Module 1: Java Basics

Premanand S

Assistant Professor
School of Electronics Engineering (SENSE)
Vellore Institute of Technology
Chennai Campus

premanand.s@vit.ac.in

December 18, 2024

Topics covered in Module 1,

- OOP Paradigm
- Features of JAVA Language
- JVM
- Bytecode
- Java Program Structure
- Basic Programming Construct
- Data Types
- Variables
- Java naming conventions
- Operators

Verdict of Module 1,

- Foundational knowledge required to write basic Java programs
- Java's core concepts
- How code execution work?
- How to use variables, operators, and basic constructs effectively?

Platforms to skill,

- OnlineGDB JAVA Compiler

Paradigm

What is a Programming Paradigm?

- A **programming paradigm** is a style or way of thinking about and structuring programs.
- It defines how developers write, organize, and structure code based on specific concepts and principles.
- Programming paradigms are not languages or tools. You can't "build" anything with a paradigm.
- They're more like a set of ideals and guidelines that many people have agreed on, followed, and expanded upon.

Types of Programming Paradigms

- **Imperative Paradigm:** Focuses on how to perform tasks step by step. (e.g., C, Java)
- **Declarative Paradigm:** Focuses on what the outcome should be. (e.g., SQL, Prolog)
- **Object-Oriented Paradigm:** Based on objects representing data and behavior. (e.g., Java, Python)
- **Functional Paradigm:** Emphasizes immutability and functions. (e.g., Haskell, Lisp)
- **Logic Paradigm:** Based on formal logic and rules. (e.g., Prolog)
- **Event-Driven Paradigm:** Focuses on responding to events. (e.g., JavaScript)
- **Concurrent Paradigm:** Manages multiple processes running simultaneously. (e.g., Go, Rust)

Imperative Programming

- It's called "imperative" because as programmers we dictate exactly what the computer has to do, in a very specific way.

Example (Understanding)

```
1- Pour flour in a bowl
2- Pour a couple eggs in the same bowl
3- Pour some milk in the same bowl
4- Mix the ingredients
5- Pour the mix in a mold
6- Cook for 35 minutes
7- Let chill
```

Procedural Programming

- Procedural programming is a derivation of imperative programming, adding to it the feature of functions (also known as "procedures" or "subroutines").
- User is encouraged to subdivide the program execution into functions, as a way of improving modularity and organization.

Procedural Programming (Contd...)

Example (Understanding)

```
function pourIngredients() {
    - Pour flour in a bowl
    - Pour a couple eggs in the same bowl
    - Pour some milk in the same bowl}

function mixAndTransferToMold() {
    - Mix the ingredients
    - Pour the mix in a mold}

function cookAndLetChill() {
    - Cook for 35 minutes
    - Let chill}
pourIngredients()
mixAndTransferToMold()
cookAndLetChill()
```

Functional Programming

- In functional programming, functions are treated as first-class citizens, meaning that they can be assigned to variables, passed as arguments, and returned from other functions.
- Another key concept is the idea of pure functions. A pure function is one that relies only on its inputs to generate its result. And given the same input, it will always produce the same result.

Functional Programming (Contd...)

Example (Understanding)

```
# Pure functions
def pour_ingredients():
    return ["flour", "eggs", "milk"]

def mix_ingredients(ingredients):
    return f"Mixed {', '.join(ingredients)}"

def pour_into_mold(mix):
    return f"{mix} poured into mold"

def bake(mix_in_mold):
    return f"{mix_in_mold} baked for 35 minutes"

def let_chill(baked_item):
    return f"{baked_item} chilled"
```

Functional Programming (Contd...)

Example (Understanding)

```
# Composing functions
ingredients = pour_ingredients()
mixed = mix_ingredients(ingredients)
molded = pour_into_mold(mixed)
baked = bake(molded)
final_product = let_chill(baked)

print(final_product)
```

Functional vs Procedural Programming Paradigms

Key Differences

- **Core Concept:**
  - Functional: *What* to do (declarative)
  - Procedural: *How* to do it (imperative)
- **State Management:**
  - Functional: Immutable data, no state modification
  - Procedural: Mutable data, state is modified
- **Functions:**
  - Functional: Functions are pure, stateless
  - Procedural: Functions may have side effects

Examples Use Cases

- **Functional:**
  - Use cases: Data transformation, ML, pipelines
  - Example: Transforming ingredients without modifying them
- **Procedural:**
  - Use cases: System tasks, step-by-step instructions
  - Example: Mixing and modifying ingredients directly

Object-Oriented Programming Paradigm

Key Concepts in OOP

- **Classes:** Templates to create objects (recipes in our case).
- **Objects:** Instances of classes that maintain state (ingredients, bowl, etc.).
- **Encapsulation:** Bundling data and methods that work on the data in one unit (class).
- **Inheritance:** Creating a new class that reuses properties and methods from another class.
- **Polymorphism:** Methods that do different things depending on the object type.

