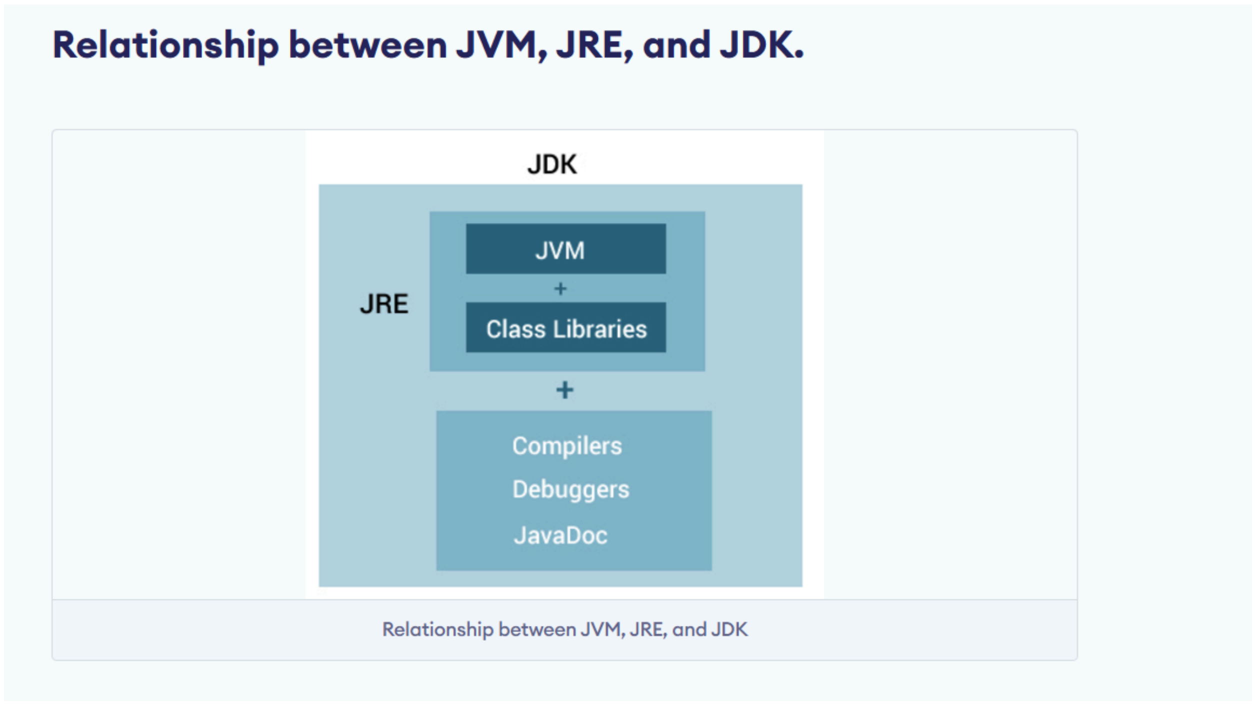
JVM architecture


Relationship between JVM, JRE, and JDK.
JDK
JRE
JVM
+
Class Libraries
+
Compilers
Debuggers
JavaDoc
Relationship between JVM, JRE, and JDK

JDK (Java Development Kit) = JRE (Java Runtime Environment) + Development Tools

- Development Tools include the Java compiler (javac), debugger, and other utilities required for Java development.

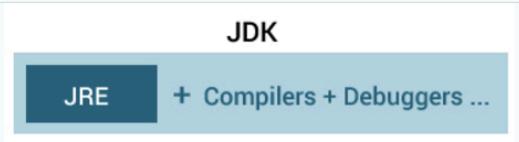
JRE (Java Runtime Environment) = JVM (Java Virtual Machine) + Libraries

- JVM: The engine that runs Java bytecode (responsible for execution).
- Libraries: Pre-written classes and methods (like java.util, java.io, etc.) that Java programs use.

What is JDK?

JDK (Java Development Kit) is a software development kit required to develop applications in Java. When you download JDK, JRE is also downloaded with it.

In addition to JRE, JDK also contains a number of development tools (compilers, JavaDoc, Java Debugger, etc).

