Programming in Java

1. Introducing Java

Evolution of Software Development

- How we write code has gone through a number of eras
  - Each era is characterized by a specific approach to “How We Write Code”
- These eras are the result of several factors:
  - **Business drivers** – What the consumers (who pay for software) want it to do for them
  - **Technology** – Limitations and capabilities of existing hardware and infrastructure (like networks)
  - **Tools and techniques** – The theoretical and practical engineering methodologies and toolsets
- Main Eras
  - **1940s to 1950s** – *Hardware Rules* – Machine code and instruction sets
  - **1950s to 1970s** – *Code Cowboys* – Coding by instinct “Real programmers only use assembler”
  - **1970s to 1990s** – *Software Engineering* – Adaptation of engineering practices to software
  - **1990s to 2010s** – *Human Cyber Systems* – Massively networked: eg. Internet and on-line users
  - **2010s to Present** – *Cyber Physical Systems* – Internet of Things, big data, massive scale

Programming Paradigms

- A programming paradigm is a set of:
  - Assumptions about what the components of a program should be
  - Techniques and principles used in designing programs
  - Assumptions about what sort of problems are solved best by that paradigm
  - Best practices for software design and code style
- There are multiple programming paradigms
  - Most have been around since the start of high level programming in the 1960s
- A paradigm becomes “main-stream” when a set of problems arise that the paradigm is better suited to solve than the paradigm currently in use
- The main-stream paradigms in use today – and supported in Java are:
  - Structured or procedural programming
  - Object Oriented Programming
  - Functional Programming

Structured Programming

- Code is structured into reusable modules
  - Called subroutines or functions or procedures
  - Standard libraries of procedure are part of the programming language
- Users can define their own procedures and libraries
  - Procedures are the highest level of organization
  - Implementation of DRY
- Many Java class libraries are organized like this.
- Image is a FORTAN subroutine

```
C ***************************************************************
C
      SUBROUTINE TRISOL(A,B,C,D,H,N)
C
C ************** TRI-DIAGONAL MATRIX SOLVER ************************
C
C *** THIS TRIDIAGONAL MATRIX SOLVER USES THE THOMAS ALGORITHM ********
C
      dimension A(250),B(250),C(250),D(250),H(250),W(250),R(250),G(250)
      W(1)=A(1)
      G(1)=D(1)/W(1)
      do 100 I=2,N
      I1=I-1
      R(I1)=B(I1)/W(I1)
      W(I)=A(I)-C(I)*R(I1)
      G(I)=(D(I)-C(I)*G(I1))/W(I)
  100 continue
      H(N)=G(N)
      N1=N-1
      do 200 I=1,N1
      II=N-I
      H(II)=G(II)-R(II)*H(II+1)
  200 continue
      return
      end
```

OO Programming

- Procedures are methods within class definitions
  - There are no stand-alone procedures
  - Java methods tend to be where the imperative style of coding is mostly found
- Some Java classes allow for static methods
  - The class is used to define an API that works just like a library in a structured programming language
- OO programming is about where we write and store our reusable code
  - Specifically, in class definitions
  - Image is Simula67 code from the mid 1960s
  - From the official Simula reference material

```
Begin
   Class Glyph;
      Virtual: Procedure print Is Procedure print;;
   Begin
   End;

   Glyph Class Char (c);
      Character c;
   Begin
      Procedure print;
        OutChar(c);
   End;

   Glyph Class Line (elements);
      Ref (Glyph) Array elements;
   Begin
      Procedure print;
      Begin
         Integer i;
         For i:= 1 Step 1 Until UpperBound (elements, 1) Do
            elements (i).print;
         OutImage;
      End;
   End;

   Ref (Glyph) rg;
   Ref (Glyph) Array rgs (1 : 4);

   ! Main program;
   rgs (1):- New Char ('A');
   rgs (2):- New Char ('b');
   rgs (3):- New Char ('b');
   rgs (4):- New Char ('a');
   rg:- New Line (rgs);
   rg.print;
End;
```

