A Comparative Study of Object-Oriented Principles in Creative Coding: Java AWT vs p5.js

Abhinav Mehrotra, Vishnu Vardhan Mahajan
University of Colorado Boulder
Boulder, CO 80309

Abstract

This research conducts a comparative analysis of object-oriented principles in creative coding by evaluating two environments: Java AWT and p5.js. The study examines how each platform implements key OO concepts such as classes, inheritance, and modular design in the development of interactive graphics and animations. Performance tests and usability assessments highlight the trade-offs between Java's comprehensive yet intricate approach and p5.js's streamlined, accessible methodology. The findings provide valuable insights for developers and digital artists in selecting the most suitable tool for their creative coding projects.

Introduction

In today's digital landscape, creative coding has become an essential tool for artists and developers working in interactive media, digital art, and data visualization. This research explores how object-oriented programming (OOP) principles are applied in two different environments Java AWT and p5.js each offering unique approaches to coding creative projects. Java AWT is a well-established programming language known for its strong support of OOP concepts such as classes, inheritance, and encapsulation. Its robust framework allows for detailed and precise control over program structure and behavior. In contrast, p5.js is a JavaScript library designed specifically for creative coding, providing a more accessible and streamlined framework that simplifies the process of building interactive graphics and animations. The purpose of this study is to compare these two environments by examining how each one implements core OOP principles in the context of creative projects. By conducting practical coding experiments and performance assessments, the research aims to identify the strengths and limitations of each approach. Ultimately, the findings will help developers, digital artists, and educators choose the most suitable tool for their creative coding projects, balancing factors like ease of use, performance, and code maintainability.

Literature Review

Object-Oriented Programming in Java

The fundamental concepts of classes, inheritance, abstraction, polymorphism, and encapsulation form the foundation of Java's object-oriented programming (OOP). These guidelines help programmers organize their work into modular parts, which improves the readability and maintainability of the code. Java is a popular choice in both industry and academia because of its static type system and rigorous enforcement of class-based architecture, which also support reliable, large-scale software development. Java uses frameworks like AWT, Swing, and JavaFX to generate and control visual elements in graphical or interactive applications. These libraries were originally designed for conventional desktop applications, but they have also been modified for use in creative coding. Programmers can separate user interaction, animation loops, and drawing logic into separate modules by using classes and interfaces. Because of this design pattern's facilitation of scalability and reusability, developers can expand or alter creative projects without having to completely redo the coding. However, quick prototyping—a critical component of creative coding can occasionally be hampered by the verbose syntax and stringent type restrictions. As a result, even while Java offers a strong framework for organized development, those wishing to swiftly experiment with visual or interactive art may find it difficult to understand.

Object-Oriented Programming in p5.js

In contrast to Java, p5.js is a JavaScript library designed with creative coding in focus. Although class syntax was added as syntactic sugar for prototypical inheritance in ES6, JavaScript is prototype-based rather than strictly class-based (p5.js Foundation, n.d.). p5.js extends Processing, a Java-based creative coding platform, to the online environment while retaining much of its philosophy. The development of interactive drawings is made easier by p5.js's user-friendly setup and draw methods, which free up developers and artists to concentrate on interactivity and graphics rather than complex boilerplate code. The flexibility of p5.js's approach to OOP stems from its foundation in JavaScript. ES6 syntax can be used to define classes, or developers can use conventional prototype patterns to structure their code. This

adaptability can help novices who are still learning the basics of programming and for rapid experimentation (Shiffman, 2012). Because of the library's integrated event handling (such as mouse and keyboard events) and continuous draw loop, p5.js is particularly attractive in educational settings because it lowers the entry barrier. Some sophisticated OOP features, like completely segregated namespaces or stringent type enforcement, are more difficult to implement with JavaScript. Due to Java's more stringent OOP limitations, p5.js may be less appropriate for large-scale, highly structured applications, even while it excels at accessibility and rapid prototyping.

