01_2 Introduction to Java

Object-Oriented Programming

이번 강의에서는 이번 코스에서 사용하게 될 Java 언어에 대해 간단하게 소개하도록 하겠습니다.

Why Java in This Course? (1/3)

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

```
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);
    }
}
```

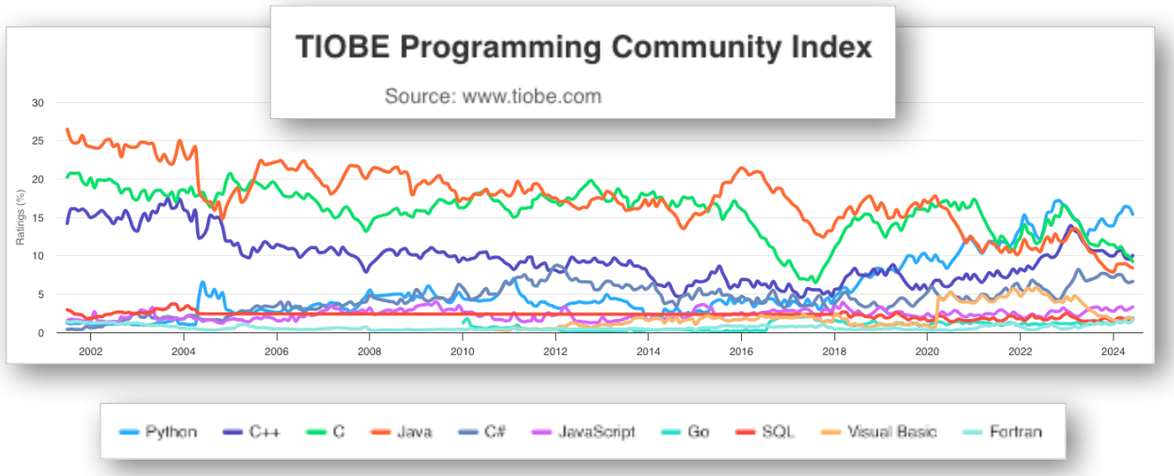

먼저 왜 우리의 OOP 과목에서 Java를 선택하였는가에 대해 설명이 필요할 것 같습니다.
Java는 순수한 OO language 그 자체라고 볼 수 있습니다.
때문에 OOP의 주요 concept들을 공부하기에 이상적인 언어 입니다.
예를 들면 Java 프로그램에서는 모든 변수나 method들이 반드시 어떤 class 안에 속해야 합니다.
소속 class가 없는 code는 Java에서는 존재할 수 없습니다.
이 Java 프로그램을 보아도 모든 코드는 class IdealOOP 안에 구현되고 있습니다.
따라서 Java 프로그래밍은 class들을 계속 구현해 나가는 것이라 할 수 있습니다.
우리 course에서 Java를 사용하는 또 다른 이유는 Java가 오래된 언어이기는 하지만 아직도 많은 곳에 사용되고 있다는 점 때문입니다.
예를 들면 대규모 기업 시스템 구현, 서버 구현, Android 환경에서의 앱 구현 등에 많이 사용되고 있습니다.

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

우리 course에서 Java를 사용하는 또 다른 이유는 Java가 오래된 언어이기는 하지만
아직도 많은 곳에 사용되고 있다는 점 때문입니다.
예를 들면 대규모 기업 시스템 구현,
서버 구현에 많이 사용되며,
Android operating system이 Java로 구현되어 있고,
클라우드 시스템의 구현에도 많이 사용됩니다.

Why Java in This Course? (3/3)



이것은 programming language들이 얼마나 사용되는지에 따른 ranking을 보여주는 TIOBE index 입니다.
여기서 2002년부터 2010년대 중반까지 주황색으로 표시된 Java는 거의 ranking 1, 2위를 넘나들고 있습니다.
하늘색으로 표시된 Python이 Deep learning이 출현한 2010년대 중반부터 약진하기 시작하여,
2024년 현재, 단연 랭킹 1위를 차지하고 있습니다.
그 아래로 전통적으로 많이 쓰이던 C, C++, Java가 뒤따르고 있는 것을 볼 수 있습니다.

