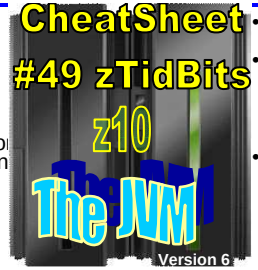
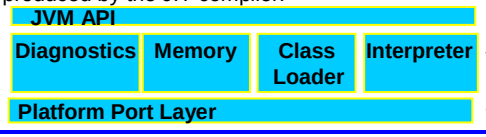


•The Java Virtual machine (JVM) is the application that executes a Java program and it is included in the Java package.
•The JVM is called "*virtual*" because it provides a machine interface that does not depend on the underlying operating system and machine hardware architecture.
•This independence from hardware and operating system is a cornerstone of the write-once run-anywhere value of Java programs. Java programs are compiled into "bytecodes" that target the abstract virtual machine where the JVM is responsible for executing the bytecodes on the specific operating system and hardware combination.
•The installable Java package supplied by IBM comes in two versions:
 -The Java Runtime Environment (JRE)
 - The Java Software Development Kit (SDK)
•The JRE provides runtime support for Java applications.
 -The SDK provides the Java compiler and other development tools.
 - The SDK includes the JRE.
  >The JRE (and, therefore, the SDK) includes a JVM.
  >This is the application that executes a Java program.
• A Java program requires a JVM to run on a particular platform, such as z/OS, Linux or even Windows ☹ .
•The IBM SDK Version 5.0 began a different implementation of the JVM and the Just-In-Time compiler (JIT) than its earlier releases apart from the version 1.4.2 implementation on z/OS 64-bit.
•The JVM specification also defines several other runtime characteristics where all JVMs:
 - Execute code that is defined by a standard known as the class file format
 - Provide fundamental runtime security such as bytecode verification
 - Provide intrinsic operations such as performing arithmetic and allocating new objects
•A Java application uses the Java class libraries that are provided by the JRE to implement the application-specific logic. The class libraries, in turn, are implemented in terms of other class libraries and, eventually, in terms of primitive native operations that are provided directly by the JVM.
•The JVM API encapsulates all the interaction between external programs and the JVM. Examples of this interaction include:
 - Creation and initialization of the JVM through the invocation APIs.
 - Interaction with the standard Java launchers, including handling command-line directives.
 - Presentation of public JVM APIs such as JNI and JVMTI.
 - Presentation and implementation of private JVM APIs used by core Java classes.
•**The diagnostics component** provides Reliability, Availability, and Serviceability (RAS) facilities to the JVM.
 - The IBM Virtual Machine for Java is distinguished by its extensive RAS capabilities.
 - The IBM Virtual Machine for Java is designed to be deployed in business-critical operations and includes several trace and debug utilities to assist with problem determination.
 - If a problem occurs in the field, it is possible to use the capabilities of the diagnostics component to trace the runtime function of the JVM and help to identify the cause of the problem.
 -The diagnostics component can produce output selectively from various parts of the JVM and JIT.
•**The memory management** component is responsible for the efficient use of system memory by a Java application.
 - Java programs run in a managed execution environment and when a Java program requires storage, the memory management component allocates the application a discrete region of unused memory (HEAP).
 - After the application no longer refers to the storage, the memory management component must recognize that the storage is unused and reclaim the memory for subsequent reuse by the application or return it to the operating system.
•**The class loader** component is responsible for supporting Java's dynamic code loading facilities. The dynamic code loading facilities include:
 - Reading standard Java .class files.
 - Resolving class definitions in the context of the current runtime environment.
 - Verifying the bytecodes defined by the class file to determine whether bytecodes are language-legal.
 - Initializing the class definition after it is accepted into the managed runtime environment.
 - Various reflection APIs for introspection on the class and its defined members.
•**The interpreter** is the implementation of the stack-based bytecode machine that is defined in the JVM specification and each bytecode affects the state of the machine and, as a whole, the bytecodes define the logic of the application.
 - The interpreter executes bytecodes on the operand stack, calls native functions, contains and defines the interface to the JIT compiler, and provides support for intrinsic operations such as arithmetic and the creation of new instances of Java classes.
 - The interpreter is designed to execute bytecodes very efficiently.
 - It can switch between running bytecodes and handing control to the platform-specific machine-code produced by the JIT compiler.



