Programming Languages and Concepts

Siegfried Benkner

Research Group Scientific Computing

Universität Wien

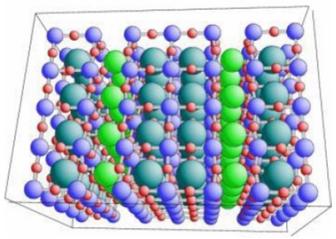
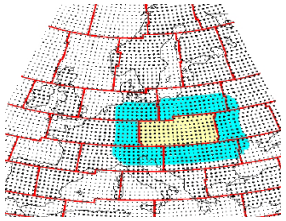
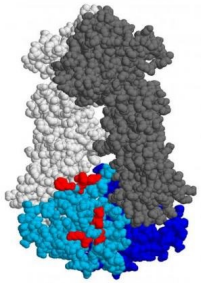
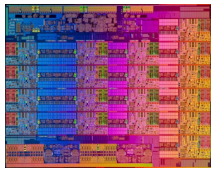
Research Group Scientific Computing (SC)

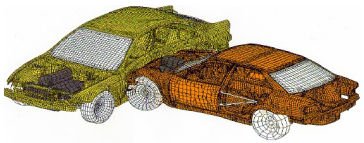
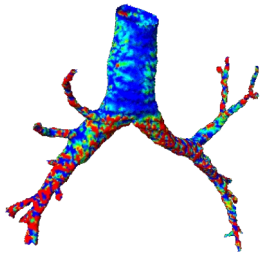
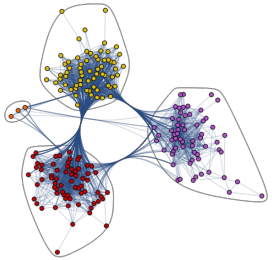
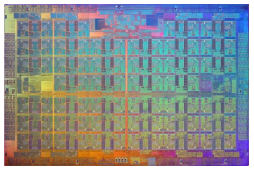
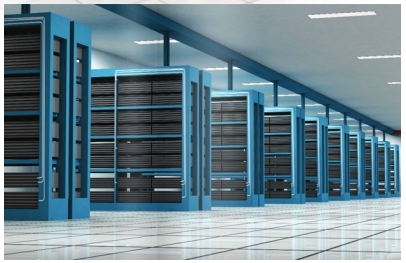
Research in

https://sc.cs.univie.ac.at/

- programming paradigms,
- languages, compilers, runtime systems, and
- software infrastructures

for complex compute- and data-intensive applications on **high-performance parallel and distributed computing** platforms.

Goals

- **Learn and understand main programming paradigms**
  - Object-oriented, Functional, Logic, Parallel/Concurrent, Aspect-Oriented
- **Understand main concepts of programming languages**
  - to effectively use programming languages.
  - to facilitate learning new languages.
- **Gain insight** into main aspects of the **design, compilation and runtime** support of programming languages.
- **Learn and understand main aspects of compiler development.**
- Understand applicability of different languages, constructs and paradigms for practical application scenarios.

Study Programmes

- **Bachelor Informatik** (3. Semester)
  - Informatik
  - Data Science
  - Medieninformatik
  - Medizininformatik
  - Scientific Computing
- **Master Computational Science**
  - APMG-A Foundations of Computational Science A
  - APMG-A Foundations of Computational Science B
- **Significant Implementation Requirements (SIR)**
  - Hands-on program development with Java, Rust, Antlr, …

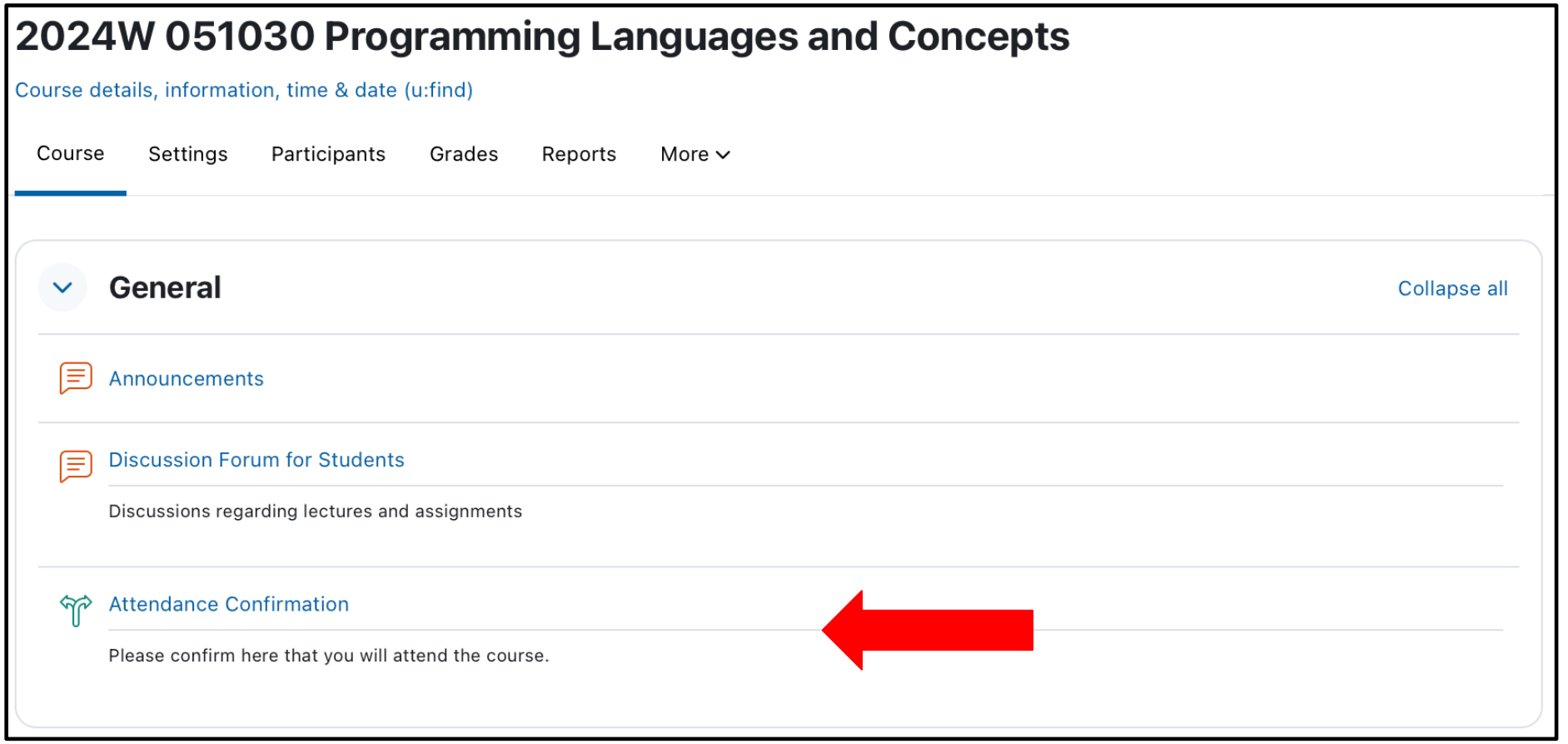
Prerequisites

- Mandatory
    - StEOP
- Recommended
    - Programmierung 2
    - Algorithmen und Datenstrukturen

Registration

In order to claim your seat in the lecture you need to

- sign the attendance list in the first lecture, *or*
- register on Moodle by Oct. 3



Languages/Tools used in PLC

Java

- Object-Oriented and Functional Programming Concepts
- Concurrent Programming Concepts

Rust

- Performance-Oriented Programming, Systems Programming
- Memory Safety, Freedom of Data Races, Move Semantics, Borrow Checker

Prolog

- Logic Programming, Declarative Programming
- Unification, Backtracking, Recursion

Aspect-Oriented Programming

- Separation of concerns; modularity

Compiler Construction

- ANTLR parser generator for context free grammars

Schedule (tentative)

Date	Topic	
2./3.10.	Introduction, **Assignment 1**	OO Concepts I
9./10.10.	OO Concepts II	Generic Types
16./17.10.	Concurrent Programming, **Assignment 2**	Selected Concepts
23./24.10.	On-site Lab	*Homework Assignment 1&2*
30.10/31.10.	Remote Q/A	*Homework Assignment 1&2*
6.11/7.11.	**Deadline Assignment 1 & 2, Test 1**	Feedback Assignment 1 & 2
13./14.11.	Rust	Rust, **Assignment 3**
20./21.11	On-site Lab	*Homework Assignment 3*
27.11./28.11.	On-site Lab	*Homework Assignment 3*
4./5.12.	Selected Concepts	Selected Concepts + Q&A
11./12.12.	**Deadline Assignment 3, Feedback Assignment 3**	Compiler Construction, **Assignment 4**
8./9.1.	Aspect-Oriented Programming	Logic Programming
15./16.1.	On-site Lab	Remote Q/A
22./23.1.	**Deadline Assignment 4, Test 2**	No Lecture

Grading

Grading is based on tests and assignments:

