Google Summer of Code 2025

Final Work Product Report

Modernizing Java PathFinder Extensions: Support for Java 11/17 Features

Student: Mahmoud Khawaja
GitHub: Mahmoud-Khawaja
Email: mahmoud.khawaja97@gmail.com

Organization: The JPF team
Mentors: Cyrille Artho

Project Duration: May 2025 - September 2025

Contents

1 Executive Summary

This report presents a comprehensive overview of my Google Summer of Code 2025 contributions to the Java PathFinder (JPF) project. Over the course of five months, I successfully modernized multiple JPF extensions to support Java 11 and Java 17 features, addressing critical compatibility issues that prevented JPF from analyzing modern Java applications.

The project encompassed 16 significant pull requests across three key repositories: JPF-Core (14 pull requests), JPF-NAS (1 major pull request), and JPF-NHandler (1 pull request). Key achievements include implementing Java Records support, Bootstrap Methods for dynamic method invocation, Sealed Classes functionality, SharedSecrets interfaces, and comprehensive build system modernization from Ant to Gradle.

The work has successfully enabled JPF to verify Java applications using modern language features, significantly expanding its applicability to contemporary software development practices. This modernization effort represents a substantial contribution to the Java verification community and establishes a foundation for future Java version support.

2 Project Overview and Objectives

2.1 Background

Java PathFinder (JPF) is a model checker specifically designed for Java applications, developed by NASA. Despite its powerful verification capabilities, JPF extensions had fallen behind in supporting modern Java versions, limiting their usefulness for analyzing contemporary Java applications that leverage Java 11+ features.

2.2 Project Goals

The primary objectives of this project were:

- Enable JPF-Core to support Java 11 and Java 17 language features
- Modernize JPF-NAS for network simulation with current Java APIs
- Update JPF-NHandler for compatibility with modern Java runtimes
- Migrate build systems from Ant to Gradle for better dependency management
- Implement comprehensive testing and continuous integration workflows
- Ensure backward compatibility while adding new functionality

3 JPF-Core Contributions

JPF-Core serves as the foundation engine for all Java PathFinder extensions. My contributions to this repository focused on implementing critical Java 11/17 language features and enhancing the overall robustness of the verification engine.

3.1 Pull Requests Summary

PR #	Title & Link	Description
556	Enhanced SharedSecrets for Java 11 integration	Added JavaNetInetAddressAccess and JavaSecurityAccess interfaces to SharedSecrets with missing accessor methods for Java 11 compatibility
554	Added direct execution for records	Implemented direct execution optimization for Java record types in INVOKEDYNAMIC bytecode handling
553	Added JavaNetSocketAccess support to SharedSecrets	Added socket access support to SharedSecrets for network-related JPF extensions to function with Java 11
552	Added the direct approach for handling lambda	Implemented direct approach for lambda expression handling with enhanced bootstrap method processing
550	String concat call site	Implemented call site generation for string concatenation operations using StringConcatFactory
545	Sealed classes full	Complete implementation of Java 17 sealed classes with inheritance constraints and validation
543	Enabled ParallelTesting	Enhanced build system with parallel testing capabilities and improved CI/CD workflows
540	Fixed the field access for records	Fixed field access mechanisms for Java record types with proper final field handling
530	Adding support for internals	Major implementation of Java record internals including bootstrap methods, equals(), hashCode(), toString()
528	Class version check	Updated class version validation to accept Java 17 bytecode (version 61)
522	Fixes #355: Added Method dispatch test scenarios	Enhanced test coverage for method dispatch mechanisms addressing issue #355
520	Record support	Initial comprehensive record support implementation serving as foundation
519	Refactored the old testing class for records	Refactored existing record tests for better structure and coverage
518	Implemented toGenericString() in Field.java and Method.java	Implemented toGenericString() methods in Field.java and Method.java for improved reflection API compatibility

3.2 Technical Deep Dive

3.2.1 Records Implementation

The implementation of Java Records represented one of the most complex aspects of this project. Records in Java are immutable data carriers with automatically generated methods. My implementation included:

- Automatic generation of equals(), hashCode(), and toString() methods
- Deep equality checking for record components containing arrays or other complex types
- Support for generic record types with proper type parameter handling
- Integration with JPF's object lifecycle management and garbage collection
- Proper handling of record component access and validation

The technical challenge involved ensuring that the generated methods behaved identically to the JVM's native implementation while working within JPF's model checking environment. This was accomplished through PRs #520, #530, #540, and #554.

3.2.2 Bootstrap Methods and Call Site Generation

Modern Java relies heavily on bootstrap methods for dynamic feature implementation. My implementation covered:

- String concatenation via StringConcatFactory (PR #550)
- Lambda expression initialization through LambdaMetafactory (PR #552)
- Method handle resolution for dynamic method invocation
- Call site caching and invalidation strategies
- Direct execution optimization for record methods (PR #554)

This work required deep understanding of the JVM's invokedynamic instruction and how it interfaces with bootstrap method handles.