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

Java 언어의 역사를 응용 분야 중심으로 간략히 소개해 보겠습니다.
1990년대 초반 Java는 Sun Microsystems에서 가전제품용 프로그래밍 언어로 개발 되었습니다.
1995년에 Java가 공식적으로 발표되는데, "Write Once, Run Anywhere" 라는 철학을 가지고,
초기에는 웹에서 구동되는 어플리케이션, 즉, 애플릿의 개발에 주로 사용되었습니다.
1990년대 후반부터 2000년대 들어서 차츰 기업 애플리케이션 개발에 사용되기 시작하였으며,
2000년대 중반 부터는 대규모 기업 시스템 개발의 주력 언어로 부상하였습니다.

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.

2000년대 중반에는 또한 모바일 애플리케이션 개발 분야에 이용되기 시작하였으며,
스마트 폰 이전 시대인 피처폰 애플리케이션 개발에 많이 사용되었습니다.
2010년대에 들어서는 안드로이드 시스템이 Java로 구현되면서, 안드로이드 앱 개발의 주요 언어로 자리매김하게 됩니다.
Java는 현재까지도 다양한 분야에서 활용되고 있는데, 웹 서버, 엔터프라이즈 시스템,
안드로이드 앱,
빅데이터 처리를 위해 여러 개의 컴퓨터들을 하나로 묶어 대용량 데이터를 처리하는 기술인 Hadoop,
Internet of Things (IOT)의 디바이스 컨트롤 등이 대표적인 분야들 입니다.

Future Outlook

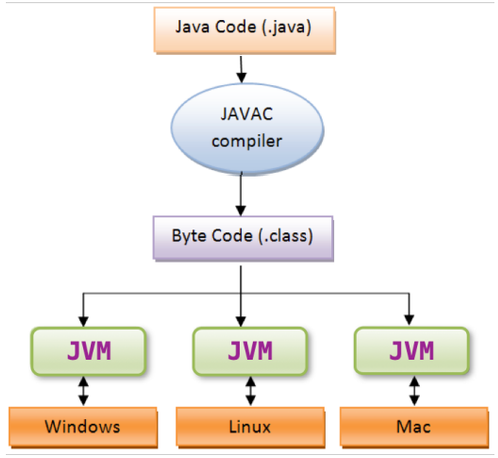
- Cloud-native application development
- Artificial intelligence and machine learning
- Micro-services architecture
  - Modularization with reusable classes
- Ecosystem through interoperability with JVM languages
  - Ex) Kotlin, Scala, ...
- Performance improvements and support for new hardware architectures

Java는 앞으로도 계속 많이 사용될 것으로 예측됩니다.
Java 언어는 클라우드에서 실행되는 다양한 애플리케이션 개발에서의 지속적인 역할이 기대되고,
인공지능, 머신러닝 분야에서의 활용이 점차 확대되며,
마이크로서비스 아키텍처에서의 중요성이 계속 유지되어, 세부 기능을 담당하는 reusable class들이 점차 증가되고,
Byte code로 compile되는 다른 JVM 언어들(Kotlin, Scala 등)과의 상호운용성을 통해 Java 생태계 자체의 확장과 함께
성능 개선 및 새로운 하드웨어 아키텍처 지원을 통해 계속적으로 경쟁력이 유지될 것으로 예상됩니다.

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



Java는 크로스 플랫폼 프로그래밍 언어입니다.
"WORA", 즉, "Write Once Run Anyware" 는 Java의 이러한 특징을 일컫는 말 입니다.
Java 언어로 작성된 프로그램을 실행하기 위해서는, JAVAc 컴파일러로 컴파일하여,
모든 유형의 프로세서 또는 운영 체제에 대해 동일한 중간 언어인 "Byte Code"로 표현되는 ".class" 파일을 얻습니다.
이 .class 파일은 JVM (Java Virtual Machine)이라는 소프트웨어 가상 머신에서 실행됩니다.
즉, 바이트 코드는 JVM의 machine language로 볼 수 있습니다.
물론 JVM은 다른 플랫폼마다 다르게 구현된 platform-dependent 소프트웨어이지만,
바이트 코드는 매우 간단한 형식을 가지고 있기 때문에 JVM을 구현하는 난이도가 많이 높은 것은 아닙니다.

