# Object Oriented Programming Based Game Design: Java vs Python

**Grayson Hubbell**[1], **Yuri Fung**[1]

[1]University of Colorado Boulder
grayson.hubbell@colorado.edu, yuri.fung@colorado.edu

### Abstract

This research explores the pros and cons of using Java and Python for Object-oriented (OO) game design. The study analyzes how each language handles fundamental OO principles such as encapsulation, inheritance, and polymorphism. The focus is on how these features affect the structure and performance of game applications and the implications for game designers. The research covers implementing game design elements such as graphics, sounds, and user input and story-driven elements such as characters, dialogue, and maps. This paper compares the differences in runtime environment, game performance, and ease of development with OOP tools between Java and Python. Accompanying code repository can be found here: https://github.com/HubbelGrayso24/java-python-snake

## Introduction

Object-oriented programming (OOP) has become integral to modern game development because it manages complex systems by organizing code into reusable, modular components. OOP enables game developers to represent in-game objects such as players, enemies, and environmental elements with classes. Those classes can then be instantiated as objects or inherited by subclasses and utilized in flexible, understandable, yet highly complex programming flows. As the gaming industry grows, the choice of which programming language to use plays a vital role in defining the efficiency and scalability of game development projects. This research paper will compare and analyze two of the most popular programming languages used in OOP game development: Java and Python.

This paper evaluates these languages based on their adherence to OOP principles, the tools and libraries they provide for game development, runtime performance, and ease of development. The evaluations and comparisons target game developers trying to select a language for their next game development project.

## OOP Concepts in Java & Python

### Encapsulation

Encapsulation in OOP refers to bundling and protecting data with methods that operate on that data, keeping both safe from outside interference and misusage. In game design, encapsulation is critical for managing the state of objects such as NPCs (non-player characters), items, game states, and other complex game-specific data structures. Java and Python allow developers to implement encapsulation but with critical differences.

Java enforces encapsulation strictly with access modifiers like private, public, and protected (Stemkoski 2018). For example, classes in Java model game elements like characters or maps, with sensitive game data like player health or inventory stored using private variables within the class. This private enforcement ensures that data is not easily modified outside of class, helps maintain stability, and prevents bugs. Game development teams are often huge, and for those teams, encapsulation ensures that any variables are correctly modified and consistently across a large and complex codebase.

In Python, encapsulation is more "relaxed." Although it "supports" encapsulation through private members using naming conventions (prefix variables with "_" and "__"), these private variables can still be accessed externally, which makes it more flexible but more prone to misuse and bugs (Kalb 2022). Python relies on a trust-the-developer solution compared to Java's strict compile failure implementation. In Python-based game development, this flexibility can help develop prototypes faster, but skirting OOP principles can become a significant problem as bugs compound in development. Despite these problems, Python's simple approach allows developers to be swift during development, which may be attractive for small-scale game projects where a small team is highly familiar with the codebase and thus can comprehend encapsulation violations.

### Inheritance

Inheritance allows the creation of subclasses from the base class, which helps with code reuse and adaptivity. In game development, inheritance is beneficial for defining hierarchical relationships between game objects. For instance, if a game has multiple character types, like player, enemy, or NPC, each character can inherit common properties and behaviors from a Character class while defining their unique attributes. This DRYs up the code and reduces development complexity, making the broader architecture easier for individual developers to digest. Inheritance is a core of OOP, and for game development, it is instrumental, but both Java and

Listing 1: Java Encapsulation `Snake.java`

```java
public class Snake implements Movable,
    Renderable {
    private final List<SnakeSegment>
        segments;
    private Direction direction =
        Direction.RIGHT;
    private final MovementStrategy
        movementStrategy;

    ...

    public SnakeSegment getHead() {
        return segments.getFirst();
    }
}
```

Listing 2: Python Encapsulation `encapsulation.py`

```python
class Character:
    def __init__(self):
        self._health = 100

    @property
    def health(self):
        return self._health

    @health.setter
    def health(self, value):
        if value < 0:
            self._health = 0
        else:
            self._health = value
```

Python have intricacies in their specific implementations.

