CENG204 - Programming Languages Concepts

Asst. Prof. Dr. Emre ŞATIR

Lecture 6
Language Implementation Systems

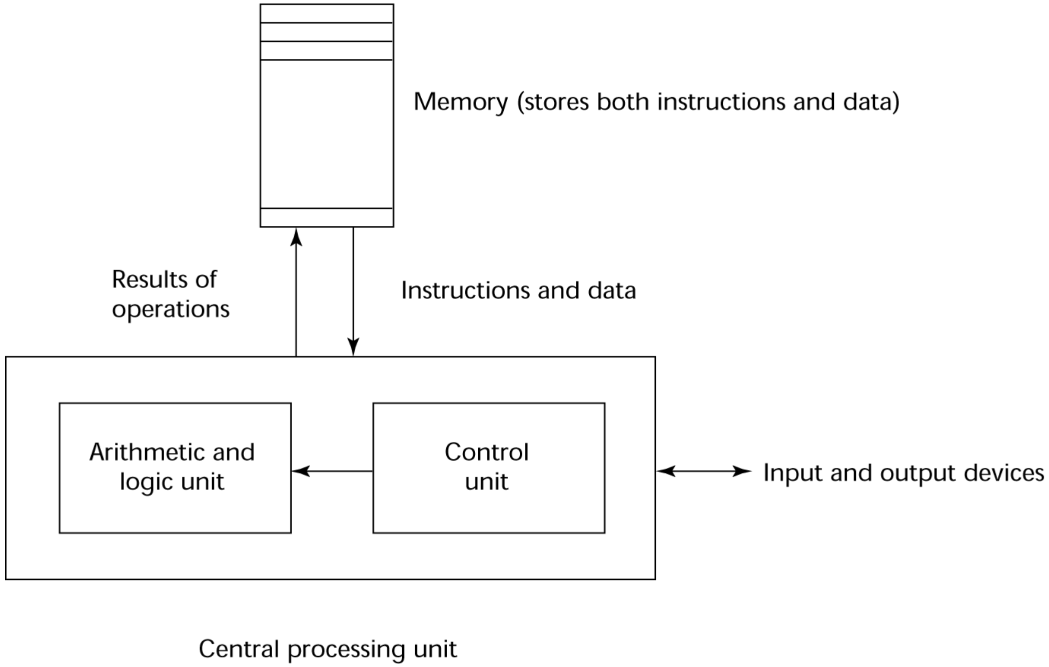
Lecture 6 Topics

- Implementation Methods
  - Compilation
  - (Pure) Interpretation
  - Hybrid Implementation
- Programming Environments
  - Framework vs Library
  - Combining Different Programming Languages

Implementation Methods

Implementation Methods

- As described before, two of the primary components of a computer are its internal **memory** and its **processor**.
- The internal memory is used to store programs and data (*The von Neumann Architecture*).
- The processor is a collection of circuits that provides a realization of a set of primitive operations, or machine instructions, such as those for arithmetic and logic operations.
- In most computers, some of these instructions, which are sometimes called macroinstructions, are actually implemented with a set of instructions called microinstructions, which are defined at an even lower level.
- Microinstructions are never seen by software.

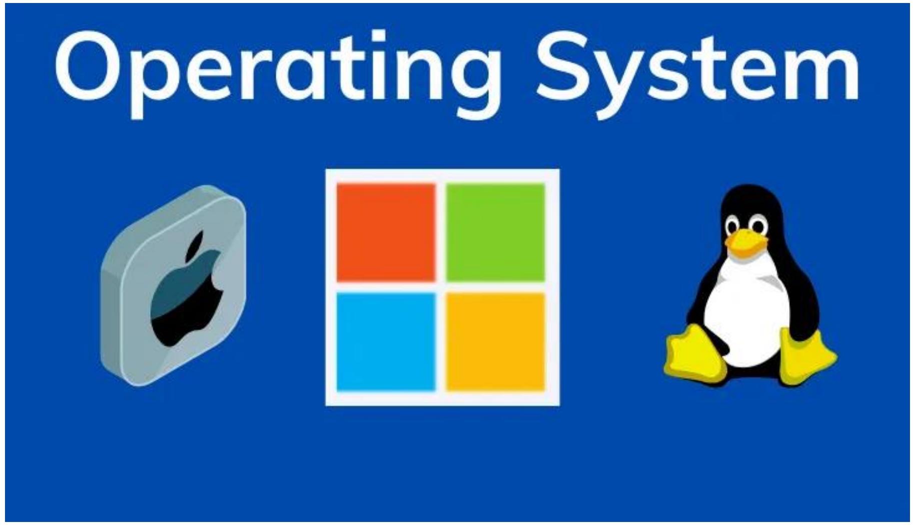


Implementation Methods

- The machine language of the computer is its set of instructions.
- In the absence of other supporting software, its own machine language is the only language that hardware computers "understand."
- Theoretically, a computer could be designed and built with a particular high-level language as its machine language, but it would be very complex and expensive ("dependent" on that language).
- The more practical machine design choice implements in hardware a very low-level language that provides the most commonly needed "primitive operations" and requires system software to create an interface to programs in higher-level languages.

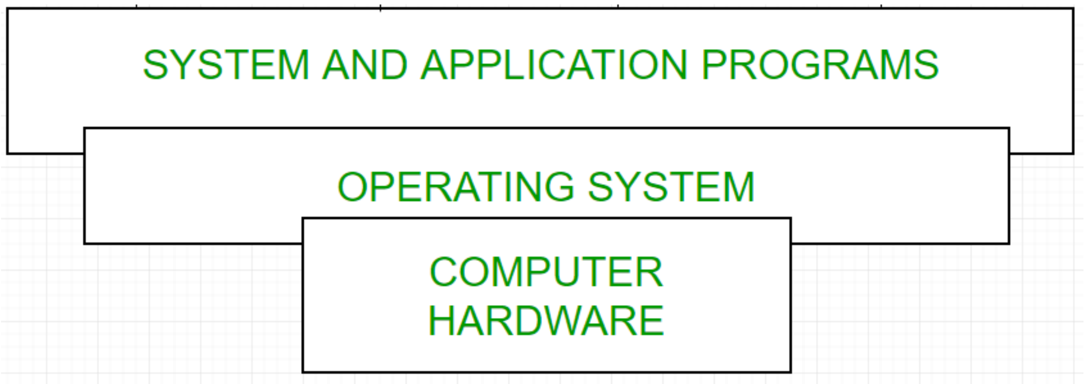
Implementation Methods

- A language implementation system cannot be the only software on a computer. Also required is a large collection of programs, called the operating system, which supplies higher-level primitives than those of the machine language.
- These primitives provide system resource management, input and output operations, a file management system, text and/or program editors, and a variety of other commonly needed functions.
- Because language implementation systems need many of the operating system facilities, they interface with the operating system rather than directly with the processor.



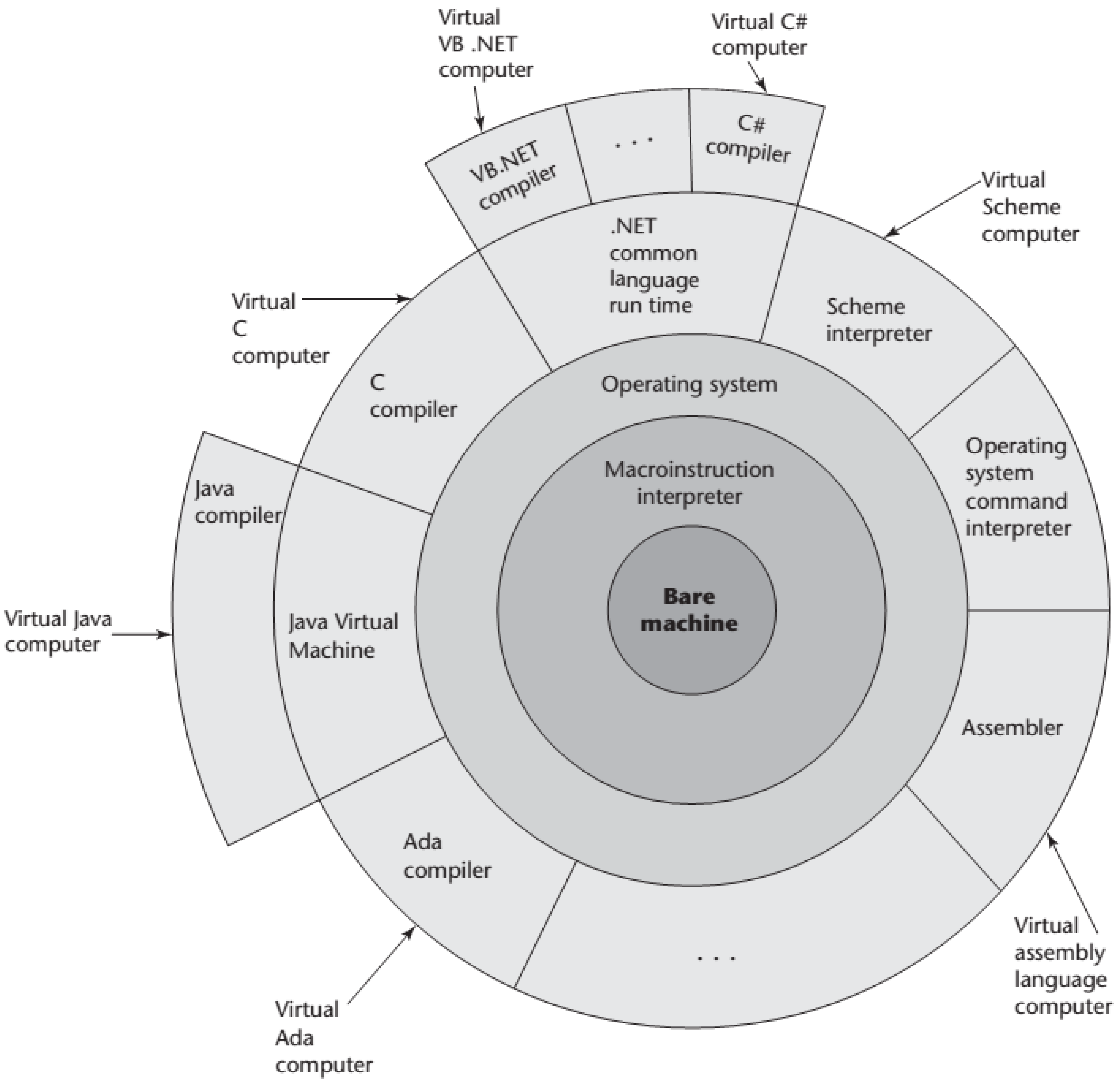
Implementation Methods

- The operating system and language implementations are layered over the machine language interface of a computer.
- These layers can be thought of as virtual computers, providing interfaces to the user at higher levels.
- For example, an operating system and a C compiler provide a “virtual C computer”. With other compilers, a machine can become other kinds of virtual computers.
- Most computer systems provide several different virtual computers. The layered view of a computer is shown in figure below:

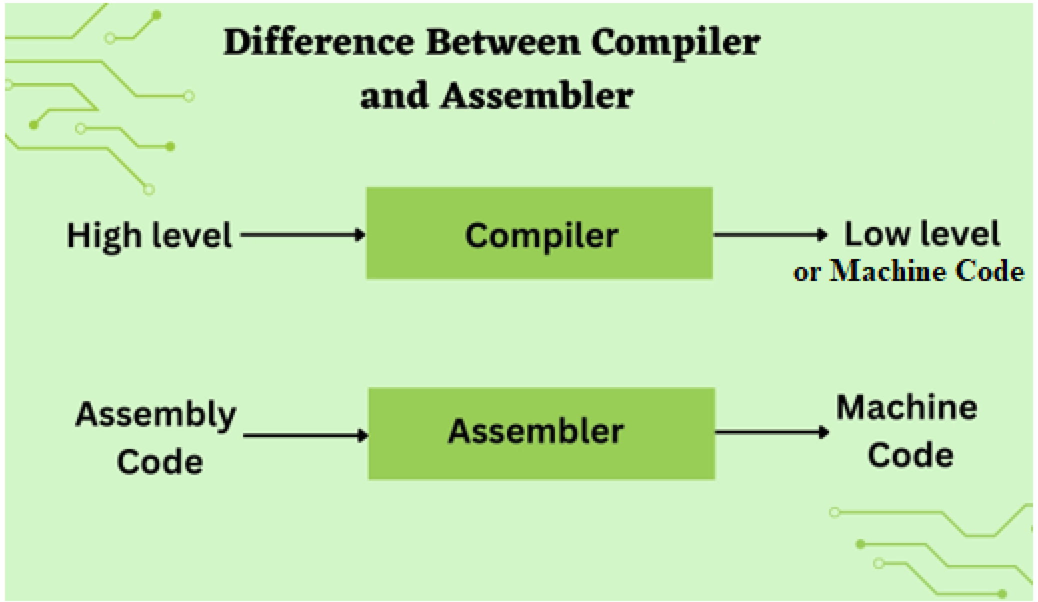


Layered View of Computer

The operating system and language implementation are layered over machine interface of a computer



