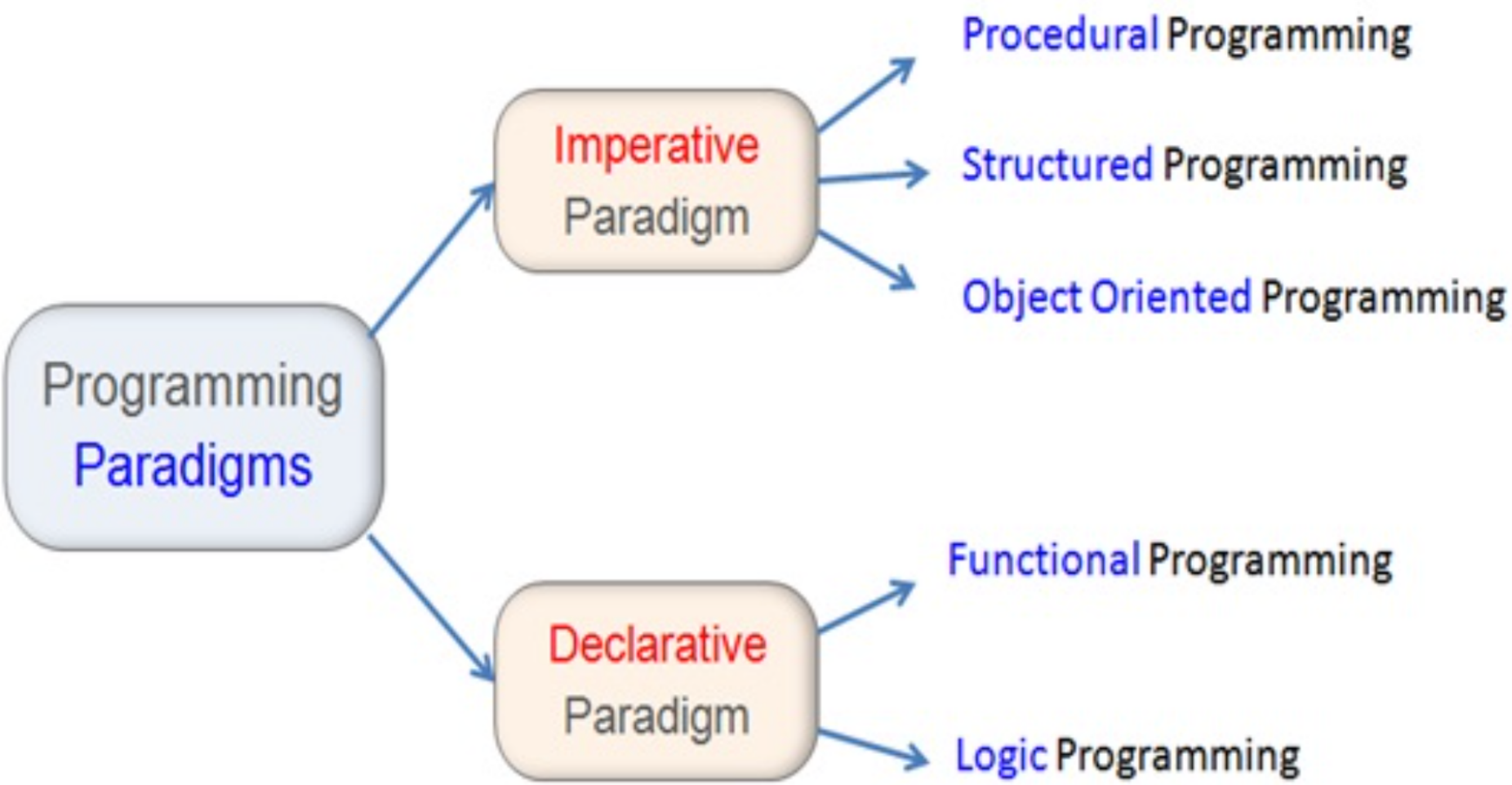
JAVA NEW FEATURES

Java Architecture and New Features

- Java Release Cycles
- Overview of JDK/JRE/JVM
- JVM Architecture and Internals
- Java 8 Features Recap
- Java 9 Features
- Java 10 Features
- Java 11 Features
- Java 12 - 16 Features
- Java 17 Features

Intro to Programming Language Paradigms

Programming paradigms are a way to classify programming languages based on their features
Imperative Paradigm - programmer instructs the machine how to change its state
Declarative Paradigm - programmer declares properties of the desired result, but not how to compute it



What is Java and it’s Background?

Java is a high-level object-oriented programming language with platform independent deployment.