3.2.3 Sealed Classes Support

Java 17's sealed classes restrict which classes can extend or implement them. The implementation (PR #545) included:

- Compile-time verification of sealed class constraints
- Runtime enforcement of inheritance restrictions
- Integration with pattern matching infrastructure for future expansion
- Proper handling of permitted subclasses declarations

3.2.4 SharedSecrets Enhancement

Critical for Java 11+ compatibility, the SharedSecrets enhancement (PRs #553, #556) provided:

- JavaNetSocketAccess interface for network operations
- JavaNetInetAddressAccess for address resolution
- JavaSecurityAccess for security framework integration
- Proper accessor method implementation for JPF extensions

4 JPF-NAS Contributions

JPF-NAS provides network simulation capabilities for JPF. My work focused on modernizing this extension to work with Java 11+ networking APIs and resolving critical compatibility issues.

4.1 Pull Request Summary

PR #	Title & Link	Description
5	Gradle support	Comprehensive migration to Gradle build system with Java 11 support, including 24 commits covering build modernization, socket implementation, and peer class development (+1,952 additions, -1,202 deletions)

4.2 Technical Implementation

4.2.1 Build System Modernization

The migration from Ant to Gradle involved:

- Creating comprehensive build.gradle configuration with Java 11 target compatibility
- Establishing proper dependency management for JPF-Core integration
- Implementing automated testing with JUnit framework
- Setting up GitHub Actions CI/CD workflow for continuous integration
- Proper source set configuration for main, test, and examples
- Custom JAR creation tasks for model classes and examples

4.2.2 Socket Implementation Enhancement

The Java 11 socket support implementation included:

- Implementation of PlainSocketImpl peer classes to intercept socket initialization
- Creation of comprehensive Java 11 socket API support including:
  - JPF_java_net_PlainSocketImpl: Socket initialization and ExtendedSocketOptions bypass
  - JPF_java_net_InetAddress: Hostname resolution and address management
  - JPF_java_net_SocketCleanable: Java 11 automatic resource cleanup bypass
  - JPF_java_lang_ref_Reference: Reference management with reachabilityFence() support
  - JPF_jdk_internal_ref_CleanerFactory: Cleaner system integration management
- Resolution of SharedSecrets compatibility issues for Java 11
- Enhanced ServerSocket.java with comprehensive timeout and connection handling
- Proper module system integration using –patch-module compilation approach

4.2.3 Java 11 Compatibility Resolution

The project addressed critical Java 11 compatibility issues:

- **SharedSecrets Issue**: Resolved NoSuchMethodException for setJavaNetSocketAccess() by implementing proper interfaces
- **Module System Conflicts**: Used –patch-module approach to resolve conflicts between JPF model classes and java.base module
- **ExtendedSocketOptions**: Created peer classes to handle Java 11 extended socket options initialization
- **Classpath Problems**: Fixed build configuration to include proper class directories for JPF test execution

5 JPF-NHandler Contributions

JPF-NHandler manages native method handling within JPF. This extension required updates to support modern Java native method interfaces.

5.1 Pull Request Summary

PR #	Title & Link	Description
15	Gradle java 11	Migrated build system to Gradle with Java 11 compatibility, resolved API dependencies, and excluded deprecated Google Translate examples

5.2 Technical Implementation

The JPF-NHandler update involved:

- Migrating build configuration with proper Java 11 compatibility
- Updating dependency management to use jpf.Jar.getFullPath() for dynamic JPF-Core path resolution
- Excluding problematic example code that depended on deprecated Google APIs (GoogleAPI and ReceiverAdapter)
- Ensuring proper integration with updated JPF-Core APIs and SharedSecrets interfaces
- Testing native method handling compatibility with modern Java applications

6 Challenges and Solutions

6.1 Java Module System Integration

One of the most significant challenges was resolving conflicts between JPF's model classes and the Java Platform Module System (JPMS). The solution involved:

- Using –patch-module compilation approach to overlay JPF model classes onto java.base module
- Careful management of module path vs. class path dependencies
- Ensuring proper reflection access for JPF's introspection needs while respecting module boundaries
- Maintaining compatibility with both modular and non-modular applications

6.2 Bootstrap Method Complexity

Implementing bootstrap methods required understanding the intricate relationship between:

- JVM bytecode generation and invokedynamic instructions
- Method handle creation and resolution mechanisms
- Call site linking and invalidation strategies
- Memory management for dynamically generated code within JPF's verification environment

The solution involved creating a robust call site generation framework that could handle various bootstrap method patterns while maintaining JPF's state tracking capabilities.

