CHAPTER 2

JAVA AND THE JAVA VIRTUAL MACHINE

2.1 The Java Programming Language

A new object-oriented programming language called Java was introduced by Sun Microsystems in 1995 which was close to C/C++ [28]. Its features of portability, robustness, simplicity and security [39] have made it increasingly popular within the software community. Java combines a wide range of language features found in different programming languages, for example, an object-oriented model, multithreading, exception handling and automatic garbage collection. A disadvantage is that these features come at the expense of a decrease in performance [53].

In recent years the Java programming language has integrated itself into our every day lives [56]. Java technology is involved in the creation of devices such as televisions, VCR's, audio components, fax machines, scanners, printers, cell phones, smart personal digital assistants, pagers, keys to homes and cars, watches or smart cards. Figure 2.1 shows the wide range of applications of Java technology [58].

At the heart of Java technology lies the Java Virtual Machine (JVM). This is the abstract computer on which all Java programs run and with which Java *class* files, Java's Application Program Interface (API), and the Java language work together to make the Java phenomena possible [41]. The ability to implement the JVM on various platforms is what makes Java portable.

In traditional compiled languages, such as C/C++, Pascal or Fortran, the source code is directly complied to the instruction set of the target Operating System's Central Processing Unit (OS-CPU). In contrast Java's complied code, known as bytecode, is platform independent [5]. The JVM is the interface between compiled Java programs and any target hardware platform. It is also the component of the Java technology responsible for the small size of its compiled code, and Java's ability to protect users from malicious programs [57].

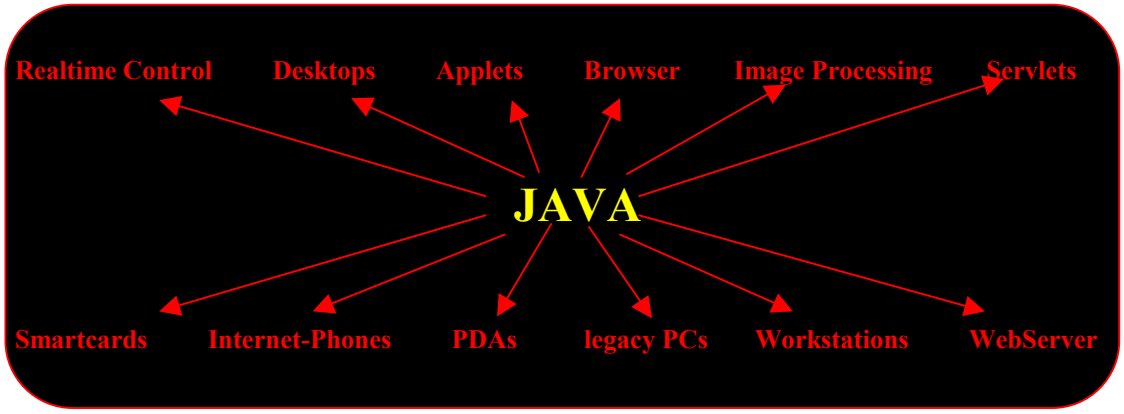

Figure 2.1: Uses of the Java Programming Language.

With respect to runtime services, Java is much more a "Middleware Operating System" than just a language. It includes enough functionality to act as a "standalone" Operating System (native libraries, optional platform Operating System and Hardware are completely hidden). It moves the boundary of platform specific code out of the applications.

2.1.1 The Execution of a Java Program

A Java program is a collection of class definitions written in the Java language [3]. A Java compiler then complies or translates this into a collection of bytes that are represented in a form known as the *class* file format. This is a platform independent intermediate representation of the program. As an alternative to being stored in a file these bytes can also be kept in a database, across a network or as part of a Java Archive File (JAR).

The *class* file contains bytecode instructions for the JVM [32]. This file will have the same semantics as the original Java source code. The JVM will execute these instructions to produce executable code. Figure 2.2 illustrates the steps involved in the execution of a Java program. As long as the original semantics of the instructions are obeyed, the JVM is free to perform the actions specified by the bytecode in any way it sees fit. There are a number of implementation techniques a JVM has to choose from and the memory layout is also determined by the JVM implementation. The bytecode instructions can be interpreted or alternatively the JVM can translate them into native machine code.