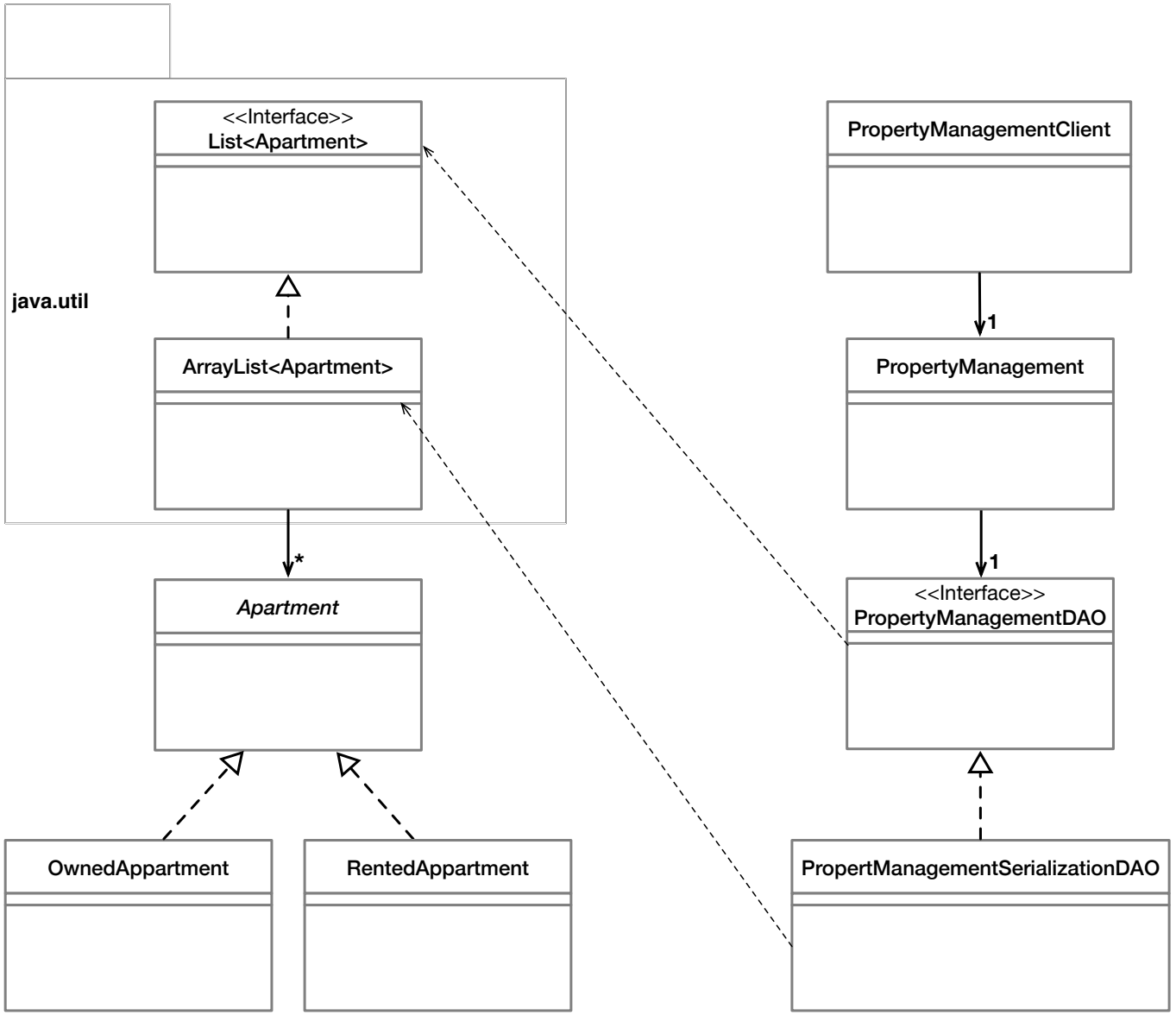
- Assignments: pass/fail

  an assignment is passed if it is successfully submitted on the online platform
- Tests: 100 (=50+50) points in total

Grading Scheme:

5: 0 – 49 points	or	more than 1 assignment failed
4: 50 – 61	and	at most 1 assignment failed
3: 62 – 74	and	at most 1 assignment failed
2: 75 – 87	and	all assignments passed
1: 88 – 100	and	all assignments passed

Assignment 1



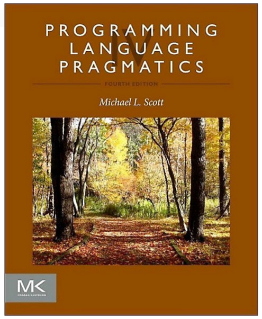
PLC – Introduction and Overview

- **Classification of Programming Languages**
- **Compilation vs. Interpretation**
- **Popular Programming Languages**
- **Future Perspectives**

Literature

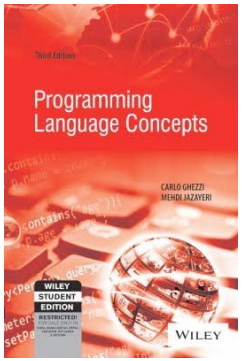
- Michael L. Scott,

  Programming Language Pragmatics Fourth Edition, 2016



- Mehdi Jazayeri, Carlo Ghezzi,

  Programming Language Concepts, 3ed, 2008



Classification of Programming Languages

Imperative Languages

- Express sequence/control flow of computation ("how")
- Statements may change state of program
  - **Procedural**/Structured (Fortran, Pascal, Basic, C, C++, Rust)
  - **Object-Oriented** (Smalltalk, C++, Java, Scala)
  - **Scripting languages** (Perl, Python, JavaScript, PHP)

Declarative Languages

- Express the logic of a computation without describing its control flow;
- Focus on "what" instead of "how"
  - **Logic**, constraint-based (Prolog)
  - **Functional** (Scheme, Lisp, Scala)
  - **Query Languages** (SQL subset)

Classification of Programming Languages

- **Procedural** (C)

```
int gcd(int a, int b) {
   while (a != b)
      if (a > b) a = a-b; else b = b-a;
   return a;
}
```

- **Functional** (Haskell)

```
gcd a b
| a == b = a
| a > b = gcd (a-b) b
| a < b = gcd a (b-a)
```

- **Logical** (Prolog)

```
gcd(A, A, A).
gcd(A, B, G) :- A > B, N is A-B, gcd(N, B, G).
gcd(A, B, G) :- A < B, N is B-A, gcd(A, N, G).
```

Programming Styles – e.g., Java

```
ArrayList<Person> pl = ...
...
/*
 * imperative (old) style
 */
for (Iterator<Person> it = pl.iterator(); it.hasNext(); ) {
    Person p = it.next();
    if (p.age > 30) System.out.println(p.name);
}

/*
 * functional (new) style
 */
pl.stream().filter(p -> p.age > 30)
           .forEach(p -> System.out.println(p.name));
```

Programming Paradigms

A paradigm is a style of programming, characterized by certain key concepts.

- **Imperative** programming: variables, commands, procedures
- **Object-oriented** programming: classes, objects, inheritance, polymorphism
- **Functional** programming: values, expressions, functions; no side-effects; immutable values, immutable state
- **Logic** programming: predicates, rules, assertions, relations
- **Concurrent** programming: threads, synchronization
- **Parallel** programming: processes, data decomposition, distribution, coordination, communication/synchronization
- **Reactive** programming: asynchronous data flows/streams; message-driven

Classification of Programming Languages

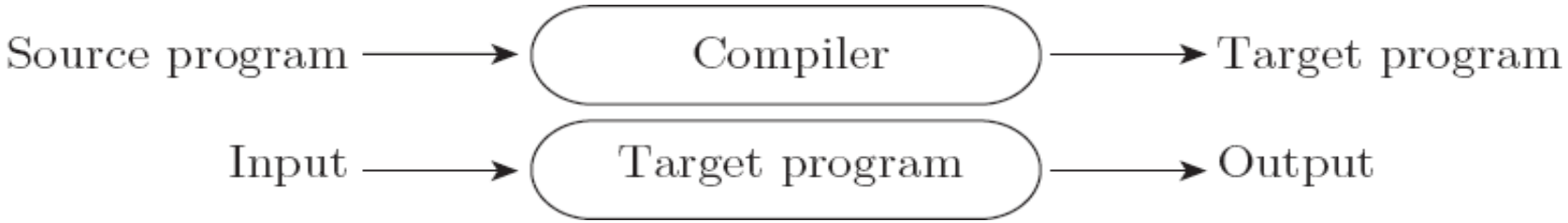
Most languages support multiple programming paradigms

- Procedural
- Object-oriented
- Functional
- Logic
- Data-Flow
- Aspect-Oriented
- Actor-based
- Concurrent
- Parallel

Classification of Programming Languages

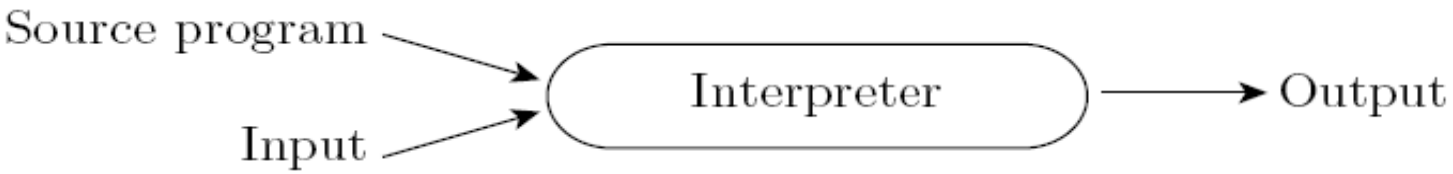
Compiled Languages

- Compiler translates the high-level source program into an equivalent target program (typically in machine language)



