(3, 5, "blue");
console.log(cr.describe());
console.log(cr.doubleArea());
console.log(cr.area());
```

What is printed to the console? (in 1 line)

```
Rectangle with area 30, 30, 15
```

5-Minute Break!

Design Patterns

What are Design Patterns?

Throughout the history of web development and programming, standardized **design patterns** have been codified as reusable solutions to coding and organization problems.

These design patterns help with:

- Maintainability and Scalability of code
- Reusable and Modular code
- Collaboration and the "vocabulary" of projects
- Providing proven, tested Solutions

Singleton Pattern

Single instances of objects and classes with global access

Singleton Pattern

The **Singleton Pattern** ensures classes or objects will only have one instance, but will provide the program global access to that instance.

This is used for many purposes:

- Prevent duplication (single database connection, single UI managers)
- Centralize app configurations and resources

Singleton with Objects

```
const AppConfig = {
  theme: "dark",
  language: "en",
};

AppConfig.language = "fr";
```

Objects

```
const Singleton = (function () {
  let instance;
  function createInstance() {
    return { id: Math.random() };
  }
  return {
    getInstance() {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
  };
})();

const a = Singleton.getInstance();
const b = Singleton.getInstance();
console.log(a === b); // true
```

Closures

Singleton with Classes

```
class ModalManager {
  static instance;
  constructor() {
    if (ModalManager.instance) {
      return ModalManager.instance;
    }
    this.modals = [];
    ModalManager.instance = this;
  }

  open(modal) {
    this.modals.push(modal);
  }
}

// Usage:
const m1 = new ModalManager();
const m2 = new ModalManager();
console.log(m1 === m2); // true
```

Observer Pattern

Subjects notify observers about when they change

Observer Pattern

The **Observer Pattern** allows one object, as a subject, to notify a list of dependents, as observers, automatically when its state changes.

This pattern is useful for UI updates and reactivity in web applications, and some frameworks even have this pattern built into their systems (React states)

Observer Examples

```
button.addEventListener('click', handler)
```

Event Handlers

```
function Counter() {
  const [count, setCount] =
React.useState(0);

  return <button onClick={() =>
setCount(count + 1)}>{count}</button>;
}
```

React States

```
class Subject {
  constructor() {
    this.observers = [];
  }
  subscribe(fn) {
    this.observers.push(fn);
  }
  unsubscribe(fn) {
    this.observers =
this.observers.filter(obs => obs !== fn);
  }
  notify(data) {
    this.observers.forEach(fn => fn(data));
  }
}

const subject = new Subject();

function logger(data) {
  console.log("Logged:", data);
}

subject.subscribe(logger);
subject.notify("Hello World!"); // Output:
Logged: Hello World!
```

Factory Pattern

Create objects dynamically without exposing implementation details

Factory Pattern

The **Factory Pattern** allows you to create objects without exposing implementation details, through a defined factory function or method.

This pattern decouples object creation from code that uses the object, and improves scalability by making it simple to add new configurations into the factory.

Factory Examples

```
function createUser(type, name) {
  if (type === "admin") {
    return
      { role: "admin", name, canDelete: true };
  } else if (type === "guest") {
    return
      { role: "guest", name, canDelete: false };
  }
  return
      { role: "user", name, canDelete: false };
}

// Usage:
const admin = createUser("admin", "Voravich");
const guest = createUser("guest", "Randy");

console.log(admin.role); // "admin"
console.log(guest.canDelete); // false
```

Objects

```
class CarFactory {
  static createCar(type) {
    if (type === "sports") {
      return { type, maxSpeed: 200 };
    }
    if (type === "family") {
      return { type, maxSpeed: 120 };
    }
    return { type: "standard", maxSpeed: 150 };
  }
}

// Usage:
const sportsCar = CarFactory.createCar("sports");
console.log(sportsCar.maxSpeed); // 200
```

Classes

MVC/MVVM Pattern

Linking data and interface using Controllers and ViewModels

Models and Views

MVC (Model-View-Controller) and **MVVM (Model-View-ViewModel)** are patterns that split programs and projects into 3 parts, 2 of which are shared between both:

- **Model**: A data layer that handles business and application logic. Handles data changes that may eventually change the view.
- **View**: The user interface, that handles only visual elements, and changes depending on data changes in the model

Models and Views: Example

```
class TodoListModel {
  constructor() {
    this.todos = [];
  }

  addTodo(text) {
    this.todos.push({ text, done: false });
  }

  getTodos() {
    return this.todos;
  }

  markDone(index) {
    if (this.todos[index]) {
      this.todos[index].done = true;
    }
  }
}
```

Model: Business Logic

