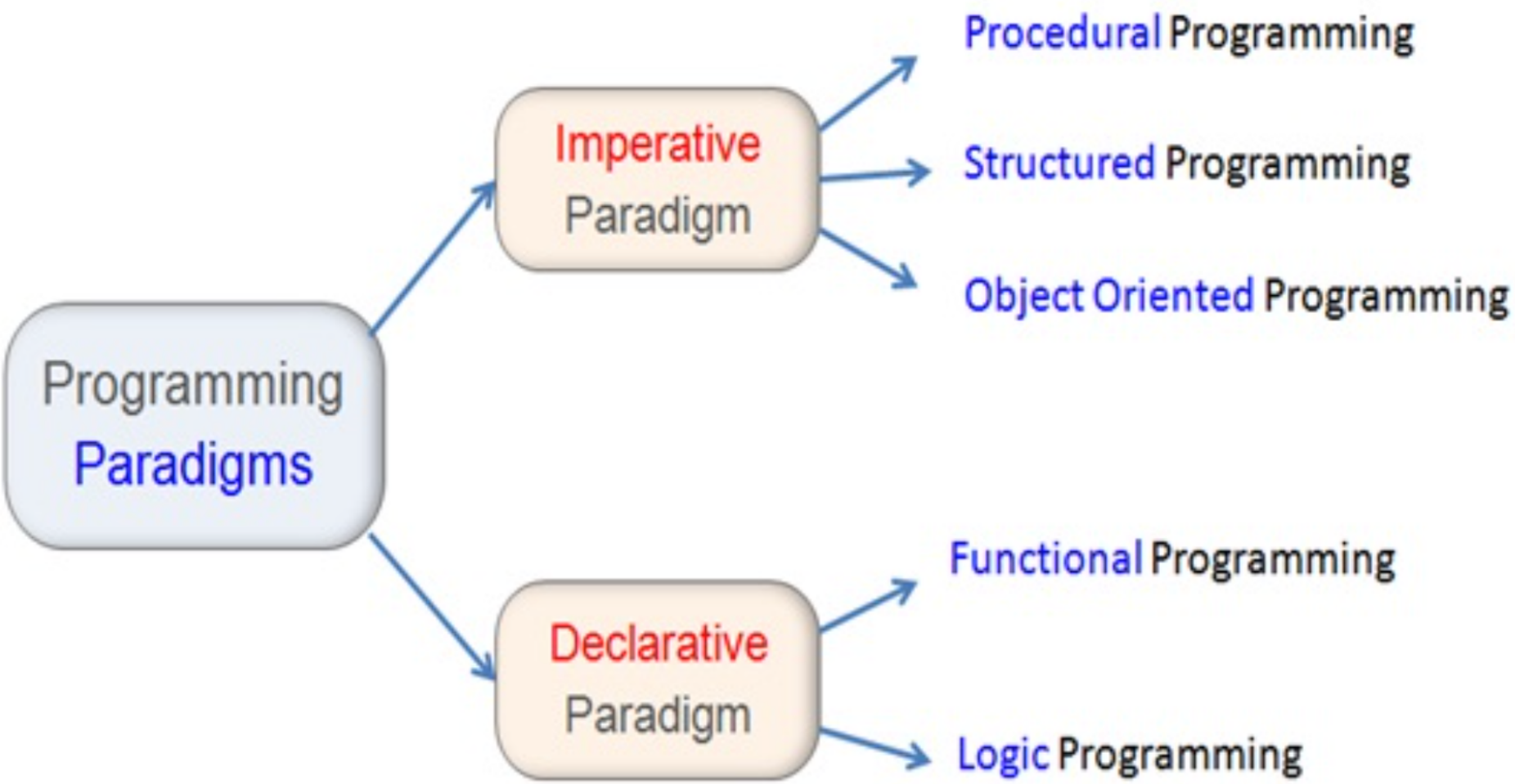
INTRODUCTION TO JAVA

Java programming fundamentals

- Introduction to Java
- Overview of JDK/JRE/JVM
- Java Language Constructs
- Object Oriented Programming with Java
- Exception Handling

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.

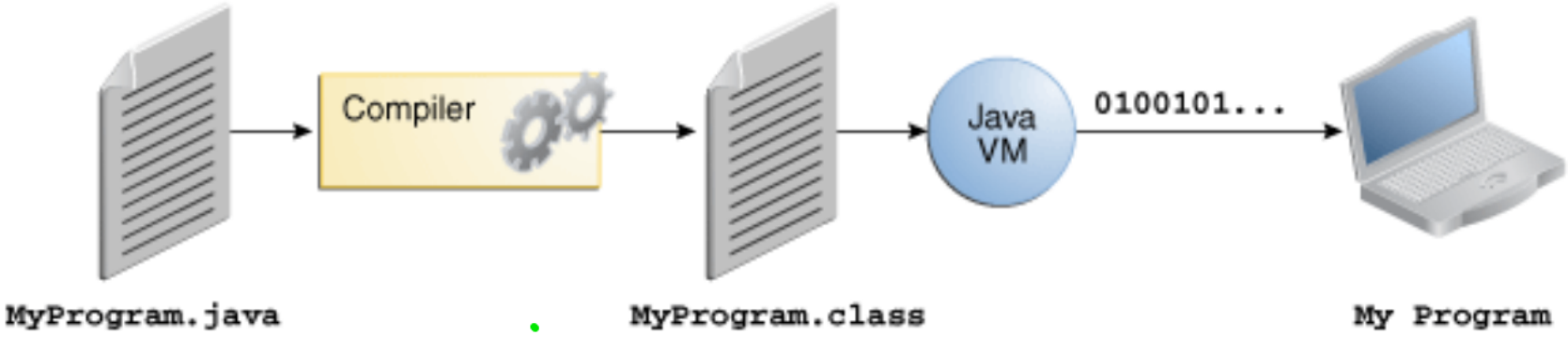
- Project started on 1991 by Sun Microsystems
- Developed by James Gosling with support from Mike Sheridan, Patrick Naughton
- v1.0 released on 1996
- JVM become open source on 2006/07 under FOSS (Free & Open Source Software)
- Oracle acquired Sun Microsystems and become owner of Java on 2009/10
- Latest version 20 and LTS versions are 8, 11 and 17