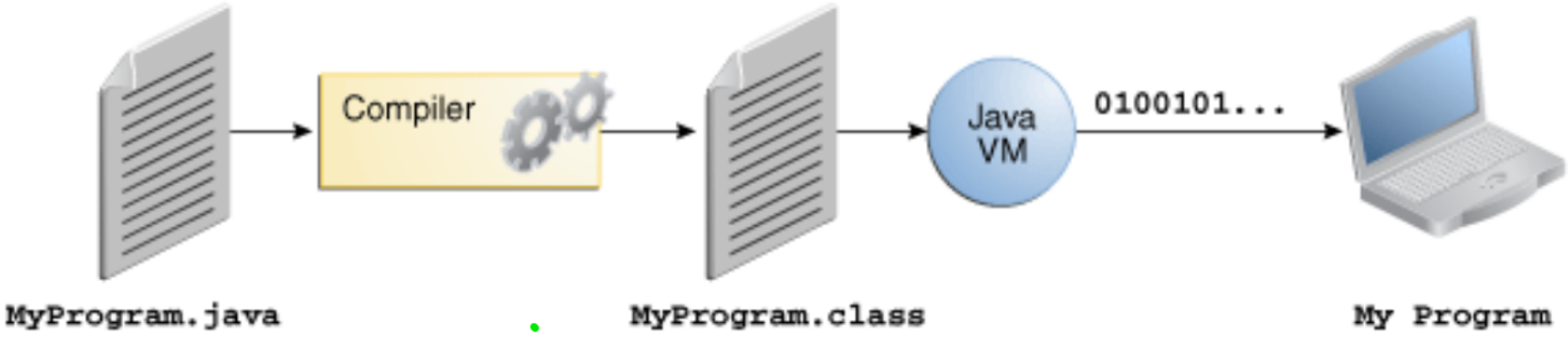
- Project started on 1991 by Sun Microsystems
- Developed by James Gosling with support from Mike Sheridan, Patrick Naughton
- v1.0 released on 1996
- JVM become open source on 2006/07 under FOSS (Free & Open Source Software)
- Oracle acquired Sun Microsystems and become owner of Java on 2009/10
- Latest version 23 and LTS versions are 8, 11, 17 and 21

Java Design Goals

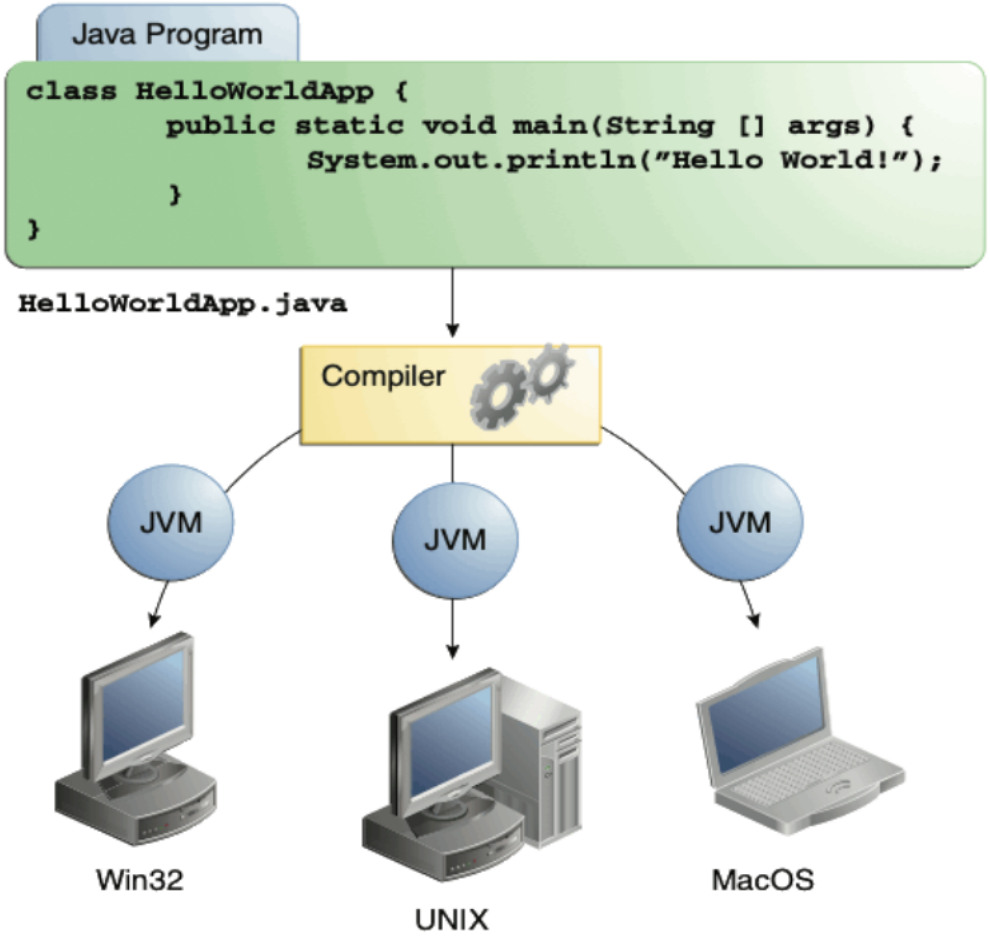
- simple, object oriented, familiar
- robust and secure
- architectural neutral and portable
- high performance (JIT)
- interpreted, threaded and dynamic

