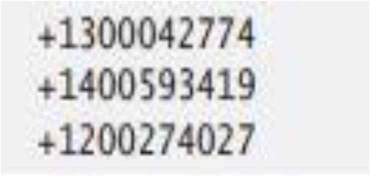
Introduction to Programming Languages

Types of Languages

- Programmers write instructions in various programming languages, some d**irectly understandable by computers** and others **requiring intermediate translation steps**.
- Hundreds of such languages are in use today. These may be divided into three general types:
  - Machine Language
  - Assembly Language
  - High-level Language

Machine Language

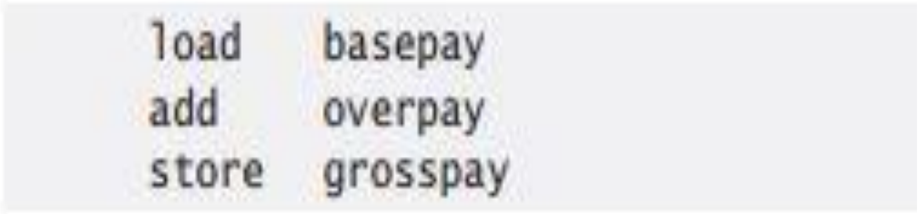
- Any computer can directly understand only its own machine language, defined by its hardware design
- Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time.
- Machine languages are machine dependent (a particular machine language can be used on only one type of computer).
- For example, here’s a section of an early machine-language payroll program that adds overtime pay to base pay and stores the result in gross pay:



```
+1300042774
+1400593419
+1200274027
```

Assembly Language

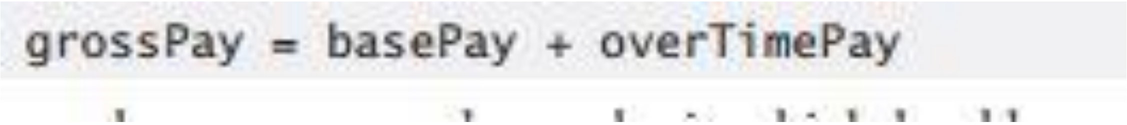
- Programming in machine language was simply too slow and tedious for most programmers.
- Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent elementary operations. These abbreviations formed the basis of assembly languages.
- ***Translator programs* called assemblers** were developed to convert early assembly-language programs to machine language at computer speeds.
- The following section of an assembly-language payroll program also adds overtime pay to base pay and stores the result in gross pay



```
load    basepay
add     overpay
store   grosspay
```

High-level Language

- With the advent of assembly languages, computer usage increased rapidly, but programmers still had to use numerous instructions to accomplish even the simplest tasks.
- To speed the programming process, high-level languages were developed in which single statements could be written to accomplish substantial tasks.
- **Translator programs called compilers** convert high-level language programs into machine language.
- High-level languages allow you to write instructions that look almost like everyday English and contain commonly used mathematical notations.
- A payroll program written in a high-level language might contain a single statement such as



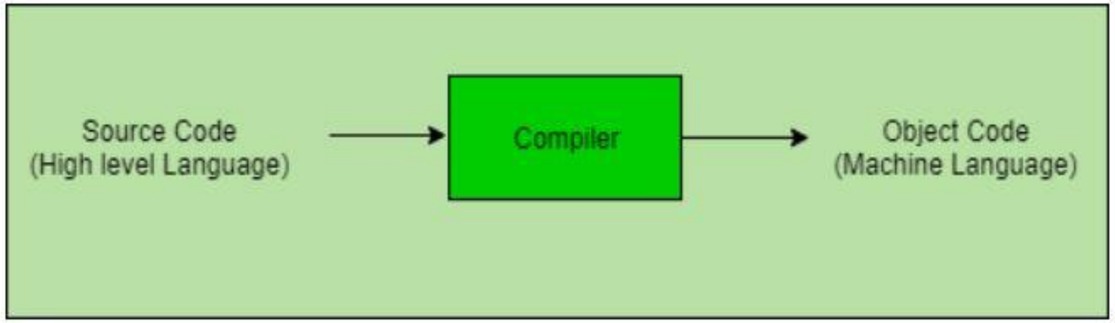
```
grossPay = basePay + overTimePay
```

