

Computer Language

Course overview

Intro

■ Tutor

- Jin-Woo Jeong
- Tel:6483, e-mail: jinw.jeong at seoultech.ac.kr
- Office: Changhak 103

■ Class homepage

- E-class (<http://eclass.seoultech.ac.kr>)

■ Pre/Post-requisites

- Programming language
- Data structure, Mobile programming, Algorithm, etc.

Curriculum

■ Goal

- This module aims to provide a basic understanding of software solutions: their analysis, design, development and implementation including appropriate methodologies and skills in the use of an event driven language for open source development (Java)
- This module focuses on Object Oriented Programming using Java

■ Schedule

- Java basics (W2~W7)
 - Types, Basic operators, Conditions, Loop, OOD/P
- Java advances (W9~W14)
 - Exception, Collections, Generic, IO

Curriculum (cont'd)

■ Assignment

- Laboratory coursework / week (application programming)
- Submission: e-class

■ Evaluation

- Midterm: 30%
- Final: 40%
- Homework: 30%

Curriculum (cont'd)

■ Textbook

- Slide
- Reference
 - 혼자공부하는 자바, 한빛미디어
 - 이것이 자바다, 한빛미디어
 - Java: An Introduction to Problem Solving & Programming, 7th Ed., Savitch & Mock, Pearson, 2015.
 - OCA Java SE 8 Programmer I Exam Guide (Exams 1Z0-808) (Certification & Career - OMG) , Kathy Sierra, Bert Bates, McGraw-Hill Education (May 5, 2017)

■ Important Note

- No cheating, please!

Curriculum (cont'd)

■ Development environment

- Java 11 (LTS)
- IDE: IntelliJ
- Windows 10+

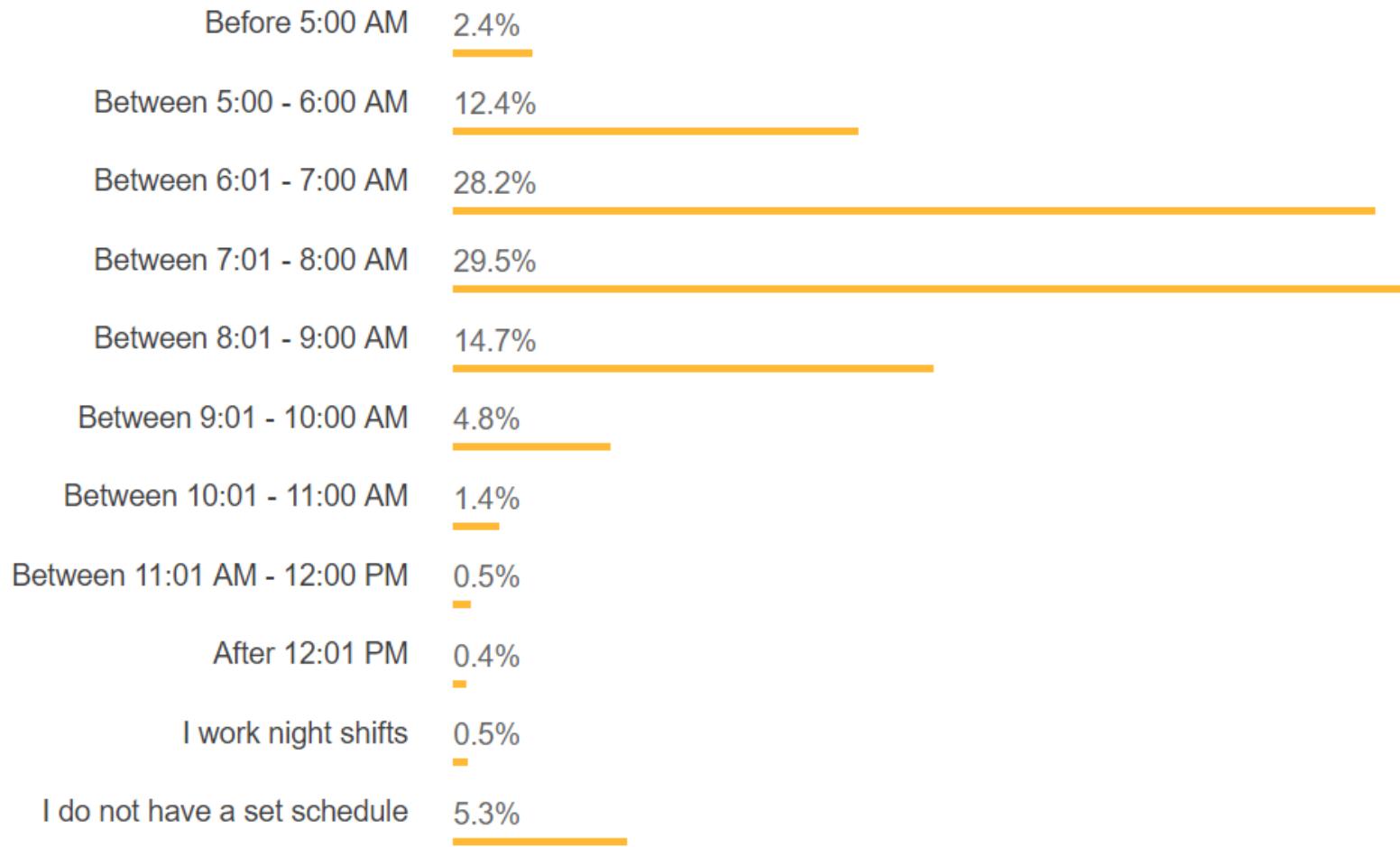
Stack Overflow Developer Survey 2020

■ <https://insights.stackoverflow.com/survey/2021>



Developer Survey - Developer Profile

What Time Do Developers Wake Up? 2018



Developer Survey - Developer Profile (cont'd)

How Much Time Do Developers Spend on a Computer? 2018

How Much Time Do Developers Spend Outside? 2018



Developer Survey - Developer Profile (cont'd)

Healthy Habits 2018

How Often Do Developers Skip Meals To Be Productive?

How Often Do Developers Exercise?

I don't typically exercise 37.4%

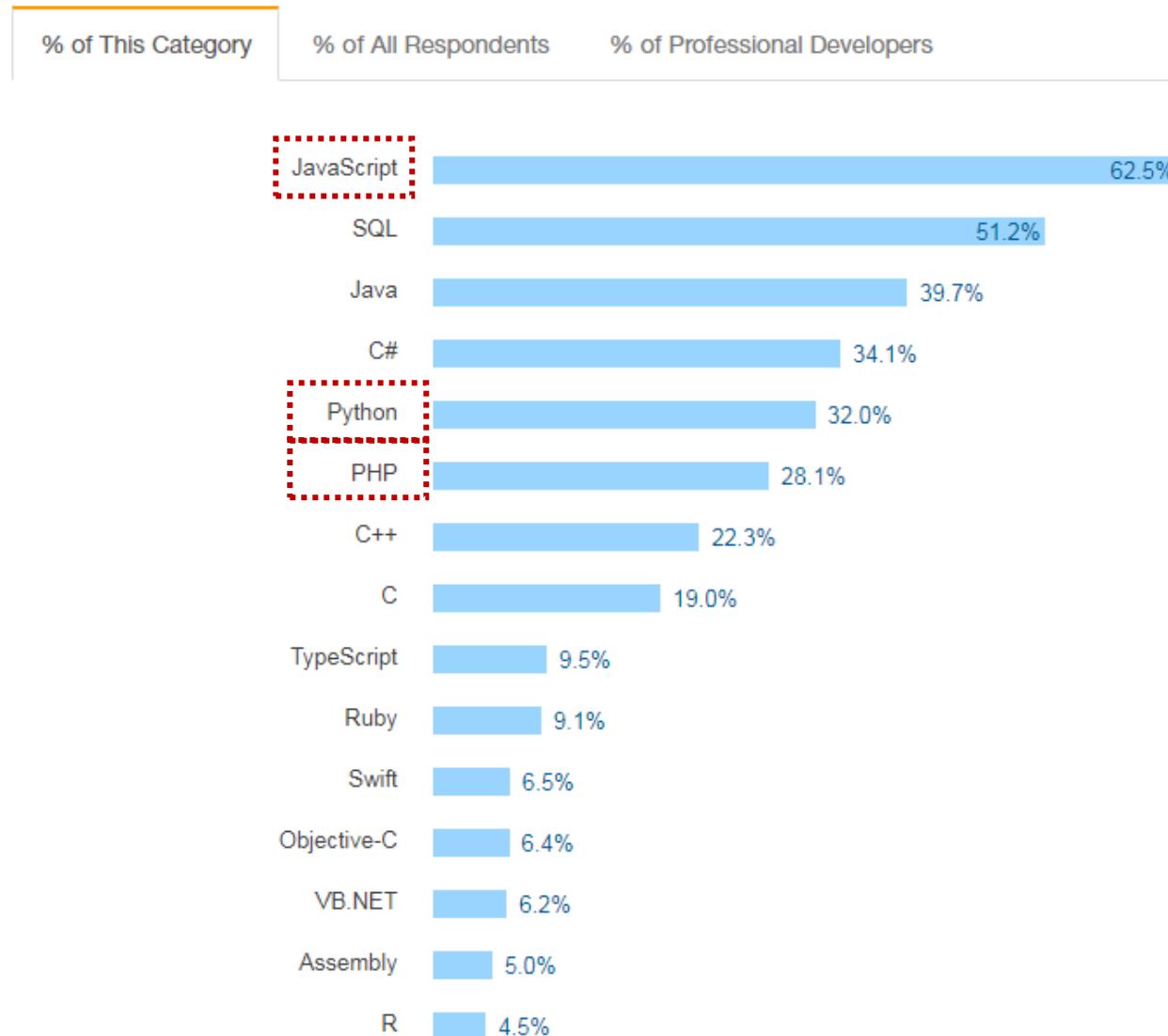
1 - 2 times per week 29.0%

3 - 4 times per week 19.9%

Daily or almost every day 13.7%

Developer Survey - Technologies

Programming Languages 2017

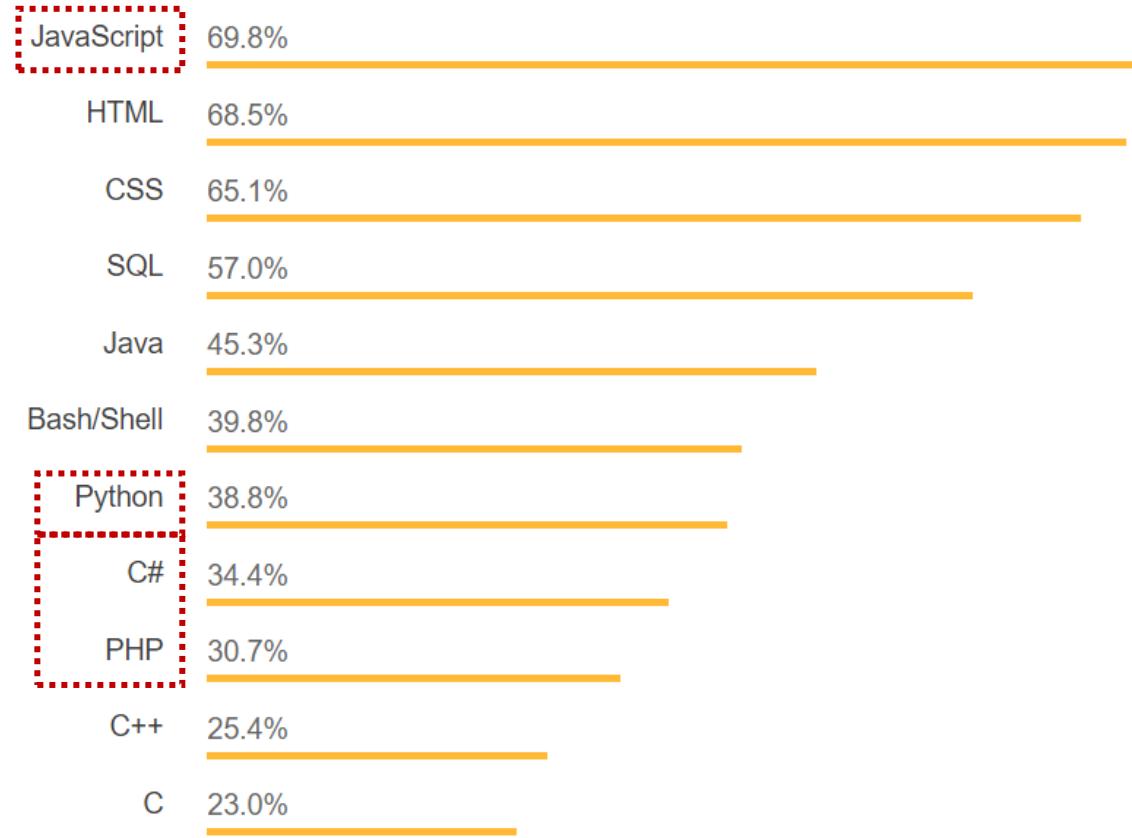


Developer Survey - Technologies (cont'd)

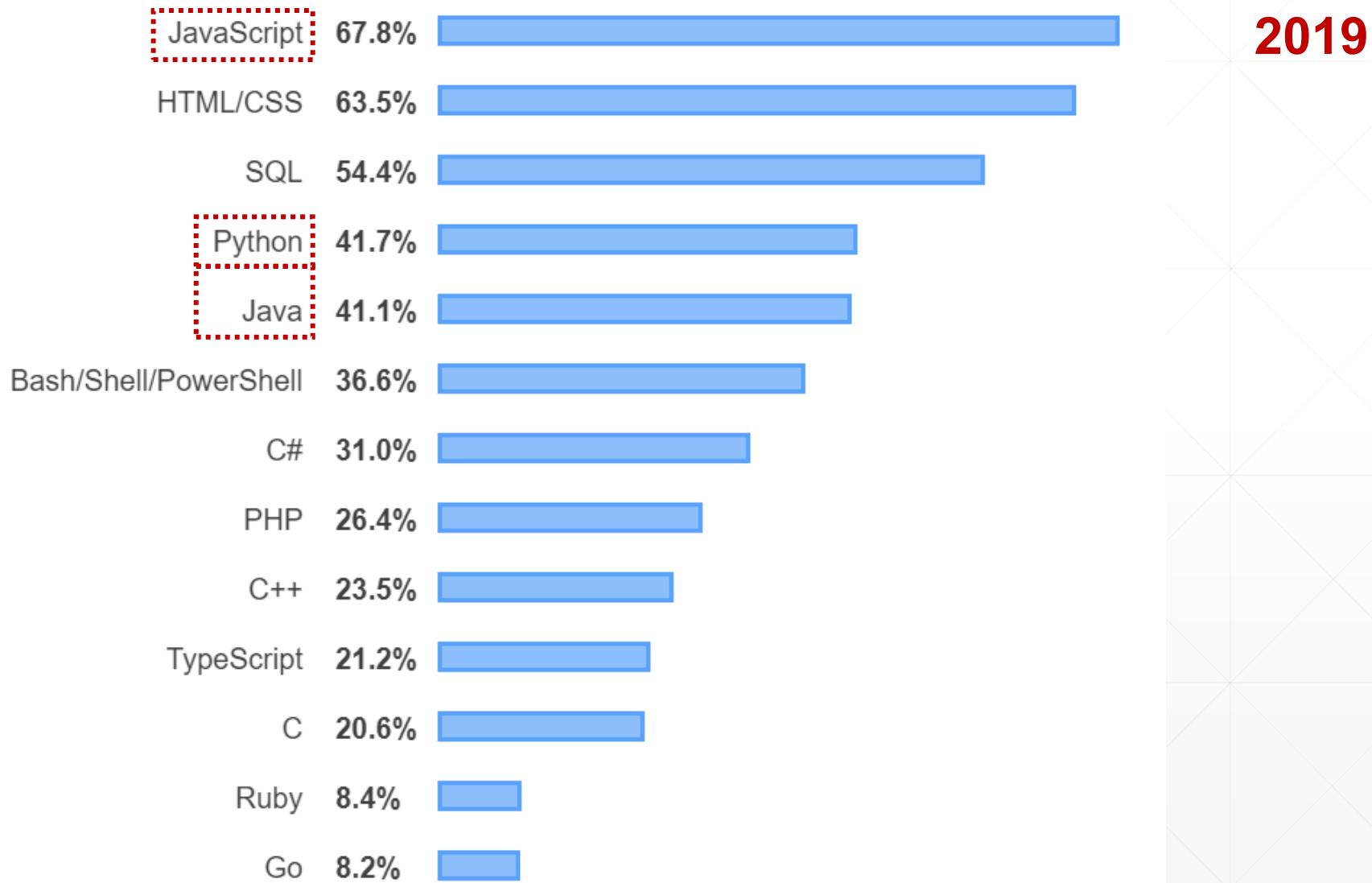
Programming, Scripting, and Markup Languages 2018

All Respondents

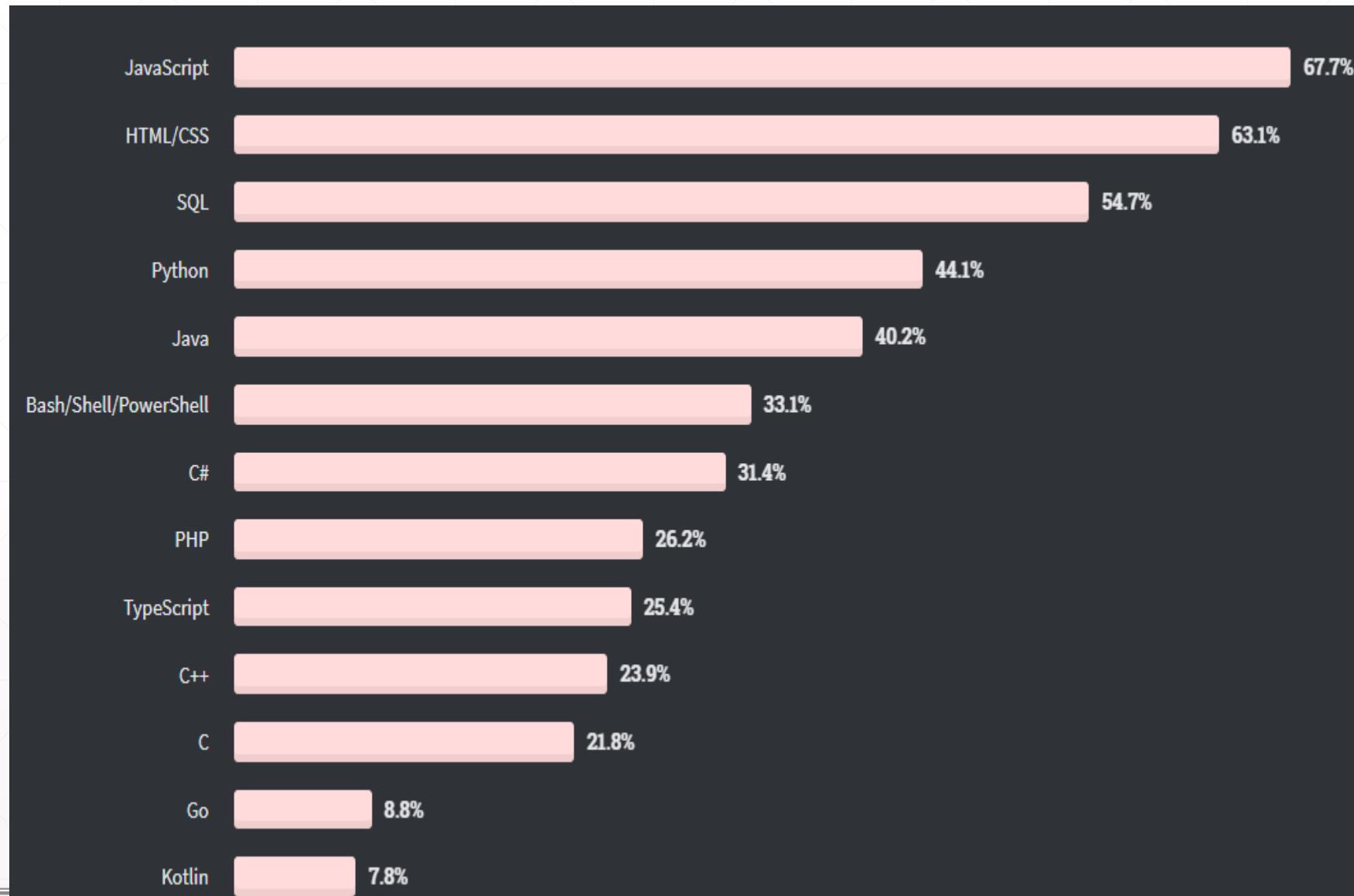
Professional Developers



Developer Survey - Technologies (cont'd)



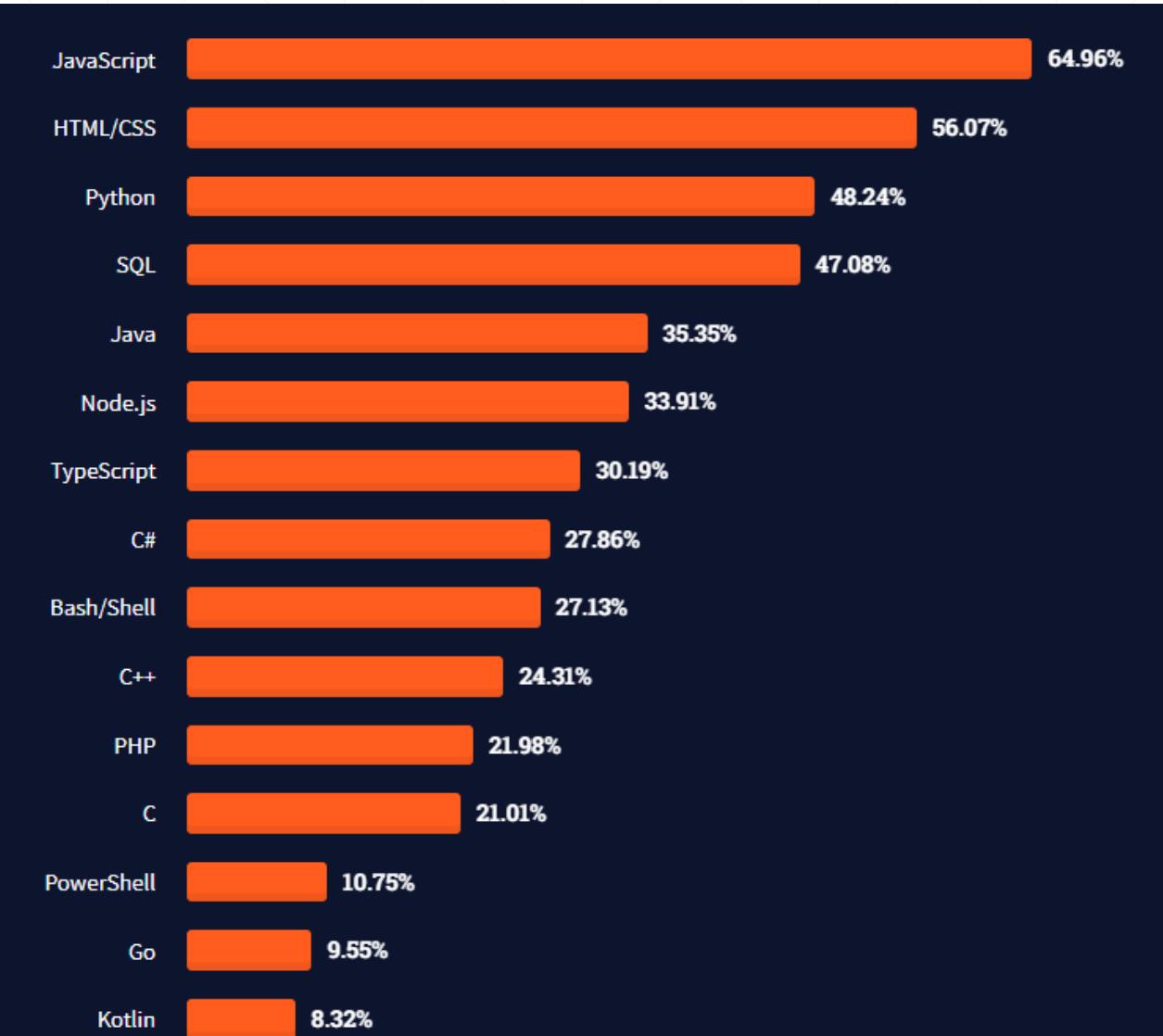
Developer Survey - Technologies (cont'd)



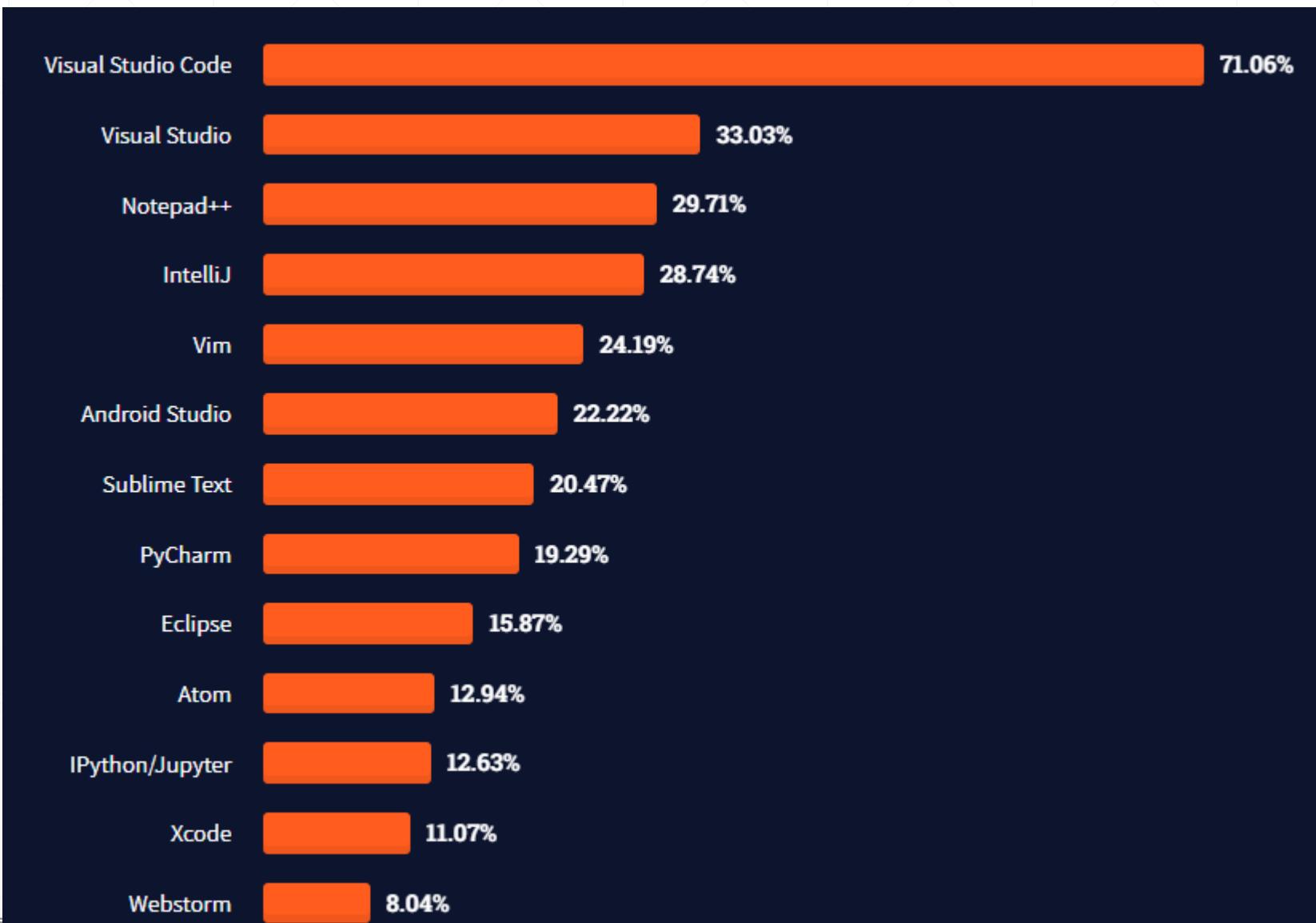
2020

Developer Survey - Technologies (cont'd)

2021

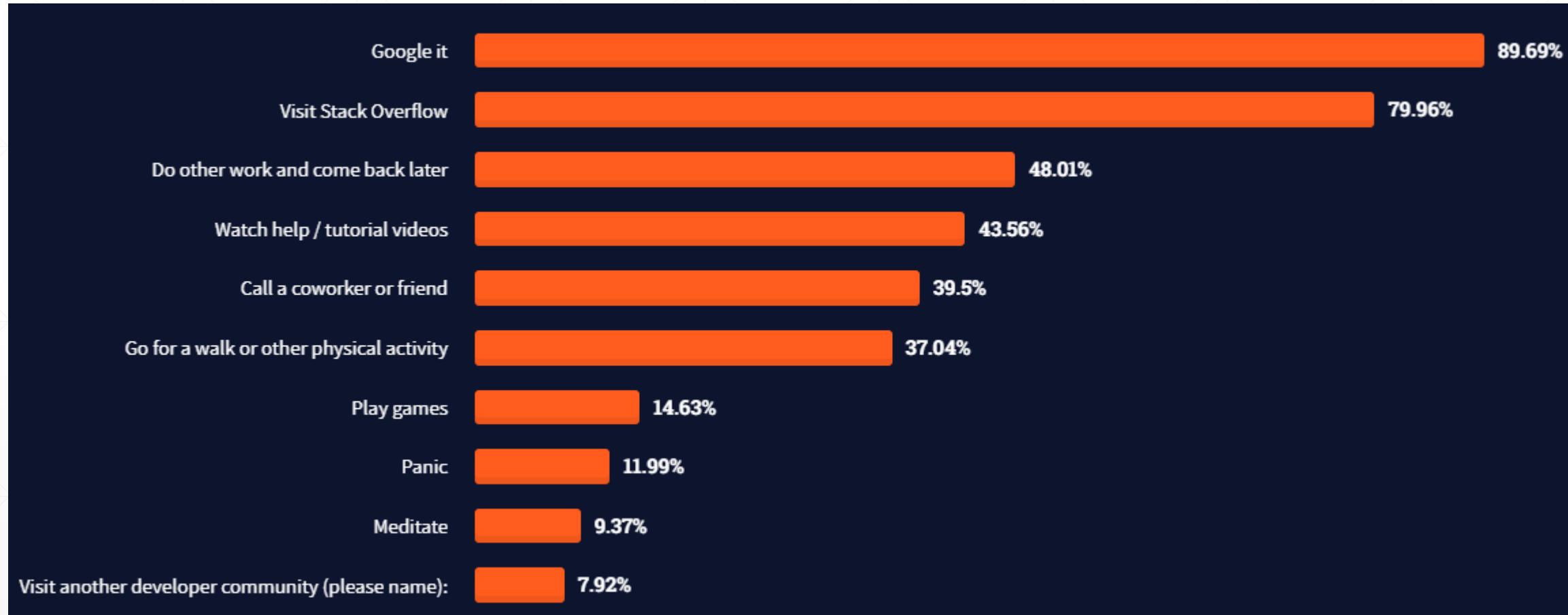


Developer Survey - IDE

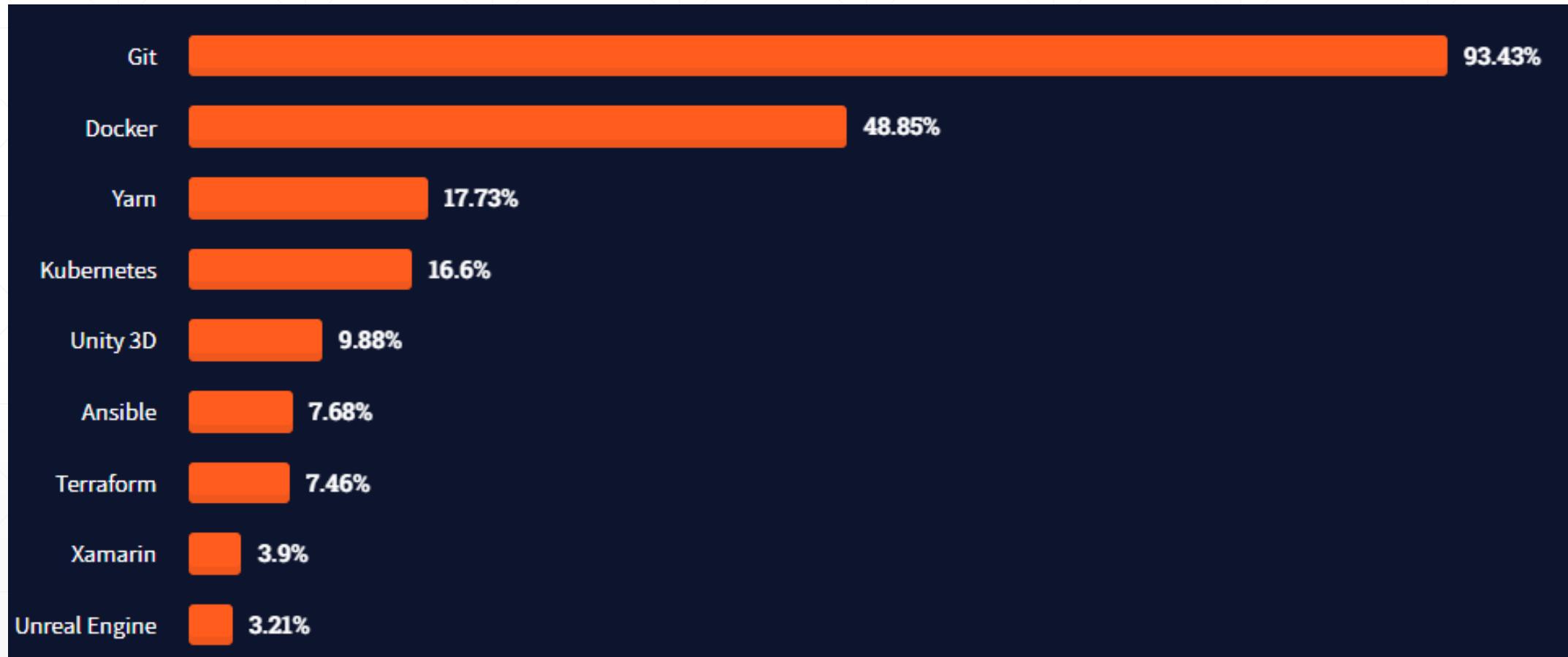


Developer Survey – Learning & Problem Solving

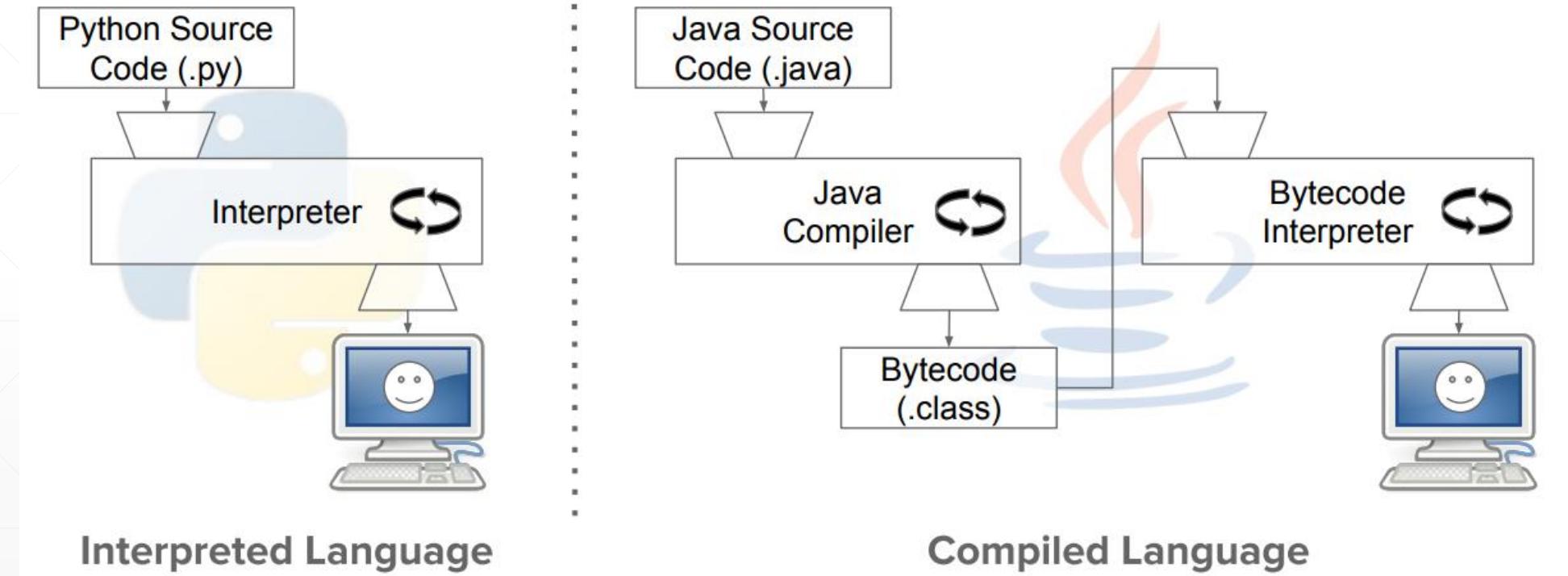
■ What do you do when you get stuck



Developer Survey – Tools



Java?



Resource from PyCon 2017

Java?

```
print("Hello, World! ")
```



```
D:\W> python hello.py
Hello, World!!
```

Python

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



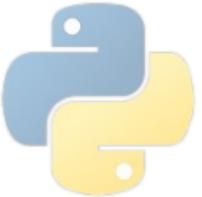
```
D:\W>javac Hello.java
```



```
D:\>java -classpath . Hello  
Hello, World!!
```

Java

Java?



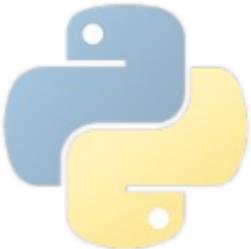
```
print("Hello, World!!")
```



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

Resource from PyCon 2017

Java?



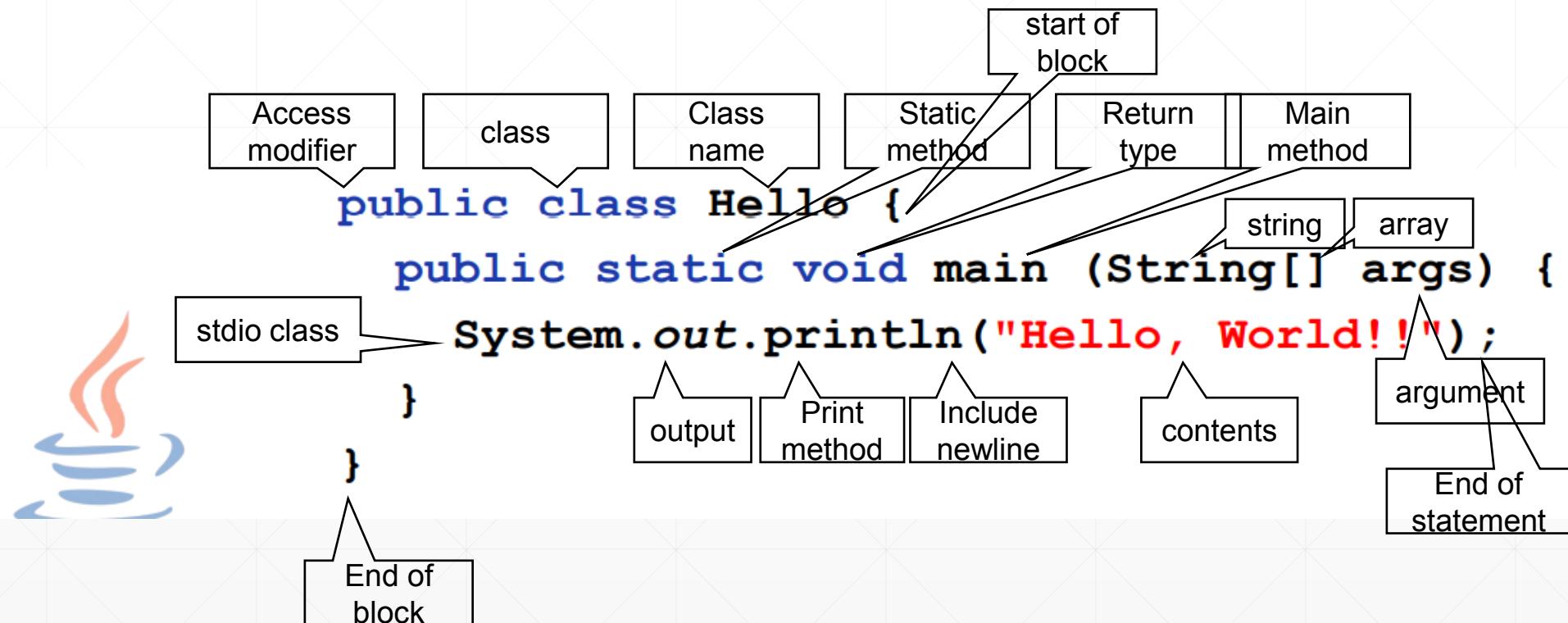
```
print("Hello, World!!")
```

function

contents

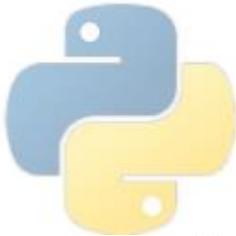
Resource from PyCon 2017

Java?



Resource from PyCon 2017

Java?



```
"-".join(str(n) for n in range(100))
```



```
private static void join (int num) {  
    for(int i = 0; i < num; i++) {  
        StringBuilder sb = new StringBuilder();  
        for(int j: IntStream.range(0, 100).toArray()) {  
            if(sb.length() == 0) sb.append(j);  
            else sb.append("-").append(j);  
        }  
    }  
}
```

Resource from PyCon 2017

Java?

■ Runtime efficiency



⇒ 0.323s

x 1.6

⇒ 0.198s (IntStream.range())



x 2.3

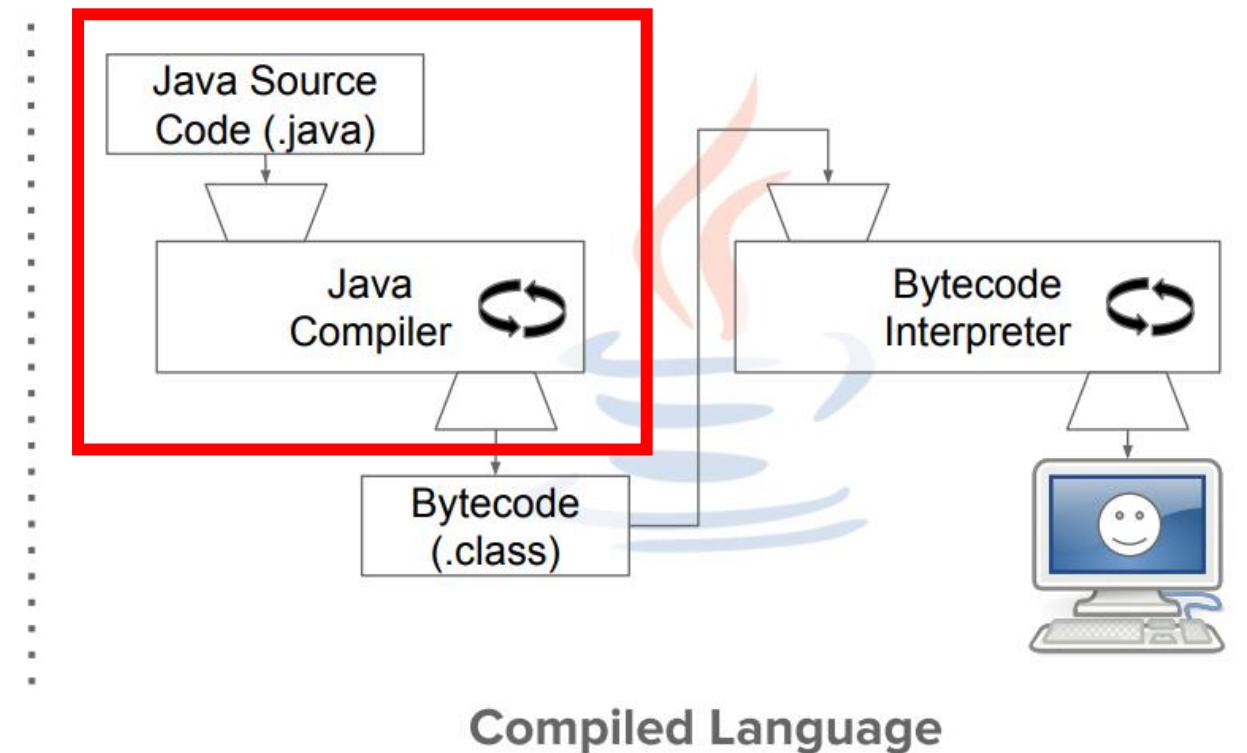
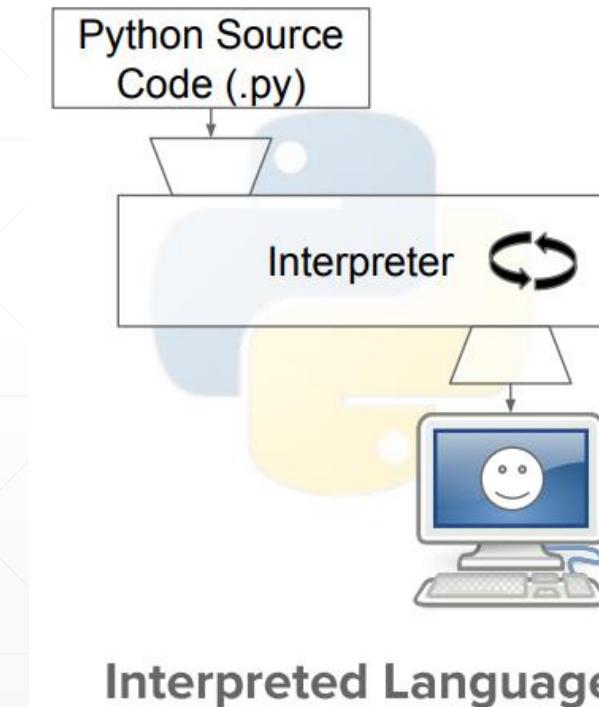
⇒ 0.086s (Traditional Loop)

x 3.8

Resource from PyCon 2017

Java?

■ Development time efficiency



Resource from PyCon 2017

Java?

Free-Style

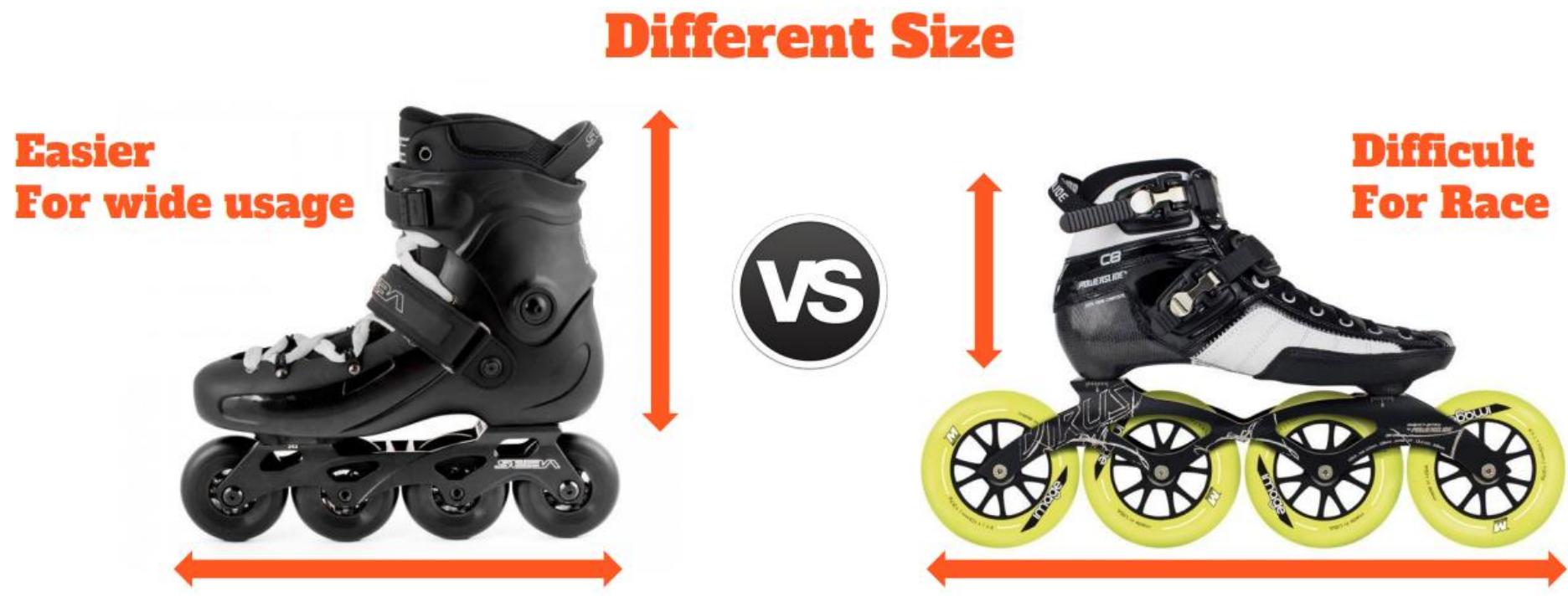


High-end



Resource from PyCon 2017

Java?



Resource from PyCon 2017

Java?

Build FAST



vs



Run FAST

Resource from PyCon 2017

Q&A

■ Next... in this week

- JDK/IDE Setup (eClass video)

■ Next week

- Introduction to Java
- Types

Computer Language



Setup

Agenda

- IDE installation
- Hello World!

IntelliJ Installation

■ Download IntelliJ

➤ <https://www.jetbrains.com/ko-kr/idea/download>

The screenshot shows the IntelliJ IDEA download page on the JetBrains website. At the top, there's a navigation bar with links for '개발자 도구' (Developer Tools), '팀 도구' (Team Tools), '학습 도구' (Learning Tools), '솔루션' (Solutions), '스토어' (Store), a search icon, and user account icons. Below the navigation bar, the 'IntelliJ IDEA' logo is displayed, along with a message 'Coming in 2021.1' and links for '새로운 기능' (New Features), '기능' (Features), '리소스' (Resources), and '구매' (Purchase). A prominent blue '다운로드' (Download) button is located on the right.