Java Characteristics / Features

- Simple
- Object oriented
- Distributed
- Multithreaded
- Dynamic
- Architecture neutral
- Portable
- High performance
- Robust
- Secure



Java is Platform Independent



Java Release History

- v1.0 -> 1996
- v1.1 -> 1997
- v1.2 -> 1998 => J2SE, J2EE, J2ME
- v1.3 -> 2000
- v1.4 -> 2002
- v5.0 -> 2004 => JSE, JEE, JME
- v6.0 -> 2006
- v7.0 -> 2011
- v8.0 -> 2014 (LTS) => OOP + FP (Lambda Expr + Stream API)
- v9.0 -> 2017
- v10 -> 2018(Mar)
- v11 -> 2018(Sep) (LTS)
- v12 -> 2019(Mar)
- v13 -> 2019(Sep)
- v14 -> 2020(Mar)
- v15 -> 2020(Sep)
- v16 -> 2021(Mar)
- v17 -> 2021(Sep) (LTS)
- v18 -> 2022(Mar)
- v19 -> 2022(Sep)
- v20 -> 2023 (Mar)
- v21 -> 2023 (Sep)
- v22 -> 2024 (Mar)
- v23 -> 2024 (Sep)

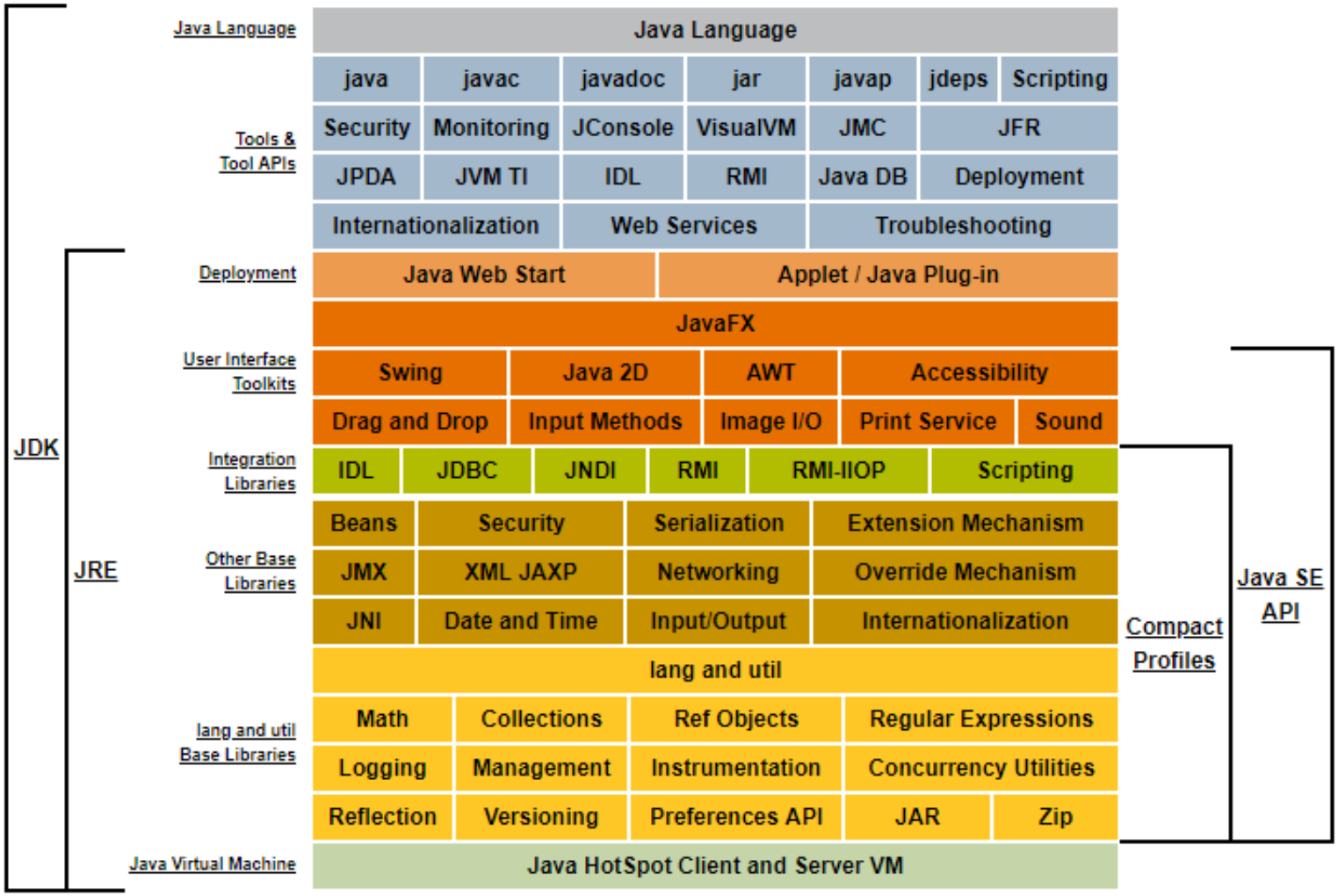
Java Flavors

- Java SE (Standard Edition)
- Java EE (Enterprise Edition) / Jakarta EE - Servlet, JSP, EJB, JAX-RS, etc..
- Java ME (Micro Edition)