As with encapsulation, Java has a strict approach to ensure OOP best practices. Java uses type checking to ensure the correctness of inheritance, which prevents errors at compile time (Sarcar and Mullick 2020). Java classes can extend other classes, and with abstract classes and interfaces, developers can define flexible hierarchies. For example, an abstract Enemy class might define an attack method, while specific enemies like Vampire or Dragon override and implement unique attack methods. Inheritance allows for consistent behavior across different enemies while integrating specialized actions.

Python supports single and multiple inheritance, allowing developers to combine behaviors from multiple base classes. While this flexibility is a powerful tool, it can make the class hierarchy more challenging to manage with larger games (Kelly 2019). With Python's implementation of encapsulation, support for unrestricted multiple inheritance can lead to poor usage of OOP best practices. Multiple inheritance, however powerful, can dramatically increase the complexity of the codebase and introduce bugs that Python will not catch nor mark as errors because it makes an arbitrary decision at runtime.

Listing 3: Java Inheritance `Food.java`

```java
public abstract class GameEntity {
    private float x;
    private float y;
    ...
    public void setPosition(float x,
        float y) {
        setX(x);
        setY(y);
    }
}
public class Food extends GameEntity
    implements Renderable, Collidable {
    ...
    public Food(float x, float y) {
        super(x, y);
    }
    ...
}
```

Listing 4: Python Inheritance `inheritance.py`

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a
            noise."

class Dog(Animal):
    def speak(self):
        return f"{self.name} barks."
```

## Polymorphism

Polymorphism enables a single function to handle objects of different types, which is helpful in games for interacting with a diverse variety of similar yet distinct objects. Polymorphism allows runtime flexibility in instantiated objects actions based on the specific subclass chosen. An example could be how a game handles armor and reduces damage for the player. If there are different types of armor, the specific subclass is passed through. A standard function of the superclass, reduceDamage(int damage), is invoked. This function returns a reduced damage value determined by the properties and mechanisms of each subclass. Polymorphism allows game developers to create various complex objects with unique properties that all interact as the same superclass, decreasing coupling and increasing the legibility of the codebase.

Java attains polymorphism through method overriding and interfaces (Stemkoski 2018). Developers define standard interfaces for unique game elements. These interfaces allow game systems like fighting or interaction to operate polymorphically. For example, a parent class Weapon could have subclasses Sword and Wand with a standard method attack() that can be called without knowing whether the player is wielding a sword or casting a fireball. In either case, there is a consistent and roughly similar outcome independent of the function that calls the method attack.

Listing 5: Java Polymorphism `Renderable.java`

```
public interface Renderable {
    void render(ShapeRenderer renderer);
}

public class SnakeSegment implements
    Renderable {
    @Override
    public void render(ShapeRenderer
        renderer) {
        renderer.setColor(Color.GREEN);
        renderer.rect(getX(), getY(),
            WIDTH, HEIGHT);
    }
}
```

Listing 6: Python Polymorphism `polymorphism.py`

```
class Shape:
    def area(self):
        pass

    def perimeter(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2
```

Python, again, allows users to break some principles of OOP because of its polymorphism implementation. Python achieves polymorphism with dynamic typing. This dynamic typing enables more flexibility with polymorphism but does not ensure correctness as with Java's implementation. Dynamic typing achieves polymorphism by running a method so long as the object supports the method's operation. For example, a fireball and lion could have an attack method, but they are not required to inherit from the same class for Python to execute the method. Python's approach using duck-typing increases flexibility in game design. As with the fireball and lion example, the attack method may have the same outcome; thus, both classes could be relevant. That said, duck-typing reduces type safety compared to Java, and runtime errors can occur if the called object's method is not correctly defined or the return from the method call is not as expected (Miller, Settle, and Lalor 2015). In the fireball and lion example, the correct OOP practice would be inheriting both classes from a Weapon class or interface.

## Implementation of Game Design Elements

### Graphics & Sound

Java and Python offer libraries for implementing graphics and sound, which are essential game development components. There are many examples of such libraries, but this analysis focuses on Java's LibGDX and Python's Pygame. These libraries are standard and well-documented tools that provide developers with both sample implementations, as analyzed in this paper, and tools for complex graphics and sound implementation.

Listing 7: LibGDX `EatFoodSound.java`

