TALLINN UNIVERSITY OF TECHNOLOGY
Department of computer engineering
Chair of Digital Systems Design

Oleg Dmitrijev
121998 IASM

Essential characteristics of Java

Analysis of Programming Languages (IAG0450)
Essay

Presented: 19.11.2013

Tallinn 2013

Foreword. Evolution of Java.

The Java project started by James Gosling, Mike Sheridan, and Patrick Naughton in June 1991. The initial name of the language was Oak and it was intended to implement in smart devices and TV. Sun Microsystems released first version of Java JDK 1.0 in January 1996. According to Wikipedia definition "Java is a computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another. Java applications are typically compiled to bytecode (class file) that can run on any Java virtual machine (JVM) regardless of computer architecture." Already in February 1997 next edition of Java JDK 1.1 was released. This version got extensive retooling of the AWT event model, nested classes, Java Database Connectivity (JDBC) API, Java Remote Method Invocation (RMI) – Java analogue of remote procedure calls (RPC), reflection technology and JavaBeans classes. In same year, Sun Microsystems did its first attempt to standardize Java, but soon refused from this idea. Subsequently, Java became a standard controlled through the Java community process[1].

The next Java version was 1.2 (also called as Java 2) or J2SE 1.2 (Standard Edition). This release almost tripled the size of the platform to 1520 in 59 packages. It contained such significant addition and modification as first of all JIT(Just-In-Time) compiler, completely new graphical library Swing, Java IDL (Interface Definition Language) as implementation of CORBA specification, Java plugins and JCF(Java Collection Framework).

In May 2000 Sun Microsystems released J2SE 1.3. This release included HotSpot JVM, Java Platform Debugger architecture, Java Naming and Directory Interface, JavaSound, synthetic proxy classes and RMI was modified to support CORBA optionally.

Since J2SE 1.4, released in February 2002, the evolution of the Java language has been governed by the Java Community Process (JCP), which uses Java Specification Requests (JSRs) to propose and specify additions and changes to the Java platform[2]. J2SE 1.4 included modification and additions as regular expressions, IPv6 support, JAXP (Java API for XML processing), NIO (New I/O) collection of Java APIs for non-blocking I/O operations, image

[1] http://www.jcp.org/en/home/index
[2] http://en.wikipedia.org/wiki/Java_version_history

API, logging API, preferences API, Java Web Start and integrated security and cryptography extensions.

Next J2SE 5.0 released in September 2004 brought such additions and improvements as generics and enumerations, components of Swing library got skins, boxing and unboxing of types, variable arguments of method called Varargs, static imports and enhanced "for each" loop. Also firstly, Java platform got tools for advanced metadata generation called annotations.

Java SE 6 started in December 2006 had the next major improvements and new features as significant JVM performance improvements, as result overall performance of the platform and Swing increased, Java compiler API, support of scripting languages, improved support of web services with JAX-WS, JDBC 4.0, JAXB 2.0 with integration of StAX parser. In addition, Sun and then Oracle released 65 bug-fixing and minor updates of Java SE 6, the last one was launched in October 2013.

Java SE 7 released in July 2011 included compressed 64-bits pointers, syntax enhancements like strings in "switch" statement, automatic resource management in "try" statement, improved type inference for generic instance creation, simplified Varargs method declaration, binary integer literals, allowing underscores in numeric literals, catching multiple exception types and re-throwing exceptions with improved type checking[3]. Also were added and improved such features as new file I/O library, support of elliptic curve cryptography algorithms, support of new network protocols (like SCTP and Socket Direct Protocol), JVM support for dynamic languages, XRender support for Java2D and upstream updates to XML and Unicode.

Next Java SE 8 is expected in March 2014.

Syntactic and semantic definition of language.