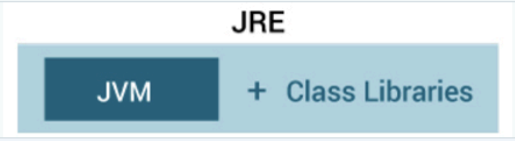


Java Development Kit

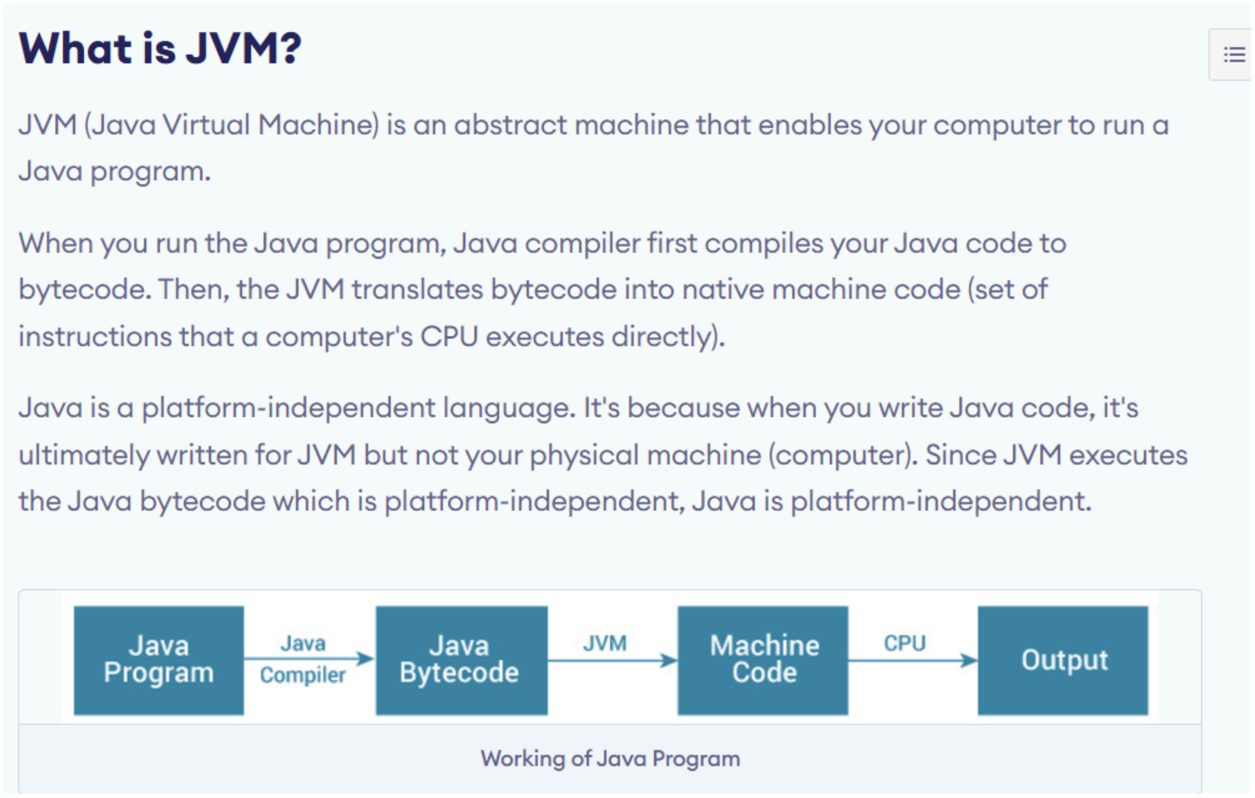
What is JRE?

JRE (Java Runtime Environment) is a software package that provides Java class libraries, Java Virtual Machine (JVM), and other components that are required to run Java applications.

JRE is the superset of JVM.



Java Runtime Environment



What is JVM?

JVM (Java Virtual Machine) is an abstract machine that enables your computer to run a Java program.

When you run the Java program, Java compiler first compiles your Java code to bytecode. Then, the JVM translates bytecode into native machine code (set of instructions that a computer's CPU executes directly).