6.3 SharedSecrets API Changes

Java 11 introduced significant changes to internal networking APIs that JPF-NAS relied upon:

- **Problem**: NoSuchMethodException for SharedSecrets.setJavaNetSocketAccess()
- **Root Cause**: Java 11 changed internal networking API structure
- **Solution**: Implemented JavaNetSocketAccess interface and proper registration in SharedSecrets (PRs #553, #556)
- **Result**: Enabled JPF's network interception architecture to function with Java 11+

6.4 Record Type Semantics

Implementing records correctly required careful attention to:

- Ensuring immutability constraints were properly enforced
- Generating methods that precisely matched JVM behavior
- Handling complex nested equality scenarios and generic type parameters
- Integrating with JPF's object creation and lifecycle management
- Fixing field access issues for final record components

7 Testing and Validation

Each implementation underwent rigorous testing to ensure correctness and compatibility:

7.1 Unit Testing

- Comprehensive test coverage for all new Java 11/17 features
- Edge case testing for complex record scenarios and nested structures
- Integration testing with existing JPF functionality to ensure backward compatibility
- Socket simulation testing with timeout handling and connection management
- Enhanced test scenarios for method dispatch (PR #522)
- Parallel testing capabilities for improved CI performance (PR #543)

7.2 Build System Validation

- GitHub Actions CI/CD workflows for automated testing
- Multi-platform compatibility verification
- Gradle wrapper ensuring consistent build environments
- Dependency resolution testing for JPF-Core integration

8 Impact and Future Work

8.1 Immediate Impact

This work has immediately enabled JPF to:

- Analyze modern Java applications using Java 11/17 features including Records, Sealed Classes, and enhanced APIs
- Support contemporary frameworks and libraries that depend on modern Java language features
- Maintain relevance in the current Java ecosystem and development practices
- Provide verification capabilities for cutting-edge Java development with proper build automation
- Handle complex string concatenation and lambda expressions through proper bootstrap method support

8.2 Community Benefits

- Expanded JPF's user base to include developers working with modern Java versions
- Established patterns and best practices for future Java version support
- Improved build and deployment processes for all JPF extensions through Gradle modernization
- Enhanced documentation and testing practices for the JPF ecosystem
- Created foundation for further Java language feature support

9 Lessons Learned

This project provided valuable insights into:

- The complexity of modern JVM feature implementation and internal API evolution
- The importance of maintaining backward compatibility during major modernization efforts
- The value of comprehensive testing in verification tool development and reliability
- The challenges of integrating with rapidly evolving language specifications
- The critical importance of proper build system design for complex multi-module projects
- The intricate relationship between bootstrap methods and modern Java language features

10 Repository Links and Documentation

- **JPF-Core:** https://github.com/javapathfinder/jpf-core
  - Java 17 branch: https://github.com/javapathfinder/jpf-core/tree/java-17
  - Call site generation branch: https://github.com/javapathfinder/jpf-core/tree/call-site-generation
- **JPF-NAS:** https://github.com/javapathfinder/jpf-nas
  - Java 11 branch: https://github.com/javapathfinder/jpf-nas/tree/java-11
- **JPF-NHandler:** https://github.com/javapathfinder/jpf-nhandler

11 Acknowledgments

I would like to express my sincere gratitude to:

- **Cyrille Artho** - Primary mentor who provided invaluable technical guidance, thorough code reviews, and project direction throughout the entire GSoC period
- **The JPF Team** - For accepting me into the GSoC program and providing access to this incredible verification tool ecosystem
- **Google Summer of Code** - For facilitating this amazing learning and contribution opportunity
- **JPF Community** - For their support, feedback, and collaborative development environment

This project has been an extraordinary learning experience that significantly enhanced my understanding of:

- Java virtual machine internals and bytecode manipulation
- Modern Java language feature implementation at the JVM level
- Software verification and model checking concepts and applications
- Open source collaboration, code review processes, and community development
- Build system design and continuous integration best practices
- Bootstrap method implementation and dynamic method invocation

12 Conclusion

This Google Summer of Code 2025 project successfully modernized the Java PathFinder ecosystem to support Java 11 and Java 17 features. Through 14 merged pull requests to JPF-Core, 1 comprehensive pull request to JPF-NAS, and 1 pull request to JPF-NHandler, I implemented complex language features including Records, Bootstrap Methods, Sealed Classes, and comprehensive API enhancements.

The work represents a significant milestone in JPF's evolution, ensuring its continued relevance and applicability to modern Java development. The implementations are production-ready, thoroughly tested, and well-documented, providing a solid foundation for future enhancements and community contributions.

Key achievements include:

- Complete Java Records support with all generated methods (equals, hashCode, toString)
- Bootstrap method implementation enabling string concatenation and lambda expressions
- Java 17 Sealed Classes with full constraint validation
- SharedSecrets interfaces enabling Java 11+ networking compatibility
- Comprehensive build system modernization across all extensions
- Enhanced testing frameworks and parallel test execution

The project not only achieved its primary technical objectives but also established improved development practices, comprehensive testing frameworks, and modernized build systems that will benefit the JPF community for years to come. This contribution ensures that Java PathFinder remains a vital tool for Java software verification in the contemporary development landscape.

The successful resolution of critical Java 11 compatibility issues in JPF-NAS, implementation of modern language features in JPF-Core, and modernization of build systems across all extensions demonstrates the project's comprehensive scope and lasting impact on the Java verification community.

This report was prepared for Google Summer of Code 2025 final evaluation.
All source code and documentation are available in the respective GitHub repositories.