Functional Programming

- Procedures are defined like mathematical functions
  - Functions act like data
  - Said to be first class citizens
- Programs are treated like function composition in math
  - LISP introduced functional programming in the 1950s – before FORTRAN
  - Top listing shown is 1960s APL code for computing a matrix determinant
  - The code below it assigns a function to a variable Avg that averages a vector of numbers
  - You can try APL interactively at
    - https://tryapl.org/

```
      ∇DET[□]∇
     ∇ Z←DET A;B;P;I
[1]    I←□IO
[2]    Z←1
[3]   L:P←(|A[;I])⍳⌈/|A[;I]
[4]    →(P=I)/LL
[5]    A[I,P;]←A[P,I;]
[6]    Z←-Z
[7]   LL:Z←Z×B←A[I;I]
[8]    →(0 1 ∨.=Z,1↑⍴A)/0
[9]    A←1 1 ↓A-(A[;I]÷B)∘.×A[I;]
[10]   →L
[11]  ⍝EVALUATES A DETERMINANT
     ∇
```

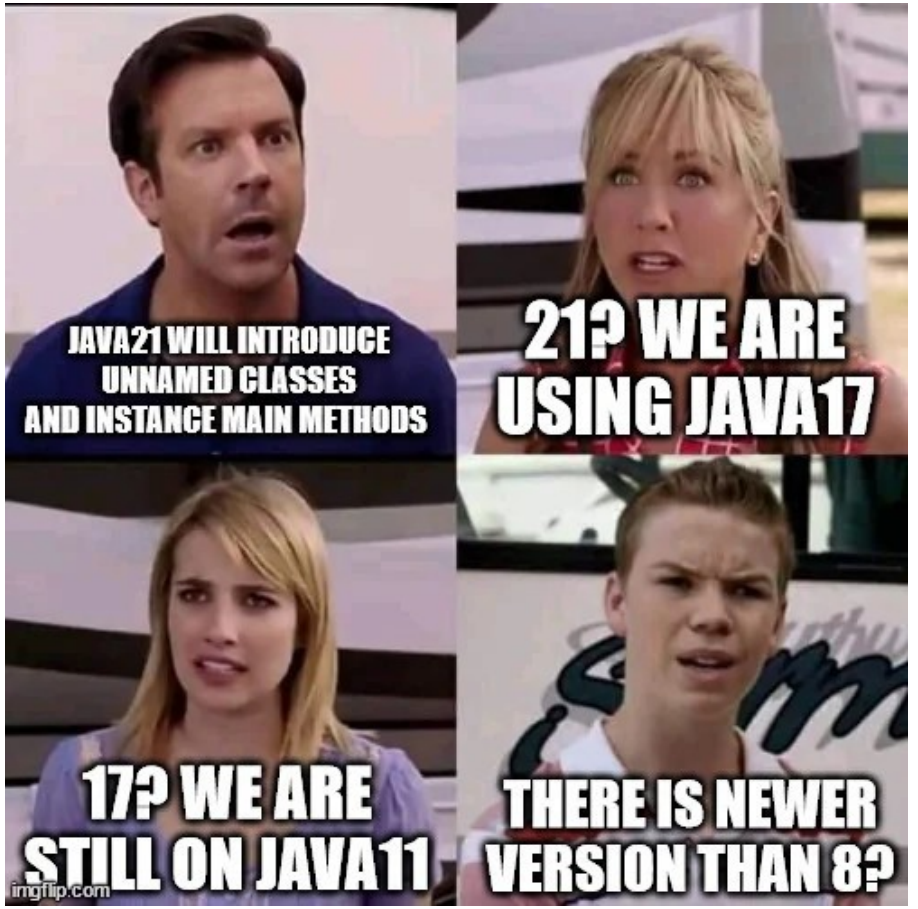
```
      Avg←{(+⌿⍵)÷≢⍵}
      Avg 1 6 3 4
3.5
      |
```

Paradigm Focus

- Structured programming
  - Developed to respond to the need to automate the business processes in the 1960s
  - COBOL = Common Business Oriented Language, FORTRAN and ALOGOL
  - A data set was read in, algorithms used to process it, then the results written out
  - Typically done in batch mode on a mainframe
  - There is still a massive installed based of COBOL and other structured code running business operations in the public and private sector
- Object Oriented programming
  - Structured programming doesn't do distributed computing and networks well
  - The rise of the Internet in the 1990s made this a requirements
  - OO languages, led by Java, had the right paradigm to do this sort of computing
- Functional programming
  - Ideal tool to handle streaming data at scale which became a need in the mid 2010s with the rise of big data
  - OO and structured programming don't do this well

