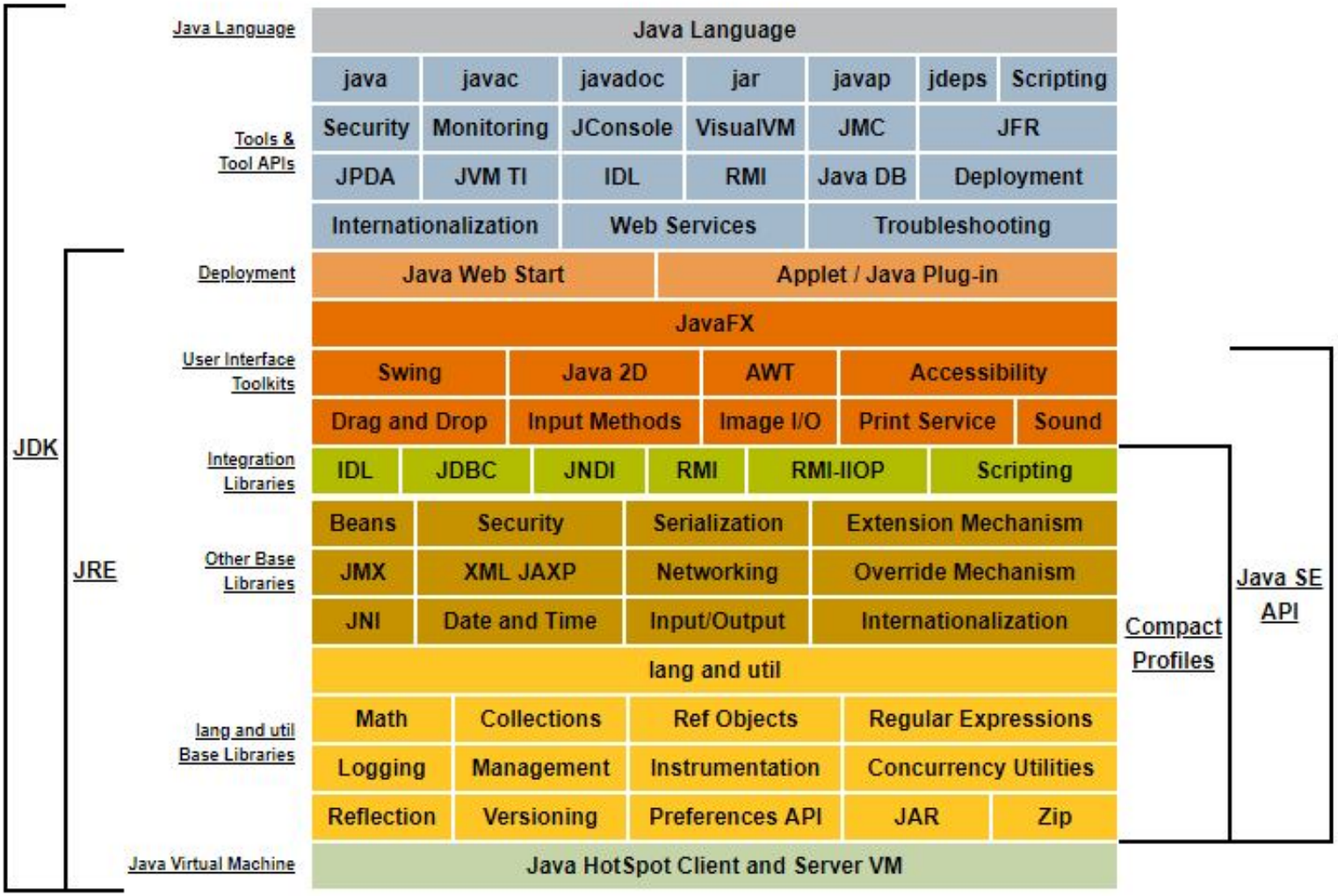
INTRODUCTION TO JAVA

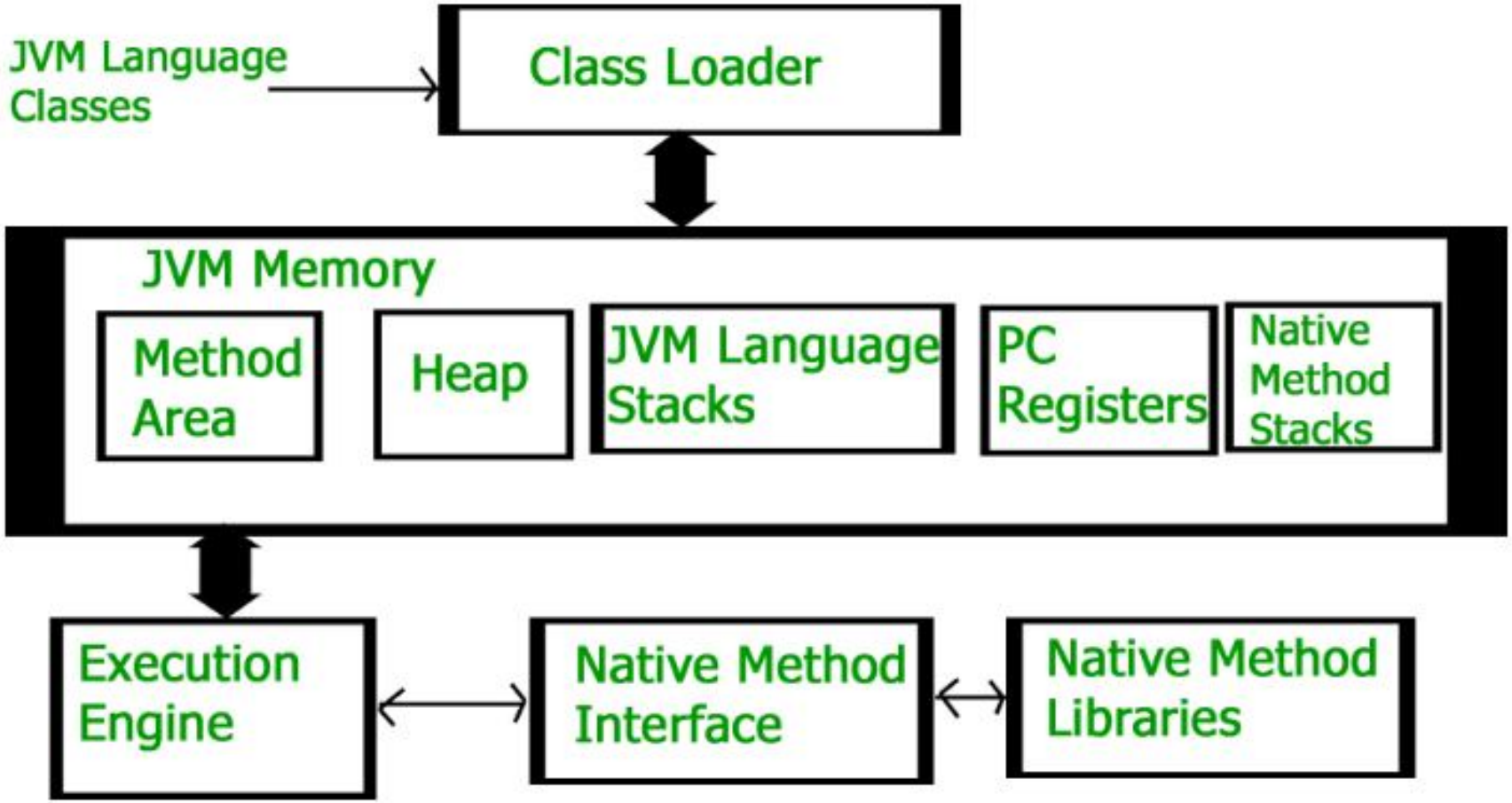
Java programming fundamentals