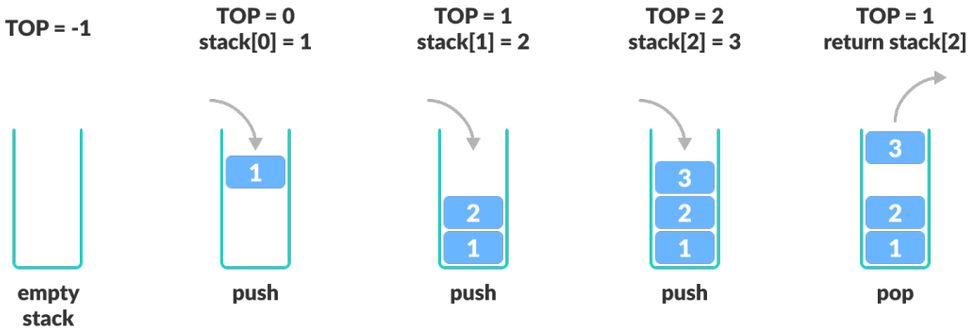
Java Source Code and Byte Code (1/3)

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


이 프로그램은 Add 라는 class로서 두 정수 a, b의 값을 더하여 출력하는 간단한 프로그램입니다.
나중에 더 자세히 설명하겠지만 class가 하나의 프로그램으로 실행되기 위해서는 'main' method를 가져야 하고
Main method 부터 프로그램이 실행되게 됩니다.
먼저 Integer type의 두 variable a와 b에 각각 5와 3 이라는 값을 assign 합니다.
또다른 integer type variable인 result는 a와 b를 더한 값을 가지게 되고
결과 값인 result를 screen에 출력하게 됩니다.
이 프로그램이 Byte code 로 컴파일 된 후에 JVM 에서 실행되는 과정을 조금 더 자세히 살펴보겠습니다.

Java Source Code and Byte Code (2/3)

Stack Architecture in JVM

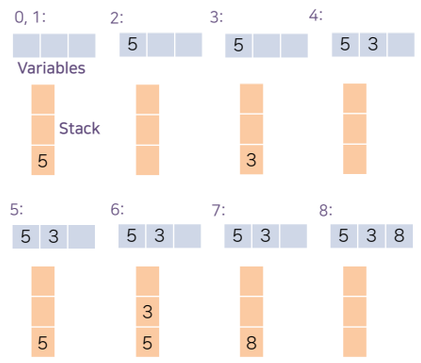


JVM은 간단한 Stack machine architecture를 가지고 있습니다.
그림에서 보듯 Stack에는 input과 output이 한 방향에서만 이루어 집니다.
Empty stack일 때 Stack Top을 가르치는 TOP index는 -1이고,
data 하나를 push하면 stack[TOP] 에 저장되며, TOP index를 증가시킵니다.
이런 식으로 push operation은 기존의 Top element 위에 새로운 data를 하나씩 쌓아 나가게 됩니다.
Pop operation은 현재의 Top element를 꺼내고 TOP index를 하나 감소 시킵니다.

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);
     }
}
```



이 프로그램은 Java source code를 javac를 통해 컴파일하여
그 출력으로 얻은 Byte Code 프로그램의 한 예입니다.
Byte Code의 각 line에는 line number가 붙어 있으며,
모든 line은 단 하나의 byte (즉, 8bits 또는 두 개의 16진수 digits) 로만 표시될 수 있습니다.
Byte Code Program의 시작 전에
JVM의 Variable array와 Stack은 initialize 되어 있습니다.
Line 0에서 bypush 명령 (operation) 을 만나는데,
이 명령은 바로 뒤 byte의 수 (operand)를 stack에 push하라는 명령입니다.
Line 1까지 실행 한 후 Variable memory와 Stack의 상태를 오른쪽 그림에서 보여주고 있습니다.
그림에서 볼 수 있듯, Stack에는 5가 push되어 있으며, top index = 0 입니다.
Line 2에서는 Stack의 top element를 pop하여 V{0]에 저장합니다.
Line 3에서는 constant 3을 stack에 push합니다.
Line 4에서는 Stack의 top element를 pop하여 V[1]에 저장합니다.
Line 5, 6에서는 V[0]와 V[1]을 Stack에 push합니다.
Line 7에서는 Stack에서 두 개의 element를 pop하여 이를 더한 후 그 결과를 다시 push합니다.
즉, 여기서는 3과 5를 pop하여 더한 후, 그 결과인 8을 push하게 됩니다.
Line 8에서는 Stack의 top element를 pop하여 V[2]에 save합니다.
Line 9 부터 12까지는 결과인 V[2] = 8을 display하는 부분입니다.

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
```