Java belongs to so-called C family of programming languages and has similar syntax to other members of this family like C, C++ and C#. That is the reason why many features of C or C++ syntax take place in Java. Similarities with other members of C-family are next: syntax of all basic operations (mathematical, logical), names of build-in elementary data type (except

[3] http://en.wikipedia.org/wiki/Java_version_history

boolean), beginning and end of the blocks are defined by parenthesis, the end of the statement is semi-colon, syntax of conditional statements like if/else or switch/operations, syntax loop statements (for, while and do/while), declaration of variables in style <access_modifier>[type] [name_of_variable], names and usage of access modifiers (public, private and protected) are same like in C++ or C#, enumerations reserved word “enum” (since version J2SE 5.0).Java has three types of comments: classical // or /* */ and Javadoc style that since version 5.0 can be used for project documentation. Although "goto" is reserved word in Java hasn't command "goto". Instead it has similar mechanism called "label". From Object-C Java got operator “interface” means set of signatures without implementation of lasts. Interfaces are used for multiple inheritance in Java. Operator “implements” used in Java has analogue in C++ and C# operator “:” after name of the class meaning that class inherits from object interface behind the operator. In case of inheritance from class used operator “extends”.

Semantically Java is algorithmic, concurrent, class-based, object-oriented programming language. Like in case of syntax semantic similar to all representatives of C family, especially to C#. Here are set of statements comparing semantic of Java with other ones:

- Unlike C++ semantic of Java does not allow default values of arguments.
- All methods of the classes are virtual by default and can be overridden in inherited classes.
- In Java all primitive parameters are passed by value, it has not C++ style passing by reference.
- Pointers, which are milestones of C/C++ programming that, allow direct manipulating with memory addresses don exist in Java. Java (like in C#) has references to objects (not to primitive types), other references and arbitrary memory locations.
- In Java like in C# and unlike in C++ native types have value semantics and user-defined types like classes have reference semantic. For example objects should be created using operator new (), otherwise user gets another one reference to an object or an empty reference.
- Java does not support operator overloading.
- Java has such immutable type of data like string - a sequence of Unicode symbols. Once created it can't be modified, any attempts of modification will lead to creation of new modified string in memory, and same time first original one will be kept memory. Exactly same semantic mechanism is implemented in C#.
- Java doesn't support unsigned integer types like C/C++/C# languages do it.

For many users it's rather disadvantage.

Authors of Java meaningly refused from preprocessor directives and headers (*.h) files. The goal was first to simplify Java because using of preprocessor command may lead to creation of difficultly understandable code. And at the second using of header files distributed separately from their binary code is the security risk because malicious users created this code can get access to private data object. Instead of header files Java uses packages with directive "import".

The interesting fact is that all types, even built-in primitives like integers or doubles, of data in Java are objects and located in memory heap. For instance, the concept of C# is that primitive objects located in stack, even structures are primitive object and can be located in stack, and operations responsible for relocations from/to stack are called boxing and unboxing. But in Java all data located in heap, in other words creation of integer means that initially JVM creates object Int that has attributes int and two methods one for assignment and second for getting the value.

Reliability

The C++ has greatest influence on Java, but some of its low-level operations like direct operations with memory were meaningly excluded from Java by reliability reasons.

On the one hand direct memory operations are very powerful tool, but on other it brings additional vulnerability due to human errors. For example in C actually has not native support of arrays in its nowadays meaning and C programmer has deal with set of pointers to the memory addresses.

JVM translation of Java codes to byte code is closed from user. It means the method how the code will be translated depends on the JVM translation algorithms. This measure helps to skip such familiar for C programmers problems like buffers overflow and memory leaks. All user defined objects in Java is dynamically allocated and garbage collector is responsible for de-allocation.

Java is strongly-typed language, it has set of rules and certain operation can be performed only with certain data types. Java supports widening implicit types conversion if there is no threat of data loss, an user can convert for example integer value to double. Also Java supports explicit cast-type conversion of similar types like double to float meaning that user is ware that data loss is possible. However, you cannot convert string to integer in Java.

Java does not support such feature of C++ like multiple inheritance from classes, it helps to avoid ambiguity in child classes when both parents has method or variable with same name. Despite on this statement that members of the class has same names can be achieved with implementation of multiple interfaces, this is allowed in Java, but Java has built-in defense mechanism from such situations - methods with same name and signature will not be implemented.

A strongly-typed programming language means that there is certain set of built-in data types, so called primitive data types, and all operations are only allowed with these data types. For example such data types in Java boolean, byte, char, short, int, long, float, double, also there void and string. Such approach can guarantee some kind of reliability of computation result. Only those types can be used, a programmer can not define his/her own types of data for instance some unsigned integer in his/her system. It sounds like standardization of types and the goal is to enforce reliability of entire Java platform.