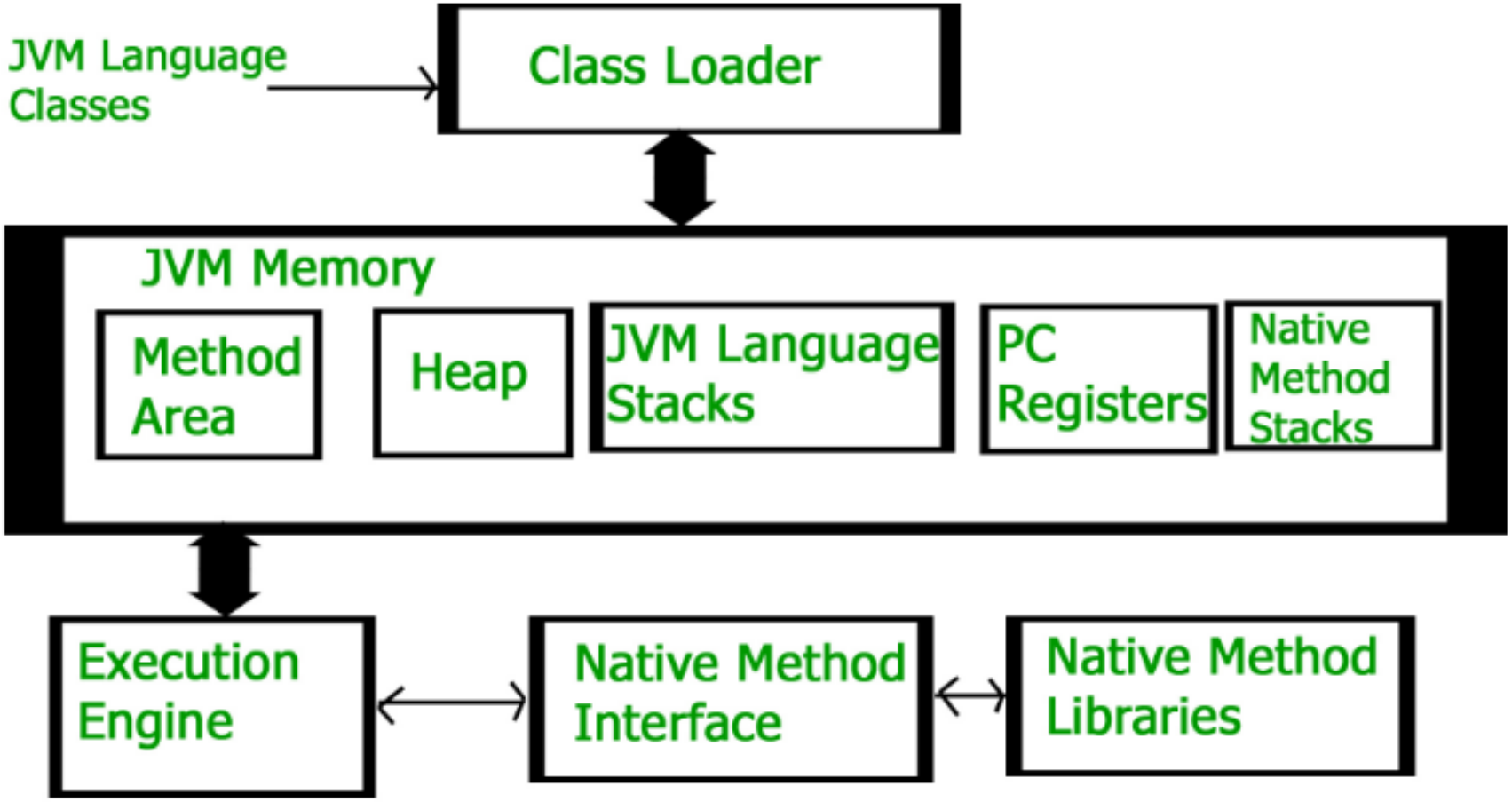
Java Benefits

- Get started quickly
- Write less code
- Write better code
- Develop programs more quickly
- Avoid platform dependencies