앞 슬라이드의 Byte Code는 JVM이라 불리는 interpreter에서 실행됩니다.
Compiler는 일단 source program을 target code로 완전히 translate하여
output target code file로 저장해 줍니다.
그러나 Interpreter는 compiler와 달리 command를 하나씩 바로 바로 실행해 나갑니다.
즉, compiler 처럼 translation을 마친 output file을 만들지 않는다는 것입니다.
JVM은 비교적 단순하기 때문에 어렵지 않게 구현할 수 있습니다.
여기서는 Python으로 JVM을 구현해 보았습니다.

이 "main" part는 사실 Python 프로그램에서 뒤 쪽에 위치해야 하지만,
이해를 쉽게하기 위해 먼저 살펴 보도록 하겠습니다.
JVM의 stack과 variable을 list로 정의하기 위해 그 size를 각각 100으로 잡았습니다.
Stack pointer sp는 초기값을 -1로 하는데, 현재 stack top의 index를 유지합니다.
그리고 stack과 variable list의 element들을 모두 0으로 초기화 했습니다.

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]}")
```


Bytecode program은 앞에서 설명한 것 처럼, 두 자리수 16진수의 list로 저장되어 있습니다.
이 16진수들을 순서대로 읽어 JVM의 상태를 바꾸며 프로그램을 interpreting하면서 실행하게 됩니다.
실행을 담당하는 function이 jvm_execute입니다.
실행이 다 끝난 후, v[2]의 value를 결과로 print합니다.

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
```


jvm_push function은 주어진 value를 stack에 push하는 function입니다.
현재 stack pointer (sp) 의 값을 하나 증가시키고 stack list의 그 인덱스에 value를 assign합니다.

jvm_pop function은 현재 stack의 top, 즉, stack[sp] 의 value를 return하면서
sp의 값을 감소시켜 stack의 top index를 하나 감소 시킵니다.

Simple JVM Implementation (4/4)

```
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])
        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
```

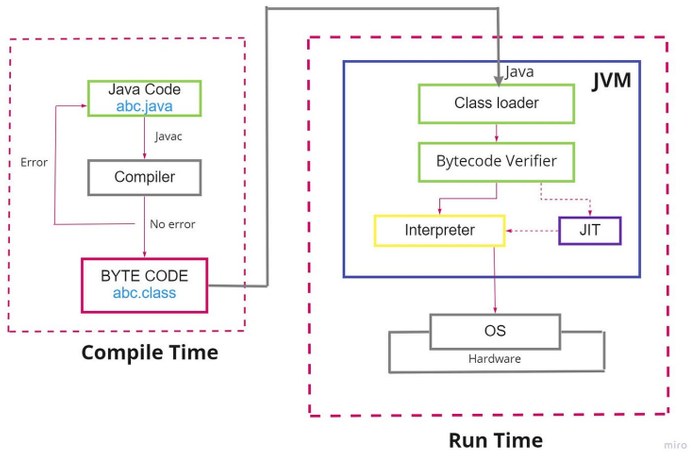

jvm_execute 함수는 bytecode 프로그램이 저장된 list를 parameter로 받아서
한 byte씩 명령을 해석하고 이를 실행하는 일을 합니다.
Global variable인 sp는 JVM의 stack pointer를 말하며
v는 JVM 내부의 variable array 입니다.
pc는 "program counter" 의 약자인데,
bytecode 프로그램의 명령들을 하나씩 읽어나가기 위한 index로서
프로그램을 읽어나가면서 증가하게 됩니다.
While 문의 body에서 bytecode list의 entry들을 하나씩 해석하고
그에 맞는 작업을 수행합니다.
예를 들어 0x3c는 "istore_0" 명령인데, stack top에 있는 값을 v[0]로 옮기는 일을 하는 것입니다.
나머지 명령들도 이전 슬라이드에서 설명했었습니다.
한번 bytecode 프로그램을 따라가면서 확인해 보시기 바랍니다.