Implementation Methods



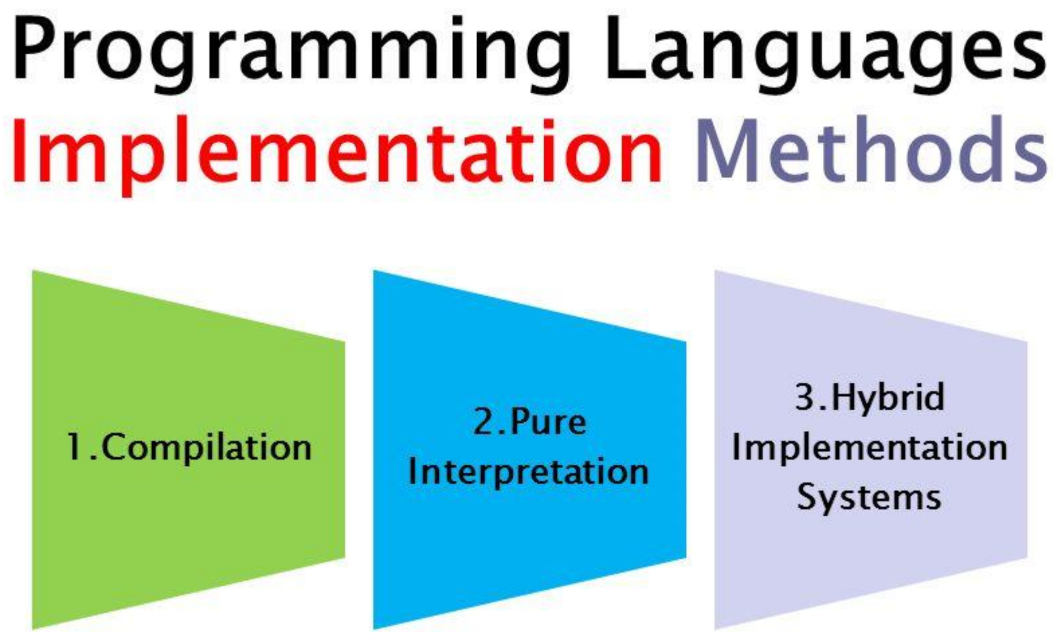
- Once this collection of high-level primitives had been identified, a program, called a **translator**, was written that translated programs expressed in these high-level primitives into machine-language programs.
- Such a translator was similar to assemblers, except that it often had to **compile** several machine instructions into short sequences to simulate the activity requested by a single high-level primitive.
- Thus, these translation programs were often called **compilers**.

Implementation Methods

- An alternative to compilers, called **interpreters**, emerged as another means of implementing third-generation languages.
- These programs were similar to compilers except that they executed the instructions as they were translated (line-by-line) instead of recording the translated version for future use.
- That is, rather than producing a machine-language copy of a program that would be executed later, an interpreter actually executed a program from its high-level form.

Implementation Methods

- Programming languages can be implemented by any of three general methods:
  1. Compilation
  2. (Pure) Interpretation
  3. Hybrid Implementation

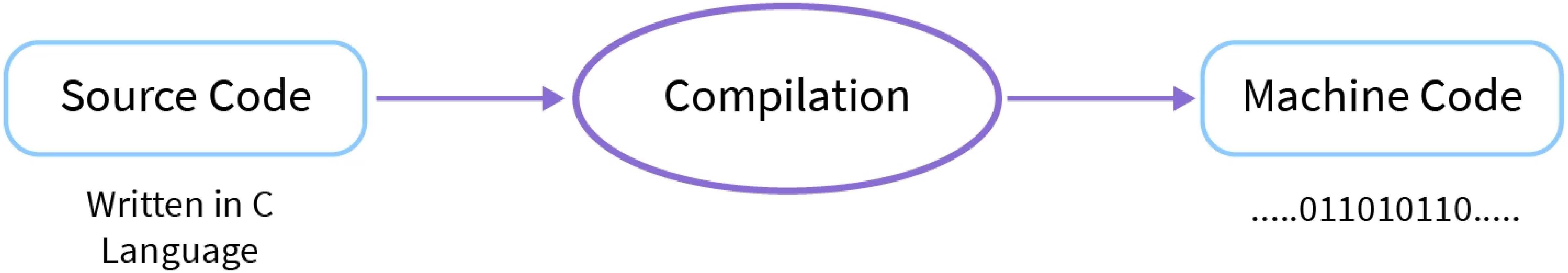


Programming Languages Implementation Methods

1.Compilation

2.Pure Interpretation

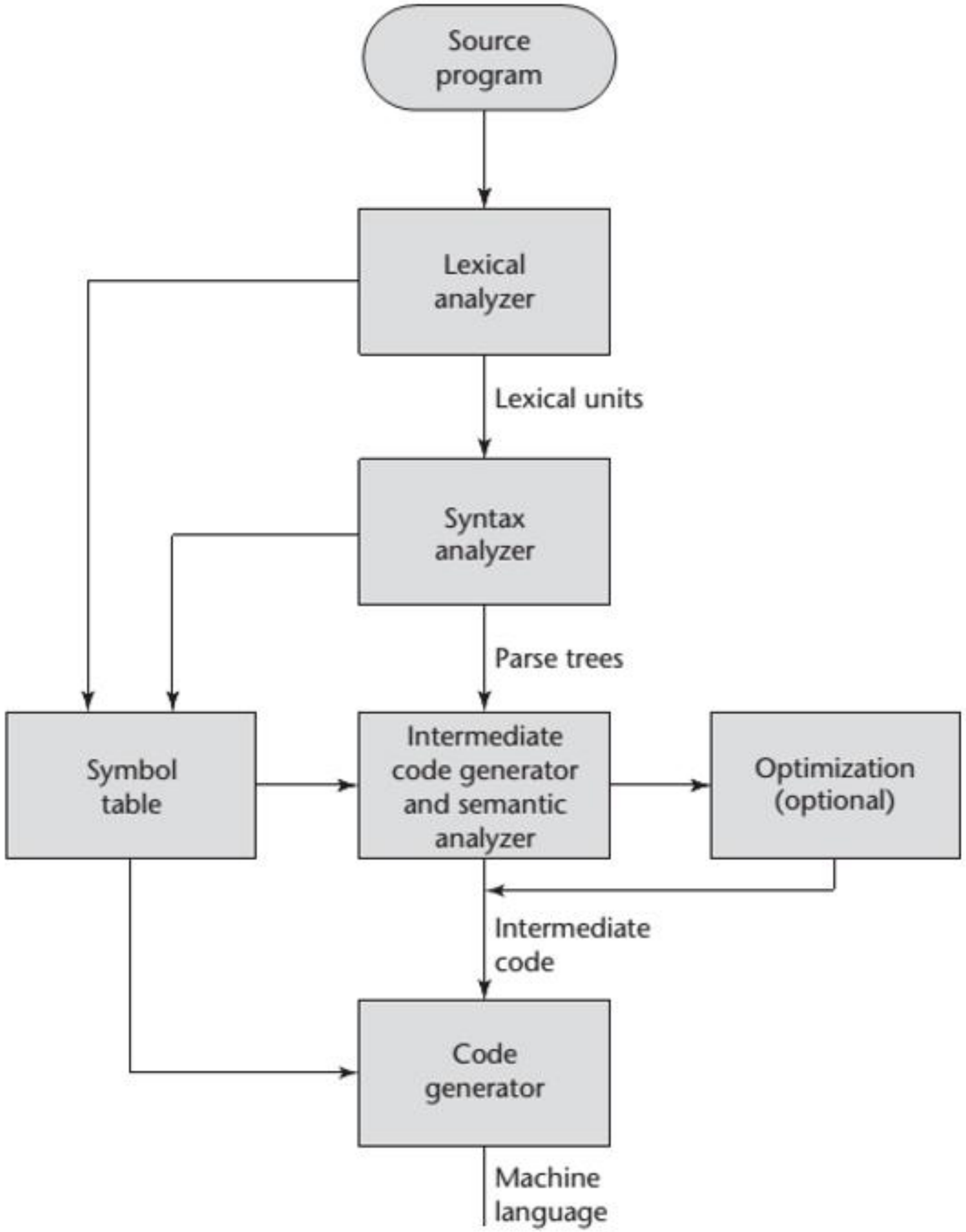
3.Hybrid Implementation Systems

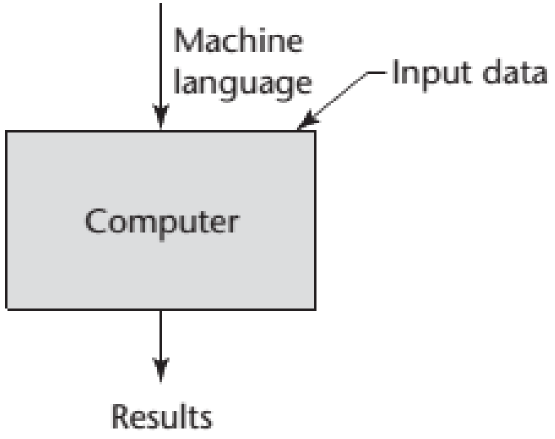


1. Compilation

- Programs (high-level program sourcecode) can be translated into machine language, which can be executed directly on the computer.
- This method is called a **compiler implementation** and has the advantage of very fast program execution, once the translation process is complete (slow translation).
- Most production implementations of languages, such as Fortran, C, COBOL, and C++, are by compilers.
- **Usage:** Large commercial applications.
- The process of compilation and program execution takes place in several phases, the most important of which are shown in the figure below:

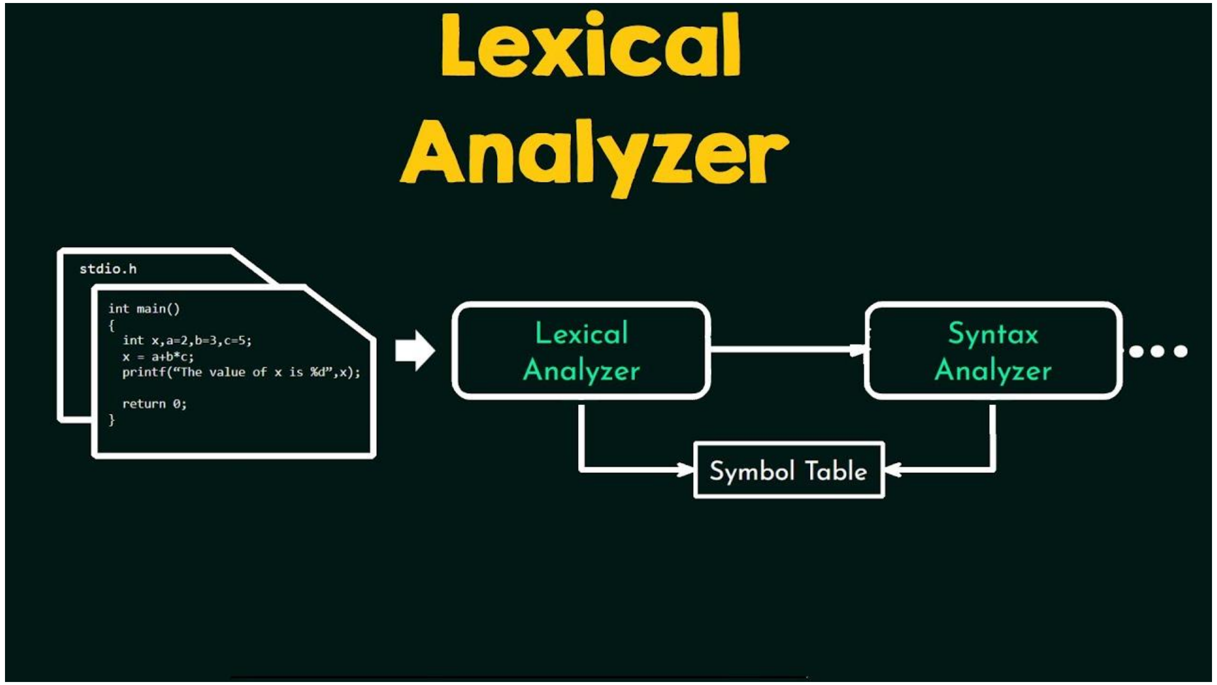
The Compilation Process





Compilation - Lexical Analysis

- The **lexical analyzer** gathers the characters of the source program into lexical units.
- The lexical units of a program are “identifiers”, “special words”, “operators”, and “punctuation symbols”.
- The lexical analyzer ignores (removes) comments in the source program because the compiler has no use for them.