```
public class EatFoodSound implements
    AudioPlayable {
    Sound sound = Gdx.audio.newSound(Gdx
        .files.internal("eat.mp3"));

    @Override
    public void play() {
        sound.play();
    }
}
```

Java's LibGDX is a robust game development tool for implementing graphics, handling user input, and managing game states. LibGDX is notable for its performance, cross-platform capabilities, and comprehensive toolkit, which makes it a popular choice for professional game developers (Stemkoski 2018). LibGDX builds comprehensive tools for game development on top of the features that Java already provides. LibGDX provides a straightforward yet powerful class architecture for audio processing that abstracts and encapsulates all system and JVM audio complexities. It allows developers to use any sound file they like and modify its playback by changing properties like speed, pitch, looping, and volume. The sound and music objects represent files, sound designed for shorter segments, and music for longer pieces, and they both follow all OOP best practices. Both interfaces have identical method coverage, and the instantiated objects are entirely encapsulated. While the audio interface is simple and understandable, due to the inherent complexity, the graphics classes built into LibGDX are numerous and highly intricate. Despite the scope of the 2D and 3D graphics engines, LibGDX uses the same conventions found in the rest of the package and provides a comprehensive set of tools for graphics rendering. The graphics tools also follow the conventions of OOP and create a well-defined hierarchy for sprites and other rendered elements. Between the audio and graphics library, LibGDX provides a complete toolset that aligns closely with OO principles for game developers who want to use Java as their platform for development (Bad Logic Games 2024).

Python uses Pygame, which is less powerful but also more approachable. Pygame is more suitable for beginners, smaller game projects, and quick prototyping for larger projects (Kelly 2019). Despite its narrower scope, Pygame still incorporates audio and graphics similar to LibGDX; in fact, Pygame's audio support is even more robust than that of LibGDX. On top of the same tools available in LibGDX, Pygame allows the developer to control the channel outputs, create spatial audio experiences, and control the package's audio processing variables such as frequency, buffer size, and output device. The audio implementation has many of the same issues in both Pygame and LibGDX, namely the unrelated classes that provide audio for short clips versus music. This separation is a minor oversight in both cases, as

Listing 8: Pygame `game.py`

```
class Game:
    def __init__(self, gameState):
        ...
        self.pickup_sound = pygame.mixer
            .Sound("Sound/ItemPickup.mp3"
            )

    def update(self):
        ...
        if self.snake.body[0] == self.
            food.position:
            ...
            self.pickup_sound.play()
```

there is no general audio type. In both libraries, the mixer and music implementations have all the same methods, but they inherit from a general superclass of game objects rather than an audio-specific class (Pygame 2024).

While the audio libraries in Pygame are more robust than LibGDX, the graphics libraries are where Pygame's scope shrinks dramatically. Pygame is a 2D game engine; it does not support 3D rendering or scenes like LibGDX. Moreover, the options are limited and focused on sprite and shape-driven game development. This simplicity is both a strength and a weakness. 2D games are much easier to develop and offer an entry point for one-person teams or small indie groups interested in prototyping 2D games (Pygame 2024).

Both LibGDX and Pygame are game development libraries with object-oriented solid principles. They provide game developers with launchpads to build their imaginations. For both packages, the audio and graphics tools would allow near-infinite flexibility and provide an abstraction for some of the most complex elements of game design. Pygame is generally a well-built and detailed library for developing simple to complex 2D games with an elegant toolset, while LibGDX is a fully-fledged game development tool with robust support for audio, 2D, and 3D graphics.

## User Input & Interaction

While handling user input is crucial, Java and Python offer different approaches to it. Again, this paper will discuss the LibGDX and Pygame libraries and the corresponding support for user input. It will focus on two key differences: events vs. input processing and multiplatform support.

The Pygame implementation of user input relies on the event queue, a list of events the system compiles as the game progresses. This event-driven style has some benefits. It is a straightforward system to interact with, and it is reliable as each event the system could record will be there; the developer can then choose to execute based on that list. It also follows solid OO principles, namely polymorphism, as the events can be any user input, although Pygame only supports a mouse, keyboard, and controller (Kelly 2019). The downsides of this design are speed and simultaneous inputs. The event queue must be continuously monitored for events, which then must be processed. Waiting for the event queue to update and then doing something is inefficient and a significant slowdown for the Pygame user input. Beyond speed, the nature of event queues with discrete actions means there is no support for pressing input combinations, as they will register as single inputs in the event queue (Pygame 2024).

Listing 9: LibGDX `MovementStrategy.java`