Creative Coding Frameworks and Paradigms

The history of creative coding is extensive, dating back to John Maeda's work at the MIT Media Lab and continuing with programs like Cinder, openFrameworks, and Processing. These frameworks emphasize visual experimentation, interactive installations, and artistic expression in an effort to democratize programming. In contrast to conventional software engineering settings, creative coding frameworks frequently place an emphasis on ease of use and instantaneous visual feedback, which can encourage a more experimental and iterative development approach. An excellent illustration of this mentality is p5.js, which builds upon the fundamental concepts of Processing while utilizing JavaScript's widespread use on the web. P5.js provides a user-friendly starting point for novice programmers and artists by displaying interactive sketches straight in the browser. A more rigorous, class-based approach is provided by Java-based frameworks, such as the original Processing, which can help with a more thorough investigation of OOP concepts (Shiffman, 2012). According to, the choice of framework frequently depends on a project's magnitude, complexity, and performance requirements. Java's rigorous type system and structured OOP paradigm may be more dependable for large-scale interactive systems, whereas p5.js's quick prototyping capabilities may be more advantageous for smaller, art-focused sketches. When choosing tools for creative coding projects, it is crucial to comprehend the advantages and disadvantages of each framework due to the conflict between system robustness and convenience of use.

Methodology

To investigate how p5.js and Java AWT use object-oriented concepts in creative coding tasks, this study uses a comparative research approach. A collection of small-scale programs that incorporate form rendering, simple animations, and user interactions are created in both contexts in order to do this. The study seeks to distinguish how the fundamental elements of each platform—such as classes, inheritance, and encapsulation—are implemented by preserving comparable functionality between the two implementations.

Four important factors direct the analysis while assessing these projects. Initially, frame-rate monitoring, CPU consumption, and memory profiling are used to assess each platform's performance in handling rendering and animation operations. Second, by analyzing each environment's learning curve, setup difficulty, and debugging experience, ease of use and development speed are evaluated. Third, the study looks into how object-oriented ideas are implemented, with an emphasis on how modular, maintainable code structures are produced by utilizing classes, inheritance, and encapsulation. Lastly, by examining how readily each strategy can be expanded to support larger codebases and more intricate creative projects, scalability and maintainability are assessed.

Both quantitative and qualitative methods are used in data collecting and analysis. Lines of code, memory consumption data, and frame-rate benchmarks are examples of quantitative measurements. These unbiased metrics provide information about each environment's complexity and efficiency. The subjective elements of programming, like clarity, debugging difficulties, and workflow, are illuminated by qualitative observations obtained through code reviews, user feedback, and quick developer surveys. In order to help developers, artists, and educators choose the best tool for their creative coding requirements, the study will synthesize these data points and provide a fair assessment of each platform's advantages and disadvantages.

Coding Excersizes

1. Particle System with Forces

- **Description:** Implement a system where particles react to forces like gravity, wind, or attraction to a point.
- **Comparison Points:**
  - Ease of handling object-oriented programming for particles
  - Performance in rendering and updating many particles
  - Flexibility in adding interactivity (e.g., mouse-controlled forces)

2. L-System (Lindenmayer System) for Procedural Growth

- **Description:** Use an L-system to generate fractal-like plant growth patterns based on iterative string rewriting rules.
- **Comparison Points:**
  - Efficiency in handling recursive rule application
  - Rendering performance of complex branching structures
  - Ease of visualizing different rule sets dynamically

3. Flow Field Simulation

- **Description:** Create a vector field that influences particles, simulating natural phenomena like wind currents or ocean currents.
- **Comparison Points:**
  - Performance in calculating and applying per-pixel/per-cell flow
  - Smoothness of particle motion along the flow
  - Ease of visualizing and modifying the vector field dynamically

4. Boids (Flocking Behavior Simulation)

- **Description:** Implement Craig Reynolds' flocking algorithm with rules for separation, alignment, and cohesion to simulate bird or fish movement.
- **Comparison Points:**
  - Performance in handling multiple agents interacting in real time
  - Ease of implementing behavior rules in a structured way
  - Rendering efficiency for smooth flock motion

5. Reaction-Diffusion System (Turing Patterns)

- **Description:** Simulate chemical pattern formation using Gray-Scott reaction-diffusion equations.
- **Comparison Points:**
  - Computational efficiency in handling grid-based diffusion
  - Visual complexity achievable in both libraries
  - Flexibility in tweaking parameters for emergent patterns

Comparative Study of L-System

Our implementation of L-Systems in both Java AWT and p5.js offers an illuminating case study of how object-oriented principles manifest in different programming environments. This comparison highlights the structural and philosophical differences between these frameworks when applied to creative coding tasks.

Code Structure and Organization

The Java AWT implementation exemplifies classical object-oriented programming with a clear class hierarchy. The main `LSystem` class contains an inner `LSystemPanel` class, demonstrating encapsulation of the rendering functionality within the application structure. This nested approach creates a well-defined boundary between the UI components and the rendering logic.

