Introduction to Java

1. Modern Software Development and Java

Introduction

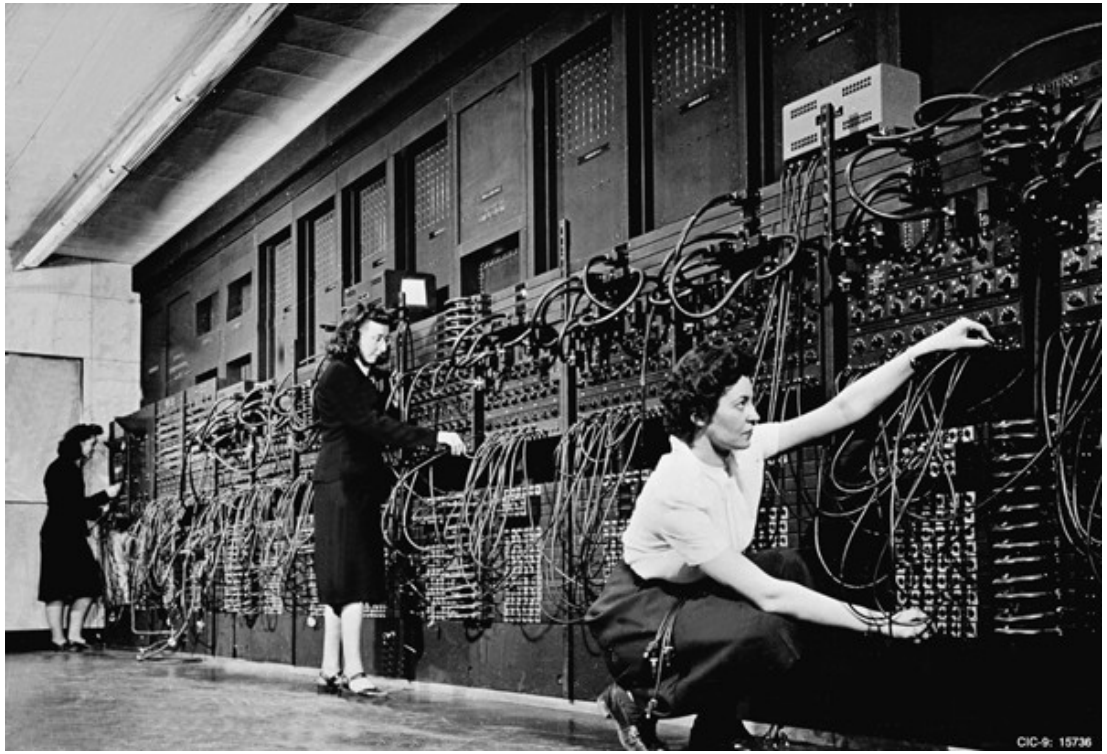
- This is one of the few theory heavy modules in the course
- This module covers some general topics like
  - The evolution of computing and different programming paradigms
  - A bit of Java history to explain why it wound up looking like it does today
- The main goal of this module is to set the context for the rest of the material

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 driver**s – The types of tasks that the consumers (who pay for software) want it to do for them
  - **Available Technology** – The 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 and high level language
  - **1990s to 2010s** – *Human Cyber Systems* – Massively networked systems, Internet and on-line users
  - **2010s to Present** – *Cyber Physical Systems* – Internet of Things, big data, massive scale architectures, AI