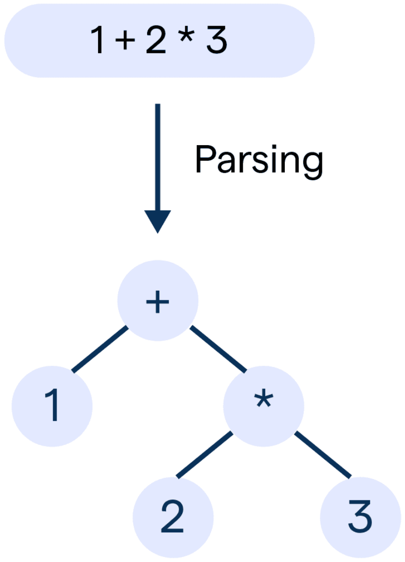


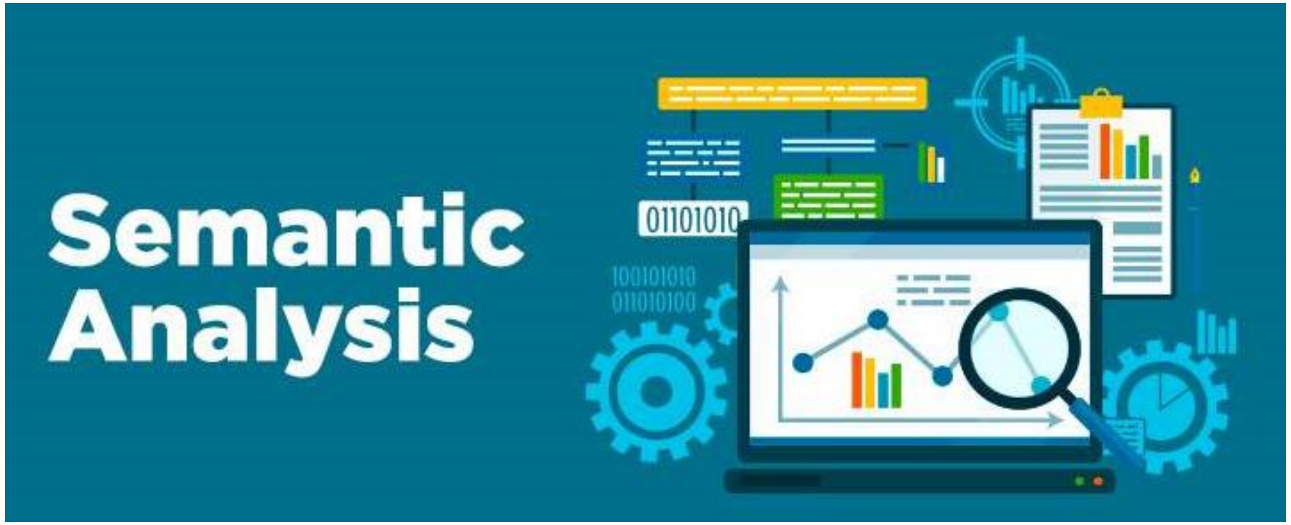
```
int main()
{
  int x,a=2,b=3,c=5;
  x = a+b*c;
  printf(“The value of x is %d”,x);

  return 0;
}
```

Compilation - Syntax Analysis

- The **syntax analyzer** takes the lexical units from the lexical analyzer and uses them to construct hierarchical structures called **parse trees**.
- Syntax analyzers also known as “**parsers**”.
- These parse trees represent the syntactic structure of the program.

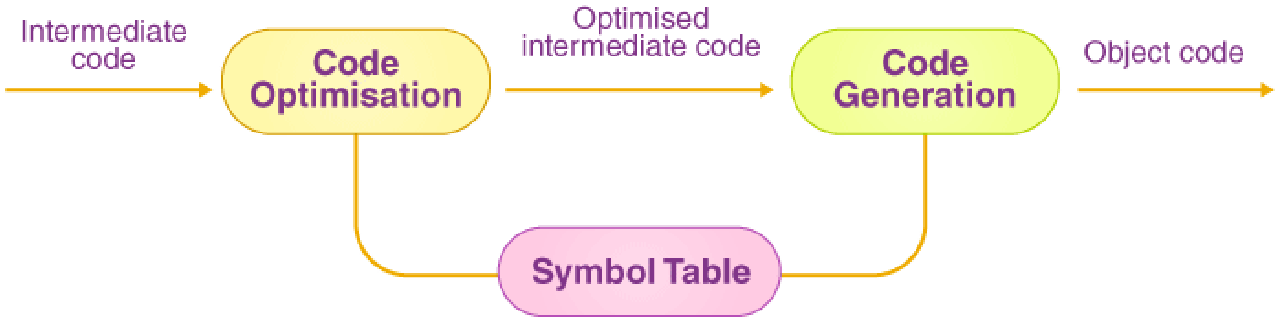
Syntax Analysis





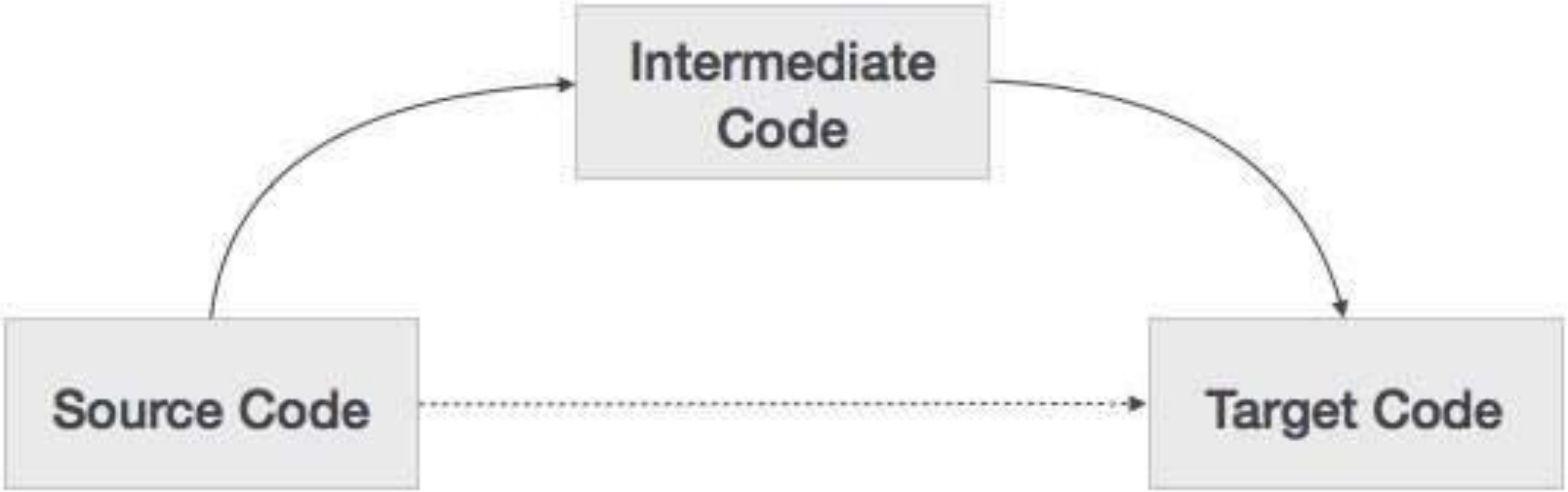
Compilation - Semantics Analysis

- The **intermediate code generator** produces a program in a different language, at an intermediate level between the source program and the final output of the compiler (the machine language program).
- Intermediate languages sometimes look very much like assembly languages.
- The **semantic analyzer** is an integral part of the intermediate code generator.
- The semantic analyzer checks for errors, such as type errors, that are difficult to detect during syntax analysis.



Compilation - Optimization

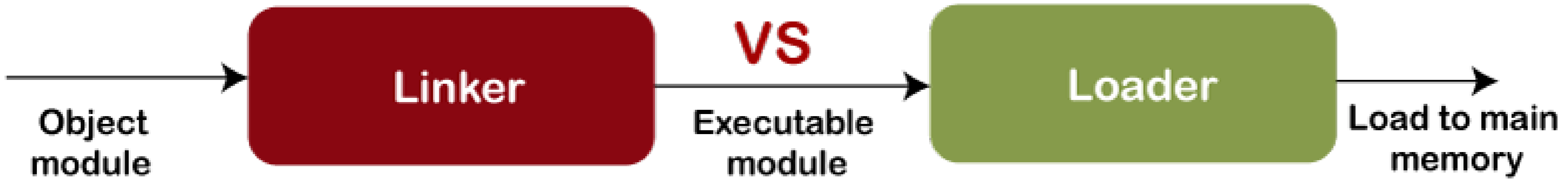
- Optimization, which improves programs by making them smaller or faster or both.
- Because many kinds of optimization are difficult to do on machine language, most optimization is done on the intermediate code.
- Optimization is usually an optional operation.



Compilation – Code Generation

- The **code generator** translates the optimized intermediate code version of the program into an equivalent machine language program.

- *** "The symbol table" serves as a database for the compilation process. The primary contents of the symbol table are the type and attribute information of each user-defined name in the program. This information is placed in the symbol table by the lexical and syntax analyzers and is used by the semantic analyzer and the code generator.