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

이 표는 Compiler와 Interpreter의 차이를 나타내고 있습니다.
먼저 translation 시점 면으로, compiler는 target code가 실행되기 전에 compile이 다 끝나야 합니다.
하지만 interpreter는 실행 중에 line-by-line으로 translation이 됩니다.
실행 속도 면에서는 compiler가 interpreter보다 빠릅니다.
compiler는 일단 translation이 다 끝난 후 프로그램을 실행하기 때문에
한 라인씩 실행하는 interpreter보다는 확연히 빠릅니다.
Error를 찾는 시점은 compiler의 경우 compile time에 찾을 수 있는 error들을 모두 찾아내는 반면
interpreter의 경우는 현재 실행하고 있는 line에서의 error만 찾을 수 있습니다.
Compiler의 경우 실행 파일이 생기지만, Interpreter는 한 라인씩 실행되기 때문에 실행 파일이 만들어 지지 않습니다.
Compiler를 사용하는 언어로는 대표적으로 C, C++, Java (from source to bytecode) 가 있습니다.
Interpreting 언어로 대표적인 것들은 Java (bytecode 실행), Python, JavaScript, Ruby 등이 있습니다.

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

Java 프로그램의 실행 과정을 다시 살펴보면,
Compile time에는 java source code를 compiler가 bytecode 프로그램으로 translate합니다.
이 bytecode 프로그램은 software로 구현된 Java Virtual Machine (JVM) 내에서
한 줄씩 interpreting되면서 실행됩니다.
Interpreting 과정이 상대적으로 느리기 때문에 속도를 보완할 여러가지 장치들이 연구되었습니다.
JVM 에는 JIT, 즉 Just-In-Time Compiler라는 장치로 interpretation의 속도를 빠르게 하고 있습니다.
JIT는 자주 나올 법한 byte code 부분을 미리 JVM이 구동 중인 실제 컴퓨터의 machine 언어로 compile 해 둡니다.
나중에 다시 그 byte code 부분이 실제로 출현하면 즉시 미리 compile 해 둔 machine 언어 버전을 사용하여
빠른 실행이 가능하게 하는 것입니다.

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

이제 Java 언어의 특징들을 정리해 보겠습니다.

먼저 Java는 class를 기본 단위로 하는 object-oriented 언어 입니다.
또, bytecode와 JVM을 사용한 메카니즘으로
똑같은 source code를 컴퓨터 하드웨어의 종류나 OS에 상관없이 실행할 수 있습니다.
이같은 특징을 "platform independent" 라고 합니다.
Java 언어는 프로그램 실행 중에 더 이상 쓰이지 않고 있는 memory를 자동으로 모아서
재 사용 가능하도록 관리해 주는데
이러한 기능을 "자동 Garbage Collection" 이라 합니다.
이 기능 덕분에 Java는 C++ 에서와 같은 복잡한 pointer나 dynamic memory management가 없기 때문에
배우기 쉽습니다.
Static typing을 사용하기 때문에 엄격한 데이터 type에 대한 규칙이 지켜져야 하는데
이는 programmer의 실수를 방지하는데 도움이 됩니다.
Java는 또한 process보다 작은 thread 단위의 job들이 동시에 실행되도록 프로그래밍 할 수 있는 기능을 가지고 있습니다.
Java는 뛰어난 Networking 기능을 제공하며,
JIT 등으로 interpreter의 한계를 극복하여 빠른 실행이 가능합니다.
JVM만 구현되면 새로운 platform에서도 쉽게 실행이 가능하며,
유용한 많은 standard library 들이 제공되고 있습니다.