```
public class DefaultMovementStrategy
    implements MovementStrategy {
    @Override
    public void updateDirection(Snake
        snake) {
        if (Gdx.input.isKeyPressed(UP))
            { snake.setDirection(UP); }
        else if ...
    }
}
```

Listing 10: Pygame `Game.py`

```
for event in pygame.event.get():
    if event.type == pygame.KEYDOWN
        if event.key == pygame.K_UP:
            self.snake.change_direction
                ((0, -1))
        elif event.key == pygame.K_DOWN:
            self.snake.change_direction
                ((0, 1))
```

LibGDX also has an event queue-like system that could be used similarly; however, there is a more robust implementation that LibGDX enables. Using a subclass of InputProcessor, LibGDX can proactively execute methods when the system receives inputs rather than placing them in an event queue. Developers can then chain together InputProcessors using the InputMultiplexer class, which enables simultaneous input to run complex methods (Stemkoski 2018). Furthermore, InputProcessor is faster than the event queue as it replaces the placeInQueue method with the actual resultant method. The implementation of InputProcessor and InputMultiplexer are both solid. While there is only native support for keyboard, mouse, and touch input, they define a framework for an adaptor pattern that allows a developer to integrate the InputProcessor with any input device (Bad Logic Games 2024).

In addition to the benefits of InputProcessor over an event queue, LibGDX has an additional advantage over Pygame for user input. LibGDX supports multiplatform game development, and as part of that goal, the InputProcessor and general Input classes have robust support for unique input devices. Critical is LibGDX's strong support for touchscreen devices, which broadens the audience a game can reach. The classes also allow developers to create uncommon interfaces, such as joysticks or accessible controllers. The support for multiplatform development may seem like an insignificant goal, but it further highlights the robustness and general completeness of LibGDX (Bad Logic Games 2024).

## Story Elements

Developing a story has become an integral component of game design. Stories drive games, but they also introduce

significant program complexity as there can be an enormous quantity of paths an end user can take. Object-oriented programming is a valuable tool to integrate story components into a game. Polymorphism allows runtime changes in a player's actions, inheritance will enable developers to create a unified story element structure, and encapsulation ensures compliance with requirements when developing the story. As discussed earlier, both Java and Python strongly support SOLID OOP principles and thus enable developers to create story structures using OOP. Despite both languages supporting object-oriented programming, some key differences separate Python and Java when creating story elements.

While LibGDX is a powerful tool for deploying rendering, audio, and input tools, it does not have internal classes designed as story elements. Instead, standalone Java is used to develop the story components, which, using patterns, can interact with the LibGDX pipeline. Command pattern Java classes can include information about a character or action item to pass into the LibGDX pipeline as a universal executable that can render new elements or change the story arc. Implementing story elements with the command pattern is also possible and encouraged in Pygame, as its event queue runs commands similarly to the pattern. In both cases, the strong support and encouragement for the command pattern for story elements indicates the commitment to SOLID principles and support for common object-oriented patterns.

Beyond the command pattern, inheritance is a powerful tool for story development that both Java and Python take advantage of similarly. Inheritance allows developers to create story elements like non-playable characters (NPCs) and quests that inherit their standard features from a superclass. The inheritance simplifies the development of these standard features and enables polymorphism, as discussed earlier. That polymorphism allows developers to create simple executable code that abstracts the potential implementation complexity to a class, making the overall codebase and, thus, the story structure simpler to understand when developing. Java and Python support inheritance similarly, but for story elements, Python's implementation is more straightforward for developers to adapt for story design. Multiple inheritance, while potentially problematic, is a powerful tool to enable story design. NPCs, for example, could inherit from several parents, including Invulnerable, Interactable, Moveable, and Human. In Java, there are severe restrictions on multiple inheritance, and while it is possible with interfaces, it is not as developer-friendly as Python's implementation. A critical issue when developing with multiple inheritance is overriding methods, where it is possible to create the diamond problem. Java solves this but fails to compile, something that can require significant refactoring; Python instead uses a priority queue to resolve the diamond problem. While this implementation is imperfect, it enables developers to use multiple inheritance and prioritize where subclasses will pull methods from. Python defines a list to reference, with the first parent selected first, then the remaining parents in order, then their parents, and so on. This implementation is tractable and predictable, efficiently enabling multiple inheritance for story development.