Already in JDK 1.1 were added classes with digital signature. It gave an opportunity to authenticate authority was done for security reasons. From this point of view in addition to the fact that JIT analyzes user code for optimization and malicious code will not be performed (stack overflow, intruding to memory areas not belonging to the process etc.) Java has mechanism to run code only from trustable source.

Also for security reasons Java applets running in so-called sandbox – environment with right limitation (no access to file system, cannot change security police, cannot load native libraries end so on).

Fast translation

Although Java is not compiled language and paradigm of compiled languages insists on advantages of native compiled languages in speed of data processing, this assert is rather controversial. Already since 1998 in J2SE 1.2 was presented JIT (Just-In-Time) compiler that translates bytecode to native command in running time. From this started permanent speed optimization model of Java. In Java SE 6 released on 11 December 2006 computation speed increased up to 70% comparing to previous versions. Now this speed is comparable with computation speed of native languages being 20% behind of them and even exceeds in some circumstances. Also from version Java SE 6 was significantly improved workability of Swing graphical library in OpenGL and DirectX.

Java HotSpot Performance Engine was introduced April 27, 1999. Initially developed by Longview Technology small company that were acquired by Sun Microsystems in 1997. In the begging HotSpot was planned as improvement of J2SE 1.2, but from J2SE 1.2 the engine became main JVM. The general idea of HotSpot was determing of "hottest" spots in byte code and providing more computation resources to these spots at the expenses of reducing of resources value for the rest of the program code.

Overall performance of Java code execution depends on how JVM optimizes it according features of this particular OS and hardware. Since J2SE 5.0 compressed oops technology was introduced. An “oop” or “ordinary object pointer” is a managed pointer in Java terminology. Originally in many systems it was same size as machine native pointer. For instance, on an LP64 system a machine word is 64 while on a LP32 system it’s accordingly 32 bits. It means on LP32 heap size limitation (less than 4 GB). Compressed oops represent managed pointers (in many but not all places in the JVM) as 32-bit values which must be scaled by a factor of 8 and added to a 64-bit base address to find the object they refer to. This allows applications to address up to four billion objects (not bytes), or a heap size of up to about 32 GB. At the same time, data structure compactness is competitive with ILP32 mode.[4] This is significantly reduce memory consumption compares with using 64-bit references as Java uses references much more than some languages like C++[5].

JVM optimization has even benefits over compiled C and C++ languages because some optimization techniques are not available or significantly disturbed due to the nature of compiled languages. For example, usage of pointers can interfere optimization in those languages. In Java widely used escape analysis, a method for determing the dynamic scope of pointers. Nevertheless, in C/C++ the technique is limited because a compiler does not know whether an object will be modified in this block or not due to pointers. Also Java can access to derived instance method faster than C++ can access derived virtual methods due to C++'s extra Virtual-Table look-up. However, non-virtual methods in C++ do not suffer from V-Table performance bottlenecks, and thus exhibit performance similar to that of Java

On November 1997 started collaboration between Seoul National University and IBM T.J. Watson Research Center – the LaTTe project. The goal of this collaboration was to create an open-source Java VM JIT compiler for RISC processor. As result of that on October 1999 first release of LaTTe code. LaTTe has such features like lightweight monitor, efficient exception handling and efficient garbage collection and memory management. It translates

[4] https://wikis.oracle.com/display/HotSpotInternals/CompressedOops
[5] http://en.wikipedia.org/wiki/Java_performance

bytecode into SPARC native code in five stages. In the first stage, LaTTe identifies all control join points and subroutines in the bytecode via a depth-first traversal. In the second stage, the bytecode is translated into a control low graph (CFG) of pseudo SPARC instructions with symbolic registers. In the optional third stage, LaTTe performs some "traditional" optimizations. In the fourth stage, LaTTe performs fast register allocation, generating a CFG of real SPARC instructions. Finally, the CFG is converted into SPARC code, which is allocated in the heap area of the JVM and is made ready for execution[6].
The code of LaTTe is freely available with BSD-like license.
The new technology called Jazelle DBX (Direct Bytecode eXecution) developed by ARM was announced in June 2005. This technology performs direct execution of Java bytecode in hardware that promises significant acceleration of execution of Java bytecode in application written in Java.