다운로드 IntelliJ IDEA

버전: 2020.3.2
빌드: 203.7148.57
2021년 1월 26일
[릴리스 노트](#)

시스템 요구 사항
[설치 안내](#)
[기타 버전](#)

Windows Mac Linux

Ultimate
웹 및 엔터프라이즈 개발용
[다운로드](#) .exe

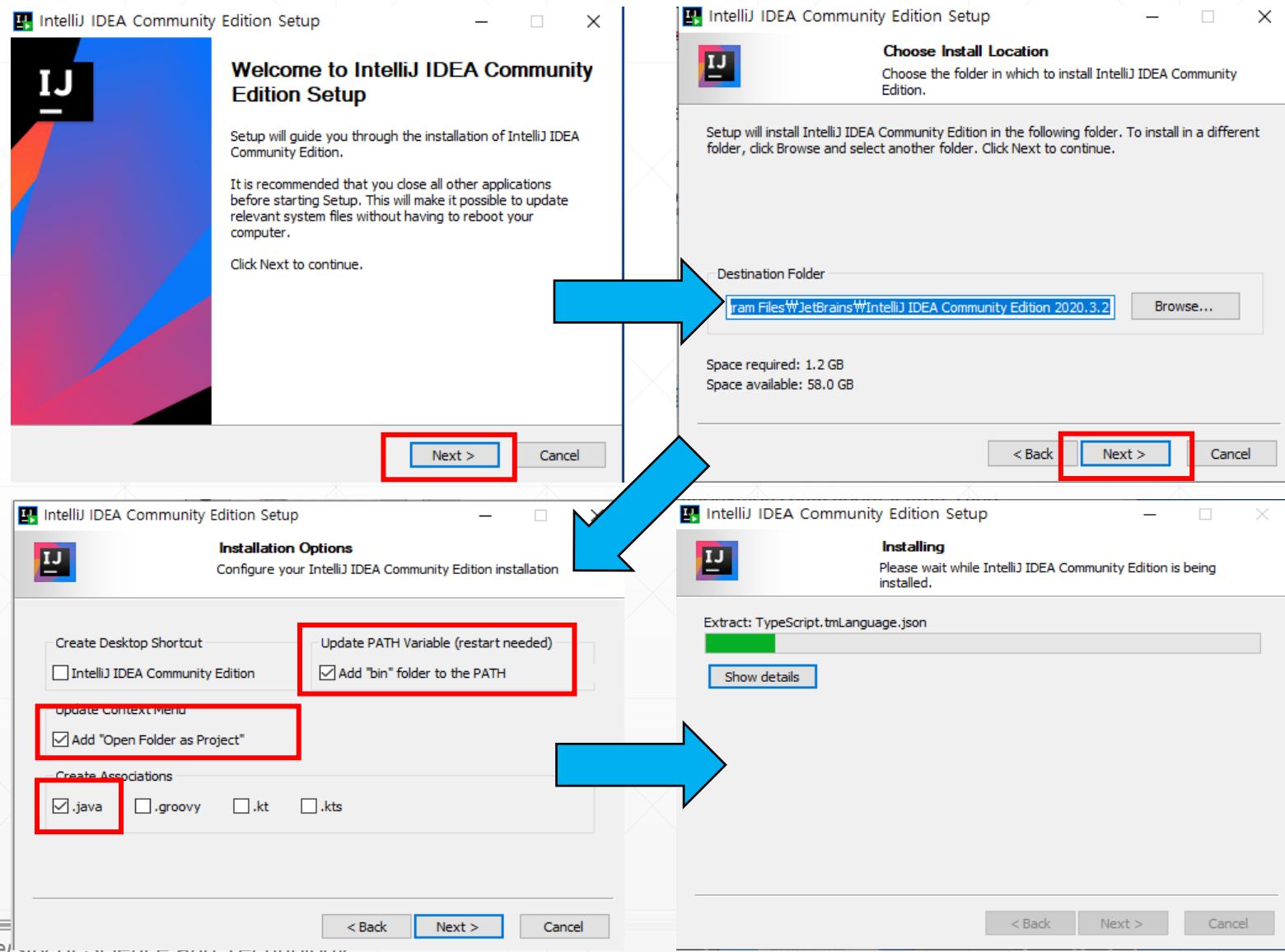
30일 무료 평가판

Community
JVM 및 Android 개발용
[다운로드](#) .exe
무료, 오픈 소스

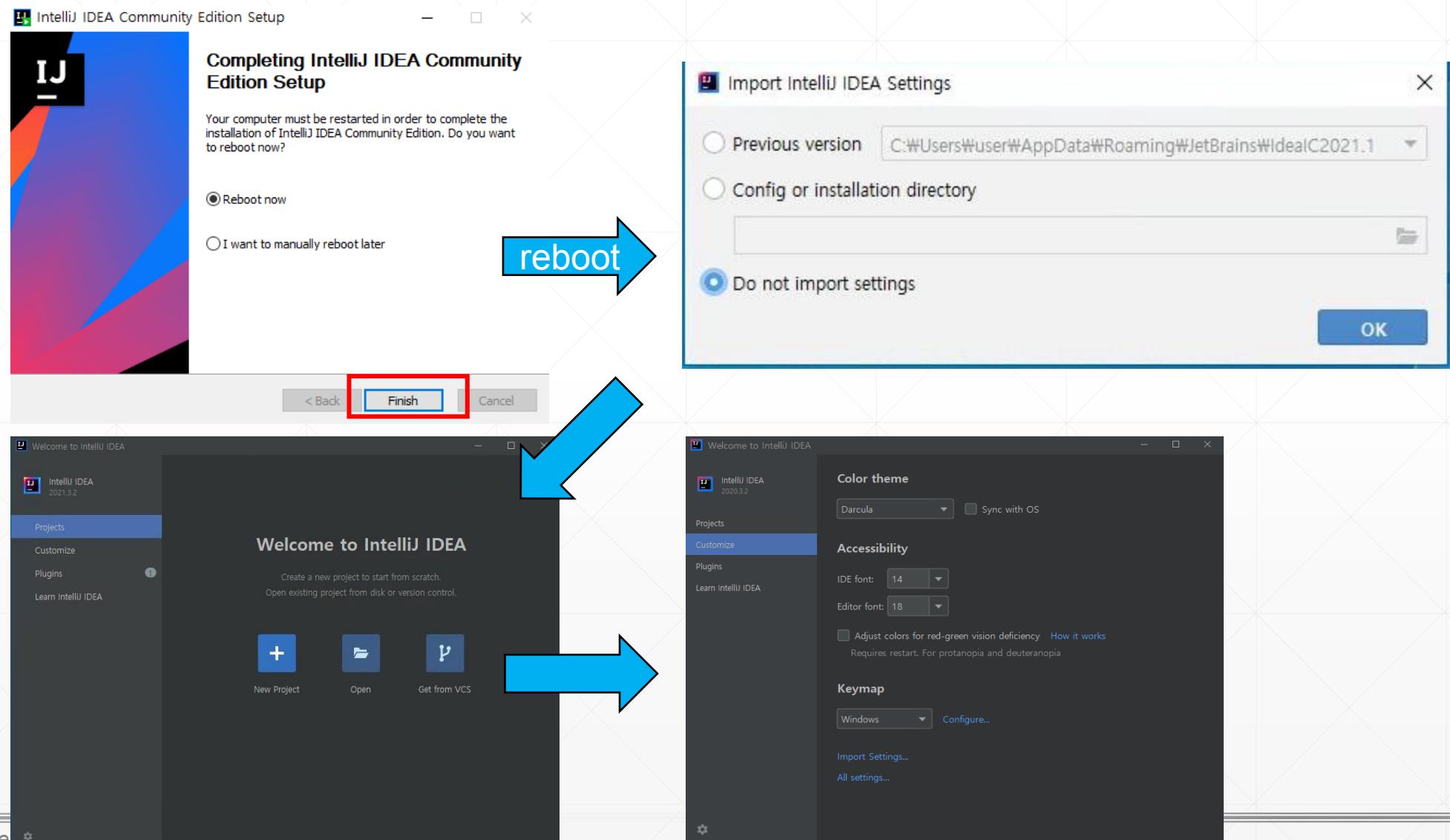
IntelliJ IDEA Ultimate IntelliJ IDEA Community 에디션 ⓘ

Java, Kotlin, Groovy, Scala	✓	✓
Android ⓘ	✓	✓

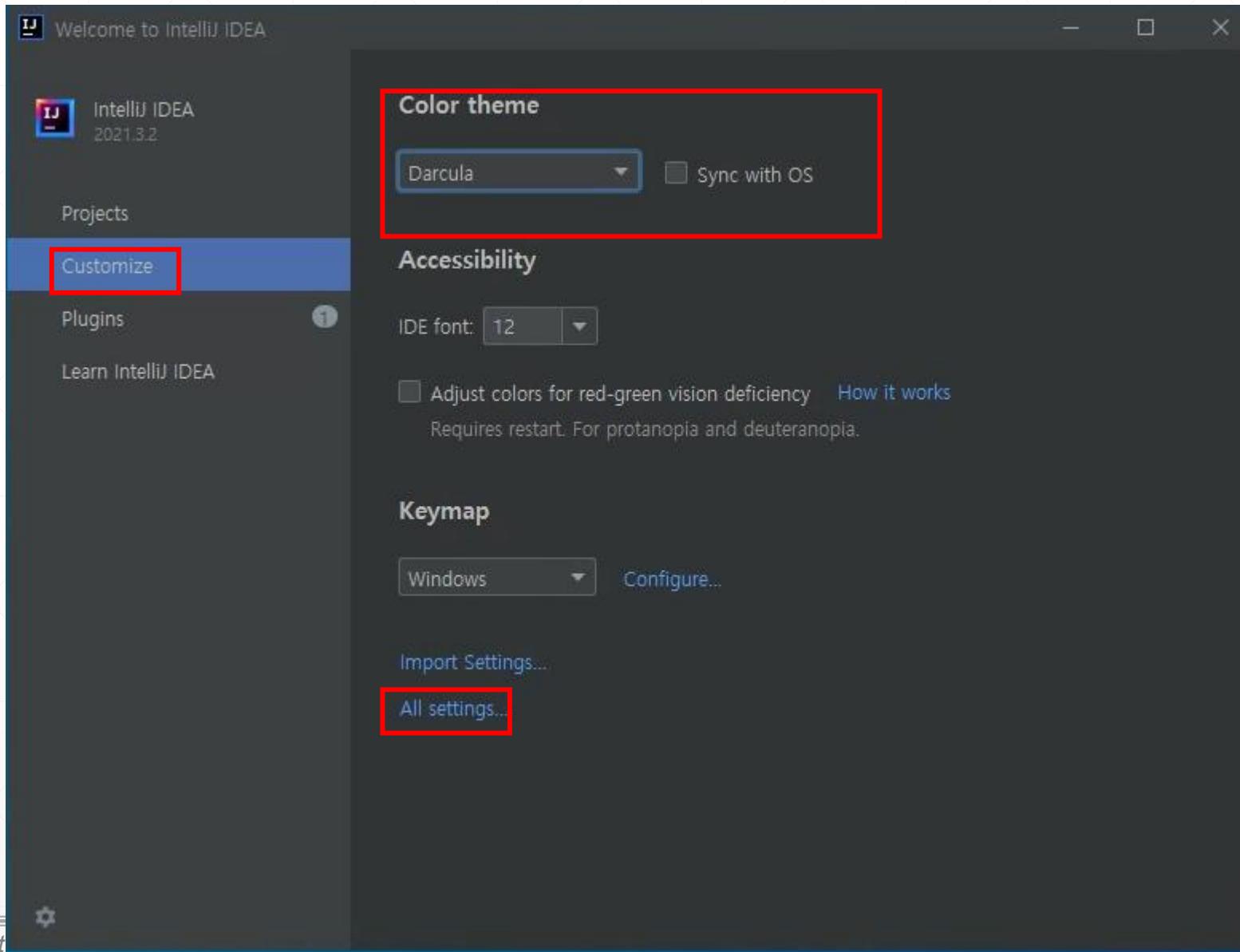
IntelliJ IDEA Installation (cont'd)



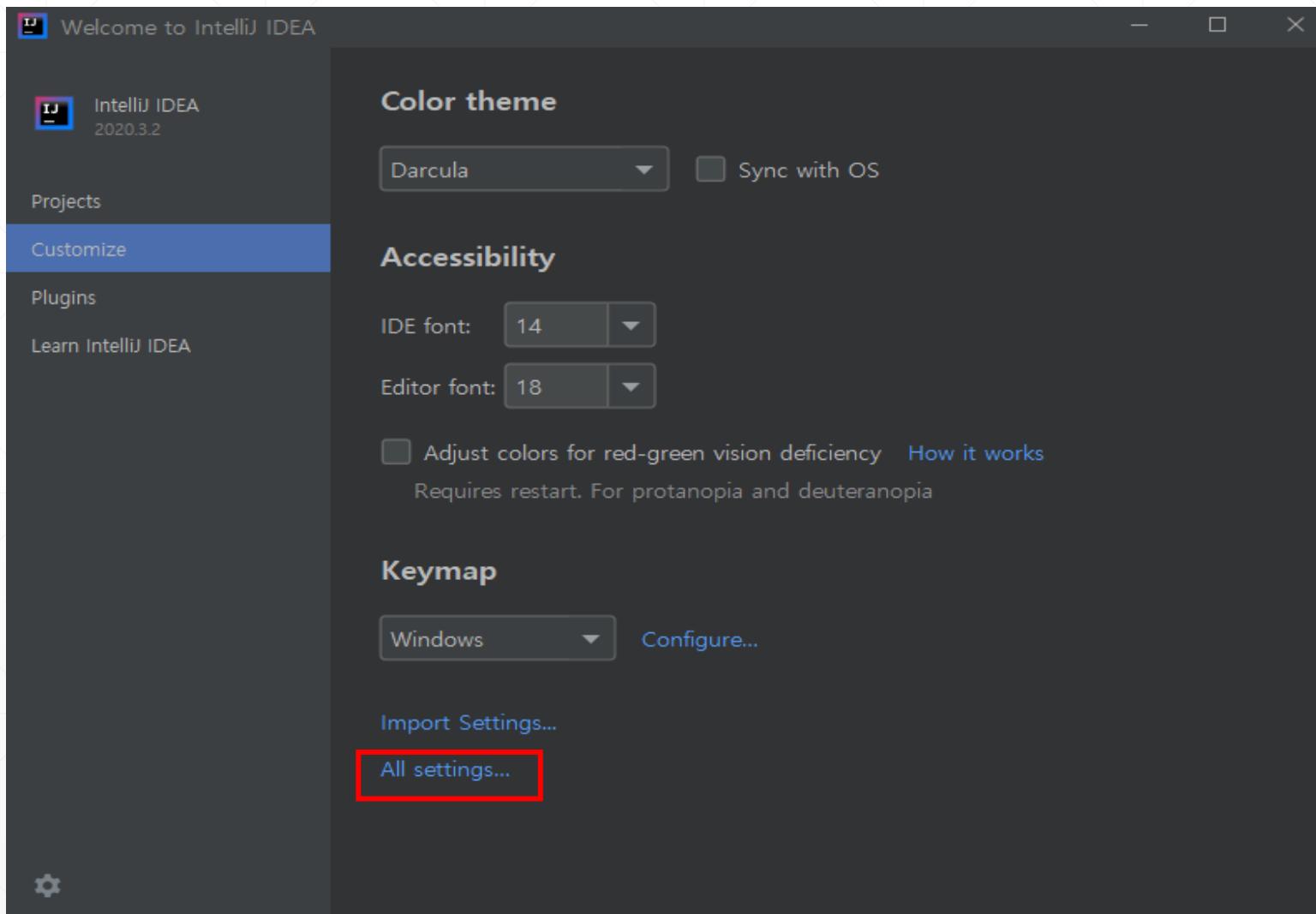
IntelliJ IDEA Installation (cont'd)



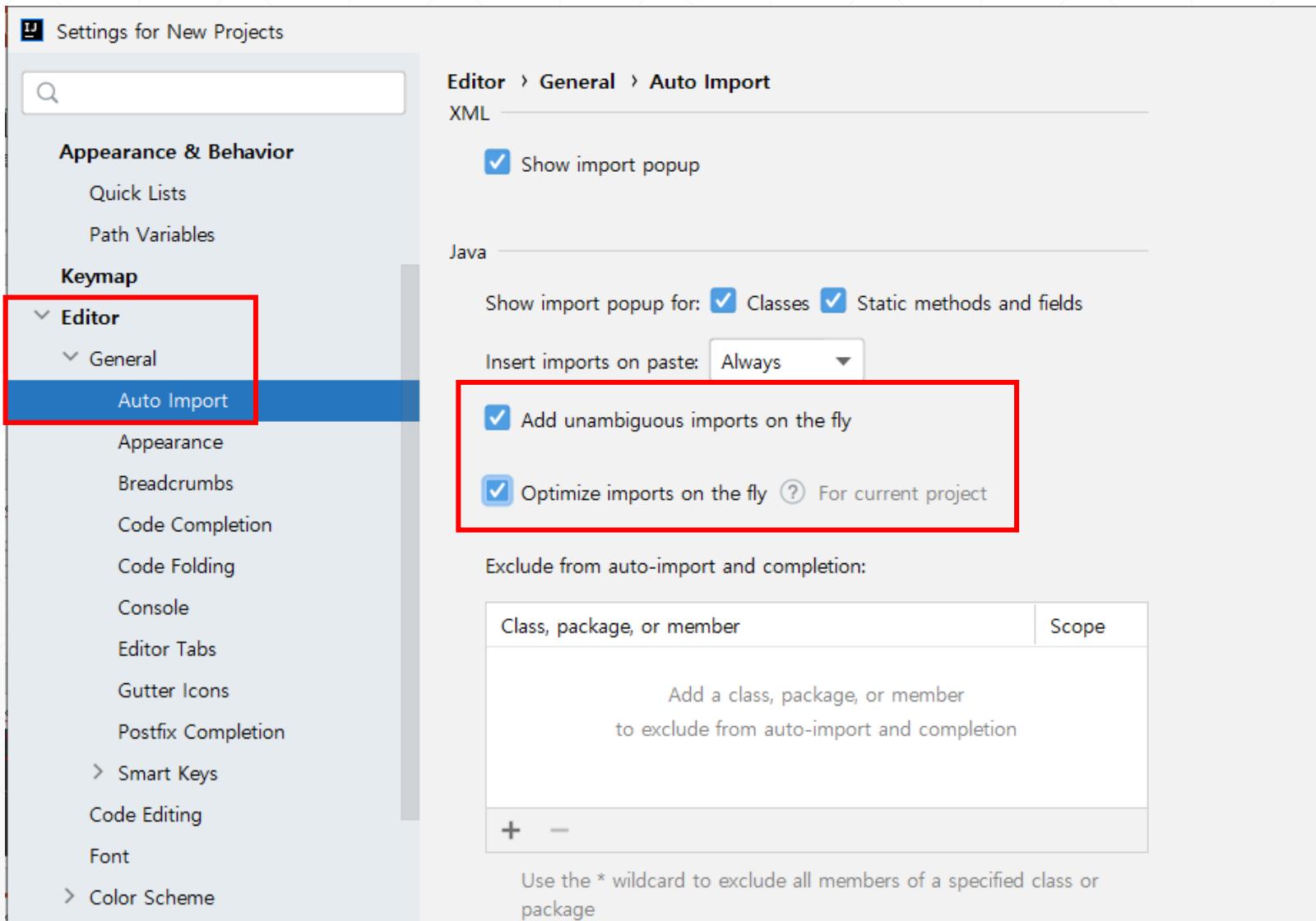
IntelliJ IDEA Configuration



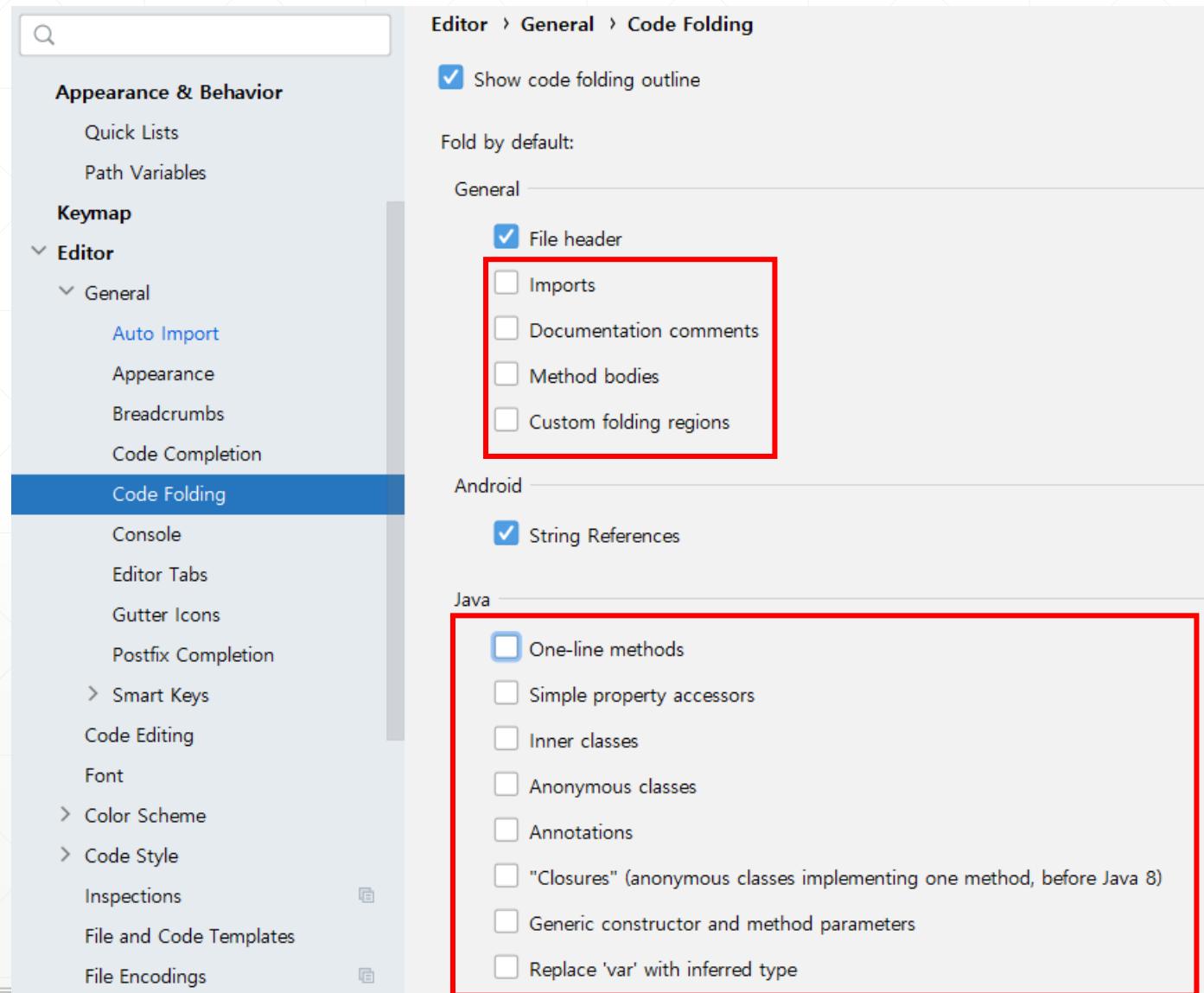
IntelliJ IDEA Configuration (cont'd)



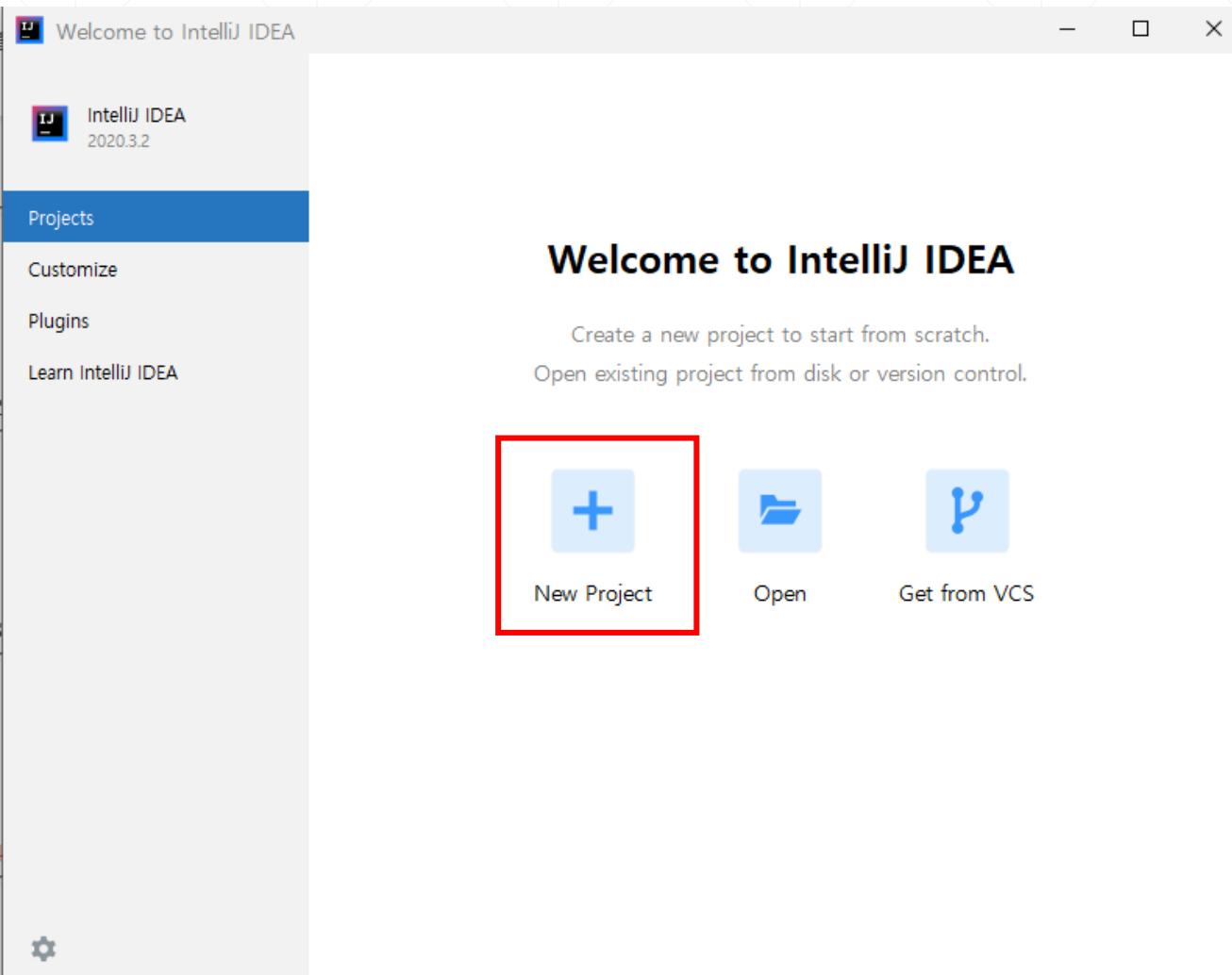
IntelliJ IDEA Configuration (cont'd)



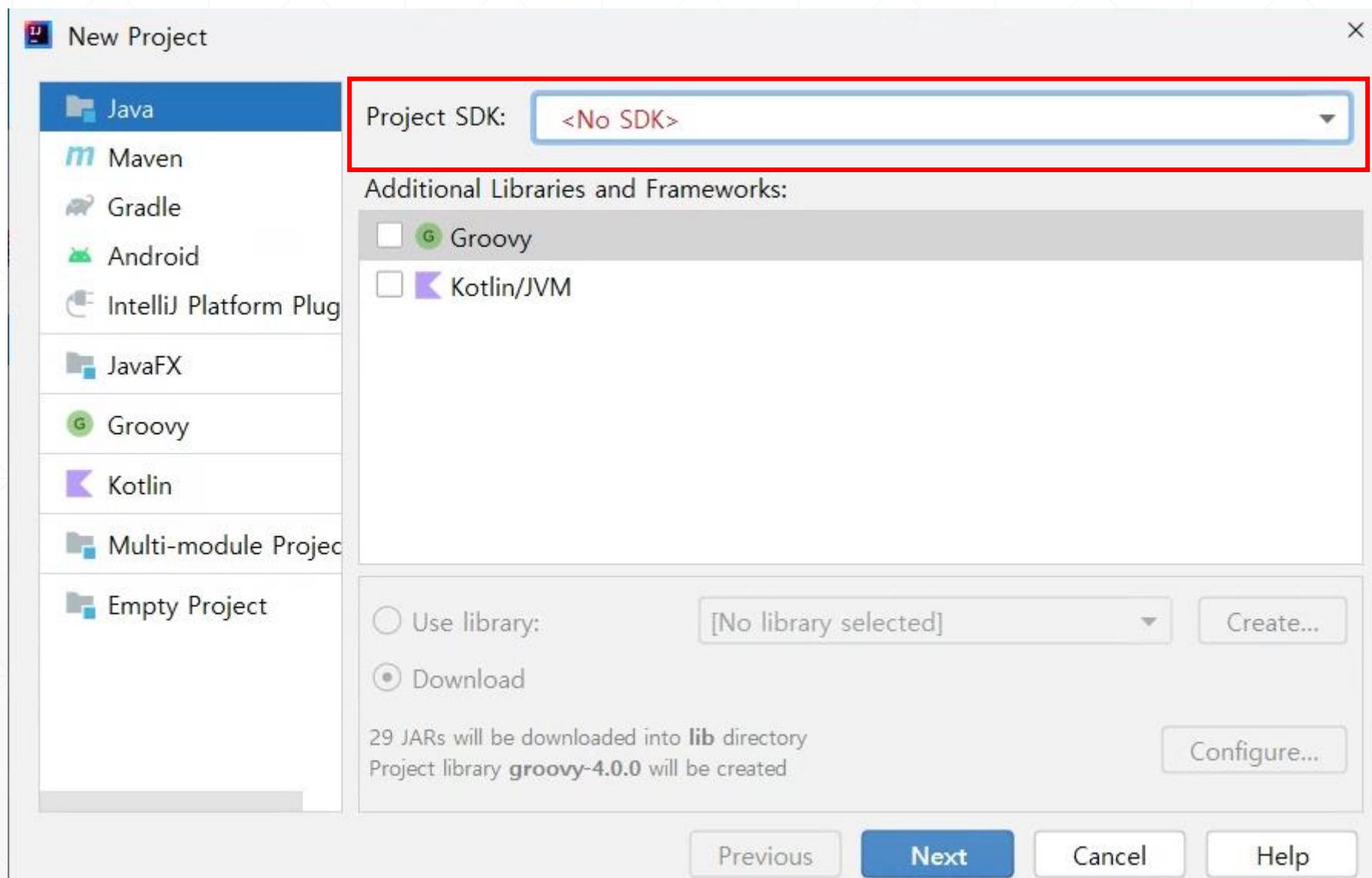
IntelliJ IDEA Configuration (cont'd)



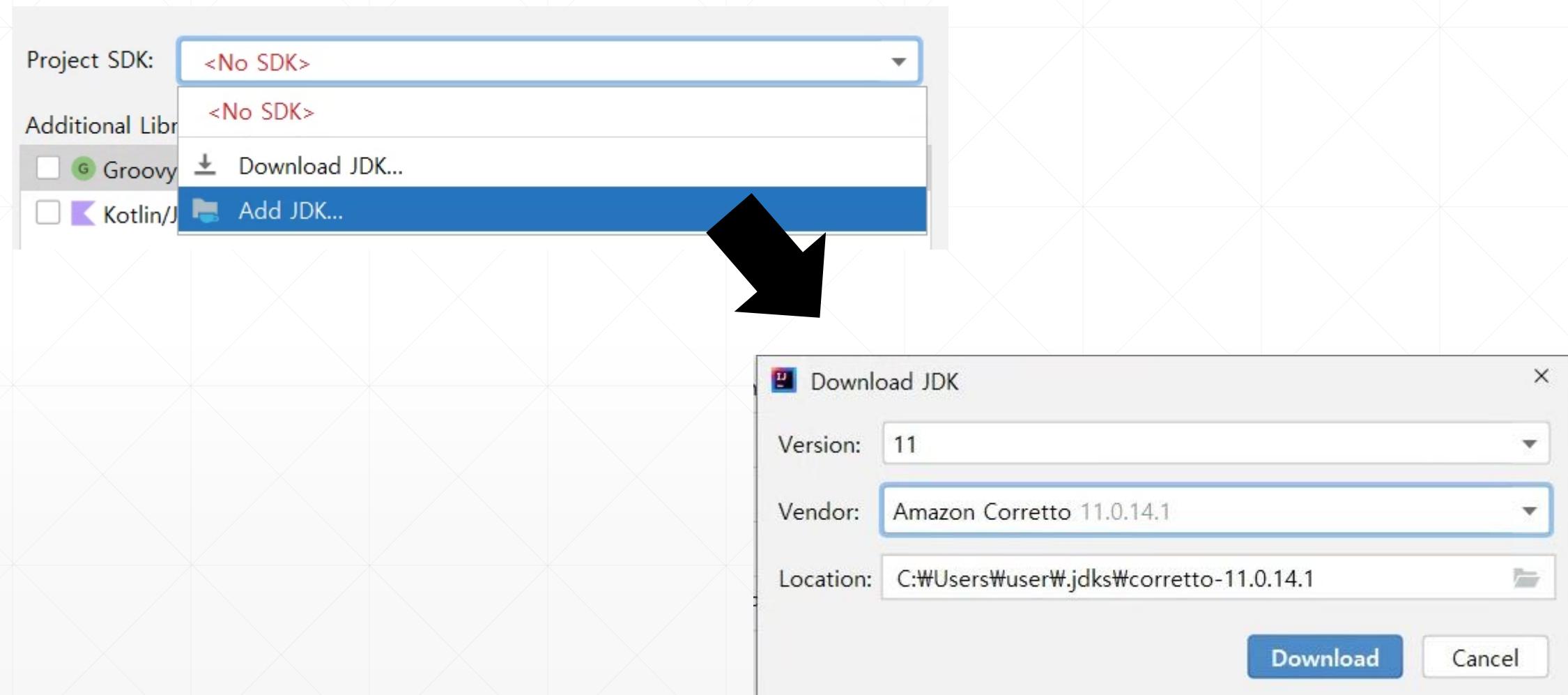
Hello, World!



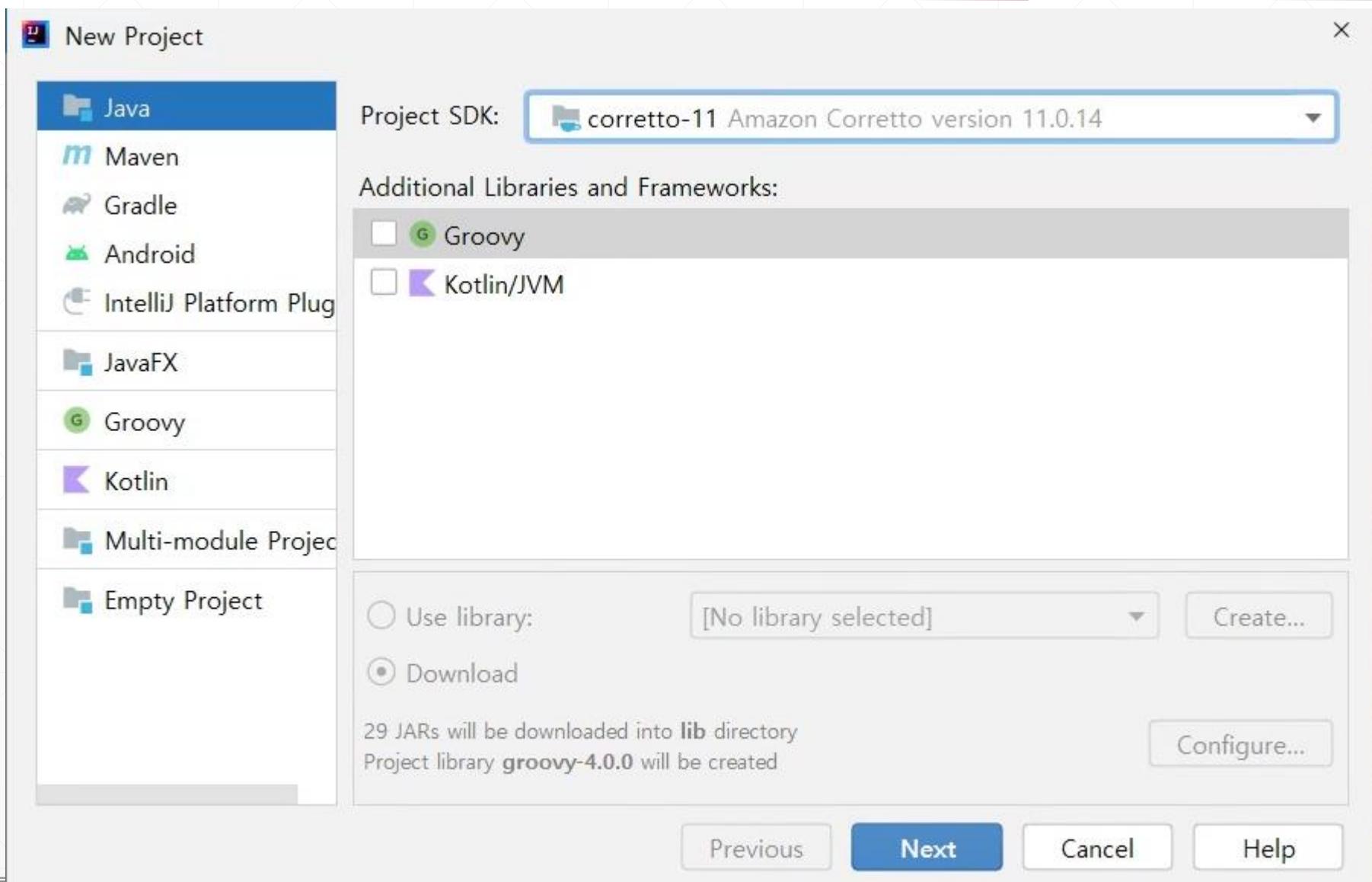
Hello, World! (cont'd)



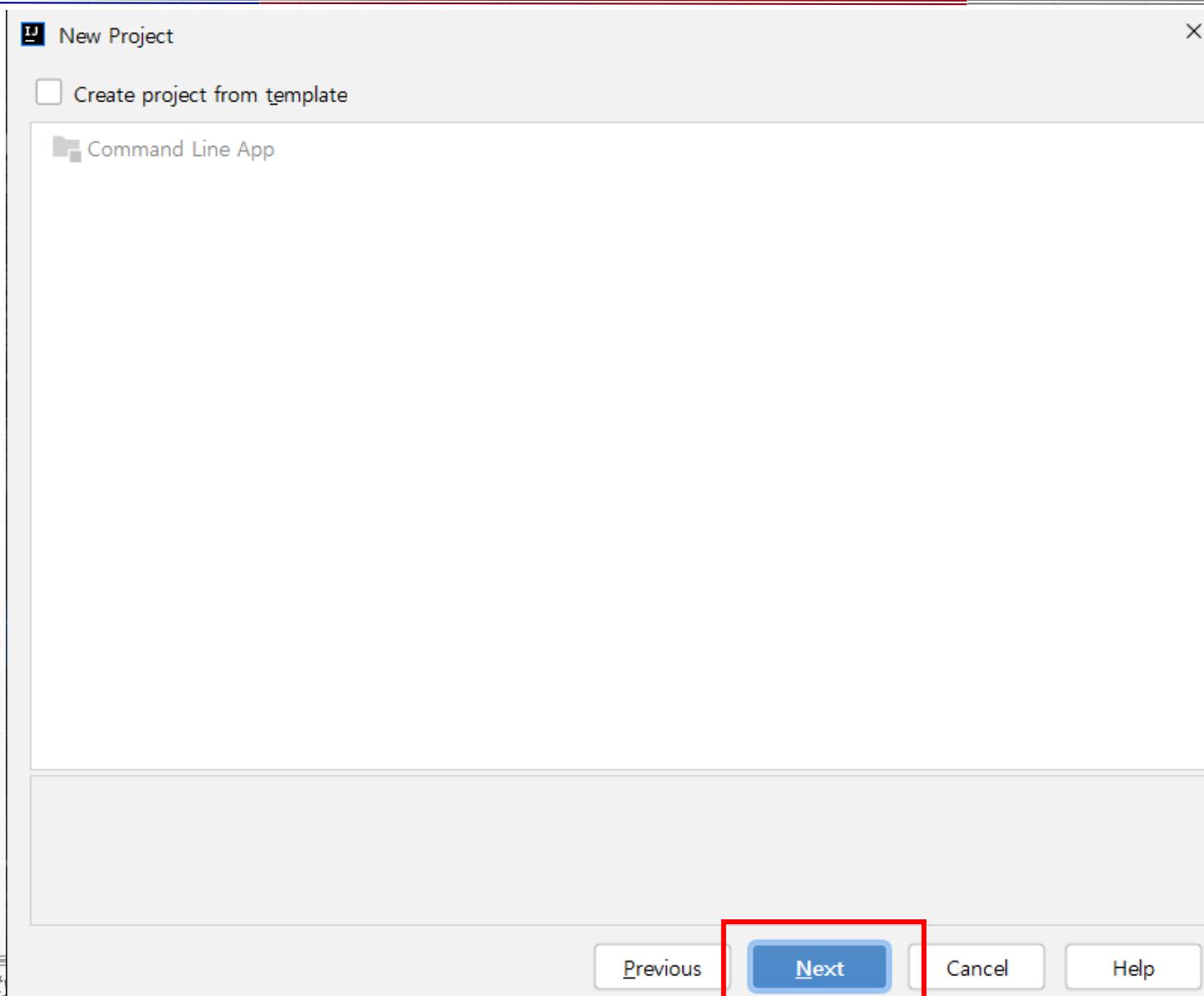
Hello, World! (cont'd)



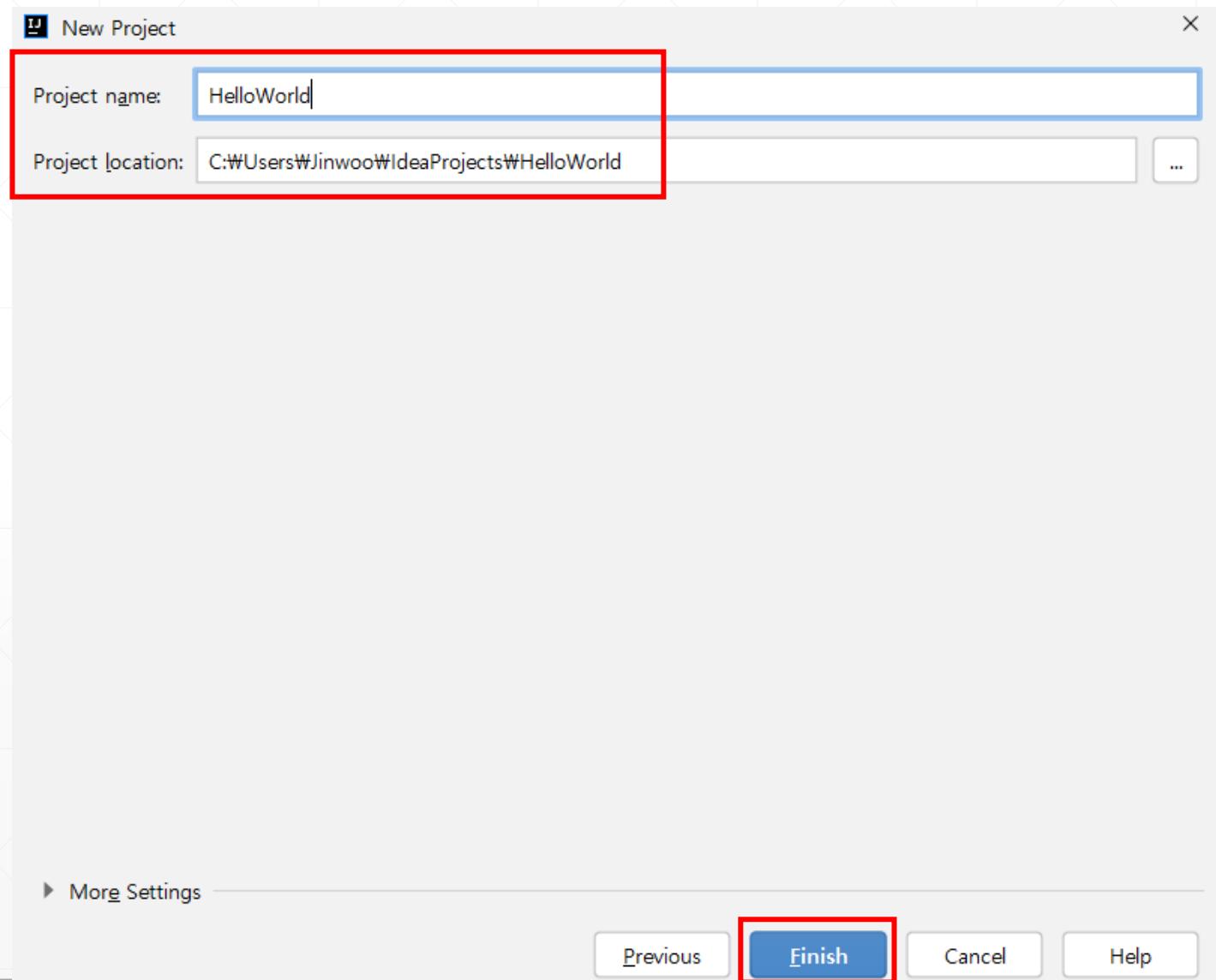
Hello, World! (cont'd)



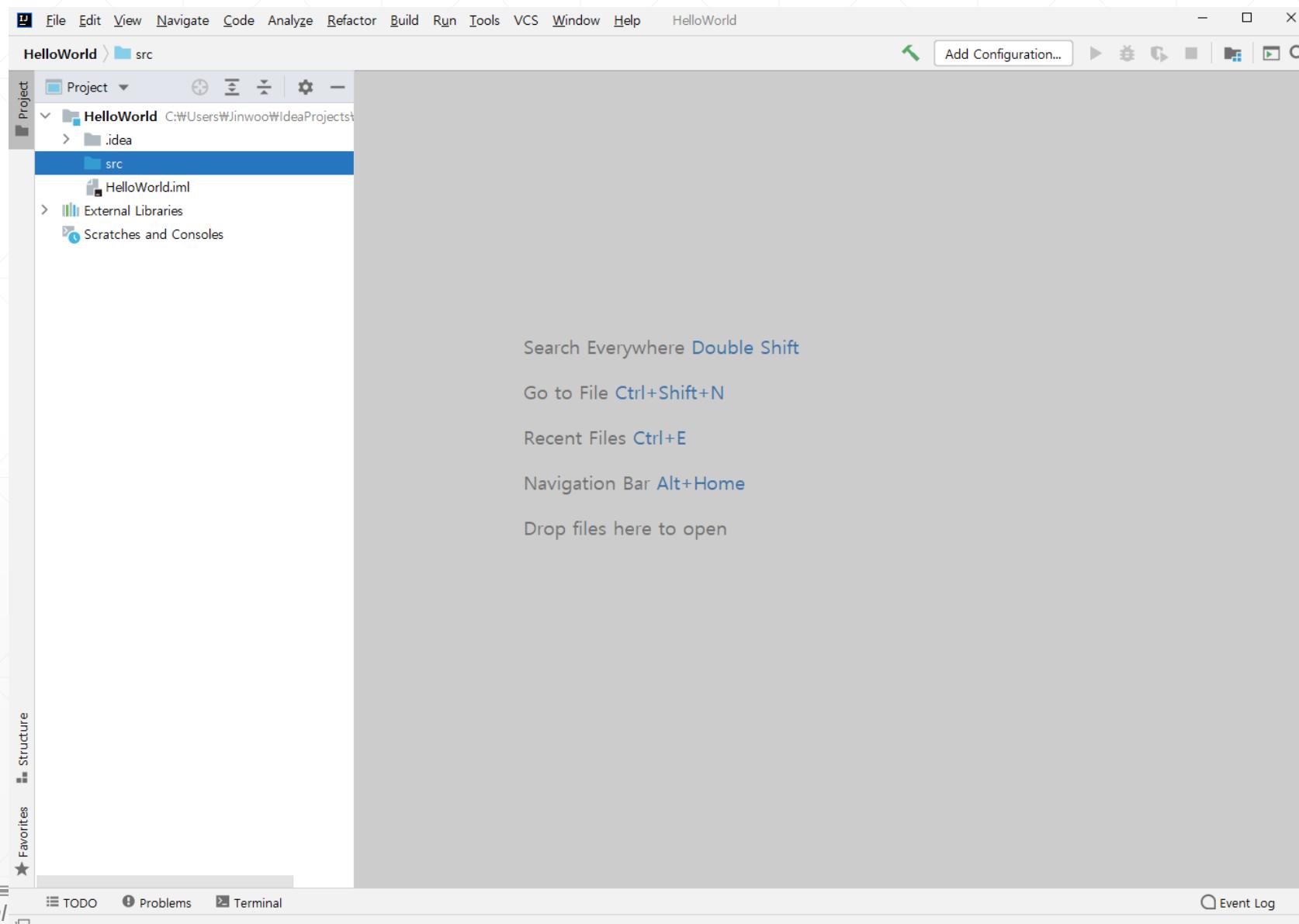
Hello, World! (cont'd)



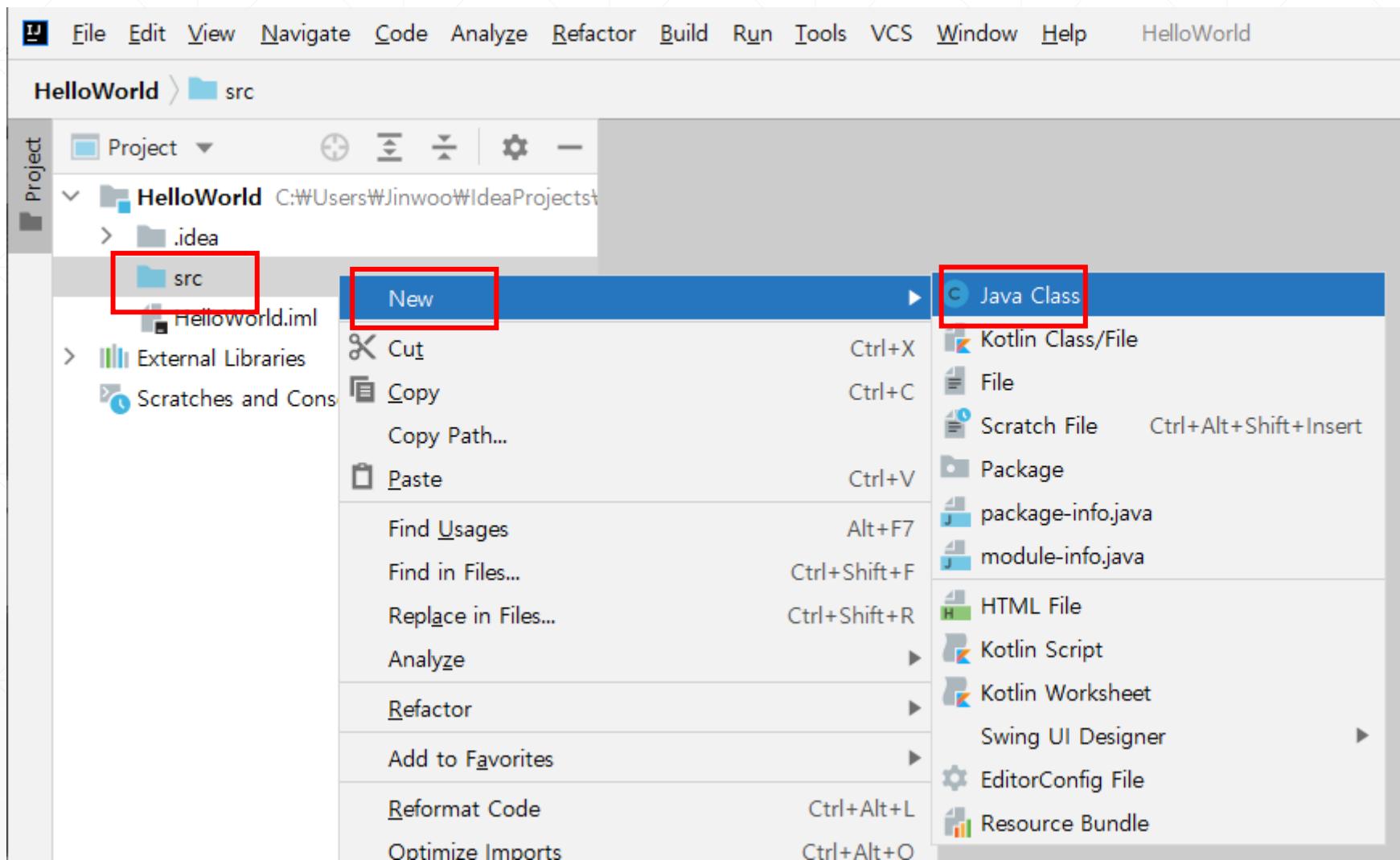
Hello, World! (cont'd)



Hello, World! (cont'd)

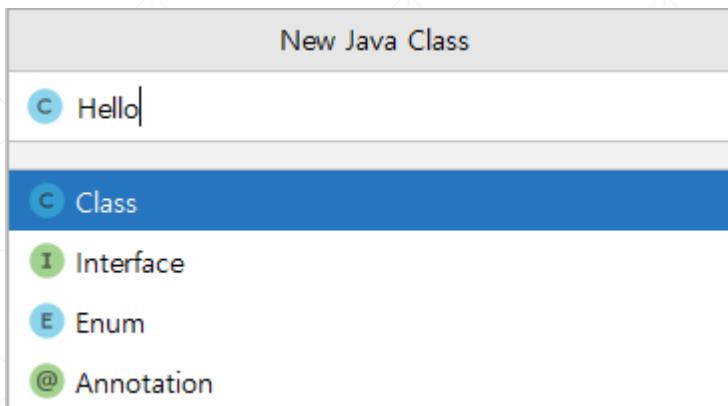


Hello, World! (cont'd)



Hello, World! (cont'd)

■ New class



■ Hello, World!

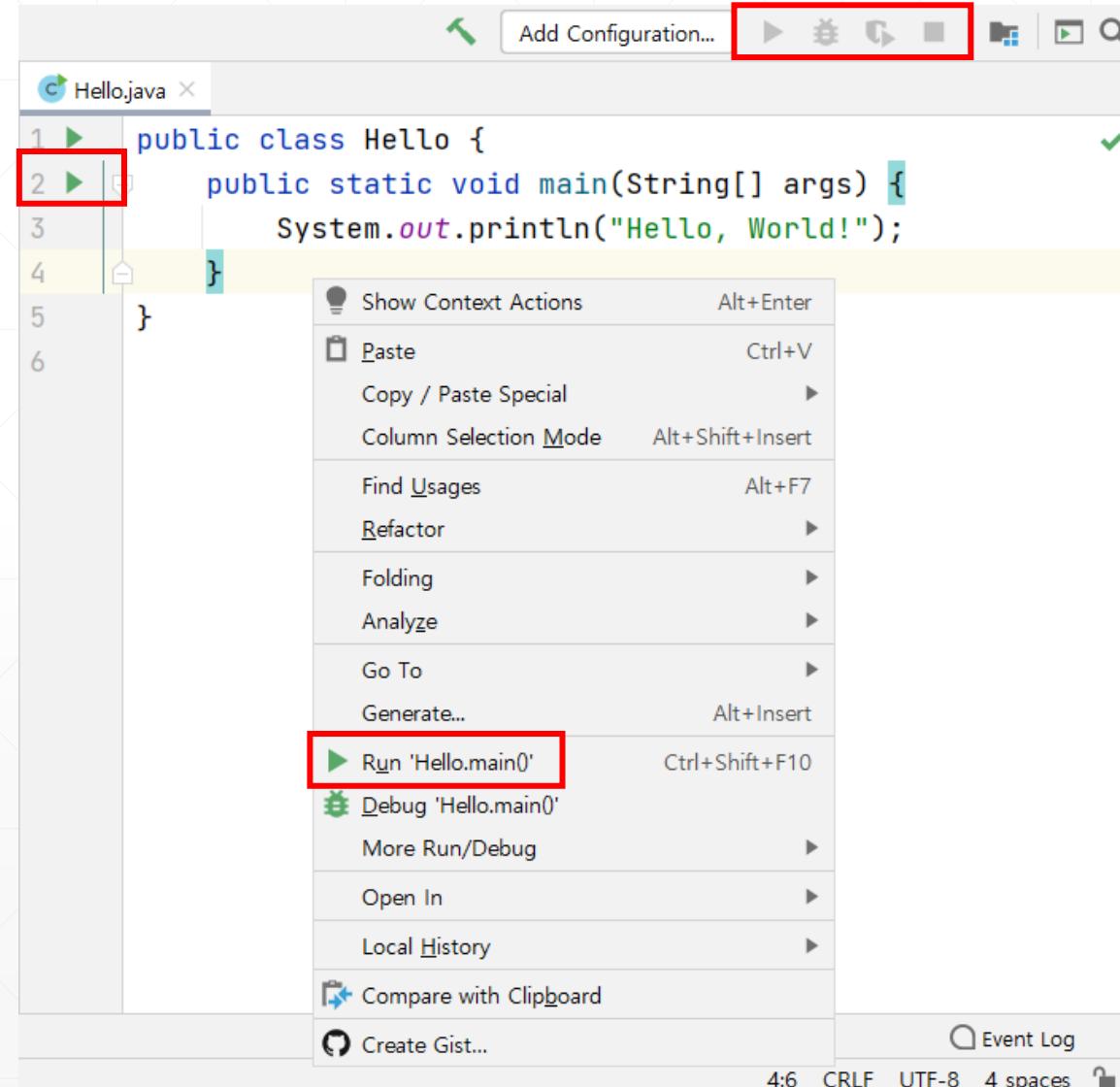
The screenshot shows an IDE interface with a window titled 'HelloWorld - Hello.java'. The menu bar includes File, Edit, View, Navigate, Code, Analyze, Refactor, Build, Run, Tools, VCS, Window, Help. The title bar shows 'HelloWorld > src > Hello'. The code editor displays the following Java code:

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

The code editor has syntax highlighting: 'Hello' is blue, 'public', 'class', 'main', 'String', 'System.out.println', and 'Hello, World!' are green, and the curly braces are black. The file path 'HelloWorld > src > Hello' is also highlighted in the project tree on the left.

Hello, World! (cont'd)

Run the program



Hello, World! (cont'd)

■ Good Job ☺

The screenshot shows a software interface, likely an IDE, with a central window titled "Run: Hello". The window displays the command used to run the program: "C:\Program Files\Java\jdk-11.0.10\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2019.2.3\lib\javaprofiler.jar" "Hello, World!". Below this, the output of the program is shown: "Hello, World!" followed by "Process finished with exit code 0". On the left side of the window, there is a vertical toolbar with icons for Run, Stop, Step Over, Step Into, Step Out, and others. At the bottom of the window, there are tabs for Run, TODO, Problems, Terminal, Build, and Event Log. The Run tab is currently selected. The status bar at the bottom of the screen shows the message "Build completed successfully in 4 sec, 594 ms (moments ago)" and "Checking for JDK updates". It also displays settings for line endings (6:1 CRLF), character encoding (UTF-8), and code style (4 spaces).