An additional key element of game design is a global state space that includes variables and information that determines both the current state of the game and how the game will play out in the future. As both LibGDX and Pygame pass the development of the story structure to the developer, the native language support for global variables, pointers, and the singleton pattern is essential to allow complex game structures to access game states without passing through game states at every method call. As game states are inherently global, access can be problematic, and overlapping calls can muddle data. The singleton pattern is a great way to alleviate this problem; critically, Python does not support the singleton pattern implementation or global variables. Without simple, controlled access to global variables, either via the global namespace for smaller teams or with the singleton pattern for larger teams, Python's story implementation is more complex. It requires numerous and confusing variables passing between methods. As Java does have a global namespace and robust support for singleton patterns, Java is the superior language for developing most story-structured games.

Python and Java have pros and cons for developing story elements, and neither package this paper discusses natively supports story structure. Python is generally simpler, as continuously discussed, and while it supports inheritance and polymorphism as Java does and multiple inheritance better than Java, the lack of the singleton pattern and truly global variables hurts story state space significantly. Due to the lack of support for pointer passing required for the singleton pattern and global variable, Python is inferior to Java for developing story structure in games because global state spaces are integral to story design.

## Performance Comparison

### Runtime & Execution Speed

Games are often large and complex runtime environments. Between graphics rendering and managing all the moving parts that could be in a game, such as NPCs or world simulations, games are some of the most computationally intensive programs. Programming languages can play an essential role in the execution speed of programs and thus significantly impact game performance. In both Java and Python, this paper heavily relies on OOP principles to design SOLID games, which may add additional processing time and memory requirements, a potential downside to this approach. To measure the performance difference between the languages, this paper relies on the findings of Wingqvist, Wickström, and Memeti and Filho et al.. These papers measure the computing performance of the two languages. They do not measure the most compute-intensive aspect of a game: graphics. Not including graphics in this comparison may appear incomplete, but LibGDX and Pygame rely on OpenGL for their graphics pipeline. OpenGL interacts with the graphics processor independent of the language type and relies on its bindings independent of language (Overvoorde 2019). This built-in offloading is a critical feature of both packages and without it, the bar to develop games would be significantly higher.

Compared to Java, Python is, in almost all cases, orders of

magnitude slower. When running a series of algorithms with large integer, float, and pointer computing, Python was between 72.83 and 249.87 times slower than Java, a vast gulf (Filho et al. 2024). The algorithms explored in this paper do not perfectly represent a game, but they indicate just how stark the difference can be. Java is the more performative language on every platform, which could be essential when deciding which language to use. The additional speed Java provides is the difference between an accessibly playable game and a game that cannot maintain significant framerates.

While this paper focuses on OOP and the SOLID principles for developing flexible, scalable games, there is a performance cost to coding with this approach. Redirecting the development focus from OOP to data-oriented design can significantly uplift performance. Wingqvist, Wickström, and Memeti finds that using a data-oriented design over OOP can improve performance 13.25-fold on identical hardware. That additional speed is another vast improvement demonstrating the efficacy of data-oriented programming. While this approach benefits performance, it deviates from the OOP standard design, which can create development complexity and significant headaches for large development teams working on one game together. While data-oriented design is valuable, object-oriented programming is the most helpful tool for most game developers.

## Concurrency & Multithreading

As games become more complex, integrating multithreading or multiprocessing becomes increasingly essential to maximize game performance. Parallelization has a dramatic potential performance advantage on modern hardware with many CPU cores. That said, writing multithread-compatible programs is significantly more complex and expensive. Native language support and ease of implementation are two key requirements for developers to consider the additional effort of developing a multithreaded game. Java and Python support multithreading natively and use universal nomenclature, so both are appealing options for multithreading programming. Despite the standard implementations, there are significant downsides to multithreading as neither language is natively thread-safe, and neither are the packages that we discuss, LibGDX and Pygame.

Java's multithreading implementation relies on OOP. It contains an abstract class, Thread, and an interface, Runnable, which allows SOLID-designed classes to run on user-definable threads. Implementations of the Runnable interface operate similarly to a command pattern, where a run() method is called when the thread containing the Runnable class is started. To ensure that access to variables is appropriately synchronized, Java utilizes a keyword, synchronized, to indicate that multiple threads should access that method one at a time. This simple implementation prevents race conditions but could cause deadlock for complex functions that may define game elements. More complex Java utilities also allow developers to implement locks and semaphores, which can prevent race conditions and deadlock but require significantly more commitment to developing for a potentially slight performance improvement (Goetz 2006).

Listing 11: Java `MultiThread.java`

