01_2 Introduction to Java

Object-Oriented Programming

Why Java in This Course? (1/3)

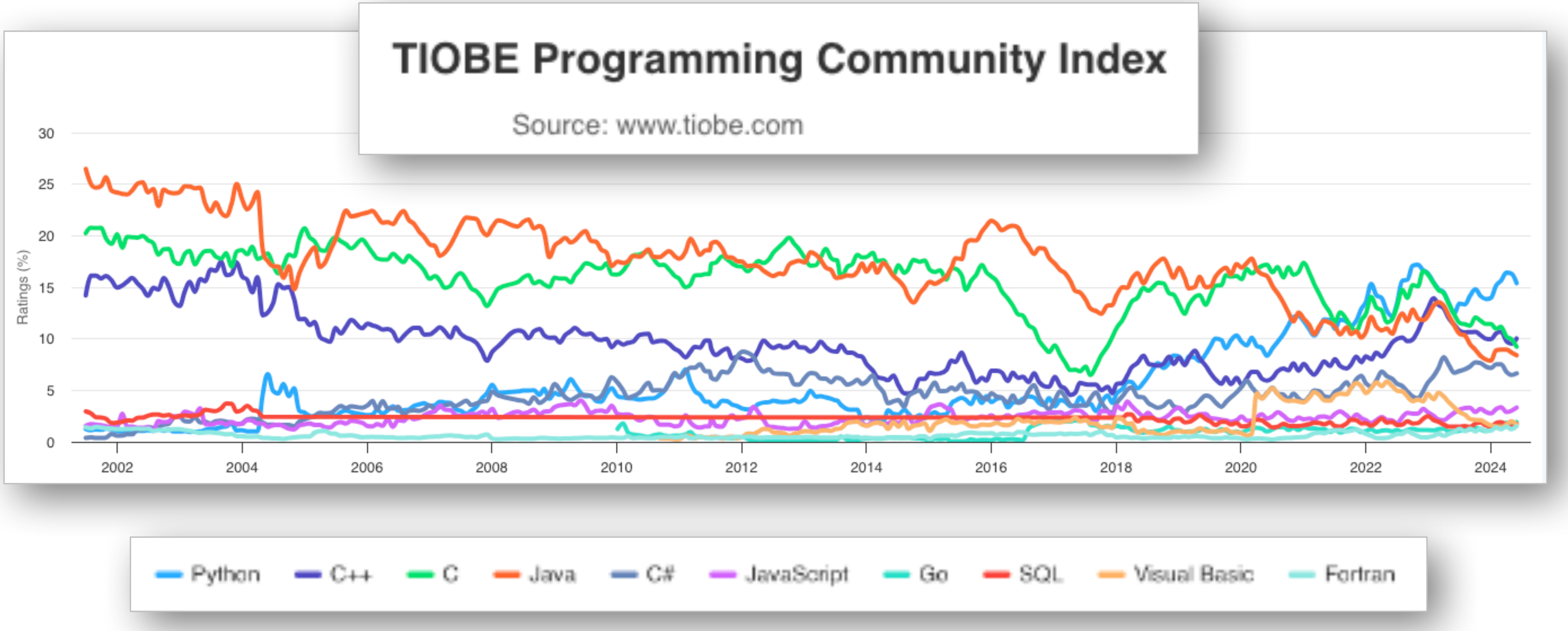
- Java:
  - Pure ideal object-oriented language
  - Everything should be in any class (object)

```java
public class IdealOOP {
    public static void main(String[] args) {
        int[] array = {3, 4, 5, 6, 7};
        float average = MyUtil.averageArray(array);
        System.out.println("average: " + average);
    }
}
```

Why Java in This Course? (2/3)

- Still popular
    - For large enterprise system
    - Backend (server) implementation
    - Native implementation language of Android
    - For cloud system development

Why Java in This Course? (3/3)



TIOBE Programming Community Index

Source: www.tiobe.com

History of Java (1/2)

- Early 1990s
  - Developed by Sun Microsystems, initially as a language for consumer electronics.
- 1995
  - Officially released, used to develop web applets with a "Write Once, Run Anywhere" philosophy
- Late 1990s - 2000s
  - Emerged as a mainstream language for enterprise application development
- Mid-2000s
  - Mainly used for developing large-scale enterprise systems

History of Java (2/2)

- Mid-2000s
  - Mobile application (feature phone app with J2ME) development
- 2010s
  - Dominant language for android app development
- Today
  - Utilized in a variety of fields
  - Web servers, enterprise systems, Android apps, Big data processing (Hadoop), IoT devices, etc.

