

Computer Language



Introduction to Java



Computer: Hardware



Computer: Software



Computer: Programming Language

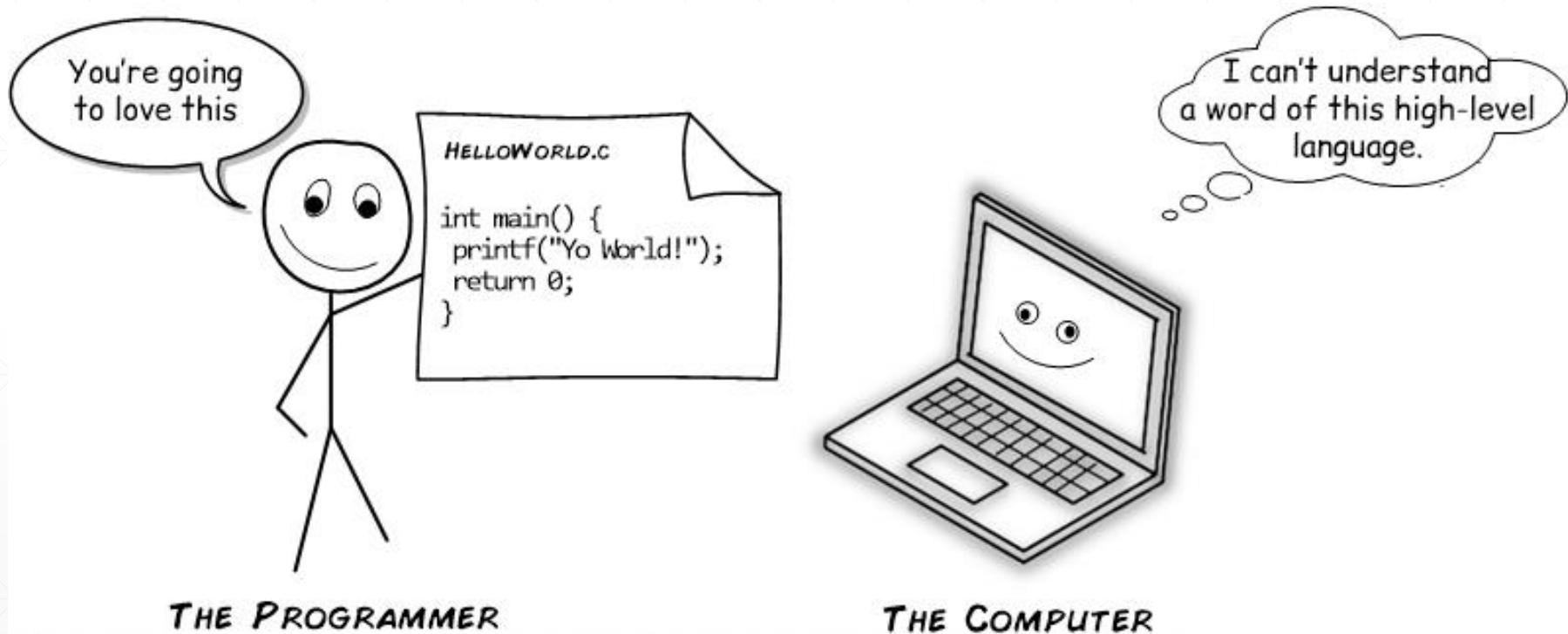
■ Machine language

- Machine-readable code
- Binary representation (0 & 1)
- CPU only understands this language

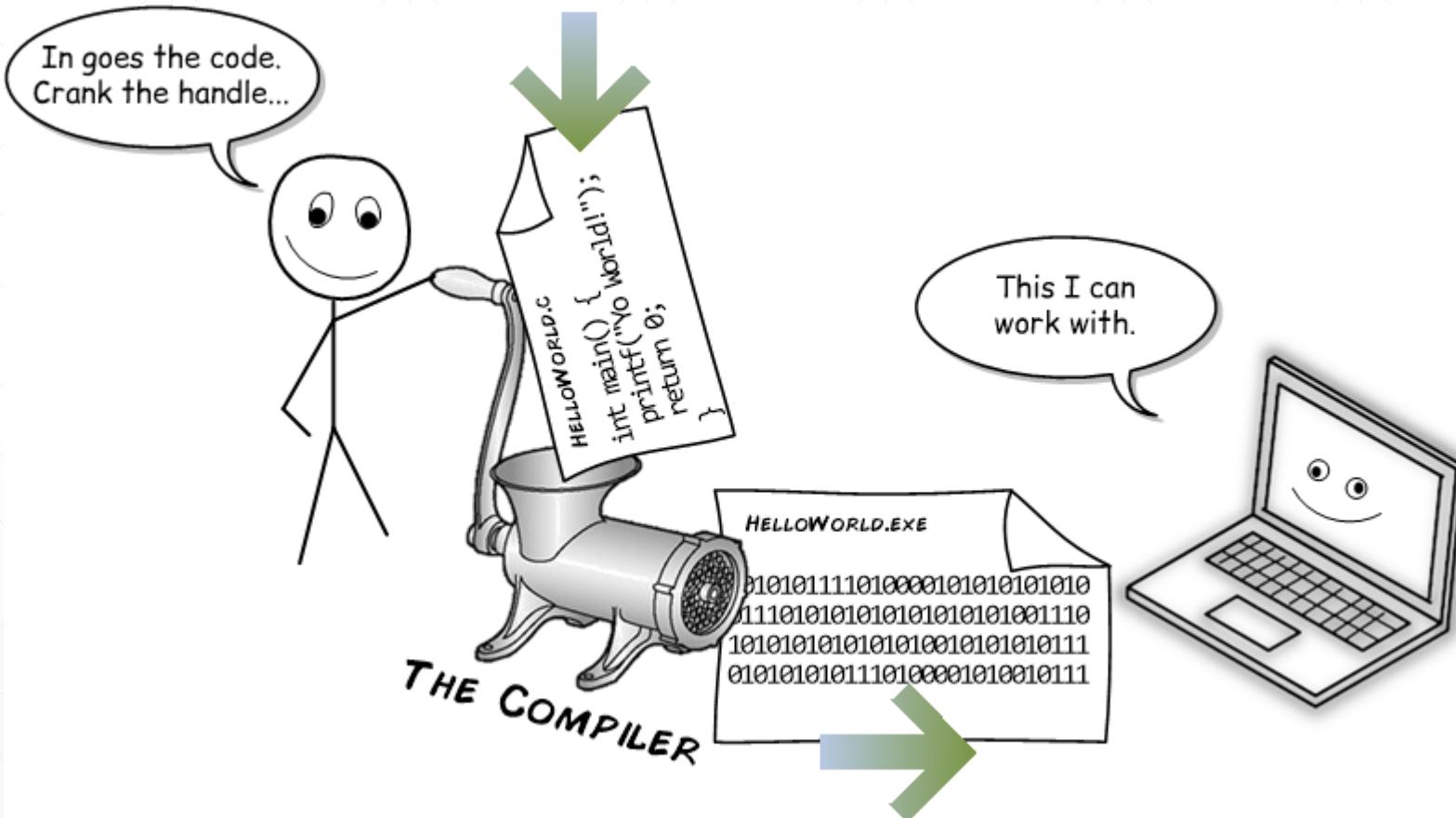
■ (High-level) Programming language

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

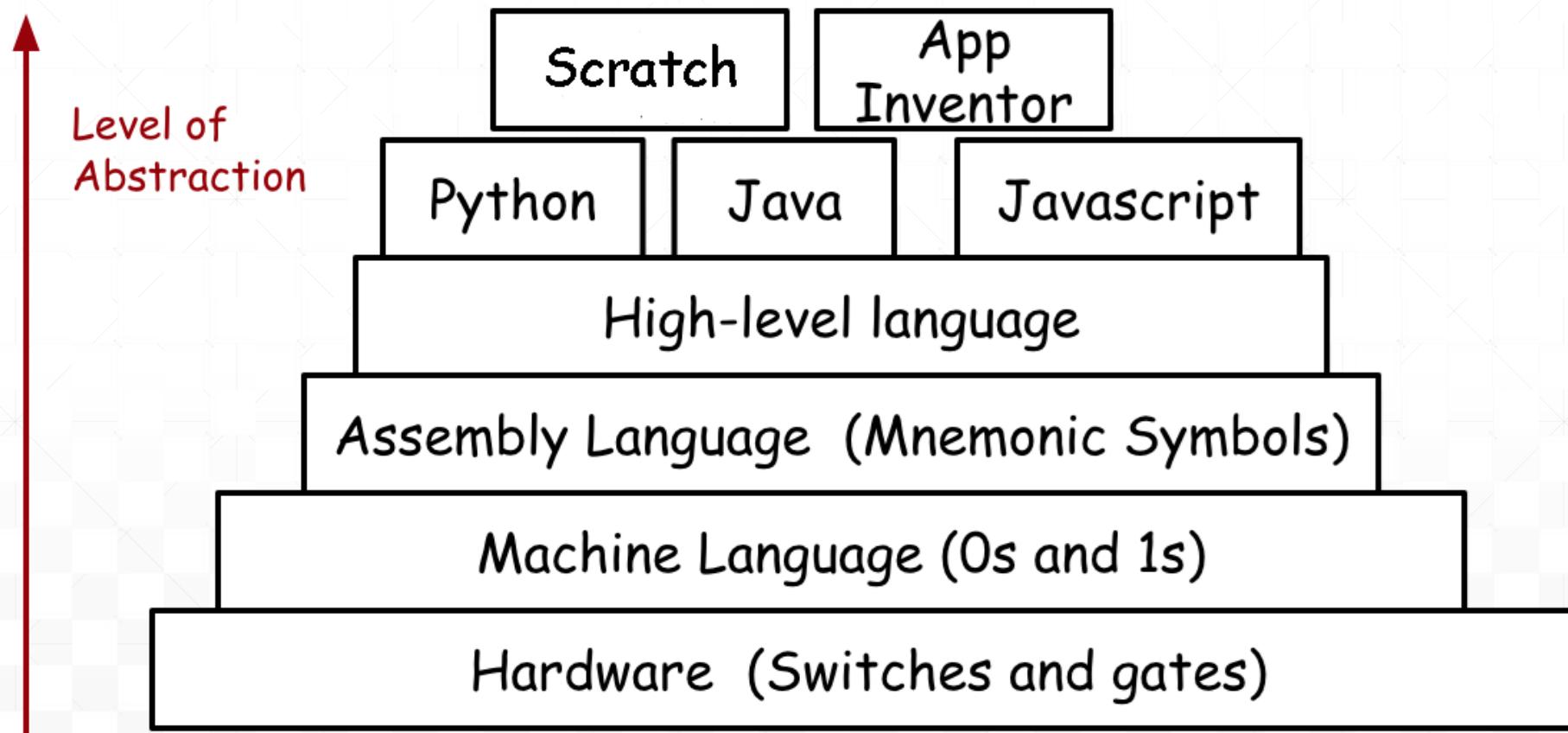
Computer: Programming Language (cont'd)



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



James Gosling

- Father of Java programming language

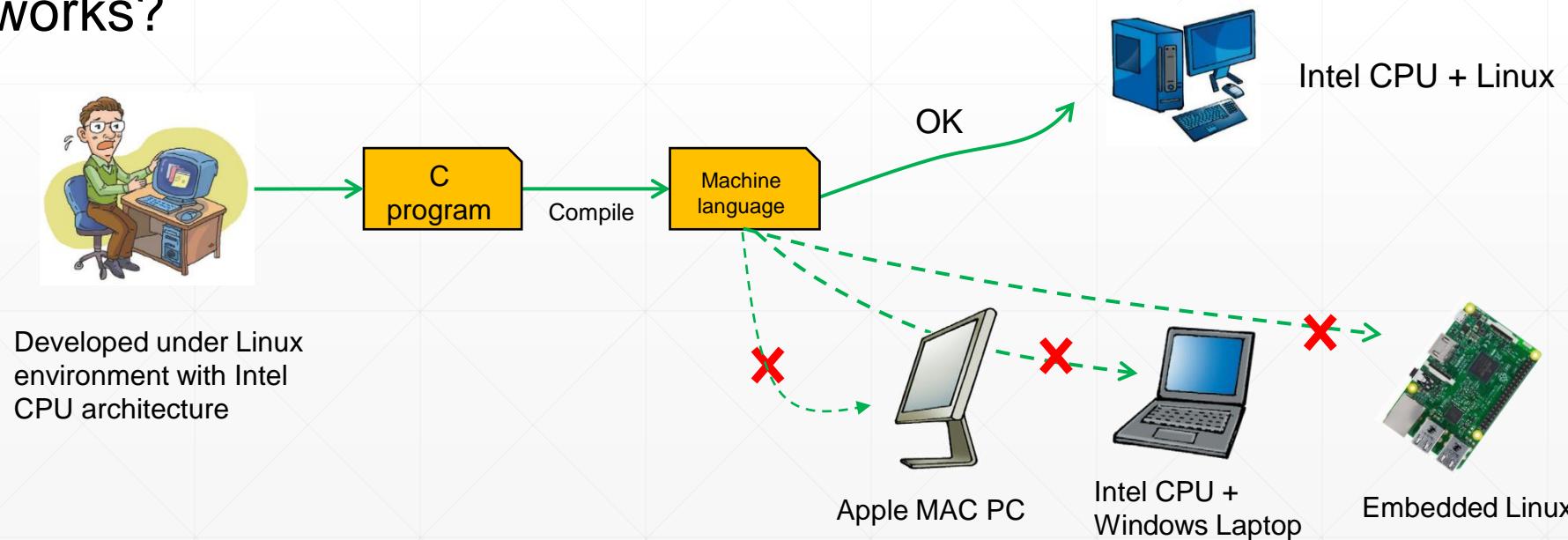


Java (cont'd)

■ WORA (Write Once Run Anywhere)

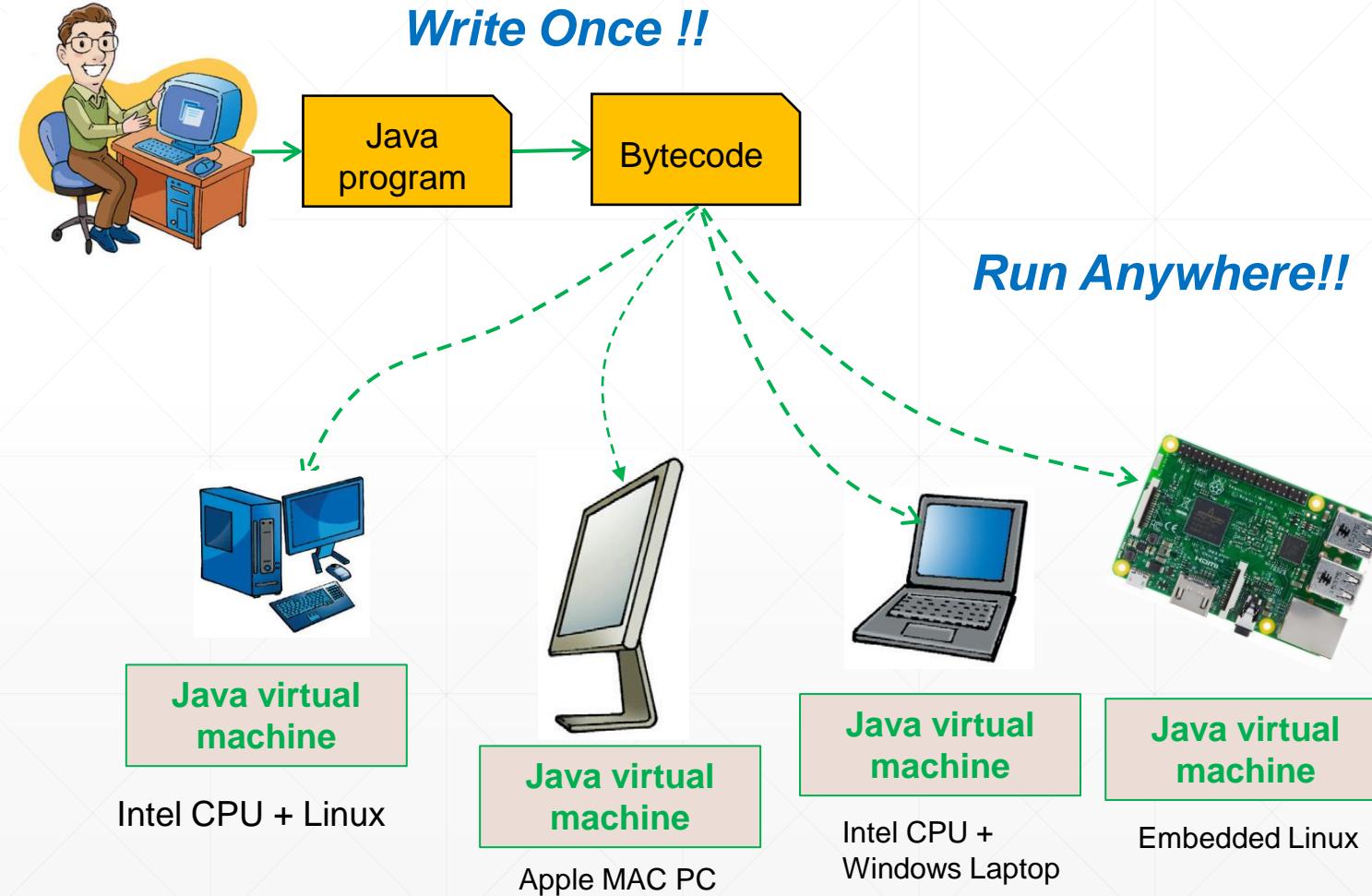
- Platform independent (cross-platform)
 - Weakness of traditional languages like C, C++, etc.
- A program written in Java can be executed on any architecture

■ How C works?



Java (cont'd)

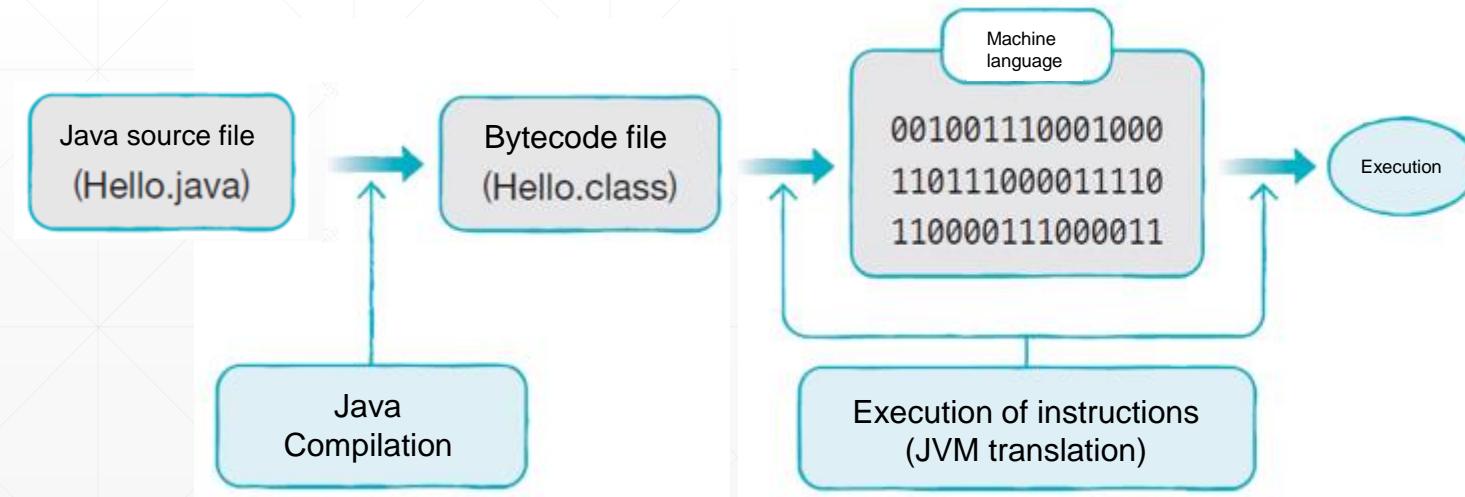
■ How Java works?



Java (cont'd)

■ WORA (Write Once Run Anywhere)

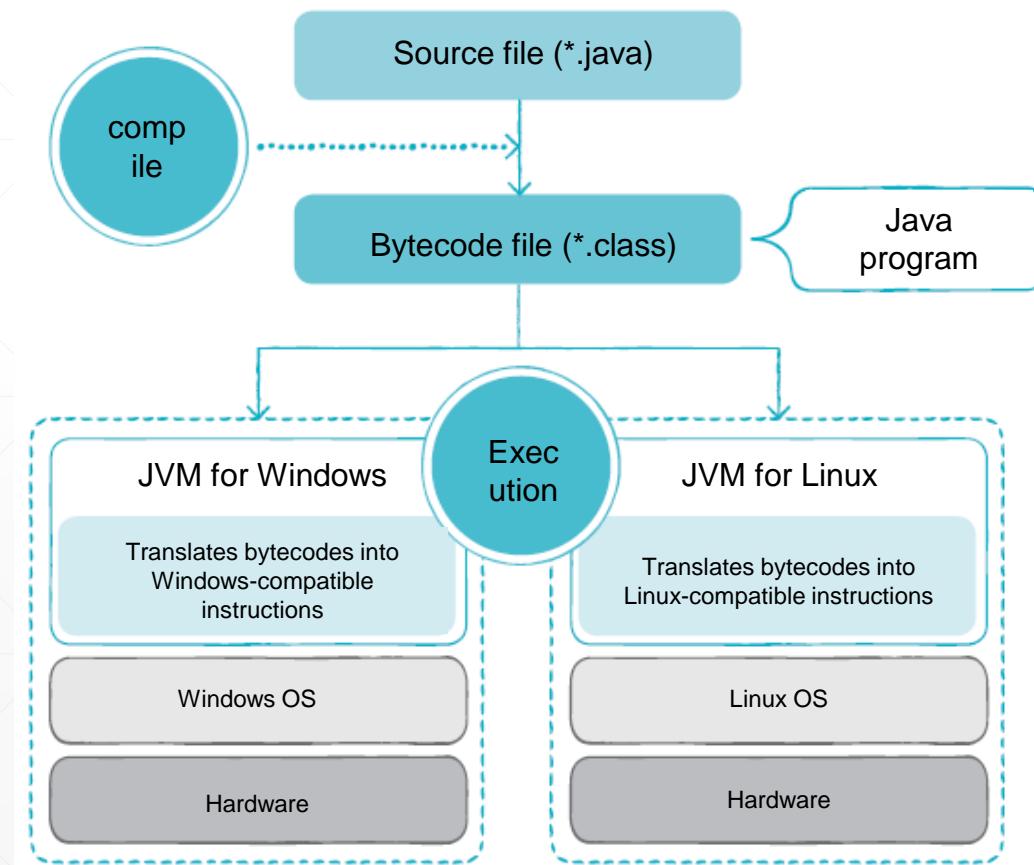
- Java programs consist of byte code files (.class), not complete machine language
 - Bytecode files cannot be executed directly on the OS
- Java Virtual Machine (JVM) translates and executes complete machine language



Java (cont'd)

■ Java Virtual Machine (JVM)

- Provides an equivalent Java execution environment
- JVM itself is platform dependent



Java (cont'd)

- Object-oriented programming (OOP)

- Encapsulation
- Inheritance
- Polymorphism
- ...

- 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 /* ... */

Computer Language



Variables & Types

Agenda

- Variables
- Types

Variables

Types

Variables

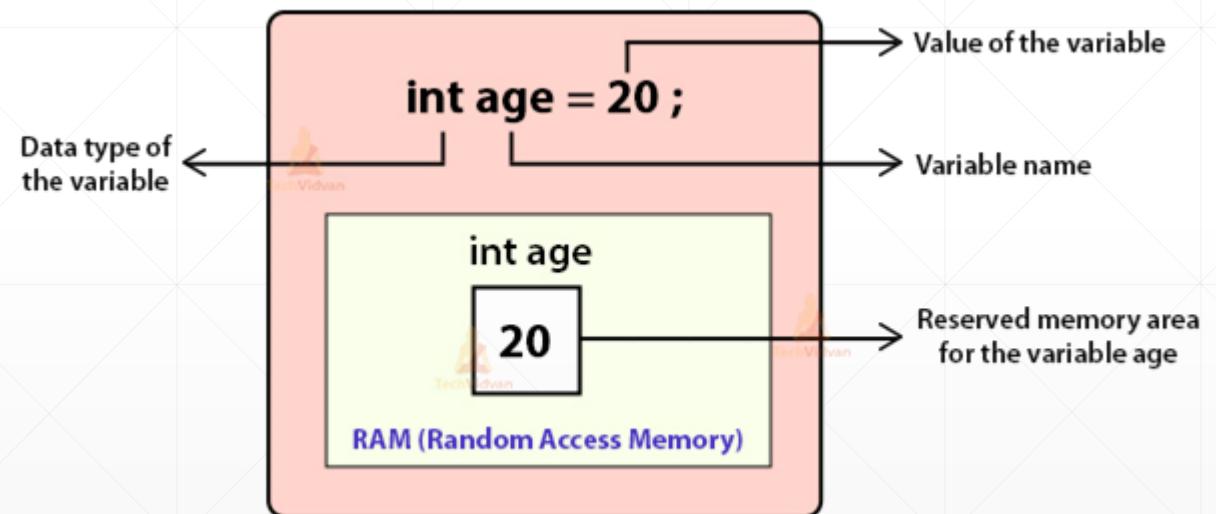
- A way to store data in a computer

- 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
- 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
 - Example) age, grade
- 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

```
int radius; // declaration  
radius = 10; // initialization
```

```
int radius = 10; // declaration & initialization  
char c1 = 'a', c2 = 'b', c3 = 'c';  
double weight = 75.56;
```

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	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

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

- 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	
365	

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

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

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 ±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

Types: Number (cont'd)

■ Floating point types

➤ Floating point literal

- Basically, 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      → 5.0 × 102 = 500.0  
0.12E-2 → 0.12 × 10-2 = 0.0012
```

- Example)

```
float var = 3.14; // OK?  
  
double var = 3.14;  
double var = 314e-2; // OK?
```

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'

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
- 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 \tID \tAge");
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;
```



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

[Translate from Korean](#)

Following



Constant

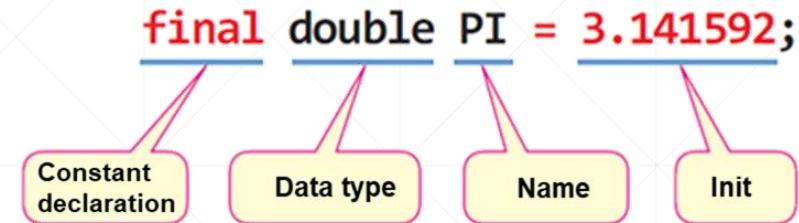
■ Final variable (constant)

- 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
 - 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`

- 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
 - 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);  
System.out.println(doubleValue);
```

```
char chValue = 'A';  
int intValue = chValue;  
System.out.println(intValue);  
short shortValue = 10;  
char chValue2 = shortValue;
```

Type Conversion: Promotion (cont'd)

■ Automatic conversion

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

```
short x= 10;  
short y = 20;  
  
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);
```

Type Conversion: Casting

■ Manual conversion

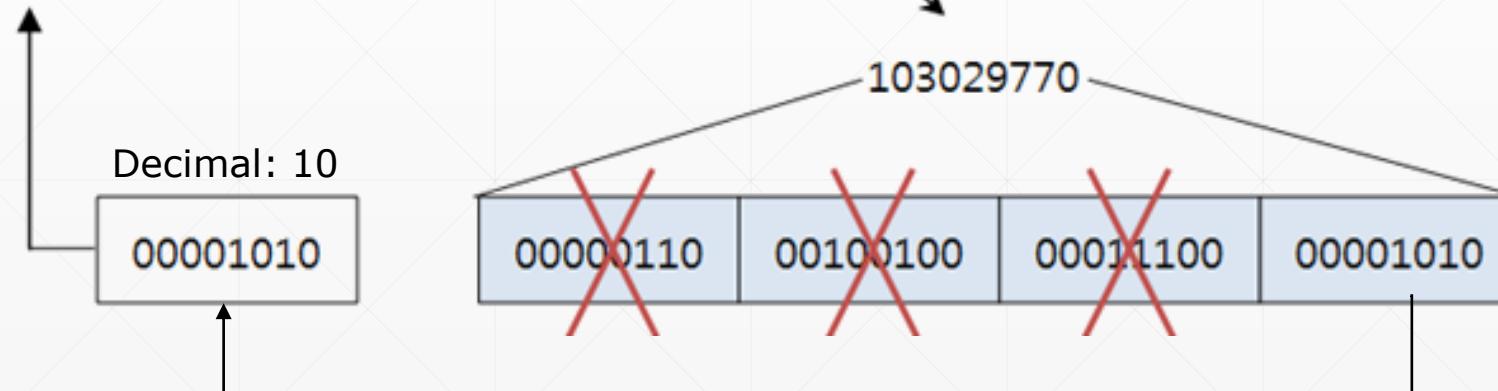
- Converting a larger size type to a smaller size type

byte -> short -> int -> long -> float -> double

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

Smaller type = (smaller type) larger type
casting

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



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);
System.out.println(myInt);
```

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

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

System.out.println(myInt);
System.out.println(myChar);
```

Q&A

■ Next week

- 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

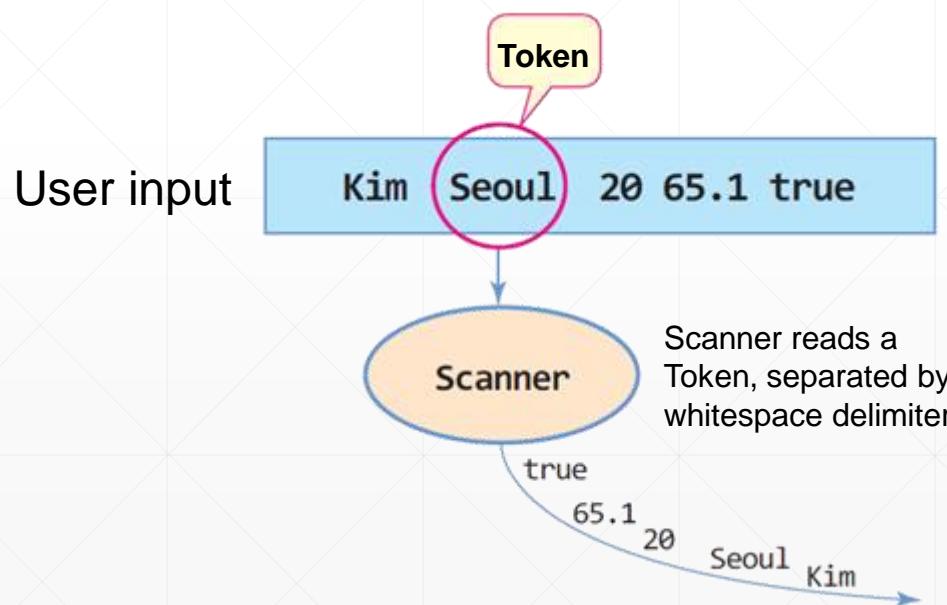
- Need to import `java.util.Scanner` class
- Ask `System.in` to take a sequence of bytes from the user
- Convert input bytes to data with an arbitrary type and then return!



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
 - ‘+’ : includes sign, whether positive or negative
 - ‘0’ : zero padding
 - ...
- Conversion: determines how the argument should be formatted
 - ‘d’: integer
 - ‘f’, ‘g’: floating-point number
 - ‘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~");
```

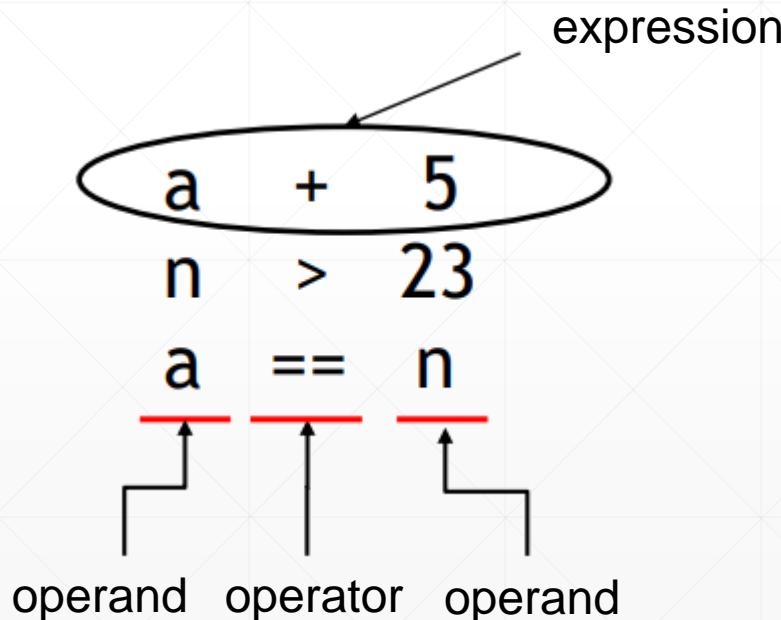
```
System.out.print("Hay Hay~");
```

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	$++ --$	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

➤ 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

■ Postfix operators (`a++`, `a--`)

- Return the value first, and then in/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)	&&	(3<5) && (1==1)	True
		(3>5) && (1==1)	False
Conditional OR (returns true if one of the statements is true)		(3<5) (1==1)	True
		(3>5) (1==1)	True
Complement (inverts the value of a Boolean)	!	!(3<5)	False
		!(3>5)	True

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));

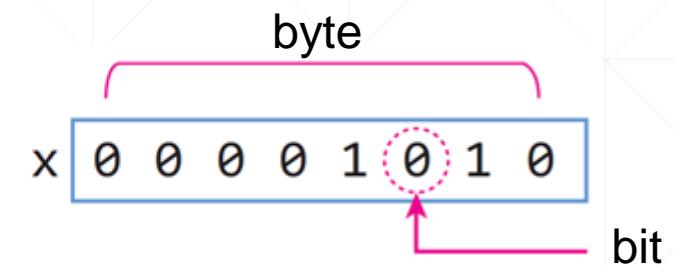
System.out.println((3 != 2) || (-1 > 0));
```

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

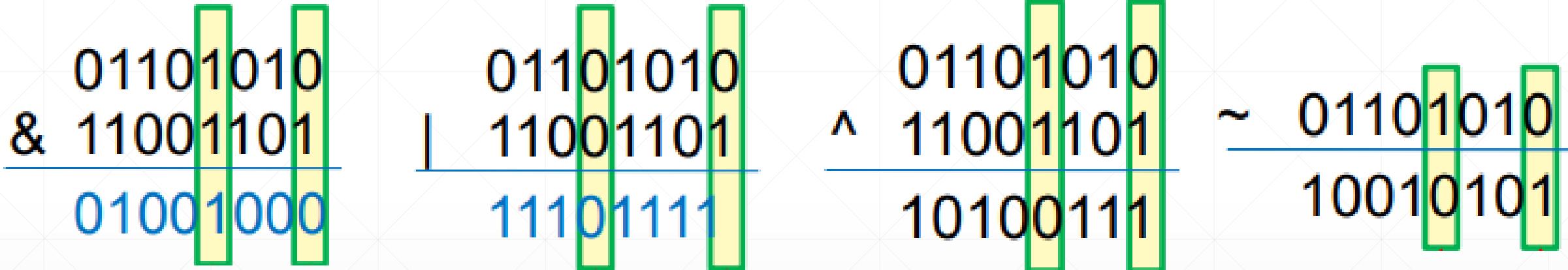
`byte x = 10;`



Description	Operator
AND (returns true if both bits are 1)	<code>a & b</code>
OR (returns true if one of the bits is 1)	<code>a b</code>
NOT (inverts a bit pattern)	<code>~ a</code>
XOR (returns true if two bits are different)	<code>a ^ b</code>

Operator: Bit & Shift (cont'd)

- Operators for the bits of operands



Operator: Bit & Shift (cont'd)

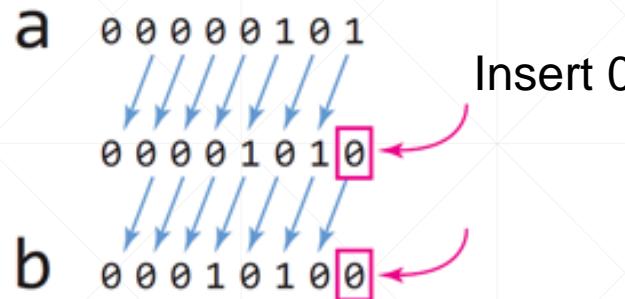
■ Operators for shifting the bits of operands

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 <i>copied</i>	$a \gg b$
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	$a \ggg b$

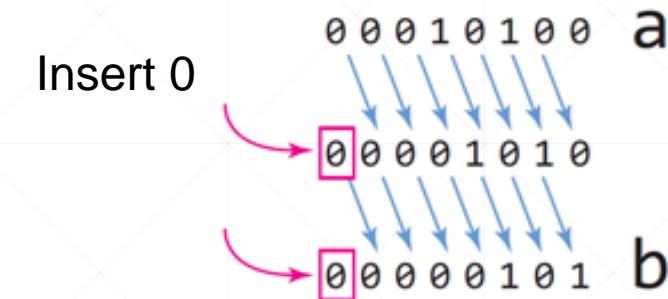
Operator: Bit & Shift (cont'd)

■ Example)

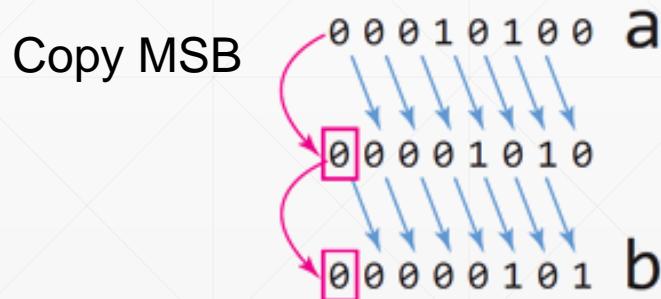
```
byte a = 5; // 5  
byte b = (byte)(a << 2); // 20
```



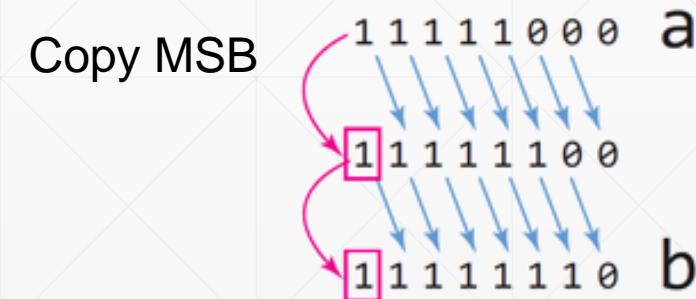
```
byte a = 20; // 20  
byte b = (byte)(a >> 2); // 5
```



```
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;
```

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)

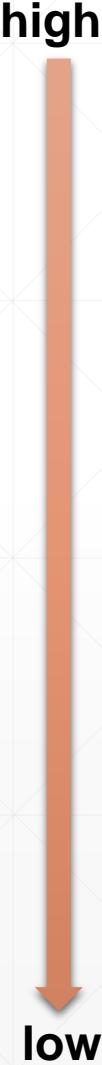
- Operators to assign values to variables
- Compound assignment

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)



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

- Conditions & Loop

Computer Language



Conditions & Loop

Agenda

- Condition
- Loop

Condition Loop

Program's Flow

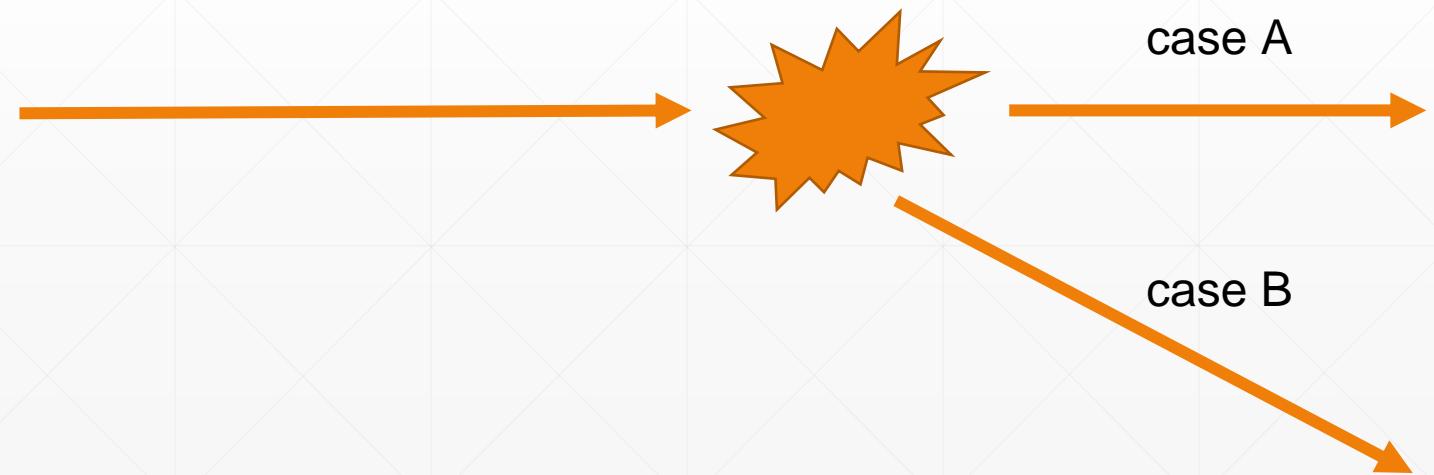
- Our simple program so far

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

Normal, sequential flow



- What if we want to handle various cases?



Program's Flow (cont'd)

■ How can we define a condition?

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

■ 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*

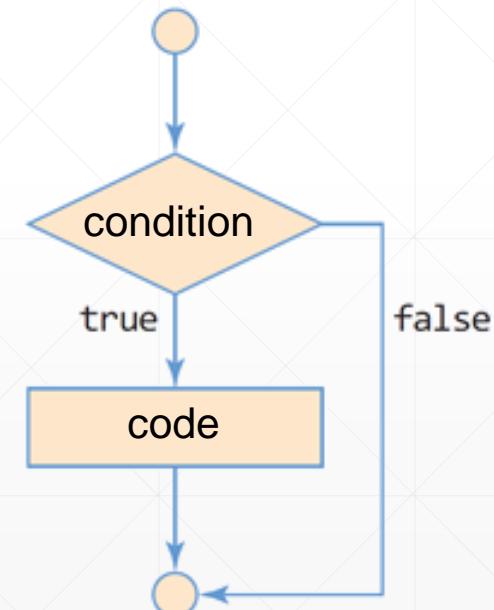
Condition: IF

■ 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)

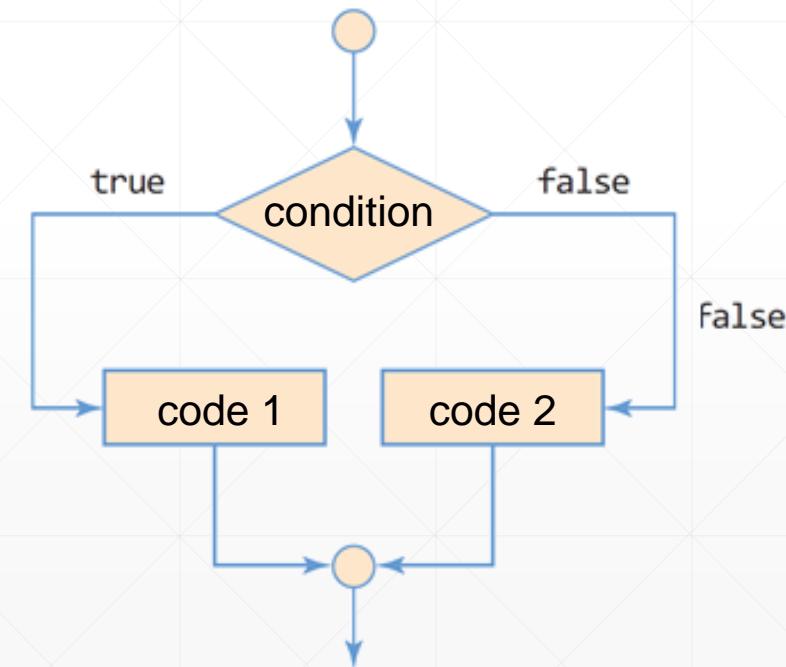
■ IF-else statement

- 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

- 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?

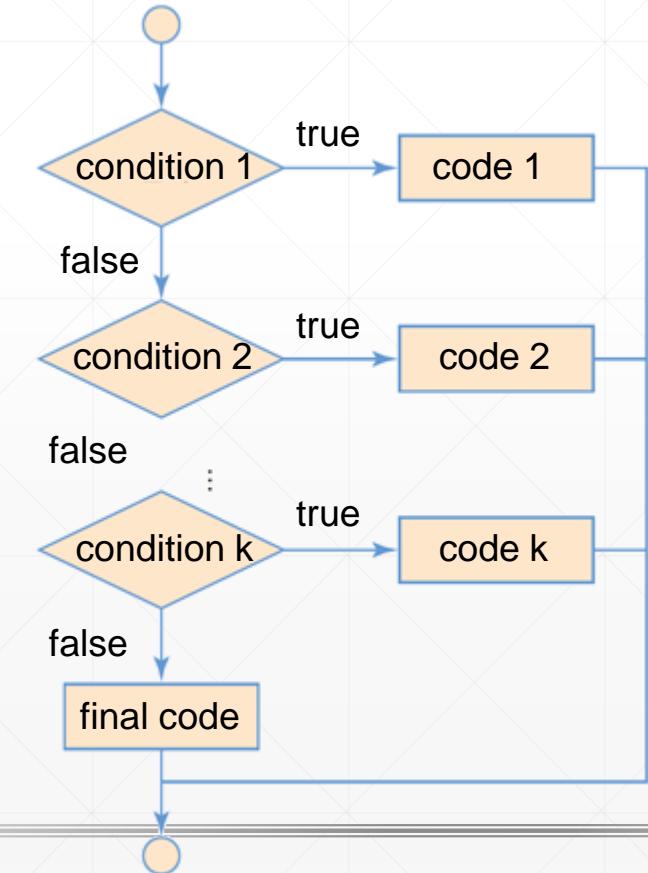
Condition: IF (cont'd)

■ 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 section 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';
}
```

Condition: IF (cont'd)

■ Nested if statement

- 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!");

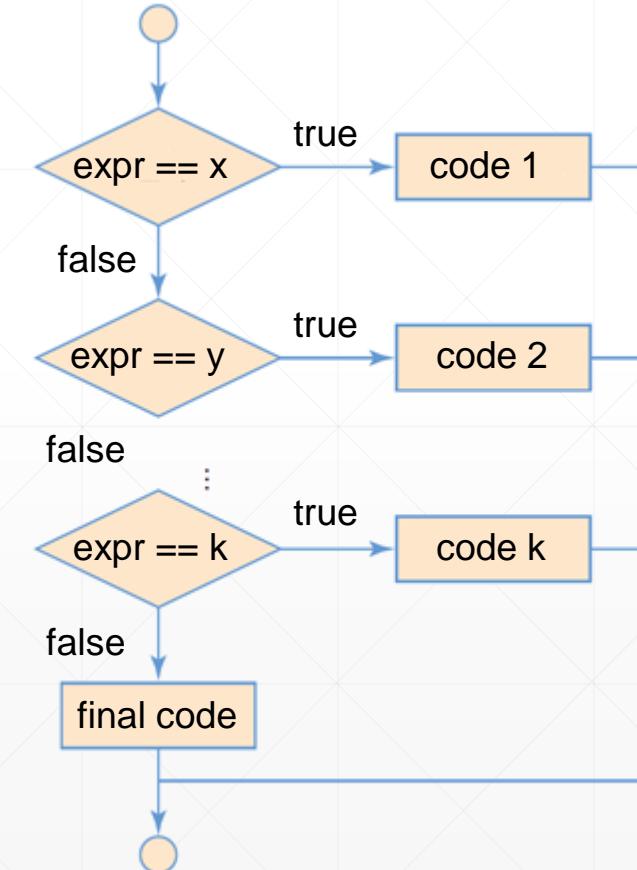
scanner.close();
```

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);
```



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);
```

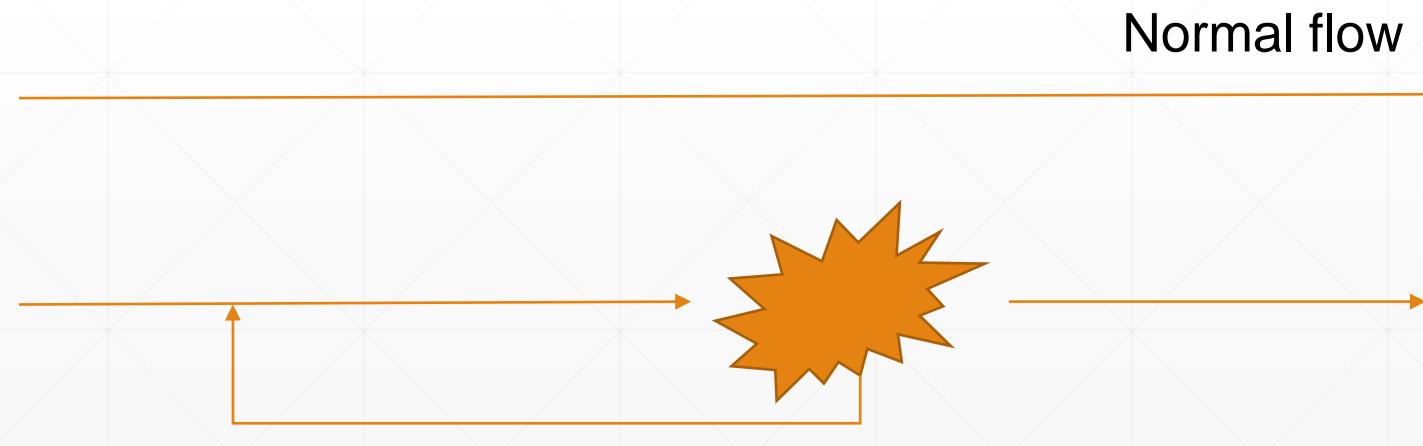
Umm... am I doing correct programming?

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



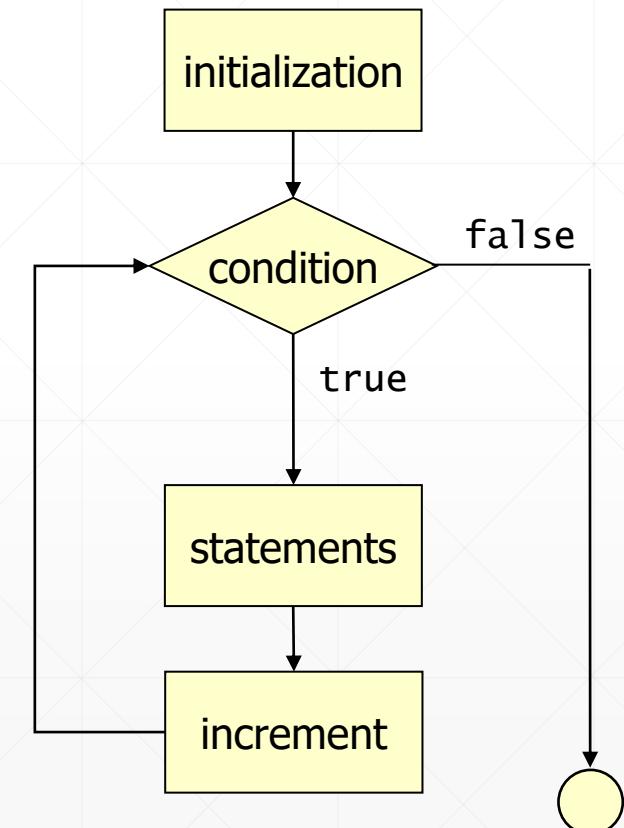
Loop! until a condition matches!

Loop: For

■ Most frequently used loop statement

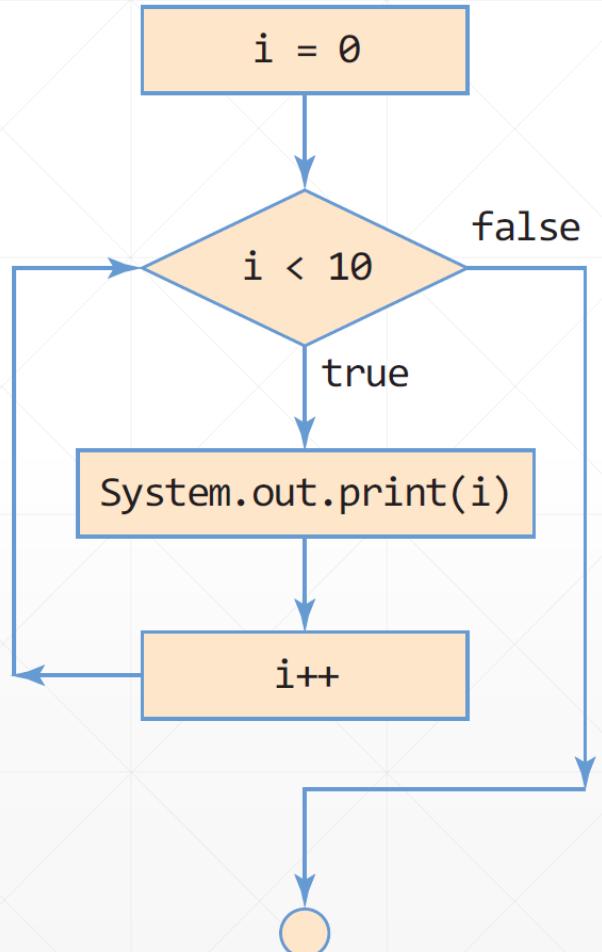
- The initialization expression initializes the loop
- When the condition evaluates to false, the loop terminates
 - Generally, used with counting
- The increment is invoked after each iteration through the loop
 - It is acceptable for this expression to increment or decrement a value