Re-inventing Java

- As new paradigms become mainstream Java incorporates features from those paradigms
  - What Java code looks like has changed over time as the language evolves
- Java programmers often have to both
  - Write new Java code for modern architectures like microservices
  - Support legacy Java code written in earlier versions of Java

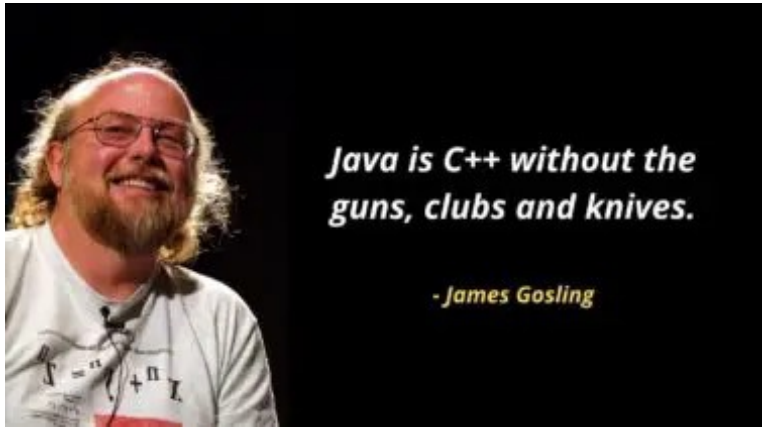


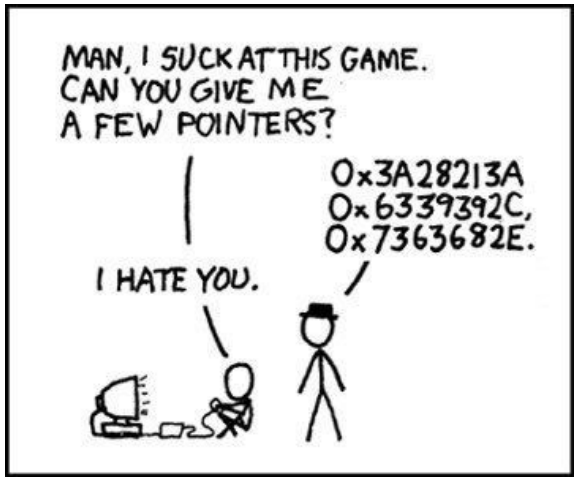
The Java Story

- Java was a project developed at SUN Microsystems to support interactive TV
  - Hardware wasn't powerful enough to do the required processing
  - Repurposed to do client side computing in browsers on the web
  - Up until that time, all computing took place on the servers
  - This created bandwidth issues and performance bottlenecks
- Java would run code called "Applets" in a special browser plugin or sandbox
  - Much like JavaScript does now
  - Principle designer was Canadian software engineer James Gosling

The Language Design

- 1990s – C++ and OO were very popular
- Gosling intended Java to be a better C++
  - Easy for C++ programmers to learn and adopt
- Threw away the non-OO features of C++
- “Fixed” the two most problematic features of C++
  - Direct access to memory pointers
  - Memory leaks
- Wanted portability
  - Write once, run anywhere
  - Designed around a fan-out VM architecture that had proven to be effective in other languages
- More rigorous error checking
  - For example, array bounds checking





Java Social Engineering

- Very concerned about keeping a Java Standard
  - *'My impression is that a really, really high-order concern for the whole development community is interoperability and consistency."* James Gosling
  - Did not want to open source the language but still make it free to use to speed adoption by programmers
  - Did not want to be in the business of "building" Java compilers
- Published two specifications
  - The Java Language Specification that describes how Java works,
    - Like other standard language specifications (C++ ISO standard for example)
  - The Java Virtual Machine Specification that describes how the runtime environment (JRE) works
    - JRE included the Java Virtual Machine and local native libraries
- SUN protected their IP using very restrictive trademarking
  - Anyone could write and distribute their own Java development tools (JDK) and JRE
  - Provided it passed the Java Compliance Suit tests
  - If your product didn't, you were not allowed to call it Java. Period. Or SUN would come after you

Fool Around and Find Out

- Microsoft developed a version of Java called J++ that was not compliant to the Java spec
  - SUN sued to enforce their trademark
  - SUN won and Microsoft had to drop J++
  - J++ later became the basis for .Net and C#

Sun, Microsoft settle Java suit

Sun Microsystems and Microsoft have settled their long-running lawsuit over Microsoft's use of Sun's Java software.