Python vs Java: Program Structure

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

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

printHello
```

```java
// 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!");
    }
}
```


Java의 특징을 한눈에 훑어보기 위해 Java와 Python 언어의 차이점들을 알아보도록 하겠습니다.
먼저 program 구조를 살펴보면
Python 은 class를 사용하지 않고도 program 작성이 가능합니다.
물론 Python 언어도 OOP 개념을 가지고 있으며, class 도 정의할 수 있습니다만
class를 반드시 사용해야 하는 것은 아닙니다.
그러나 Java 프로그램의 경우에는 모든 것이 class에 속해 있어야 합니다.
이 example 프로그램의 경우는 PythonVSJava 라는 class 하나가
프로그램 전체를 이루고 있습니다.

Python vs Java: Type System

```python
# Python: Dynamic Typing

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

```java
// Java: Static Typing

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

```python
# Python: No Type Declaration

name = "Alice"
age = 30
```

```java
// Java: Type Declaration

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


Type system 측면에서 보면, Python 에서는 variable의 type이 고정되지 않습니다.
이 예제에서 보면 variable x는 value 5가 assign이 되는 순간 "integer" type이 되며,
"Hello" 를 다시 x에 assign 하는 순간에 x는 String type으로 변화합니다.
그러나 Java에서는 이러한 자유로운 type 변화는 불가능 합니다.
모든 variable들은 사용되기 전에 그 type이 고정되며,
프로그램 실행 중간에 variable의 type을 바꿀 수 없습니다.

또 Python에서는 variable이 사용되기 전에 반드시 그 type을 알리는 declaration이 필요하지 않습니다.
그러나 Java에서 variable은 반드시 declare 된 이후 사용되어야 합니다.

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 + "!";
}
```


Python에서는 function이 "def" 문을 사용하여 정의됩니다.
Java에서는 class에 속한 method로 function이 구현됩니다.

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);
    }
}
```


Python에도 class구조가 존재합니다.
class는 variable들과 function들을 포함합니다.
이 example에서는 name과 age라는 variable이 class에 포함되고
__init__ 과 say_hello 는 function들 입니다.
특히 __init__ 은 constructor (생성자) 로서
Class object가 생성되는 순간에 자동으로 실행 됩니다.
Java는 프로그램 전체가 한 개 이상의 class들로 구성됩니다.
이 예에서도 class Person은 name과 age라는 두 개의 data를 가지고 있고
constructor인 Person method와 sayHello() method를 가지고 있습니다.

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");
```


Python에는 전통적인 array의 개념과 유사한 list 구조가 있습니다.
List 구조는 array보다 훨씬 더 유연하며,
많은 기능을 가지고 있습니다.
Java에는 각 type마다 array를 만드는 메카니즘이 존재합니다.
또 ArrayList와 같은 collection framework을 사용하면 훨씬 더 많은 기능들을 사용할 수 있습니다.

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");
}
```


Code 블럭의 경우 Python은 별다른 구분자를 사용하지 않고 띄어쓰기로 같은 block을 표현하게 됩니다.
Java의 경우는 중괄호 (brace) { } 로 begin과 end를 명확히 표시하고 있습니다.

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);
}
```


Python의 경우 for 문은 list안에 속하는 element들에 대해 반복적인 연산을 수행합니다.
이 예에서는 fruits라는 list 안의 fruit 들에 대해서
또 0부터 4까지의 integer에 대해서 반복되는 연산을 수행합니다.

Java의 경우에도 비슷하게 for문은 array내의 각 element들에 대한 반복적인 연산을 수행할 수 있게 합니다.
또한, 정수 i를 0부터 4까지 증가시키면서 연산을 수행할 수 있게 할 수 있습니다.

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

Python과 Java는 모두 interpretation을 사용하기 때문에 기본적으로 프로그램 수행이 느려집니다.
그러나 Java의 경우에는 JIT와 같은 메카니즘을 사용하여 수행 속도를 더 빠르게 할 수 있습니다.
Python은 비교적 간단한 문법을 가지고 있고 배우기 쉽습니다.
그에 비하면 Java는 매우 엄격한 문법을 가지고 있는데,
이 때문에 프로그래머가 의식하지 못한 채 error를 포함한 프로그램을 작성하는 것을 방지할 수 있습니다.
Python은 과학, 공학에 광범위하게 사용되는 library 함수들을 포함하고 있으며
Java에는 기본적으로 제공되는 standard library가 풍부하고,
기업 어플리케이션을 위한 써드파티 library들이 많이 존재합니다.
따라서 Python은 수행속도가 중요시되지 않는 연구 분야의 프로토타이핑, 데이터 분석, 기계학습 등에 널리 사용되고 있으며,
Java는 서버 구현과 같은 기업 솔루션, 클라우드 구현, 안드로이드 어플리케이션 구현 등에 많이 사용되고 있습니다.