Java Design Goals

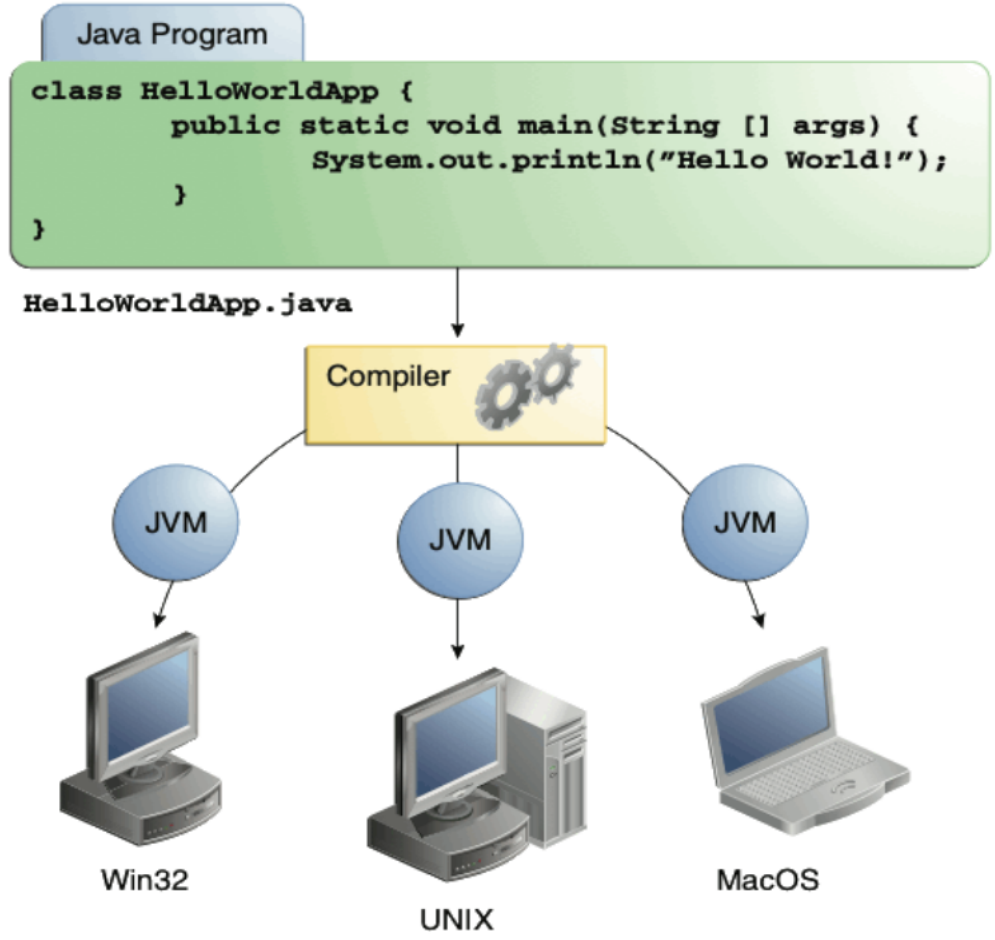
- simple, object oriented, familiar
- robust and secure
- architectural neutral and portable
- high performance (JIT)
- interpreted, threaded and dynamic

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)