- Write once, run anywhere (WORA)
- Distribute software more easily

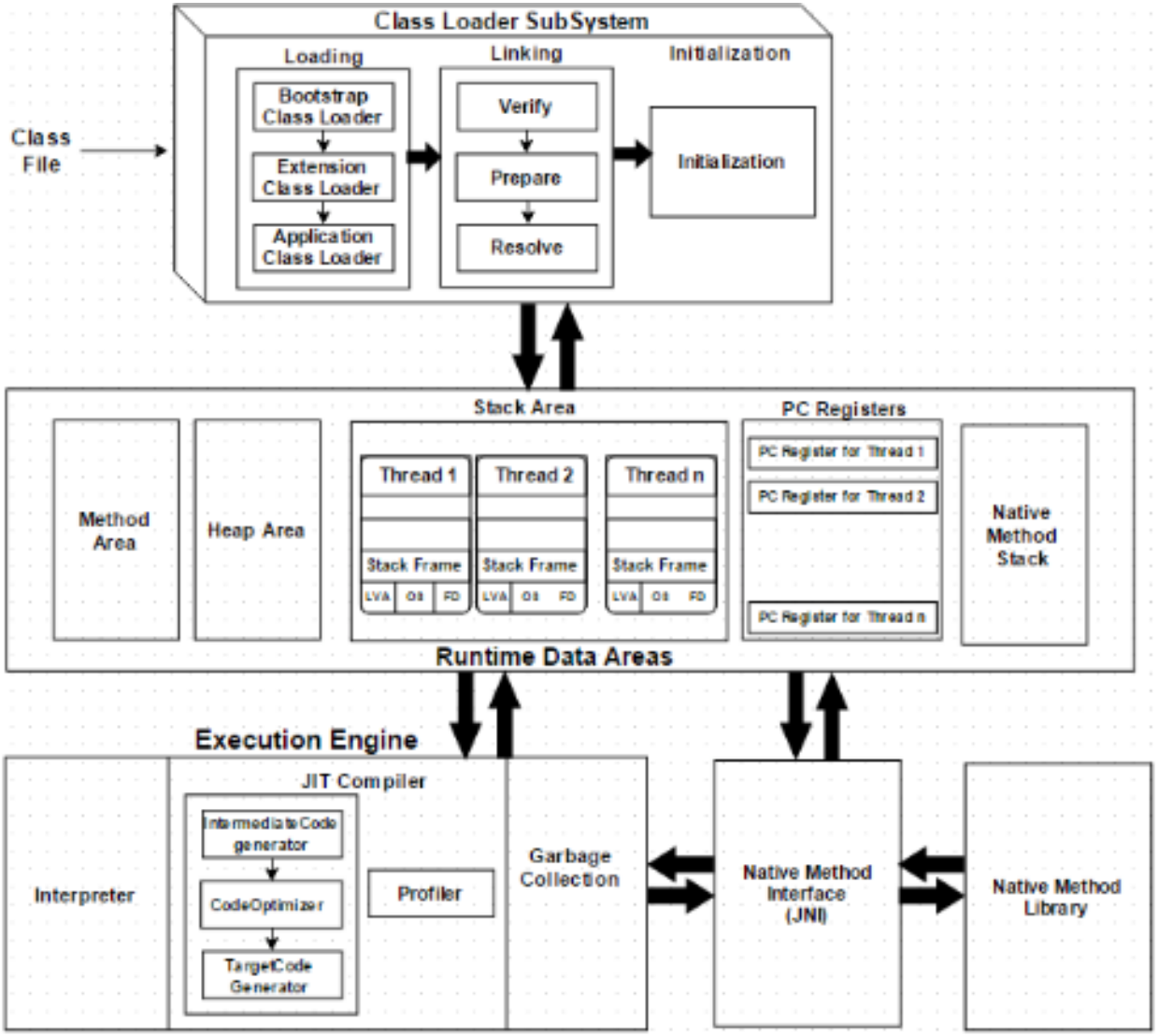
Java Conceptual Model (JVM/JRE/JDK)