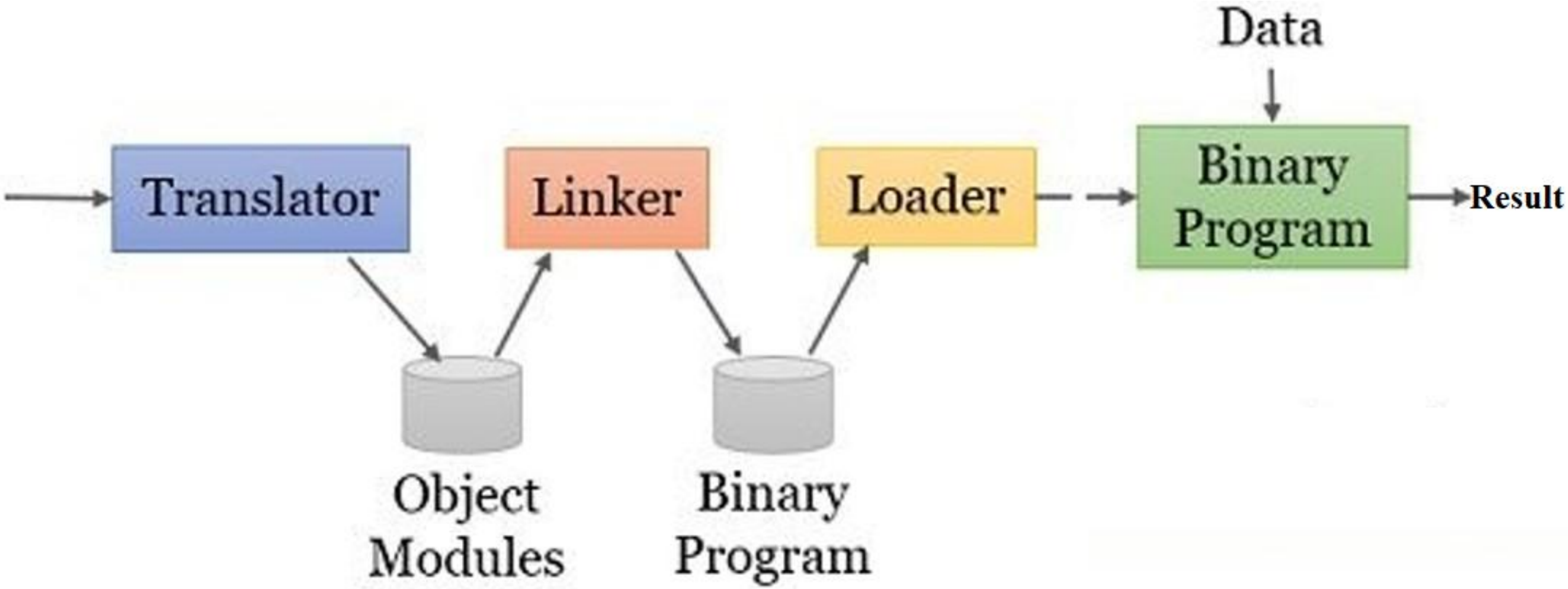


Linking and Loading

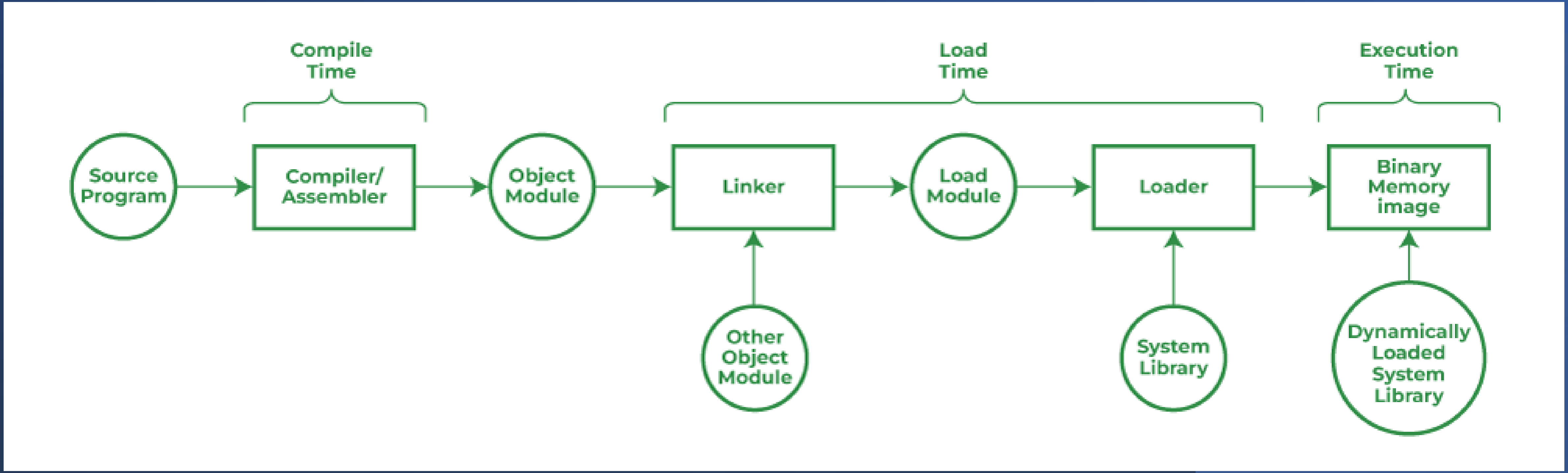
- Although the machine language generated by a compiler can be executed directly on the hardware, it must nearly always be run along with some other code.
- Most user programs also require programs from the operating system.
- Among the most common of these are programs for input and output.
- The compiler builds calls to required system programs when they are needed by the user program.
- Before the machine language programs produced by a compiler can be executed, the required programs from the operating system must be found and linked to the user program.

Linking and Loading

- Establishing the linking between all the modules or all the functions of the program in order to continue the program execution is called **linking**.
- Linking is a process of collecting and maintaining pieces of code and data into a “single file”.
- Linker takes “**object module**” as input and forms an executable file as output for the loader.
- Linking is performed at the last step in compiling a program.



Schematic Execution of the Program


Compile Time
Load Time
Execution Time
Source Program
Compiler/ Assembler
Object Module
Linker
Load Module
Loader
Binary Memory image
Other Object Module
System Library
Dynamically Loaded System Library

Linking and Loading

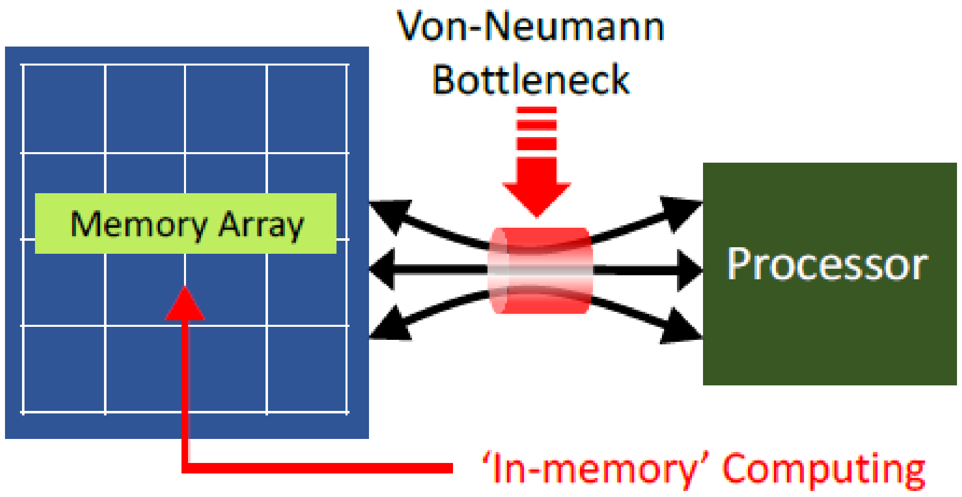
- The linking operation connects the user program to the system programs by placing the addresses of the entry points of the system programs in the calls to them in the user program.
- The user and system code together are sometimes called a **load module** or **executable image**.
- The process of collecting system programs and linking them to user programs is called **linking**.
- It is accomplished by a systems program called a **linker**.
- In addition to systems programs, user programs must often be linked to previously compiled user programs that reside in libraries. So, the linker not only links a given program to system programs, but also it may link it to other user programs.

Linking and Loading

- **Loading** is the process of loading the program from secondary memory to the main memory for execution.
- The **loader** not only loads the program into memory, but also performs:
  - **Address translation** (conversion of relative/relocatable codes to absolute addresses)
  - **Loads the necessary dynamic libraries** (Dynamic Linking): Although a static linker combines all libraries, in systems that use dynamic linking, the loader loads the required shared libraries (e.g., .dll) into memory during execution and establishes the program's connections with them.
  - And **starts the execution of the executable file**: Determines the entry point of the executable file and transfers control from the operating system to the program.

The von Neumann Bottleneck

- "Connection speed" between a computer's memory and its processor determines the speed of a computer.
- Program instructions often can be executed much faster than the speed of the connection; the connection speed thus results in a *bottleneck.*
- Known as the ***von Neumann bottleneck***; it is the primary limiting factor in the speed of computers.

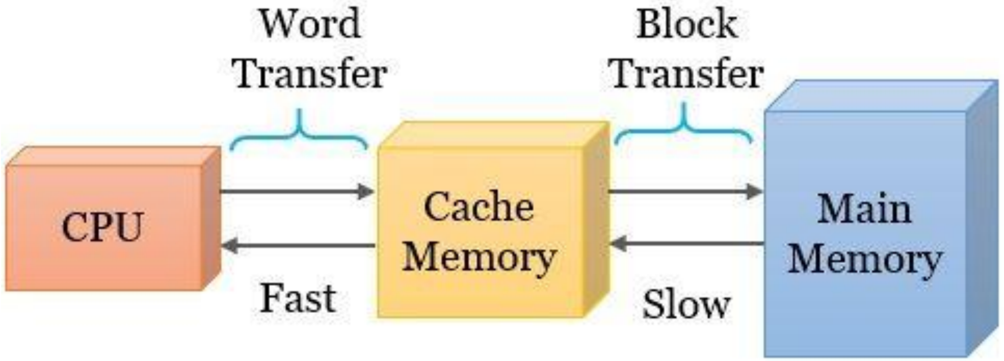


The von Neumann Bottleneck

- The von Neumann Bottleneck is a problem of von Neumann architecture that grows as today's technology advances.
- It occurs as processor speeds increase.
- The processor, which is faster than the memory, starts waiting to use the data from the memory after receiving the commands.
- As technology progresses, memory technology that cannot keep up with faster processors has increased the magnitude of this problem.

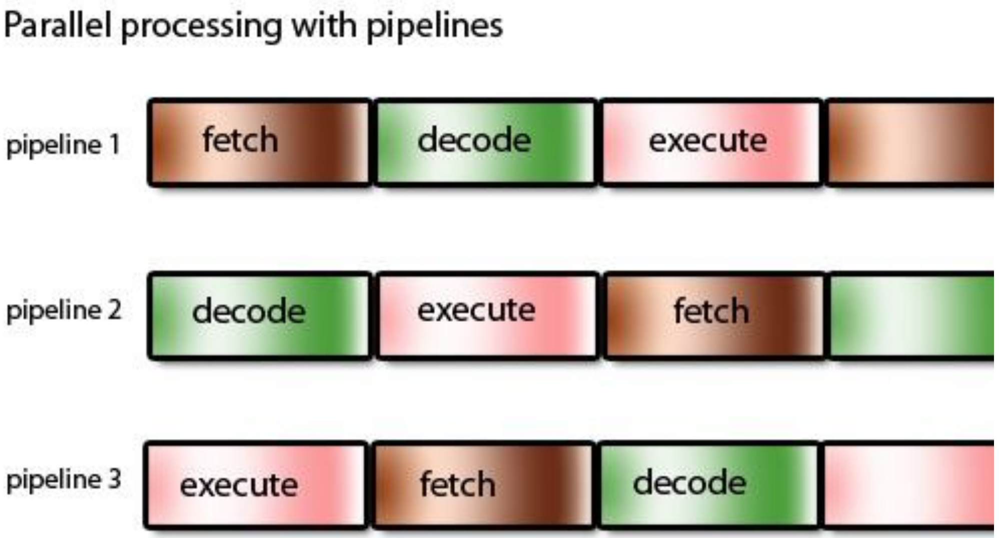
The von Neumann Bottleneck

- The most logical solution is to use **cache** (memories).
- Since caches keep the data that the processor has used recently, the processor does not need to access memory and the expected time is compensated.
- Caches reduce the need for frequent memory access by keeping the data closer to the CPU, thereby improving performance.

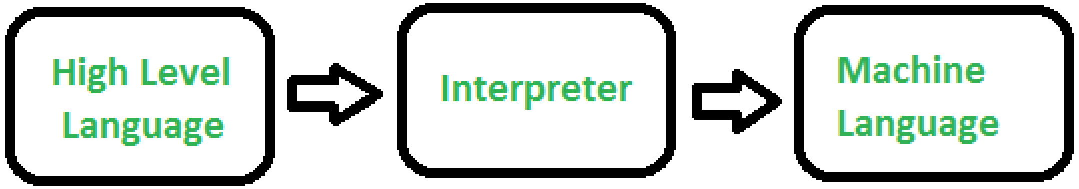


The von Neumann Bottleneck

- **Pipelining** is another technique used to overcome the von Neumann bottleneck.
- Essentially, instead of doing one instruction at a time, and waiting for each to complete, you would have a production line, where each instruction was started and then passed down the line (executing instructions concurrently).
- By "overlapping" different stages of instruction execution, pipelining reduces the impact of memory access latency on overall performance.