Translator Programs

- Compiler
- Assembler
- Interpreters

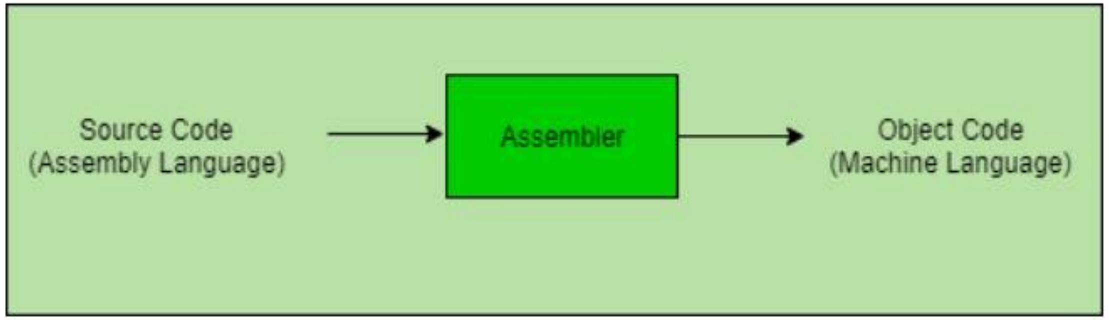
Compiler

- The language processor **that reads the complete source program written in high-level language as a whole in one go** and translates it into an equivalent program in machine language is called a Compiler. Example: C, C++, C#, Java.
- In a compiler, the source code is translated to object code successfully if it is free of errors. The compiler specifies the errors at the end of the compilation with line numbers when there are any errors in the source code.
- The errors must be removed before the compiler can successfully recompile the source code again.
- Example: C, C++, JAVA