```
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);
```

■ 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;
```

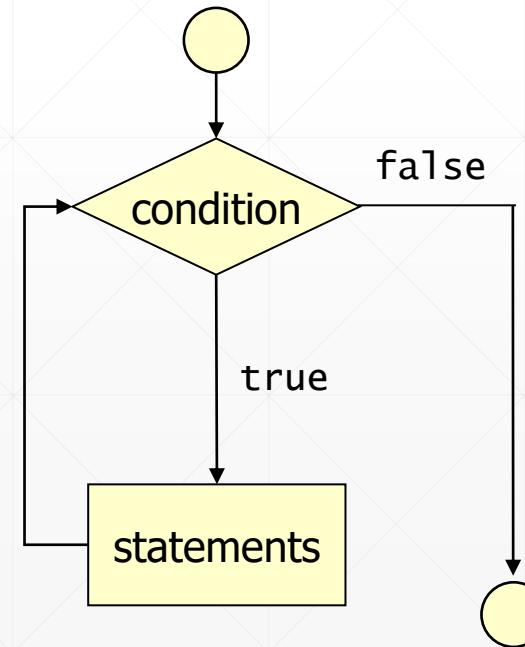
```
int sum = 0;  
for(int i = 100; i >= 0; i--)  
    sum += i;
```

Loop: While

■ Another loop statement

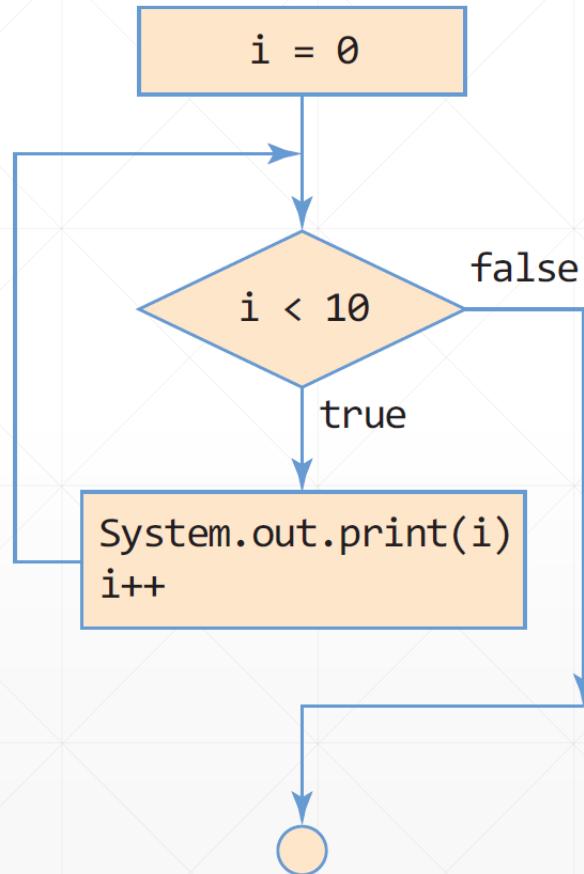
- While statement evaluates expression, which must return a boolean value
- If the expression evaluates to true,
the while statement executes the statement(s) in the while block
- While statement continues testing the expression and executing its block
until the expression evaluates to false

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



Loop: While (cont'd)

■ Example)



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

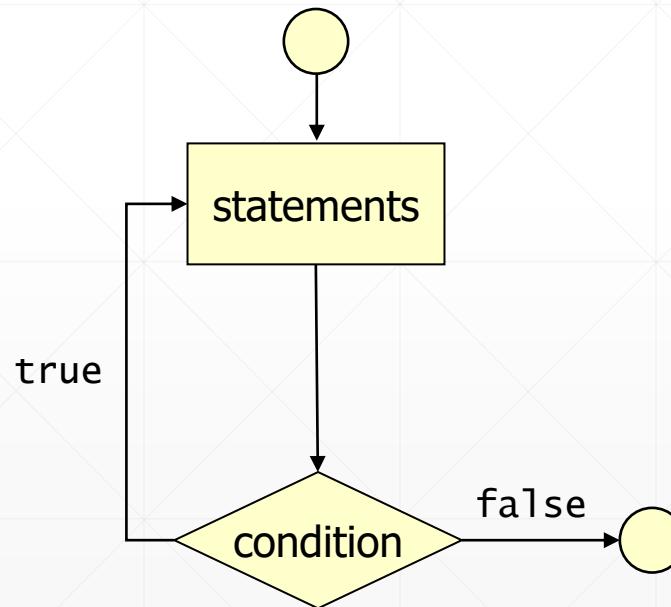
0123456789

Loop: Do-While

■ Another while statement

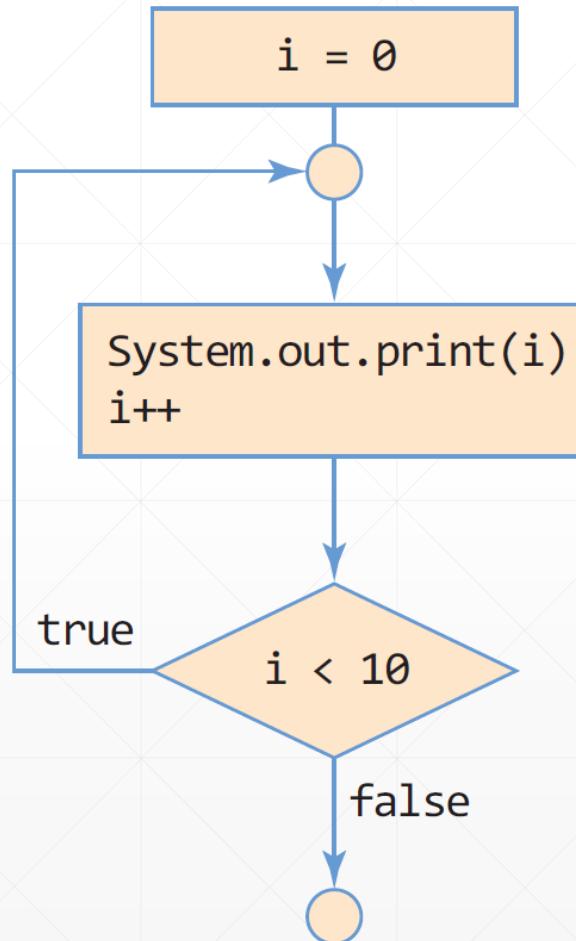
- Do-while evaluates its expression at the bottom of the loop instead of the top!
- The statements within the do block are always executed **at least once**

```
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 );
```

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;
```



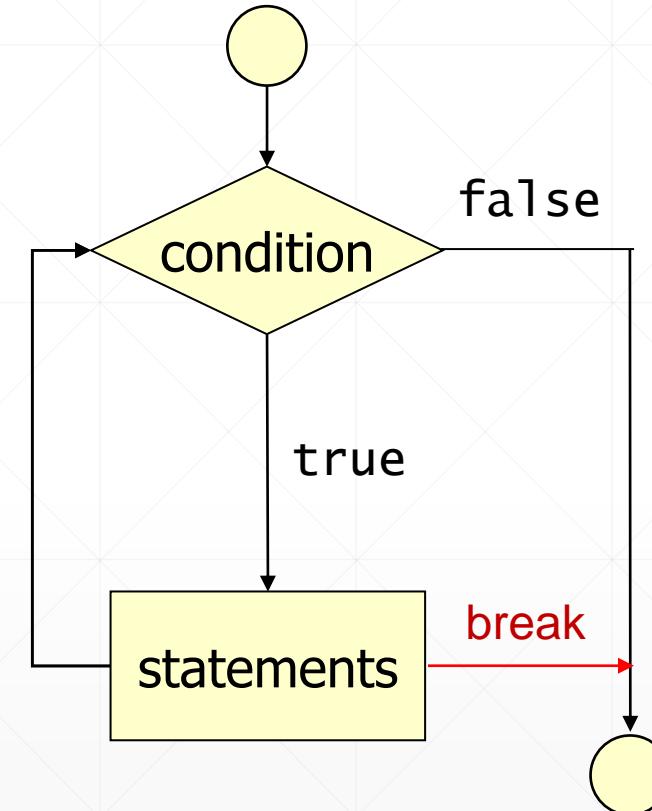
Loop: Continue & Break (cont'd)

■ 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){ ← Infinite loop!  
        break; ← Break the loop!  
    }  
    System.out.println("Here!"); ← Where should we go?  
}  
System.out.println("There!"); ←
```

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;



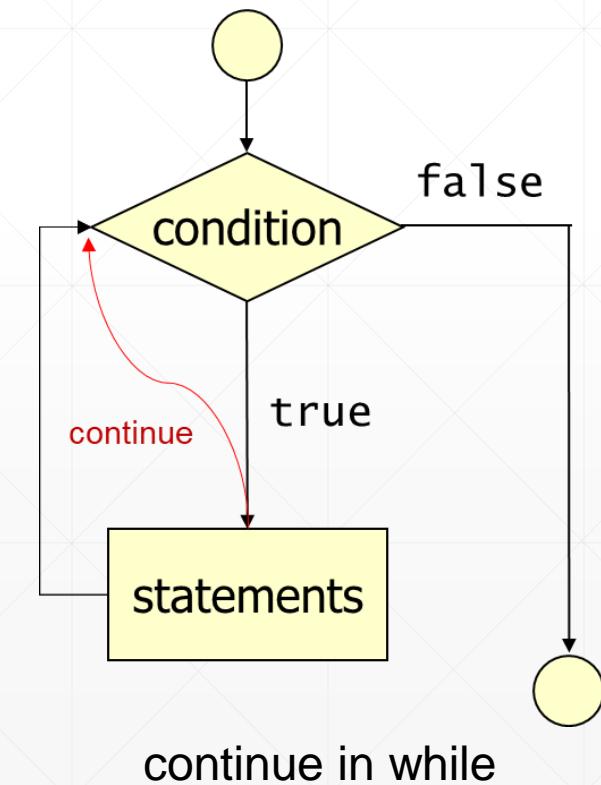
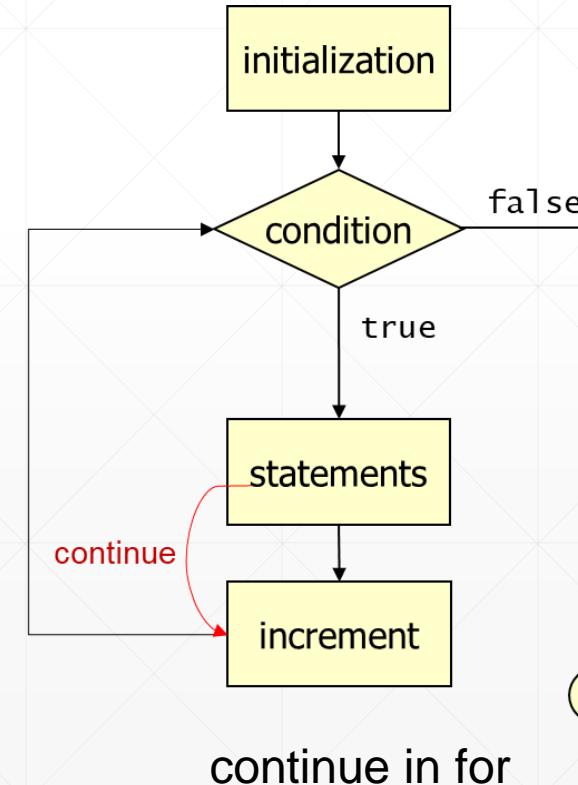
Loop: Continue & Break (cont'd)

■ 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!");
```

Q&A

■ Next week

- Reference Types

Computer Language



Reference Type



Agenda

■ Reference Type

- Basics
- Array

Basics

Array

Reference Type

■ Data type

➤ Primitive type

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

➤ Reference type

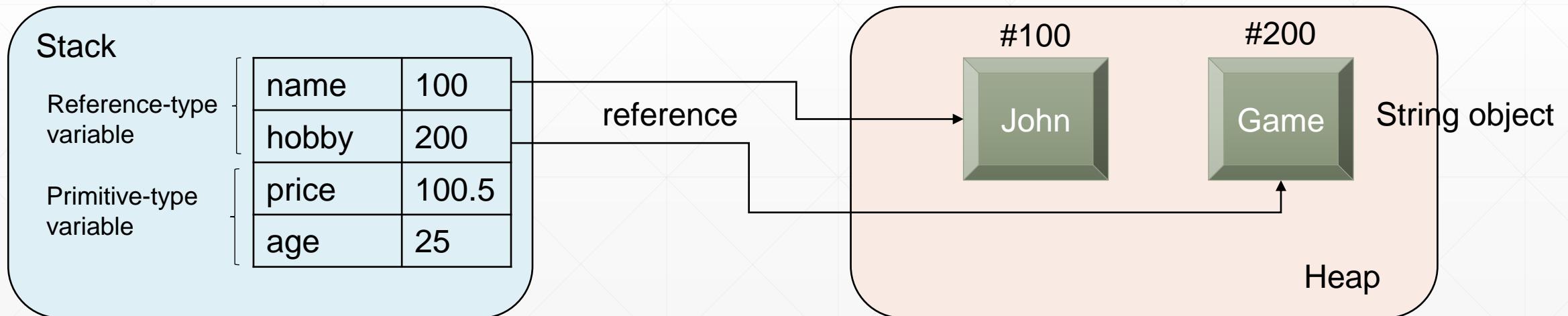
- Array
- Enum
- Class (+String, Wrapper)
- Interface

Reference Type (cont'd)

■ Data type

- Primitive type variable
 - Store a value in the variable (stack)
- Reference type variable
 - Store a value in the memory (heap)
 - Store an address of the memory for further reference

```
int age = 25;  
double price = 100.5;  
  
String name = "John";  
String hobby = "Game";
```



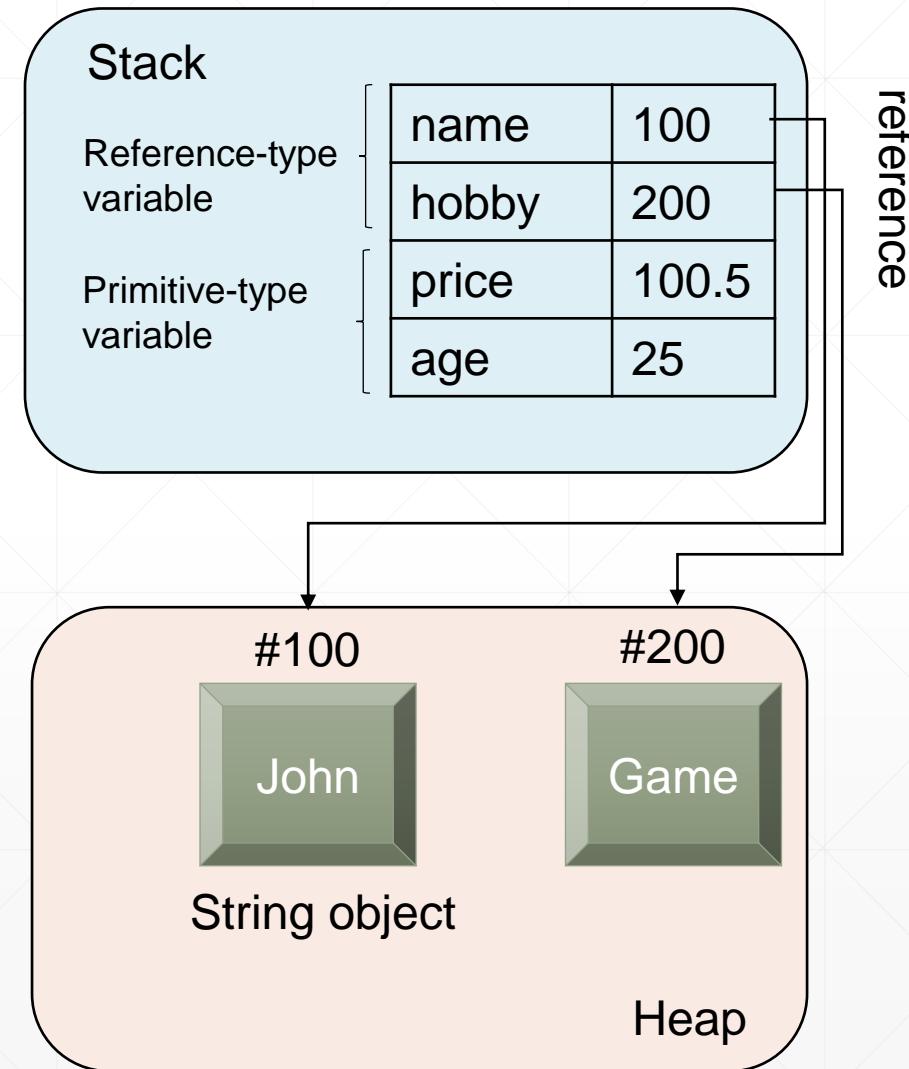
Reference Type (cont'd)

■ Stack

- 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
- After the method is finished, the memory space for the data is cleared

■ Heap

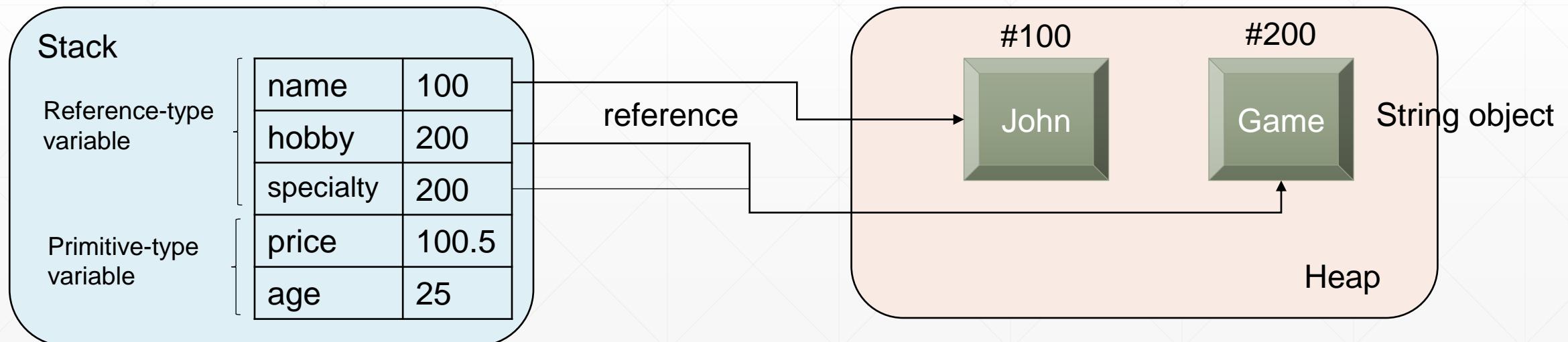
- Created by JVM
- Uses as long as the application is running
- All objects are stored in a heap with a global access, therefore, can be referenced from anywhere in the app



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

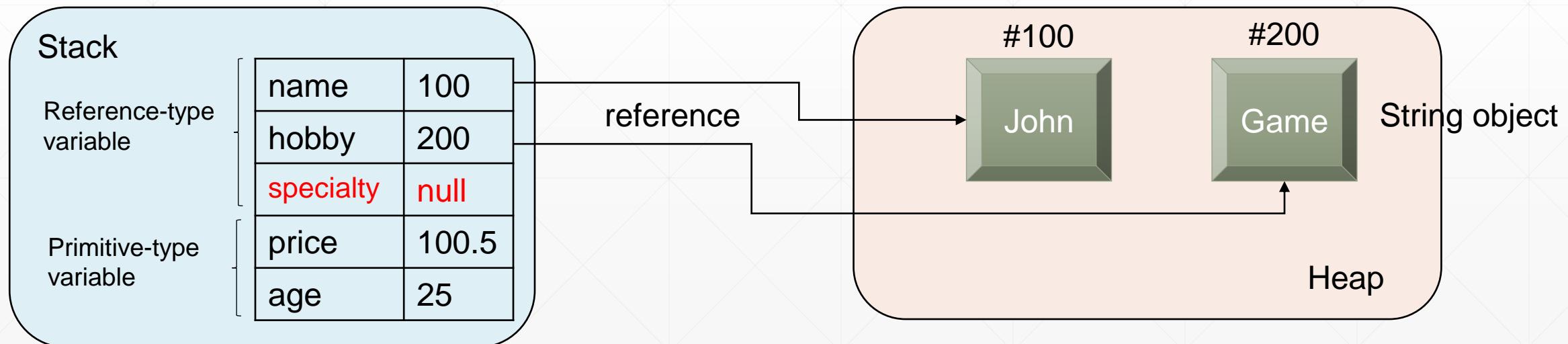


Reference Type (cont'd)

■ null

- Can be used for a reference type
- 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

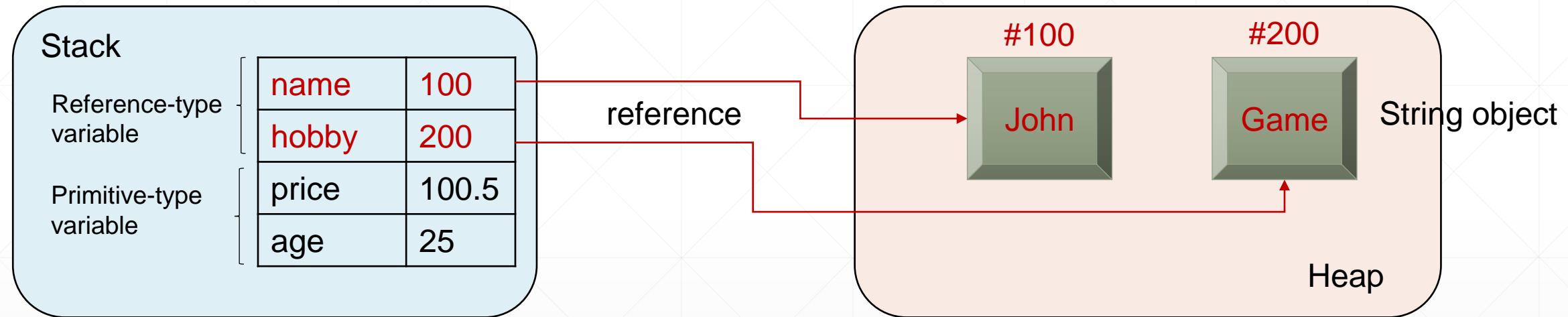
```
int[] intArray = null;  
intArray[0] = 10;      //NullPointerException
```

```
String str = null;  
System.out.println(" Length: " + str.length()); //NullPointerException
```

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

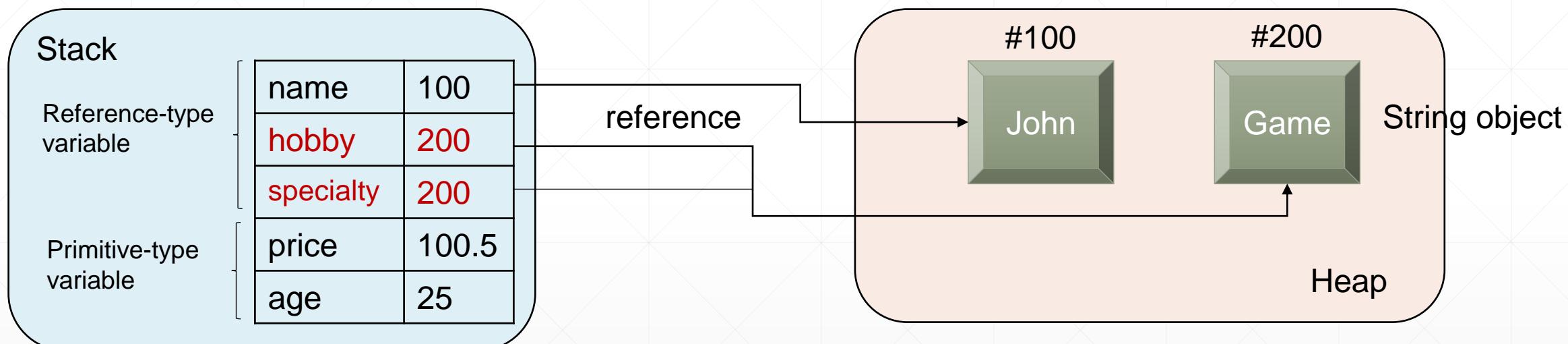
```
int age = 25;  
double price = 100.5;
```

```
String name = "John";  
String hobby = "Game";
```

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

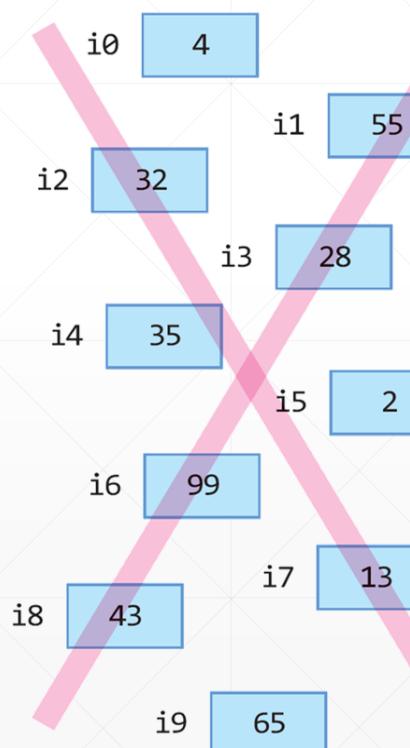
When We need an Array? (cont'd)

- 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;
```

```
int i[] = new int[10];
```

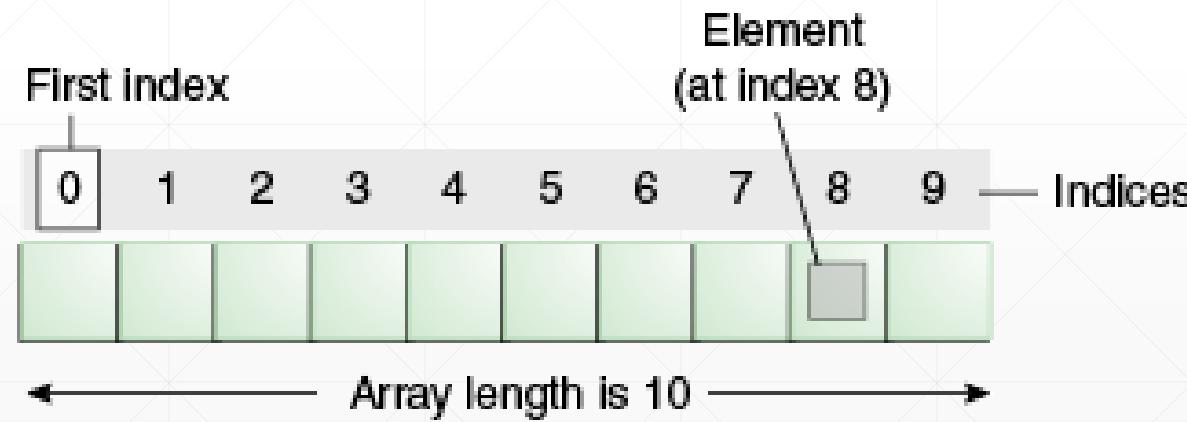


```
for(sum=0, n=0; n<10; n++)  
    sum += i[n];
```

Array: Index

■ 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!

Array: Declaration and Creation (cont'd)

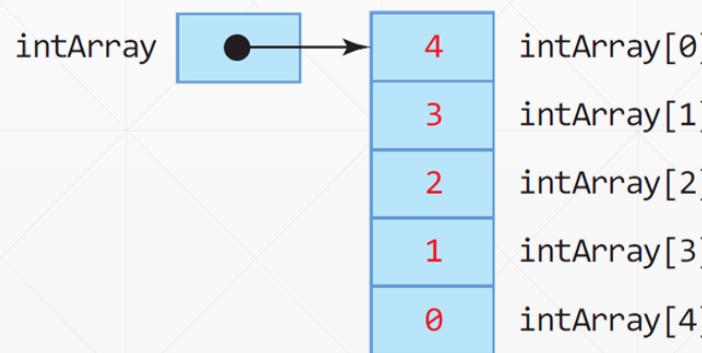
■ Declaration with initialization

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

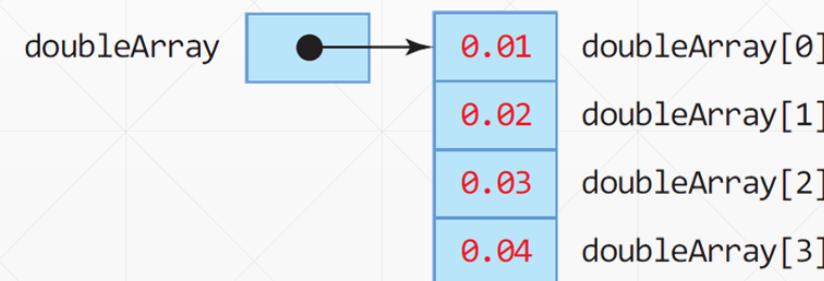
➤ Example)

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

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



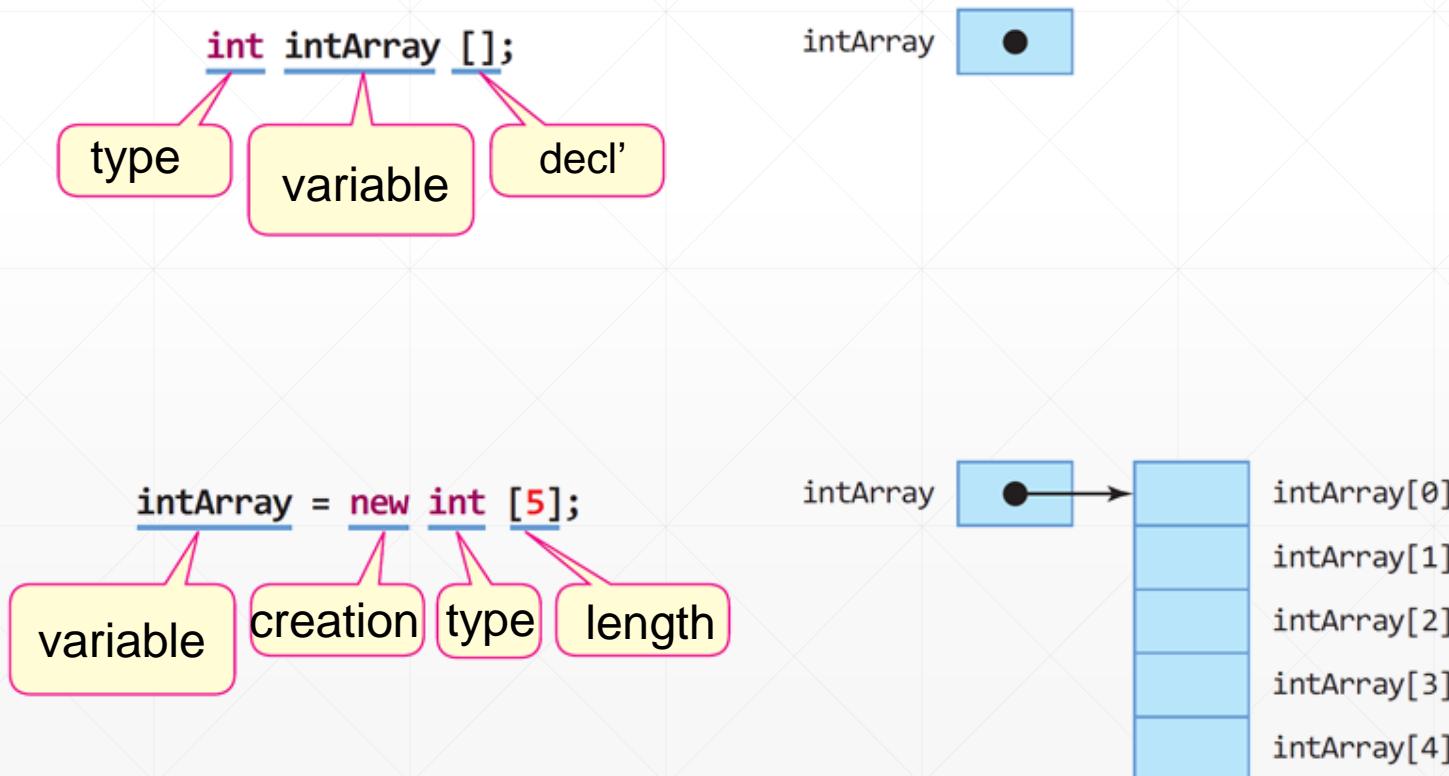
```
double doubleArray[] = {0.01, 0.02, 0.03, 0.04};
```



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

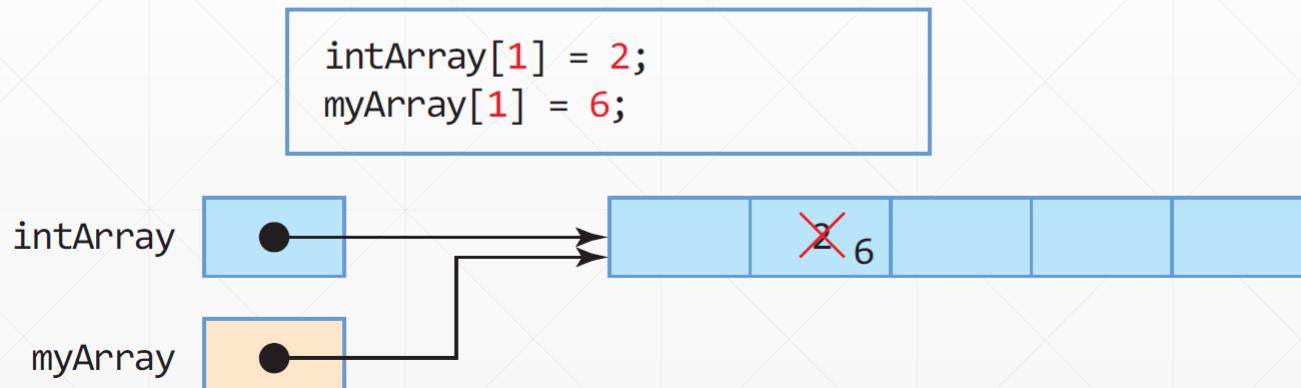
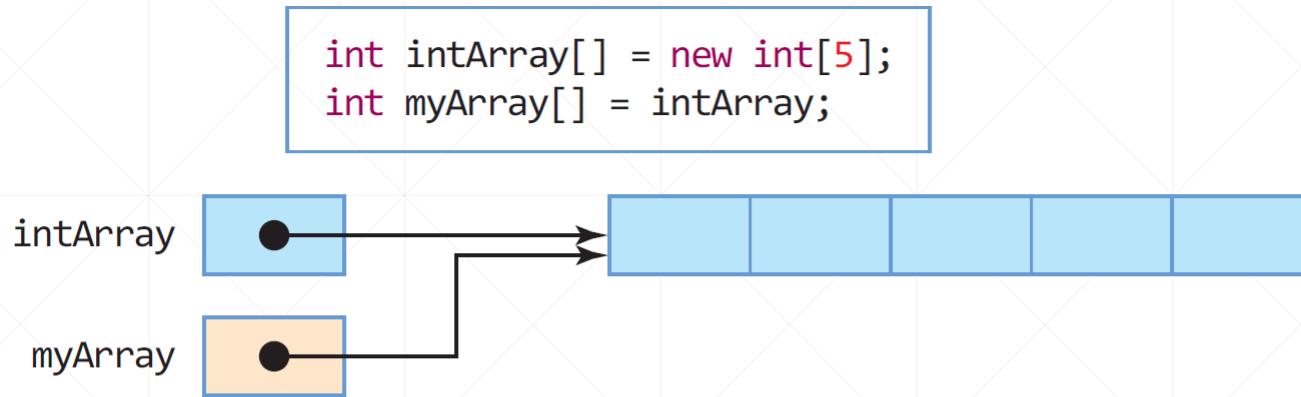
ArrVariable[index]

- 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

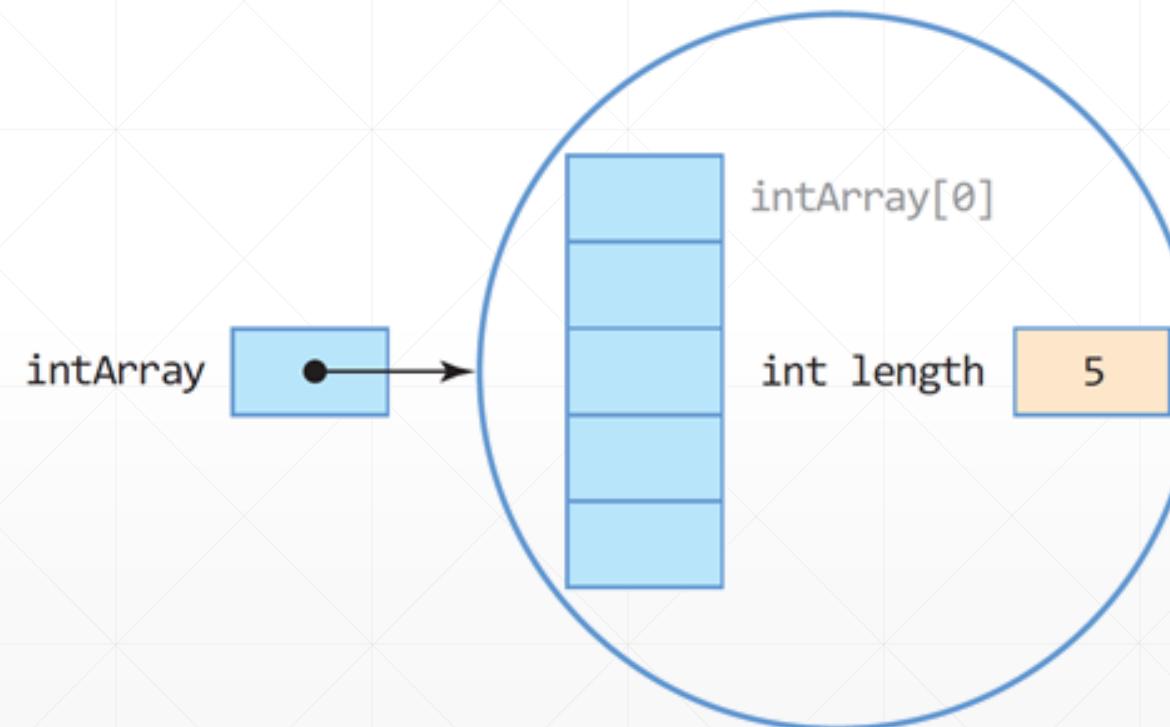
int max = 0; // current max value
System.out.println("Input 5 Positive Numbers.");
for (int i = 0; i < 5; i++) {
    intArray[i] = scanner.nextInt(); // store the input value to the array
    if (intArray[i] > max) // if intArray[i] is greater than the current max
        max = intArray[i]; // then set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

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

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

■ Creation

```
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];
```

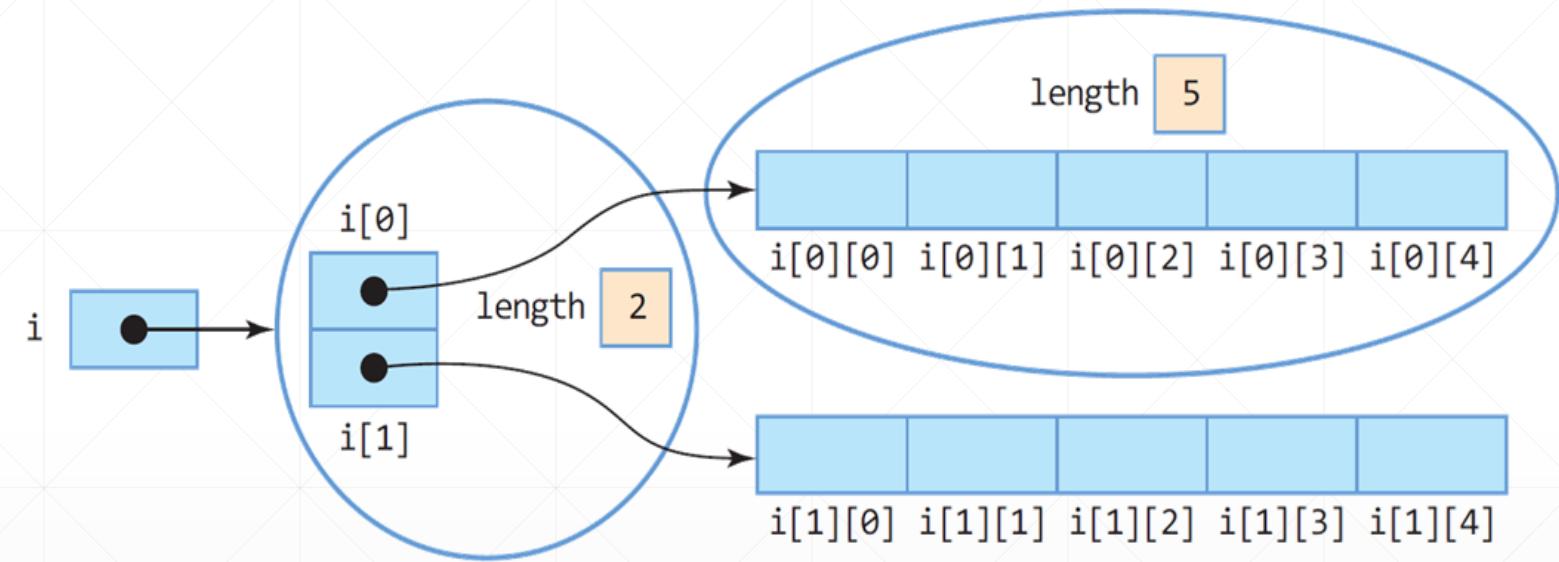
■ Declaration with initialization

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

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)

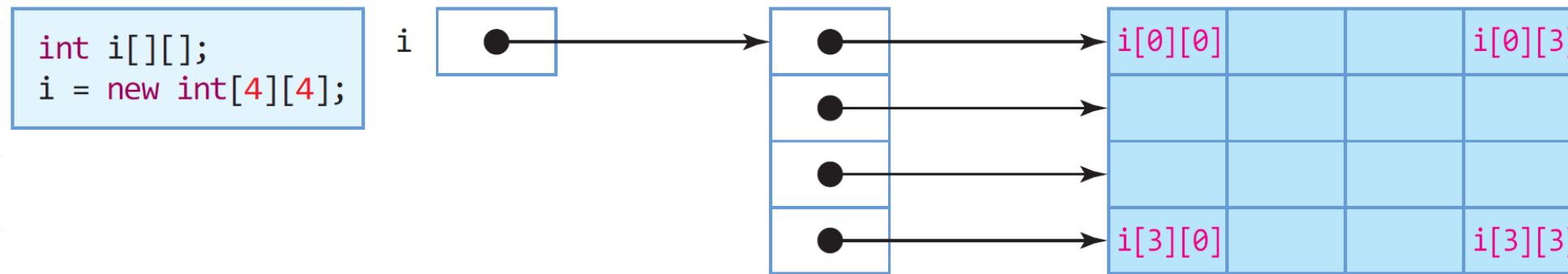
Array: 2D-array (cont'd)

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

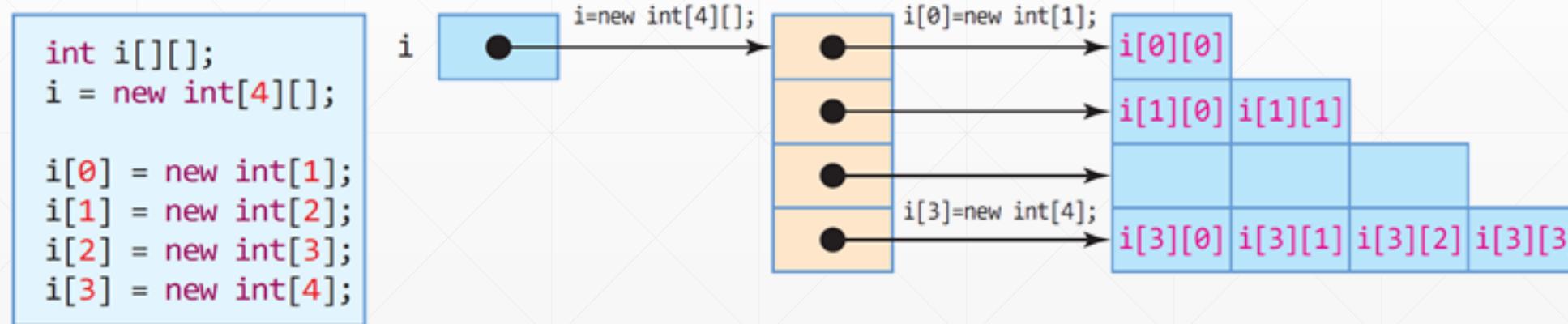
```
double score[][] = {{3.3, 3.4},      // GPA for the 1st year  
                     {3.5, 3.6},    // GPA for the 2nd year  
                     {3.7, 4.0},    // GPA for the 3rd year  
                     {4.1, 4.2}};   // GPA for the 4th year  
  
double sum = 0;  
for (int year = 0; year < score.length; 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



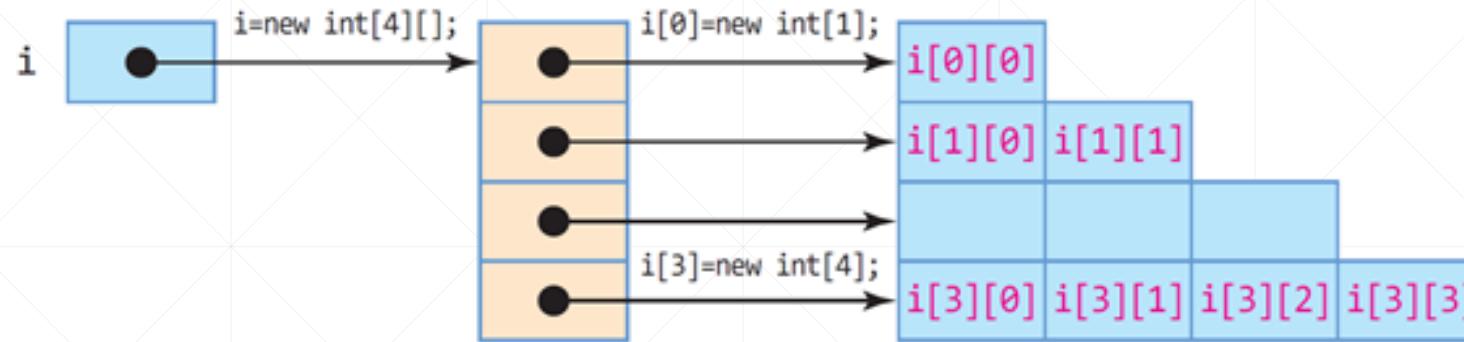
■ Non-square array



Array: 2D-array (cont'd)

■ Non-square array

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



■ 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	

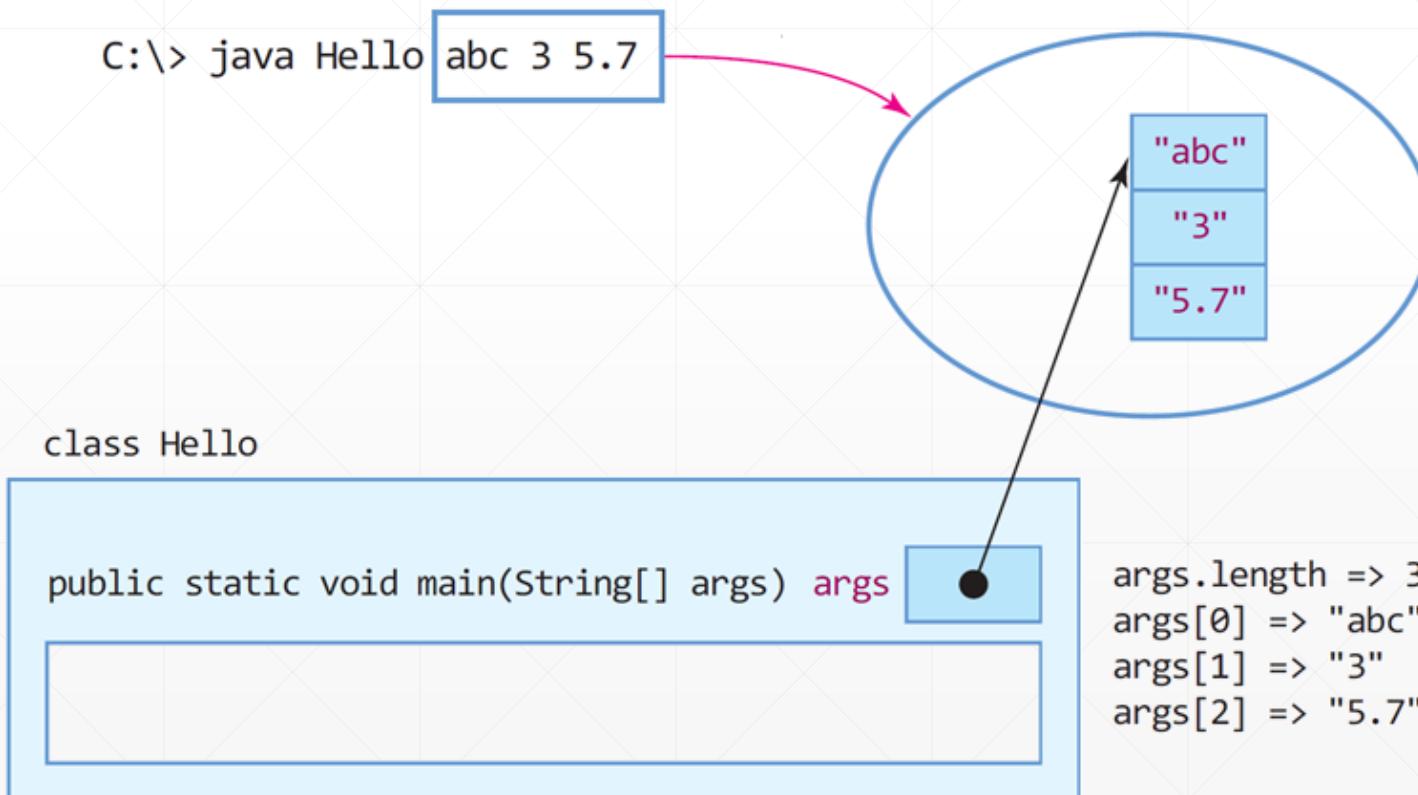
```
int intArray[][] = new int[4][];
intArray[0] = new int[3];
intArray[1] = new int[2];
intArray[2] = new int[3];
intArray[3] = new int[2];

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.

