



Modern Java

전체 목차

1. Intro to Java 8
2. 클래스의 이해
3. 클래스 관계
4. Java API
5. Java 8

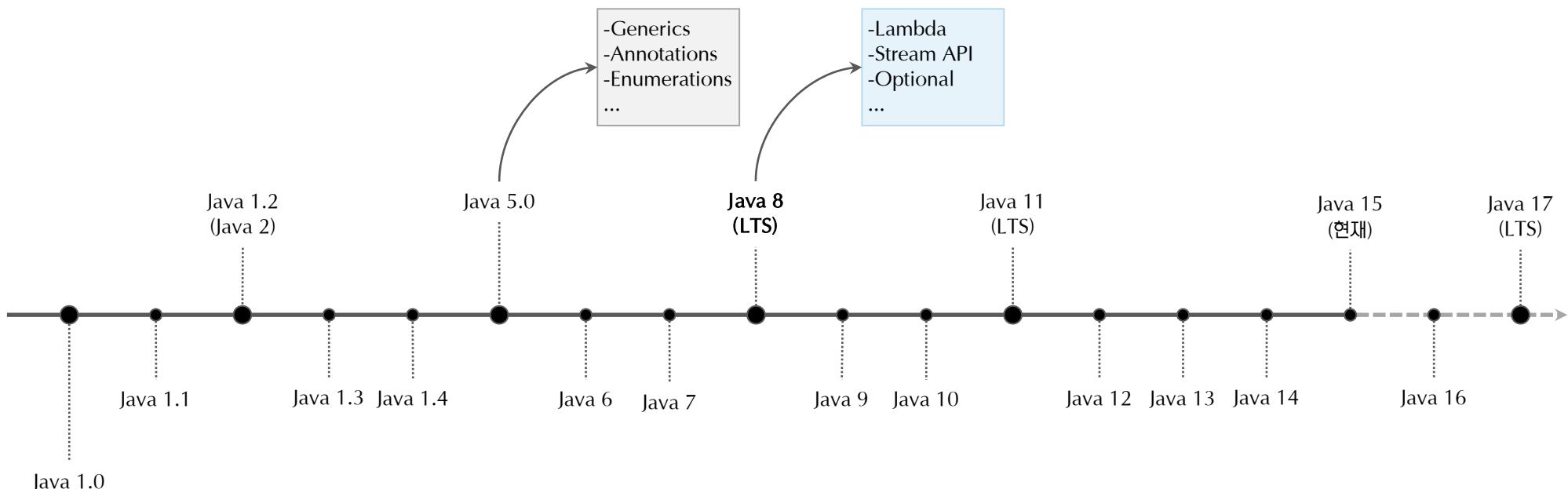


1. Intro to Java 8

- 1.1 Java 소개
- 1.2 Java 주요 특징
- 1.3 Java Application 동작 방식
- 1.4 환경 구성
- 1.5 첫번째 자바프로그래밍

1.1 Java 소개

- ✓ 자바 언어는 1996년 JDK(Java Development Kit) 1.0이 발표된 이후 현재까지 발전을 거듭하고 있습니다.
- ✓ 각각의 버전마다 크고 작은 변화들이 있었으며 Java 5과 Java 8 버전에서 가장 큰 변화가 있었습니다.
- ✓ Java 8 버전부터는 LTS(Long Term Support) 버전과 Non-LTS 버전으로 나뉘어 릴리즈 되고 있습니다.
- ✓ Java 9부터 6개월 단위로 새로운 버전이 나오고 있으며 Non-LTS 버전은 새로운 버전이 출시되면 더이상 업데이트가 지원되지 않습니다.

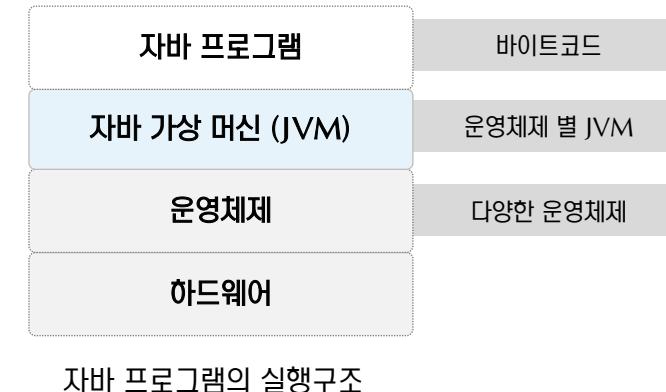
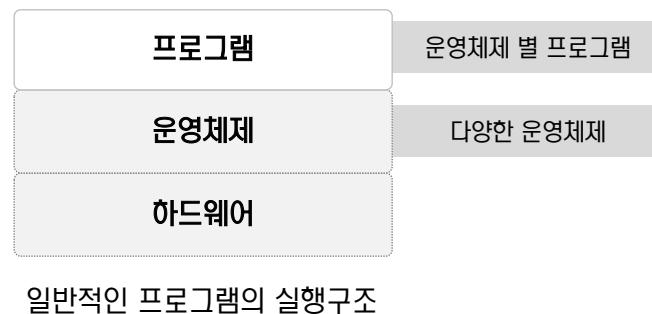


1.2 Java 주요 특징

✓ 자바 언어의 대표적인 특징은 다음과 같습니다.

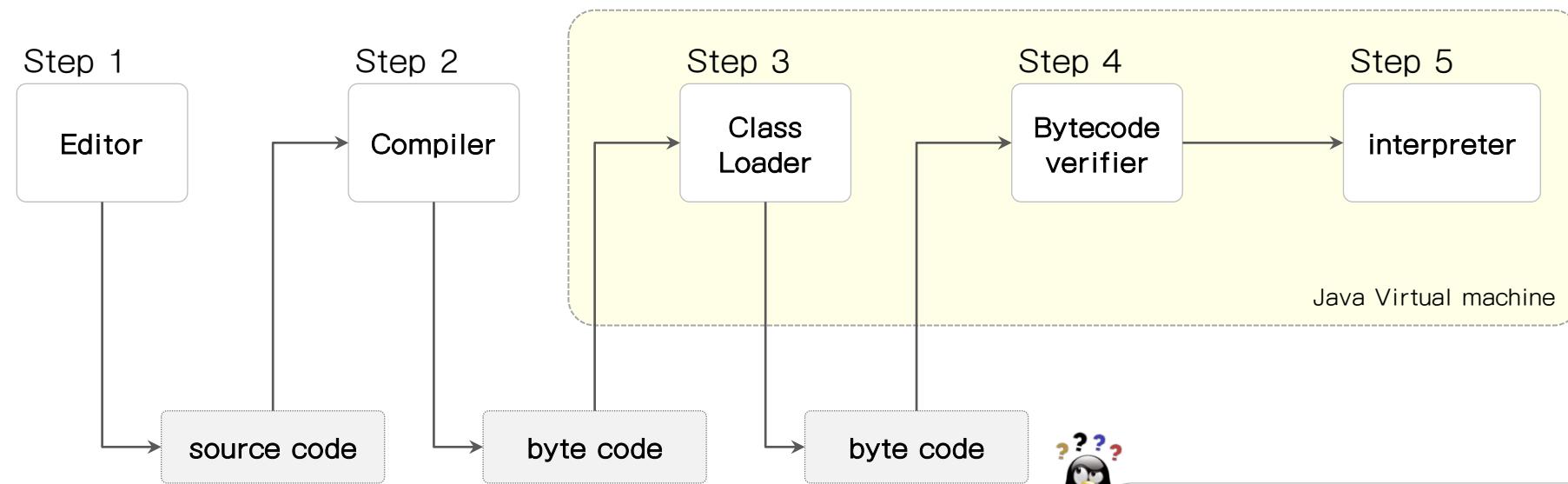
- 객체지향 언어(Object Oriented Programming)
- 멀티스레드 지원
- OS 플랫폼 독립성
- Garbage Collection

✓ 자바로 작성된 프로그램은 자바 가상 머신(Java Virtual Machine) 위에서 실행되며 이런 특징으로 인해 자바 프로그램은 어떤 운영체제(OS)에서도 실행할 수 있습니다. (write once, run anywhere)



1.3 Java Application 동작 방식

- ✓ 일반적인 프로그램은 소스코드를 컴파일하면 컴퓨터가 직접 해석 및 실행 할 수 있는 실행 파일이 만들어집니다.
- ✓ 작성된 코드를 번역하는 방식은 컴파일 방식과 인터프리터 방식이 있으며 자바 언어는 두 방식을 모두 수행합니다.
- ✓ 자바 언어로 작성한 코드는 자바 컴파일러(javac.exe)를 통해 컴파일을 수행하고 그 결과물은 *.class 파일이 됩니다.
- ✓ *.class 파일은 프로그램이 실행되면 JVM 내부에서 라인(line) 단위로 번역되어 실행됩니다.



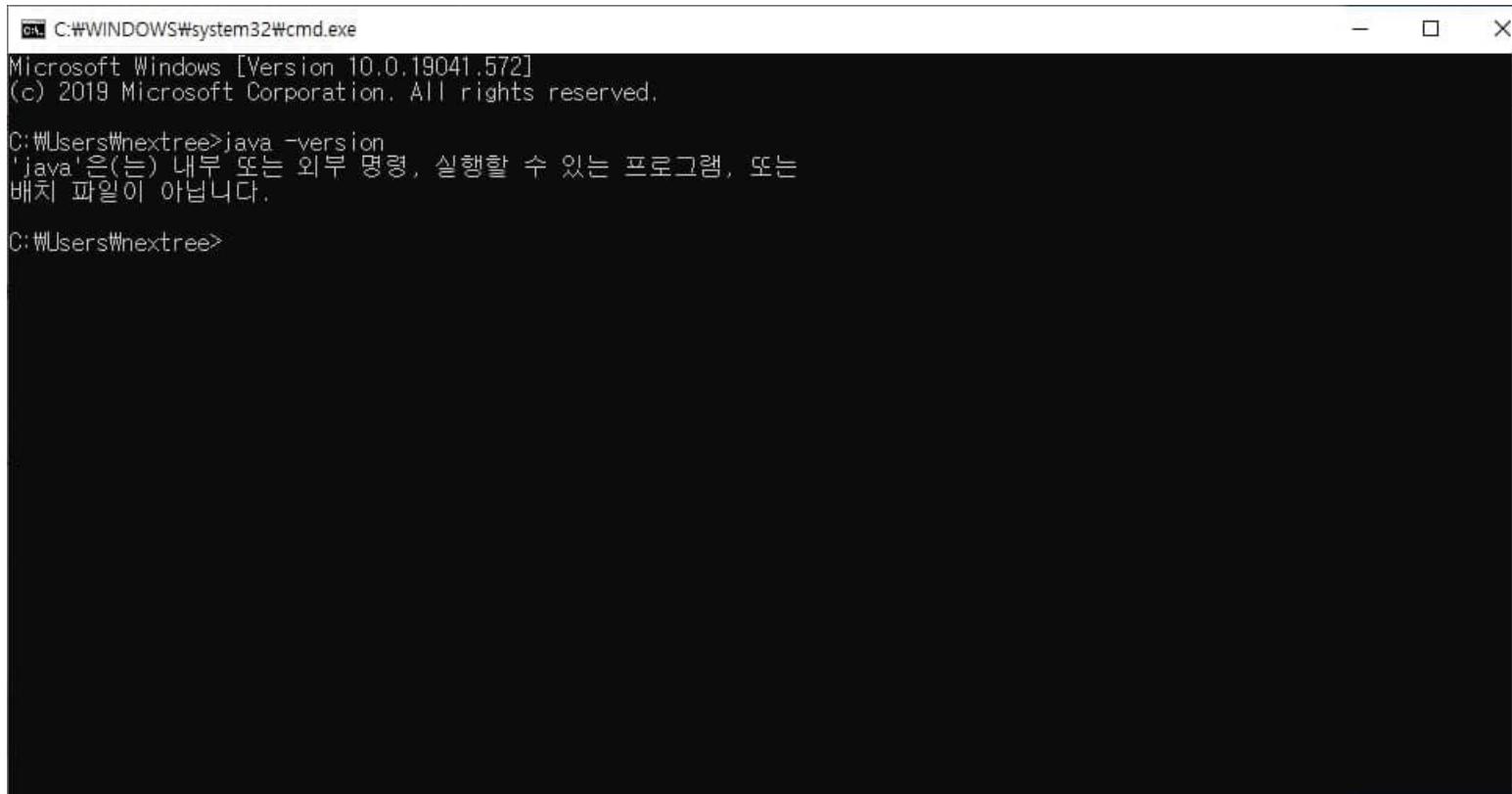
Byte Code 와 Native code의 차이는 무엇일까요?
컴퓨터는 0,1(2진수)만 읽을 수 있습니다. 2진수로 이루어진
코드를 보통 바이너리 코드(Binary code)라고하는데
자바에서는 이 바이너리 코드를 네이티브 코드(Native code)
라고 부릅니다. Byte code는 JVM이 읽고 번역하여
Native code로 변환하고 이렇게 변환된 코드를 컴퓨터가
읽고 실행하게 됩니다.

1.4 환경 구성 (1/5)

✓ 자바 개발 환경 구성을 위해 필요한 요소는 다음과 같습니다.

- JDK 11 : <https://www.oracle.com/java/technologies/javase-jdk11-downloads.html>
- IDE : IntelliJ(<https://www.jetbrains.com>) 또는 Eclipse(<https://www.eclipse.org/downloads/>)

✓ JDK 11 설치 과정 (Windows 10 기준)



A screenshot of a Windows Command Prompt window titled 'cmd C:\WINDOWS\system32\cmd.exe'. The window shows the following text:

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.19041.572]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\nextree>java -version
'java'은(는) 내부 또는 외부 명령, 실행할 수 있는 프로그램, 또는
배치 파일이 아닙니다.