JVM Architecture



JVM Architecture (detailed)



JVM Components – Class Loader Subsystem

- **Loading** - Classes will be loaded by this component
  - Boot Strap – Loads classes from the bootstrap classpath
  - Extension – Loads classes which are inside the ext folder
  - Application – Loads from Application Level Classpath, Environment Variable etc
- **Linking**
  - Verify - Bytecode verifier will verify whether the generated bytecode is proper or not
  - Prepare - For all static variables memory will be allocated and assigned with default values
  - Resolve - All symbolic memory references are replaced with the original references from Method Area
- **Initialization**
  - All static variables will be assigned with the original values, and the static block will be executed

JVM Components – Runtime Data Area

- Method Area - All the class level data will be stored here, including static variables
- Heap Area - All the Objects and their corresponding instance variables and arrays will be stored here
- Stack Area - For every thread, a separate runtime stack will be created.
  All local variables will be created in the stack memory.
- PC Registers - Each thread will have separate PC Registers, to hold the address of current executing
  instruction once the instruction is executed the PC register will be updated with the next instruction
- Native Method Stacks - Native Method Stack holds native method information.
  For every thread, a separate native method stack will be created.

JVM Components – Execution Engine

- Interpreter
- JIT Compiler
  - Intermediate Code Generator
  - Code Optimizer
  - Target Code Generator
  - Profiler
- Garbage Collectors
- Java Native Interface
- Native Method Libraries

JVM Internals - Memory Management

- Memory Spaces
  - Heap - Primary storage of the Java program class instances and arrays
    - Young Generation [Eden Space, Survivor Space]
    - Old Generation
  - PermGen/Metaspace - Primary storage for the Java class metadata
  - Native Heap - native memory storage for the threads, stack, code cache including objects such as MMAP files and third party native libraries

JVM Internals – Garbage Collectors

- Serial Garbage Collector - Single threaded. Freezes all app threads during GC
- Parallel Garbage Collector - Multi threaded. Freezes all app threads during GC
- Concurrent Mark Sweep - Multi threaded with shorter GC pauses
- G1 Garbage Collector - Divides heap space into many regions and GCs region have more garbage

Java 8 Features

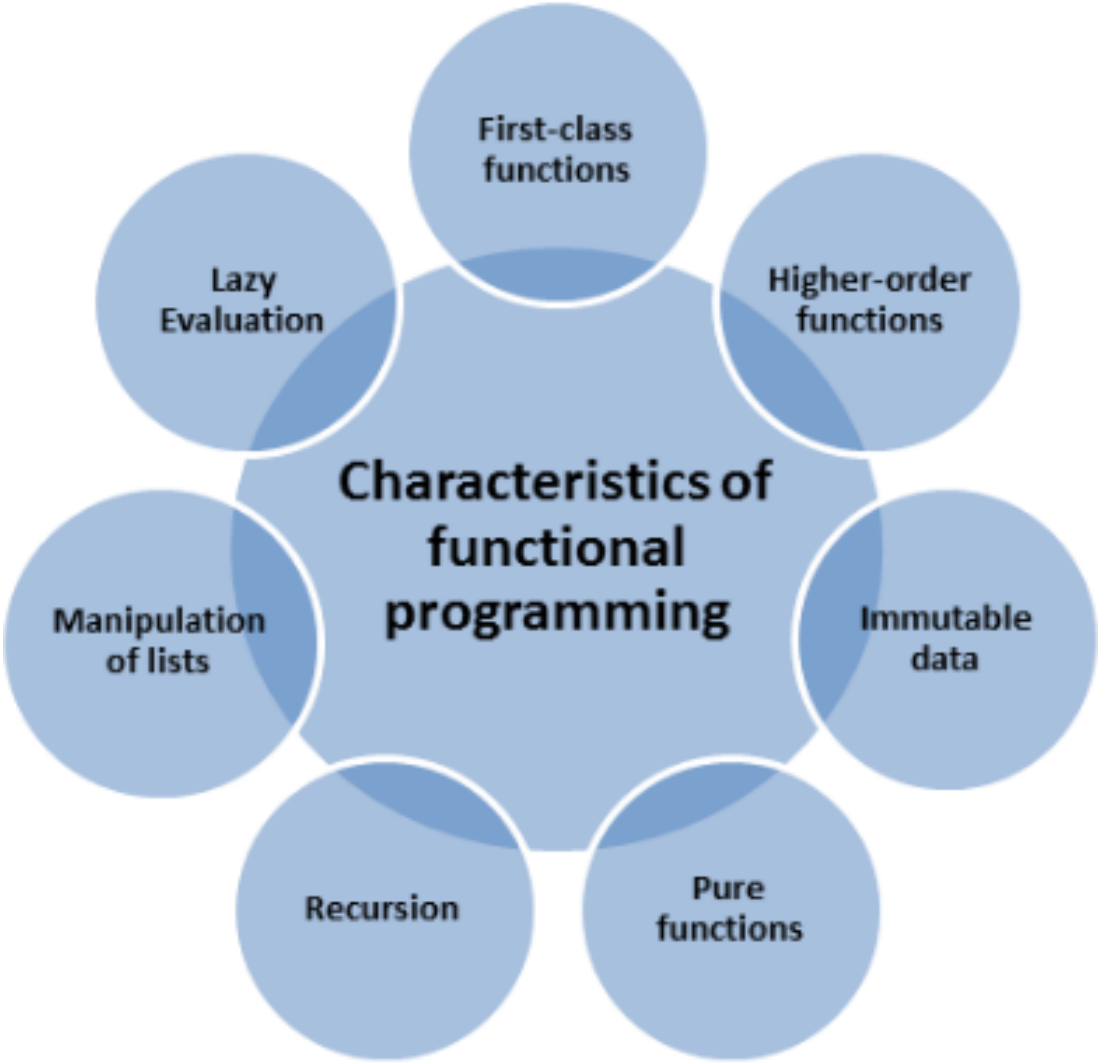
- Fundamentals of Functional Programming
- Lambda Expressions
- Functional Interfaces
- Method References
- Stream API - foreach, map, filter, parallel processing, collectors, etc.
- Default Methods
- Optional
- New DateTime package