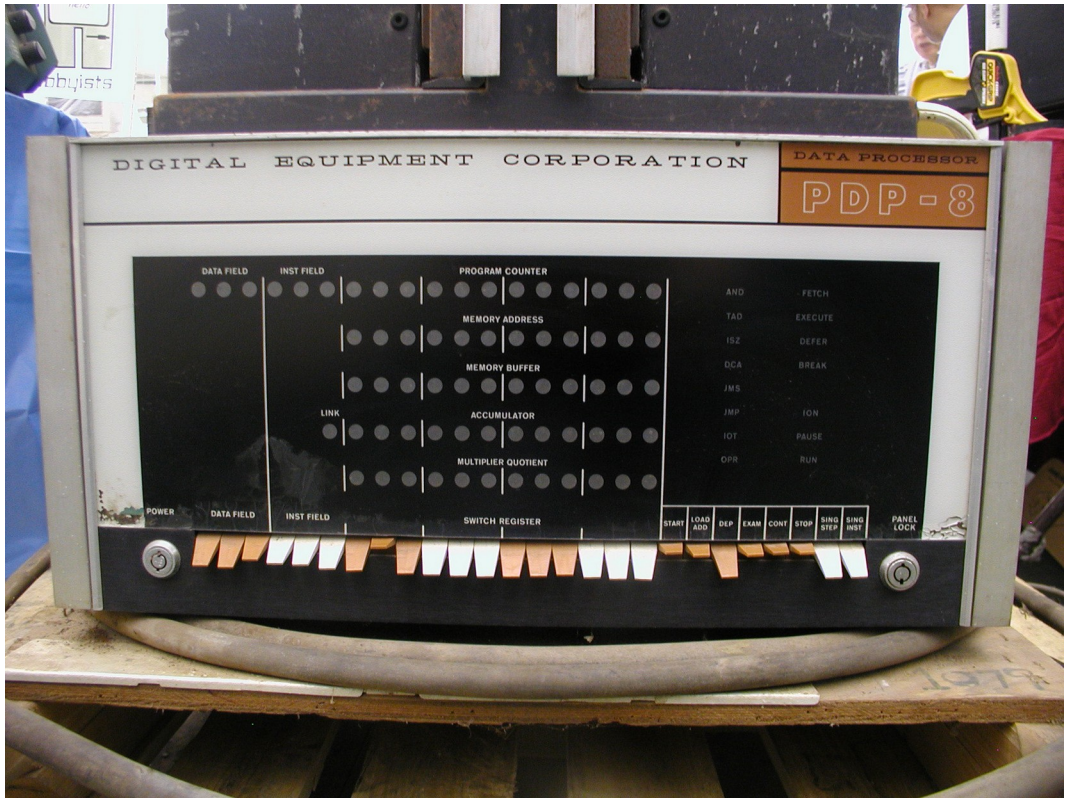
Hardware Rules 1940s - 1950s

- There was little or no software – what software that existed was often written in machine code
  - Programs were entered by physically modifying the hardware – the image shows developers programming an early ENIAC computer



Code Cowboys 1950s - 1960s

- Primitive operating systems
- Code cowboys could write better assembler code than compilers of the time
  - Called *programming to bare metal*
- Hardware was very expensive
  - Program quality measured by how efficiently it used physical resources
  - Programming done as close to machine code as possible
- Primary use of software was to automate repetitive and tedious manual tasks
  - Eg. Printing out account statements
- Image is a DEC PDP-8 showing the switches that were used to enter code in binary



Software Engineering 1960s - 1990s

- OS created a layer of abstraction between hardware and applications
- Demand was for automation of business workflows and procedures
  - Eg. Automate a payroll system
- Required adapting engineering construction principles to software development
- Programming languages were procedural (ALGOL, FORTRAN, C)
- Mainframes represented “islands of computing”
- Hardware still very expensive
  - Mainframe memory was $10/byte
  - Image is an IBM/370 CPU and magnetic core memory being unpacked