C:\Users\nextree>
```

1.4 환경 구성 (2/5)

✓ JDK 11 다운로드 (URL: <https://www.oracle.com/kr/java/technologies/javase-jdk11-downloads.html>)

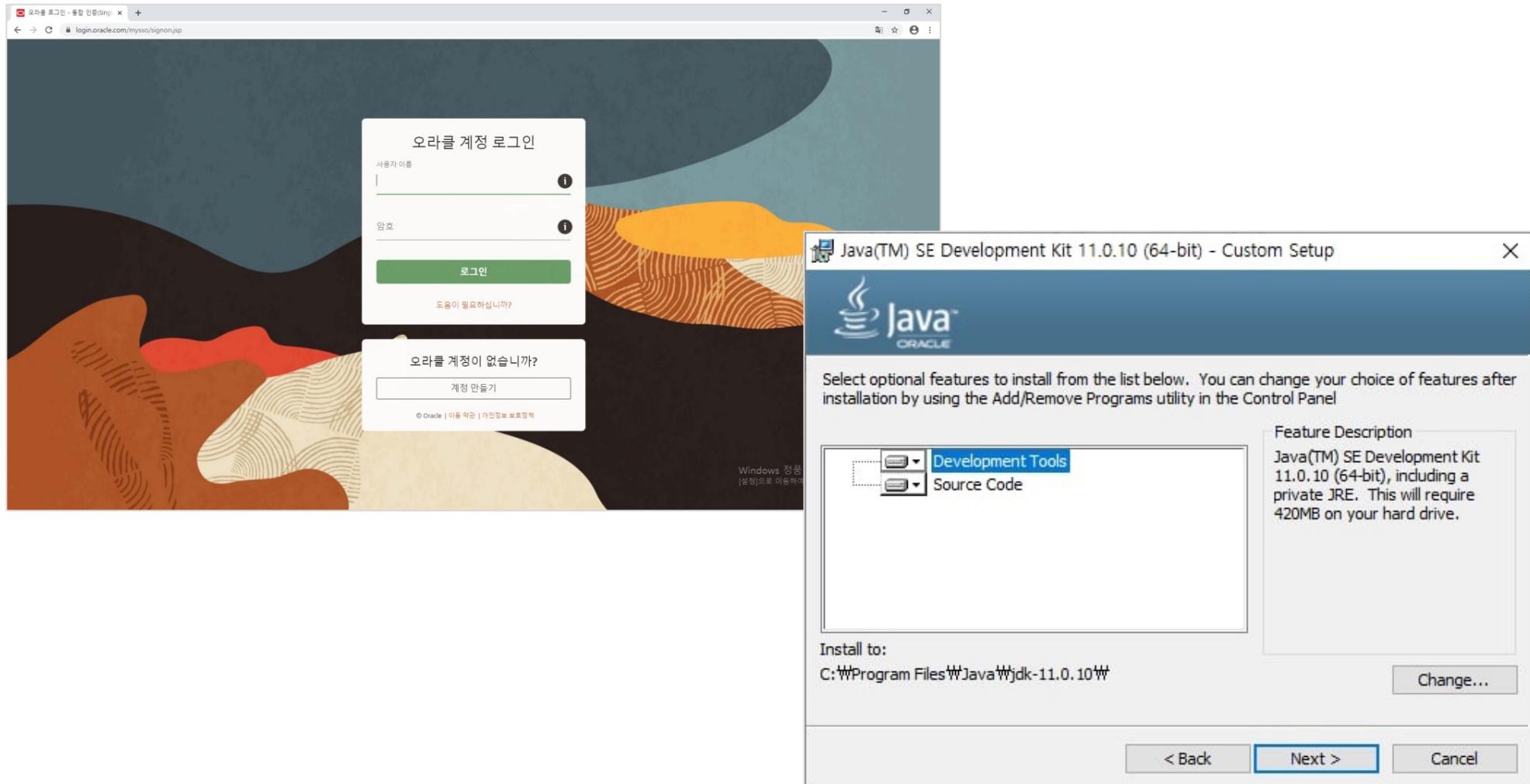
The screenshot shows a web browser displaying the Oracle Java SE Development Kit 11.0.10 download page. The page title is "Java SE Development Kit 11.0.10". It includes links for "Java Developer Day hands-on workshops (free) and other events", "Java Magazine", and "JDK 11.0.10 checksum". Below this, there is a section titled "Java SE Development Kit 11.0.10" with a note about the license. A table lists ten download options:

| Product / File Description | File Size | Download |
|----------------------------------|-----------|--|
| Linux ARM 64 Debian Package | 145.64 MB | jdk-11.0.10_linux-aarch64_bin.deb |
| Linux ARM 64 RPM Package | 152.22 MB | jdk-11.0.10_linux-aarch64_bin.rpm |
| Linux ARM 64 Compressed Archive | 169.37 MB | jdk-11.0.10_linux-aarch64_bin.tar.gz |
| Linux x64Debian Package | 149.39 MB | jdk-11.0.10_linux-x64_bin.deb |
| Linux x64 RPM Package | 156.12 MB | jdk-11.0.10_linux-x64_bin.rpm |
| Linux x64 Compressed Archive | 173.31 MB | jdk-11.0.10_linux-x64_bin.tar.gz |
| macOS Installer | 167.51 MB | jdk-11.0.10_osx-x64_bin.dmg |
| macOS Compressed Archive | 167.84 MB | jdk-11.0.10_osx-x64_bin.tar.gz |
| Solaris SPARC Compressed Archive | 184.82 MB | jdk-11.0.10_solaris-sparcv9_bin.tar.gz |
| Windows x64 Installer | 152.32 MB | jdk-11.0.10_windows-x64_bin.exe |
| Windows x64 Compressed Archive | 171.67 MB | jdk-11.0.10_windows-x64_bin.zip |

The "Windows x64 Installer" link is highlighted with a red rectangular box.

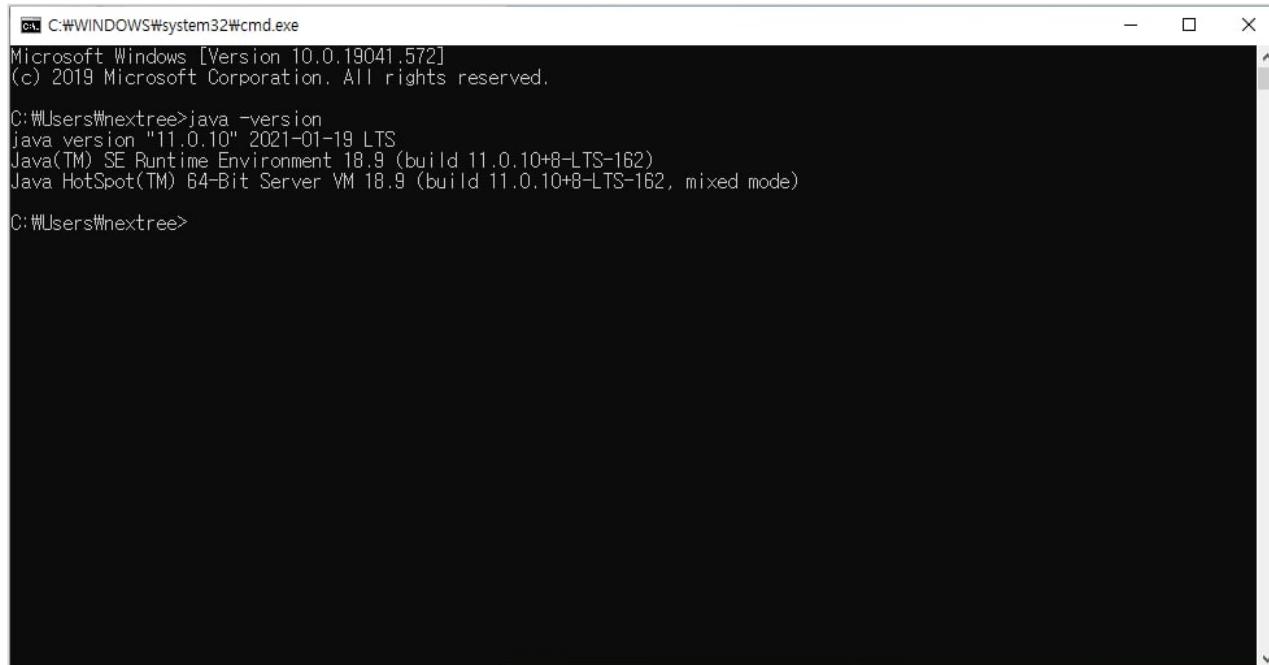
1.4 환경 구성 (3/5)

✓ 오라클 계정 확인 후 다운로드를 진행하며 다운로드가 완료된 이후에는 installer를 통해 설치를 진행합니다.



1.4 환경 구성 (4/5)

✓ JDK 설치 확인



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.19041.572]
(c) 2019 Microsoft Corporation. All rights reserved.

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

C:\Users\nextree>
```



환경변수 설정

특정 프로그램을 전역에서 실행 할 수 있도록 설정하는 것이 환경변수입니다. 기존에는 JDK 설치 이후에 Windows 고급 설정을 통해 환경변수 설정을 진행해야 했습니다.

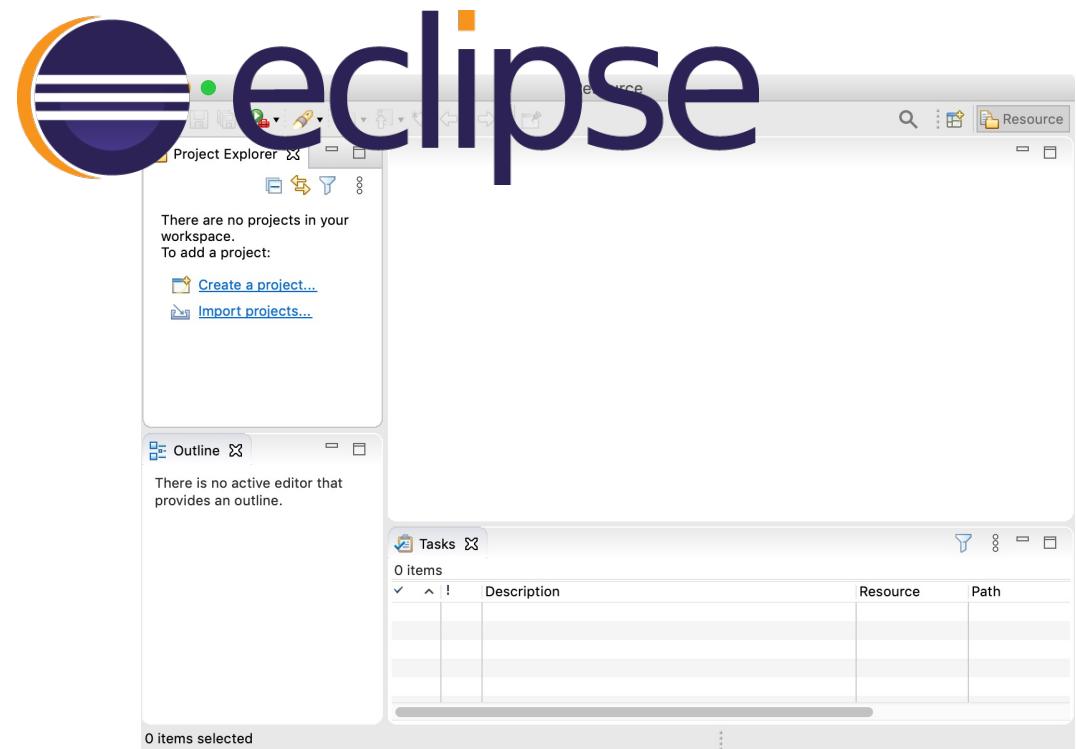
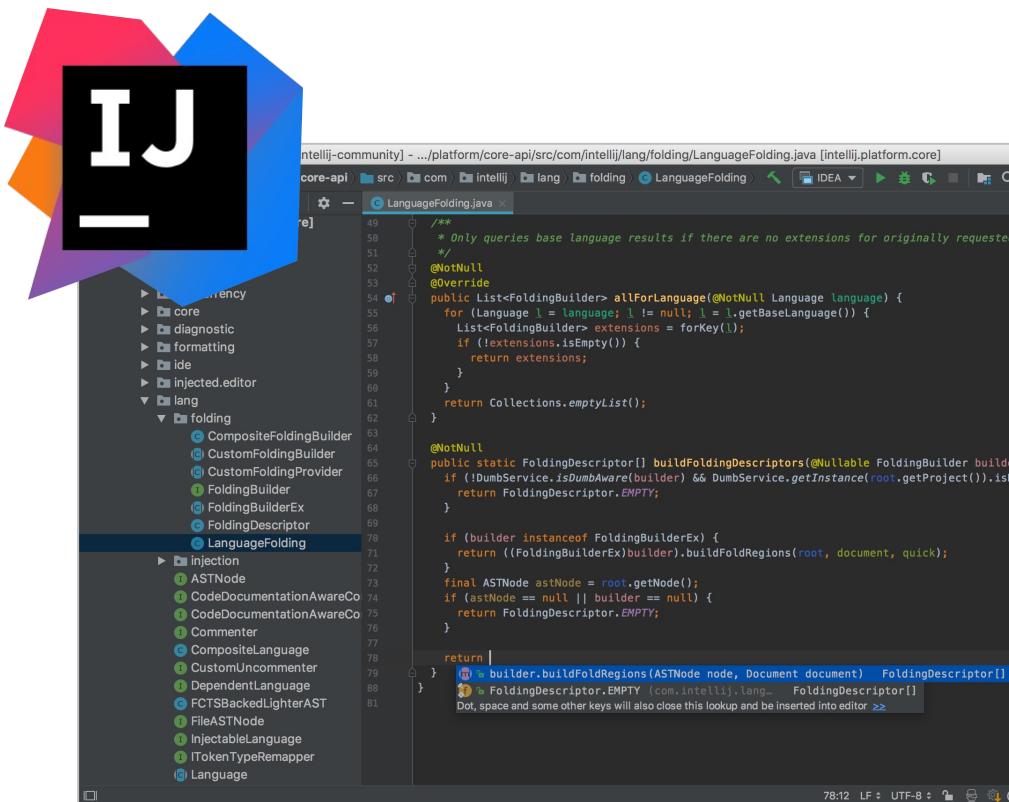
현재는 JDK가 설치되면 Program files > Common files > Oracle > Java > javapath 가 기본적으로 환경변수에 설정됩니다.

여기에는 javac.exe, java.exe, javaw.exe, jshell.exe 도구가 기본적으로 포함되어 있습니다.

[참고] 이 도구들 이외의 프로그램을 전역으로 사용하고 싶다면 별도의 환경변수 설정이 필요합니다.

1.4 환경 구성 (5/5)

- ✓ IDE는 프로그램의 작성과 컴파일 및 실행을 돋는 개발 도구입니다.
- ✓ 다양한 IDE가 존재하며 가장 대표적인 IDE는 intelliJ와 eclipse이며 익숙한 도구를 선택해 사용합니다.



1.5 첫번째 자바프로그래밍(1/3) – 초간단 자바 프로그램

- ✓ 자바 프로그램은 하나 이상의 클래스와 하나의 메인(main) 메소드로 이루어 집니다.
- ✓ 자바 프로그래밍은 편집 → 컴파일 → 실행 단계를 거쳐 완성합니다.
- ✓ .java는 텍스트 형식의 소스파일이며, 컴파일을 통해서 바이트코드(.class)로 변환합니다.
- ✓ java 명령어는 main() 메소드를 갖고 있는 클래스(바이트코드)를 실행합니다.

1) 소스코드 작성 (편집기)

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



소스파일 (.java)

HelloWorld.java

2) 컴파일 (컴파일러)

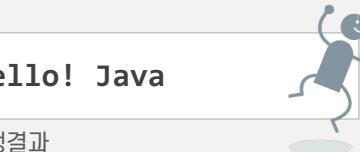
javac HelloWorld.java



바이트코드 (.class)

3) 실행 (Java 실행기)

java HelloWorld



실행결과

1.5 첫번째 자바프로그래밍(2/3) – 자바 프로그램 구조

- ✓ 파일 이름은 대소문자를 구분하며 클래스 이름과 동일해야합니다.
- ✓ 클래스에서 사용하는 다른 클래스는 import 문으로 추가합니다.
- ✓ 클래스 선언 다음에 있는 { 는 } 와 한 쌍입니다.
- ✓ 자바코드에서 하나의 문장이 끝날 때에는 세미콜론(;)을 적어줍니다.

Java 소스 파일

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

HelloWorld.java

Java 프로그램 구조

```
package 패키지 경로;  
  
import 패키지_경로1;  
import 패키지_경로2;  
import static 패키지_경로3;  
  
class 클래스명1 {  
    내용부;  
}  
  
public class 클래스명2 {  
    내용부;  
}
```

1.5 첫번째 자바프로그래밍(3/3) – main 메소드

- ✓ 자바 클래스의 **main()** 메소드는 java 명령어를 통해 처음 실행되는 메소드(method)입니다.
- ✓ 예를 들어, **java HelloWorld** 를 실행하면, **HelloWorld** 클래스의 **main** 메소드가 실행됩니다.
- ✓ **main()** 메소드는 객체를 생성하지 않고도, 외부에서도 접근할 수 있어야 합니다.
- ✓ **main()** 메소드는 반환 값이 없으며, java 명령어의 전달인자를 받는 매개변수(parameter)를 갖습니다.

static은 main 메소드가 객체 생성 없이도 정적으로 로드 될 수 있음을 나타냅니다.

main은 Java 프로그램의 시작점을 나타내는 메소드 이름입니다.

public은 외부에서도 접근할 수 있음을 탄내는 접근 제한자입니다. java 명령어가 main() 메소드를 실행하기 위해 해당 메소드에 접근 가능함을 명시합니다. public 이외의 접근 제한자를 사용하면 실행할 수 없습니다.

```
public class HelloWorld {  
    public static void main (String[] args) {  
        ...  
    }  
}
```

(**String[] args**) 는 메소드가 java 명령어를 통해 실행될 때 전달하는 인자들에 대한 매개변수를 나타냅니다. 전달인자가 여러 개일 수 있으므로 배열을 사용합니다.

void는 반환하는 값이 없음을 나타내는 키워드입니다. main 메소드는 실행 후 반환하는 값이 없습니다.

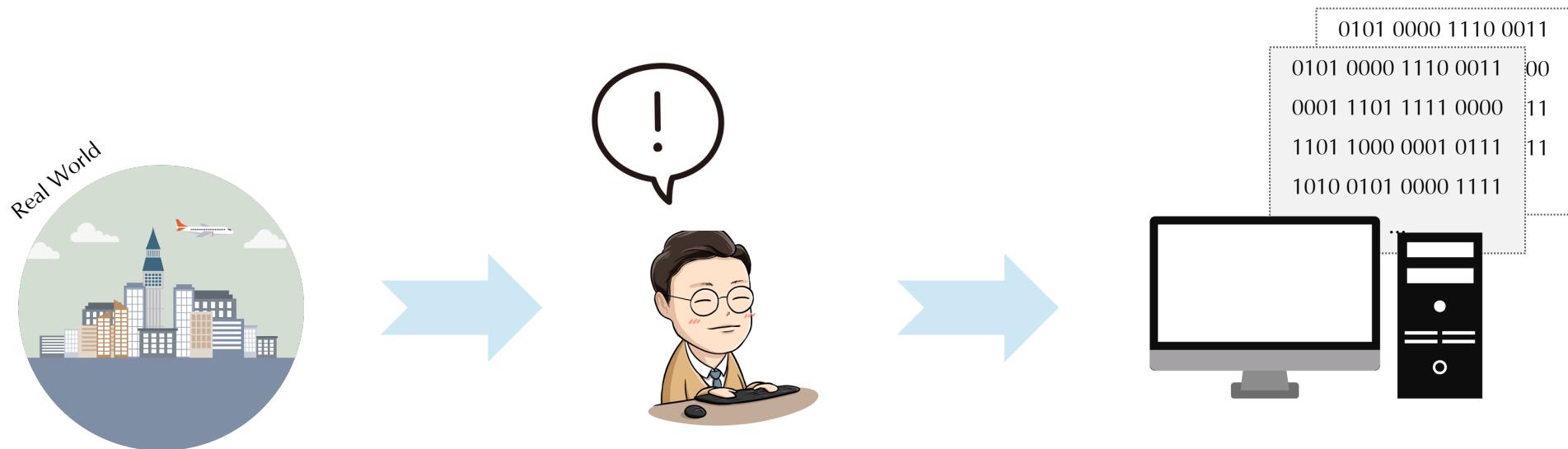


2. 클래스의 이해

-
- 2.1 객체지향 프로그래밍
 - 2.2 클래스 그리고 객체
 - 2.3 클래스의 구성요소
 - 2.4 필드(Field)의 정의
 - 2.5 연산자(Operators)의 이해
 - 2.6 메소드(Method) 정의
 - 2.7 메소드 호출의 이해
 - 2.8 생성자(Constructor) 정의
 - 2.9 생성자 호출의 이해
 - 2.10 Java 메모리 모델
 - 2.11 static과 final