Interpreted Languages

- Interpreter stays around for the execution of the program
- Interpreter is the locus of control during execution

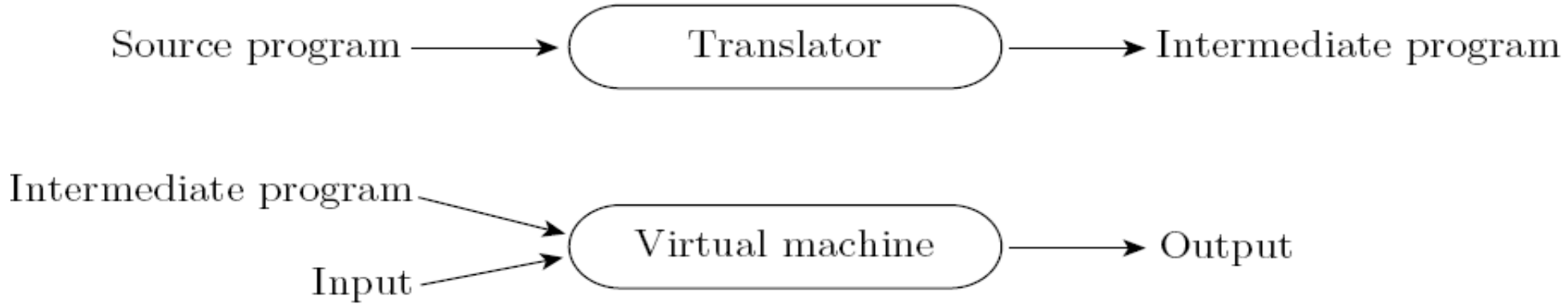


See: Programming Language Pragmatics, Fourth Edition, Michael L. Scott

Classification of Programming Languages

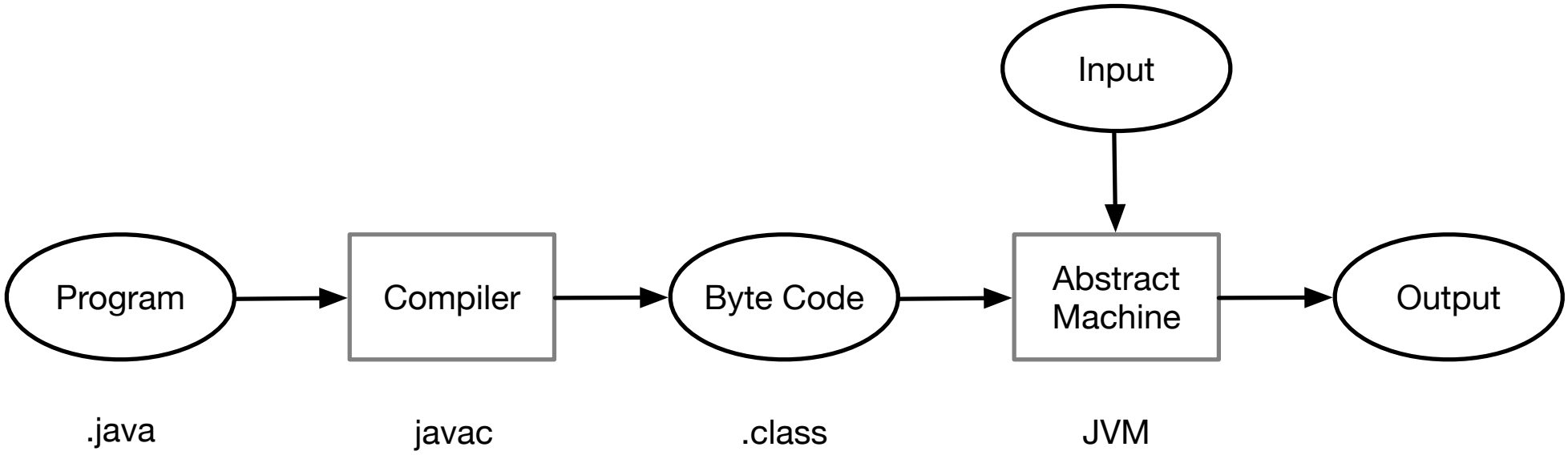
Compilation vs. Interpretation

- Common case is compilation or simple pre-processing, followed by interpretation.
- Most language implementations include a mixture of both compilation and interpretation.



See: Programming Language Pragmatics, Fourth Edition, Michael L. Scott

Compilation vs. Interpretation - Java



Why Compilation?

Better Safety/Correctness

- check all names are defined (classes, methods, fields, variables, ...); check that names have correct type; check visibility rules (public, private)
- optimize code (method inlining, ...)

Better Performance

- Checks/optimizations done at compile time not at runtime
- Execute optimized code

Why not compilation?

- May reduce flexibility through static bindings and static type checks
- Web programming often requires more flexibility (e.g., JavaScript, PHP, ...)

Classification of Programming Languages

Dynamic and Just-in-Time Compilation

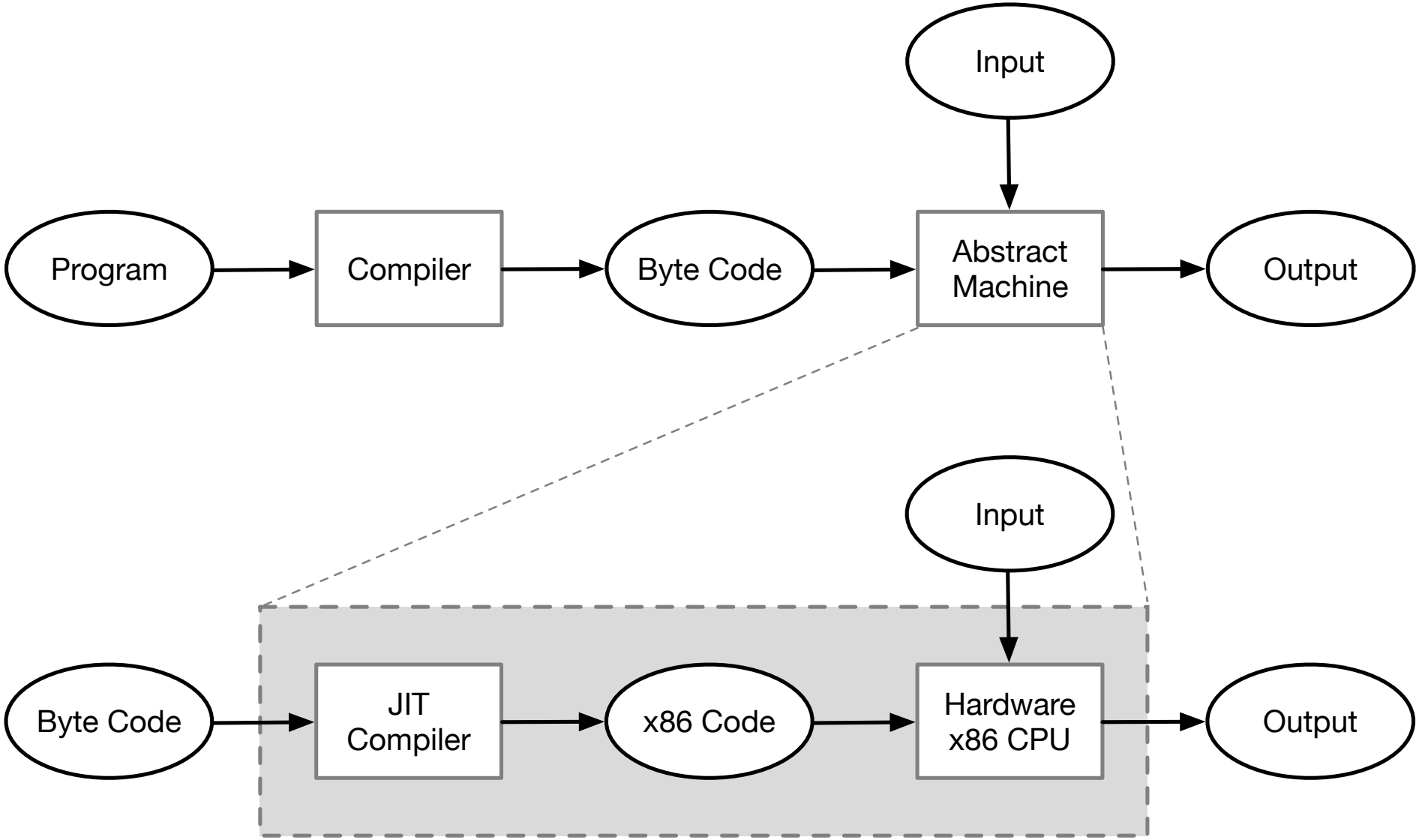
- The Java compiler generates platform-independent byte code. Just-in-Time (JIT) compiler may compile byte code into machine code at runtime when the program is executed.

- Lisp or Prolog invoke the compiler on the fly, to translate newly created source into machine language, or to optimize the code for a particular input set.