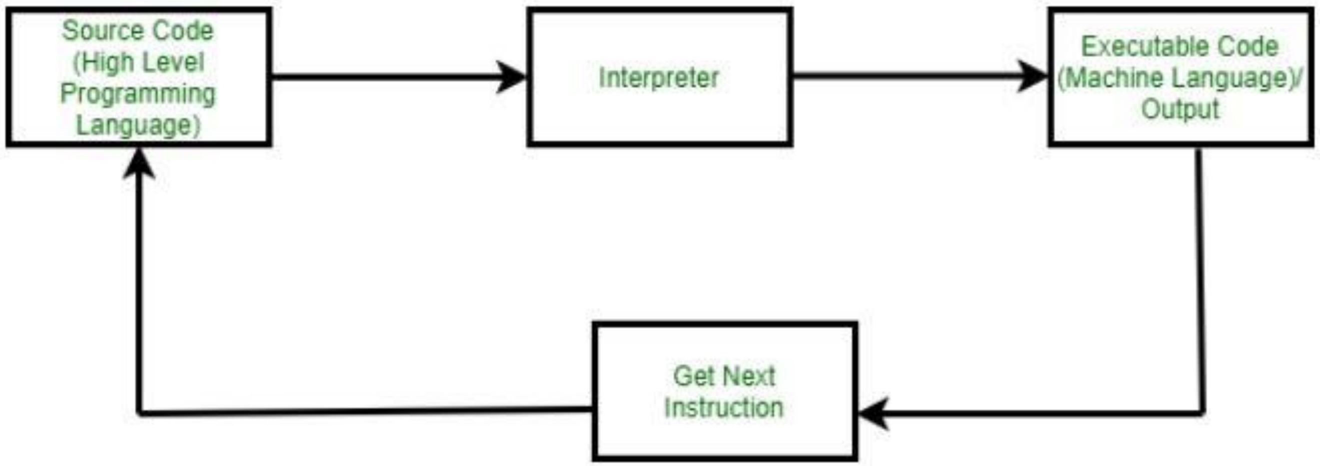
Assembler

- The Assembler is used to **translate the program written in Assembly language into machine code**. The source program is an input of an assembler that contains assembly language instructions.
- The output generated by the assembler is the object code or machine code understandable by the computer.
- Code written in assembly language is some sort of mnemonics(instructions) like **ADD, MUL, MUX, SUB, DIV, MOV, LOAD** and so on. and the assembler is basically able to convert these mnemonics in Binary code.
- Example: MIPS (Million Instructions Per Second)



Interpreter

- The translation **of a single statement of the source program into machine code is done by a language processor** and executes immediately before moving on to the next line is called an interpreter.
- If there is an error in the statement, the interpreter terminates its translating process at that statement and displays an error message.
- The interpreter moves on to the next line for execution only after the removal of the error. An Interpreter directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code.
- Example: Perl, Python, Matlab


Source Code (High Level Programming Language)
Interpreter
Executable Code (Machine Language)/ Output
Get Next Instruction

Structured Programming

- Structured programming is a paradigm that aims to make programs easier to **comprehend from a reader's point of view**. It does this by linearising the flow of control through a program. In structured programming, **execution follows the writing order of the code**.
- Limitations
  - The target of coding is code readability rather than usability
  - A change in one place will affect all other sequential code execution
  - It is difficult to modify and re-use code

Rise of Java

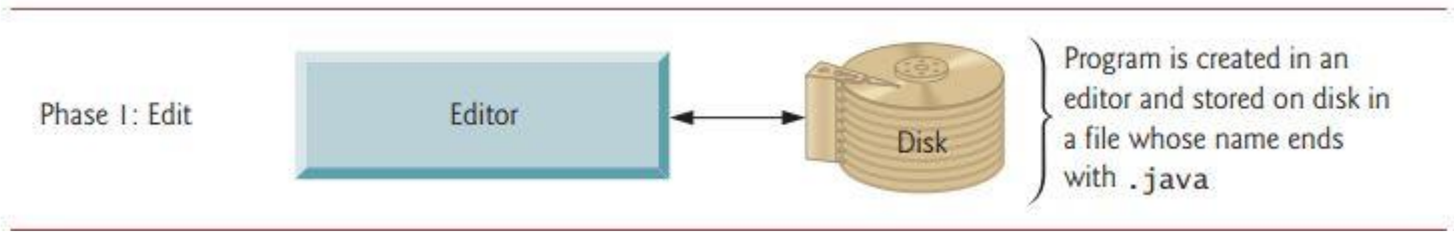
- **Sun Microsystems in 1991** funded an internal corporate research project led by James Gosling, which resulted in a C++- based object-oriented programming language that Sun called Java.
- A key goal of Java is to be able to write programs that will run on a great variety of computer systems and computer-controlled devices. This is sometimes called **“write once, run anywhere.”**

A Typical Java Development Environment

- Normally there are five phases—edit, compile, load, verify and execute.
- Phase 1: Creating a Program
- Phase 2: Compiling a Java Program into **Bytecodes**
- Phase 3: Loading a Program into Memory
- Phase 4: Bytecode Verification
- Phase 5: Execution