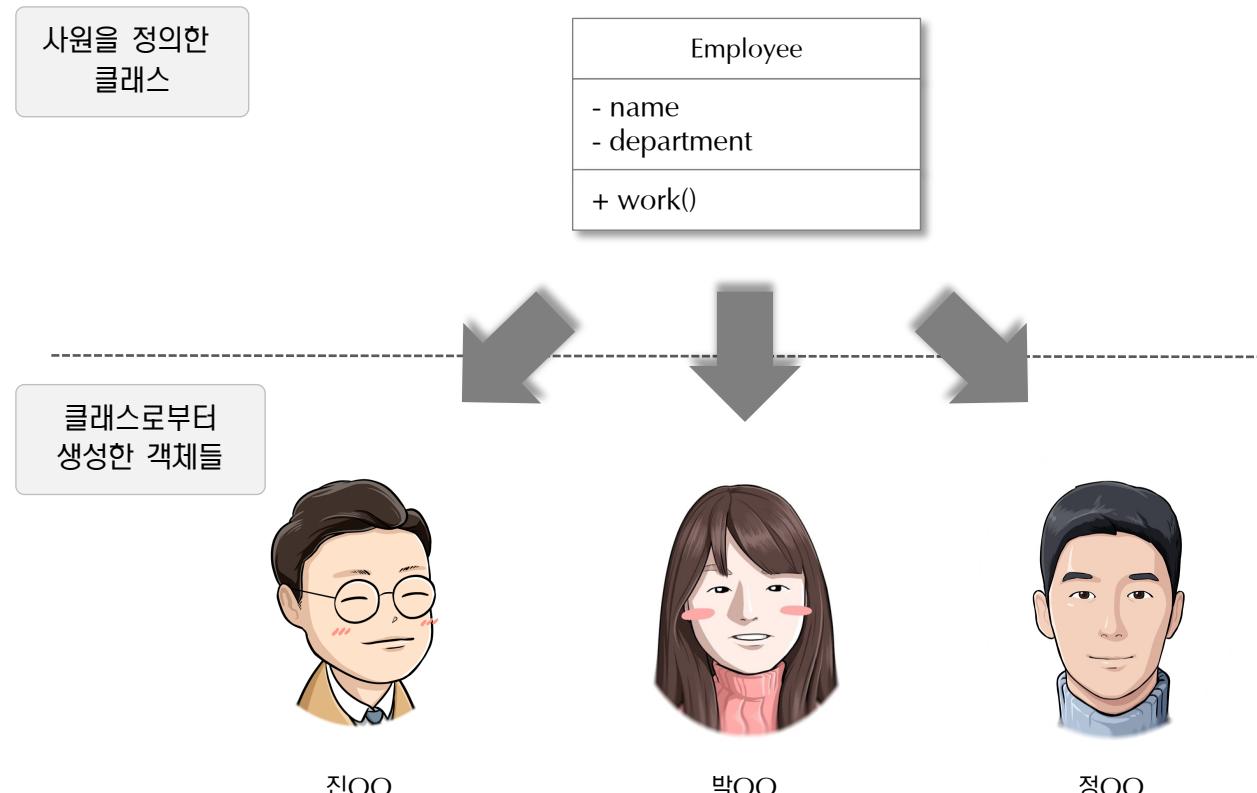
2.1 객체지향 프로그래밍

- ✓ 프로그래밍 언어의 패러다임을 이해하는 과정은 해당 프로그램 언어를 익히고 사용하는데 매우 중요한 절차입니다.
- ✓ 프로그래밍은 현실 세계의 특정 문제를 컴퓨터 세계를 통해 풀어가는 방법이라고 할 수 있습니다.
- ✓ 현실 세계의 문제를 기능 또는 구조위주의 관점으로 보고, 기능을 세분화하여 풀어가는 것을 절차지향이라고 합니다.
- ✓ 객체지향은 문제를 데이터의 관점으로 보며, 데이터들의 상호 관계를 정의함으로써 해결책을 찾아갑니다.



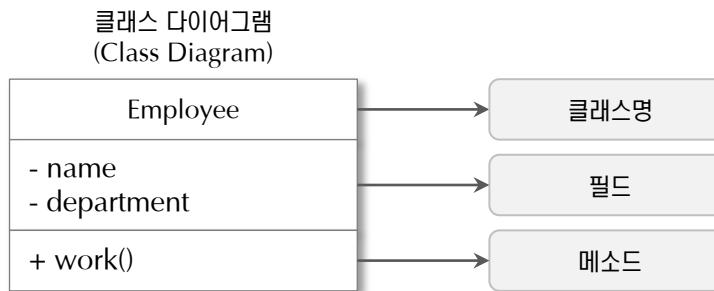
2.2 클래스(Class) 그리고 객체(Object)

- ✓ 자바 언어로 구현하는 프로그램은 다수의 클래스들로 이루어지며 이 클래스를 이용해 객체를 만들고 사용합니다.
- ✓ 클래스를 정의한다는 것은 객체를 만들기 위한 과정이며 클래스는 객체에 대한 청사진 또는 템플릿이라 할 수 있습니다.
- ✓ 클래스로부터 만들어지는 객체를 인스턴스(instance) 혹은 인스턴스 객체(instance object)라고 합니다.



2.3 클래스의 구성요소

- ✓ 자바 프로그램의 기본인 클래스를 잘 정의하기 위해서는 클래스를 구성하는 구성요소에 대한 이해가 필요합니다.
- ✓ 클래스는 상태와 행위를 가지며 상태를 필드(Field), 행위를 메소드(Method)라고 합니다.
- ✓ 클래스에는 필드와 메소드 외에 생성자(Constructor)라는 특수한 메소드도 반드시 하나 이상 갖습니다.



Employee Class

```
public class Employee {  
  
    private String name;  
    private String department;  
  
    public Employee(String name, String department){  
        this.name = name;  
        this.department = department;  
    }  
  
    public void work(){  
        ...  
    }  
}
```

Fields

Constructor

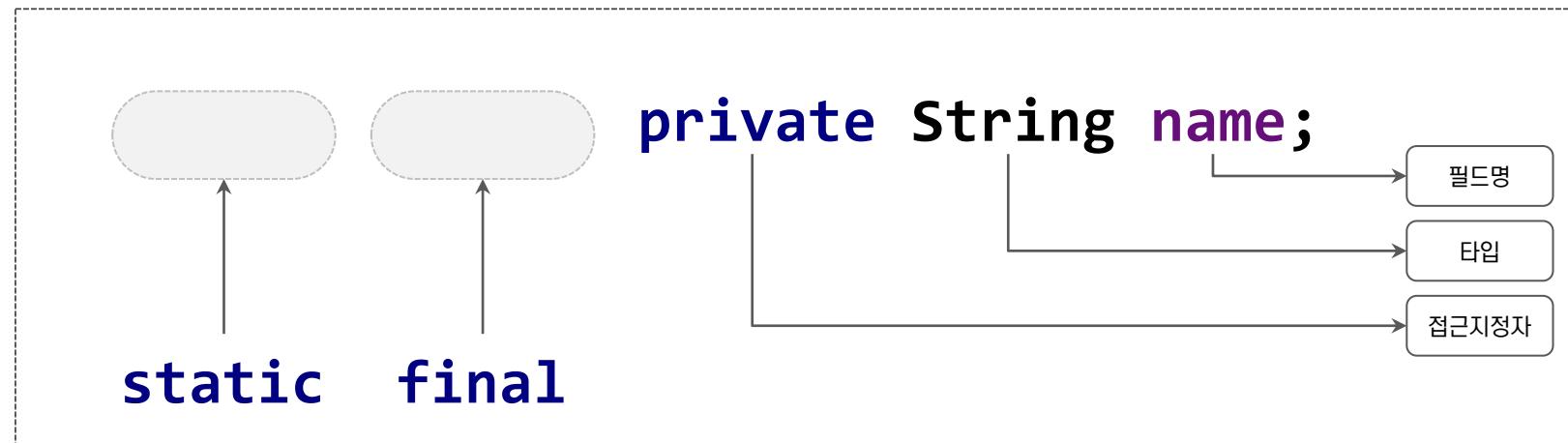
Method

This section displays the generated Java code for the Employee class. It highlights the fields ('name', 'department') with a red dashed box and the constructor ('Employee') with another red dashed box. The code is annotated with labels: 'Fields' next to the field declarations, 'Constructor' next to the constructor definition, and 'Method' next to the method declaration.

2.4 필드(Field)의 정의 (1/10) – 개요

- ✓ 클래스에 정의하는 속성은 특정한 값을 가지며, 객체의 속성 값은 해당 객체의 상태를 표현합니다.
- ✓ 현실 세계의 객체는 다양하고 광범위한 속성들을 갖고 있기 때문에 정의하고자 하는 클래스의 특성을 잘 이해하고 해당 클래스의 핵심 속성들을 정의합니다.
- ✓ 필드를 정의할 때는 반드시 접근지정자(Access Modifier), 타입, 필드명을 명시합니다.

class



2.4 필드(Field)의 정의 (2/10) – 변수(Variable)의 이해

✓ 변수란 데이터를 담는 그릇이며, 데이터의 저장과 참조를 위해 메모리 공간을 할당 받습니다.

- 변수는 자료형과 변수의 이름을 사용하여 선언합니다.

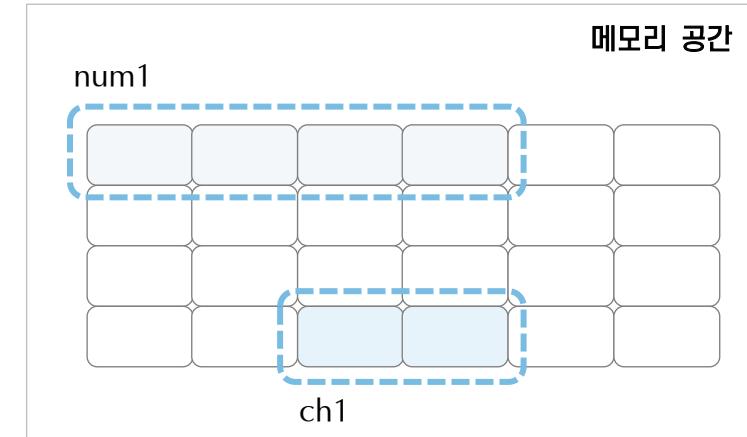
```
int num1;  
char ch1;
```

✓ 변수 이름

- 첫 글자는 문자만 올 수 있으며, 이어서 문자 또는 숫자가 올 수 있습니다.
- 문자는 대소문자를 구분하며, 유니코드로 정의된 어떠한 문자도 사용할 수 있습니다.
- 특수문자는 \$와 _만 쓸 수 있습니다.
단, \$는 컴파일러가 생성하므로 사용하지 않습니다.
- 자바의 예약어는 사용할 수 없습니다.
예: boolean, class, switch, return 등

✓ 변수 초기화

- 선언한 변수에 처음으로 값을 할당하는 것을 변수 초기화라고 합니다.
- 선언과 동시에 변수를 초기화할 수 있습니다.
- 초기화하지 않은 변수를 사용하면 자바 컴파일러는 에러를 발생시킵니다.



```
int num1;  
num1 = 10;  
System.out.println(num1);
```

```
int num1 = 10;  
System.out.println(num1);
```

```
int num1;  
System.out.println(num1); //Compile Error
```

2.4 필드(Field)의 정의 (3/10) – 변수의 유형

✓ 변수는 정의된 위치에 따라 4가지의 유형으로 구분하며 각 유형에 따라 갖는 특성에 차이가 있습니다.

- 지역변수(Local variables)
- 매개변수(Parameter variables)
- 인스턴스 변수(Instance variables)
- 정적 변수(Class variables)

```
public class VariableTypes {

    /** 정적 변수 */
    public static int classVar = 1;

    /** 인스턴스 변수 */
    private int instanceVar;

    /**
     * @param paramVar 매개변수
     */
    public static void main(String[] paramVar) {

        // 지역변수
        int localVal = 10;
    }
}
```

2.4 필드(Field)의 정의 (4/10) – 자료형(Data Type)의 이해

✓ 자료형(타입)은 변수가 가지는 자료의 종류입니다.

- 자바는 자료형 검사가 엄격한 언어이므로, 모든 변수는 자료형을 갖습니다.
- 자료형에 따라 할당하는 메모리 크기가 달라집니다.

```
int count = 10;
char ch = 'Y';
double reality = 99.9;
boolean pass = true;
```

✓ 자바의 기본 자료형 (Primitive type)

- 정수형 : int, short, long, byte
- 실수형 : float, double
- 문자형 : char
- 부울형 : boolean

2.4 필드(Field)의 정의 (5/10) – 정수 자료형 (Integer Type)

✓ 정수형은 소수부가 없는 숫자를 나타내는 자료형입니다.

- 자바는 데이터의 표현범위에 따라 4가지의 정수형을 제공합니다.

| 자료형 | 메모리 크기 | 표현범위 | 표현 범위(지수표현) |
|-------|--------|--|---------------------------|
| byte | 1 바이트 | -128 ~ 127 | $-2^7 \sim 2^7 - 1$ |
| short | 2 바이트 | -32,768 ~ 32,767 | $-2^{15} \sim 2^{15} - 1$ |
| int | 4 바이트 | -2,147,483,648 ~ 2,147,483,647 | $-2^{31} \sim 2^{31} - 1$ |
| long | 8 바이트 | -9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807 | $-2^{63} \sim 2^{63} - 1$ |

✓ 정수 표현

- 정수 리터럴은 기본적으로 int 형을 표현하므로, long 타입을 명시하기 위해서는 접미사 L을 사용합니다.
- 필요에 따라 접두사(prefix)를 사용하여 8진수(0), 16진수(0x), 2진수(0b)를 표현합니다.
- Java 7부터는 정수 리터럴에 밑줄(_) 기호를 사용하여 자릿수를 구분할 수 있습니다. (컴파일러는 밑줄을 무시 합니다.)

```
int num = 10;
int num1 = 010;      // 8을 나타내는 8진수 표현
int num2 = 0x11;    // 17을 나타내는 16진수 표현
int num3 = 0b0011;  // 3을 나타내는 2진수 표현
```

```
// Long 값을 나타내는 접미사 L
long longNumber = 123_123_123_123L;
// Java 7부터 제공되는 UnderScore 표기법
int numberUsingUnderScore = 1_234_567_000;
int alsoUsedForBinaryLiteral = 0b1111_0000_1010_0000;
```

2.4 필드(Field)의 정의 (6/10) – 실수 자료형 (Floating-Point Type)

✓ 실수형은 소수부를 가진 숫자를 표현하는 자료형입니다.

- 자바는 데이터의 표현범위에 따라 2가지의 실수형을 제공합니다.

| 자료형 | 메모리 크기 | 표현범위 |
|--------|--------|---|
| float | 4 바이트 | 대략 $\pm 3.40282347E+38F$ (소수점 이하 7자리의 정밀도를 가짐) |
| double | 8 바이트 | 대략 $\pm 1.79769313486231570E+308$ (소수점 이하 15자리의 정밀도를 가짐) |

✓ 실수의 표현

- 실수 리터럴은 기본적으로 double 형을 표현하므로 float 타입 값을 표현하려면 접미사 F를 사용해야 합니다.
- double을 나타내는 접미사는 D이나 생략할 수 있습니다.

```
float f1 = 123.123F; // float 타입을 표현하는 접미사 F
double d1 = 123.123;
```

2.4 필드(Field)의 정의 (7/10) – 문자 자료형 (Character Type)

- ✓ 문자형은 개별 문자를 나타내는 자료형입니다.
- ✓ Java는 유니코드라는 표준을 사용하여 2바이트로 문자를 표현합니다. (www.unicode.org 참고)
- ✓ 유니코드의 문자체계는 2바이트(16비트)이므로 유니코드 표현법과 16진수 표기법은 같습니다.

J 한 A さ

```
char ch1 = 'J';  
  
char han = '한';  
System.out.println(han);  
  
char english = 0x0041; // A  
System.out.println(english);  
  
char hangul = 0xAC00; // 가  
System.out.println(hangul);  
  
char japanese = 0x3055; // さ  
System.out.println(japanese);
```

| | 000 | 001 | 002 | 003 | 004 | 005 | 006 | 007 |
|---|------|-----|-----|-----|-----|-----|-----|-----|
| 0 | HU: | DLE | SP | 0 | @ | P | ` | p |
| 1 | SOH: | DC1 | ! | 1 | A | Q | a | q |
| 2 | STX: | DC2 | " | 2 | B | R | b | r |
| 3 | ETX: | DC3 | # | 3 | C | S | c | s |
| 4 | EOT: | DC4 | \$ | 4 | D | T | d | t |
| 5 | ENO: | NAK | % | 5 | E | U | e | u |

| | AC0 | AC1 | AC2 | AC3 | AC4 | AC5 | AC6 | AC7 |
|---|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 가 | 감 | 갠 | 겠 | 갈 | 깥 | 깽 | 거 |
| 1 | 각 | 갑 | 깱 | 깱 | 깱 | 깱 | 깱 | 깱 |
| 2 | 깎 | 깎 | 깎 | 깎 | 깎 | 깎 | 깎 | 깎 |
| 3 | 깟 | 깟 | 깟 | 깟 | 깟 | 깟 | 깟 | 깟 |
| 4 | 간 | 잤 | 깰 | 깰 | 꺌 | 꺌 | 꺌 | 꺌 |

| | 304 | 305 | 306 | 307 | 308 | 309 |
|---|-----|-----|-----|-----|-----|-----|
| 0 | ぐ | だ | ば | む | ゐ | |
| 1 | あ | け | ち | ぱ | め | ゑ |
| 2 | あ | げ | ぢ | ひ | も | を |
| 3 | い | こ | つ | び | や | ん |

출처 : 유니코드(<http://www.unicode.org/charts/PDF/UAC00.pdf>)

2.4 필드(Field)의 정의 (8/10) – 논리 자료형 (Boolean Type)

✓ 논리 자료형은 참과 거짓을 표현하는 자료형입니다.

- true : 참을 표현하는 값
- false : 거짓을 표현하는 값

```
public class BooleanType {  
    public static void main(String[] args) {  
  
        boolean.isTrue = true;  
        boolean.isFalse = false;  
  
        System.out.println(isTrue);  
        System.out.println(isFalse);  
  
    }  
}
```

2.4 필드(Field)의 정의 (9/10) – 상수 (Constants)

- ✓ 상수는 값이 변하지 않는 수를 의미하며 자바에서는 두가지 방식으로 상수를 구분합니다.
 - 리터럴 상수(Literals Constants) : 155, -5 'a', "Java Programming"
 - 사용자 정의 상수(User Define Constants) : final double PI = 3.141592;
- ✓ 리터럴 상수도 그 타입에 따라 기본 타입이 정해져 있습니다.
 - 155, -5와 같은 정수형 타입은 int type 입니다.
 - 0.5, 3.14와 같은 실수 타입은 double type 입니다.
- ✓ 사용자 정의 상수는 변수를 선언하고 여기에 final 키워드를 붙이면 한번 초기화 후 그 값을 변경할 수 없습니다.

```
final double PI = 3.14;  
System.out.println("PI is " + PI);
```

2.4 필드(Field)의 정의 (10/10) – 참조 자료형 (Reference Type)

- ✓ 참조 자료형을 갖는 변수는 특정 객체의 참조 정보를 저장합니다.
- ✓ 기본 데이터 타입(byte, short, int, long, float, double 등) 이외의 타입을 의미합니다.
- ✓ 참조 자료형 변수는 4byte의 크기를 갖으며 인스턴스 객체에 접근할 수 있는 정보를 갖습니다.
- ✓ 객체의 삭제는 객체가 더 이상 사용되지 않을 때 자바의 Garbage Collector에 의해 자동적으로 제거됩니다.