In contrast, the p5.js implementation adopts a more functional and less strictly object-oriented approach. State is managed primarily through JavaScript object literals (`lSystem` and `presets`), and the program flow is organized around p5.js's setup/draw lifecycle rather than through class hierarchies. This reflects JavaScript's more flexible approach to object-oriented concepts, focusing on prototypal inheritance and dynamic object composition rather than rigid class structures.

Rendering Implementation

Both implementations interpret the L-System string similarly, but their rendering approaches differ significantly:

- **Java AWT** uses manual coordinate calculations with explicit trigonometry in the `drawLSystem` method. Position tracking requires maintaining x, y, and angle variables and manually calculating new positions using `Math.cos` and `Math.sin`.
- **p5.js** leverages the library's built-in transformation functions (`translate`, `rotate`, `push`, `pop`), which abstract away the mathematical complexity and more naturally represent the turtle graphics paradigm common in L-Systems.

This difference illustrates how p5.js provides domain-specific abstractions for creative coding that reduce the cognitive load on the programmer, whereas Java requires more explicit mathematical implementation.

State Management and Memory Considerations

Java's approach to state management is more formal and controlled:

- Private class variables with getter/setter methods
- Explicit instantiation of objects (e.g., `new HashMap<>()`)
- Manual memory management for data structures (e.g., `Stack<double[]>`)

The p5.js implementation demonstrates a more relaxed approach to state:

- Direct property access on global objects
- Dynamic property assignment
- Automatic garbage collection without explicit memory management

User Interface Implementation

The UI implementations reflect the fundamental differences between the platforms:

- Java AWT creates UI components programmatically using Swing's component hierarchy, with explicit layout managers (`BorderLayout`, `BoxLayout`) and event listeners.
- p5.js integrates with HTML DOM elements, binding event listeners to existing HTML elements and using browser-native form controls.

This difference highlights how Java maintains a strict separation between UI and logic within a single language, while p5.js embraces the web platform's natural separation of concerns across HTML, CSS, and JavaScript.

Code Verbosity and Development Efficiency

The Java implementation required approximately 258 lines of code, while the p5.js implementation required only about 162 lines—roughly 37% less code. This difference in verbosity impacts development speed and maintenance overhead.

Java's verbosity stems from:

- Explicit type declarations
- Formalized class structures
- Detailed component configuration
- Boilerplate for event handling

p5.js achieves greater conciseness through:

- Dynamic typing
- Simplified function-based structure
- Built-in methods designed specifically for creative applications
- Integration with browser capabilities

Error Handling and Robustness

The Java implementation includes explicit error handling with try/catch blocks and user feedback through dialog boxes, demonstrating a more defensive programming approach. The p5.js version has minimal error handling, relying more on JavaScript's forgiving nature and browser-based debugging tools.

Conclusions

This comparison demonstrates how the choice of platform significantly impacts the expression of object-oriented principles in creative coding:

- Java AWT enforces stricter OOP paradigms with formal class structures, encapsulation, and explicit type checking, leading to more verbose but potentially more maintainable code for larger projects.
- p5.js offers a more flexible, less formal approach to OOP that emphasizes rapid development and concise expression, making it particularly suited for prototyping and smaller creative projects.

The underlying L-System algorithm remains essentially identical across both implementations, but the architectural patterns surrounding it differ dramatically. This illustrates how the choice of programming environment can shape the application of object-oriented principles even when the core computational task remains constant.

Comparative Study of Particle Systems

This analysis compares implementations of particle systems using Java AWT and p5.js, highlighting how object-oriented principles manifest differently in these environments when applying physics-based animation.

Class Structure and Object-Oriented Design

The Java AWT implementation exemplifies a classic object-oriented approach with three distinct classes:

- `Vector2D`: A utility class encapsulating 2D vector operations
- `Particle`: Representing individual particles with physics properties
- `ParticleController`: Managing collections of particles

This structure follows object-oriented design principles of encapsulation and separation of concerns, with each class having clearly defined responsibilities and interfaces.

The p5.js implementation also uses classes, leveraging JavaScript's ES6 class syntax:

- `Particle`: Similar to Java's version but using p5.js's built-in `PVector`
- `System`: Equivalent to Java's ParticleController

However, the p5.js version doesn't require a custom vector class since it utilizes the library's built-in vector functionality (`createVector()`, `p5.Vector`). This demonstrates how framework-provided abstractions can reduce the need for custom implementations while maintaining object-oriented principles.