Phase 1: Creating a Program

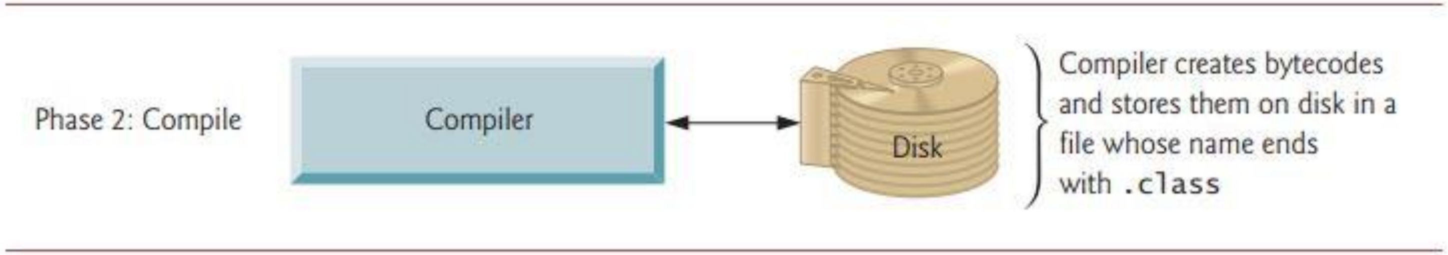
- Phase 1 consists of editing a file with **an editor program**, normally known simply as an editor
- Using the editor, you type a Java program (typically referred to as source code), make any necessary corrections and save it on a secondary storage device, such as your hard drive.
- Java source code files are given a name ending with the .java extension.
- Editors – Netbeans, Eclipse.



Program is created in an editor and stored on disk in a file whose name ends with `.java`

Phase 2: Compiling a Java Program into Bytecodes

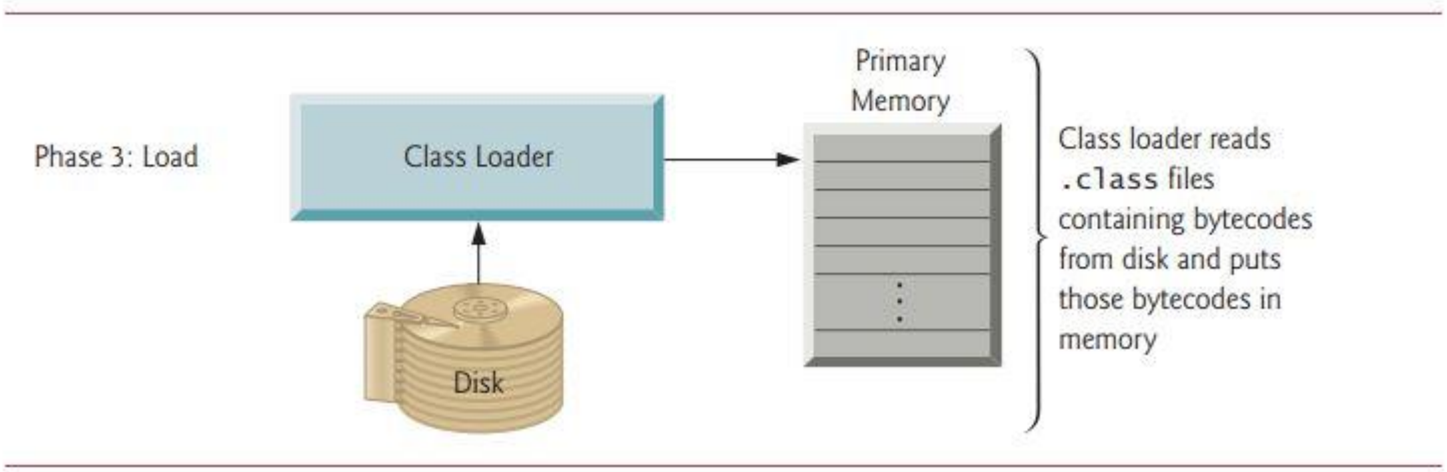
- Compile the already written program using **javac command** line or **editor**.
- If the program compiles, the compiler produces a .class file **containing bytecode.** For example – “Welcome.java” file will be converted to “Welcome.class” that contains the compiled version.
- IDEs typically provide a menu item, such as Build or Make, that invokes the javac command for you.
- If the compiler detects errors, you’ll need to go back to Phase 1 and correct them.