Functional Programming

Functional programming is just a style, is a programming paradigm that treats computation as the evaluation of functions and avoids state and mutable data…

- *"Functions as primary building blocks"* (first-class functions)
- programming with *"immutable"* variables and assignments, no s
  - Programs work by returning values instead of modifying data

Functional Programming Characteristics



Functional vs Object Oriented Programming

Functional Programming	OOP
Uses Immutable data.	Uses Mutable data.
Follows Declarative Programming Model.	Follows Imperative Programming Model.
Focus is on: "What you are doing"	Focus is on "How you are doing"
Supports Parallel Programming	Not suitable for Parallel Programming
Its functions have no-side effects	Its methods can produce serious side effects.
Flow Control is done using function calls & function calls with recursion	Flow control is done using loops and conditional statements.
It uses "Recursion" concept to iterate Collection Data.	It uses "Loop" concept to iterate Collection Data. For example: For-each loop in Java
Execution order of statements is not so important.	Execution order of statements is very important.
Supports both "Abstraction over Data" and "Abstraction over Behavior".	Supports only "Abstraction over Data".

Lambda Expression

parameter -> expression body

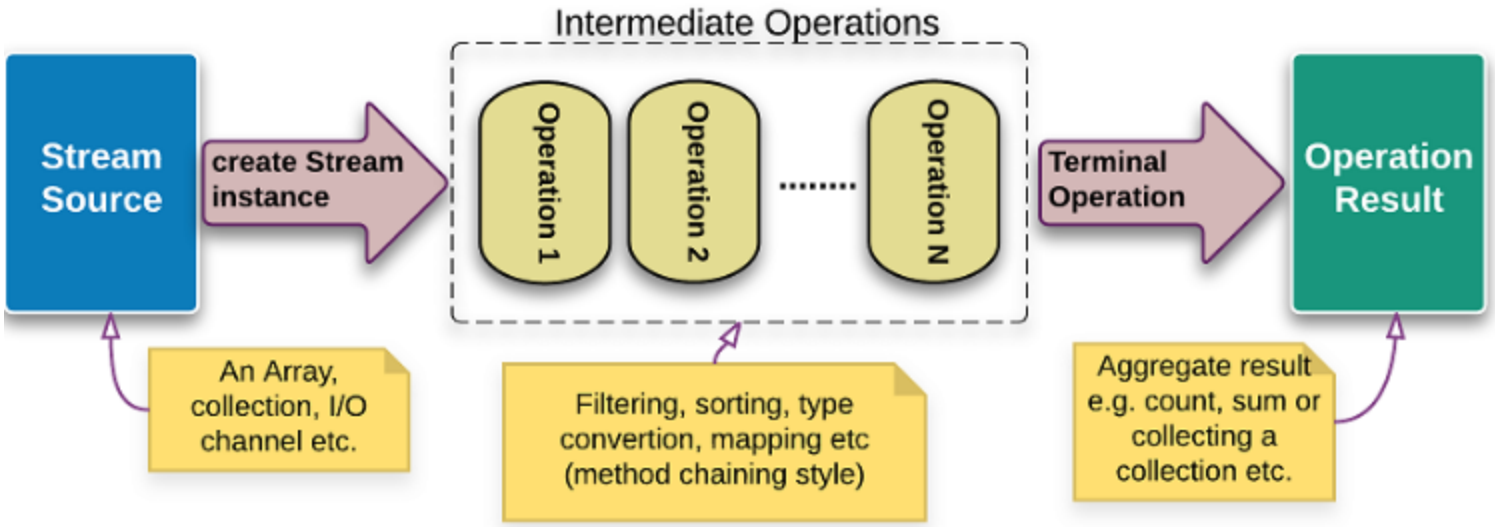
- No arguments:
```
() -> System.out.println("Hello")
```
- One argument:
```
s -> System.out.println(s)
```
- Two arguments:
```
(x, y) -> x + y
```
- With explicit argument types:
```
(Integer x, Integer y) -> x + y
```
- Multiple statements:
```
(x, y) -> {
    System.out.println(x);
    System.out.println(y);
    return (x + y);
}
```