2.5 연산자(Operators)의 이해 (1/9)

- ✓ 연산자는 피연산자(operand)를 대상으로 특정 기능을 수행하고 결과를 반환하는 특수한 기호입니다.
- ✓ 연산자의 구분은 피연산자의 수에 따라 단항, 이항, 삼항으로 구분하거나 기능의 종류에 따라 산술, 관계, 비트, 논리, 대입, 기타 연산자로 구분합니다.

| 연산자 종류 | 설명 |
|--|---|
| 산술연산자 (+, -, *, /, %) | 피 연산자가 두 개인 대표적인 이항 연산자로 사칙연산 및 나머지(%) 연산을 수행합니다. |
| 대입연산자 (=) | 산술연산의 결과를 변수에 할당합니다. |
| 관계연산자 (<, >, <=, >=, =, !=) | 피 연산자의 크기 및 동등 관계를 비교하는 이항 연산자로, 비교연산자라고도 합니다. 연산의 결과가 모두 boolean 형입니다. |
| 논리연산자 (&&, , !) | 논리 연산을 수행하는 연산자로 이항 연산자인 AND(논리곱, &&), OR(논리합,) 연산자와 단항 연산자인 NOT(논리부정, !)연산자가 있습니다. 피 연산자와 연산의 결과가 모두 boolean 형입니다. |
| 부호연산자 (+, -) | 양수(+) 및 음수(-)로 부호를 바꾸어 주는 단항 연산자입니다. |
| 증가/감소 연산자 (++, --) | 변수에 저장된 값을 증가(++)시키거나 또는 감소(--)시키는 단항 연산자입니다. 연산자의 위치에 따라 연산이 수행되는 시점이 다르게 적용됩니다. |
| 비트연산자 (&, , ^, ~) | 비트 단위의 연산을 수행하는 연산자로 이항 연산자인 비트 AND(&), 비트 OR(), 비트 XOR(^)연산자와 단항 연산자인 비트 NOT(~) 연산자가 있습니다. |
| 비트 쉬프트(Shift) 연산자 (<<, >>, >>>) | 피 연산자의 비트 열을 왼쪽 또는 오른쪽으로 이동시키는 연산자입니다. |
| 기타연산자 ((), [], .(dot operator), ?:) | () - 특정 연산자들을 묶어서 먼저 처리할 수 있도록 만들어 주는 연산자입니다. [] - 자료형이나 클래스와 함께 사용되어 배열로 선언되었음을 알리는 연산자입니다. . - 특정 범위(클래스) 내에 속해 있는 멤버를 지칭하는 연산자입니다. |

2.5 연산자(Operators)의 이해 (2/9) – 대입연산자

- ✓ 대입 연산자는 이항 연산자로 = 기호를 기준으로 오른쪽 값을 왼쪽 기억 공간(변수)에 저장하는 기능을 합니다.

L value **=** **R value**

- ✓ 대입 연산자는 산술 연산자와 함께 복합 연산자 형태로 사용할 수 있습니다.

| 연 산 자 | 예 | 의 미 |
|-----------------|----------------------|-------------------------|
| <code>+=</code> | <code>x += 10</code> | <code>x = x + 10</code> |
| <code>--</code> | <code>x --</code> | <code>x = x - 10</code> |
| <code>*=</code> | <code>x *= 10</code> | <code>x = x * 10</code> |
| <code>/=</code> | <code>x /= 10</code> | <code>x = x / 10</code> |
| <code>%=</code> | <code>x %= 10</code> | <code>x = x % 10</code> |

2.5 연산자(Operators)의 이해 (3/9) – 산술연산자, 관계연산자

| 연산자 | 연산식 | 의미 |
|-----|------|-----------------------------------|
| + | 10+5 | 10과 5를 더한 15 답이다 |
| - | 10-5 | 10에서 5의 차이인 5이다 |
| * | 10*5 | 10*5는 두 피연산자의 곱셈이므로 50이 답이다 |
| / | 10/5 | 10/5는 두 피연산자의 나눗셈이므로 2 답이다 |
| % | 10%5 | 10%5는 10을 5로 나누었을 때 나머지이므로 0이 답이다 |

| 연산자 | 예 | 의미 | 결과 |
|-----|--------|----------------|-----------------------|
| = | a = b | a와 b가 같다. | 같으면 1 다르면 0 |
| != | a != b | a와 b가 같지 않다. | 같지 않으면 1 같으면 0 |
| > | a > b | a가 b보다 크다 | a가 더크면 1 아니면 0 |
| >= | a >= b | a가 b보다 크거나 같다. | a가 크거나 같으면 1 아니면 0 |
| < | a < b | a가 b보다 작다 | a가 b보다 작으면 1 아니면 0 |
| <= | a <= b | a가 b보다 작거나 같다. | a가 b와 같거나 작으면 1 아니면 0 |

2.5 연산자(Operators)의 이해 (4/9) – 증감 연산자

- ✓ 변수에 저장된 값을 증가(++)시키거나 또는 감소(--)시키는 단항 연산자입니다.
- ✓ 연산자를 정의하는 위치에 따라 연산이 수행되는 시점이 다르게 적용됩니다.

| 종류 | 예 | 의 미 |
|-----|--------------|-----------------------|
| 전위형 | $\text{++}a$ | $a = a+1$ 또는 $a += 1$ |
| | $\text{--}a$ | $a = a-1$ 또는 $a -= 1$ |
| 후위형 | $a\text{++}$ | $a = a+1$ 또는 $a += 1$ |
| | $a\text{--}$ | $a = a-1$ 또는 $a -= 1$ |

```
int x = 1;  
int y = x++;  
  
y = 1
```

1
2
 $y = x++;$

```
int x = 1;  
int y = ++x;  
  
y = 2
```

2
1
 $y = ++x;$

2.5 연산자(Operators)의 이해 (5/9) – 논리 연산자

- ✓ 논리 연산을 수행하는 연산자로 이항 연산자인 AND(논리곱, `&&`), OR(논리합, `||`) 연산자와 단항 연산자인 NOT(논리부정, `!`)연산자가 있습니다. ⚡ 연산자와 연산의 결과가 모두 boolean 형입니다.
- ✓ 우선 순위는 NOT > AND > OR 입니다.

| x | y | <code>x && y</code> | 의미 |
|---|---|-----------------------------|----|
| 0 | 0 | 0 | 거짓 |
| 0 | 1 | 0 | 거짓 |
| 1 | 0 | 0 | 거짓 |
| 1 | 1 | 1 | 참 |

| x | <code>!x</code> | 의미 |
|---|-----------------|----|
| 0 | 1 | 참 |
| 1 | 0 | 거짓 |

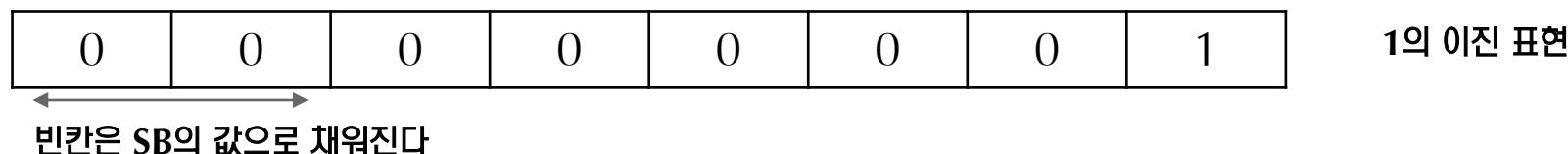
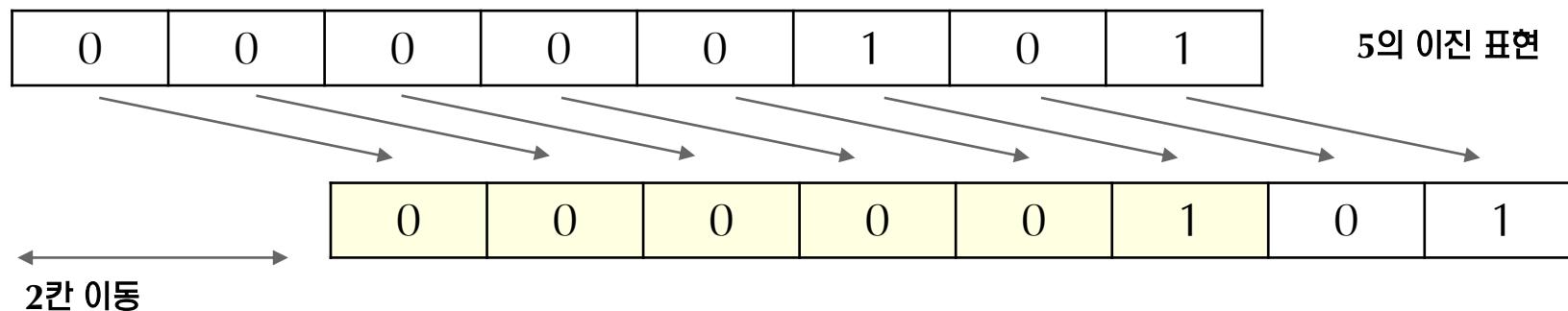
| x | y | <code>x y</code> | 의미 |
|---|---|---------------------|----|
| 0 | 0 | 0 | 거짓 |
| 0 | 1 | 1 | 참 |
| 1 | 0 | 1 | 참 |
| 1 | 1 | 1 | 참 |

| x | y | <code>x ^ y</code> | 의미 |
|---|---|--------------------|----|
| 0 | 0 | 0 | 거짓 |
| 0 | 1 | 1 | 참 |
| 1 | 0 | 1 | 참 |
| 1 | 1 | 0 | 거짓 |

2.5 연산자(Operators)의 이해 (6/9) – 비트 연산자(Shift)

- ✓ 정수형 피연산자에 대해 비트를 왼쪽, 혹은 오른쪽으로 이동시키는 연산자입니다.

| 연산자 | 뜻 | 예 | 의미 |
|-----|----------|---------|----------------------------------|
| >> | 오른쪽으로 이동 | 5 >> 2 | 5의 이진수표현에서 오른쪽으로 2칸 쉬프트 |
| << | 왼쪽으로 이동 | 5 << 2 | 5의 이진수표현에서 왼쪽으로 2칸 쉬프트 |
| >>> | 오른쪽으로 이동 | 5 >>> 2 | 5의 이진수표현에서 오른쪽으로 2칸 쉬프트(0으로 채워짐) |



2.5 연산자(Operators)의 이해 (7/9) – 삼항 연산자

- ✓ 삼항 연산자는 피연산자를 3개 갖는 연산자입니다.
- ✓ 조건의 참, 거짓 여부에 따라 서로 다른 수식을 실행합니다.

조건 ? 수식1 : 수식2

```
int x, y, z;  
  
x = 10; y = 20;  
z = ( x > y ? x : y);  
System.out.println(z); //z : 20  
  
z = ( y > x ? x : y);  
System.out.println(z); // z : 10
```

2.5 연산자(Operators)의 이해 (8/9) – 연산자 우선순위

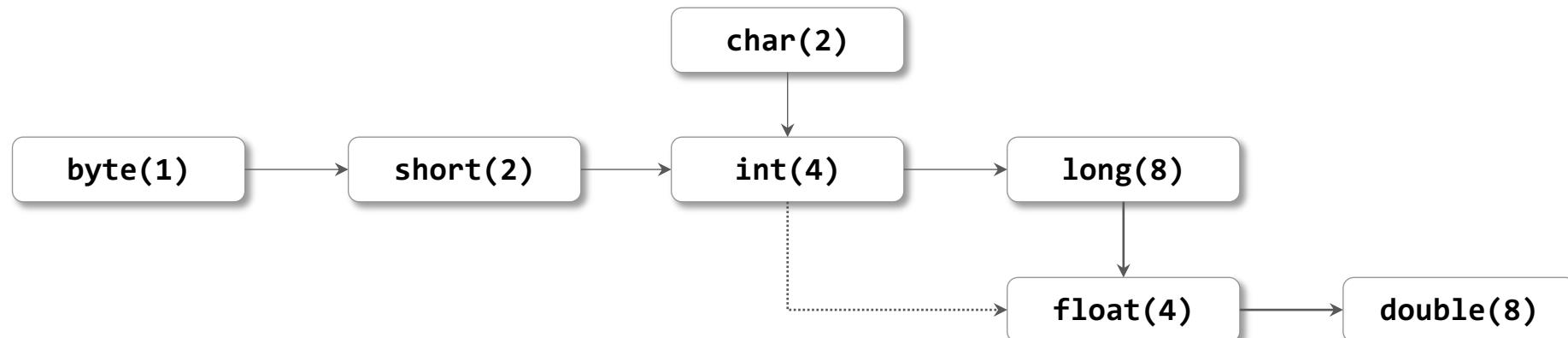
- ✓ 하나의 실행문에 다수의 연산자가 함께 사용될 경우 어떤 연산을 먼저 수행할지를 결정하는 것이 연산자 우선순위입니다.
- ✓ 모든 연산자의 우선순위를 고려하여 사용하기 보다는 소괄호() 연산자를 이용해 우선순위를 명확히 합니다.

| 연산자 | 결합방향 | 우선순위 |
|--|------|--------|
| () , [] , .(dot operator) | → | 1(높음) |
| expr++, expr-- | ← | 2 |
| ++expr, --expr, +expr, -expr, ~, !, (type) | ← | 3 |
| *, /, % | → | 4 |
| +, - | → | 5 |
| <<, >>, >>> | → | 6 |
| <, >, <=, >=, instanceof | → | 7 |
| ==, != | → | 8 |
| & | → | 9 |
| ^ | → | 10 |
| | → | 11 |
| && | → | 12 |
| | → | 13 |
| condition ? expr : expr | → | 14 |
| =, +=, -=, *=, /=, %=, &=, ^=, =, <=>, >>>= | ← | 15(낮음) |

2.5 연산자(Operators)의 이해 (9/9) – 형변환(casting) 연산자

✓ 암묵적 형 변환 (Implicit Conversion)

- Java는 데이터가 손실되지 않거나, 손실이 제한적인 범위 내에서 암묵적으로 형을 변환합니다.
- 자료형이 다른 변수에 값을 할당하는 경우 (float f = 10;)
- 자료형이 다른 값을 연산하는 경우 (double d = 10.0 + 20;)



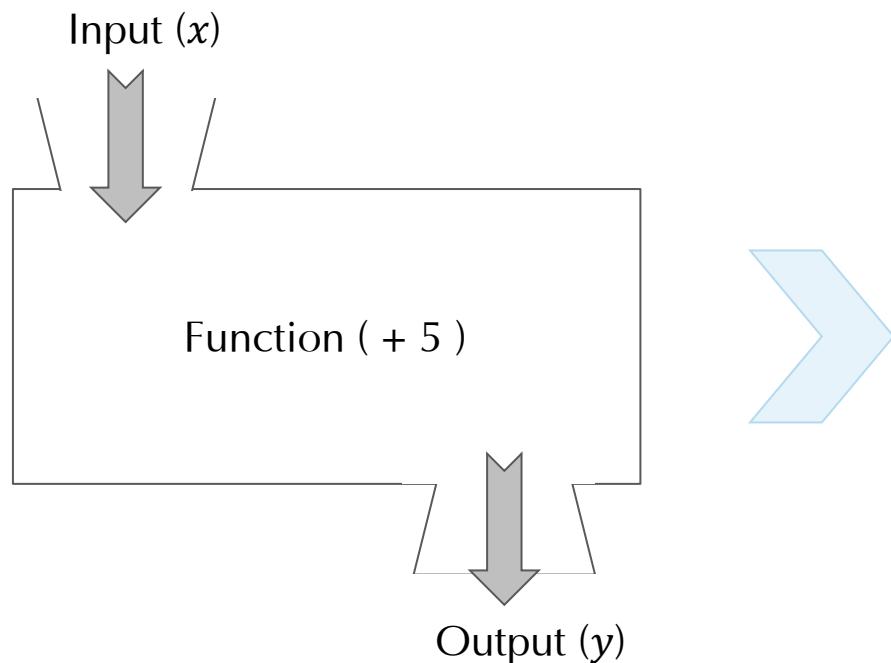
✓ 명시적 형 변환 (Explicit Conversion)

- 명시적으로 형 변환하는 것을 캐스팅(Casting)이라고 합니다.
- 자동 형 변환 규칙에 위배되는 상황임에도 강제로 형 변환할 경우 데이터 손실의 위험이 있습니다.

```
int i1 = 20.5; // Error
int i2 = (int) 20.5; // 20
```

2.6 메소드(Method) 정의(1/3) – 메소드 개요

- ✓ 클래스를 구성하는 구성요소에서 해당 클래스의 행위를 의미하는 것이 메소드(Method)입니다.
- ✓ 메소드를 이해하기 위해서는 먼저 함수(function)가 무엇인지 이해해야 합니다.
- ✓ 함수는 입력(input), 기능(function), 출력(output)으로 구성되며 함수의 기능에 따라 입력과 출력은 없을 수 있습니다.
- ✓ 함수와 메소드의 구분은 클래스의 소속여부로 나누며 특정 클래스에 소속된 경우 메소드, 그렇지 않을 경우 함수라 합니다.



```
int myFunction(int x){  
    return x + 5;  
}
```



함수는 Java 8 이전 버전에서는 그 용어가 사용되지 않았습니다. 람다(Lambda) 함수가 탄생하기 이전에는 클래스에 소속된 메소드만 존재했기 때문이며, 현재는 두 용어 모두 사용되고 있습니다.