OOP Example: Recipe Class

Class: Recipe

- Attributes: ‘ingredients‘, ‘bowl‘, ‘mold‘, etc.
- Methods:
  - ‘pourIngredients()‘
  - ‘mix()‘
  - ‘bake()‘
  - ‘chill()‘

Object Instantiation:

- ‘myRecipe = Recipe()‘
- ‘myRecipe.pourIngredients()‘
- ‘myRecipe.mix()‘
- ‘myRecipe.bake()‘
- ‘myRecipe.chill()‘

OOP Paradigm

Object-Oriented Programming (OOP)

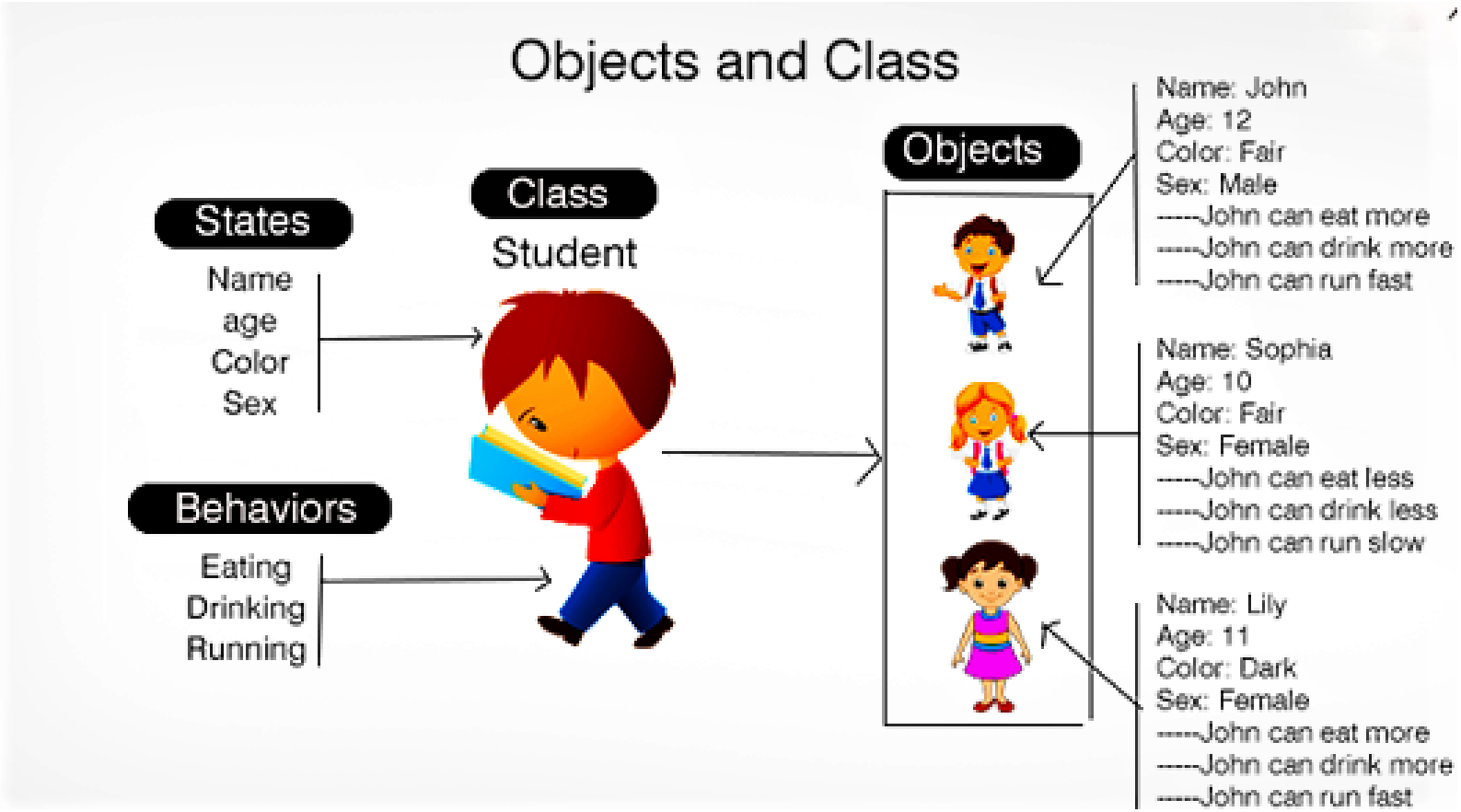
- Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which are instances of classes.
- It focuses on organizing code into reusable, self-contained units, modeling real-world entities and their relationships, and making it easier to design and maintain software systems.

Object-Oriented Programming (OOP)

Classes and Objects

- **Class**: A blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that the objects created from the class will have.
- **Object**: An instance of a class. It contains actual values for the properties defined by the class and can call its methods.

Classes and Objects (Contd...)



Class and Objects - code

Example (Understanding)

```
class Car:
    # Constructor to initialize object properties
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    # Method to display car details
    def display_info(self):
        print(f"Car Details: {self.year} {self.make}
                                    {self.model}")

my_car = Car("Toyota", "Corolla", 2020)

my_car.display_info()
```