2. (Pure) Interpretation



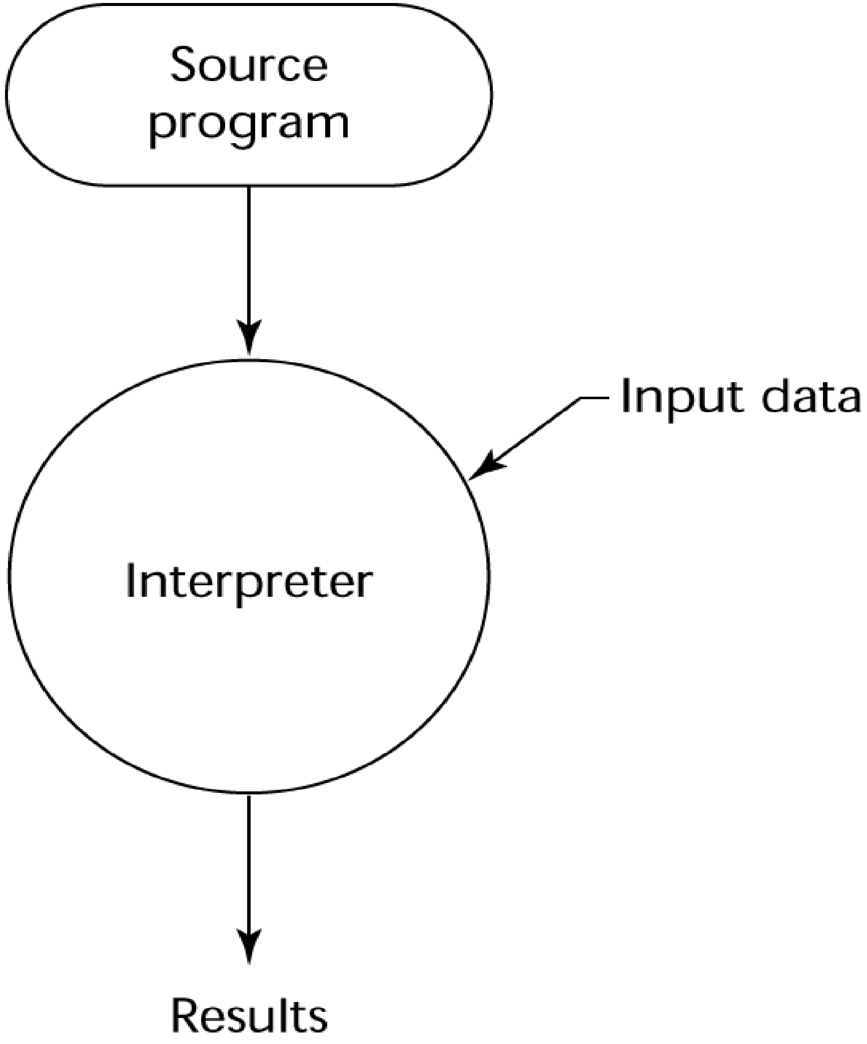
High Level Language

Interpreter

Machine Language

- With this approach, programs are interpreted by another program called an **interpreter**, with no translation whatever.
- The interpreter program acts as a software simulation of a machine whose “fetch-decode-execute cycle” deals with high-level language program statements rather than machine instructions.
- Pure interpretation has the advantage of allowing easy implementation of many source-level debugging operations, because all run-time error messages can refer to source-level units.
- For example, if an array index is found to be out of range, the error message can easily indicate the source line of the error and the name of the array.

Pure Interpretation Process



Pure Interpretation

- On the other hand, this method has the serious disadvantage that execution is 10 to 100 times slower than in compiled systems.
- The primary source of this slowness is the decoding of the high-level language statements, which are far more complex than machine language instructions.
- Furthermore, regardless of how many times a statement is executed, it must be decoded every time. Therefore, statement decoding, rather than the connection between the processor and memory, is the “bottleneck of a pure interpreter”.

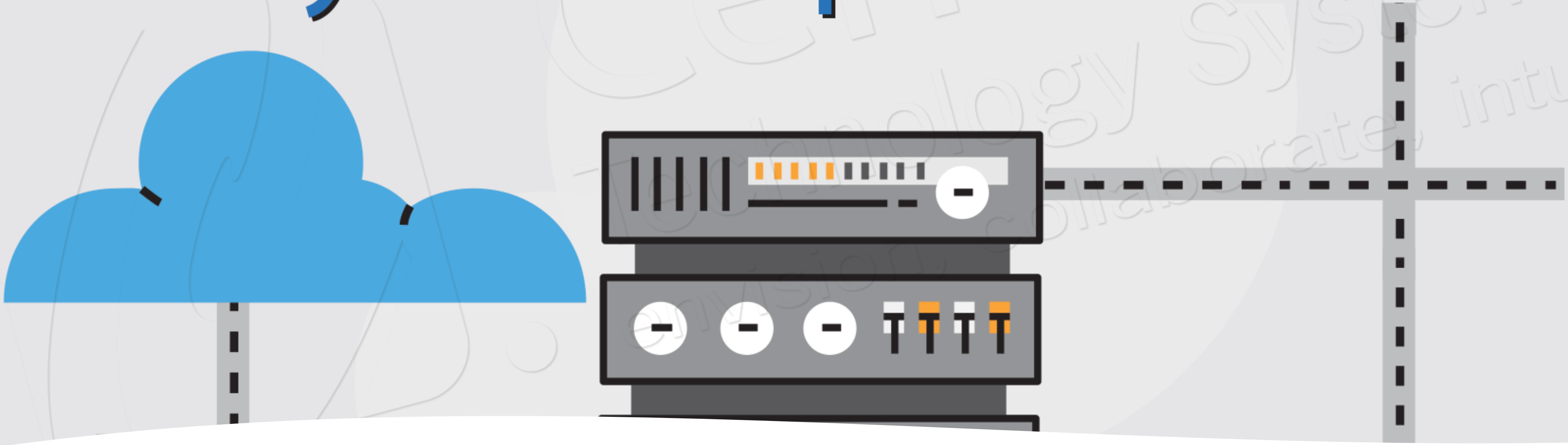
Pure Interpretation

- Although some simple early languages of the 1960s (APL, SNOBOL, and Lisp) were purely interpreted, by the 1980s, the approach was rarely used on high-level languages.
- However, in recent years, pure interpretation has made a significant comeback with some Web scripting languages, such as JavaScript and PHP, which are now widely used.
- **Usage:** Small programs or when efficiency is not an issue.

*** Recall that “Scripting Languages” are implemented partial or full interpretation.

*** Python is a relatively recent object-oriented interpreted scripting language.

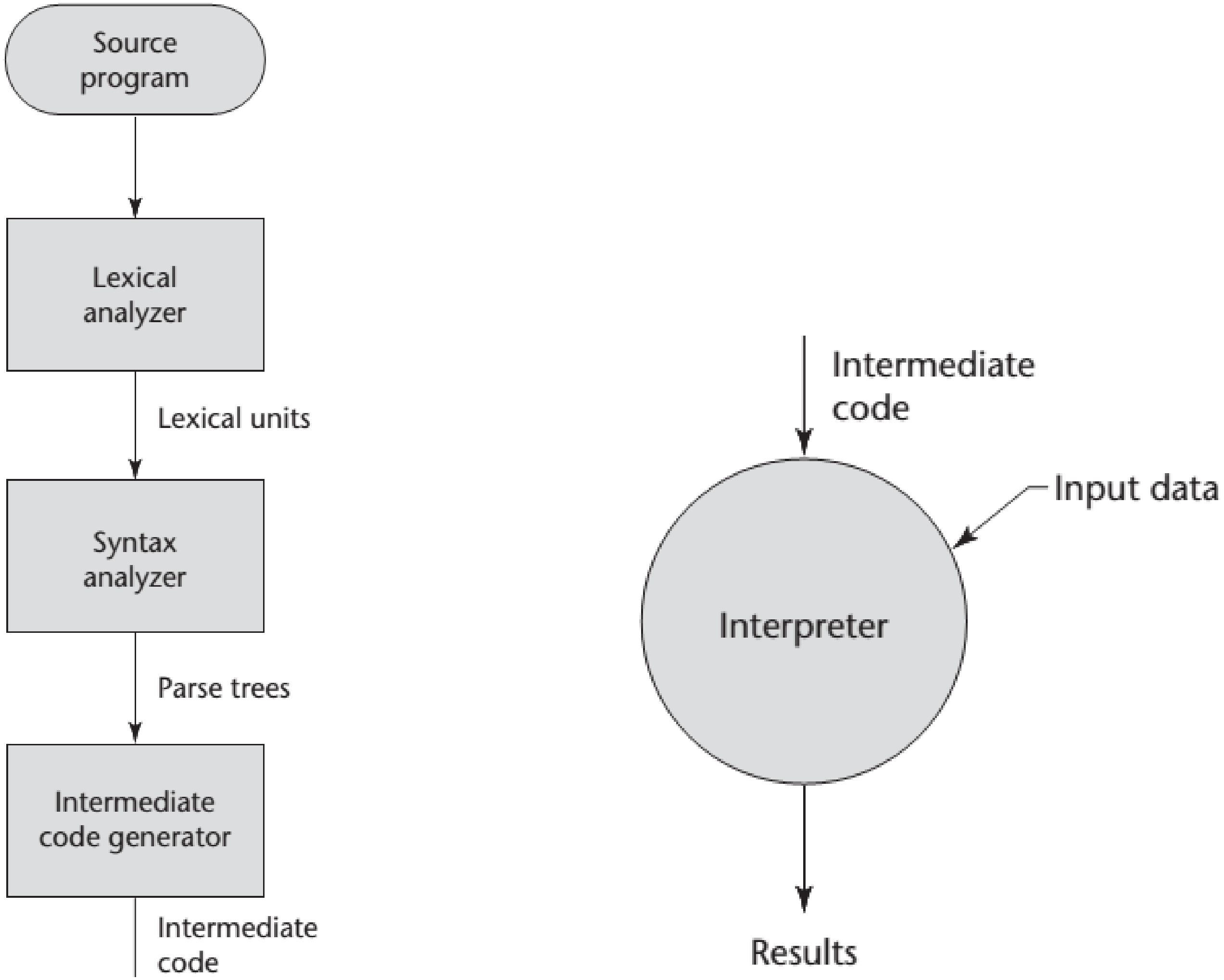
Hybrid Implementation



3. Hybrid Implementation Systems

- Some language implementation systems are a compromise between compilers and pure interpreters; they translate high-level language programs to an intermediate language designed to allow easy interpretation.
- This method is faster than pure interpretation because the source language statements are decoded only once.
- The process used in a hybrid implementation system is shown in figure below. Instead of translating intermediate language code to machine code, it simply interprets the intermediate code.

Hybrid Implementation Process



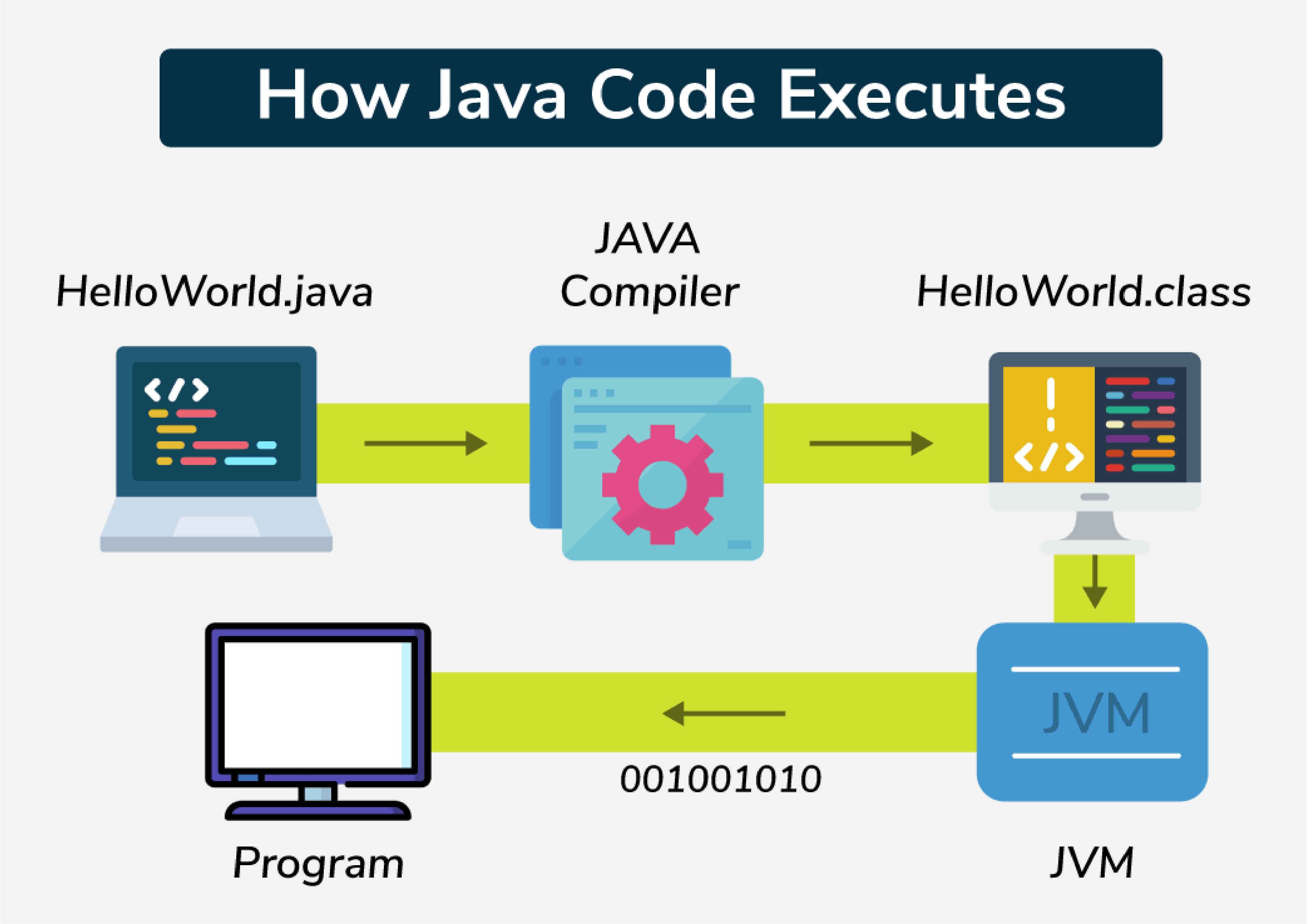
Hybrid Implementation Systems

- For example, **Perl** is implemented with a hybrid system. Perl programs are partially compiled to detect errors before interpretation and to simplify the interpreter.
- Initial implementations of **Java** were all hybrid. Its intermediate form, called **bytecode**, provides portability to any machine that has a bytecode interpreter and an associated run-time system. Together, these are called the "Java Virtual Machine (JVM)".
- *** There are now systems that translate Java source code or bytecode into machine code for faster execution.
- **Usage**: Small and medium systems when efficiency is not the first concern.