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

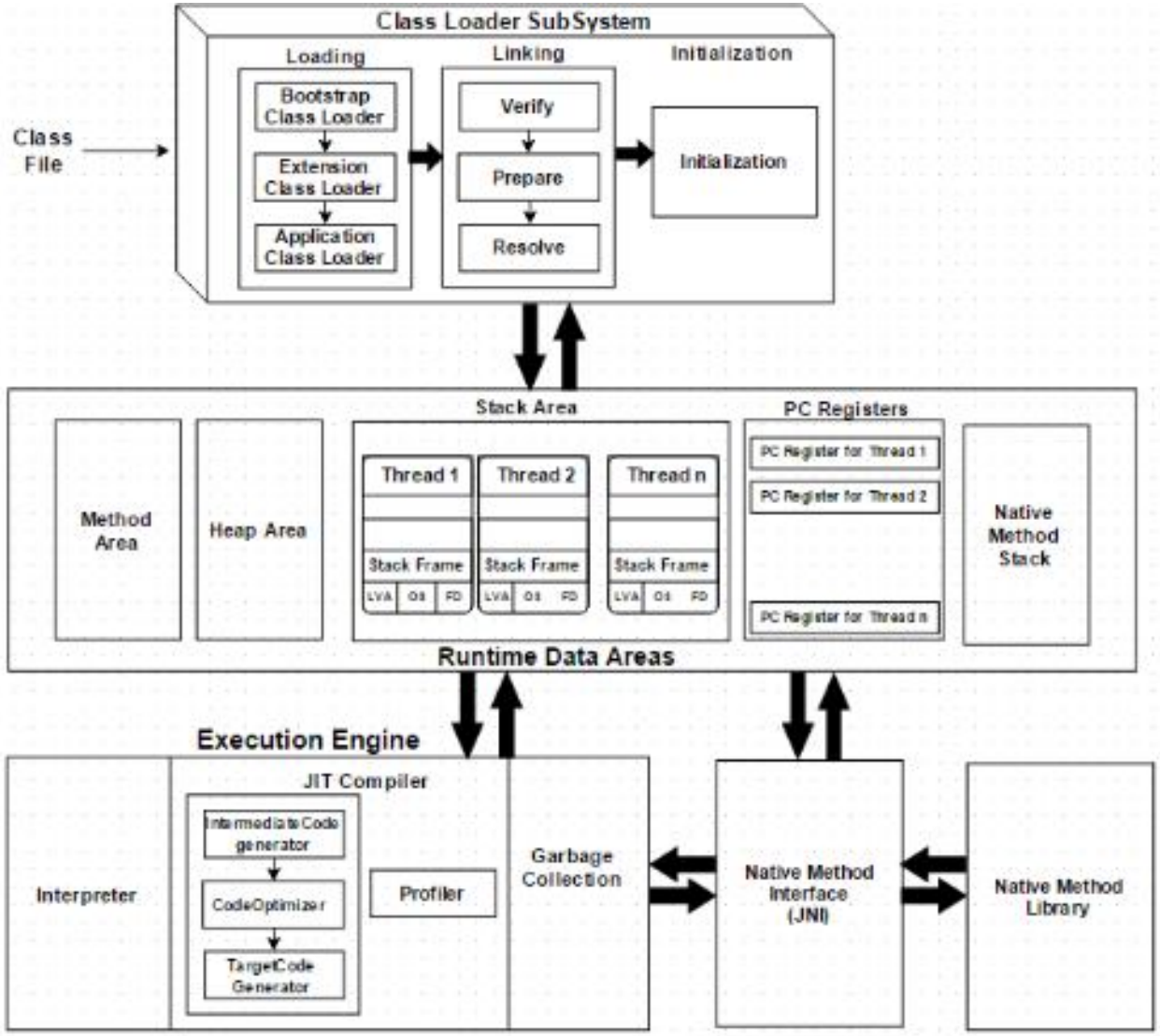
Java Conceptual Model (JVM/JRE/JDK)