2.6 메소드(Method) 정의(2/3) – 메소드 구성 및 정의(1/2)

- ✓ 클래스의 구성요소로 메소드의 역할은 해당 클래스의 데이터에 대한 제어입니다.
- ✓ 특정 클래스의 데이터, 즉 필드 값의 변경은 곧 해당 클래스의 특정 행위가 수행되는 것을 의미합니다.
- ✓ 계좌(Account)의 잔액(balance)을 특정 금액(amount)만큼 빼는 메소드는 곧 출금(withdraw) 행위를 의미합니다.

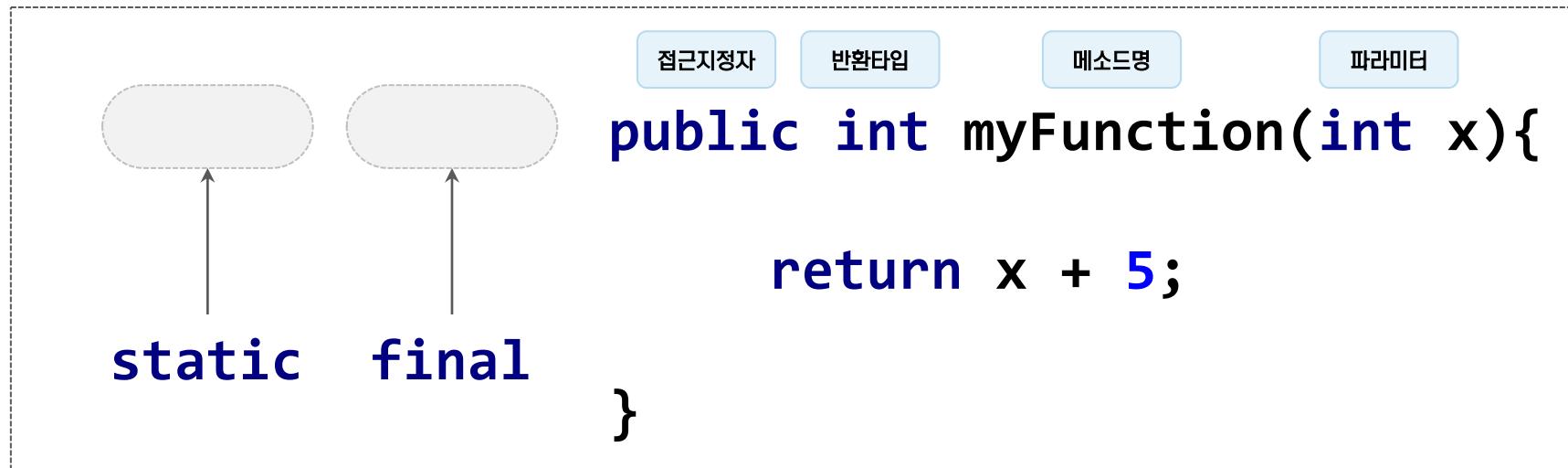
```
public class Account {  
    private double balance;  
  
    public Account(double balance){  
        this.balance = balance;  
    }  
  
    public double withdraw(double amount) {  
        if( balance < amount ) {  
            System.out.println("잔액이 부족합니다.");  
        } else {  
            balance -= amount;  
        }  
        return balance;  
    }  
}
```

Method

2.6 메소드(Method) 정의(3/3) – 메소드 구성 및 정의(2/2)

- ✓ 클래스는 다수의 메소드를 가질 수 있으며, 그 메소드의 기능은 메소드 호출을 통해 수행됩니다.
- ✓ 필드와 마찬가지로 메소드에도 접근 지정자를 지정하여 메소드 호출에 대한 범위를 정할 수 있습니다.
- ✓ 메소드는 static 키워드를 이용해 클래스 메소드와 인스턴스 메소드로 정의할 수 있습니다.

class



2.7 메소드 호출의 이해(1/2)

```
public class Account {  
    private double balance;  
    public Account(double balance){  
        this.balance = balance;  
    }  
  
    public void deposit(double amount){  
        this.balance += amount;  
    }  
  
    public double withdraw(double amount) {  
        if(balance < amount) {  
            System.out.println("잔액이 부족합니다.");  
        } else {  
            balance -= amount;  
        }  
        return balance;  
    }  
  
    public double getBalance(){  
        return this.balance;  
    }  
}
```

```
public class NamoosoriBank {  
    public static void main(String[] args) {  
        Account newAccount = new Account(10000);  
        System.out.println("1.Total Balance : " +  
                           newAccount.getBalance());  
        newAccount.deposit(10000);  
        System.out.println("2.Total Balance : " +  
                           newAccount.getBalance());  
        newAccount.withdraw(15000);  
        System.out.println("3.Total Balance : " +  
                           newAccount.getBalance());  
    }  
}
```

```
1.Total Balance : 10000.0  
2.Total Balance : 20000.0  
3.Total Balance : 5000.0
```

2.7 메소드 호출의 이해(2/2) – 오버로딩(Overloading)

- ✓ 메소드의 이름을 같게 하고, 파라미터를 달리 하여 여러 메소드를 정의하는 방법을 메소드 오버로딩이라 합니다.
- ✓ 자바의 메소드 시그니처(signature)는 메소드의 이름부터 파라미터까지입니다.
- ✓ 메소드의 반환타입은 메소드 시그니처에 포함되지 않기 때문에 메소드 오버로딩과는 관련이 없습니다.
- ✓ 메소드 오버로딩은 하나의 클래스내에서의 기능이며 파라미터는 타입, 순서, 개수를 달리하여 구분할 수 있어야 합니다.

```
public class Calculator {  
  
    public int sum(int x, int y){  
        return x + y;  
    }  
  
    public int sum(int x, int y, int z){  
        return x + y + z;  
    }  
  
    public double sum(double x, double y){  
        return x + y;  
    }  
}
```

```
public class OverloadingExam {  
  
    public static void main(String[] args) {  
        Calculator calc = new Calculator();  
        System.out.println(calc.sum(10, 20)); //30  
        System.out.println(calc.sum(10, 20, 30)); //60  
        System.out.println(calc.sum(10.0, 20.0)); //30.0  
    }  
}
```

2.8 생성자(Constructor) 정의(1/3) – 생성자 개요

- ✓ 생성자(Constructor)는 **클래스 구성요소 중 하나로, 객체를 인스턴스화** 할 때 가장 먼저 호출되는 특수한 메소드입니다.
- ✓ 생성자의 역할은 객체가 갖는 필드의 초기화이며 따라서 모든 클래스는 하나 이상의 생성자를 갖습니다.
- ✓ 생성자의 이름은 해당 클래스의 이름과 같고 반환 타입을 갖지 않습니다.
- ✓ 생성자도 모든 접근제어자를 적용할 수 있으며 일반적으로는 public 접근제어자가 적용됩니다.

Employee Class

```
public class Employee {  
  
    // fields  
    private String name;  
    private String department;  
  
    // default constructor  
    public Employee(){  
        System.out.println("Employee() 생성자 호출");  
    }  
  
    // user defined constructor  
    public Employee(String name){  
        this.name = name;  
    }  
}
```

```
public static void main (String[] args) {  
  
    /** new 연산자를 이용한 객체 인스턴스화 */  
    Employee employee1 = new Employee();
```

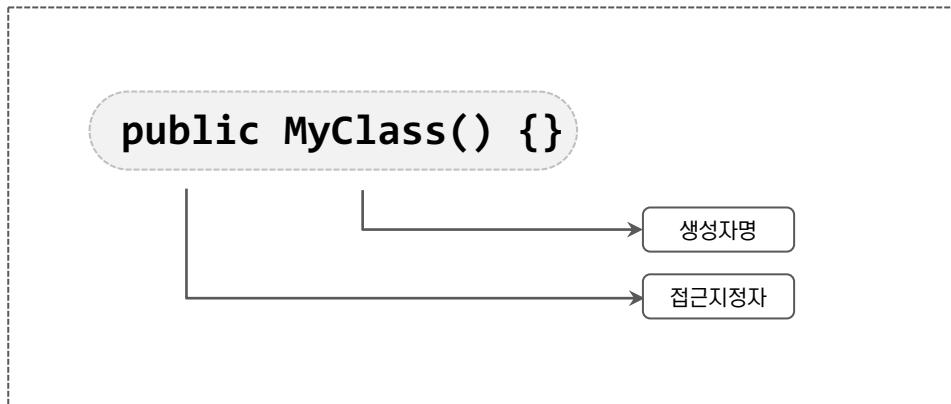
Employee() 생성자 호출

실행결과

2.8 생성자(Constructor) 정의(2/3) – 디폴트 생성자 (Default Constructor)

- ✓ 매개 변수가 없고 구현 내용 없이 정의하는 생성자를 기본 생성자 혹은 디폴트 생성자(Default Constructor)라고 합니다.
- ✓ 클래스는 반드시 하나 이상의 생성자를 가지며 사용자가 생성자를 정의하지 않으면 디폴트 생성자가 자동으로 생성됩니다.
- ✓ 생성자에서 명시적으로 필드의 값을 설정하지 않으면 디폴트 값으로 초기화 됩니다.
 - 디폴트 초기값 : 숫자 타입(0), 논리 타입(false), 참조 타입(null)

class



```
public class Student {
```

```
    private String name;
```

```
    /** 컴파일이 자동생성 해주므로 생략 가능  
     * public Student() {}  
     */
```

Default Constructor

```
}
```

```
public class Employee {  
    private String name;
```

```
    public Employee(String name){  
        this.name = name;  
    }  
}
```

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

```
    Student student = new Student();
```

```
    Employee employee = new Employee(); // compile Error
```

```
}
```

Employee 클래스에 디폴트 생성자가 없어 컴파일 에러가 발생합니다. 이러한 경우, 개발자가 추가로 정의해야 합니다.

2.8 생성자(Constructor) 정의(3/3) – User Defined Constructor

- ✓ 클래스 필드의 초기화를 위해서 정의하는 생성자를 사용자 정의 생성자(User Defined Constructor)라 합니다.
- ✓ 사용자 정의 생성자는 해당 클래스가 갖는 필드의 초기화를 위해 정의하며 여러 형태로 정의가 가능합니다.
- ✓ 만일, 사용자가 하나 이상의 사용자 정의 생성자를 만들었다면 디폴트 생성자는 자동으로 생성되지 않습니다.

class

```
public 클래스명(매개변수 타입 매개변수이름) {  
    . . .  
}
```

Student Class

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

User Defined Constructor

2.9 생성자 호출의 이해(1/2) – 생성자 오버로딩(Constructor Overloading)

- ✓ 클래스 필드의 초기화는 다양한 방식으로 이루어질 수 있으며 이를 위해 다양한 형태의 생성자를 정의할 수 있습니다.
- ✓ 생성자의 이름은 반드시 클래스의 이름과 동일해야 하기 때문에, 다수의 생성자 정의는 **생성자 오버로딩**으로 정의합니다.
- ✓ 생성자 오버로딩은 해당 클래스를 구성하는 생성자들의 **매개변수 개수와 타입, 순서를 달리하여** 정의합니다.
- ✓ 객체의 인스턴스화 과정에서 생성자를 호출하며 이때 전달인자를 다르게 하여 필요한 생성자를 호출합니다.

```
public class Employee {  
  
    private String name;  
    private int age;  
    private String department;  
  
    // 생성자 오버로딩 (constructor overloading)  
  
    public Employee() {  
        System.out.println("Employee() 생성자 호출");  
    }  
  
    public Employee(String name) {  
        System.out.println("Employee(String name) 생성자 호출");  
    }  
  
    public Employee(String name, int age){  
        System.out.println("Employee(String name, int age) 생성자 호출");  
    }  
}
```

Employee.java

```
// 오버로딩 생성자 호출  
public class Test {  
    public static void main (String args[]) {  
        Employee employee1 = new Employee();  
        Employee employee2 = new Employee("김수현");  
        Employee employee3 = new Employee("홍길동", 20);  
  
        System.out.println(employee1);  
        System.out.println(employee2);  
        System.out.println(employee3);  
    }  
}
```

실행결과

```
Employee() 생성자 호출  
Employee(String name) 생성자 호출  
Employee(String name, int age) 생성자 호출
```

2.9 생성자 호출의 이해(2/2) – this() constructor

- ✓ 하나의 클래스에 정의된 다수의 생성자 간에 this() 생성자를 통해 호출이 가능합니다.
- ✓ this() 생성자는 중복되는 코드를 제거하고 생성자를 재사용하기 위해 사용합니다.
- ✓ this() 생성자의 호출은 반드시 생성자 이름의 바로 아래에 위치해야 합니다.

```
public class Employee {  
  
    private String id;  
    private String name;  
    private String department;  
  
    public Employee() {}  
  
    public Employee(String id){  
        this.id = id;  
        System.out.println("Employee(id) 호출");  
    }  
  
    public Employee(String id, String name){  
        this(id); → 생성자 이름 바로 아래에 위치해야 합니다.  
        this.name = name;  
        System.out.println("Employee(id, name) 호출");  
    }  
  
    public Employee(String id, String name, String department){  
        this(id, name);  
        this.department = department;  
        System.out.println("Employee(id, name, department) 호출");  
    }  
}
```

Employee.java

```
public class Test {  
    public static void main (String args[]){  
        Employee employee1 = new Employee("0001", "홍길동", "나무소리");  
  
        System.out.println(employee1);  
    }  
}
```

실행결과

Employee(id) 호출
Employee(id, name) 호출
Employee(id, name, department) 호출

2.10 Java 메모리 모델

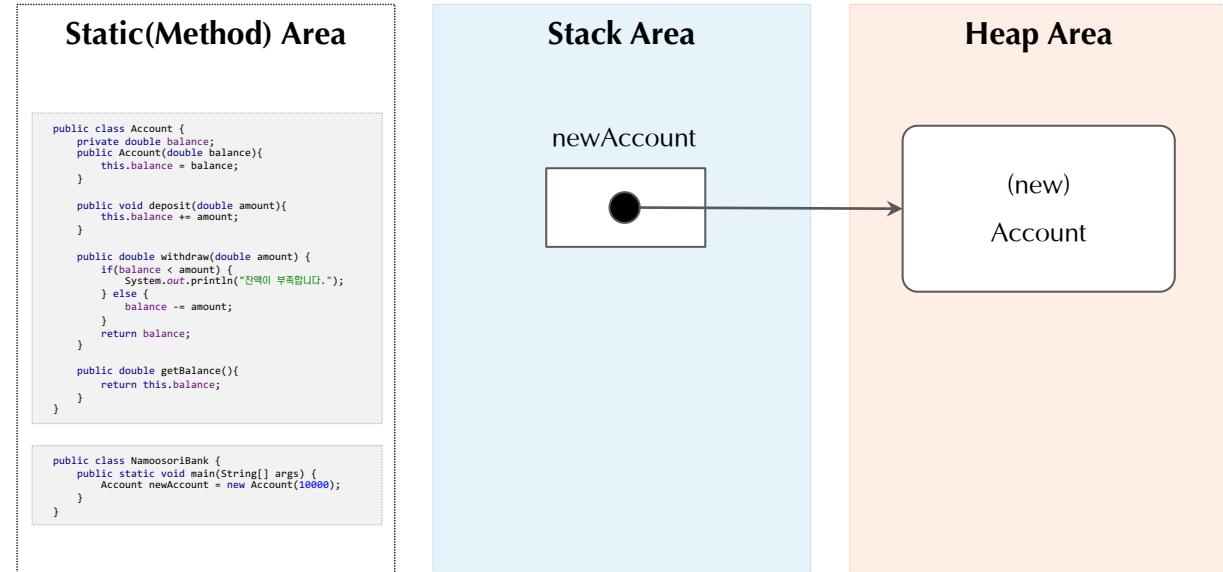
- ✓ Java의 JVM이 관리하는 메모리 공간은 크게 3가지 영역으로 나눌 수 있습니다.

- **스택 영역**(Stack Area) : 지역 변수(Local Variable), 매개 변수(Parameter)가 할당되는 영역으로 초기화가 진행되지 않습니다.
 - **힙 영역**(Heap Area) : 배열과 모든 인스턴스 객체가 할당되는 영역으로 자동 초기화가 진행됩니다.
 - **스태틱 영역**(Static Area) 또는 메소드 영역 : 메소드의 바이트 코드, static 변수가 할당 됩니다.

```
public class Account {  
    private double balance;  
    public Account(double balance){  
        this.balance = balance;  
    }  
  
    public void deposit(double amount){  
        this.balance += amount;  
    }  
  
    public double withdraw(double amount) {  
        if(balance < amount) {  
            System.out.println("잔액이 부족합니다.");  
        } else {  
            balance -= amount;  
        }  
        return balance;  
    }  
  
    public double getBalance(){  
        return this.balance;  
    }  
}
```

```
public class NamooSoriBank {  
    public static void main(String[] args) {  
        Account newAccount = new Account(10000);  
    }  
}
```

“모든 객체는 힙(Heap)영역에 저장됩니다.”