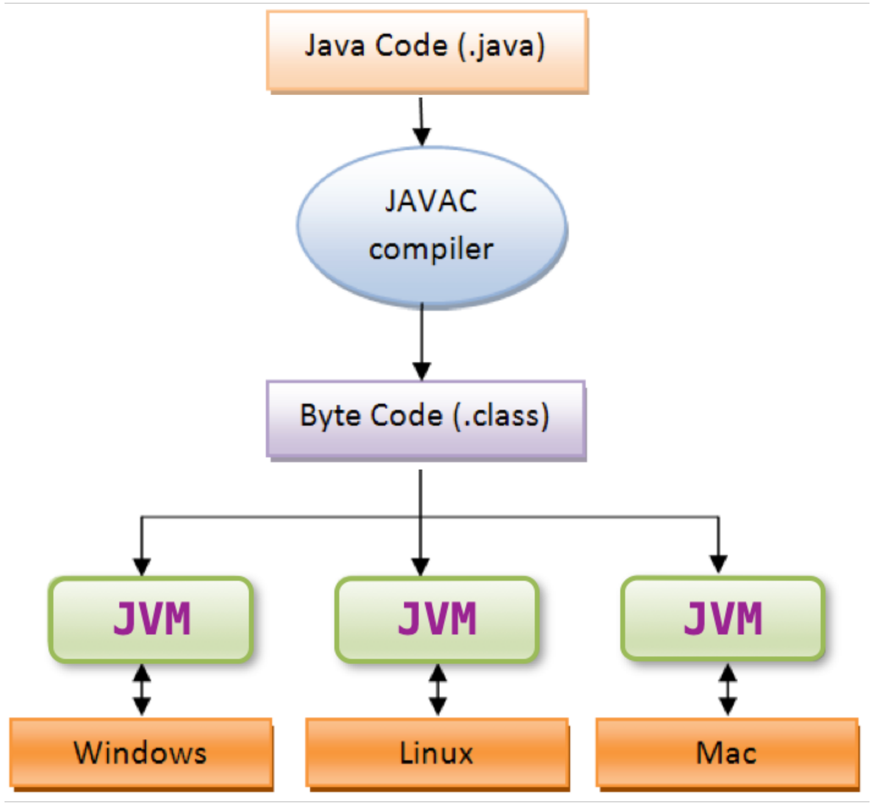
Future Outlook

- Cloud-native application development
- Artificial intelligence and machine learning
- Micro-services architecture
  - Modularization with reusable classes
- Ecosystem through interoperability with JVM languages
  - Ex) Kotlin, Scala, ...
- Performance improvements and support for new hardware architectures

Platform-Independent Features in Java

- Java: Cross-platform language
  - ○ WORA: Write-Once, Run-Anywhere

- Execution Process
  - ① Java source program (.java)
  - ② Compiled by “Javac” compiler
  - ③ Converted to Byte Code program (.class)
  - ④ Executed on JVM (Java Virtual Machine) in each different platform