Java is a platform-independent language. It's because when you write Java code, it's ultimately written for JVM but not your physical machine (computer). Since JVM executes the Java bytecode which is platform-independent, Java is platform-independent.

$$\begin{tikzcd}\text{Java Program} \arrow[r, "\text{Java Compiler}"] & \text{Java Bytecode} \arrow[r, "\text{JVM}"] & \text{Machine Code} \arrow[r, "\text{CPU}"] & \text{Output}\end{tikzcd}$$

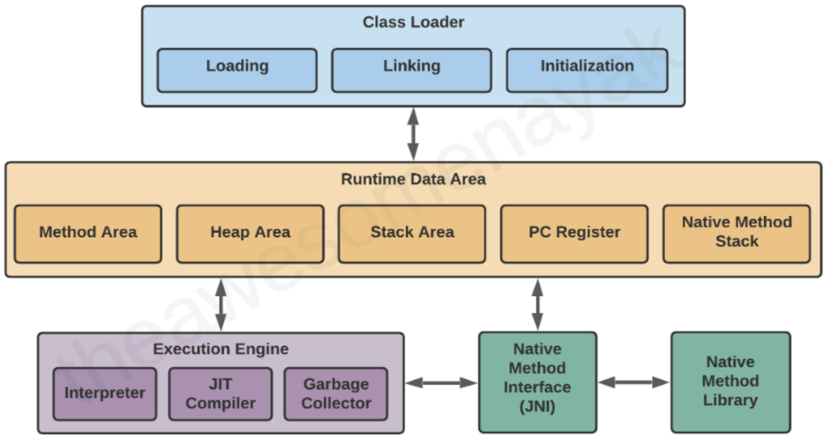
Working of Java Program

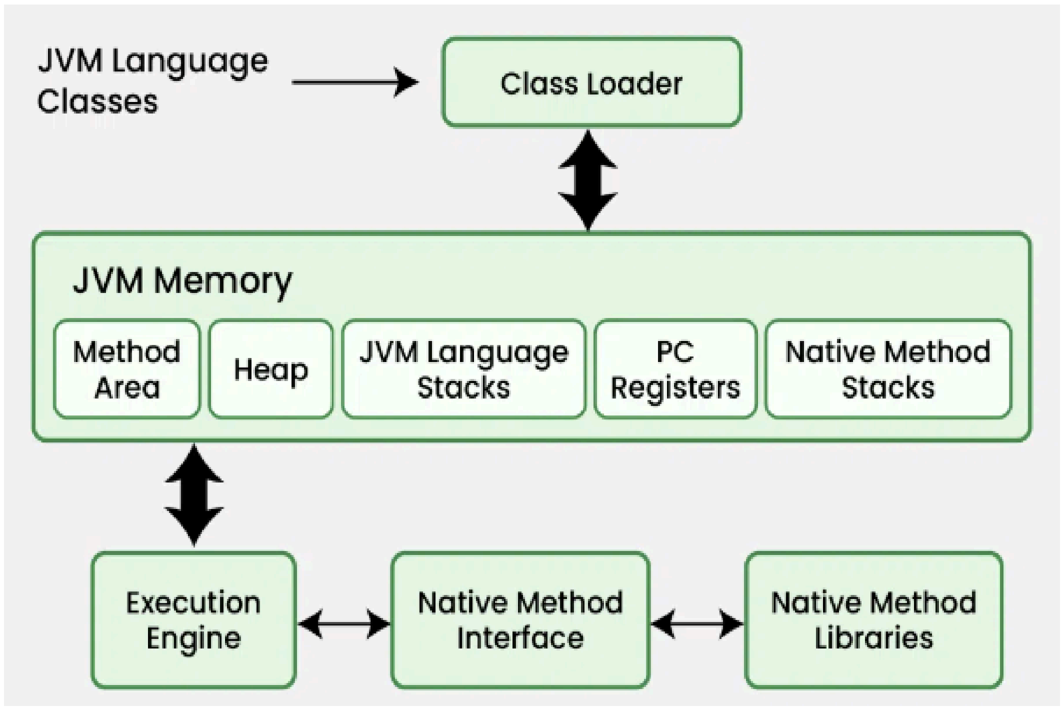
What is JVM?

- The JVM provides a runtime environment to execute Java applications.
- It converts Java bytecode into machine language. JVM is a part of Java Runtime Environment (JRE).

Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and expect it to run on any other Java-enabled system without any adjustment. This is all possible because of JVM.

When we compile a .java file, the Java compiler generates .class files containing bytecode for each class defined in the .java file. When we run the .class file, it goes through several steps, which together define how the JVM (Java Virtual Machine) works.





Class Loader

When you compile a .java source file, it is converted into byte code as a .class file. The JVM's class loader loads the .class file into memory.

There are three phases in the class loading process: loading, linking, and initialization.