- The main C# compiler produces .NET Common Intermediate Language (CIL), which is then translated into machine code immediately prior to execution.

See: Programming Language Pragmatics, Fourth Edition, Michael L. Scott

Just-in-Time Compilation



Classification of Programming Languages

Managed Languages

- Memory management (and other aspects of resource management) is automatically managed by the **runtime system**.
- The runtime system plays an important role during program execution.
- Examples: Java, C#

Unmanaged languages

- Memory management (and other aspects of resource management) are under control of the programmer.
- The program executes natively without a (or only a very lightweight) runtime system.
- Examples: C/C++, Rust

Classification of Programming Languages

- **Machine languages**: interpreted directly in hardware
- **Assembly languages**: thin wrappers over a machine language
- **System languages**: designed for writing low-level tasks, like memory and process management; compiled, statically typed
- **High-level languages**: machine-independent; compiled/interpreted, statically typed
- **Scripting languages**: dynamic, high-level; clue things together; interpreted/compiled
- **Domain-specific languages**: used in highly special-purpose areas only

Classification of Programming Languages

Statically vs. Dynamically Typed Languages

- Static Typing

  type checking at compile time;

  e.g., Java, C/C++, Scala

```
// Java
int i;
i = 1;
i = "abc"; // type mismatch
```

- Dynamic Typing

  type checking performed at runtime;

  e.g., Python, JavaScript

```
//JavaScript
var i;
i = 1;      // i is a Number
i = "abc"; // i is a String
```

Strongly vs. Weakly Typed Languages

- Strong vs. weak typing usually means to what extent a language enables type checks such that errors are detected

Successful Programming Languages

- **Expressive Power**
  - Theoretically, all languages are equally powerful (Turing complete)
- **Productivity**
  - Abstraction facilities enhance expressive power
  - Ease to learn and use (write, maintain, analyze)
  - Tool support
- **Portability**
  - Ease of Implementation
  - Runs on virtually everything
- **Performance**
  - Fortran has extremely good compilers; JIT; parallel execution
- **Availability, Economics**
  - Open Source; Standards; Large user base

Programming Language Concepts

- **Structuring the Data**

  Names, bindings, scopes, data types, values, variables, type inference, generic types, data abstractions, conversion and contexts, **...**

- **Structuring the Computation**

  Expressions, statements, control flow, procedures, functions, methods, control abstractions, concurrency, parallelism, exception handling, **...**

- **Structuring the Program**

  Encapsulation, modularity, classes, interfaces, packaging, **...**

Some “important” languages

- Fortran (1954)
- Lisp (1958)
- Cobol (1959)
- **C** (1972)
- Prolog (1972)
- Ada (1980)
- **C++** (1983)
- Objective C (1983)
- Haskell (1990)
- **Python** (1991)
- Ruby (1993)
- R (1993)
- **Java** (1995)
- **JavaScript** (1995)
- PHP (1995)
- C# (2001)
- Scala (2003)
- Go (2012)
- Julia (2012)
- Swift (2014)
- Rust (2015)
- Kotlin (2016)

PYPL Index 2024

PopularitY of Programming Language Index https://pypl.github.io/PYPL.html

Rank	Change	Language	Share	1-year trend
1		Python	29.66 %	+1.6 %
2		Java	15.64 %	-0.2 %
3		JavaScript	8.3 %	-1.0 %
4		C#	6.64 %	-0.1 %
5		C/C++	6.46 %	-0.2 %
6	↑	R	4.66 %	+0.2 %
7	↓	PHP	4.35 %	-0.5 %
8		TypeScript	2.96 %	-0.0 %
9		Swift	2.69 %	+0.0 %
10	↑	Rust	2.65 %	+0.6 %

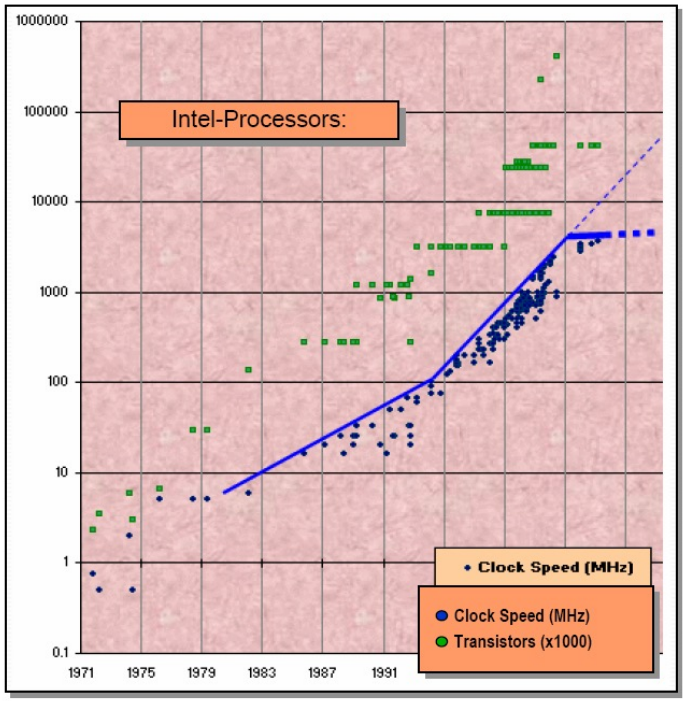
Rank	Change	Language	Share	1-year trend
10	↑	Rust	2.65 %	+0.6 %
11	↓	Objective-C	2.45 %	+0.2 %
12		Go	2.08 %	+0.2 %
13		Kotlin	1.95 %	+0.2 %
14		Matlab	1.45 %	-0.1 %
15		Ruby	0.99 %	-0.1 %
16	↑↑	VBA	0.97 %	+0.0 %
17	↑↑	Powershell	0.96 %	+0.1 %
18	↓↓	Ada	0.96 %	-0.1 %
19	↓↓	Dart	0.94 %	-0.0 %
20	↑	Lua	0.69 %	+0.1 %

Evolution of Processors and Performance

- The performance of processors has increased exponentially, in line with Moore's law.

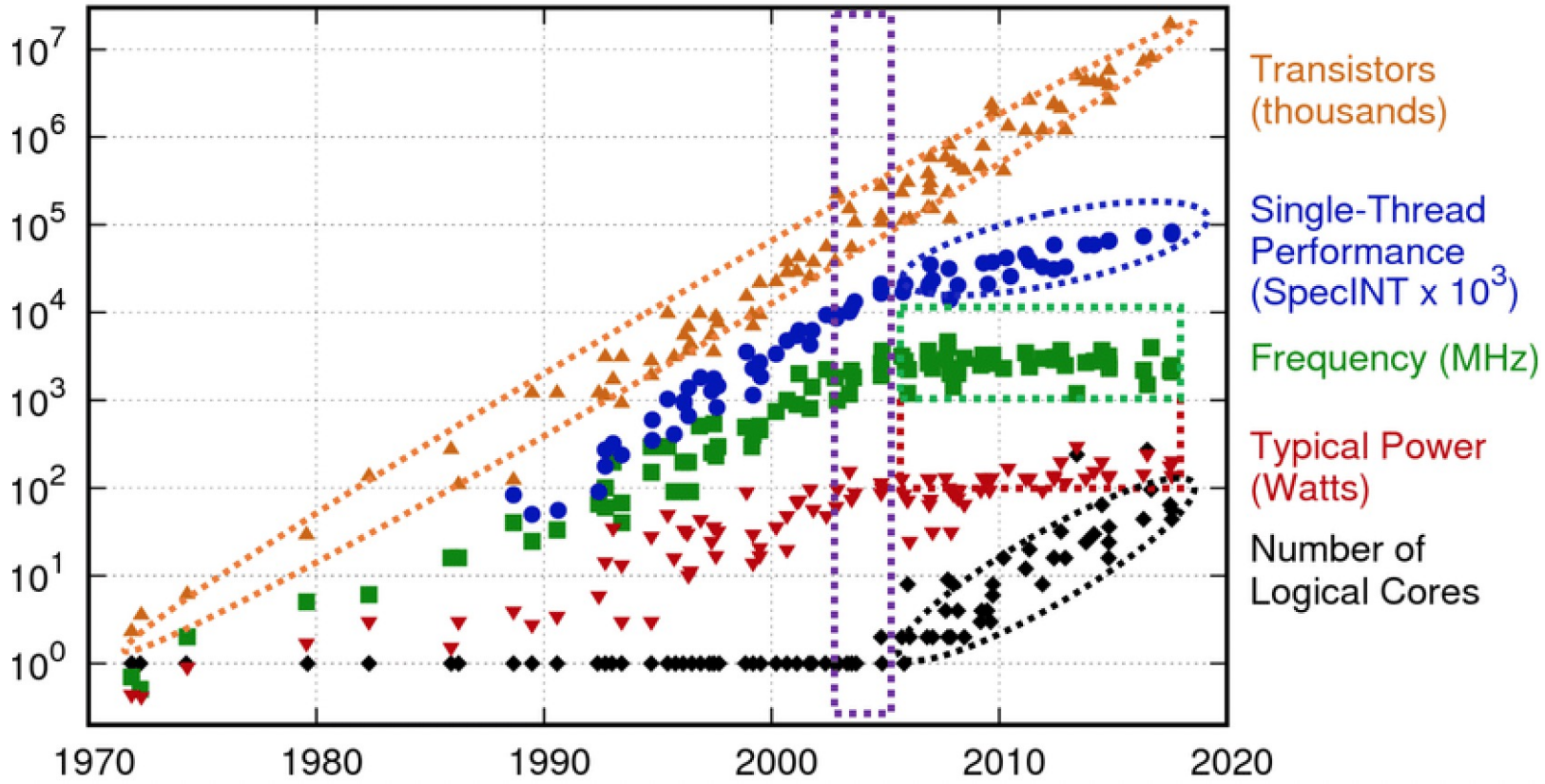
Moore's Law:
The number of transistors on a chip doubles every ≈18 months.