Q&A

■ Next week

- Introduction to Java
- Types

JDK Installation (Optional)

■ Download JDK

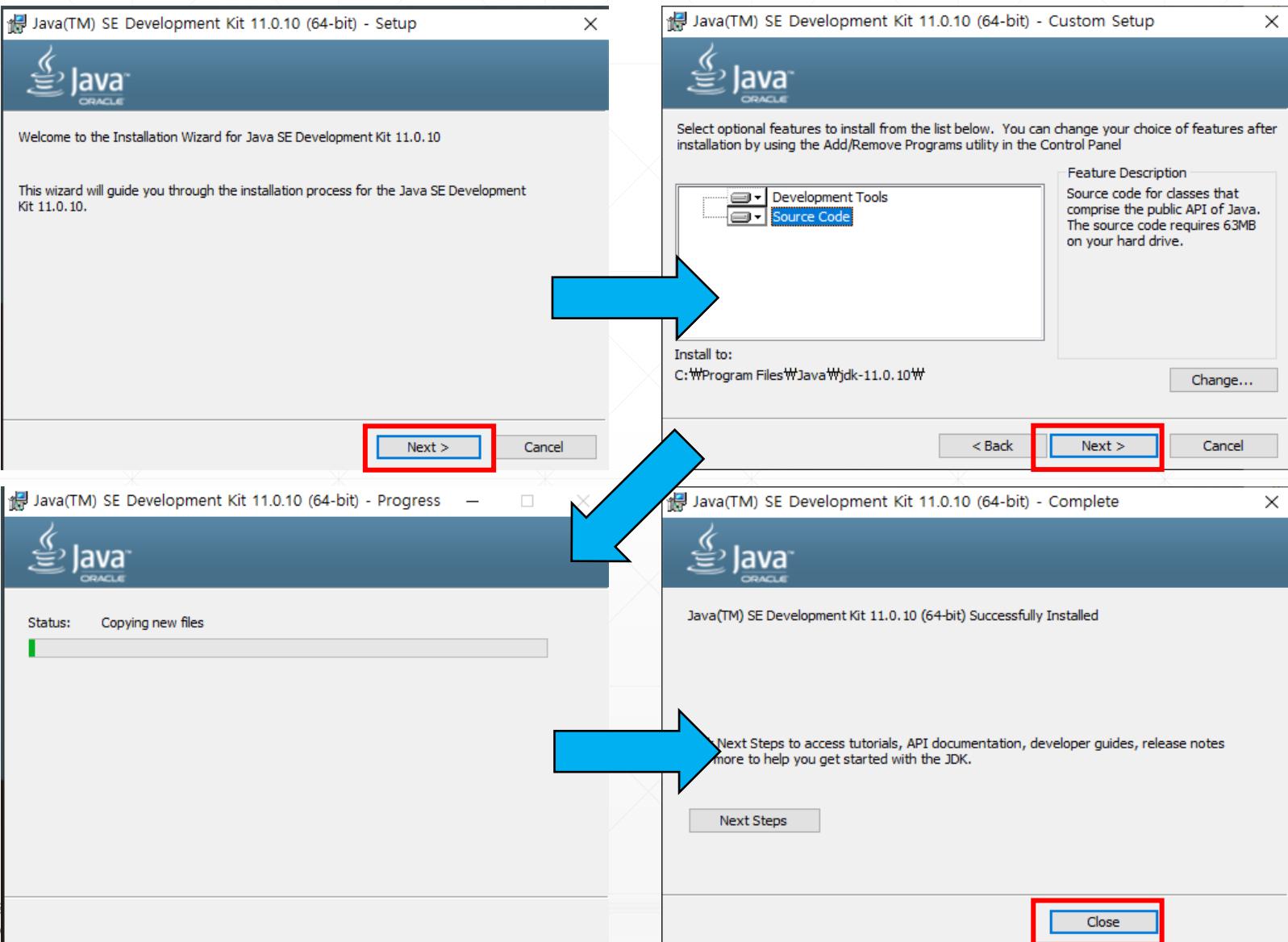
➤ Oracle Java SE 11

- <https://www.oracle.com/kr/java/technologies/javase-downloads.html>

The screenshot shows the Oracle Java SE Downloads page. At the top, there's a navigation bar with links for Products, Industries, Resources, Support, Events, Developer, and Partners. Below the navigation bar, a banner states "Java SE subscribers have more choices" and notes that Java SE subscribers will receive JDK 11 updates until at least September of 2026. The page features tabs for Java 8 and Java 11, with Java 11 being active. A section titled "Java SE Development Kit 11.0.14" provides information about the license and checksums. Below this, there are tabs for Linux, macOS, Solaris, and Windows, with Windows being active. A table lists two download options: "x64 Installer" (140.24 MB) and "x64 Compressed Archive" (157.79 MB). The "x64 Compressed Archive" link is highlighted with a red box.

Product/file description	File size	Download
x64 Installer	140.24 MB	jdk-11.0.14_windows-x64_bin.exe
x64 Compressed Archive	157.79 MB	jdk-11.0.14_windows-x64_bin.zip

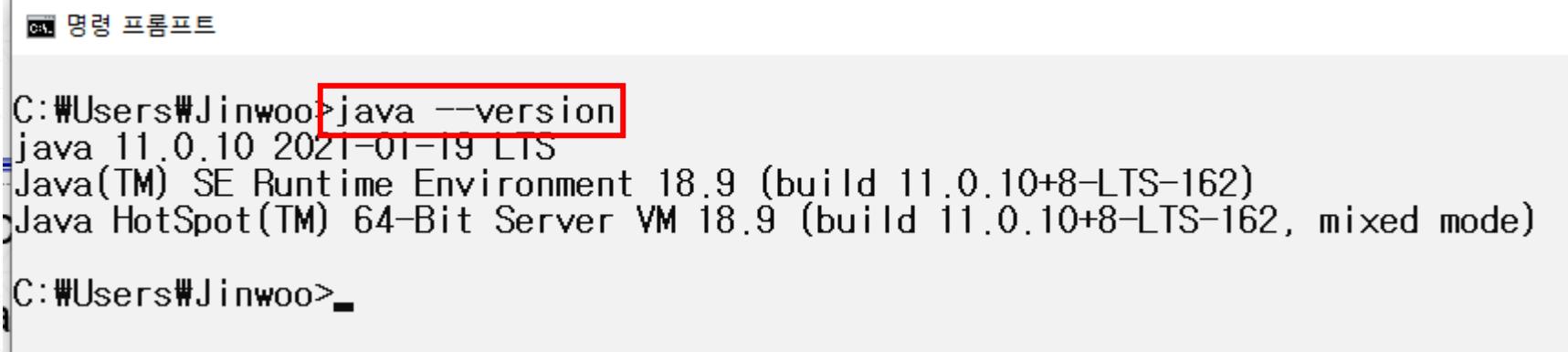
JDK Installation (Optional)



JDK Installation (Optional)

■ Checking Java version

- Start – cmd



```
C:\Users\Jinwoo>java --version
java 11.0.10 2021-01-19 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.10+8-LTS-162)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.10+8-LTS-162, mixed mode)

C:\Users\Jinwoo>
```

Computer Language



Introduction to Java

Hardware + Software

Computer: Hardware

*machine , device
physical.*



Computer: Software

Kind of program.
↳ computer program, application.) ⇒ running on HW.



Computer: Programming Language

to make program!

■ Machine language \rightarrow low-level

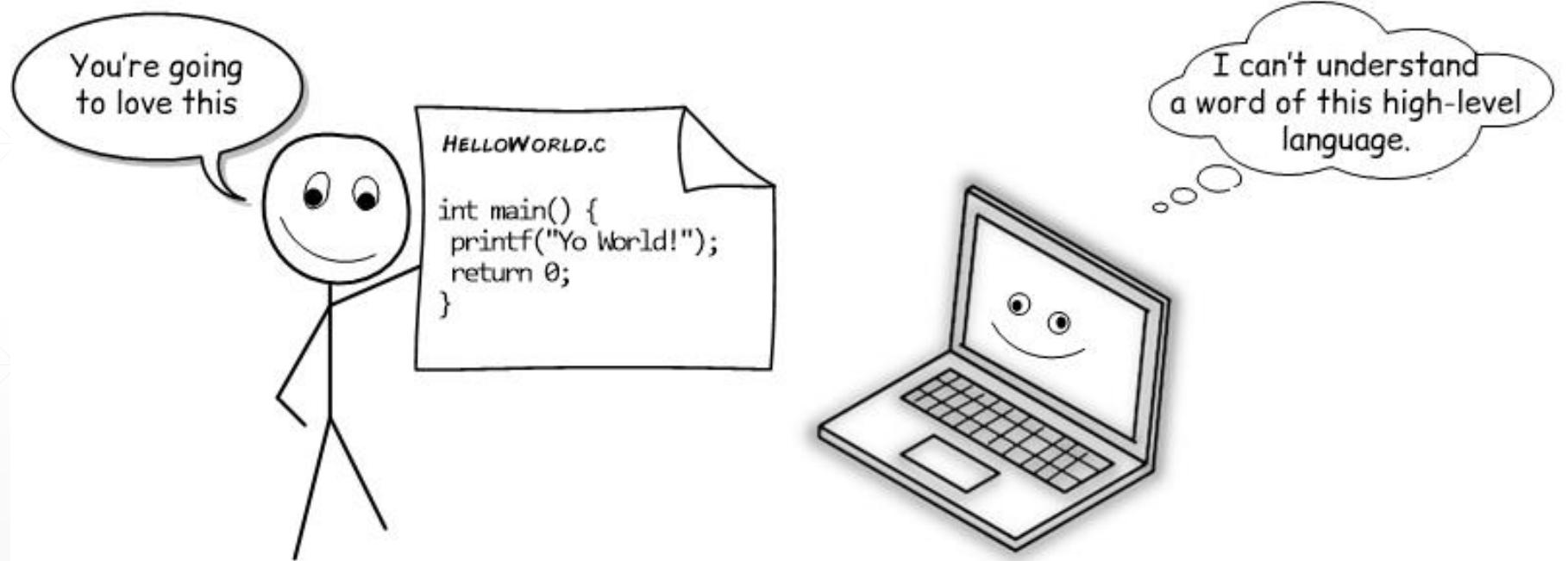
- Machine-readable code
- Binary representation (0 & 1)
- CPU only understands this language \rightarrow human can not understand

■ (High-level) Programming language

like (C C++ C# Java JS Python ...)

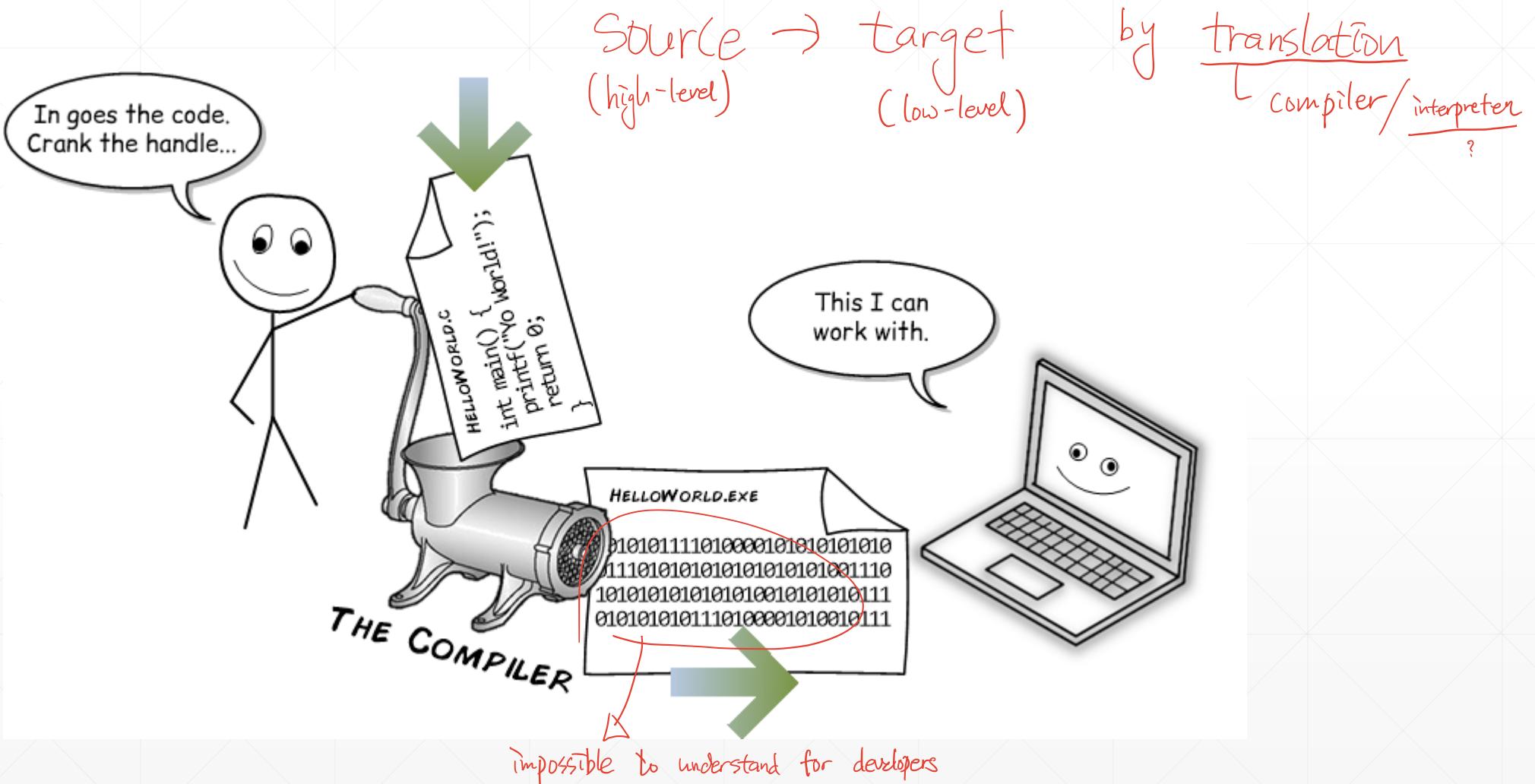
- Human-readable code
- Bridge between a machine and a human
 - But, CPU still cannot understand human-readable representation

Computer: Programming Language (cont'd)

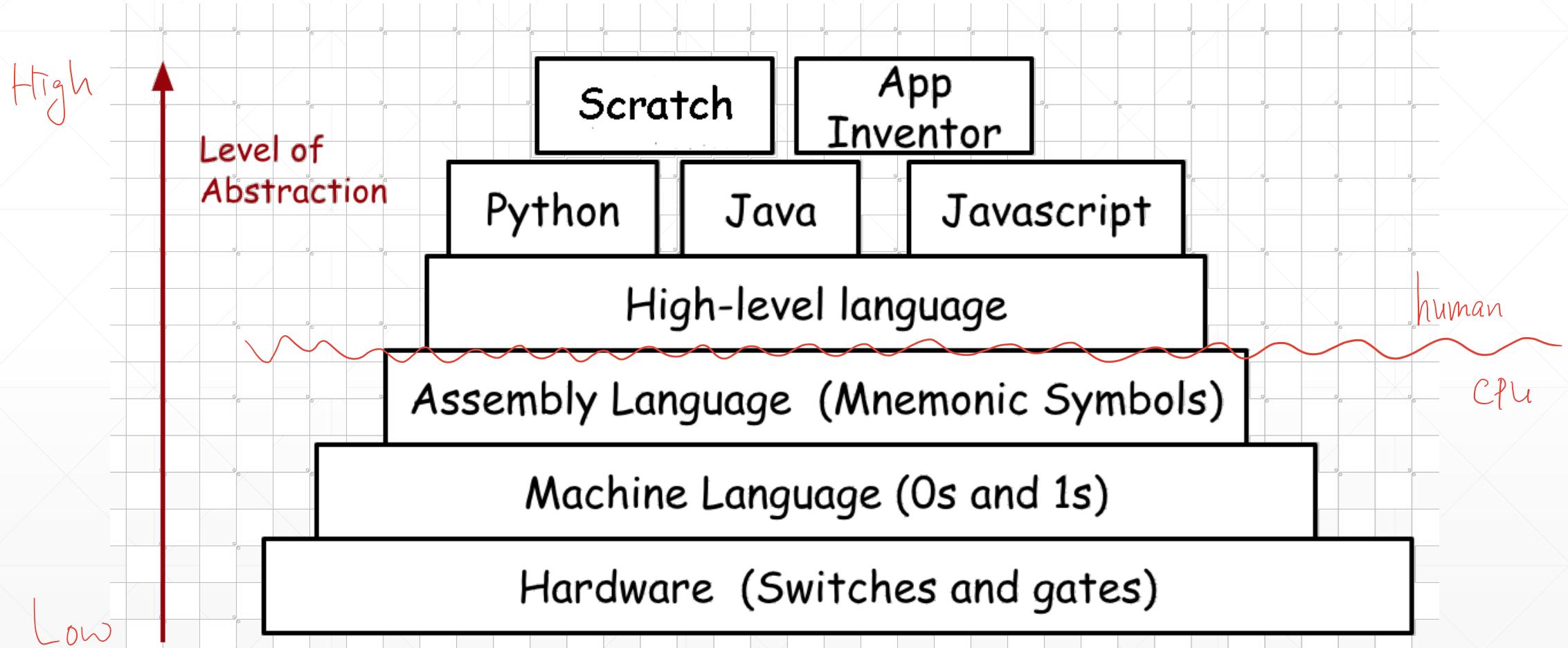


→ Converting is needed

Computer: Programming Language (cont'd)



Computer: Programming Language (cont'd)



Computer: Programming Language (cont'd)

■ Source file

- Text file written in a (High-level) programming language
- Example) .c, .java file

■ Compile

- Translation from high-level to low-level
- Translates source codes into machine instructions

Java

■ Class-based, object-oriented programming language

- Originally, “Oak” programming language
- Developed by James Gosling
- Released in 1995 by Sun Microsystems
- Still very popular language to develop various kinds of applications
- Maintained by Oracle since 2010

in *Android*



James Gosling

- Father of Java programming language

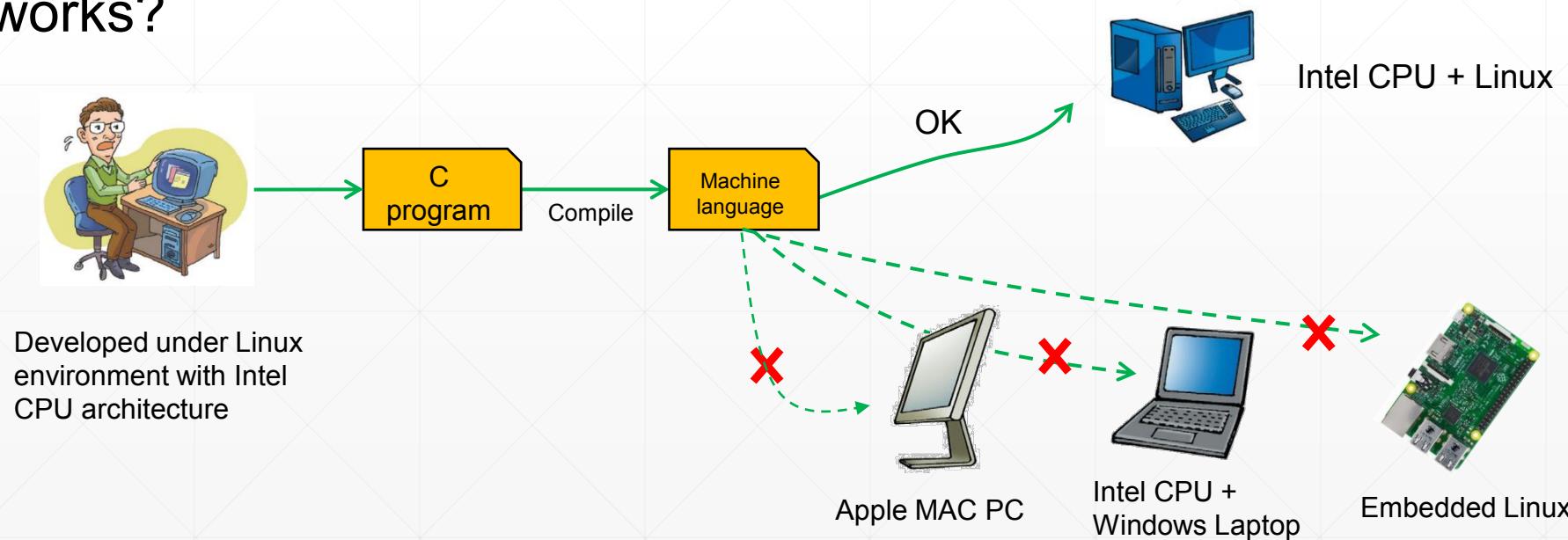


Java (cont'd)

■ WORA (Write Once Run Anywhere) *- slogan of Java.*

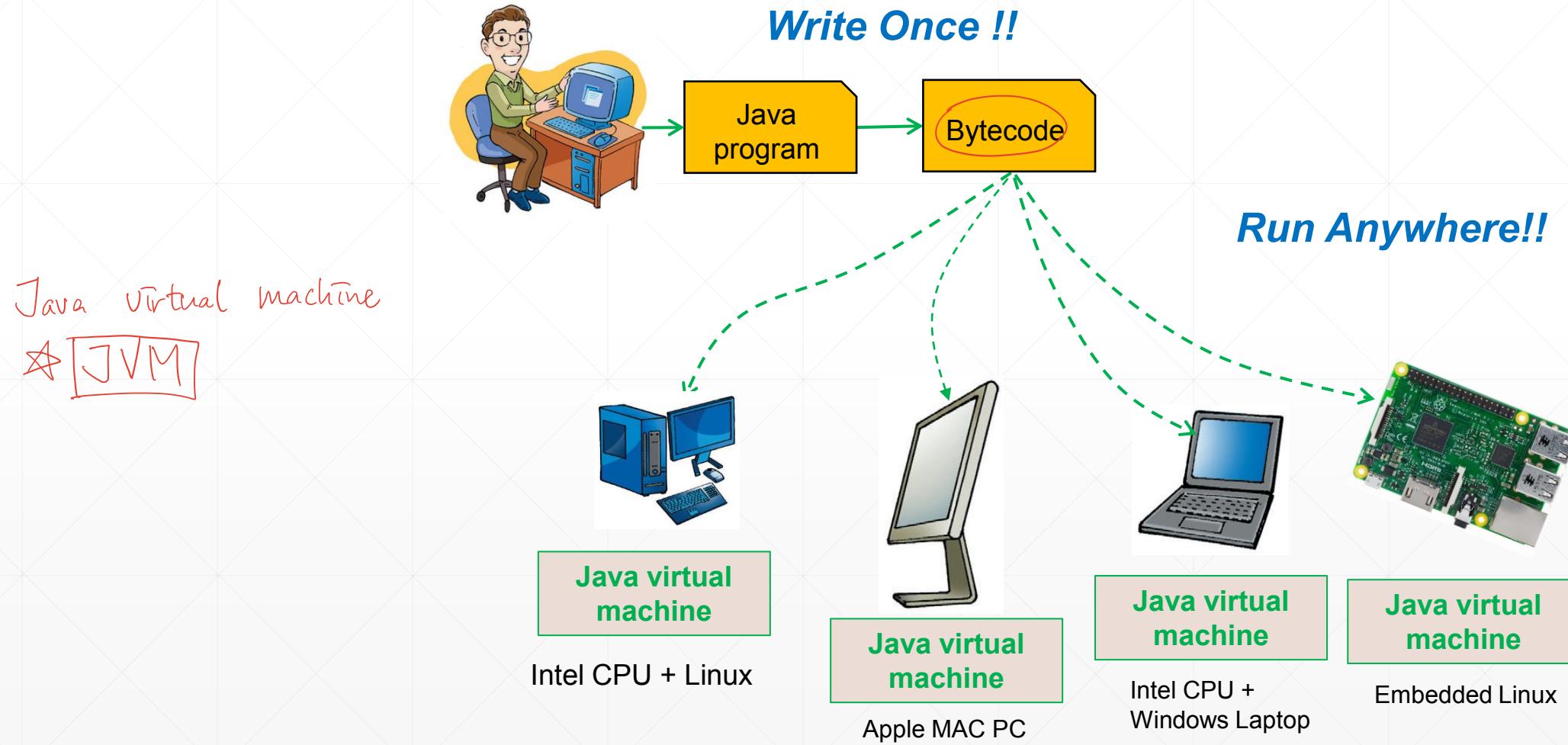
- Platform independent (cross-platform) ↗ x86, x64, ARM 등 (architecture) 상관없이 실행가능!
HW, OS, ...
- Weakness of traditional languages like C, C++, etc.
- A program written in Java can be executed on any architecture

■ How C works?



Java (cont'd)

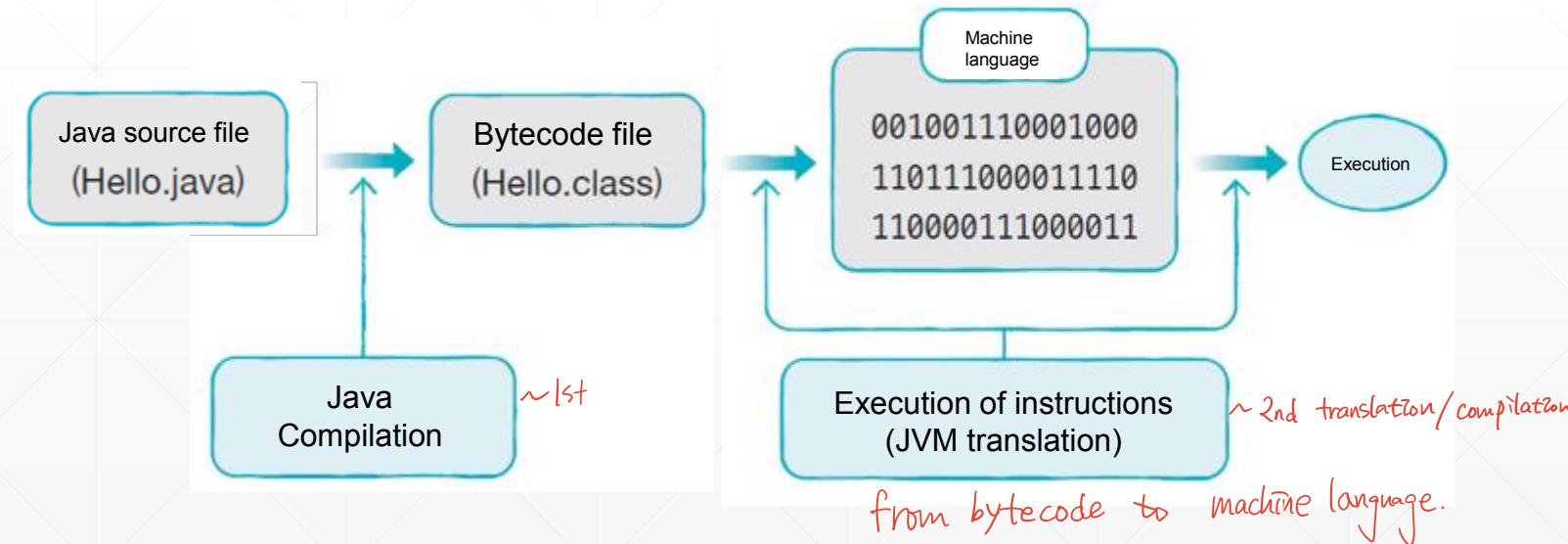
■ How Java works?



Java (cont'd)

■ WORA (Write Once Run Anywhere)

- Java programs consist of byte code files (.class), not complete machine language
 - Bytecode files cannot be executed directly on the OS ↗ bytecode is not complete machine readable language.
written
- Java Virtual Machine (JVM) translates and executes complete machine language



Java (cont'd)

■ Java Virtual Machine (JVM)

➤ Provides an equivalent Java execution environment

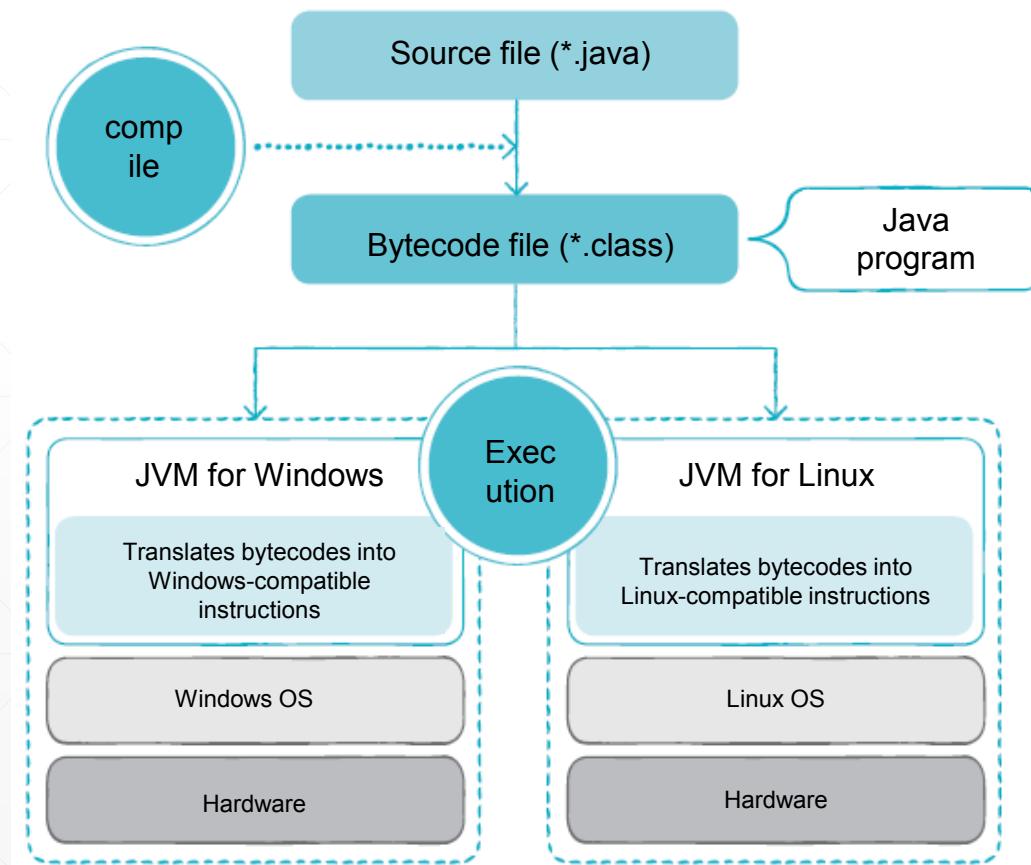
➤ JVM itself is platform dependent

Java code is platform independent

TC/C++
compile → in Intel Cpu + Windows

(machine language
only for Intel CPU & Windows.)

⇒ for other config, other compilation required.
... platform dependency?



Java (cont'd)

■ Object-oriented programming (OOP)

- Encapsulation 개체화
- Inheritance 상속
- Polymorphism 다형성
- ...

■ Memory management

■ Multi-threaded

■ Opensource libraries

- ...

Java (cont'd)

■ JDK/JRE

- JDK (Java Development Kit)
 - Environment for developing and executing Java programs
 - JRE + Development Kit
- JRE (Java Runtime Environment)
 - Environment for executing Java programs
 - JVM + Java standard class libraries

■ API/API reference

- Application Programming Interface (API)
 - Pre-built Java class libraries
- API Reference
 - Specification document for APIs
 - <https://docs.oracle.com/en/java/javase/11/docs/api/>

Getting Started

■ Java project

```
1 > public class Hello {  
2 >   public static void main(String[] args) {  
3 >     System.out.println("Hello, World!");  
4 >     // This is our first  
5 >   }  
6 > }
```

method block

class block

- Defining class and methods
- Statement
 - Semicolon (;) must be appended to represent the end of statements
- Comment
 - Single line (// ...)
 - Multiple lines /* ... */

Q&A

■ Next... in this week (eClass video)

- Variable
- Types

Computer Language



Variables & Types

Agenda

- Variables
- Types

Variables

Types

Variables

building blocks for programming language

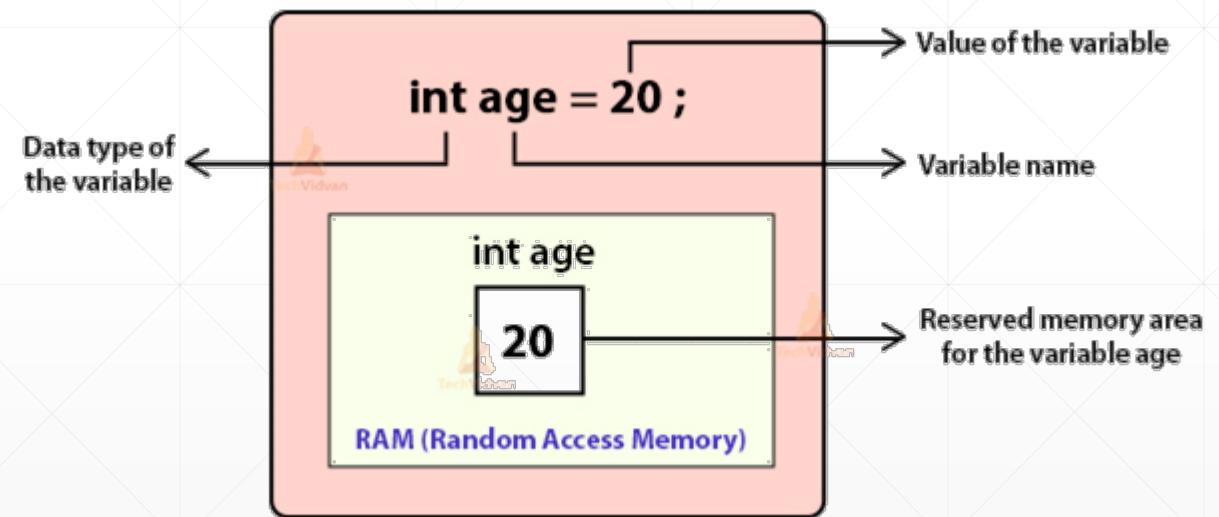
■ A way to store data in a computer

- Declaration of variables
 - What kind (type) of data?
 - How to call it? (name)

type Variable name

int age ;

double value ;



- The contents of a variable can be changed (variable)

Variables (cont'd)

■ Naming rule

- Variable names are case-sensitive ↗ `int age ; != int AGE ;`
- Unlimited-length sequence of Unicode letters and digits ↗ *길이 무제한!*
- Can begin with a **letter**, the dollar sign **"\$"**, or the underscore character **"_"**
- Special characters like **"@"**, **"!"**, **"#"**, and whitespaces are not allowed
 - Example) price, \$price, _price (possible)
 - Example) 1v, @speed, \$#value (impossible)
- Java keywords are not allowed
- Boolean literal (true/false), null literals are not allowed

Variables (cont'd)

■ Naming rule

- Java keywords are not allowed

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Variables (cont'd)

■ Naming convention

- Begin your variable names with a letter, not "\$" or "_"
- Use full words instead of cryptic abbreviations
 - Speed, gear, age are more intuitive than s, g, and a
- If your variable name consists of only one word, spell that word in all lowercase letters
- If your variable name consists of more than one word, capitalize the first letter of each subsequent word
 - Example) myAge, myFinalGrade

Variables (cont'd)

■ Valid variable names

```
int name;  
char student_ID;  
int whatsYourNameMyNameIsKitae;  
int barChart; int barchart;  
int 가격;
```

■ Invalid variable names

```
int 3Chapter;           // use of number for the first character  
double if;             // java keyword (if)  
char false;            // java keyword (false)  
float null;            // java keyword (null)  
short ca%lc;           // special character (%)
```

Variables (cont'd)

■ Initialization

- Variables needs to be **initialized** before being used
- Use assignment operator ('=') to assign/initialize a value to the variable

type variable
int radius;
 value
radius = 10;

assign value to variable

// declaration
// initialization

int radius = 10;

char c1 = 'a', c2 = 'b', c3 = 'c';

double weight = 75.56;

// declaration & initialization

Variables (cont'd)

■ Access

- Variables needs to be initialized before being used
- Variables can be accessed by its name
- The value of variables can be used for printing, calculation, etc
- The value of a variable can be copied to another variable

■ Example)

```
int hour = 3, minute = 5;  
  
System.out.println(hour + "h" + minute + "m");  
  
System.out.println(hour * 60 + minute + "m");  
  
int totalMinute = hour * 60 + minute;  
  
System.out.println(totalMinute);
```



Variables **Types**

Types

■ Java data types

➤ Primitive types

- Types for number, character, boolean

➤ Non-primitive types

- String, array, class, etc.

■ Literal

- Source code representation of a fixed value
- Represented directly in the code without requiring computation
- Can be assigned to a variable

Type	Size in bytes	Range	Default Value
byte	1 byte	-128 to 127	0
short	2 bytes	-32,768 to 32,767	0
int	4 bytes	-2,147,483,648 to 2,147,483,647	0
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0
float	4 bytes	approximately ±3.40282347E+38F (6-7 significant decimal digits) Java implements IEEE 754 standard	0.0f
double	8 bytes	approximately ±1.79769313486231570E+308 (15 significant decimal digits)	0.0d
char	2 bytes	0 to 65,536 (unsigned)	'\u0000'
boolean	Not precisely defined*	true or false	false

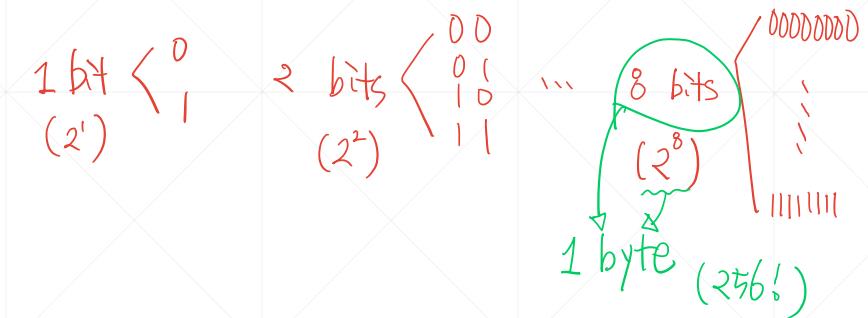
Types: Number

■ Integer types

- Stores whole numbers without decimals (fraction)
 - Includes positive and negative



Type	Size in bytes	Range	Default Value
byte	1 byte	-128 to 127 2^8	0
short	2 bytes	-32,768 to 32,767 2^{16}	0
int	4 bytes	-2,147,483,648 to 2,147,483, 647 2^{32}	0
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 2^{64}	0



Types: Number (cont'd)

■ Integer types

➤ Integer literal

- Type of 'int' unless the literal ends with the letter 'L' or 'l'
- Binary system (stating with 0b or 0B, 0/1 representation)

0b1011	$\rightarrow 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \rightarrow 11$
0b10100	$\rightarrow 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \rightarrow 20$

suffix for "long" type

L l

모든 정수는 기본적으로 int.
정수 L ⇒ Long

- Octal system (stating with 0, 0-7 representation)

013	$\rightarrow 1 \times 8^1 + 3 \times 8^0 \rightarrow 11$
0206	$\rightarrow 2 \times 8^2 + 0 \times 8^1 + 6 \times 8^0 \rightarrow 134$

- Decimal system

$$12 = 1 \times 10^1 + 2 \times 10^0$$

$$365 = 3 \times 10^3 + 6 \times 10^1 + 5 \times 10^0$$

10ⁿ

- Hexadecimal system (starting with 0x, 0-F representation)

0xB3	$\rightarrow 11 \times 16^1 + 3 \times 16^0 \rightarrow 179$
0x2A0F	$\rightarrow 2 \times 16^3 + 10 \times 16^2 + 0 \times 16^1 + 15 \times 16^0 \rightarrow 10767$

16ⁿ

0	10 A
1	11 B
2	12 C
3	13 D
4	14 E
5	15 F
6	
7	
8	
9	

Types: Number (cont'd)

■ Integer types

- Use of underscore ('_') character in Integer literal
 - Use of underscore to separate groups of digits is allowed
 - Use of underscore can improve the readability of the code

```
int price = 20_100;                                // 20100
long cardNumber = 1234_5678_1357_9998L;           // 1234567813579998L
long controlBits = 0b10110100_01011011_10110011_11110000;
long maxLong = 0x7fff_ffff_ffff_ffffL;
int age = 2____5;                                  // 25
```

- Underscores cannot be used in the following places:
 - At the beginning or end of a number
 - Adjacent to a decimal point in a floating-point literal
 - Prior to an F or L suffix
 - Inside prefix 0b and 0x
 - In positions where a string of digits is expected

```
int x = 15_;                                     // error. At the end of a number
double pi = 3_.14;                               // error. Adjacent to a decimal point
long idNum = 981231_1234567_L; // error. Prior to an F or L suffix
int y = 0_x15;                                   // error. Inside the prefix '0x'
```

Types: Number (cont'd)

■ Floating point types

- Represents numbers with a fractional part, containing one or more decimals

Type	Size in bytes	Range	Default Value
float	4 bytes	approximately $\pm 3.40282347E+38F$ <u>(6-7 significant decimal digits)</u> Java implements IEEE 754 standard	0.0f
double	8 bytes	approximately $\pm 1.79769313486231570E+308$ <u>(15 significant decimal digits)</u>	0.0d

Significant decimal digit ↑ accuracy range ↑

Types: Number (cont'd)

■ Floating point types

➤ Floating point literal

- Type of 'double' and it can optionally end with the letter 'D' or 'd'
- Type of 'float' if the literal ends with the letter 'F' or 'f'
- Can represent scientific (floating-point) number with an "e"

5e2 $\rightarrow 5.0 \times 10^2 = 500.0$ E $\rightarrow 10^0$
0.12E-2 $\rightarrow 0.12 \times 10^{-2} = 0.0012$ E $\rightarrow 10^{-2}$

- Example)

```
float var = 3.14; // OK?  $\rightsquigarrow 3.14f;$   
double var = 3.14; ok  
double var = 314e-2; ok // OK?
```

\rightsquigarrow result: 3.14

Types: Character

■ Char type

- Used to store a single character (0000 to FFFF)
 - Unicode (utf-16) character
 - Unicode table: <https://unicode-table.com/en/blocks/>
 - The character must be surrounded by single quotes, like 'A' or 'c'

{ 16 bit =
2 byte

Type	Size in bytes	Range	Default Value
char	2 bytes	0 to 65,536 (unsigned)	'\u0000'

```
char a = 'A';
char b = '글';
char c = '\u0041'; // Unicode of 'A'
char d = '\uae00'; // Unicode of '글'
```

Types: String

■ String type

- Non-primitive type *reference type*
- Used to store a sequence of characters (i.e., string)
- The string must be surrounded by double quotes, like “Hello, Java!”
- String literal can be assigned to a String object

```
String str = "Good";
```

■ Escape character

- A character starting with backslash ('\\')
- Can be used to represent special character
- Can be used to control printing of a string

Types: String (cont'd)

■ Escape character

Escape character	Purpose	Escape character	Purpose
\b	Backspace	\n	Line feed
\r	Carriage return	\t	Tab
'	Print '	"	Print '
\\"	Print \	\u(Unicode)	Print character based on Unicode

```
System.out.println("I love \"Java\"");
System.out.println("Name "+ID+" "+Age);
System.out.println("Computer\nLanguage\u2661");
```

Types: Boolean

■ Boolean type

- Represents *true* or *false*
- Can be stored in a Boolean type variable or used with condition statements

```
boolean myValue = true;  
System.out.println(myValue);  
myValue = 10 < 15;  
System.out.println(myValue);  
myValue = 10 == 15;  
System.out.println(myValue);
```

Types: Null

■ Null literal

- Represents “not existent”
- Can be used for a reference type (will be discussed later)

```
int n = null;           // error!  
String str = null;
```



기자는 나쁜 문痞
@mold_bread

Following

Learned that the difference between null and 0 in a programming language.
What ...?

[Translate from Korean](#)



Constant

HT
BT

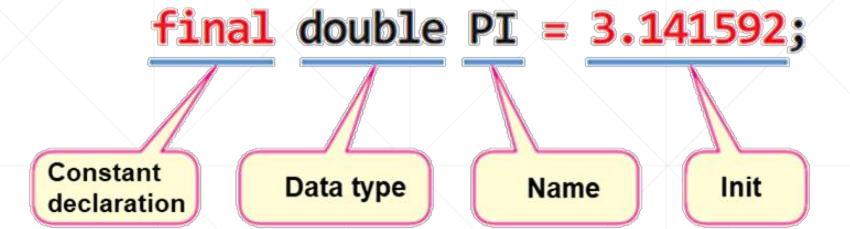
■ Final variable (constant)

- Unchangeable, read-only variable
- Can be declared by adding *final* keyword

```
final double PI = 3.141592;  
  
System.out.println(PI);  
  
PI = 5.00;
```

- Naming convention *of Constant*
 - All uppercase with words separated by underscores ("_")

```
static final int MIN_WIDTH = 4;  
static final int MAX_WIDTH = 999;  
static final int GET_THE_CPU = 1;
```



Type Conversion: Promotion

■ Automatic conversion

- Converting a smaller size type to a larger size type

byte -> short -> int -> long -> float -> double *Smaller size* *Larger size*

- Done automatically when,

- Passing a smaller size type to a larger size type
- Performing an arithmetic operation with integer-type values
 - Byte, short, char type values are automatically converted to int type values
- Performing an arithmetic operation with different types of values
 - Arithmetic operation is only performed with the same type *operands* *피연산자*
 - Smaller type value is automatically converted to a larger type value

promotion

Larger type = smaller type

Type Conversion: Promotion (cont'd)

■ Automatic conversion

- Example) Passing a smaller size type to a larger size type

```
long longValue = 500000L;  
double doubleValue = longValue;  
System.out.println(longValue); ↳ 500000 (long type)  
System.out.println(doubleValue); ↳ 500000.0 (double type)
```

```
char chValue = 'A';  
int intValue = chValue;  
System.out.println(intValue); ↳ 65 (unicode of 'A')
```

```
short byteValue = 10;  
char chValue = byteValue;
```

↳ size of byte : same
range : different

char (2 bytes)	short (2 bytes)
(+)	(+, -)

↳ A/R 가능

Type Conversion: Promotion (cont'd)

■ Automatic conversion

- Example) Performing an arithmetic operation with integer-type values

```
short x= 10;  
short y = 20;  
  
int 가 되어야 함  
short total = x + y;  
System.out.println(total);
```

- Example) Performing an arithmetic operation with different types of values

```
int intValue = 10;  
int anotherValue = 3;  
double doubleValue = 3;  
  
System.out.println(intValue / anotherValue);  
System.out.println(intValue / doubleValue);
```

→ 3

→ 3.333333333333335

16 significant digits

Type Conversion: Casting

not automatic, manual

■ Manual conversion

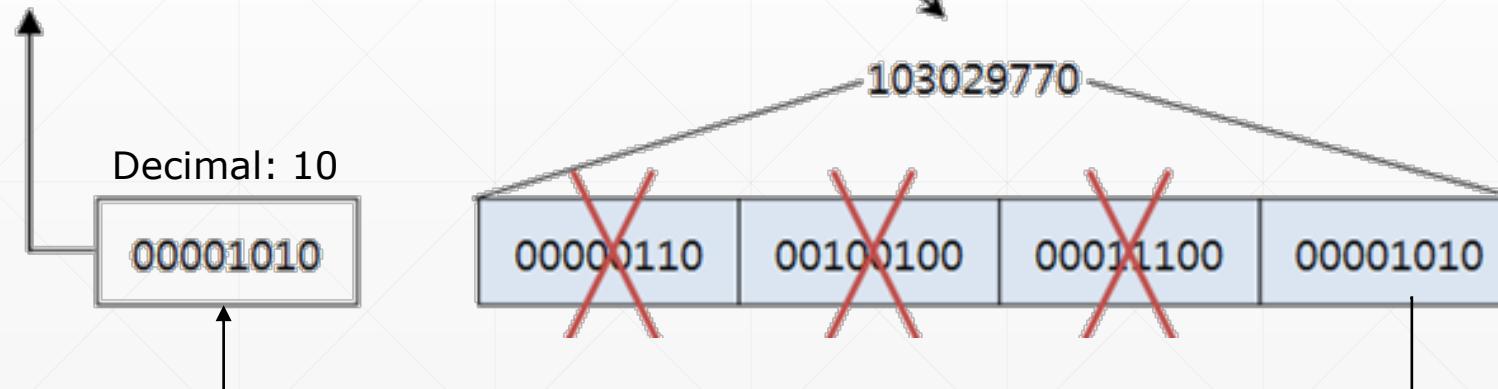
- Converting a larger size type to a smaller size type

Small byte -> short -> int -> long -> float -> double / large

- Done manually by casting operation '(type)'
- May result in the loss of a value

casting
Smaller type = (smaller type) larger type

```
int intValue = 103029770;  
byte byteValue = (byte) intValue;
```



/ byte 만 보조됨

Type Conversion: Casting (cont'd)

■ Manual conversion

- Example) casting from double to int

```
double myDouble = 11.50;
int myInt = (int) myDouble;

System.out.println(myDouble); ↗ 11.5
System.out.println(myInt); ↗ 11
```

- Example) casting from int to char (to print a character!)

```
int myInt = 67;
char myChar = (char) myInt;

System.out.println(myInt); ↗ 67
System.out.println(myChar); ↗ C (capital C)
```

Q&A

■ Next week (eClass video)

- Basic operators

Computer Language



Basic operator



Agenda

- Scanner & Print
- Basic Operators

Scanner & Print

Basic Operators

Scanner

Let's take an input from the user!

➤ System.in

- Standard input stream in Java
- Return byte-type data
- Not developer-friendly



➤ Scanner class

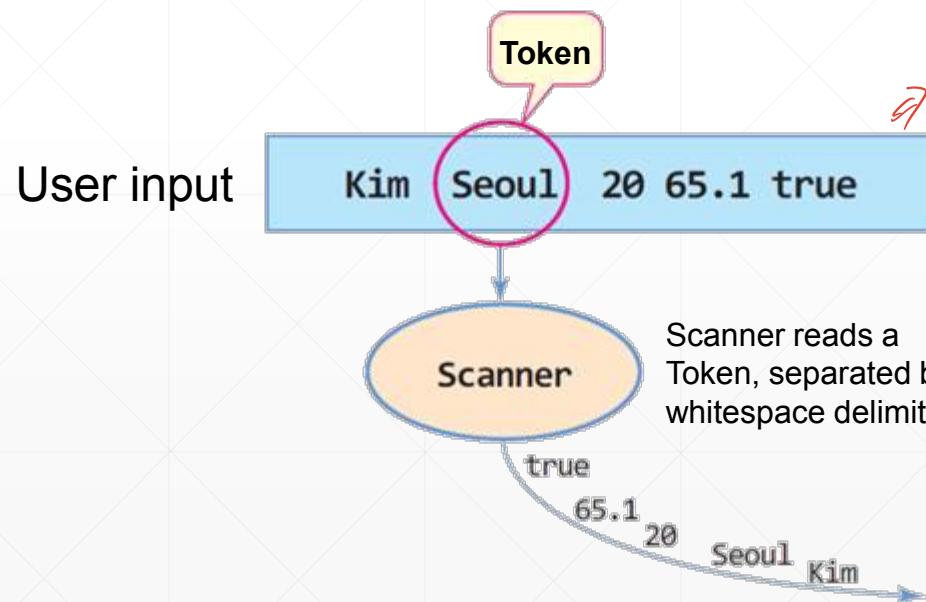
- Need to import java.util.Scanner calss
- Ask System.in to take a sequence of bytes from the user
- Convert input bytes to data with an arbitrary type and then return!



Scanner (cont'd)

■ Reading key inputs

- Scanner reads an item based on the whitespace delimiter
 - Whitespace character: '\t', '\f', '\r', '\n', " "
- Scanner can read byte streams and convert it to various data types



```
Scanner scanner = new Scanner(System.in);  
  
String name = scanner.next();           // "Kim"  
String city = scanner.next();          // "Seoul"  
int age = scanner.nextInt();          // 20  
double weight = scanner.nextDouble(); // 65.1  
boolean single = scanner.nextBoolean(); // true
```

Scanner (cont'd)

■ Scanner methods

Method	Description
next()	Reads a value as a String from the user
nextBoolean()	Reads a boolean value from the user
nextByte()	Reads a byte value from the user
nextDouble()	Reads a double value from the user
nextFloat()	Reads a float value from the user
nextInt()	Reads an int value from the user
nextLine()	Reads one line (before '\n') from the user
nextLong()	Reads a long value from the user
nextShort()	Reads a short value from the user
close()	Close a Scanner
hasNext()	Returns True if a token is given, otherwise waits for a new input. CTRL-Z will break this loop.

Scanner (cont'd)

■ Example)

```
System.out.println("Input your name, city, age, and weight, separated by a single whitespace");
Scanner scanner = new Scanner(System.in);

String name = scanner.next(); // Read a string
System.out.print("My name is " + name + ", ");

String city = scanner.next(); // Read a string
System.out.print("city is " + city + ", ");

int age = scanner.nextInt(); // Read an integer value
System.out.print("age is " + age + "-years old, ");

double weight = scanner.nextDouble(); // Read a floating-point value
System.out.print("Weight is " + weight + "kg, ");

System.out.println("\nOk. Are you single?");
boolean single = scanner.nextBoolean(); // Read a boolean value
System.out.println(single);

System.out.println("\nAny comment?");
String comment = scanner.nextLine(); // Read a line
System.out.println("Your answer: " + comment);

scanner.close(); // Close the scanner
```

Print

■ Basic functions to print out the contents

System.out.[print methods]

- Print out something to the system's standard output

■ Methods

- `println(contents)`: print out the contents and make a newline
- `print(contents)`: print out the contents
- `printf("formatting string", val1, val2, ...)`: print out the values using the formatting string
- Contents can be either literals or variables

Print (cont'd)

■ `printf("formatting string", val1, val2, ...)`

- Prints the values using the formatting string

■ Formatting string

```
% [argument_index$] [flags] [width] [.precision] conversion
```

- Only %conversion is mandatory
- argument_index\$: the position of the argument in the argument list (e.g., 1\$, 2\$, ...)
- flags: controls the modification of output
- width: the minimum number of characters to be written
- precision: the digits after the radix point
- conversion: determines how the argument should be formatted

Print (cont'd)

■ Formatting string

➤ Flags: controls the modification of output

- ‘-’ : left-justified *왼쪽 정렬*
- ‘+’ : includes sign, whether positive or negative
- ‘0’ : zero padding \Rightarrow width가 5인 때, 123이 출력되면 00123 출력.
- ...

➤ Conversion: determines how the argument should be formatted

- ‘d’: integer
- ‘f’, ‘g’: floating-point number $f \rightarrow \text{float}$, $g \rightarrow \text{double}$
- ‘s’: string
- ...