Stephen Shankland

March 15, 2002 5:10 a.m. PT

5 min read

Sun Microsystems and Microsoft have settled their long-running lawsuit over Microsoft's use of Sun's Java software.
Under the settlement, Microsoft will pay Sun $20 million and is permanently prohibited from using "Java compatible" trademarks on its products, according to Sun. Sun also gets to terminate the licensing agreement it signed with Microsoft.

Reference Implementations

- SUN also provided “reference implementations”
  - These were a JDK and JRE that demonstrated compliance to the specs and compliance test suite
  - These were not intended to be commercial products
- Many versions of JRE were developed for different platforms
  - IBM maintains versions of JRE to run on AIX, IBM Linux, z/OS and IBM i
- Java was originally free but proprietary
  - Eventually migrated into open source reference implementation
  - Java and the JVM are now open source
  - These are available at https://jdk.java.net/
  - These are supported by the openJDK project and Oracle
  - Also motivated by the fear that when Oracle bought SUN they might implement expensive licensing for using Java
- Oracle has opted more for a commercial support and redistribution plan for enterprises

The JCP – Java Community Process

- SUN created the Java Community Process (JCP)
  - Open committee that was the only authority to issue Java specifications
  - Would decide the direction Java would take based on input from the user community
- It resulted re-inventing Java as an enterprise application delivery platform instead of remaining a JavaScript like browser automation engine
  - Resulted in J2EE (EE) which introduced server side Java, servlets, Java Server Pages and a lot more
- Responsible for introducing new features like generics, functional programming and other innovations
  - Done in response to the requests to the JCP
  - Also responsible for deprecating Java features that are no longer useful
- From their website
    - *The JCP is the mechanism for developing standard technical specifications for Java technology. Anyone can register for the site and participate in reviewing and providing feedback for the Java Specification Requests (JSRs), and anyone can sign up to become a JCP Member and then participate on the Expert Group of a JSR or even submit their own JSR Proposals.*