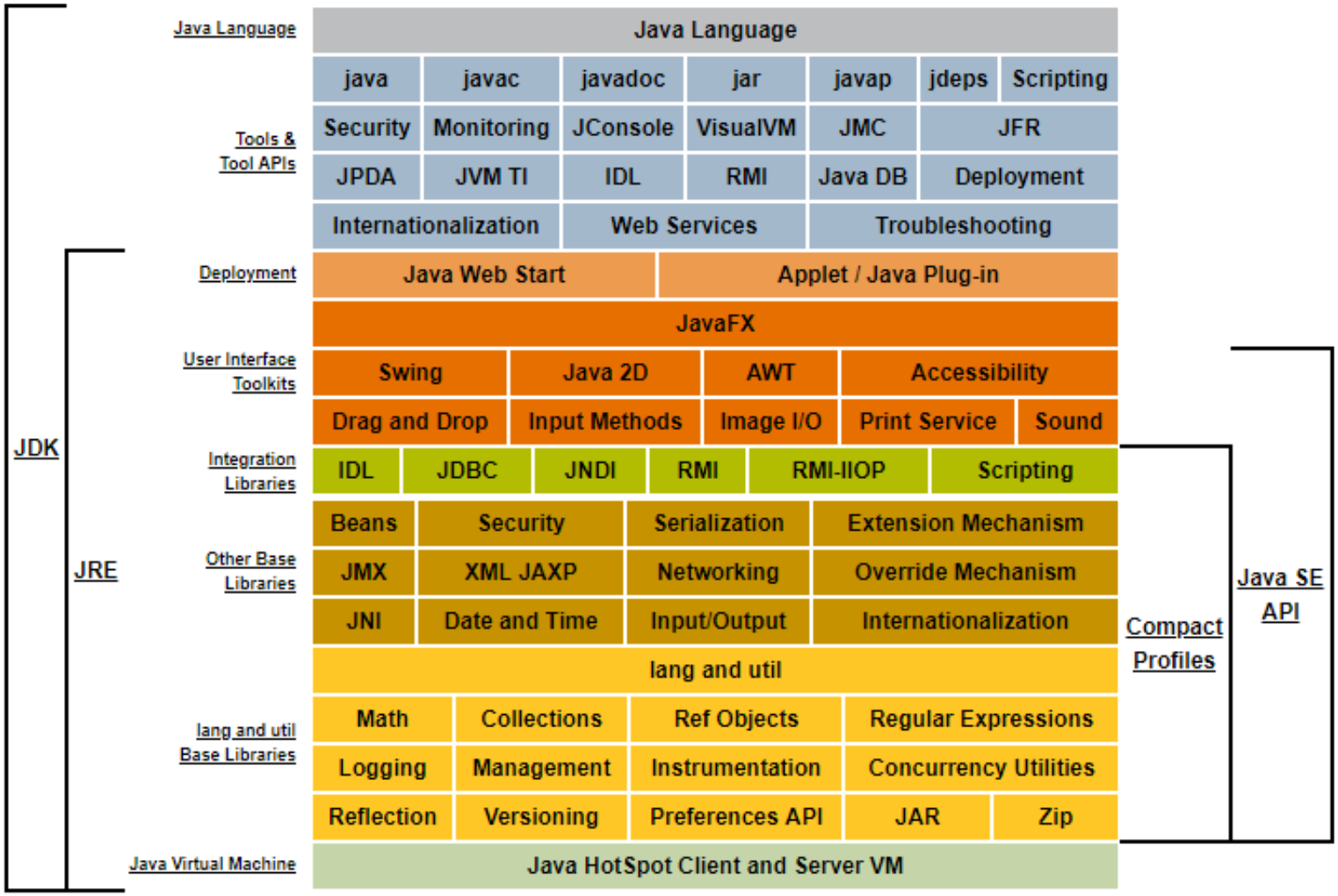
Java Flavors

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

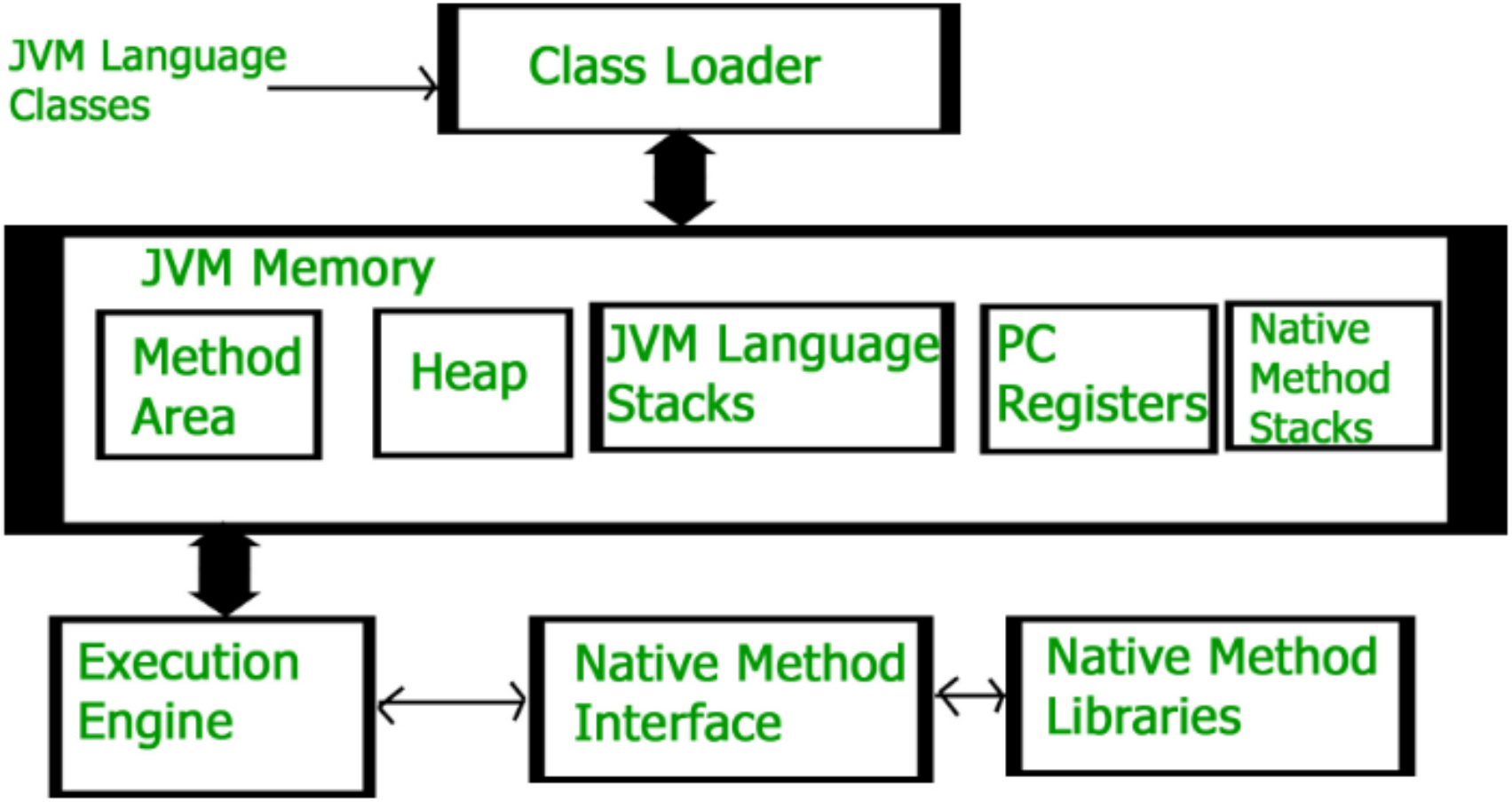
Java Benefits

- **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**

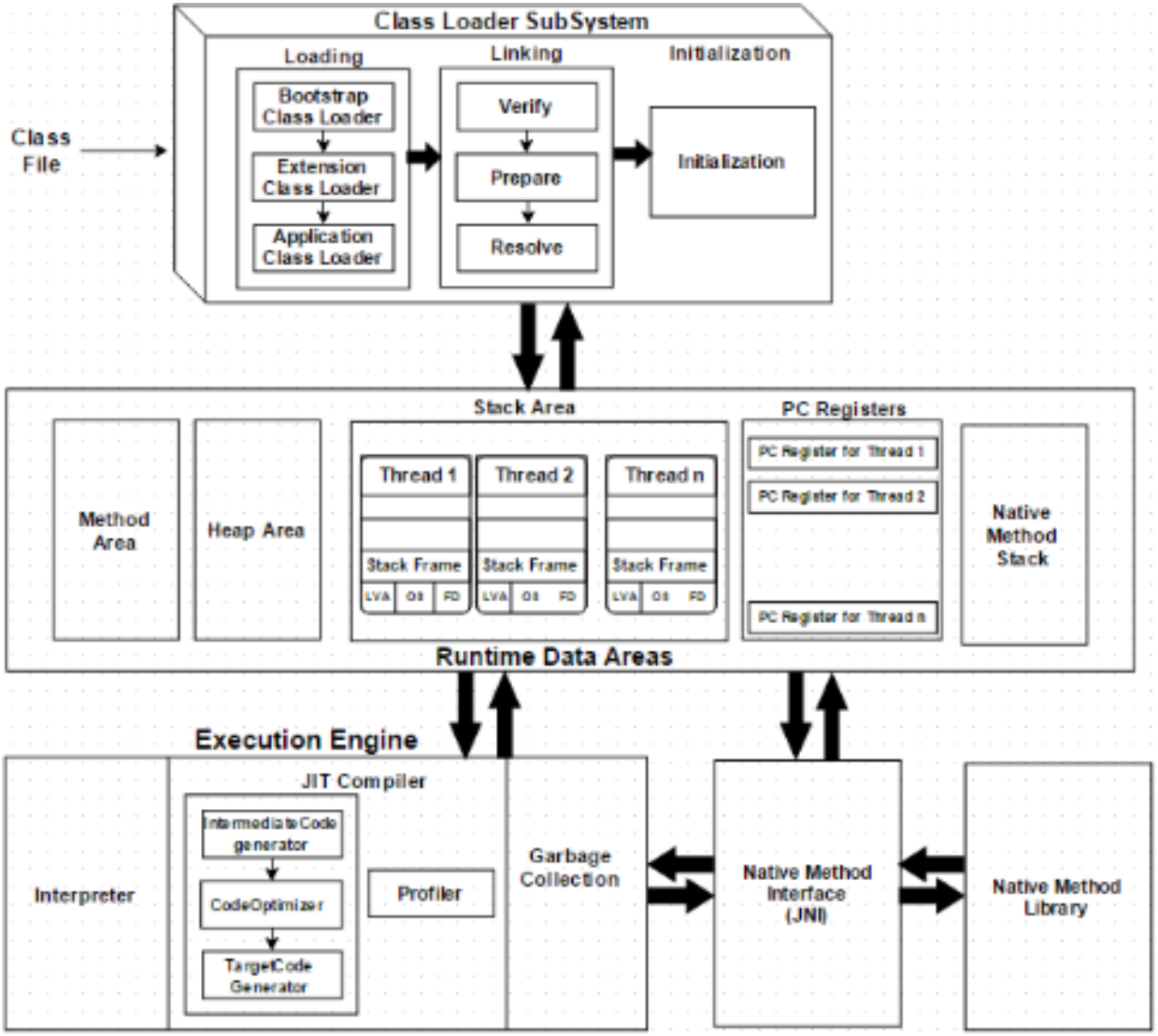
Java Conceptual Model (JVM/JRE/JDK)



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

JVM Internals – Hotspot

- Region of a computer program where a high proportion of executed instructions occur or where most time is spent during the program's execution
- **Client VM** - Tuned for quick loading. It makes use of interpretation.
- **Server VM** - Loads more slowly, putting more effort into producing highly optimized JIT compilations to yield higher performance
- **Tiered Compilation** - uses both the client and server compilers in tandem to provide faster startup time than the server compiler, but similar or better peak performance

Java Keywords

abstract	default	if	private	this
assert	do	implements	protected	throw
boolean	double	import	public	throws
break	else	instanceof	return	transient
byte	enum	int	short	try
case	extends	interface	static	void
catch	final	long	strictfp	volatile
char	finally	native	super	while
class	float	new	switch	
continue	for	package	synchronized	

Language Basic Constructs

- Data Types
- Variables
- Constants
- Operators
- Expressions, Statements, Blocks
- Control Flow Statements
- Loop Statements
- Branching Statements
- Naming Conventions
- Comments
- Arrays
- Strings