JVM Architecture



JVM Architecture (detailed)



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

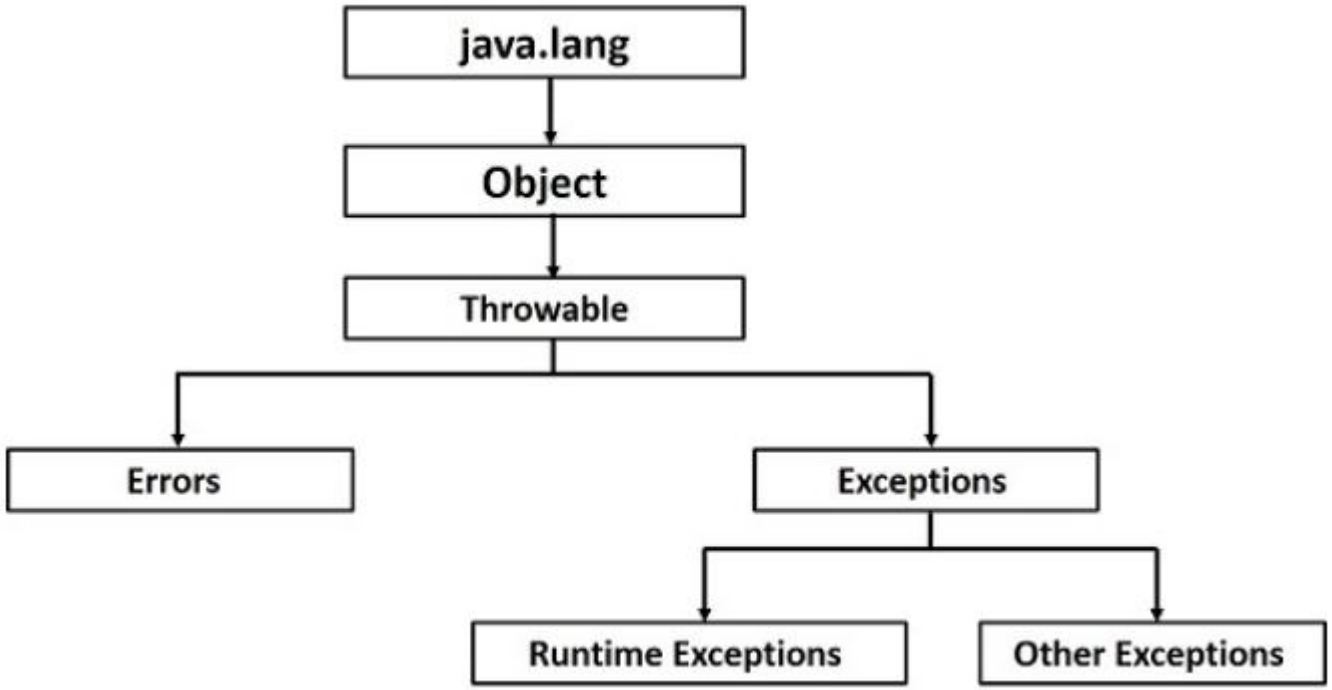
- Class
- Object
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