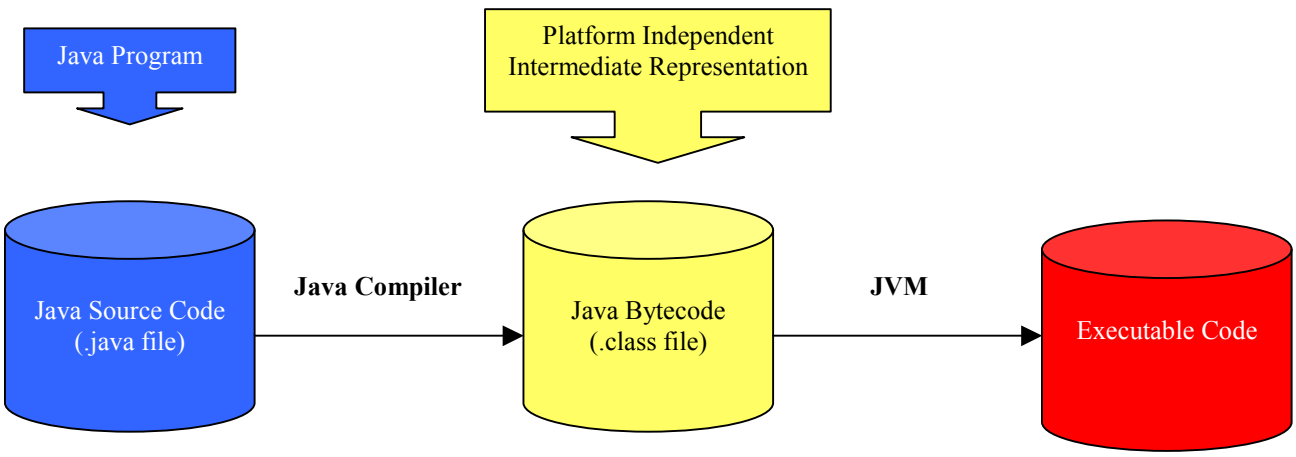

Figure 2.2: Steps in the execution of a Java program.

2.2 The Java Virtual Machine Specification

The Java Virtual Machine Specification, by Tim Lindholm and Frank Yellin, outlines what determines a JVM [57]. The specification defines three components:

Firstly, the JVM must accept bytecode instructions. Bytecodes define the instruction set that executes the user's application program. Each of the JVM's stack-based instructions consist of a one-byte opcode immediately followed by zero or more operands. The JVM's instruction set defines 200 standard opcodes, 25 quick variations of some opcodes (to support dynamic binding), and three reserved opcodes. The opcodes dictate to the JVM what action to perform. [53]

Secondly, a binary format known as the *class* file format, must be used to convey bytecodes and related class infrastructure in a platform- independent manner. Static tools such as *class* file viewers can look at this representation.

Finally, a *verification* algorithm must be used for identifying programs that will not compromise the integrity of the JVM. A verifier is a key component of the Java security architecture. It will check that the code respects the syntax of the bytecode language and that it respects the language rules [29].