Object Oriented Programming and Related Concepts

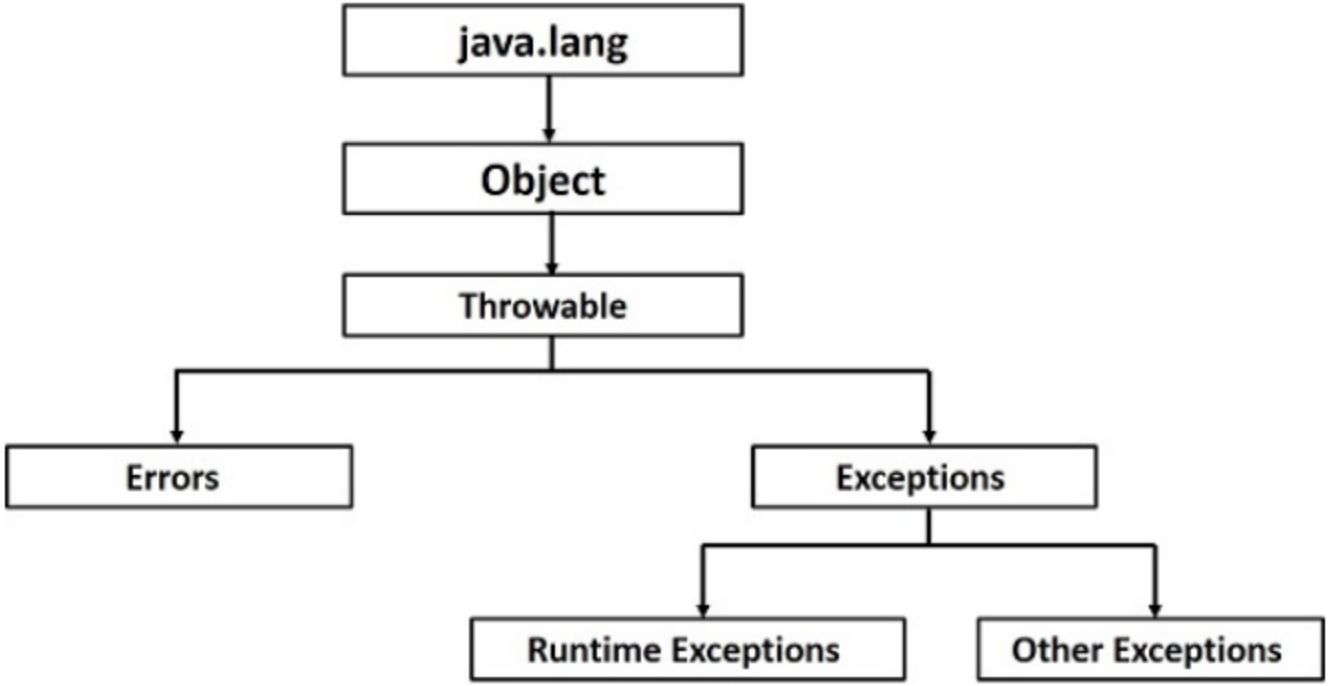
- Class
- Object
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

- Interface
- Package
- Wrapper Classes
- Object Class
- Methods
- Access Modifiers

Exception Handling

- Method call-stack and Exception
- Exception Hierarchy
- Exception vs Error
- Checked vs Unchecked Exception
- try…catch..finally block
- throws
- throw
- Custom Exception

Exception Hierarchy



Collection Framework and other features

- Java Collection Overview
- Generics Overview
- Reflection API
- Annotations
- Inner Classes

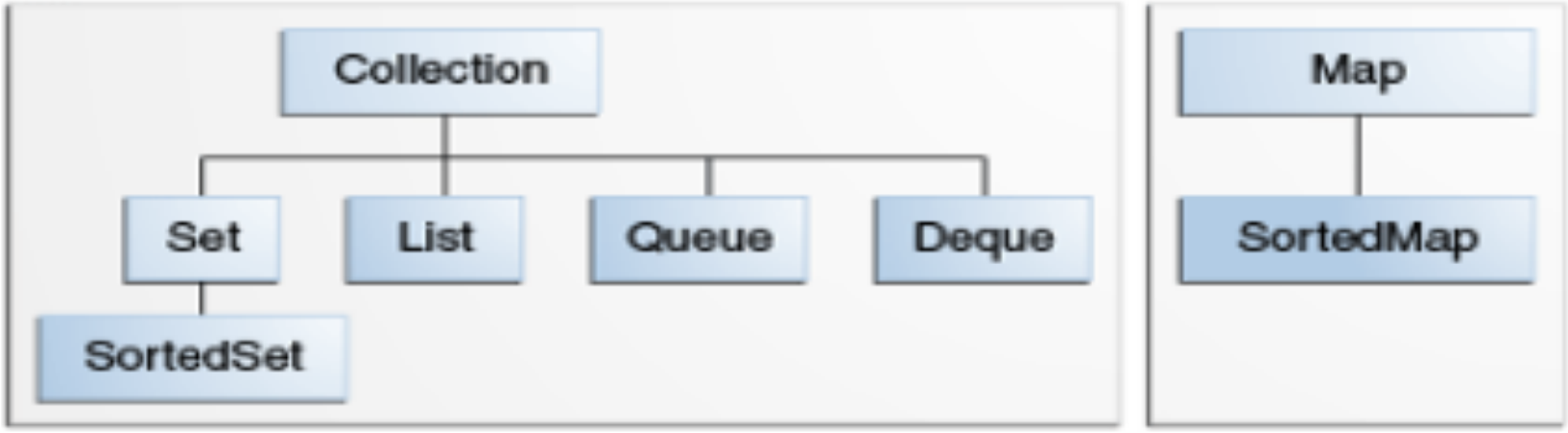
Collections Framework Overview

- A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve, manipulate, and communicate aggregate data
- A collections framework is a unified architecture for representing and manipulating collections. It consists of
  - Interfaces
  - Implementations
  - Algorithms