- 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

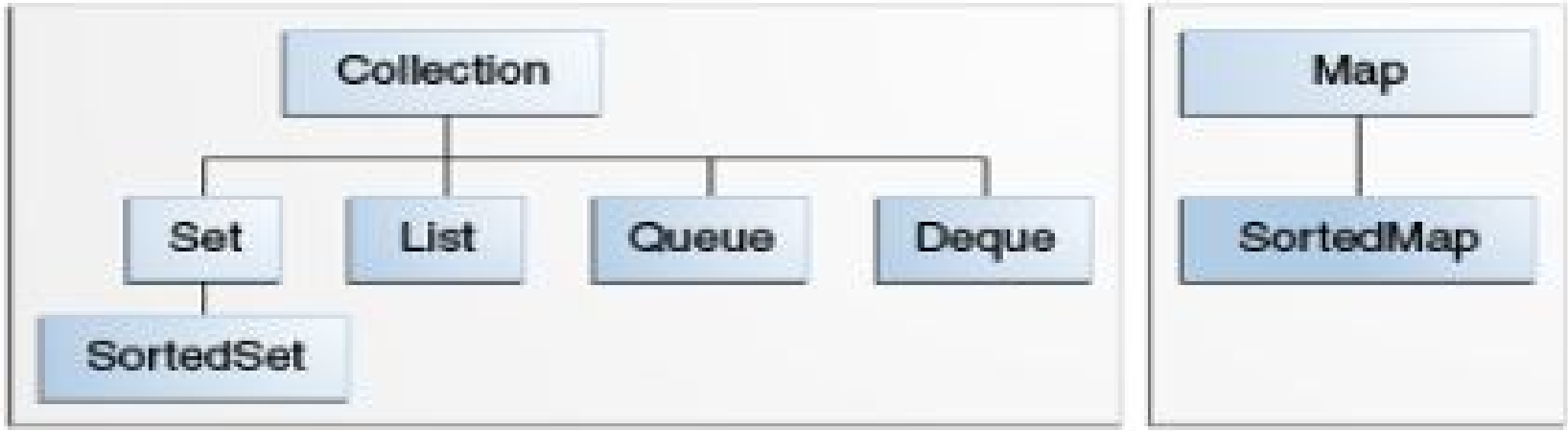
Collections Framework Overview

- 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

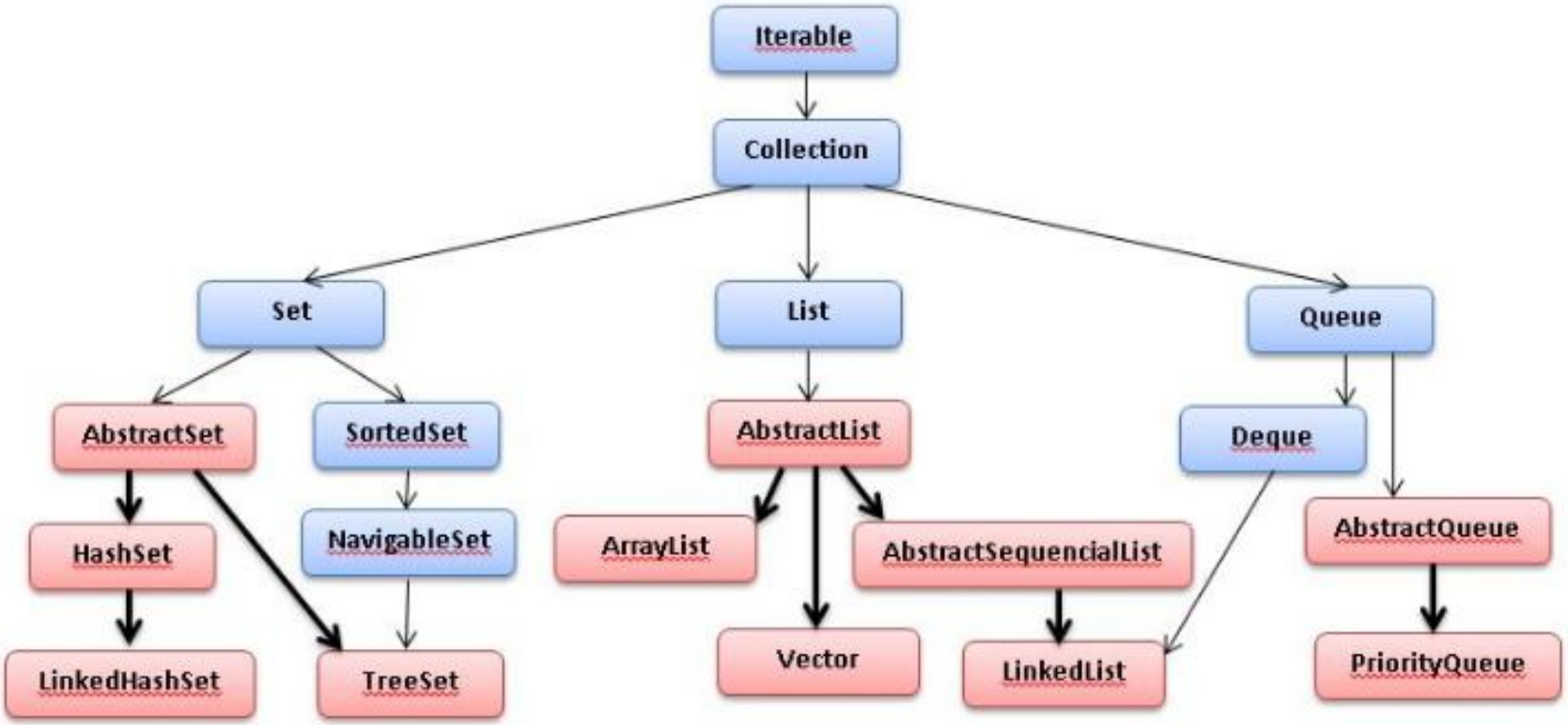
Collections Framework Benefits

- 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

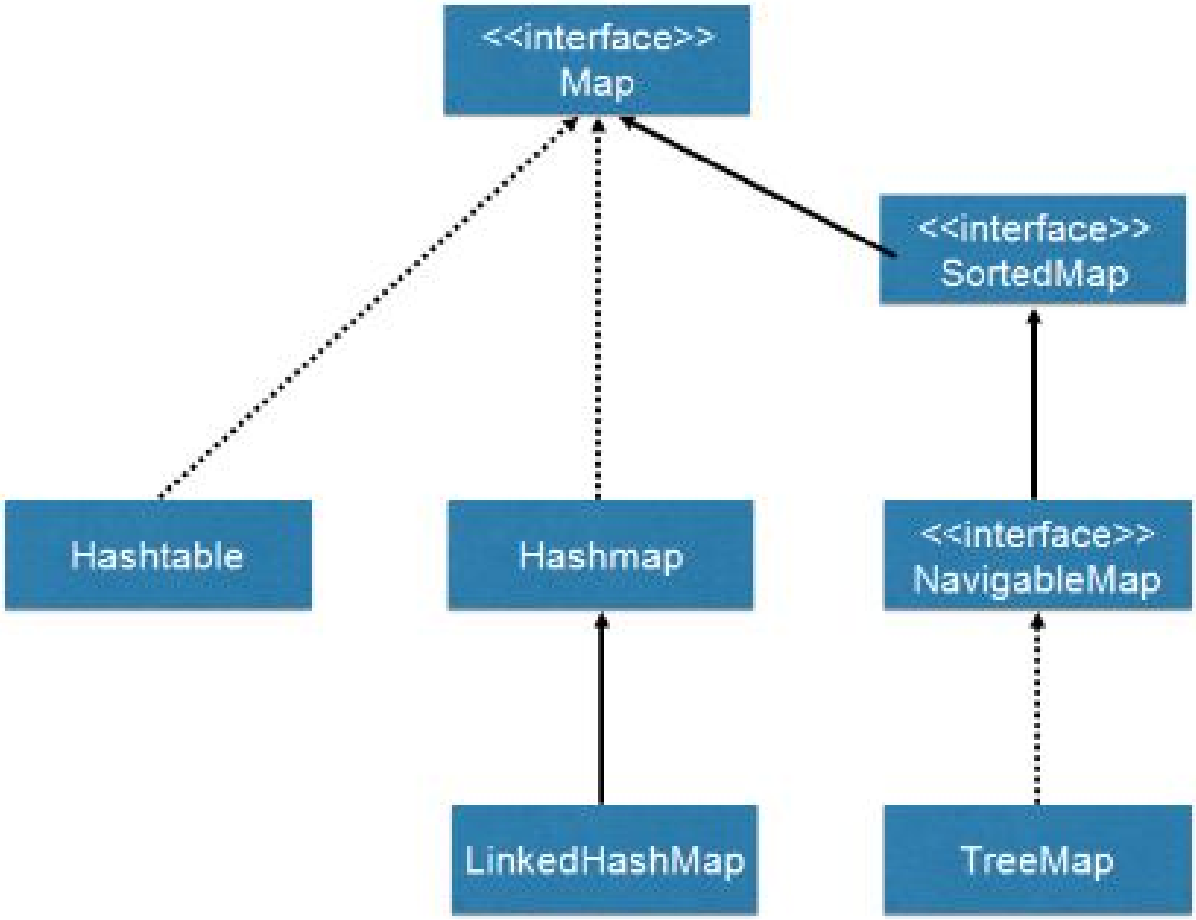
Collection Hierarchy (Interfaces)