Vector Mathematics Implementation

The Java implementation requires a custom `Vector2D` class with explicitly defined vector operations:

- Manual implementation of vector addition (`add`)
- Manual implementation of scalar multiplication (`multiply`)
- Explicit vector copying (`copy`)

In contrast, the p5.js implementation leverages the library's comprehensive vector support:

- `createVector` for vector instantiation
- Built-in methods like `add`, `mult`, and `random2D`
- Automatic handling of vector operations

This difference highlights how domain-specific libraries can abstract away common mathematical operations, allowing developers to focus on higher-level logic while still maintaining object-oriented structure.

Animation and Rendering Approach

The Java AWT implementation:

- Uses Swing's `Timer` class for animation control
- Overrides `paintComponent` for custom rendering
- Requires explicit casting to `Graphics2D` and manual setting of rendering properties
- Creates a new `Color` object for each particle with alpha calculation

The p5.js implementation:

- Relies on p5.js's built-in `draw` function for the animation loop
- Uses high-level functions like `ellipse`, `fill`, and `noStroke`
- Takes advantage of p5.js's `colorMode(HSB)` for more expressive color manipulation
- Implements visual effects like scaling and clipping regions with less code

This comparison demonstrates how p5.js abstracts the rendering pipeline, allowing for more concise expressive code while Java requires more explicit handling of graphics contexts.

Event Handling and User Interaction

The Java implementation captures user interaction through explicit event handling:

- Uses an anonymous `MouseAdapter` subclass
- Explicitly calculates forces based on mouse position
- Requires boilerplate for event registration

The p5.js version simplifies interaction:

- Uses p5.js's built-in `mouseIsPressed` state variable
- Integrates interaction directly in the main loop
- Automatically handles mouse position tracking

This contrast highlights how p5.js provides a more streamlined approach to user interaction, reducing the boilerplate code needed while still maintaining the event-driven programming model.

Particle System Management

Both implementations manage collections of particles, but with different approaches:

Java AWT:

- Uses `ArrayList<Particle>` for explicit type safety
- Systematically updates particles, removes dead ones, and renders in separate method calls
- Creates particles at a fixed origin with random velocity

p5.js:

- Uses JavaScript arrays with more flexible typing
- Creates batches of particles in response to user interaction
- Implements a nested system structure allowing for multiple independent particle systems

The Java implementation demonstrates stronger type safety and more explicit memory management, while the p5.js version offers more flexibility in system organization and particle creation patterns.

Visual Effects and Creative Expression

The Java implementation focuses on basic functionality:

- White particles with opacity fading
- Simple circular rendering
- Functional physics without embellishment

The p5.js implementation incorporates more creative elements:

- HSB color mode with evolving hue values for each particle
- Visual enhancements like scaling effects and circular clipping regions
- More expressive visual design focused on aesthetics

This highlights p5.js's orientation toward creative coding and visual expression, which is reflected in both the library's design and how it influences implementation choices.

Comparitive study of Flow Field Simulation

This section presents a detailed comparison of the Flow Field simulation implementations in Java AWT and p5.js. We examine four key aspects: class structure and design, performance of the vector-field computation, smoothness of particle motion and trails, and dynamic visualization and interaction.

Class Structure and Object-Oriented Design

Java AWT implementation:

- `FlowFieldSimulation` (main `JFrame`)
- Inner class `FlowFieldPanel` handles rendering and the animation loop.
- `FlowField` encapsulates the grid of `Vector2D` vectors, with methods for noise-based update and local disturbances.
- `Particle` represents individual particles, each holding its own position, velocity, acceleration, and previous position for trail effects.
- `Vector2D` provides vector arithmetic (`add()`, `mult()`, `normalize()`, `limit()`, etc.).
- Follows strict encapsulation: each class has a single responsibility and communicates via well-defined methods.
- Uses Swing's layout managers and listeners to separate UI setup from simulation logic.

p5.js implementation:

- Typically consists of:
- A `FlowField` class that builds an array of `p5.Vector` entries via `noise()`.
- A `Particle` class using `createVector()`, with methods `applyForce()`, `update()`, and `display()`.
- Global `setup()` and `draw()` functions orchestrating initialization and per-frame updates.
- Leverages p5.js's built-in vector support, removing the need for a custom `Vector2D` class.
- UI controls (sliders, buttons) are standard HTML elements bound via DOM or p5.js helper functions, decoupling markup from logic.