Functional Interfaces

- An interface which performs single task (have single method) is called functional interface.

- Standard Functional Interfaces
  - Predicate - takes an argument and returns boolean value
  - Consumer - takes an argument and process the logic, no return value
  - Supplier - takes no argument, returns a value
  - Function - takes an argument and returns a value

Java Streams Overview



Java 9 Features

- JShell - REPL (Read Evaluate Process Loop)
- Module System - Project Jigsaw
- Reactive Streams - Flow API
- Private Methods in Interfaces
- Try with Resources Enhancements
- Factory Methods for Immutable List, Set and Map - of()
- Stream API Enhancements - takeWhile(), dropWhile()
- Optional Class Improvements - stream()
- Process API Improvements
- CompletableFuture API Improvements
- JVM Enhancements / G1 GC set as default GC

Java 10 Features

- Local Variable Type Inference
- Collection Enhancements - copyOf() factory method
- Stream API Enhancements - Collectors.unmodifiableList|Set|Map()
- Optional Enhancements - orElseThrow()
- Container Awareness
- Time-Based Release Versioning
- Application Class-Data Sharing
- Root Certificates
- Garbage Collector Interface / G1 GC Enhancements
- Remove the Native-Header Generation Tool (javah)

Java 11 Features

- Compile Free Launch
- var in Lambda
- Optional isEmpty support
- Not Predicate
- New methods in String - lines, isBlank, strip, stripTrailing, stringLeading, repeat
- New HttpClient
- New Files and Path methods - writeString(), readString()
- Nest Based Access
- Dynamic Class File Constants
- No-Op Garbage Collector
- Flight Recorder

Java 12 Features

- String class new method - intent(), transform()
- New File mismatch method - mismatch()
- Stream Enhancement - Collectors.teeing(ds1, ds2, mergr)
- CompactNumberFormat
- Unicode 11 Support
- JVM Constants API
- Micro Benchmark Suite
- Default CDS Archive - Reduce Start Time and Memory Footprint
- JVM Enhancements / G1 GC Enhancements

Java 13 Features

- NIO - ByteBuffer Enhancements - bulk get/put
- Legacy Socket API Reimplemented
- FileSystems.newFileSystem() Method
- Unicode 12 Support
- Dynamic CDS Archive
- JVM Enhancements

Java 14 Features

- Switch Expression
- Helpful NullPointerException
- JFR Event Streaming
- JVM Enhancements
- Deprecated / Removed Features
  - CMS Garbage Collector
  - ParallelScavenge + SerialOld GC Combination
  - Solarix and SPARC ports

Java 15 Features

- Text Blocks
- Hidden Classes
- ZGC Garbage Collector - low latency (~10 ms), non-generational
- Shenandoah GC - low latency (~10 ms), generational (17+ onwards)
- Deprecated / Removed Features
  - Javascript Nashhorn Engine

Java 16 Features

- Records
- Pattern Matching for instanceof
- Stream Enhancement - toList()
- Packaging Tool
- Strong Encapsulation by Default
- JVM Enhancements

Java 17 Features

- Sealed Classes
- Restore Always-Strict Floating-Point Semantics > strictfp
- Enhanced Pseudo-Random Number Generators
- New macOS Rendering Pipeline > OpenGL - Metal API
- macOS/AArch64 Port > Intel (AMD64) - M1/M2 (ARM64)
- Strongly Encapsulate JDK Internals > -illegal-access
- Context-Specific Deserialization Filters
- LTS Release Cycle reduced to two years
- Deprecation of the Applet API
- Removal of Experimental AOT and JIT Compiler

Thank You!