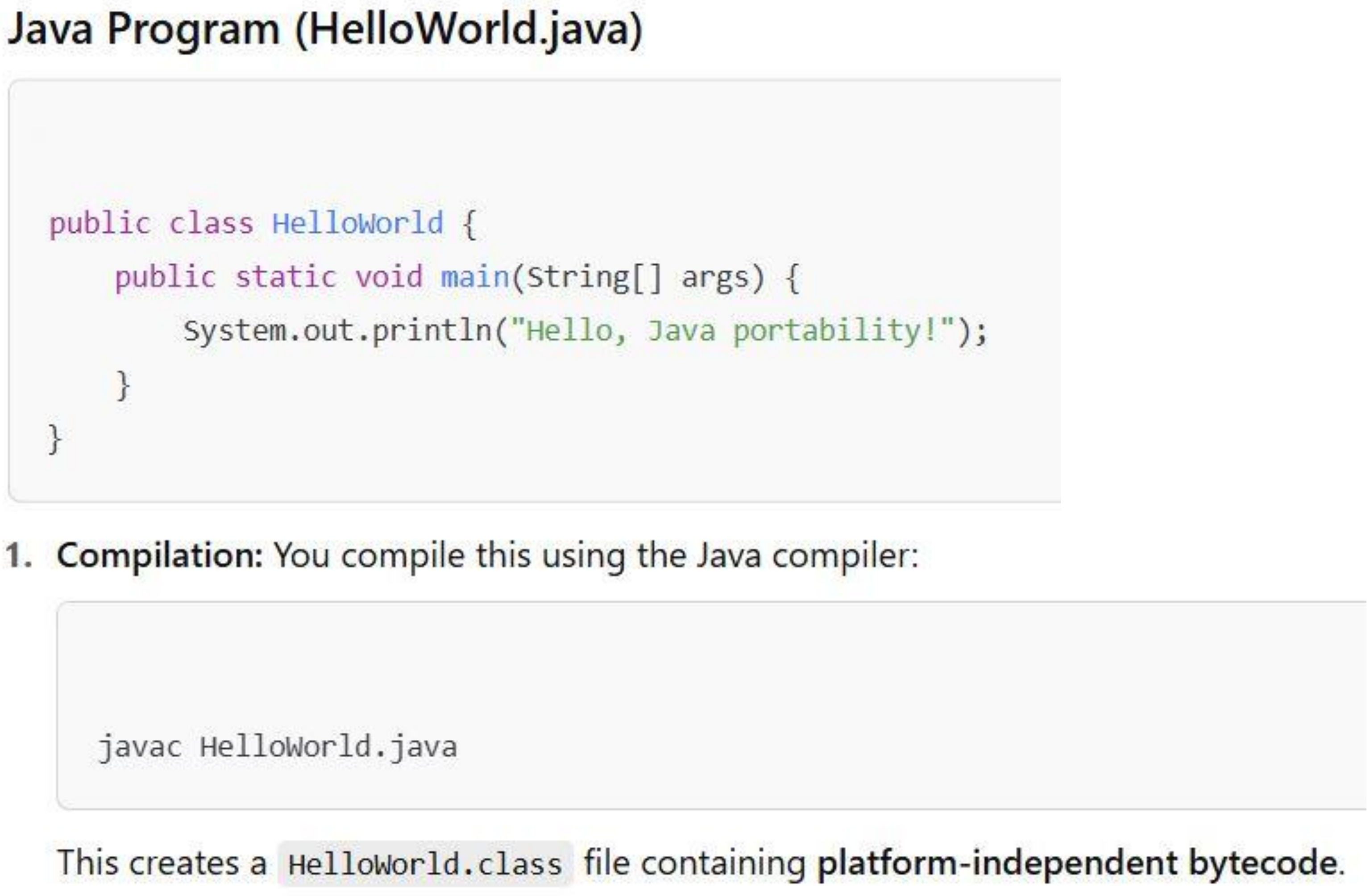
How does Java Ensure Portability?

* Java programs are compiled into an intermediate form called bytecode, which is platform-independent. This bytecode runs on the JVM, which is available for many hardware and operating system platforms.

* Since the bytecode does not depend on the underlying system architecture, any machine with a compatible JVM can run the same Java program without modification.



How does Java Ensure Portability?



Java Program (HelloWorld.java)

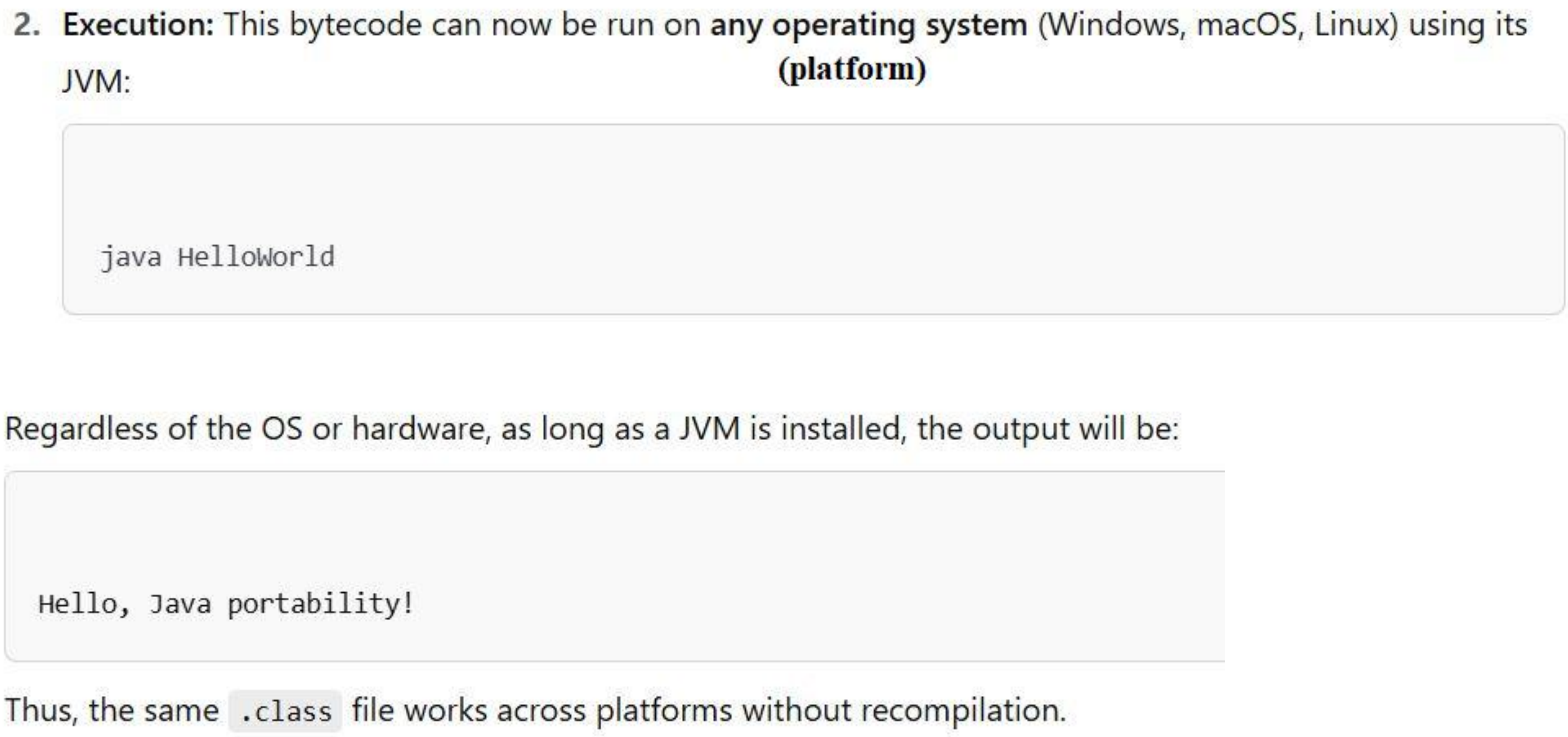
```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, Java portability!");
    }
}
```

1. **Compilation:** You compile this using the Java compiler:

```
javac HelloWorld.java
```

This creates a `HelloWorld.class` file containing **platform-independent bytecode**.

How does Java Ensure Portability?



2. **Execution:** This bytecode can now be run on **any operating system** (Windows, macOS, Linux) using its JVM: **(platform)**

```
java HelloWorld
```

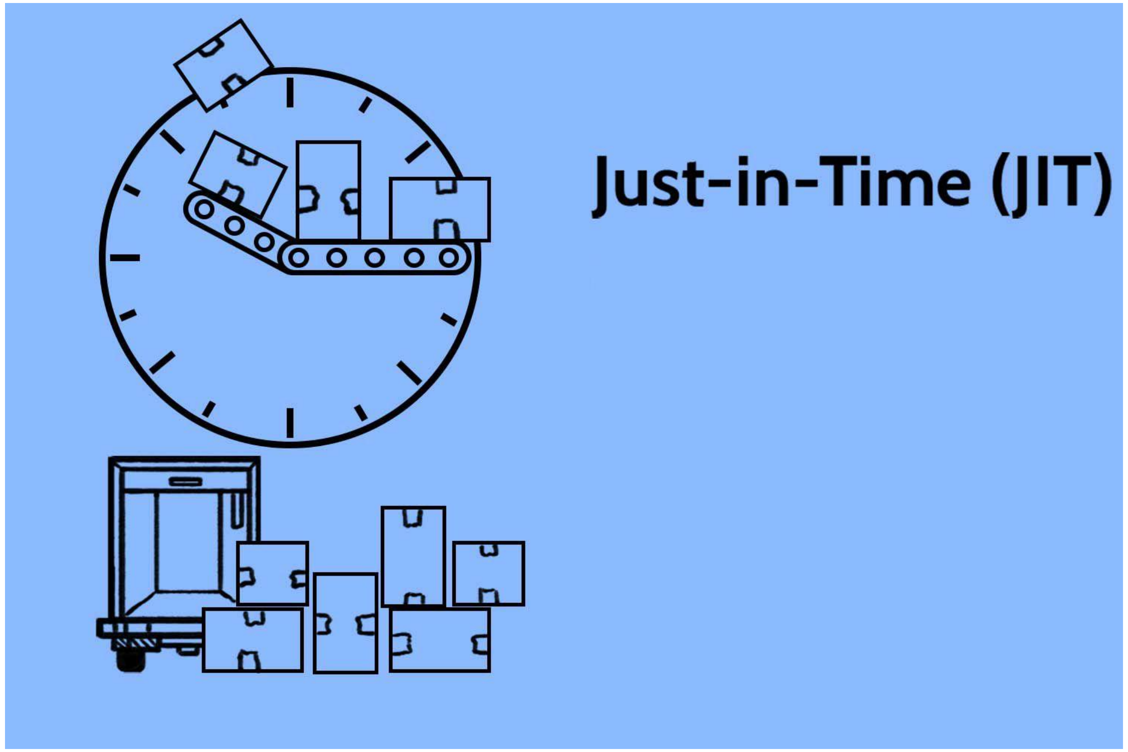
Regardless of the OS or hardware, as long as a JVM is installed, the output will be:

```
Hello, Java portability!
```

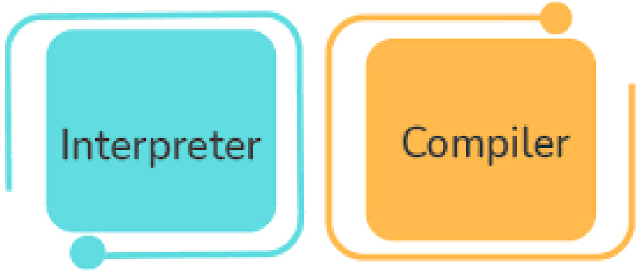
Thus, the same `.class` file works across platforms without recompilation.