Print (cont'd)

■ Example)

```
System.out.printf("%1$d %3$d %2$d\n", 10, 20, 30);
```

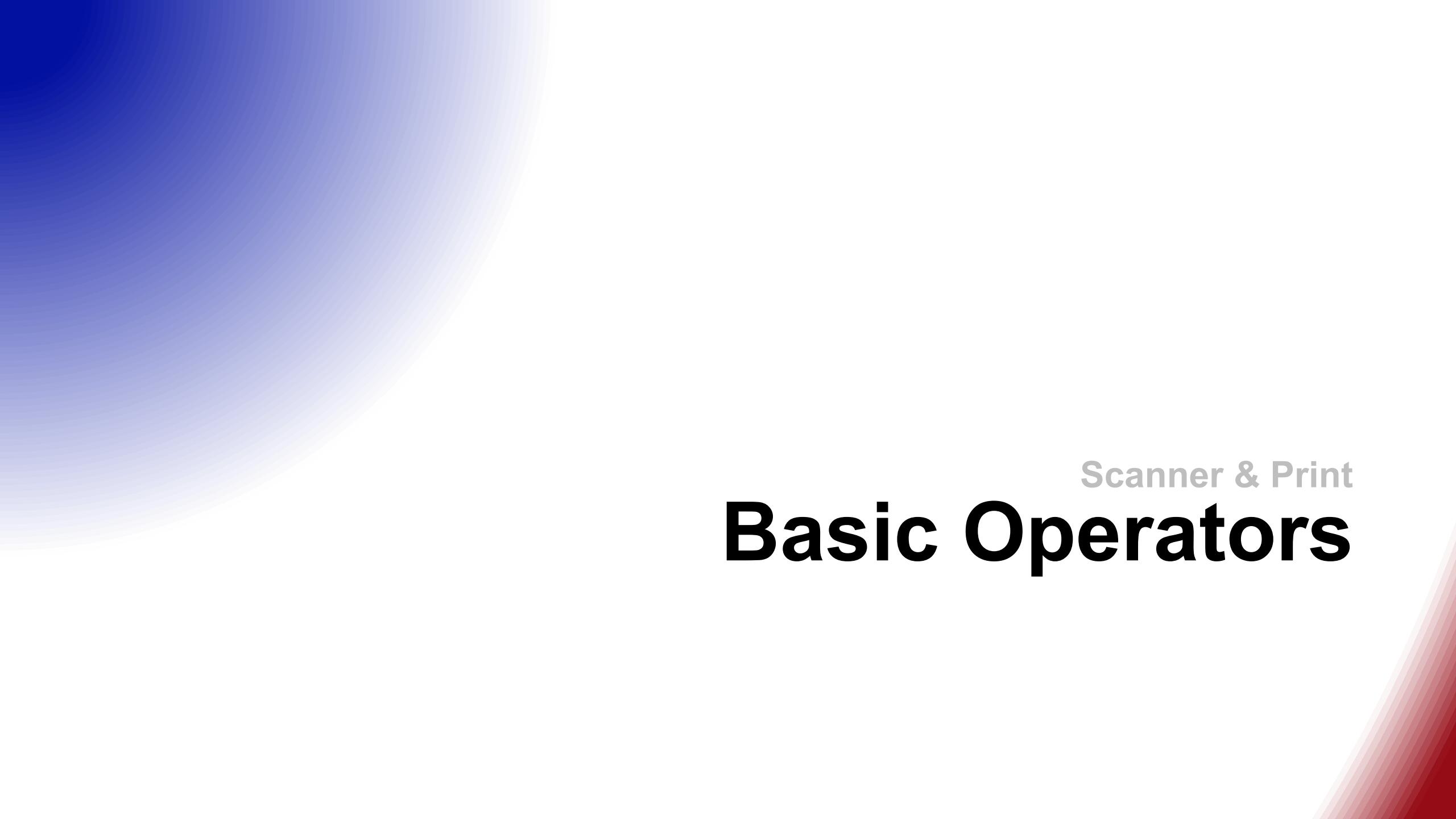
```
System.out.printf("%1$d %3$f %2$s\n", 10, "Hi", 20.5);
```

```
System.out.printf("%1$+5d %3$.2f %2$s\n", 10, "Hi", 20.5);
```

```
System.out.print("Hoy Hoy~");
```

```
System.out.println("Hey Hey~");
```

```
System.out.print("Hay Hay~");
```

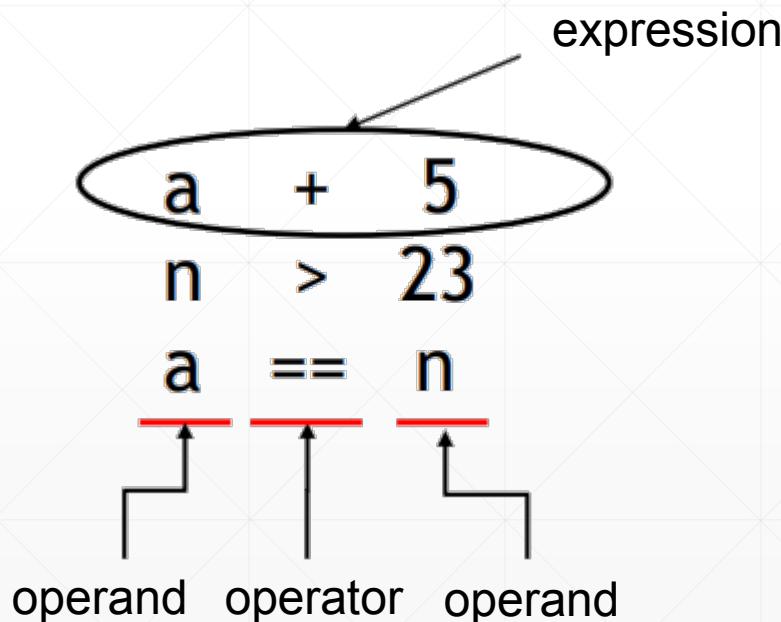


Scanner & Print

Basic Operators

Operator

- First step to do something with values!
- Operators are special symbols that perform specific operations on one, two, or three *operands (literals or variables)*, and then return a result



Type	Operator	Type	Operator
In/decrement	$\textcolor{red}{++ \ --}$ ^{unary}	Bit	$\& ^ \sim$
Arithmetic	$+ - * / \%$	Conditional	$\&\& !$
Shift	$>> << >>>$		$= *= /= += -=$
Relational	$> < >= <= == !=$	Assignment	$\&= ^= =$ $<<= >>= >>>=$

Operator: Arithmetic

■ Arithmetic computation

Description	Operator	Example	Result
Additive	+	$25.5 + 3.6$	29.1
Subtraction	-	$3 - 5$	-2
Multiplication	*	$2.5 * 4.0$	10.0
Division	/	$5/2$	2
Remainder	%	$5\%2$	1

몫

나머지

➤ Ex) check if x is an odd or not

```
int x = n % 2; // if x is 1, n is an odd number, otherwise even
```

Operator: Arithmetic (cont'd)

■ String concatenation

- When one of operands for ‘+’ operation is String type
- Example)

```
System.out.println("30"+5);  
System.out.println(30+5);  
System.out.println("Java "+11.0);
```

Operator: Increment/Decrement

■ Unary increment and decrement operators

- A single operand is required
- Increase or decrease the value by 1

■ Prefix operators ($++a$, $--a$)

- Increase the value by 1 first, and then return the value

증가, 감소 순서 반환?

■ Postfix operators ($a++$, $a--$)

- Return the value first, and then increase/decrease the value by 1

Operator: Increment/Decrement (cont'd)

Example)

```
int myNum = 1;  
  
System.out.println(myNum++);  
  
System.out.println(++myNum);  
  
System.out.println(--myNum);  
  
System.out.println(myNum--);  
  
System.out.println(myNum);
```

Operator: Relational

- Used to determine if one operand is greater than, less than, equal to, or not equal to another operand
 - Returns true or false

Description	Operator	Example	Result
Equal to	<code>==</code>	<code>1 == 3</code>	False
Not equal to	<code>!=</code>	<code>1 != 3</code>	True
Greater than	<code>></code>	<code>3 > 5</code>	False
Greater than or equal to	<code>>=</code>	<code>10 >= 10</code>	True
Less than	<code><</code>	<code>3 < 5</code>	True
Less than or equal to	<code><=</code>	<code>1 <= 0</code>	False

Operator: Conditional

- Used to determine the logic between variables or values

- Returns true or false

Description	Operator	Example	Result
Conditional AND (returns true if both operands are true) $\begin{array}{l} tt \rightarrow t \\ tf \rightarrow f \\ ft \rightarrow f \\ ff \rightarrow f \end{array}$	$\&\&$	$\begin{array}{l} (3 < 5) \&\& (1 == 1) \\ (3 > 5) \&\& (1 == 1) \end{array}$	$\begin{array}{l} \text{True} \\ \text{False} \end{array}$
Conditional OR (returns true if one of the statements is true) $\begin{array}{l} tt \rightarrow t \\ tf \rightarrow t \\ ft \rightarrow t \\ ff \rightarrow f \end{array}$	$\ $	$\begin{array}{l} (3 < 5) \ (1 == 1) \\ (3 > 5) \ (1 == 1) \end{array}$	$\begin{array}{l} \text{True} \\ \text{True} \end{array}$
Complement (inverts the value of a Boolean)	!	$\begin{array}{l} !(3 < 5) \\ !(3 > 5) \end{array}$	$\begin{array}{l} \text{False} \\ \text{True} \end{array}$

Operator: Relational + Conditional

■ Example)

```
// true if one is 20s  
(age >= 20) && (age < 30)
```

```
// what about 20 <= age < 30 ?
```

```
// true if a character c is a capital letter  
(c >= 'A') && (c <= 'Z')
```

```
// true if (x,y) is inside the rectangle from (0,0) to (50,50)  
(x>=0) && (y>=0) && (x<=50) && (y<=50)
```

Operator: Relational + Conditional (cont'd)

■ Example)

```
System.out.println('a' > 'b');

System.out.println(3 >= 2);

System.out.println(-1 < 0);

System.out.println(3.45 <= 2);

System.out.println(3 == 2);

System.out.println(3 != 2);

System.out.println(!(3 != 2));

System.out.println((3 > 2) && (3 > 4));

System.out.println((3 != 2) || (-1 > 0));
```



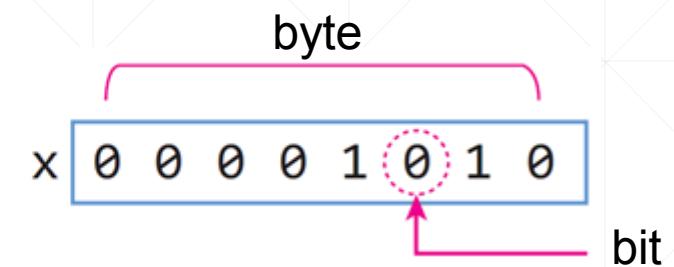
understanding required!

Operator: Bit & Shift

Operators for the bits of operands

- Bitwise conditional operators
 - AND, OR, XOR, NOT operation on bits
- Bit shift operators
 - Operations to shift the bits to the left/right

byte x = 10;



Description	Operator
AND (returns true if both bits are 1)	$a \& b$
OR (returns true if one of the bits is 1)	$a b$
NOT (inverts a bit pattern)	$\sim a$
XOR (returns true if two bits are different)	$a ^ b$

Operator: Bit & Shift (cont'd)

- Operators for the bits of operands

The diagram illustrates four binary operations:

- AND:** $01101010 \& 11001101 = 01001000$. The result is shown with green vertical bars highlighting the 4th and 7th bits from the left.
- OR:** $01101010 | 11001101 = 11101111$. The result is shown with green vertical bars highlighting the 4th and 7th bits from the left.
- XOR:** $01101010 ^ 11001101 = 10100111$. The result is shown with green vertical bars highlighting the 4th and 7th bits from the left.
- NOT:** $\sim 01101010 = 10010101$. The result is shown with green vertical bars highlighting the 4th and 7th bits from the left.

Operator: Bit & Shift (cont'd)

■ Operators for shifting the bits of operands

Description	Operator
Arithmetic Left shift When shifting left, the most-significant bit is lost, and a 0 bit is inserted on the other end	$a \ll 1$
Arithmetic Right shift When shifting right with an arithmetic right shift , the least-significant bit is lost, and the most-significant bit is copied	$a \gg b$
Logical Right shift When shifting right with a logical right shift , the least-significant bit is lost, and a 0 bit is inserted on the other end	$a \ggg b$

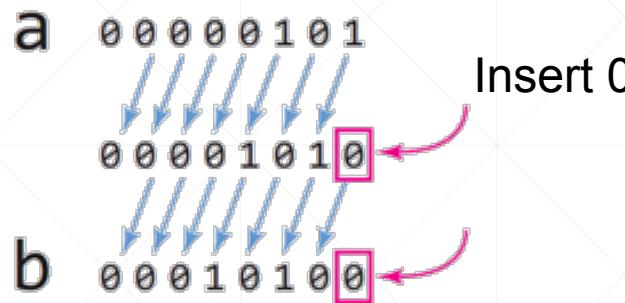
Operator: Bit & Shift (cont'd)

MSB \Rightarrow 자기부분

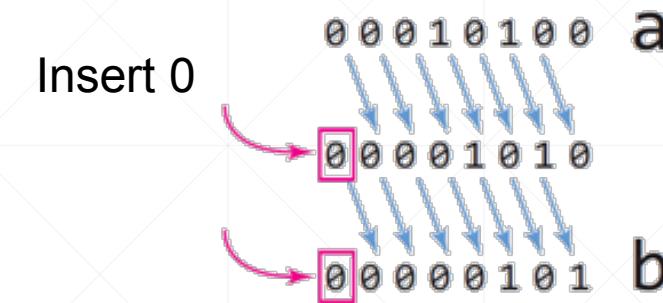
LSB \Rightarrow 자기부분

■ Example)

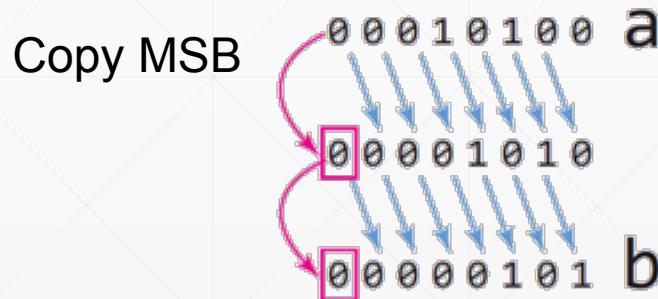
```
byte a = 5; // 5  
byte b = (byte)(a << 2); // 20
```



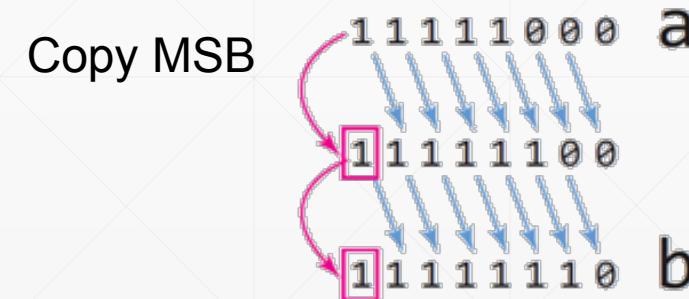
```
byte a = 20; // 20  
byte b = (byte)(a >> 2); // 5
```



```
byte a = 20; // 20  
byte b = (byte)(a >> 2); // 5
```



```
byte a = (byte)0xF8; // -8  
byte b = (byte)(a >> 2); // -2
```



Operator: Bit & Shift (cont'd)

■ Example)

```
short a = (short)0b0101010111111111;  
short b = (short)0x00ff;
```

0b 00000000 11111111

printf("%04x"): print a 4-digit number with
a hexadecimal (0~f) format

```
System.out.printf("%04x\n", (short)(a & b)); // bitwise AND  
System.out.printf("%04x\n", (short)(a | b)); // bitwise OR  
System.out.printf("%04x\n", (short)(a ^ b)); // bitwise XOR  
System.out.printf("%04x\n", (short)(~a)); // bitwise NOT
```

```
int c = 20;  
int d = -8;
```

```
System.out.println(c <<2);  
System.out.println(c >>2); // arithmetic right shift  
System.out.println(d >>2); // arithmetic right shift  
System.out.println(d >>>2); // logical right shift
```

Operator: Assignment

- Operators to assign values to variables
- Simple assignment (=)
 - Ex) myValue = 5; // assign 5 to the variable *myValue*

Operator: Assignment (cont'd)

- Operators to assign values to variables

- Compound assignment**

Operator	Example	Same As
<code>+=</code>	<code>x += 3</code>	<code>x = x + 3</code>
<code>-=</code>	<code>x -= 3</code>	<code>x = x - 3</code>
<code>*=</code>	<code>x *= 3</code>	<code>x = x * 3</code>
<code>/=</code>	<code>x /= 3</code>	<code>x = x / 3</code>
<code>%=</code>	<code>x %= 3</code>	<code>x = x % 3</code>
<code>&=</code>	<code>x &= 3</code>	<code>x = x & 3</code>
<code> =</code>	<code>x = 3</code>	<code>x = x 3</code>
<code>^=</code>	<code>x ^= 3</code>	<code>x = x ^ 3</code>
<code>>>=</code>	<code>x >>= 3</code>	<code>x = x >> 3</code>
<code><<=</code>	<code>x <<= 3</code>	<code>x = x << 3</code>

Operator: Precedence

- Operators with higher precedence are evaluated before operators with relatively lower precedence

- Top priority: ()

- Associativity

- A rule for the operators with equal precedence
- All binary operators except for the assignment operators are evaluated from left to right
 - Assignment operators are evaluated right to left

Operator: Precedence (cont'd)

high
↓
low

Operators	Precedence	Associativity
postfix	expr++ expr--	
unary	++expr --expr +expr -expr ~ !	←
multiplicative	* / %	
additive	+ -	
shift	<< >> >>>	
relational	< > <= >= instanceof	
equality	== !=	→
bitwise AND	&	
bitwise XOR	^	
bitwise OR		
logical AND	&&	
logical OR		
ternary	? :	
assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=	←

Q&A

- Next week (eClass video)

- Conditions & Loop

Computer Language



Conditions & Loop

Agenda

- Condition
- Loop

Condition Loop

Program's Flow

■ Our simple program so far

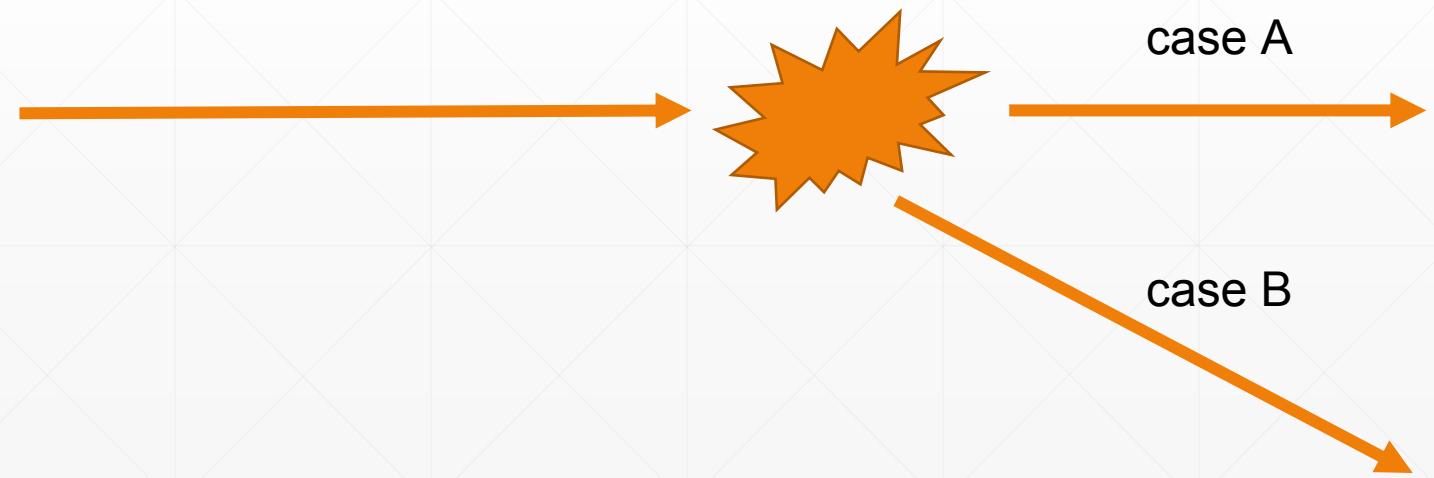
- Take input from the user
- Perform some operations
- Print the results

Normal, sequential flow



■ What if we want to handle various cases?

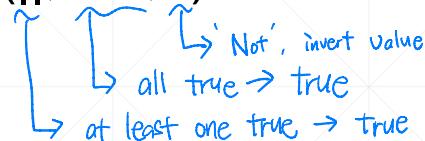
make some branches by conditions!



Program's Flow (cont'd)

■ How can we define a condition?

- {
 - Relational operators (`==`, `!=`, `>`, `<`, `<=`, `>=`)
 - Conditional operators (`||`, `&&`, `!`)


'Not', invert value.
all true → true
at least one true → true

■ How can we make a branch based on the condition?

- {
 - **IF** statements
 - **Switch** statements
- Conditional statements (logic) execute a certain section of code only if a particular test (condition) evaluates to *true*

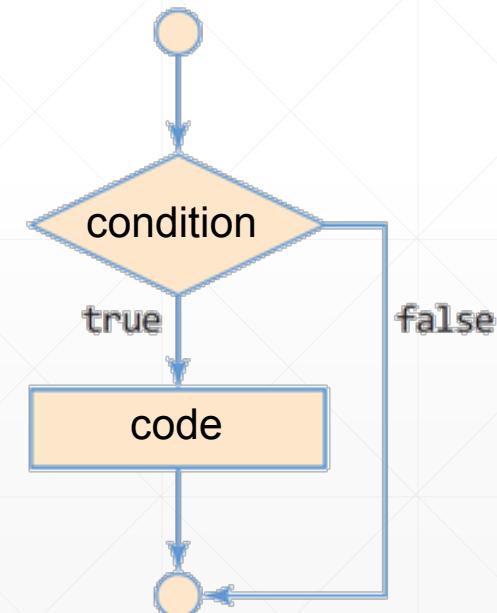
Condition: IF

■ Simple IF statement

- If an evaluation of the condition is true, then execute a code section
- Brackets can be omitted, if a code to be executed is a single line

```
if(n%2 == 0) {  
    System.out.print(n);  
    System.out.println("is an even number.");  
}  
  
if(score >= 80 && score <= 89)  
    System.out.println("Your grade is B!");
```

```
if(condition){  
    ... code to be executed ...  
}
```



Condition: IF (cont'd)

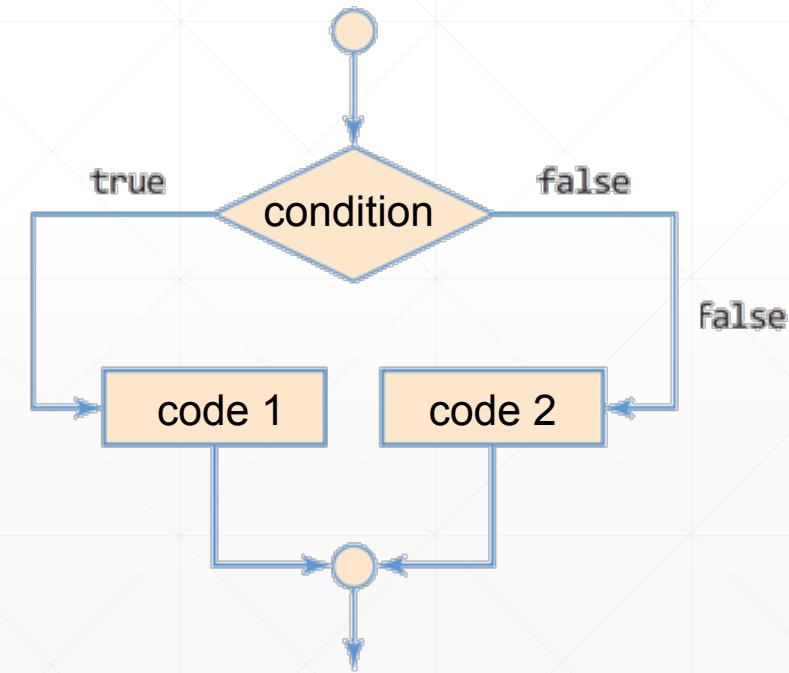
■ IF-else statement

- If an evaluation of the condition is true, then execute a code section 1
- If false, then execute a code section 2

```
int score = 55;

if(score >= 90)
    System.out.println("A+!");
else
    System.out.println("F!");
```

```
if(condition){
    code section 1
}
else {
    code section 2
}
```



Condition: IF (cont'd)

■ Ternary statement

3가지 형태로

→ simple, reduce lines, but also readability).

- Condition ? opr2 : opr3
- If an evaluation of the condition is true, then the result will be opr2
- If false, then result will be opr3
- Alternative of IF-else statement

```
int x = 5;  
int y = 3;  
  
int s;  
if(x>y)  
    s = 1;  
else  
    s = -1;
```

```
int s = (x>y) ? 1 : -1;
```

Condition: IF (cont'd)

■ Ternary statement

- Example)

```
int a = 3, b = 5;  
System.out.println("The diff between two numbers is " + ((a>b)?(a-b):(b-a)));
```

- How to implement this statement using if-else statement?

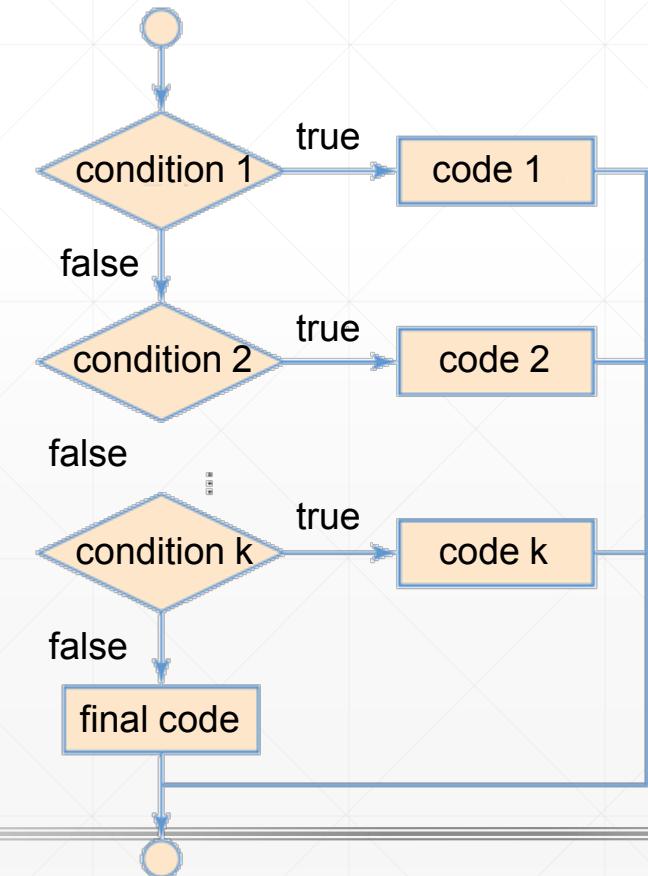
Condition: IF (cont'd)

■ Multiple if-else statement

- In case we need to have multiple branches
- The conditions are mutually exclusive (only one section will be executed)
- If – (else if)* - else

```
if(score >= 90) { // 90 <= score  
    grade = 'A';  
}  
else if(score >= 80) { // 80 <= score < 90  
    grade = 'B';  
}  
else if(score >= 70) { // 70 <= score < 80  
    grade = 'C';  
}  
else if(score >= 60) { // 60 <= score < 70  
    grade = 'D';  
}  
else { // < 60  
    grade = 'F';  
}
```

```
if(condition){  
    code secction 1  
}  
else if(condition2){  
    code section 2  
}  
else if(condition3){  
    code section 3  
}  
...  
else {  
    final section  
}
```



Condition: IF (cont'd)

■ Multiple if-else statement

- Example) what happens we just use multiple if statements?

```
if(score >= 90) { // 90 <= score  
    grade = 'A';  
}  
if(score >= 80) { // 80 <= score < 90  
    grade = 'B';  
}  
if(score >= 70) { // 70 <= score < 80  
    grade = 'C';  
}  
if(score >= 60) { // 60 <= score < 70  
    grade = 'D';  
}  
else { // < 60  
    grade = 'F';  
}
```

} given SCORE = 100
grade result = D
↳ grade history: A → B → C → D

Condition: IF (cont'd)

■ Nested if statement

[if in if]

- If statement can be used inside another if statement
- Example)

Nested

```
Scanner scanner = new Scanner(System.in);

System.out.print("Input your score (0~100): ");
int score = scanner.nextInt();

System.out.print("Input your grade (1~4): ");
int year = scanner.nextInt();

if(score >= 60) { // greater than or equal to 60
    if(year != 4)
        System.out.println("PASS!"); // if not a senior, pass!
    else if(score >= 70)
        System.out.println("PASS!"); // if senior && greater than or equal to 70, pass!
    else
        System.out.println("FAIL!"); // if senior && less than 70, fail!
} else // less than 60, fail!
    System.out.println("FAIL!");

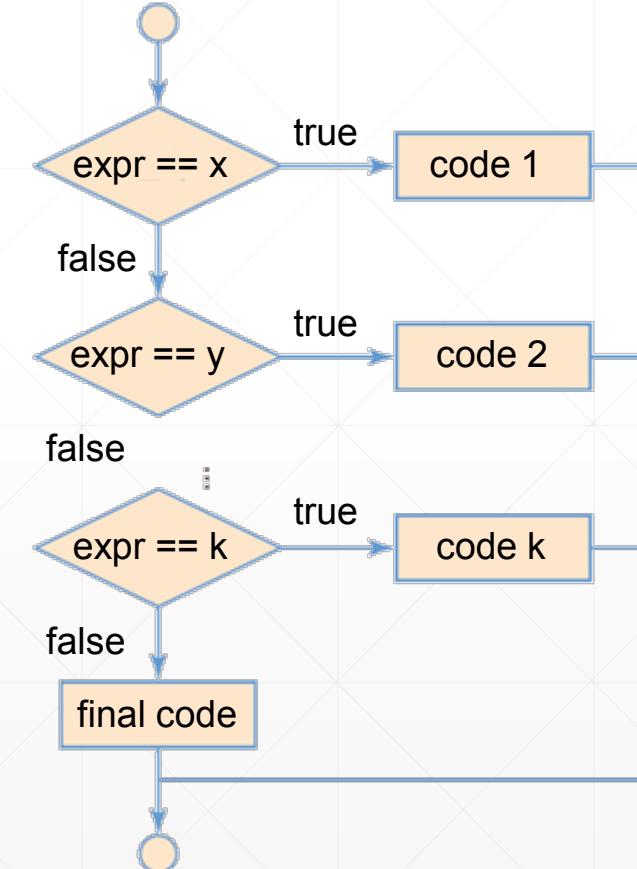
scanner.close();
```

Condition: Switch

■ Switch-case statement

- Evaluates if a given expression matches each case value
- Case value
 - Only char, integer, String literals are allowed
 - Floating-point literal is not allowed
- If matched, execute its code block
 - Break literally breaks the switch statement
- If nothing matched, execute a code block of the default section

```
switch (expression){  
    case x:  
        code block 1  
        break;  
  
    case y:  
        code block 2  
        break;  
    ...  
    default:  
        final code block  
}
```



Condition: Switch (cont'd)

■ Example)

```
int b;
switch(b%2) {
    case 1 : ...; break; // integer literal is allowed
    case 2 : ...; break;
}

char c;
switch(c) {
    case '+' : ...; break; // char literal is allowed
    case '-' : ...; break;
}

String s = "Yes";
switch(s) {
    case "Yes" : ...; break; // String literal is allowed
    case "No" : ...; break;
}
```

```
switch(a) {
    case a :      // Error!
    case a > 3 : // Error!
    case a == 1 : // Error!
}
```

Condition: Switch (cont'd)

■ Example)

```
Scanner scanner = new Scanner(System.in);
```

```
char grade;
System.out.print("Input your score (0~100): ");
int score = scanner.nextInt();
switch (score / 10) {
    case 10: // score = 100
    case 9: // score 90~99
        grade = 'A';
        break;
    case 8: // score 80~89
        grade = 'B';
        break;
    case 7: // score 70~79
        grade = 'C';
        break;
    case 6: // score 60~69
        grade = 'D';
        break;
    default: // score < 60
        grade = 'F';
}
System.out.println("Your grade is " + grade);
```

Condition: Switch (cont'd)

- Example) What happens if we remove break?

```
Scanner scanner = new Scanner(System.in);
```

```
char grade;
System.out.print("Input your score (0~100): ");
int score = scanner.nextInt();
switch (score / 10) {
    case 10: // score = 100
    case 9: // score 90~99
        grade = 'A';
    case 8: // score 80~89
        grade = 'B';
    case 7: // score 70~79
        grade = 'C';
    case 6: // score 60~69
        grade = 'D';
    default: // score < 60
        grade = 'F';
}
System.out.println("Your grade is " + grade);
```

always 'F'

[break]

important in Switch statement



Conditions **Loop**

Why We Need a Loop?

- Suddenly, wanna make a multiplication table!
 - Let's compute 1×1 , 1×2 , ..., 1×9 !

- How to do this?

```
System.out.println("1x1=" + 1*1);
System.out.println("1x2=" + 1*2);
System.out.println("1x3=" + 1*3);
System.out.println("1x4=" + 1*4);
System.out.println("1x5=" + 1*5);
System.out.println("1x6=" + 1*6);
System.out.println("1x7=" + 1*7);
System.out.println("1x8=" + 1*8);
System.out.println("1x9=" + 1*9);
```

Ok, I can do this.

Why We Need a Loop? (cont'd)

- But, a multiplication table consists of one, two, ..., nine times table!
- How to do this?

```
System.out.println("1x1=" + 1*1);
System.out.println("1x2=" + 1*2);
System.out.println("1x3=" + 1*3);
System.out.println("1x4=" + 1*4);
System.out.println("1x5=" + 1*5);
System.out.println("1x6=" + 1*6);
System.out.println("1x7=" + 1*7);
System.out.println("1x8=" + 1*8);
System.out.println("1x9=" + 1*9);
```

```
System.out.println("2x1=" + 2*1);
System.out.println("2x2=" + 2*2);
System.out.println("2x3=" + 2*3);
System.out.println("2x4=" + 2*4);
System.out.println("2x5=" + 2*5);
System.out.println("2x6=" + 2*6);
System.out.println("2x7=" + 2*7);
System.out.println("2x8=" + 2*8);
```

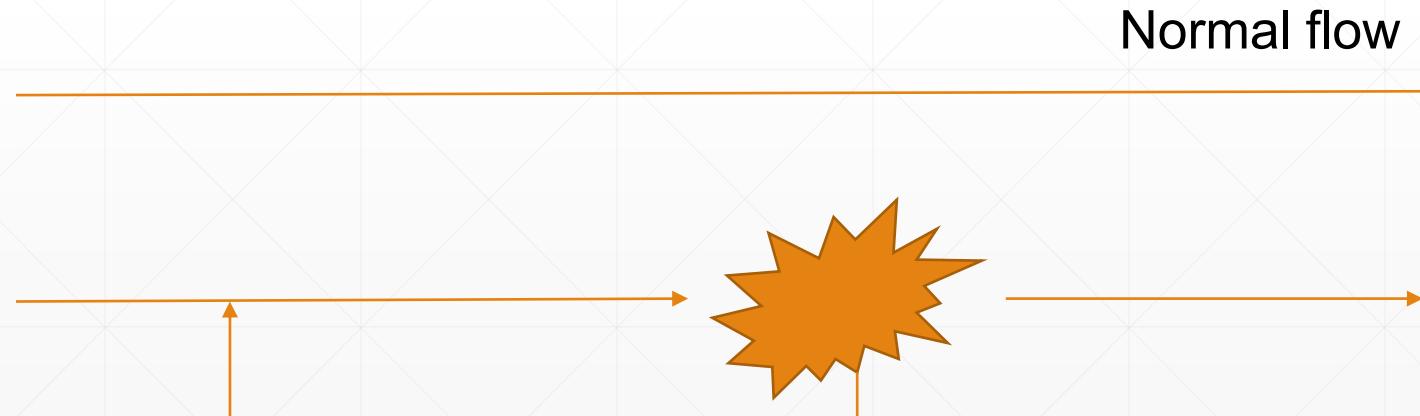
Umm... am I doing correct programming?

Why We Need a Loop? (cont'd)

- How can we do the same/similar tasks iteratively?

- Use Loop statements

- For statement
- While statement
- Do-while statement



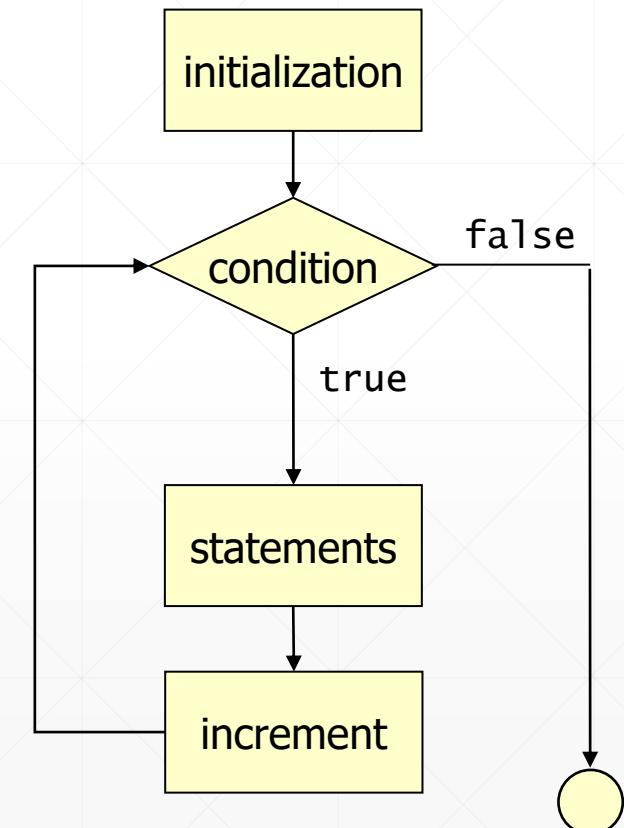
Loop! until a condition matches!

Loop: For

■ Most frequently used loop statement

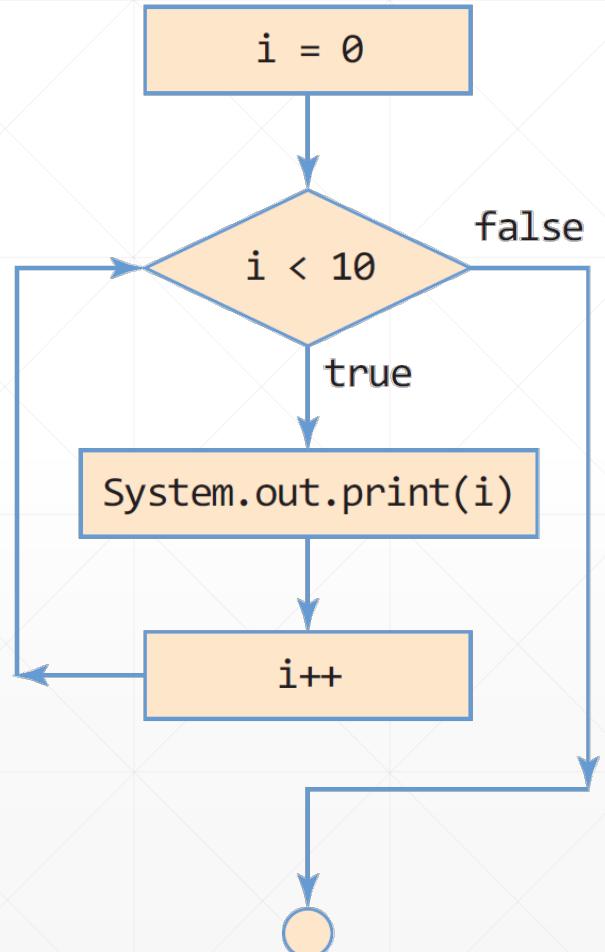
- The initialization expression initializes the loop
- When the condition evaluates to false, the loop terminates
 - Generally, used with counting
- The increment is invoked after each iteration through the loop
 - It is acceptable for this expression to increment or decrement a value

```
for (initialization; condition; increment) {  
    ... statement(s) ...  
}
```



Loop: For (cont'd)

■ Example)



```
for(i=0; i<10; i++) {  
    System.out.print(i);  
}
```

0123456789

Loop: For (cont'd)

■ Example)

```
for(initialization; true; increment) { // if a condition is true, then this is an infinite loop
.....
}
```

```
for(initialization; ; increment) { // if a condition is empty, recognize it as true, so this is also an infinite loop
.....
}
```

```
// initialization and increment can have multiple statements, separated by ','
for(i=0; i<10; i++, System.out.println(i)) {
.....
}
```

```
// It is possible to declare a local variable inside the for loop
for(int i=0; i<10; i++) { // variable i can be only used inside this loop
.....
}
```

Loop: For (cont'd)

■ Example) print from 0 to 9

```
int i;  
for(i = 0; i < 10; i++) {  
    System.out.print(i);  
}
```

```
int i;  
for(i = 0; i < 10; i++)  
    System.out.print(i);
```

↗ Single Line → } 가로 줄에.
→ } 가로 줄에.

■ Example) summate from 0 to 100

```
int sum = 0;  
for(int i = 0; i <= 100; i++)  
    sum += i;
```

```
int i, sum;  
for(i = 0, sum=0; i <= 100; i++)  
    sum += i;
```

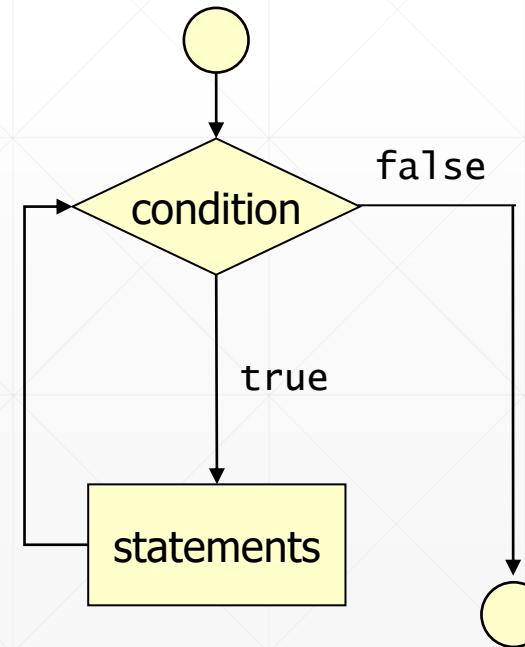
```
int sum = 0;  
for(int i = 100; i >= 0; i--)  
    sum += i;
```

Loop: While

■ Another loop statement

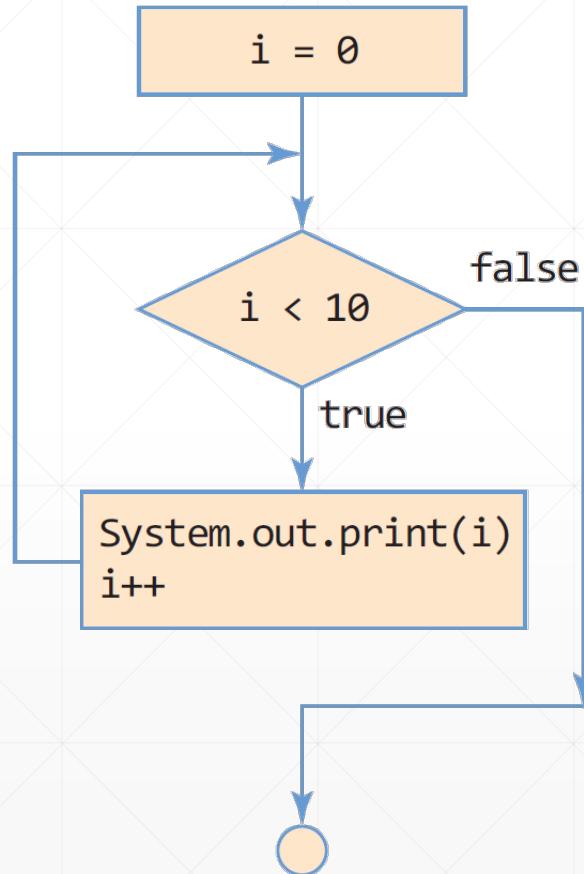
- While statement evaluates expression, which must return a boolean value
- If the expression evaluates to true,
the while statement executes the statement(s) in the while block
- While statement continues testing the expression and executing its block
until the expression evaluates to false

```
while (condition) {  
    ... statement(s) ...  
}
```



Loop: While (cont'd)

■ Example)



```
i = 0;  
while(i<10) {  
    System.out.print(i);  
    i++;  
}
```

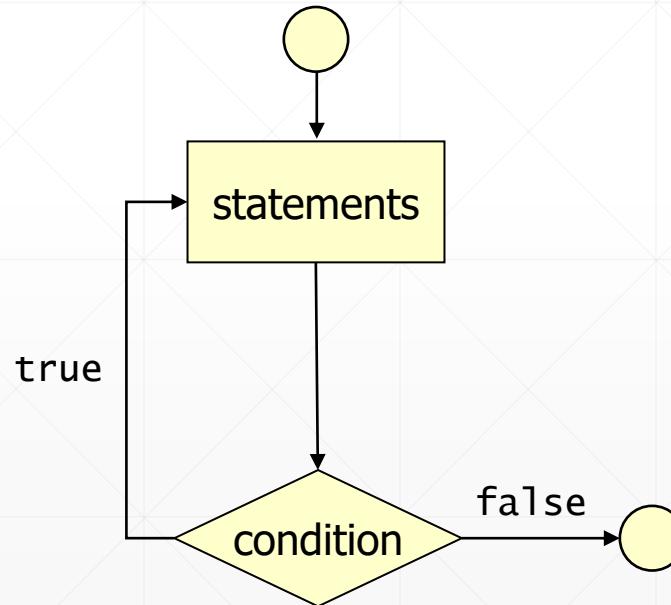
0123456789

Loop: Do-While

■ Another while statement

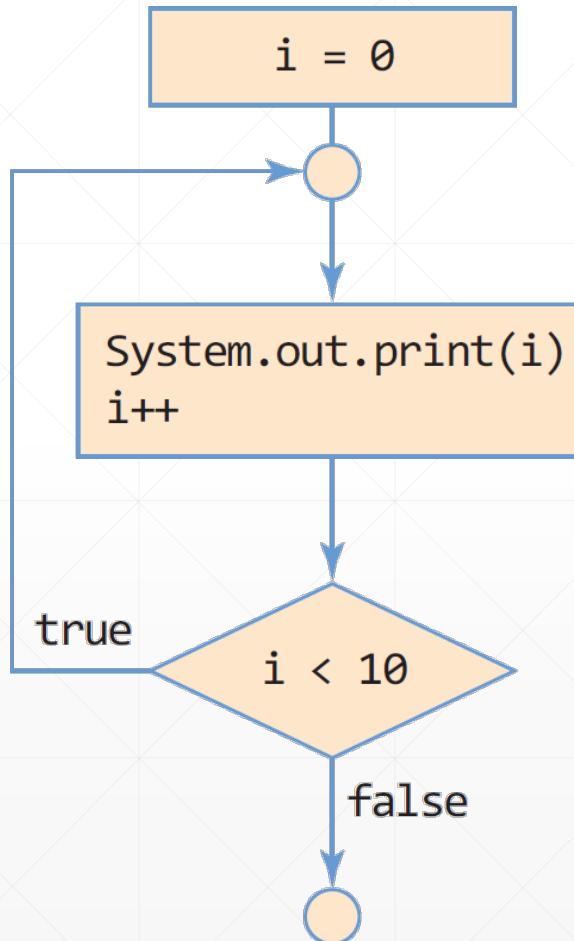
- Do-while evaluates its expression at the bottom of the loop instead of the top!
- The statements within the do block are always executed **at least once**

```
Do {  
    ... statement(s) ...  
} while (condition);
```



Loop: Do-While (cont'd)

■ Example)



```
i = 0;  
do {  
    System.out.print(i);  
    i++;  
} while(i<10);
```

0123456789

Loop: Summary

```
for(initialization; condition ; increment)  
{  
    statements  
}
```

```
while( condition )  
{  
    statements  
}
```

```
do  
{  
    statements  
} while( condition );
```

white
do - white] sequence
] $\frac{1}{2}$ n.

- white \rightarrow condition \neq \perp
statement \perp \perp
- do - white \rightarrow statement \neq \perp
 \perp \neq condition \perp .

Loop: Nested Loop

- Similar to the nested if-statement, loop statements can be used in a nested way
 - i.e., Loop inside another loop

```
Get in a department store;  
while (any interesting store??){  
    Get in the store;  
    Look around the store;  
    while (any interesting toy?){  
        Take a closer look at the toy;  
        if (Love it?) Buy it!;  
        Move to another toy;  
    }  
    Get out of the store;  
}  
Leave the department store;
```



Loop: Nested Loop (cont'd)

- Similar to the nested if-statement, loop statements can be used in a nested way
 - i.e., Loop inside another loop

```
for(int i=0; i<100; i++) { // sum up the scores of 100 schools  
    ....  
    for(int j=0; j<10000; j++) { // sum up the scores of 10,000 students, for each school  
        ....  
        ....  
    }  
    ....  
}
```

Doubly nested loop to add up the scores of 100 schools,
each of which has 10,000 students.

Loop: Nested Loop (cont'd)

- Example) print a multiplication table using double nested loop

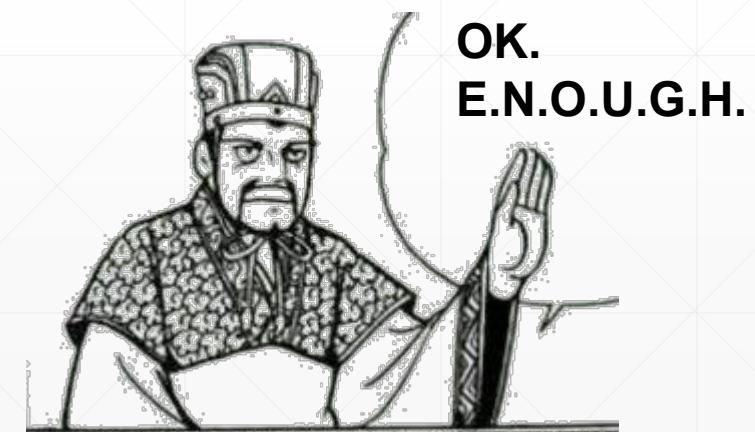
```
public class NestedLoop {  
    public static void main(String[] args) {  
        for(int i=1; i<10; i++) { // from 1 times table to 9 times table  
            for(int j=1; j<10; j++) { // for each table  
                System.out.print(i + "*" + j + "=" + i*j); // print multiplication  
                System.out.print('\t'); // print tab  
            }  
            System.out.println(); // nextline  
        }  
    }  
}
```

1*1=1	1*2=2	1*3=3	1*4=4	1*5=5	1*6=6	1*7=7	1*8=8	1*9=9
2*1=2	2*2=4	2*3=6	2*4=8	2*5=10	2*6=12	2*7=14	2*8=16	2*9=18
3*1=3	3*2=6	3*3=9	3*4=12	3*5=15	3*6=18	3*7=21	3*8=24	3*9=27
4*1=4	4*2=8	4*3=12	4*4=16	4*5=20	4*6=24	4*7=28	4*8=32	4*9=36
5*1=5	5*2=10	5*3=15	5*4=20	5*5=25	5*6=30	5*7=35	5*8=40	5*9=45
6*1=6	6*2=12	6*3=18	6*4=24	6*5=30	6*6=36	6*7=42	6*8=48	6*9=54
7*1=7	7*2=14	7*3=21	7*4=28	7*5=35	7*6=42	7*7=49	7*8=56	7*9=63
8*1=8	8*2=16	8*3=24	8*4=32	8*5=40	8*6=48	8*7=56	8*8=64	8*9=72
9*1=9	9*2=18	9*3=27	9*4=36	9*5=45	9*6=54	9*7=63	9*8=72	9*9=81

Loop: Continue & Break

■ Oh...Please...

Get in a department store;
while (any interesting store??){
 Get in the store;
 Look around the store;
 while (any interesting toy?) {
 Take a closer look at the toy;
 if (Love it?) Buy it!;
 Move to another toy;
 }
 }
 Get out of the store;
}
STOP IT!!!!!!!
Leave the department store;



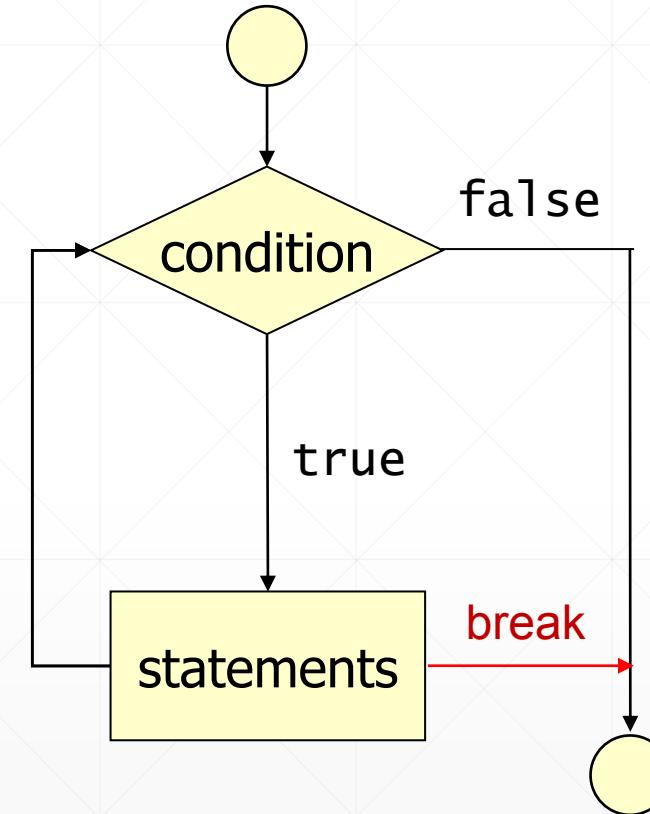
Loop: Continue & Break (cont'd)

■ Break statement

- Can be used to break the loop!
- Only applied to the current loop

■ Example)

```
for(int i=0;i<5;i++){  
    if(i==3) break;  
    System.out.println(i);  
}  
  
int j=0;  
while(j<5){  
    if(j==3) break;  
    System.out.println(j);  
    j++;  
}
```



Loop: Continue & Break (cont'd)

■ Break in the nested loop

- Only applied to the current loop

■ Example)

```
while(true){  
    while(true){  
        break;           ← Infinite loop!  
    }  
    System.out.println("Here!");   ← here?  
}  
System.out.println("There!");   ← Where should we go?
```

Loop: Continue & Break (cont'd)

■ Oh...Please...