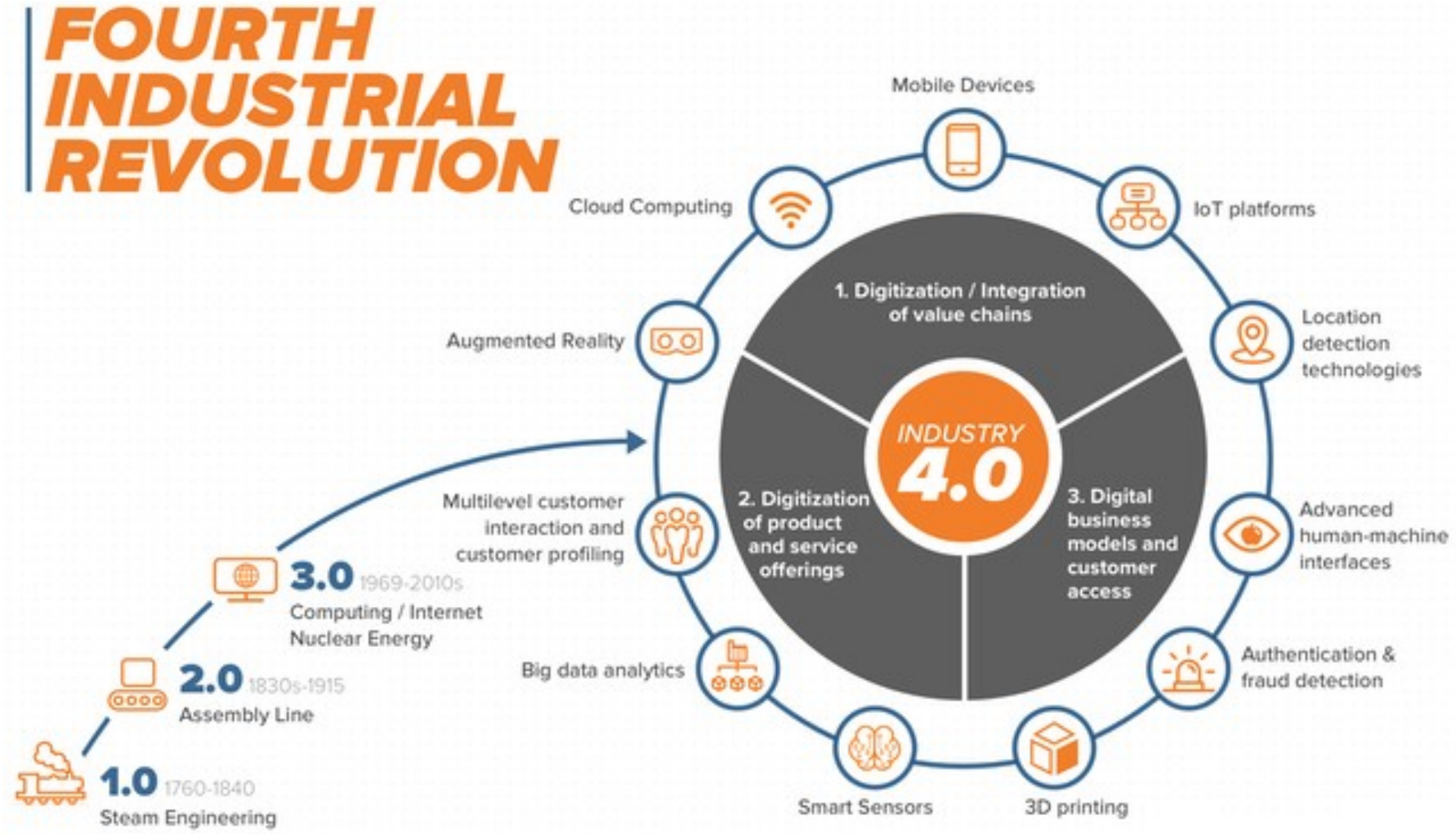
Cyber Human Systems 1990s - 2010s

- Drivers
  - Rise of personal computing
  - Rise of the Internet and networking
  - Businesses going “on-line”
  - Cheaper Hardware
- Programming
  - Increased used of OO languages like Java
  - Separated front-end users from the back-end business
    - *Backends were usually mainframe based*
    - *New front end technologies explode*
  - Emphasis was on network based infrastructure

Drivers of the Current Era

- Exponential increases in the amount of data being generated and needing to be processed - generally referred to as “Big Data”
  - Direct result of devices being connected in the Internet of Things
- Support for new paradigms of computing – machine learning and analytics
  - Processing large amounts of data at scale and often in real time
- Rapidly increasing hardware capabilities
  - Virtualization, cloud computing, containerization and microservices
- Reduced time to market for new or upgraded applications
- Deployment of applications at massive scales
  - Reformulation of the basic ideas of software architecture
- No down-time or disruption of services
  - High reliability software
  - Critical systems driven by software
- The increasing use of functional and metaprogramming