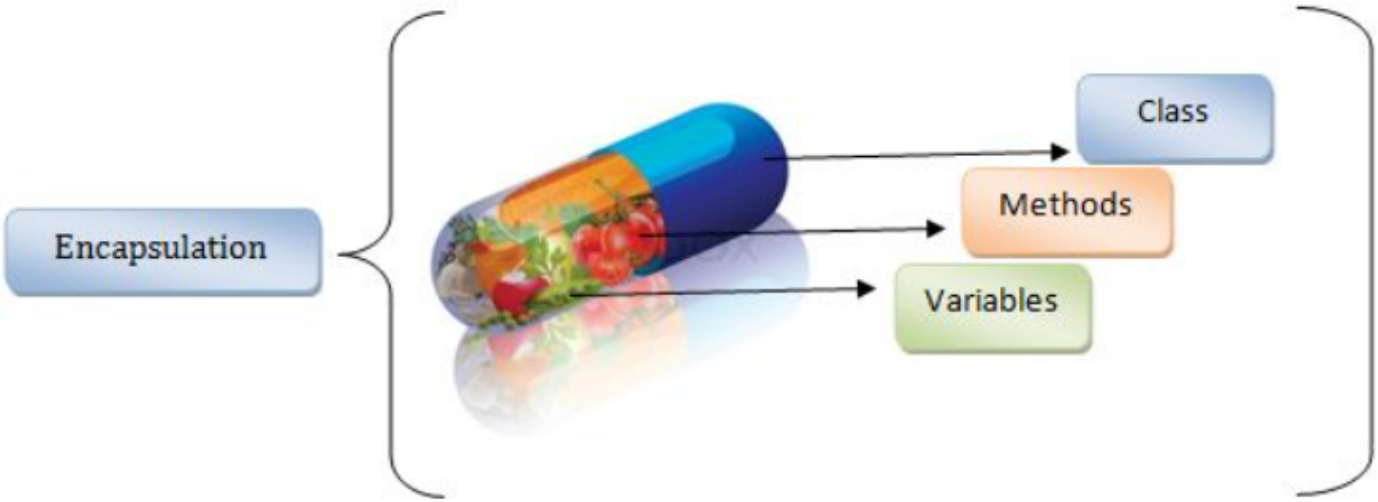
Encapsulation

Encapsulation is the process of bundling data (attributes) and methods (functions) that operate on the data within one unit, a class. It also involves restricting direct access to some of the object's components to protect the integrity of the data.
Access modifiers like:

- `private`
- `protected`
- `public`

are used to control access.

Encapsulation (Contd...)



Encapsulation (Contd...)

Encapsulation

A college can have many departments CS Department, Accounts Department etc.

All these departments together makes a college.

Encapsulation - code

Example (Understanding)

```
# Define a class with encapsulation
class Car:
    def __init__(self, brand, model):
        self.__brand = brand  # Private variable
        self.__model = model  # Private variable

    # Method to set the brand
    def set_brand(self, brand):
        self.__brand = brand

    # Method to get the brand
    def get_brand(self):
        return self.__brand
```

Encapsulation - code (Contd...)

Example (Understanding)

```
    # Method to display car info
    def display_info(self):
        print(f"Car Brand: {self.__brand}")
        print(f"Car Model: {self.__model}")

my_car = Car("Toyota", "Camry")

# Access public method
my_car.display_info()

# Modify brand using setter method
my_car.set_brand("Honda")

# Access the updated brand using getter method
print(f"Updated Car Brand: {my_car.get_brand()}")
```

Abstraction

- Abstraction involves hiding the complex implementation details and exposing only the essential features of the object.
- This helps in reducing complexity and allows the programmer to focus on higher-level functionalities.

Abstraction (Contd...)

Abstraction

With a laptop you can do many things like play games, watch movies, editing etc.

It doesn't show the inside process of how its doing the things. Implementation parts are hidden

Abstaction - code

Example (Understanding)

```
from abc import ABC, abstractmethod

# Define an abstract class
class Car(ABC):

    # Abstract method (this must be implemented by any subclas
    @abstractmethod
    def fuel_efficiency(self):
        pass

    def start_engine(self):
        print("The car's engine is starting.")

# Define a subclass for Electric Car
class ElectricCar(Car):
```

Abstaction - code (Contd...)

Example (Understanding)

```
    # Implement the abstract method
    def fuel_efficiency(self):
        print("Electric car has high energy efficiency.")

# Define a subclass for Petrol Car
class PetrolCar(Car):

    # Implement the abstract method
    def fuel_efficiency(self):
        print("Petrol car has moderate fuel efficiency.")

# Create objects of the subclasses
electric_car = ElectricCar()
petrol_car = PetrolCar()
```

Abstaction - code (Contd...)

Example (Understanding)

```
# Call the methods
electric_car.start_engine()
electric_car.fuel_efficiency()

petrol_car.start_engine()
petrol_car.fuel_efficiency()
```

Inheritance

Inheritance is a mechanism by which one class can inherit the attributes and methods of another class. This promotes code reusability.

- **Subclass** or **Derived class**: Inherits from another class.
- **Parent class** or **Base class**: The class being inherited from.

Inheritance (Contd...)

Inheritance

The cats can have same colour, same size, same name

but they are not same they are identical.

Inheritance - code

Example (Understanding)

```
# Define a parent class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound")

# Define a child class that inherits from Animal
class Dog(Animal):
    def __init__(self, name, breed):
        # Call the parent class constructor
        super().__init__(name)
        self.breed = breed
```

Inheritance - code (Contd...)

Example (Understanding)

```
    # Overriding the speak method
    def speak(self):
        print(f"{self.name} barks!")

# Create an object of the Dog class
my_dog = Dog("Buddy", "Golden Retriever")

# Access methods from both the parent and child class
my_dog.speak()
```

Polymorphism

Polymorphism means "many forms." It allows methods to do different things based on the object calling them.

- **Method Overriding**: Same method name, but different behavior in subclasses.
- **Method Overloading**: Same method name, but different parameters.

Polymorphism (Contd...)

"A boy starts love with the word friendship, but girl ends love with the same word friendship".

Word is the same but the attitude is different. This beautiful concept of oop is nothing but polymorphism.