Performance in Field Computation

- **Java AWT:**
- Precomputes the entire flow field in `FlowField.update()`, iterating over a grid of size `cols` × `rows`.
- Uses a simple hash-based noise function to generate each vector angle once per 60 frames, then reuses the same field for all particles.
- Executes the particle updates and field recomputation on the Event Dispatch Thread; performance remains high due to optimized Java math and the relatively coarse cell resolution (e.g., 20px).

p5.js:

- Can either:
- Recompute the field in each `draw()` call by sampling `noise()` on a grid, or
- Precompute once into a JavaScript array and reuse it, similar to Java.
- Single-threaded environment means large grids may cause frame drops unless WebGL shaders or Web Workers are employed.
- p5.js's built–in `noise()` is optimized in C++ under the hood, but JS looping overhead can still introduce jank.

Smoothness of Particle Motion & Trails

Java AWT:

- Each `Particle` stores a `prevPosition` and draws a semi-transparent rectangle (`new Color(0,0,0,20)`) on each frame to produce smooth fading trails.
- The `Timer` at 16ms per tick maintains a steady 60fps, ensuring consistent visual flow.

p5.js:

- Typical pattern: call `background(0, 20)` at the start of `draw()` to fade out previous frames.
- Use of `drawingContext.globalAlpha` or `fade` in `background()` achieves trail effects with concise code.
- Frame rate controlled via `frameRate(60)`, but dependent on browser rendering performance.

Dynamic Visualization & Interaction

Java AWT:

- `FlowFieldPanel` registers a `MouseAdapter`; `mousePressed()` calls `flowField.addDisturbance(x, y)`, creating a localized radial disturbance in the vector grid.
- Additional Swing UI components (buttons, sliders) allow on-the-fly adjustments to cell size, noise parameters, or particle count.

p5.js:

- Harnesses `mousePressed()` callback directly; modifying the `field` array or global parameters in response to clicks or drags.
- HTML sliders and color pickers bound via `select()` or `createSlider()`, with listeners in JS to update simulation variables dynamically.

Comparitive study of Boids Simulation

Overview

The Boids simulation models flocking behavior via three simple rules—alignment, cohesion, and separation—applied to large numbers of agents ("boids"). Both Java AWT and p5.js implementations realize the same algorithmic core but differ significantly in design, verbosity, and interactivity.

Class Structure and Object-Oriented Design

Java AWT The Java version defines a `FlockingSimulation JFrame` containing a `FlockPanel` inner class for rendering and UI, plus separate `Flock`, `Boid`, and `Vector2D` classes. This strict layering enforces single–responsibility: the panel handles Swing events and repainting, `Flock` manages the boid collection, `Boid` encapsulates behavior rules, and `Vector2D` provides basic vector arithmetic (**?**; **?**).

p5.js In p5.js, a single ES6 `Boid` class uses the library's built-in `p5.Vector` for all math. The simulation loop lives in `setup()` and `draw()`, and UI sliders are standard HTML inputs bound to global variables. This flatter structure reduces boilerplate but retains conceptual OO boundaries (**?**).

Vector Mathematics and Behavior Rules

Java's `Vector2D` implements methods `add()`, `sub()`, `mult()`, `normalize()`, and `limit()` manually, with each boid computing neighbors within a perception radius via explicit loops (**?**). In p5.js, `createVector()` plus `p5.Vector.add()`, `mult()`, `normalize()`, and `limit()` serve the same roles, reducing custom code and potential errors (**?**).

Rendering and Visualization

The Java implementation uses `Graphics2D` to draw and rotate a `Path2D` triangle per boid, optionally visualizing force vectors in different colors. Swing's `Timer` at 16ms intervals ensures smooth animation (**?**). In p5.js, `push()`/`pop()`, `translate()`, `rotate()`, and `triangle()` draw each boid, with hardware-accelerated canvas (or WebGL) handling performance (**?**).

Interactivity and Controls

Java wires `JSliders` and a `JCheckBox` to adjust alignment, cohesion, and separation weights in real time, and adds boids via a `MouseListener` (**?**). p5.js binds HTML `<input type="range">` elements (or uses `createSlider()`) to global parameters and leverages the `mousePressed()` callback to spawn new boids, simplifying event handling (**?**).