8086	1978	29K
80286	1982	134K
80386	1985	275K
80486	1991	1185K
Pentium	1993	31K00
Pentium II	1998	75K00
Pentium 4	2001	42M
Core 2 Extreme (4)	2006	582M
Nehalem (4)	2008	731M
Nehalem-EX (8)	2010	2300M
AMD EPYC (32)	2017	19200M
Apple M2 Ultra (24)	2023	134000M



- These single processor performance increases ended in 2002, mainly since the clock frequency could not be improved anymore at the rate of Moore's law.

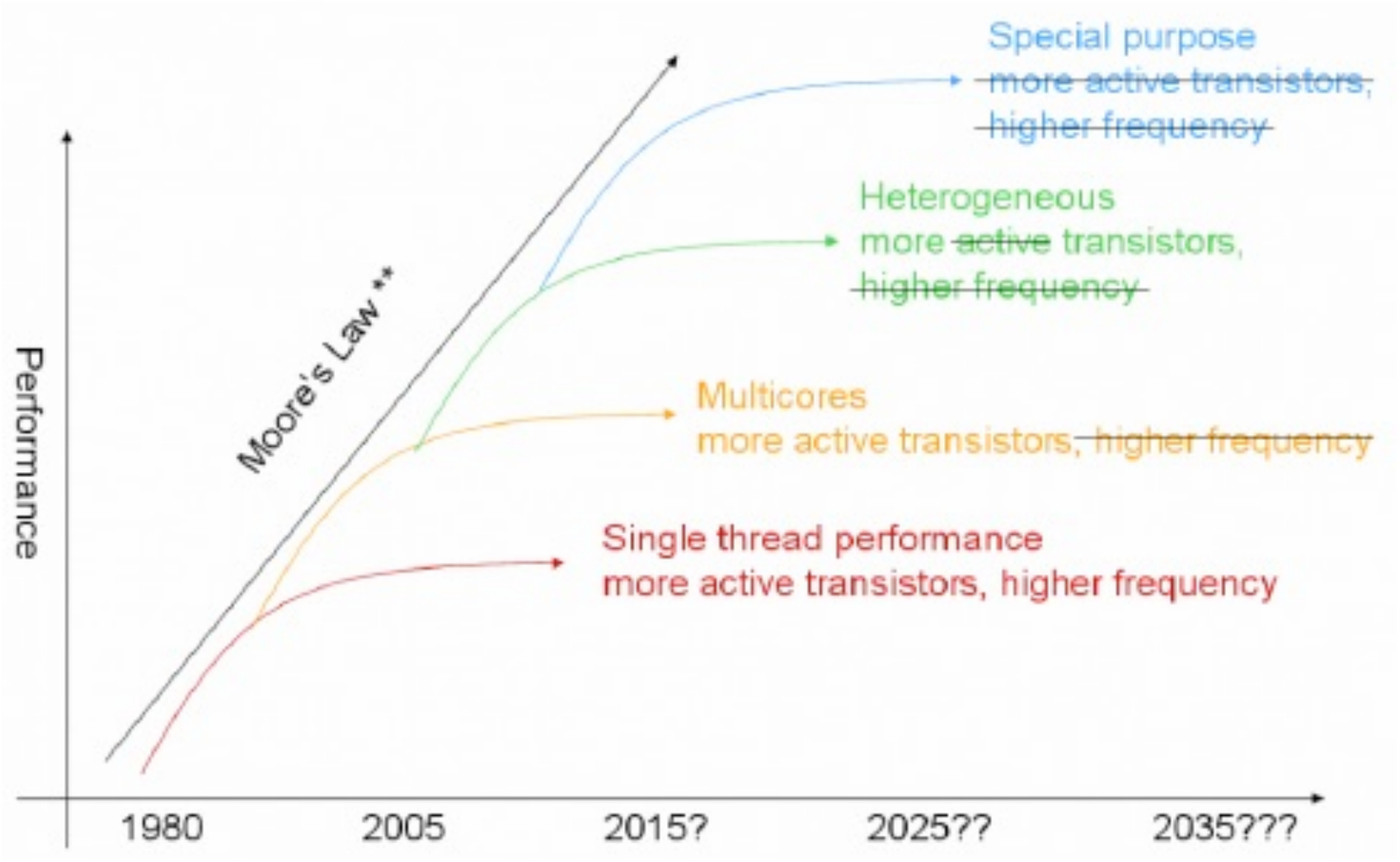
Evolution of Processors and Performance



M. Rusanovsky, et al (2019). BACKUS: Comprehensive High-Performance Research Software Engineering Approach for Simulations in Supercomputing Systems.

- CPU frequency stalled; Performance increase now mainly through parallelism.
- Sequential programs will not run faster anymore.
- Only parallel programs may run faster on multicore processors.

Evolution of Processors and Perfromance

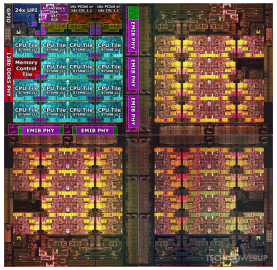


Source: H. Peter Hofstee (2005)

Processors – Parallelism & Spezialization

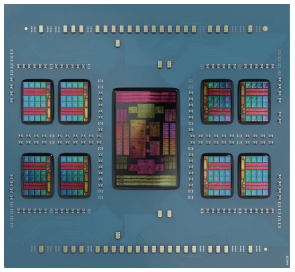
CPUs

Intel Xeon 8490H (2023)



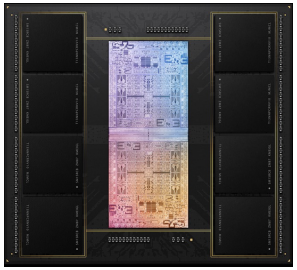
60 cores

AMD EPYC 9754 (2023)



128 cores

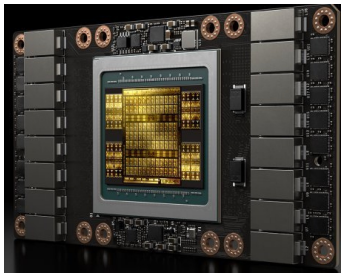
Apple M2 Ultra (2023)



16+8 cores + 76 GPU cores + NE

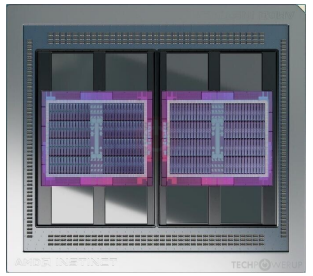
GPUs

NVIDIA Ampere GH100 (2023)



18432 CUDA cores + 576 Tensor cores

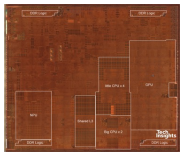
AMD Instinct MI250X (2023)



14080 stream processors

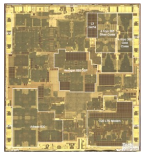
Mobile

Apple A17 (2023)



2+4 cores + 6 GPU cores + NE (16 cores)

Snapdragon 8 Gen2 (2023)



1+2+2+3 cores + GPU + NPU

Top 500: #1 Frontier (ORNL, USA)

Performance (Linpack) : 1.2 **ExaFlop/s** (=10^{18} Flop/s)

Cores: 8,699,904 (9,472 AMD EPYC 64C CPUS + 37,888 AMD Instinct MI250X GPUs)

Power: 22.8 MW

Domain-Specific Architectures (DSAs)

- Trade generality for performance and energy efficiency
- Come with their own domain-specific programming language (DSL)/API

GPU – Graphics Processing Unit (NVIDIA, AMD, …)

- Domain: Graphics, Scientific Computing
- Programmed with: CUDA, OpenCL

TPU – Tensor Processing Unit (Google Cloud TPU, …)

