Security in Java Features That Differentiates Java from Other Languages

Kajal Kanika & Monika Malhotra
Information Technology Branch Dronacharya College of Engineering Gurgaon, India
monikamalhotra15@yahoo.com

Abstract—

Java is a modern programming language that consists of new age programming features like Data Abstraction, Object Oriented Programming, Multicore Programming, and Thread management and is Platform Independent. Since the beginning, Java has been at the centre of a culture of innovation. Its original release redefined programming for the Internet. The Java Virtual Machine (JVM) and byte-code changed the way we think about security and portability.
Java's approach of write once and run everywhere caught everyone's attention. It reduces the amount of effort programmers put to deploy their code on different platforms. While is java is platform independent, it raises the question of how secure it is on different platforms. One way, the security could be compromised on different platforms is via illegally accessing memory. For that reason, Java doesn't allow the use of Pointers in the code. One may think, its one of the core feature of a programming language and even C++ allows to use pointers. But people can perform most of their task in java without the need of pointers.

Keywords—

JAVA; Security; Portability; Pointer-less

INTRODUCTION

The fundamental forces that necessitated the invention of Java are portability and security, other factors also played an important role in moulding the final form of the java language. The key following considerations were summed up by the Java team:

A. Simple
B. Secure
C. Portable
D. Object-oriented
E. Robust
F. Multithreaded
G. Architecture-neutral
H. Interpreted
I. High performance
J. Distributed
K. Dynamic

We will look into Security and Portability in detail one by one later in this paper. Let's start with the byte-code first.

I. BYTE-CODE

The key point that allows Java to take care of both the security and the portability problems is the Output of the compiler. The output of a Java compiler is not executable code. Rather, it is a bytecode. Bytecode is a highly optimised set of instructions designed to be executed by the Java run-time system, called the Java Virtual Machine (JVM). The original JVM was designed as an interpreter for bytecode. This looks like a surprise because most modern programming languages are made to be compiled into executable code because of performance concerns. However, the fact that a Java program is executed by the JVM helps solve the major problems associated with web-based applications.

Translating a Java program into bytecode makes it easier to run a program ina wide range of platforms because only the JVM needs to be implemented for each platform. Once the runtime package exists for a given system, any Java program can run on it. Although the details of the JVM will differ from platform to platform, but all of them understand the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU and platform connected to the Internet. This is not a feasible solution. Thus, the execution of bytecode by the JVM is the easy and efficient way to create portable programs.

The fact that a Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it acts like a sandbox that contain the program and prevent it from generating side effects outside of the system. As we will see, safety is also enhanced by certain restrictions that exist in the Java language.

In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code. However, with Java, the differential between the two is not so great. Because byte-code has been highly optimised, the use of bytecode enables the JVM to execute programs much faster than one might expect.

Although Java was created as an interpreted language, there is nothing about Java that prevents on-the-fly compilation of bytecode into native code in order to enhance performance. For this reason, Sun began supplying its Hotspot technology not long after Java's initial release. Hotspot provides a Just-In-Time (JIT) compiler for bytecode. When a JIT compiler is a part of the JVM, selected portions of bytecode are compiled into executable code in real time, on a piece-by-piece and demand basis. It is important to understand that it is not practical to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only during the program run time. Instead, a JIT compiler compiles code as it is needed, during execution. Furthermore, not all sequences of bytecode are compiled, only those that will favor good from compilation. The remaining code is simply interpreted as it is. However, the just-in-time approach still yields a significant performance boost. Even when dynamic compilation is applied to bytecode, the portability and safety features still apply, because the JVM is still in charge of the execution environment.

II. ROBUSTNESS OF JAVA

The multi-plat formed environment of the internet put huge demands on a program, because the program must execute reliably in various systems. Thus, the ability to create robust programs was given a high priority in the design of Java language. To gain reliability, Java restricts us in a few key areas that force us to find our mistakes early in software development. At the same time, Java frees us from worrying about many of the most common programming errors. Because Java is a strictly typed language, it checks the code at compile time. However, it also checks the code at run time. Many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what we have written will behave in a predictable way under diverse conditions is a key feature of Java language.

To better understand how Java is robust, let's consider two main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and deallocation for us. (Deallocation is completely automatic, because Java provides garbage collection for unused objects.) Exceptional conditions in traditional environments often arise in situations such as division by zero or "file not found," and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object-oriented exception handling.

In a well-written Java program, all run-time errors can and should be managed by our program.

III. Security in Java

We all are likely aware, every time we download a “normal” program, we take a risk, because the code we are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources. For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer’s local file system. In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack.

Java achieved this protection by confining an applet to the Java execution environment only and not allowing it access to other parts of the computer. The ability to download applets with confidence that no harm will be done and that no security will be breached is considered by many to be the single most innovative aspect of Java.

Applets can be very useful in java, but it serves as just one half of the client/server architecture equation. Not long after the initial release of Java, it became obvious that Java would also be useful on the server side as well. The result was the servlet. A servlet is a small program that executes on the server. Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server. Thus, with the advent of the servlet, Java spanned both sides of the client/server connection.

Servlets are used to create dynamically generated contention the server that is then served to the client on the client side. For example, an online store might use a servlet to look up the price or description for an item in the database. The price and description information is then used to dynamically generate a web page that is sent to the browser. Although dynamically generated content is available through mechanisms such as CGI (Common Gateway Interface), the servlet offers several advantages, including increased performance and efficiency.

Because servlets (like all Java programs) are compiled into bytecode and executed by the JVM, they are highly portable. Thus, the same servlet can be used in a variety of different server environments. The only requirements are that the server supports the JVM and a servlet container.

V. Conclusion

In this paper we have presented that how java is a secure language and the features that differentiate java from other languages out there in use. Since Java uses bytecode which in turn gets executed in a separate environment (JVM) generates a confidence that java is secure and robust. Java does limit us from using various programming features available in other languages for the sake of security. For example, Pointers but in java one don’t need a pointer. It also serves well in client-server architecture and is multi-platform but still maintain the security level properly in all environment which is commendable.

References

[1]. Dragoni, N., Massacci, F., Naliuka, K., & Siahaan, I. (2007). Security-by-contract: Toward a semantics for digital signatures on mobile code. In *Public Key Infrastructure* (pp. 297-312). Springer Berlin Heidelberg.

[2]. Caromel, D., & Vayssière, J. (2001). Reflections on MOP s, Components, and Java Security. In *ECOOP 2001—Object-Oriented Programming* (pp. 256-274). Springer Berlin Heidelberg.

[3]. Amme, W., Dalton, N., von Ronne, J., & Franz, M. (2001). *SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form* (Vol. 36, No. 5, pp. 137-147). ACM.

[4]. Flenner, R. (Ed.). (2003). *Java P2P unleashed*. Sams Publishing.