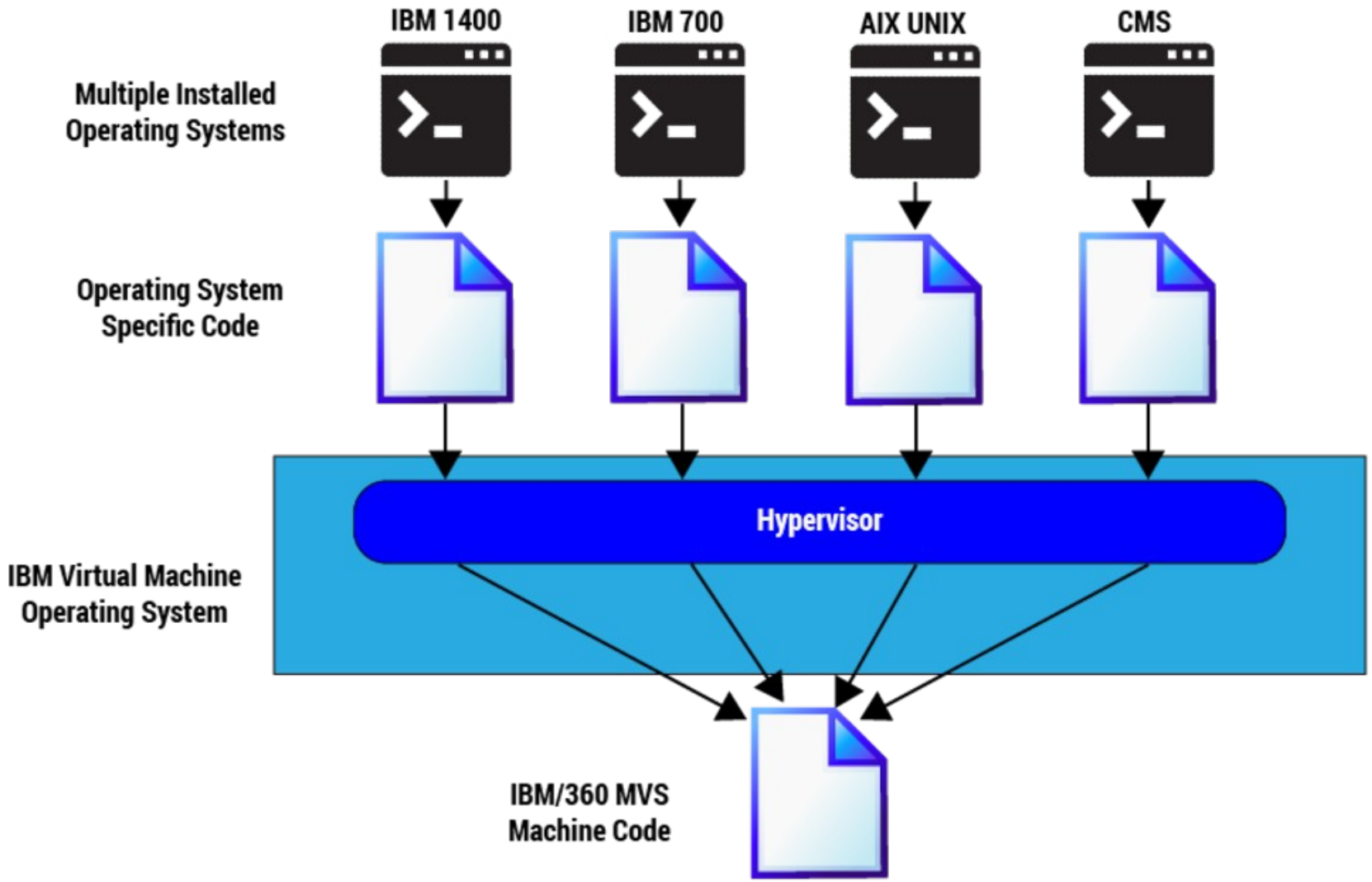
Virtual Machines

- Java runs in a Java Virtual Machine, or JVM
- VMs are ubiquitous in modern software architectures
  - The JVM is not like those VMs
- There are two basic types of architecture called virtual machines
- The first is what we might call fan-in architecture which is the most common type in use
  - VMWare, VirtualBox and cloud providers like AWS and Azure use this model
  - Characterized the use of a hypervisor
- The other is a fan-out architecture
  - It’s like a distributed hypervisor in reverse
  - It is implemented as a client (JVM) on a target system

Fan-in VMs

- Developed by IBM for the IBM/370 MVS OS in 1972
  - Called the Virtual Machine Operating System VM/OS
  - Coined the term hypervisor
- IBM had a massive installed base of customers
  - Used different versions of the IBM mainframes, eg. 700 series, 1400 series
  - IBM wanted to retire all those models and move clients to the new IBM/370
  - IBM only rented hardware to clients so there would be minimal hardware costs
- The problem
  - Each OS used software tightly coupled to the underlying hardware
  - Moving would entail massive software rewrites and so customer balked
- The VM/OS could emulate all the legacy systems
  - Customer software from a IBM 1400 would run in the VM/OS emulating a 1400

IBM VM/OS


IBM 1400
IBM 700
AIX UNIX
CMS
Multiple Installed Operating Systems
Operating System Specific Code
Hypervisor
IBM Virtual Machine Operating System
IBM/360 MVS Machine Code