- 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



Array: Misc. (cont'd)

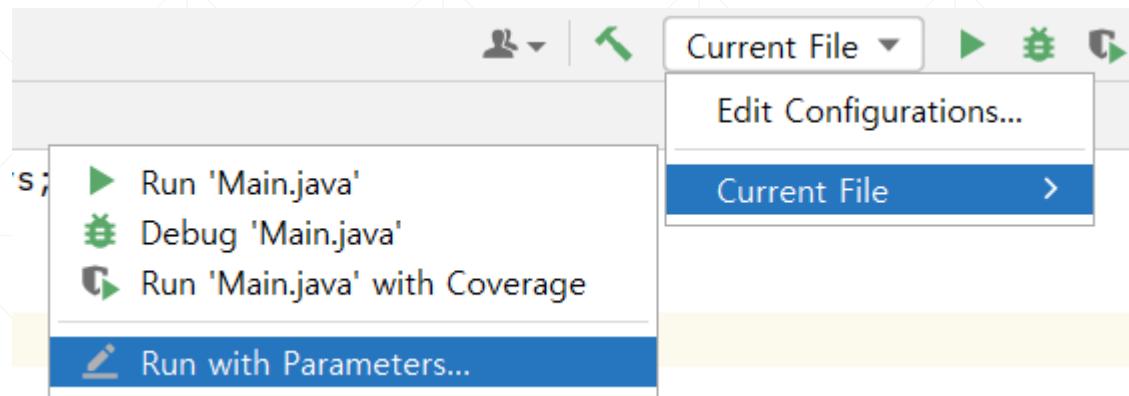
- Example) print out the contents of args array in the main method

```
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

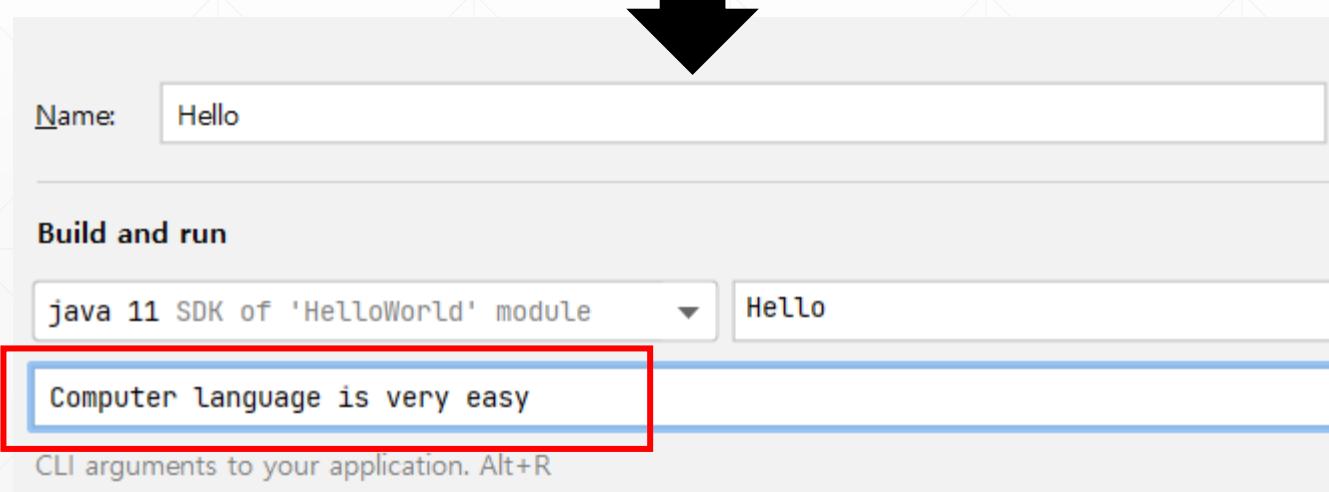
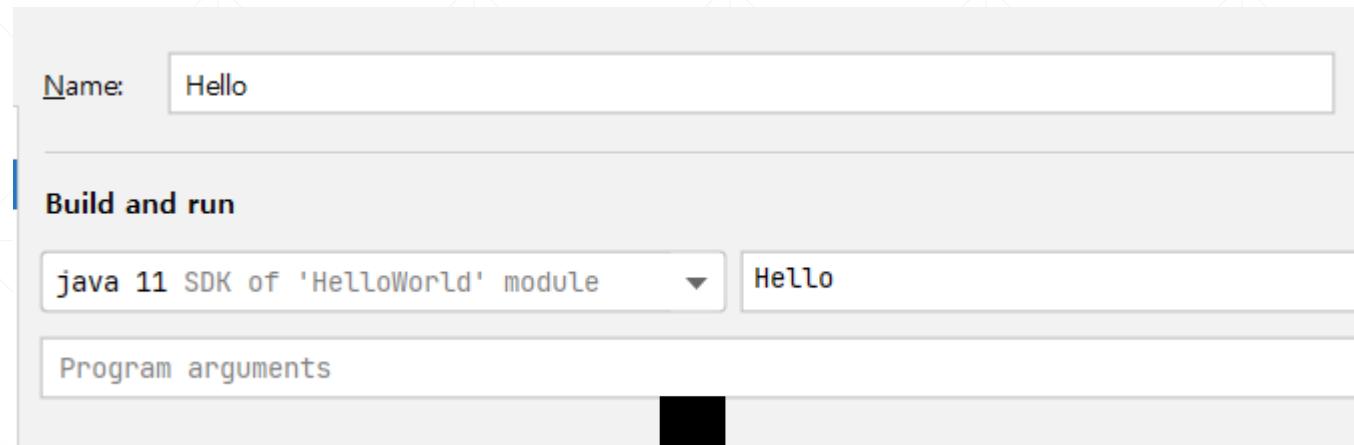
Array: Misc. (cont'd)

■ Run with parameters



Array: Misc. (cont'd)

■ Edit Configuration: passing arguments



What happens?

Q&A

■ Next week

- OOD/P: Class and Methods

Computer Language



OOP 1: Class



Agenda

- OOP
- Class

OOP

Class

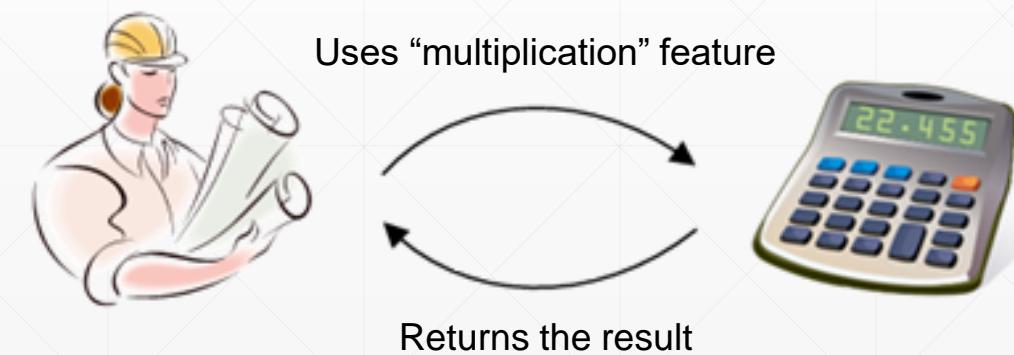
OOP: Basics

■ Object-Oriented Programming (OOP)

- The world is composed of objects (thing)



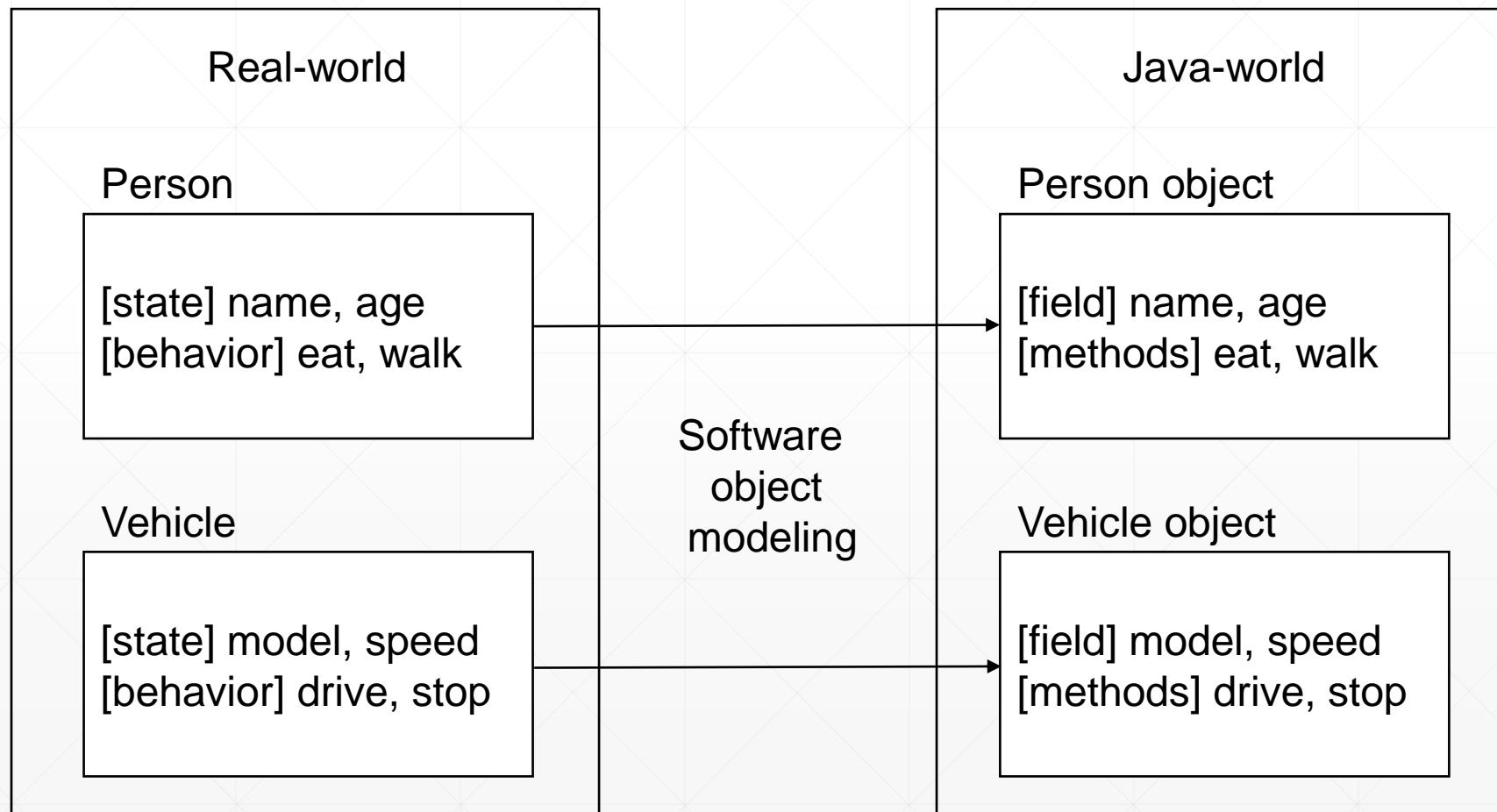
- Characteristics of objects
 - Each object has its own state and behavior
 - Objects interact with each other



OOP: Basics (cont'd)

■ Objects in Java world

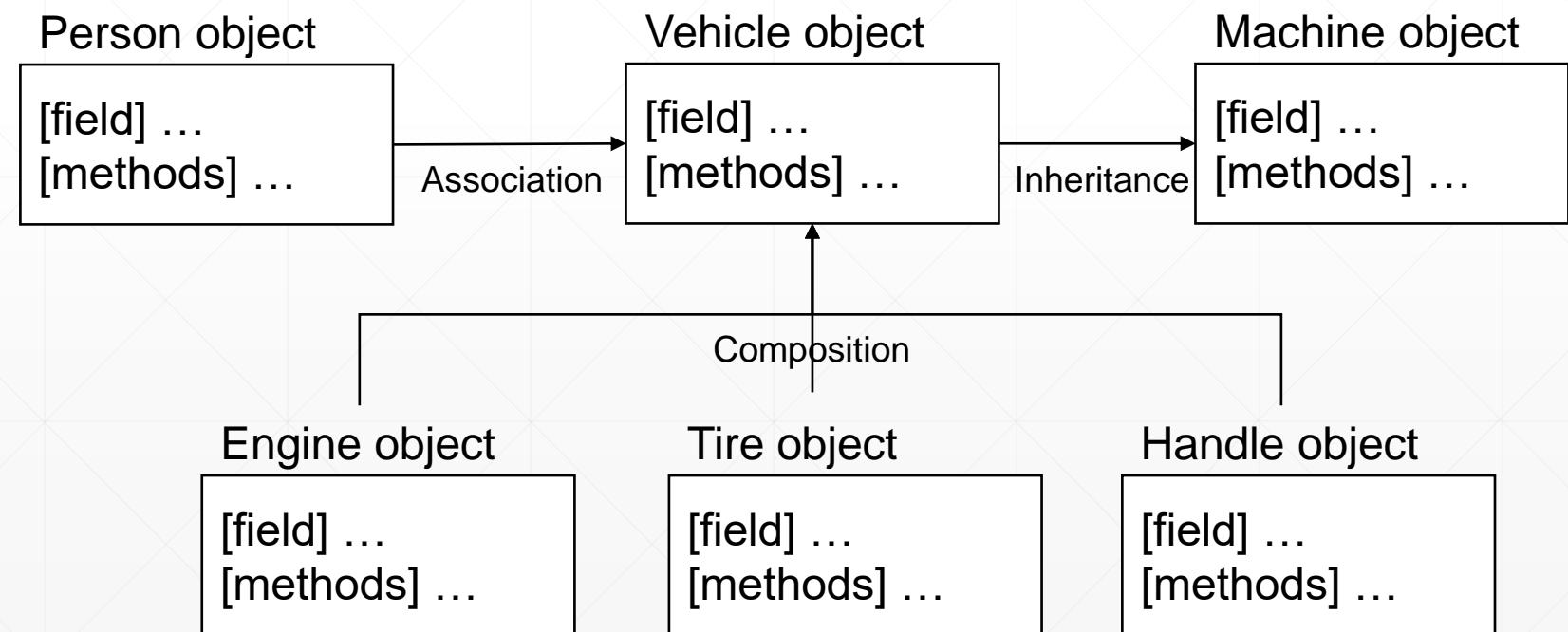
- Java object models the real-world object by defining fields (states) and methods (behaviors)



OOP: Basics (cont'd)

■ Objects in Java world

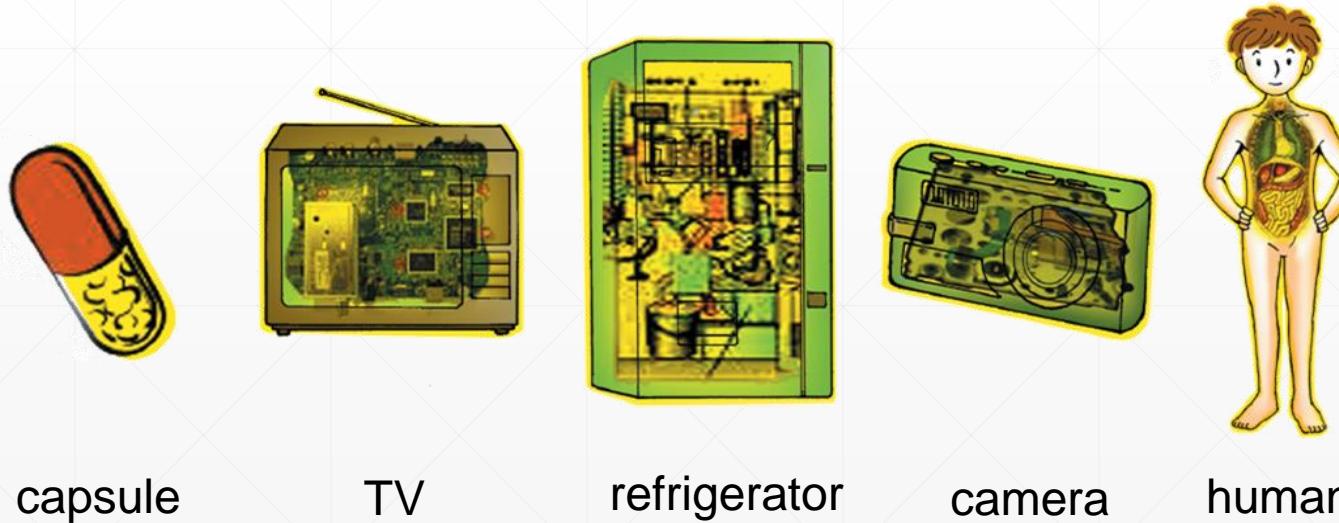
- Relationships between the objects
 - Association
 - Composition/Aggregation
 - Inheritance



OOP: Characteristics

■ Encapsulation

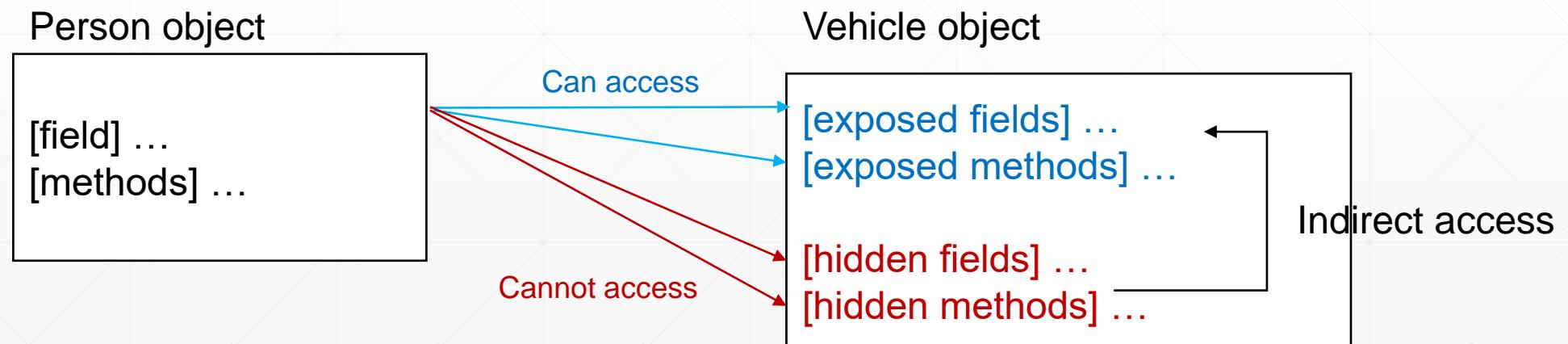
- Information/Implementation hiding
 - External objects cannot know the details (internal structure) of an object they want to use
 - External objects cannot access the private fields/methods of an object they want to use
 - External objects can only access the fields and methods explicitly exposed by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



OOP: Characteristics (cont'd)

■ Encapsulation: Benefits

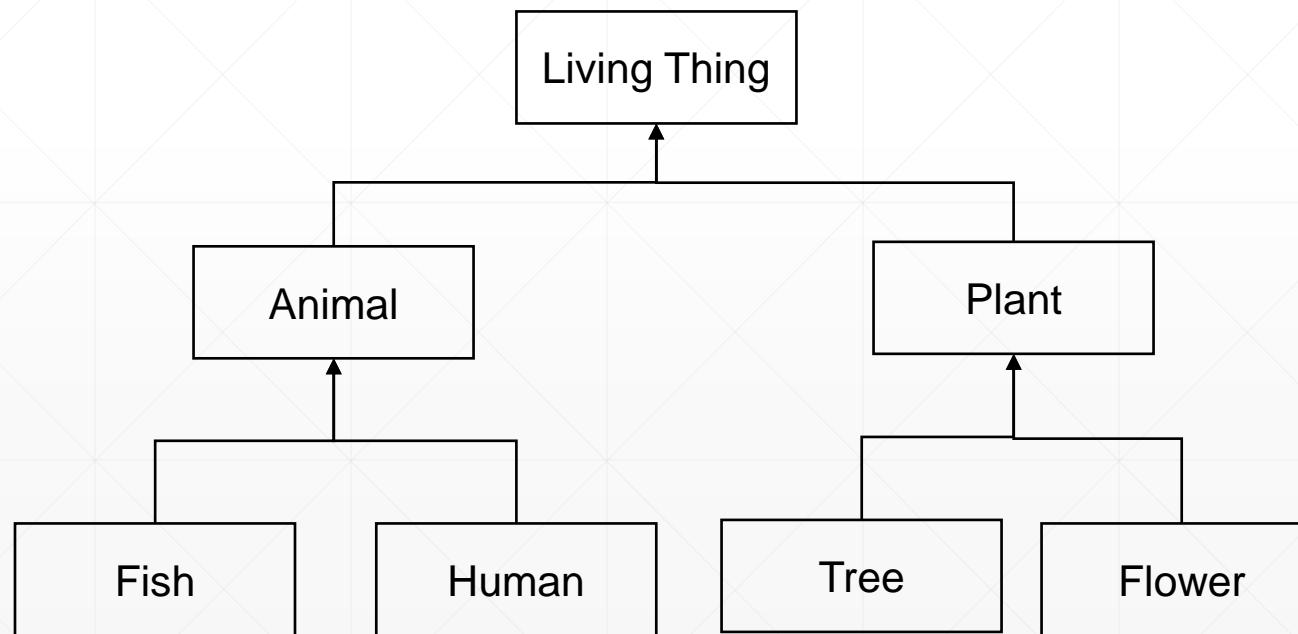
- Better control of class attributes and methods
- Flexible: the programmer can change one part of the code without affecting other parts
- Increased security of data



OOP: Characteristics (cont'd)

■ Inheritance

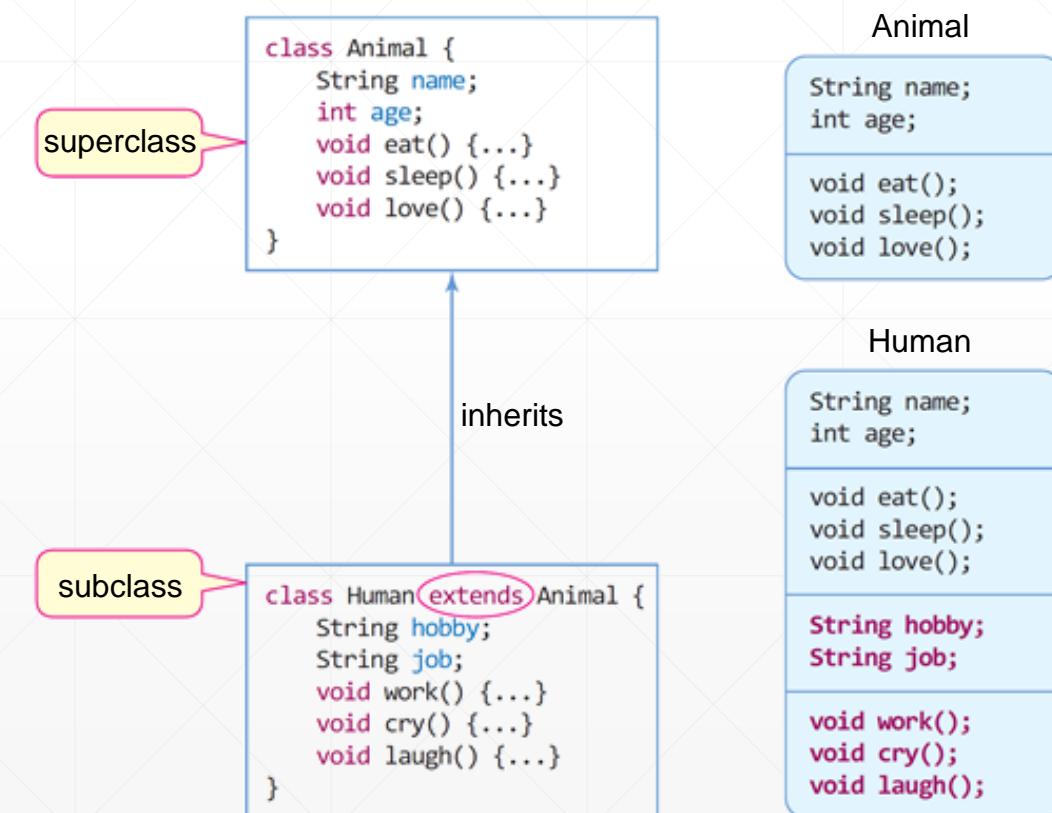
- It is possible to inherit attributes and methods from one class to another
 - Subclass: a class derived from another class (child/derived/extended class)
 - Superclass: the class from which the subclass is derived (parent/base class)



OOP: Characteristics (cont'd)

■ Inheritance

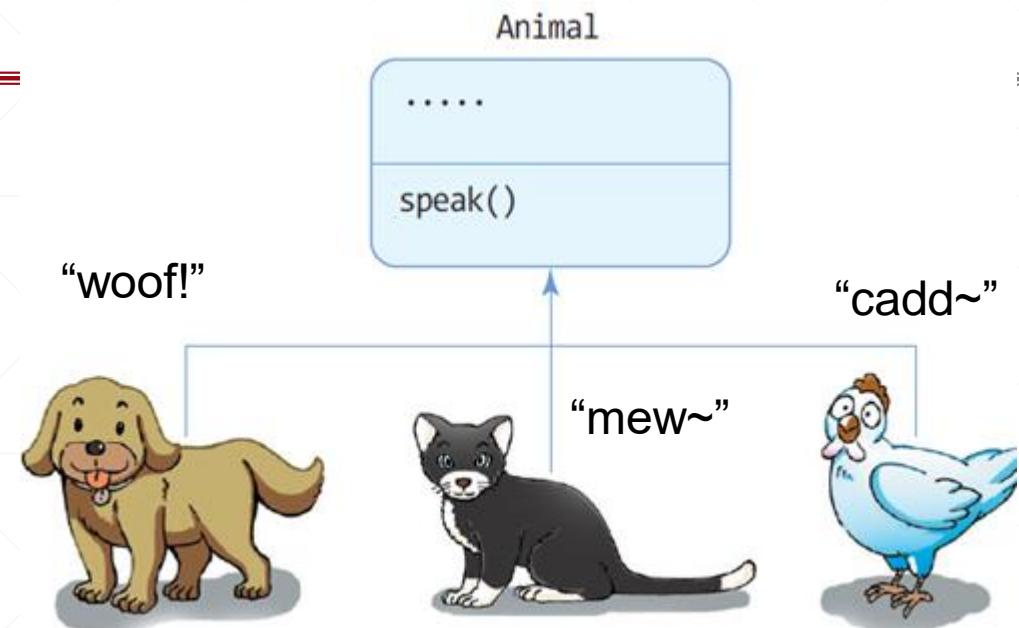
- Subclass inherits superclass's members and can extend its own features
- We can reuse the fields and methods of the existing class without having to re-write
- Benefits
 - Rapid implementation using existing classes
 - Reduced redundant codes
 - Better, efficient maintenance
 - Polymorphism



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



■ 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.

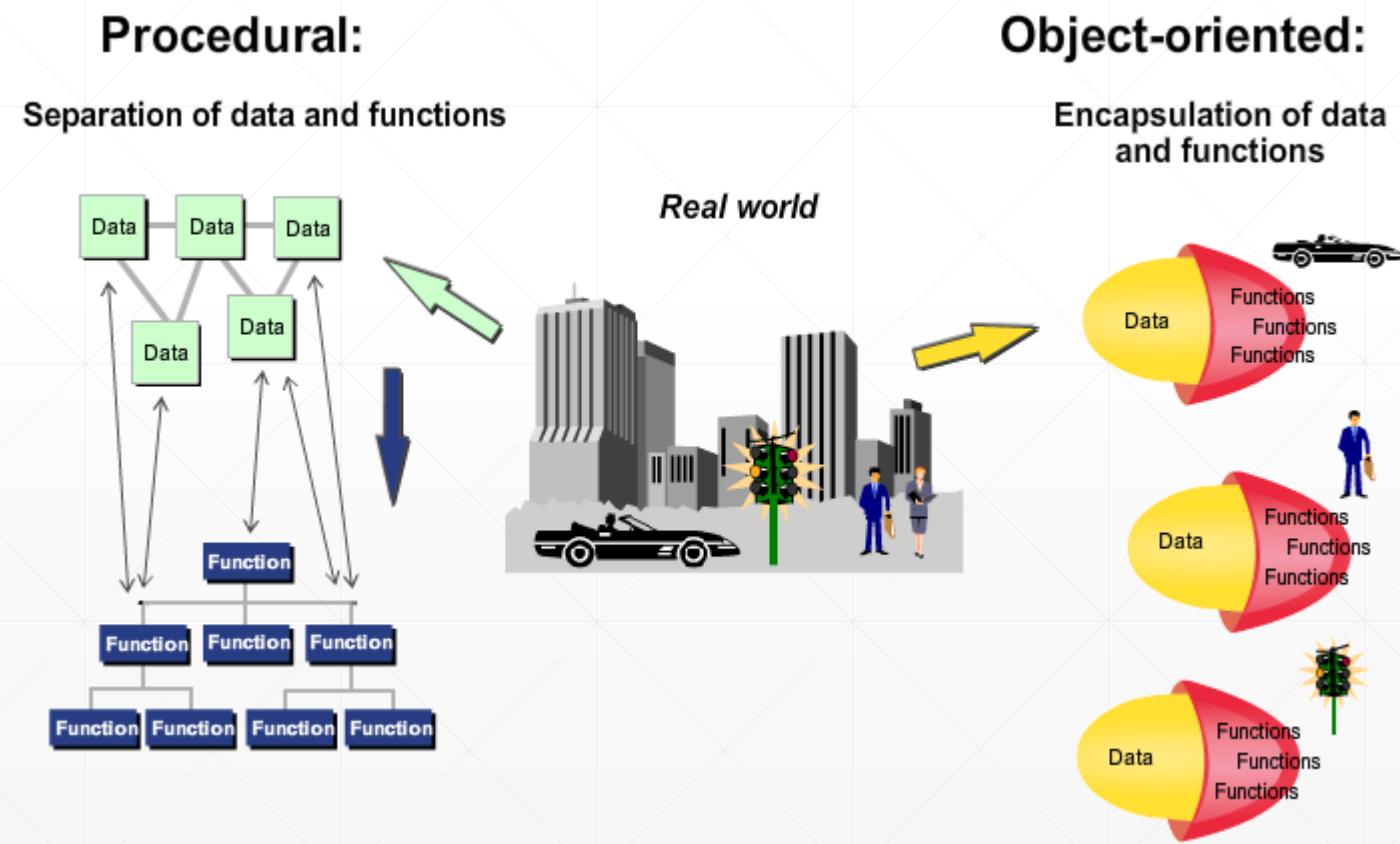
OOP: Procedural vs Object-Oriented Programming

■ Procedural

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

■ Object-Oriented

- Models a set of objects
- 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

OOP
Class

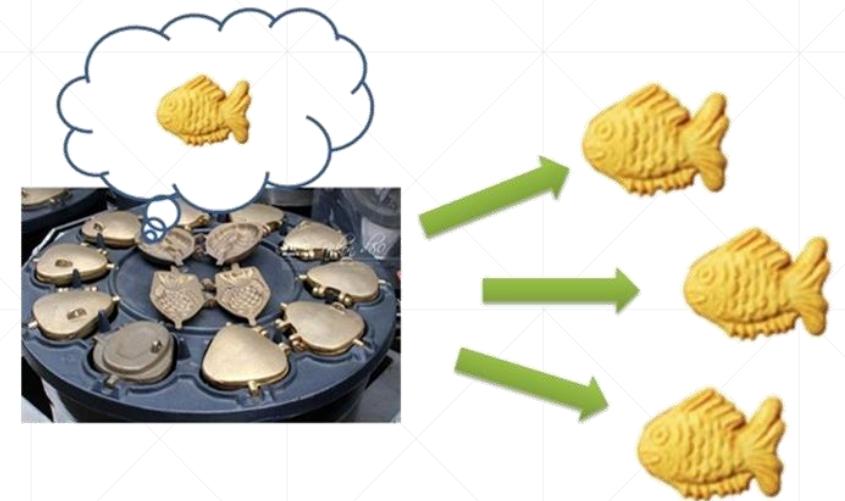
Class

■ Class

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

■ 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)

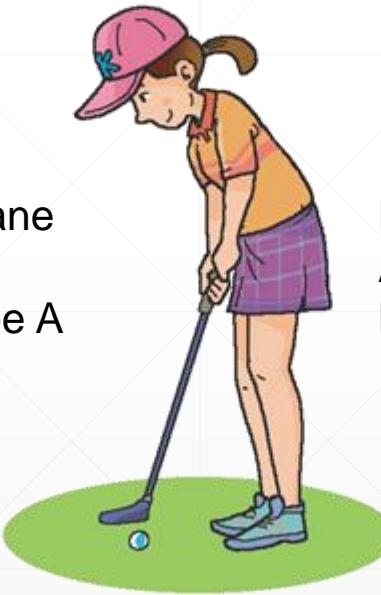
Class: Person

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



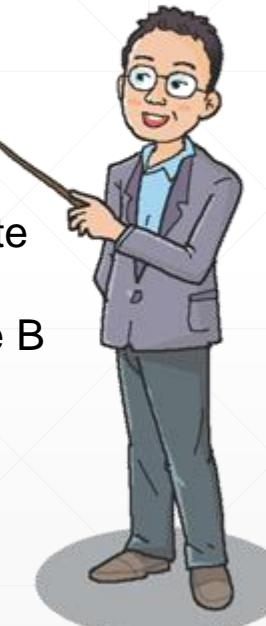
Name Jane
Age 40
Bloodtype A

Object: Jane



Name Kate
Age 30
Bloodtype B

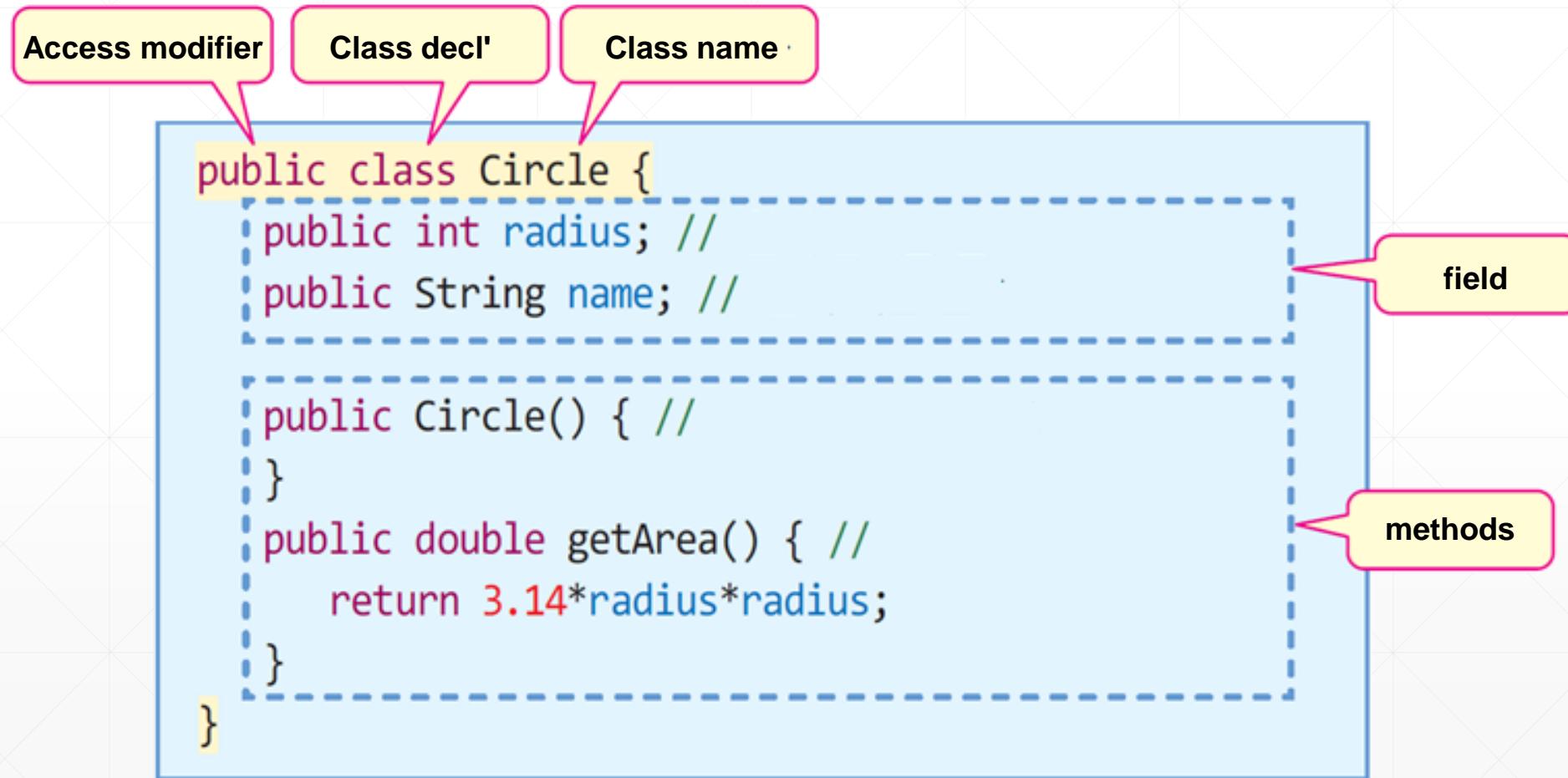
Object: Kate



Name John
Age 35
Bloodtype AB

Object: John

Class: Structure



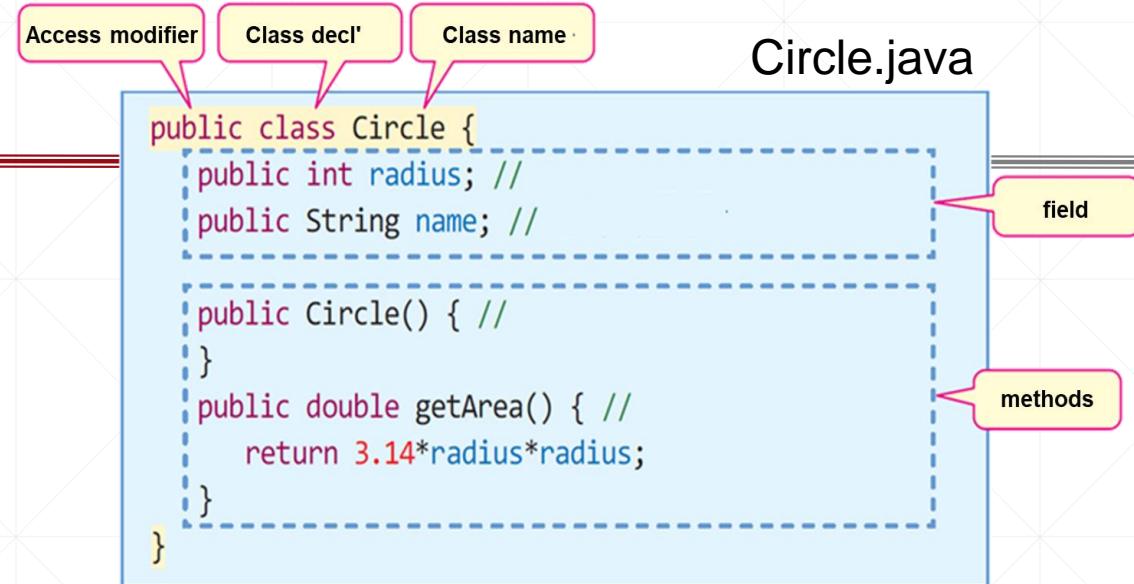
Class: Structure (cont'd)

■ 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

■ Object instantiation

- Use “new” keyword to instantiate an object
 - 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
 - Access the object members

```
new className();
```

```
objReference.member;
```

Class: Instantiation (cont'd)

1) Declaration of a reference variable

(1) Circle pizza;

pizza



2) Object instantiation using new keyword

(2) pizza = new Circle();

pizza



Circle-type object

radius



name

Memory
allocation
Instantiation

getArea() { ... }

3) Accessing object member (field)

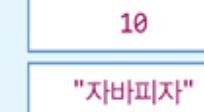
(3) pizza.radius = 10;

pizza



pizza.name = "자바피자"

radius



name

Set value
Set value

getArea() { ... }

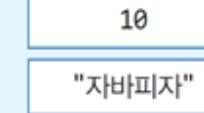
4) Accessing object member (method)

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

pizza



radius



name

Invoke
getArea()

getArea() {

...
return 3.14*radius*radius;
}

area

314.0

Class: Instantiation (cont'd)

■ Example)

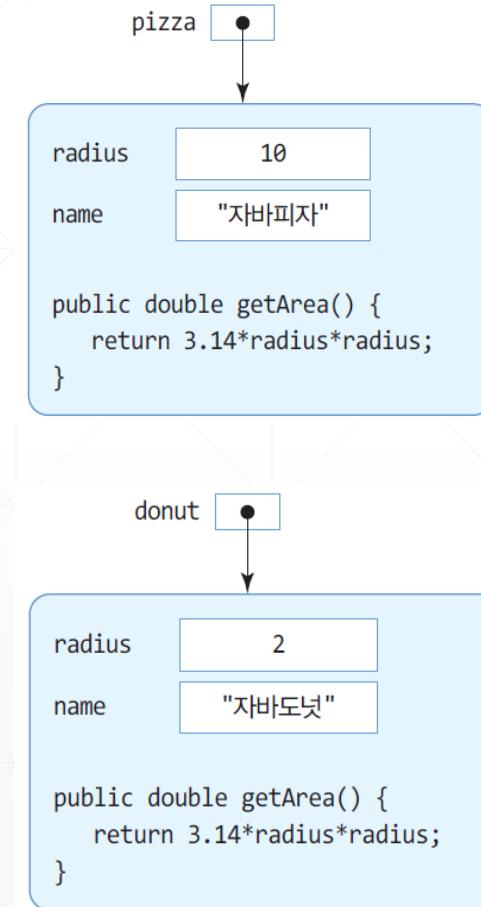
```
public class Circle { // Circle class
    int radius;           // radius field
    String name;          // name field

    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);
    }
}
```



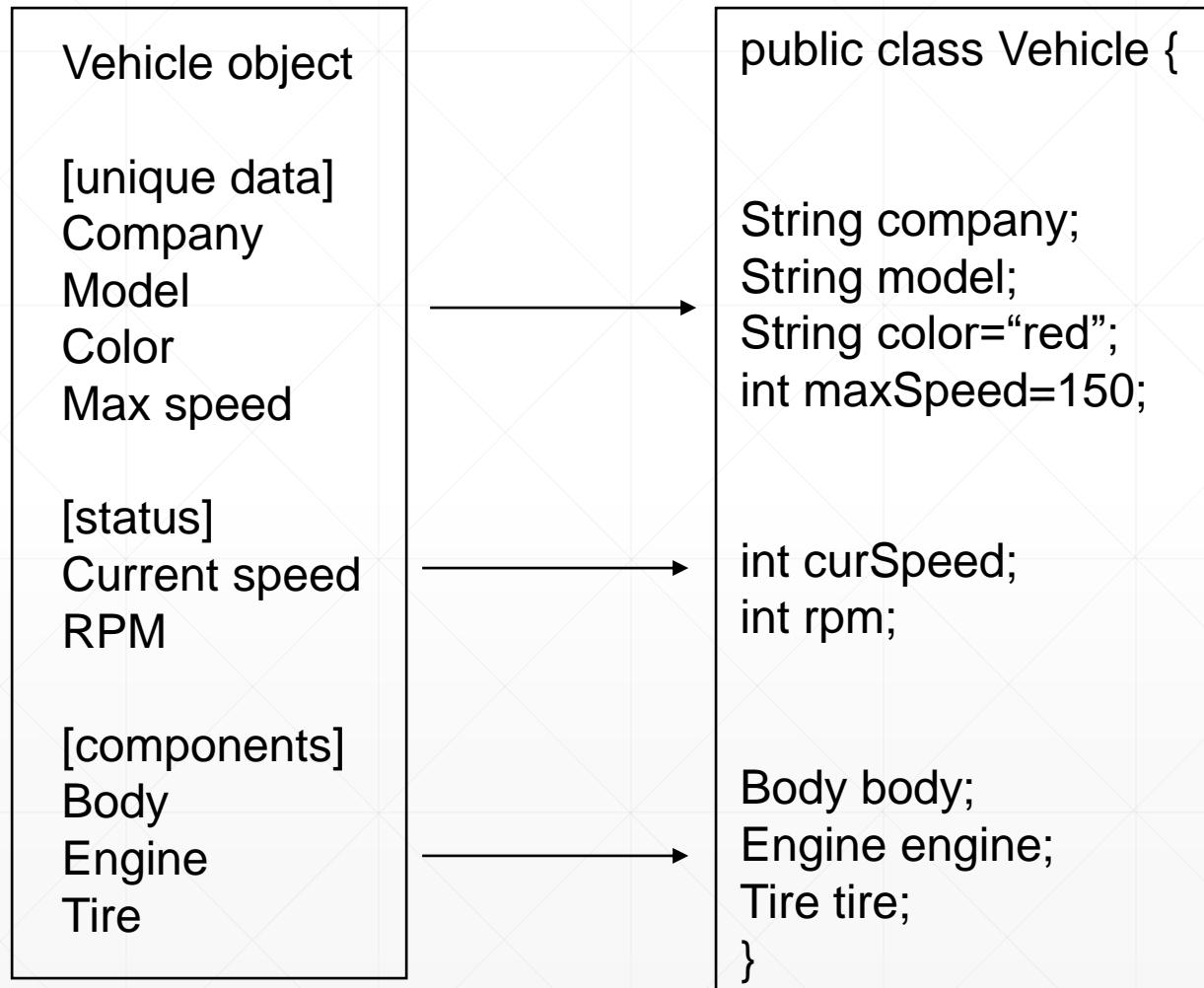
Class: Field

■ 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
		char
		short
		int
		long
	Floating-point	float
		double
	boolean	boolean
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"

```
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)

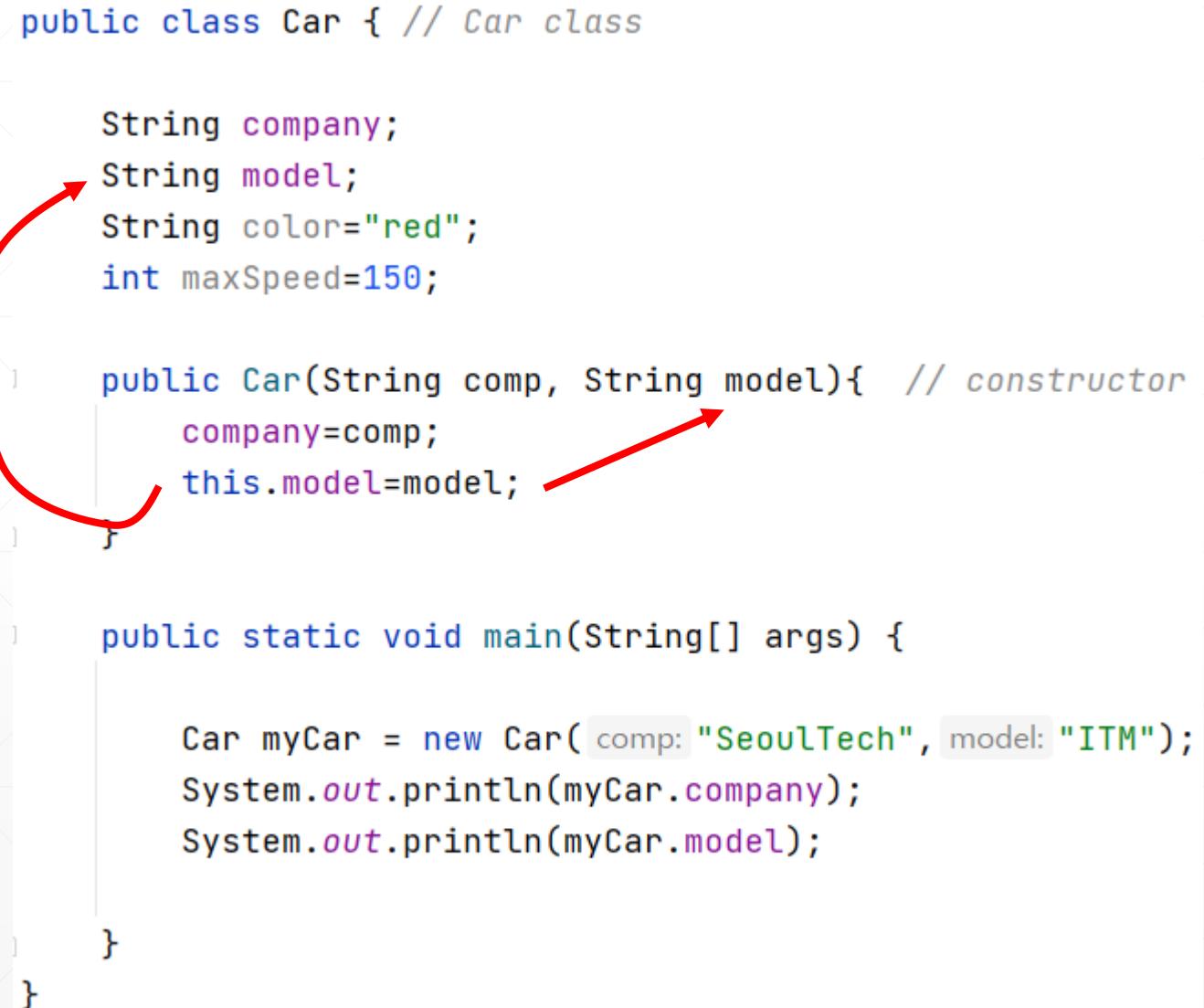
■ Name conflict

- 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;  
}
```

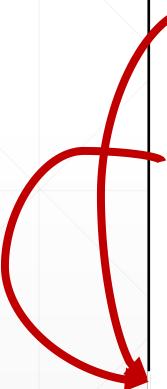
[Constructor Overloading]

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

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

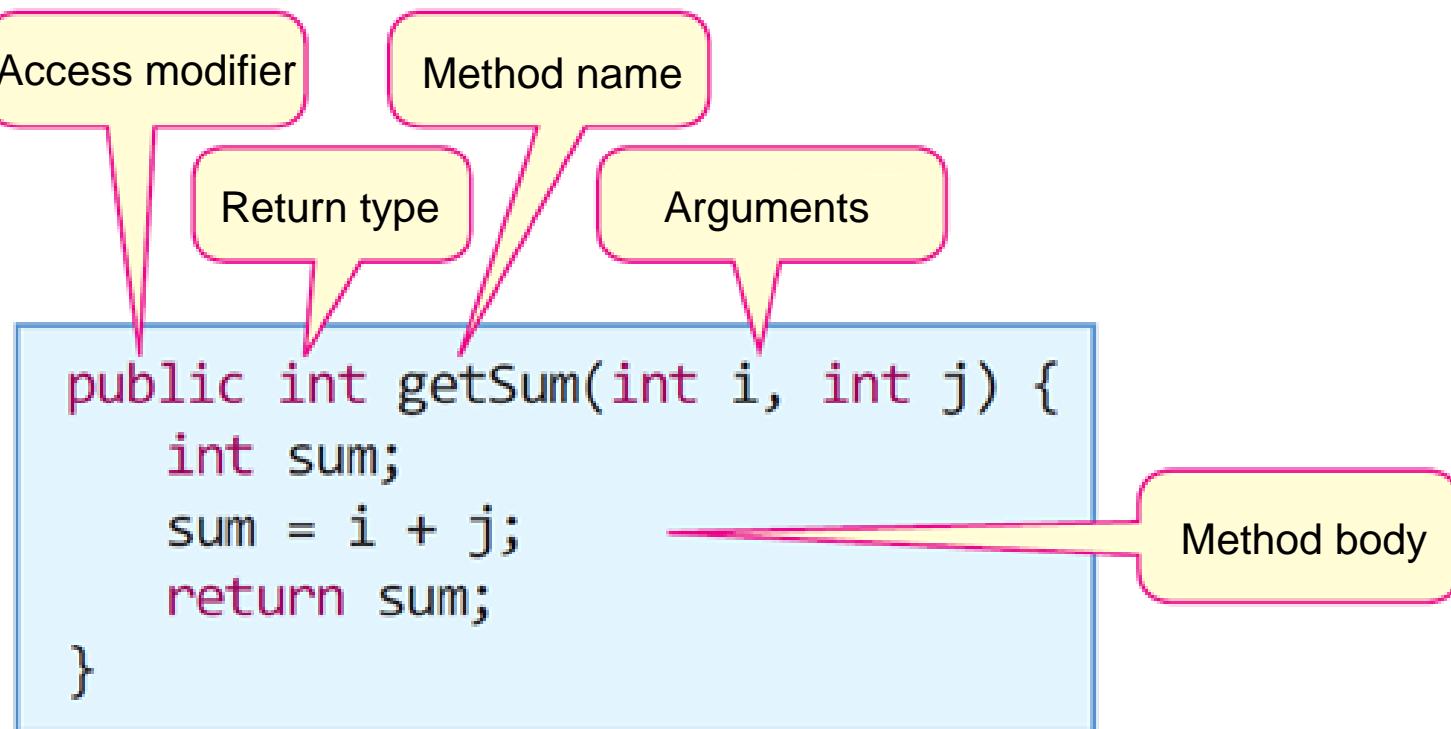


```
Car(String model) {  
    this("SeoulTech",model,150);  
}  
  
Car(String company, String model) {  
    this(company,model,100);  
}  
  
Car(String company, String model, int maxSpeed) {  
    this.company = company;  
    this.model = model;  
    this.maxSpeed = maxSpeed;  
}
```

Class: Method

■ Behavior of a class

- Behavior of a class is implemented through a method



Class: Method (cont'd)

■ Access modifier

- public, private, protected, default

■ Return type

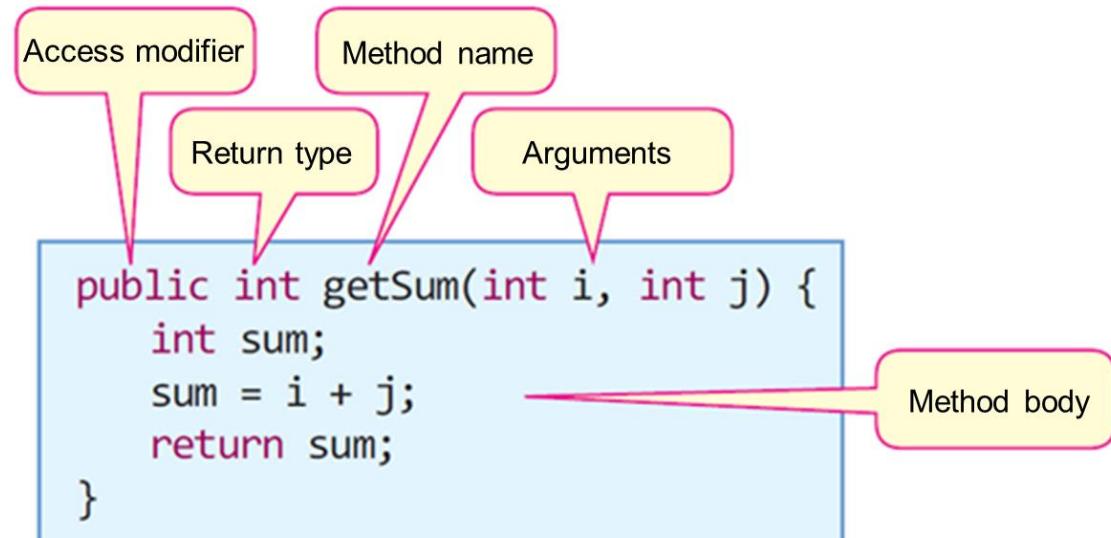
- The type of the data returned by a method
- Method can have no return values (void)

■ 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
```

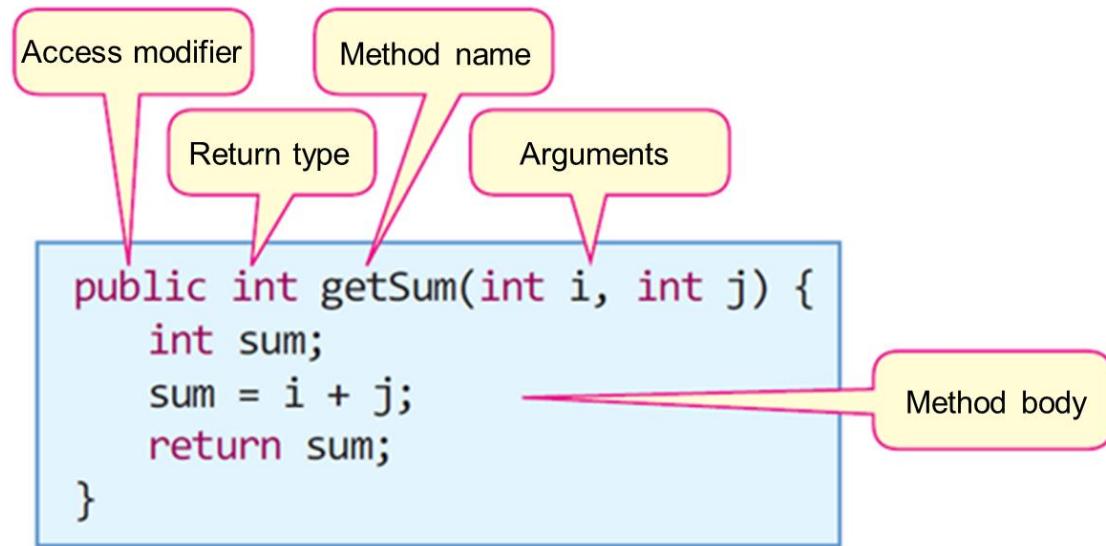
```
byte b1 = 10;
byte b2 = 20; // method call
double result = divide(b1, b2);
```