```
1  public synchronized void increment(
       String threadName) {
2      counter++;
3      System.out.println(counter);
4  }
```

Listing 12: Pygame `Game.py`

```
1  def multiprocessing_worker(counter, lock
       , process_name):
2      for _ in range(5):
3          with lock:
4              counter.value += 1
5              print(f"{counter}")
```

LibGDX explicitly warns against using multithreading with the package but points out that the tool uses some inherent multithreading (LibGDX Threading). Graphics are rendered via OpenGL, which runs as a separate thread. User input and audio processing both run on separate threads, allowing the game to run, play audio, and process user input simultaneously. LibGDX's multithreading implementation focuses on concurrent execution instead of performance uplift. Still, it highlights the potential to use the Running interface and synchronized keyword to run multiple game elements simultaneously, such as two characters fighting in real time.

Compared to Java's native threading implementations, Python's is more confusing, more difficult to implement, and easier to create race conditions and deadlock. Instead of creating OOP SOLID classes like Java, Python relies on a replica of the C and C++ threading implementations. These implementations are robust but inaccessible to newer developers and smaller teams that cannot rigorously develop multithreading. This contrasts with Python's general ease of development and discourages developers from using multithreading with Python for game development. Moreover, Python includes a GIL or global interpreter lock, which limits what threads can access and, more importantly, the number of cores the program can run on. With the GIL and the multithreading package, Python remains a single-core program that enables concurrency but is unlikely to accelerate game performance. To run a Python program on multiple threads, multiprocessing is required, which has a similar implementation but bypasses the GIL. In addition to Python's inaccessible multithreading support, Pygame is designed to be purely single-threaded. The event processing pipeline that Pygame relies on executes all functions in the order they are passed through, namely user input. Audio and graphics are still technically threaded, but they are because of the native implementation of audio and graphics at a system level.

This paper's snake game is very limited in scale and thus does not include a multithreaded implementation. Code snippets are included to highlight how Java and Python threaded implementations work. Due to the snake game's limited scale, multithreading would likely harm perfor-

mance, as the system would have additional overhead to create and manage threads compared to the single-threaded implementation.

## Developer Experience & Ease of Use

### Development Tools & Libraries

Python and Java are popular languages with long-standing support from the development community. As such, they have created many development tools for various projects, including game development. This paper discusses LibGDX for Java and Pygame for Python, the most popular game development tools on their respective platform. Still, many other tools, like PyOpenGL and LWJGL, provide lower-level access to the hardware functions, and higher-level tools, like Slick2D and Pyglet, abstract the complexities of game development for beginners. Both LibGDX and Pygame are complete, OO-supported game development tools that demonstrate the completeness of the game development experience for their respective languages and highlight the deep community support for tools enabling game development on their respective platforms (Bad Logic Games 2024) (Pygame 2024).

Beyond the existence of libraries like LibGDX and Pygame, the syntax and structure of the packages are important as they define the ease of use for the tools. Developers looking to develop in a specific language, Python or Java, intend to build with the language's syntax. Thus, packages must align with the syntax and programming structure of the language. Pygame's package closely aligns with Python's general syntax (Kelly). Still, for a developer focusing on object-oriented design, Pygame is designed primarily with a linear implementation and does not expose many objects but package methods. To develop with Pygame using object-oriented principles, developers must define their class hierarchy and object-oriented structure. This flexibility can be appealing as no existing constraints limit a developer's ability, but it is also inherently easier to deviate from the SOLID principles. Java and LibGDX have a tighter implementation of the object-oriented tenets and expect developers to expensively use SOILD principles in their game design, following the steps that LibGDX initially defines. LibGDX's implementation of sprites, sounds, maps, inputs, and almost every other package element is defined as abstract classes or interfaces for developers to implement (Bad Logic Games 2024). This design enforces object-oriented principles and strongly encourages developers to create SOLID game code. This enforcement is key to ensuring that development teams effectively use object-oriented principles. Pygame is the more straightforward language to read and write. Still, LibGDX's implementation of Java uses strong OOP principles and benefits from the substantial code reuse and the easy development with large teams that it provides.