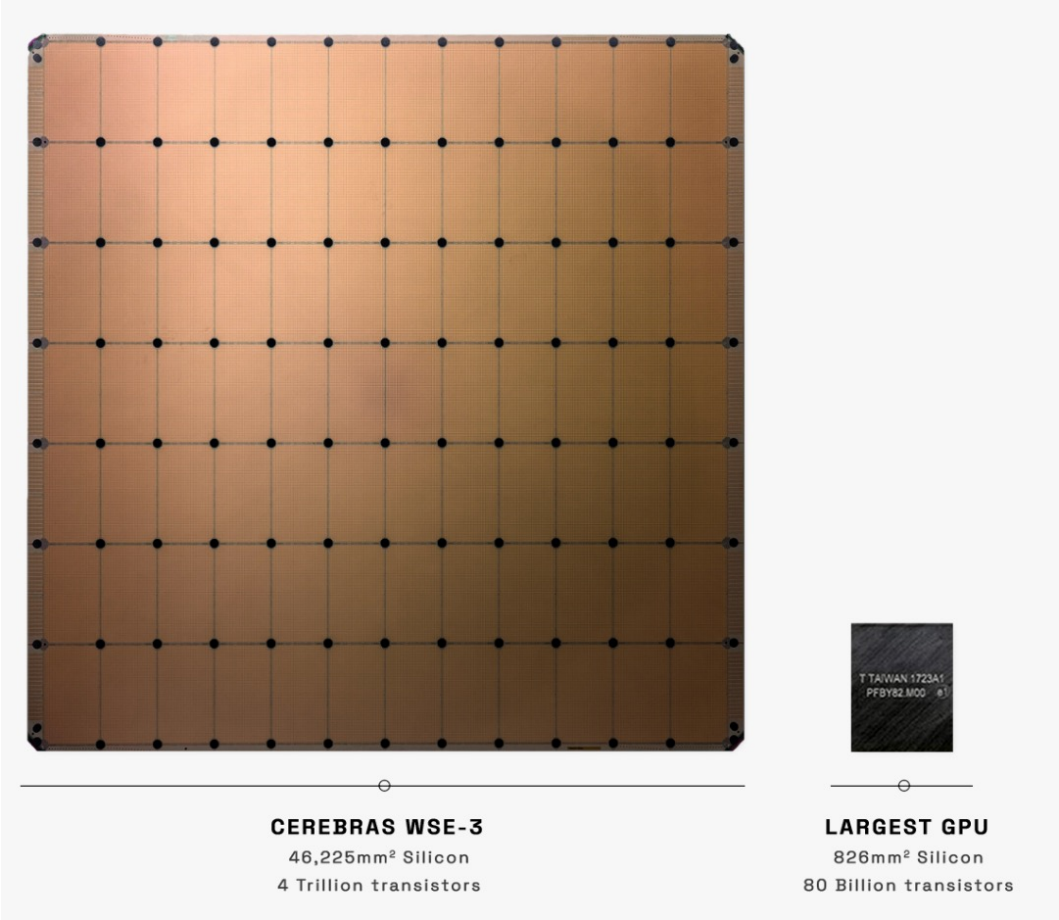
- Domain: Machine Learning, Deep Neural Networks
- Programmed with: TensorFlow, Caffe, PyTorch, …

VPU – Vision Processing Unit (Google Pixel, Movidius Myriad X, …)

- Domain: Computer Vision, AI
- Programmed with: Halide, …

Example: Cerebras WSE-3

Cores: 900,000
On-chip memory: 44 GB
Peak Performance (AI): >100 PFlop/S



PLC - Java Plattform and Technologies

- **Java Platform, Bytecode, JVM**
- **Java Technologies**
- **Organization of Java Programs**

Literature

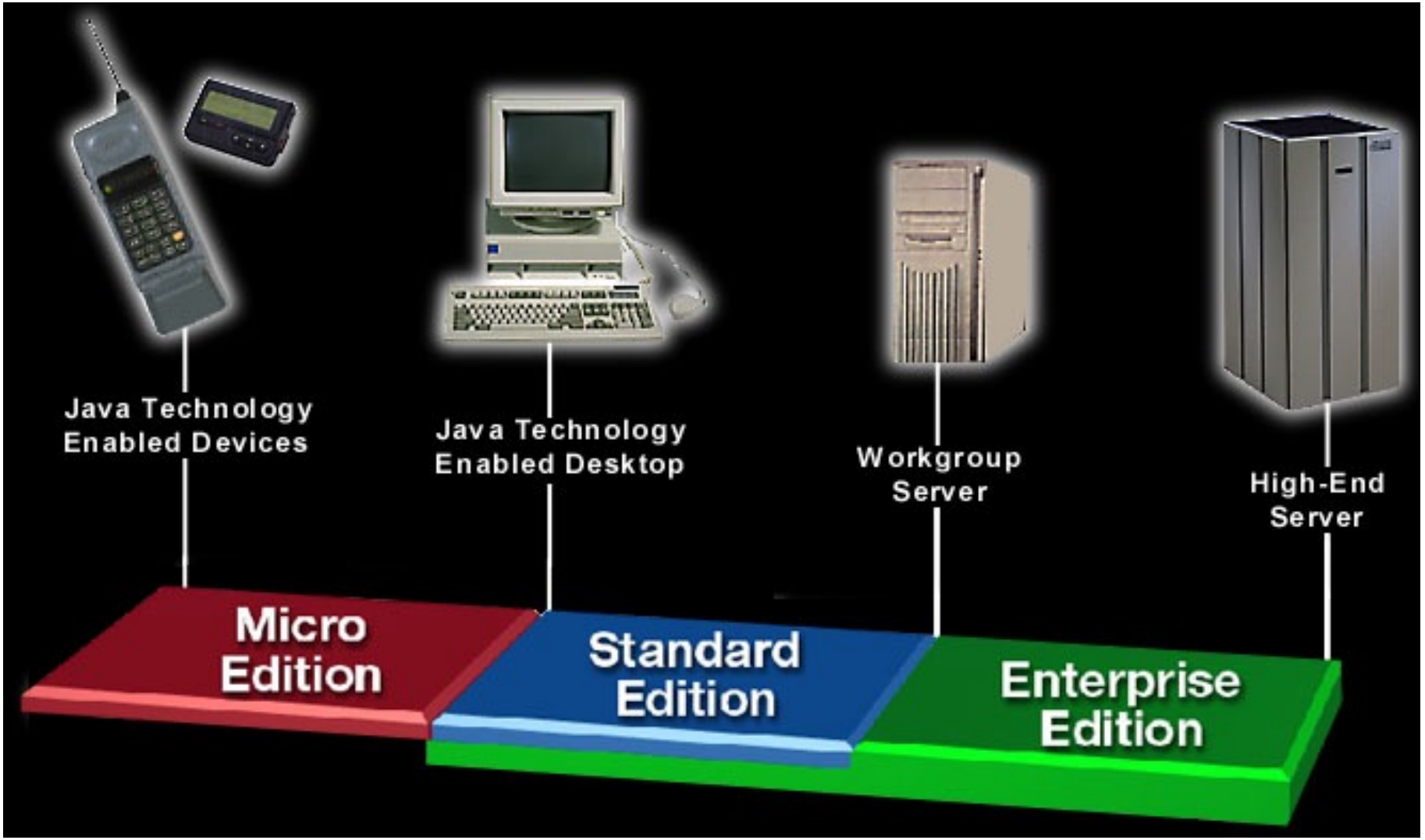
- The Java Tutorials
  http://docs.oracle.com/javase/tutorial/

- The Java Language Specification, J. Gosling et al.
  http://docs.oracle.com/javase/specs/jls/se18/html/index.html

- The Java Virtual Machine Specification, Tim Lindholm, et al.
  https://docs.oracle.com/javase/specs/jvms/se18/html/index.html

- JDK 18 Documentation
  https://docs.oracle.com/en/java/javase/18/index.html

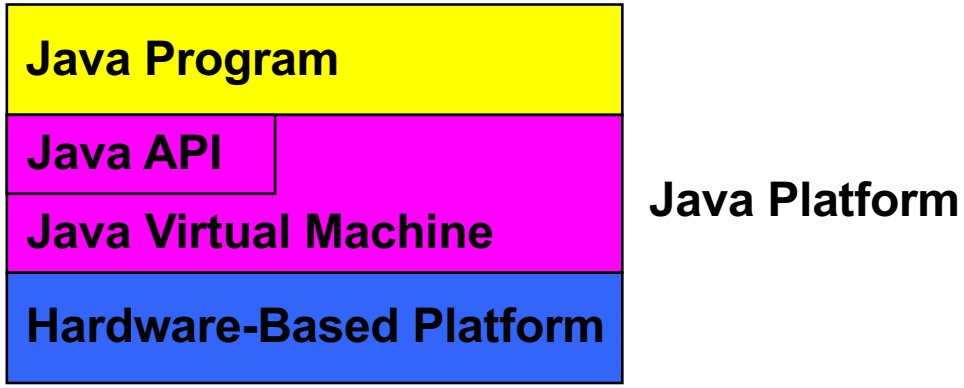
Java Technology



Java Platform

- Software-only platform, which runs on all types of hardware.