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

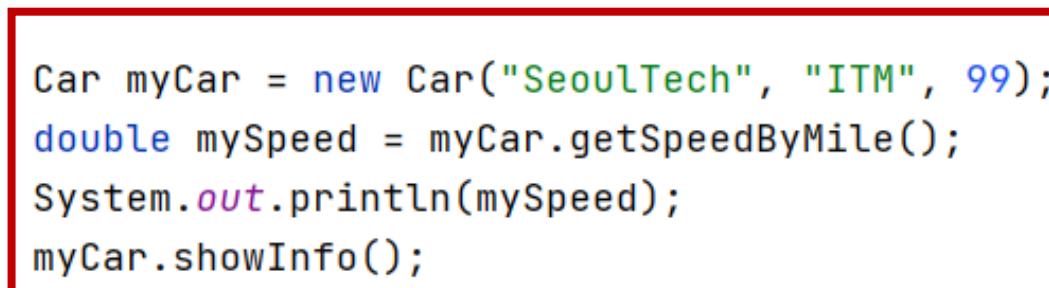
```
public double getSpeedByMile(){  
    return maxSpeed * 0.62137;  
}  
  
public void showInfo(){  
    if(maxSpeed<100) return;  
    System.out.println(company+"_"+model);  
    System.out.println(maxSpeed);  
}  
  
public static void main(String[] args) {  
  
    Car myCar = new Car("SeoulTech",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

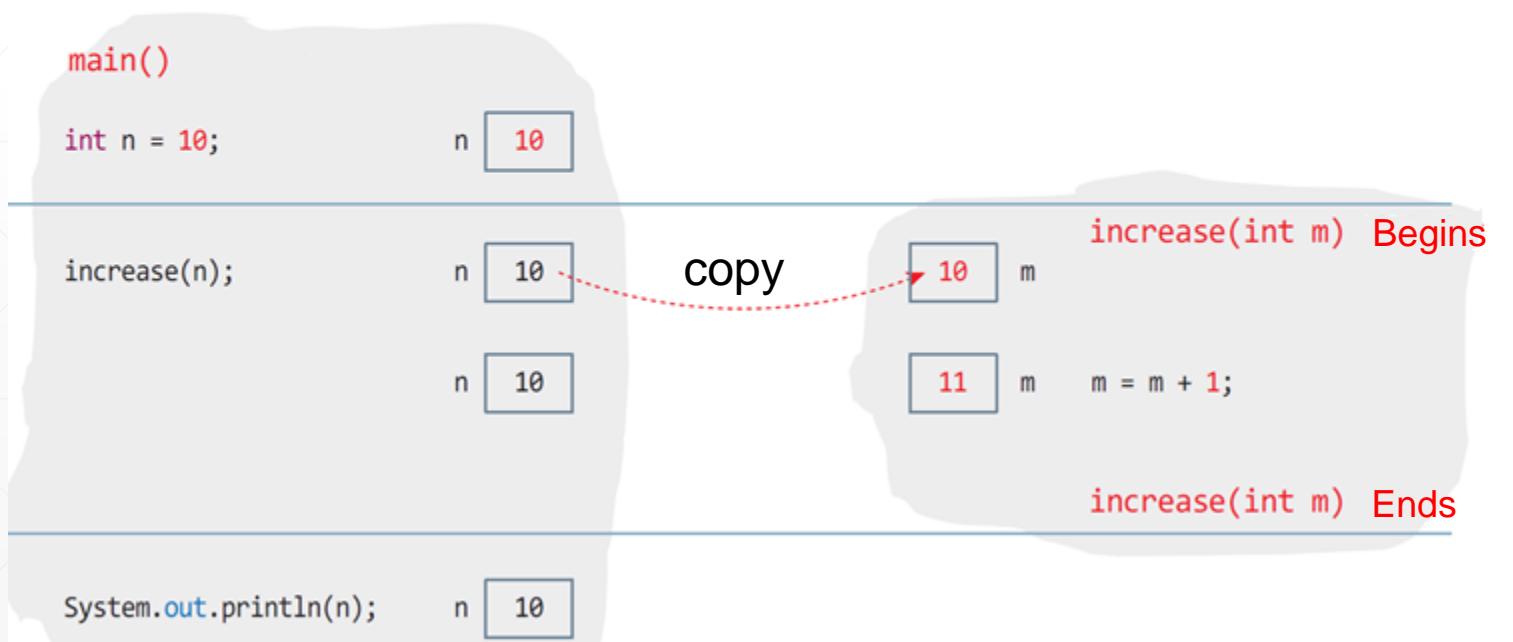
- Passing primitive-type values
 - A value is copied and then passed to the method
 - Change of the argument does not affect the original value
- Passing reference-type values (e.g., object, array, etc.)
 - A reference is passed to the method
 - Change of the argument affects the original value

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

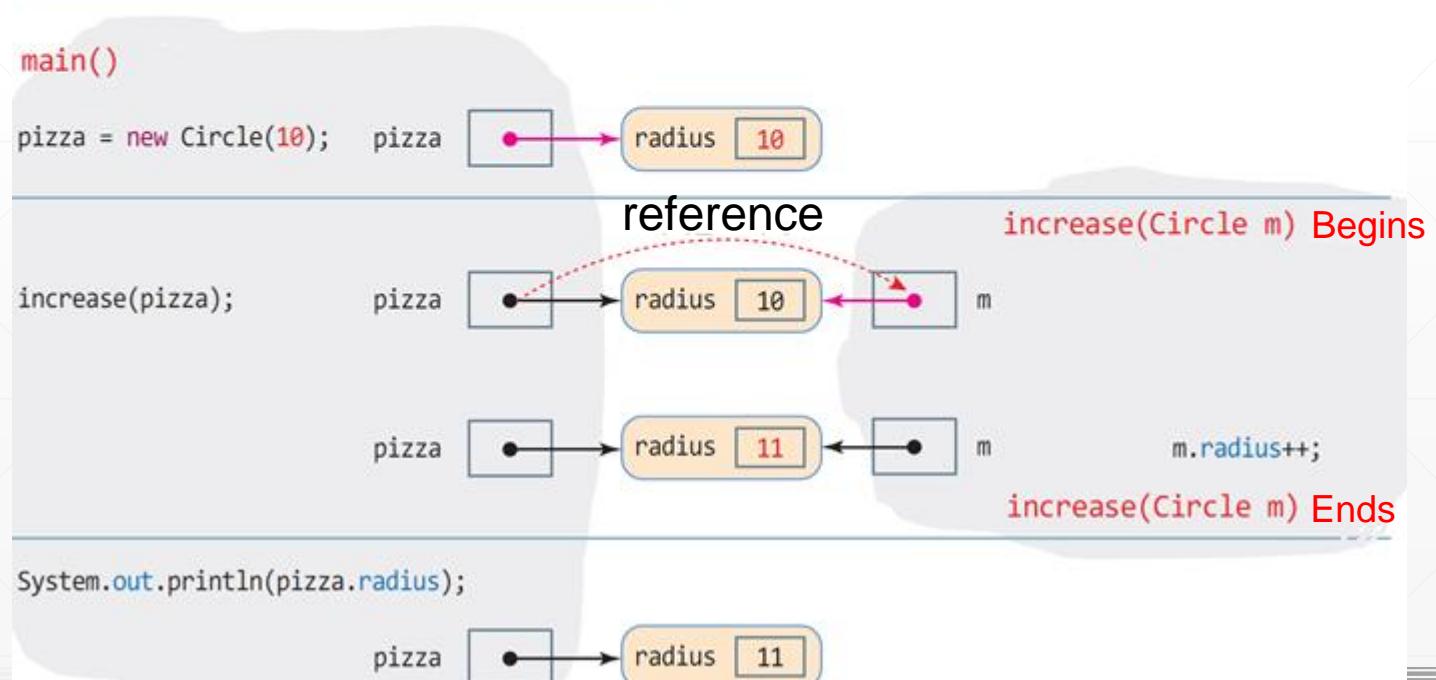
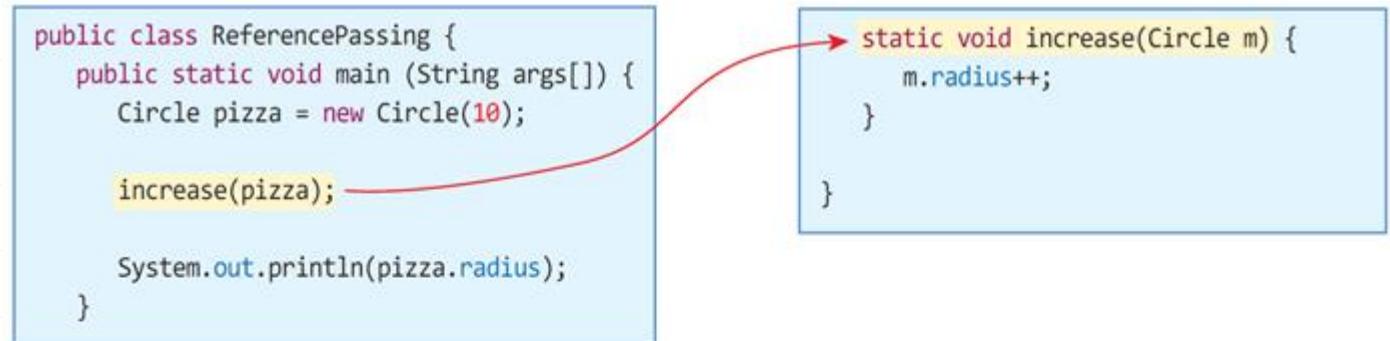
```
public class ValuePassing {  
    public static void main(String args[]) {  
        int n = 10;  
  
        increase(n);  
  
        System.out.println(n);  
    }  
  
    static void increase(int m) {  
        m = m + 1;  
    }  
}
```



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

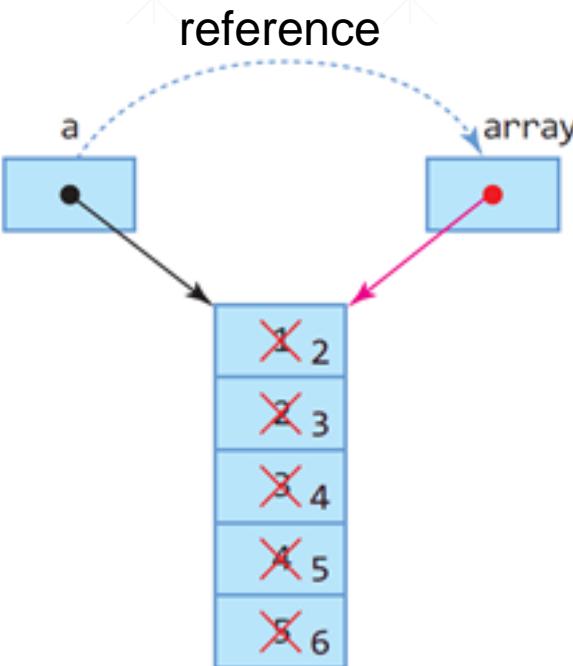


Class: Method (cont'd)

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

// Fail!

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

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

- 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 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

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

```
String company;  
String model;  
static int maxSpeed = 150;
```

```
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;  
    void f1(int x){n=x;}  
    void f2(int x){m=x;}  
  
    static int m;  
    static void s1(int x){n=x;}  
    static void s2(int x){f1(3);}  
  
    static void s3(int x){m=x;}  
    static void s4(int x){s3(3);}  
}
```

for
for
OK
OK

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

Method: Static Member (cont'd)

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

```
class Calc {  
}  
  
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 {  
  
}  
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();  
    }  
}
```

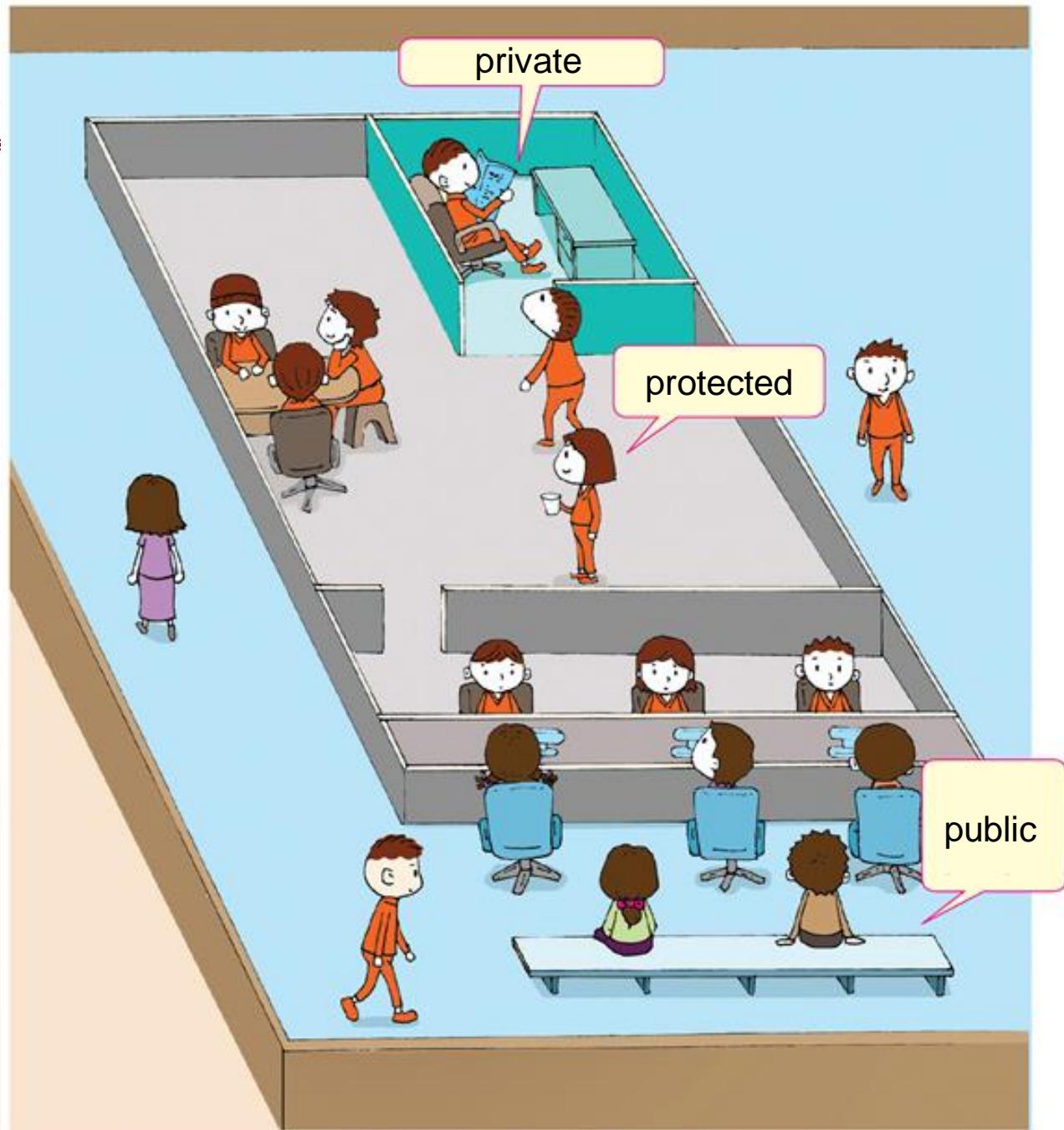
Exchange rate (1\$)>> 1200
1M Won is \$833.333333333334
\$100 is 120000.0won

Won to Dollar = Won/rate

Dollar to Won = Dollar * rate

Method: Access Modifier

- Determine who can access!



Method: Access Modifier (cont'd)

■ Java Package

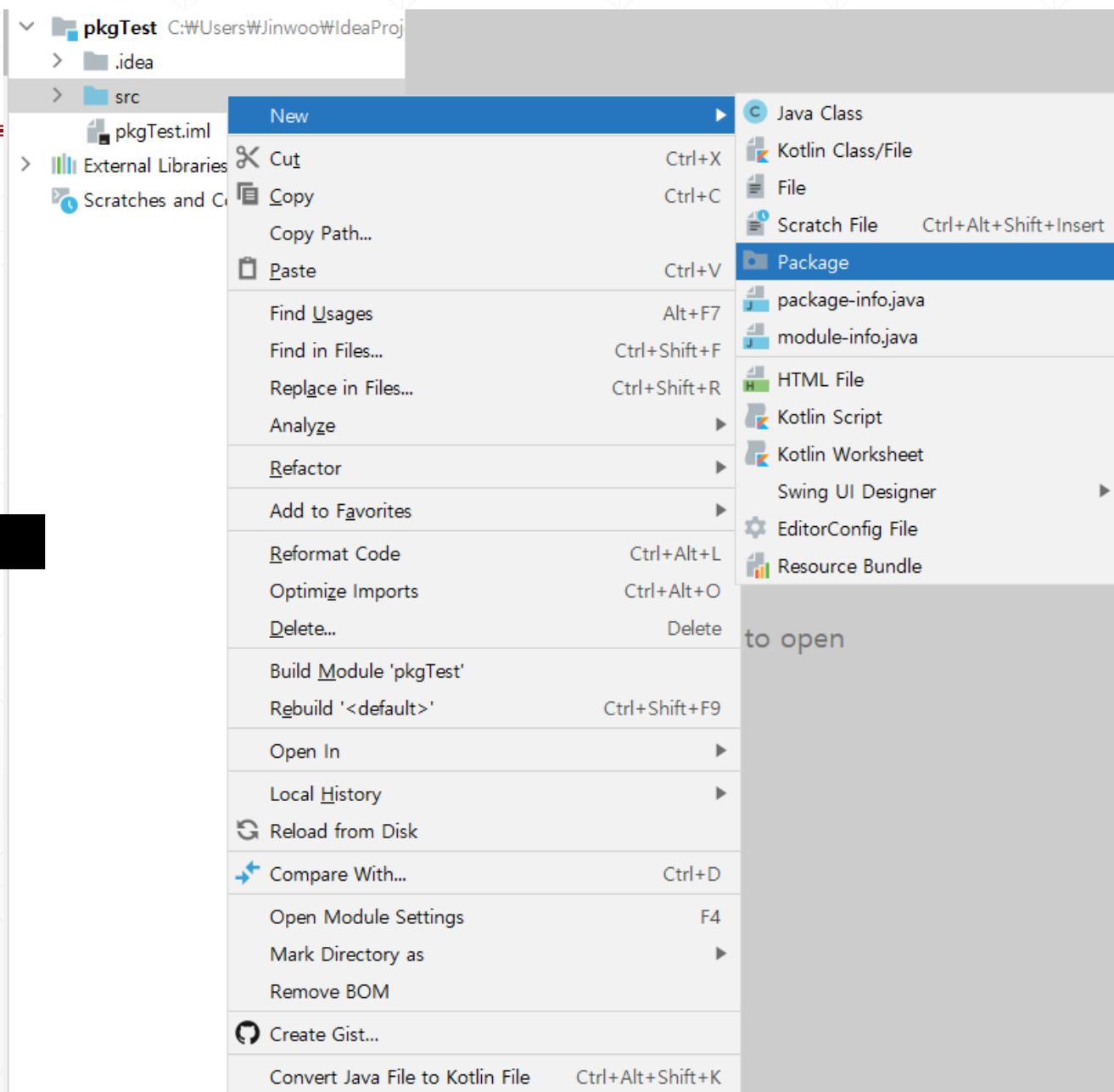
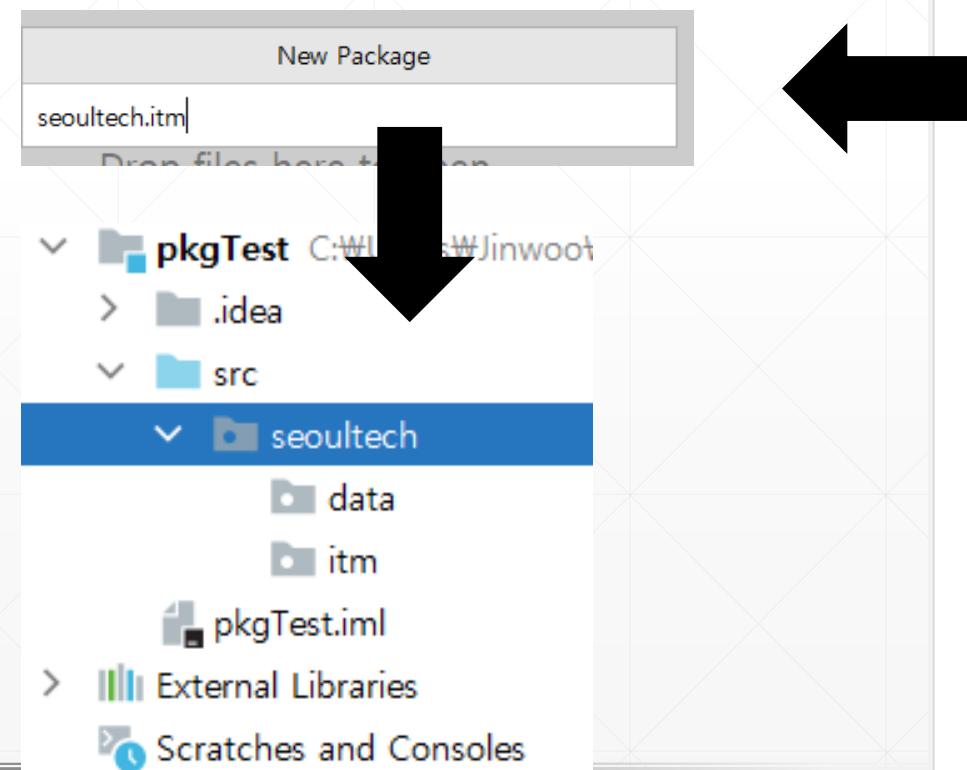
- A group of classes with similar features/categories
- Similar to “folder” in Filesystem to manage files
- Class name is composed of
 - Package names (hierarchy)
 - Class name
- Class name can be uniquely identified by its package names
 - superPkg.subPkg1.myClass
 - superPkg.subPkg2.myClass

These two classes are different!

Method: Access Modifier

Java Package

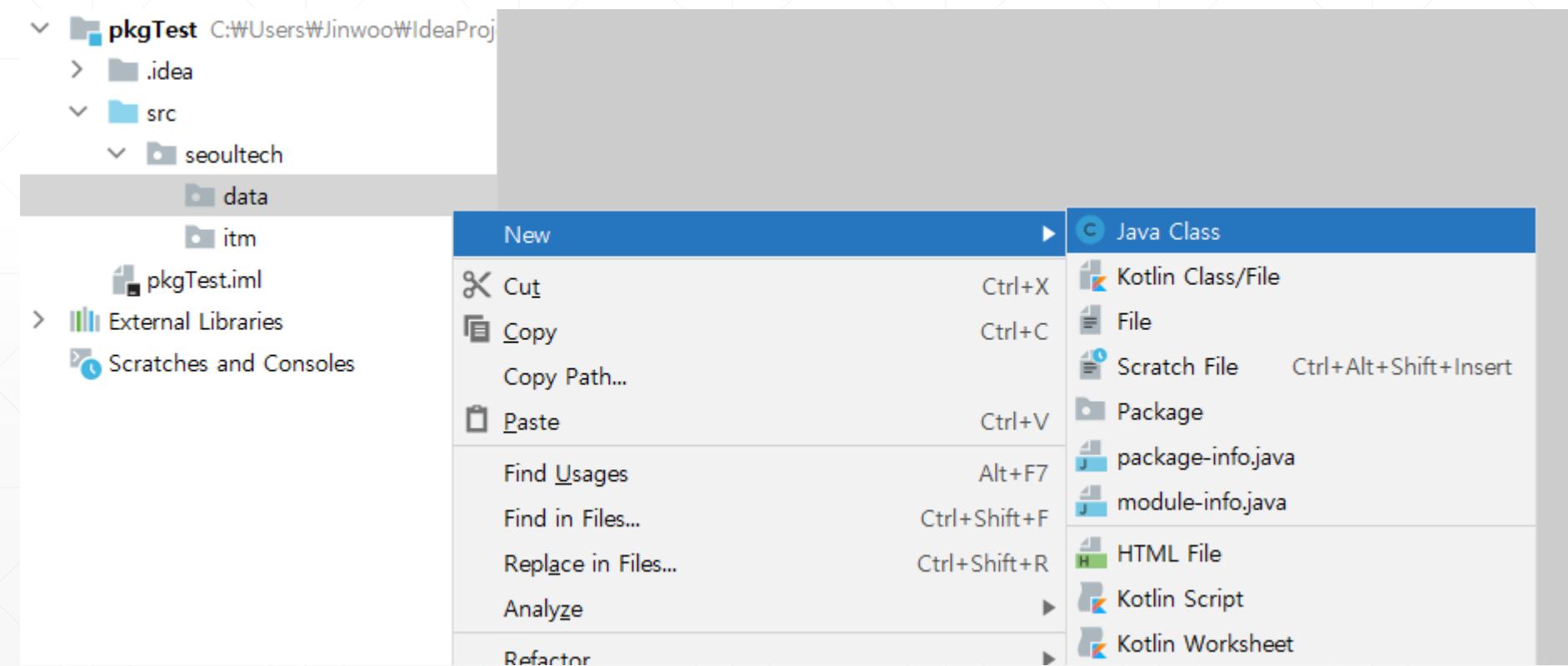
- How to create a package?
 - src → new → package
 - Choose your package name



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 file explorer on the left and a code editor on the right. The file explorer displays a project structure under 'pkgTest'. The 'src' folder contains a 'seoultech' package, which has a 'data' folder containing 'DataMath' and an 'itm' folder containing 'ComLang'. The 'ComLang' file is selected and highlighted with a blue bar at the bottom of the file list. The code editor shows the following Java code:

```
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 classes: "DataMath" and "ComLang". A third "DataMath" class is shown in the code editor.
- Code Editor (Line 1): `package seoultech.itm;` (highlighted with a red box)
- Code Editor (Line 4): `public class DataMath {`
- Code Editor (Line 5): `public String msg="ITM's datamath";`
- Code Editor (Line 6): `}`

Bottom Instance:

- File Tree: Shows the same "pkgTest" project structure.
- Code Editor (Line 1): `package seoultech.data;` (highlighted with a red box)
- Code Editor (Line 4): `public class DataMath {`
- Code Editor (Line 5): `public String msg="Data's datamath";`
- Code Editor (Line 6): `}`

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 'itm' include 'pkgTest.iml', 'External Libraries', and 'Scratches and Consoles'. On the right is the Code Editor showing 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 sections of the code are highlighted with red boxes: the 'import java.util.Scanner;' statement and the 'System.out.println(dm.msg);' line.

Method: Access Modifier (cont'd)

Java Package

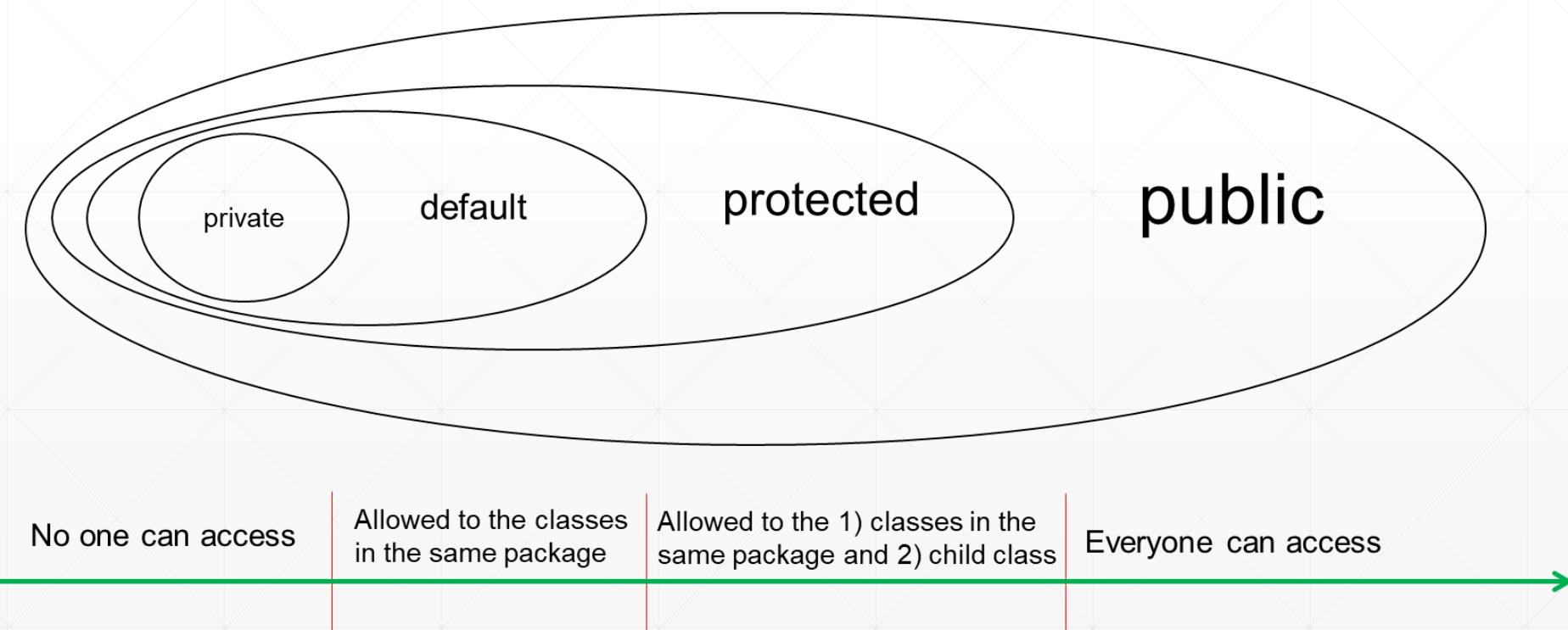
- To use a class in another package, we need to import it first!
- We can access a certain class in the same package using its class name

The screenshot shows the IntelliJ IDEA interface. On the left, the Project tool window displays a file structure for a project named 'pkgTest'. The 'src' directory contains a 'seoultech' package with 'data' and 'itm' sub-directories. Inside 'data', there is a 'DataMath' class. Inside 'itm', there is a 'ComLang' class and another 'DataMath' class. The 'ComLang' class is currently selected. On the right, the Code Editor shows the Java code for 'ComLang'. The code imports 'DataMath' from the same package and 'Scanner' from the standard library. It defines a 'ComLang' class with a main method that uses a scanner to read input and prints the result using a 'DataMath' object.

```
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 }
```

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 this **class**

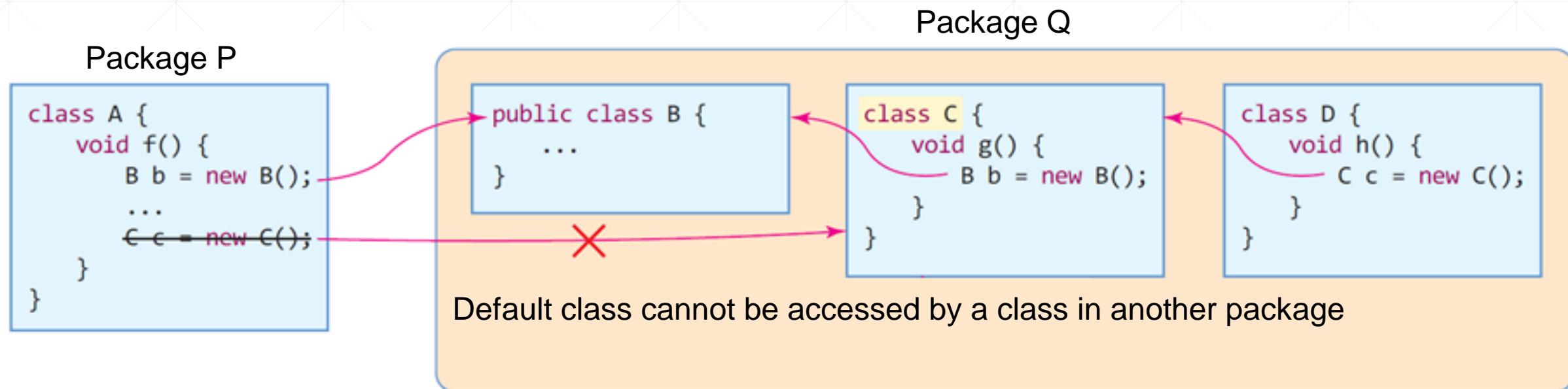
■ Member-level modifiers

- 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 even in another package can access
- Default (package-private): only the class in the same package can access

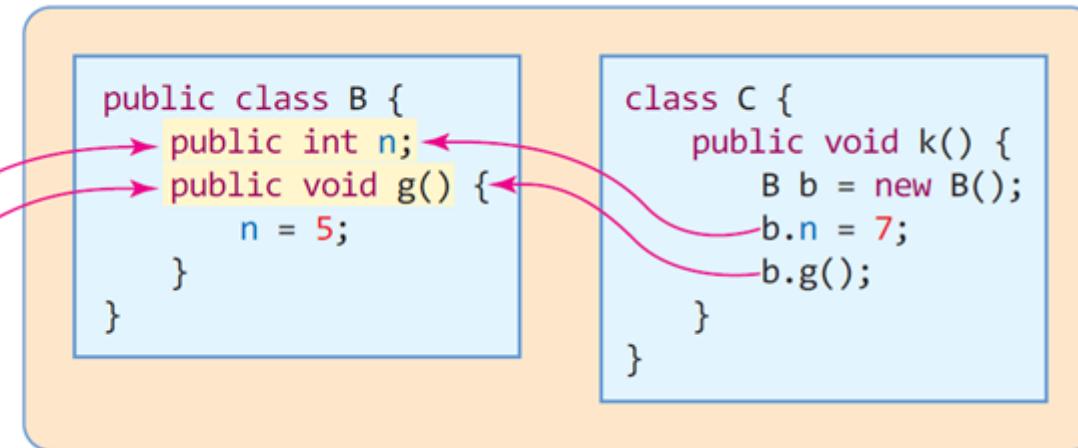
Accessor	Access modifier of a member			
	private	default	protected	public
Class in the same package	X	O	O	O
Class in another package	X	X	X O (child)	O

Method: Access Modifier

■ Member-level access modifier

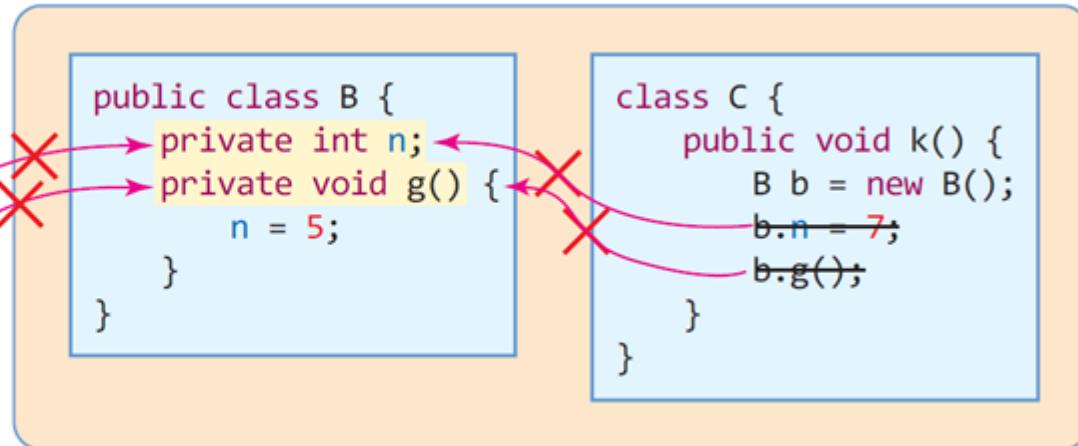
➤ Example of public modifiers

```
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();  
    }  
}
```



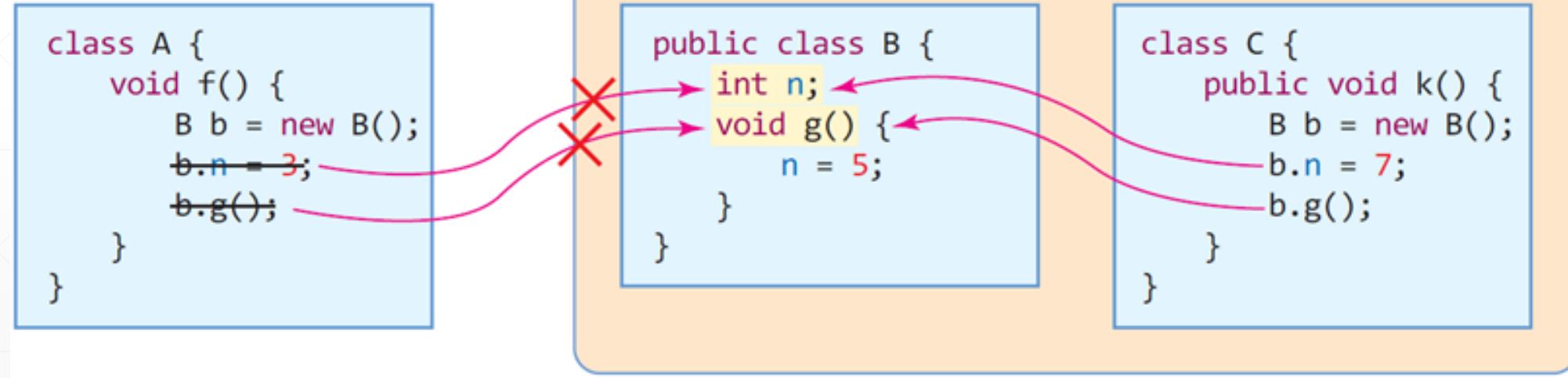
Accessor	Access modifier of a member			
	private	default	protected	public
Class in the same package	X	O	O	O
Class in another package	X	X	X	O

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

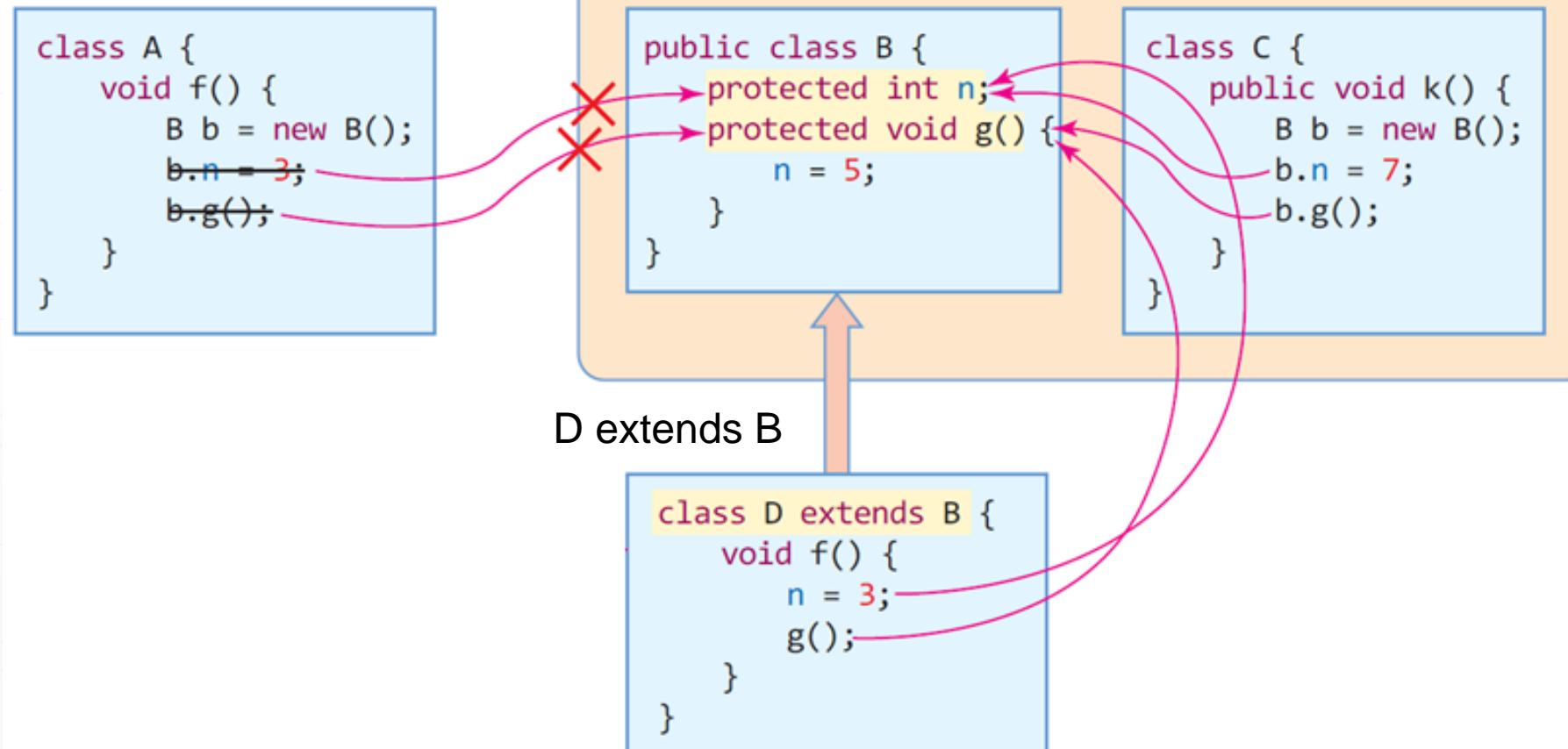


Method: Access Modifier

■ 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;
        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



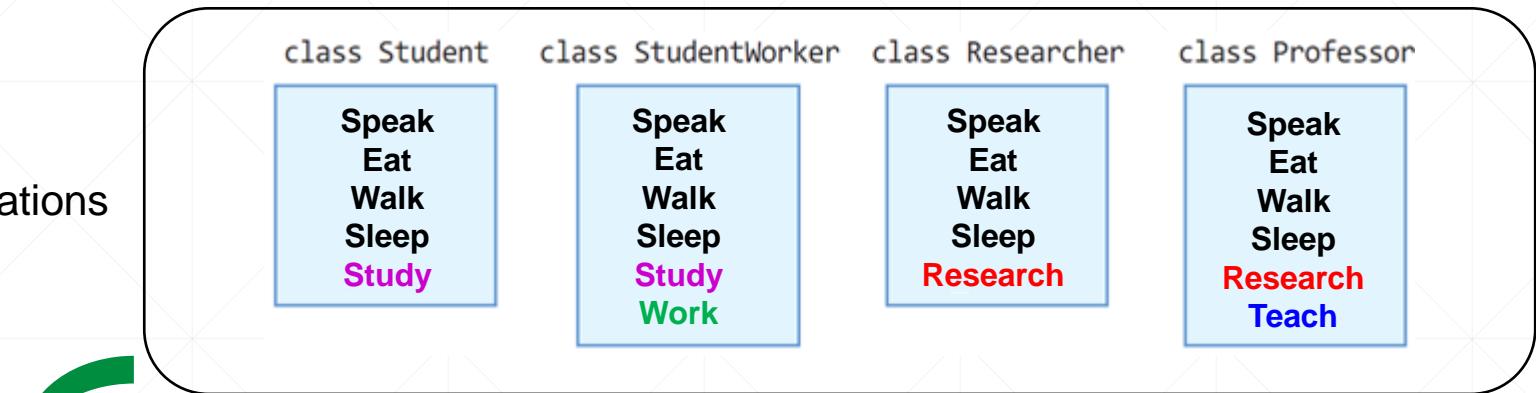
■ Inheritance in the Java world

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

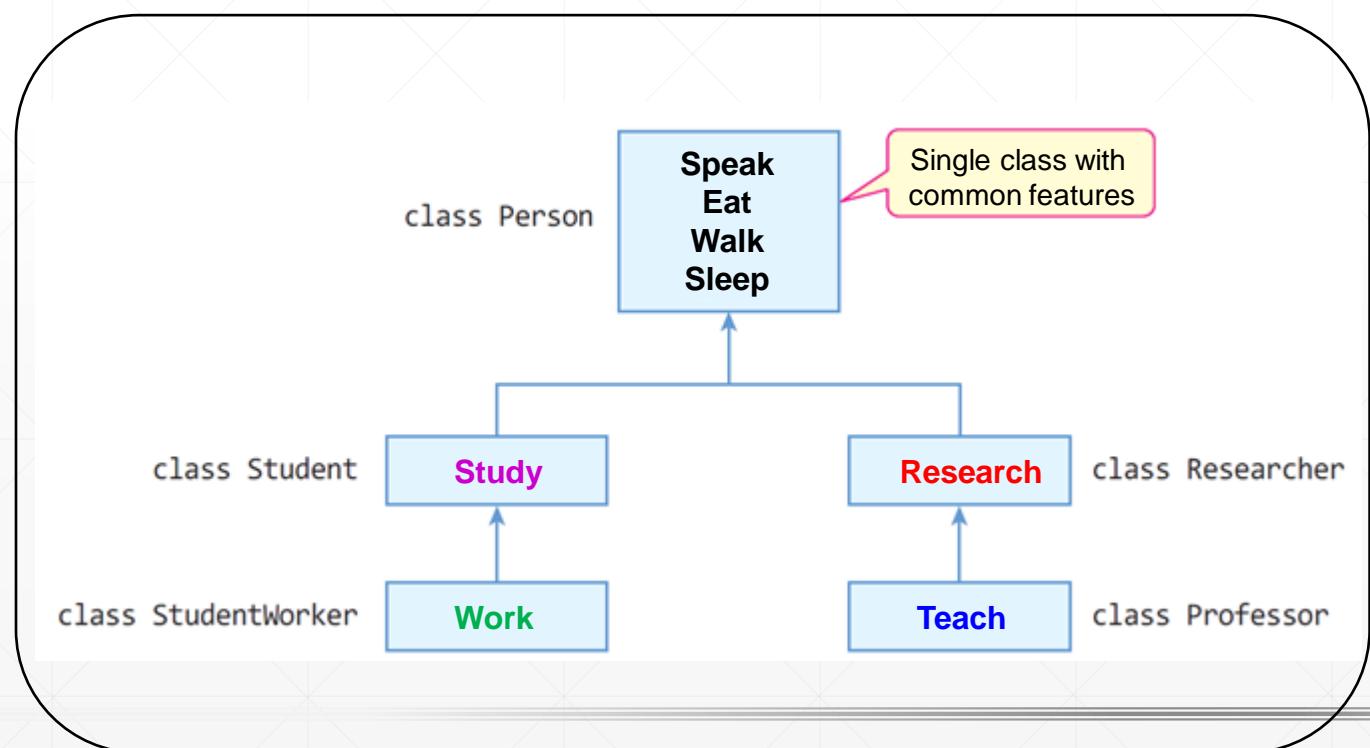
Inheritance: Concept (cont'd)

■ Example of Inheritance

4 Classes with duplicated members/implementations



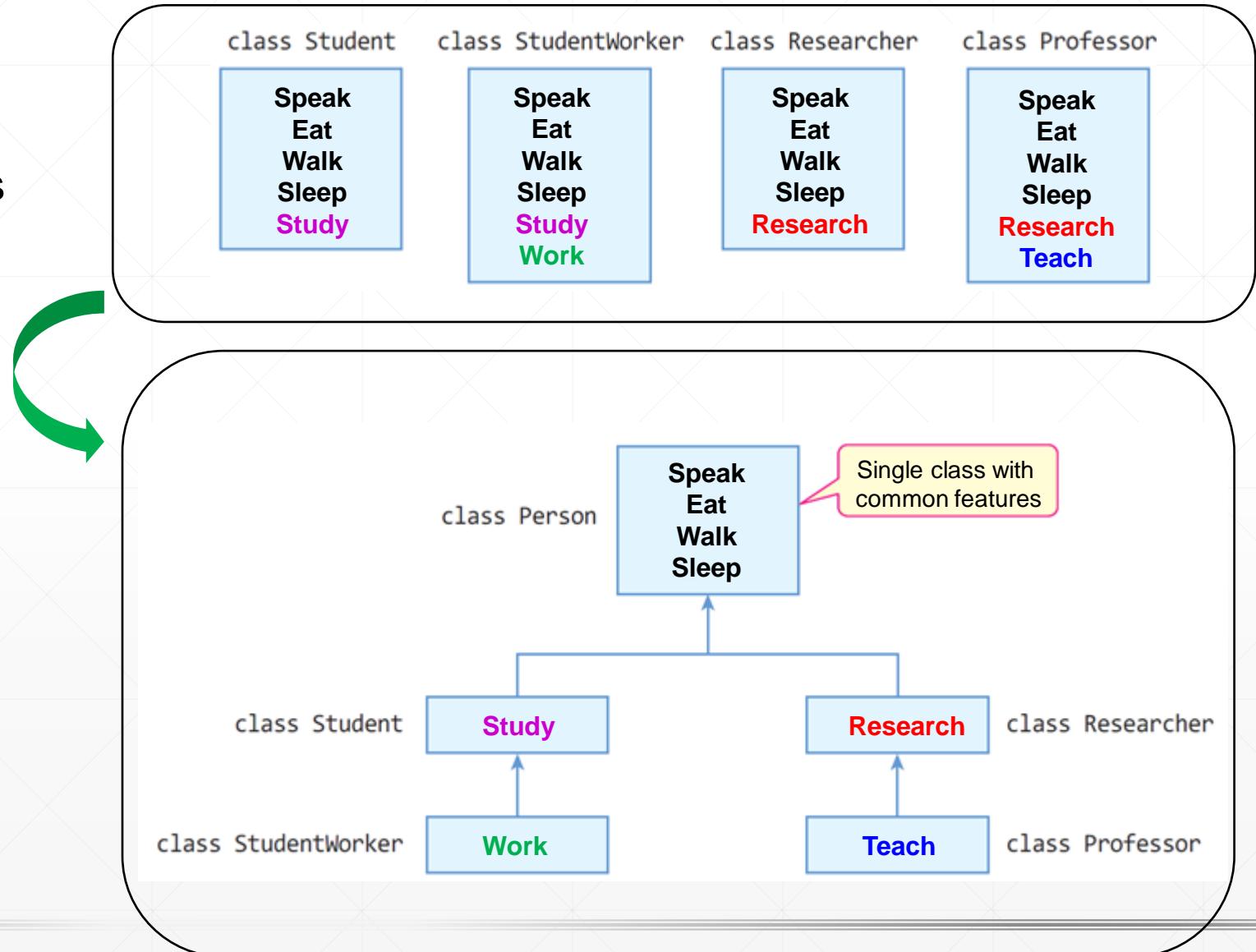
One class with common features
+
Specialized sub-classes without duplicated members



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

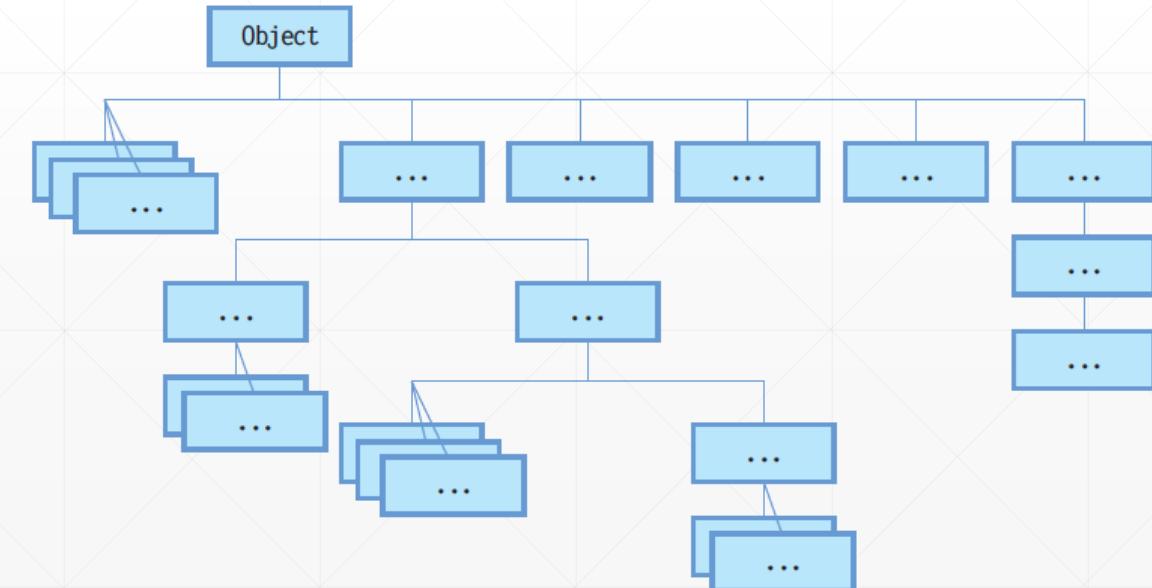
■ Super (parent) class

- 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)