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 8 Features

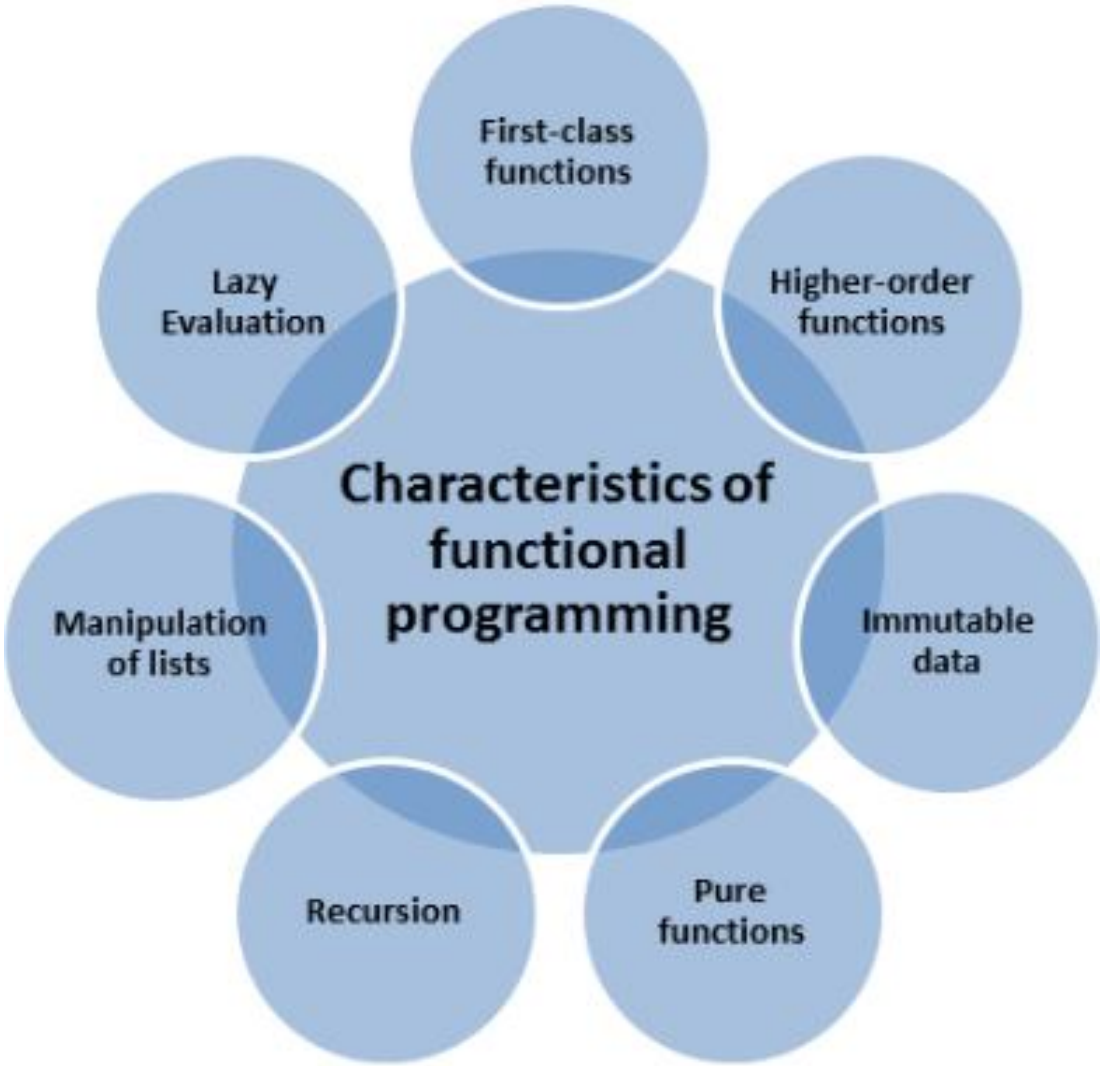
- Fundamentals of Functional Programming
- Lambda Expressions
- Functional Interfaces
- Method References
- Stream API - foreach, map, filter, parallel processing, collectors, etc.
- Default Methods
- Optional
- New DateTime package