Polymorphism - code

Example (Understanding)

```
# Define a class for Car
class Car:
    def start(self):
        print("Car is starting.")

# Define a class for ElectricCar that inherits from Car
class ElectricCar(Car):
    def start(self):
        print("Electric car is starting silently.")

# Define a class for PetrolCar that inherits from Car
class PetrolCar(Car):
    def start(self):
        print("Petrol car is starting with a roar.")
```

Polymorphism - code (Contd...)

Example (Understanding)

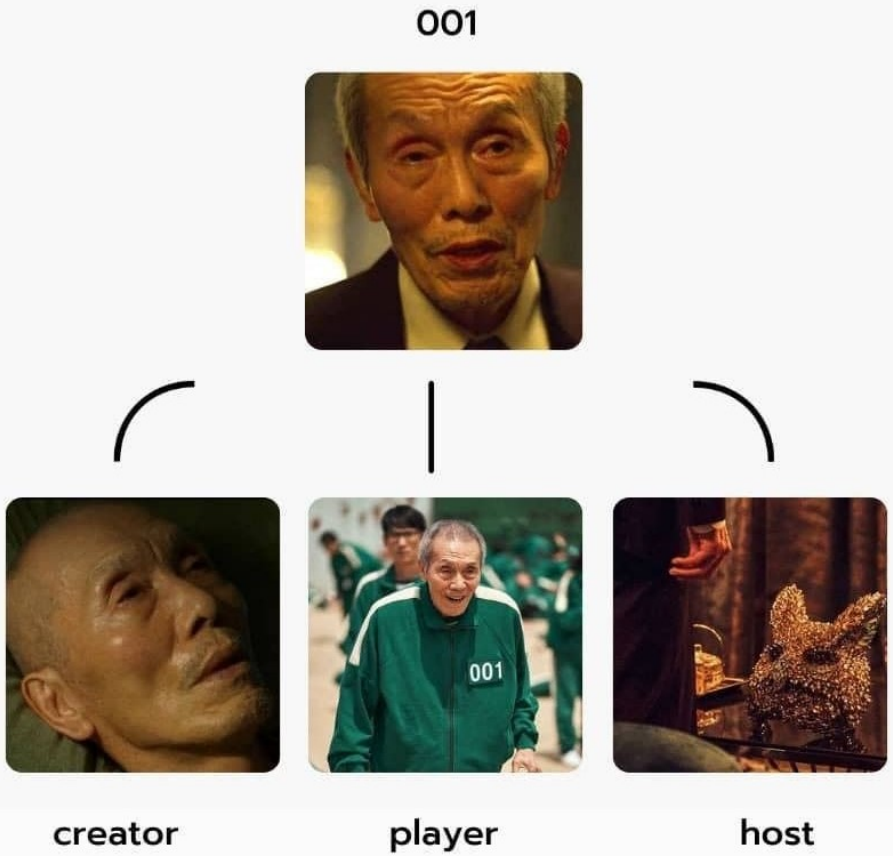
```
# Create objects of the subclasses
car1 = ElectricCar()
car2 = PetrolCar()

# Call the start method for both objects
# (polymorphism in action)
car1.start()
car2.start()
```

Squid Game - Polymorphism

1 . Polymorphism

Polymorphism is the ability to exit in many forms

The old man played roles in many forms as the creator , player and host of the game



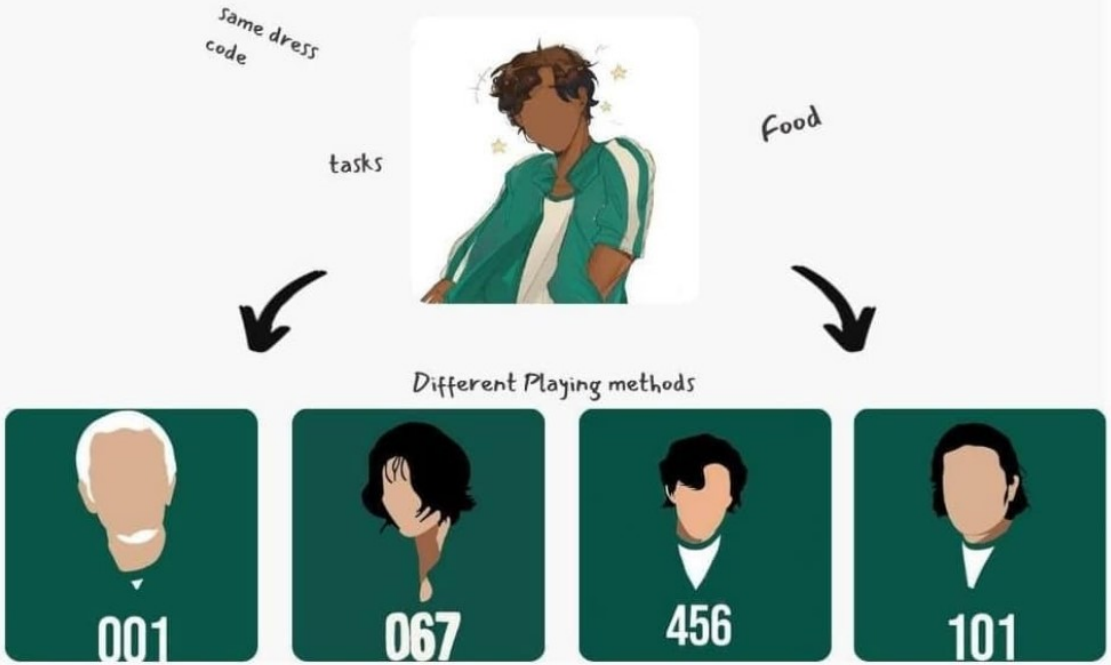
Squid Game - Inheritance

2 . Inheritance

Inheritance allows classes to inherit common properties from the parent class

Players :

All Players inherit the same parent properties like dress code,food,tasks.With that we can have different Players with different playing strategical methods



Squid Game - Encapsulation

3. Encapsulation

Encapsulation means it binds data and code together into one unit.