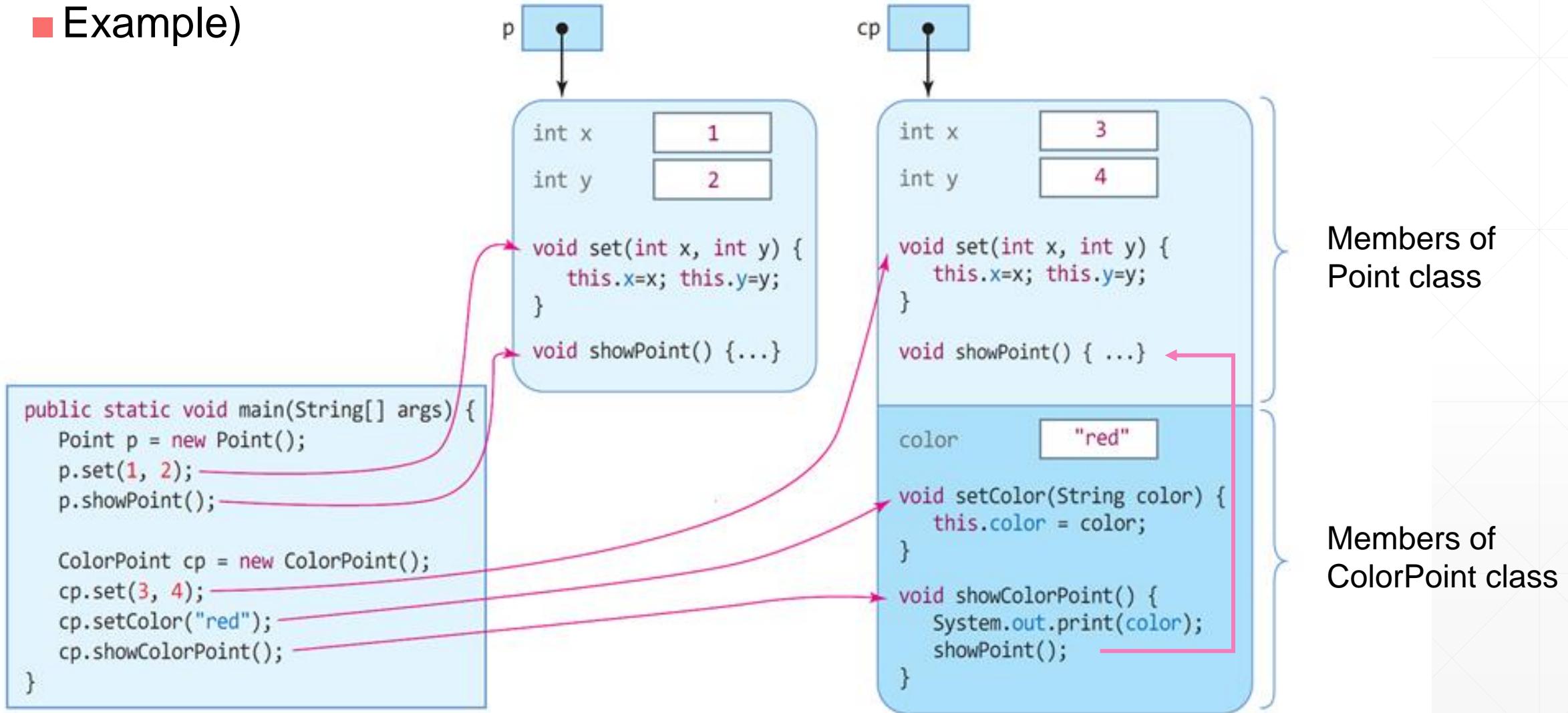
```
class Point {  
    private int x, y;  
    public void set(int x, int y) {  
        this.x = x; this.y = y;  
    }  
    public void showPoint() {  
        System.out.println("(" + x + "," + y + ")");  
    }  
}
```

```
// 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)

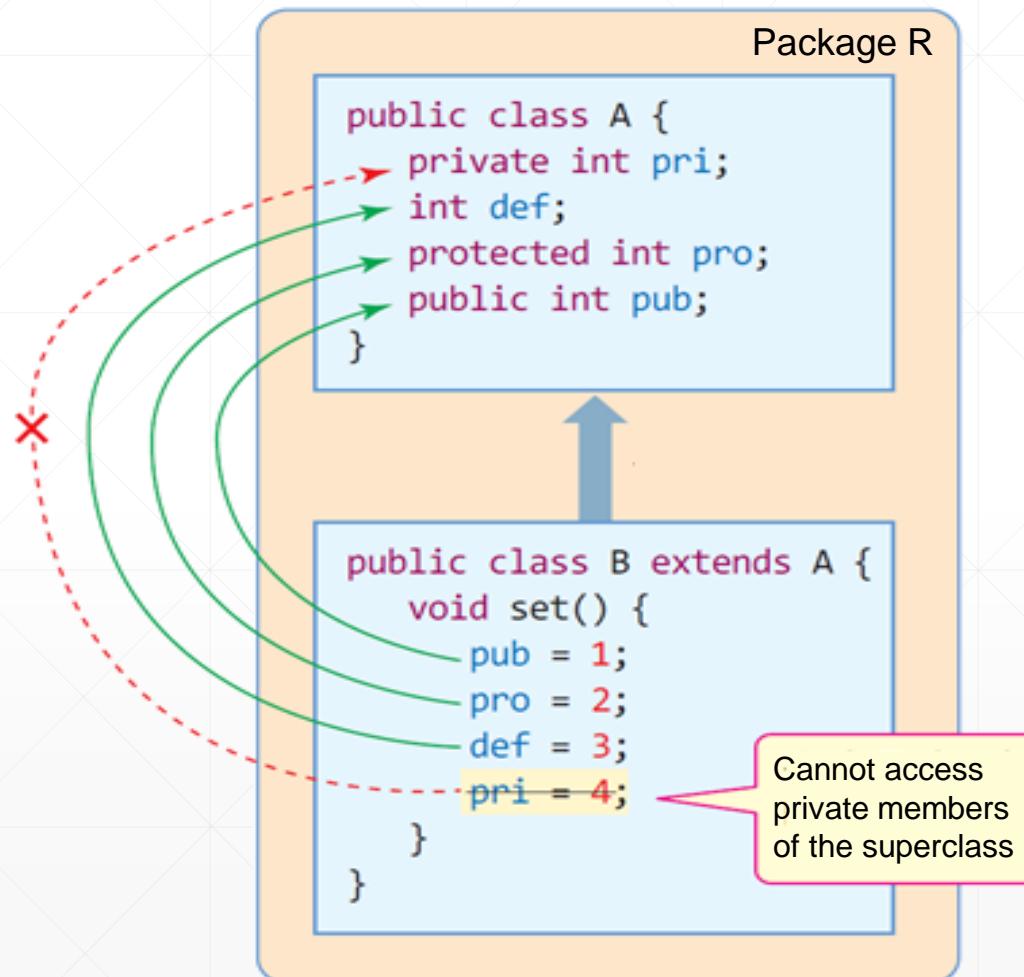


Inheritance: Access Modifier

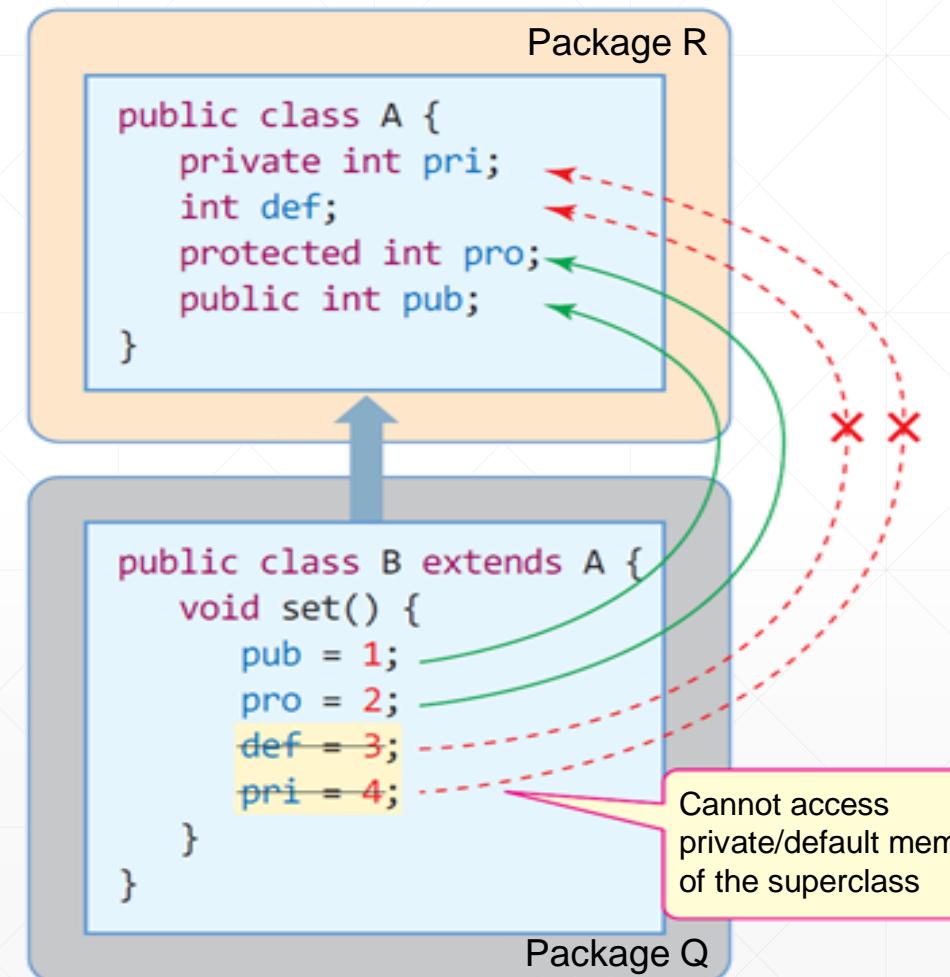
■ Access modifiers in the super class

- 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();  
    }  
}
```

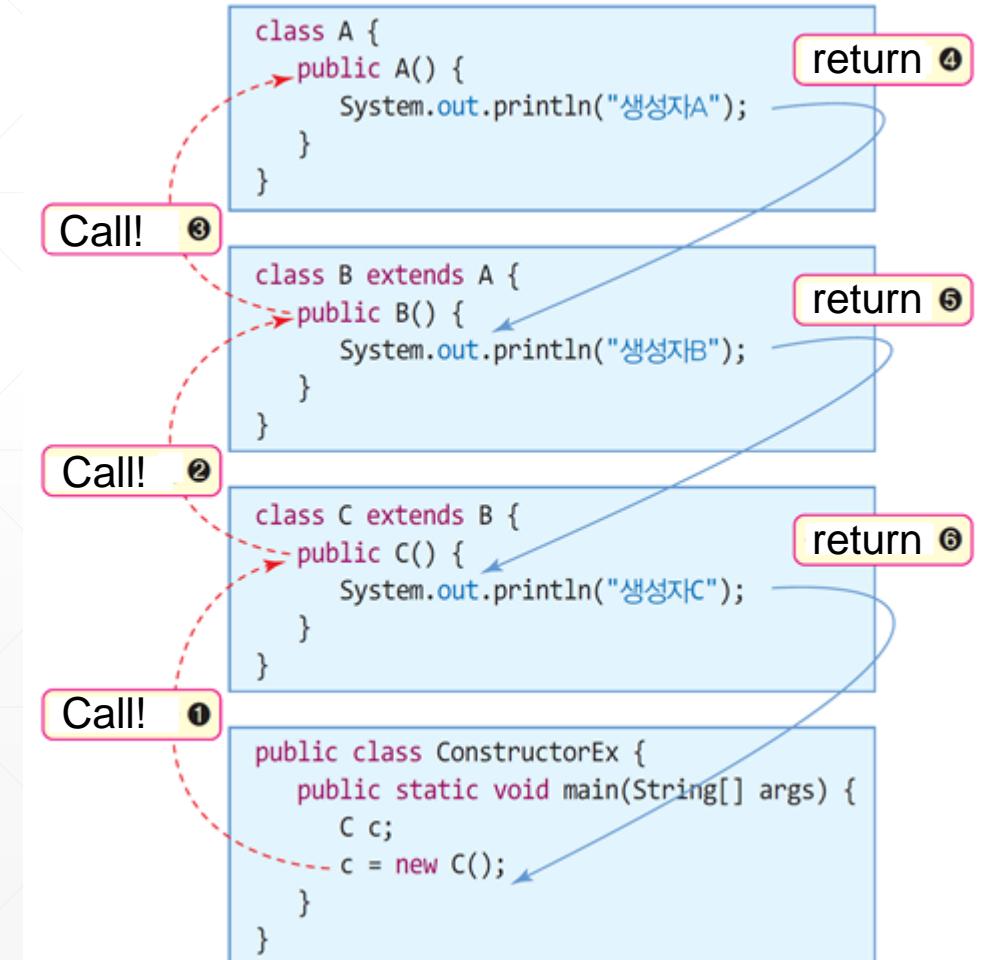
```
class Student extends Person {  
    public void set() {  
        age = 30; // OK  
        name = "Jinwoo"; // OK  
        height = 175; // OK  
        // weight = 99; // Error. Private member of superclass  
        setWeight(99); // OK  
    }  
}
```

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?



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!

“There is no default constructor available in [Superclass]”

- 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 is 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

- Up/Downcasting
- Method Overriding

Computer Language

OOP 3: Casting and Overriding



Agenda

- Casting
- Method Overriding

Class: Up/Downcasting

- Type conversion between classes
 - Similar to promotion/casting concept for primitive types

- Upcasting
 - Type conversion from sub-class to super-class

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

```
Parent parent = new Parent();
Child child = (Child) parent; ← Impossible!
```

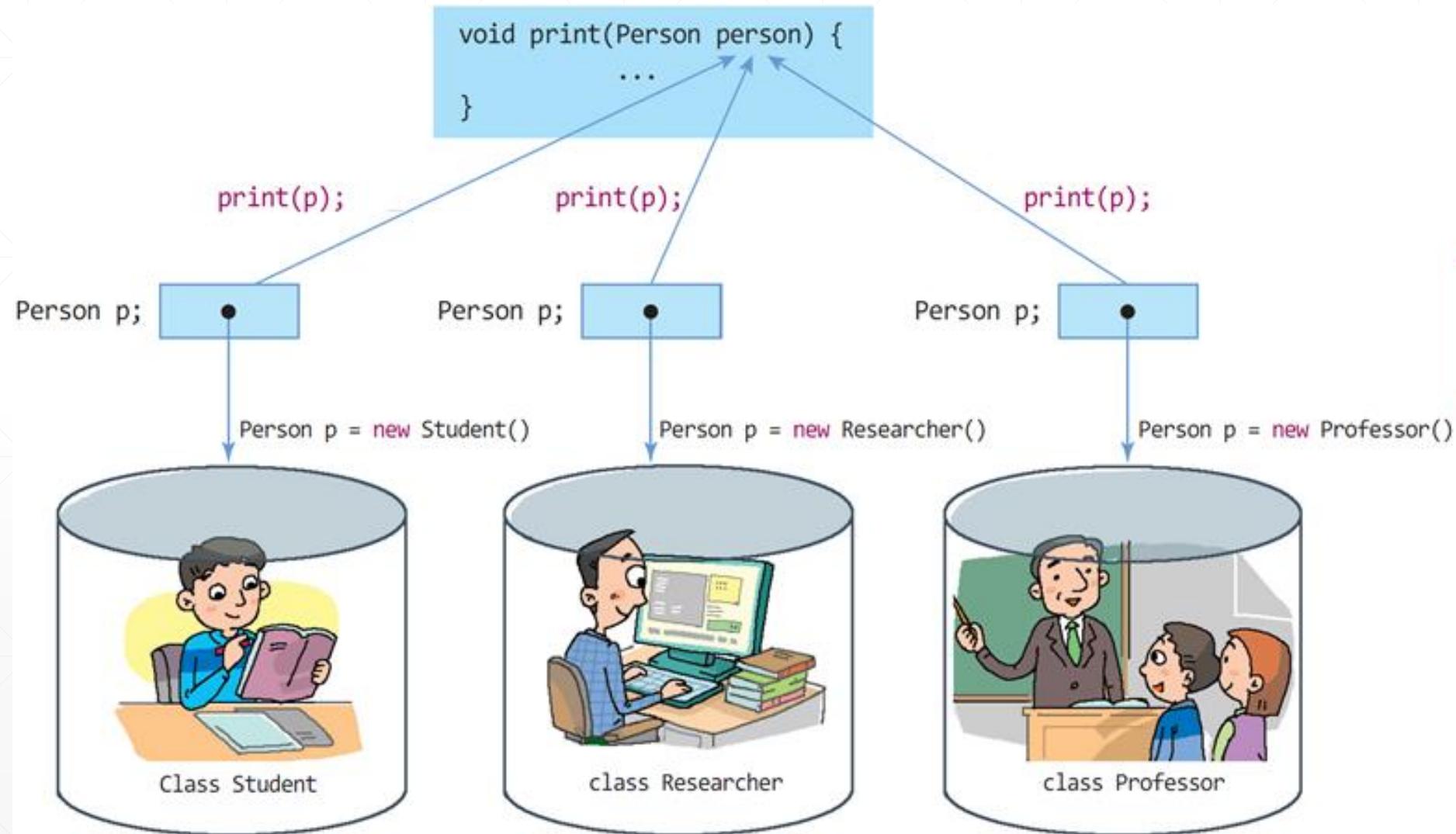
- It is impossible to infer the actual type of a upcasting reference

■ instanceof operator

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

```
objRef instanceof Classtype
```

Class: Up/Downcasting (cont'd)



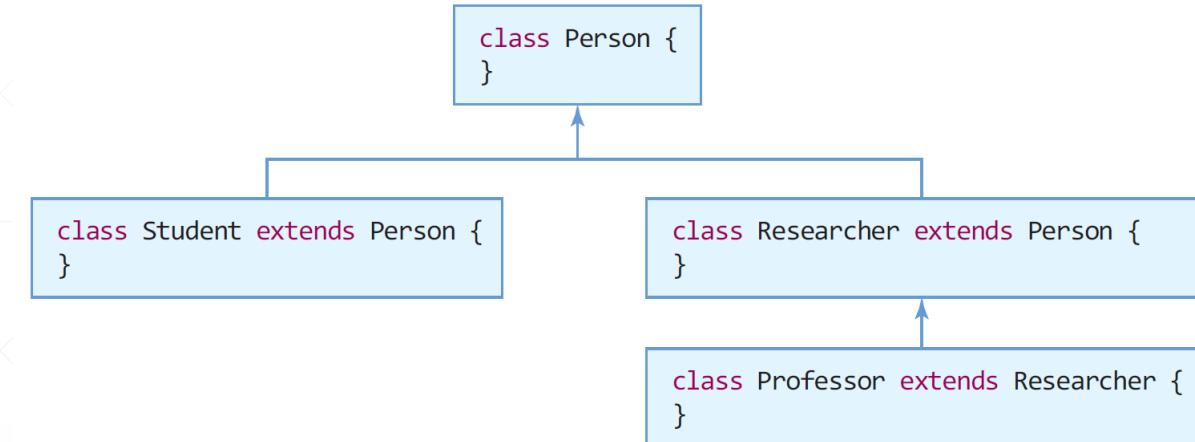
Class: Up/Downcasting (cont'd)

■ 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!
```

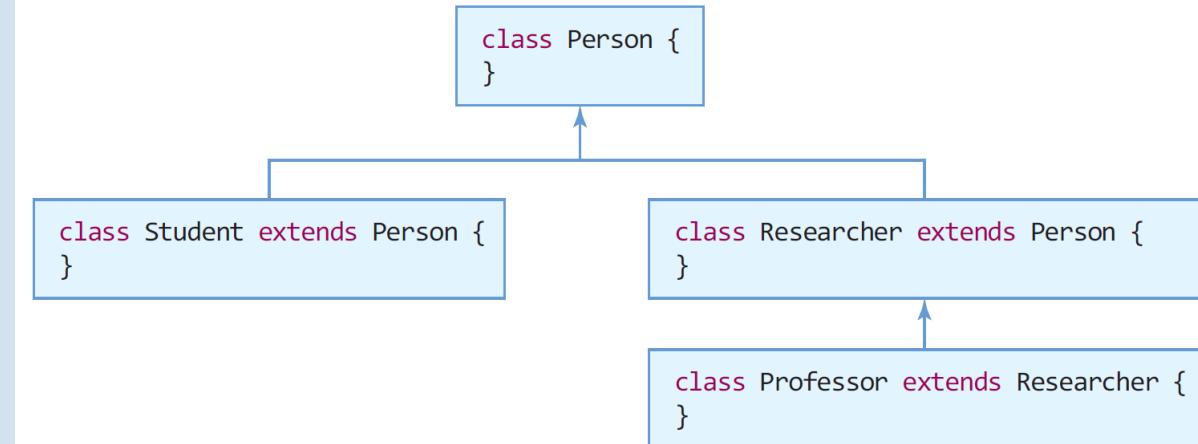
```
if("java" instanceof String)    // true
```



Class: Up/Downcasting (cont'd)

■ Example of using instanceof operator

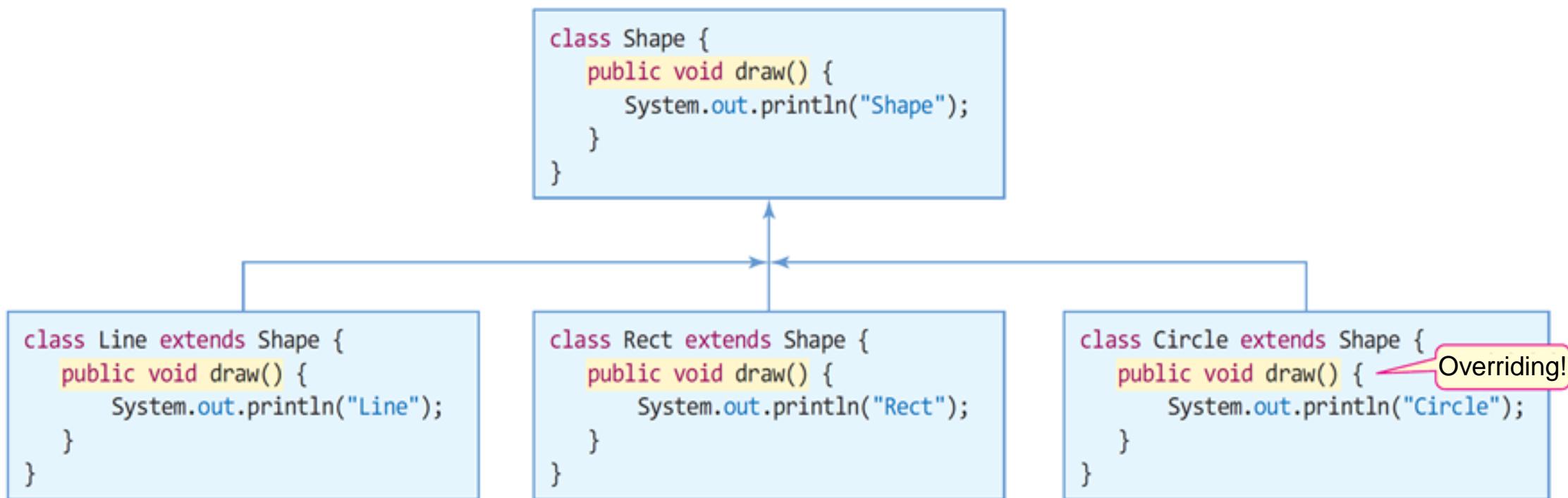
```
class Person {}  
class Student extends Person {}  
class Researcher extends Person {}  
class Professor extends Researcher {}  
  
public class InstanceOfEx {  
    static void print(Person p) {  
        if(p instanceof Person)  
            System.out.print("Person ");  
        if(p instanceof 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());  
        System.out.print("new Researcher() ->"); print(new Researcher());  
        System.out.print("new Professor() ->"); print(new Professor());  
    }  
}
```



Method Overriding

■ Redefinition of superclass's method in the subclass

- Same method signature, but different behaviors



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?

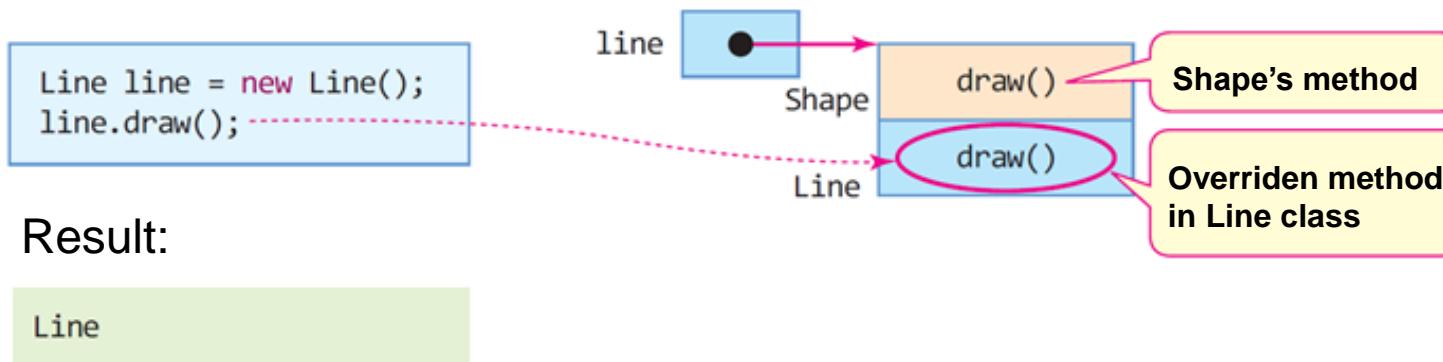
```
public class MethodOverridingEx {  
    static void paint(Shape p) {  
        p.draw(); // call overridden draw()  
  
    }  
  
    public static void main(String[] args) {  
        Line line = new Line();  
        paint(line);  
        paint(new Shape());  
        paint(new Line());  
  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

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

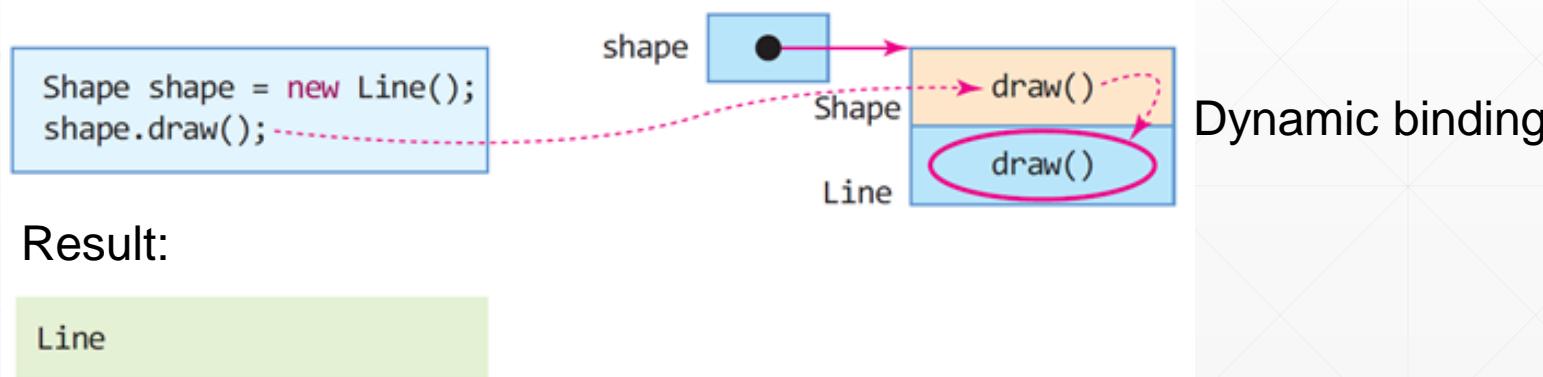
Method Overriding (cont'd)

■ 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

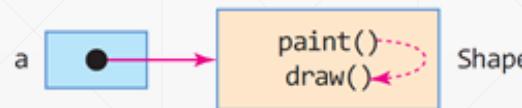
- Runtime association of method calling
- “Who should be invoked?” is determined at runtime

```
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();  
    }  
}
```

```
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:

Shape



Result:

Circle



Method Overriding (cont'd)

■ Static binding

- Compile-time association of method calling
- “Who should be invoked?” is determined at compile time (e.g, static method)

```
class Shape {  
    static void clear(){ System.out.println("Clear!"); }  
    void draw() { System.out.println("Shape"); }  
}  
  
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

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)

Computer Language



OOP 4: Abstraction



Agenda

- Abstract Class
- Interface

Abstract Class

Interface

Abstract Class: Goal

■ Abstraction

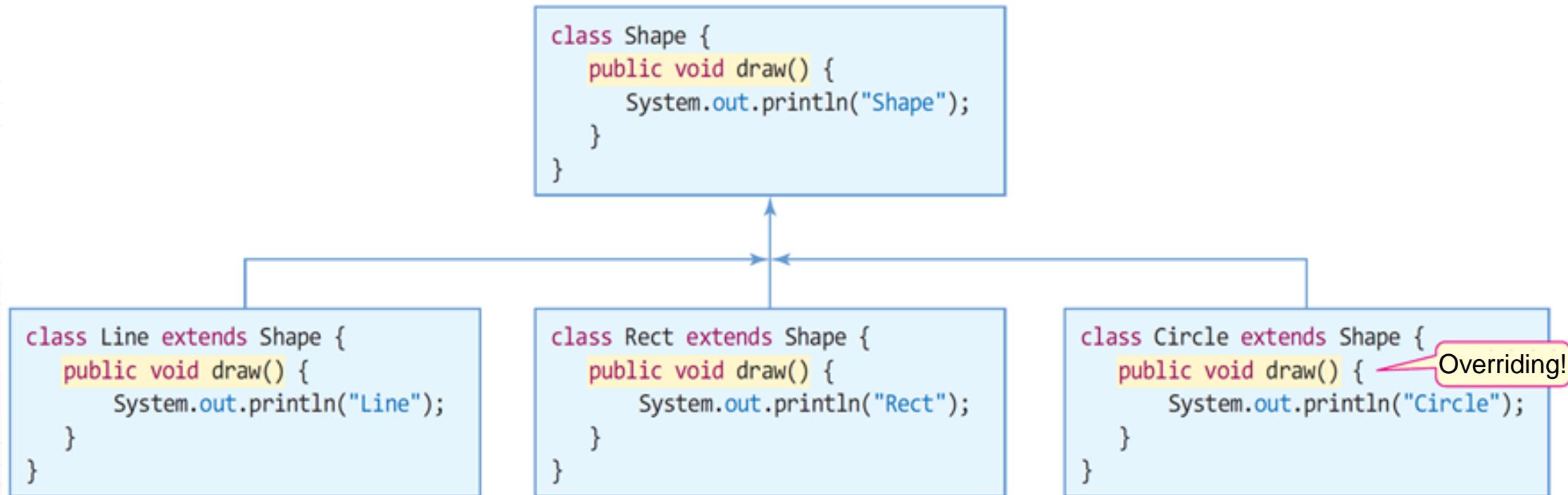
- Extraction of common features from a set of similar instances
 - Example 1) bird, insect, fish → animal (abstraction)
 - Example 2) Samsung, Hyundai, LG → company (abstraction)

■ Abstract class

- Class to define the common fields and methods of concrete classes
- Act as a parent (base) class for concrete classes

Abstract Class: Goal (cont'd)

- What's different? Compared to standard inheritance relationship?



Abstract Class: Goal (cont'd)

■ Goal of Abstract class

- Separate interface (design) from implementation!

■ Abstract class

- Define common concepts
- Declare 'abstract' methods that MUST be implemented in the subclasses
 - Abstract method does not have implementations!

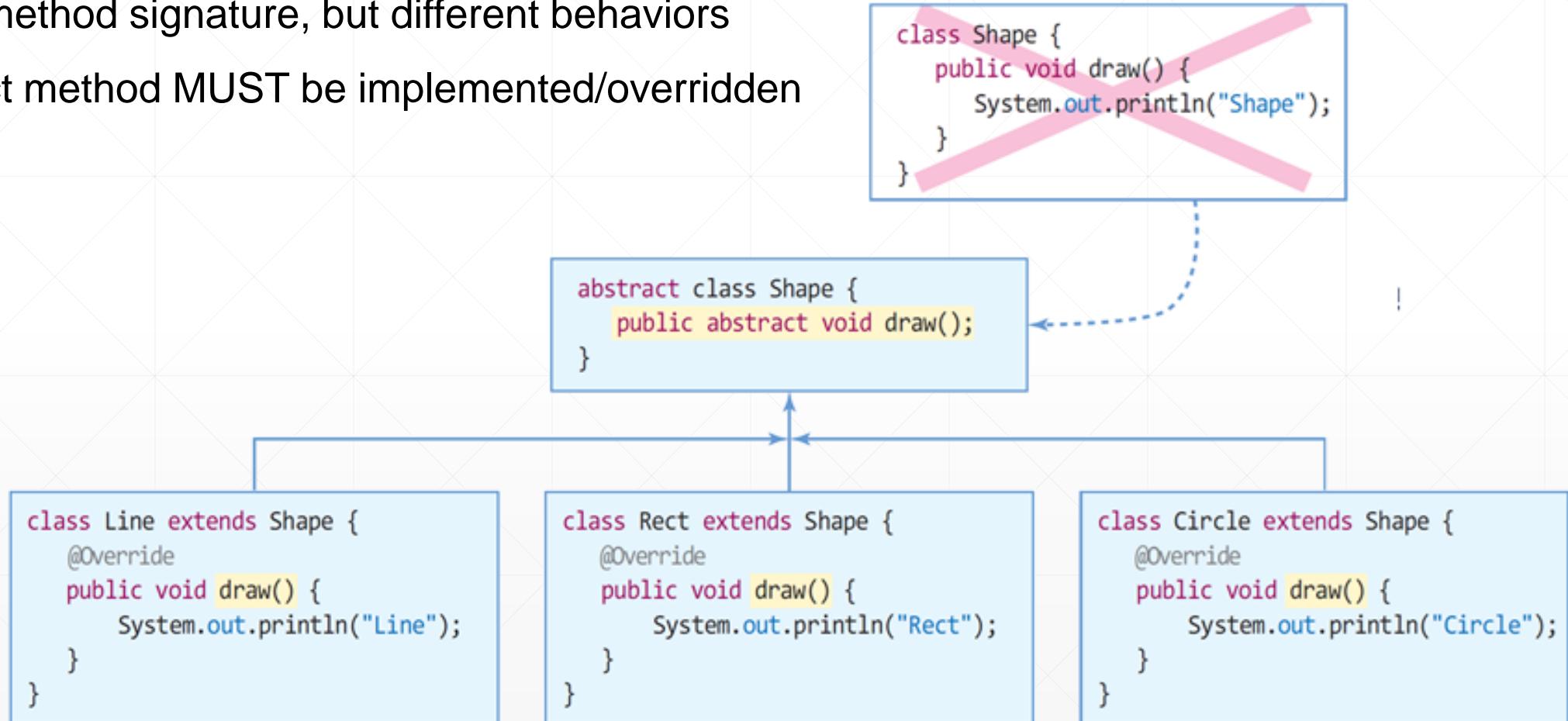
■ Concrete classes

- Implement class-specific behaviors

Abstract Class: Goal (cont'd)

■ Redefinition of superclass's method in the subclass

- Same method signature, but different behaviors
- Abstract method MUST be implemented/overridden



Abstract Class: Goal (cont'd)

- Redefinition of superclass's method in the subclass

- Same method signature, but different behaviors
 - Abstract method MUST be implemented/overridden

- 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

Abstract Class: Definition

■ Use abstract keyword!

- Abstract Class
- Abstract method
 - Defined but not implemented method
 - MUST be overridden by subclasses

■ Characteristics of abstract classes

- Cannot be instantiated by new() keyword
- May or may not include abstract methods
- If a class includes abstract methods, then the class MUST be declared abstract

Abstract Class: Definition (cont'd)

■ Characteristics of abstract classes

- May or may not include abstract methods

```
// 1. abstract class containing abstract methods
```

```
abstract class Shape { // declaration of abstract class
    public Shape() { }
    public void paint() { draw(); }
    abstract public void draw(); // declaration of abstract method
}
```

No implementation for abstract methods

```
// 2. abstract class without abstract methods
```

```
abstract class MyComponent { // declaration of abstract class
    String name;
    public void load(String name) {
        this.name = name;
    }
}
```

Abstract Class: Definition (cont'd)

■ Inheritance of abstract classes

- Abstract class inheriting another abstract class
 - Subclass cannot be instantiated

```
abstract class Shape { // abstract class
    public Shape() { }
    public void paint() { draw(); }
    abstract public void draw(); // abstract method
}
abstract class Line extends Shape { // abstract class, not implementing draw() method
    public String toString() { return "Line"; }
}
```

- Concrete class inheriting an abstract class
 - All abstract methods MUST be implemented (overriding)
 - Concrete subclass can be instantiated

Abstract Class: Usecases

■ 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);  
    }  
}
```

Abstract Class: Usecases (cont'd)

■ Example)

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

```
abstract class Payment {  
    abstract void pay(int money);  
}  
  
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!"); }  
}
```

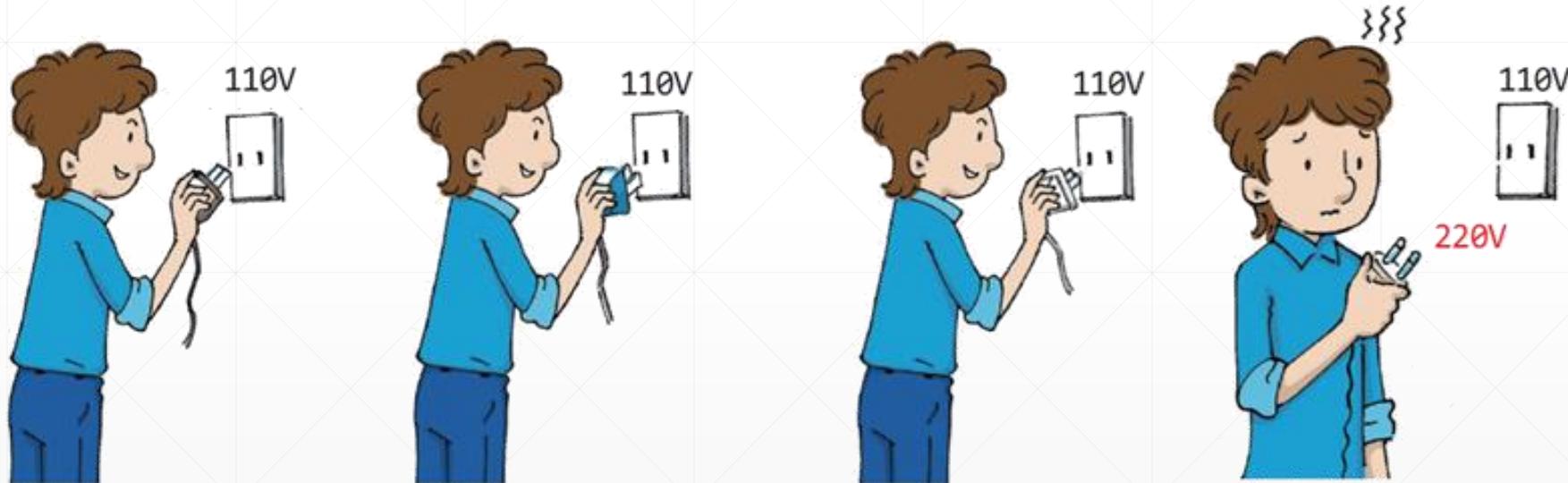


Abstract Class **Interface**

Interface: Goal

■ Interface in real life

- Define a standard for interaction between devices



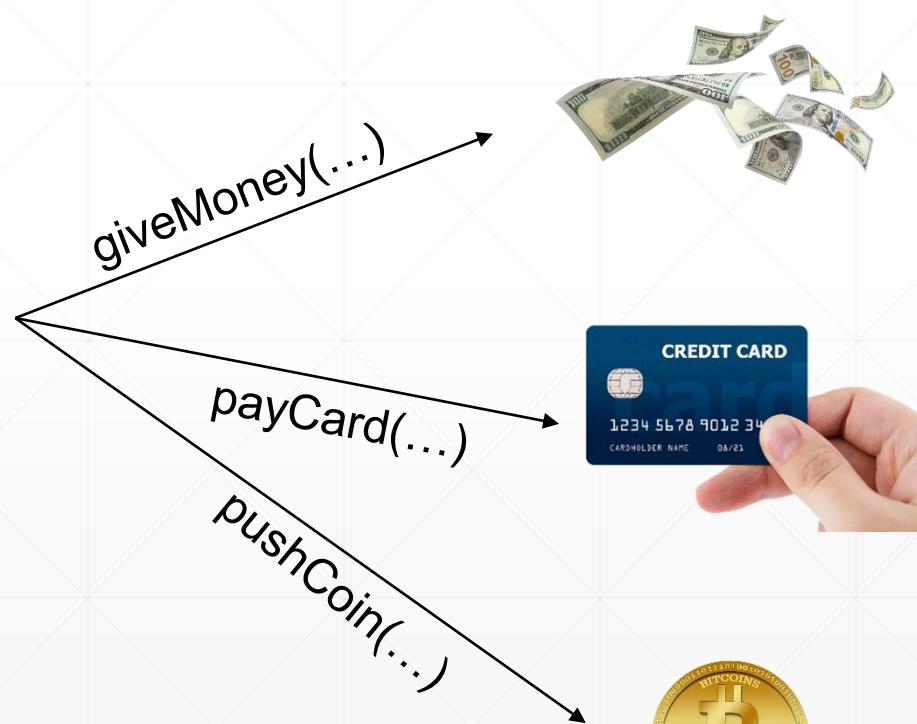
Interface: Goal (cont'd)

■ Interface in Java world

- Define a standard (contract) for interaction between class/objects



POS



Need to know all the details of each counterpart!

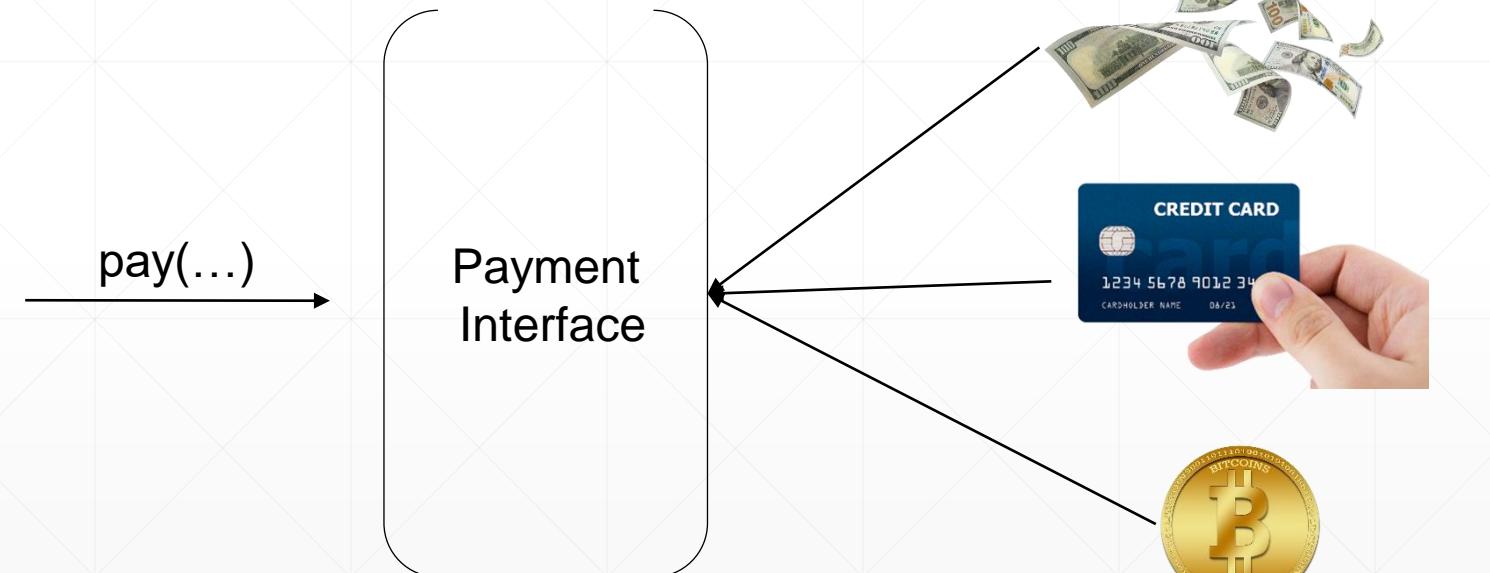
Interface: Goal (cont'd)

■ Interface in Java world

- Define a standard (contract) for interaction between class/objects
- Achieves polymorphism



POS



Classes implementing
Payment interface

Only interact with Payment interface using pay() method!

Interface: Definition

- Declaration of interface

- Use *interface* keyword!

- Declaration of interface members

- Constants
 - Abstract methods
 - Default, private, static methods
 - Variables are **NOT** allowed

Interface: Definition (cont'd)

■ Declaration of interface

- Use *interface* keyword!

```
interface PhoneInterface { // interface
    public static final int TIMEOUT = 10000; // constant
    public abstract void sendCall(); // abstract methods
    public abstract void receiveCall();
    public default void printLogo() { // default method
        System.out.println("** Phone **");
    }
}
```

■ Declaration of interface members

- Constants
- Abstract methods
- Default, private, static methods
- Variables are **NOT** allowed

Interface: Definition (cont'd)

■ Declaration of interface members

➤ Constants

- All fields defined in an interface are automatically declared as **public static final**
- Naming convention: USE CAPITAL LETTERS

```
interface PhoneInterface { //  
    public static final int TIMEOUT = 10000; //  
    public abstract void sendCall(); //  
    public abstract void receiveCall();  
    public default void printLogo() { //  
        System.out.println("** Phone **");  
    }  
}
```

➤ Abstract methods

- All methods defined in the interface are basically abstract without implementation
- ‘public abstract’ keywords can be omitted

Interface: Definition (cont'd)

■ Declaration of interface members

➤ Default methods

- Methods with implementations
- MUST be declared as “default”
- Can be overridden by subclasses or interface realizations
- Access modifier: public

➤ Private methods

- Methods with implementations
- Access modifier: private
- Only accessible by the methods inside the same interface

➤ Static methods

- Can be either public or private (default: public)

```
interface PhoneInterface { //  
    public static final int TIMEOUT = 10000; //  
    public abstract void sendCall(); //  
    public abstract void receiveCall();  
    public default void printLogo() { //  
        System.out.println("## Phone ##");  
    };  
}
```

Interface: Characteristics

■ Instantiation

- Interfaces cannot be instantiated, like abstract classes
- Cannot use new() keyword for object instantiation



```
new PhoneInterface();
```

■ Reference

- Reference variable of a certain interface type can be declared

```
PhoneInterface galaxy; OK!
```

■ Inheritance

- Can extend another interface
- Can extend multiple interfaces

Interface: Characteristics (cont'd)

■ Example of interface definition and inheritance

```
interface PhoneInterface {  
    final int TIMEOUT = 10000;  
    void sendCall();  
    void receiveCall();  
}
```

```
interface MP3Interface {  
    void play();  
    void stop();  
}
```

```
interface MobilePhoneInterface extends PhoneInterface {  
    void sendSMS(); // additional abstract methods  
    void receiveSMS(); // additional abstract methods  
}
```

Interface can extend another interface!

```
interface MusicPhoneInterface extends MobilePhoneInterface, MP3Interface {  
    void playMP3RingTone(); // additional abstract methods  
}
```

Interface can extend multiple interfaces!

Interface: Realization

■ Use 'implements' keyword to realize a certain interface

- Multiple realization is also allowed
- All abstract methods defined in the interface MUST be implemented

■ Example of interface realization

```
class SamsungPhone implements PhoneInterface {  
  
    public void sendCall() { System.out.println("RRRiiinnnngg~~"); }  
    public void receiveCall() { System.out.println("Incoming call!!!"); }  
  
    public void flash() { System.out.println("Mmmmmyyy Flash----!!"); }  
}
```

```
interface PhoneInterface {  
  
    final int TIMEOUT = 10000;  
  
    void sendCall();  
    void receiveCall();  
}
```

 → SamsungPhone's additional method

Interface: Realization (cont'd)

■ Example of interface realization (cont'd)

```
class LGPhone implements PhoneInterface {  
  
    public void sendCall() { System.out.println("Yap, my call!!"); }  
    public void receiveCall() { System.out.println("Give me a call!!"); }  
  
    public void knock() { System.out.println("knock, knock!"); }   
}  
  
public class InterfaceEx{  
    public static void main(String[] args) {  
        SamsungPhone myPhone = new SamsungPhone();  
        LGPhone yourPhone = new LGPhone();  
  
        myPhone.sendCall();  
        myPhone.receiveCall();  
        myPhone.flash();  
  
        yourPhone.sendCall();  
        yourPhone.receiveCall();  
        yourPhone.knock();  
    }  
}
```

LGPhone's additional method

Interface: Realization (cont'd)

■ Example) default and private methods

- New requirement: Every phone should be able to print Phone logo!
 - 1) adding abstract printLogo() method? → existing interface broken
 - 2) adding default printLogo() method! → method with implementation in the interface

```
interface PhoneInterface {  
    final int TIMEOUT = 10000;  
  
    void sendCall();  
  
    void receiveCall();  
  
    default void printLogo() {  
        System.out.println("** Phone **");  
    }  
}
```

```
class IPhone implements PhoneInterface {  
  
    public void sendCall() { System.out.println("Yap, my call!!"); }  
    public void receiveCall() { System.out.println("Give me a call!!"); }  
  
    public void printLogo(){  
        System.out.println("|||| PPPPHONE!");  
    }  
  
    public void watch() { System.out.println("Apple watch activated!"); }  
}
```

Overriding the default method

Interface: Realization (cont'd)

■ Example) default and private methods (cont'd)

- New requirement: Every phone should be able to print Phone logo!
 - 1) adding abstract printLogo() method? → existing interface broken
 - 2) adding default printLogo() method! → method with implementation in the interface

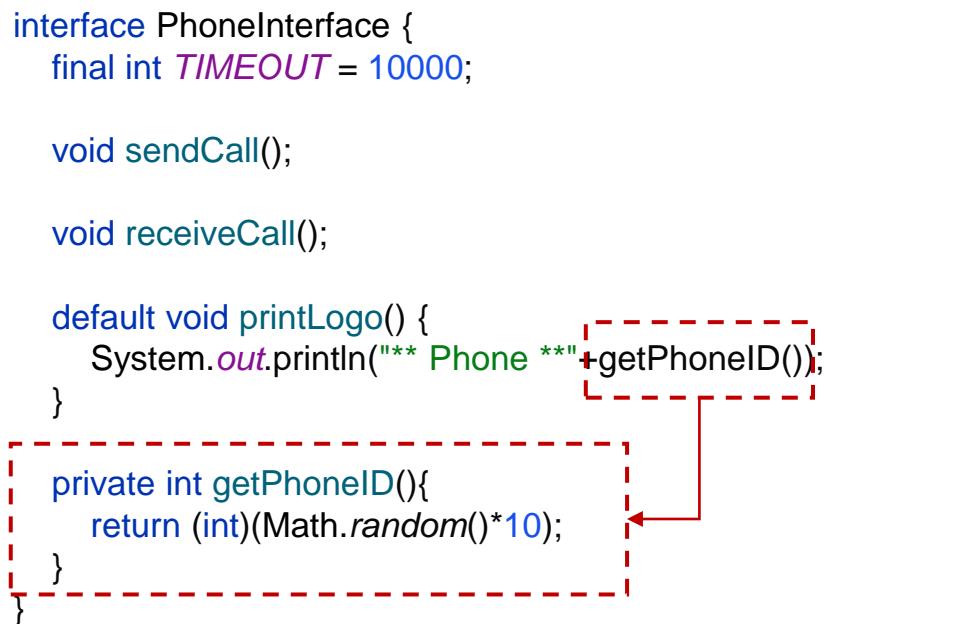
```
public class InterfaceEx{  
    public static void main(String[] args) {  
        SamsungPhone myPhone = new SamsungPhone();  
        LGPhone yourPhone = new LGPhone();  
        IPhone hisPhone = new IPhone();  
  
        myPhone.printLogo();  
        yourPhone.printLogo();  
        hisPhone.printLogo();  
    }  
}
```

Interface: Realization (cont'd)

■ Example) default and private methods (cont'd)

- New requirement: Every phone should be able to print Phone logo!
 - 1) adding abstract printLogo() method? → existing interface broken
 - 2) adding default printLogo() method! → method with implementation in the interface
 - Even we can define and invoke a private method in the interface!