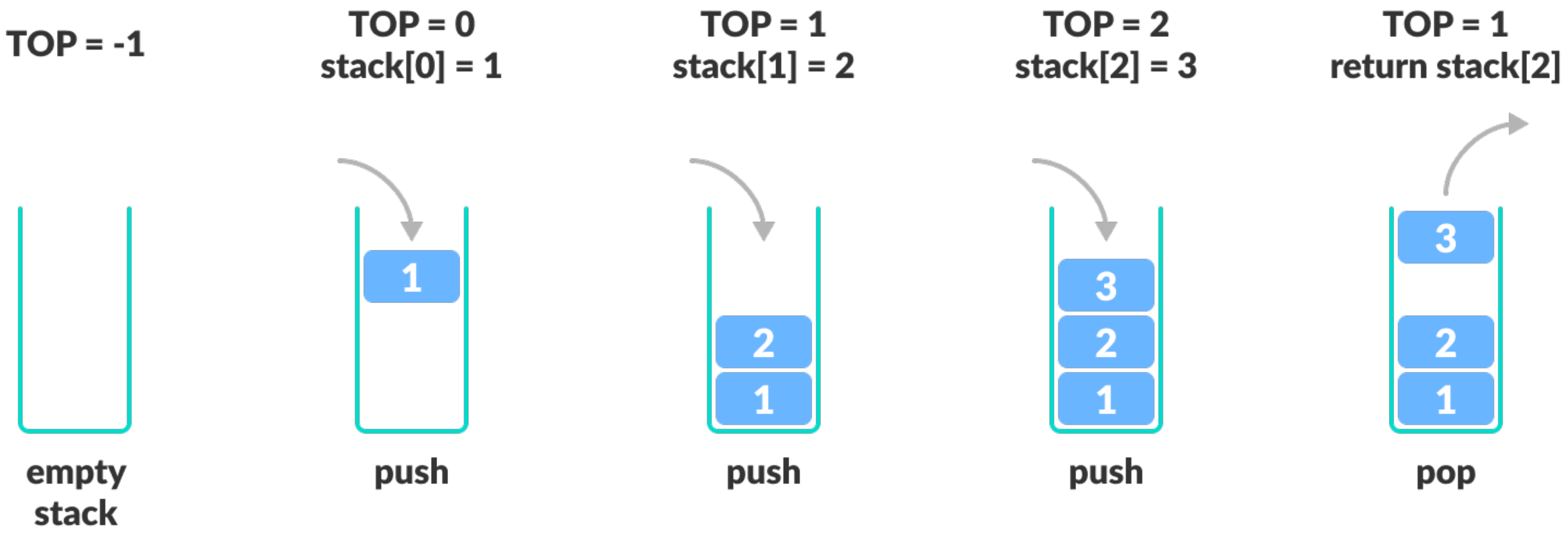


Java Source Code and Byte Code (1/3)

```java
public class Add {
    public static void main(String[] args) {
        int a = 5;
        int b = 3;
        int result = a + b;
        System.out.println(result);
    }
}
```

Java Source Code and Byte Code (2/3)

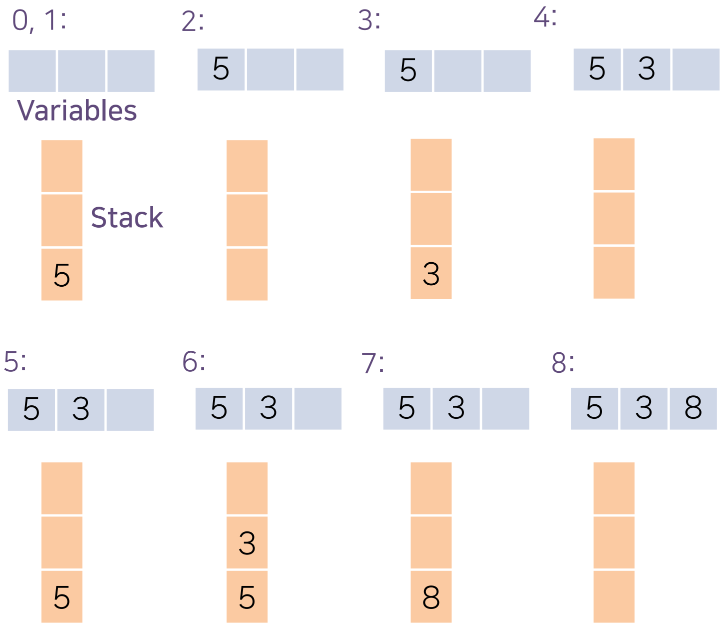
Stack Architecture in JVM



Java Source Code and Byte Code (3/3)

```
public class Add {
  public static void main(java.lang.String[]);
    Code:
       0: bypush     // push next byte to the Stack
       1: 5          // operand of previous “bypush”
       2: istore_0   // pop the top of Stack to v[0]
       3: iconst_3   // push constant 3 to the stack
       4: istore_1   // pop the top of Stack to v[1]
       5: iload_0    // push v[0] to the Stack
       6: iload_1    // push v[1] to the Stack
       7: iadd       // pop two values, add, and push
       8: istore_2   // pop the top of Stack to v[2]
       9: getstatic #2   // load class of ‘System.out’
      10: iload_2        // push v[2] to the Stack
      11: invokevirtual #3 // execute method ‘println’
      12: return
}
```

```
public class Add {
     public static void main(String[] args) {
          int a = 5;
          int b = 3;
          int result = a + b;
          System.out.println(result);
     }
}
```



