Introduction to Java - 2 Hour Course Materials

Course Overview

Duration: 2 hours
Target Audience: Beginners to programming or developers new to Java
Prerequisites: Basic understanding of programming concepts

---

1. History of Java (15 minutes)

Timeline of Java Development

1991 - The Green Project

- Started by James Gosling, Mike Sheridan, and Patrick Naughton at Sun Microsystems
- Originally called "Oak" - designed for embedded systems and consumer electronics
- Goal: Create a platform-independent programming language

1995 - Java is Born

- Renamed to "Java" (after Java coffee)
- First public release: Java 1.0
- "Write Once, Run Anywhere" (WORA) philosophy introduced

Key Milestones:

- **1996:** Java 1.1 - Inner classes, JavaBeans, JDBC
- **1998:** Java 2 (J2SE 1.2) - Swing, Collections Framework
- **2004:** Java 5 - Generics, Annotations, Autoboxing
- **2010:** Oracle acquires Sun Microsystems
- **2014:** Java 8 - Lambda expressions, Stream API
- **2017:** Java 9 - Module system (Project Jigsaw)
- **Present:** Java follows 6-month release cycle

Why Java was Created

- **Platform Independence:** Solve the "write once, run anywhere" problem
- **Security:** Safe execution in networked environments
- **Simplicity:** Easier than C++ but powerful
- **Internet-ready:** Perfect timing for web development boom

---

2. What is Java (20 minutes)

Definition

Java is a high-level, object-oriented, platform-independent programming language designed for building robust, secure, and portable applications.

Core Philosophy

- **Simple:** Easy to learn and use
- **Object-Oriented:** Everything is an object (except primitives)
- **Platform Independent:** "Write Once, Run Anywhere"
- **Secure:** Built-in security features
- **Robust:** Strong memory management and error handling
- **Multithreaded:** Built-in support for concurrent programming

How Java Works

```
Source Code (.java) → Compiler (javac) → Bytecode (.class) → JVM → Machine Code
```

Java vs Other Languages

Feature	Java	C++	Python
Platform Independence	✓	✗	✓
Memory Management	Automatic	Manual	Automatic
Compilation	Compiled to Bytecode	Compiled to Machine Code	Interpreted
Speed	Medium	Fast	Slow
Learning Curve	Medium	Hard	Easy

---

3. Java Flavors (15 minutes)

Java Editions

1. Java SE (Standard Edition)

- Core Java platform
- Includes basic libraries and APIs
- Desktop and standalone applications
- **Components:** JVM, JRE, JDK

2. Java EE (Enterprise Edition) / Jakarta EE

- Built on top of Java SE
- Enterprise-level applications
- Web services, servlets, EJBs
- **Use cases:** Large-scale enterprise applications

3. Java ME (Micro Edition)

- Subset of Java SE
- Mobile devices and embedded systems
- Limited resources environments

- **Use cases:** IoT devices, mobile apps (legacy)

Java Distributions

- **Oracle JDK:** Commercial support, official distribution
- **OpenJDK:** Open-source reference implementation
- **Amazon Corretto:** Free, production-ready distribution
- **Azul Zulu:** Enterprise-focused distribution
- **Eclipse Temurin:** AdoptOpenJDK successor

Version Naming

- **Legacy:** Java 1.x (1.0, 1.1, 1.2, etc.)
- **Modern:** Java x (8, 11, 17, 21, etc.)
- **LTS Versions:** 8, 11, 17, 21 (Long Term Support)

---

4. Characteristics of Java (20 minutes)

1. Platform Independence

- **Bytecode:** Intermediate representation
- **JVM:** Platform-specific interpreter
- Same bytecode runs on any OS with JVM

2. Object-Oriented Programming

```
// Everything is an object (except primitives)
public class Car {
    private String brand;
    private int speed;

    public void accelerate() {
        speed++;
    }
}
```

3. Automatic Memory Management

- **Garbage Collection:** Automatic memory cleanup
- **No manual memory allocation/deallocation**
- Prevents memory leaks and dangling pointers