Compiler creates bytecodes and stores them on disk in a file whose name ends with `.class`

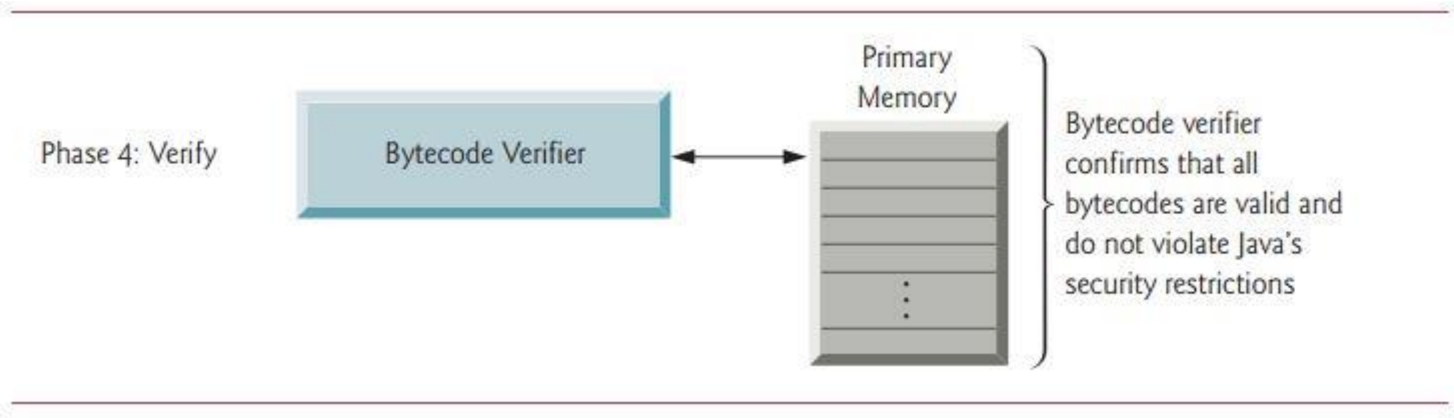
Phase 3: Loading a Program into Memory

- In Phase 3, the program in memory is loaded to execute it—this is known as loading
- The Java Virtual Machine (JVM) class loader takes the .class files containing **the program's bytecodes** and transfers them to primary memory. It also loads any of the .class files provided by Java that your program uses. The .class files can be loaded from a disk on your system or over a network (e.g., your local college or company network, or the Internet)


Phase 3: Load
Class Loader
Disk
Primary Memory
Class loader reads .class files containing bytecodes from disk and puts those bytecodes in memory