```
interface PhoneInterface {  
    final int TIMEOUT = 10000;  
  
    void sendCall();  
  
    void receiveCall();  
  
    default void printLogo() {  
        System.out.println("** Phone **");  
        getPhoneID();  
    }  
  
    private int getPhoneID(){  
        return (int)(Math.random()*10);  
    }  
}
```



Interface: Realization (cont'd)

■ Example) static methods

➤ Utility method of an interface

- Static methods of an interface can be only accessed via Interface name

```
interface PhoneInterface {  
    final int TIMEOUT = 10000;  
  
    void sendCall();  
  
    void receiveCall();  
  
    default void printLogo() {  
        System.out.println("** Phone **"+getPhoneID());  
    }  
  
    private int getPhoneID(){  
        return (int)(Math.random()*10);  
    }  
  
    static int getTimeout(){  
        return TIMEOUT;  
    }  
}
```

```
public class InterfaceEx{  
    public static void main(String[] args) {  
        SamsungPhone myPhone = new SamsungPhone();  
        LGPhone yourPhone = new LGPhone();  
  
        System.out.println(PhoneInterface.getTimeout());  
        System.out.println(myPhone.getTimeout());  
    }  
}
```

```
interface MobilePhoneInterface extends PhoneInterface {  
    void sendSMS();  
    void receiveSMS();  
}
```

Interface: Realization (cont'd)

■ Realization of multiple interfaces

- A class can implement multiple interfaces

```
interface AllInterface {  
    void recognizeSpeech();  
    void synthesizeSpeech();  
}  
  
class iPhone implements MobilePhoneInterface, AllInterface { // realization of multiple interfaces  
    // realize MobilePhoneInterface  
    public void sendCall() { ... }  
    public void receiveCall() { ... }  
    public void sendSMS() { ... }  
    public void receiveSMS() { ... }  
  
    // realize AllInterface  
    public void recognizeSpeech() { ... }  
    public void synthesizeSpeech() { ... }  
  
    // can add class-specific methods  
    public int touch() { ... }  
}
```

Interface: Realization (cont'd)

■ Example) Extending abstract class + implementing multiple interfaces

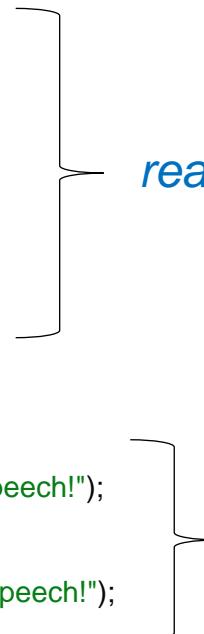
```
interface PhoneInterface {  
    final int TIMEOUT = 10000;  
    void sendCall();  
    void receiveCall();  
    default void printLogo() {  
        System.out.println("** Phone **");  
    }  
}
```

```
interface MobilePhoneInterface extends  
PhoneInterface {  
    void sendSMS();  
    void receiveSMS();  
}
```

```
interface AllInterface {  
    void recognizeSpeech();  
    void synthesizeSpeech();  
}
```

```
abstract class PDA {  
    public int calculate(int x, int y) {  
        return x + y;  
    }  
}
```

```
class AIPhone extends PDA implements MobilePhoneInterface, AllInterface { // realization of multiple interfaces  
  
    public void sendCall() {  
        System.out.println("AI Sends call!");  
    }  
    public void receiveCall() {  
        System.out.println("AI receives call!");  
    }  
    public void sendSMS() {  
        System.out.println("AI sends sms!");  
    }  
    public void receiveSMS() {  
        System.out.println("AI receives sms!");  
    }  
  
    public void recognizeSpeech() {  
        System.out.println("AI recognized your speech!");  
    }  
    public void synthesizeSpeech() {  
        System.out.println("AI synthesized your speech!");  
    }  
  
    // can add class-specific methods  
    public void touch() {  
        System.out.println("Don't touch me!");  
    }  
}
```



realize MobilePhoneInterface

realize AllInterface

Interface: Realization (cont'd)

- Example) Extending abstract class + implementing multiple interfaces (cont'd)

```
public class InterfaceEx{  
    public static void main(String[] args) {  
        AIPhone myPhone = new AIPhone();  
        myPhone.touch();  
        myPhone.sendCall();  
        myPhone.receiveSMS();  
        myPhone.synthesizeSpeech();  
        System.out.println("My Phone can calculate: 10 + 10 = "+myPhone.calculate(10,10));  
    }  
}
```

Interface: Usecases

■ Example)

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

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

Interface: Usecases (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);  
    }  
}
```

Abstract Class vs Interface

Abstract class	Interface
Abstract class does not support multiple inheritance	Interface supports multiple inheritance
Abstract class can have final, non-final, static and non-static variables	Interface has only static and final variables (constants)
abstract keyword is used to declare abstract class	interface keyword is used to declare interface
Abstract class can extend another class and implement multiple interfaces	Interface can extend another interface only
Abstract class can be extended using keyword "extends"	Interface can be implemented using keyword "implements"
Abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default
Example: <pre>public abstract class Shape{ public abstract void draw(); }</pre>	Example: <pre>public interface Drawable{ void draw(); }</pre>

Abstract Class vs Interface (cont'd)

■ Which should you use, abstract classes or interfaces?

- Consider using abstract classes if any of these statements apply to your situation:
 - You want to share code among several **closely related classes**
 - You expect that classes that extend your abstract class have **many common methods or fields**, or require access modifiers other than public (such as protected and private)
 - You want to declare non-static or non-final fields

- Consider using interfaces if any of these statements apply to your situation:
 - You expect that **unrelated classes** would implement your interface
 - You want to **specify the behavior of a particular data type**, but **not concerned about who implements its behavior**
 - You want to take advantage of multiple inheritance of type

Q&A

■ Next week

- Exception handling
- Java basic packages

Computer Language



Java Basic Packages

Agenda

- Exception Handling
- Enumeration
- Java Packages – Part I

Exception Handling

Enumeration
Java Packages

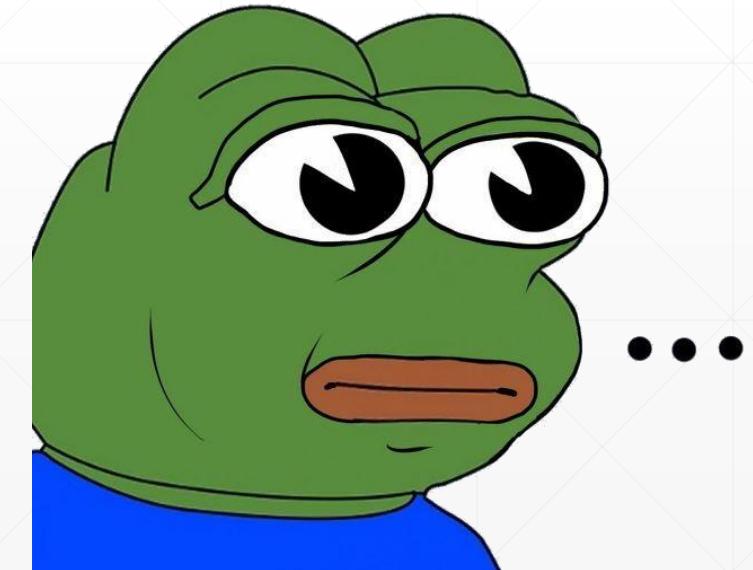
Exception

- Event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions
 - Shorthand for the phrase “exceptional event”
 - Generally related with “error”
 - Invalid manipulation of a program by the user
 - Developer’s incorrect logics
 - ...
- What happens if exception occurs?
 - Your program will be crashed
 - Before crashing, your program can handle the exceptions!

Exception (cont'd)

■ When exception occurs?

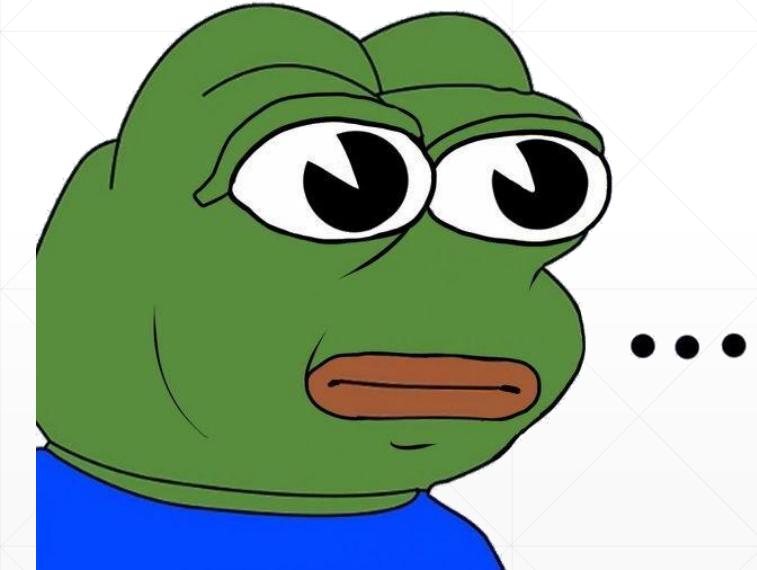
- Dividing an integer by zero
- Accessing an element of an array with an index greater than the length of the array
- Reading a file that does not exist
- Entering a string value to the position where an integer value is required.
- ...



Exception (cont'd)

■ What kind of exception occurs?

Exception class	When?
ArithmaticException	Dividing an integer by zero
NullPointerException	Referencing a null reference
ClassCastException	Casting to the invalid type
OutOfMemoryError	Not enough memory
ArrayIndexOutOfBoundsException	Accessing an invalid index of an array
IllegalArgumentException	Passing invalid arguments
IOException	IO operation failure
NumberFormatException	Invalid number conversion
InputMismatchException	Invalid use of Scanner methods

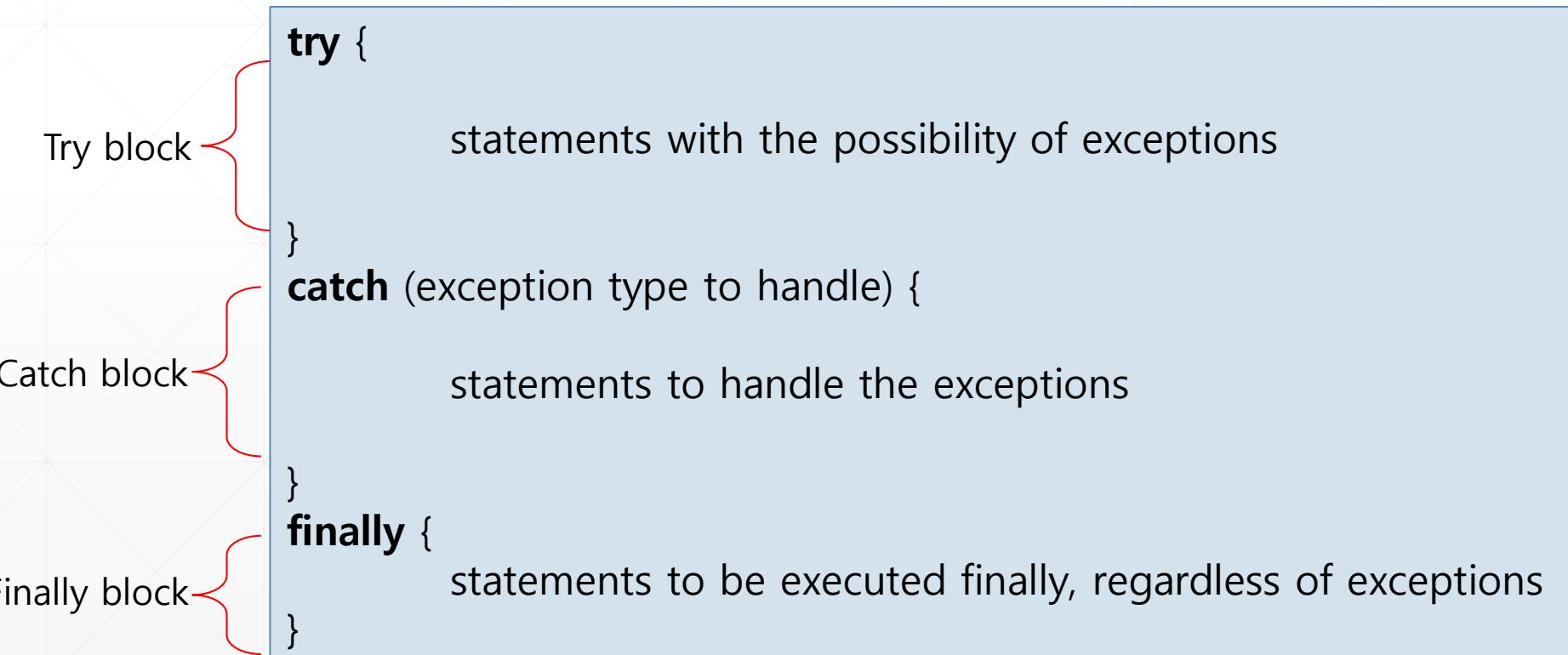


“Exception” class is a superclass
of all other specific exception classes!

Exception: Try-Catch-Finally (cont'd)

■ So, how to handle exceptions?

- Use Try-Catch(-Finally) statement!
 - Finally block can be omitted



The diagram illustrates the structure of the Try-Catch-Finally statement. It consists of three nested curly braces, each labeled with its corresponding block type: 'Try block', 'Catch block', and 'Finally block'. The code within each block is described by text to its right.

```
try {  
    statements with the possibility of exceptions  
}  
catch (exception type to handle) {  
    statements to handle the exceptions  
}  
finally {  
    statements to be executed finally, regardless of exceptions  
}
```

Try block: statements with the possibility of exceptions

Catch block: statements to handle the exceptions

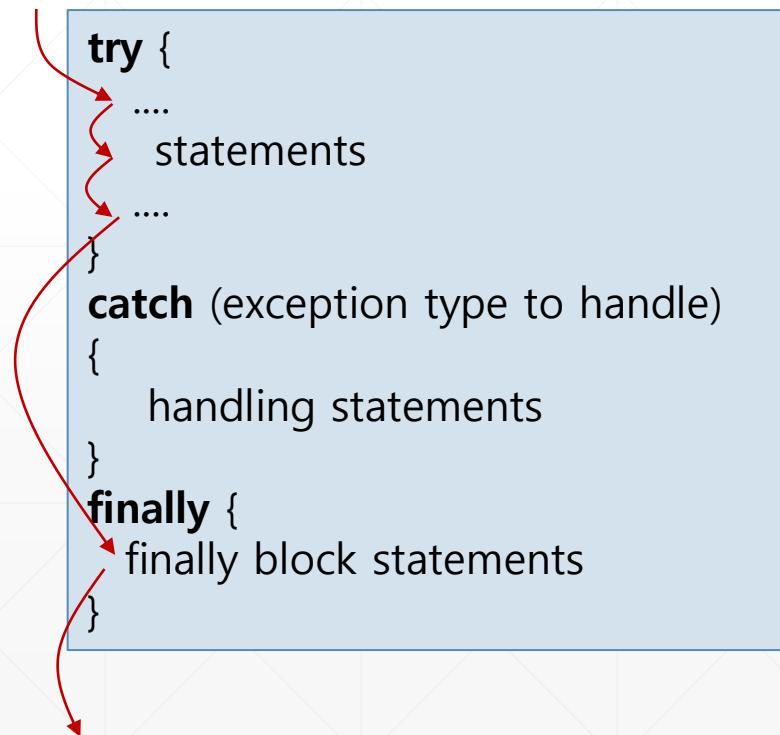
Finally block: statements to be executed finally, regardless of exceptions

Exception: Try-Catch-Finally (cont'd)

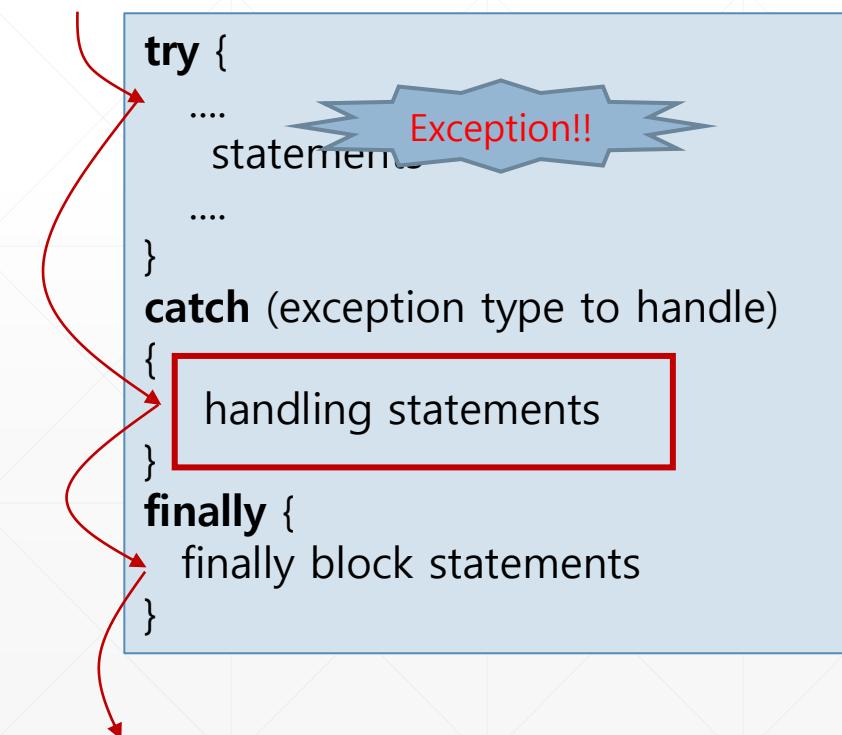
■ So, how to handle exceptions?

- Use Try-Catch(-Finally) statement!

Normal case that no exceptions occur in the try block

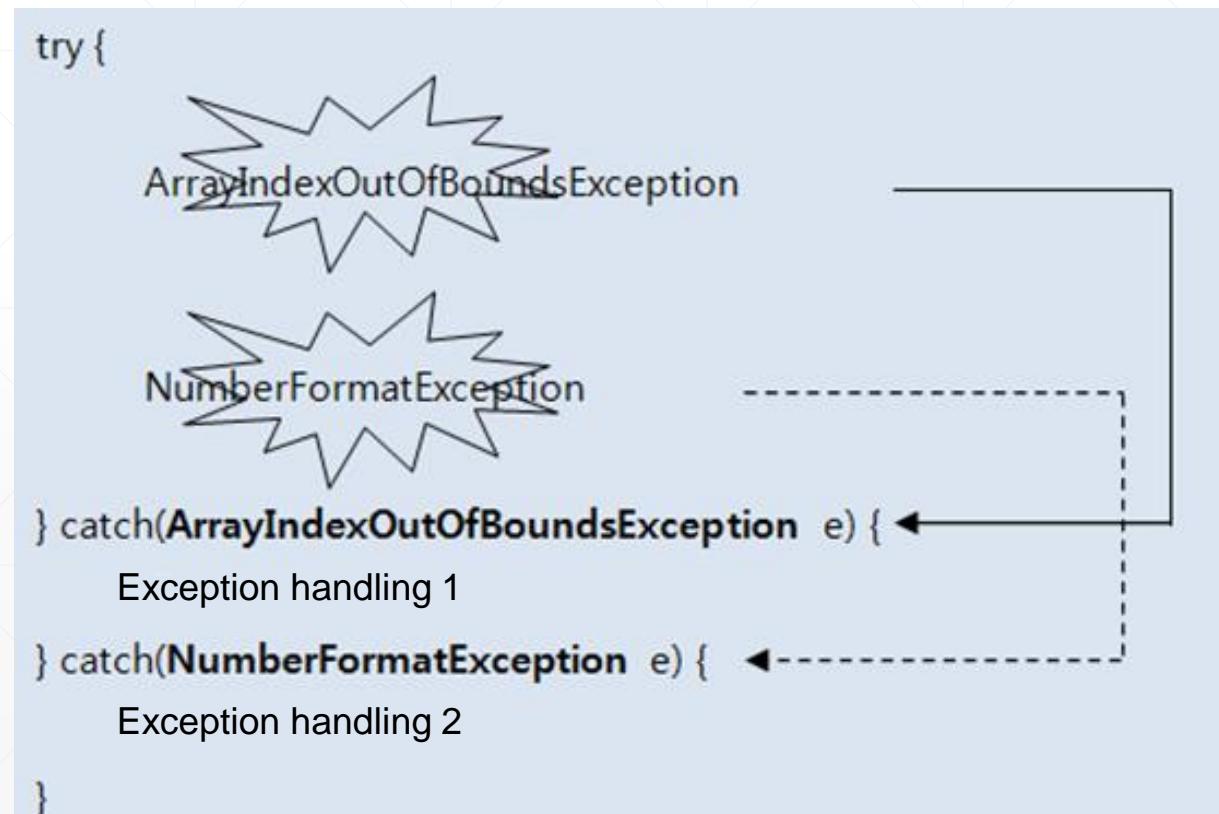


Error case that an exception occurs in the try block



Exception: Try-Catch-Finally (cont'd)

- So, how to handle exceptions?
 - Use Try-Catch(-Finally) statement!
 - Multiple catch statements are also allowed



Exception: Examples

■ ArithmeticException

```
import java.util.Scanner;

public class DivideByZero {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int dividend;
        int divisor;

        System.out.print("Input your number:");
        dividend = scanner.nextInt();
        System.out.print("Input your divisor:");
        divisor = scanner.nextInt();
        System.out.println(dividend+ " divided by " + divisor + " is " + dividend/divisor );
        scanner.close();
    }
}
```

Exception occurs
when divisor is 0

Exception: Examples (cont'd)

■ **ArrayIndexOutOfBoundsException**

```
public class ArrayException {  
    public static void main (String[] args) {  
        int[] intArray = new int[5];  
        intArray[0] = 0;  
        try {  
            for (int i=0; i<5; i++) {  
                intArray[i+1] = i+1 + intArray[i];  
                System.out.println("intArray["+i+"]"+" = "+intArray[i]);  
            }  
        }  
        catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("out of index!");  
        }  
    }  
}
```

Exception occurs
when i is 4

Exception: Examples (cont'd)

■ NumberFormatException

```
public class NumException {  
    public static void main (String[] args) {  
        String[] stringNumber = {"23", "12", "3.141592", "998"};  
        String test = null;  
        int i=0;  
        try {  
            for (i=0; i<stringNumber.length; i++) {  
                int j = Integer.parseInt(stringNumber[i]);  
                System.out.println("The value after converting to integer number is " + j);  
                if(i % 2 == 1) System.out.println(test.length());  
            }  
        }  
        catch (NumberFormatException e) {  
            System.out.println(stringNumber[i] + " cannot be converted to integer number.");  
        }  
        catch (NullPointerException e){  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

Exception occurs when
converting "3.141592"

Exception occurs when
i is an odd-number

Exception: Error Information

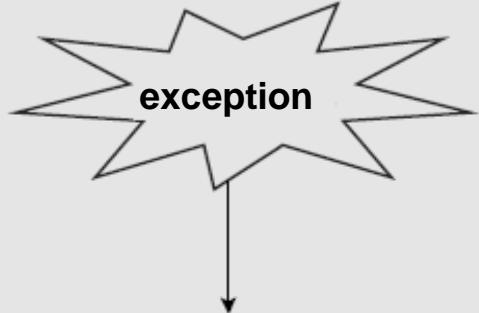
■ getMessage()

- Can take the error message for the exception
- Used in the catch block

■ printStackTrace()

- Print all the history of tracing the exception source to the console

```
try {  
    // ...  
}  
catch(Exception type e) {  
    // take the message of the exception  
    String message = e.getMessage();  
  
    // trace the path of exception  
    e.printStackTrace();  
}
```



Exception: Debugging

■ Breakpoint using IDE



The screenshot shows a Java code editor window titled "NumException.java". The code is as follows:

```
1 public class NumException {
2     public static void main (String[] args) {
3         String[] stringNumber = {"23", "12", "3.141592", "998"};
4         String test = null;
5         int i=0;
6         try {
7             for (i=0; i<stringNumber.length; i++) {
8                 int j = Integer.parseInt(stringNumber[i]);
9                 System.out.println("The value after converting to integer number is " + j);
10                //if(i % 2 == 1) System.out.println(test.length());
11            }
12        } catch (NumberFormatException e) {
13            System.out.println(stringNumber[i] + " cannot be converted to integer number.");
14        } catch (NullPointerException e){
15            System.out.println(e.getMessage());
16        }
17    }
18 }
```

A red box highlights line 8, which contains the statement `int j = Integer.parseInt(stringNumber[i]);`. A red dot at the start of this line indicates a breakpoint has been set. The code editor interface includes a toolbar at the top, a status bar with a warning icon and the number 1, and a vertical scrollbar on the right.

Exception: Debugging (cont'd)

■ Breakpoint using IDE

The screenshot shows a Java code editor and a debugger interface. The code is as follows:

```
6     try {
7         for (i=0; i<stringNumber.length; i++) {
8             int j = Integer.parseInt(stringNumber[i]);  stringNumber: ["23", "12", "3.141592", "998"]
9             System.out.println("The value after converting to integer number is " + j);
10            //if(i % 2 == 1) System.out.println(test.length());
11        }
12    }
13    catch (NumberFormatException e) {
14        System.out.println(stringNumber[i] + " cannot be converted to integer number.");
15    }
16    catch (NullPointerException e){
17        System.out.println(e.getMessage());
18    }
19
20
21
22 }
```

The line 8 is highlighted with a blue background, and there is a red breakpoint icon on the left margin next to line 8. The variable `stringNumber` is shown in the Variables panel with its elements: `["23", "12", "3.141592", "998"]`. The variable `i` is shown with a value of 0.

Below the code editor is a debugger window titled "Debug: NumException". The "Debugger" tab is selected. The "Frames" panel shows the current frame: "main:8, NumException". The "Variables" panel displays the current status of variables, which matches the information shown in the code editor's sidebar.

A red arrow points from the text "Current status of variables" to the "Variables" panel in the debugger window.

Current status of variables



Exception Handling **Enumeration** Java Packages

Enumeration: Review

■ 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 }
```

Enumeration: Review (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;
```

Enumeration: Review (cont'd)

■ Example)

Hello.java

```
public class Hello {  
    public static void main(String[] args) {  
  
        Weekday myDay = Weekday.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;  
        }  
    }  
}
```

Weekday.java

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

Enumeration: Class-usage

■ Enumeration is actually a kind of Java class

- So, we can use various features of Java class!
- Fields, constructor, methods, etc

■ Constructor

- Access modifier: private
 - We cannot create an enum object explicitly
- Called for each constant definition
 - At the time of enum class loading!
- ‘this’ keyword refers to the created constant itself

```
public enum Weekday {  
    MONDAY,  
    TUESDAY,  
    WEDNESDAY,  
    THURSDAY,  
    FRIDAY,  
    SATURDAY,  
    SUNDAY;  
  
    Weekday(){  
        System.out.println(this + " was called!");  
    }  
}
```

Enumeration: Class-usage (cont'd)

■ Custom fields

- We can assign some custom values to the constant
 - Syntax: CONSTANT(...values...)
- Field definition for a custom value required
- Constructor with arguments required
 - To assign custom values to the field

```
public enum Weekday {  
    MONDAY("NO"),  
    TUESDAY("No"),  
    WEDNESDAY("no"),  
    THURSDAY("yes"),  
    FRIDAY("YES!"),  
    SATURDAY("YEAH~!"),  
    SUNDAY("SAD...");  
  
    public String text; // field for custom message  
  
    Weekday(String text){  
        System.out.println(this + " was called!");  
        this.text = text; // assigning custom messages  
    }  
}
```

- If enum class has fields/methods, then the constant definitions must end with a semicolon

Enumeration: Class-usage (cont'd)

■ Custom fields

```
public class Hello {  
    public static void main(String[] args) {  
  
        Weekday myDay = Weekday.SATURDAY;  
  
        switch (myDay) {  
            case MONDAY:  
            case FRIDAY:  
            case SATURDAY:  
            case SUNDAY:  
                System.out.println(myDay.name() + " is " + myDay.text);  
                break;  
            default:  
                System.out.println("Midweek days are so-so.");  
                break;  
        }  
    }  
}
```

```
public enum Weekday {  
    MONDAY("NO"),  
    TUESDAY("No"),  
    WEDNESDAY("no"),  
    THURSDAY("yes"),  
    FRIDAY("YES!"),  
    SATURDAY("YEAH~!"),  
    SUNDAY("SAD...");  
  
    public String text; // field for custom message  
  
    Weekday(String text){  
        System.out.println(this + " was called!");  
        this.text = text; // assigning custom messages  
    }  
}
```

Enumeration: Class-usage (cont'd)

■ Method

- Getter for a custom value
 - Encapsulation purpose

```
public enum Weekday {  
    MONDAY("NO"),  
    TUESDAY("No"),  
    WEDNESDAY("no"),  
    THURSDAY("yes"),  
    FRIDAY("YES!"),  
    SATURDAY("YEAH~!"),  
    SUNDAY("SAD...");  
  
    private String text;  
    Weekday(String text){  
        System.out.println(this + " was called!");  
        this.text = text;  
    }  
  
    public String getText() {  
        return text;  
    }  
}
```

```
switch (myDay) {  
    case MONDAY:  
    case FRIDAY:  
    case SATURDAY:  
    case SUNDAY:  
        System.out.println(myDay.name() + " is " + myDay.getText());  
        break;  
    default:  
        System.out.println("Midweek days are so-so.");  
        break;  
}
```

Enumeration: Class-usage (cont'd)

■ Method

➤ Enum class methods

- `name()`: returns the defined name of an enum constant in string form
- `values()`: returns an array of enum type containing all the enum constants
- `valueOf()`: takes a string and returns an enum constant having the same string name

Enumeration: Class-usage

■ Example)

- Two custom values assigned
- Fields and getters added
- Constructor changed

```
for(Weekday w: Weekday.values()) {  
    System.out.print(w.getCode());  
    System.out.print(w.getText());  
    System.out.println(w.name());  
}  
  
Weekday someday = Weekday.valueOf("THURSDAY");  
System.out.println(someday.getText());
```

```
public enum Weekday {  
    MONDAY(0,"NO"),  
    TUESDAY(1,"No"),  
    WEDNESDAY(2,"no"),  
    THURSDAY(3,"yes"),  
    FRIDAY(4,"YES!"),  
    SATURDAY(5,"YEAH~!"),  
    SUNDAY(6,"SAD...");
```

```
private int code;  
private String text;
```

```
Weekday(int code, String text){  
    System.out.println(this + " was called!");  
    this.code = code;  
    this.text = text;  
}
```

```
public String getText() {  
    return text;  
}
```

```
public int getCode() {  
    return code;  
}
```

Exception Handling
Enumeration

Java Packages

Java API Packages

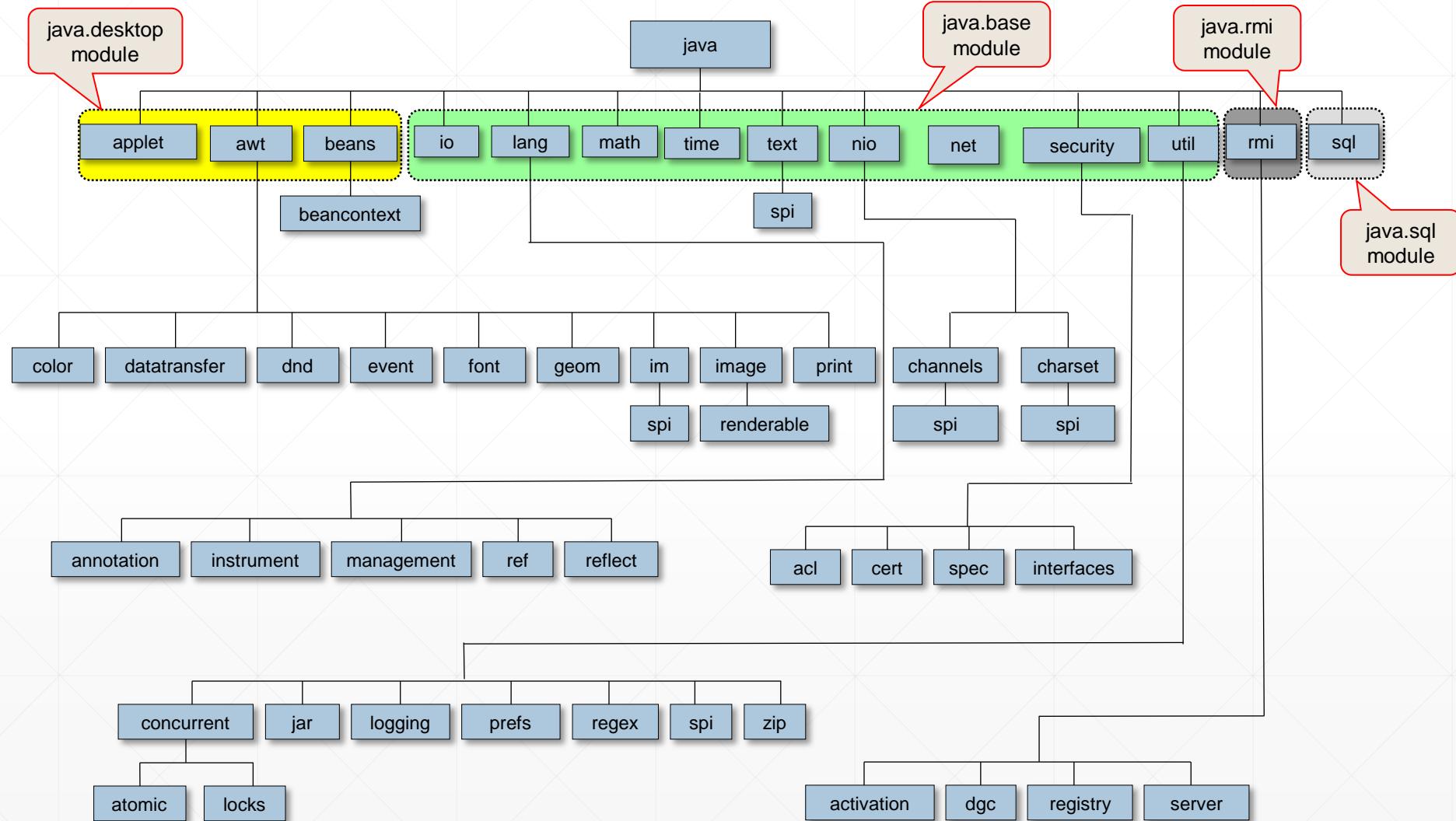
■ Java API (Application Programming Interface)

- Java's built-in software library to develop java programs
- A collection of frequently used classes and interfaces

■ Java API document (reference/specification)

- Document on how to use APIs
- Online reference
 - <https://docs.oracle.com/en/java/javase/11/docs/api/index.html>

Java API Packages (cont'd)



Java API Packages (cont'd)

■ java.lang

- Java language package
 - Basic classes and interfaces for developing Java programs, including String, Math, etc.

■ java.util

- Utility package
 - Various utility classes and interfaces including Date, Time, Vector, HashMap, etc.

■ java.io

- IO classes and interfaces for interacting with keyboard, monitor, printer, disk, etc.

■ java.awt

- Classes and Interfaces for Java GUI programming

■ javax.swing

- Swing package for Java GUI programming

■ ...

Object class

■ Root class of Java

- All classes implicitly, automatically inherit Object class
- All classes can use the methods of Object class

■ Methods

Method	Description
<code>boolean equals(Object obj)</code>	Returns true if this object is the same as obj
<code>Class getClass()</code>	Returns the runtime class of this object
<code>int hashCode()</code>	Returns a hashCode value for this object
<code>String toString()</code>	Returns a string representation of this object

- ... and more!

Object class (cont'd)

- Example) Get the class name, hashCode, string representation of Object

```
class Point {  
    private int x, y;  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}  
  
public class ObjectPropertyEx {  
    public static void print(Object obj) {  
        System.out.println(obj.getClass().getName()); // class name  
        System.out.println(obj.hashCode()); // hashCode  
        System.out.println(obj.toString()); // string representation  
        System.out.println(obj); // object itself  
    }  
    public static void main(String [] args) {  
        Point p = new Point(2,3);  
        print(p);  
    }  
}
```

Object class (cont'd)

■ `toString()` method

- Returns a string representation of an object
- `toString()` method implementation of Object class

```
public String toString() {  
    return getClass().getName() + "@" + Integer.toHexString(hashCode());  
}
```

- Automatically invoked when a string manipulation with an object reference is required
- Overriding `toString()` for each class
 - Can return a class-specific string representation

Object class (cont'd)

■ `toString()` method

```
class Point {  
    private int x, y;  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public String toString() {  
        return "Point(" + x + "," + y + ")";  
    }  
}  
  
public class ToStringEx {  
    public static void main(String [] args) {  
        Point p = new Point(2,3);  
        System.out.println(p.toString());  
        System.out.println(p);  
        System.out.println("Info: "+p);  
    }  
}
```

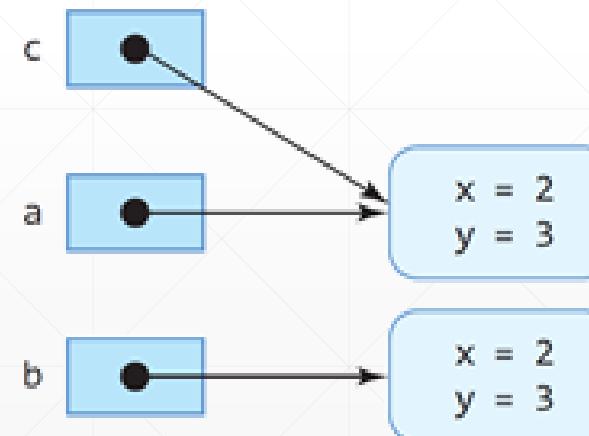
Object class (cont'd)

■ equals() method

- Returns true if this object is the same as obj
- Object's equals() method basically works like == operator
 - == operator: returns true if two operands point the same address (for reference type)

```
class Point {  
    private int x, y;  
    public Point(int x, int y) {  
        this.x = x; this.y = y;  
    }  
}
```

```
Point a = new Point(2,3);  
Point b = new Point(2,3);  
Point c = a;  
if(a == b) // false  
    System.out.println("a==b");  
if(a == c) // true  
    System.out.println("a==c");
```



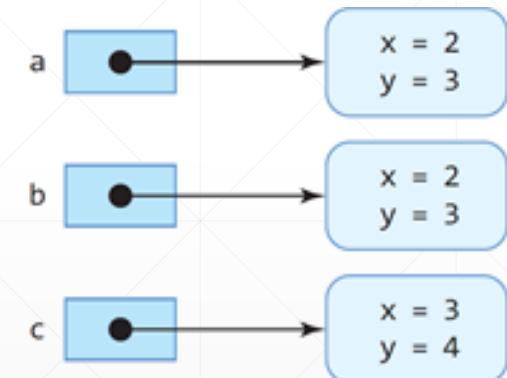
Object class (cont'd)

■ equals() method

- Can be overridden in the class to return true if this object is the same as obj in terms of contents, rather than address

```
class Point {  
    private int x, y;  
    public Point(int x, int y) {  
        this.x = x; this.y = y;  
    }  
  
    public boolean equals(Object obj) {  
        Point p = (Point)obj;  
        if(x == p.x && y == p.y)  
            return true;  
        else return false;  
    }  
}
```

```
Point a = new Point(2,3);  
Point b = new Point(2,3);  
Point c = new Point(3,4);  
  
if(a == b) // false  
    System.out.println("a==b");  
if(a.equals(b)) // true  
    System.out.println("a is equal to b");  
if(a.equals(c)) // false  
    System.out.println("a is equal to c");
```



Object class (cont'd)

■ equals() method: Example)

- Implement Rect class with width and height fields. If the areas of two Rect objects are same, then equals() method of Rect should return true.

```
class Rect {  
    private int width;  
    private int height;  
    public Rect(int width, int height) {  
        this.width = width;  
        this.height = height;  
    }  
    public boolean equals(Object obj) {  
        Rect p = (Rect)obj;  
        if (width*height == p.width*p.height)  
            return true;  
        else  
            return false;  
    }  
}
```

```
public class EqualsEx {  
    public static void main(String[] args) {  
        Rect a = new Rect(2,3);  
        Rect b = new Rect(3,2);  
        Rect c = new Rect(3,4);  
        if(a.equals(b))  
            System.out.println("a is equal to b");  
        if(a.equals(c))  
            System.out.println("a is equal to c");  
        if(b.equals(c))  
            System.out.println("b is equal to c");  
    }  
}
```

Wrapper Class

- Dedicated class for primitive datatypes
 - “Wrap” the primitive data type into an object of that class

Primitive	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
int	Integer
float	Float
double	Double
long	Long
short	Short

Wrapper Class (cont'd)

■ Methods

- All wrapper classes have similar methods

■ Main methods of Integer class

Modifier and Type	Method	Description
boolean	equals(Object obj)	Compares this object to the specified object
byte	byteValue()	Returns the value of this Integer as a byte after a narrowing primitive conversion
double	doubleValue()	Returns the value of this Integer as a double after a widening primitive conversion
float	floatValue()	Returns the value of this Integer as a float after a widening primitive conversion
int	intValue()	Returns the value of this Integer as an int
long	longValue()	Returns the value of this Integer as a long after a widening primitive conversion
short	shortValue()	Returns the value of this Integer as a short after a narrowing primitive conversion
static int	sum(int a, int b)	Adds two integers together as per the + operator
static int	max(int a, int b)	Returns the greater of two int values as if by calling Math.max.
static int	min(int a, int b)	Returns the smaller of two int values as if by calling Math.min.

Wrapper Class (cont'd)

■ Main methods of Integer class (cont'd)

Modifier and Type	Method	Description
static int	parseInt(String s)	Parses the string argument as a signed decimal integer
static int	parseInt(String s, int radix)	Parses the string argument as a signed integer in the radix specified by the second argument
static String	toBinaryString(int i)	Returns a string representation of the integer argument as an unsigned integer in base 2
static String	toHexString(int i)	Returns a string representation of the integer argument as an unsigned integer in base 16
static String	toOctalString(int i)	Returns a string representation of the integer argument as an unsigned integer in base 8
String	toString()	Returns a String object representing this Integer's value
static String	toString(int i)	Returns a String object representing the specified integer
static String	toString(int i, int radix)	Returns a string representation of the first argument in the radix specified by the second argument
static Integer	valueOf(int i)	Returns an Integer instance representing the specified int value
static Integer	valueOf(String s)	Returns an Integer object holding the value of the specified String
static Integer	valueOf(String s, int radix)	Returns an Integer object holding the value extracted from the specified String when parsed with the radix given by the second argument

Wrapper Class (cont'd)

■ Instantiation of wrapper class

static Integer valueOf (int i)	Returns an Integer instance representing the specified int value
static Integer valueOf (String s)	Returns an Integer object holding the value of the specified String
static Integer valueOf (String s, int radix)	Returns an Integer object holding the value extracted from the specified String when parsed with the radix given by the second argument

```
Integer i = Integer.valueOf(10);
Character c = Character.valueOf('c');
Double f = Double.valueOf(3.14);
Boolean b = Boolean.valueOf(true);
```

```
Integer l = Integer.valueOf("10");
Double d = Double.valueOf("3.14");
Boolean b = Boolean.valueOf("false");
```

int	intValue()	Returns the value of this Integer as an int
-----	-------------------	---

```
Integer i = Integer.valueOf(10);
int ii = i.intValue(); // ii = 10

Character c = Character.valueOf('c' );
char cc = c.charValue(); // cc = 'c'
```

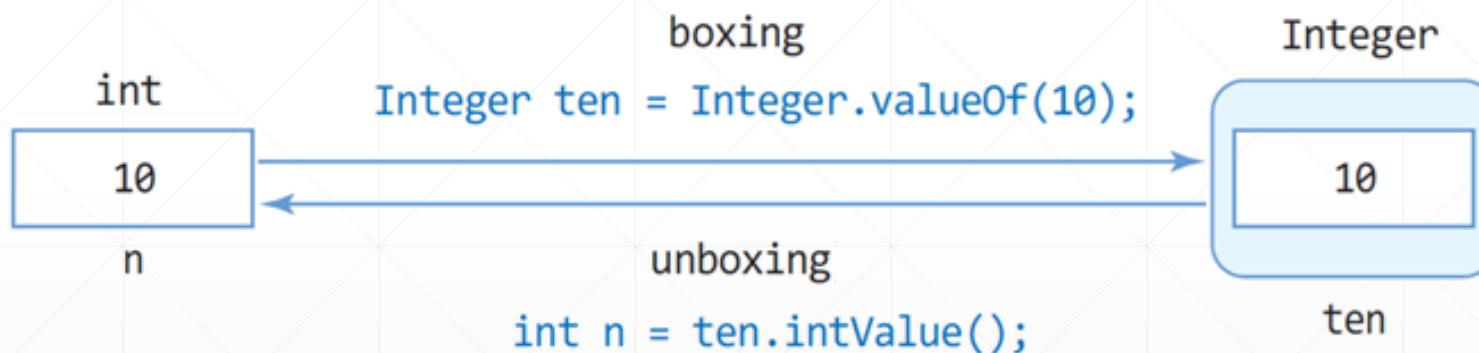
```
Double f = Double.valueOf(3.14);
double dd = d.doubleValue(); // dd = 3.14

Boolean b = Boolean.valueOf(true);
boolean bb = b.booleanValue(); // bb = true
```

Wrapper Class (cont'd)

■ Boxing & Unboxing

- Boxing: conversion from primitive types to wrapper classes
- Unboxing: conversion from wrapper classes to primitive types



- Automatic boxing & unboxing

```
Integer i = 10; // auto boxing (Integer.valueOf(10))
int ival = i; // auto unboxing (i.intValue())
System.out.println(ival);
```

Wrapper Class (cont'd)

■ String \leftrightarrow primitive types

static int	parseInt(String s)	Parses the string argument as a signed decimal integer
static int	parseInt(String s, int radix)	Parses the string argument as a signed integer in the radix specified by the second argument

```
int i = Integer.parseInt("123");           // i = 123
boolean b = Boolean.parseBoolean("true");   // b = true
double f = Double.parseDouble("3.14" );    // d = 3.14
```

String	toString()	Returns a String object representing this Integer's value
static String	toString(int i)	Returns a String object representing the specified integer

```
String s1 = Integer.toString(123);        // from integer 123 to string "123"
String s2 = Integer.toHexString(123);      // from integer 123 to HexString "7b"
String s3 = Double.toString(3.14);         // from double 3.14 to string "3.14"
String s4 = Character.toString('a');       // from character 'a' to string "a"
String s5 = Boolean.toString(true);        // from boolean true to string "true"
```

Wrapper Class (cont'd)

■ When to use?

- Utility class for primitive types
- Generic syntax
- ...

Q&A

■ Next week

- Generic & Collection

Computer Language



String APIs

Agenda

- String APIs

String

■ String class

- java.lang.String
- String representation

```
// generation of a string object using string literals  
String str1 = "abcd";  
  
// generation of a string object using String class  
char data[] = {'a', 'b', 'c', 'd'};  
String str2 = new String(data);  
String str3 = new String("abcd");
```

■ String constructor

constructor	description
String()	Create an empty string object
String(char[] value)	Create a String object with text values in char[]
String(String original)	Create a String object with the given String value
String(StringBuffer buffer)	Create a String object with text values in StringBuffer

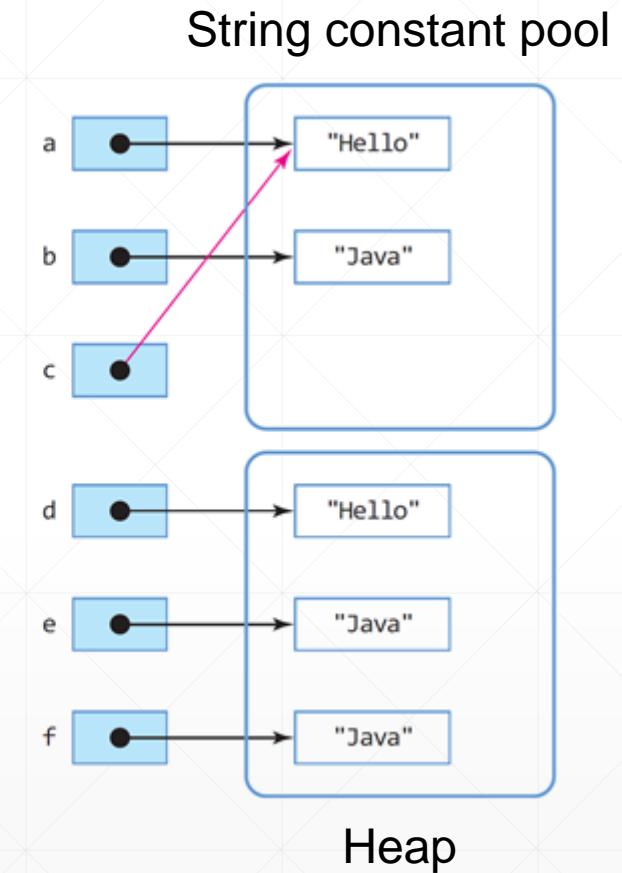
String: Literal vs new String()

■ How to create a String?

➤ Literal (String str = "Hello";)

- String by literal is **sharable** (String constant pool)
- Managed by JVM

```
String a = "Hello";
String b = "Java";
String c = "Hello";
String d = new String("Hello");
String e = new String("Java");
String f = new String("Java");
```



➤ String object (String str = new String("Hello");)

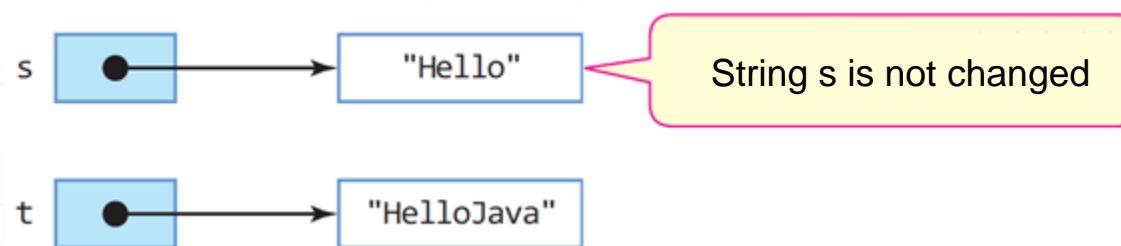
- A **new object is created** in the Heap memory area

String: Characteristics

■ Immutable

- String object **CANNOT** be modified

```
String s = new String("Hello");
String t = s.concat("Java"); // Concatenation of two strings. Returns a NEW string
```



■ Comparison

- String should be compared using `equals()` method
- `==` operator compares two objects based on **their address**

String: Methods

Method	Description
<code>char charAt(int index)</code>	Returns the char value at the specified index.
<code>int codePointAt(int index)</code>	Returns the character (Unicode code point) at the specified index.
<code>int compareTo(String anotherString)</code>	Compares two strings lexicographically.
<code>String concat(String str)</code>	Concatenates the specified string to the end of this string.
<code>boolean contains(CharSequence s)</code>	Returns true if and only if this string contains the specified sequence of char values.
<code>int length()</code>	Returns the length of this string.
<code>String replace(CharSequence target, CharSequence replacement)</code>	Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
<code>String[] split(String regex)</code>	Splits this string around matches of the given <u>regular expression</u> .
<code>String subString(int beginIndex)</code>	Returns a string that is a substring of this string.
<code>String toLowerCase()</code>	Converts all of the characters in this String to lower case using the rules of the default locale.
<code>String toUpperCase()</code>	Converts all of the characters in this String to upper case using the rules of the default locale.
<code>String trim()</code>	Returns a string whose value is this string, with all leading and trailing space removed.

String: Comparison

■ int compareTo(String anotherString)

➤ Returns:

- 0, if the argument string is equal to this string
- A value less than 0 if this string is lexicographically less than the string argument
- A value greater than 0 if this string is lexicographically greater than the string argument

```
String java= "Java";
String cpp = "C++";
int res = java.compareTo(cpp);
if(res == 0)
    System.out.println("the same");
else if(res <0)
    System.out.println(java + " < " + cpp);
else
    System.out.println(java + " > " + cpp);
```

"Java" is lexicographically greater
than "C++"

Java > C++

String: Concatenation

■ Concatenation using + operator

- In case where operands include a string or an object
 - Object is converted to String using `toString()` method and then concatenated
 - Primitive types are converted to String and then concatenated

```
System.out.print("abcd" + 1 + true + 3.13e-2 + 'E'+ "fgh" );
// abcd1true0.0313Efgh
String a = "Java";
String b = "Final Exam ";
System.out.println(a+b+"score:"+100);
// Java Final Exam score:100
```

■ Concatenation using `concat(String str)`

- Returns a new string where the string values are concatenated

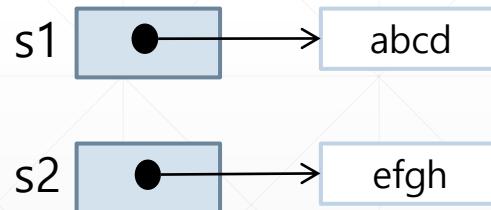
```
System.out.println("I like ".concat("Java"));
String str = "But, I love ";
System.out.println(str.concat("Python!"));
```

String: Concatenation (cont'd)

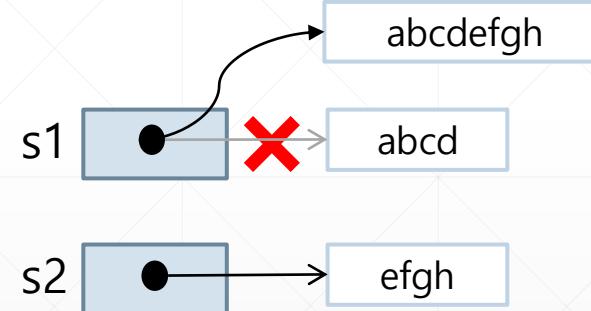
■ Concatenation using concat(String str)

- Returns a new string where the string values are concatenated

```
String s1 = "abcd";  
String s2 = "efgh";
```



```
s1 = s1.concat(s2);
```



A new string object returned by s1.concat(s2)

String: Accessing Values

■ Accessing a character in the string

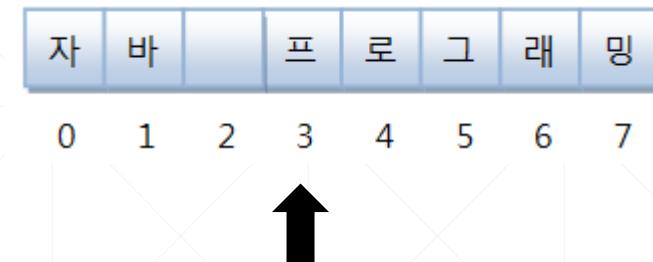
➤ Char charAt(int index)

- Returns the char value at the specified index
- An index ranges from 0 to length() - 1

```
String a = "class";
char c = a.charAt(2); // c = 'a'
```

```
// to count the number of 's' in "class"
int count = 0;
String a = "class";
for(int i=0; i<a.length(); i++) { // a.length() = 5
    if(a.charAt(i) == 's')
        count++;
}
System.out.println(count); // 2
```

```
String subject = "자바 프로그래밍";
char charValue = subject.charAt(3);
```



String: Accessing Values (cont'd)

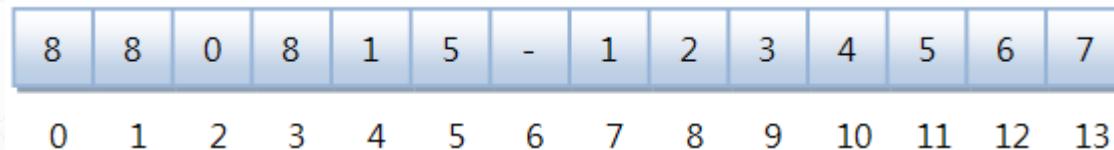
■ Extracting a sub-string from the string

➤ String substring(int beginIndex)

- Returns a substring that begins with the character at the specified index and extends to the end of this string

➤ String substring(int beginIndex, int endIndex)

- Returns a substring that begins at the specified beginIndex and extends to the character at index endIndex - 1