1) **Loading:**

The Class loader reads the "*.class*" file, generates the corresponding binary data and saves it in the method area. For each "*.class*" file, JVM stores the following information in the method area.

- **The fully qualified name of the loaded class and its immediate parent class.**
- **Whether the "*.class*" file is related to Class or Interface or Enum.**
- **Modifier, Variables and Method information etc**.

2) **Linking**

Performs verification, preparation, and (optionally) resolution.

- ***Verification***: It ensures the correctness of the *.class* file i.e. it checks whether this file is properly formatted and generated by a valid compiler or not. If verification fails, we get a run-time exception *java.lang.VerifyError*. This activity is done by the component ByteCodeVerifier. Once this activity is completed then the class file is ready for compilation.
- ***Preparation***: JVM allocates memory for class static variables and initializes the memory to default values.
- ***Resolution***: It is the process of replacing symbolic references from the type with direct references. It is done by searching into the method area to locate the referenced entity.

3) **Initialization**

In this phase, all static variables are assigned with their values defined in the code and static block(if any). This is executed from top to bottom in a class and from parent to child in the class hierarchy. In general, there are three class loaders:

1. **Bootstrap ClassLoader**: This is the first classloader which is the super class of Extension classloader. It loads the *rt.jar* file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes etc.
2. **Extension ClassLoader:** This is the child classloader of Bootstrap and parent classloader of System classloader. It loades the jar files located inside the *$JAVA_HOME/jre/lib/ext* directory.

3. **System/Application ClassLoader**: This is the child classloader of Extension classloader. It loads the class files from classpath. By default, classpath is set to the current directory. You can change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.

```
// Java code to demonstrate Class Loader subsystem
public class Test {
    public static void main(String[] args)
    {
        // String class is loaded by bootstrap loader, and
        // bootstrap loader is not Java object, hence null
        System.out.println(String.class.getClassLoader());


        // Test class is loaded by Application loader
        System.out.println(Test.class.getClassLoader());
    }
}
```

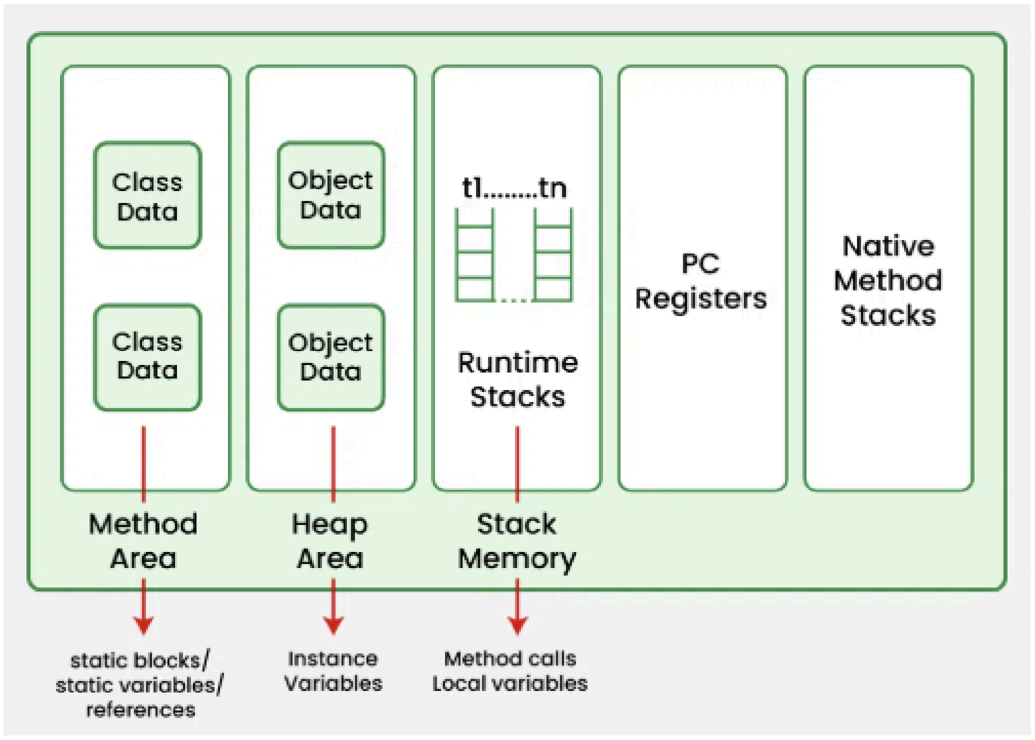
Output

```
null
jdk.internal.loader.ClassLoaders$AppClassLoader@8bcc55f
```