4. Strong Type System

```
int number = 10;        // Primitive type
String text = "Hello";  // Reference type
// number = text;       // Compilation error!
```

5. Exception Handling

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Division by zero!");
}
```

6. Multithreading Support

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running");
    }
}
```

7. Security Features

- **Bytecode verification**
- **Security Manager**
- **Access control mechanisms**
- **Sandbox execution**

8. Rich Standard Library

- Collections Framework
- I/O operations
- Networking
- GUI development (Swing/AWT)
- Database connectivity (JDBC)

---

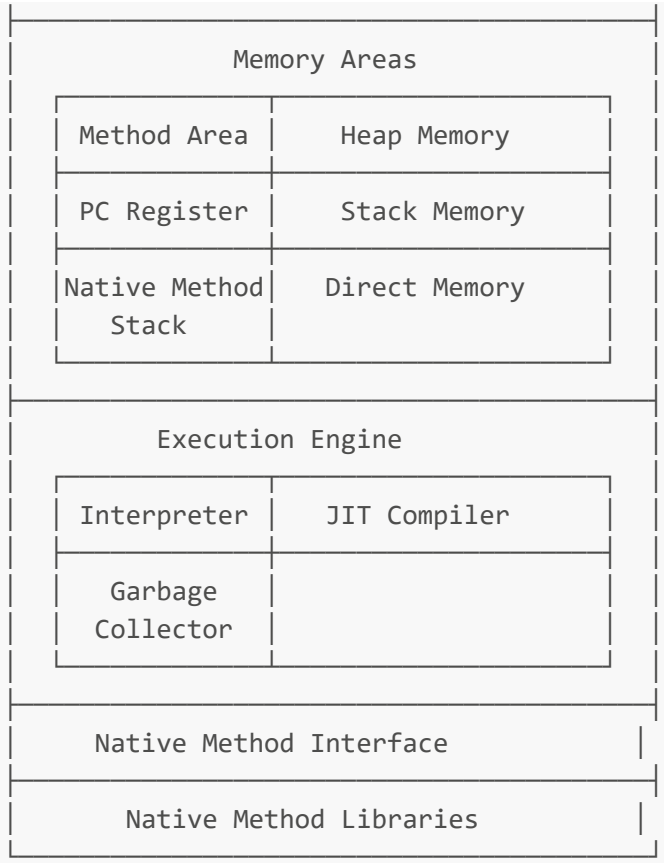
5. JVM Architecture (25 minutes)

What is JVM?

The Java Virtual Machine is a runtime environment that executes Java bytecode. It provides platform independence by abstracting the underlying operating system.

JVM Components

```
┌─────────────────────────────────────────┐
│                JVM                      │
├─────────────────────────────────────────┤
│  Class Loader Subsystem                 │
```



1. Class Loader Subsystem

Responsibilities:

- Loading classes from various sources
- Linking classes and interfaces
- Initializing classes

Types of Class Loaders:

- **Bootstrap Class Loader:** Loads core Java classes
- **Extension Class Loader:** Loads extension libraries
- **Application Class Loader:** Loads application classes

2. Memory Areas

Method Area (Metaspace in Java 8+)

- Stores class-level information
- Method bytecode, constant pool
- Static variables

Heap Memory

- Stores objects and instance variables
- Divided into Young Generation and Old Generation
- Garbage collected

Stack Memory

- Thread-specific memory
- Stores method calls and local variables
- LIFO structure

PC (Program Counter) Register

- Stores address of currently executing instruction
- Thread-specific

Native Method Stack

- For native method calls (JNI)