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

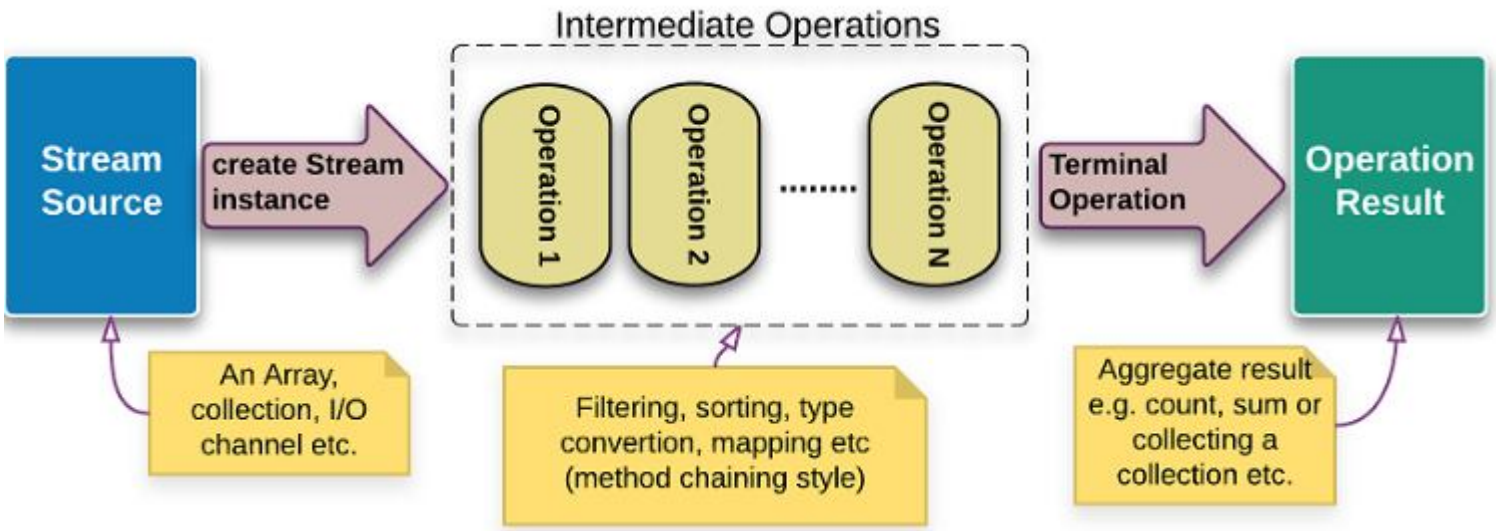
- No arguments:
```
() -> System.out.println("Hello")
```
- One argument:
```
s -> System.out.println(s)
```
- Two arguments:
```
(x, y) -> x + y
```
- With explicit argument types:
```
(Integer x, Integer y) -> x + y
```
- Multiple statements:
```
(x, y) -> {