```
String ssn = "880815-1234567";
String firstNum = ssn.substring(0, 6);
String secondNum = ssn.substring(7);
```

String: Replace

■ Replaces the original text with the new text

- String replace(CharSequence target, CharSequence replacement)
 - Replaces each substring of this string that matches the *target* sequence with the specified *replacement* sequence
 - The replacement proceeds from the beginning of the string to the end
 - E.g., replacing "aa" with "b" in the string "aaa" will result in "ba" rather than "ab"
- Example)

```
String java = "Java is the best";
System.out.println(java.replace("best","worst"));
String comLang = java.replace("Java","Nothing");
System.out.println(comLang);
```

String: Split

- Splits this string around matches of the given regular expression

- String[] split(String regex)

- Example)

- String “boo:and:foo”

Regex	Result
:	{ "boo", "and", "foo" }
o	{ "b", "", ":and:f" }

```
String lang ="Java,C++,C#,Python,Javascript";
String[] langs = lang.split(",");
System.out.println(langs.length);
for(String l : langs) System.out.println(l);
```

More about regular expression
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java.util.regex.Pattern.html#sum](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Pattern.html#sum)

String: Trim & Lower/Upper Case

- Removes all leading and trailing spaces (tab, enter, space)

- String trim()

```
String a = "      abcd def      ";
String b = "      xyz₩t";
String c = a.trim(); // c = "abcd def". Space inside the string is not removed
String d = b.trim(); // d = "xyz". Spaces and '₩t' removed
```

- Case conversion

- String toLowerCase()/toUpperCase()

```
String original = "Java Programming";
String lowerCase = original.toLowerCase();
String upperCase = original.toUpperCase();
```

String: Example

- Removes all leading and trailing spaces (tab, enter, space)

```
public class StringEx {  
    public static void main(String[] args) {  
        String a = new String(" C#");  
        String b = new String(",C++ ");  
  
        System.out.println(a + "'s length is" + a.length()); //  
        System.out.println(a.contains("#")); // if String a contains '#'?  
  
        a = a.concat(b); // string concatenation  
        System.out.println(a);  
  
        a = a.trim(); // string trimming  
        System.out.println(a);  
  
        a = a.replace("C#", "Java"); // string replacement  
        System.out.println(a);  
  
        String s[] = a.split(","); // string split  
        for (int i=0; i<s.length; i++)  
            System.out.println("split string is" + i + ": " + s[i]);  
  
        a = a.substring(5); // accessing a substring  
        System.out.println(a);  
  
        char c = a.charAt(2); // accessing a character  
        System.out.println(c);  
    }  
}
```

StringTokenizer

■ Alternative class to split a string value into tokens using delimiters

- Delimiter: character to separate tokens
- Token: split substring

■ Constructor

Constructor	Description
<code>StringTokenizer(String str)</code>	Constructs a string tokenizer for the specified string. The tokenizer uses the default delimiter set, which is " \t\n\r\f"
<code>StringTokenizer(String str, String delim)</code>	Constructs a string tokenizer for the specified string. The characters in the delim argument are the delimiters for separating tokens.
<code>StringTokenizer(String str, String delim, boolean returnDelims)</code>	Constructs a string tokenizer for the specified string. All characters in the delim argument are the delimiters for separating tokens. If the returnDelims flag is true, then the delimiter characters are also returned as tokens.

StringTokenizer (cont'd)

[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java.util.StringTokenizer.html#method.summary](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/StringTokenizer.html#method.summary)

■ Methods

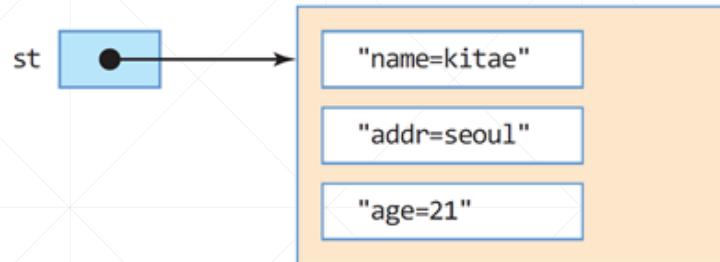
Method	Description
<code>int countTokens()</code>	Calculates the number of times that this tokenizer's <code>nextToken()</code> method can be called
<code>boolean hasMoreTokens()</code>	Tests if there are more tokens available from this tokenizer's string
<code>String nextToken()</code>	Returns the next token from this string tokenizer

StringTokenizer (cont'd)

■ Example)

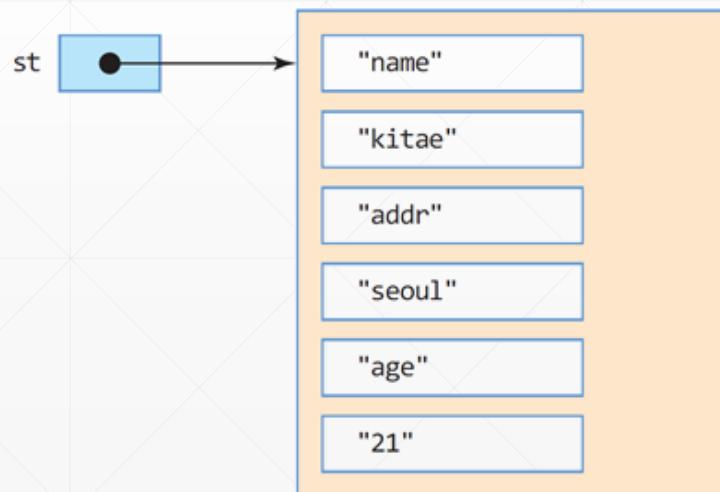
```
String query = "name=kitae&addr=seoul&age=21";  
StringTokenizer st = new StringTokenizer(query, "&");
```

Delimiter: '&'



```
StringTokenizer st = new StringTokenizer(query, "&=");
```

Delimiter: '&', '='



StringTokenizer (cont'd)

■ Example)

```
import java.util.StringTokenizer;

public class StringEx {
    public static void main(String[] args) {
        String lang ="Java,C++,C#,Python,Javascript";
        StringTokenizer st = new StringTokenizer(lang, ",");
        System.out.println("We have "+st.countTokens()+" tokens!");
        while(st.hasMoreTokens()) System.out.println(st.nextToken());
    }
}
```

StringBuffer

■ Mutable sequence of characters (String)

- String values **can be** modified

■ Constructor

Constructor	Description
<code>StringBuffer()</code>	Constructs a string buffer with no characters in it and an initial capacity of 16 characters
<code>StringBuffer(charSequence seq)</code>	Constructs a string buffer that contains the same characters as the specified CharSequence
<code>StringBuffer(int capacity)</code>	Constructs a string buffer with no characters in it and the specified initial capacity
<code>StringBuffer(String str)</code>	Constructs a string buffer initialized to the contents of the specified string

StringBuffer (cont'd)

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/StringBuffer.html#method.summary>

■ Methods

Method	Description
<code>StringBuffer append(String str)</code>	Appends the specified string to this character sequence
<code>StringBuffer append(StringBuffer sb)</code>	Appends the specified StringBuffer to this sequence
<code>int capacity()</code>	Returns the current capacity
<code>StringBuffer delete(int start, int end)</code>	Removes the characters in a substring of this sequence
<code>StringBuffer insert(int offset, String str)</code>	Inserts the string into this character sequence
<code>StringBuffer replace(int start, int end, String str)</code>	Replaces the characters in a substring of this sequence with characters in the specified String
<code>StringBuffer reverse()</code>	Causes this character sequence to be replaced by the reverse of the sequence
<code>void setLength(int newLength)</code>	Sets the length of the character sequence

StringBuffer (cont'd)

■ Example)

```
StringBuffer sb = new StringBuffer("a");
```

```
sb.append(" pencil");
```

```
sb.insert(2, "nice ");
```

```
sb.replace(2, 6, "bad");
```

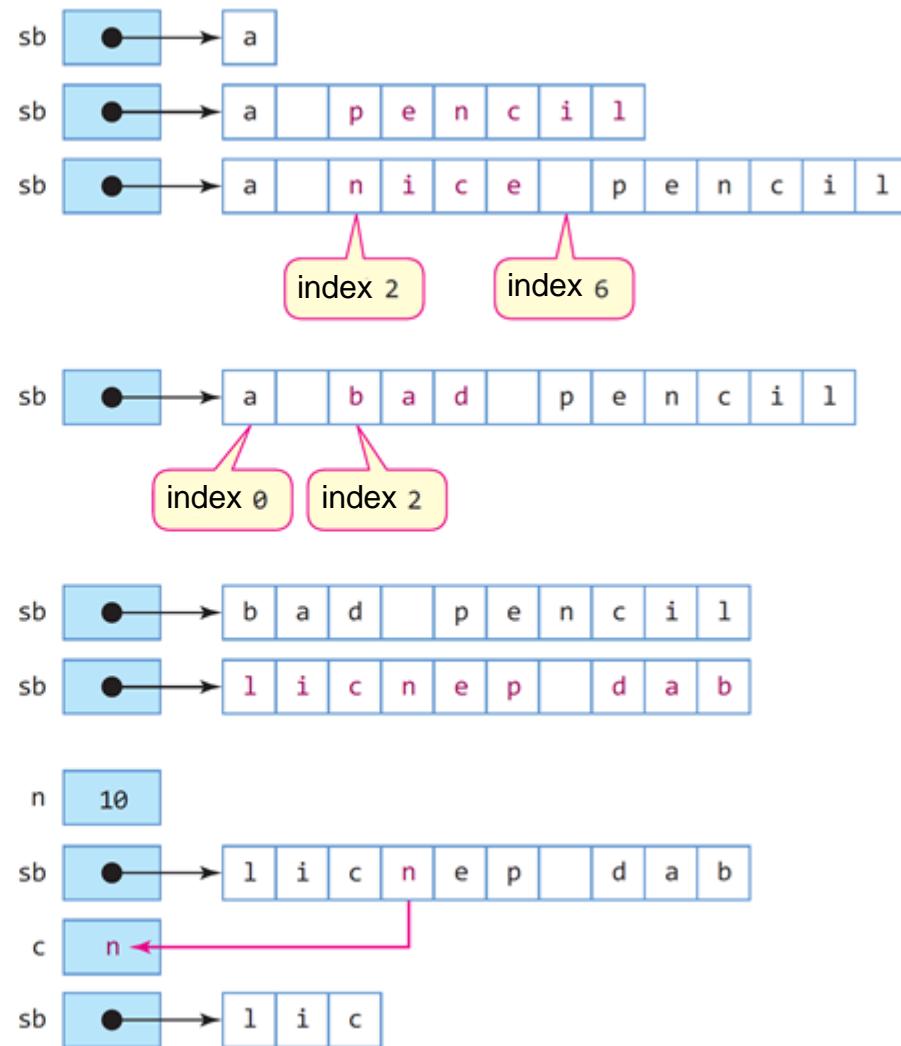
```
sb.delete(0, 2);
```

```
sb.reverse();
```

```
int n = sb.length();
```

```
char c = sb.charAt(3);
```

```
sb.setLength(3);
```



StringBuffer (cont'd)

■ Example)

```
public class StringBufferEx {  
    public static void main(String[] args) {  
        StringBuffer sb = new StringBuffer("This");  
  
        sb.append(" is pencil"); // appending a string  
        System.out.println(sb);  
  
        sb.insert(7, " my"); // inserting "my"  
        System.out.println(sb);  
  
        sb.replace(8, 10, "your"); // replacing "my" with "your"  
        System.out.println(sb);  
  
        sb.delete(8, 13); // deleting "your "  
        System.out.println(sb);  
  
        sb.setLength(4); // setting a new length  
        System.out.println(sb);  
    }  
}
```

More APIs

- Math
- Calendar
- Date
- ...

Computer Language



Generic & Collections

Agenda

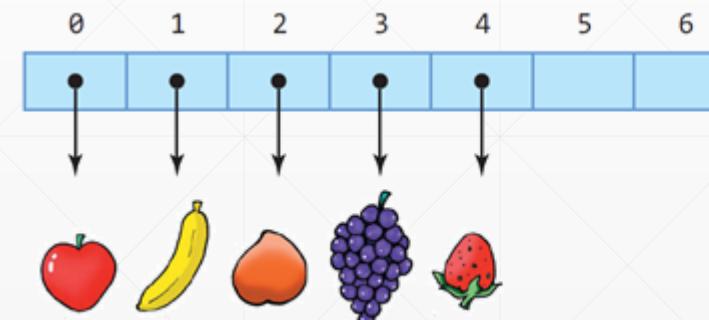
- Generic & Collection

- Collections

- Vector
 - ArrayList
 - HashMap

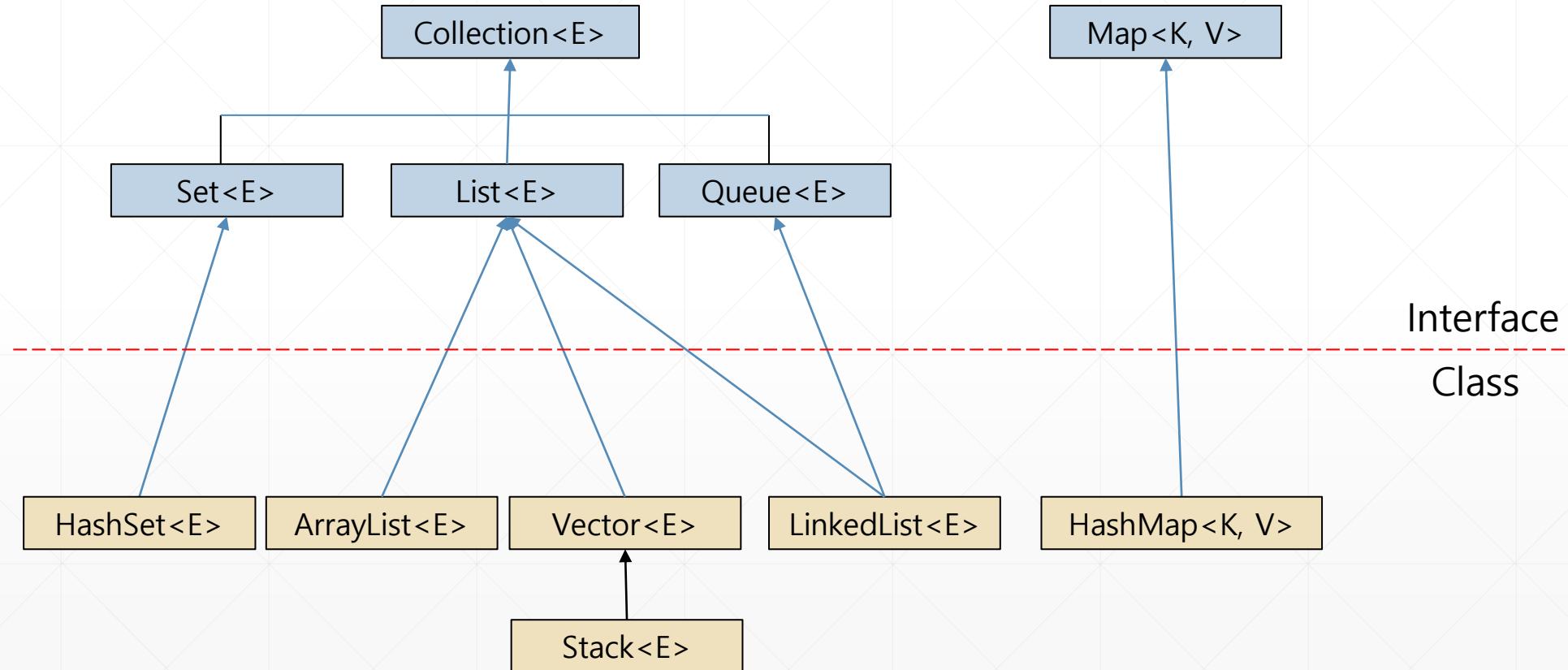
Collection

- Storage of elements
 - Container of elements
 - Dynamically update the length based on the number of elements
 - Automatically update the position of elements according to the result of insert/delete operations
- Can overcome the limitation of a fixed length array
- Ease the Insertion, deletion, and search operations for various objects



Collection (cont'd)

■ Interface/class hierarchy



Collection & Generic

■ Collections are implemented based on Generics

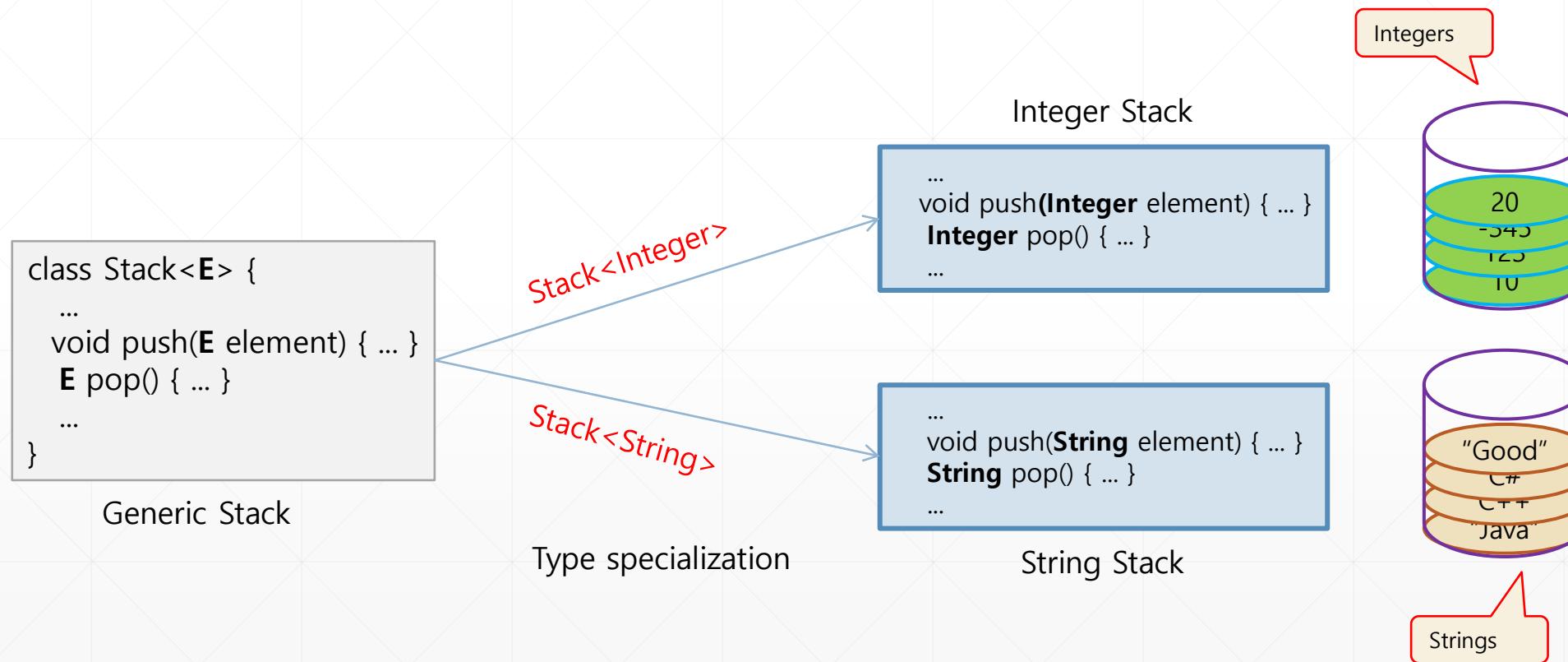
- Only objects can be elements of a collection
- Primitive types cannot be used

■ Generic

- Enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods
- Type parameters provide a way to re-use the same code with different inputs

Generic

- Provides a way to handle various types using generalized type parameters



Generic (cont'd)

■ Stack example

OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
ALL CLASSES
SEARCH: Search X

Module `java.base`

Package `java.util`

Class `Stack<E>`

`java.lang.Object`
`java.util.AbstractCollection<E>`
`java.util.AbstractList<E>`
`java.util.Vector<E>`
`java.util.Stack<E>`

All Implemented Interfaces:

`Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess`

```
public class Stack<E>
extends Vector<E>
```

The `Stack` class represents a last-in-first-out (LIFO) stack of objects. It extends class `Vector` with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top.

When a stack is first created, it contains no items.

A more complete and consistent set of LIFO stack operations is provided by the `Deque` interface and its implementations, which should be used in preference to this class. For example:

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

Since:

1.0

Generic (cont'd)

■ Defining a generic class/interface

```
public class Box {  
    private Object object;  
  
    public void set(Object object) { this.object = object; }  
    public Object get() { return object; }  
}
```

Non-generic Box class

```
public class Box<T> {  
    // T stands for "Type"  
    private T t;  
  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```

Generic version of Box class

Generic (cont'd)

■ Type parameter naming convention

- By convention, type parameter names are single, uppercase letters
 - E - Element (used extensively by the Java Collections Framework)
 - K - Key
 - N - Number
 - T - Type
 - V - Value
 - S,U,V etc. - 2nd, 3rd, 4th types

Generic: Specialization

■ Defining a generic class/interface

- Object instantiation of a generic class using specific type
- Object instantiation of a generic class using primitive type is impossible

```
public class Box<T> {  
    // T stands for "Type"  
    private T t;  
  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```

```
Box<String> s = new Box<String>(); // Setting String for generic type T  
s.set("hello");  
System.out.println(s.get()); // "hello"
```

```
Box<Integer> n = new Box<Integer>(); // Setting Integer for generic type T  
n.set(5);  
System.out.println(n.get()); // 5
```

Generic: Specialization (cont'd)

■ Defining a generic class/interface

- What happens after specialization?

```
public class MyClass<T> {  
    T val;  
    void set(T a) {  
        val = a;  
    }  
    T get() {  
        return val;  
    }  
}
```



T to String

```
public class MyClass<String> {  
    String val;  
    void set(String a) {  
        val = a;  
    }  
    String get() {  
        return val;  
    }  
}
```

Generic: Specialization (cont'd)

■ Before Java 7

```
Box<Integer> v = new Box<Integer>();
```

■ After Java 7

```
Box<Integer> v = new Box<>();
```

- Type inference feature of compiler
- Can skip type parameters in <> (diamond) as long as the compiler can determine the type arguments from the context

Generic (cont'd)

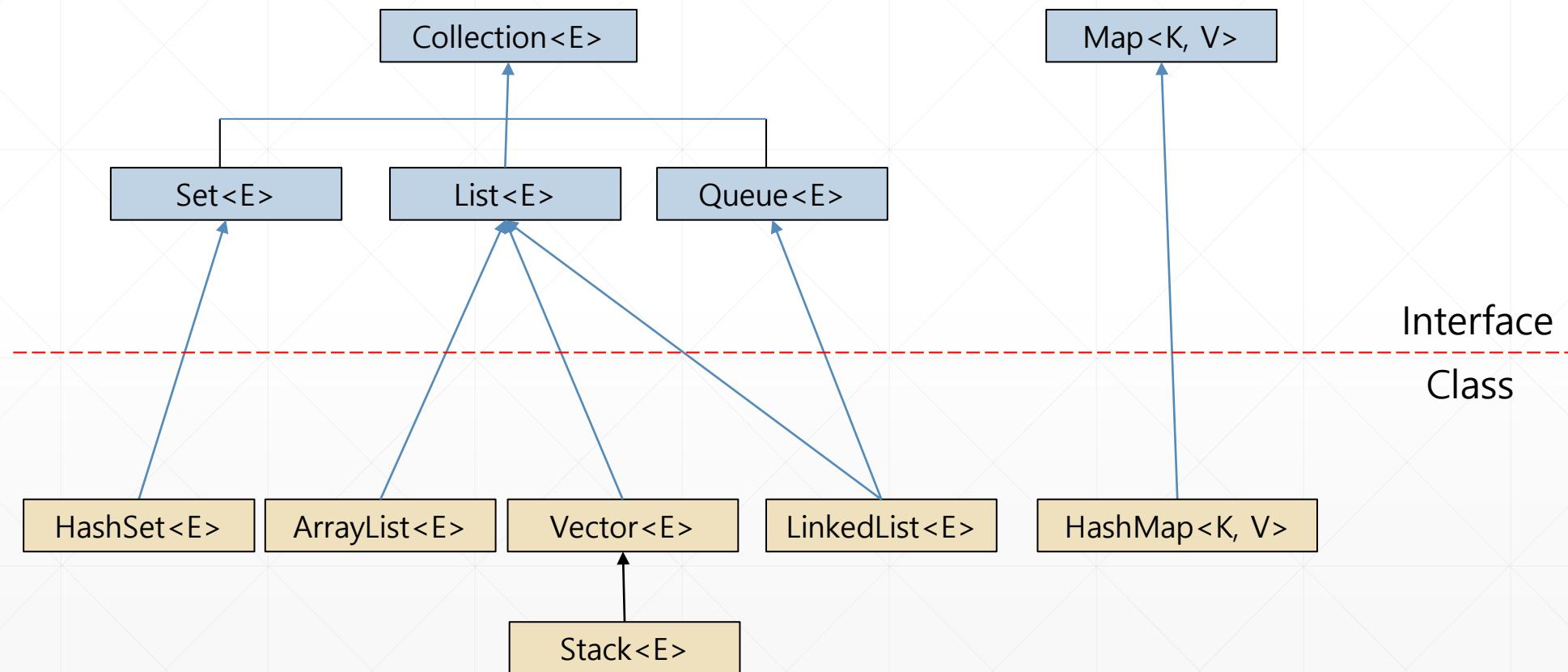
■ Defining a generic class/interface with multiple type parameters

```
class Box<K, V> {  
    private K k;  
    private V v;  
  
    public void set(K k, V v) {  
        this.k = k;  
        this.v = v;  
    }  
  
    public K getKey() {  
        return k;  
    }  
  
    public V getValue() {  
        return v;  
    }  
}
```

```
public class BoxEx{  
    public static void main(String[] args) {  
        Box<String, Integer> myBox = new Box<>();  
        myBox.set("hey",5);  
        System.out.println(myBox.getKey());  
        System.out.println(myBox.getValue());  
  
        Box<Double, Double> dBox = new Box<>();  
        dBox.set(3.14, 3.14);  
        System.out.println(dBox.getKey());  
        System.out.println(dBox.getValue());  
    }  
}
```

Collections

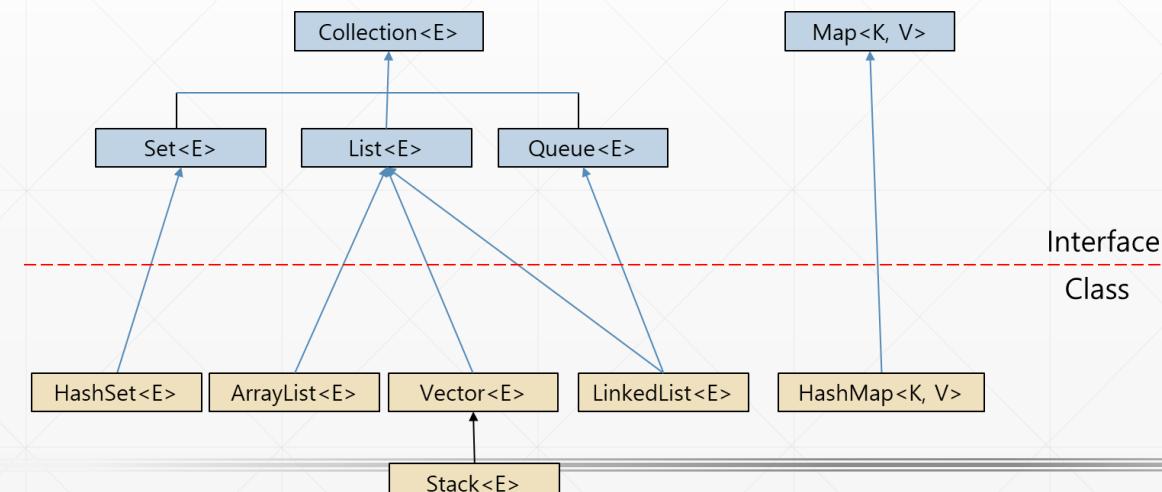
■ Interface/class hierarchy



Vector<E>

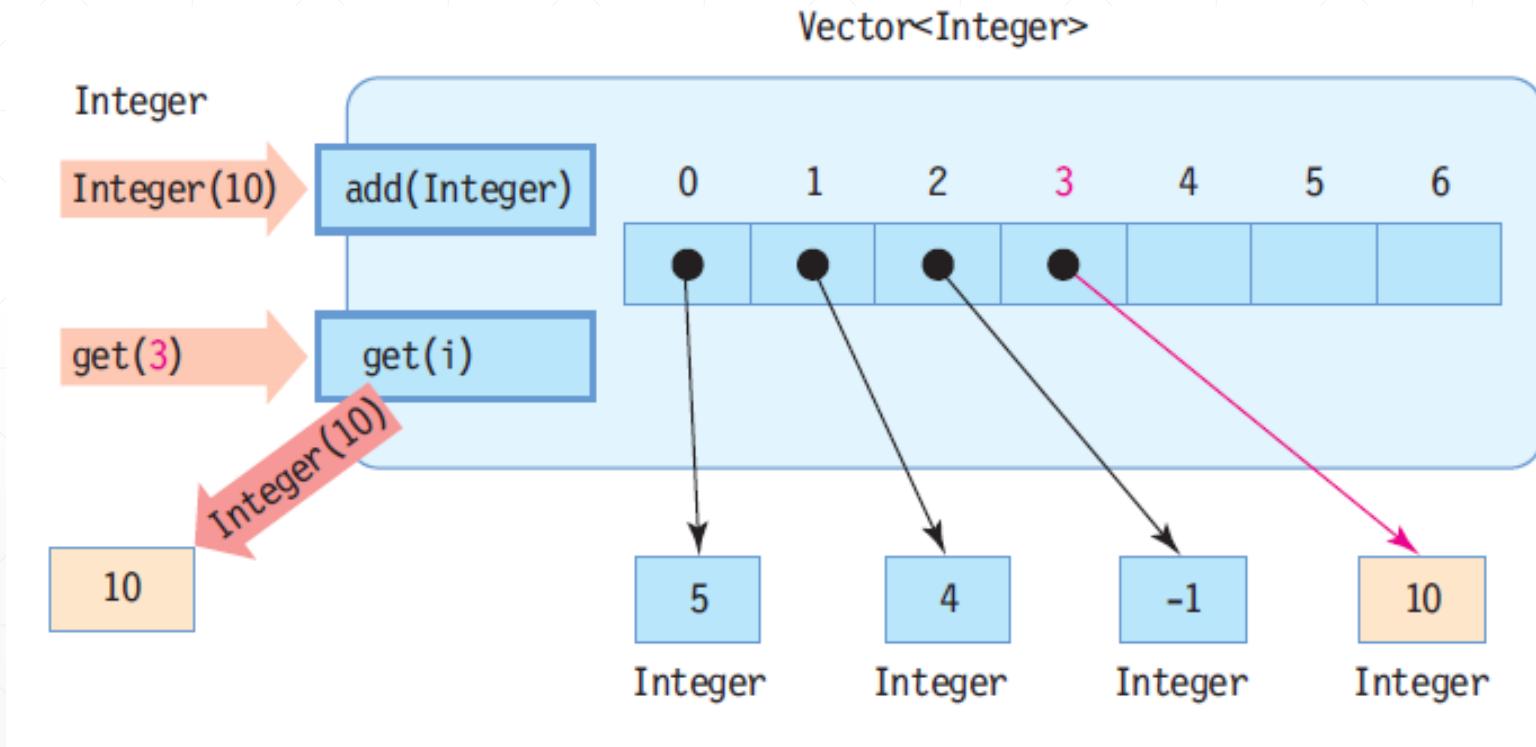
■ Characteristics

- java.util.Vector
 - Can be specialized for <E>
- Container class to insert, delete, search for multiple objects
 - Overcome the limitation of the fixed length of an array
 - Length is dynamically updated when overflow occurs
- Vector can contain:
 - Object, null
 - Primitive types after boxing (i.e., wrapper class)
- Support various collection features
 - insert/delete operations
 - contains() operation
 - Getters
 - ...



Vector<Integer>

```
Vector<Integer> v = new Vector<Integer>();
```



Vector<E>: Methods

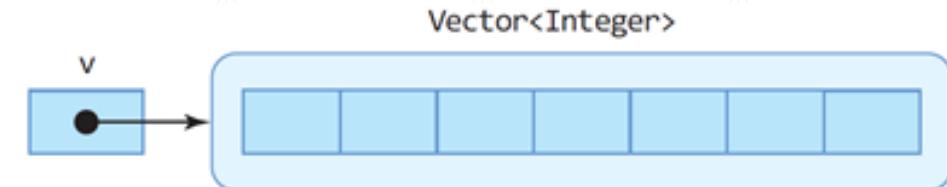
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Vector.html#method.summary>

Method	Description
boolean add(E element)	Appends the specified element to the end of this Vector
void add(int index, E element)	Inserts the specified element at the specified position in this Vector
int capacity()	Returns the current capacity of this Vector
boolean addAll(Collection<? extends E> c)	Appends all of the elements in the specified collection to the end of this Vector
void clear()	Removes all of the elements from this Vector
boolean contains(Object o)	Returns true if this Vector contains the specified element
E elementAt(int index)	Returns the component at the specified index
E get(int index)	Returns the element at the specified position in this Vector
int indexOf(Object o)	Returns the index of the first occurrence of the specified element in this Vector
boolean isEmpty()	Returns true if this Vector contains no elements
E remove(int index)	Removes the element at the specified position in this Vector
boolean remove(Object o)	Removes the first occurrence of the specified element from this Vector, if it is present
void removeAllElements()	Removes all components from this vector and sets its size to zero
int size()	Returns the number of elements in this Vector
Object[] toArray()	Returns an array containing all of the elements in this Vector in proper sequence

Vector<Integer>

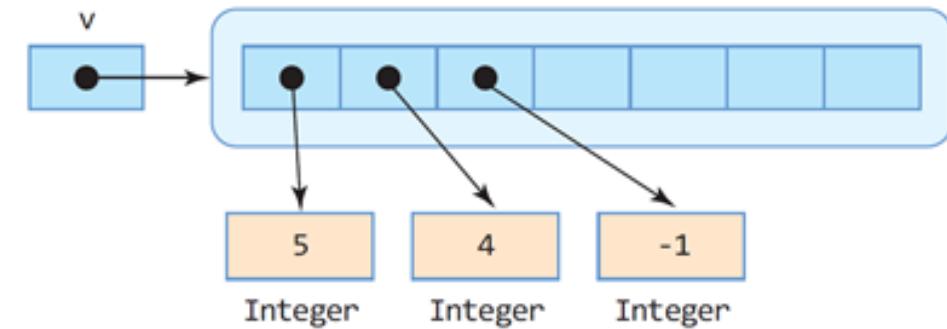
Create Vector

```
Vector<Integer> v = new Vector<Integer>(7);
```



Adding elements

```
v.add(5);  
v.add(4);  
v.add(-1);
```



Counting elements

```
int n = v.size();  
int c = v.capacity();
```

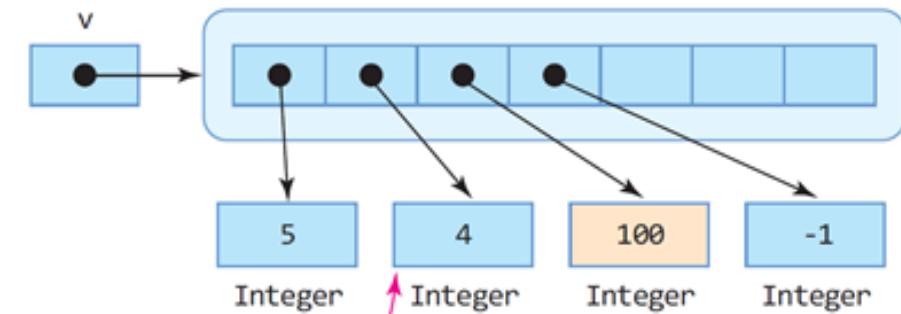
n = 3
c = 7

Vector<Integer> (cont'd)

Adding elements

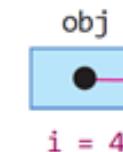
```
v.add(2, 100);
```

```
-v.add(5, 100);
```



Getting element

```
Integer obj = v.get(1);
int i = obj.intValue();
```

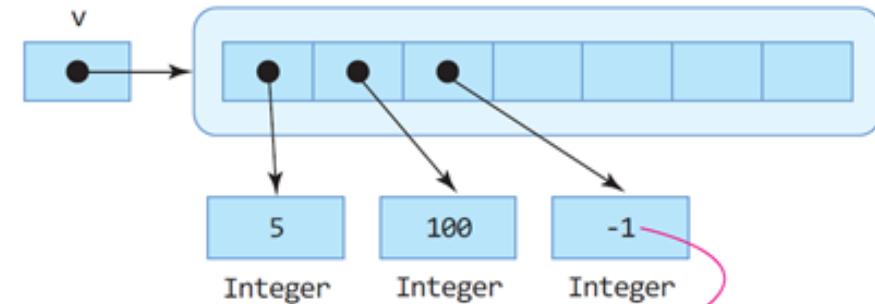


Vector<Integer> (cont'd)

Removing elements

```
v.remove(1);
```

```
-v.remove(4);
```



Removing all elements

```
int last = v.lastElement();
```

last = -1

```
v.removeAllElements();
```



Vector<Integer> (cont'd)

■ Auto boxing/unboxing

- Boxing: primitive type → wrapper class
- Unboxing: wrapper class → primitive type

```
Vector<Integer> v = new Vector<Integer>();  
v.add(4); // 4 → Integer.valueOf(4), auto boxing  
int k = v.get(0); // Integer → int, auto unboxing (k = 4)
```

■ Vector initialization with primitive type is impossible!

```
Vector<int> v = new Vector<int>(); // Error!
```

Vector<Integer>: Example

■ Basic usage of Vector<Integer>

```
import java.util.Vector;

public class VectorEx {
    public static void main(String[] args) {

        Vector<Integer> v = new Vector<Integer>();

        v.add(5);
        v.add(4);
        v.add(-1);

        // add element at specified index
        v.add(2, 100); // insert 100 between 4 and -1

        System.out.println("number of elements: " + v.size());
        System.out.println("current capacity: " + v.capacity());

        for(int i=0; i<v.size(); i++) {
            int n = v.get(i);
            System.out.println(n);
        }
    }
}
```

```
// sum all the number in the vector
int sum = 0;
for(int i=0; i<v.size(); i++) {
    int n = v.elementAt(i);
    sum += n;
}
System.out.println("sum of all integers in the vector: " + sum);
}
```

Vector<Integer>: Example (cont'd)

■ Usage of Vector with a custom class

```
import java.util.Vector;

class Point {
    private int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public String toString() {
        return "(" + x + "," + y + ")";
    }
}
```

```
public class PointVectorEx {
    public static void main(String[] args) {
        // Vector with Point class
        Vector<Point> v = new Vector<Point>();

        // adding 3 point instances
        v.add(new Point(2, 3)); // 0
        v.add(new Point(-5, 20)); // 1
        v.add(new Point(30, -8)); // 2

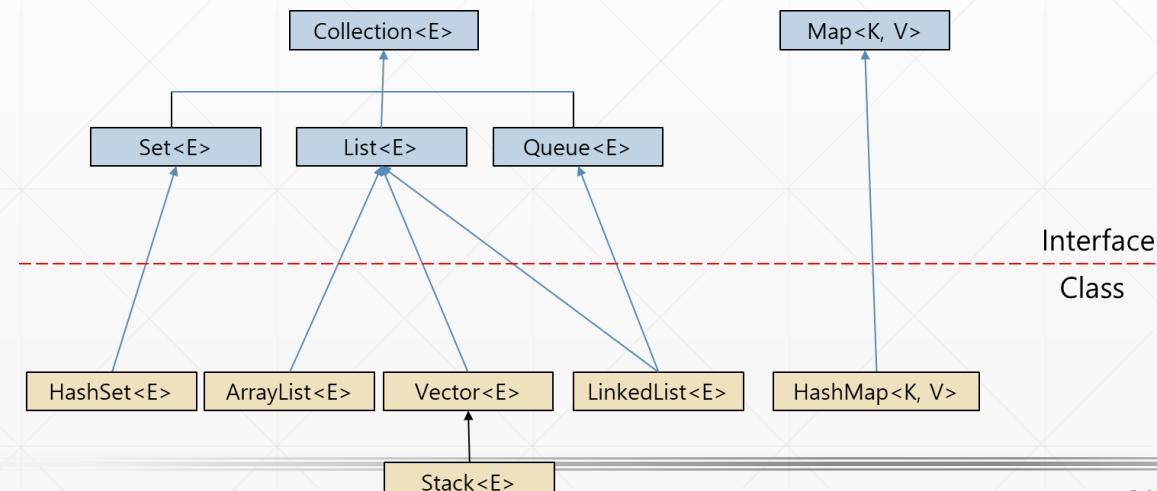
        v.remove(1); // remove a specific element

        //
        for(int i=0; i<v.size(); i++) {
            Point p = v.get(i); // getting i-th Point element
            System.out.println(p);
        }
    }
}
```

ArrayList<E>

■ Characteristics

- java.util.ArrayList, resizable-array implementation
 - Can be specialized for <E>
- ArrayList can contain:
 - Object, null
 - Primitive types after boxing (i.e., wrapper class)
- Support various collection features
 - Insert/delete operations
 - contains() operation
 - Getters
 - ...

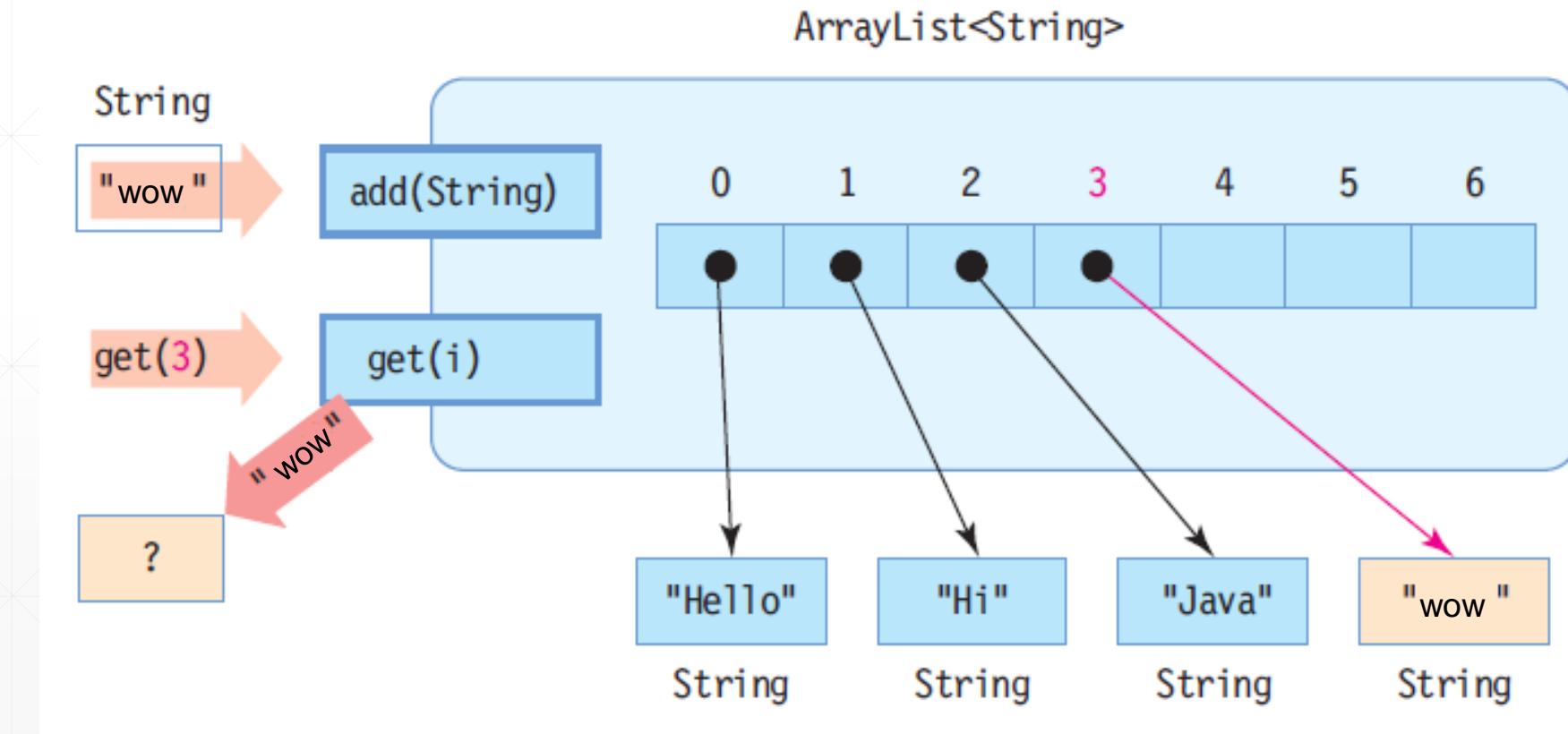


ArrayList<E>: Methods

Method	Description
boolean add(E element)	Appends the specified element to the end of this list
void add(int index, E element)	Inserts the specified element at the specified position in this list
boolean addAll(Collection<? extends E> c)	Appends all of the elements in the specified collection to the end of this list
void clear()	Removes all of the elements from this list
boolean contains(Object o)	Returns true if this list contains the specified element
E get(int index)	Returns the element at the specified position in this list
int indexOf(Object o)	Returns the index of the first occurrence of the specified element in this list
boolean isEmpty()	Returns true if this list contains no elements
E remove(int index)	Removes the element at the specified position in this list
boolean remove(Object o)	Removes the first occurrence of the specified element from this list, if it is present
int size()	Returns the number of elements in this list.
Object[] toArray()	Returns an array containing all of the elements in this list in proper sequence

ArrayList<String>

```
ArrayList<String> al = new ArrayList<String>();
```

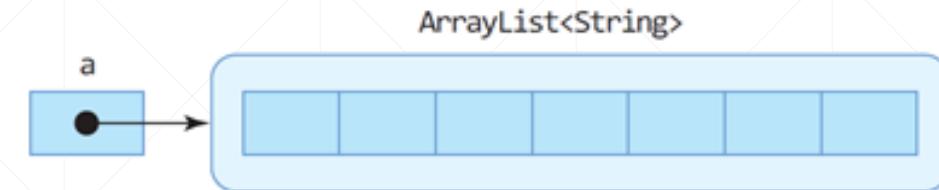


ArrayList<String> (cont'd)

■ Example)

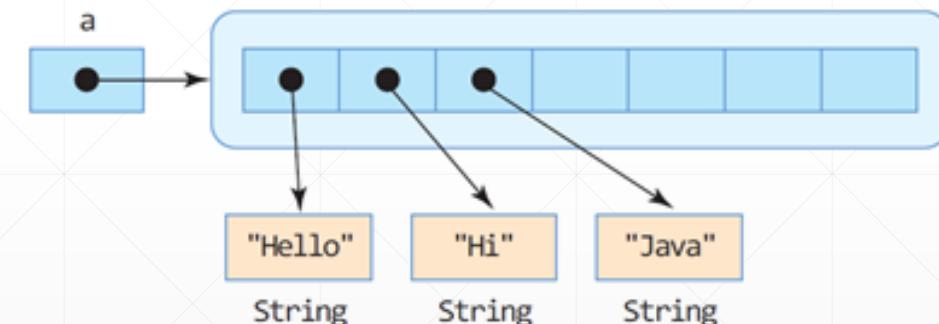
Create ArrayList

```
ArrayList<String> a = new ArrayList<String>(7);
```



Adding elements

```
a.add("Hello");
a.add("Hi");
a.add("Java");
```



Counting elements

```
int n = a.size();
int c = a.capacity();
```

n = 3

ArrayList<String> (cont'd)

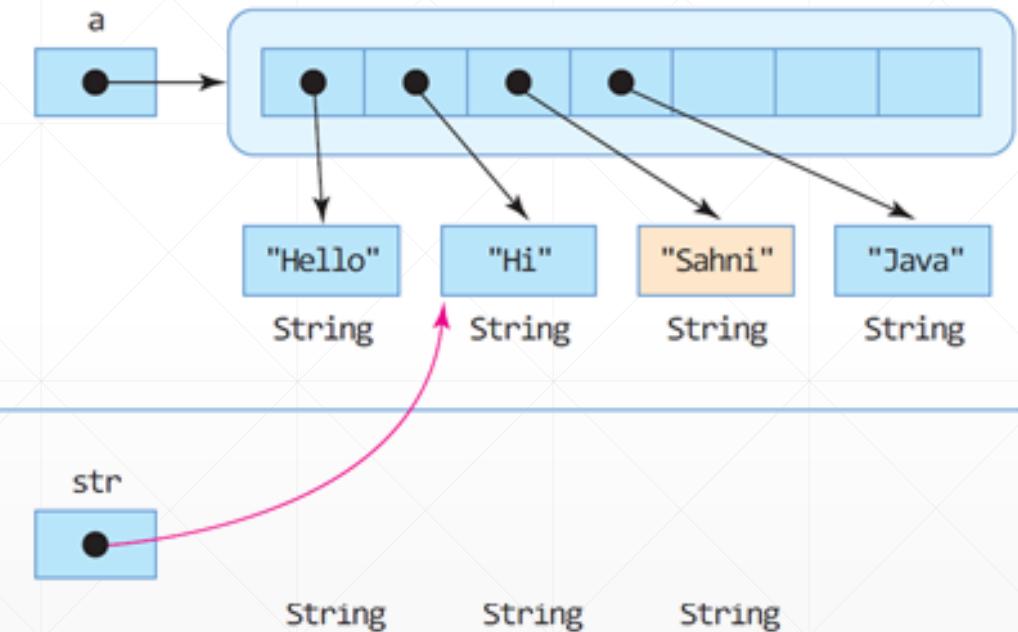
■ Example)

Adding elements

```
a.add(2, "Sahni");  
a.add(5, "Sahni");
```

Getting element

```
String str = a.get(1);
```



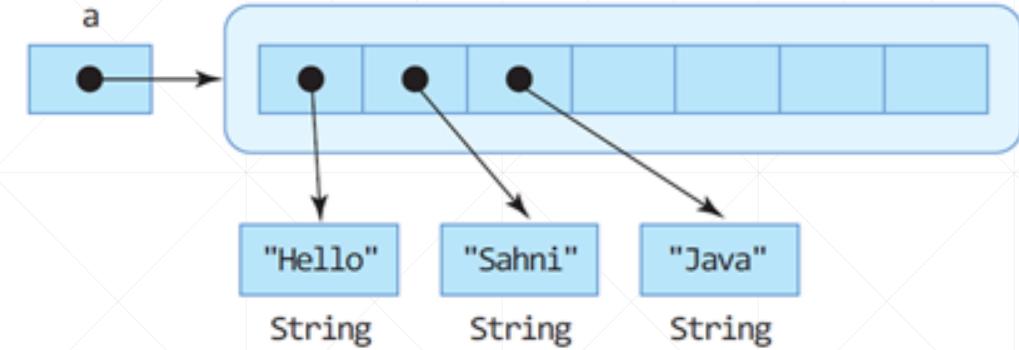
ArrayList<String> (cont'd)

■ Example)

Removing elements

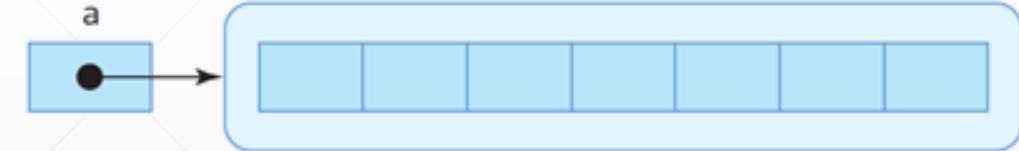
```
a.remove(1);
```

```
a.remove(4);
```



Removing all elements

```
a.clear();
```



ArrayList<String> (cont'd)

- Example) take 4 names and store them into ArrayList. Then, print all the names and the longest one.