Get in a department store;
while (any interesting store??){
 Get in the store;
 Look around the store;

 Hmm, not interesting! **while** (any interesting toy?) {
 Take a closer look at the toy;
 if (Love it?) Buy it!;
 Move to another toy;
 }
 Get out of the store;
}
Leave the department store;



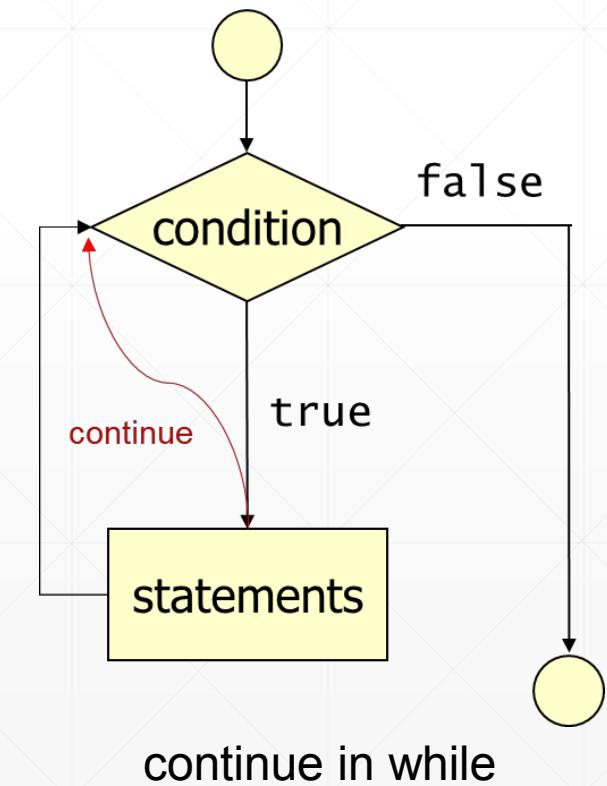
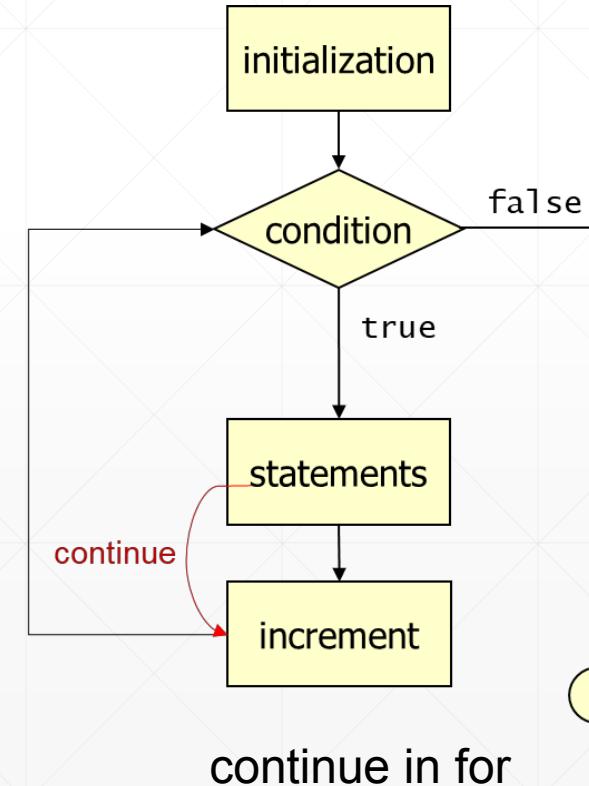
Loop: Continue & Break (cont'd)

■ Continue statement

- Skip the remaining statements
- Move to check the condition of the loop
- Only applied to the current loop

■ Example)

```
for(int i=0;i<5;i++){  
    if(i%2==0) continue;  
    System.out.println(i);  
}  
  
int j=0;  
while(j<5){  
    if(j%2==0) { j++; continue; }  
    System.out.println(j);  
    j++;  
}
```



Loop: Continue & Break (cont'd)

■ Continue in the nested loop

- Only applied to the current loop

■ Example) int i,j=0;

```
int i,j=0;

while (j<10) {
    while (j<5) {
        System.out.println("first!");
        j++;
        if (j %2 == 1) continue;
        System.out.println("second!");
    }
    System.out.println("third!");
    j++;
}
System.out.println("fourth");
```

0	f
1	f
2	Sf
3	f
4	Sf
5	t
6	t
7	t
8	t
9	t
10	fourth

Q&A

■ Next week (eClass video)

- Reference Types

Computer Language

Reference Type

Agenda

■ Reference Type

- Basics
- Array

primitive type VS reference type

String & enumeration

Basics

Array

Reference Type

■ Data type

➤ Primitive type

- Integer (byte, char, short, int, long)
- Floating-point (float, double)
- Boolean

difference in capabilities (size)

1 byte 2 byte 2 byte 4 byte 8 byte

➤ Reference type

- Array
- Enum ↗enumeration
- Class (+String, Wrapper)
- Interface

Reference Type (cont'd)

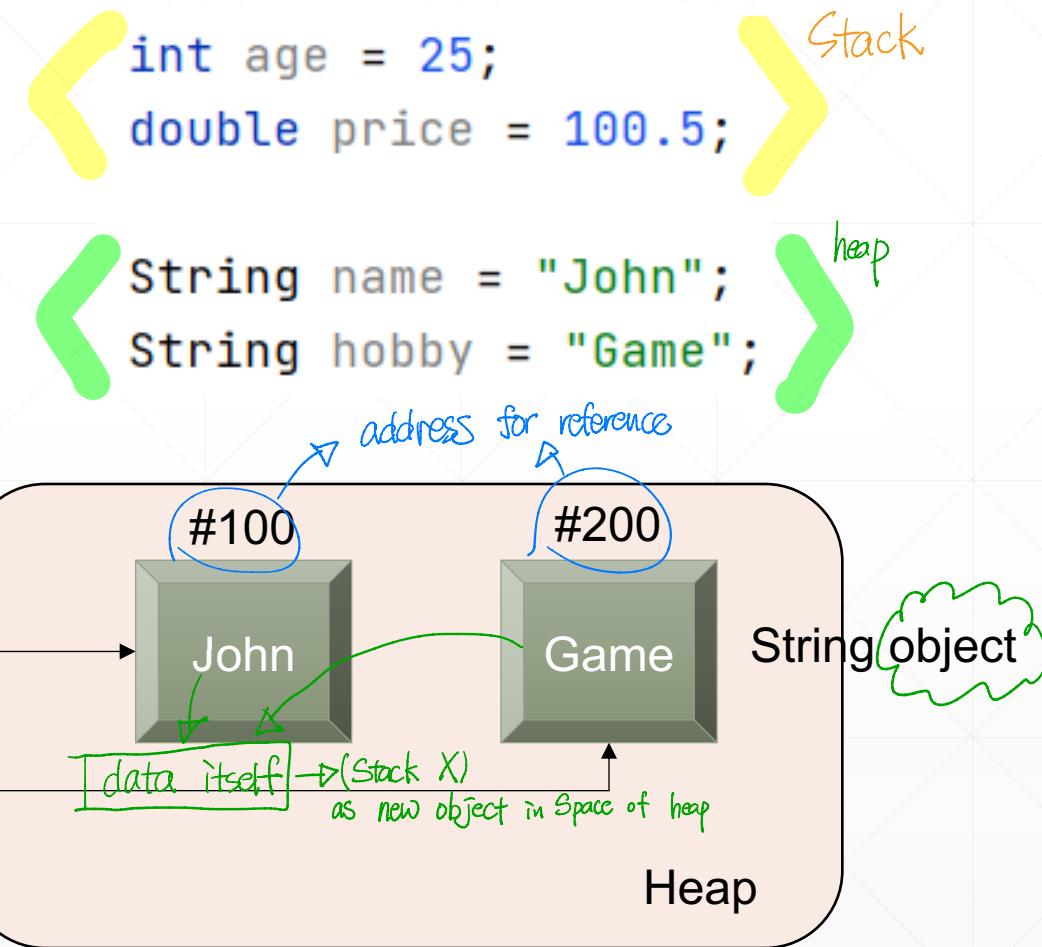
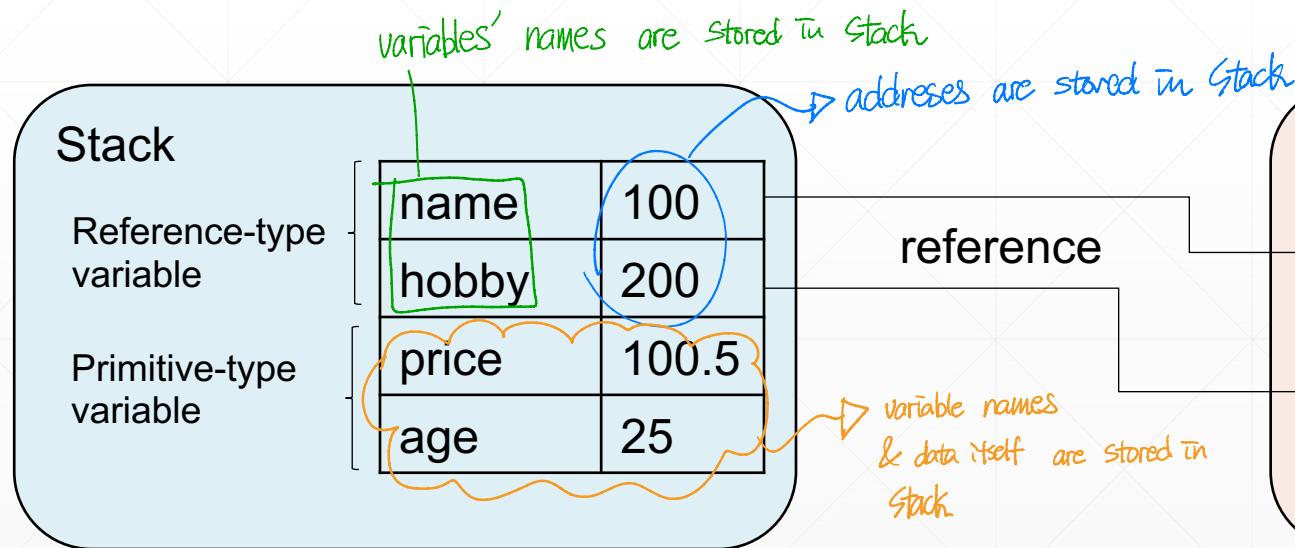
■ Data type

➤ Primitive type variable

- Store a value in the variable (stack)

➤ Reference type variable NP More complex structures are inserted

- Store a value in the memory (heap)
- Store an address of the memory for further reference



Reference Type (cont'd)

■ Stack < thread-based, temporary, short-term memory >

- Allocated for each thread
- Temporary memory based on the method call
- Memory blocks for the method are stacked
 - Local variables, references, etc
 - Only accessible in the block

because...

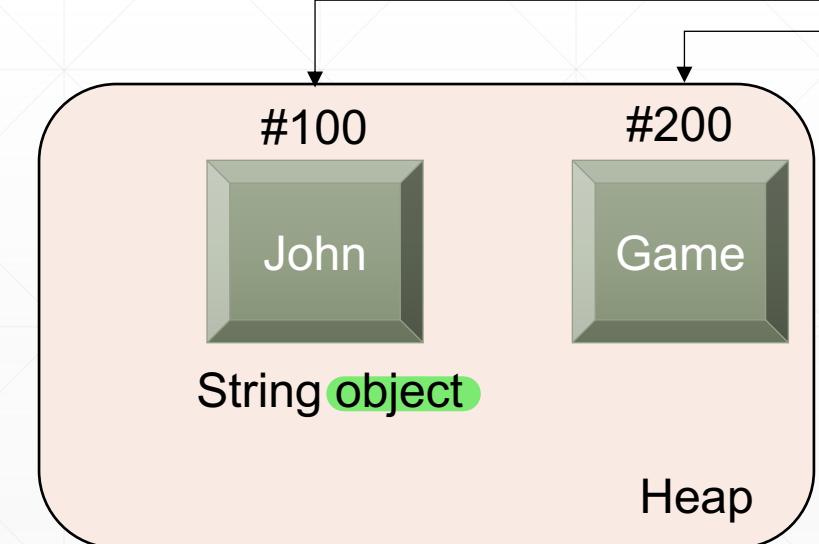
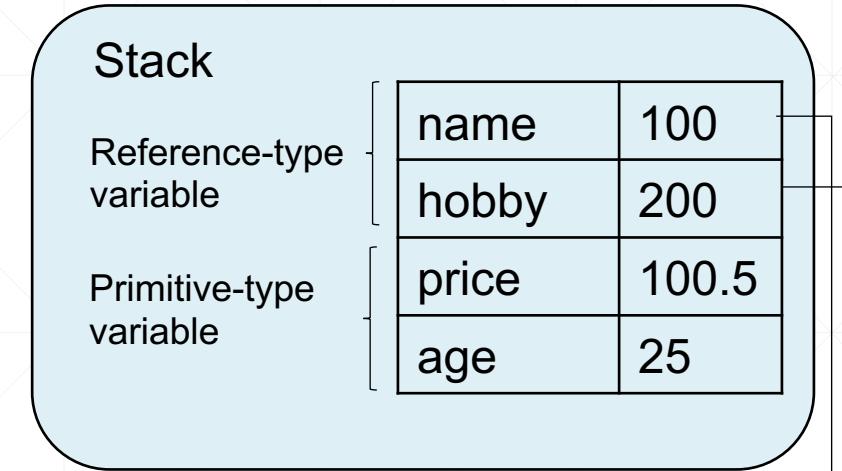
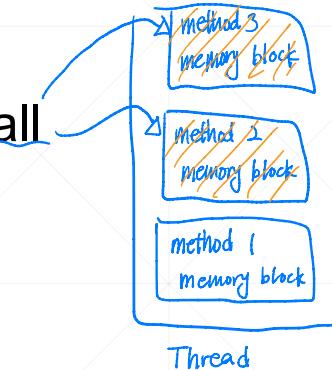
- After the method is finished, the memory space for the data is cleared ↳ method calling finish ⇒ discard memory block

⇒ More spaces for stack are preserved in thread

"Stack overflow" error ⇒ needed memory block for method calling > size of stack in thread
... request too much memories

■ Heap

- Created by JVM
- Uses as long as the application is running
- All objects are stored in a heap with a global access, therefore, can be referenced from anywhere in the app< long-term, global, sharable memory >



Reference Type (cont'd)

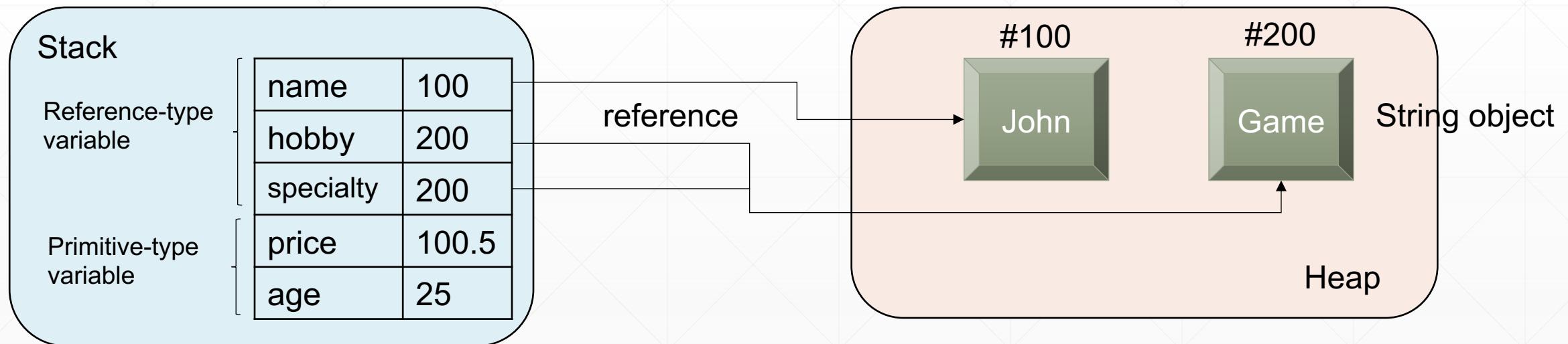
■ ==, != operations

➤ Primitive type variable

- Check if two values are same or not

➤ Reference type variable

- Check if two variables are pointing the same reference or not
in terms of memory, not data itself.

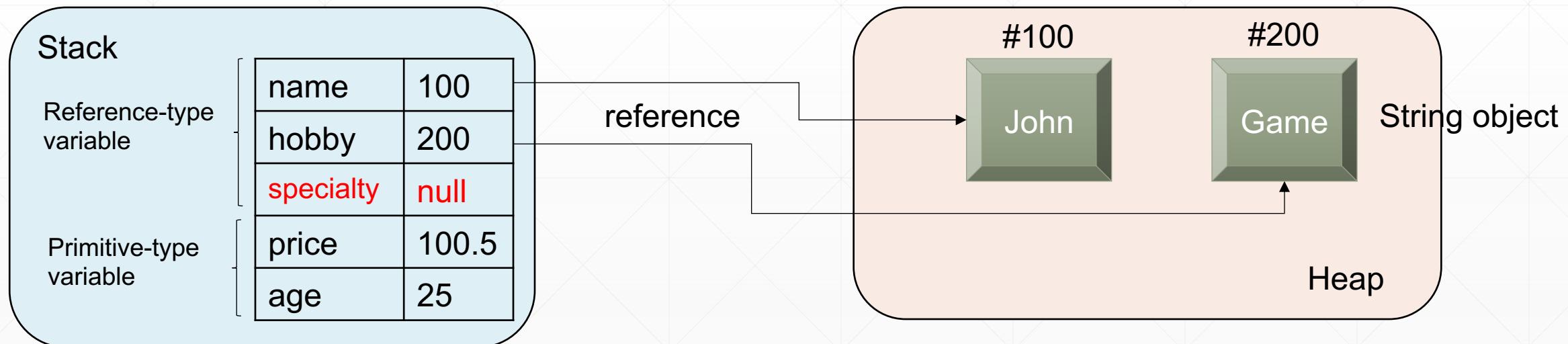


Reference Type (cont'd)

null

- Can be used for a reference type only!
- Can be used as a default value when a reference-type variable does not point anything
- !=, == operations can be used for null type

name == null (false)
specialty == null (true)



Reference Type (cont'd)

■ NullPointerException (NPE)

- One of Exceptions (will be discussed later)
 - Program error
- When if we try to use variables/methods of null

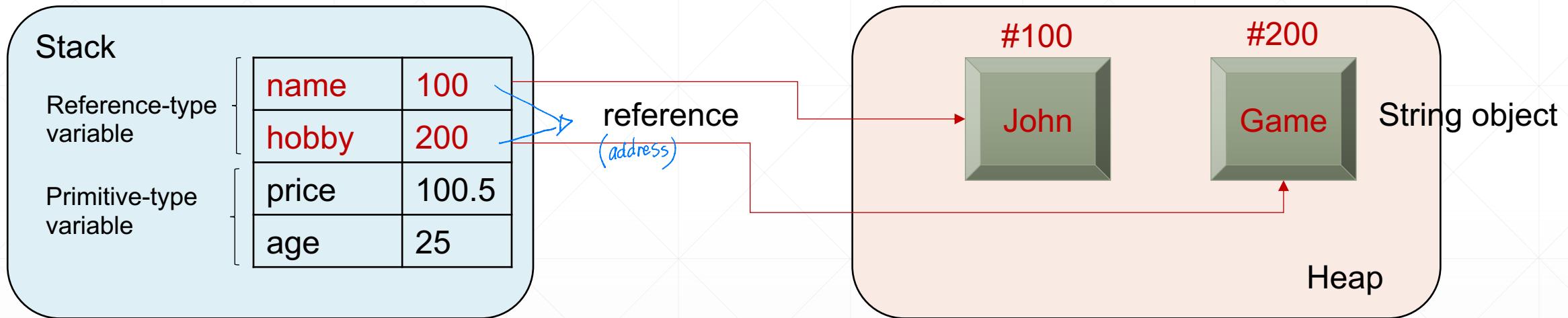
```
int[] intArray = null;           No memory space for 'intArray'  
intArray[0] = 10;               //NullPointerException  
  
String str = null;             impossible to be allocated to intArray  
                                [No memory!  
  
System.out.println(" Length: " + str.length()); //NullPointerException
```

method, feature, function of "null" is impossible.
using feature of "nothing" is impossible.

Reference Type (cont'd)

■ String

- Class to store a string value

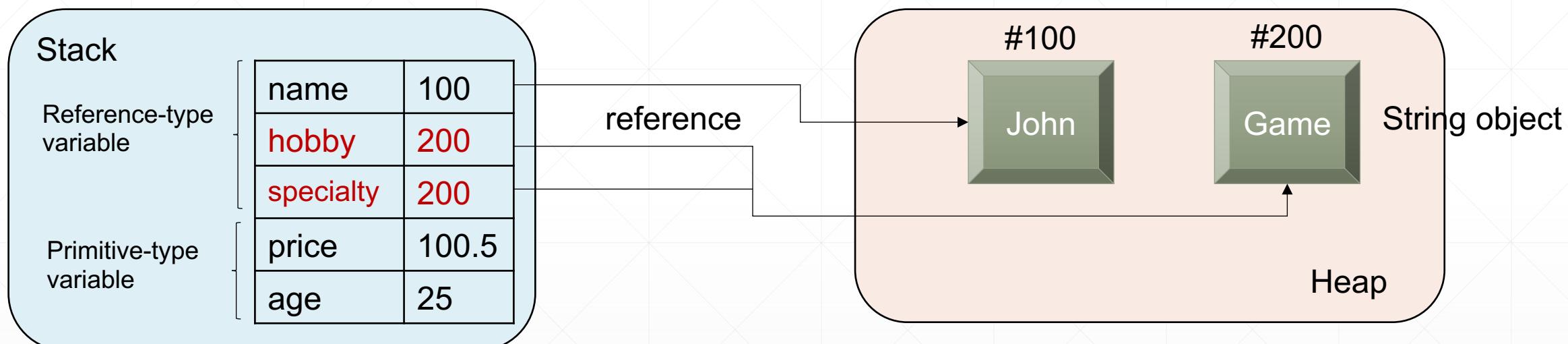


- Generation of **String object** using “**new**” keyword
 - Generate a new String object in Heap area
 - The address of the memory is returned

Reference Type (cont'd)

■ String

- String object can be shared if string literals are same



```
String hobby = "Game";
String specialty = "Game";
```

Reference Type (cont'd)

■ Enumeration

- Special data type to store a set of constants
- Common example
 - Representing compass directions: {NORTH, SOUTH, EAST, WEST}
 - Representing the days of a week: {SUNDAY, MONDAY, TUESDAY, ..., SATURDAY}
- Enum-type variable must be equal to one of the values that have been predefined for it
- Declaration `public enum Enumtype { ... (a set of enum constants) }`
 - Need to be declared in the java file with the same Enumtype name
 - **Enum constant should be CAPITAL** (naming convention)

```
public enum Week { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, ... }
```

```
public enum LoginResult { LOGIN_SUCCESS, LOGIN_FAILED }
```

Reference Type (cont'd)

■ Enumeration

- Declaration of Enum type variable

```
Enumtype variableName;
```

```
Week today;
```

```
Week reservationDay;
```

- Assigning a value to Enum type variable

- Value must be equal to one of the values that have been predefined for it

```
Enumtype variableName = Enumtype.constant;
```

```
Week today = Week.SUNDAY;
```

- Enum type is a kind of reference type

- Enum-type variable can use null literal

```
Week birthday = null;
```

Reference Type (cont'd)

■ Example)

Hello.java

```
public class Hello {  
    public static void main(String[] args) {  
  
        Week myDay = Week.FRIDAY;  
  
        switch (myDay) {  
            case MONDAY:  
                System.out.println("Mondays are bad.");  
                break;  
            case FRIDAY:  
                System.out.println("Fridays are better.");  
                break;  
            case SATURDAY: case SUNDAY:  
                System.out.println("Weekends are best.");  
                break;  
            default:  
                System.out.println("Midweek days are so-so.");  
                break;  
        }  
    }  
}
```

Week.java

```
public enum Week {  
    MONDAY,  
    TUESDAY,  
    WEDNESDAY,  
    THURSDAY,  
    FRIDAY,  
    SATURDAY,  
    SUNDAY  
}
```

Basics
Array

When We need an Array?

- We want to calculate the grades for 5 students

- Store each student's score, then
 - Calculate mean, min, max of all students' scores!

- We need five variables to do this!

- score1
 - score2
 - score3
 - score4
 - score5

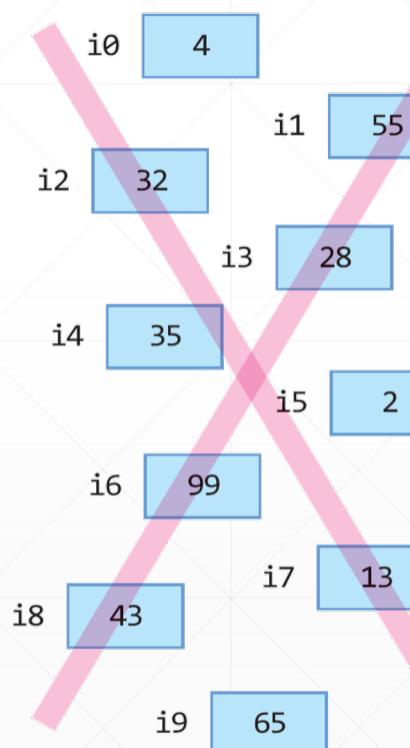
When We need an Array? (cont'd)

- What about 50 students
 - Store each student's score, then
 - Calculate mean, min, max of all students' scores!
- We need fifty variables to do this...?
 - score1, score2, score3, score4, score5 ... score50?
 - Ok..
- What about 500 students?
 - We need five hundred variables then?
- So, we need a data structure to store a list of elements! (like, array!)

Array: Characteristics

- Data structure used to store multiple values in a single variable, instead of declaring separate variables for each value
- A container object that holds a **fixed number of values** of a **single type**

```
int i0, i1, i2, i3, i4, i5, i6, i7, i8, i9;
```



```
sum = i0+i1+i2+i3+i4+i5+i6+i7+i8+i9;
```

single type
fixed number
int i[] = new *int*[10];

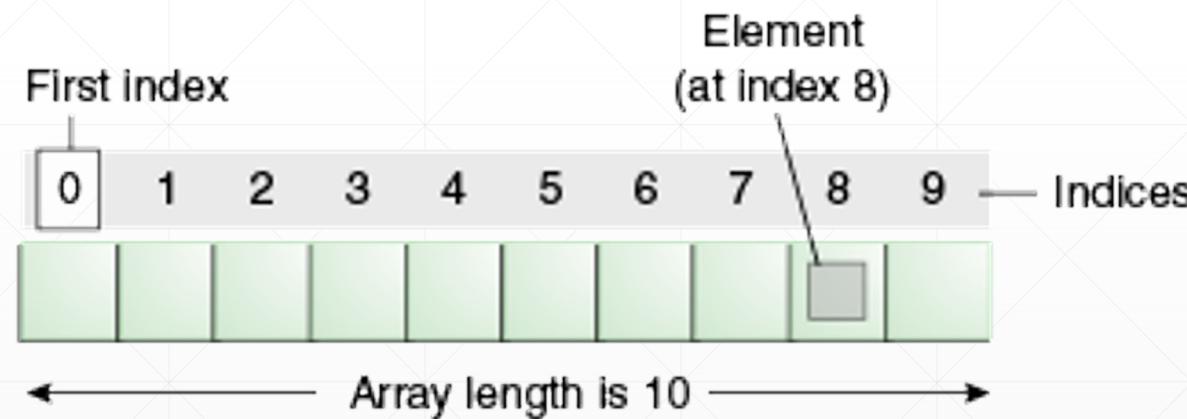


```
for(sum=0, n=0; n<10; n++)  
    sum += i[n];
```

Array: <Index> is address ?

■ Element

- Each item in an array
- Accessed by its numerical index
- Index number begins with 0



Array: Declaration and Creation

■ Declaration

type[] variable;

```
int[] intArray;  
double[] doubleArray;  
String[] strArray;
```

type variable[];

```
int intArray[];  
double doubleArray[];  
String strArray[];
```

■ Array variable can be null

- If null, accessing array elements (i.e., array[index]) is not possible
 - NullPointerException NPE occurs!

Not exist

error

Array: Declaration and Creation (cont'd)

■ Declaration with initialization

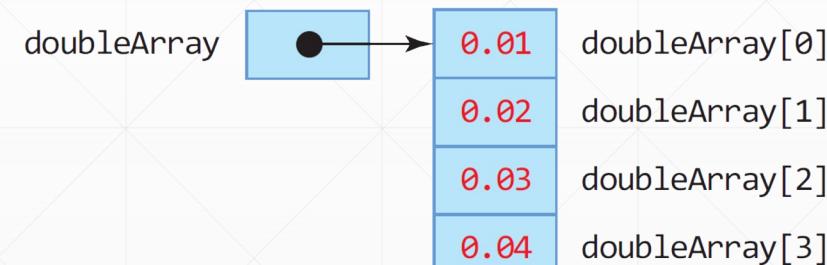
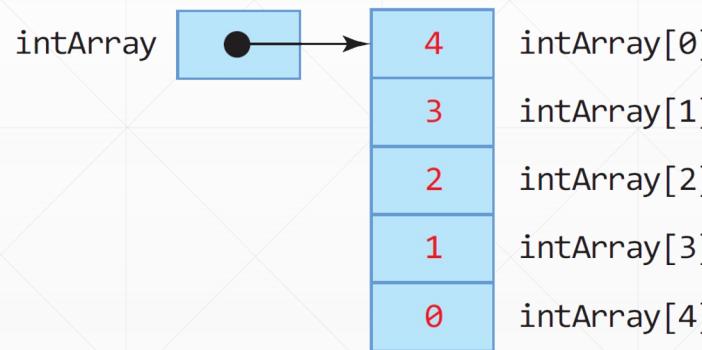
```
type[] variable = { value0, value1, value2, ...};
```

➤ Example)

```
int intArray[] = {4,3,2,1,0};  
double doubleArray[] = {0.01, 0.02, 0.03, 0.04};
```

```
int intArray[] = {4, 3, 2, 1, 0};
```

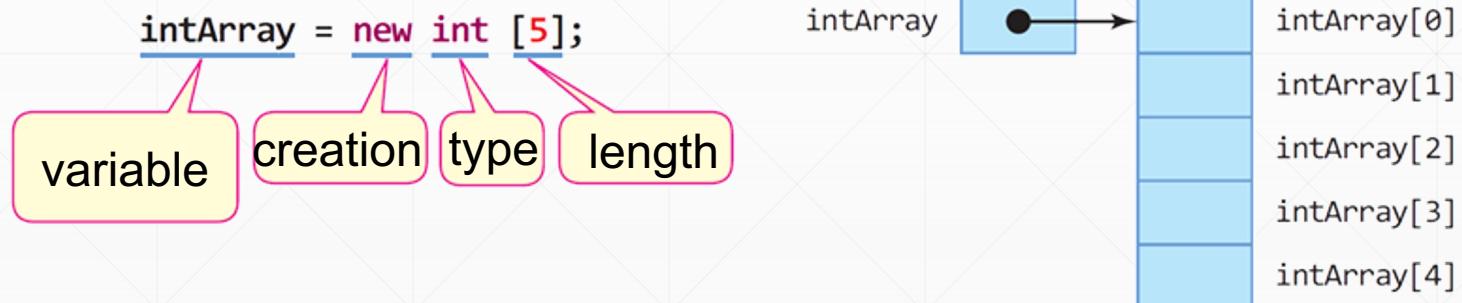
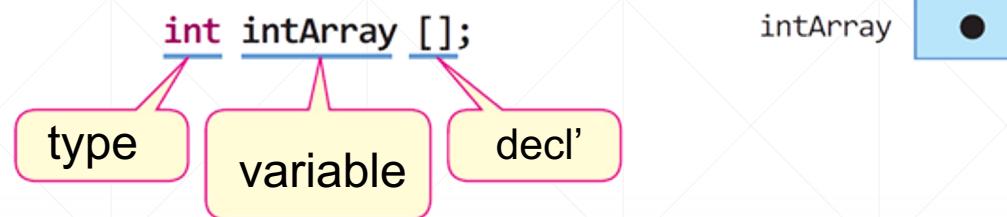
```
double doubleArray[] = {0.01, 0.02, 0.03, 0.04};
```



Array: Declaration and Creation (cont'd)

■ Creation after Declaration

```
type[] variable; // array declaration  
variable = new type[length]; // array creation (memory allocation)
```



Array: Access

■ Accessing array element

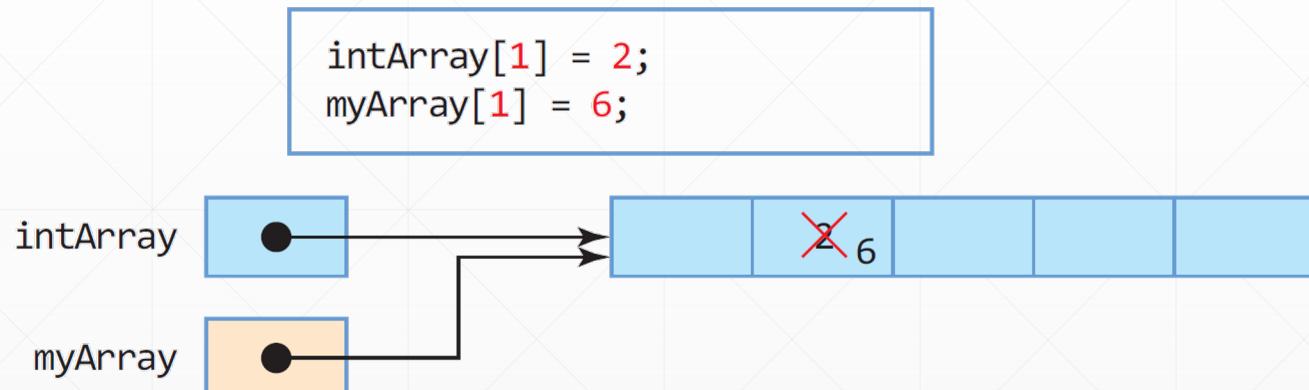
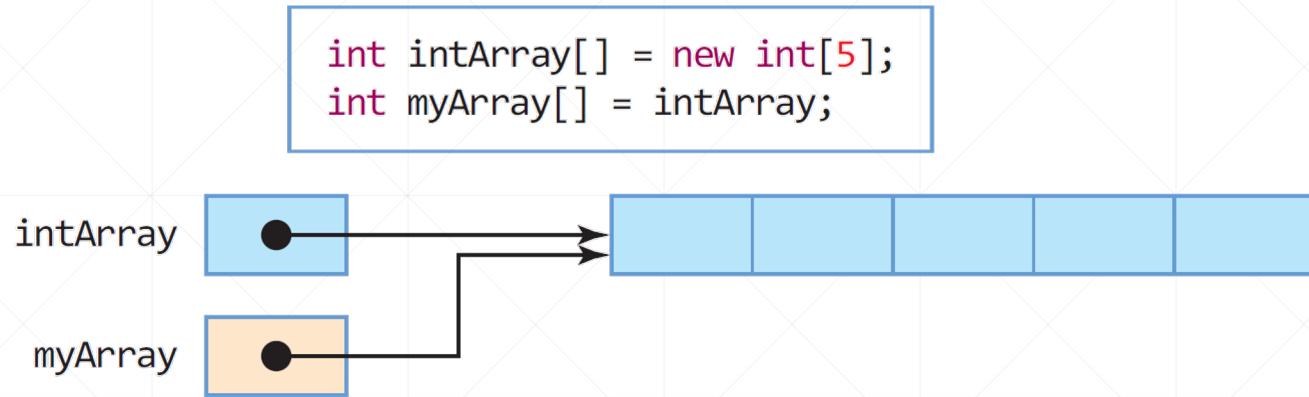
ArrVariable[index]

- Get the element located in the position of *index* in the array
- Index begins with 0
- The index of the last element in an array is (the length of the array -1)

```
int intArray [] = new int[5]; // create an array with size of 5 (index: 0~4)
intArray[0] = 5; // store 5 to index 0
intArray[3] = 6; // store 6 to index 3
int n = intArray[3]; // access index 3 of intArray and the assign the value to variable n
n = intArray[-2]; // error!
n = intArray[5]; // error!
```

Array: Access (cont'd)

- A single array can be shared with multiple references



Array: Access (cont'd)

- Example) take 5 positive integers from the user, store them in an array, and print the max value!

```
Scanner scanner = new Scanner(System.in);

int intArray[] = new int[5];    // create an array

int max = 0;      // current max value
System.out.println("Input 5 Positive Numbers.");
for (int i = 0; i < 5; i++) {
    intArray[i] = scanner.nextInt();    // store the input value to the array
    if (intArray[i] > max)        // if intArray[i] is greater than the current max
        max = intArray[i];        // then set intArray[i] as current max
}
System.out.print("The maximum value is " + max + ".");

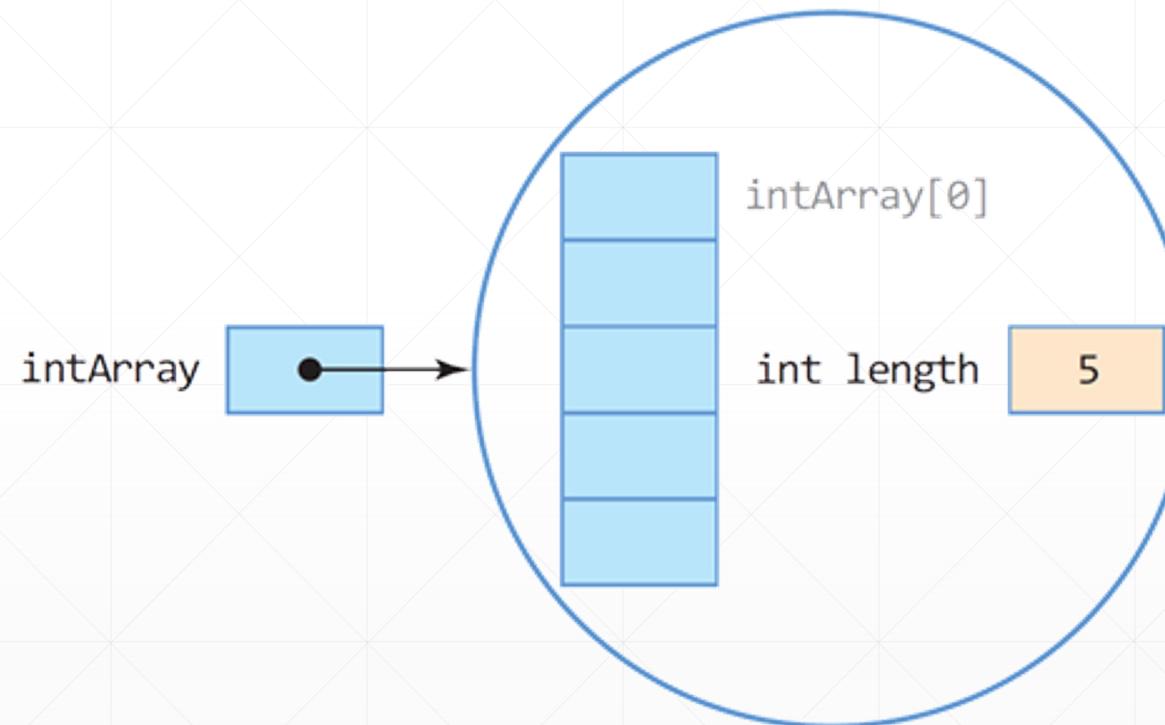
scanner.close();
```

Array: Length

- Array is a kind of Object in Java

- Length field of an array represents the size of the array

```
int intArray[];  
intArray = new int[5];  
  
int size = intArray.length;
```



Array: Length

■ Example)

- Take a set of integers from the user to fill out the array
- Use the length of an array to determine how many times a user needs to type the number!

```
int intArray[] = new int[5];
int sum = 0;

Scanner scanner = new Scanner(System.in);
System.out.println("Input " + intArray.length + " numbers:");
for (int i = 0; i < intArray.length; i++)
    intArray[i] = scanner.nextInt(); // store an integer value to the array

for (int i = 0; i < intArray.length; i++)
    sum += intArray[i]; // sum up all the values in the array using for statement

System.out.print("Average is " + (double) sum / intArray.length);
scanner.close();
```

Array: For-Each

■ Advanced for statement to iterate each element in the array or enum

- Do not need to check the length of an array in the loop!

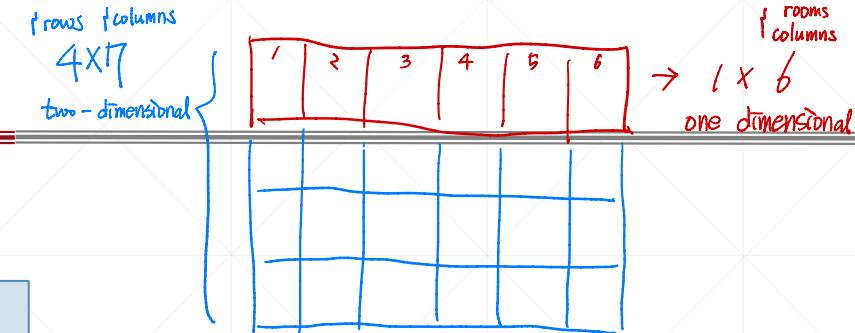
```
for (type variable : array)
```

```
int[] num = { 1,2,3,4,5 };
int sum = 0;
for (int k : num) // for each iteration, k is set to num[0], num[1], ..., num[4]
    sum += k;
System.out.println("Sum: " + sum);
```

```
String names[] = { "apple", "pear", "banana", "cherry", "strawberry", "grape" } ;
for (String s : names)
    System.out.print(s + " ");
```

```
enum Week { MON, TUE, WED, THU, FRI, SAT, SUN }
for (Week day : Week.values())
    System.out.print(day + " ");
```

Array: 2D-array



■ Declaration

2 brackets!

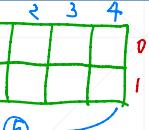
```
int intArray[][];
char charArray[][];
double doubleArray[][];
```

```
int[][] intArray;
char[][] charArray;
double[][] doubleArray;
```

■ Creation

rows columns

```
intArray = new int[2][5];
charArray = new char[5][5];
doubleArray = new double[5][2];
```

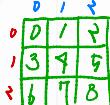


```
int intArray[][] = new int[2][5];
char charArray[][] = new char[5][5];
double doubleArray[][] = new double[5][2];
```

■ Declaration with initialization

3 rows

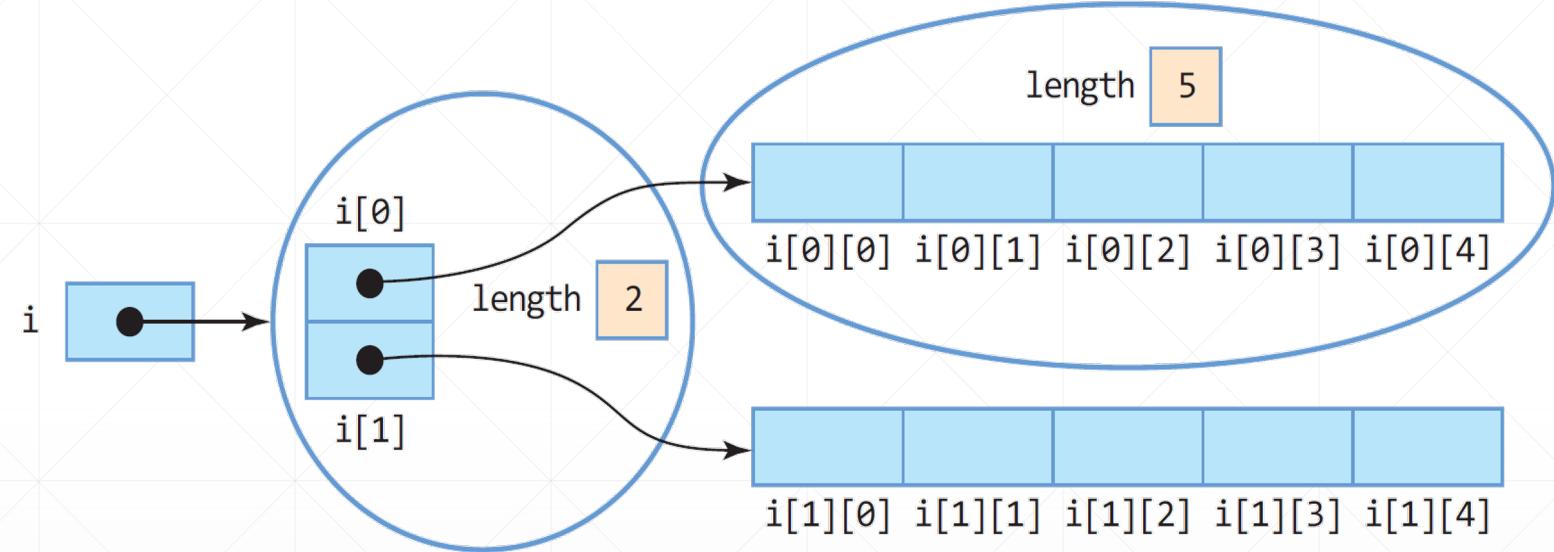
```
int intArray[][] = {{0,1,2},{3,4,5},{6,7,8}};
char charArray[][] = {{'a', 'b', 'c'},{'d', 'e', 'f'}}; 2 rows 3 columns.
double doubleArray[][] = {{0.01, 0.02}, {0.03, 0.04}}; 2 rows 2 columns
```



Array: 2D-array (cont'd)

■ Conceptual view

```
int i[][] = new int[2][5];
int size1 = i.length; // 2
int size2 = i[0].length; // 5
int size3 = i[1].length; // 5
```



■ The length of 2D-array

- `i.length = 2` (the number of row)
 - `i[0].length = 5` (length of n-th 1D-array)
 - `i[1].length = 5` (length of n-th 1D-array)

Array: 2D-array (cont'd)

- Store GPA scores for each year/semester in a 2d array, and then calculate their average

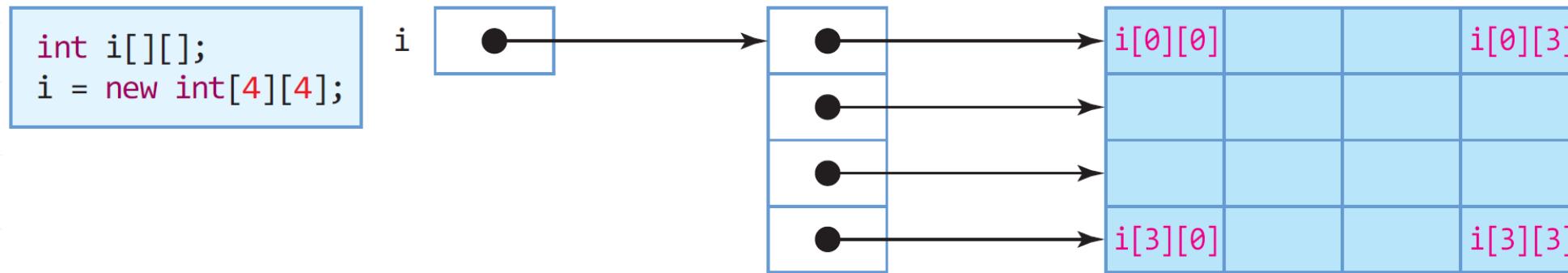
```
double score[][] = {{3.3, 3.4},      // GPA for the 1st year
                    {3.5, 3.6},      // GPA for the 2nd year
                    {3.7, 4.0},      // GPA for the 3rd year
                    {4.1, 4.2}};    // GPA for the 4th year

double sum = 0;
for (int year = 0; year < score.length; year++) // for each year
    for (int term = 0; term < score[year].length; term++) // for each semester
        sum += score[year][term]; // sum-up!

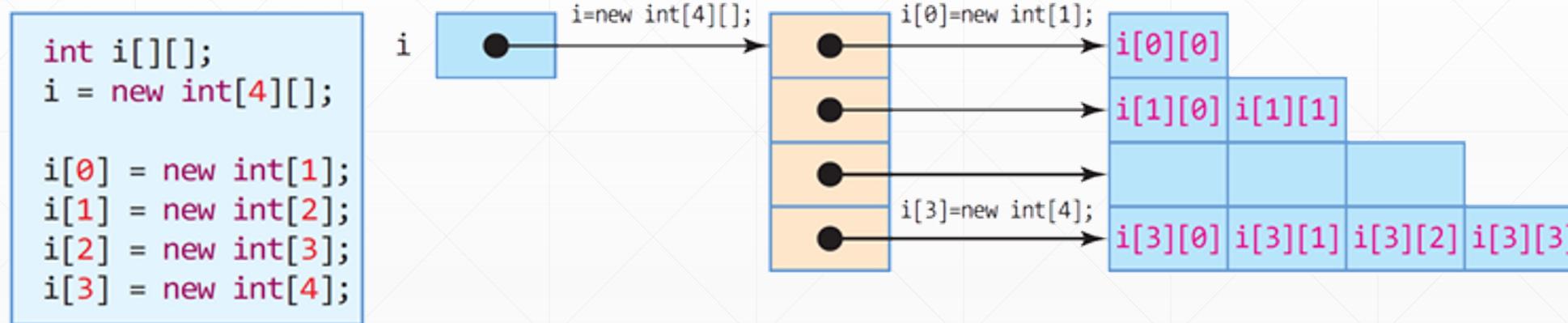
int n = score.length; // the number of rows
int m = score[0].length; // the number of columns
System.out.println("Total GPA is " + sum / (n * m));
```

Array: 2D-array (cont'd)

■ Square array

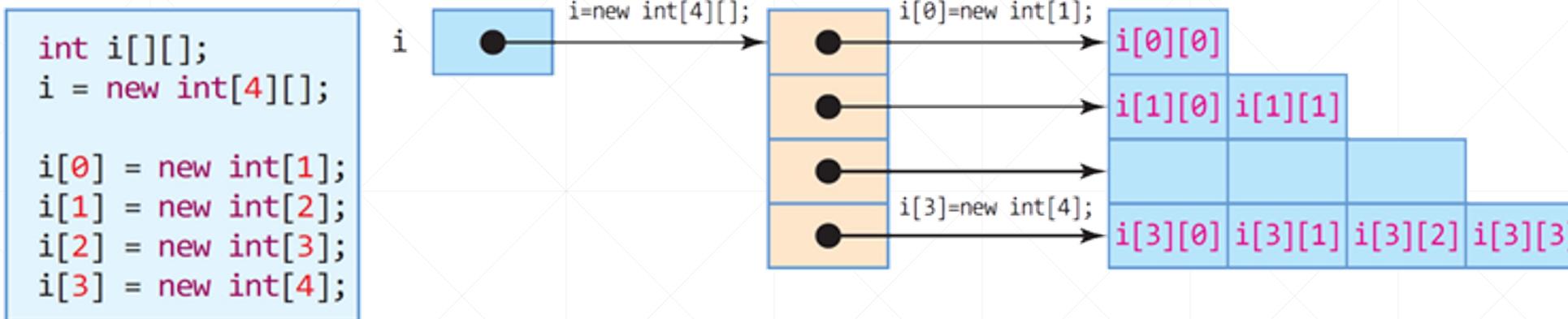


■ Non-square array



Array: 2D-array (cont'd)

■ Non-square array



■ The length of a non-square 2D-array

- `i.length = 4` (the number of row)
 - `i[0].length = 1` (length of n-th 1D-array)
 - `i[1].length = 2` (length of n-th 1D-array)
 - `i[2].length = 3` (length of n-th 1D-array)
 - `i[3].length = 4` (length of n-th 1D-array)

Array: 2D-array (cont'd)

- Example) Create a non-square array as shown in the following figure, initialize the array, and print it.

10	11	12
20	21	
30	31	32
40	41	

```
int intArray[][] = new int[4][];
intArray[0] = new int[3];
intArray[1] = new int[2];
intArray[2] = new int[3];
intArray[3] = new int[2];

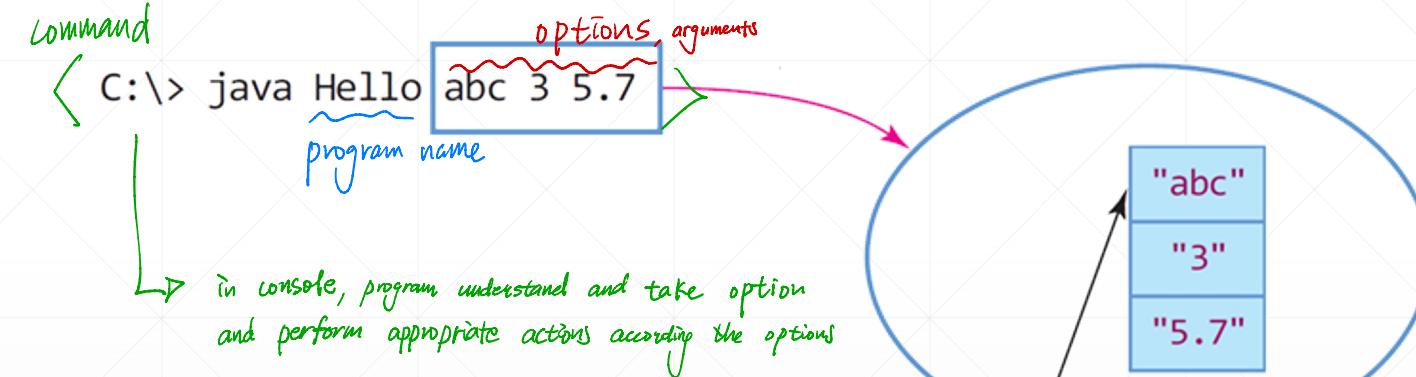
for (int i = 0; i < intArray.length; i++)
    for (int j = 0; j < intArray[i].length; j++)
        intArray[i][j] = (i + 1) * 10 + j;

for (int i = 0; i < intArray.length; i++) {
    for (int j = 0; j < intArray[i].length; j++)
        System.out.print(intArray[i][j] + " ");
    System.out.println();
}
```

Array: Misc.

without IDE, compile & run java file at console
java
java c g command required!

- What is `String[] args` in the main method?
- `main()` is the entry point of Java application
 - Arguments for starting Java application is passed through `args` String array



class Hello

public static void main(String[] args)

args.length => 3
args[0] => "abc"
args[1] => "3"
args[2] => "5.7"

Array: Misc. (cont'd)

IDE can pass some arguments when starting Java program.

- Example) print out the contents of args array in the main method

```
public class Hello {  
    public static void main(String[] args) {  
  
        for(String arg : args)  
            System.out.println(arg);  
  
    }  
}
```

- If no arguments are set, then nothing is printed

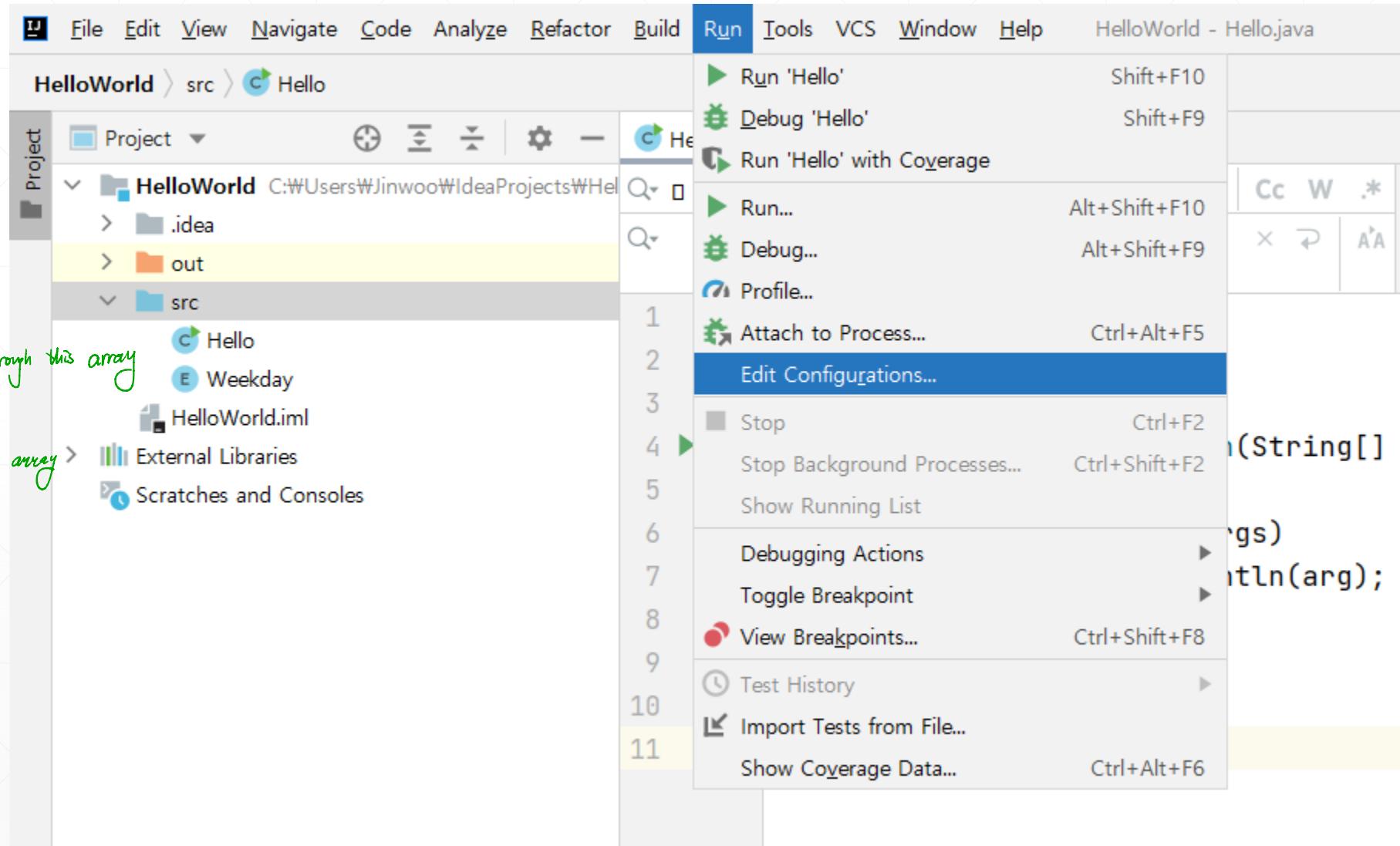
Array: Misc. (cont'd)

■ Edit Configuration

- Empty → no option
no arguments are passed
* Nothing print out

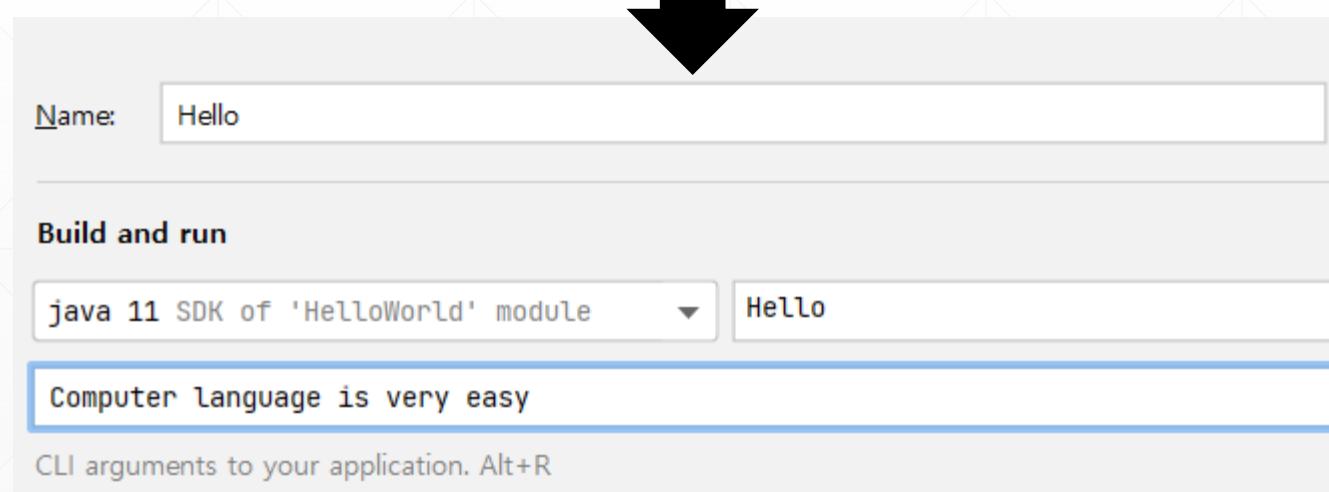
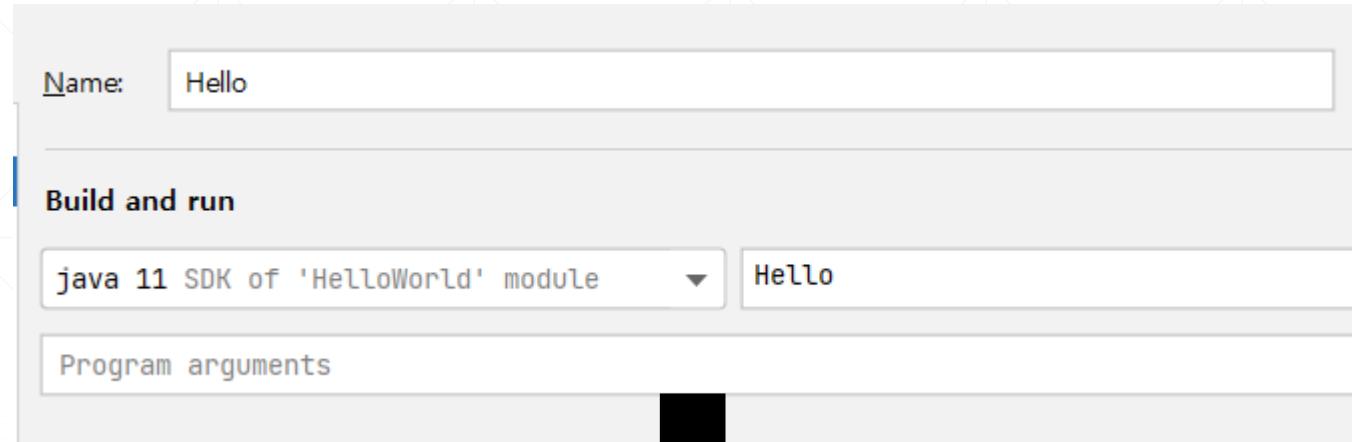
- put some arguments → passed to program through this array
in main method.