    System.out.println(x);
    System.out.println(y);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



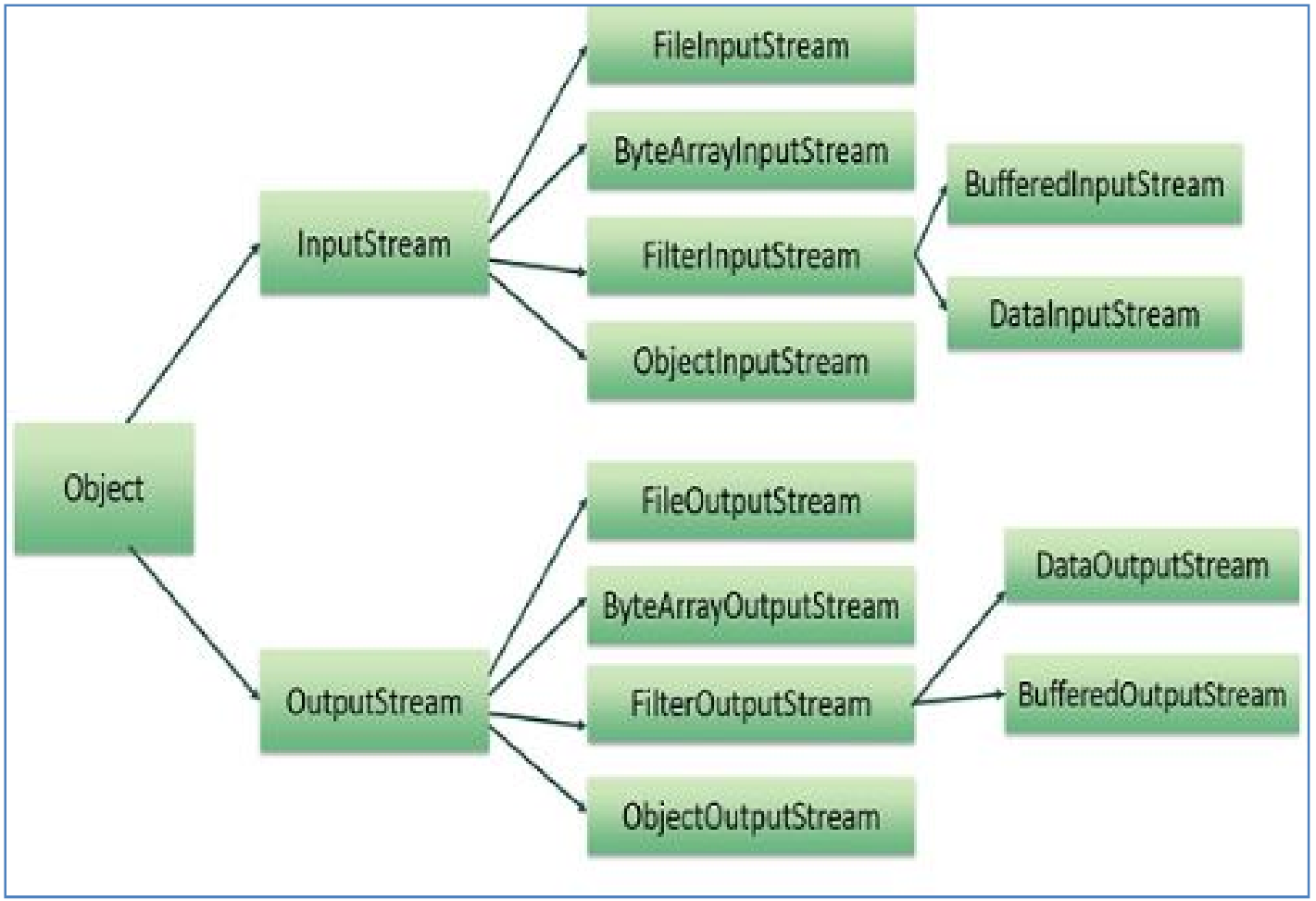
Java Concurrent Programming

- Introduction to Concurrent Programming
- Java Multi-Threading Overview
- Java Concurrency API Overview
- Enhanced Concurrency API with Lambdas

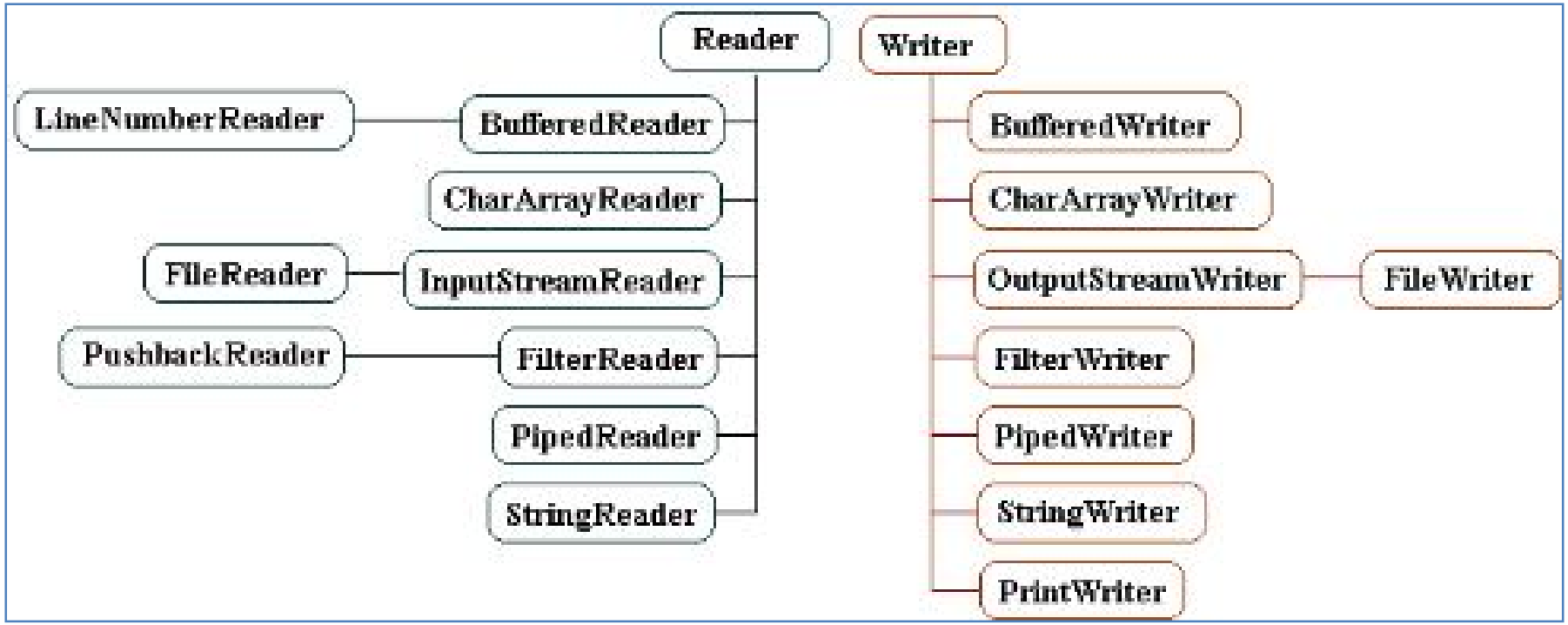