```
import java.util.*;  
  
public class ArrayListEx {  
    public static void main(String[] args) {  
  
        ArrayList<String> a = new ArrayList<String>();  
        Scanner scanner = new Scanner(System.in);  
  
        for (int i = 0; i < 4; i++) {  
            System.out.print("Input your name >> ");  
            String s = scanner.next();  
            a.add(s);  
        }  
  
        // printing all names  
        for (int i = 0; i < a.size(); i++) {  
            // Getting i-th element  
            String name = a.get(i);  
            System.out.print(name + " ");  
        }  
    }  
}
```

```
...  
// finding the longest name  
int longestIndex = 0;  
for (int i = 1; i < a.size(); i++) {  
    if (a.get(longestIndex).length() < a.get(i).length())  
        longestIndex = i;  
}  
System.out.println("\n the longest one is : " + a.get(longestIndex));  
scanner.close();
```

Vector vs ArrayList

■ Speed Test

```
ArrayList<Integer> list = new ArrayList<Integer>();
Vector<Integer> vec = new Vector<Integer>();

new Thread(() -> {
    long startTime = System.currentTimeMillis();

    for (int i = 0; i < 10000000; i++) {
        list.add(1);
    }

    long endTime = System.currentTimeMillis();

    long durationTimeSec = endTime - startTime;
    System.out.println("ArrayList: " + durationTimeSec + "m/s");
}).start();

new Thread(() -> {
    long startTime = System.currentTimeMillis();

    for (int i = 0; i < 10000000; i++) {
        vec.add(1);
    }

    long endTime = System.currentTimeMillis();

    long durationTimeSec = endTime - startTime;
    System.out.println("Vector: " + durationTimeSec + "m/s");
}).start();
```

Iterator

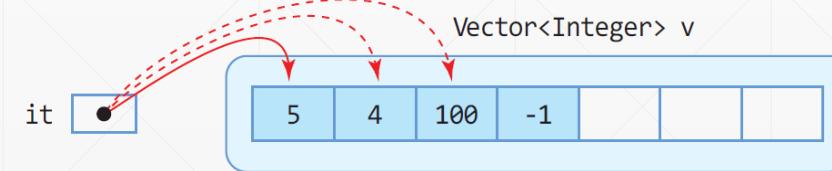
■ Iterator<E> interface

- Vector<E>, ArrayList<E>, LinkedList<E>
- Declare methods to iteratively visit the elements in the list-type data structure
- Methods

Method	Description
boolean hasNext()	Returns true if the iteration has more elements
E next()	Returns the next element in the iteration
void remove()	Removes from the underlying collection the last element returned by this iterator (optional operation)

- iterator() method: returns an iterator instance
 - Can use this instance to iteratively visit each element in the collection

```
Vector<Integer> v = new Vector<Integer>();  
Iterator<Integer> it = v.iterator();  
while(it.hasNext()) { //  
    int n = it.next(); //  
    ...  
}
```



Iterator: Example

■ Iterator<E> interface

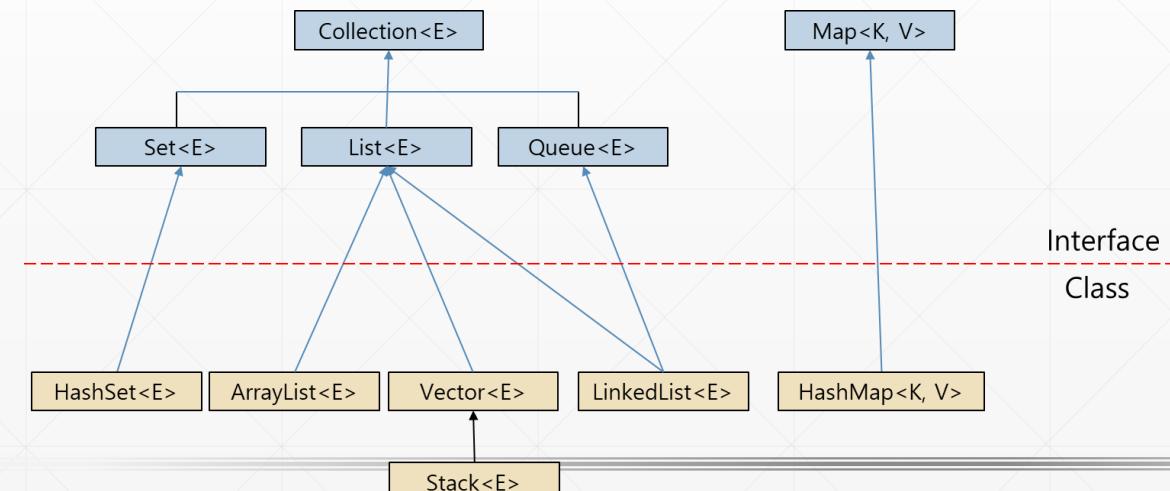
```
import java.util.*;  
  
public class IteratorEx {  
    public static void main(String[] args) {  
  
        Vector<Integer> v = new Vector<Integer>();  
        v.add(5);  
        v.add(4);  
        v.add(-1);  
        v.add(2, 100);  
  
        // print all elements using Iterator  
  
        Iterator<Integer> it = v.iterator();  
        while(it.hasNext()) {  
            int n = it.next();  
            System.out.println(n);  
        }  
    }  
}
```

```
// sum all the elements using Iterator  
  
int sum = 0;  
it = v.iterator();  
while(it.hasNext()) {  
    int n = it.next();  
    sum += n;  
}  
System.out.println("sum: " + sum);  
}  
}
```

HashMap<K, V>

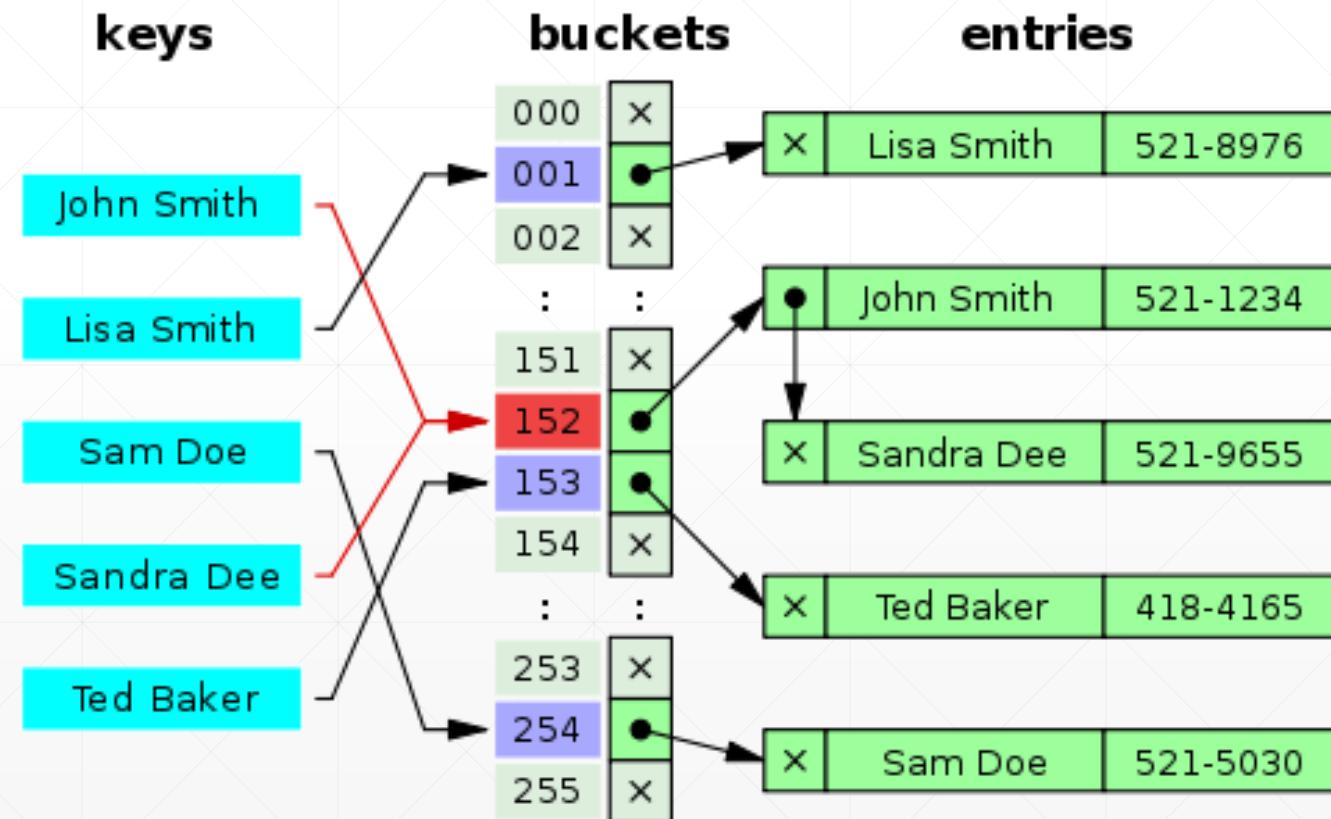
■ Characteristics

- java.util.HashMap
- Container class to manage key-value pairs
 - K: type to be used for keys, V: type to be used for values
 - Key determines a position where the element is located (therefore, **key must be unique**)
 - Values can be searched based on the key
- Support various collection features
 - Insert: put() method
 - Search: get() method
 - ...



HashMap<String, String>

```
HashMap<String, String> map = new HashMap<String, String>();
```



HashMap<K,V>

[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java.util/HashMap.html#method.summary](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/HashMap.html#method.summary)

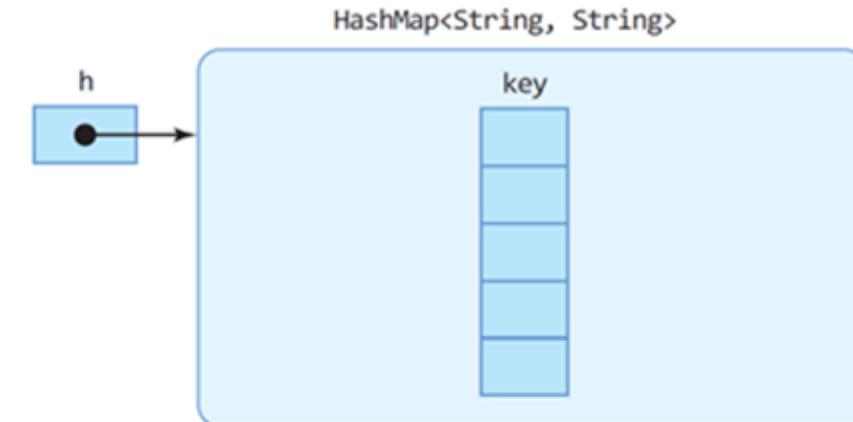
■ Methods

Method	Description
<code>void clear()</code>	Removes all of the mappings from this map
<code>boolean containsKey(Object key)</code>	Returns true if this map contains a mapping for the specified key
<code>boolean containsValue(Object value)</code>	Returns true if this map maps one or more keys to the specified value
<code>V get(Object key)</code>	Returns the value to which the specified key is mapped
<code>boolean isEmpty()</code>	Returns true if this map contains no key-value mappings
<code>Set<K> keySet()</code>	Returns a Set view of the keys contained in this map
<code>V put(K key, V value)</code>	Associates the specified value with the specified key in this map
<code>V remove(Object key)</code>	Removes the mapping for the specified key from this map if present
<code>int size()</code>	Returns the number of key-value mappings in this map

HashMap<String, String> (cont'd)

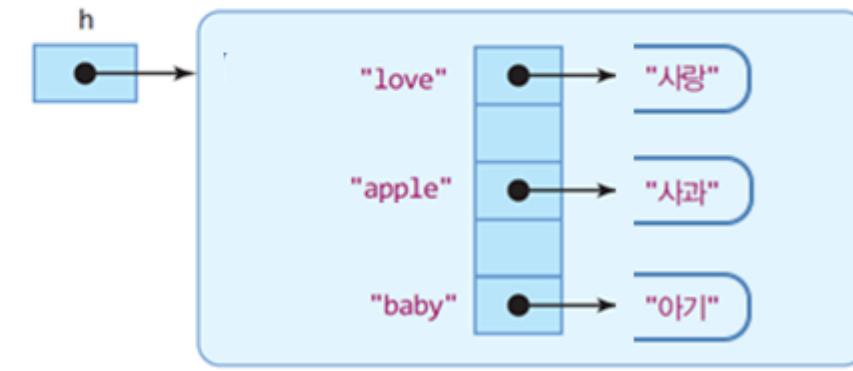
Create HashMap

```
HashMap<String, String> h =  
    new HashMap<String, String>();
```



Adding elements

```
h.put("baby", "아기");  
h.put("love", "사랑");  
h.put("apple", "사과");
```



HashMap<String, String> (cont'd)

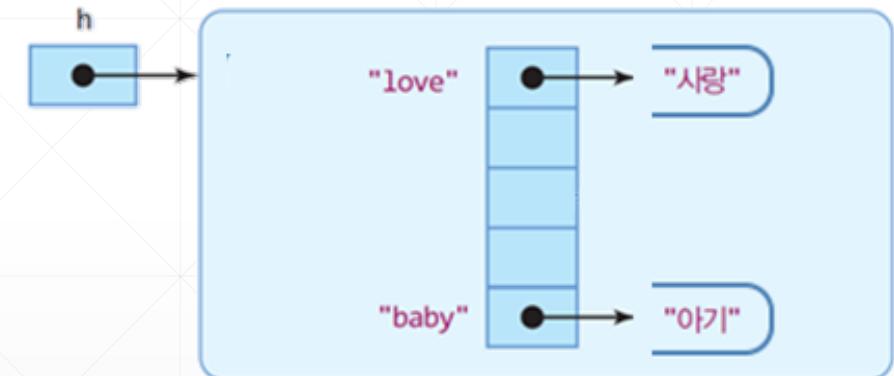
Getting elements

```
String kor = h.get("love");
```

kor = "사랑"

Removing elements

```
h.remove("apple");
```



Counting elements

```
int n = h.size();
```

n = 2

HashMap<K,V>: Example

■ Dictionary implementation

```
import java.util.*;

public class HashMapDicEx {
    public static void main(String[] args) {
        // HashMap for <String, String> pairs
        HashMap<String, String> dic = new HashMap<String, String>();

        // add 3 pairs
        dic.put("baby", "아기");
        dic.put("love", "사랑");
        dic.put("apple", "사과");

        // take English word and return its corresponding Korean word
        Scanner scanner = new Scanner(System.in);
        while(true) {
            System.out.print("which word?");
            String eng = scanner.next();
            if(eng.equals("exit")) {
                System.out.println("exit...");
                break;
            }
        }
    }
}
```

```
String kor = dic.get(eng);
if(kor == null)
    System.out.println(eng +
                      " does not exist.");
else
    System.out.println(kor);
}
scanner.close();
}
```

HashMap<K,V>: Example

■ ScoreTable implementation

```
public class HashMapScoreEx {  
    public static void main(String[] args) {  
        // HashMap for <String, Integer>  
        HashMap<String, Integer> javaScore =  
            new HashMap<String, Integer>();  
  
        javaScore.put("jinwoo", 97);  
        javaScore.put("jinhee", 88);  
        javaScore.put("jinha", 98);  
        javaScore.put("jinkoo", 70);  
        javaScore.put("jindo", 99);  
  
        System.out.println("HashMap's size :" + javaScore.size());  
  
        // print all (key, value) pairs in javaScore HashMap  
        // get Set collection containing all keys of HashMap  
        Set<String> keys = javaScore.keySet();  
  
        // get an Iterator for Set collection  
        Iterator<String> it = keys.iterator();
```

```
while(it.hasNext()) {  
    String name = it.next();  
    int score = javaScore.get(name);  
    // get the value for that key from HashMap  
    System.out.println(name + " : " + score);  
}  
}
```

HashMap<K,V>: Example

■ StudentTable implementation

```
class Student {  
    int id;  
    String tel;  
    public Student(int id, String tel) {  
        this.id = id; this.tel = tel;  
    }  
}
```

```
public class HashMapStudentEx {  
    public static void main(String[] args) {  
        // HashMap for <String, Student> pairs  
        HashMap<String, Student> map = new HashMap<String, Student>();  
  
        map.put("jinwoo", new Student(1, "010-111-1111"));  
        map.put("jindo", new Student(2, "010-222-2222"));  
        map.put("jinha", new Student(3, "010-333-3333"));  
  
        Scanner scanner = new Scanner(System.in);  
        while(true) {  
            System.out.print("name?");  
            String name = scanner.nextLine();  
            if(name.equals("exit"))  
                break; // exit the program  
            Student student = map.get(name);  
            if(student == null)  
                System.out.println(name + " does not exist.");  
            else  
                System.out.println("id:" + student.getId() + ", tel:" + student.getTel());  
        }  
        scanner.close();  
    }  
}
```

Collections

■ Java.util.collections

- Operates on collections and return collections
- Only has static methods

■ Methods

- sort()
- reverse()
- min()/max()
- ...

Collections: Example

■ Usage of Collections class

```
import java.util.*;

public class CollectionsEx {
    static void printList(Vector<String> l) {
        Iterator<String> iterator = l.iterator();
        while (iterator.hasNext()) {
            String e = iterator.next();
            String separator;
            if (iterator.hasNext())
                separator = "->";
            else
                separator = "\n";
            System.out.print(e+separator);
        }
    }
}
```

```
public static void main(String[] args) {
    Vector<String> myList = new Vector<String>();
    myList.add("Transformer");
    myList.add("StarWars");
    myList.add("Matrix");
    myList.add(0,"Terminator");
    myList.add(2,"Avatar");

    Collections.sort(myList); // sorting elements
    printList(myList);

    Collections.reverse(myList); // reversing elements
    printList(myList);

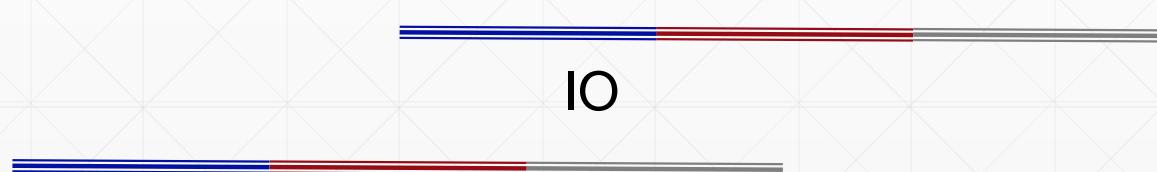
    System.out.println(Collections.min(myList));
}
```

Q&A

■ Next week

- File IO

Computer Language



Agenda

- IO
- Exercises

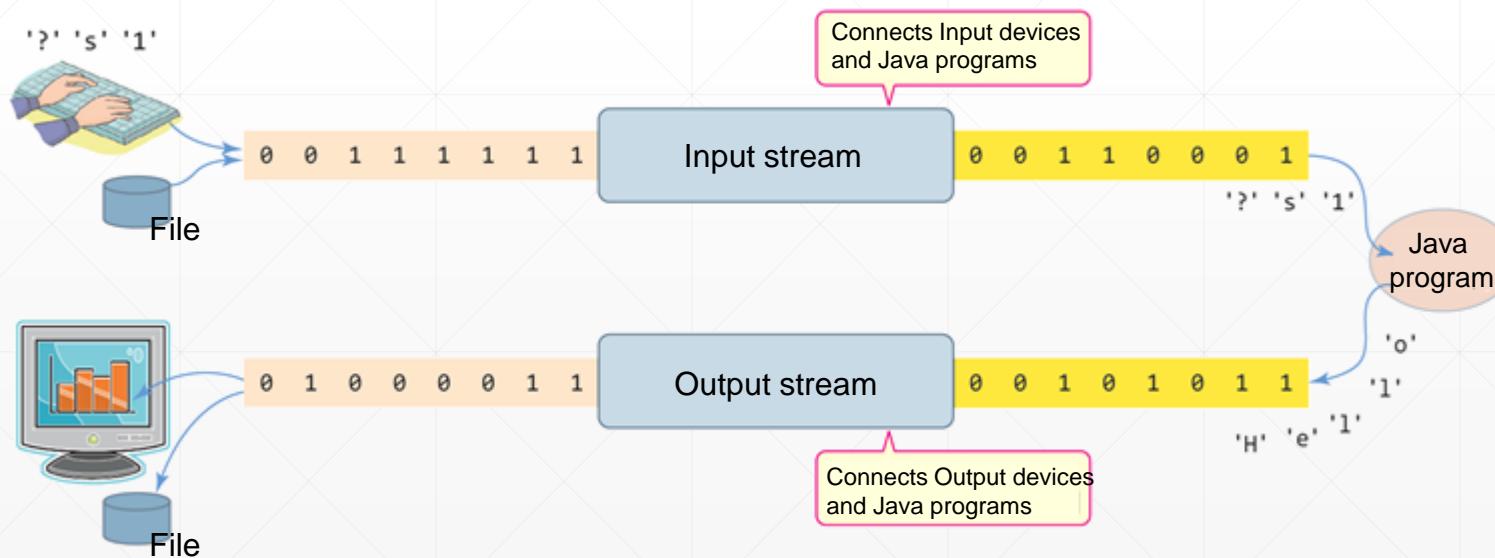
Stream

■ Stream IO

- Input and Output (IO) processing based on the buffer

■ Java's IO Stream

- Input stream: takes data from input devices and bring it to the java program
- Output stream: pass data to output devices



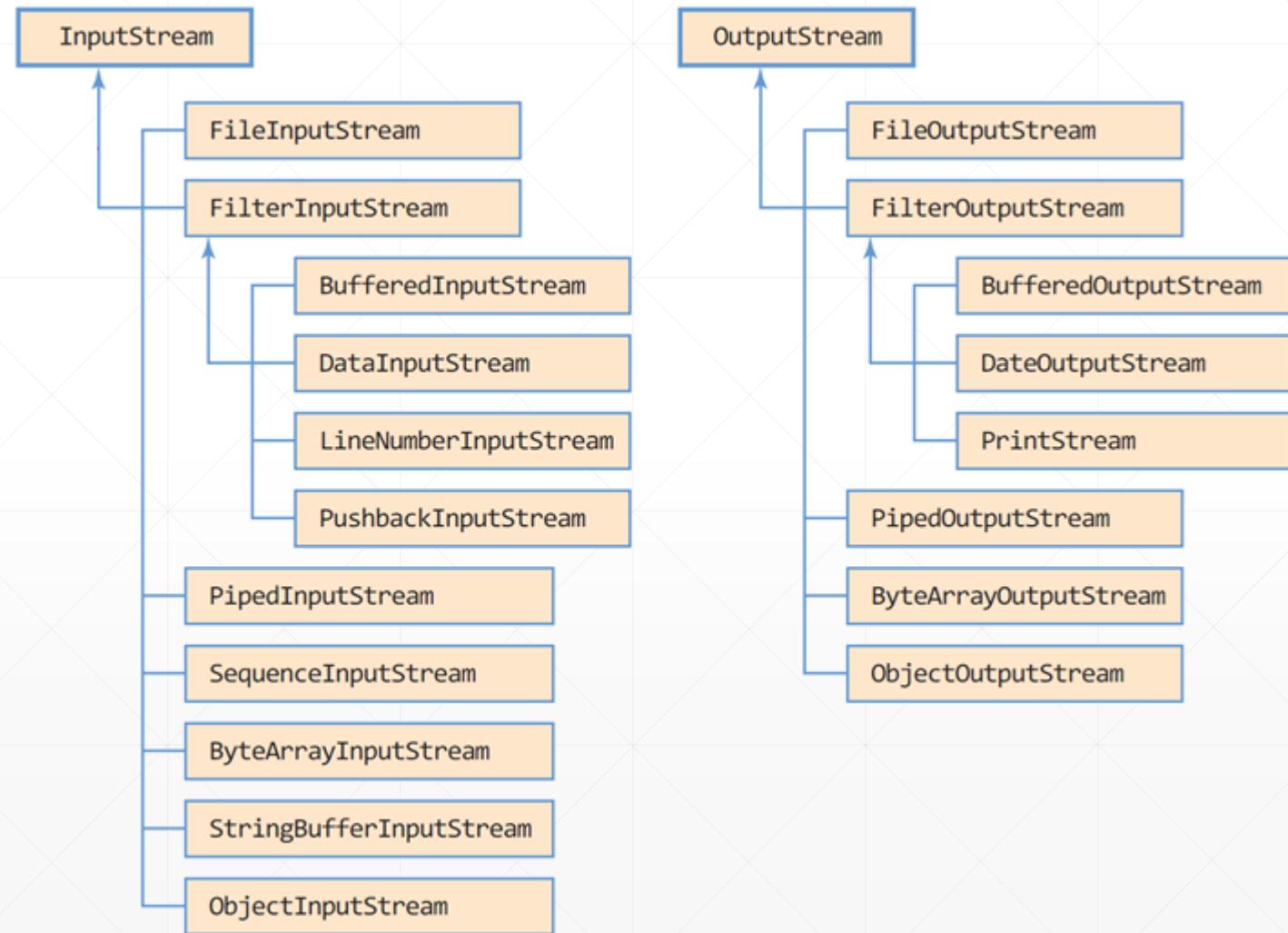
Stream (cont'd)

■ Characteristics

- Uni-directional
- Basic unit
 - Byte for byte stream
 - Transmits any type of data (e.g., image, video, etc.)
 - Character for character stream
 - Transmits character data only (e.g., text file)
- FIFO
 - Frist-in first-out

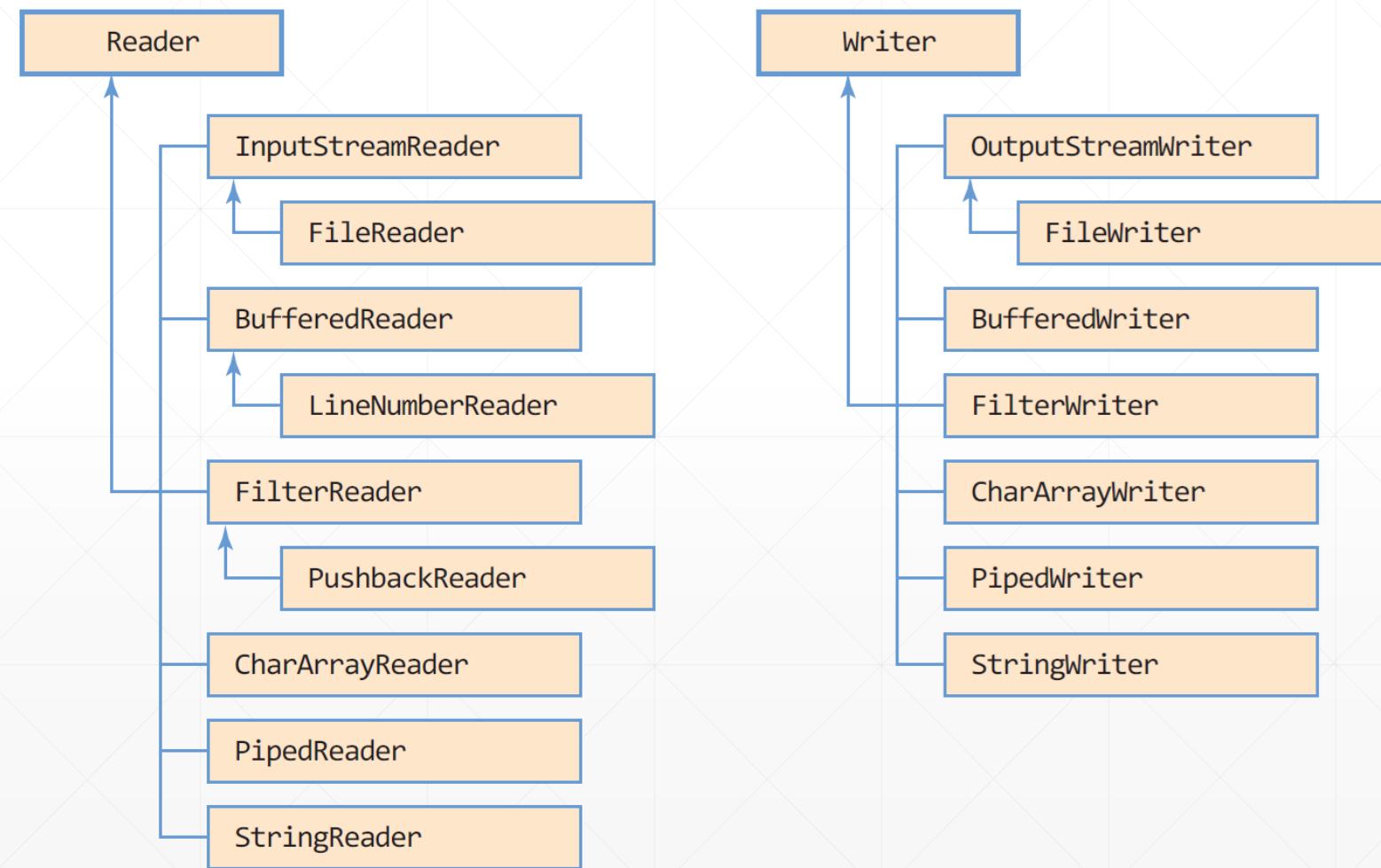
Stream (cont'd)

■ Byte stream hierarchy



Stream (cont'd)

■ Character stream hierarchy



Stream (cont'd)

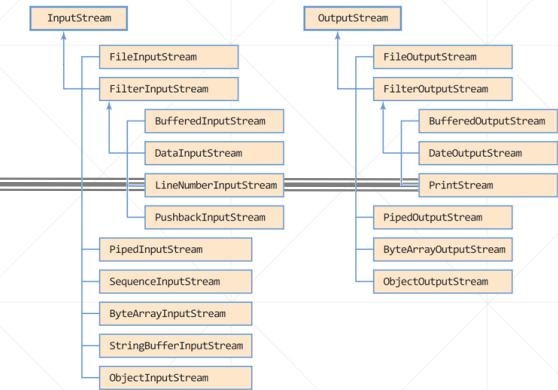
■ Root abstract classes of byte streams

➤ InputStream

void	<code>close()</code>	Closes this input stream and releases any system resources associated with the stream.
abstract int	<code>read()</code>	Reads the next byte of data from the input stream.
int	<code>read(byte[] b)</code>	Reads some number of bytes from the input stream and stores them into the buffer array b.
int	<code>read(byte[] b, int off, int len)</code>	Reads up to len bytes of data from the input stream into an array of bytes.

➤ OutputStream

void	<code>close()</code>	Closes this output stream and releases any system resources associated with this stream.
void	<code>flush()</code>	Flushes this output stream and forces any buffered output bytes to be written out.
void	<code>write(byte[] b)</code>	Writes b.length bytes from the specified byte array to this output stream.
void	<code>write(byte[] b, int off, int len)</code>	Writes len bytes from the specified byte array starting at offset off to this output stream.
abstract void	<code>write(int b)</code>	Writes the specified byte to this output stream.

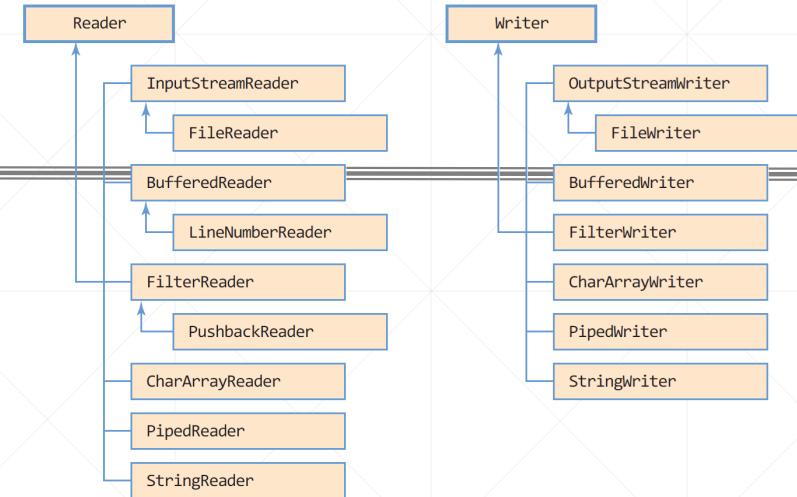


Stream (cont'd)

■ Root abstract classes of character streams

➤ Reader

int	<code>read()</code>	Reads a single character
int	<code>read(char[] cbuf)</code>	Reads characters into an array
abstract int	<code>read(char[] cbuf, int off, int len)</code>	Reads characters into a portion of an array



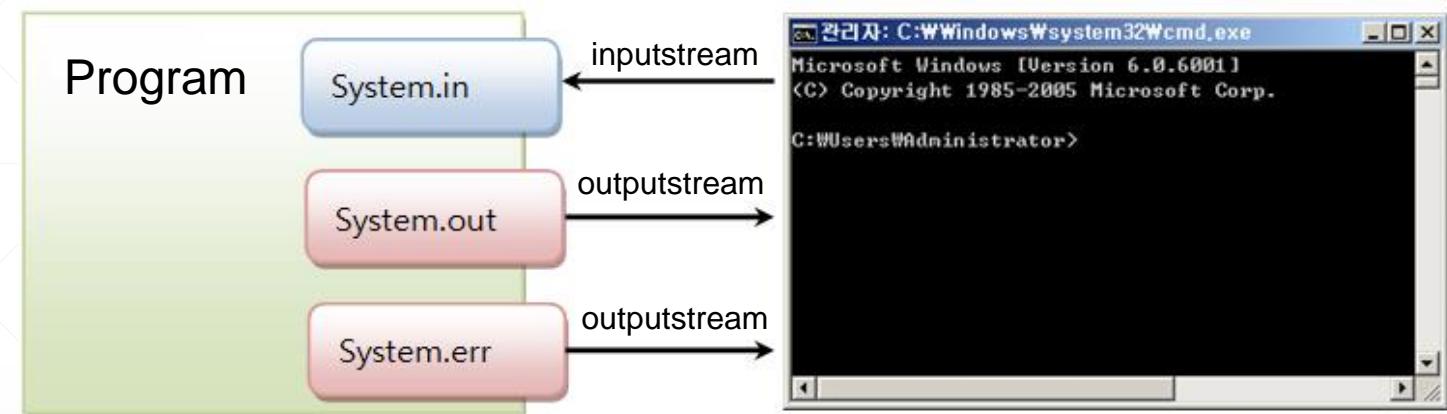
➤ Writer

abstract void	<code>flush()</code>	Flushes the stream
void	<code>write(char[] cbuf)</code>	Writes an array of characters
abstract void	<code>write(char[] cbuf, int off, int len)</code>	Writes a portion of an array of characters
void	<code>write(int c)</code>	Writes a single character
void	<code>write(String str)</code>	Writes a string
void	<code>write(String str, int off, int len)</code>	Writes a portion of a string

Stream: Console

■ System software, application interface

- Linux terminal, windows prompt, IntelliJ/Eclipse console, etc.
- Take input from the keyboard
- Output the data to display



■ System class

- “in” field: standard input stream (`InputStream`)
- “out” field: standard output stream (`PrintStream`)
- “err” field: standard error error stream (`PrintStream`)

FileReader

■ Reading a text file

Constructor	Description
FileReader(File file)	Creates a new FileReader, given the File to read
FileReader(File file, Charset charset)	Creates a new FileReader, given the File to read and the charset .
FileReader(String fileName)	Creates a new FileReader, given the name of the file to read
FileReader(String fileName, Charset charset)	Creates a new FileReader, given the name of the file to read and the charset .

```
public class FileReaderEx {  
    public static void main(String[] args) {  
        FileReader fin = null;  
        try {  
            fin = new FileReader("c:\windows\system.ini");  
            int c;  
            while ((c = fin.read()) != -1) { // read a character  
                System.out.print((char)c);  
            }  
            fin.close();  
        }  
        catch (IOException e) {  
            System.out.println("IO error!");  
        }  
    }  
}
```

FileWriter

■ Writing to a text file

Constructor	Description
<code>FileWriter(File file)</code>	Constructs a FileWriter given the File to write
<code>FileWriter(File file, boolean append)</code>	Constructs a FileWriter given the File to write and a boolean indicating whether to append the data written
<code>FileWriter(File file, Charset charset)</code>	Constructs a FileWriter given the File to write and <code>charset</code> .
<code>FileWriter(File file, Charset charset, boolean append)</code>	Constructs a FileWriter given the File to write, <code>charset</code> and a boolean indicating whether to append the data written.
<code>FileWriter(String fileName)</code>	Constructs a FileWriter given a file name
<code>FileWriter(String fileName, boolean append)</code>	Constructs a FileWriter given a file name and a boolean indicating whether to append the data written
<code>FileWriter(String fileName, Charset charset)</code>	Constructs a FileWriter given a file name and <code>charset</code> .
<code>FileWriter(String fileName, Charset charset, boolean append)</code>	Constructs a FileWriter given a file name, <code>charset</code> and a boolean indicating whether to append the data written.

■ Character or Block writing is possible

```
FileWriter fout = new FileWriter("c:\\Temp\\test.txt");
fout.write('A'); // writing character 'A' to the file
fout.close();
```

```
char [] buf = new char [1024];
// writing the contents of buf[] (1024 characters) to the file
fout.write(buf, 0, buf.length);
```

FileWriter (cont'd)

■ Writing to a text file

```
import java.io.*;
import java.util.*;

public class FileWriterEx {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        FileWriter fout = null;
        int c;
        try {
            fout = new FileWriter("c:\Temp\test.txt");
            while(true) {
                String line = scanner.nextLine();
                if(line.length() == 0)
                    break;
                fout.write(line);
                fout.write("\r\n");
            }
            fout.close();
        } catch (IOException e) {
            System.out.println("IO error!");
        }
        scanner.close();
    }
}
```

Inserts "\r\n" escape characters to insert a new line

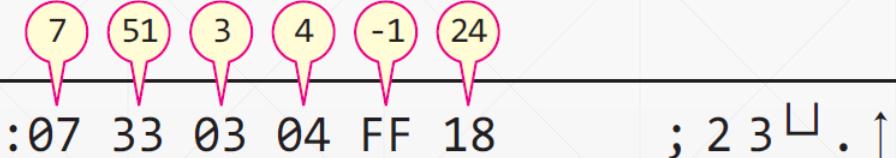
FileOutputStream

■ Writing to a binary file

Constructor	Description
FileOutputStream(File file)	Creates a file output stream to write to the file represented by the specified File object.
FileOutputStream(File file, boolean append)	Creates a file output stream to write to the file represented by the specified File object.
FileOutputStream(String name)	Creates a file output stream to write to the file with the specified name.
FileOutputStream(String name, boolean append)	Creates a file output stream to write to the file with the specified name.

■ Binary data is not human-readable (not text!)

```
byte b[] = {7, 51, 3, 4, -1, 24};  
try {  
    FileOutputStream fout =  
        new FileOutputStream("c:\\Temp\\test.out");  
    for (int i = 0; i < b.length; i++)  
        fout.write(b[i]);  
    fout.close();  
} catch (IOException e) {  
    System.out.println("could not save the file!");  
    return;  
}  
System.out.println("saved to c:\\Temp\\test.out");
```



FileInputStream

■ Reading a binary file

Constructor	Description
FileInputStream(File file)	Creates a FileInputStream by opening a connection to an actual file, the file named by the File object file in the file system.
FileInputStream(String name)	Creates a FileInputStream by opening a connection to an actual file, the file named by the path name name in the file system.

```
byte b[] = new byte [6];
try {
    FileInputStream fin = new FileInputStream("c:\\Temp\\test.out");
    int n=0, c;
    while((c = fin.read())!= -1) {
        b[n] = (byte)c;
        n++;
    }
    System.out.println("Printing the contents from c:\\Temp\\test.out");
    for(int i=0; i<b.length; i++) System.out.print(b[i] + " ");
    System.out.println();
    fin.close();
} catch(IOException e) {
    System.out.println( "could not read c:\\Temp\\test.out!!");
}
```

Auxiliary Stream

■ Bridge between the streams

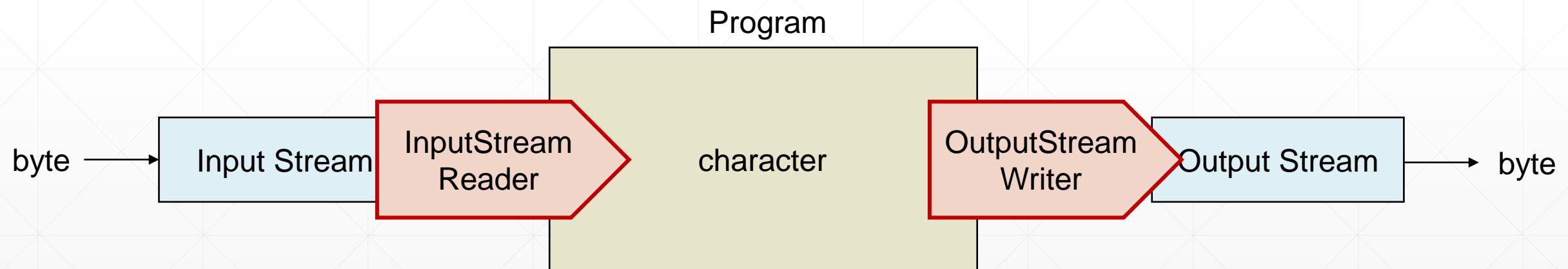
- Provides useful stream features
 - Character conversion
 - Buffered I/O
 - Object I/O
 - ...
- Can be chained



Auxiliary Stream: Character Conversion

■ InputStreamReader / OutputStreamWriter

- Converts byte data from Input stream to character data
- Converts character data to byte data for Output stream
- Can set a specific character set

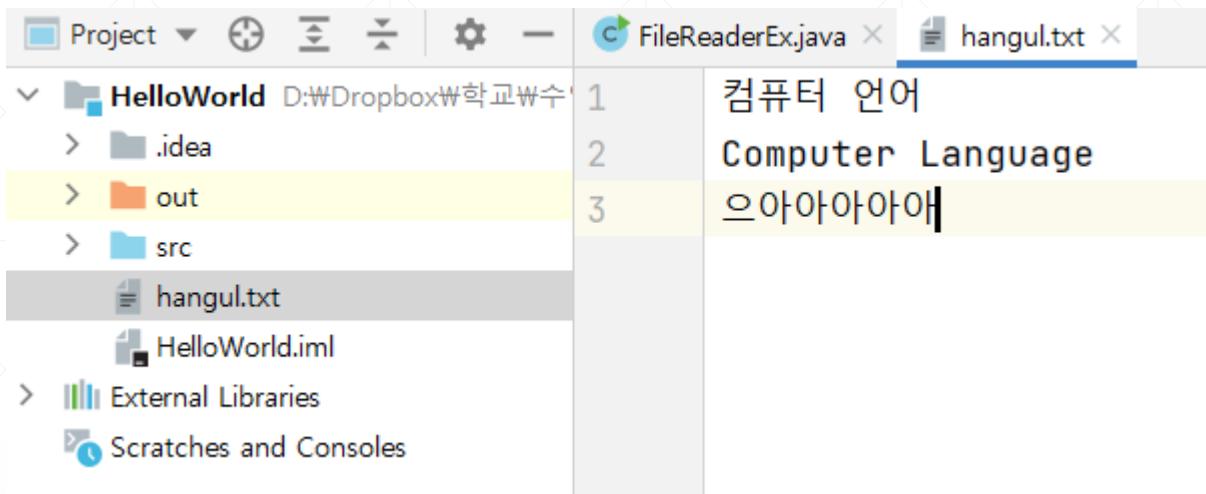


Auxiliary Stream: Character Conversion (cont'd)

■ Example)

```
InputStreamReader in = null;
FileInputStream fin = null;
try {
    fin = new FileInputStream("hangul.txt");
    in = new InputStreamReader(fin, "utf-8");
    int c;

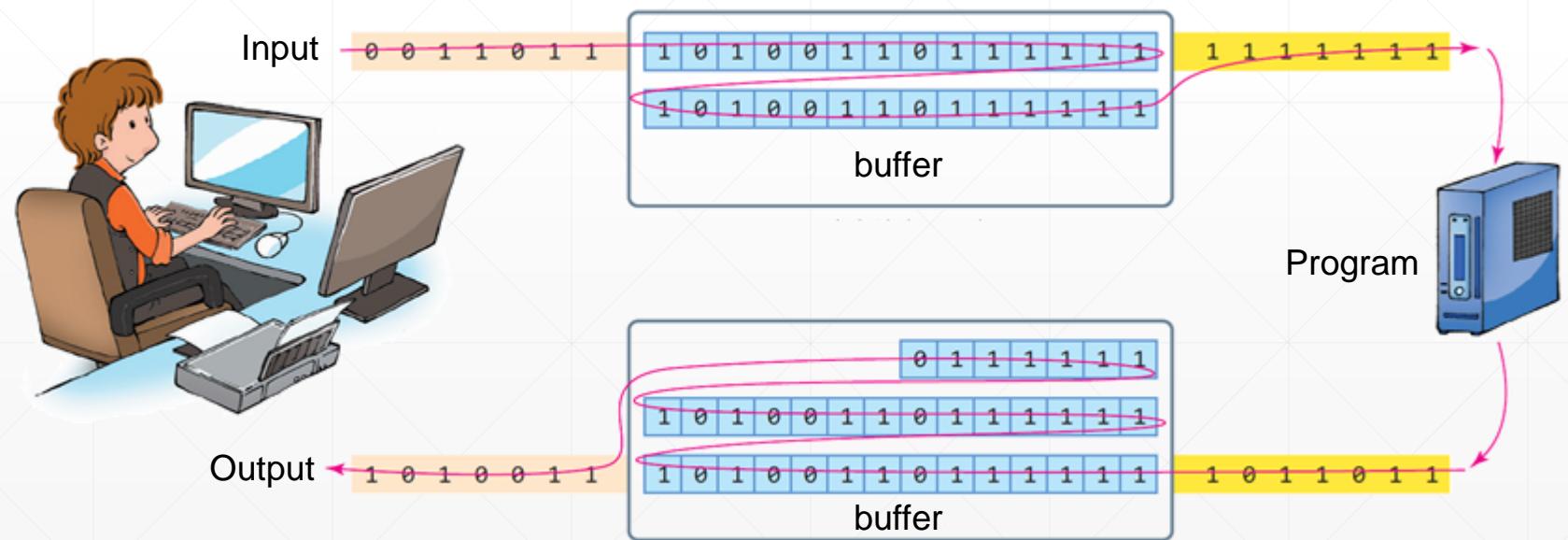
    System.out.println("encoding: " + in.getEncoding());
    while ((c = in.read()) != -1) {
        System.out.print((char)c);
    }
    in.close();
    fin.close();
} catch (IOException e) {
    System.out.println("IO error!");
}
```



Auxiliary Stream: Buffering I/O

■ Buffered Streams

- `BufferedInputStream / BufferedOutputStream` (for binary data)
- `BufferedReader / BufferedWriter` (for character data)
- Improves I/O efficiency by reducing native I/O operations
 - Keep the data in the buffer!



Auxiliary Stream: Buffering I/O (cont'd)

■ Example)

```
FileReader fin = null;  
int c;  
try {  
    fin = new FileReader("c:\\windows\\system.ini");  
    BufferedOutputStream out = new  
        BufferedOutputStream(System.out, 128);  
    while ((c = fin.read()) != -1) {  
        out.write(c);  
    }  
  
    new Scanner(System.in).nextLine(); // waiting for Enter  
    out.flush(); // flushing buffer!  
    fin.close();  
    out.close();  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

File

■ java.io.File

- Class handling a file's information (metadata)
 - Path of file/directory
- Class handling file management
 - Renaming, removing, creating, etc
- Does not support read/write functionalities

■ File instance

```
File f = new File("c:\windows\system.ini");
```

File (cont'd)

■ Methods

➤ Creation and deletion

Modifier and Type	Method	Description
boolean	createNewFile()	Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
boolean	mkdir()	Creates the directory named by this abstract pathname.
boolean	mkdirs()	Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories.
boolean	delete()	Deletes the file or directory denoted by this abstract pathname.

File (cont'd)

■ Methods

➤ Get information of files and directories

Modifier and Type	Method	Description
boolean	canExecute()	Tests whether the application can execute the file denoted by this abstract pathname.
boolean	canRead()	Tests whether the application can read the file denoted by this abstract pathname.
boolean	canWrite()	Tests whether the application can modify the file denoted by this abstract pathname.
String	getName()	Returns the name of the file or directory denoted by this abstract pathname.
String	getParent()	Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
File	getParentFile()	Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.
String	getPath()	Converts this abstract pathname into a pathname string.

File (cont'd)

■ Methods

➤ Get information of files and directories

Modifier and Type	Method	Description
boolean	isDirectory()	Tests whether the file denoted by this abstract pathname is a directory.
boolean	isFile()	Tests whether the file denoted by this abstract pathname is a normal file.
long	length()	Returns the length of the file denoted by this abstract pathname.
String[]	list()	Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
File[]	listFiles()	Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.

File (cont'd)

■ Example)

- Create a File instance
- Get File path
- Check
- Get files and subdirectories

```
File f = new File("c:\windows\system.ini");
```

```
String filename = f.getName(); // "system.ini"  
String path = f.getPath(); // "c:\windows\system.ini"  
String parent = f.getParent(); // "c:\windows"
```

```
if(f.isFile()) // in case of file  
    System.out.println(f.getPath() + "is a file.");  
else if(f.isDirectory()) // in case of directory  
    System.out.println(f.getPath() + "is a directory.");
```

```
File f = new File("c:\Temp");  
File[] subfiles = f.listFiles(); // get files and subdirectories of c:\Temp  
  
for(int i=0; i< subfiles.length; i++) {  
    System.out.print(subfiles[i].getName()); // print names  
    System.out.println("File size: " + subfiles[i].length()); // print length  
}
```

File (cont'd)

■ Example)

```
import java.io.File;

public class FileEx {
    public static void listDirectory(File dir) {
        System.out.println("----" + dir.getPath() + "s sub list ----");
        File[] subFiles = dir.listFiles();
        for(int i=0; i<subFiles.length; i++) {
            File f = subFiles[i];
            long t = f.lastModified();
            System.out.print(f.getName());
            System.out.print("File Size: " + f.length());
            System.out.printf("Modified time: %tb %td %ta %tT%n",t, t, t, t);
        }
    }

    public static void main(String[] args) {
        File f1 = new File("c:\windows\system.ini");
        System.out.println(f1.getPath() + ", " + f1.getParent() + ", " + f1.getName());
        String res="";
        if(f1.isFile()) res = "File";
        else if(f1.isDirectory()) res = "Directory";
        System.out.println(f1.getPath() + " is " + res);
    }
}
```

```
File f2 = new File("c:\Temp\java_sample");
if(!f2.exists()) {
    f2.mkdir(); // if not exist, make a new directory
}
listDirectory(new File("c:\Temp"));
f2.renameTo(new File("c:\Temp\javasample"));
listDirectory(new File("c:\Temp"));
}
```

Example: Copying Text Files

```
import java.io.*;

public class TextCopyEx {
    public static void main(String[] args){
        File src = new File("c:\windows\system.ini"); // source file
        File dest = new File("c:\Temp\system.txt"); // destination file
        int c;
        try {
            FileReader fr = new FileReader(src);
            FileWriter fw = new FileWriter(dest);
            while((c = fr.read()) != -1) { // read a single character
                fw.write((char)c); // write a single character
            }
            fr.close(); fw.close();
            System.out.println(src.getPath() + " copied to " + dest.getPath());
        } catch (IOException e) {
            System.out.println("IO error!");
        }
    }
}
```

Example: Copying Binary Files

```
import java.io.*;

public class BinaryCopyEx {
    public static void main(String[] args) {
        File src = new File("img1.jpg");
        File dest = new File("copyimg.jpg");
        int c;
        try {
            FileInputStream fi = new FileInputStream(src);
            FileOutputStream fo = new FileOutputStream(dest);
            while((c = fi.read()) != -1) {
                fo.write((byte)c);
            }
            fi.close();
            fo.close();
            System.out.println(src.getPath() + " copied to " + dest.getPath());
        } catch (IOException e) {
            System.out.println("IO Error!");
        }
    }
}
```



Example: Copying Binary Files with Buffer

```
import java.io.*;

public class BinaryCopyEx {
    public static void main(String[] args) {
        File src = new File("img1.jpg");
        File dest = new File("copyimg.jpg");
        int c;
        try {
            FileInputStream fi = new FileInputStream(src);
            FileOutputStream fo = new FileOutputStream(dest);

            BufferedInputStream bi = new BufferedInputStream (fi);
            BufferedOutputStream bo = new BufferedOutputStream (fo);

            while((c = bi.read()) != -1) {
                bo.write((byte)c);
            }
            bi.close();
            bo.close();
            fi.close();
            fo.close();
            System.out.println(src.getPath() + " copied to " + dest.getPath());
        } catch (IOException e) {
            System.out.println("IO Error!");
        }
    }
}
```



Scanner with File

■ A simple text scanner

➤ Java.util.scanner

Constructors	
Constructor	Description
<code>Scanner(File source)</code>	Constructs a new Scanner that produces values scanned from the specified file.
<code>Scanner(File source, String charsetName)</code>	Constructs a new Scanner that produces values scanned from the specified file.
<code>Scanner(File source, Charset charset)</code>	Constructs a new Scanner that produces values scanned from the specified file.
<code>Scanner(InputStream source)</code>	Constructs a new Scanner that produces values scanned from the specified input stream.
<code>Scanner(InputStream source, String charsetName)</code>	Constructs a new Scanner that produces values scanned from the specified input stream.
<code>Scanner(InputStream source, Charset charset)</code>	Constructs a new Scanner that produces values scanned from the specified input stream.
<code>Scanner(Readable source)</code>	Constructs a new Scanner that produces values scanned from the specified source.
<code>Scanner(String source)</code>	Constructs a new Scanner that produces values scanned from the specified string.
<code>Scanner(ReadableByteChannel source)</code>	Constructs a new Scanner that produces values scanned from the specified channel.
<code>Scanner(ReadableByteChannel source, String charsetName)</code>	Constructs a new Scanner that produces values scanned from the specified channel.
<code>Scanner(ReadableByteChannel source, Charset charset)</code>	Constructs a new Scanner that produces values scanned from the specified channel.
<code>Scanner(Path source)</code>	Constructs a new Scanner that produces values scanned from the specified file.
<code>Scanner(Path source, String charsetName)</code>	Constructs a new Scanner that produces values scanned from the specified file.
<code>Scanner(Path source, Charset charset)</code>	Constructs a new Scanner that produces values scanned from the specified file.

Scanner with File (cont'd)

■ Reading Text files using Scanner

- Scanner(File)
- Scanner(FileReader)

```
try {  
    Scanner scn = new Scanner(new File("c:\\windows\\system.ini"));  
    while (scn.hasNext()) {  
        String tmp = scn.nextLine();  
        System.out.println(tmp);  
    }  
    scn.close();  
  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

```
FileReader fin = null;  
try {  
    fin = new FileReader("c:\\windows\\system.ini");  
    int c;  
    while ((c = fin.read()) != -1) { // read a character  
        System.out.print((char)c);  
    }  
    fin.close();  
}  
catch (IOException e) {  
    System.out.println("IO error!");  
}
```

```
FileReader fin = null;  
try {  
    fin = new FileReader("c:\\windows\\system.ini");  
    Scanner scn = new Scanner(fin);  
    while(scn.hasNext()) {  
        String tmp = scn.nextLine();  
        System.out.println(tmp);  
    }  
    fin.close();  
    scn.close();  
  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

Q&A

■ Next week (Offline Test)

- Final exam (Openbook lab test)
- 29/May, 10:00 ~ 13:00