Beyond the game community, Python is the most popular language on GitHub (Zapponi 2024). That popularity is a significant advantage as the community is developing additional packages and resources for Python game development to cater to a large audience. On the other hand, Java is still very popular but has been around for considerably longer and has support for legacy systems and codebases that could add significant features to a game's design. Packages like LibGDX have also been in development markedly longer than their Python equivalents and, thus, are more robust. Between Pygame and LibGDX, LibGDX is the clear winner, but there are pros and cons to the age and popularity of both languages. Either language has robust development support for developing games with excellent tools.

### Community & Ecosystem Support

The communities for both Java and Python support game development, but their focus and depth differ. Java has a professional and enterprise-level ecosystem with long-standing forums, such as Stack Overflow, Github repository, and documentation of libraries like LibGDX. These libraries are well-documented and include Javadoc references, and make sure that developers can access detailed information about methods, classes, and implementation strategies. Furthermore, developers can benefit from tutorials and guides tailored for scalable, performance-intensive game development projects. Corporate use of Java for software development reinforces the stability and reliability of community support. This ensures consistent updates, bug fixes, and active forums.

Python's community is much broader, from beginners to educators. Platforms such as Github, Reddit, and Python forums supply resources for Pygame and similar libraries. The simplicity of Python syntax and a wide variety of open-source projects make it easy for developers to contribute and learn more effectively. Python's community offers resources for quick prototyping and smaller game development projects in which simplicity and creativity are prioritized over performance. Furthermore, YouTube tutorials are valuable resources for Python game developers, which offer more visual walkthroughs for building games using Pygame and related libraries. Python is also a more popular language than Java. While both are widely adopted, Python does have considerably more community adoption, particularly at an educational level (Zapponi 2024).

A notable advantage of Python's community is its openness to experimentation and accessibility for new developers. This inclusiveness encourages a broader audience to engage in game development, encouraging innovation in smaller-scale projects. However, lacking professional-level tools compared to Java limits Python's suitability for large-scale and complex projects. Despite this, Python's community is consistent about its support, collaboration, and learning-oriented environment, which is ideal for developers or those creating smaller-scaled games.

Both Java and Python communities are rich in resources, catering to different needs. Java offers a professional and structured ecosystem, perfect for developers seeking scalability and robustness, while Python thrives as a beginner-friendly platform motivated by simplicity and creativity. Platforms such as YouTube, GeeksForGeeks, and w3schools provide foundational programming insights for both languages, which helps developers of all skill levels for support and inspiration for their projects.

## Conclusion

The comparison of Java and Python for OOP-based game development emphasizes their distinct advantages and limitations. Java's strength lies in its performance, scalability, and strict adherence to OOP principles, which make it optimal for large, complex games. Conversely, Python is well-suited for smaller projects, rapid prototyping, and educational purposes, offering simplicity and flexibility. Both languages are complemented by tools like LibGDX and Pygame, providing viable options for game development depending on the scope and team expertise.