→ print each element of array



Array: Misc. (cont'd)

■ Edit Configuration: passing arguments



What happens?

Q&A

■ Next week (eClass video)

- OOD/P: Class and Methods

Object - oriented design/program.

Computer Language



OOP 1: Class



Agenda

- OOP
- Class

OOP

Class

OOP: Basics

■ Object-Oriented Programming (OOP)

- The world is composed of objects (thing)



- Characteristics of objects
 - Each object has its own state and behavior
 - Objects interact with each other

→ have own state or attributes or property

action



Uses “multiplication” feature

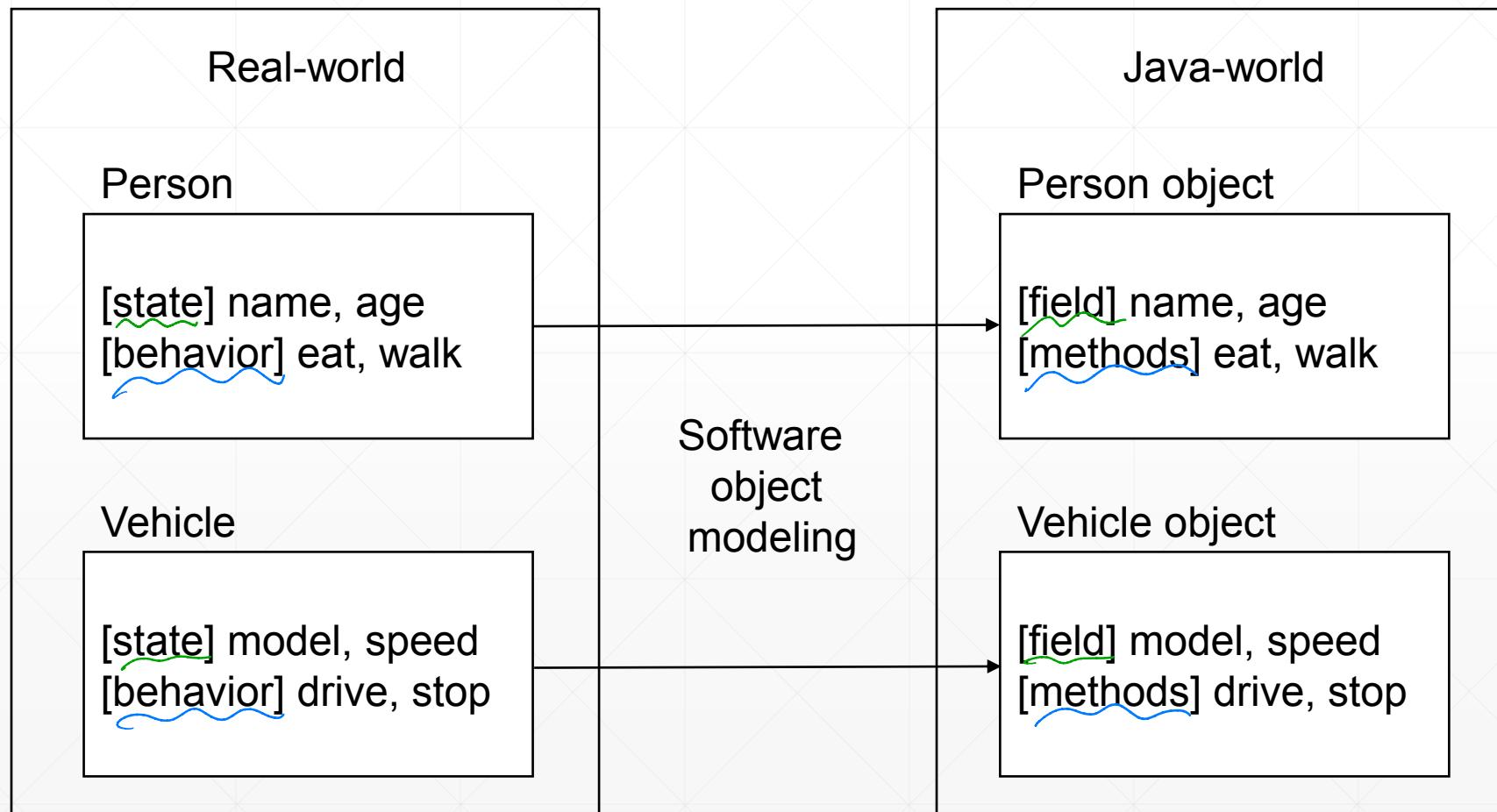


Returns the result

OOP: Basics (cont'd)

■ Objects in Java world

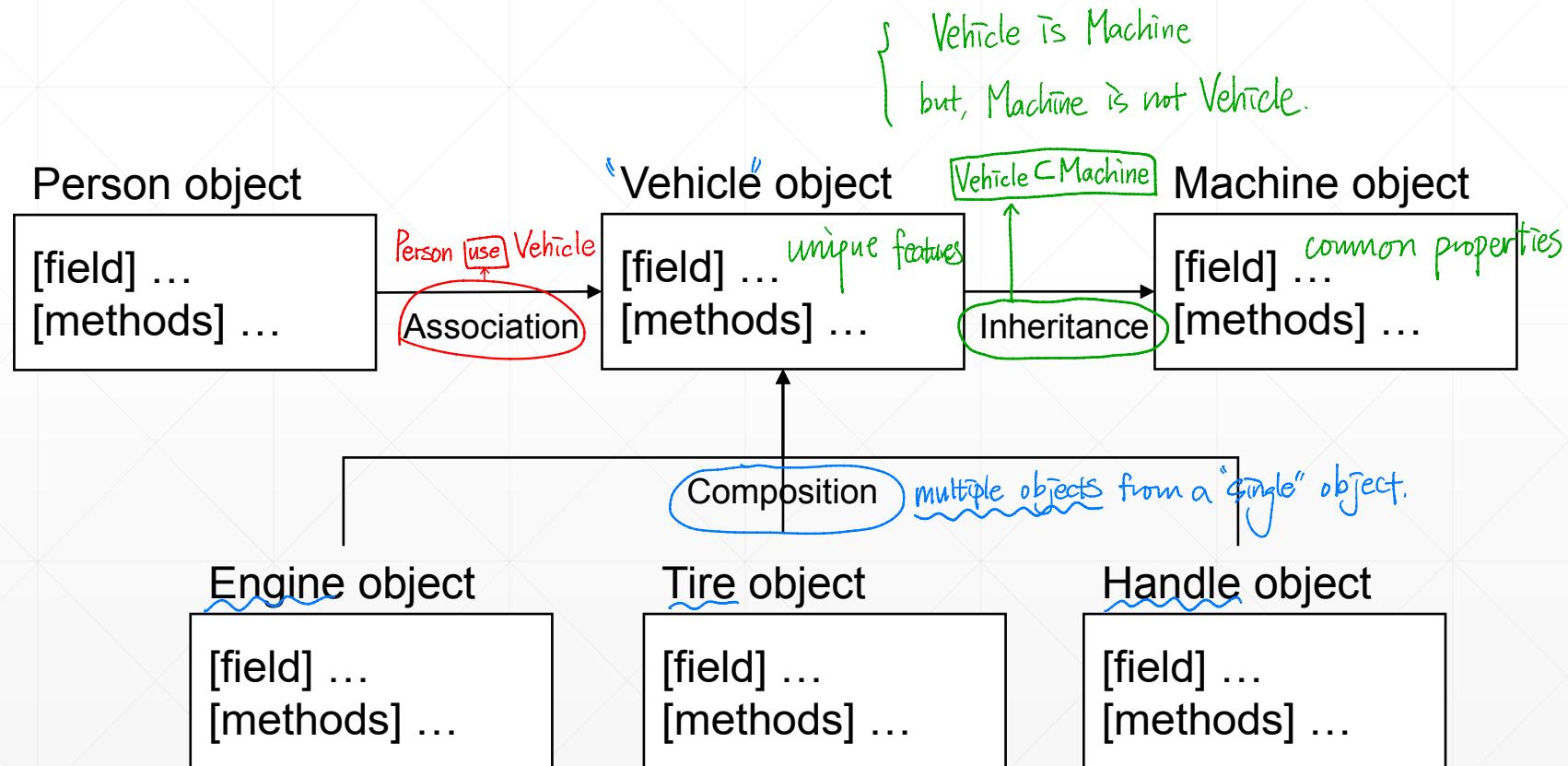
- Java object models the real-world object by defining fields (states) and methods (behaviors)



OOP: Basics (cont'd)

■ Objects in Java world

- Relationships between the objects
 - Association
 - Composition/Aggregation
 - Inheritance



OOP: Characteristics

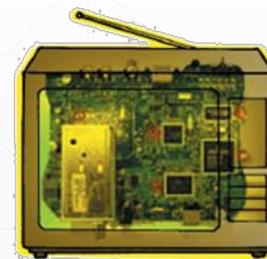
■ Encapsulation

External objects cannot provide internal details, structures.

- Information/Implementation hiding ↗ try to hide details inside of object
 - External objects cannot know the details (internal structure) of an object they want to use
 - External objects cannot access the private fields/methods of an object they want to use
 - External objects can only access the fields and methods explicitly exposed by an object they want to use
- Access modifier is used to determine the visibility of class members (discussed later)



capsule



TV



refrigerator



camera



human

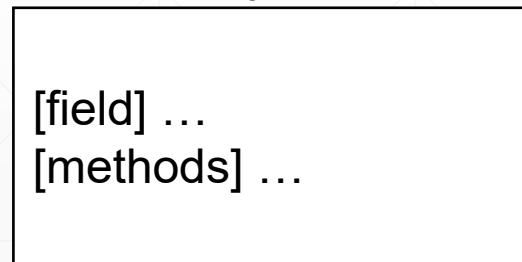
OOP: Characteristics (cont'd)

■ Encapsulation: Benefits

- Better control of class attributes and methods
- Flexible: the programmer can change one part of the code without affecting other parts
- Increased security of data
 - ↳ hide important parts.
 - ↳ access only public parts.

External object → interface → internal process → interface → External Object.

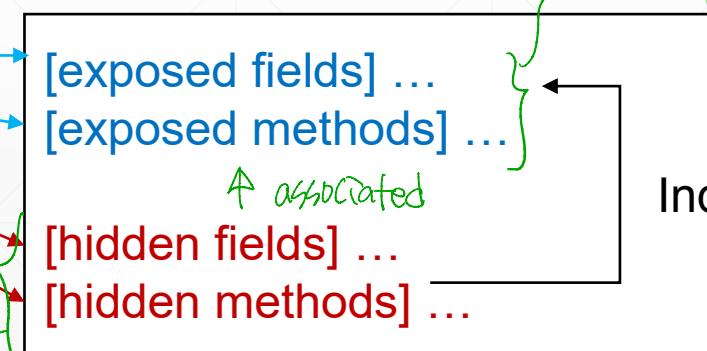
Person object



Can access

Cannot access

Vehicle object



External objects interact with exposed sectors
detailed implementation could be
accessed through exposed sector
indirectly.

impossible to change some codes.

because hidden sector could be still used in other part.

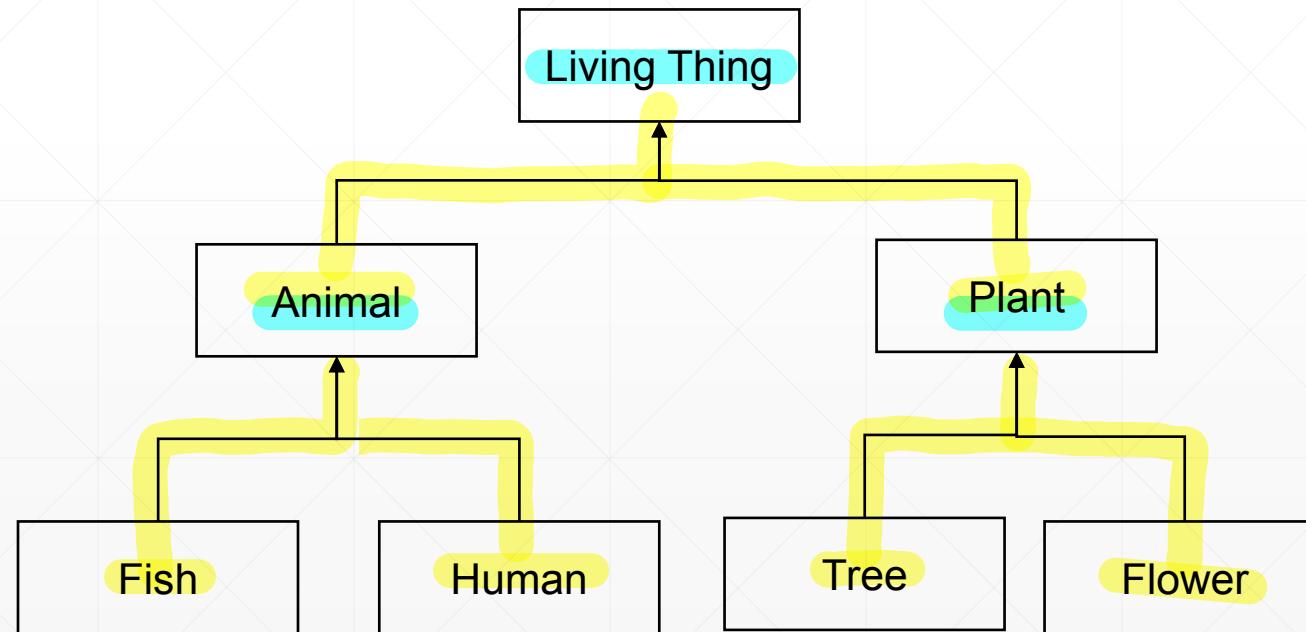
OOP: Characteristics (cont'd)

■ Inheritance

- It is possible to inherit attributes and methods from one class to another

- Subclass: a class derived from another class (child/derived/extended class)
- Superclass: the class from which the subclass is derived (parent/base class)

(common features in superclass
are given to subclasses.)

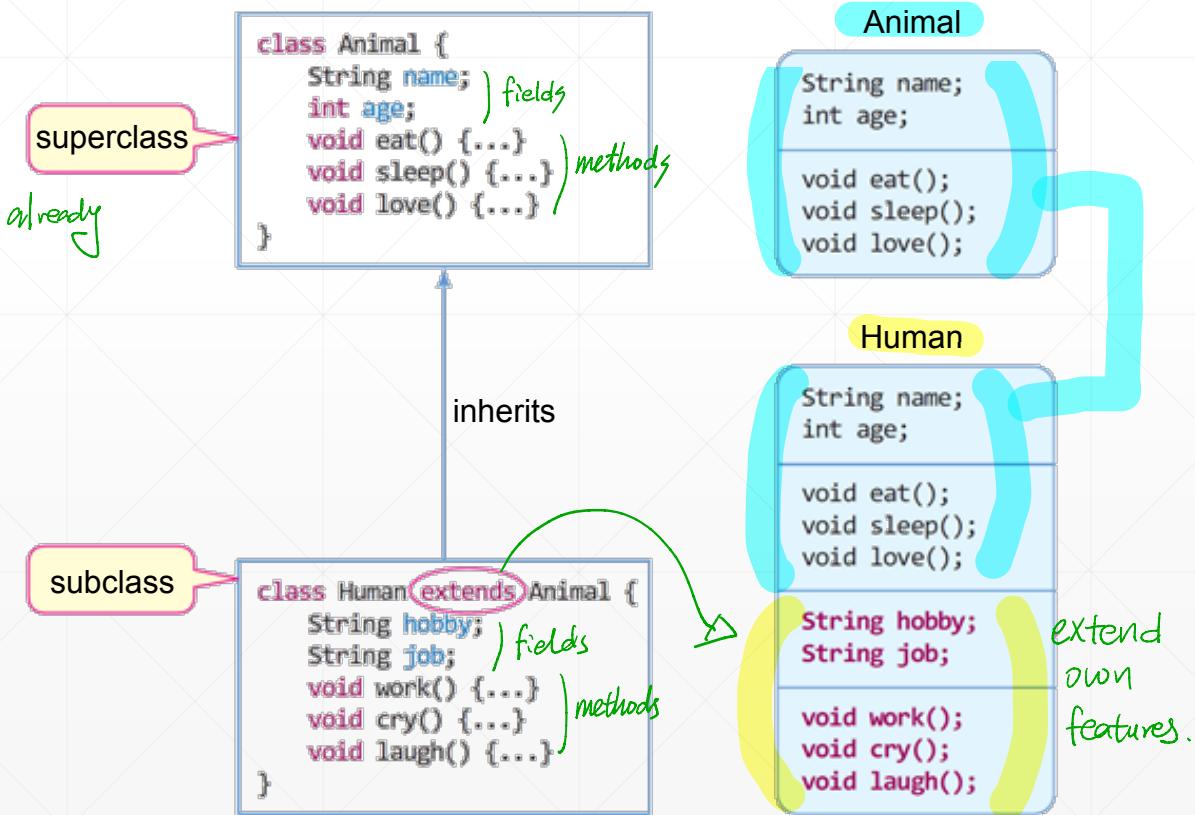


OOP: Characteristics (cont'd)

Inheritance

- Subclass inherits superclass's members and can extend its own features
- We can reuse the fields and methods of the existing class without having to re-write
- Benefits
 - Rapid implementation using existing classes
 - Reduced redundant codes ↗ existing common features already
 - Better, efficient maintenance
 - Polymorphism

State/ behavior
field/ method

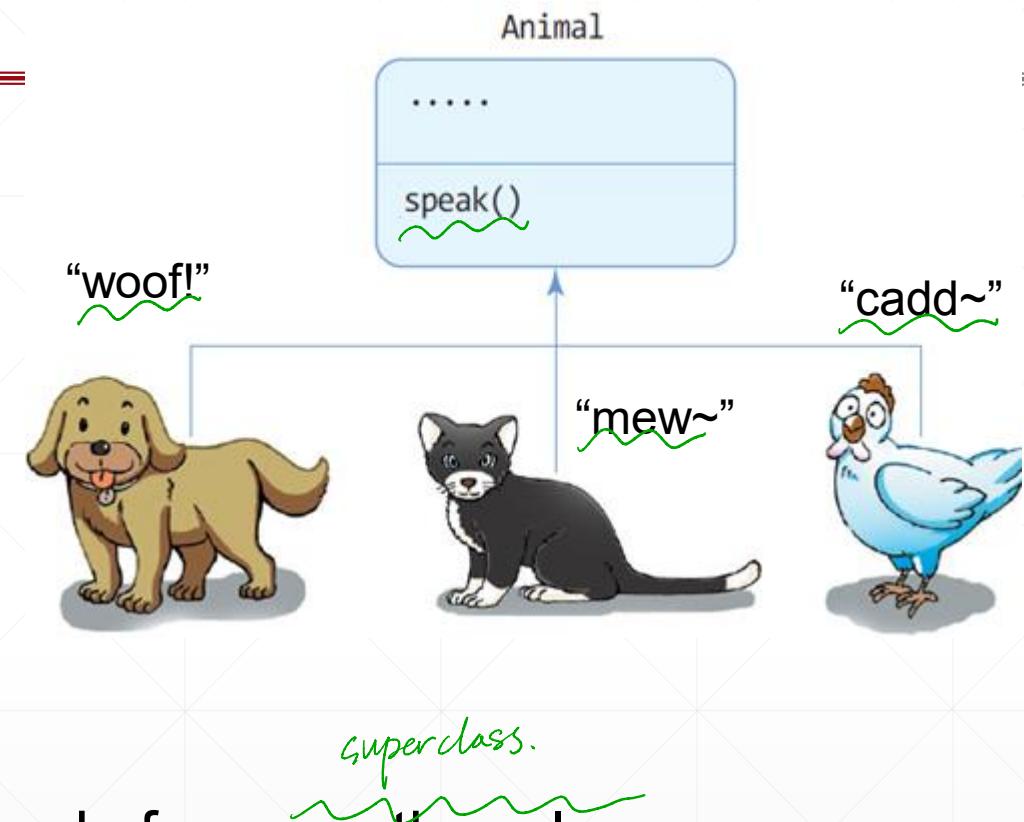


OOP: Characteristics (cont'd)

■ Polymorphism

- Definition: “having many forms”
 - “an organism or species can have many different forms or stages”
- Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class

→ Same function in different way!



■ Inheritance lets us inherit attributes and methods from another class

■ Polymorphism uses those methods to perform different tasks

■ Inheritance and Polymorphism allow us to perform a single action in different ways

- Benefits: flexible codes, better maintenance, etc.

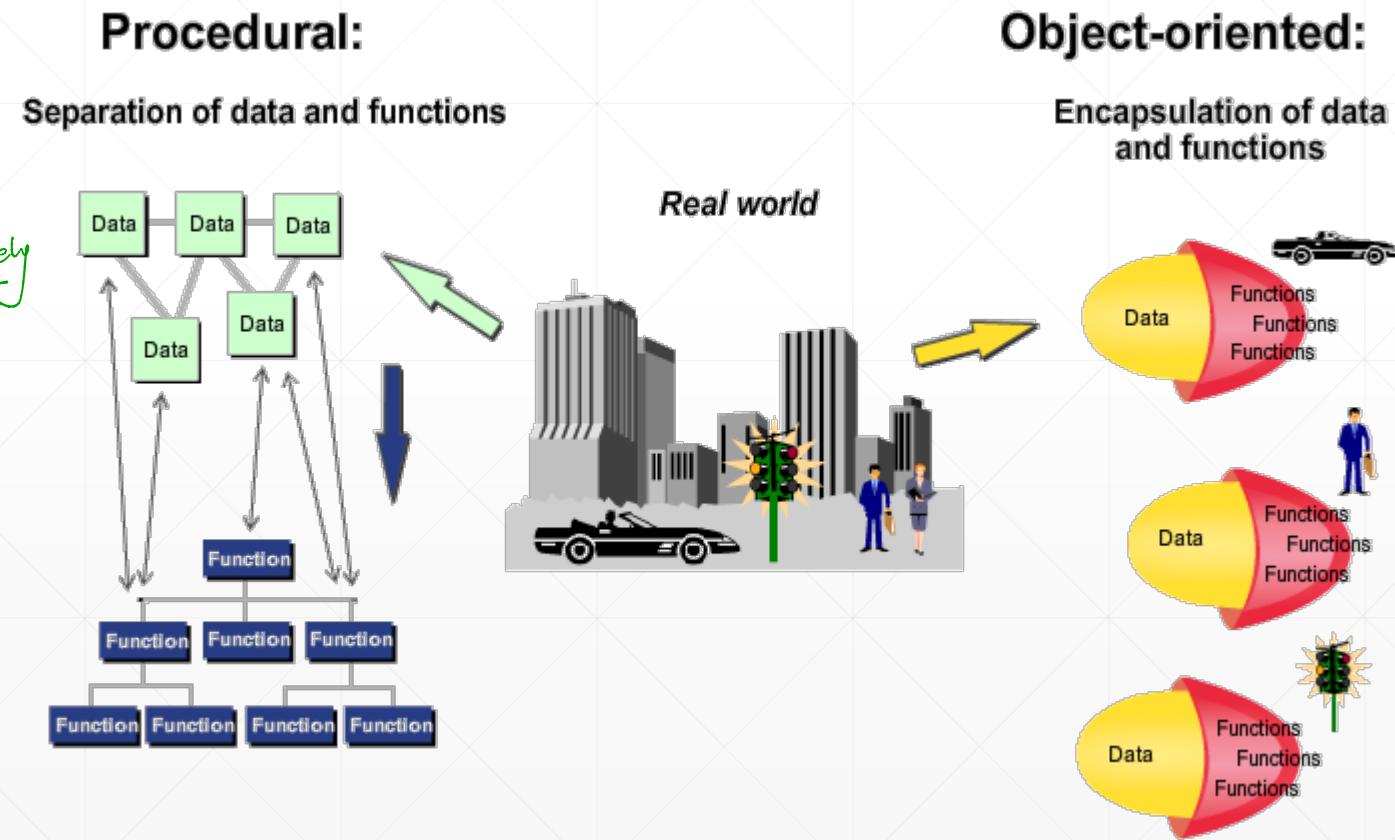
OOP: Procedural vs Object-Oriented Programming

■ Procedural *→ focus on data itself, define series of functions for data.*

- Describes a procedure of a task on the data
- ex) C, Fortran, pascal, etc.

■ Object-Oriented

- Models a set of objects
include functions respectively
- Interaction between the objects
- Ex) Java, C++/C#, Python, etc.



OOP: Procedural vs Object-Oriented Programming (cont'd)

■ Advantages of OOP over procedural programming

- Modularity
 - Source code for an object can be written and maintained independently of the source code for other objects
- Information-hiding
 - Details of internal implementation remain hidden from the outside world
- Code re-use
 - If an object already exists, you can use that object in your program
- Pluggability and debugging ease
 - If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement because of modularity.
productivity ↑



OOP
Class

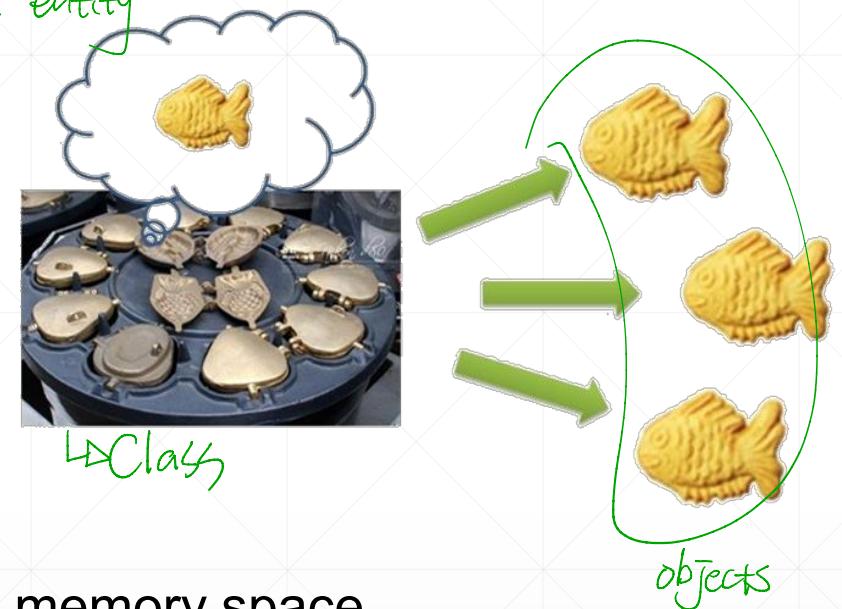
Class

■ Class definition of objects or instances which are complete entity

- Frame/Blueprint to create an object
- Includes states and behaviors of an object
 - fields
 - methods

■ Object

- Instance created based on the class definition
- A concrete entity, created in program runtime, occupying a memory space
- Multiple instances can be created from a single class
 - Class: Student Object/Instance: 201001, 201002, ...



Class (cont'd)

Class: Person

name, age, blood-type
eat, sleep, speak



Name Jane
Age 40
Bloodtype A



Name Kate
Age 30
Bloodtype B



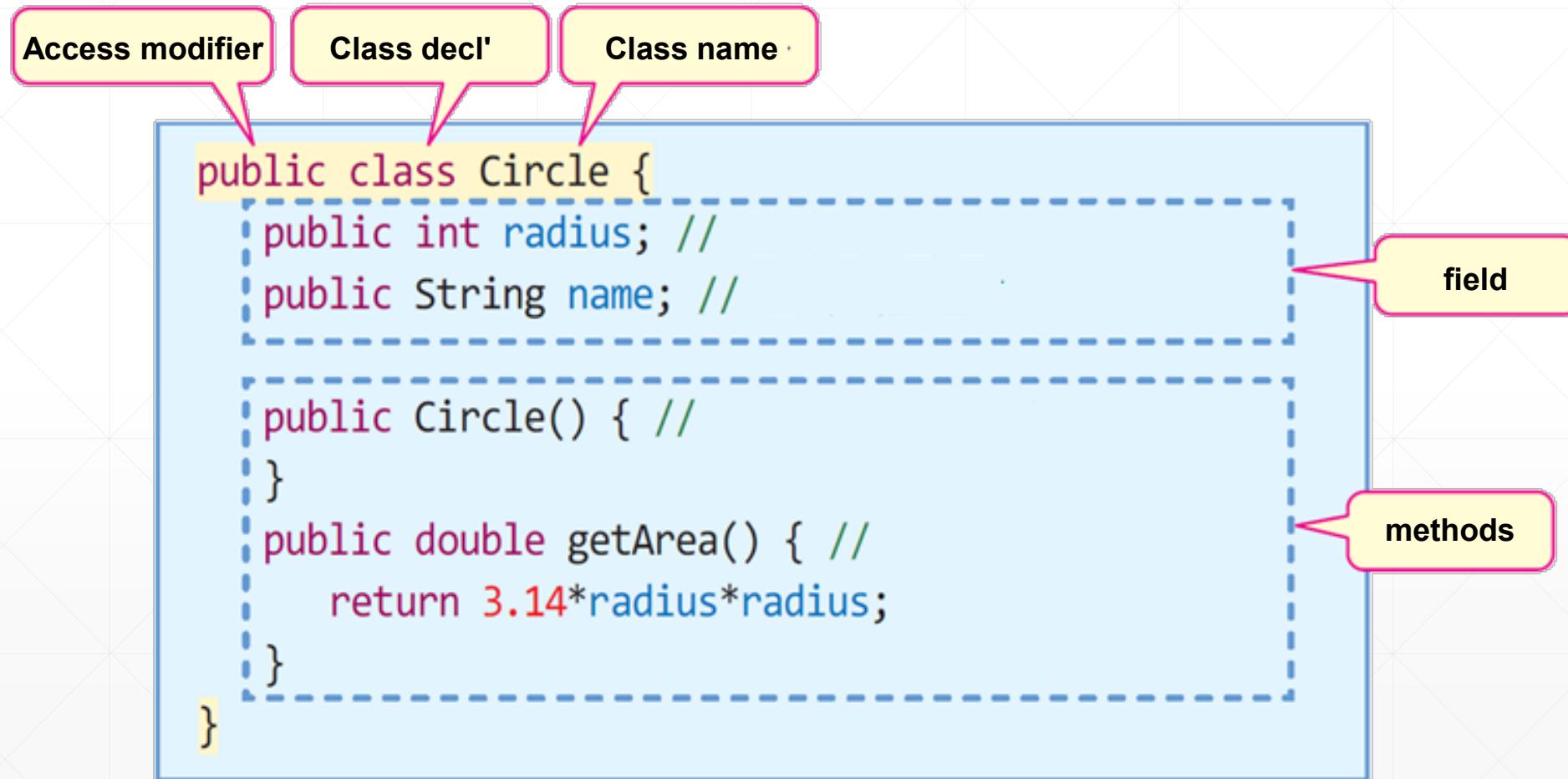
Name John
Age 35
Bloodtype AB

Object: Jane

Object: Kate

Object: John

Class: Structure



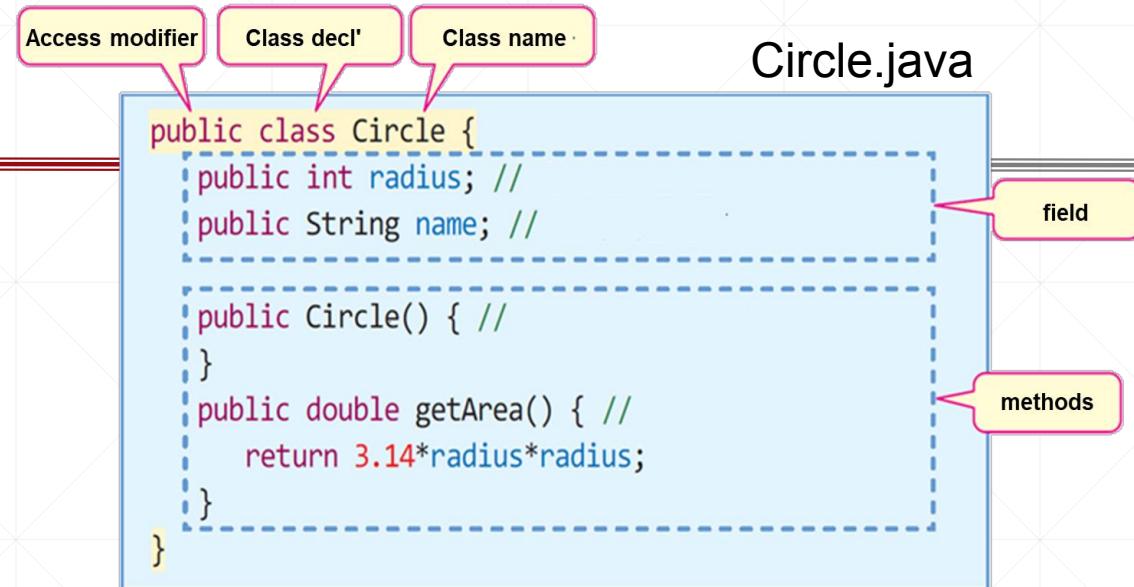
Class: Structure (cont'd)

■ Class declaration in [ClassName].java

- Use “class” keyword
- Begins with “{“ and ends with “}”

■ Fields and methods

- **Field**: member variable to store values (state)
- **Method**: function in which the behavior is implemented
 - **Constructor**
 - Method whose name is identical to that of the class
 - Automatically invoked upon the object is created
 - Instance initialization logic can be implemented in the constructor



■ Access modifier

- Represents the accessibility of a class, fields, methods, etc.

Class: Instantiation

↗ create objects.

■ Object instantiation

- Use “new” keyword to instantiate an object

```
new className();
```

- Constructor of an object is invoked
- The memory address for the created object is returned

- Sequence of object instantiation
 - Declare a reference variable for the object
 - Instantiate an object
 - Memory allocated
 - Constructor invoked for initialization
 - Access the object members

```
objReference.member;
```

ref. variable

field or method

Class: Instantiation (cont'd)

1) Declaration of a reference variable

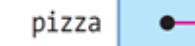
(1) Circle *variable*
class, type
pizza;



Circle-type object

2) Object instantiation using new keyword

(2) pizza = *new* Circle();



Memory allocation
Instantiation

3) Accessing object member (field)

(3) pizza.radius = 10;
pizza.name = "자바피자"



Set value
Set value

4) Accessing object member (method)

(4) double area = pizza.getArea();



getArea()

return 3.14*radius*radius;

Invoke
getArea()

Class: Instantiation (cont'd)

■ Example)

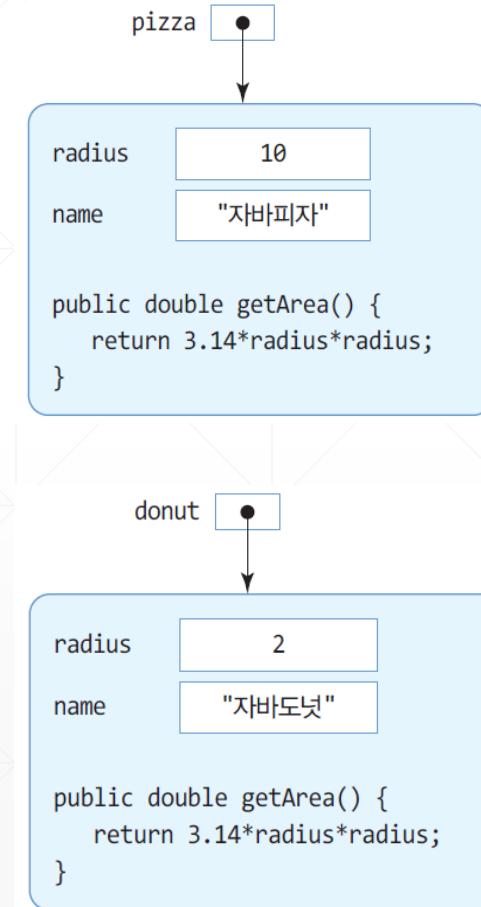
```
public class Circle { // Circle class
    int radius;           // radius field
    String name;          // name field

    public Circle() {}     // constructor

    public double getArea() { // method
        return 3.14*radius*radius;
    }

    public static void main(String[] args) {
        Circle pizza;
        pizza = new Circle();           // Circle object instantiation
        pizza.radius = 10;             // set radius
        pizza.name = "자바피자";       // set name
        double area = pizza.getArea();  // invoke getArea()
        System.out.println(pizza.name + "'s area is " + area);

        Circle donut = new Circle();    // Circle object instantiation
        donut.radius = 2;              // set radius
        donut.name = "자바도넛";        // set name
        area = donut.getArea();         // invoke getArea()
        System.out.println(donut.name + "'s area is " + area);
    }
}
```



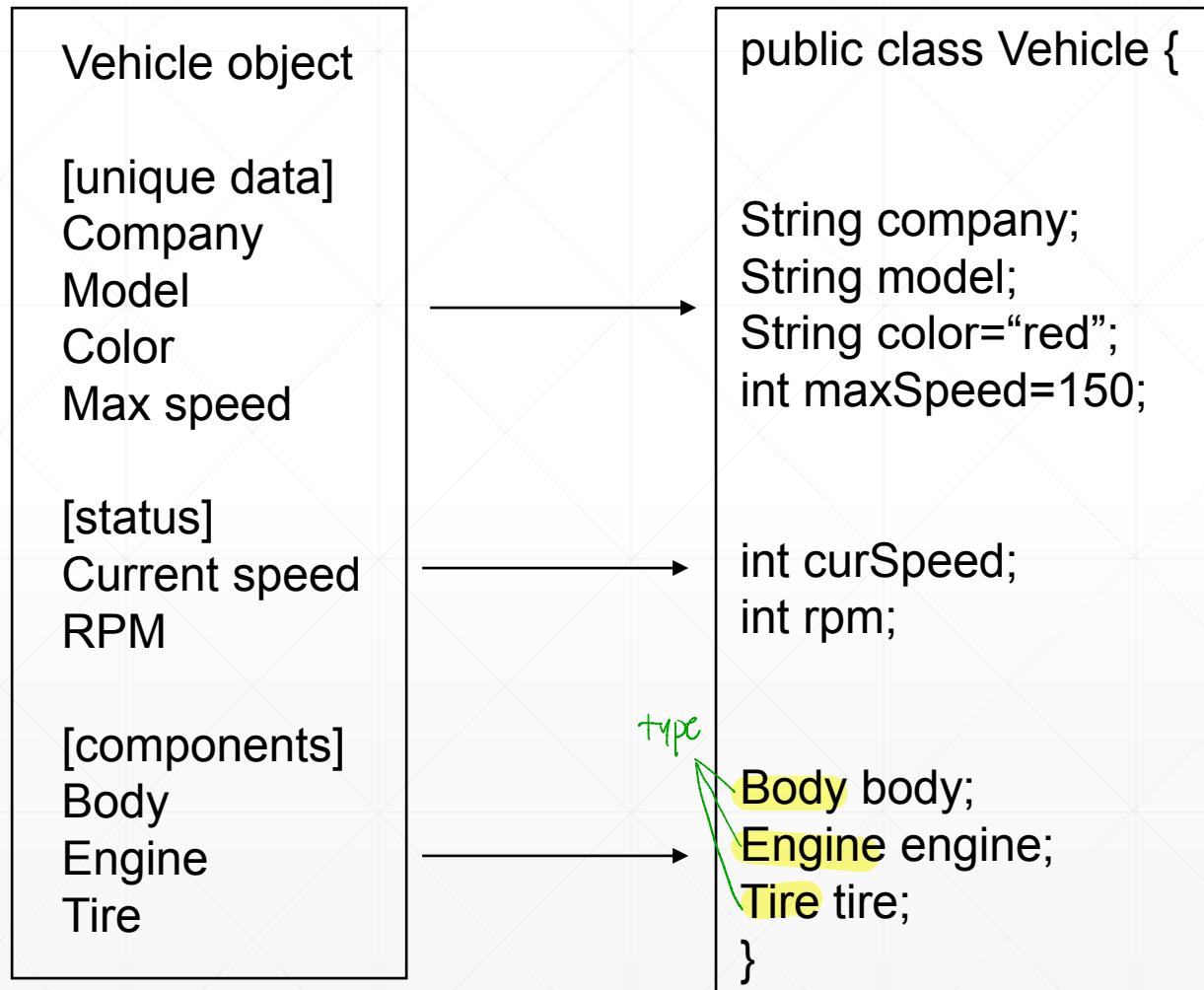
Class: Field

■ What can be the field?

- Object's unique data
- Object's current status
- Sub-objects (components)
- ...

■ Declaration

- Same as variable declaration
- Fields are initialized with the default values upon object instantiation, if no values were set



Class: Field (cont'd)

■ Default values

Category	Type	Default value	
Primitive	Integer	byte	0
		char	'\u0000'
		short	0
		int	0
		long	0L
	Floating-point	float	0.0F
		double	0.0
	boolean	boolean	false
Reference	Array	null	
	Class (including String)	null	
	Interface	null	

Class: Field (cont'd)

■ How to access the fields?

- How to read/write the contents of a field?
- Inside an object
 - Direct access to a field by "fieldName"
- Outside an object
 - Can access by "refVariable.fieldName"

instantiate object first!

```
public class Car { // Car class
```

```
String company;  
String model;  
String color="red";  
int maxSpeed=150;
```

```
public Car(){ // constructor  
    company="SNUTECH";  
    model="ITM";  
}
```

```
public static void main(String[] args) {
```

```
    Car myCar = new Car();  
    System.out.println(myCar.company);  
    myCar.company="SeoulTech";  
    System.out.println(myCar.company);
```

Setting the values
of the fields

Class: Constructor

■ Who initializes an object?

- Constructor!

■ Constructor

- Invoked when a new object is generated
- Multiple definitions allowed
 - At least one constructor MUST be defined
- Name of the constructor MUST be identical to the name of the class
 - ex) public className(...)
- Cannot declare a return type
- Can have arguments

```
public class Car { // Car class
```

```
    String company;  
    String model;  
    String color="red";  
    int maxSpeed=150;
```

```
    public Car(){ // constructor  
        company="SNUTECH";  
        model="ITM";  
    } // initialization
```

```
    public static void main(String[] args) {
```

```
        Car myCar = new Car();  
        System.out.println(myCar.company);  
        myCar.company="SeoulTech";  
        System.out.println(myCar.company);
```

```
}
```

Constructor
block

Class: Constructor (cont'd)

■ Constructor with arguments

- Arguments can be used for initialization

■ Name conflict

- Car's model field
- Constructor's model argument
- How to differentiate it?

■ this keyword

- Reference to a class/object itself

```
public class Car { // Car class  
    String company;  
    String model;  
    String color="red";  
    int maxSpeed=150;  
  
    public Car(String comp, String model){ // constructor  
        company=comp;  
        model=model;  
    }  
  
    public static void main(String[] args) {  
  
        Car myCar = new Car( comp: "SeoulTech", model: "ITM");  
        System.out.println(myCar.company);  
        System.out.println(myCar.model);  
    }  
}
```

Class: Constructor (cont'd)

■ Name conflict

- Car's model field
- Constructor's model argument
- How to differentiate it?

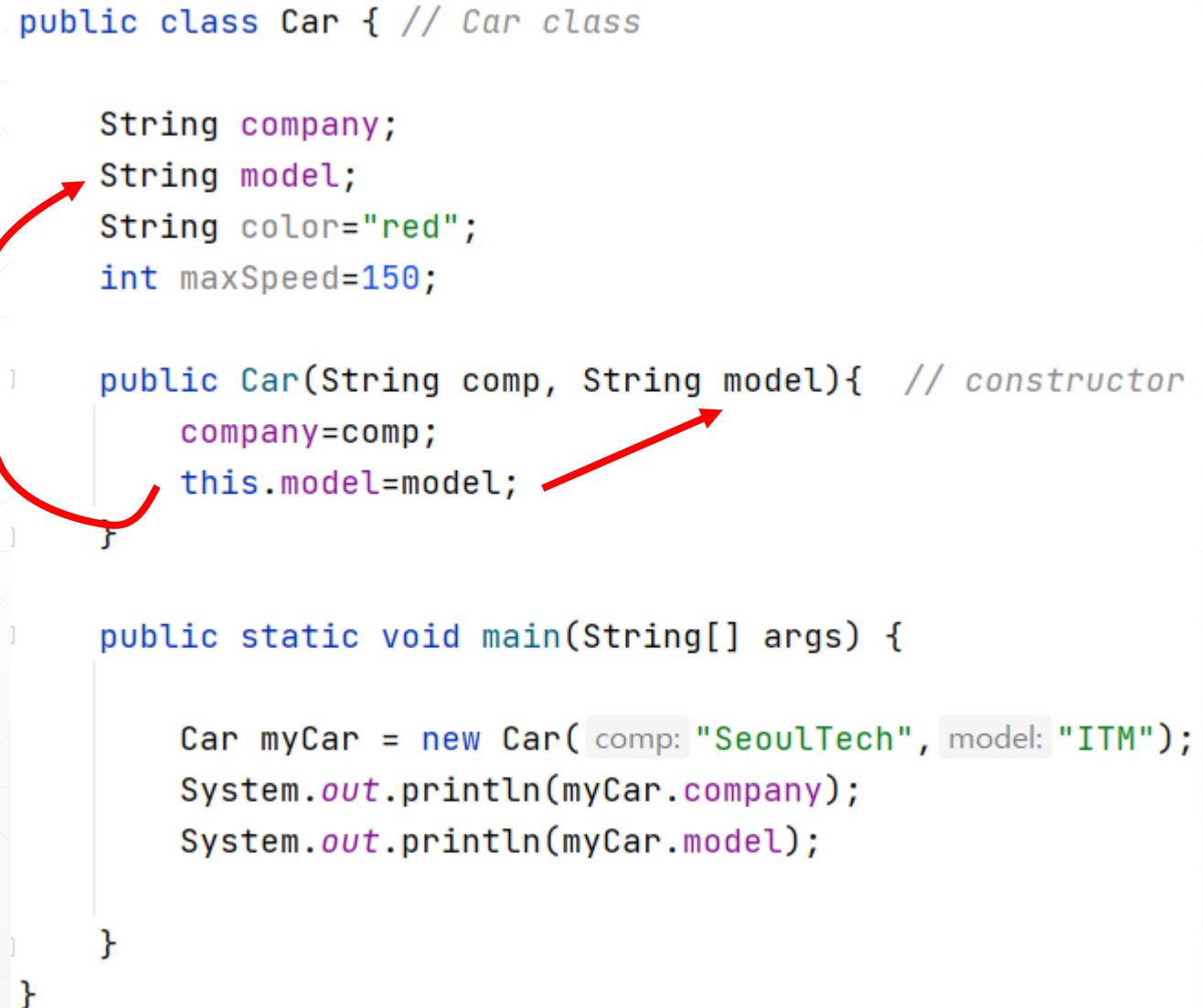
■ this keyword

- Reference to a class/object itself
- Use `this.field` syntax to represent a field of an object itself

```
public class Car { // Car class
    String company;
    String model;
    String color="red";
    int maxSpeed=150;

    public Car(String comp, String model){ // constructor
        company=comp;
        this.model=model;
    }

    public static void main(String[] args) {
        Car myCar = new Car( comp: "SeoulTech", model: "ITM");
        System.out.println(myCar.company);
        System.out.println(myCar.model);
    }
}
```



Class: Constructor (cont'd)

■ Default constructor

- Invoked when a new object is generated
- At least one constructor MUST be defined
- If no constructor is defined by developers, a compiler automatically inserts a default constructor

```
public class Circle {  
    int radius;  
    void set(int r) { radius = r; }  
    double getArea() { return 3.14*radius*radius; }  
  
    public static void main(String [] args){  
        Circle pizza = new Circle();  
        pizza.set(3);  
    }  
}
```

No constructor?
It's impossible!

```
public class Circle {  
    int radius;  
    void set(int r) { radius = r; }  
    double getArea() { return 3.14*radius*radius; }  
  
public Circle() {}  
  
    public static void main(String [] args){  
        Circle pizza = new Circle();  
        pizza.set(3);  
    }  
}
```

Class: Constructor (cont'd)

■ Default constructor: exception

- The default constructor is NOT added if a developer-defined constructor exists
- Developer-defined constructor MUST be used when instantiating an object

```
public class Circle {  
    int radius;  
    void set(int r) { radius = r; }  
    double getArea() { return 3.14*radius*radius; }  
  
    public Circle(int r) {  
        radius = r;  
    }  
    public static void main(String [] args){  
        Circle pizza = new Circle(10);  
        System.out.println(pizza.getArea());  
  
        Circle donut = new Circle();  
        System.out.println(donut.getArea());  
    }  
}
```

// Default constructor (i.e., public Circle()) is not defined

Class: Constructor (cont'd)

■ Multiple constructors

- Multiple constructors can be defined in a class
- Various types of initialization can be performed

```
public class myClass {  
    myClass (arguments, ...){  
        ...  
    }  
  
    myClass (arguments, ...){  
        ...  
    }  
}
```

[Constructor Overloading]
Same name, but the type and number of the arguments are different!

```
public class Car {  
  
    Car(){ ... }  
    Car(String model){ ... }  
    Car(String model, String color) { ... }  
    Car(String model, String color, int speed) { ... }  
}
```

```
Car car1 = new Car();  
  
Car car2 = new Car("WowCar");  
  
Car car3 = new Car("WowCar", "gold");  
  
Car car4 = new Car("WowCar", "gold", 300);
```

Class: Constructor (cont'd)

■ Multiple constructors

- Some codes may be duplicated in the multiple constructors

```
Car (String model){  
    this.company = "SeoulTech";  
    this.model = model;  
    this.maxSpeed=100;  
}
```

```
Car (String company, String model){  
    this.company = company;  
    this.model = model;  
    this.maxSpeed=100;  
}
```

```
Car (String company, String model, int maxSpeed){  
    this.company = company;  
    this.model = model;  
    this.maxSpeed=maxSpeed;  
}
```

[Constructor Overloading]

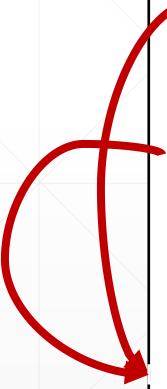
Same name, but the type and number of the arguments are different!