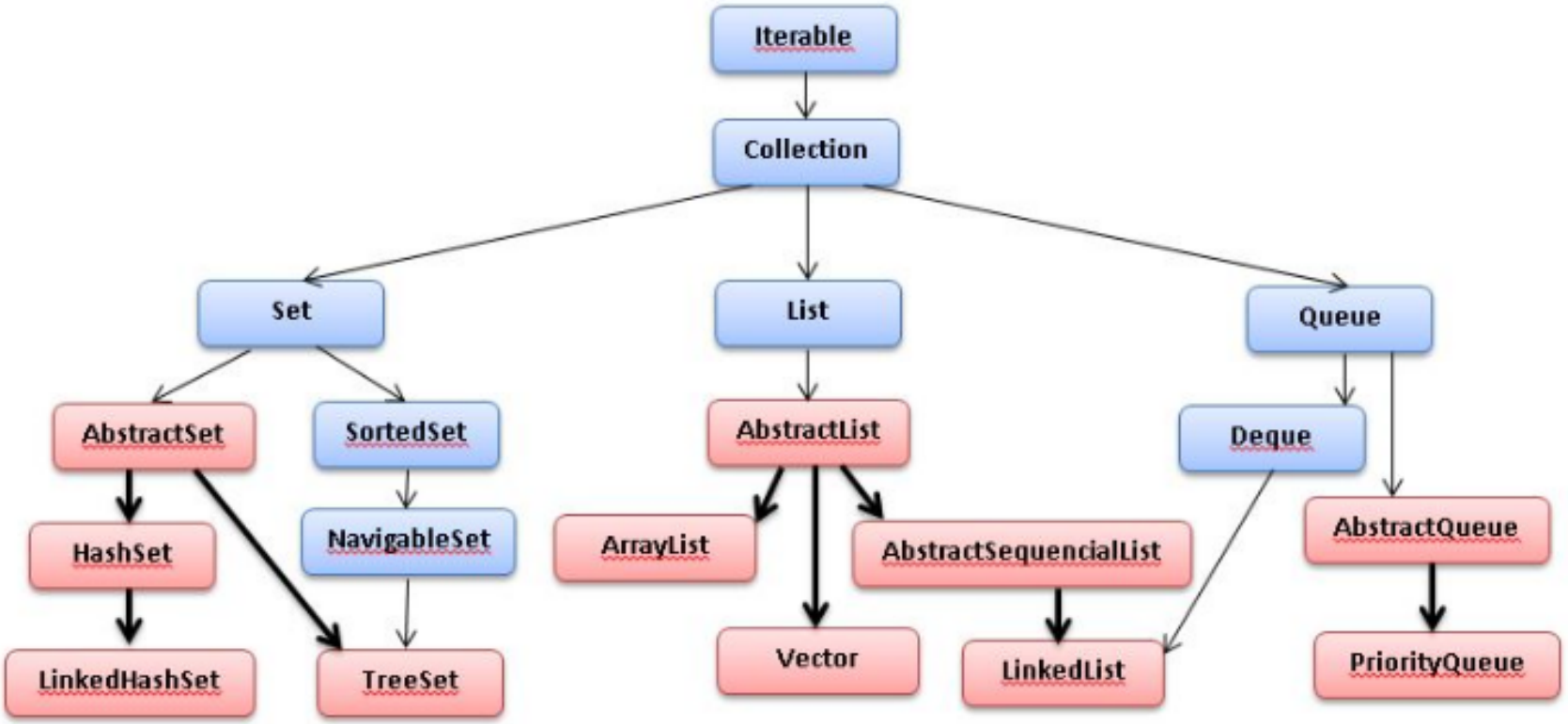
Collections Framework Benefits

- Reduces Programming Effort
- Increases Program Speed and Quality
- Allows interoperability among unrelated APIs
- Reduces effort to learn and to use new APIs
- Reduces effort to design new APIs
- Fosters software reuse

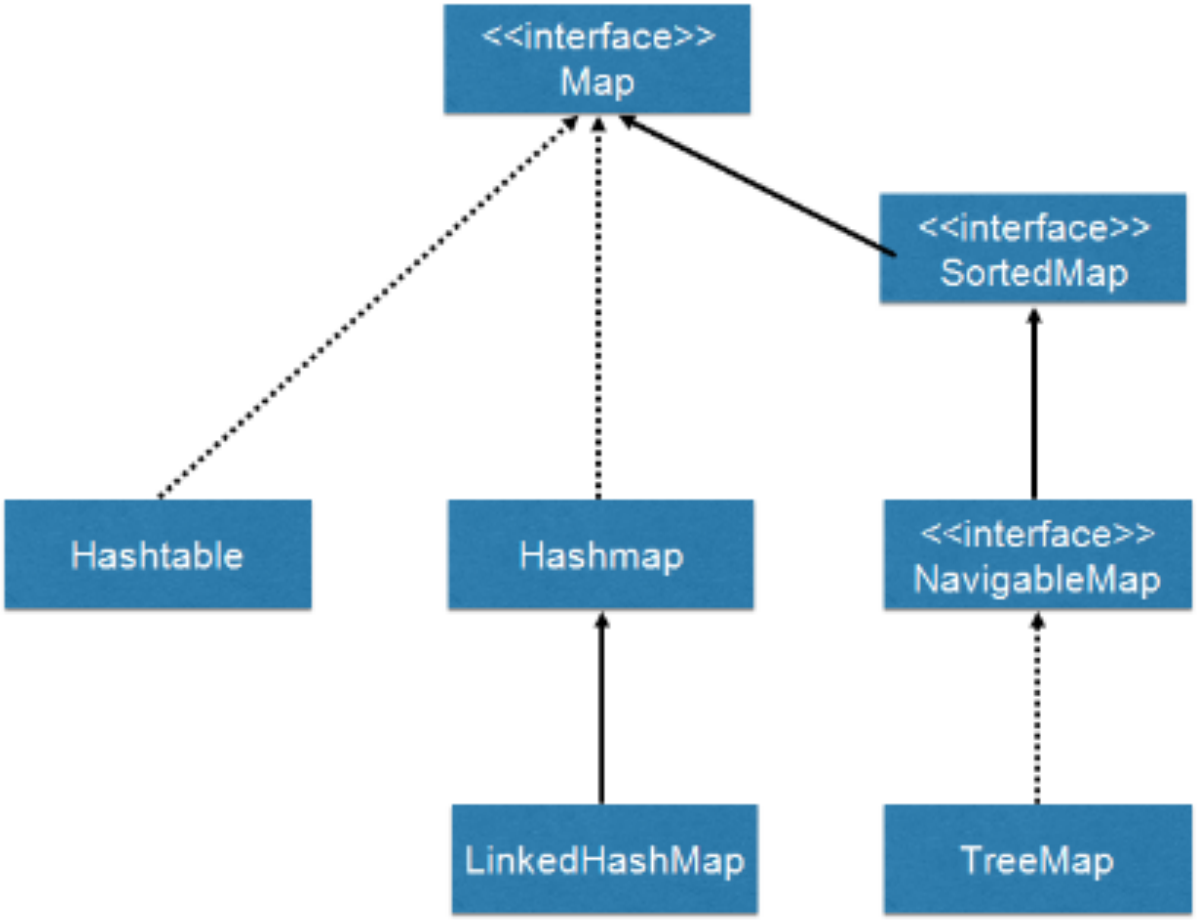
Collection Hierarchy (Interfaces)



Collection Hierarchy (Implementations)



Collection Hierarchy (contd.)



Generics

- Generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods.
- Much like the more familiar *formal parameters* used in method declarations, type parameters provide a way for you to re-use the same code with different inputs.
- The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types
- Benefits
  - Stronger type checks at compile time
  - Elimination of casts
  - Enabling programmers to implement generic algorithms