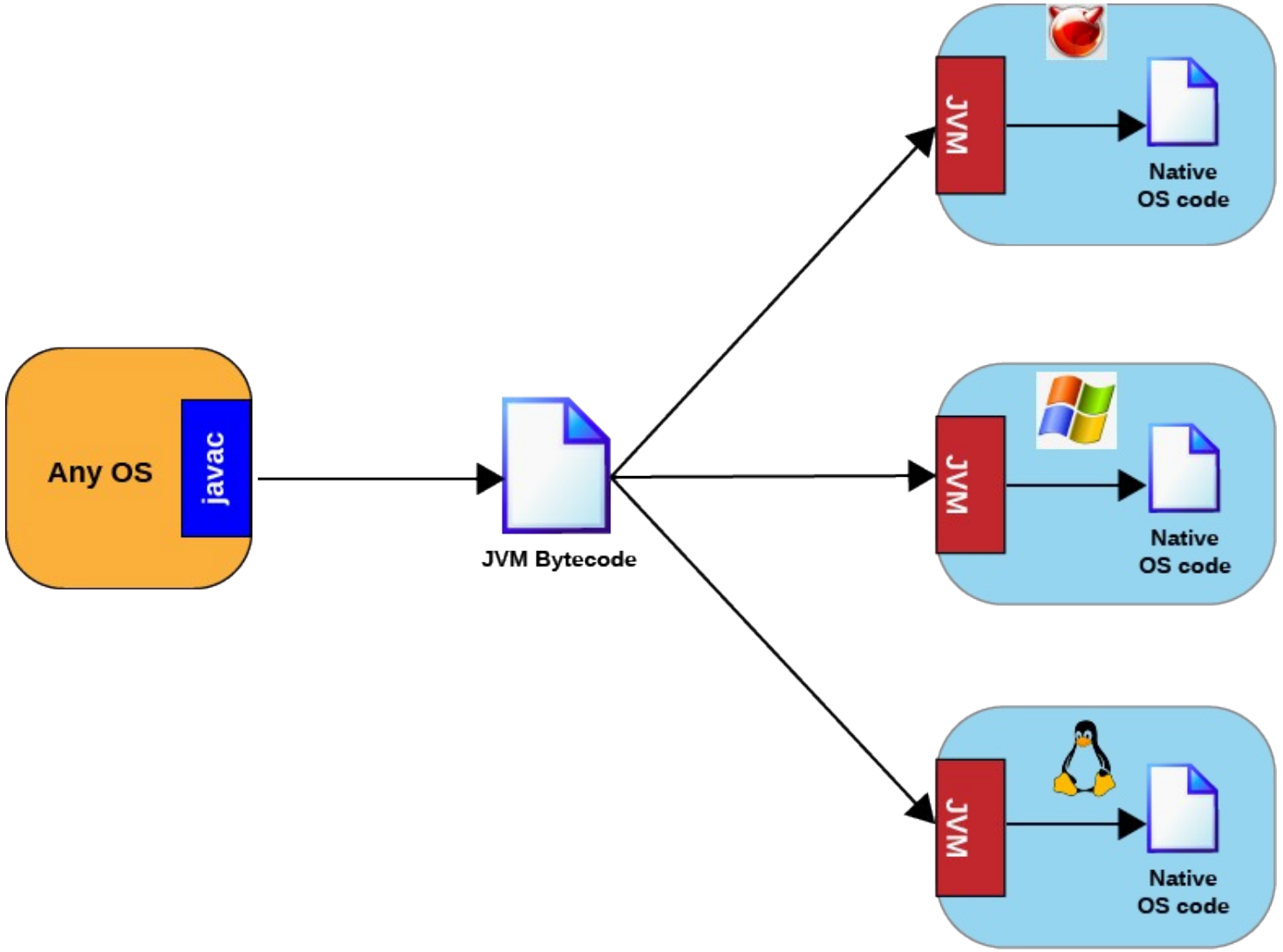
Time Traveling VMs

- John Titor was the name used by a person who claimed to be from the year 2036
  - His internet posts and interviews were very popular in 2000-2001
  - He claimed he traveled back in time to 1975 get an IBM 5100 portable computer
  - The computer ran APL and Basic
  - But it had an early version of a VM called an emulator that allowed it to be used to debug mainframes
  - He claimed this VM was needed to fix problem with the future legacy systems
  - Currently the Titor story is considered to be a very literary experiment fiction writing