Java Serialization and I/O

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

Byte Stream Hierarchy



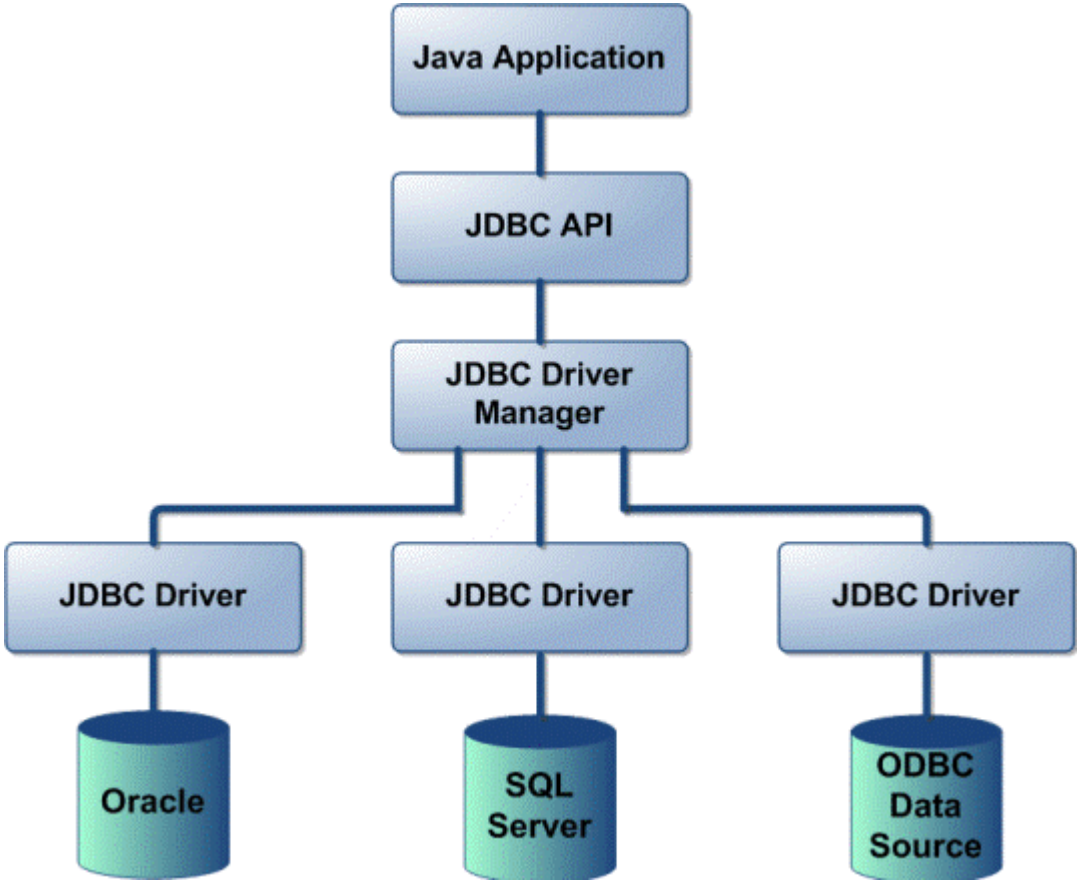
Character Stream Hierarchy



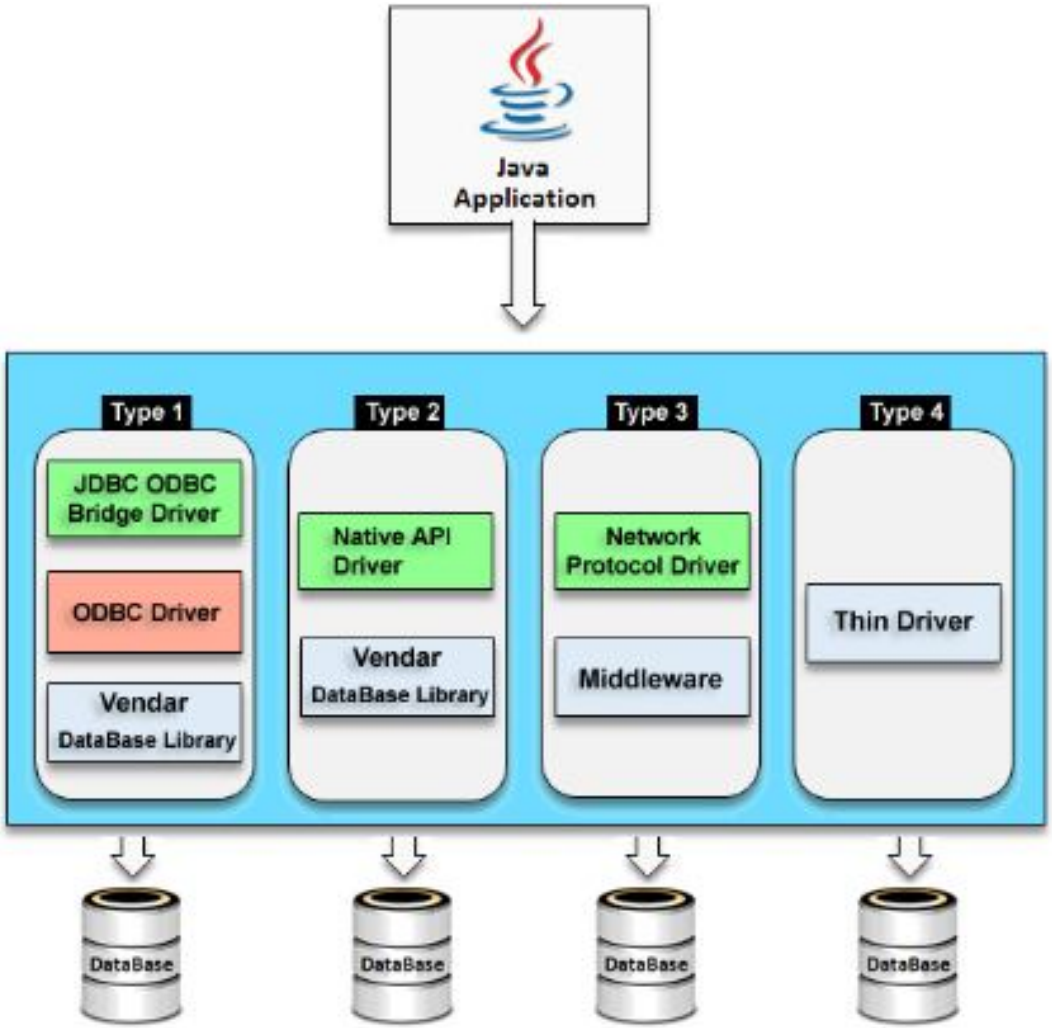
JDBC Programming

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

JDBC Overview



JDBC Drivers



Thank You!