Squid Game itself is a big Encapsulation. Combination of Frontman, VIP, Host, Players, Money (consider as **variables**), Tasks(consider as **functions**) makes the Squid Game

Squid Game - Abstraction

5/5

4. Abstraction

Abstraction displays only the important information by hiding the implementation part

Tasks :

Though everyone performs the tasks no one knows the process behind them . From soldiers , moto to the behind the scenes of preparing everything for the task, which are all hidden.

JAVA Language

Introduction to Java

- Java is a high-level, object-oriented, and platform-independent programming language that is widely used for building web applications, mobile applications (especially Android), and enterprise-level solutions.
- It was developed by Sun Microsystems (now owned by Oracle) and released in 1995.
- Java is known for its reliability, security, and portability.
- Java is also used in cloud computing and IoT applications, thanks to its cross-platform capabilities.

Who Invented Java?

Java was invented by **James Gosling** in **1991** at **Sun Microsystems**, later acquired by **Oracle Corporation**. He is known as the "father of Java," with key contributions from **Mike Sheridan** and **Patrick Naughton** as part of the Green Team.

Why Was Java Invented?

- **Platform Independence:** Java was designed to be platform-independent, allowing applications to run on any device without modification, using bytecode and the Java Virtual Machine (JVM).
- **Internet Integration:** Java enabled the creation of web-based applications, especially for interactive content on the growing internet.
- **Security:** Java incorporated security features, such as bytecode verification and sandboxing, to allow secure execution of code over the internet.
- **Simplicity and Reliability:** Java aimed to be simpler and more reliable than languages like C++, eliminating complexities like manual memory management.
- **Object-Oriented Programming (OOP):** Java was designed with OOP principles to promote modular, reusable, and maintainable code.

Features of JAVA Language

Features of Java Language

- **Simple:** Java is designed to be easy to use and removes the complexity of C and C++ (e.g., no explicit pointers or operator overloading).
- **Object-Oriented:** Everything in Java is treated as an object, supporting key OOP principles like inheritance, polymorphism, abstraction, and encapsulation.
- **Platform-Independent:** Java follows the "Write Once, Run Anywhere" principle, with code compiled into platform-independent bytecode.
- **Distributed Computing:** Java supports distributed computing with built-in libraries like Java RMI and JavaBeans for networked applications.
- **Multithreaded:** Java provides built-in support for multithreading to execute multiple tasks simultaneously.

Features of Java Language (Contd.)

- **High Performance:** Java achieves high performance with Just-In-Time (JIT) compilation that optimizes bytecode during execution.
- **Security:** Java provides a secure environment through features like a security manager, preventing unauthorized access to system resources.
- **Robust:** Java is designed to be reliable, with features like automatic memory management, exception handling, and type checking.
- **Portable:** Java code is platform-independent and can run on any device with a Java Virtual Machine (JVM).
- **Dynamic:** Java adapts to changing environments and supports runtime class loading and flexible features.

Features of Java Language (Contd.)

- **Rich Standard Library:** Java provides a comprehensive API with utilities for networking, data structures, I/O, GUI, and more.
- **Garbage Collection:** Java includes automatic memory management and garbage collection to prevent memory leaks.
- **Multiplatform Support:** Java runs on various platforms like desktops, Android devices, and embedded systems.
- **Network-Centric:** Java offers powerful networking APIs for building internet-based applications.
- **Backward Compatibility:** Java ensures compatibility with older versions, allowing newer versions to run legacy applications.

Java Ecosystems

Java Ecosystems?

- The reason JVM, JDK, JRE, and other tools are called the Java ecosystem is because they all interact together to provide a complete solution for Java development and execution.
- Just like an ecosystem in nature, each part of the Java ecosystem plays a role, and they work together to provide a robust, cross-platform, and secure environment for building and running applications.
- The Java ecosystem encompasses the JVM (The engine that allows Java programs to be executed on different platforms.), the JDK (The toolkit for developing Java applications.), and the JRE (The environment required to run Java applications.).
- The combination of these components, along with frameworks, libraries, and tools, allows Java to thrive as a powerful platform for application development across various domains.

JDK - JRE - JVM - JIT

- JDE: Workspace for building Java programs.
- JDK: Toolbox with everything needed to create and run Java programs.
- JVM: Machine that runs Java programs by understanding bytecode.
- JIT: Speed booster that makes Java programs run faster by compiling bytecode on the fly.