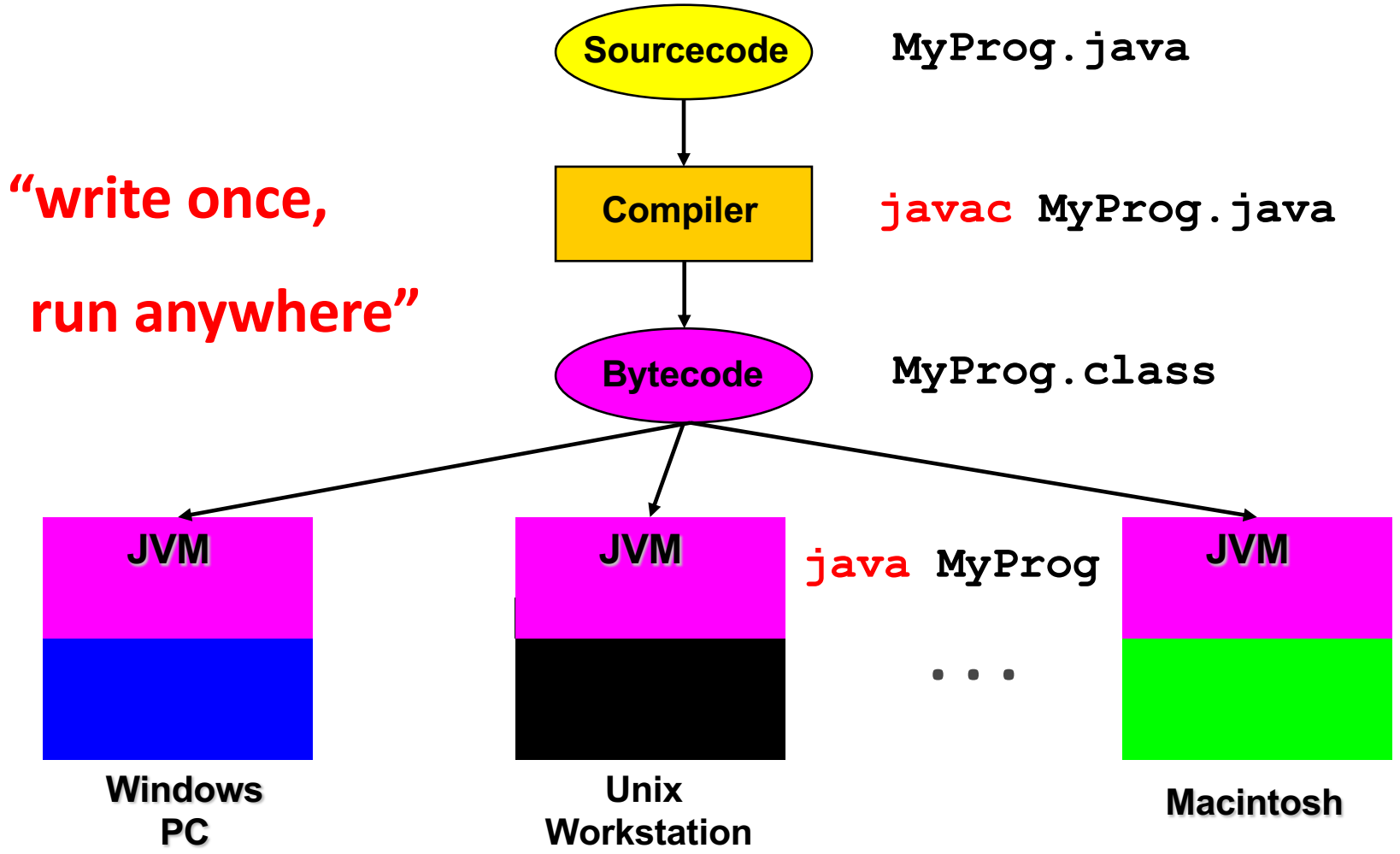
- Comprised of
  - Java Virtual Machine (JVM)
  - Java Application Programming Interface (Java API)



Main Features of Java

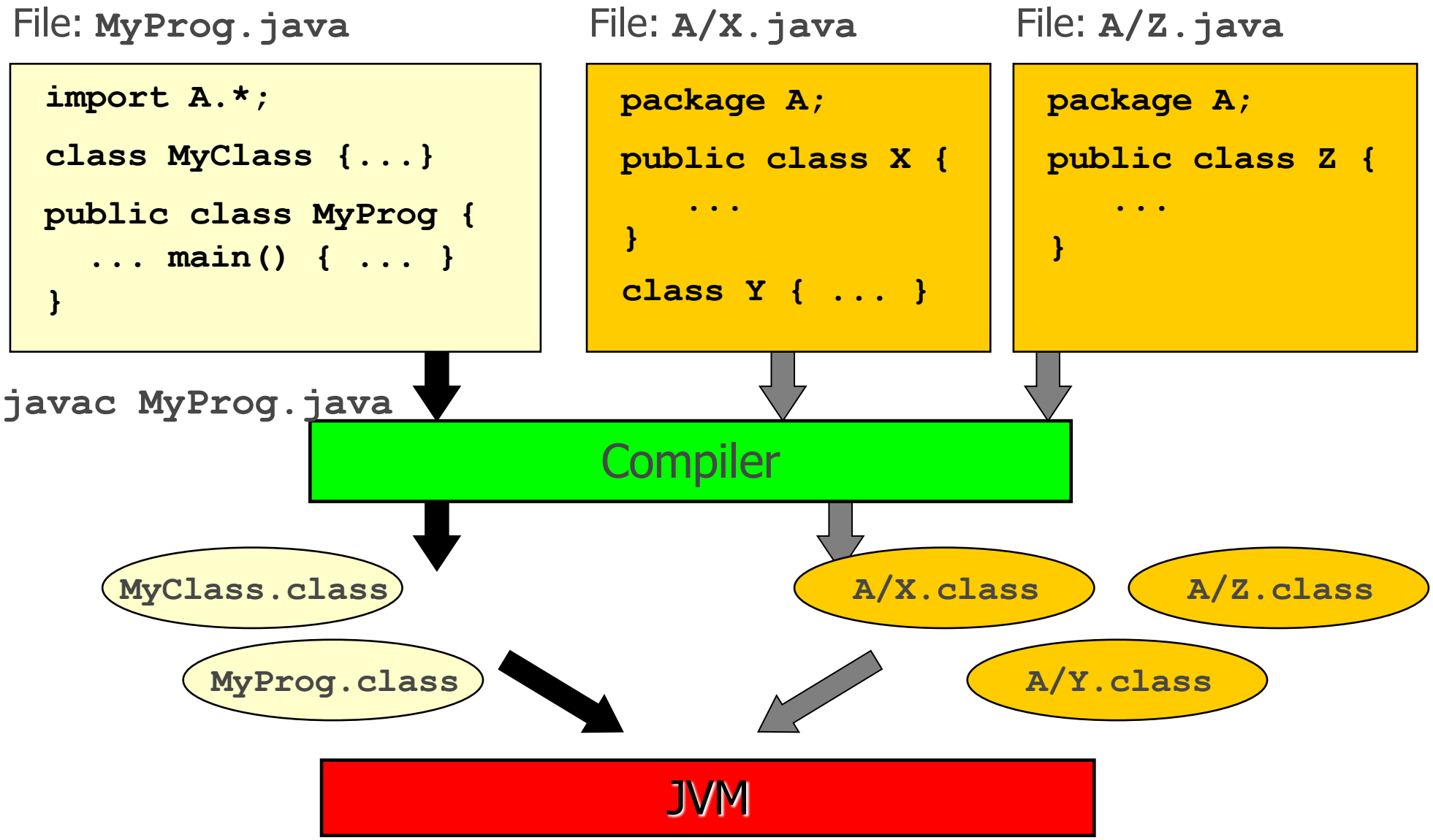
- **Portable; platform independent**
- **Object-oriented**
- **Dynamic**
- **Robust**
- **Secure**
- **”Simple”**

Portability



- Portability is also facilitated through the language specification, e.g., representation of data types.

Organization of Java Programs



Dynamic and distributed

- Java applications may be comprised of different components, which may be distributed over the internet.

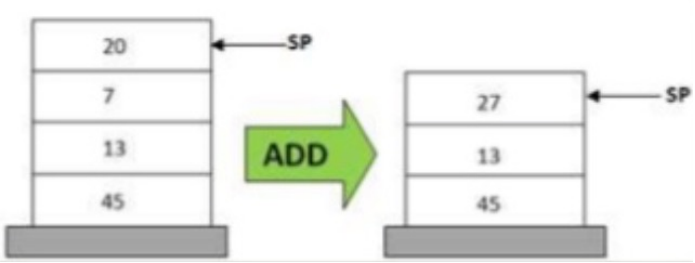
- During runtime of program, the JVM can load additional bytecode (class files) over a network.

- Remote Method Invocation (RMI) is Java's base mechanisms for developing distributed applications.

- RMI realizes high-level communication between distributed objects, hiding low-level communication details between distributed Java components.