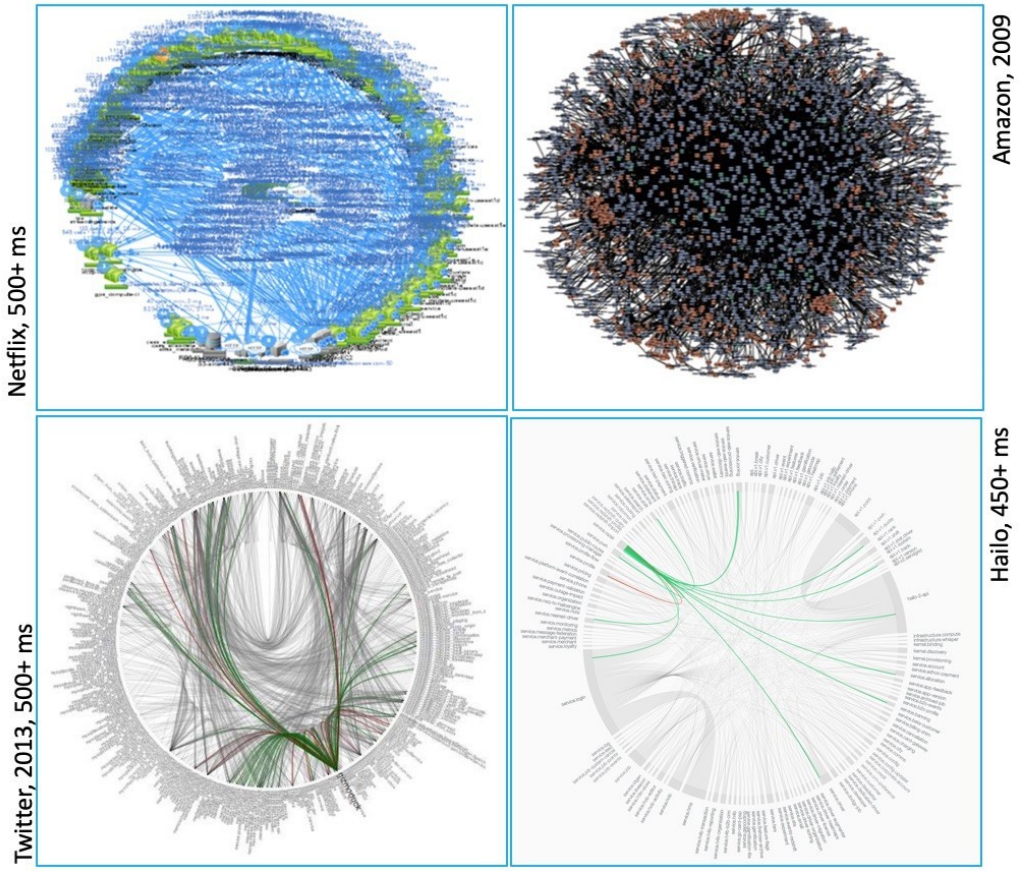
Cyber Physical Systems



LearnQuest

Rethinking Application Development

- Requires new ways of thinking about app development
  - Release time needs to be days, not months
  - Robustness is essential – *no downime*
- Massive scales
  - Hundreds of millions of transaction per second
  - Zeta-bytes of data
  - Data is generated in various forms from user interactions and connected devices
- Applications microservices
  - Clusters of smaller scalable components working together
  - Requires writing code for containerized deployment as opposed to stand alone applications
  - Often result in a “death star” architecture



Programming Paradigms

- A programming paradigm is a set of:
  - Assumptions about what the components of a program should be
  - Techniques and principles for building 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” and supported in programming languages when a set of problems arise that the paradigm is better suited to solve than the other paradigms in use
- The main-stream paradigms in use today – and supported in Java are:
  - Structured or procedural programming
  - Object Oriented Programming
  - Functional Programming

Programming Language “Styles”

- Imperative programming
  - Code is a series of instructions that specify the computational steps to be executed
  - Usually said to be expressible as a Turing machine
  - Intended to be directly compiled more or less directly into assembly code
  - Imperative code is usually bundled into reusable “procedures”
  - DRY principle – “Do not repeat yourself”
- Declarative programming
  - Code is a description of what a final result should be
  - Usually done by calling an existing procedure
- This is a continuum of style, not discrete, mutually exclusive categories
  - Most code is a mixture of imperative and declarative styles
  - Eg. x = sin(y) * 3.14159
    - *The call to sin() is declarative since we don’t know (or care) how the sin is computed*
    - *The multiplication is imperative code*

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
- 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
```

Image Credit: https://www.researchgate.net/figure/Subroutine-Thomas-written-in-FORTRAN-95_fig5_330504432

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 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
      |
```

Image Credit: https://computerhistory.org/blog/the-apl-programming-language-source-code/?key=the-apl-programming-language-source-code

Other Paradigms

- Two other paradigms that we will not deal with since Java does not support them
- Symbolic Programming or meta-programming
  - Treats code and data as the same thing – sets of symbols
  - Allows code to rewrite itself or to write new code
  - Introduced in LISP in the 1960s
- Logical Programming
  - Treats code as propositional logic
  - Developed by the Prolog language in 1972
  - Image shows a basic Prolog program

```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

This results in the following query being evaluated as true:

```
?- sibling(sally, erica).
Yes
```

Image Credit: https://codedocs.org/what-is/prolog

Modern Programming Languages

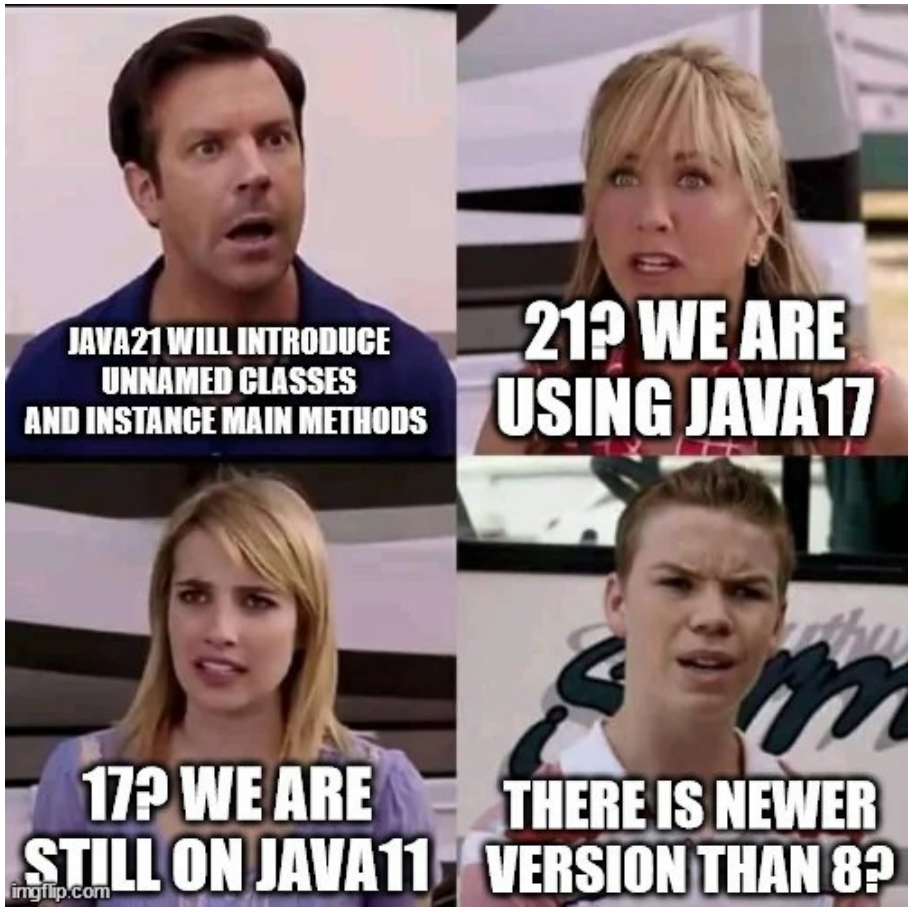
- Most modern programming languages support more than one paradigm
  - Modern languages like Rust, Go and Julia are designed to support multiple paradigms
- Legacy Languages are often revised to add support for a paradigm
  - COBOL added object oriented support
  - Java added support for functional programming in Java 8
- Why paradigms go mainstream
  - Most of the different paradigms have existed for over 50 years
  - They are designed to solve a particular class of problems
  - The types of problems industry deals with change over time
  - Changes often result from changes in technology and user requirements
  - Existing paradigms may not be able to solve these new problems
  - A different paradigm is main-streamed that can solve the problems

Paradigm Focus

- Structured programming
  - Developed to respond to the need to automate the business processes in the 1960s
  - COBOL = *Common Business Oriented Language*