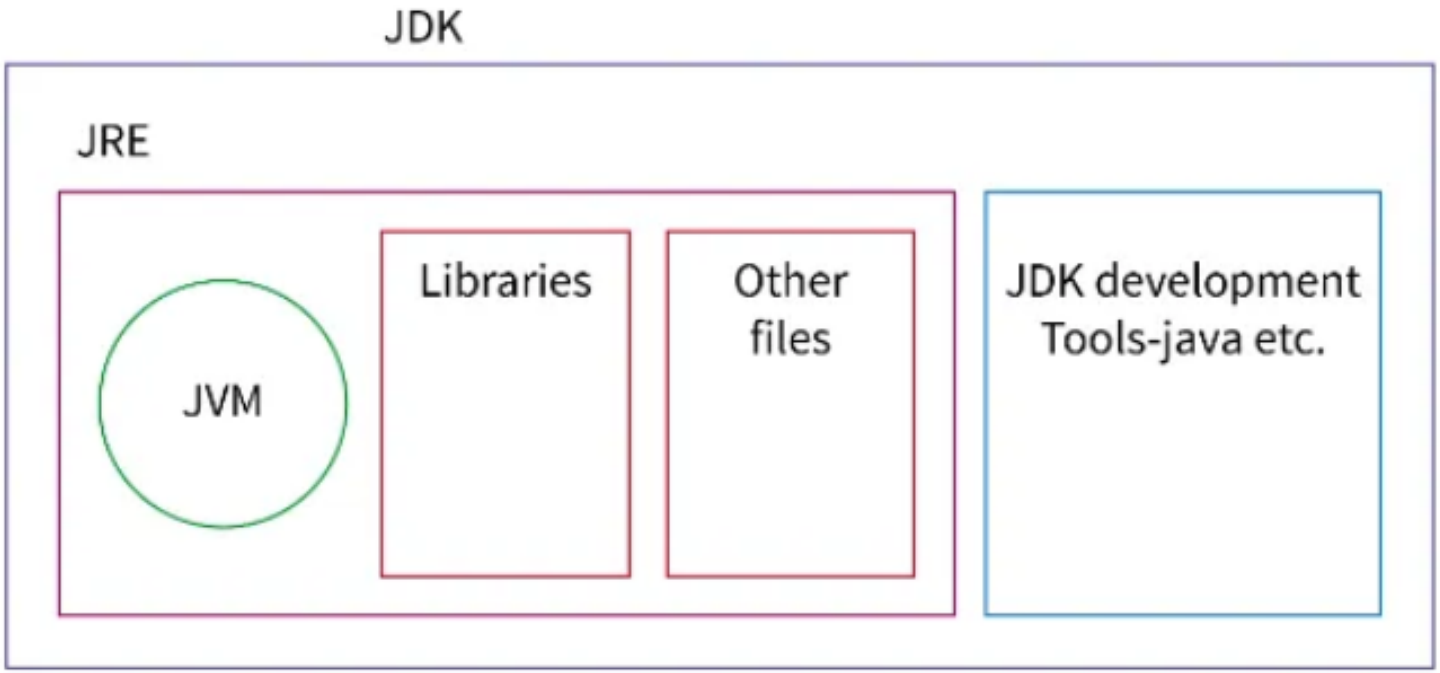
JDK - JRE - JVM - JIT (Contd...)

Java Development Kit (JDK):

- Complete development package for Java applications.
- JDK stands for Java Development Kit
- It is used to build and develop the java program
- It internally contains JRE
- It contains the compiler and debugger
- It contains all the related set of libraries and files to build and compile the java program
- Without JDK we can't build any java program

JDK Architecture

- Compilation, Execution and Debugging



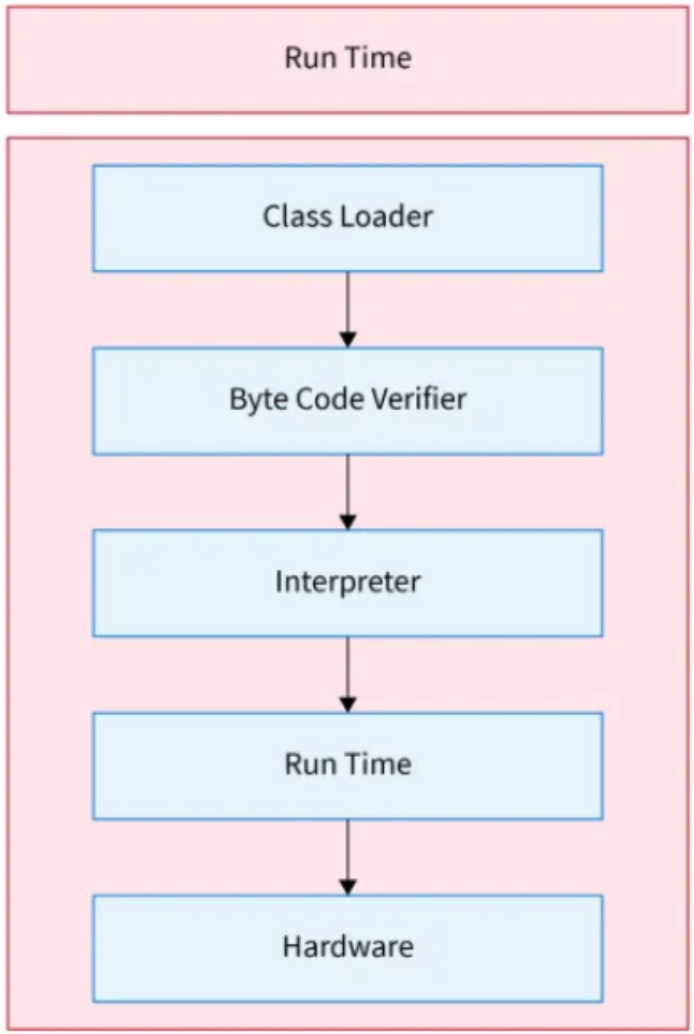
JDK - JRE - JVM - JIT (Contd...)