2.3 The Components of a Java Virtual Machine

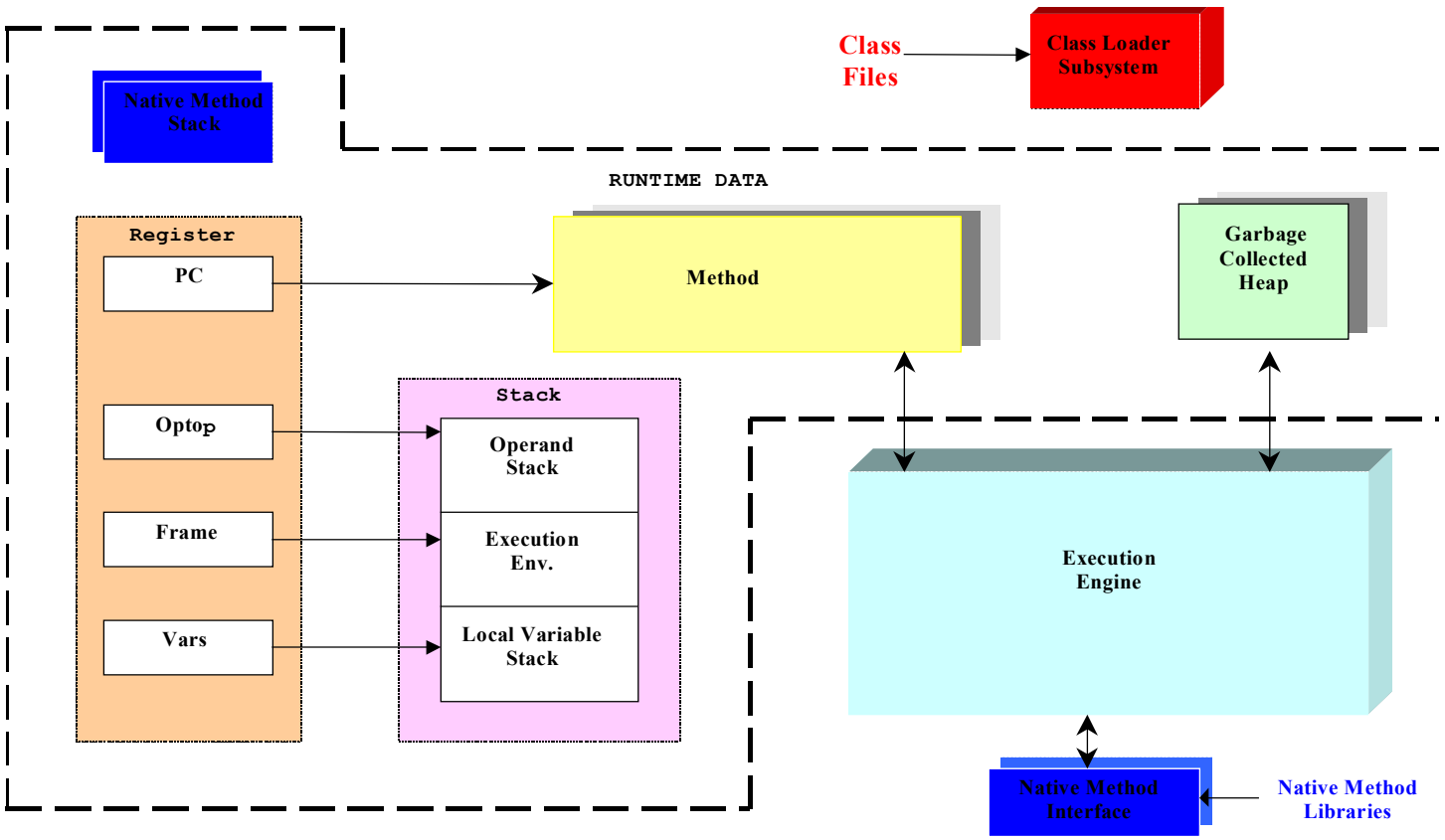

Figure 2.3: The Components of a JVM

The basic components of a JVM are illustrated by Figure 2.3. The ***execution engine*** is the core of the JVM. The behaviour of the *execution engine* is defined in terms of an instruction set. The JVM specification defines what the implementation should do when it is given an instruction as it executes bytecode.

A variety of execution techniques have been proposed to decrease the execution time of Java programs. The 'virtual processor' can be implemented as an interpreter, a compiler or a Java-specific processor [60]. Interpreters and compilers are software implementations of the JVM. A Java specific processor is a hardware implementation.

The *class area* stores classes that are loaded into the system by the *class loader*. The class definitions serve as templates for objects. The properties

of classes are immutable, i.e. it they cannot be changed once they have been brought into the system. This adds to the stability of the JVM.