Simple JVM Implementation (1/4)

```
if __name__ == “__main__":

    # Initialize JVM
    # Define constants
    STACK_SIZE = 100
    VAR_SIZE = 100

    # Initialize global variables
    stack = [0] * STACK_SIZE
    sp = -1  # Stack pointer (index of top)
    v = [0] * VAR_SIZE  # variables
```

Simple JVM Implementation (2/4)

```
# Bytecode program
bytecode = [
    0x10, 0x05,  # bipush 5
    0x3c,        # istore_0
    0x04,        # iconst_2
    0x3d,        # istore_1
    0x1a,        # iload_0
    0x1b,        # iload_1
    0x60,        # iadd
    0x3e,        # istore_2
    0x1c,        # iload_2
    0xb1         # return
]

# Execute bytecode
jvm_execute(bytecode)

# Print result
print(f"Result: {v[2]}")
```

Simple JVM Implementation (3/4)

```
def jvm_push(value):  # function jvm_push: push given value to the top of the stack
    global sp         # sp is the global variable we’ve defined
    sp += 1           # Increment the stack pointer (stack full case ignored)
    stack[sp] = value # Copy given value to the top of the stack

def jvm_pop():        # function jvm_pop: pop and return the top element of stack
    global sp
    value = stack[sp] # Copy the top element to the value
    sp -= 1           # Decrement the stack pointer (sp = sp - 1)
    return value      # Return the value
```

Simple JVM Implementation (4/4)

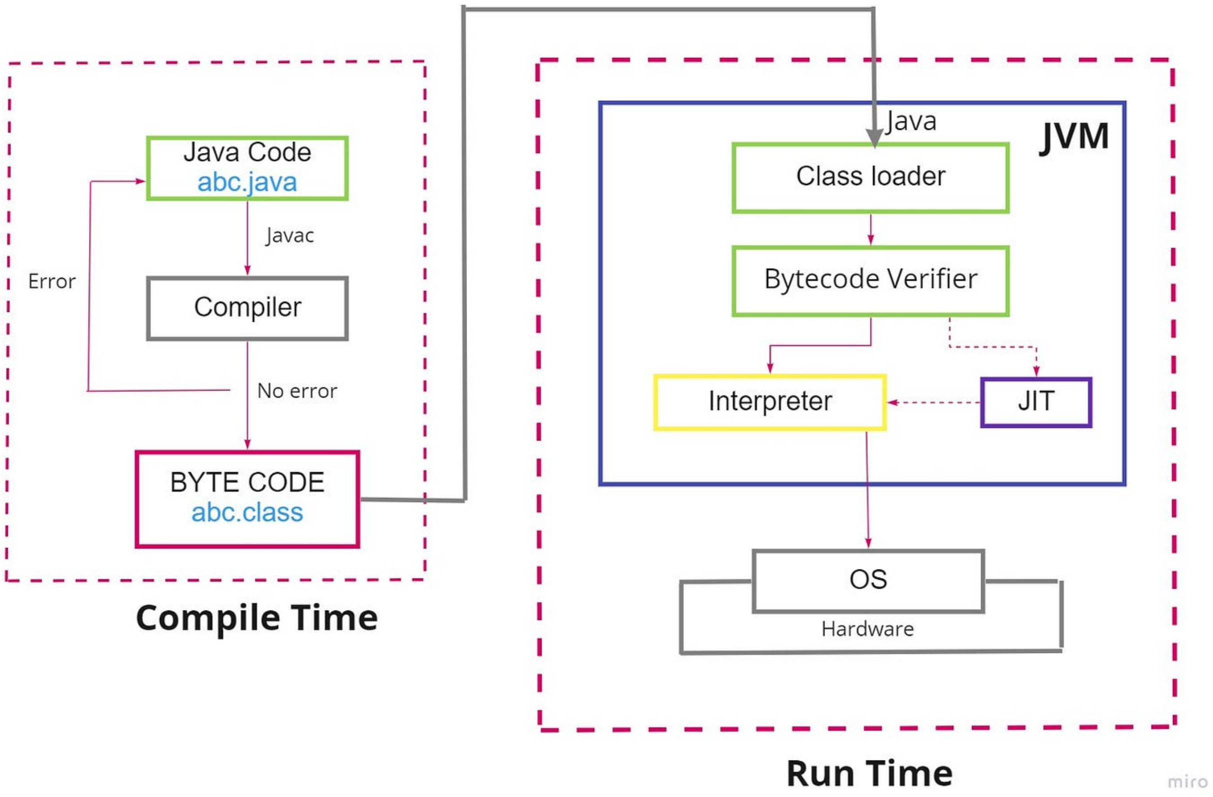
```python
def jvm_execute(bytecode):
    global sp, v
    pc = 0  # Program counter

    while pc < len(bytecode):
        opcode = bytecode[pc]
        pc += 1
        if opcode == 0x10:  # bipush
            jvm_push(bytecode[pc])
            pc += 1
        elif opcode == 0x3c:  # istore_0
            v[0] = jvm_pop()
        elif opcode == 0x04:  # iconst_3
            jvm_push(3)
        elif opcode == 0x3d:  # istore_1
            v[1] = jvm_pop()
        elif opcode == 0x1a:  # iload_0
            jvm_push(v[0])
```

```python
        elif opcode == 0x1b:  # iload_1
            jvm_push(v[1])
        elif opcode == 0x60:  # iadd
            b = jvm_pop()
            a = jvm_pop()
            jvm_push(a + b)
        elif opcode == 0x3e:  # istore_2
            v[2] = jvm_pop()
        elif opcode == 0x1c:  # iload_2
            jvm_push(v[2])
        elif opcode == 0xb1:  # return
            return
        else:
            print("Unsupported opcode")
            return
```