  - A data set was read in, algorithms used to process it, 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 changes
- 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

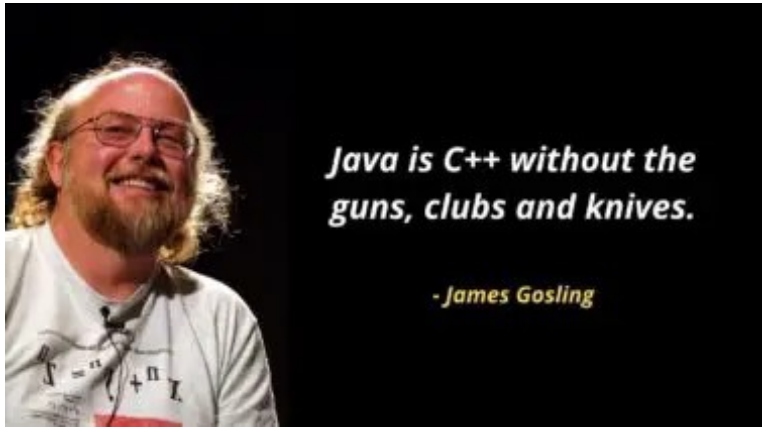


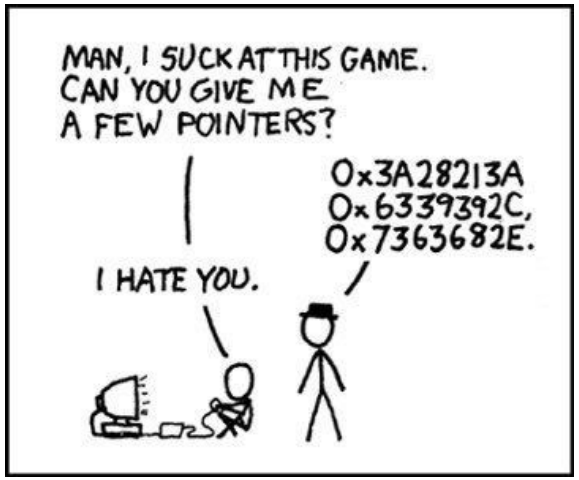
The Java Story

- Java was a project developed at SUN Microsystems to support interactive TV
  - Hardware wasn't powerful enough to do 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

- 19950s – C++ and OO were very popular
- Gosling intended Java to be a better C++
  - Easy for C++ programmers to learn and adopt
- Threw away all 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
  - Did not want to be in the business of “building” Java compilers
- Published two specifications
  - The Java Language Specification that describes how Java works,
    - *Llike other standard language specification (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

Mess Around with Java 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++
  - Although 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

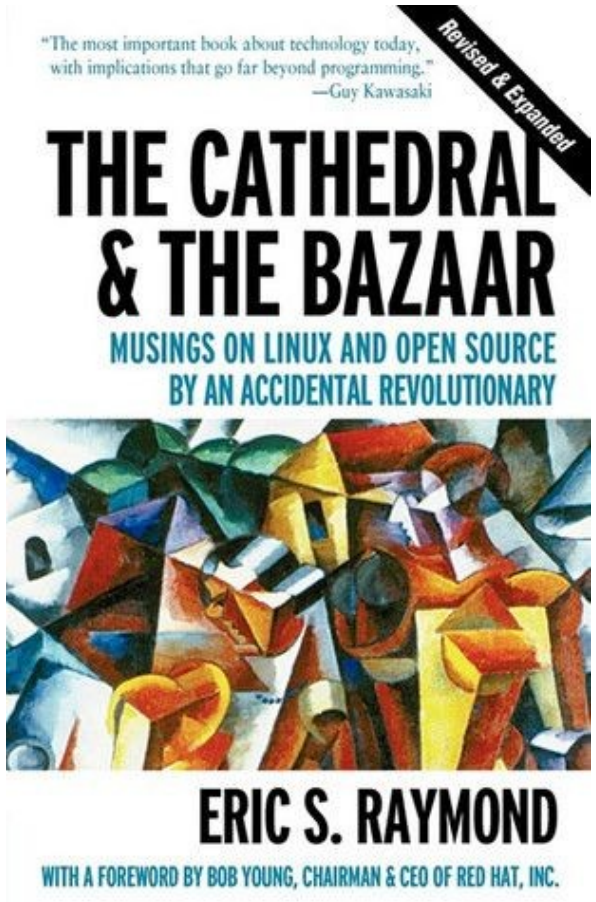
Sun Microsystems and Microsoft have settled their long-running lawsuit over Microsoft's use of Sun's Java software.