JVM Memory Areas

- **Method area:** In the method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

- **Heap area:** Information of all **objects** is stored in the heap area. There is also one Heap Area per JVM. It is also a shared resource.
- **Stack area:** For every thread, JVM creates one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which stores methods calls. All **local variables** of that method are stored in their corresponding frame. After a thread terminates, its run-time stack will be destroyed by JVM. It is not a shared resource.
- **PC Registers:** Store address of current execution instruction of a thread. Obviously, each thread has separate PC Registers.
- **Native method stacks:** For every thread, a separate native stack is created. It stores native method information.



Execution Engine

Execution engine executes the "*.class*" (bytecode). It reads the byte-code line by line, uses data and information present in various memory areas and executes instructions. It can be classified into three parts:

- ***Interpreter***: It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
- ***Just-In-Time Compiler(JIT)*** :

  The JIT Compiler overcomes the disadvantage of the interpreter. The Execution Engine first uses the interpreter to execute the byte code, but when it finds some repeated code, it uses the JIT compiler . It is used to increase the efficiency of an interpreter. It compiles the entire bytecode and changes it to native code so whenever the interpreter sees repeated method calls, JIT provides direct native code for that part so re-interpretation is not required, thus efficiency is improved.

- ***Garbage Collector***: It destroys un-referenced objects. For more on Garbage Collector, refer Garbage Collector.

Java Native Interface (JNI)

Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.

6. Native Method Libraries

These are collections of native libraries required for executing native methods. They include libraries written in languages like C and C++.

Interview Questions

Explain the architecture of JVM.

- JVM consists of:
    - **ClassLoader**: Loads class files.
    - **Memory Area**: Method area, heap, stack, program counter, native method stack.
    - **Execution Engine**: Interprets or compiles bytecode into machine code.
    - **Garbage Collector**: Manages memory by removing unused objects

What are the components of the JVM memory model?

- **Heap**: Stores objects and class-level variables.
- **Method Area**: Stores class-level data like runtime constant pool, field, method data, and constructor code.
- **Stack**: Stores local variables, method call information.
- **Program Counter (PC) Register**: Holds the address of the JVM instruction being executed.
- **Native Method Stack**: Stores native method information.

What is the role of ClassLoader in JVM?

- It loads .class files into JVM, breaking it into three parts: **Bootstrap ClassLoader**, **Extension ClassLoader**, and **Application ClassLoader**.

Explain the different memory areas in JVM.

- **Heap**: Object memory.
- **Stack**: Thread-specific memory for method invocations.
- **Method Area**: Stores class-related data.
- **PC Register**: Keeps track of JVM instruction addresses.
- **Native Method Stack**: Used for native (non-Java) code.

What is the difference between Stack and Heap memory?

- **Stack**: Stores local variables and method calls; operates on LIFO.
- **Heap**: Stores objects; shared among all threads.

How does JVM handle method invocation?

- JVM uses the **Stack** memory for method invocation, creating a new stack frame for every method call.

What is the Execution Engine in JVM?

- Converts bytecode to machine-specific code, consisting of:
    - **Interpreter**: Executes bytecode line by line.

    - **JIT Compiler**: Converts bytecode into native machine code for high performance.
    - **Garbage Collector**: Manages memory deallocation.

What is JIT (Just-In-Time) Compilation?

- Part of the execution engine, **JIT** compiles frequently used bytecode to machine code at runtime for improved performance.

How does JVM manage memory?

- JVM uses **Garbage Collection** to automatically free memory by identifying and deleting unused objects in the heap.

What are the different types of ClassLoaders?

- **Bootstrap ClassLoader**: Loads core Java classes (java.lang.*).
- **Extension ClassLoader**: Loads classes from the Java extension libraries.
- **Application ClassLoader**: Loads application-specific classes from the classpath.

What is the role of the Garbage Collector in JVM?

- It automatically reclaims memory by removing objects that are no longer in use, ensuring efficient memory management in the heap.

What are the phases of Garbage Collection in JVM?

- **Mark**: Identifies objects that are in use.
- **Sweep**: Removes objects that are not marked.
- **Compact**: Reorganizes memory by moving active objects together.