Fan-out VMs

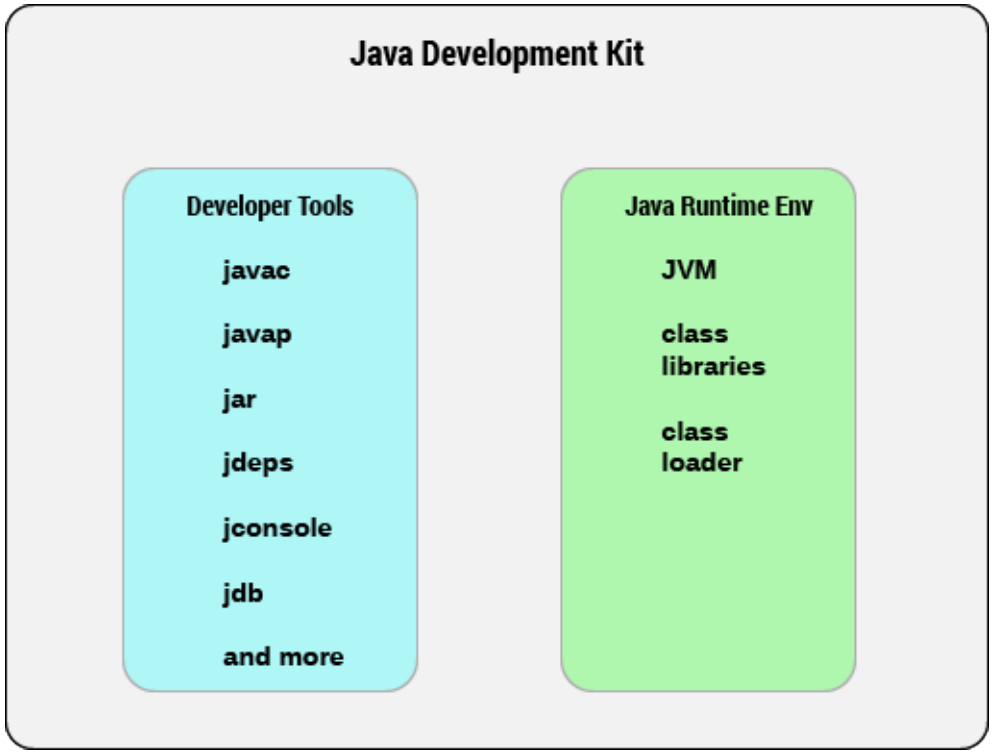
- Common write-once, run-anywhere model
- Application code is complied to a intermediate representation called bytecode
  - Sort of a generic assembler language
  - Makes it easy to translate into real assembler
- Client machines run the bytecode in a platform specific interpreter
  - The interpreter translates the bytecode into platform specific assembler
- Became very popular in the 1970s
  - Many early computer companies went belly up (like Wang laboratories)
  - Left users with masses of code that would run only on those machines
  - Solution was to create a virtual version of the hardware that would compile source code into bytecode
  - Then run the bytecode on a PC or a Mac
- Since the original intent of Java was to run on multiple platforms
  - This was a natural model at the time to use as the basis for a design

Java Architecture



The JRE

- The platform specific JRE consists of:
  - The JVM executable named *java*
  - Local copies of the Java Class libraries in platform specific code
    - A “platform” is the combination of hardware and operating systems
  - A class loader responsible for loading complied java classes into the JVM
- The JRE is all that is needed to run Java applications
- The Java Development Kit JDK
  - Contains a JRE
  - But also contains the java compiler and other development tools



Java Code Lifecycle

- Java source code in **.java* files is compiled (*hello.java*)
- The bytecode output is in a **.class* file (*hello.class*)
- The JVM starts
- The class loader loads the requested **.class files* into the JVM
  - This is the reason for the rule we will see later about public class name and file names
- If the source files are in a package structure
  - The compiler outputs a corresponding package structure for class files
  - We will explore packages later
- The problem is that this structure had to be sent somehow to a JRE
  - This could be very complicated
- The modern approach is to zip all the out put in to a Jar file
  - Just one thing to send to the JRE
  - Regular Jar files still need the local class libraries to run
  - Fat or Shaded Jar files, like the ones Spring produces, contain the needed libraries
  - Fat Jar files can run as standalone Java applications

Writing Java Code

- All Java code must be contained in a class definition.
  - These are the units of byte code that are loaded into the JVM
- The JVM needs to know where to start executing the code
  - It looks through all the class definition until if finds a class with a method called *main()*
  - Just like the *main()* function in C, C++ or Golang
  - If there is more than one main() method defined, or no main() method defined, a run time error occurs

Demo

Introduction to IDEs and Hello World

Lab 1-1

Hello World and Lab Setup

Running Java Code

- In the first lab, you let the IDE run the Java code for you
- Java is designed to be run at the command line on in some environment with out a GUI
- In this case, we execute ‘javac HelloWord.java’
  - This creates the compiled bytecode ‘HelloWorld.class’
- The class file can be run with the JVM with the command ‘java HelloWorld’

Demo

Working with bytecode


Lab 1-2
Grokking Bytecode


Java™