Java Runtime Environment (JRE):

- JRE stands for Java Runtime Environment
- JRE is the responsible unit to run the java program
- Without JRE we can't run java program (JDK is used to build java program, where as JRE is used to run java program. Such that without JDK, I mean with only JRE we can run the java program without JDK)
- JRE contains JVM
- JRE contains all the inbuilt packages and library files (lang, io, util, etc. all packages are present in JRE)

Working Process of JRE

- file.java - file.class - JVM

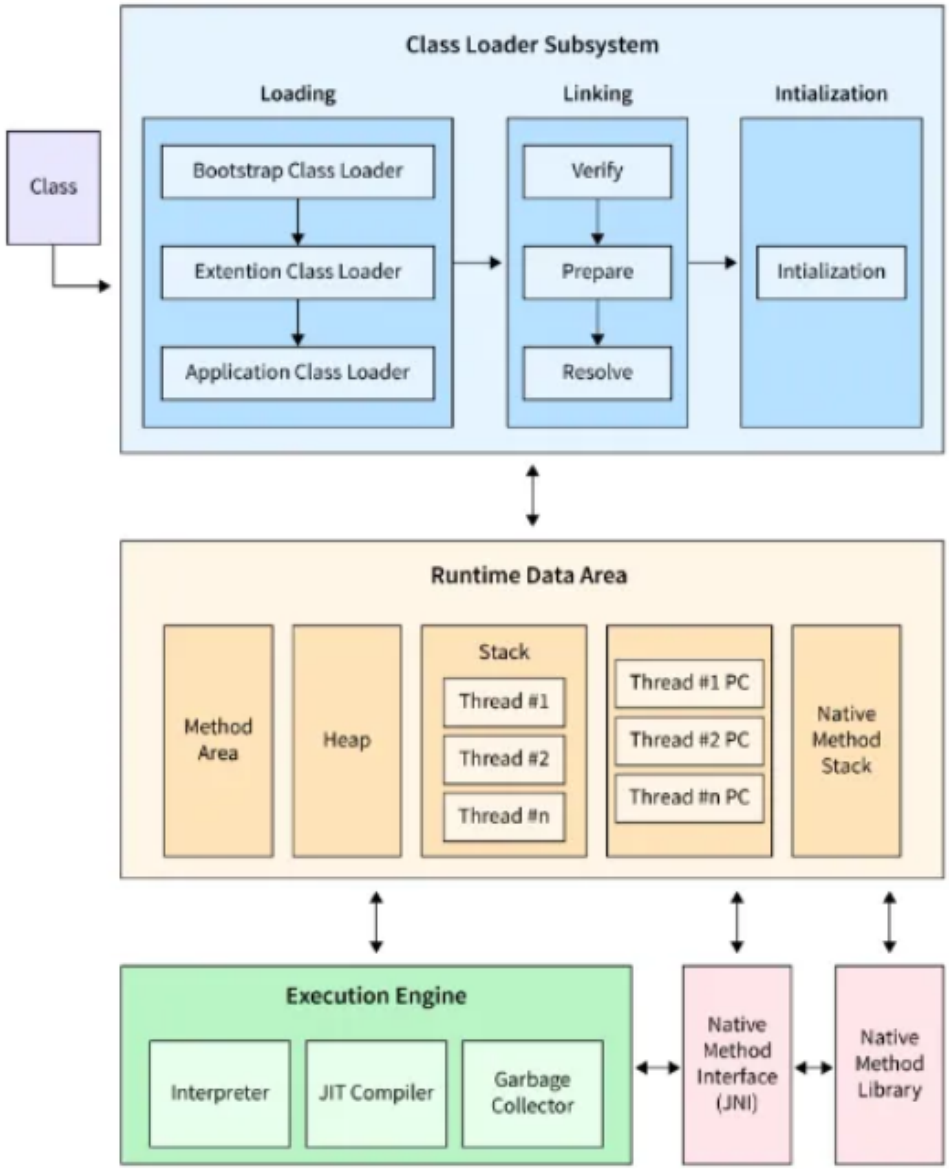


JDK - JRE - JVM - JIT (Contd...)

Java Virtual Machine (JVM):

- JVM stands for Java Virtual Machine
- JVM is platform independent
- JVM is responsible for converting the byte code to machine code.
- JVM takes (.class) files and executes it by managing the memory
- JVM contains JIT
- JVM loads, verifies and executes the code and provides the runtime environment
- JVM plays a major role in java memory management
- It is known as a virtual machine as it is not present physically.