Compiler vs Interpreter

	Compiler	Interpreter
When to translate	Before running the program	Line -by -line during execution
Execution speed	Fast	Slow
Error detection	Detect all errors at compile time	During execution of the current line
Execution file	Executable file generated	No executable file generated
Example languages	C, C++, Java	Java, Python, JavaScript, Ruby

JIT (Just-In-Time) Compiler



- JIT
  - Compiles frequent Byte Code into machine code
  - Save the machine code
  - Use pre-compiled machine code later
  - Fast execution

Features of Java

- Object-Oriented
- Platform Independent
- Automated Garbage Collection
- Simple and Easy to Learn
- Robust and Secure
- Multithreaded
- Distributed
- High Performance
- Portable
- Rich Standard Library

Python vs Java: Program Structure

```
#Python: program without any class is possible

def printHello():
    print("Hello World!")

printHello
```

```
// Java: Everything should be in a class
// PythonVSJava.java: class name = file name

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

Python vs Java: Type System

```
# Python: Dynamic Typing

x = 5
x = "Hello" # OK
```

```
// Java: Static Typing

int x = 5;
x = "Hello";     // Compile Error
```

```
# Python: No Type Declaration

name = "Alice"
age = 30
```

```
// Java: Type Declaration

String name = "Alice";
int age = 30;
```

Python vs Java: Function / Method

```python
# Python: Function Definition

def greet(name):
    return f"Hello, {name}!"
```

```java
// Java: Method Definition

public static String greet(String name)
{
    return "Hello, " + name + "!";
}
```

Python vs Java: Class Definition

```python
# Python: Class Definition

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def say_hello(self):
        print(f"Hello, I'm {self.name}")
```

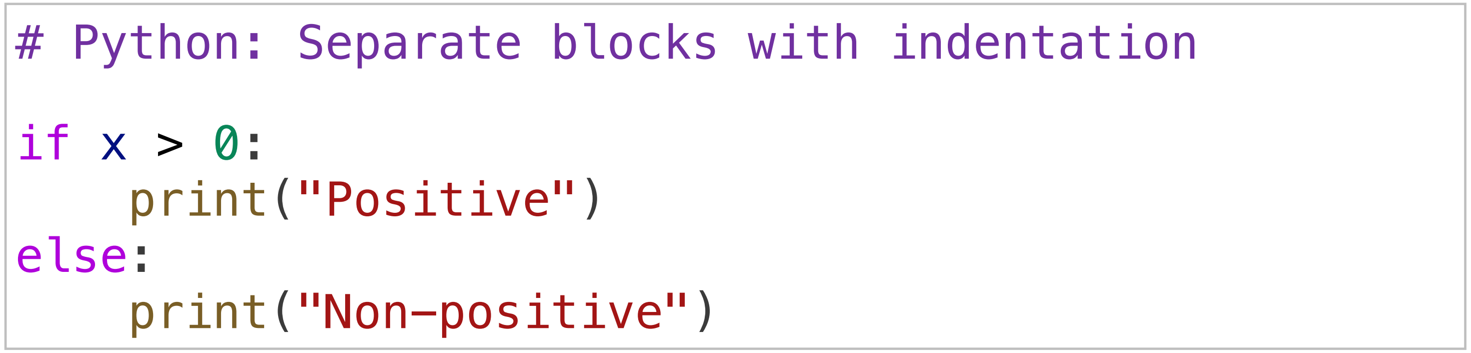
```java
// Java: Class Definition