```
function renderTodos(todoList) {
  const ul = document.createElement('ul');
  todoList.forEach((todo, i) => {
    const li = document.createElement('li');
    li.textContent = todo.done ? `[✔]
${todo.text}` : todo.text;
    ul.appendChild(li);
  });
  const container =
document.getElementById('todoContainer');
  container.innerHTML = '';
  container.appendChild(ul);
}
```

View:Visual Rendering

Controllers and ViewModels

MVC and MVVM differ in how view and model are linked: through a **Controller** (MV**C**) or a **ViewModel** (MV**VM**)

Controllers are bridges between the view and model for MVC. They receive information from the model to update the view.

In MVVM, Views are often bound directly to the ViewModel, allowing **automatic** updating of the view depending on changes to the model or and visual data present in the ViewModel

MVC Example

```
class TodoListModel {
  constructor() {
    this.todos = [];
  }

  addTodo(text) {
    this.todos.push({ text, done: false });
  }
}
```

```
class TodoController {
  constructor(model, view) {
    this.model = model;
    this.view = view;
  }

  addTodo(text) {
    this.model.addTodo(text);
    this.view(this.model.todos);
  }
}
```

```
function renderTodos(todoList) {
  const ul = document.createElement('ul');
  todoList.forEach((todo, i) => {
    const li = document.createElement('li');
    li.textContent = todo.done ? `[✔]
${todo.text}` : todo.text;
    ul.appendChild(li);
  });
  const container =
document.getElementById('todoContainer');
  container.innerHTML = '';
  container.appendChild(ul);
}
```

```
// Usage
const model = new TodoListModel();
const controller = new TodoController(model,
renderTodos);

controller.addTodo('Learn JS Patterns');
// Renders: Learn JS Patterns
```

MVVM Example

```
class TodoListModel {
  constructor() {
    this.todos = [];
  }

  addTodo(text) {
    this.todos.push({ text, done:
false });
  }
}
```

```
function renderTodos(todoList) {
  const ul =
document.createElement('ul');
  todoList.forEach((todo, i) => {
    const li =
document.createElement('li');
    li.textContent = todo.done ? `[✔]
${todo.text}` : todo.text;
    ul.appendChild(li);
  });
  const container =
document.getElementById('todoContainer
');
  container.innerHTML = '';
  container.appendChild(ul);
}
```

```
class TodoListViewModel {
  constructor(model) {
    this.model = model;
    this.observers = [];
  }

  get todos() {
    return this.model.todos;
  }

  addTodo(text) {
    this.model.addTodo(text);
    this.notify();
  }

  subscribe(observerFn) {
    this.observers.push(observerFn);
  }

  notify() {
    this.observers.forEach(fn =>
fn(this.todos));
  }
}
```

```
// Usage
const model = new TodoListModel();
const viewModel = new
TodoListViewModel(model);

// Bind the View to the ViewModel
viewModel.subscribe(renderTodos);


viewModel.addTodo('Learn MVVM!');
// "Learn MVVM!" appears in the UI
```

Why MVVM? Frameworks!

Modern JavaScript frameworks often uses MVVM or MVVM-like patterns to allow for views to be updated easily.

For instance, React features a one-way data flow version of MVVM, that lets changes in states(ViewModel) change UI (view).

```
function TodoList() {
  // ViewModel & Model: states, setting, and adding todos
  const [todos, setTodos] = React.useState([]);

  function addTodo(newTodo) {
    setTodos([...todos, newTodo]);
  }

  // View: binds to state/ViewModel via "todos" variable
  return (
    <div>
      <ul>
        {todos.map(todo => <li key={todo}>{todo}</li>)}
      </ul>
      <button onClick={() => addTodo('Learn JS
Patterns')}>Add</button>
    </div>
  );
}
```

OPEN POLL QUESTION

You are building a collaborative online document editor. Multiple users may edit and view a document simultaneously. When one user makes a change (like typing a character), that change must immediately be reflected in real-time for all users currently viewing the document.

Which of the following design patterns is the best fit for implementing the mechanism that keeps all users' views synchronized with the document's state?

Observer Pattern

OPEN POLL QUESTION

You are implementing a game using JavaScript. Within your code, various game components will emit "signals" when certain things occur, such as a character moving into a region, or a certain boolean flag is triggered. You also want to make sure no signals get duplicated over time.

You decide to maintain a global "signal bus" that other scripts can refer to in order to detect when certain signals get triggered. What design pattern best describes this "signal bus"?

Singleton Pattern

Live* Coding: Drawing Shapes with OOP and MVC

*Much will be prewritten because it's a larger app!