MSCS-632: Practical Connection Assignment: Essay: Reflection on Applying Advanced Programming Language Concepts in the Workplace

---

Sandesh Pokharel

University of the Cumberlands

MSCS-632 Advanced Programming Languages

Dr. Vanessa Cooper

April 04, 2025

The Advanced Programming Languages course has been one of the most directly applicable classes I've taken so far in my MSCS program. As someone currently working as an Enterprise Application Developer at Paychex, I've found a lot of overlap between the theories, programming paradigms, and language concepts covered in this course and the real-world decisions I face daily while building and maintaining secure, scalable login systems. This course gave me a deeper understanding of the reasoning behind many language-level choices and trade-offs that I often had to make intuitively at work, especially around language selection, security, and performance.

One major area where I've seen a clear connection is the comparison of programming paradigms—object-oriented, procedural, functional, and scripting. While Java is inherently an object-oriented language and is the main technology stack in my current role, we've started using more functional features since Java 8, like lambdas, streams, and immutability. Before this course, I used these tools for convenience, but now I understand the core principles behind them—how functional paradigms reduce side effects, make concurrent systems more reliable, and help avoid shared state issues (Sebesta, 2016). This has changed the way I approach designing login flows, especially when dealing with stateless authentication and token validation in a multi-threaded environment.

This course also helped me appreciate language design and feature comparisons through our work with different languages like C++, Smalltalk, Rust, and Go. Learning Rust's ownership model made me more aware of memory safety and how crucial it is in secure systems. Even though I don't use Rust at work, the idea of strict compile-time checks to prevent null references and data races made me think about how we could tighten similar guarantees in Java—like avoiding NullPointerExceptions by using Optional, or applying stricter immutability principles to security-sensitive code (Matsakis & Klock, 2014). Similarly, understanding Go's simplicity

and built-in concurrency features made me reflect on how we use threads and executors in Java, especially when scaling login services to handle thousands of requests per second.

Ethical decision-making was also subtly embedded in our discussions on language trade-offs. For example, choosing a language for a project isn't just about speed or popularity—it's about safety, maintainability, and long-term impact on the team. At Paychex, as we migrate to Auth0 and handle authentication across multiple products, our decisions directly affect user security and privacy. This course made me more mindful of those ethical responsibilities, reinforcing the importance of selecting languages and tools that minimize vulnerabilities and support secure coding practices (NIST, 2020).

Additionally, concepts like memory management and garbage collection, which I had always taken for granted in managed languages like Java, now make more sense. Knowing how different languages handle memory—manual in C++, automatic in Java, hybrid in Rust—helped me optimize performance when integrating third-party libraries or configuring memory-sensitive applications. It's no longer just about "getting it to work," but getting it to work correctly, securely, and efficiently.

In summary, this course didn't just teach language theory—it taught me how to think like a language designer and an ethical software architect. It deepened my understanding of programming paradigms, influenced how I approach concurrency and security, and made me more conscious of the trade-offs in real-world language decisions. I now find myself reflecting more on the "why" behind the code, not just the "how." These insights are already shaping the way I write code, make design choices, and review pull requests at work.

References

Matsakis, N., & Klock, F. S. (2014). The Rust language. Retrieved from https://www.rust-lang.org/

NIST. (2020). Zero Trust Architecture (SP 800-207). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-207

Sebesta, R. W. (2016). Concepts of programming languages (11th ed.). Pearson.