Core Java : An Object Oriented Language

Ananya Anikesh
Department of Information & Technology Engineering
Dronacharya College Of Engineering, Khentawas, Gurgaon

Abstract- **This research paper is a brief overview about Core Java an object oriented language. Objects are key to understanding object oriented technology. The common practice of object oriented prototyping is to define the fragments of java. High-level languages allow some English-like words and mathematical expressions that facilitate better understanding of the logic involved in a program. While solving problems using high-level languages, importance was given to develop an algorithm (step-by-step instructions to solve a problem). While solving complex problems, a lot of difficulties were faced in the algorithmic approach. Hence, object oriented programming languages such as C++ and Java were evolved with a different approach to solve the problems. Object-oriented languages are also high-level languages with concepts of classes and objects.**

I. INTRODUCTION

Java is a set of computer software and specifications developed by Sun Microsystems, later acquired by Oracle Corporation, that provides a system for developing application software and deploying it in a cross-platform computing environment. Java is used in a wide variety of computing platforms from embedded devices and mobile phones to enterprise servers and supercomputers. While less common, Java applets run in secure, sandboxed environments to provide many features of native applications and can be embedded in HTML pages.

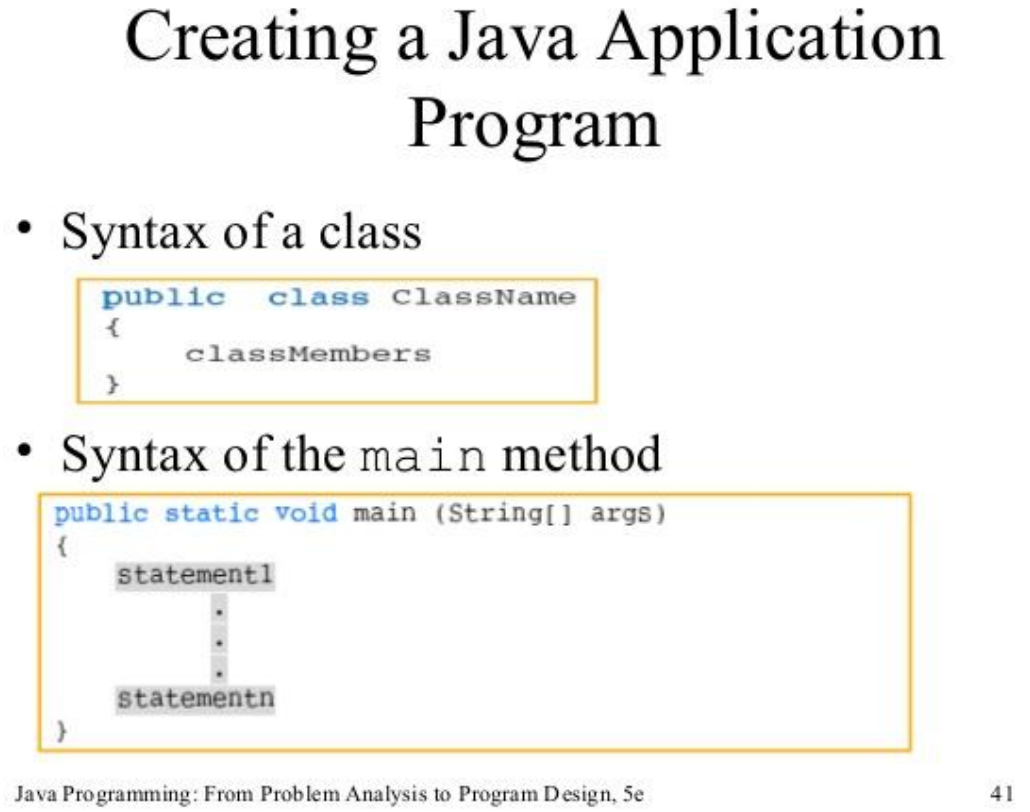
Writing in the Java programming language is the primary way to produce code that will be deployed as byte code in a Java Virtual Machine (JVM); byte code compilers are also available for other languages, including Ada, JavaScript, Python, and Ruby. In addition, several languages have been designed to run natively on the JVM, including Scala, Clojure and Groovy. Java syntax borrows heavily from C and C++, but object-oriented features are modeled after Smalltalk and Objective-C. Java eschews certain low-level constructs such as pointers and has a very simple memory model where every object is allocated on the heap and all variables of object types are references. Memory management is handled through integrated automatic garbage collection performed by the JVM.

II. CORE JAVA LANGUAGE

The syntax of Java is largely influenced by C++. Unlike C++, which combines the syntax for structured, generic, and object-oriented programming, Java was built almost exclusively as an object-oriented language. All code is written inside classes, and every data item is an object, with the exception of the primitive data types, *i.e.* integers, floating-point numbers, boolean values, and characters, which are not objects for performance reasons. Java reuses some popular aspects of C++ (such as printf() method).

Unlike C++, Java does not support operator overloading or multiple inheritance for classes, though multiple inheritance is supported for interfaces. This simplifies the language and aids in preventing potential errors and anti-pattern design.

Java uses comments similar to those of C++. There are three different styles of comments: a single line style marked with two slashes (//), a multiple line style opened with /* and closed with */, and the Javadoc commenting style opened with /** and closed with */. The Javadoc style of commenting allows the user to run the Javadoc executable to create documentation for the program.



III. JAVA TO CORE JAVA TRANSLATION

We formulate the translation as a set of type-directed rules that follow the syntax of the Java source language. The rules are type preserving, that is, they guarantee that both programs, the Java input and the Core-Java output, have the same type. Our algorithm consists of three main steps: Generating Descriptors of Compilation Units. For each compilation unit, we generate its attached descriptor file. A descriptor consists of typing information of each interface, class and their fields and methods, but without the method body. Computing the Global Dependency Graph. Our translation is required to process the class and interface declarations in some particular order given by the complex inter-dependency among classes, interfaces and methods. The dependency graph has the class and interface declarations organised into a hierarchy of strongly connected components (SCCs). Through a bottom-up processing of each SCC, we perform the translation in a systematic fashion. The global dependency graph is also kept after the translation to be used by the subsequent program analyses. Translation. The type-based translation is formulated as a modular type inference for the Java input program. The main judgement has the following form: D, Γ e $\Rightarrow$ exp e : τ denoting the translation of a Java expression e into a Core-Java expression e , where e and e have the type τ with respect to the type environment, Γ and the set of descriptors, D. Details about our translation rules can be found in the companion technical report.

IV. CONCLUSION AND FUTURE WORK

We have implemented the translator in Haskell and we used it to help two analyses: a region inference for Java and a type checker of a variant parametric type system for Java . The translator was very useful in extending the experiments of our projects to real world applications. Our goal is to have an integrated framework: translator, global dependency graph and any other specific data structure that can help and simplify the program analyses. Another aspect is the correctness of the translation rules. We experimentally validated that translated programs are correct and currently we are working on a formal proof of the translation rules.

REFERENCES

[1] Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical report, Cambridge University, 2003.

[2] Wei-Ngan Chin, Florin Craciun, Siau-Cheng Khoo, and Corneliu Popeea. A Flow-Based Approach for Variant Parametric Types. In ACM OOPSLA, Portland, 2006.

[3] Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin Rinard. Region Inference for an Object-Oriented Language. In ACM PLDI, Washington, 2004.

[4] Florin Craciun, Hong Yaw Goh, and Wei-Ngan Chin. A Framework for Object-Oriented Program Analyses via Core-Java. Technical report, National University of Singapore, 2006. avail. at http://www.comp.nus.edu.sg/~chinwn/papers/corejava.ps.