Just-in-Time Implementation Systems

- A Just-in-Time (JIT) implementation system initially translates programs to an intermediate language. Then, during execution, it compiles intermediate language methods into machine code when they are called. The machine code version is kept for subsequent calls.
- JIT systems now are widely used for Java programs. Also, the .NET languages are all implemented with a JIT system.
- In essence, JIT systems are delayed compilers.



An Idea for Language Implementation



- Sometimes an implementor may provide both compiled and interpreted implementations for a language.
- In these cases, the interpreter is used to develop and debug programs. Because source-level debugging is easier with this method.
- Then, after a (relatively) bug-free state is reached, the programs are compiled to increase their execution speed.

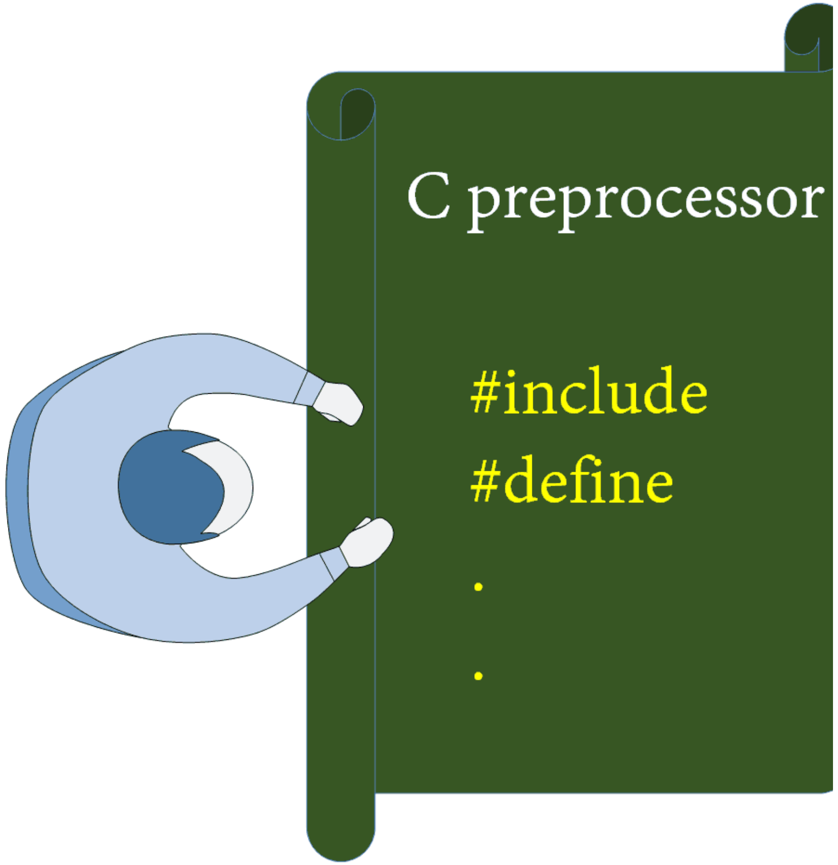
Preprocessors

- A **preprocessor** is a program that processes a program just before the program is compiled.
- Preprocessor instructions are embedded in programs.
- Preprocessor instructions are commonly used to specify that the code from another file is to be included. For example, the C preprocessor instruction

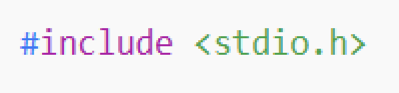
```
#include "myLib.h"
```

causes the preprocessor to copy the contents of myLib.h into the program at the position of the `#include`.

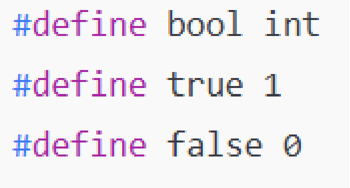
- `#include`, `#define`, and similar instructions are called as **macros** (The preprocessor is essentially a macro expander).



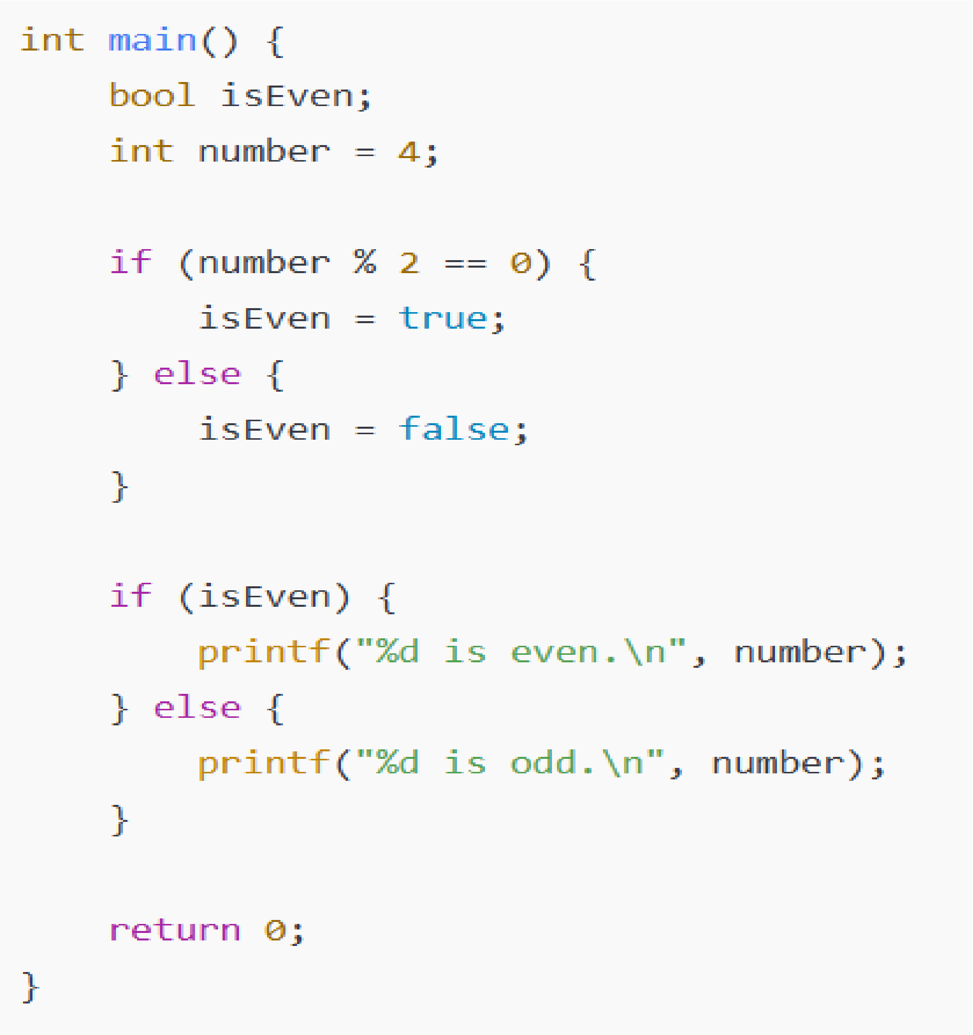
An Example C Program



```c
#include <stdio.h>
```



```c
#define bool int
#define true 1
#define false 0
```



```c
int main() {
    bool isEven;
    int number = 4;

    if (number % 2 == 0) {
        isEven = true;
    } else {
        isEven = false;
    }

    if (isEven) {
        printf("%d is even.\n", number);
    } else {
        printf("%d is odd.\n", number);
    }

    return 0;
}
```

Programming Environments

Programming Environments

- Programming environments have become important parts of software development systems, in which the language is just one of the components.
- A **programming environment** is the collection of tools used in the development of software.
- An **Integrated Development Environment (IDE)** is a software application that provides comprehensive facilities for software development.
- Netbeans, PyCharm, IntelliJ IDEA, Eclipse, Visual Studio, CodeBlocks are some examples of IDEs.

Programming Environments

- This collection may consist of a file system, a text editor, a linker, a debugger, a compiler (or interpreter) and a large collection of some other integrated tools, each accessed through a uniform user interface.
- The development and maintenance of software is greatly enhanced with these tools. Therefore, the characteristics of a programming language are not the only measure of the software development capability of a system.

Programming Environments

- UNIX is an older programming environment, first distributed in the middle 1970s, built around a portable multiprogramming operating system.
- It provides a wide array of powerful support tools for software production and maintenance in a variety of languages.
- In the past, the most important feature absent from UNIX was a uniform interface among its tools. This made it more difficult to learn and to use.
- However, UNIX is now often used through a GUI that runs on top of UNIX. Examples of UNIX GUIs are the Solaris Common Desktop Environment (CDE), GNOME, and KDE.
- These GUIs make the interface to UNIX appear similar to that of Windows and Macintosh systems.

UNIX®

A Standard of The Open Group®

Programming Environments

- **Microsoft Visual Studio .NET** is a relatively recent step in the evolution of software development environments.
- It is a large and elaborate collection of software development tools, all used through a windowed interface.
- This system can be used to develop software in any one of the .NET languages: C#, Visual Basic.NET, JScript (Microsoft’s version of JavaScript), F# (a .NET functional language), and C++/CLI.

- *** C++/CLI is a variant of the C++ programming language, modified for “Common Language Infrastructure”. It has been part of Visual Studio 2005 and later and provides interoperability with other .NET languages such as C#.

Programming Environments

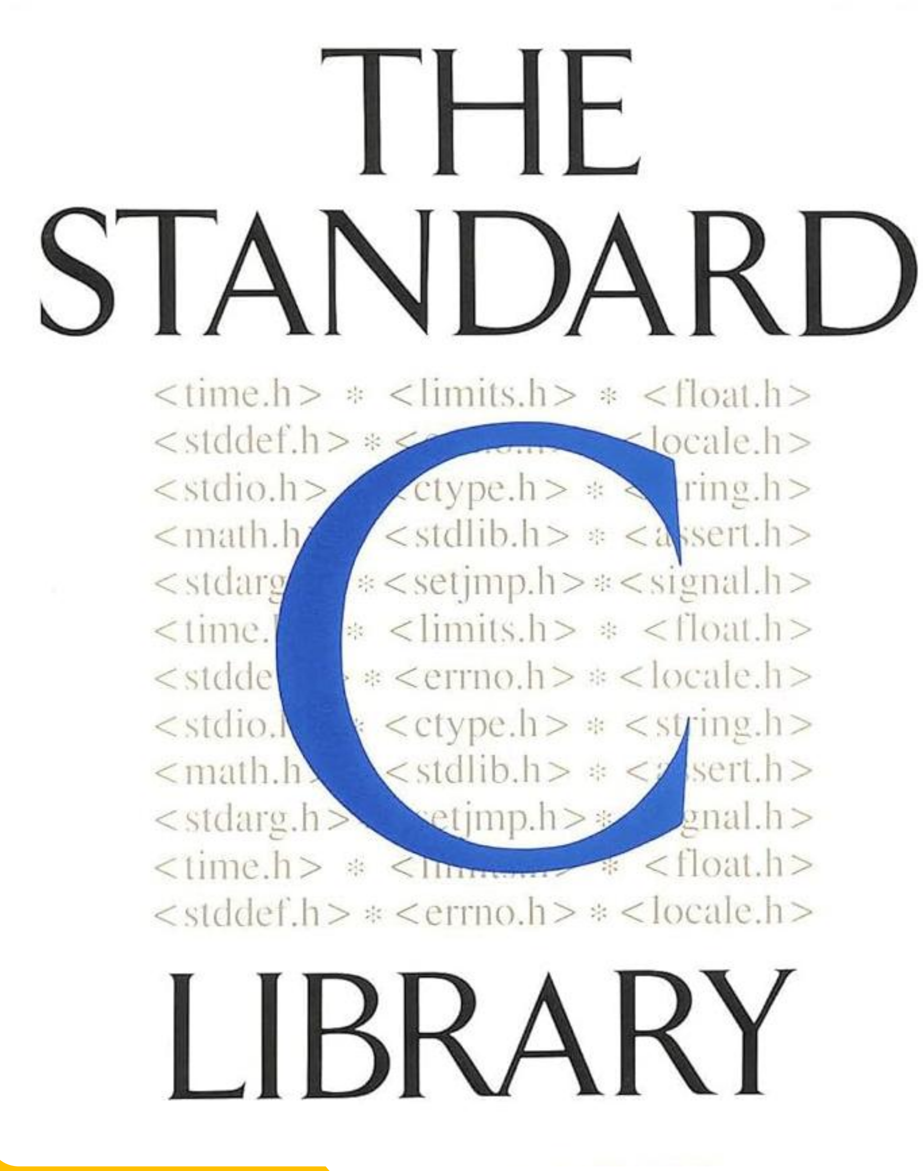
- In the Microsoft .NET environment, the bytecode equivalent of Java is called **Common Intermediate Language (CIL)**, or formerly Microsoft Intermediate Language (MSIL).
- CIL is the bytecode format into which code compiled from .NET languages (C#, VB.NET, etc.) is transformed before it is executed.
- At runtime, this CIL code is translated into machine code by a Just-In-Time (JIT) compiler and executed.