Can you manually trigger Garbage Collection in Java?

- You can request garbage collection using System.gc(), but there is no guarantee that it will run immediately.

What is the role of the Program Counter (PC) Register in JVM?

- PC Register holds the address of the next instruction that the JVM will execute, maintaining the execution flow for each thread.

What happens during JVM startup?

- The JVM:
    - Loads the main class.
    - Uses the ClassLoader to load classes.
    - Initializes class and instance variables.
    - Executes the main() method of the class.

What are the different types of Garbage Collectors in JVM?

- **Serial GC**: Suitable for single-threaded applications.
- **Parallel GC**: Uses multiple threads for garbage collection.
- **CMS GC (Concurrent Mark Sweep)**: Minimizes pauses by doing most work concurrently.
- **G1 GC (Garbage First)**: Default collector in Java 9+, divides the heap into regions for better performance.

How does the JVM handle multithreading?

- JVM creates separate stacks for each thread, and manages synchronization through monitors and locks to ensure thread safety.

Important Flow

```
package com.basic;
class Car {
        String model;
        int year;
        // Constructor to initialize object
        public Car(String model, int year) {
                this.model = model;
                this.year = year;
        }
        // Method to display car information
        public void displayInfo() {
                System.out.println("Model: " + model + ", Year: " + year);
        }
}
public class JdkJreJvmFlow {
        public static void main(String[] args) {
                // Creating an object of Car
                Car myCar = new Car("Toyota", 2022);
                // Calling the method on the object
                myCar.displayInfo();
        }
}
```

Output

```
Model: Toyota, Year: 2022
```

JDK,JRE,JVM level flow execution

Let's break down the detailed information for JDK, JRE, and JVM using the above example (Main.java and Car.java).

JDK (Java Development Kit)

What is the JDK?

The JDK is a software development kit used to develop Java applications. It contains:

- Compiler (javac): Converts Java source code into bytecode.
- Libraries: Provides core Java libraries such as java.lang, java.util, etc.
- JVM: The Java Virtual Machine, which executes the bytecode.

How the JDK works in our example:

1. You write your Java program in the files Main.java and Car.java on your development machine.

You use the JDK's compiler (javac) to convert your source code (which is human-readable Java code) into bytecode that the JVM can execute.
For example:
bash
Copy code
```
javac Main.java Car.java
```

2. After running this command, the compiler generates the bytecode .class files:
   - Main.class
   - Car.class

Summary:

The JDK provides all the tools necessary for writing and compiling your Java program. It turns your .java files (source code) into .class files (bytecode) that can be executed by the JVM.

JRE (Java Runtime Environment)

What is the JRE?

The JRE is a part of the Java Development Kit (JDK) and is the environment required to run Java applications. It includes:

- Java Libraries: Pre-built classes for common functionality (e.g., java.lang, java.util).

- JVM: The component responsible for executing Java bytecode.
- Other Components: Such as the java launcher, which starts the program.

How the JRE works in our example:

1. You have already compiled your Java program (Main.class and Car.class).
2. To run the Java program, you use the JRE, which contains the necessary components to execute the bytecode.

You use the java command to launch the Java program from the command line:
For example:
bash
Copy code

```
java Main
```

3. This command triggers the JRE to:
   - Load the Main.class file (and any dependencies like Car.class).
   - Use the JVM to execute the bytecode and run the program.

Summary:

The JRE provides the environment to run Java programs. It includes the JVM and essential libraries to execute the bytecode created by the JDK.

JVM (Java Virtual Machine)

What is the JVM?

The JVM is a part of the JRE and is responsible for executing Java bytecode. It provides platform independence by abstracting the underlying operating system. The JVM:

- Loads .class files (Java bytecode).
- Verifies the bytecode for security and correctness.
- Executes the bytecode (using either an interpreter or JIT compiler).
- Handles memory management, garbage collection, and other runtime tasks.

How the JVM works in our example:

1. You've run the command java Main, which starts the JVM.
2. The JVM first loads the Main.class file into memory.
3. The JVM verifies the bytecode, ensuring there are no security violations.
4. The JVM executes the Main class, starting the main method, which:
   - Creates an instance of the Car class (Car myCar = new Car("Toyota", 2022)).