This diagram shows the component structure of the IBM Virtual Machine for Java.

•The ability to reuse the code for the JVM for numerous operating systems and processor architectures is made possible by the platform port layer.
•The platform port layer is an abstraction of the native platform functions that are required by the JVM.
 - Other components of the JVM are written in terms of the platform-neutral platform port layer functions. Further porting the JVM requires the provision of implementations of the platform port layer facilities.
• The Just-In-Time (JIT) compiler is a component of the Java Runtime Environment that improves the performance of Java applications at run time.
 - Java programs consists of classes, which contain platform-neutral bytecodes that can be interpreted by a JVM on many different computer architectures.
 - At run time, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation.
  NOTE1:The overhead of interpretation means that a Java application performs more slowly than a native application.
 - The JIT compiler helps improve the performance of Java programs by compiling bytecodes into native machine code at run time.
 - The JIT compiler is enabled by default, and is activated when a Java is called.
 - The JIT compiler compiles the bytecodes of that method into native machine code, compiling it "just in time" to run.
 - When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it.
 - Theoretically, if compilation did not require processor time and memory usage, compiling every method would allow the speed of the Java program to approach that of a native application.
  NOTE2: JIT compilation does require processor time and memory usage. When the JVM first starts up, thousands of methods are called. Compiling all of these methods can significantly affect startup time, even if the program eventually achieves very good peak performance.
  ⚠ In practice, methods are not compiled the first time they are called. For each method, the JVM maintains a call count, which is incremented every time the method is called. The JVM interprets a method until its call count exceeds a JIT compilation threshold. Therefore, often-used methods are compiled soon after the JVM has started, and less-used methods are compiled much later, or not at all. The JIT compilation threshold helps the JVM start quickly and still have improved performance. The threshold has been carefully selected to obtain an optimal balance between startup time and long term performance.

New in SDK 6

•**Ahead-Of-Time (AOT)** compilation allows the compilation of Java classes into native code for subsequent executions of the same program.
 - The AOT compiler works in conjunction with the class data sharing framework.
 - The AOT compiler generates native code dynamically while an application runs and caches any generated AOT code in the shared data cache.
 - Subsequent JVMs that wish to execute the method can load and use the AOT code from the shared data cache without incurring the compilation overhead generally experienced with JIT-compiled native code.
•The AOT compiler is *enabled by default*, but is only active when shared classes are enabled. By default, shared classes are disabled so no AOT activity will occur.
 - When the AOT compiler is active, the compiler heuristically selects the methods to be AOT compiled with the primary goal of improving startup time.
  NOTE3: Because AOT code must persist across different program executions, AOT-generated code does not perform as well as JIT-generated code, although it usually performs better than interpreted code.
• In a JVM without an AOT compiler or with the AOT compiler disabled, the JIT compiler selectively compiles frequently executed methods into optimized native code.
 - There is a time cost associated with compiling methods because the JIT compiler operates while the application is running. Because methods begin by being interpreted and most JIT compilations occur during startup, startup times can be increased.
•Startup performance can be improved by using the shared AOT data to provide native code without compiling. There is a small time cost to load the AOT code for a method from the shared data cache and bind it into a running program, but this time cost is low compared to the time it takes the JIT compiler to compile that method.
•If you suspect that a problem is occurring in the JIT or AOT compiler, disable compilation to see if the problem remains. If the problem still occurs, you know that the compiler is not the cause of it.
•Garbage collection identifies and frees previously allocated storage that is no longer in use
 - Understanding the way the Garbage Collector works will help you to diagnose problems.
•The JVM includes a Memory Manager, which manages the Java heap.
 - The Memory Manager allocates space from the heap as objects are instantiated, keeping a record of where the remaining free space in the heap is located.
 -When free space in the heap is low and an object allocation cannot be satisfied, an allocation failure is triggered and a garbage collection cycle is started.
 - When this process is complete, the memory manager retries the allocation that it could not previously satisfy.