Under the settlement, Microsoft will pay Sun $20 million and is permanently prohibited from using "Java compatible" trademarks on its products, according to Sun. Sun also gets to terminate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

- SUN separated it's business into two independent areas
  - **The Bazaar** – the commercial business of SUN hardware and software
  - T**he Cathedral** – the resources needed to develop and support Java
  - Primarily to avoid the perception that to use Java, you had to "buy SUN"
- Wanted Java to evolve in response to the user community, not sales and marketing executives at SUN
  - User requirements should dictate the evolution of the language specification



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
    - *IBM has been a major player in the JCP*
- 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.*

Java Timeline

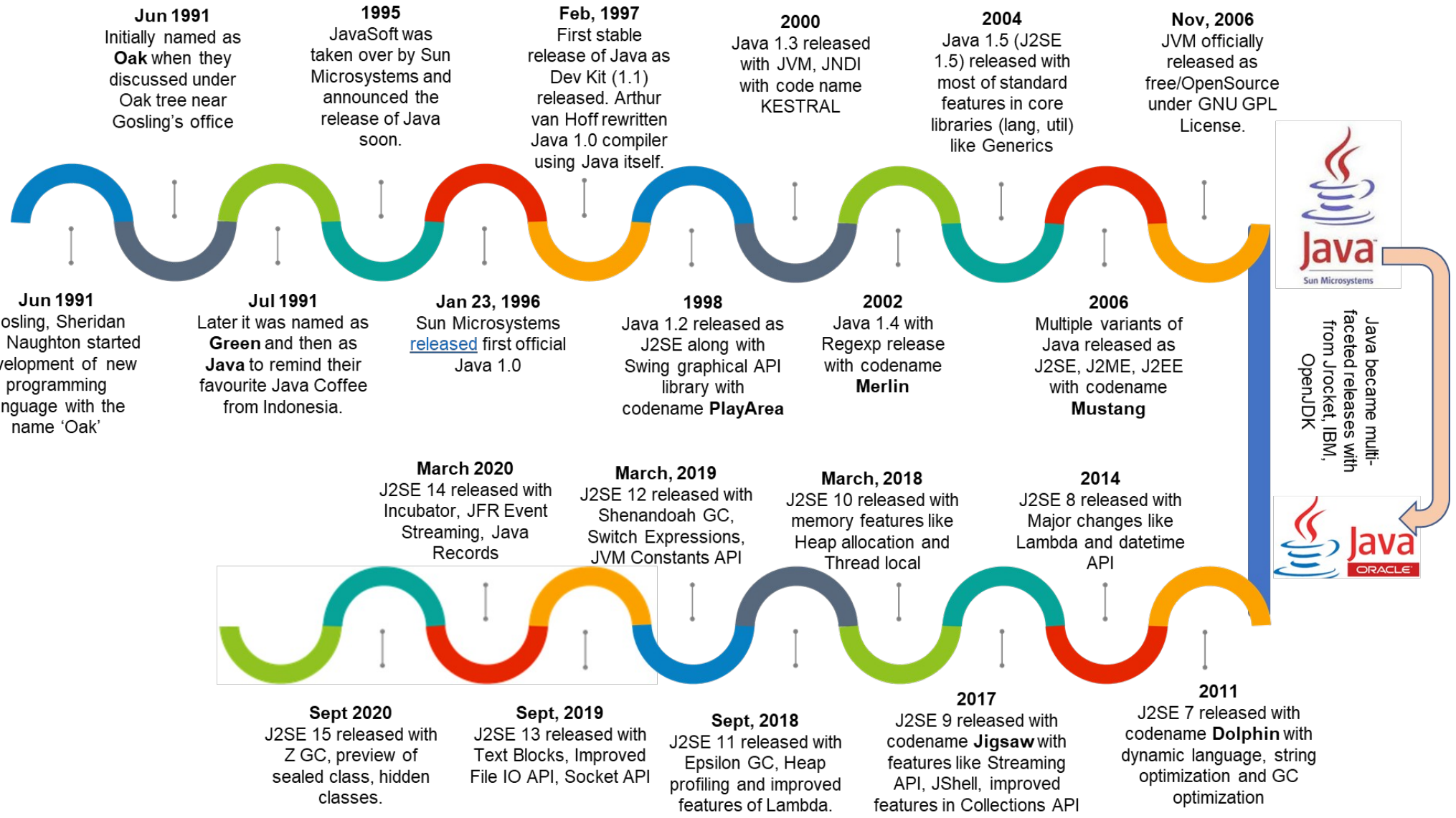


Image Credit: https://dzone.com/articles/why-java-is-so-young-after-25-years-an-architects


Questions?

Demo

Introduction to Eclipse and Hello World


Lab 1-1
Hello World


Java™