JVM Architecture

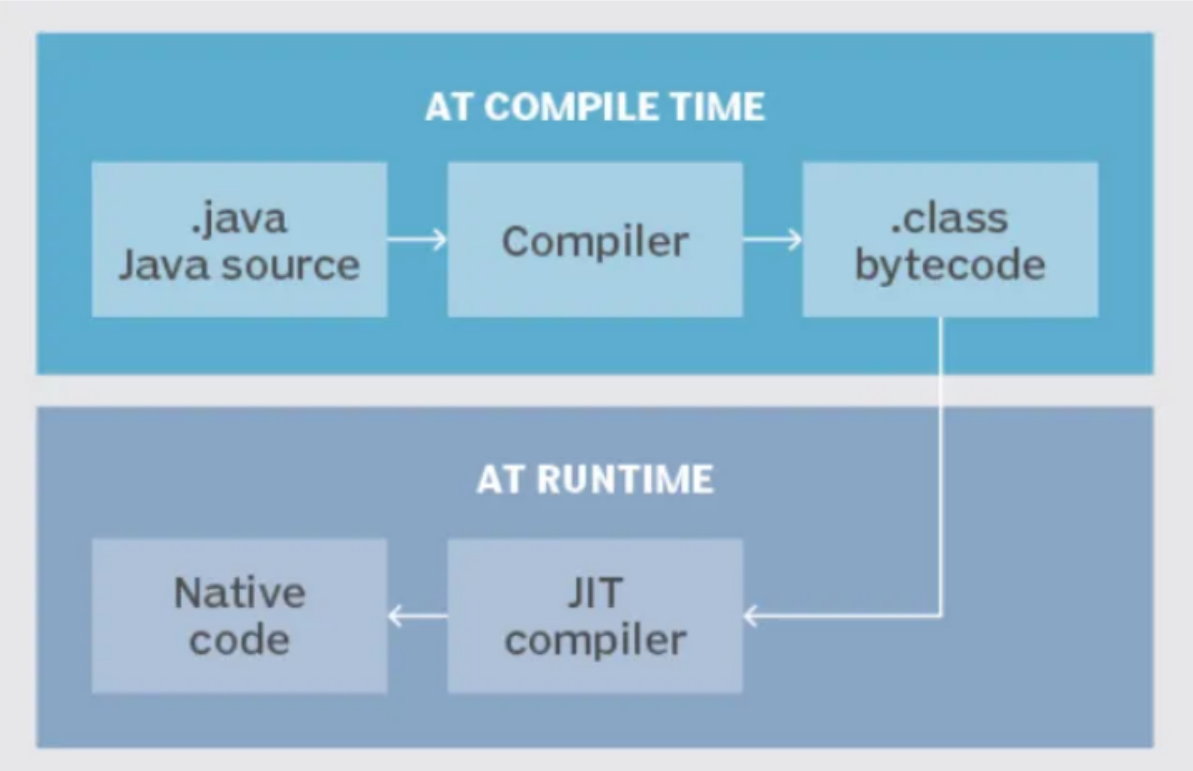


JDK - JRE - JVM - JIT (Contd...)

Java In Time (JIT):

- JIT stands for Just In Time compiler
- Java was a interpreted programming language, but after introducing JIT it was called as interpreted-compiled programming language
- JIT increased the speed of execution
- JIT helps the JVM to find the active OS (Operating System)
- JIT was introduced from 1.2 JDK

A visual of how a just-in-time (JIT) compiler works



Comparison: JVM vs JDK vs JRE

Component	Purpose	Used By
JVM	Executes Java bytecode	Developers and runtime environments
JRE	Provides environment to run Java apps	End-users running Java applications
JDK	Provides tools for Java development	Developers

Bytecode

What is Bytecode in Java?

- **Bytecode** is an **intermediate code** generated by the Java compiler after compiling a Java source file (.java).
- It is platform-independent and stored in a file with the `.class` extension.

How Bytecode Works

1. **Source Code Compilation:**
   - Write a Java program (e.g., `HelloWorld.java`).
   - Compile using `javac HelloWorld.java`.
   - Bytecode file generated: `HelloWorld.class`.

2. **Bytecode Execution:**
   - JVM reads and executes the bytecode.
   - Converts it into machine-specific instructions.

Why Use Bytecode?

- **Platform Independence:** Runs on any OS with JVM.
- **Security:** JVM ensures secure execution.
- **Performance:** JIT compilation boosts speed.
- **Portability:** Java's "Write Once, Run Anywhere" feature.

Example of Bytecode Generation

Example (HelloWorld.java)

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
# Generated Bytecode (After Compilation):
# HelloWorld.class (Sample Bytecode)
ca fe ba be 00 00 00 34 00 0d 0a 00 03 00 0a 07 ...
```

Summary

- Bytecode is an intermediate platform-independent code.
- It enables Java's **"Write Once, Run Anywhere" (WORA)** feature.
- Bytecode is executed by the JVM for secure, portable, and optimized execution.

Syntax

Syntax format - JAVA

Example (JAVA code)

```
// This java code is for displaying - 'hello world!'

/* As this is your first code in java,
unlike python programming language, it has
more than 1line to print */

public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello, World!");
    }
}
```

1. Comments

Example (JAVA code)

```
// This is a single-line comment
{ Used for short notes or explanations.
/* This is a multi-line comment */
{ Used for longer explanations or block comments.
```

2. Class declaration

Example (JAVA code)

```
public class HelloWorld {
```

- public: Means this class can be used from anywhere.
- class: Defines a class (a blueprint to create objects). HelloWorld: The name of the class. It can be anything, but in Java, the file name must match this class name (HelloWorld.java).

3. Main method declaration

Example (JAVA code)

```
public static void main(String[] args) {
```

- public: Means the method can be accessed from anywhere.
- static: Means it belongs to the class itself (no need to create an object to use it).
- void: Means the method does not return anything.
- main: The name of the method. This is fixed and required.
- String[] args: This is used to take input from the user (not needed in this simple program).

4. Printing console

Example (JAVA code)

```
System.out.println("Hello, World!");
```

- System: A built-in Java class that interacts with the system.
- out: Refers to the output stream (screen).
- println(): Means "print a line" and move to the next line after printing.

Key Rules of Java Syntax

- Case Sensitivity: Java is case-sensitive (System and system are different).
- File Name: The class name should match the file name.
- Class Structure: Every Java program must have at least one class.
- Main Method: The execution starts from the main method.
- Semicolons: Each statement must end with a semicolon (;).