2.11 static과 final(1/5) – 개요

- ✓ static과 final 키워드는 클래스, 필드, 메소드에 모두 적용할 수 있는 키워드이며 각 위치에 따라 다른 의미를 갖습니다.
- ✓ static 키워드는 정적 키워드로 정적 필드, 정적 메소드를 선언할 때 사용합니다.
- ✓ final 키워드를 필드에 정의할 경우 초기 한번의 초기화만 가능하여 이후에는 다른 값을 대입할 수 없습니다.
- ✓ static, final 키워드가 어느 위치에 있느냐에 따라 그 기능이 다른 만큼 정확히 이해하고 사용해야 합니다.

```
public class StaticExam {  
    private static String message;  
  
    static{  
        message = "Public Message~~";  
    }  
  
    public static void showMessage(){  
        System.out.println(message);  
    }  
}
```

```
public class FinalExam {  
  
    private final String message;  
  
    public FinalExam(){  
        this.message = "Final Message~";  
    }  
  
    public final void showMessage(){  
        System.out.println(message);  
    }  
  
    public void showMessage(final String message){  
        //message = "New Message!";  
        //Cannot assign a value to final variable 'message'  
        System.out.println(message);  
    }  
}
```

2.11 static과 final(2/5) – final

- ✓ final 키워드는 클래스, 필드, 메소드, 지역변수, 파라미터에 적용할 수 있습니다.
- ✓ 클래스에 final은 상속을 허용하지 않으며 메소드의 final은 오버라이딩(overriding) 금지를 의미합니다.
- ✓ 필드, 지역변수, 파라미터에 final을 적용하면 한번 초기화 한 이후에는 다른 값으로 변경할 수 없습니다.
- ✓ final 필드의 초기화 방식은 필드 선언 시점의 초기화, 초기화 블록^②, 생성자를 통한 초기화 방법 3가지가 있습니다.

```
public class FinalExam {  
    private final String message = "Final Message"; // (1)  
    {  
        message = "Final Message"; // (2)  
    }  
    public FinalExam(){  
        this.message = "Final Message~"; // (3)  
        //Variable 'message' might already have been assigned to  
    }  
    public final void showMessage(){  
        System.out.println(message);  
    }  
    public void showMessage(final String message){  
        //message = "New Message!"; → 변경 X  
        //Cannot assign a value to final variable 'message'  
        System.out.println(message);  
    }  
}
```

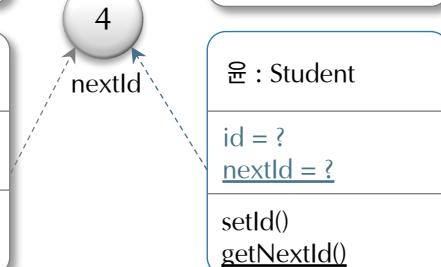
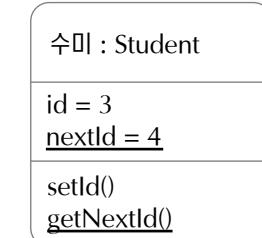
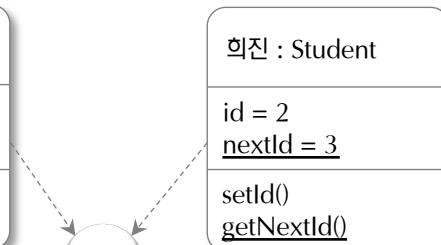
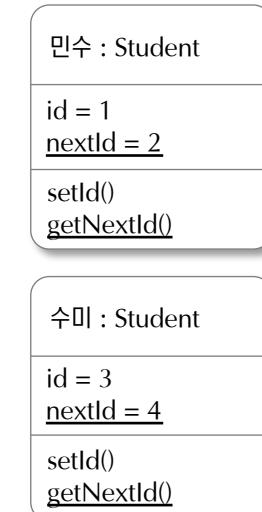
초기화 블럭 →

2.11 static과 final(3/5) – static field

- ✓ static 키워드가 적용된 필드를 정적 필드 혹은 클래스 변수라고 합니다.
- ✓ 정적 필드는 해당 클래스의 모든 인스턴스 객체들이 공유하는 변수이며 이런 의미가 바로 클래스 변수입니다.
- ✓ 정적 필드는 객체의 인스턴스화(생성) 없이 클래스 이름으로 정적 필드에 접근할 수 있습니다.
 - 단, 해당 정적 필드의 접근지정자가 무엇인지에 따라 접근 방식에 차이가 있습니다.

```
public class Student {  
  
    private static int nextId = 1;  
    private int id;  
  
    public void setId() {  
        id = nextId;  
        nextId++;  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public int getNextId() {  
        return nextId;  
    }  
}
```

```
public static void main(String[] args) {  
    Student minsu = new Student();  
    Student heejin = new Student();  
    Student sumi = new Student();  
    Student yoon = new Student();  
  
    minsu.setId();  
    System.out.println(minsu.getNextId());  
  
    heejin.setId();  
    System.out.println(heejin.getNextId());  
  
    sumi.setId();  
    System.out.println(sumi.getNextId());  
  
    yoon.setId();  
    System.out.println(yoon.getNextId());  
}
```



새로운 객체의 실행결과는?

2.11 static과 final(4/5) – 사용자 정의 상수

- ✓ 정적 필드에 final 키워드를 적용하여 값을 변경할 수 없도록 하는 것으로 사용자 정의 상수를 정의할 수 있습니다.
- ✓ 사용자 정의 상수는 정적 필드나 메소드의 접근과 마찬가지로 클래스 이름을 통해 접근하여 사용합니다.
- ✓ 사용자 정의 상수는 접근 지정자의 범위에 따라 공유하는 범위가 결정됩니다.

예) Math.PI

```
public final class Math {  
    . . .  
    public static final double PI = 3.141592653589793;  
    . . .  
}
```

예) System.out

```
public final class System {  
    . . .  
    public static final PrintStream out = . . .;  
    . . .  
}
```

2.11 static과 final(5/5) – static method

✓ 정적 메소드는 static으로 선언된 메소드로써 인스턴스 없이도 호출할 수 있습니다.

- 정적 메소드는 인스턴스 필드에는 접근할 수 없고, 정적 필드에만 접근할 수 있습니다.
- 정적 메소드는 객체를 통해 사용될 수 있지만, 반드시 클래스명과 함께 사용하기 바랍니다.

✓ 정적 메소드 예시

- Employee 클래스의 정적필드에 접근하는 정적 메소드

```
class Employee {  
    private static int nextId = 1;  
    private int id;  
  
    . . .  
    public static int getNextId() {  
        return nextId; // 정적 필드의 값을 리턴합니다.  
    }  
    . . .  
}
```

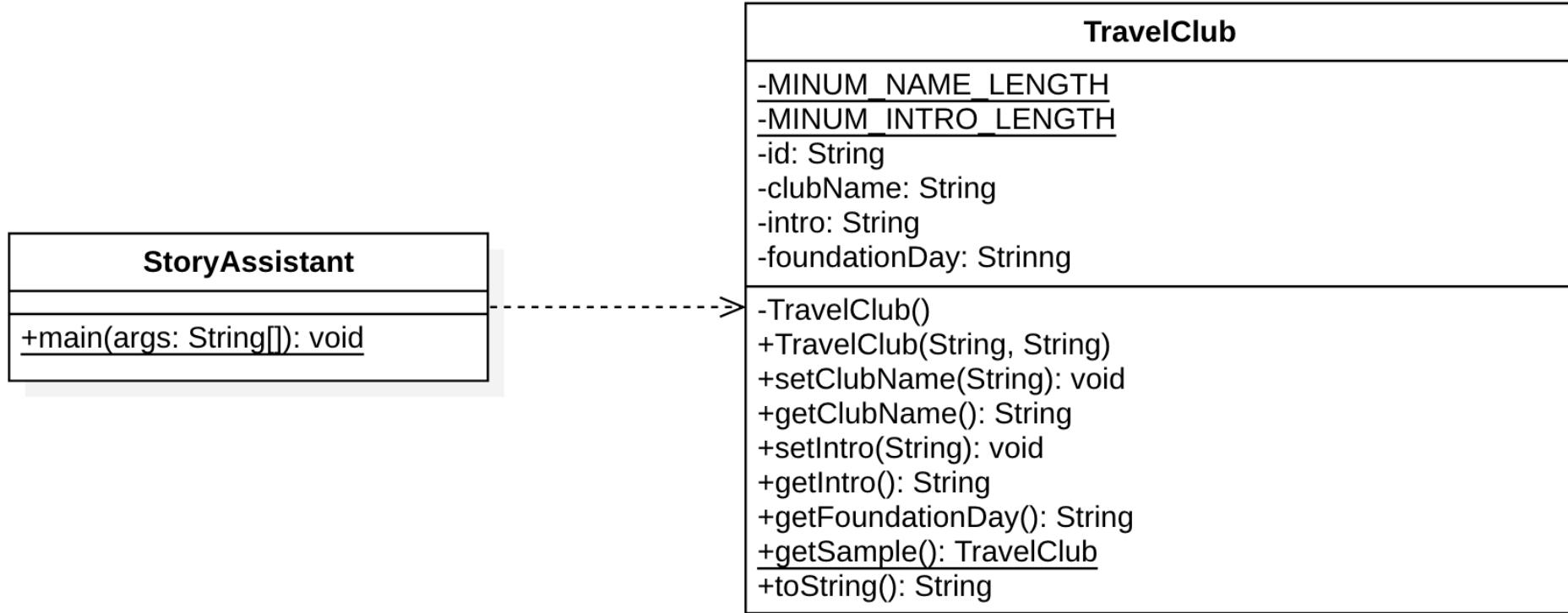
Employee.java

- Math.pow()

✓ 정적 메소드는 언제 사용할까요?

- 객체의 상태에 접근하지 않고, 필요한 파라미터가 모두 명시적 파라미터인 경우 (예, Math.pow)
- 클래스의 정적 필드에만 접근하는 경우 (예, Employee.getNextId)

[실습] TravelClub Entity



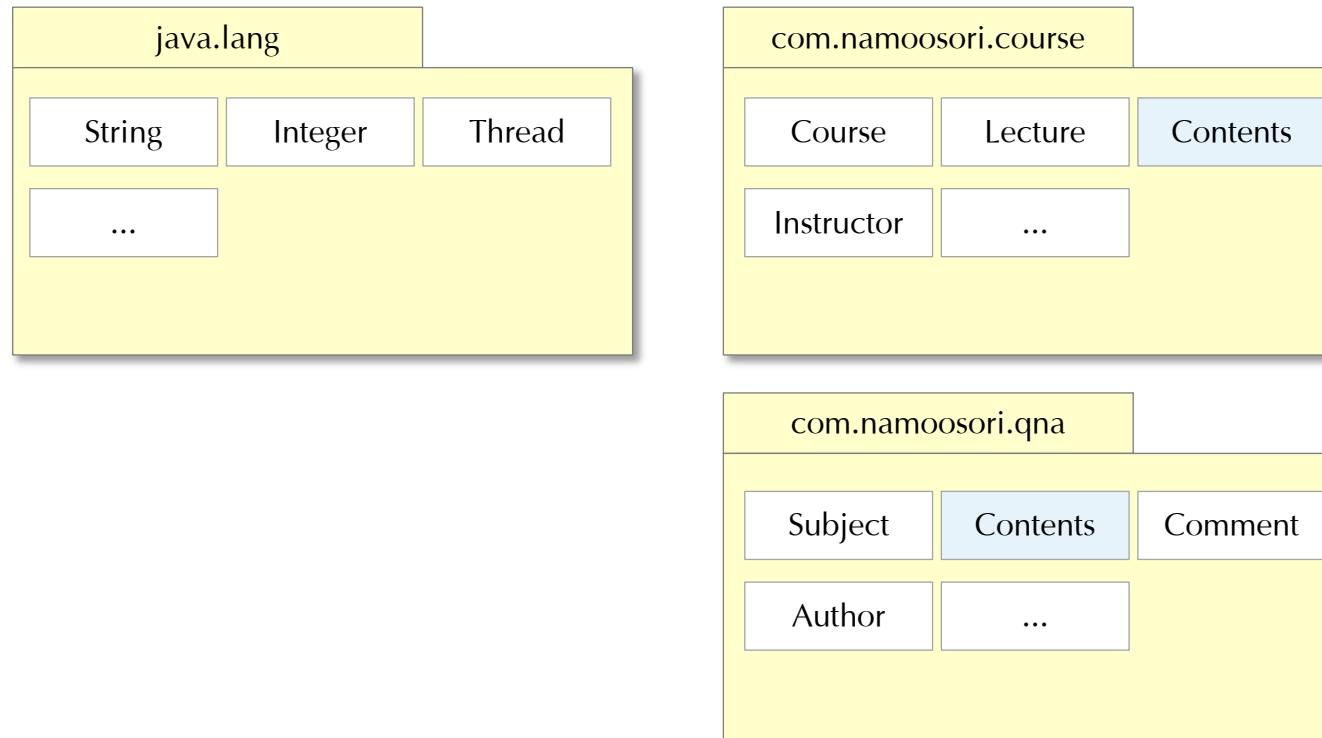


3. 클래스 관계(Relationship)

- 3.1 패키지
- 3.2 상속의 이해
- 3.3 상속 관계의 초기화 과정
- 3.4 메소드 재정의
- 3.5 객체간 타입 형변환
- 3.6 다형성
- 3.7 Object 클래스
- 3.8 추상 클래스의 이해
- 3.9 인터페이스의 이해
- 3.10 인터페이스의 활용
- 3.11 배열(Array)

3.1 패키지(1/4) – 개요

- ✓ 자바는 패키지를 통해 관련 있는 클래스들을 그룹화 합니다.
- ✓ 패키지를 사용하면 외부로부터 제공 받은 여러 클래스들과 현재 구현하는 클래스들을 구분하여 관리할 수 있습니다.
- ✓ 패키지를 사용하는 가장 중요한 이유는 클래스 이름에 대한 유일성을 보장할 수 있기 때문입니다.



3.1 패키지(2/4) – 클래스 import

- ✓ 클래스는 자신의 패키지에 있는 모든 클래스와 다른 패키지의 모든 **public** 클래스들을 사용할 수 있습니다.
- ✓ 다른 패키지의 **public** 클래스를 사용하는 두 가지 방법입니다.

- 패키지를 포함한 클래스명을 사용

```
java.util.Date today = new java.util.Date();
```

- import 문을 사용

```
import java.util.Date;  
.  
. . .  
Date today = new Date();
```

✓ import 문

- import 문은 소스파일의 가장 상단에 위치하며, 포함시키는 클래스를 정의합니다.
- 와일드카드(*)를 사용하여 특정 패키지의 모든 클래스를 포함시킬 수 있습니다. (예) `java.util.*;`
단, 사용하려는 클래스가 두 곳 이상의 패키지에 포함된 경우 컴파일 에러가 발생합니다.

```
import java.util.*;  
import java.sql.*;  
  
// compile ERROR  
Date today = new Date();
```

```
import java.util.*;  
import java.sql.*;  
import java.util.Date;  
  
Date today = new Date();
```

3.1 패키지(3/4) – static import

- ✓ `import static` 문을 사용하면 정적 메소드나 필드를 클래스명 없이 사용할 수 있습니다.
- ✓ 마찬가지로 와일드 카드를 사용하여 해당 클래스의 모든 정적 요소를 포함시킬 수 있습니다.
- ✓ 그러나, 클래스명 없이 정적 요소를 사용하는 것은 코드가 명시적이지 않아 권장하지 않습니다.

```
import static java.lang.System.out;  
.  
. . .  
out.println("Goodbye, World!"); // 예) System.out  
exit(0); // 예) System.exit
```

```
import java.lang.Math;  
.  
. . .  
Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
```

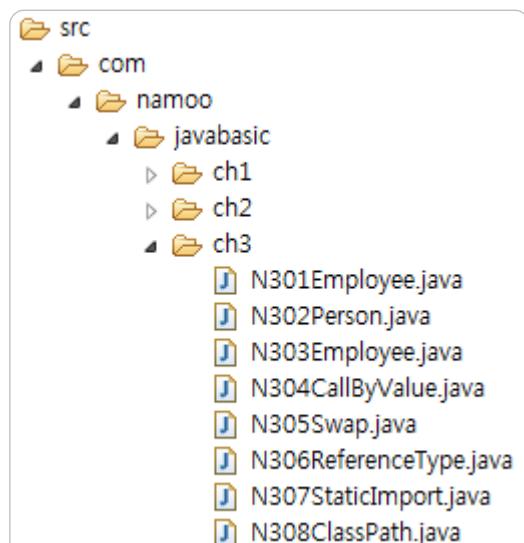
```
import static java.lang.Math.*;  
.  
. . .  
sqrt(pow(x, 2) + pow(y, 2));
```

StaticImport.java

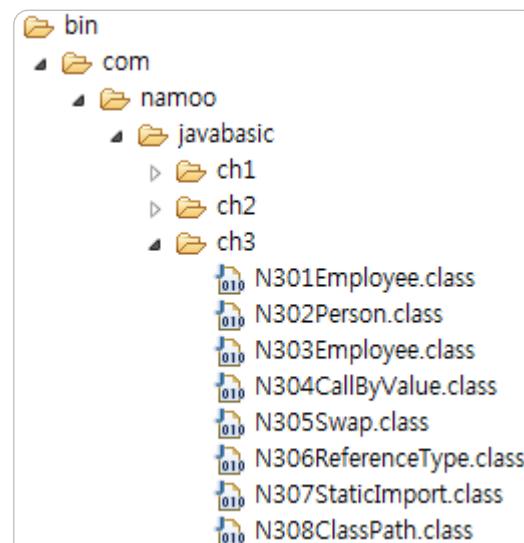
3.1 패키지(4/4) – 패키지 구성

- ✓ 패키지에 대한 정의는 클래스 소스파일의 가장 상단에 위치하며 소스 파일은 패키지명과 동일한 폴더에 위치합니다.
- ✓ 컴파일 결과(바이트코드)도 동일한 폴더에 생성됩니다.
- ✓ 통합개발환경에서는 설정을 통해 컴파일 결과를 다른 폴더에 저장할 수 있습니다.

```
package com.namoo.javabasic.ch3;
public class Student
{
    .
    .
}
```