3. Execution Engine

Interpreter

- Executes bytecode line by line
- Slower but simple

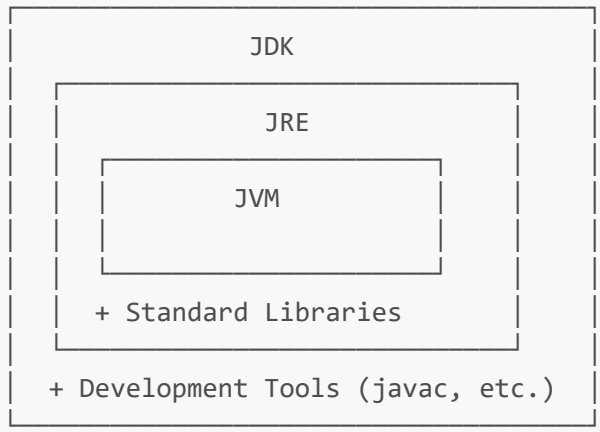
JIT (Just-In-Time) Compiler

- Compiles frequently used bytecode to native code
- Optimizes performance

Garbage Collector

- Automatic memory management
- Reclaims unused objects

JVM vs JRE vs JDK



- **JVM:** Runtime environment
- **JRE:** JVM + Standard libraries
- **JDK:** JRE + Development tools

6. Bytecode (15 minutes)

What is Bytecode?

Bytecode is an intermediate representation of Java source code that is platform-independent and executed by the JVM.

Compilation Process

```
Hello.java → javac → Hello.class (bytecode) → JVM → Machine Code
```

Bytecode Characteristics

- **Platform Independent:** Same bytecode runs on any JVM
- **Optimized:** Smaller than source code
- **Secure:** Verified before execution
- **Portable:** Can be transmitted over networks

Example: Bytecode Generation

Java Source (Hello.java):

```java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Compilation:

```
javac Hello.java  # Creates Hello.class
```

Bytecode Analysis:

```
javap -c Hello    # Disassemble bytecode
```

Sample Bytecode Output:

```
public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2    // Field java/lang/System.out
       3: ldc           #3    // String Hello, World!
```

```
        5: invokevirtual #4    // Method println
        8: return
```

Bytecode Instructions

- **Load/Store:** `iload`, `istore`, `aload`, `astore`
- **Arithmetic:** `iadd`, `isub`, `imul`, `idiv`
- **Method Calls:** `invokevirtual`, `invokestatic`
- **Control Flow:** `if_icmplt`, `goto`, `return`

Benefits of Bytecode

1. **Platform Independence**
2. **Network Mobility**
3. **Security Verification**
4. **Optimization Opportunities**

---

7. Class Loader (10 minutes)

Class Loading Process

1. Loading

- Reads .class files
- Creates Class objects in memory
- Links bytecode to JVM

2. Linking

- **Verification:** Ensures bytecode integrity
- **Preparation:** Allocates memory for static variables
- **Resolution:** Replaces symbolic references with direct references

3. Initialization

- Executes static initializers
- Initializes static variables

Class Loader Hierarchy

```
Bootstrap Class Loader (Native)
    ↓
Extension Class Loader
    ↓
Application Class Loader
    ↓
Custom Class Loaders
```

Class Loading Example

```
// Class loading happens automatically
MyClass obj = new MyClass();  // Triggers class loading

// Manual class loading
Class<?> clazz = Class.forName("com.example.MyClass");
```

Custom Class Loaders

```
public class CustomClassLoader extends ClassLoader {
    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        // Custom loading logic
        byte[] classData = loadClassData(name);
        return defineClass(name, classData, 0, classData.length);
    }
}
```

---

8. Unicode (8 minutes)

What is Unicode?

Unicode is a character encoding standard that supports characters from all writing systems worldwide.

Java and Unicode

- **Native Support:** Java uses UTF-16 internally
- **16-bit Characters:** Each char is 2 bytes
- **International Applications:** Built for global use

Unicode Examples

```
// Unicode literals
char heart = '\u2764';        // ♡
char smiley = '\u263A';       // ☺
String hindi = "नमस्ते";       // Hindi greeting
String chinese = "你好";       // Chinese greeting

// Unicode in strings
String message = "Hello \u0041\u0042\u0043";  // Hello ABC
```

Unicode Ranges

- **ASCII:** U+0000 to U+007F

- **Latin Extended:** U+0080 to U+024F
- **Greek:** U+0370 to U+03FF
- **Hindi (Devanagari):** U+0900 to U+097F
- **Chinese (CJK):** U+4E00 to U+9FFF

Benefits

1. **Global Compatibility**
2. **Consistent Encoding**
3. **No Character Set Confusion**
4. **Rich Text Support**

---

9. Class Path (10 minutes)

What is Class Path?

Class path is a parameter that tells the JVM where to look for user-defined classes and packages.

Setting Class Path

Command Line:

```
# Using -cp or -classpath
java -cp /path/to/classes MyClass
java -classpath /path/to/classes:/path/to/lib/* MyClass

# Multiple paths (Windows uses semicolon)
java -cp "C:\classes;C:\lib\*" MyClass
```

Environment Variable:

```
# Linux/Mac
export CLASSPATH=/path/to/classes:/path/to/lib/*

# Windows
set CLASSPATH=C:\classes;C:\lib\*
```

Class Path Sources

1. **Current Directory** (default)
2. **JAR files**
3. **Directory containing .class files**
4. **ZIP files containing classes**

Example Directory Structure

```
project/
├── src/
│   └── com/
│       └── example/
│           └── MyClass.java
├── classes/
│   └── com/
│       └── example/
│           └── MyClass.class
└── lib/
    └── library.jar
```

Compilation and Execution:

```
# Compile
javac -d classes src/com/example/MyClass.java

# Run
java -cp classes:lib/* com.example.MyClass
```

Common Class Path Issues

1. **ClassNotFoundException:** Class not found in class path
2. **NoClassDefFoundError:** Class was available at compile time but not runtime
3. **Wrong package structure**
4. **Missing JAR dependencies**

---

10. Path Environment Variable (7 minutes)

What is PATH?

PATH is an environment variable that tells the operating system where to find executable programs.

Setting Java PATH

Windows:

```
# Temporary
set PATH=%PATH%;C:\Program Files\Java\jdk-17\bin

# Permanent (System Properties → Environment Variables)
PATH = C:\Program Files\Java\jdk-17\bin;...
```

Linux/Mac:

```
# Temporary
export PATH=$PATH:/usr/lib/jvm/java-17-openjdk/bin

# Permanent (add to ~/.bashrc or ~/.profile)
echo 'export PATH=$PATH:/usr/lib/jvm/java-17-openjdk/bin' >> ~/.bashrc
source ~/.bashrc
```

JAVA_HOME Environment Variable

```
# Linux/Mac
export JAVA_HOME=/usr/lib/jvm/java-17-openjdk
export PATH=$JAVA_HOME/bin:$PATH

# Windows
set JAVA_HOME=C:\Program Files\Java\jdk-17
set PATH=%JAVA_HOME%\bin;%PATH%
```

Verification

```
# Check Java installation
java -version
javac -version

# Check paths
echo $JAVA_HOME    # Linux/Mac
echo %JAVA_HOME%   # Windows

which java         # Linux/Mac
where java         # Windows
```

Common PATH Issues

1. **'java' is not recognized:** Java not in PATH
2. **Wrong Java version:** Multiple Java installations
3. **JAVA_HOME not set:** Some tools require JAVA_HOME

Summary and Key Takeaways

Core Concepts Covered

1. **Java History:** From Oak to modern Java
2. **Platform Independence:** Write once, run anywhere
3. **JVM Architecture:** Understanding the runtime environment
4. **Bytecode:** Intermediate representation
5. **Class Loading:** How classes are loaded and initialized

6. **Unicode Support:** International character handling
7. **Class Path Configuration:** Finding classes at runtime
8. **Environment Setup:** PATH and JAVA_HOME

Next Steps

1. **Install Java Development Kit (JDK)**
2. **Set up development environment (IDE)**
3. **Write your first Java program**
4. **Explore object-oriented programming concepts**
5. **Practice with basic Java syntax and constructs**

Recommended Resources

- Oracle Java Documentation
- Java Tutorials (Oracle)
- OpenJDK Documentation
- Java Language Specification
- "Effective Java" by Joshua Bloch

Quick Reference Commands

```
# Compilation
javac ClassName.java

# Execution
java ClassName

# With custom class path
java -cp /path/to/classes ClassName

# View bytecode
javap -c ClassName

# Check Java version
java -version

# Set class path environment variable
export CLASSPATH=/path/to/classes
```