Classes have a number of properties which are:

- Superclass
- List of interfaces : This can be empty.
- List of fields, of which there are two types :

  Static fields – A single copy exists for the entire class.

  Non-static fields – There is a copy of the field in each object.
- List of methods and implementations
- List of constants

The ***class loader*** architecture plays an important role in security and network mobility. There can be more than one inside a JVM. Two types of *class loader* exist:

- Primordial *class loader*: There is only one of there inside a JVM. It is responsible for loading trusted classes including API classes. It is an intrinsic part of the JVM
- *Class loader* objects: At runtime a Java application can install class loader objects that load classes in custom ways, for example across a network. These load untrustworthy classes and are not an intrinsic part of the JVM. They are written in Java, converted to *class* files and loaded in to the JVM and instantiated like another object.

For each class that the *class loader* loads the JVM keeps track of which *class loader* loaded it. It will load any classes that this class requires through the same *class loader*.

The heart of the JVM's ability to load *class* files dynamically is the java.lang.ClassLoader [32].

Class loading is a two-stage process:

1. **Loading**: The name of a class is used to locate a bunch of bytes that it wants to load as a class and these are given to the JVM as the implementation of the class. The superclass of this file is also loaded and the superclass of the superclass etc. Therefore after loading the JVM will know the name of the class, where it occurs in the class hierarchy and what fields and methods it has.

2. **Linking:** Consists of
    - Verification: Extensive checks are done to ensure that the *class* file is valid, this is a vital part of the JVM security model. This is needed because of the possibility of a buggy compiler or no compiler at all, malicious intent or (class) version skew. The checks done here are independent of compiler and language.
    - Preparation: Static fields for a class are created and these are set to standard default values (but they are not explicitly initialised). Method tables are constructed for the class.
    - Resolution: Most classes refer to methods/fields from other classes, resolution translates these names into explicit references. It also checks for field/method existence and whether access is allowed.

Class initialisation then takes place. This happens just once before the first instance is created or there static variable is first used. The superclass is initialised first. Static (class) initialiser code is executed. Explicit instances for static variables are executed. However this may not need to happen if static variable is declared final.

This all takes place before anything else sees the class.

Two kinds of Java methods exist. There are those methods that are written in Java and complied to bytecode. These are stored in a *class* file and exhibit platform independence.

The code for methods is kept in the *method area.* This area is shared by all threads.

Native methods are written in another language e.g. C, C++, assembly language. They are used to allow a programmer to handle situations that cannot be handled fully in Java, e.g. interfacing to platform-dependent features or integrating legacy code written in another language. They are complied to native machine code of a particular processor and stored in dynamically linked libraries whose exact form is platform specific. When a Java program calls a native method the virtual machine loads the dynamic library that contains the method and invokes it. These provide a connection between the Java program and the host operating system [65].

When native methods are complied into machine code, they usually use a stack to keep track of their state. The *native method stacks* are provided by the JVM for this purpose. These are sometimes called C stacks, as the most common way to implement native methods is to write them in C and compile them into native code [32].

Java Native Interface (JNI) enables native methods to work with any Java Platform implementation on host, as native methods do not exist on all JVM implementations [65].

The JVM has four registers. The *Program Counter* (PC) register points into the method area. It is a pointer to the currently executing instruction if the method is not native. If the method is native the value of the PC register is undefined. Each JVM thread has its own program counter, this way the JVM can support many threads of execution at once [59]. The three other registers are used to manage the stack. The *Optop* register points to the top most cell of the operand stack and the *Frame* register points to the first cell of the execution environment. All local variables are addressed relative to the *Vars* register.

The Java ***stack*** is a stack of *stack frames* [32]. A new *stack frame* is created every time a method is invoked. Each *stack frame* consists of an operand *stack* and an array of local variables.

The *stack frame* on top of the *stack* is called the active *stack frame*; this is the current place of execution. Only the operand *stack* and the local variable array in the active *stack frame* can be used.