소스파일은 src 폴더에서 관리

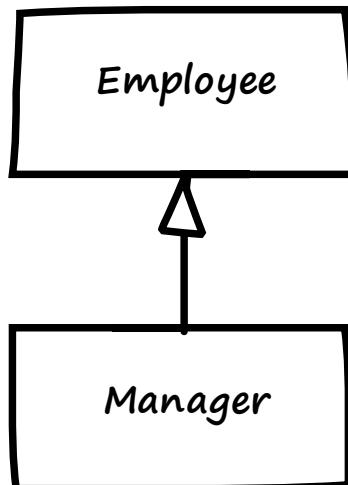


컴파일 결과는 bin 폴더에서 관리

3.2 상속(Inheritance)의 이해(1/3) – 개요

- ✓ 상속은 연관 있는 클래스들에 대해 공통적인 구성요소를 정의하고, 이를 대표하는 클래스를 정의하는 것을 의미합니다.
- ✓ 상속 관계는 “is a” 관계를 의미하며 extends 키워드를 이용해 상속 관계를 정의합니다.
- ✓ 상속 관계에서 상속을 받는 클래스를 sub class, derived class, extended class, child class라 합니다.
- ✓ 상속 관계에서 상속을 제공하는 클래스를 super class, base class, parent class라 합니다.

```
public class Manager extends Employee {  
    // 필드 및 메소드  
}
```



관리자는 직원이다 (O)
직원은 관리자이다 (X)

3.2 상속(Inheritance)의 이해(2/3)

- ✓ 자식클래스는 부모클래스를 상속받아서 부모클래스의 모든 자원(속성, 메소드)을 사용할 수 있습니다.
- ✓ 자식클래스는 부모클래스에 없는 필드와 메소드를 정의하여 기능을 추가할 수 있습니다.
- ✓ 또한, 상위클래스에 정의된 메소드를 재정의하여 다르게 동작시킬 수 있습니다. (오버라이딩)

```
public class Employee {  
  
    private String name;  
    private double salary;  
  
    public Employee(String name) {  
        this.name = name;  
    }  
    public double getSalary() {  
        return salary;  
    }  
    public void setSalary(double salary) {  
        this.salary = salary;  
    }  
    public String getName() {  
        return name;  
    }  
}
```

Employee.java

```
public class Manager extends Employee {  
  
    private double bonus;  
  
    public Manager(String name) {  
        super(name);  
    }  
    public void setBonus(double bonus) {  
        this.bonus = bonus;  
    }  
    public double getSalary() {  
        return super.getSalary() + bonus;  
    }  
}
```

Manager.java

추가적인 필드 정의

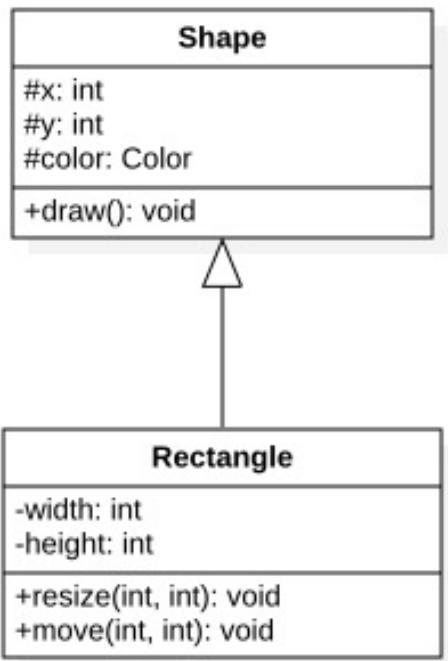
추가적인 메소드 정의

메소드 재정의 (Override)

➔ Manager 객체는 추가적으로 상여금을 가질 수 있으며,
월급 계산 시 상여금이 포함됩니다.

3.2 상속(Inheritance)의 이해(3/3) – protected

- ✓ 상속 대상이 되는 부모 클래스에 **protected** 접근 지정자로 정의된 구성요소는 자식 클래스 구성요소가 됩니다.
- ✓ 자식 클래스는 부모 클래스의 **protected**, **public** 구성요소에 대해 **this** 접근이 가능합니다.
- ✓ 즉, 상속 관계에서 자식 클래스는 부모 클래스에서 제공하는 구성요소들을 자신의 구성요소로 포함합니다.
- ✓ 부모 클래스의 **private** 구성요소는 자식 클래스에서 직접 접근할 수 없습니다.



```
public class Shape {
    protected int x;
    protected int y;
    protected Color color;

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

```
public class Rectangle extends Shape{
    private int width;
    private int height;

    public void resize(int width, int height){
        this.width = width;
        this.height = height;
    }
    public void move(int x, int y){
        this.x = x;
        this.y = y;
    }
}
```

3.3 상속 관계의 초기화 과정(1/2) – 생성자 호출

- ✓ 상속 관계에서 자식 클래스를 인스턴스화 하면 부모 클래스의 객체도 인스턴스화가 진행됩니다.
- ✓ 자식 클래스의 객체가 인스턴스화 되기 위해서는 먼저 부모 클래스의 객체가 인스턴스화되어야 합니다.
- ✓ 따라서, 상속 구조에서 가장 상위의 부모 클래스부터 차례로 인스턴스화가 진행됩니다.

```
public class Art {  
    public Art(){  
        System.out.println("Art Constructor~");  
    }  
}
```

```
public class Drawing extends Art{  
    public Drawing(){  
        System.out.println("Drawing Constructor~");  
    }  
}
```

```
public class Cartoon extends Drawing{  
    public Cartoon(){  
        System.out.println("Cartoon Constructor~");  
    }  
}
```

```
public class InheritanceAssist {  
    public static void main(String[] args) {  
        Cartoon cartoon = new Cartoon();  
    }  
}
```

```
Art Constructor~  
Drawing Constructor~  
Cartoon Constructor~
```

3.3 상속 관계의 초기화 과정(2/2) – super()

- ✓ `super()` 생성자는 자식 클래스에서 명시적으로 부모클래스의 생성자를 호출할 수 있도록 하는 방법입니다.
- ✓ 상속 관계에서 부모 클래스의 생성자 호출을 외부에서 명시적으로 지정할 수 없습니다.
- ✓ 자바는 자식 클래스의 객체가 인스턴스화 될 때 기본적으로 부모 클래스의 디폴트 생성자를 호출합니다.
- ✓ 부모 클래스에 디폴트 생성자가 정의되어 있지 않으면 자식 클래스는 명시적으로 부모 클래스의 생성자를 호출해야 합니다.

```
public class Employee {  
    private String name;  
  
    public Employee(String name){  
        this.name = name;  
    }  
}
```

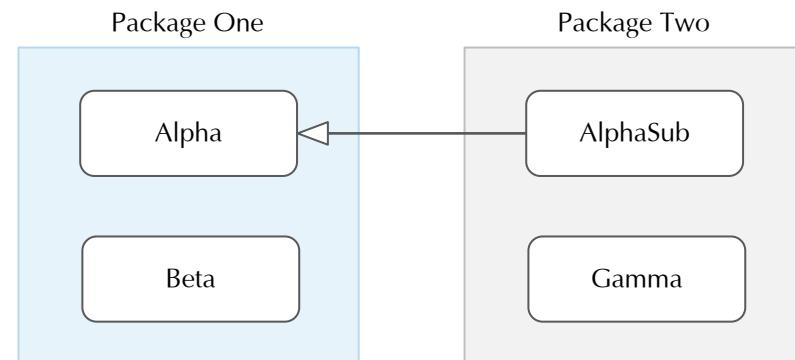
```
public class Developer extends Employee{  
    private double salary;  
  
    public Developer(String name){  
        super(name);  
    }  
  
    public void setSalary(double salary){  
        this.salary = salary;  
    }  
  
    public double getSalary(){  
        return salary;  
    }  
}
```

3.4 메소드 재정의[1/2] – 개요

- ✓ 메소드 재정의(overriding)는 부모 클래스의 메소드를 자식 클래스가 확장하거나 다시 정의하는 것을 의미합니다.
- ✓ 메소드 재정의를 구현하는 방법은 부모 클래스로부터 상속받은 메소드의 반환타입, 메소드명, 파라미터를 동일하게 하여 자식 클래스에서 정의합니다.
- ✓ 자식 클래스가 부모 클래스의 메소드를 재정의 할 때 접근 지정자의 범위는 넓거나 같아야 합니다.

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

```
public class Rectangle extends Shape{  
    @Override  
    public void draw() {  
        System.out.println("Drawing Rectangle~");  
    }  
}
```



| 부모 클래스 | 자식 클래스 |
|------------------|----------------------------|
| private | 재정의 불가 |
| protected | protected, public |
| package(default) | package, protected, public |
| public | public |

3.4 메소드 재정의[2/2] – super

- ✓ 메소드 재정의는 부모 클래스로부터 상속 받은 기능을 새롭게 변경하거나 확장하기 위해서입니다.
- ✓ 자식 클래스에서 상속 받은 메소드의 기능을 확장하기 위해서는 부모 클래스의 메소드에 대한 호출이 필요합니다.
- ✓ 자식 클래스가 부모 클래스의 구성 요소에 접근하기 위해서는 `super` 키워드를 이용합니다.

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

```
public class Rectangle extends Shape{  
    @Override  
    public void draw() {  
        super.draw();  
        System.out.println("Drawing Rectangle~");  
    }  
}
```

```
public class InheritanceAssist {  
    public static void main(String[] args) {  
        Shape shape = new Shape();  
        shape.draw();  
  
        Rectangle rect = new Rectangle();  
        rect.draw();  
    }  
}
```

Drawing Shape~

Drawing Shape~
Drawing Rectangle~

3.5 객체간 타입 형변환(1/4) – Strongly typed language

- ✓ **Strongly typed language** 개념은 데이터의 타입을 미리 정의하고 사용하고, 한번 정의된 데이터 타입은 프로그램 종료 까지 변하지 않는 것을 의미합니다.
- ✓ 이 개념은 프로그램 개발에 있어 명확함과 구체성을 제공하며 많은 프로그램 언어들이 적용하고 있습니다.
- ✓ 다만, 프로그램의 유연성 제약이라는 단점을 갖고 있으며 반대의 개념이 Loosely typed language입니다.

```
int x = 10; // assign  
int y = 20; // assign
```

double d_a = ^A
_(double)

```
public void sum(int x, int y){  
    ...  
}
```

3.5 객체간 타입 형변환(2/4)

- ✓ 자바에서 예외적으로 Strongly typed language가 적용되지 않는 경우가 있습니다.
- ✓ 상속 관계에서 자식 클래스가 부모 클래스 타입으로 참조되는 것이 허용되며 이를 up-casting이라 합니다.
- ✓ 한번 부모 클래스 타입의 클래스로 참조가 이루어진 이후 다시 자식 클래스로 참조하는 것을 down-casting이라 합니다.

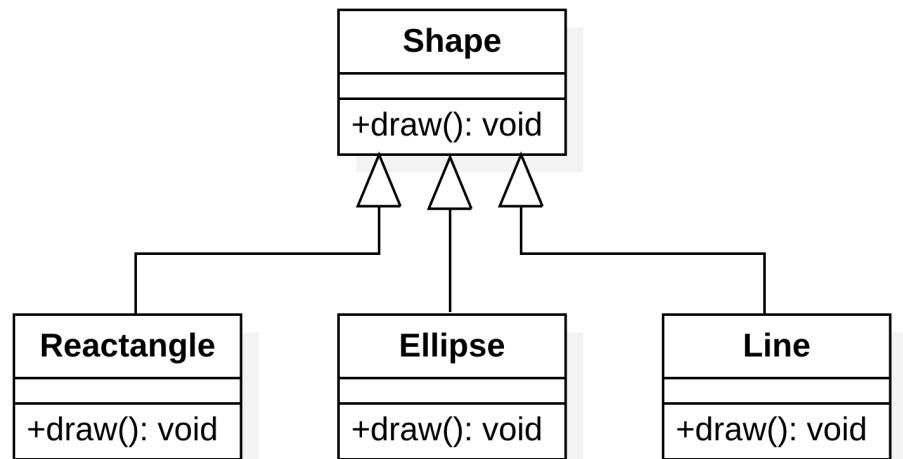
```
public class Shape {  
    public void draw(){  
        System.out.println("Drawing Shape~");  
    }  
}
```

```
public class Rectangle extends Shape{  
    @Override  
    public void draw() {  
        System.out.println("Drawing Rectangle~");  
    }  
}
```

```
public class Ellipse extends Shape{  
    @Override  
    public void draw() {  
        System.out.println("Drawing Ellipse~");  
    }  
}
```

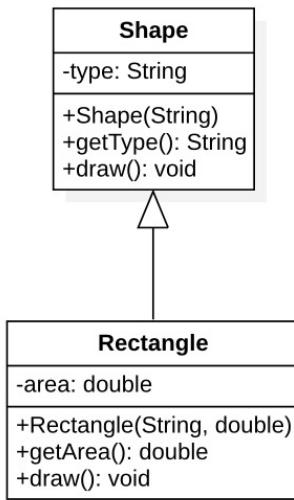
```
public class Line extends Shape{  
    @Override  
    public void draw() {  
        System.out.println("Drawing Line~");  
    }  
}
```

```
public class InheritanceAssist {  
    public static void main(String[] args) {  
        Shape shape = new Rectangle(); // up-casting  
  
        Rectangle rect = (Rectangle) shape; // down-casting  
    }  
}
```

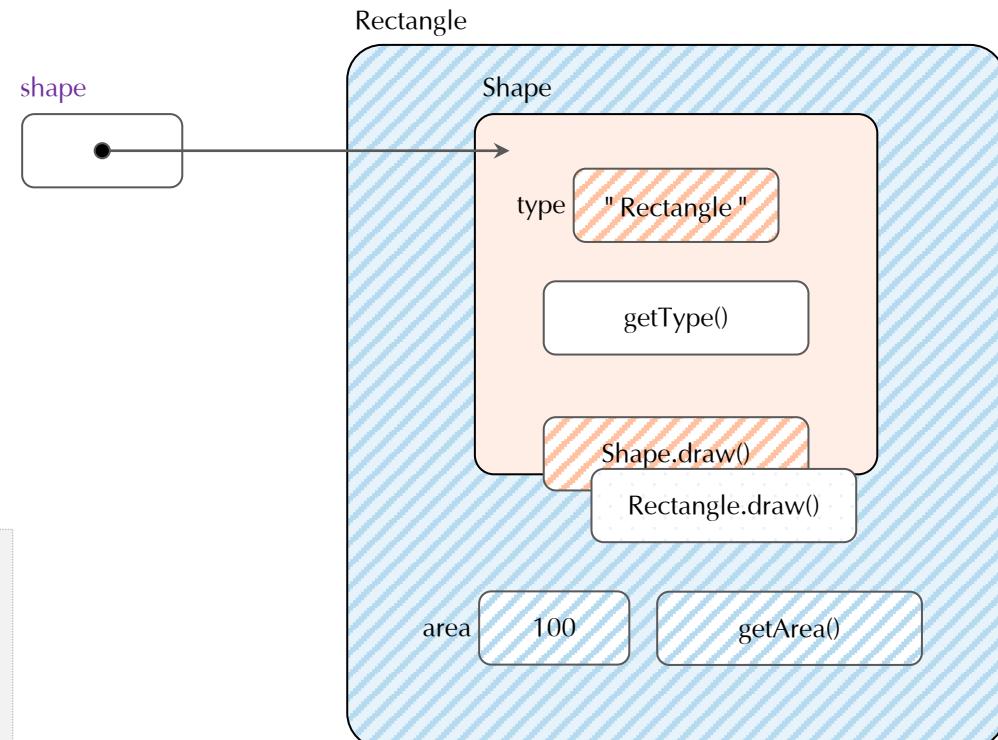


3.5 객체간 타입 형변환(3/4)

- ✓ 상속 관계가 성립되어 있고 자식 클래스의 인스턴스가 부모 클래스 타입의 변수로 참조 되면 실제 인스턴스화 객체가 자식 객체일 경우에도 자식 클래스가 갖고 있는 인스턴스 메소드는 호출 할 수 없습니다.
- ✓ 부모 클래스 타입의 변수로 참조하고 있는 자식 인스턴스 객체의 메소드를 호출하기 위해서는 down-casting이 이루어져야 합니다.
- ✓ 단, 재정의 하고 있는 메소드의 경우 up-casting 상황에서도 자식 클래스의 재정의 메소드가 호출 됩니다.

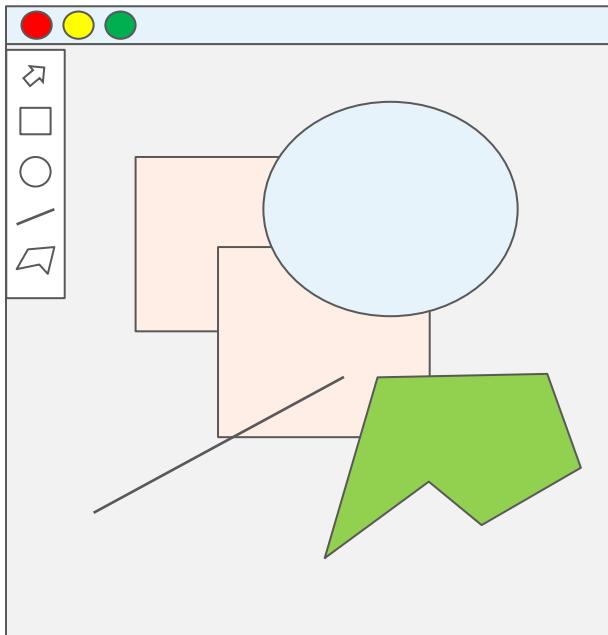


```
public class InheritanceAssist {
    public static void main(String[] args) {
        Shape shape = new Rectangle("Rectangle", 100);
        shape.getType();
        shape.draw();
    }
}
```

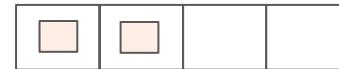


3.5 객체간 타입 형변환(4/4)

- ✓ 상속 관계에서 객체간의 타입 형변환은 객체를 관리하는데 있어 큰 이점을 제공합니다.
- ✓ 자식 클래스가 부모 클래스 타입의 변수로 참조 가능하기 때문에 같은 부모 클래스를 상속하는 모든 자식 클래스들을 하나의 타입으로 관리 할 수 있습니다.
- ✓ 이와 같은 객체의 관리는 프로그램의 복잡성을 크게 개선할 수 있습니다.



rectangles



ellipses



lines



polygons



sequence

| | | | | |
|----|----|----|----|----|
| R1 | R2 | L1 | E1 | P1 |
|----|----|----|----|----|



shapes



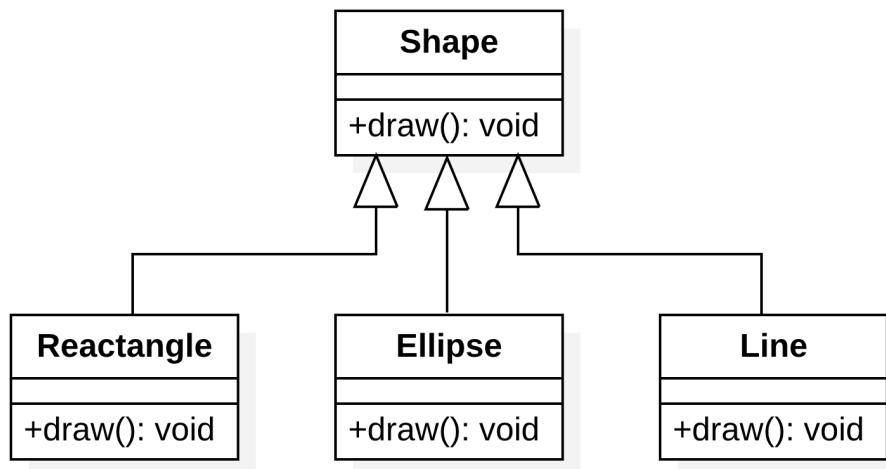
3.6 다형성

✓ **다형성(polymorphism)**은 하나의 객체가 다양한 형태로 처리될 수 있는 특성을 의미합니다.

- Poly(Many) + Morphism(Behavior)

✓ 다형성을 구현하기 위해서는 상속, 재정의 메소드, 그리고 객체간의 형변환 세가지의 조건이 필요합니다.

✓ 다형성의 이형 집합을 통한 구현 방식과 파라미터를 통한 구현 방식으로 구분 할 수 있습니다.