Same name, different
signatures. / should have different type & number
of arguments

Class: Constructor (cont'd)

■ Multiple constructors

- `this()` can be used for calling another constructor in the class
- `this()` MUST be used in the first line of a constructor

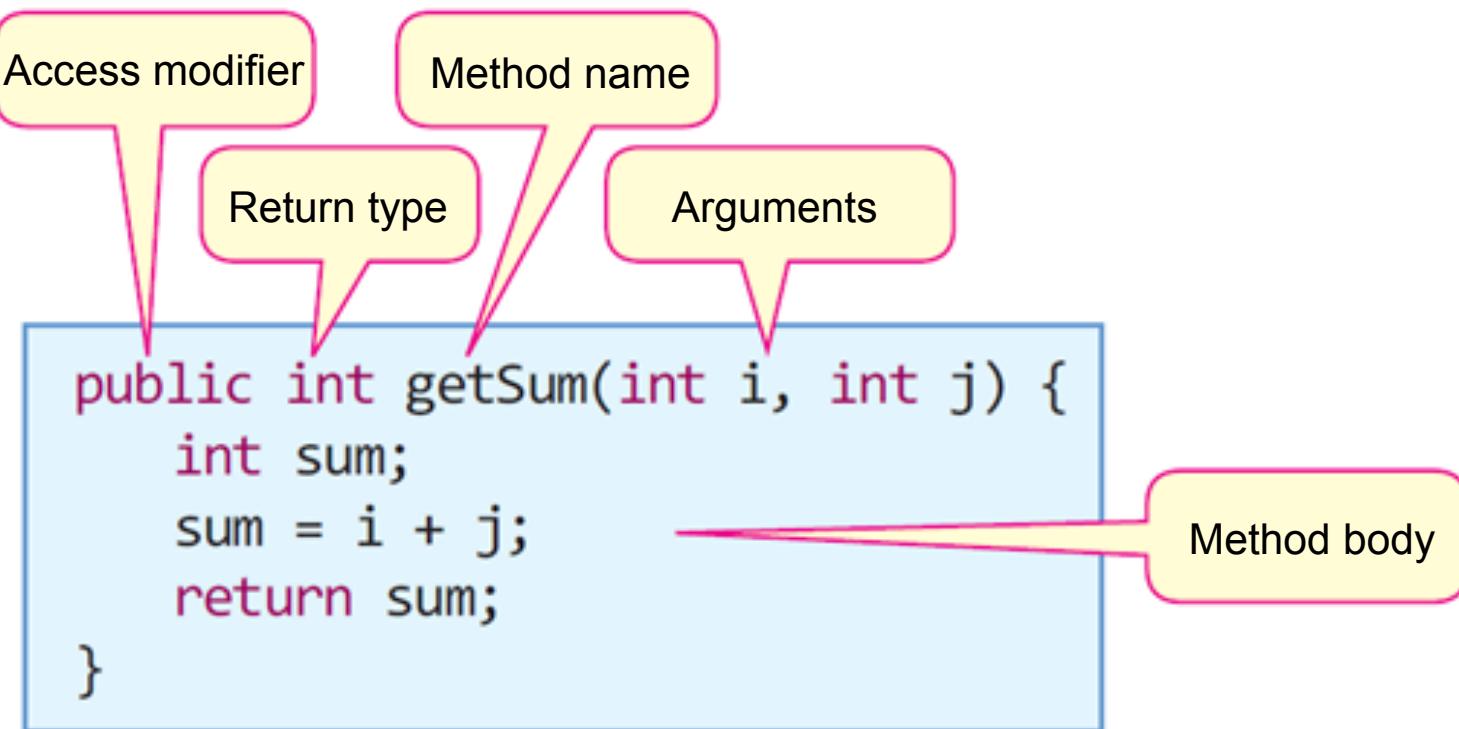


```
Car(String model) {  
    this("SeoulTech",model,150);  
}  
  
Car(String company, String model) {  
    this(company,model,100);  
}  
  
Car(String company, String model, int maxSpeed) {  
    this.company = company;  
    this.model = model;  
    this.maxSpeed = maxSpeed;  
}
```

Class: Method → define behavior of class

■ Behavior of a class

- Behavior of a class is implemented through a method



Class: Method (cont'd)

■ Access modifier ↗ restrict access level

- public, private, protected, default

everybody no external member
classes in same package + child class
classes in same package

■ Return type

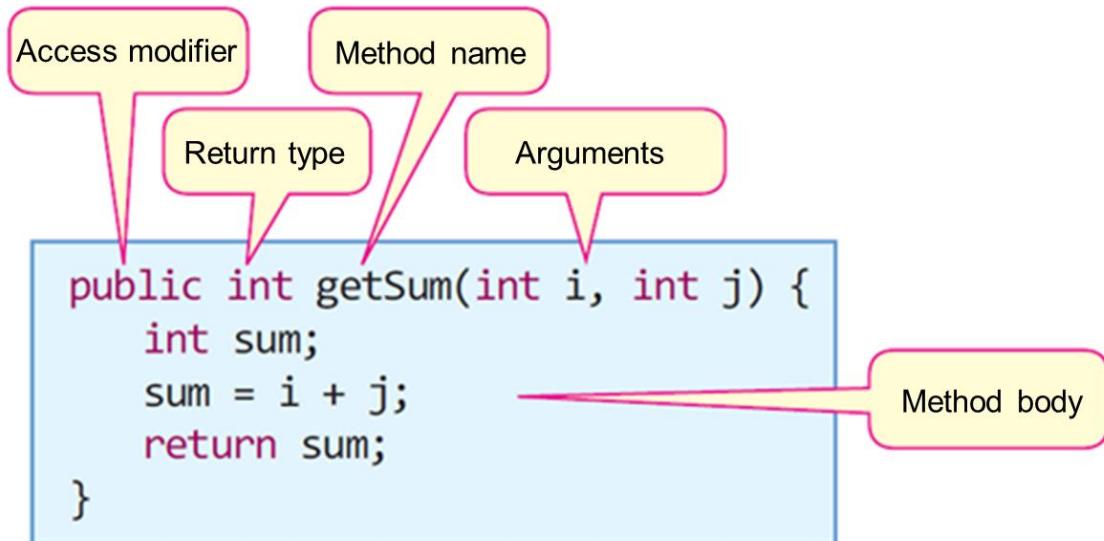
- The type of the data returned by a method
- Method can have no return values (void)

■ Method name

- Method name should follow the JAVA naming rule

■ Arguments

- Input data for a method
- Method can have no input values



```
void powerOn() { ... } // method implementation
double divide(int x, int y) { ... }
```

```
powerOn();
double result = divide( 10, 20 ); // method call
```

```
byte b1 = 10;
byte b2 = 20; // method call
double result = divide(b1, b2);
```

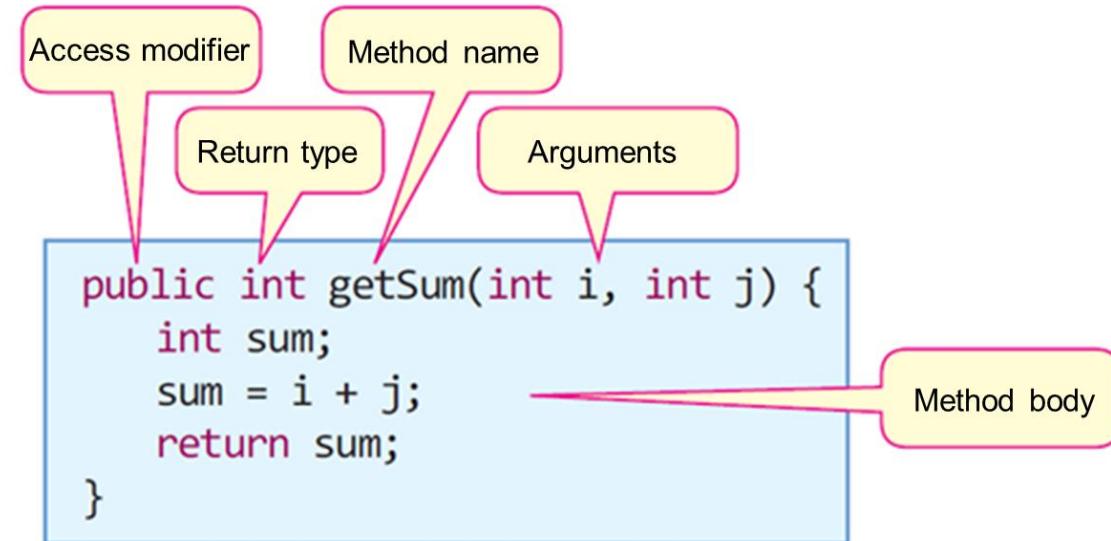
Class: Method (cont'd)



Return statement

- Method with a return type
 - Terminates the method and returns a value
 - The statements after return is not reachable

- Method without a return type
 - Terminates the method
 - The statements after return is not reachable



Class: Method (cont'd)

■ Return statement

- Method with a return type
 - Terminates the method and returns a value
 - The statements after return is not reachable
- Method without a return type
 - Terminates the method
 - The statements after return is not reachable

```
public double getSpeedByMile(){  
    return maxSpeed * 0.62137;  
}  
  
public void showInfo(){  
    if(maxSpeed<100) return;  
    System.out.println(company+"_"+model);  
    System.out.println(maxSpeed);  
}  
  
public static void main(String[] args) {  
  
    Car myCar = new Car("SeoulTech", "ITM", 99);  
    double mySpeed = myCar.getSpeedByMile();  
    System.out.println(mySpeed);  
    myCar.showInfo();  
}
```

Class: Method (cont'd)

Method invocation

- Method call inside a class

- Call a method via its name

- Method call outside a class

- After creation of a class (i.e., object instantiation)
- Call the method through a reference (obj.method)

```
public double getSpeedByMile(){  
    return maxSpeed * 0.62137;  
}  
  
public void showInfo(){  
    if(maxSpeed<100) return;  
    System.out.println(company+"_"+model);  
    System.out.println(getSpeedByMile());  
}
```

```
public static void main(String[] args) {
```

```
    Car myCar = new Car("SeoulTech", "ITM", 99);  
    double mySpeed = myCar.getSpeedByMile();  
    System.out.println(mySpeed);  
    myCar.showInfo();
```

Class: Method (cont'd)

■ Argument passing

➤ Passing primitive-type values

- A value is copied and then passed to the method
- Change of the argument **does not affect the original value**

➤ Passing reference-type values (e.g., object, array, etc.)

- A reference is passed to the method
- Change of the argument **affects the original value**

→ directly access reference (memory location), multiple variables pointing same memory area → change from any reference will affect reference & memory

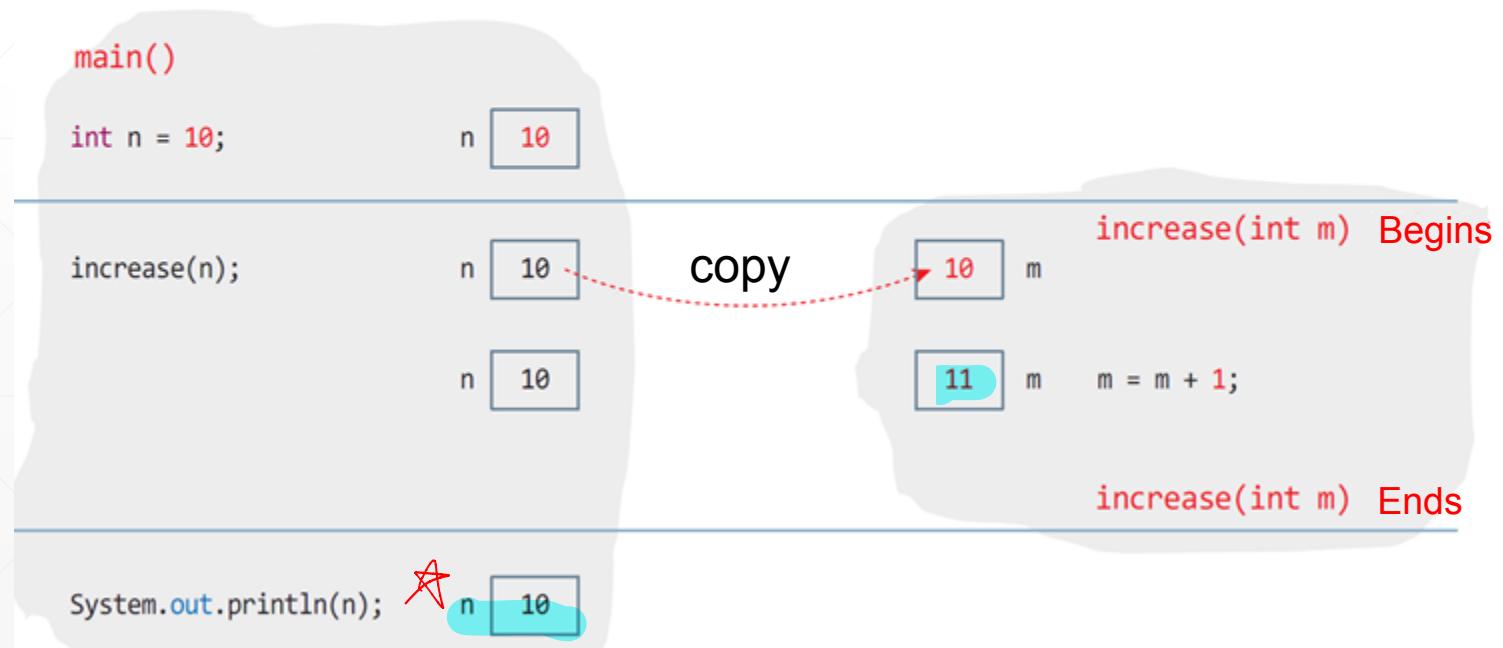
Class: Method (cont'd)

Argument passing (Passing primitive-type values)

- A value is copied and then passed to the method
- Change of the argument does not affect the original value

```
public class ValuePassing {  
    public static void main(String args[]) {  
        int n = 10;  
  
        increase(n);  
  
        System.out.println(n);  
    }  
}
```

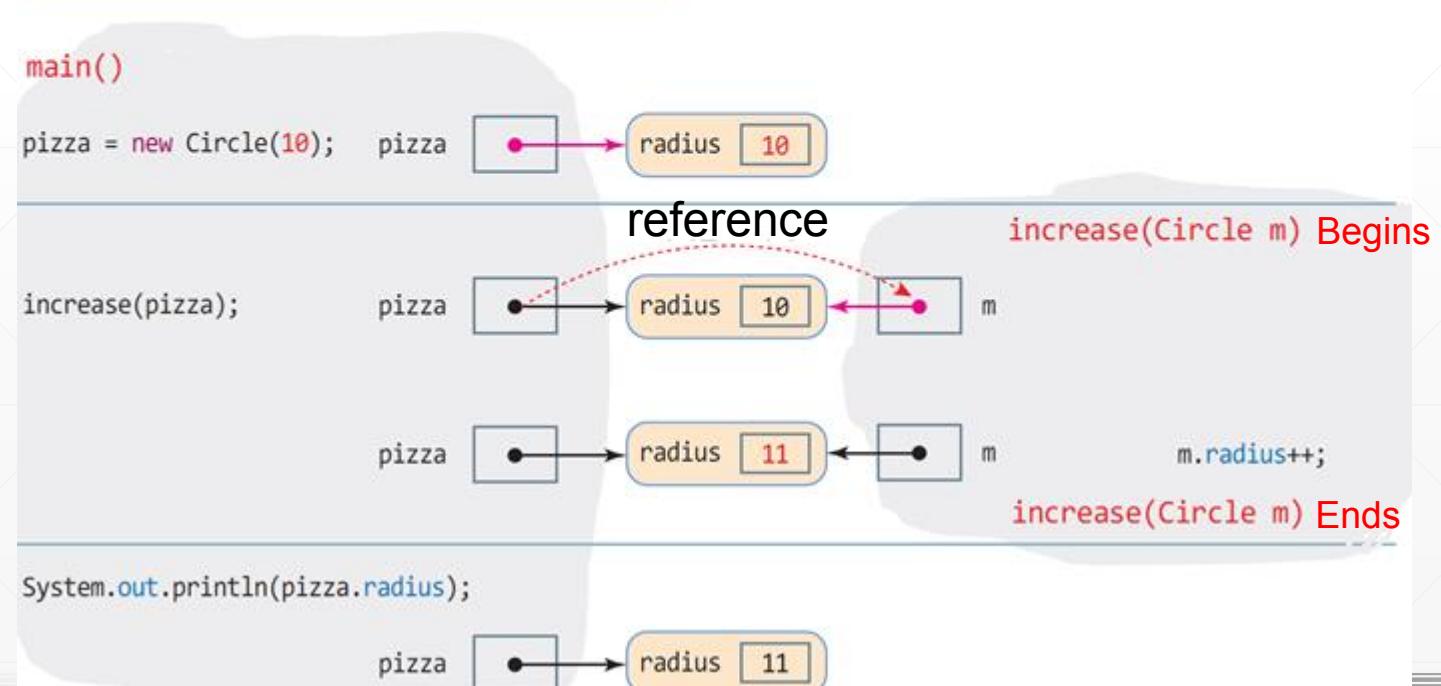
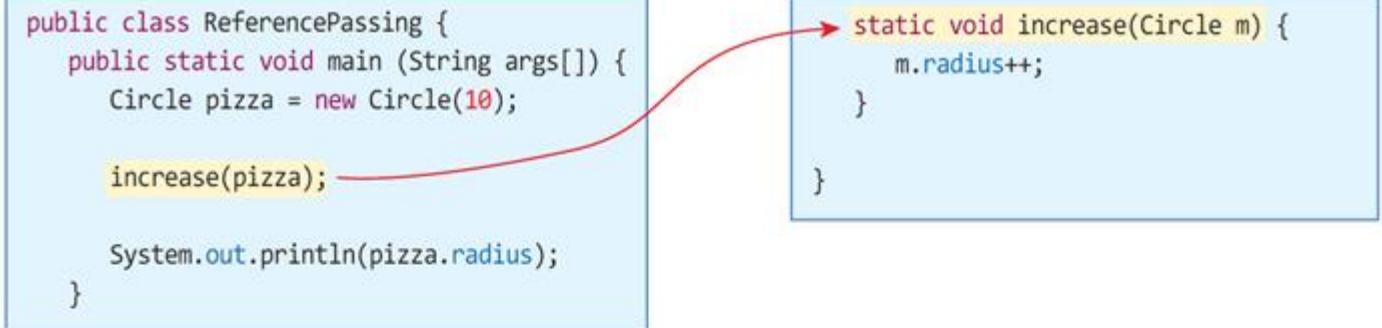
```
static void increase(int m) {  
    m = m + 1;  
}  
}
```



Class: Method (cont'd)

Argument passing (Passing reference-type values)

- A reference is passed to the method
- Change of the argument affects the original value

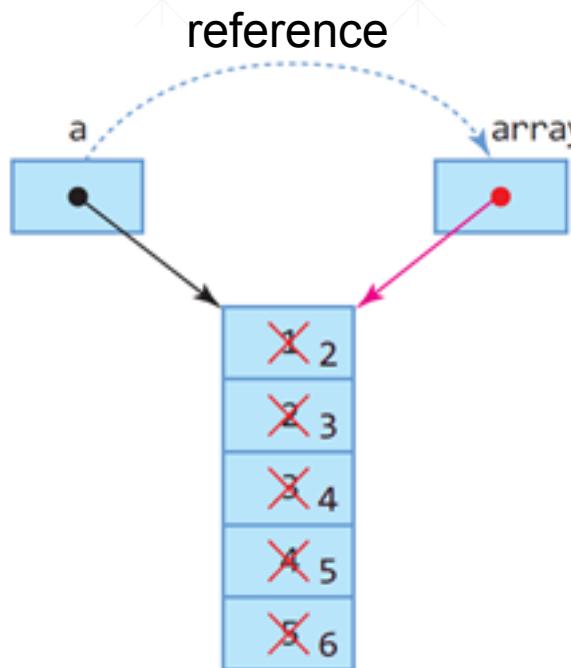


Class: Method (cont'd)

Argument passing (Passing reference-type values)

- A reference is passed to the method
- Change of the argument affects the original value

```
public class ArrayPassing {  
  
    public static void main(String args[]) {  
        int a[] = {1, 2, 3, 4, 5};  
  
        increase(a);  
  
        for(int i=0; i<a.length; i++)  
            System.out.print(a[i] + " ");  
    }  
}
```



```
static void increase(int[] array) {  
    for(int i=0; i<array.length; i++) {  
        array[i]++;  
    }  
}
```

Class: Method Overloading

- Methods with the same name, but with **the different number/type of arguments**

// successful method overloading

```
class MethodOverloading {  
    public int getSum(int i, int j) {  
        return i + j;           2 arguments  
    }  
    public int getSum(int i, int j, int k) {  
        return i + j + k;       3 arguments  
    }  
}
```

// Fail!

```
class MethodOverloadingFail {  
    public int getSum(int i, int j) {  
        return i + j;           same method! ↗ impossible to have  
    }  
    public double getSum(int i, int j) {  
        return (double)(i + j);  different return type  
    }  
}
```

Class: Method Overloading (cont'd)

- Methods with the same name, but with **the different number/type of arguments**

```
public static void main(String args[]) {  
    MethodSample a = new MethodSample();  
  
    int i = a.getSum(1, 2);  
  
    int j = a.getSum(1, 2, 3);  
  
    double k = a.getSum(1.1, 2.2);  
}
```

3 different method calls

```
public class MethodSample {  
    public int getSum(int i, int j) {  
        return i + j;  
    }  
  
    public int getSum(int i, int j, int k) {  
        return i + j + k;  
    }  
  
    public double getSum(double i, double j) {  
        return i + j;  
    }  
}
```

Q&A

■ Next week (eClass video)

- OOD/P: More about Methods
- OOD/P: Inheritance

Computer Language

OOP 2: Method and Inheritance

Agenda

- Method
- Inheritance

Method Inheritance

Method: Instance Member

- Fields and methods of an object/instance *Instance member*
- Instance field
- Instance method
- Instance members belong to an object/instance
- Therefore, instance members cannot be used without object instantiation!

```
public class Car {  
    // field  
    int gas;  
  
    // method  
    void setSpeed(int speed) { ... }  
}
```

```
Car myCar = new Car();  
myCar.gas = 10;  
myCar.setSpeed(60);  
  
Car yourCar = new Car();  
yourCar.gas = 20;  
yourCar.setSpeed(80);
```

Method: Static Member

- Fields and methods of a class

- Static field, Static method
 - Sometimes called class members

- Static members belong to a class

- Therefore, static members can be used without object instantiation!

- Declaration

- Use static keyword for the members!

```
public class Calculator {  
    static double pi = 3.14159;  
    static int plus(int x, int y) { ... }  
    static int minus(int x, int y) { ... }  
}
```

Method: Static Member (cont'd)

■ Instance vs Static members

	Instance member	Static member
Declaration	<pre>class Sample{ int n; void g(){...} }</pre>	<pre>class Sample{ static int n; static void g(){...} }</pre>
Where?	for each object-instance	for a single Class - class members - static members loaded into method area
When?	Once an object is created After object instantiation, instance members can be used	Once class is loaded Static members can be used without any object instantiation
Sharable?	No Instance members reside in each object	Yes Shared with all objects of the class

Method: Static Member (cont'd)

■ When to use Static members?

➤ Global variable/methods

- Example) Math class (java.lang.Math)
 - All the methods and fields are declared static
 - Without Math object instantiation, we can use all the features of Math class!

```
public class Math {  
    public static int abs(int a);  
    public static double cos(double a);  
    public static int max(int a, int b);  
    public static double random();  
    ...  
}
```

```
Math m = new Math(); // Error!  
int n = Math.abs(-5);
```

➤ Sharable members

- All instances of the class can share the static members

```
String company; } instance field  
String model;  
static int maxSpeed = 150; ) Static field, class member
```

```
Car(String company, String model) {  
    this.company = company;  
    this.model = model;  
}
```

```
static void bomb(){  
    System.out.println("destroyed");  
}
```

```
public static void main(String[] args) {  
  
    Car myCar = new Car("my","my");  
    Car yourCar = new Car("you","you");  
    System.out.println(myCar.company+": "+myCar.maxSpeed);  
    System.out.println(yourCar.company+": "+yourCar.maxSpeed);  
  
    Car.maxSpeed = 200;  
    System.out.println(myCar.company+": "+myCar.maxSpeed);  
    System.out.println(yourCar.company+": "+yourCar.maxSpeed);  
  
    myCar.maxSpeed = 300;  
    System.out.println(myCar.company+": "+myCar.maxSpeed);  
    System.out.println(yourCar.company+": "+yourCar.maxSpeed);  
  
    Car.bomb();  
}
```

Method: Static Member (cont.)

■ Example)

- Use of static field
- Use of static method
- What happens we modify static fields?

Method: Static Member (cont'd)

■ Restrictions

- Instance members cannot be used in a static context
- `this` keyword cannot be used in a static context

```
class StaticMethod {  
    int n; → instance field  
    void f1(int x) {n = x;} // OK  
    void f2(int x) {m = x;} // OK  
  
    static int m;  
    static void s1(int x) {n = x;} // Error!  
  
    static void s2(int x) {f1(3);} // Error!  
  
    static void s3(int x) {m = x;} // OK  
    static void s4(int x) {s3(3);} // OK  
}
```

```
class StaticAndThis {  
    int n; instance  
    static int m; static  
    void f1(int x) {this.n = x;}  
    void f2(int x) {this.m = x;} // OK  
    static void s1(int x) {this.n = x;} // Error!  
    static void s2(int x) {this.m = x;} // Error!  
}
```

Method: Static Member (cont'd)

- Example 1) Write three static functions (abs, max, and min)

```
class Calc {  
    public static int abs(int a) { return a>0?a:-a; }  
    public static int max(int a, int b) { return (a>b)?a:b; }  
    public static int min(int a, int b) { return (a>b)?b:a; }  
}  
  
public class CalcEx {  
    public static void main(String[] args) {  
        System.out.println(Calc.abs(-5));  
        System.out.println(Calc.max(10, 8));  
        System.out.println(Calc.min(-3, -8));  
    }  
}
```

5
10
-8

Method: Static Member (cont'd)

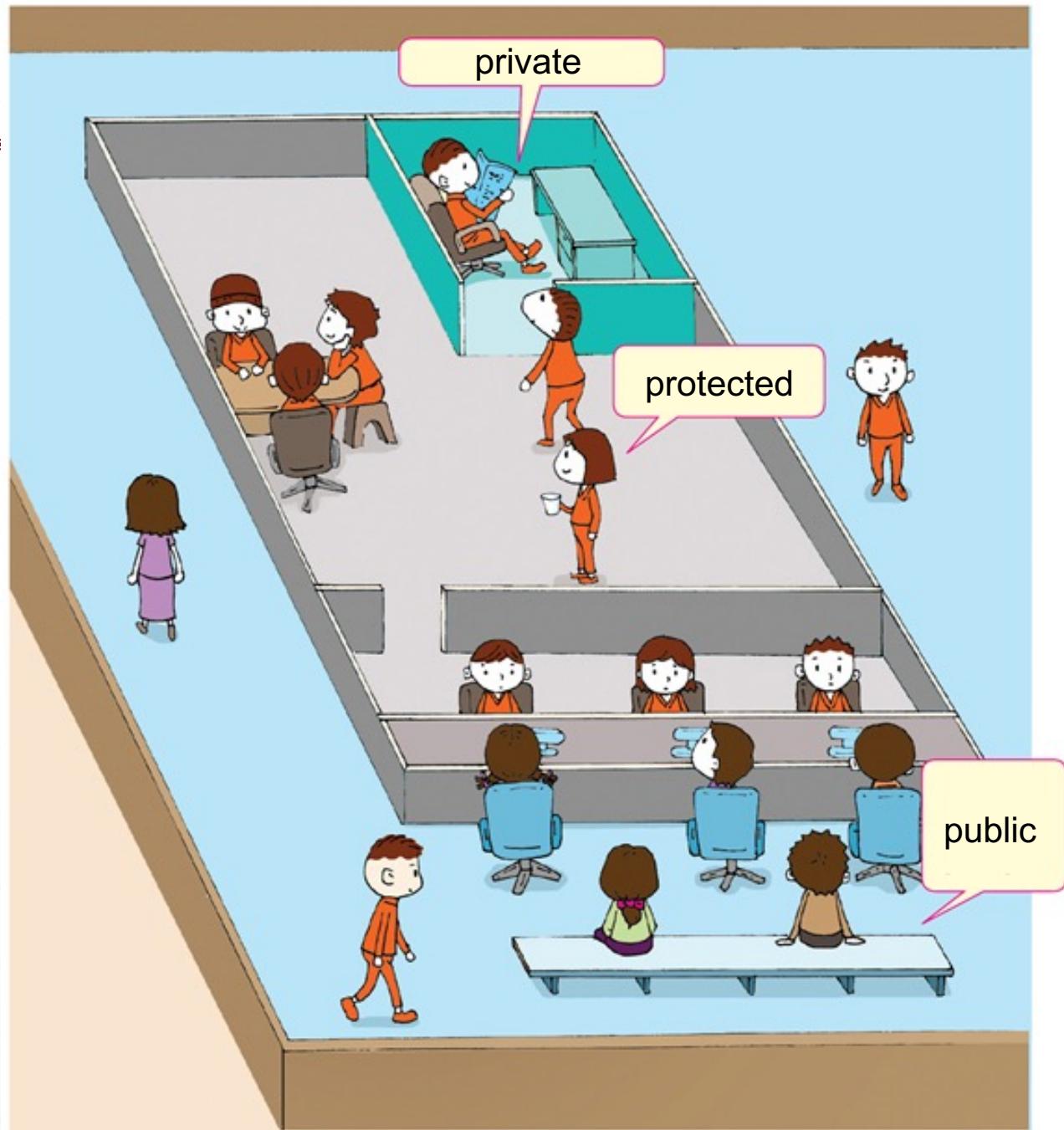
■ Example 2) Write an exchange rate calculator using static members

```
class CurrencyConverter {  
    private static double rate; // exchange rate (KRW:$)  
    public static double toDollar(double won) {  
        return won/rate; // from Won to Dollar  
    }  
    public static double toKWR(double dollar) {  
        return dollar * rate; // from Dollar to Won  
    }  
    public static void setRate(double r) {  
        rate = r; // exchange rate: KWR/$1  
    }  
}  
  
public class StaticMember {  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
        System.out.print("Exchange rate (1$)>> ");  
        double rate = scanner.nextDouble();  
        CurrencyConverter.setRate(rate); // setting exchange rate  
        System.out.println("1M Won is $" + CurrencyConverter.toDollar(1000000));  
        System.out.println("$100 is " + CurrencyConverter.toKWR(100) + "won.");  
        scanner.close();  
    }  
}
```

Exchange rate (1\$)>> 1200
1M Won is \$833.333333333334
\$100 is 120000.0won

Method: Access Modifier

- Determine who can access!



Method: Access Modifier (cont'd)

■ Java Package

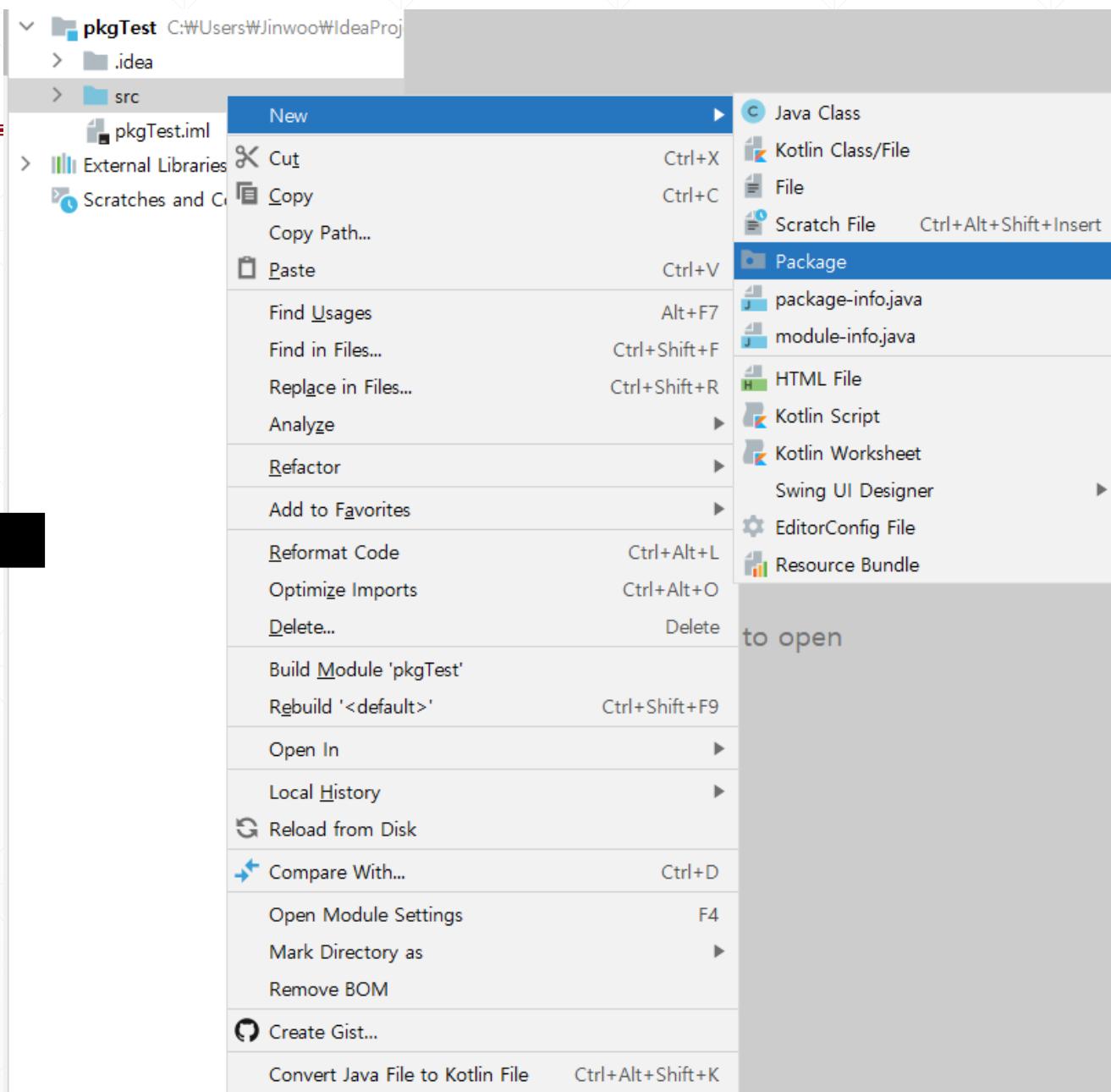
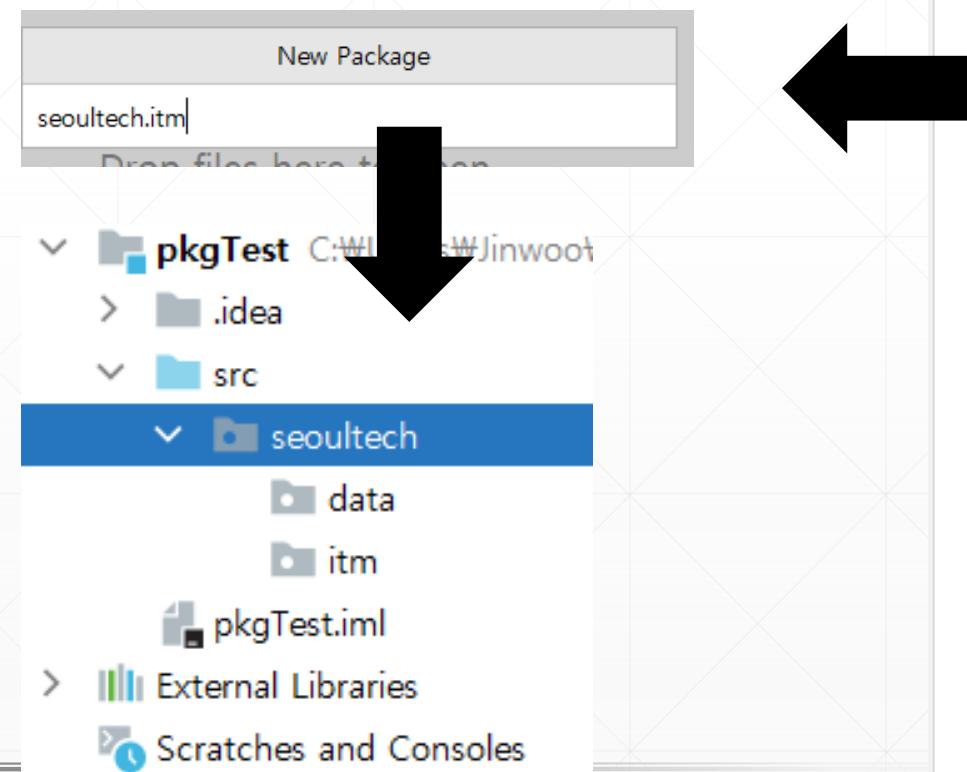
- A group of classes with similar features/categories
- Similar to “folder” in Filesystem to manage files
- Class name is composed of
 - Package names (hierarchy)
 - Class name
- Class name can be uniquely identified by its package names
 - superPkg.**subPkg1**.myClass
 - superPkg.**subPkg2**.myClass

 These two classes are different!

Method: Access Modifier

Java Package

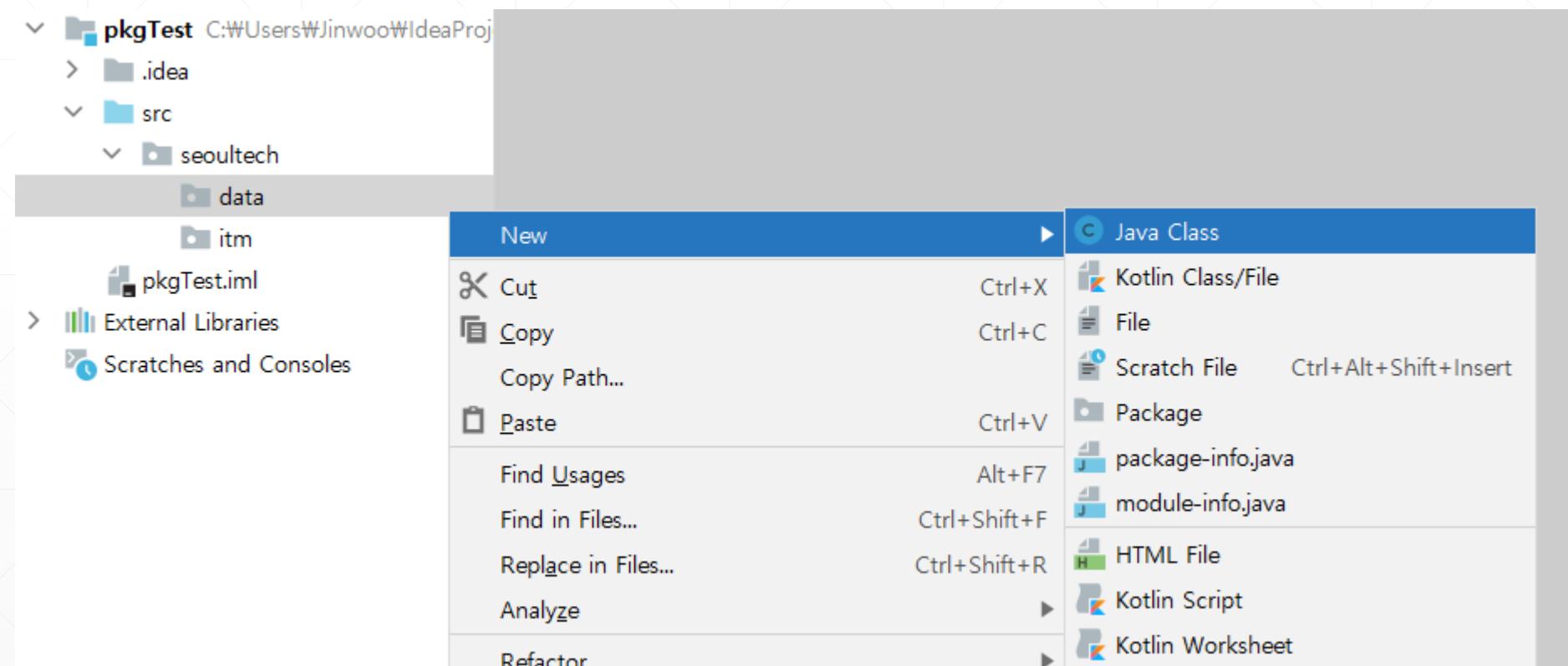
- How to create a package?
 - src → new → package
 - Choose your package name



Method: Access Modifier (cont'd)

■ Java Package

- How to create a class in a package?
 - Create a new class under a specific package



Method: Access Modifier (cont'd)

■ Java Package

- Package information is added in your class file

The screenshot shows a Java project structure in the left-hand sidebar of an IDE. The project is named 'pkgTest' and contains a file 'pkgTest.iml'. The 'src' directory contains a package named 'seoultech' which has two sub-directories: 'data' and 'itm'. Inside 'itm', there is a class named 'ComLang'. The code editor on the right shows the source code for 'ComLang':

```
1 package seoultech.itm;
2
3 public class ComLang {
4
5 }
```

The first line of code, 'package seoultech.itm;', is highlighted with a red rectangular box.

Method: Access Modifier (cont'd)

■ Java Package

- We can have multiple packages with the same name, under different packages

The screenshot shows two instances of the IntelliJ IDEA IDE interface. Both instances have a file tree on the left and a code editor on the right.

Top Instance:

- File Tree: Shows a project named "pkgTest" with a ".idea" folder, an "out" folder, and a "src" folder. The "src" folder contains a "seoultech" package which has a "data" folder and two "DataMath" files. The "DataMath" file under "itm" is selected.
- Code Editor:

```
1 package seoultech.itm;
2
3 public class DataMath {
4     public String msg="ITM's datamath";
5 }
6
```

Bottom Instance:

 - File Tree: Shows the same project structure, but the "DataMath" file under "itm" is now grayed out, indicating it is not the active file.
 - Code Editor:

```
1 package seoultech.data;
2
3 public class DataMath {
4     public String msg="Data's datamath";
5 }
6
```

A red box highlights the "package" keyword in both code snippets, and a yellow bar highlights the class body in the bottom instance's code editor.

Method: Access Modifier (cont'd)

■ Java Package

- To use a class in another package, we need to import it first!
- We can access a certain class in the same package using its class name

The screenshot shows the IntelliJ IDEA interface. On the left is the Project tool window displaying a package structure under 'pkgTest'. The 'src' folder contains a 'seoultech' package with 'data' and 'itm' subfolders. Inside 'itm', there are two classes: 'ComLang' (selected) and 'DataMath'. Other files in the 'itm' folder include 'pkgTest.iml', 'External Libraries', and 'Scratches and Consoles'. The right side shows the code editor with the following Java code:

```
package seoultech.itm;
import java.util.Scanner;

public class ComLang {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        DataMath dm = new DataMath();
        System.out.println(dm.msg);
    }
}
```

Two specific lines of code are highlighted with red boxes: 'import java.util.Scanner;' and 'System.out.println(dm.msg);'.

Method: Access Modifier (cont'd)

Java Package

- To use a class in another package, we need to import it first!
- We can access a certain class in the same package using its class name

The screenshot shows the IntelliJ IDEA interface. On the left, the project structure is displayed with the following hierarchy:

- pkgTest** (Project root)
 - .idea
 - out
 - src
 - seoultech
 - data
 - DataMath
 - itm
 - ComLang
 - DataMath
 - pkgTest.iml
 - External Libraries
 - Scratches and Consoles

The code editor on the right contains the following Java code:

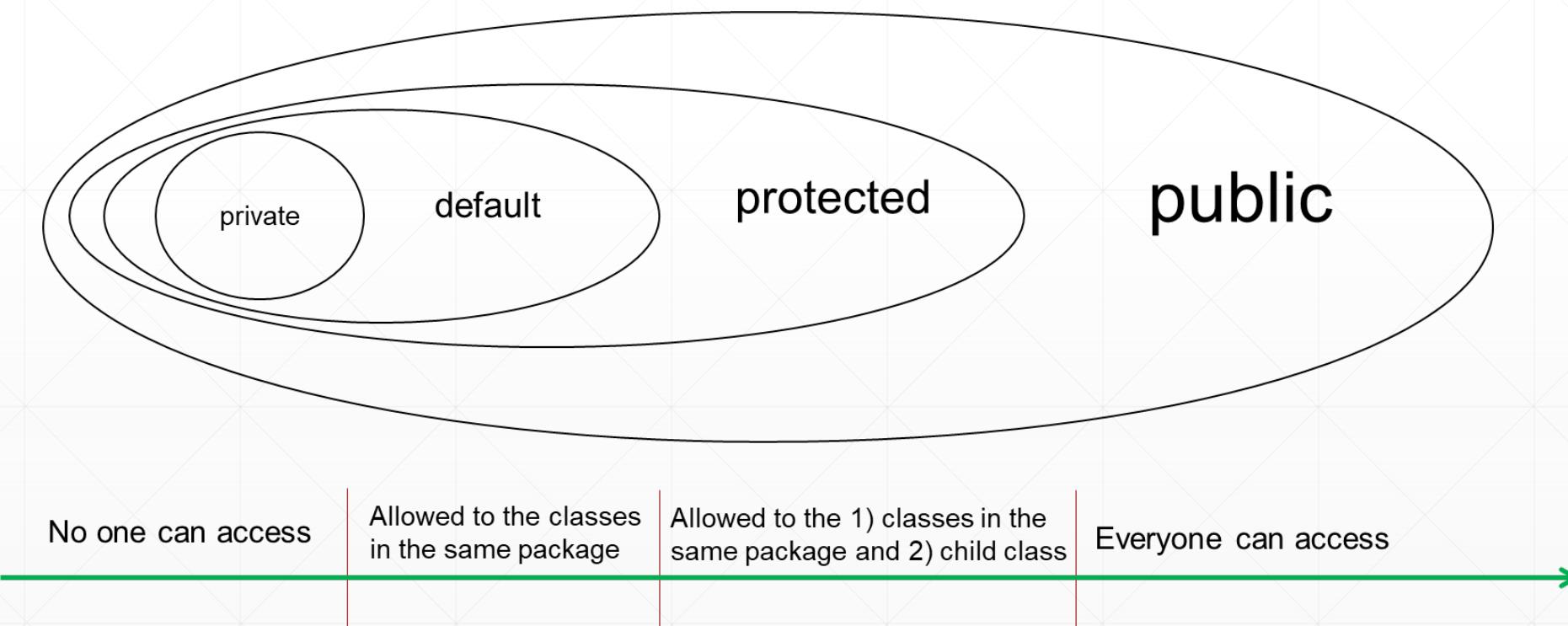
```
2 import seoultech.data.DataMath;
3
4
5 import java.util.Scanner;
6
7 public class ComLang {
8
9     public static void main(String[] args) {
10         Scanner scanner = new Scanner(System.in);
11         //DataMath dm = new DataMath();
12         DataMath dm = new DataMath();
13         System.out.println(dm.msg);
14     }
15 }
```

Two specific lines of code are highlighted with red boxes:

- `import seoultech.data.DataMath;`
- `System.out.println(dm.msg);`

Method: Access Modifier (cont'd)

- Keyword to determine whether other classes can use a particular field or invoke a particular method in the class
 - Related with encapsulation
 - Hide sensitive data, expose publicly available interfaces!



Method: Access Modifier (cont'd)

■ Top-level modifiers

- Public, Default (package-private)
- Determine who can use a class

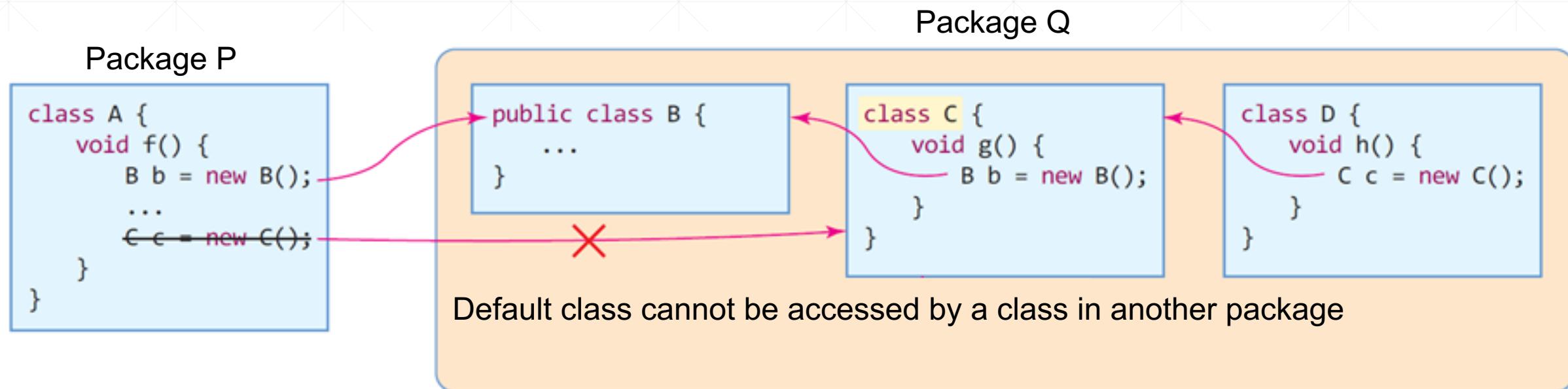
■ Member-level modifiers

- Public, Private, Protected, Default (package-private)
- Determine who can access the fields and methods of a class

Method: Access Modifier (cont'd)

■ Top-level access modifier

- Public: all classes can access
- Default (package-private): only the class in the same package can access



Method: Access Modifier (cont'd)

■ Member-level access modifier

- Public: all classes can access
- Private: no one can access
- Protected:
 - All classes in the same package can access
 - Subclass in another package can access
- Default (package-private): only the class in the same package can access

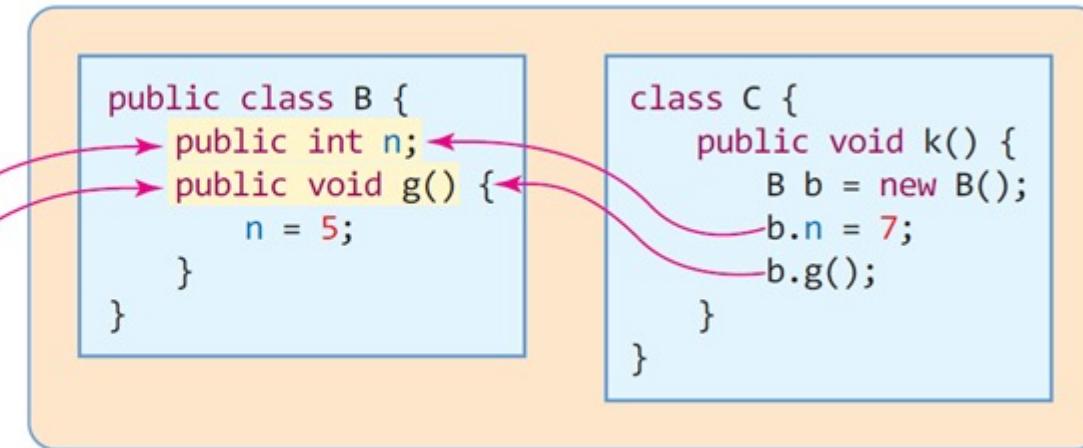
Accessor	Access modifier of a member			
	private	default	protected	public
Class in the same package	X	O	O	O
Class in another package	X	X	X O (child)	O

Method: Access Modifier

■ Member-level access modifier

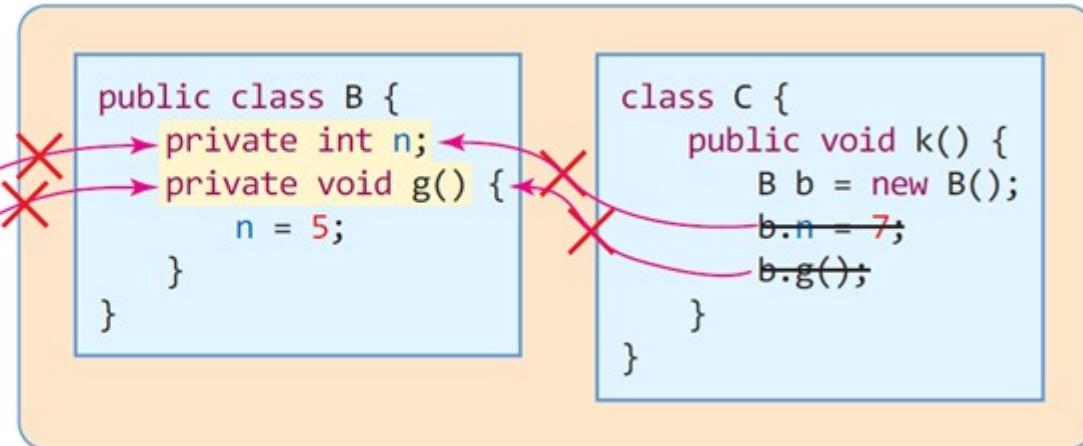
➤ Example of public modifiers

```
class A {  
    void f() {  
        B b = new B();  
        b.n = 3;  
        b.g();  
    }  
}
```



➤ Example of private modifiers

```
class A {  
    void f() {  
        B b = new B();  
        b.n = 3;  
        b.g();  
    }  
}
```



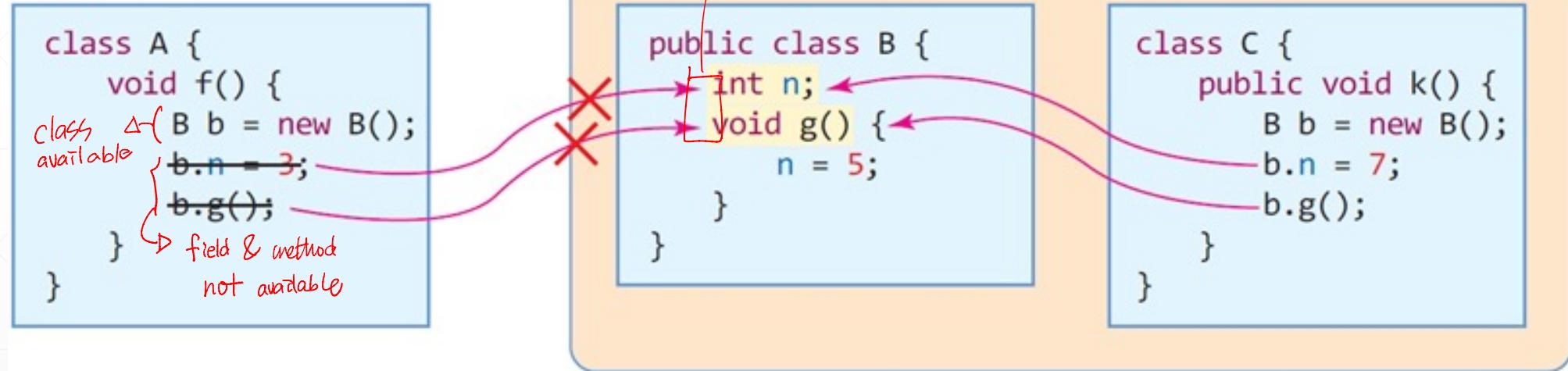
Accessor	Access modifier of a member			
	private	default	protected	public
Class in the same package	X	O	O	O
Class in another package	X	X	X	O

Method: Access Modifier

■ Member-level access modifier

- Example of default modifiers

Accessor	Access modifier of a member			
	private	default	protected	public
Class in the same package	X	O	O	O
Class in another package	X	X	X	O

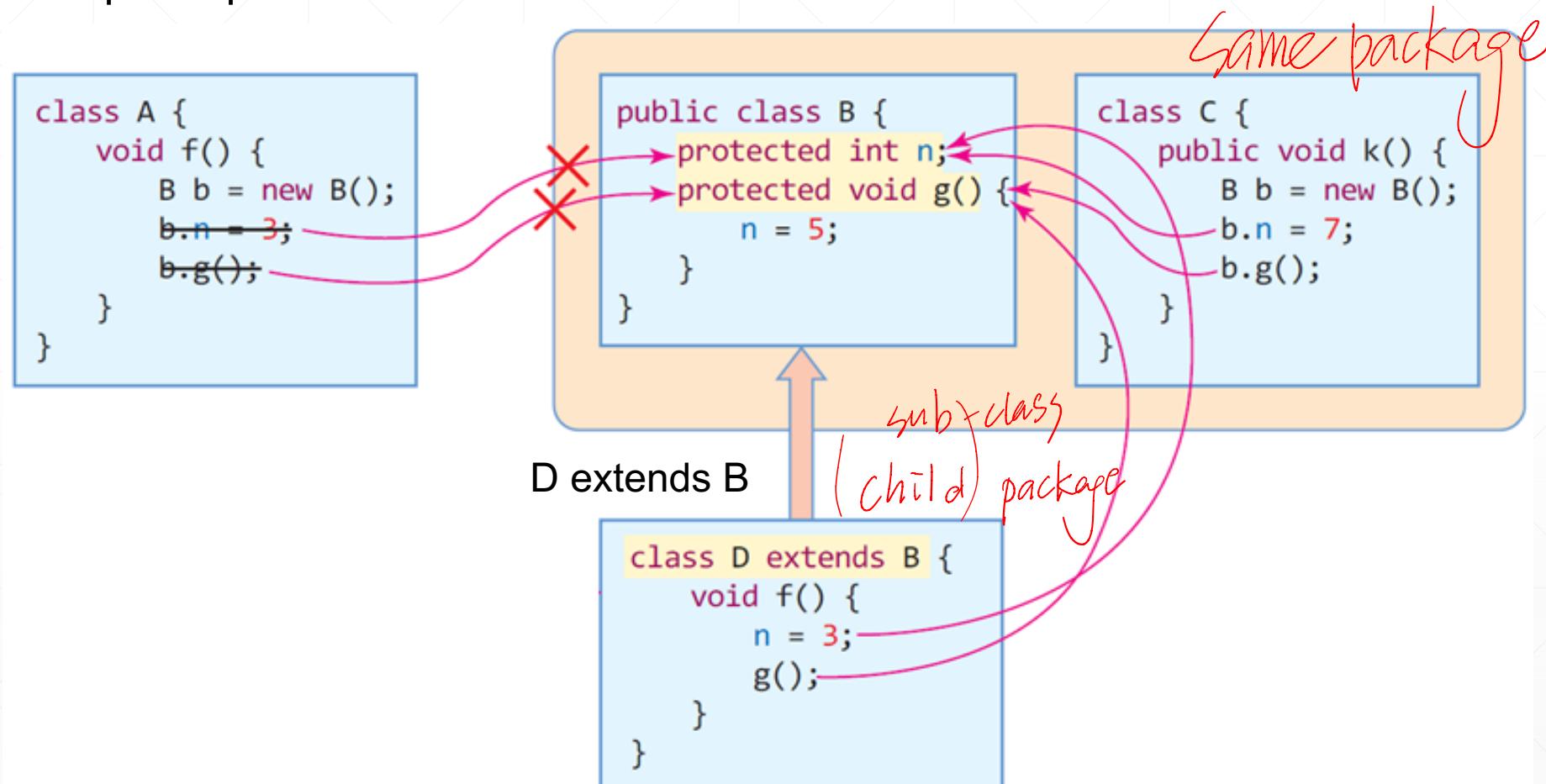


Method: Access Modifier

■ Member-level access modifier

- Example of protected modifiers

Accessor	Access modifier of a member			
	private	default	protected	public
Class in the same package	X	O	O	O
Class in another package	X	X	X	O



Method: Access Modifier (cont'd)

■ Example)

- Where a compile error occurs?