Generic Types

- A *generic type* is a generic class or interface that is parameterized over types.
- Example:

```
public class Box<T> {
      // T stands for "Type"
      private T t;
      public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

- Type Parameter Naming Convention
  - E - Element (used extensively by the Java Collections Framework)
  - K – Key
  - N – Number
  - T – Type
  - V – Value
  - S,U,V etc. - 2nd, 3rd, 4th types

Generic Concepts

- Generic Types
- Raw Types
- Bounded Type Parameters
- Type Inference
- Wildcards
  - Upper bounded wildcards — *e.g: ? extends Number*
  - Lower bounded wildcards — *e.g: ? super Integer*
  - *Unbounded* — *e.g: ?*
- Type Erasure

Reflection

- An API that represents ("reflects") the classes, interfaces, and objects in the current Java Virtual Machine.
- Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine
- **Use cases**
  - Extensibility Features
  - Class Browsers and Visual Development Environments
  - Debuggers and Test Tools
- **Limitations**
  - Performance Overhead
  - Security Restrictions
  - Exposure of Internals

Annotations

- Annotations, a form of metadata, provide data about a program that is not part of the program itself

- **Use cases**
  - Information for the compiler
  - Compile-time and deployment-time processing
  - Runtime Processing

Nested/Inner Classes

- A nested class is a member of its enclosing class.
- Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private.
- Static nested classes do not have access to other members of the enclosing class

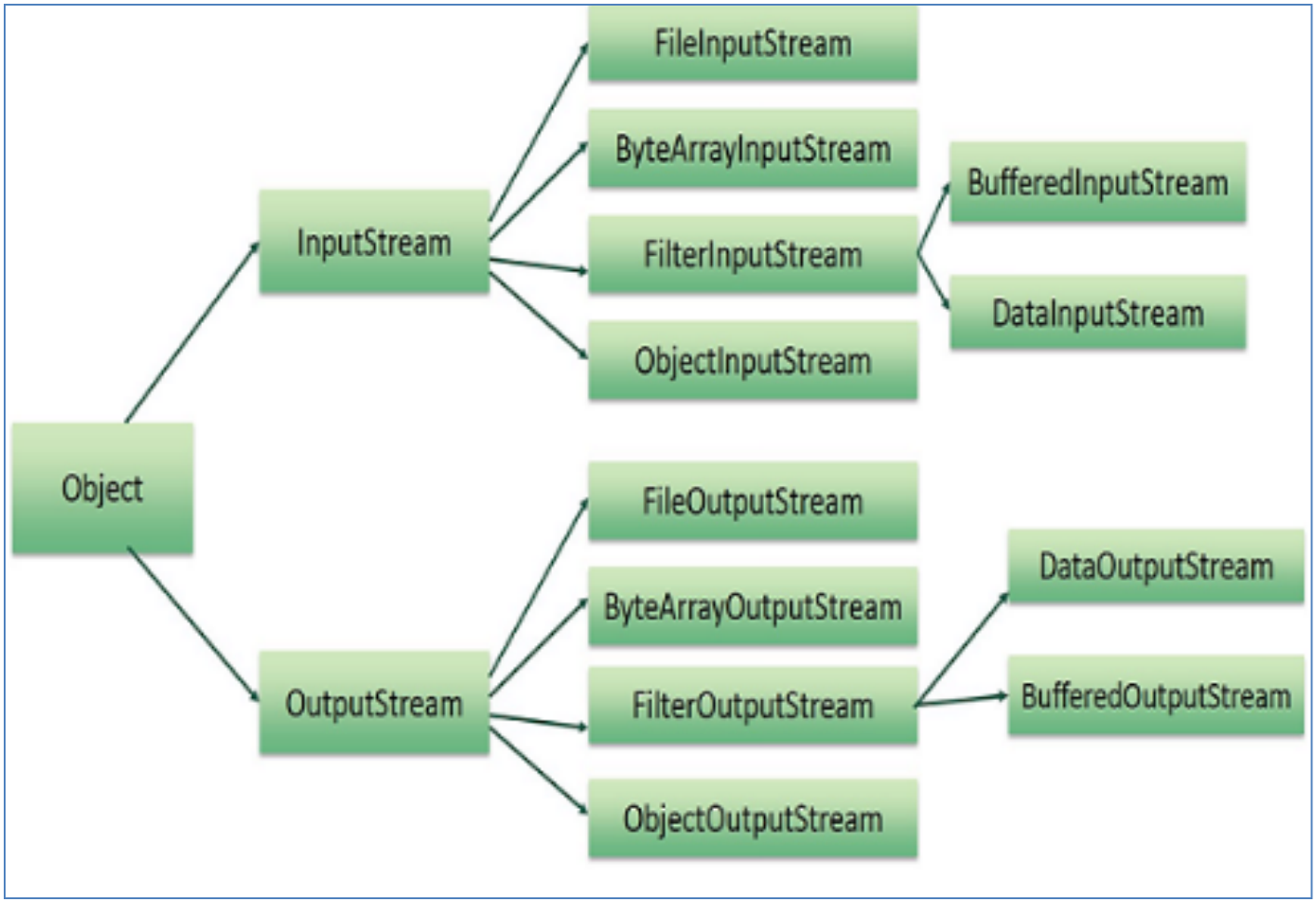
- **Why Nested Classes**
  - It is a way of logically grouping classes that are only used in one place
  - It increases encapsulation
  - It can lead to more readable and maintainable code

- **Types**
  - Static Nested Classes
  - Inner Classes (Non-static)
    - Local Inner Class -> declare an inner class within the body of a method
    - Anonymous Inner Class -> declare an inner class within the body of a method without naming the class

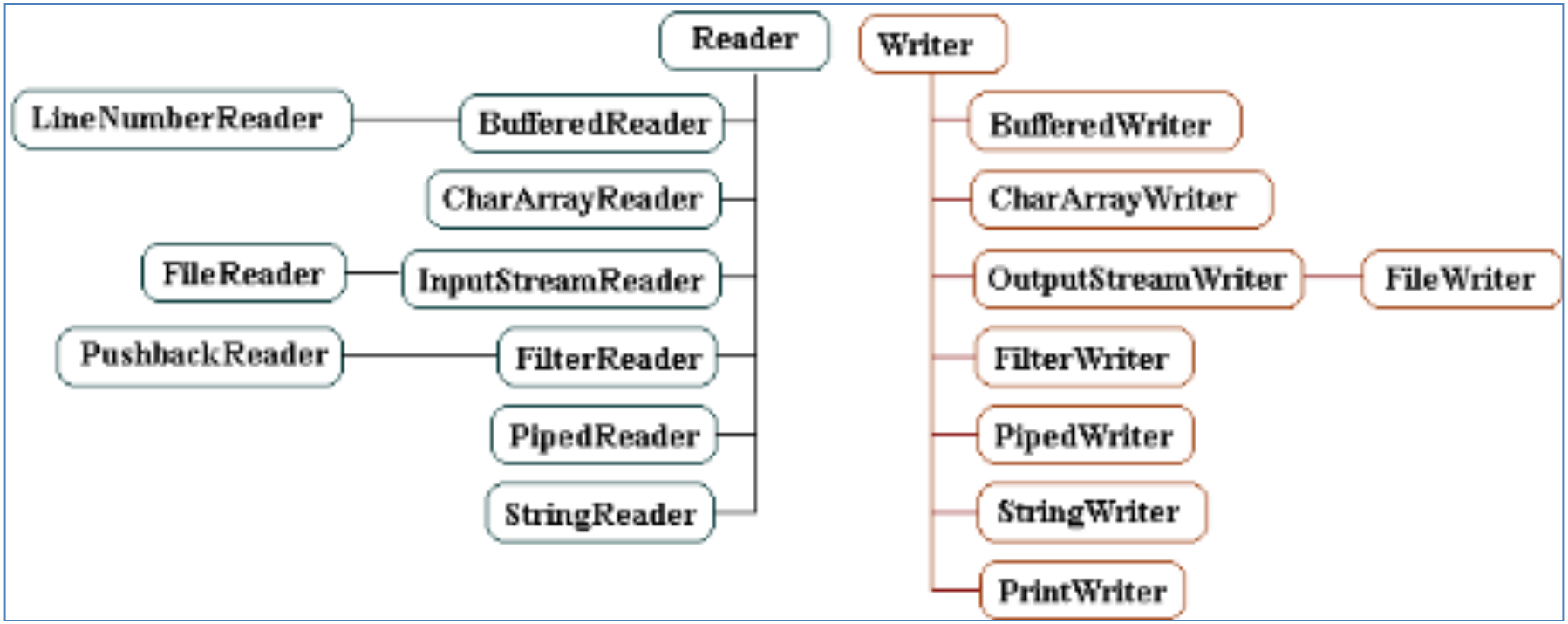
Java Serialization and I/O

- Serialization Overview
- I/O Streams Overview
- NIO (Non-blocking I/O Overview)

Byte Stream Hierarchy



Character Stream Hierarchy



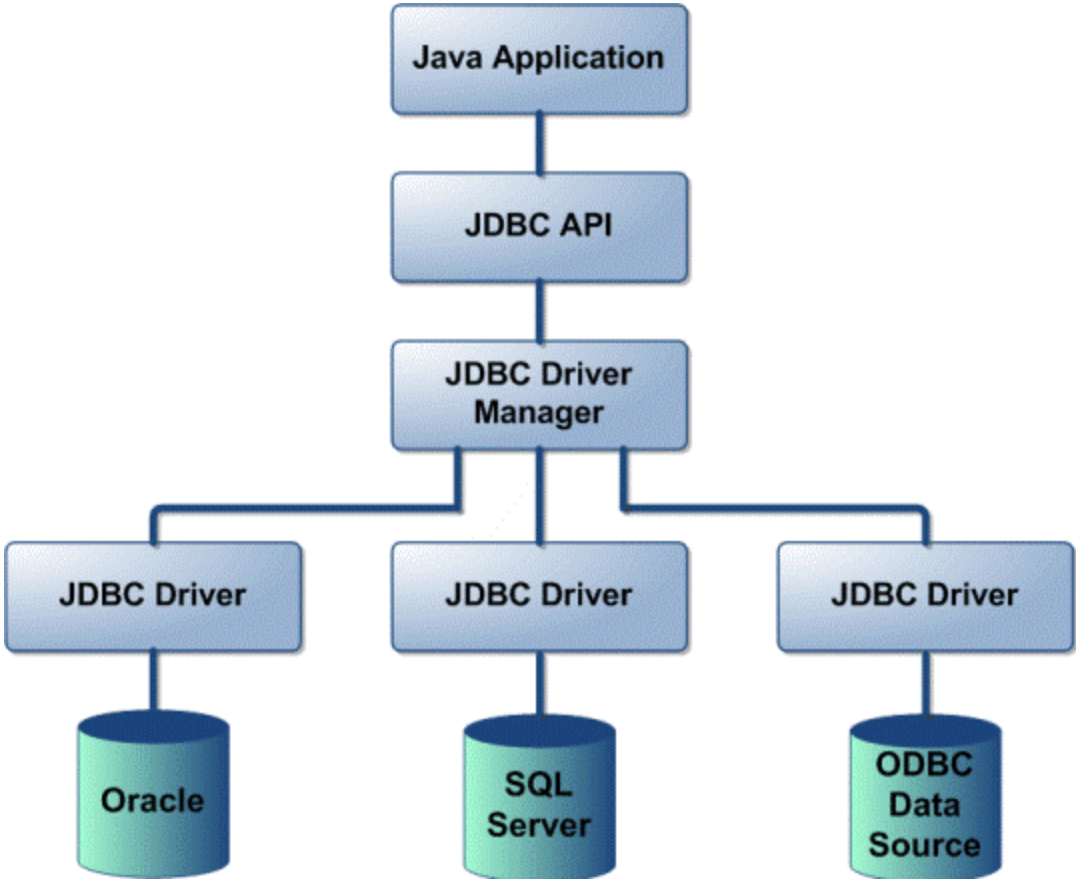