Python's core strength for game development and, in general, is its simple syntax and easy-to-run code. We have discussed the clear benefits of this approach. Python is more flexible. Its poor implementation of SOLID OOP principles allows you to break the rules and take shortcuts. Pygame is also a fantastic tool that has a purpose similar to Python. It is simple to set up and easy to program, implementing only methods and structures that are universally understandable. Rendering elements and text is straightforward; sound bites can be called at any point, and developers can access the event queue anywhere. In creating the snake implementation attached, we observed just how simple Pygame is. There are no nonstandard data structures or unique implementations of classes for display, sound, or audio; everything is built into linear calls to Pygame. From an OOD perspective, particularly when developing anything more complex than snake, these are considerable downsides to Python and Pygame.

On the other hand, Java strongly applies the principles of OOD, and LibGDX's implementation is SOLID, which encourages developers to create SOLID programs. However, the syntax is more confusing, and running the game relies on complex functions defining how it should perform in detail. Beyond the code, the game's structure is daunting and off-putting for novice developers or those unfamiliar with Java. Beyond the structure of LibGDX, the implementation itself is more complex. For a simple game, the hierarchies of inheritance and the use of polymorphism can initially be discouraging, and it costs additional development time to create with OOD. This paper's snake game implementation in Java requires three times the number of files, but the implementation is more straightforward than in Python. By relying on patterns and SOLID principles, the Java code is more flexible, easier to amend, and more straightforward to read. These are significant advantages for larger game projects and a simple snake implementation. Using Java and OOP, game design has considerable flexibility and scalability; integrating new components or refactoring old ones is more manageable and less risky than in Python. This theme has continued throughout this paper: Java requires more upfront cost, but it is worth the effort to develop the game properly, as linear programming requires more effort to expand and maintain.

Pygame and LibGDX have almost identical pros and cons to their respective languages. LibGDX is the more powerful tool and works with the more ideal game development language. It is a fantastic tool for creating custom games and provides substantial documentation and community support for those willing to learn. Pygame is very straightforward, and with little programming experience, a developer could create a robust small game. In some circumstances, particularly for single developers who want to develop small and flexible games or other interfaces, Python with Pygame may be the superior game development tool. That said, Java's structured application of OOP principles, support for critical programming tools like the singleton pattern and global variables, and enormous additional performance, coupled with the more powerful LibGDX, make it the superior language for developing games.

## References


Bad Logic Games. 2024. libGDX Javadoc.io. https://javadoc.io/doc/com.badlogicgames.gdx. Accessed: 2024-10-25.

Filho, R. M.; Bonfim, R.; Pessoa, L.; Barreto, R.; and de Freitas, R. 2024. Measuring the Execution Time of Programs from different Android Embedded Programming Languages. In *2024 L Latin American Computer Conference (CLEI)*, 1–9. ISSN: 2771-5752.

Goetz, B. 2006. *Java concurrency in practice*. Upper Saddle River, NJ: Addison-Wesley. ISBN 978-0-321-34960-6. OCLC: ocm66527178.

Kalb, I. 2022. *Object-Oriented Python: Master OOP by Building Games and GUIs*. New York: No Starch Press. ISBN 978-1-71850-207-9.

Kelly, S. 2019. *Python, Pygame, and Raspberry Pi Game Development*. Berkeley, CA: Apress L. P, 2nd ed edition. ISBN 978-1-4842-4533-0.

Miller, C. S.; Settle, A.; and Lalor, J. 2015. Learning Object-Oriented Programming in Python: Towards an Inventory of Difficulties and Testing Pitfalls. In *Proceedings of the 16th Annual Conference on Information Technology Education*, 59–64. Chicago Illinois USA: ACM. ISBN 978-1-4503-3835-6.

Overvoorde, A. 2019. *Modern OpenGL Guide*. GitHub.

Pygame. 2024. pygame v2.6.0 documentation. https://www.pygame.org/docs/. Accessed: 2024-10-25.

Sarcar, V.; and Mullick, A. 2020. *Interactive object-oriented programming in Java: learn and test your programming skills*. Springer eBook Collection. Berkeley, CA: Apress, second edition edition. ISBN 978-1-4842-5404-2.

Stemkoski, L. 2018. *Java game development with LibGDX: from beginner to professional*. Place of publication not identified: Apress, second edition edition. ISBN 978-1-4842-3324-5.

Wingqvist, D.; Wickström, F.; and Memeti, S. 2022. Evaluating the performance of object-oriented and data-oriented design with multi-threading in game development. In *2022 IEEE Games, Entertainment, Media Conference (GEM)*, 1–6. ISSN: 2766-6530.

Zapponi, C. 2024. GitHut 2.0.