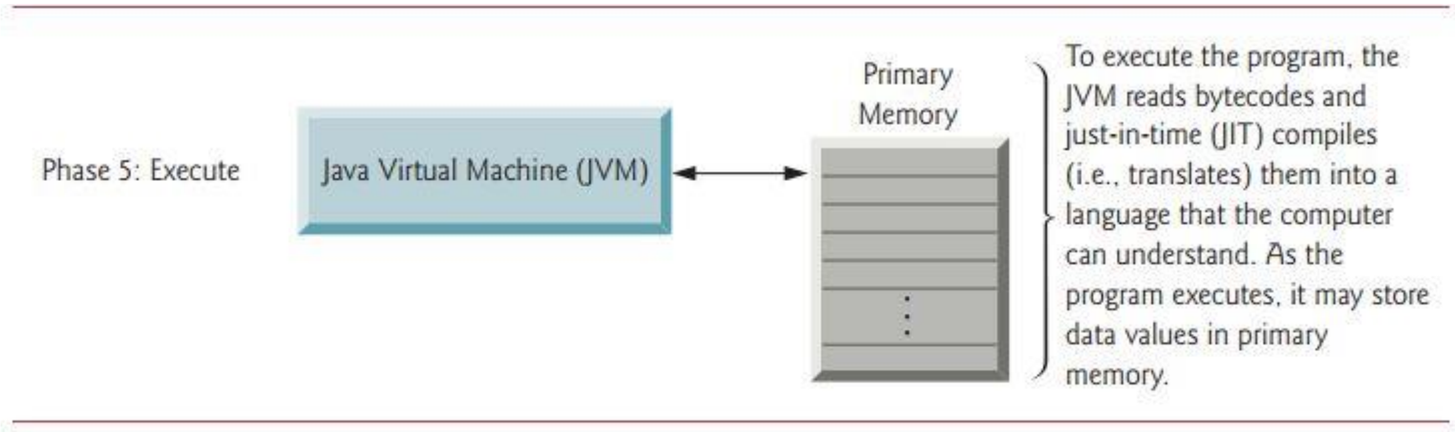
Phase 4: Bytecode Verification

- The classes are loaded, the bytecode verifier examines their bytecodes to ensure that they're valid and do not violate Java's security restrictions


Phase 4: Verify
Bytecode Verifier
Primary Memory
Bytecode verifier confirms that all bytecodes are valid and do not violate Java's security restrictions

Phase 5: Execution

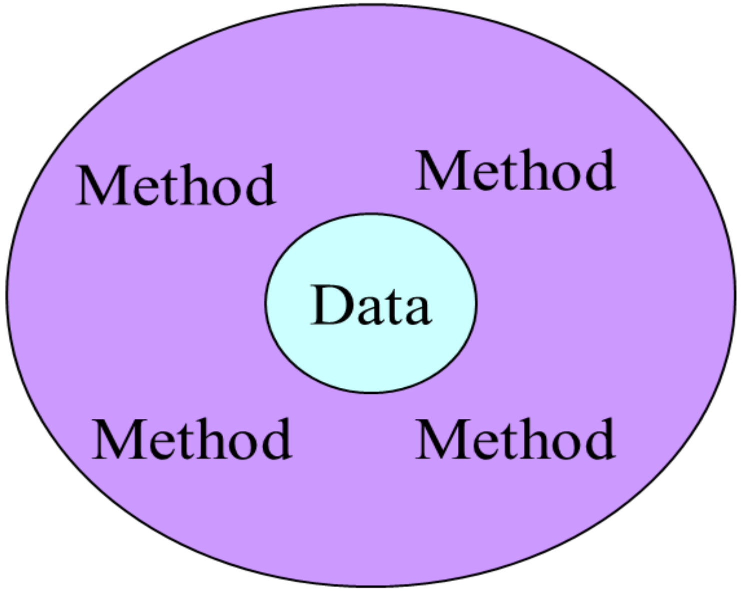
- Then the interpreter, in this case the JVM will would interpret the bytecodes and execute the byte codes.



To execute the program, the JVM reads bytecodes and just-in-time (JIT) compiles (i.e., translates) them into a language that the computer can understand. As the program executes, it may store data values in primary memory.

Object Oriented Programming (OOP)

- Object oriented programming is an approach that provides –
  - a way of modularizing programs by creating partitioned memory area for both data and functions
  - templates for creating copies of such modules on demand.
  - e.g. Smalltalk, Ada, C++, Objective C, Delphi, Java.



Object = data + method

Difference between Structured Programming and OOP

Structured Programming	OOP
Programs are divided into small programs or functions.	Programs are divided into objects or entities.
It is all about facilitating creation of programs with readable code and reusable components.	It is all about creating objects that usually contain both functions and data.
Its main aim is to improve and increase quality, clarity, and development time of computer program.	Its main aim is to improve and increase both quality and productivity of system analysis and design.
It simply focuses on functions and processes that usually work on data.	It simply focuses on representing both structure and behavior of information system into tiny or small modules that generally combines data and process both.

Difference between Structured Programming and OOP

Structured Programming	OOP
In this, methods are written globally and code lines are processed one by one i.e., Run sequentially.	In this, method works dynamically, make calls as per need of code for certain time.
It generally follows "Top-Down Approach".	It generally follows "Bottom-Up Approach".
It is more difficult to modify structured program and reuse code as compared to object-oriented programs.	It is less difficult to modify object-oriented programs and reuse code as compared to structured programs.