Performance Considerations

In Java, the typed, class-based structure and explicit loops perform reliably at hundreds of agents thanks to JVM optimizations and the Swing event thread. p5.js achieves comparable frame rates for moderate flock sizes but can exhibit frame drops in large populations without WebGL or worker offloading, owing to JS's single-threaded nature (**?**).

Comparitive study of Reaction–Diffusion System

Overview

The Reaction–Diffusion simulation models chemical pattern formation using the Gray–Scott equations on a 2D grid.

Both Java AWT and p5.js implementations recreate the same emergent patterns but differ in architecture, performance, and interactivity.

Class Structure & Concurrency

Java AWT The Java version splits responsibilities across three classes:

- `ReactionDiffusionSimulation` (`JFrame` with sliders and buttons),
- Inner `ReactionDiffusionPanel` (animation loop, rendering, FPS tracking),
- `ReactionDiffusionSystem` (grid state, parallel update logic using an `ExecutorService`).

This separation allows the core PDE updates to run on multiple threads, maximizing CPU utilization on multi-core machines (**?**).

p5.js A typical p5.js sketch embeds the entire logic within `setup()` and `draw()`, with grid arrays and update code executed sequentially in JavaScript. Concurrency can only be achieved via Web Workers or WebGL shaders, which are more complex to integrate.

Computational Efficiency & Grid Update

Java AWT Utilizes a fixed thread pool equal to the number of processor cores. Each thread processes a horizontal slice of the grid, applying a 3×3 Laplacian kernel and Gray–Scott formulas, then swaps buffers—all within a 33ms frame budget for 30fps (**?**).

p5.js Performs the same PDE loops in the single-threaded `draw()` call. Large grids or high resolutions can cause noticeable frame drops unless offloaded to GPU via ‘createShader()‘ or WebGL-assisted routines (**?**).

Rendering & Visualization

Java AWT Transfers the computed chemical B concentrations into a `BufferedImage` per frame, mapping concentrations to grayscale pixels, then draws it via `Graphics2D.drawImage()`. Feed/kill parameters and FPS are overlaid using `drawString()` (**?**).

p5.js Maps grid values to the `pixels[]` array with `loadPixels()`/`updatePixels()`, and uses `rect()` or `point()` to render patterns. Parameter display is handled with `text()` directly in the canvas.

UI Controls & Interactivity

Java AWT Implements `JSlider` for feed and kill rates, plus “Reset” and “Pause/Resume” buttons wired via `ChangeListener` and `ActionListener`. Panel mouse events inject chemical B in circular regions (**?**).

p5.js Uses HTML ‘¡input type=”range”¿‘ elements (or `createSlider()`) bound to global variables. ‘mousePressed()‘ and ‘mouseDragged()‘ callbacks add chemicals directly to the grid, with no additional boilerplate.

Performance Considerations

Java’s multi-threaded model consistently sustains real-time updates on large grids, while p5.js excels at simplicity and rapid prototyping but may require GPU-based shaders or worker threads for comparable performance in high-resolution simulations (**?**).

Conclusions

This comparison reveals significant differences in how object-oriented principles are applied in Java AWT versus p5.js:

- Java AWT enforces a more traditional, rigid object-oriented structure with explicit class hierarchies, type safety, and encapsulation.
- p5.js enables a more flexible object-oriented approach that leverages library abstractions while maintaining class-based organization.
- The underlying particle physics remains conceptually identical across both implementations, but the framework significantly influences code organization, abstraction levels, and expressive capabilities.

The p5.js implementation requires less boilerplate and provides higher-level abstractions, but potentially sacrifices some explicit control and type safety. The Java implementation offers more explicit structure and control but demands more verbose code and manual implementation of functionality that comes built into p5.js.

These differences demonstrate how programming environment choices substantially impact the expression of object-oriented principles in creative coding, even when implementing the same fundamental algorithms and behaviors.

References

Oracle(2023) *Java Documentation.* Reference from https://docs.oracle.com/en/java/

p5.js, Reference from https://p5js.org/reference/

Shiffman, D. *The Nature of Code: Simulating Natural Systems with Processing*. The Nature of Code, LLC, 2012.

McCormack, J., & d’Inverno, M. Computational creativity: A multidisciplinary approach. *Journal of Artificial Intelligence and Society*, 27(4), 397-400, 2012.

Eck, D. J. *Introduction to Computer Graphics*. Hobart and William Smith Colleges, 2019.