Java Virtual Machine (JVM)

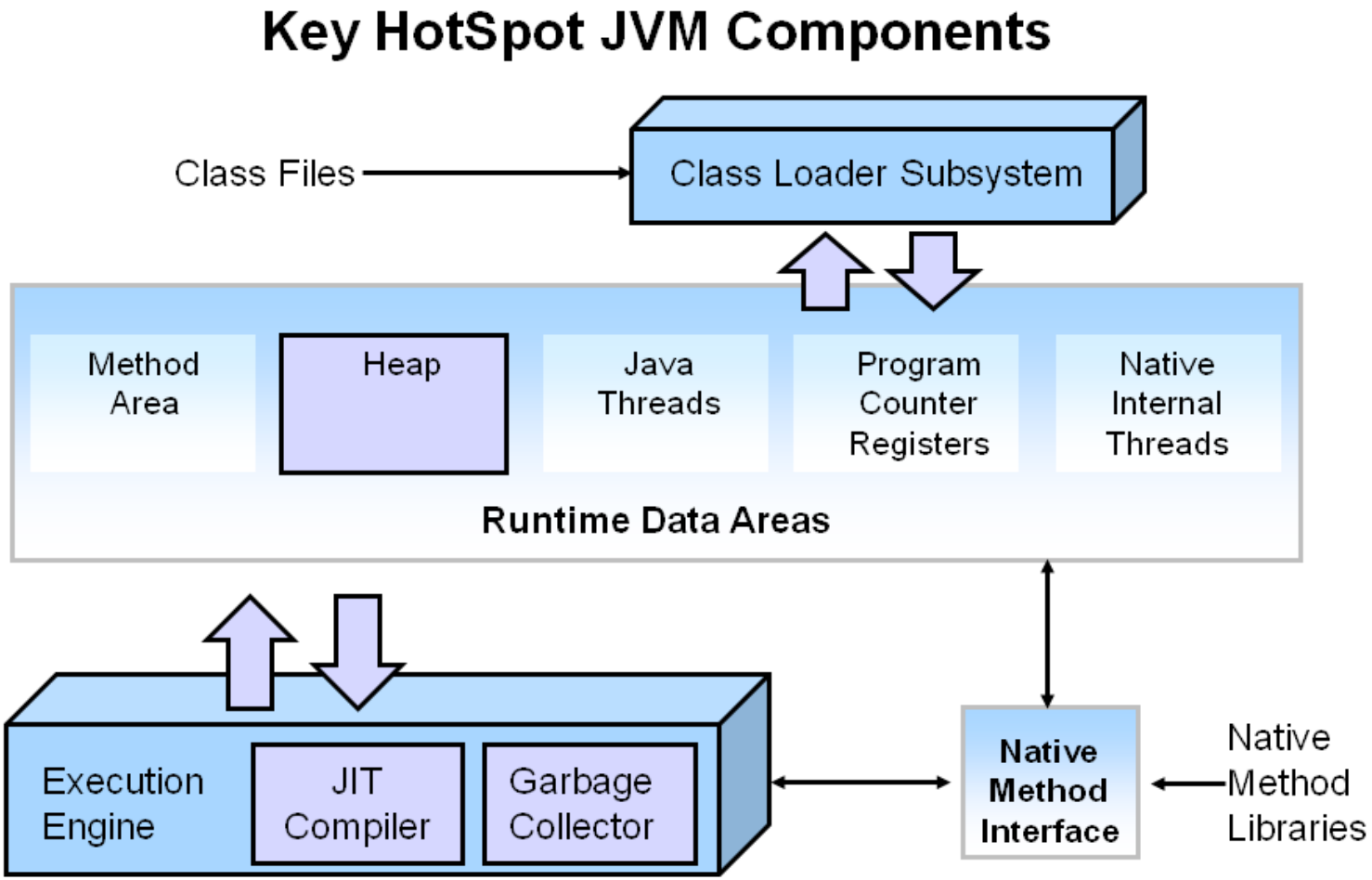
- Abstract Computing Machine
- Stack-based (LIFO) Virtual Machine



Lines	Stack-based VM Pseudo Code
0	POP 20
1	POP 7
2	ADD 20, 7, result
3	PUSH result

- Dynamic loading of class files
- Linking and initialization of classes and interfaces
- Program execution (Java Bytecode Instructions)
- JIT (just-in-time) compilation (machine code generation) and dynamic optimizations

JVM - Architecture



Robustness and Complexity

- Java is **simpler** and **more robust** than C or C++
  no pointers, no malloc, no goto, no header files …

- **Strongly and statically typed**
  no implicit method declarations, limited type conversions, only single inheritance)

- **Automatic Memory Management**
  garbage collection, runtime checks of array accesses, …

- **Exception Handling**

Strongly and Statically Typed

- The Java programming language is strongly and statically typed.

- The Java specification clearly distinguishes between the compile-time errors that can and must be detected at compile time, and those that occur at runtime (runtime errors).

- Compile time normally consists of translating programs into a machine-independent byte code representation.

- Runtime activities include loading and linking of the classes needed to execute a program, optional machine code generation and dynamic optimization of the program, and actual program execution.

Memory Management

C++ is notorious for fostering memory management bugs, including:

- Allocating insufficient memory for the intended contents;
- Accessing arrays with indexes that are out of bounds;
- Using stack-allocated structures beyond their lifetimes;
- Using heap-allocated structures after freeing them;
- Neglecting to free heap-allocated objects when they are no longer required;
- Excessive copying by copy constructors;
- Unexpected sharing due to insufficient copying by copy constructors;

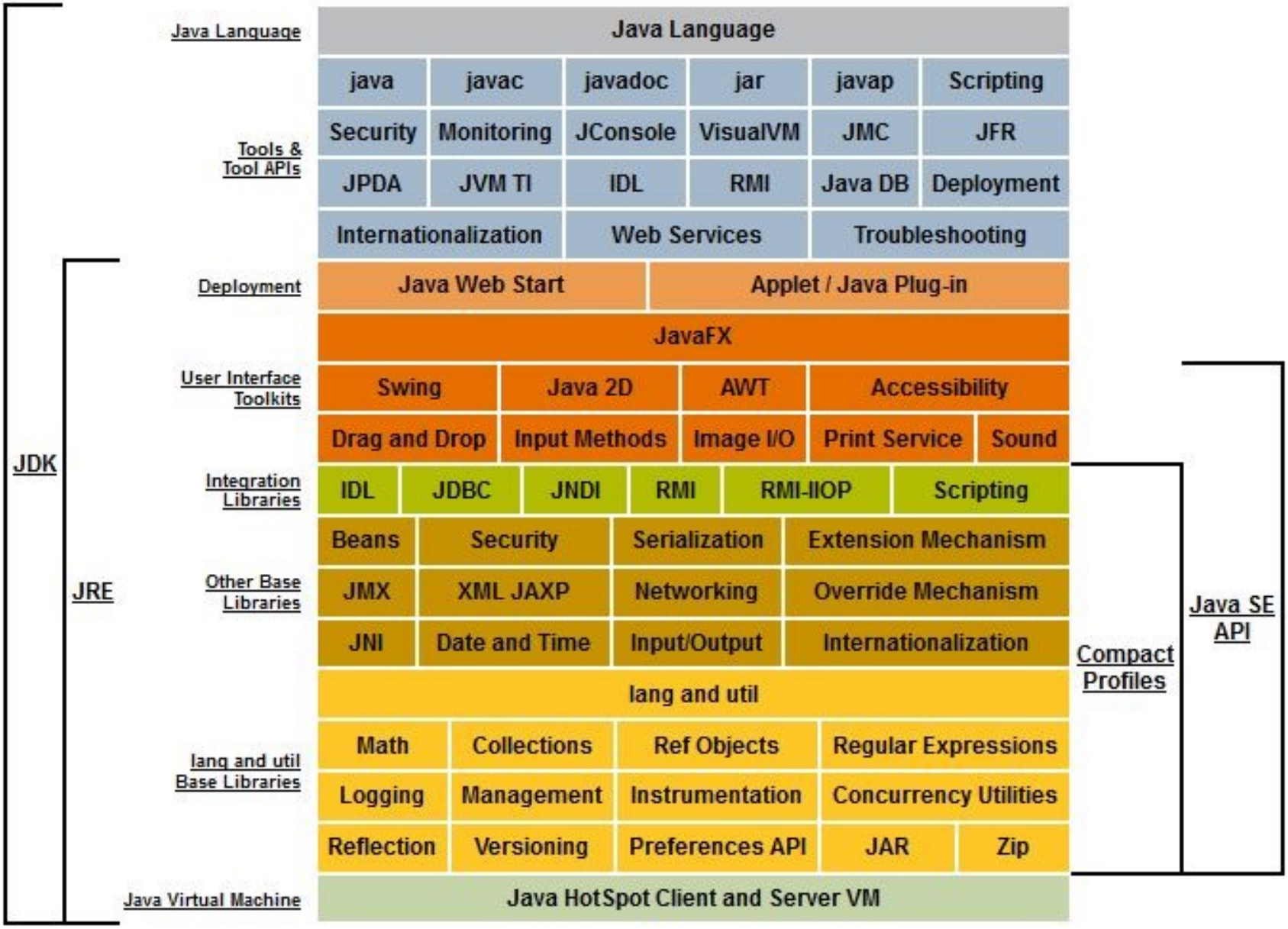
M. Ellis and B. Stroustrup. The Annotated C++ Reference Manual, 1990.

”C programmers think memory management is too important to be left to the computer. Lisp programmers think memory management is too important to be left to the user.”

Security

- **Language**
  - no pointers, runtime checks of array accesses, **...**
- **Byte-Code Verification**
  - for untrusted code bytecode-verification can be enabled
- **Digital Signatures, Digital Certificates** (X.509)
  - Class files can be digitally signed to identify their providers.
- **Sandbox**
  - Untrusted code can be executed in a sandbox with limited permissions.
- **Security Policies**
  - To control permissions of individual classes.
- **Exception Handling**

Java Standard Edition - API



Java API Documentation

http://docs.oracle.com/javase/8/docs/api/

HTML documentation of all classes and interfaces