Programming Environments

- Bytecode and CIL are mainly designed for portability;the situation is slightly different when it comes to interoperability between languages:
- Java bytecode offers only portable executability. In the Java ecosystem, cross-language interoperability (e.g. Java and Kotlin) is possible, but this is mostly thanks to the runtime support provided by the JVM (Java Virtual Machine) and standard APIs. Bytecode alone does not provide this compatibility.
- CIL in the .NET environment provides both portability and is explicitly designed for interoperability between languages. This is because:
- C#, VB.NET, F#, even other languages in .NET compile to the same CIL standard. There is a common type system (Common Type System - CTS) and a common working model (Common Language Specification - CLS). This means that you can, for example, use a class written in C# directly in VB.NET code.

Programming Environments

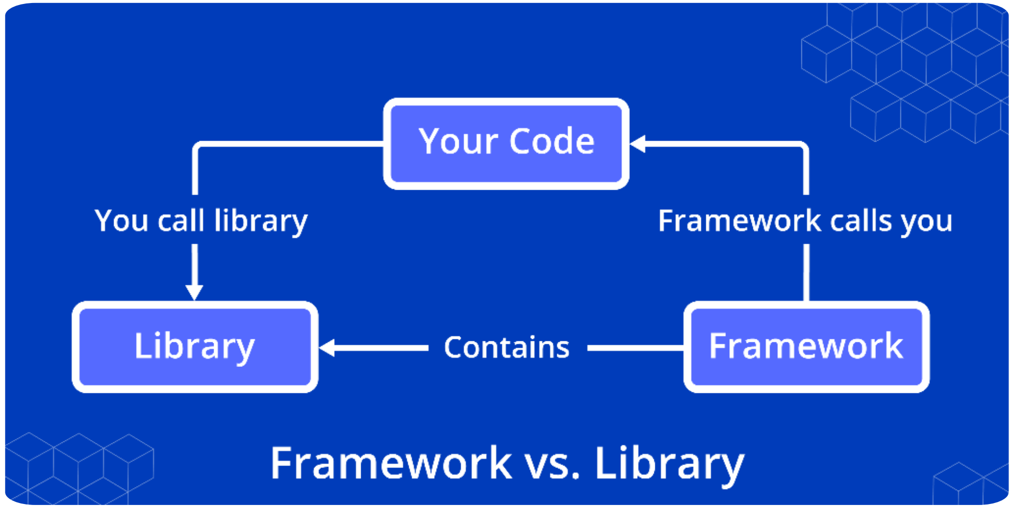
- **(Apache) NetBeans** is a development environment that is primarily used for **Java** application development but also supports JavaScript, Ruby, and PHP.
- Both Visual Studio and NetBeans are more than development environments—they are also **frameworks**, which means they actually provide common parts of the code of the application.


THE
STANDARD
C
LIBRARY
<time.h> * <limits.h> * <float.h>
<stddef.h> * <errno.h> * <locale.h>
<stdio.h> * <ctype.h> * <string.h>
<math.h> * <stdlib.h> * <assert.h>
<stdarg.h> * <setjmp.h> * <signal.h>

Framework vs Library

- A **library** is a collection of prewritten code that can be used to simplify tasks. For example, C language has a rich library.
- The “*Inversion of Control*” (IoC) describes the difference between a library and a framework.
- In some ways, you can think of a framework as a collection of libraries, but it’s entirely different.
- By using a library, you control the flow of the program. The library can be invoked whenever and wherever you like.

Framework vs Library



- Contrary to this, when you use a framework, the flow is controlled by the framework.
- The framework instructs you where to put your code, but it will call your code as required.
- Simply put, our code calls the library's code, but in a framework, it's the framework's code that calls our code.
- Developers can invoke libraries to perform specific tasks by using components, classes, and methods. A framework, however, already provides code to perform common tasks and uses code provided by a developer for custom functionality.

Combining Different Programming Languages

- While trying to solve real life problems with a programming language, in some cases it may be desired or needed to use another programming language other than the one used.
- In such cases, instead of translating the entire project into another programming language, only the relevant part can be written in a different programming language and this program can be called and run in the previously used programming language.

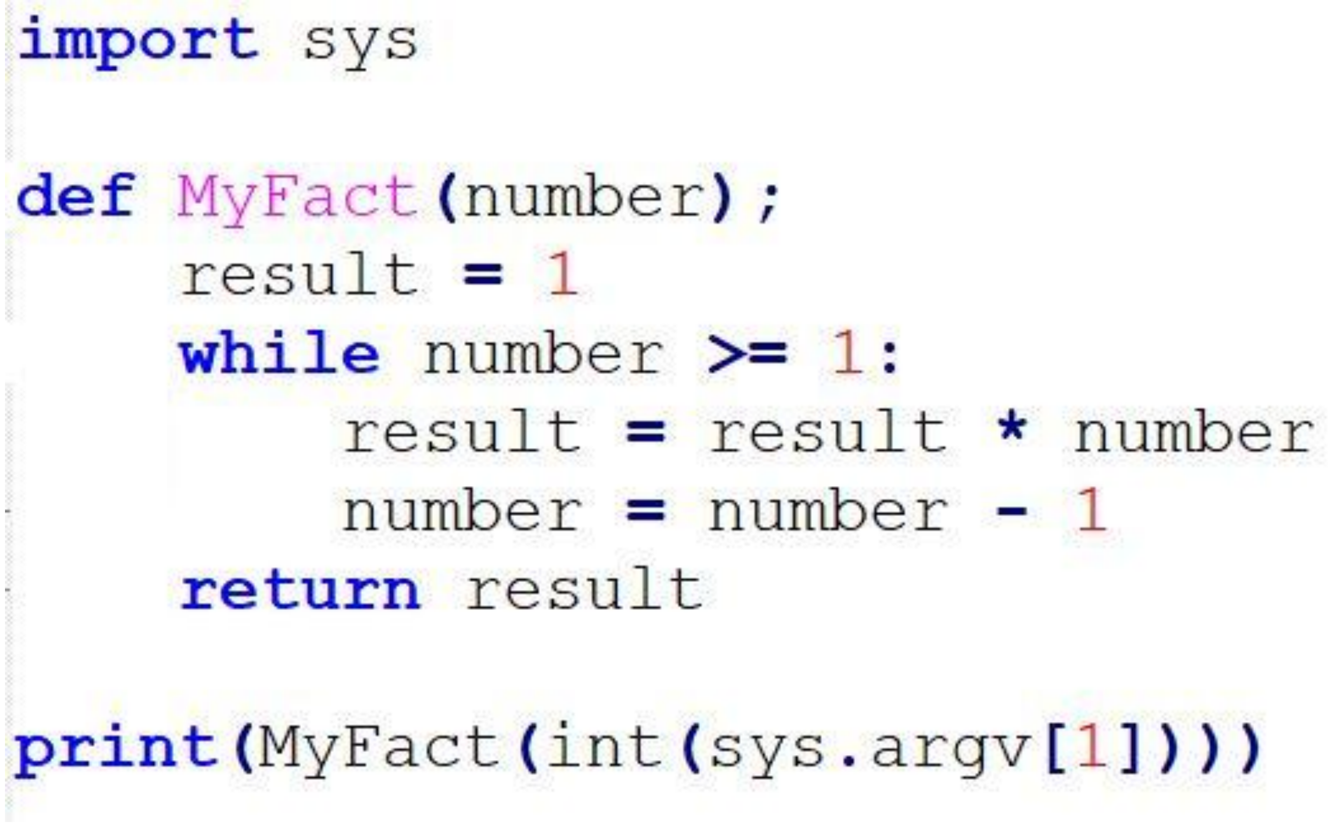
Combining Different Programming Languages

Integrating Java and Python: How to Call Python Code from Java
Java
python

- For example, if it is assumed that there is no available library in Java for a module in a program where the entire problem is written in Java language and that the Python language does this job quickly and easily, then Python can be run in Java and the result can be obtained.

Combining Different Programming Languages

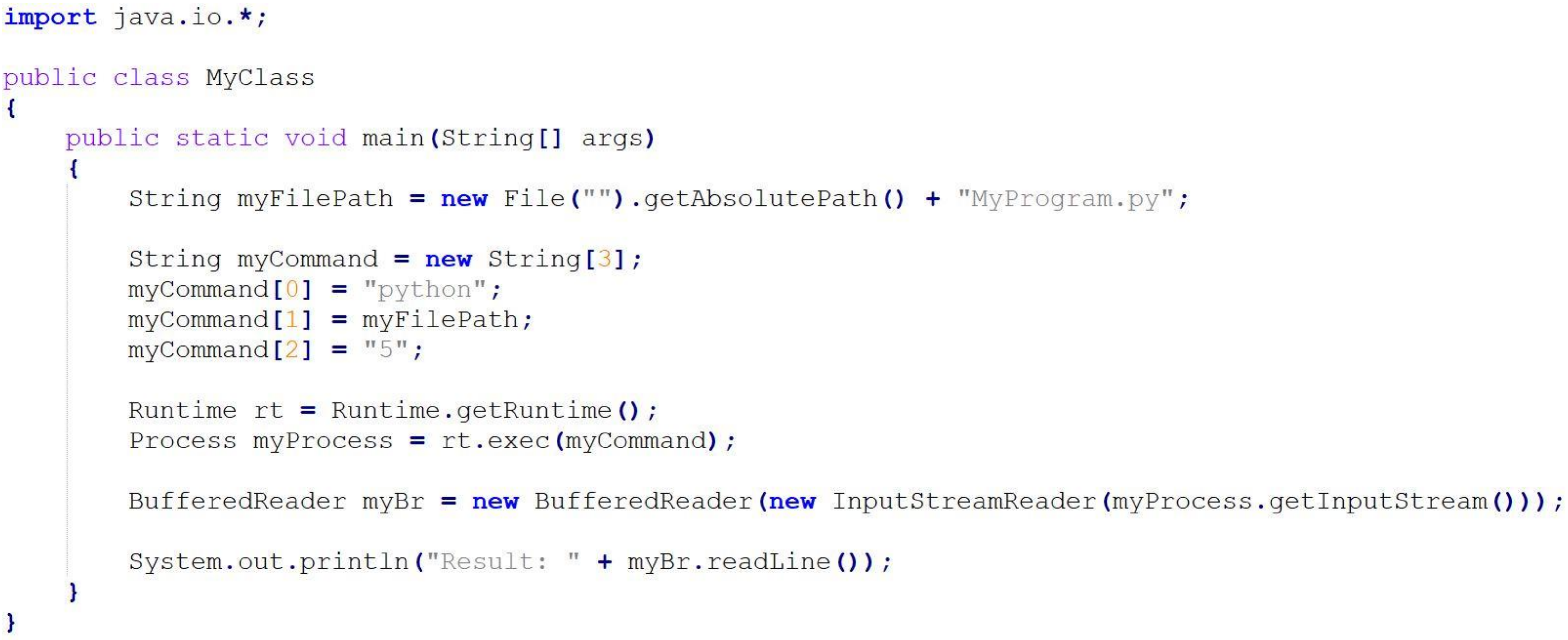
- In the figure, there is a Python code that calculates the factorial of a number.
- We assume the file is saved as “MyProgram.py”.



```
import sys

def MyFact(number);
    result = 1
    while number >= 1:
        result = result * number
        number = number - 1
    return result

print(MyFact(int(sys.argv[1])))
```

Combining Different Programming Languages

Below is an example Java program using this Python code.



```
import java.io.*;

public class MyClass
{
    public static void main(String[] args)
    {
        String myFilePath = new File("").getAbsolutePath() + "MyProgram.py";

        String myCommand = new String[3];
        myCommand[0] = "python";
        myCommand[1] = myFilePath;
        myCommand[2] = "5";

        Runtime rt = Runtime.getRuntime();
        Process myProcess = rt.exec(myCommand);

        BufferedReader myBr = new BufferedReader(new InputStreamReader(myProcess.getInputStream()));

        System.out.println("Result: " + myBr.readLine());
    }
}
```