Efficient object code.

Java allows to write object code very efficiently. As already mentioned above Java is object-oriented language from its origin. All logic of the code should be implemented in classes. Object Oriented Programming approach brings to writing of code some level of abstraction and objects created by programmer can be associated with objects from real life that simplifies understanding of program structure and re-using of the code. From this point of view Java is more efficient than for example C++ (or PHP) that is able to combine both approaches - object and non-object oriented. This combination of approaches may disorient a programmer using legacy code.
From other side Java is not more efficient than another “pure” OOP language C#, that was launched by Microsoft taking into account both positive and negative experiences of Java.

Orthogonality.

Dave Thomas, one of the authors of "The Pragmatic Programmer: From Journeyman to Master (Addison-Wesley, 1999)" in interview: "If you have a truly orthogonal system, unrelated elements are expressed independently. Here you have a business-level change: one

[6] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.153.4767&rep=rep1&type=pdf

thing changes at the business level and one thing changes in the system. If the boss says, I want negative numbers red, you change one thing and suddenly all the negative numbers in the system are red."
According to this definition Java can be named as orthogonal programming language from it's origin: it's hasn't functions outside Java classes so any implementation of Java components can be designed with certain level of isolation. The minimum unit that can be compiled in Java is a class.
Java supports encapsulation initially. As result of that, some partial realization of class methods can be hide from other side that can interact only with visible outer interface. The encapsulation is the guarantee that the most delicate parts of implementation mechanism can be protected from non-skilled user.
In addition, such feature of Java as polymorphism can be considered as having nature of orthogonality. By definition, polymorphism is the provision of a single interface to entities of different types. It means that different blocks of program code can be called by one operation. Also it means methods overloading that can modify implementation of class method, so that code can be re-used and this one of the features of orthogonality.
Object-Oriented Programming, if it's being used with certain experience and certain design patterns, is a good basement in deal of creation orthogonal systems. Once created every class can be reused latter. Java was the first original OOP language, that even cannot be implemented without OOP principles, fully brings these principles into the life.

Machine independence.

The idea of Java itself was to get language with so few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another. For example, type int always means 32-bits integer in Java. In C or C++ it can be 16-bits or 32-bits or even some other integer value (depending on compiler), the only limitation is integer cannot be greater than long 64-bits integer and shorter than 16-bits integer. Java has fixed size of numeral values and this helps to run Java program on different platforms. Binary data handling with certain format so big/little endian problem can be skipped performing same code on different platforms.

Also using of Unicode from the very beginning of Java is great advantage in deal to make Java environment machine independent.

A programmer writing in Java is focused on aims of the code he or she writes, the system were the program will be run is another level of abstraction. The bytecode will be translated to native code by JIT, like mentioned above HotSpot or LaTTe, for this particular platform.

The AWT (Abstract Windowing Toolkit) is Java original platform-independent API for creation of GUI. AWT is part of JFC (Java Foundation Classes). The main idea of implementation of AWT was to create native-looking interface on different platforms. It means Java program running on Windows platform if it was written with AWT looks like Windows program, the same code runs on Linux GNOME looks like GTK+ GUI program and so on.

In 1998 with J2SE 1.2 was introduced new library for GUI creation with name Swing. Today Swing is primary toolkit for Java GUI widgets. The Swing uses another approach for painting of graphical components. If AWT component delegates painting to OS-native widgets, Swing component renders by itself. Components of Swing library have their own platform-independent, but these components are adjustable so there is opportunity to put them look like native OS GUI components. As already mentioned in Java SE 6 Swing library was improved in OpenGL and DirectX interoperability issues, performance was significantly increased.

However, everything is good in a theory, in a practice a difference how programs behave on different platforms exists. There is no 100% warranty that Java program runs on all different systems in exactly same way. Some developers complain that their software works properly on one platform does not work another one. It may be caused by availability of a lot of different OS, Java and JVM versions.