Java Concurrent Programming

- Introduction to Concurrent Programming
- Java Multi-Threading Overview
- Java Concurrency API Overview

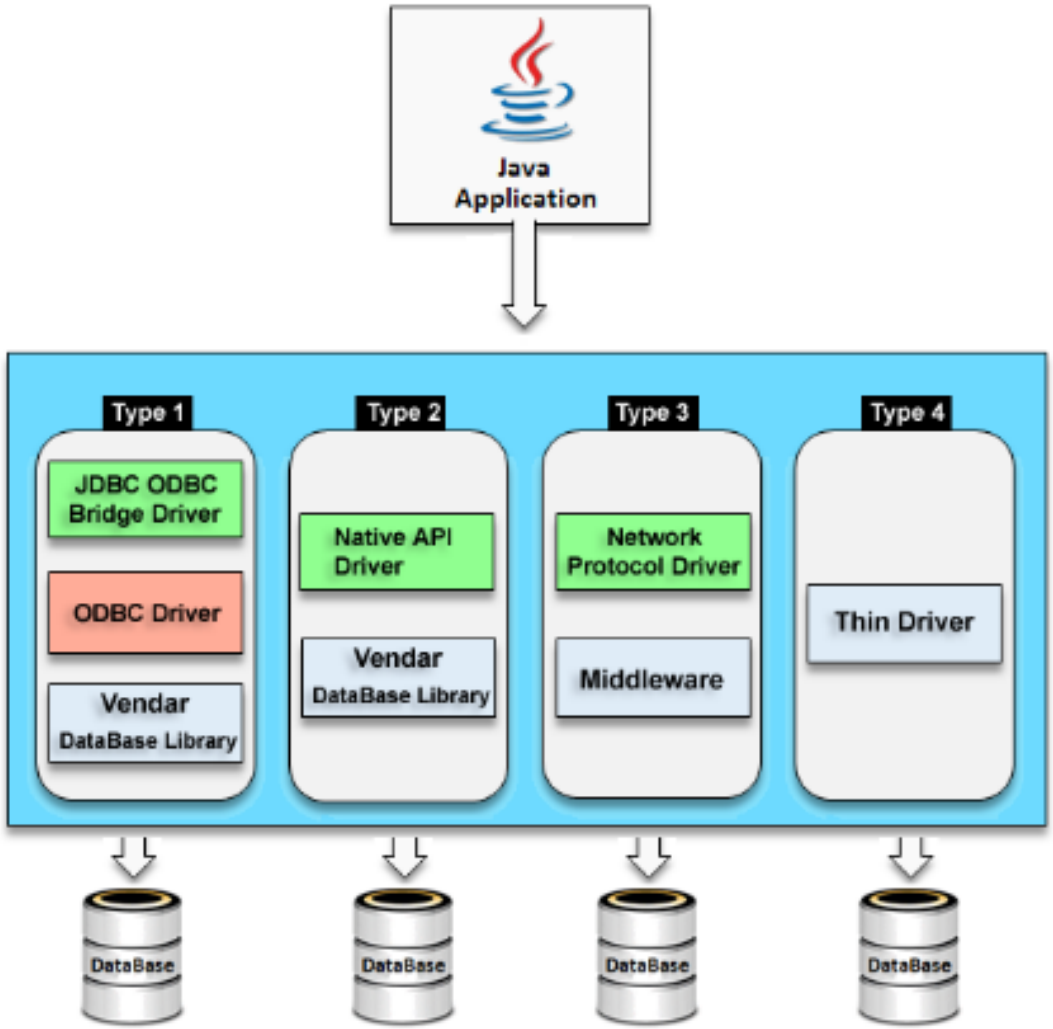
JDBC Programming

- Introduction to JDBC
- Loading Driver / Creating Data Source
- Creating Connection
- Preparing/Compiling Statements
- Executing Statements
- Processing ResultSet

JDBC Overview



JDBC Drivers



Java 8 Features

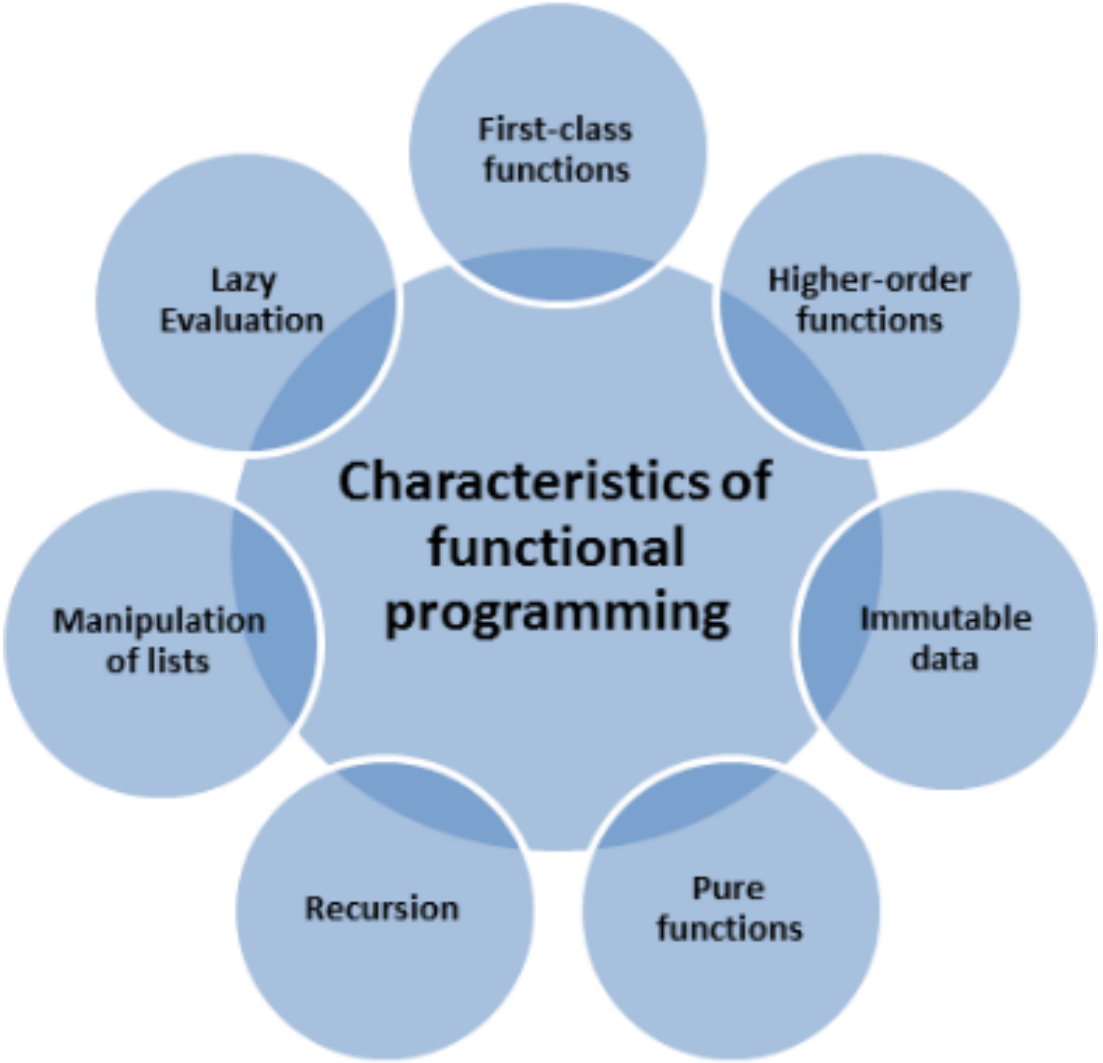
- Fundamentals of Functional Programming
- Lambda Expressions
- Functional Interfaces
- Method References
- Stream API - foreach, map, filter, parallel processing, collectors, etc.
- Default Methods
- Optional
- New DateTime package

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

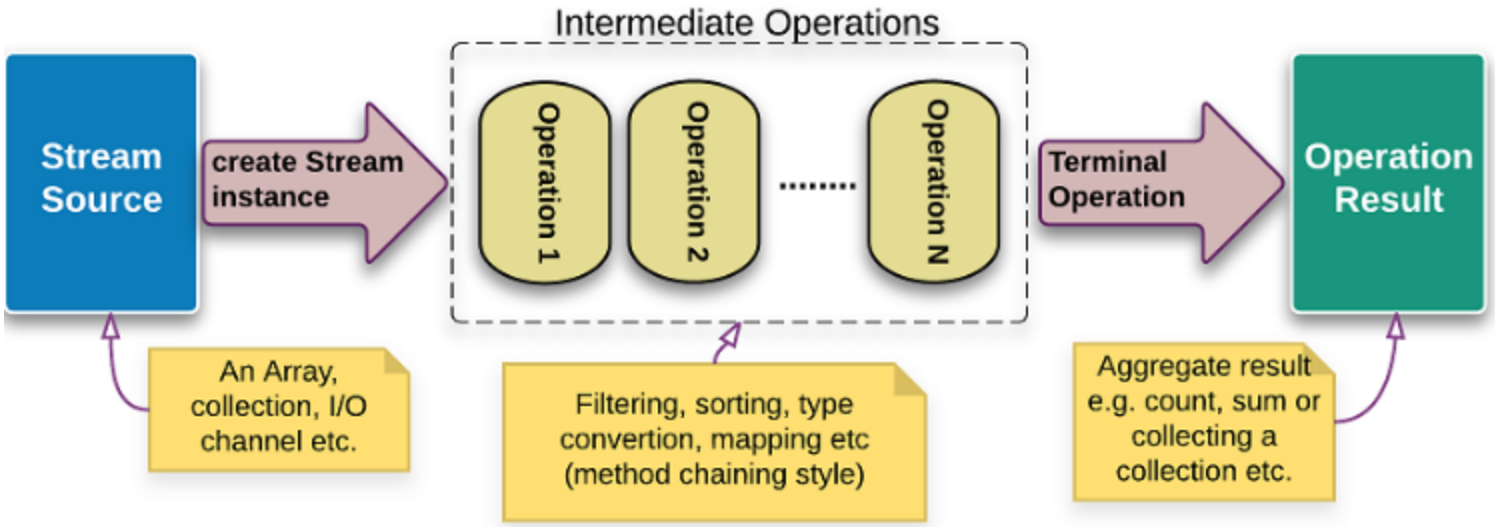
- No arguments:
```
() -> System.out.println("Hello")
```
- One argument:
```
s -> System.out.println(s)
```
- Two arguments:
```
(x, y) -> x + y
```
- With explicit argument types:
```
(Integer x, Integer y) -> x + y
```
- Multiple statements:
```
(x, y) -> {
    System.out.println(x);
    System.out.println(y);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



Thank You!