    - Calls the displayInfo() method of the Car object to print the car's details.
5. Garbage Collection: After the program finishes execution, the JVM performs garbage collection to reclaim memory used by objects that are no longer in use (like myCar).

Summary:

The JVM is the engine that runs the Java bytecode. It makes sure that the code can run consistently across different operating systems, manages memory, and performs various runtime tasks.

Full Flow Summary (JDK -> JRE -> JVM)

1. **JDK (Development Phase):**
    - You write the source code for the program (Main.java and Car.java).

You compile the source code using javac:

```
javac Main.java Car.java
```

- This generates bytecode files (Main.class and Car.class).

Summary: The JDK provides tools to write and compile your Java code into platform-independent bytecode.

2. **JRE (Execution Phase):**

To run your Java program, you use the java command:

```
java Main
```

- The JRE provides the necessary environment to run the program, including the JVM and essential libraries.

Summary: The JRE allows you to execute the compiled bytecode and ensures that your program runs in a proper environment.

3. **JVM (Execution Phase):**
    - The JVM loads the bytecode (Main.class and Car.class) into memory.
    - It verifies and executes the bytecode, running the main method and handling memory management and garbage collection.

Summary: The JVM runs the program by interpreting or compiling the bytecode into native machine instructions, making the program platform-independent.

Depth JVM flows

1. **Class Loader**:
   The JVM begins by using the **Class Loader** to load the .class files (bytecode).
   - The class loader loads the Main.class file, and if that class references other classes (like Car.class in this case), it will also load those classes.
   - The JVM looks for the Main class in the specified classpath and loads it into memory.
2. **Bytecode Verification**:
   After loading the class files, the JVM performs a **bytecode verification** process:
   - The bytecode is checked for any security violations or invalid instructions.
   - The JVM ensures that the bytecode follows the Java language specifications and does not perform unsafe operations (e.g., accessing restricted memory).
3. **JVM Memory Allocation**:
   After verification, the JVM allocates memory for the program. This includes:
   - **Method Area**: Stores class structures (metadata, constant pool, etc.).
   - **Heap**: Used for dynamic memory allocation of objects (e.g., myCar).
   - **Stack**: Each thread has a stack used for storing local variables and method calls (e.g., method execution context for main and displayInfo).
   - **PC Register**: Stores the address of the next instruction to be executed for the currently running thread.
4. **JVM Execution Starts**:
   The JVM begins execution by invoking the main method of the Main class. It follows these steps:
   Main Method Execution:
   - The main method in the Main.class file is executed.
   - The first step in main is to create a new instance of the Car class. The JVM needs to create an object of type Car (Car myCar = new Car("Toyota", 2022);).
5. **Object Creation (Heap Allocation)**:
   To create a Car object, the JVM performs the following:
   - Allocates memory for the object on the **heap**.
   - Initializes the object fields (model and year) through the constructor Car(String model, int year).
6. **Constructor Execution**:
   The JVM then calls the constructor Car(String model, int year):
   - The constructor initializes the fields of the object.
   - The constructor completes, and control is returned to the main method.
7. **Method Invocation (displayInfo)**:
   After the Car object is created, the main method invokes the displayInfo() method on the myCar object:
   - The JVM loads the bytecode for displayInfo().

    - The JVM checks the stack and pushes the displayInfo method onto the stack (since methods are executed in a LIFO order, the main method execution is paused until displayInfo completes).
8. **Method Execution**:
   The JVM now executes the displayInfo() method, which prints the car's model and year:
    - It accesses the fields model and year of the myCar object and formats the output.

The output is printed to the console:

```
Model: Toyota, Year: 2022
```

    -
    - After the displayInfo() method completes, control is returned to the main method.
9. **Method Return**:
   After printing the car's information, the displayInfo method completes its execution. The JVM removes the method call from the stack, and control returns to the main method.
10. **End of Program Execution**:
    Once the main method finishes execution, the program completes its run. The JVM now begins the process of **garbage collection** for objects that are no longer in use.
11. **Garbage Collection**:
    After the program ends, the JVM checks if any objects are no longer reachable. In this case, the myCar object is no longer needed, so it is eligible for garbage collection.
- The **garbage collector** runs in the background and reclaims memory used by objects like myCar that are no longer referenced.
13. **JVM Shutdown**:
    After garbage collection, the JVM performs any final cleanup and shuts down. This may involve freeing up resources and terminating any remaining threads.