class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void sayHello() {
        System.out.println("Hello, I'm " + this.name);
    }
}
```

Python vs Java: List / Array

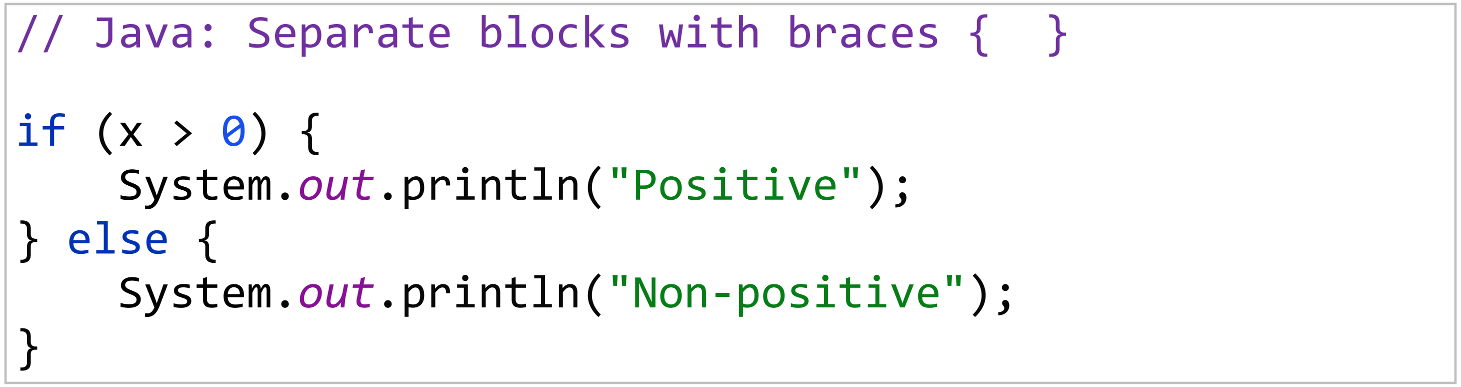
```python
# Python: List and Array

fruits = ["apple", "banana", "cherry"]
fruits.append("date")
```

```java
// Java: Array and List

// Array of Strings: Fixed size
String[] fruits = {"apple", "banana", "cherry"};

// Convert Array to ArrayList to add more elements
ArrayList<String> fruitList = new ArrayList<>(Arrays.asList(fruits));
fruitList.add("melon");
```

Python vs Java: Code Blocks



```python
# Python: Separate blocks with indentation

if x > 0:
    print("Positive")
else:
    print("Non–positive")
```



```java
// Java: Separate blocks with braces {   }

if (x > 0) {
    System.out.println("Positive");
} else {
    System.out.println("Non-positive");
}
```

Python vs Java: Iteration (for statement)

```python
# Python: For statement uses 'in' keyword
for fruit in fruits:
    print(fruit)
for i in range(5):   # i = 0, 1, 2, 3, 4
    print(i)
```

```java
// Java: For statement uses ': (for each)' or
// 'initialization; condition; update'

for (String fruit : fruits) {
    System.out.println(fruit);
}

for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

Python vs Java: Translation Method and Others

- Python
  - Interpretation, Slow execution
  - Simple grammar, Easy to learn
  - Rich scientific and engineering libraries
  - Applications: Prototyping in research, Data analysis, Machine learning, ...

- Java
  - Interpretation (but fast using JIT), Faster than Python
  - Strict grammar for reducing errors
  - Rich standard libraries and 3rd party libraries for enterprise applications
  - Applications: Enterprise applications (ex. servers), Cloud native implementation, Android's native implementation