■ Hint

- Public: everybody
- Private: no one
- Default: in the same package

```
package seoultech.itm;

class Sample{
    public int publicNum;
    private int privateNum;
    int defaultNum;
}

public class ComLang {

    public static void main(String[] args) {
        Sample mySample = new Sample();
        mySample.publicNum = 10;
        <mySample.privateNum = 20;> error
        mySample.defaultNum = 30;
    }
}
```

Method: Access Modifier (cont'd)

■ One more thing

- Default classes can be included in
 - its own java file, or
 - the java file of the other class
- Public class MUST have its own java file

ComLang.java

```
package seoultech.itm;

class Sample{
    public int publicNum;
    private int privateNum;
    int defaultNum;
}

public class ComLang {

    public static void main(String[] args) {
        Sample mySample = new Sample();
        mySample.publicNum = 10;
        mySample.privateNum = 20;
        mySample.defaultNum = 30;
    }
}
```

Method: Setter and Getter

■ Public member

- Public interface exposed to external accessors
- NEVER (rarely) changed
- Deal with private members for getting/setting values

■ Private member

- Not exposed to external accessors
- Internal use only

```
package seoultech.itm;
```

```
class Sample {  
    public int publicNum;  
    private int privateNum;  
    int defaultNum;
```

```
    public int getPrivateNum() {  
        return privateNum;  
    }
```

Getter

```
    public void setPrivateNum(int privateNum) {  
        this.privateNum = privateNum;  
    }
```

Setter

```
public class ComLang {  
    public static void main(String[] args) {  
        Sample mySample = new Sample();  
        mySample.publicNum = 10;  
        mySample.setPrivateNum(20);  
        System.out.println(mySample.getPrivateNum());  
        mySample.defaultNum = 30;  
    }  
}
```



Method **Inheritance**

Inheritance: Concept

■ Inheritance in the real world

- Biological nature of parents is inherited to their descendants
- Parent can select who will inherit their wealth

■ Inheritance in the Java world

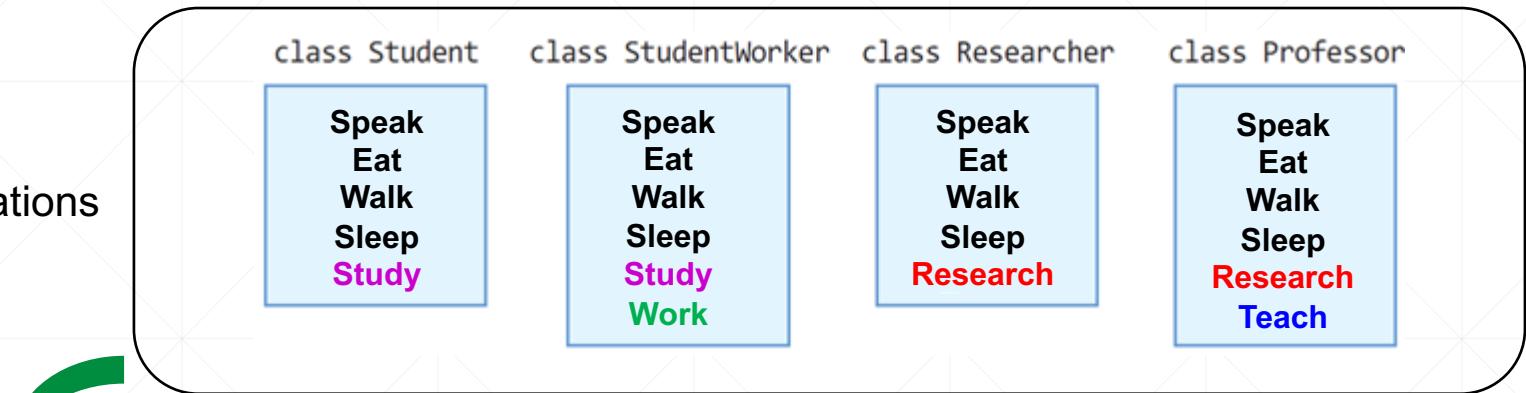
- Members of a parent class are inherited to their child classes
- Child can select who they wish to inherit!



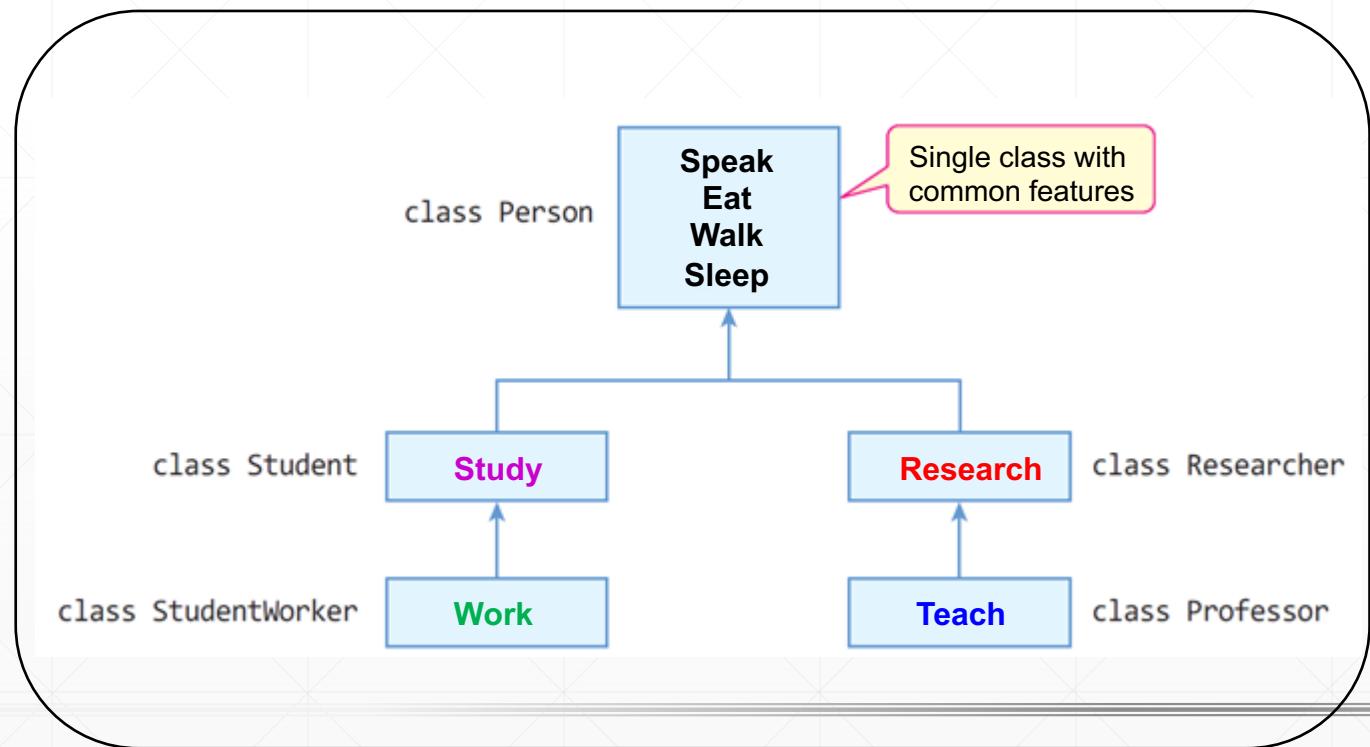
Inheritance: Concept (cont'd)

■ Example of Inheritance

4 Classes with duplicated members/implementations



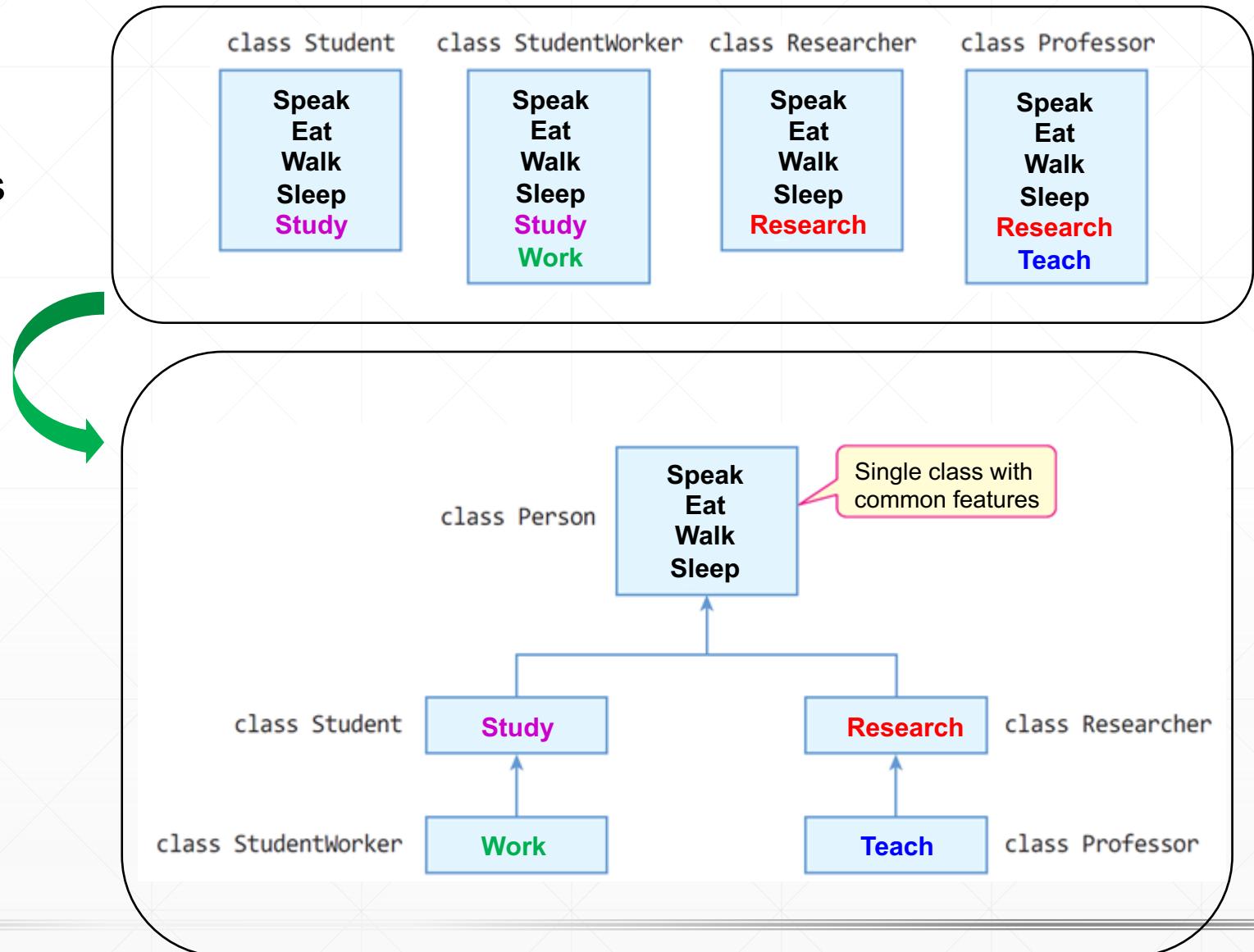
One class with common features
+
Specialized sub-classes without duplicated members



Inheritance: Concept (cont'd)

■ Advantages of Inheritance

- Reduced duplicated codes
- Better maintenance of classes
 - Hierarchical relationships
- Improved productivity
 - Class reuse and extension



Inheritance: Concept (cont'd)

■ Declaration of inheritance: “extends” keyword

```
public class Person {  
    ...  
}  
public class Student extends Person { // declares that Student inherits Person class  
    ...  
}  
public class StudentWorker extends Student { // declares that StudentWorker inherits Student  
    ...  
}
```

■ Sub (child) class

- A class that is derived from another class

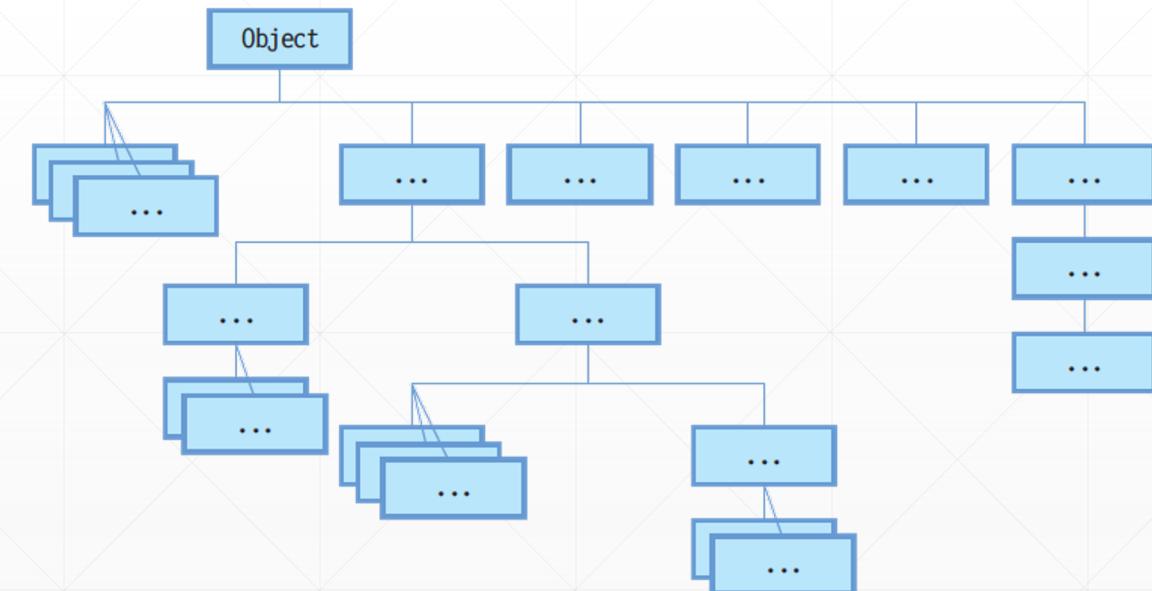
■ Super (parent) class

- A class from which the subclass is derived

Inheritance: Concept (cont'd)

■ Characteristics of Java inheritance

- Does not support multiple inheritance
- No limitation on the number of inheritance
 - E.g) Classes can be derived from classes that are derived from classes that are ..., and so on
- Every class is implicitly a subclass of Object class
 - The root: java.lang.Object
 - Automatically made by Java compiler



Inheritance: Concept (cont'd)

■ Example)

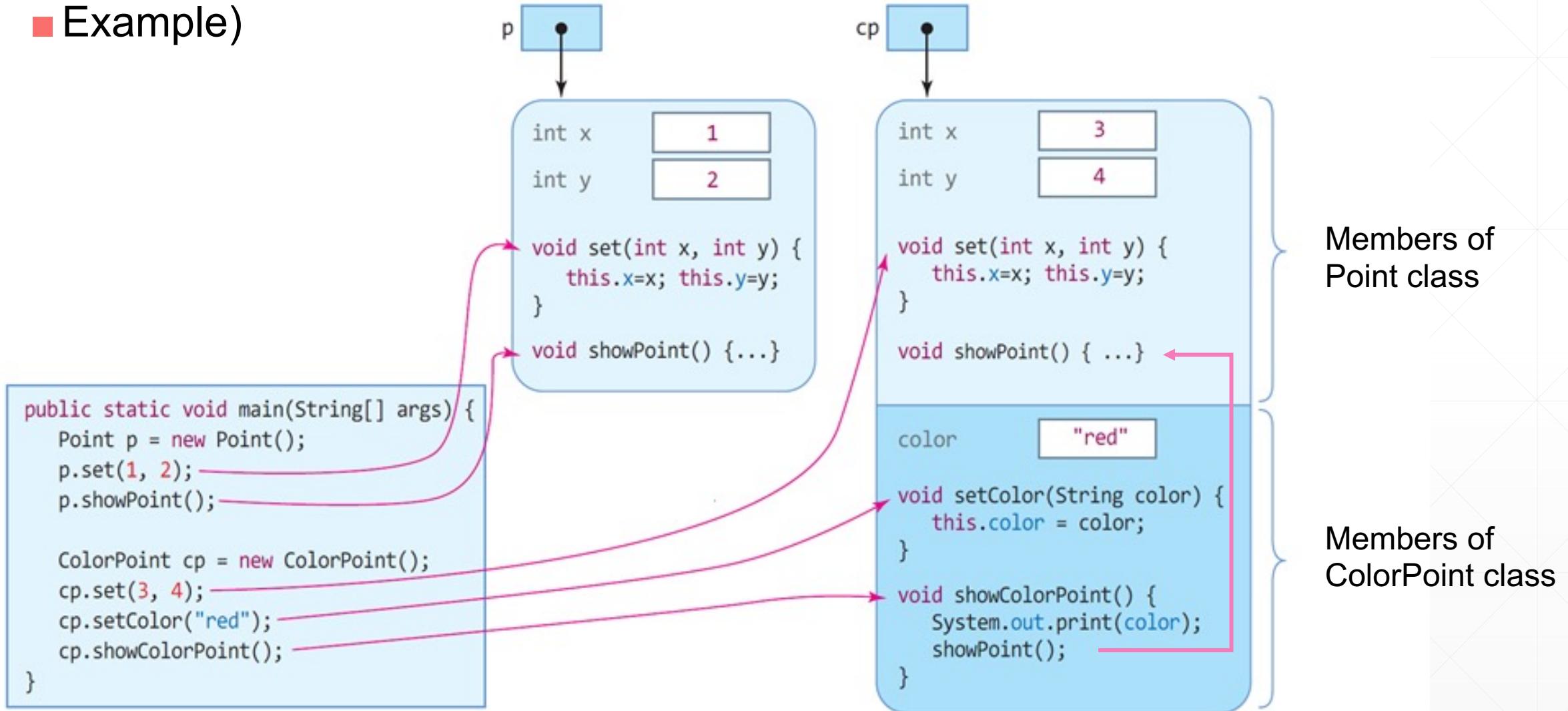
```
class Point {  
    private int x, y; ?  
    public void set(int x, int y) {  
        this.x = x; this.y = y;  
    }  
    public void showPoint() {  
        System.out.println("(" + x + "," + y + ")");  
    }  
}
```

```
// define class ColorPoint that inherits Point class  
class ColorPoint extends Point {  
    private String color;  
    public void setColor(String color) {  
        this.color = color;  
    }  
    public void showColorPoint() {  
        System.out.print(color);  
        showPoint(); // call showPoint() of Point class  
    }  
}
```

```
public class ColorPointEx {  
    public static void main(String [] args) {  
        Point p = new Point(); // instantiate Point object  
        p.set(1, 2); // call set() of Point class  
        p.showPoint();  
  
        ColorPoint cp = new ColorPoint(); // instantiate ColorPoint object  
        cp.set(3, 4); // call set() of Point class  
        cp.setColor("red"); // call setColor() of ColorPoint class  
        cp.showColorPoint();  
    }  
}
```

Inheritance: Concept (cont'd)

■ Example)

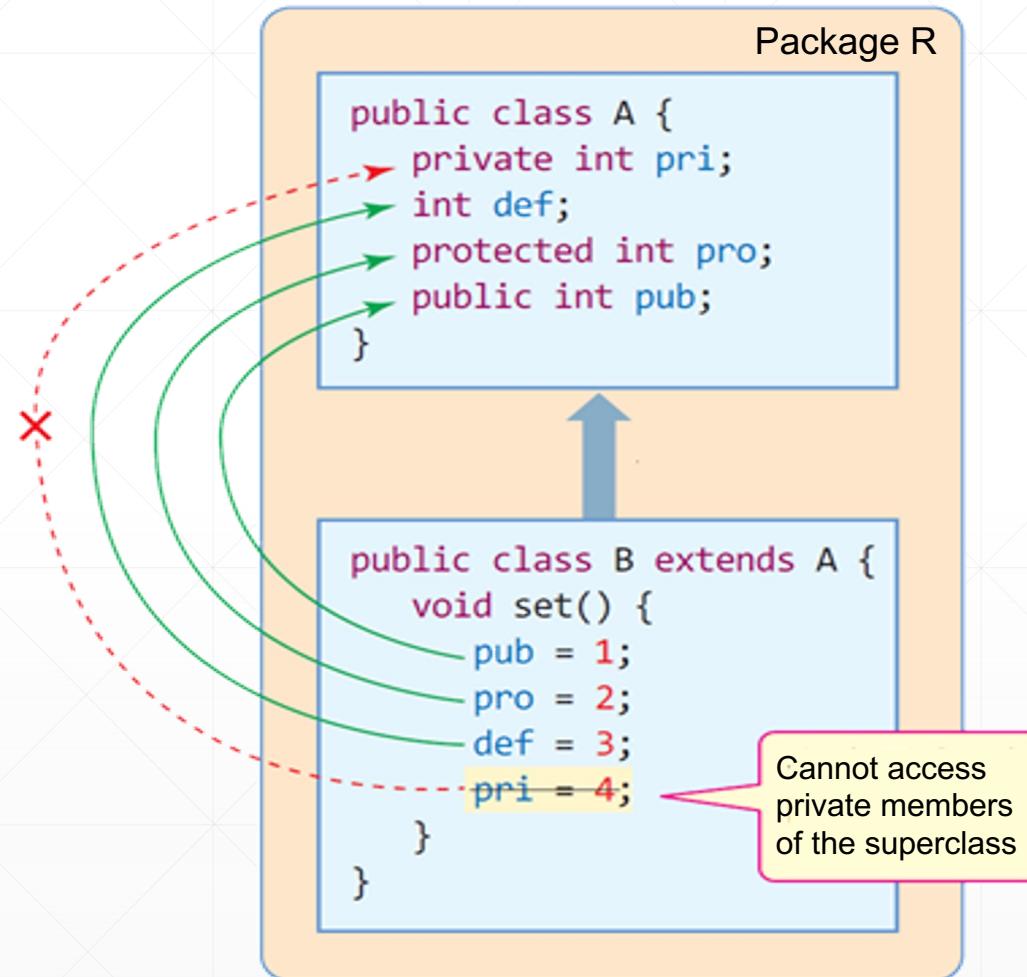


Inheritance: Access Modifier

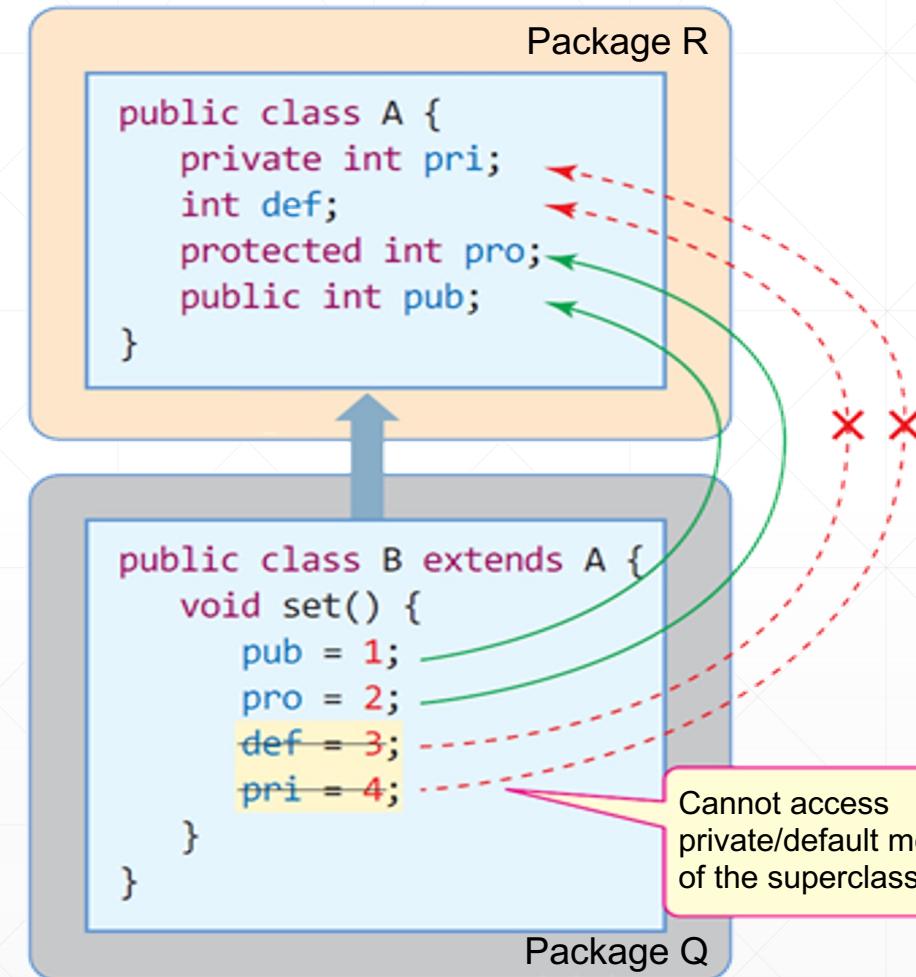
■ Access modifiers in the super class

- Public: all other classes can access these members
- Private
 - No one can access these members
 - Only other members inside the same class allowed
- Protected
 - All classes in the same package can access these members
 - Child class outside the package can access these members
- Default
 - All classes in the same package can access these members

Inheritance: Access Modifier (cont'd)



In the same package



Child is outside the package

Inheritance: Access Modifier (cont'd)

■ Example)

```
class Person {  
    private int weight;  
    int age;  
    protected int height;  
    public String name;  
  
    public void setWeight(int weight) {  
        this.weight = weight;  
    }  
    public int getWeight() {  
        return weight;  
    }  
}
```

```
public class InheritanceEx {  
    public static void main(String[] args) {  
        Student s = new Student();  
        s.set();  
    }  
}
```

```
class Student extends Person {  
    public void set() {  
        age = 30; // OK  
        name = "Jinwoo"; // OK  
        height = 175; // OK  
        // weight = 99; // Error. Private member of superclass  
        setWeight(99); // OK  
    }  
}
```

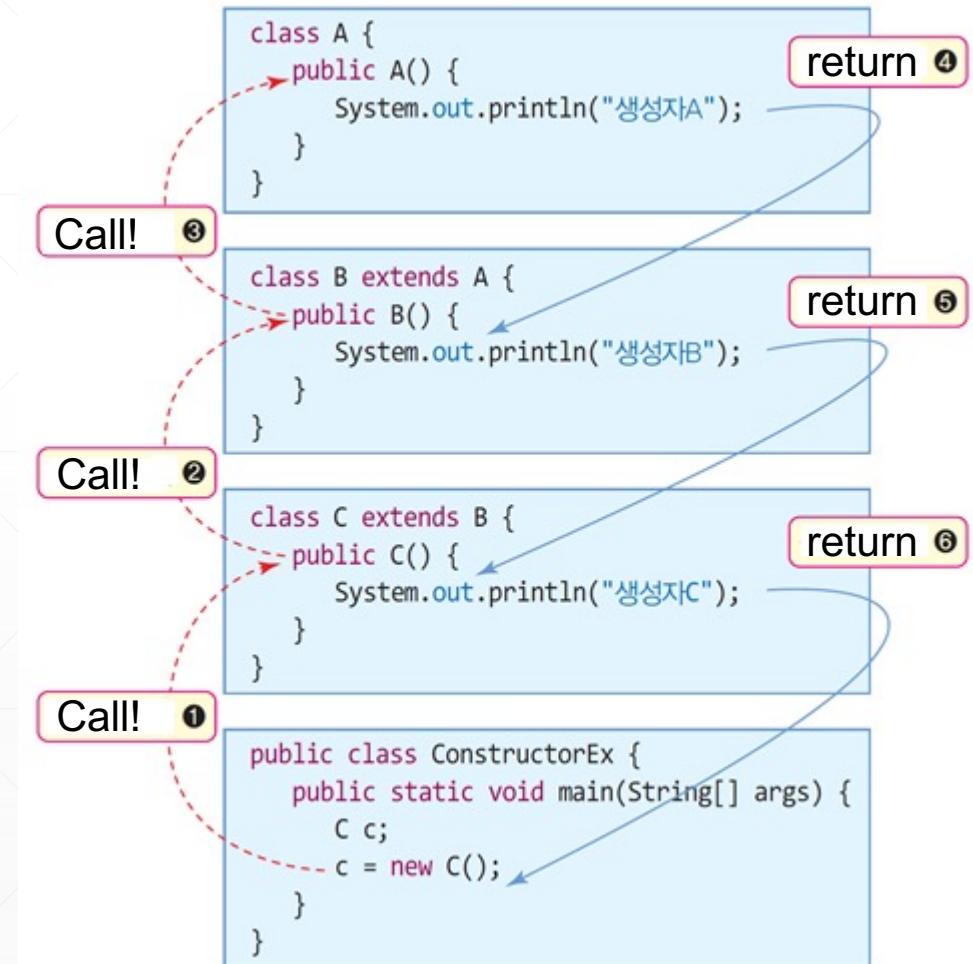
Inheritance: Constructor

■ What happens when a new object of a subclass is instantiated?

- Constructor() of Superclass is invoked first!
- Constructor() of Subclass is then invoked

■ What will be printed out from this example?

A
B
C



Inheritance: Constructor (cont'd)

- Which constructor of a superclass will be invoked?
 - Multiple constructors can be defined in both super/sub classes
- Constructor of a subclass **MUST** choose the super-constructor to invoke
 - Implicit invocation
 - Explicit invocation

Inheritance: Constructor (cont'd)

■ Implicit invocation of a super-constructor

- When a subclass constructor does not specify which one to invoke
- Java compiler automatically inserts a call to the no-argument constructor (i.e., default constructor) of the superclass

```
class A {  
    ➔ public A() {  
        System.out.println("생성자A");  
    }  
    public A(int x) {  
        ....  
    }  
}  
  
class B extends A {  
    ➔ public B() {  
        System.out.println("생성자B");  
    }  
}  
  
public class ConstructorEx2 {  
    public static void main(String[] args) {  
        B b;  
        b = new B();  
    }  
}
```

Inheritance: Constructor (cont'd)

■ Implicit invocation of a super-constructor

- Exception) What if no default constructor defined in the superclass?
 - If a superclass has constructors with parameters, then no default constructor provided by Compiler
- There will be a compile error!

“Implicit super constructor A() is undefined.
Must explicitly invoke another constructor”
- Default constructor must be defined!

```
class A {  
    public A(int x) {  
        System.out.println("생성자A");  
    }  
}  
  
class B extends A {  
    public B() {  
        System.out.println("생성자B");  
    }  
}  
  
public class ConstructorEx2 {  
    public static void main(String[] args) {  
        B b;  
        b = new B();  
    }  
}
```

Inheritance: Constructor (cont'd)

■ Implicit invocation of a super-constructor

- Exception) What if a constructor with parameters of a subclass invoked?
 - Which super-constructor in this case will be selected?
- Only default constructor of a super-class invoked!
 - This is also a case of “Implicit Invocation”

```
class A {  
    public A() {  
        System.out.println("생성자A");  
    }  
    public A(int x) {  
        System.out.println("매개변수생성자A");  
    }  
}
```

```
class B extends A {  
    public B() {  
        System.out.println("생성자B");  
    }  
    public B(int x) {  
        System.out.println("매개변수생성자B");  
    }  
}
```

```
public class ConstructorEx3 {  
    public static void main(String[] args) {  
        B b;  
        b = new B(5);  
    }  
}
```

Inheritance: Constructor (cont'd)

■ Explicit invocation of a super-constructor

- When a subclass constructor invokes a specific super-constructor using “super()” method call
- Any of superclass constructors can be selected
 - Super(): the default super constructor
 - Super(params): the super constructor with parameters
- Explicit super-constructor invocation must be in the first line of the sub-constructor
 - Same to that of this() invocation

Inheritance: Constructor (cont'd)

■ Explicit invocation of a super-constructor

- Any of superclass constructors can be selected
 - Super(): the default super constructor
 - Super(params): the super constructor with parameters
- Explicit super-constructor invocation must be in the first line of the sub-constructor
 - Same to that of this() invocation

```
class A {  
    public A() {  
        System.out.println("생성자A");  
    }  
    public A(int x) {  
        System.out.println("매개변수생성자A" + x);  
    }  
}
```

```
class B extends A {  
    public B() {  
        System.out.println("생성자B");  
    }  
    public B(int x) {  
        super(x);  
        System.out.println("매개변수생성자B" + x);  
    }  
}
```

```
public class ConstructorEx4 {  
    public static void main(String[] args) {  
        B b;  
        b = new B(5);  
    }  
}
```

Inheritance: Constructor (cont'd)

■ Explicit invocation of a super-constructor

```
class Point {  
    private int x, y;  
    public Point() {  
        this.x = this.y = 0;  
    }  
    public Point(int x, int y) {  
        this.x = x; this.y = y;  
    }  
    public void showPoint() {  
        System.out.println("(" + x + "," + y + ")");  
    }  
}  
  
class ColorPoint extends Point {  
    private String color;  
    public ColorPoint(int x, int y, String color) {  
        super(x, y);  
        this.color = color;  
    }  
    public void showColorPoint() {  
        System.out.print(color);  
        showPoint();  
    }  
}
```

```
public class SuperEx {  
    public static void main(String[] args) {  
        ColorPoint cp = new ColorPoint(5, 6, "blue");  
        cp.showColorPoint();  
    }  
}
```

Q&A

■ Next week (eClass video)

- Up/Downcasting
- Method Overriding

Computer Language



OOP 3: Casting and Overriding

Agenda

- Casting
- Method Overriding

Class: Up/Downcasting

■ Type conversion between classes

- Promotion/casting concept

■ Upcasting

- Type conversion from sub-class to super-class

```
super class  
class Person { ... }  
class Student extends Person { ... }  
sub class  
Student s = new Student();  
Person p = s; // Upcasting, automatic conversion
```

- Upcasting reference can only access the members of a superclass

Class: Up/Downcasting (cont'd)

■ Upcasting

- Type conversion from sub-class to super-class
- Upcasting reference can only access the members of a superclass

```
class Person{
    String name;
    String id;

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

class Student extends Person{
    String grade;
    String department;

    public Student(String name){
        super(name);
    }
}
```

```
public class UpcastingEx {
    public static void main(String[] args) {
        Person p;
        Student s = new Student("Jinwoo");
        p = s; //upcasting
        System.out.println(p.name);
        //p.grade = "F";
        //p.department = "ITM";
    }
}
```

Class: Up/Downcasting (cont'd)

■ Downcasting

- Type conversion from super-class to sub-class
- MUST be explicitly made by a developer

```
class Person { ... }
class Student extends Person { ... }

...
Person p = new Student("Jinwoo"); // upcasting

...
Student s = (Student) p; // downcasting (casting from Person to Student)
```

- Why downcasting?
 - When we wish to use the members of a subclass!

Class: Up/Downcasting (cont'd)

■ Downcasting

- Type conversion from super-class to sub-class
- MUST be explicitly made by a developer

```
public class UpcastingEx {  
    public static void main(String[] args) {  
        Person p = new Student("Jinwoo");  
        System.out.println(p.name);  
        //p.grade = "F";  
        //System.out.println(p.grade);  
        < Student s = (Student) p; > downcasting  
        System.out.println(s.name);  
        s.grade = "A";  
        System.out.println(s.grade);  
        //p.grade = "F";  
        //p.department = "ITM";  
    }  
}
```

Downcasting (from Person to Student)

Class: Up/Downcasting (cont'd)

■ A lot of subclasses from a single superclass available

- Invalid downcasting results in an error!

No Upcasting ... Parent parent = new Parent(); 'parent' instance is just related to 'Parent' class, not 'Child' class.
Nothing for downcasting Child child = (Child) parent; Impossible!

- It is impossible to infer the actual type of a upcasting reference

Before Downcasting, what is actual data type ?

refer superclass, has superclass object.

refer superclass, has subclass object. by upcasting.

■ instanceof operator

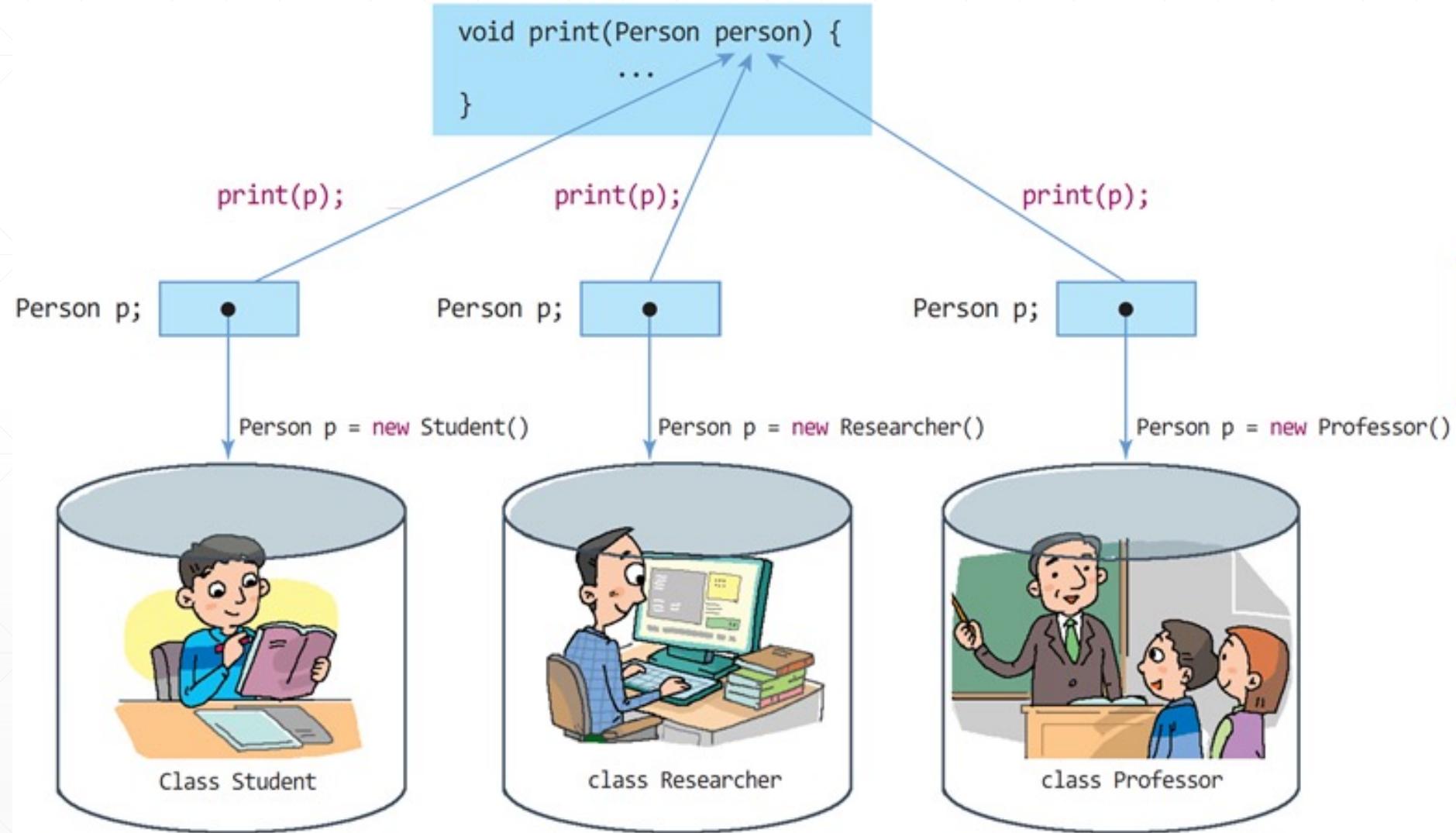
- Used to determine the type of an object
- Returns true / false

objRef **instanceof** Classtype

Person ... Super
| |
Student ≠ Researcher ... Sub

Person p = new Researcher();
Student s = (Student) p;
⇒ impossible

Class: Up/Downcasting (cont'd)



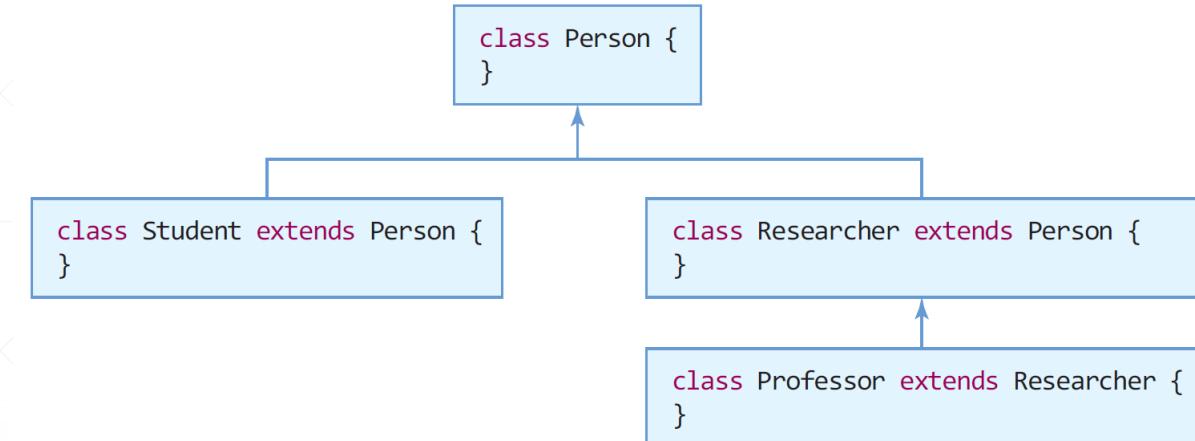
Class: Up/Downcasting (cont'd)

■ Example of using instanceof operator

```
Person jee= new Student();
Person kim = new Professor();
Person lee = new Researcher();
if (jee instanceof Person)          // true
if (jee instanceof Student)         // true
if (kim instanceof Student)         // false
if (kim instanceof Professor)       // true
if (kim instanceof Researcher)      // true
if (lee instanceof Professor)        // false
```

```
if(3 instanceof int)           // Error!
```

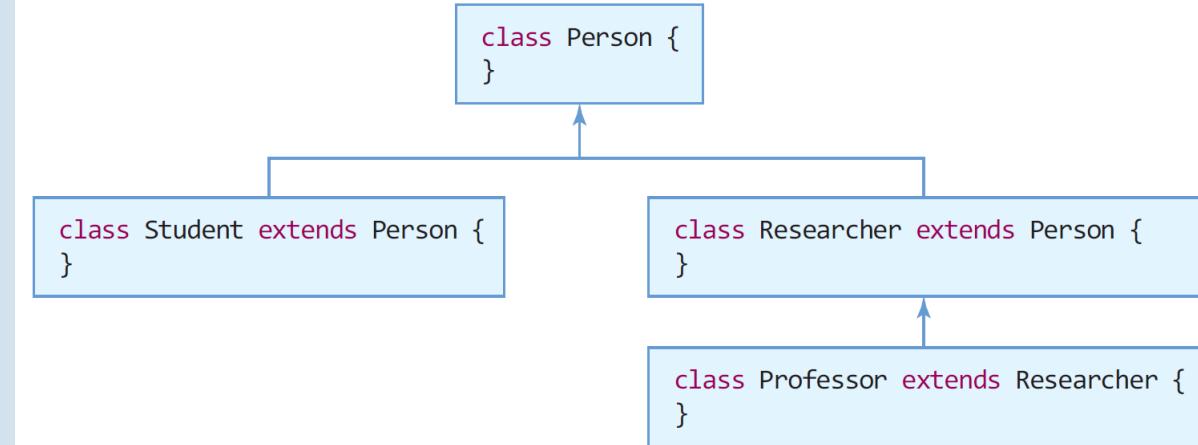
```
if("java" instanceof String)    // true
```



Class: Up/Downcasting (cont'd)

■ Example of using instanceof operator

```
class Person {}  
class Student extends Person {}  
class Researcher extends Person {}  
class Professor extends Researcher {}  
  
public class InstanceOfEx {  
    static void print(Person p) {  
        if(p instanceof Person)  
            System.out.print("Person ");  
        if(p instanceof Student)  
            System.out.print("Student ");  
        if(p instanceof Researcher)  
            System.out.print("Researcher ");  
        if(p instanceof Professor)  
            System.out.print("Professor ");  
        System.out.println();  
    }  
    public static void main(String[] args) {  
        System.out.print("new Student() ->"); print(new Student()); T T F F ... Person, Student  
        System.out.print("new Researcher() ->"); print(new Researcher()); T F T F ... Person, Researcher  
        System.out.print("new Professor() ->"); print(new Professor()); T F T T ... Person, Researcher, Professor  
    }  
}
```



T T F F ... Person, Student
T F T F ... Person, Researcher
T F T T ... Person, Researcher, Professor

Method Overriding

■ Redefinition of superclass's method in the subclass

- Same method signature, but different behaviors

```
class Shape {  
    public void draw() {  
        System.out.println("Shape");  
    }  
}
```

```
class Line extends Shape {  
    public void draw() {  
        System.out.println("Line");  
    }  
}
```

```
class Rect extends Shape {  
    public void draw() {  
        System.out.println("Rect");  
    }  
}
```

```
class Circle extends Shape {  
    public void draw() {  
        System.out.println("Circle");  
    }  
}
```

Overriding!

Method Overriding (cont'd)

- Redefinition of superclass's method in the subclass

- Same method signature, but different behaviors

- Achieves polymorphism with inheritance

- Same interface, but different behaviors
 - Line class draws a line using draw() interface
 - Circle class draws a circle using draw() interface
 - Rect class draws a rectangle using draw() interface

Method Overriding (cont'd)

■ Example of Polymorphism using method overriding

```
class Shape {  
    public void draw() {  
        System.out.println("Shape");  
    }  
}  
  
class Line extends Shape {  
    public void draw() { // method overriding!  
        System.out.println("Line");  
    }  
}  
  
class Rect extends Shape {  
    public void draw() { // method overriding!  
        System.out.println("Rect");  
    }  
}  
  
class Circle extends Shape {  
    public void draw() { // method overriding!  
        System.out.println("Circle");  
    }  
}
```

```
public class MethodOverridingEx {  
    static void paint(Shape p) {  
        p.draw(); // call overridden draw()  
    }  
  
    public static void main(String[] args) {  
        Line line = new Line();  
        paint(line);  
        paint(new Shape());  
        paint(new Line());  
        paint(new Rect());  
        paint(new Circle());  
    }  
}
```

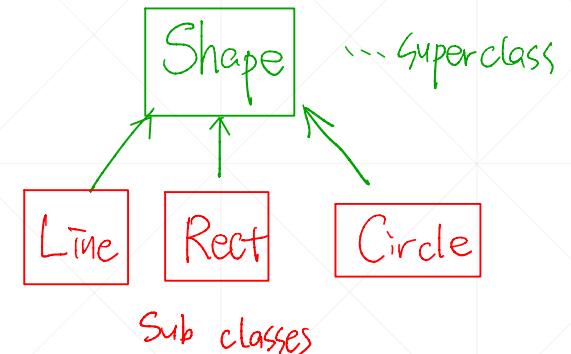
Method Overriding (cont'd)

■ Which method should be invoked?

- For input parameter with Shape type, there can be a lot of variations!
- When this association made?

```
public class MethodOverridingEx {  
    static void paint(Shape p) {  
        p.draw(); // call overridden draw()  
    }  
  
    public static void main(String[] args) {  
        Line line = new Line();  
        paint(line); // Line  
        paint(new Shape()); // Shape  
        paint(new Line()); // Line  
        paint(new Rect()); // Rect  
        paint(new Circle()); // Circle  
    }  
}
```

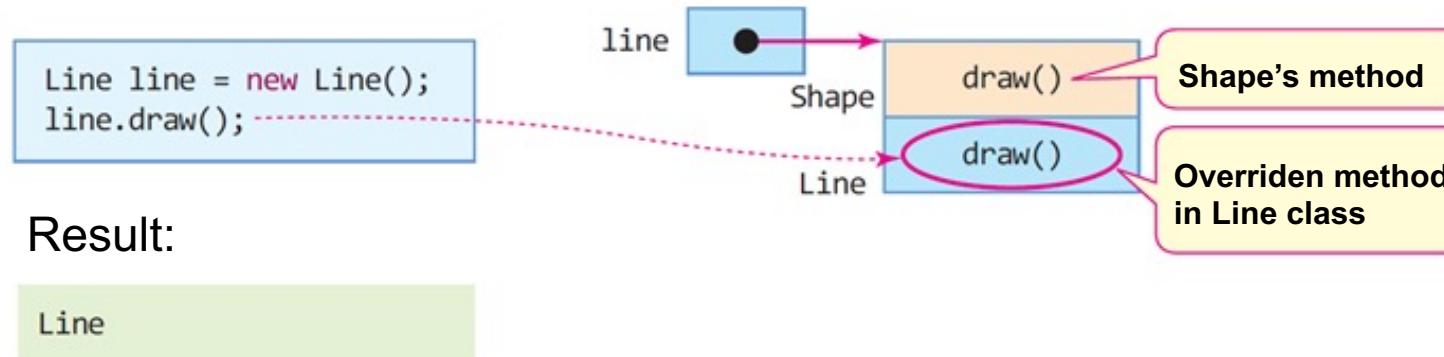
Shape's draw()
Line's draw()
Rect's draw()
Circle's draw()



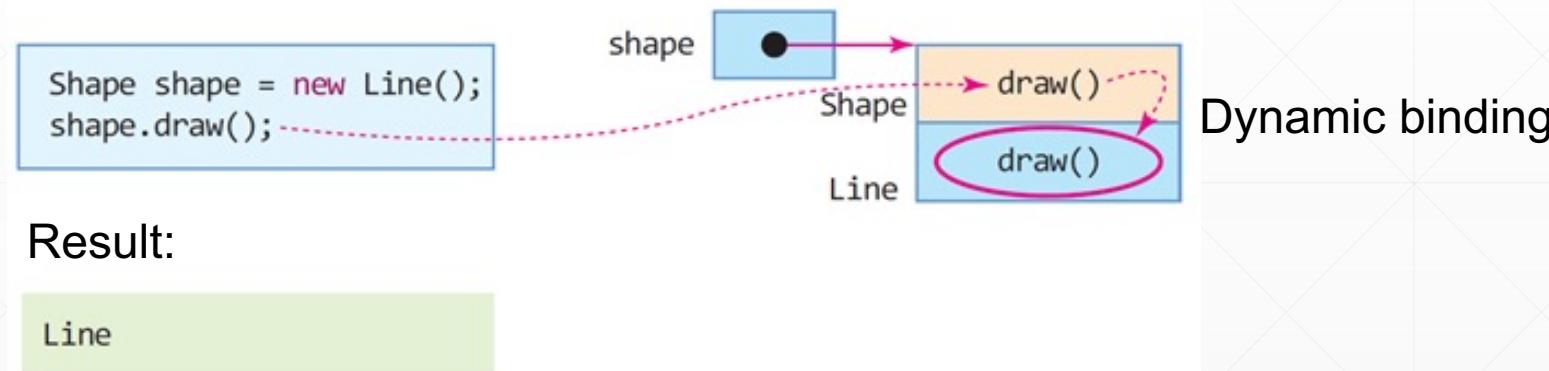
Method Overriding (cont'd)

■ Which method should be invoked?

- Calling an overridden method from the subclass



- Calling an overridden method from the (upcasting) superclass



Method Overriding (cont'd)

■ Dynamic binding

- Runtime association of method calling
- “Who should be invoked?” is determined at runtime

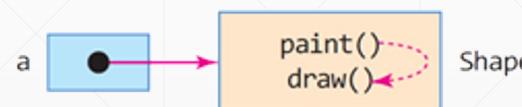
When invoking instance method or specifically for that overwritten method, target method to be invoked will be determined at runtime, not compile time.

Call some methods in the context of super class
→ dynamic binding, JVM automatically find operating methods defined in sub class

```
public class Shape {  
    protected String name;  
    public void paint() {  
        draw();  
    }  
    public void draw() {  
        System.out.println("Shape");  
    }  
    public static void main(String [] args) {  
        Shape a = new Shape();  
        a.paint();  
    }  
}
```

Result:

Shape



behavior of each sub class will be performed. -- type of runtime polymorphism.

```
class Shape {  
    protected String name;  
    public void paint() {  
        draw();  
    }  
    public void draw() {  
        System.out.println("Shape");  
    }  
}  
public class Circle extends Shape {  
    @Override  
    public void draw() {  
        System.out.println("Circle");  
    }  
    public static void main(String [] args) {  
        Shape b = new Circle();  
        b.paint();  
    }  
}
```

Result:

Circle



Method Overriding (cont'd)

■ Static binding

Static keyword \Rightarrow cannot be overwritten!

- Compile-time association of method calling Only single behavior will be performed for all the cases.
- “Who should be invoked?” is determined at compile time (e.g., static method)

```
class Shape {  
    static void clear(){ System.out.println("Clear!"); }  $\Rightarrow$  Determined to invoke when clear()  
    void draw() { System.out.println("Shape"); }      method is called in compile time  
}  
  
class Line extends Shape {  
    static void clear(){ System.out.println("Line Clear!"); }  
    void draw() { System.out.println("Line"); }  
}  
  
class Rect extends Shape {  
    static void clear(){ System.out.println("Rect Clear!"); }  
    void draw() { System.out.println("Rect"); }  
}  
  
class Circle extends Shape {  
    static void clear(){ System.out.println("Circle Clear!"); }  
    void draw() { System.out.println("Circle"); }  
}
```

public class MethodOverridingEx {
 static void paint(Shape p){ p.draw(); }
 static void clear(Shape p){ p.clear(); }

 public static void main(String[] args) {
 Line line = new Line();
 paint(line);
 paint(new Shape());
 paint(new Line());
 paint(new Rect());
 paint(new Circle());

 clear(line);
 clear(new Shape());
 clear(new Line());
 clear(new Rect());
 clear(new Circle());
 }
}

Dynamic binding

Line
Shape
Line
Rect
Circle

\Rightarrow "Clear!"
... only print out
"Clear!"

Static binding

Method Overriding (cont'd)

■ Example)

- An array containing various payment methods
- Process a series of payments using abstraction and polymorphism

```
class Payment {  
    void pay(int money) { System.out.println("Payment!"); }  
}  
  
class Cash extends Payment {  
    void pay(int money) { System.out.println("Success!" + money + " Won paid"); }  
}  
  
class Bitcoin extends Payment {  
    void pay(int money) { System.out.println("Fail! Coin destroyed!"); }  
}  
  
class Credit extends Payment {  
    void pay(int money) { System.out.println("Success! Payment made with your card!"); }  
}
```

Method Overriding (cont'd)

■ Example)

- An array containing various payment methods
- Process a series of payments using abstraction and polymorphism

```
public class MethodOverridingEx {  
    static void purchase(Payment[] pay){  
        for (Payment s: pay){  
            s.pay(1000);  
        }  
    }  
  
    public static void main(String[] args) {  
        Payment[] myPayments = new Payment[3];  
        myPayments[0] = new Cash();  
        myPayments[1] = new Bitcoin();  
        myPayments[2] = new Credit();  
  
        purchase(myPayments);  
    }  
}
```

Method Overriding (cont'd)

■ Method Overloading vs Method Overriding

	Overloading	Overriding
Declaration	Multiple definition of methods with the same name	Re-defining superclass's method in the subclass
Relationship	In the same class	Inheritance
Purpose	Improved usability through the methods with the same name Compile-time polymorphism	Re-define subclass specific behaviors Runtime polymorphism
Condition	Same method name Different number/type of arguments	Method signature (name, arguments, return type) must be same
binding	Static binding	Dynamic binding

Q&A

■ Next week

- Midterm exam (Closed written test)
- 19/Apr, 15:00 ~ 17:00