Each time a new method is invoked a new *stack frame* is created and pushed onto the stack, this then becomes the active *stack frame*. The PC is saved as part of the old Java *stack frame*. The PC in the new Java *stack frame* will point to the beginning of the method.

When a method returns, the active *stack frame* is popped from the *stack* and the one below becomes the active *stack frame*. The PC is set to the instruction after the method call, and the method continues.

Objects are stored in the *heap*. Whenever a class instance or an array is created in a running Java application, the memory for the object is allocated from a single *heap*.

Each object is associated with a class in the *method area*. There are also a number of slots for storing fields associated with each object; for example there is one slot for each nonstatic field in the class and one for each nonstatic field in the superclass etc.

Only one *heap* exists inside a JVM instance, therefore all threads share it. As a Java application runs inside its 'own' exclusive JVM instance, a separate *heap* exists for every individual running application. This means that two different Java applications cannot trample on each other's *heap* data. However two different threads of the same application can, therefore it is important to have the proper synchronisation of multithreaded access to objects.

The JVM includes an instruction that allocates memory on the *heap* for new objects but no instruction for freeing that memory. The JVM overcomes this by implementing a *Garbage Collector* to manage the *heap* .

2.3.1 Garbage Collection

Each object that is created consumes memory of which there is a limited amount. The function of the *garbage collector* is to free memory when objects are no longer in use. An object is ready to be garbage collected when it is no longer "alive", that is it is dead.

An object is deemed to be alive when:

- There is a reference to the object on the stack
- There is a reference to the object in a local variable, on the stack, or in a static field
- A field of an alive object contains a reference to the object.
- The JVM has an internal reference to the object for supporting native methods for example.

A large variety of garbage collection algorithms have been developed, for example reference counting, mark-sweep, mark-compact, copying and noncopying implicit collection.

However garbage collection imposes a time penalty on the user program, it is therefore important that it is efficient and interferes with program execution as little as possible. Therefore when selecting a garbage collection technique for JVM implementation there has to be a trade off between ease of implementation versus execution time performance [65].

As well as garbage collection freeing the programmer from the burden of explicitly reasoning about the use of memory, it also eliminates two classes of errors:

- *memory leaks* – these are errors where the application loses track of allocated memory without freeing it.
- *dangling references* – these are where the application frees memory while retaining references to it and subsequently accesses this memory through these references [1].

Garbage collection therefore leads to improvements in: safety, accuracy, simplicity, modularity and burden [4].

2.4 Performance Issues in Java

All the features that make Java desirable as a programming language, for example its platform independence, impose a significant penalty on its performance.

Java has been shown to be 10-50 times slower than an equivalent compiled C program in the standard interpreted mode of execution [53].

The main reason behind the slow performance of Java is the high degree of hardware abstraction it offers. Java source code is compiled to bytecode, that is, a platform-independent intermediate represent of the program. This makes Java programs portable across all hardware platforms as these bytecodes are generated with no knowledge of the native CPU on which the code will be executed. Because of this at run-time some translation or interpretation is needed, and this will directly add to the program's execution time.

Automatic garbage collection is an important memory management feature of the Java language. It relieves the programmer from having to deallocate memory when it is no longer needed. However this process adds to the overall execution time of a Java program. Additional overhead is added by such features as exception handling, multithreading and dynamic loading.

For Java to become accepted as a general purpose programming language, its performance must become comparable to that of languages such as C and C++. While it is currently possible to achieve a similar level of performance with Java programs through the use of various techniques such as, for example, direct Java compilers. Currently this may be achieved at the possible expense of the portability and flexibility of the Java programs, which are the main advantages of using Java as a programming language in the first place.

2.5 Conclusion

The Java programming language is relatively new, however each day sees its use being extended into a greater number of areas. As it puts an increased emphasis on reliability it is important to have methods of evaluating the quality of Java programs. Therefore the remainder of this thesis is devoted to developing a number of dynamic metrics to aid in the evaluation of the quality of Java programs.