```
Shape[] shapes = new Shape[5];
shapes[0] = new Rectangle();
shapes[1] = new Ellipse();
shapes[2] = new Line();
```

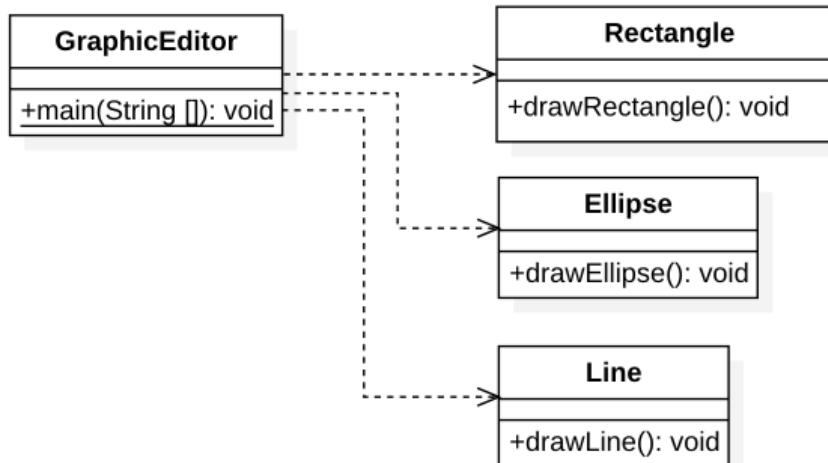
```
for(Shape shape : shapes){
    shape.draw();
}
```

```
public class InheritanceAssist {
    public static void main(String[] args) {
        drawShapes(new Rectangle());
        drawShapes(new Ellipse());
        drawShapes(new Line());
    }

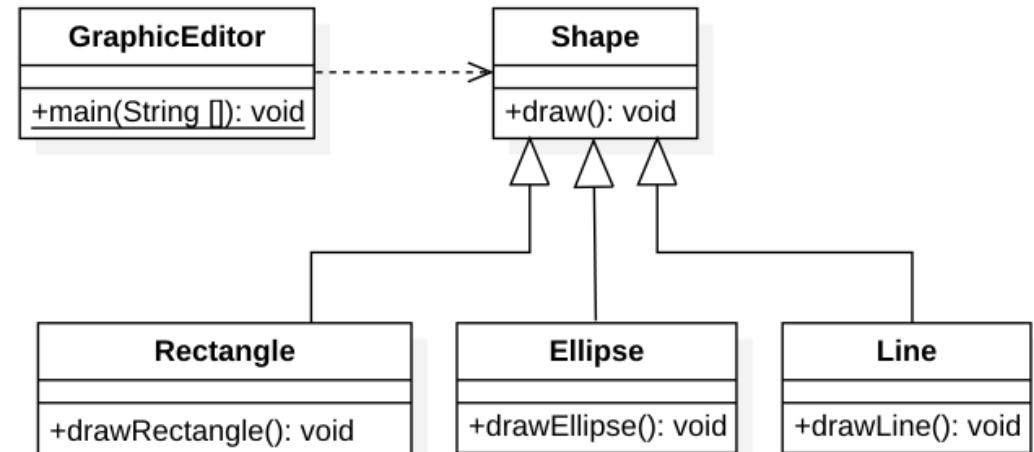
    public static void drawShapes(Shape shape){
        shape.draw();
    }
}
```

[실습] GraphicEditor

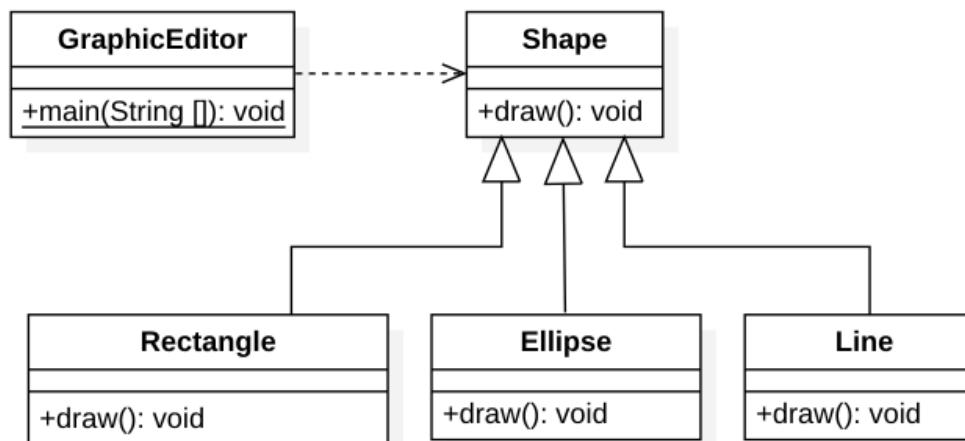
Step-01



Step-02



Step-03



3.7 Object 클래스(1/5) – 개요

- ✓ Object 클래스는 모든 자바 클래스가 상속하는 최상위 클래스입니다.
- ✓ Object 클래스는 모두 11개의 메소드를 정의하고 있으며 이 메소드들은 자바의 모든 클래스가 갖는 기능입니다.
- ✓ 자바의 모든 클래스가 상속 받아 갖는 Object 클래스 메소드에 대한 목적과 기능을 이해하는 것이 중요합니다.
- ✓ Object 클래스를 통해 상속 받는 메소드의 의미를 이해해야 그 의미에 맞게 재정의 할 수 있습니다.

```
public class EmptyClass { }

public class ObjectExam {
    public static void main(String[] args) {
        EmptyClass empty = new EmptyClass();
        empty.toString();
        empty.hashCode();
    }
}
```

| Object Class Methods |
|------------------------------------|
| protected Object clone() |
| boolean equals(Object obj) |
| protected finalize() |
| Class<?> getClass() |
| int hashCode() |
| void notify() |
| void notifyAll() |
| String toString() |
| void wait() |
| void wait(long timeout) |
| void wait(long timeout, int nanos) |

3.7 Object 클래스(2/5) – `toString` 메소드

- ✓ `toString()` 메소드는 해당 클래스에 대한 설명을 문자열 타입으로 반환하는 메소드입니다.
- ✓ 자바의 모든 클래스는 스스로에 대한 정보를 읽기 쉬운 형태로 제공 할 수 있으며 이 기능이 `toString()` 메소드입니다.
- ✓ 클래스를 정의할 때 그 클래스를 나타내는 주요 정보를 이용해 `toString()` 메소드를 재정의 합니다.

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age){  
        this.name = name;  
        this.age = age;  
    }  
    @Override  
    public String toString() {  
        return "Name:" + name + ", " + "Age:" + age;  
    }  
}
```

```
public class ObjectExam {  
    public static void main(String[] args) {  
        Person person = new Person("Kim", 25);  
        System.out.println(person.toString()); // Name:Kim, Age:25  
    }  
}
```

```
String str = new String("Java Programming~");  
  
System.out.println(str.toString());  
// Java Programming~  
System.out.println(str);  
// Java Programming~
```

```
Point point = new Point(10, 20);  
  
System.out.println(point.toString());  
//java.awt.Point[x=10,y=20]
```

3.7 Object 클래스(3/5) – equals, hashCode 메소드

- ✓ 자바의 모든 클래스는 비교가 가능해야 하며, 비교는 해당 클래스의 속성을 기준으로 합니다.
- ✓ 자바에서 비교는 동일(identity)비교와 동등(equality)비교로 구분됩니다.
- ✓ equals() 메소드는 인스턴스 객체와 파라미터로 전달되는 객체를 같은지 비교하여 그 결과는 반환합니다.
- ✓ 해시 코드(hashcode)는 객체를 식별하는 정수값을 의미하며 hashCode() 메소드는 이 정수값을 반환하는 메소드입니다.

```
public class Person {  
    private String name;  
    private int age;  
    public Person(String name, int age){  
        this.name = name;  
        this.age = age;  
    }  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (!(o instanceof Person)) return false;  
        Person person = (Person) o;  
        return Objects.equals(name, person.name);  
    }  
    @Override  
    public int hashCode() {  
        return Objects.hash(name);  
    }  
}
```

3.7 Object 클래스(4/5) – clone 메소드 [1/2]

- ✓ `clone()` 메소드는 인스턴스 객체의 복제를 위한 메소드로 해당 인스턴스 객체를 복사하여 그 참조값을 반환합니다.
- ✓ 클래스의 복제를 가능하게 하기 위해서는 **Cloneable** 인터페이스를 재정의 해야 합니다.
- ✓ **Cloneable** 인터페이스를 구현(implements) 하지 않은 클래스의 인스턴스의 `clone()` 메소드를 호출하면 **CloneNotSupportedException** 예외가 발생합니다.

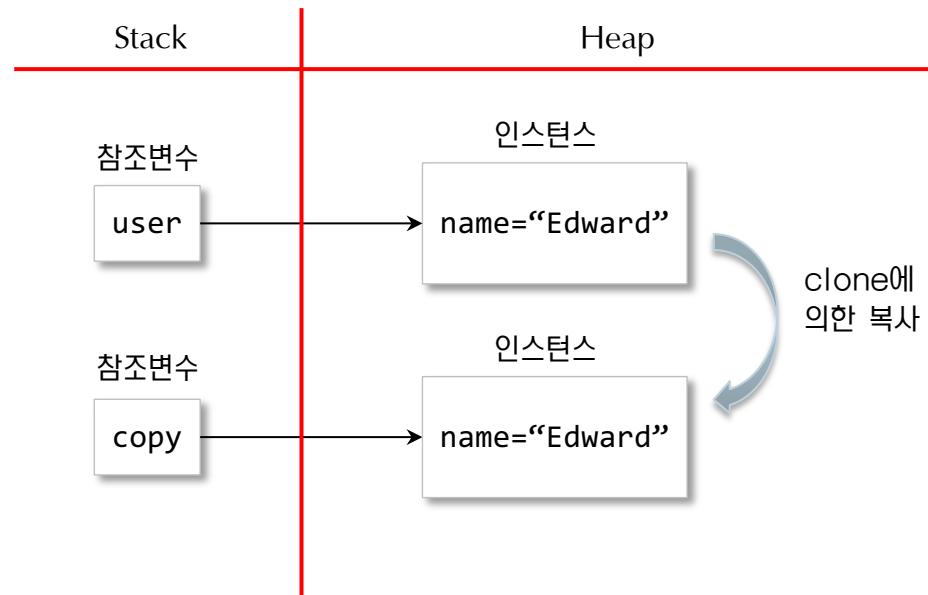
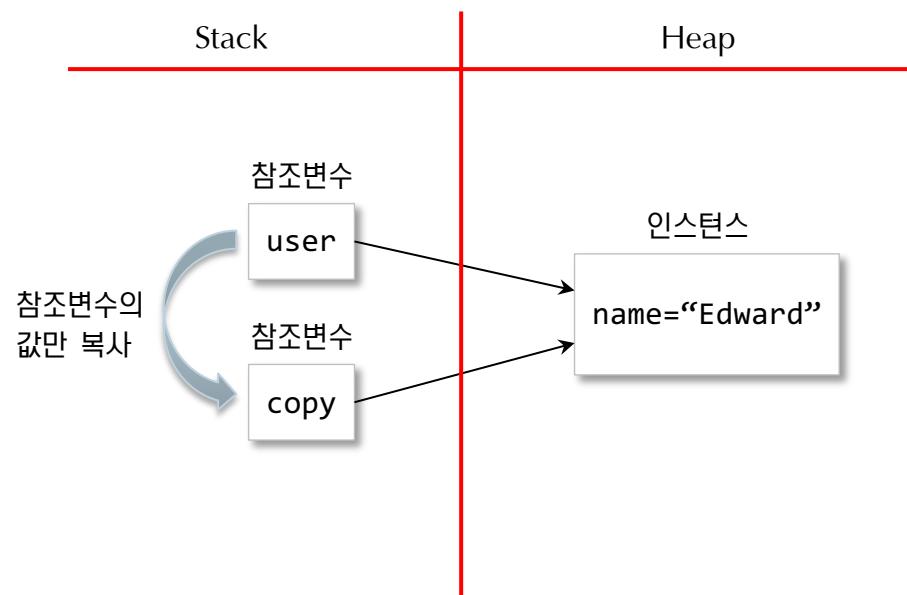
```
public class User implements Cloneable {  
  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public Object clone() throws CloneNotSupportedException {  
        return super.clone();  
    }  
}
```

3.7 Object 클래스(5/5) – clone 메소드 (2/2)

- ✓ 객체의 복사는 단순히 같은 참조 정보를 복사하는 형태와 동일한 객체의 인스턴스를 복사하는 것의 차이를 이해해야 합니다.
- ✓ Object 클래스의 `clone()` 메소드는 `protected` 접근 권한을 갖고 있으며 이를 재정의하는 클래스는 이를 `public` 접근 권한으로 재정의하여 어디서나 복제가 가능하도록 합니다.
- ✓ 객체의 복제는 얕은 복제(Shallow Copy), 깊은 복제(Deep Copy)가 있으며 이를 유의합니다.

```
User user = new User();
user.setName("Edward");
User copy = user;
```

```
User user = new User();
user.setName("Edward");
User copy = (User) user.clone();
```



3.8 추상 클래스[abstract class]의 이해

- ✓ 추상 클래스는 하나 이상의 추상 메소드(abstract method)를 갖는 클래스입니다.
- ✓ 상속 관계에서 부모 클래스의 역할을 갖기 위한 클래스이며 추상 메소드와 일반 메소드를 가질 수 있습니다.
- ✓ 추상 메소드는 메소드의 몸체(body)가 없는 메소드이며 자식 클래스에서 재정의하도록 하기 위한 메소드입니다.
- ✓ 추상 클래스는 new 동적 할당자를 통해 인스턴스 객체를 만들 수 없습니다.

```
public abstract class Shape {  
    private String type;  
  
    public Shape(String type){  
        this.type = type;  
    }  
  
    public abstract void draw();  
}
```

추상 메소드

```
public class AbstractAssist {  
    public static void main(String[] args) {  
        Shape shape = new Shape();  
        // 'Shape' is abstract; cannot be instantiated  
    }  
}
```

3.9 인터페이스(interface)의 이해(1/2) – 개요

- ✓ 인터페이스는 일반적으로 추상 메소드만 가지며 interface 키워드를 이용해 정의합니다.
- ✓ 특정 클래스가 인터페이스를 구현하기 위해서는 implements 키워드를 통해 구현합니다.
- ✓ 상속과 달리 인터페이스는 하나의 클래스가 둘 이상의 인터페이스를 동시에 구현할 수 있습니다.
- ✓ 인터페이스를 통해 설계와 구현을 완전히 분리할 수 있습니다.

```
public interface IBehavior {  
    public abstract void play();  
    생략 가능  
}
```

```
public class Soccer extends Sport implements IBehavior {  
    @Override  
    public void play() {  
        System.out.println("Playing Soccer~~");  
    }  
}
```

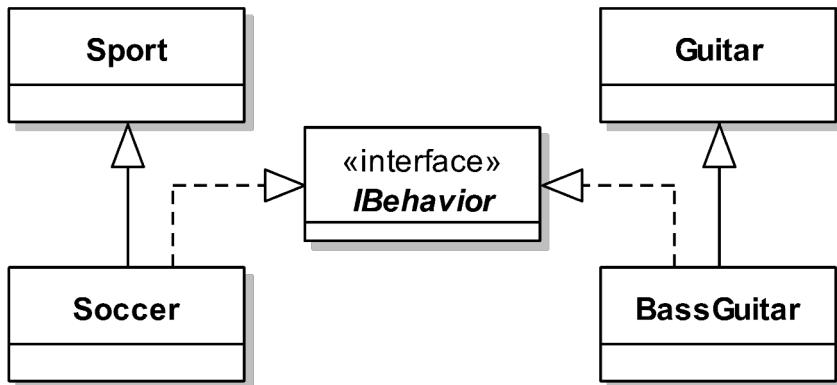
3.9 인터페이스(interface)의 이해(2/2) – 특징

- ✓ 인터페이스에는 필드, 추상메소드, static 메소드, default 메소드를 정의할 수 있습니다.
- ✓ 인터페이스에 정의하는 모든 필드는 public static final(사용자 정의 상수)이 자동으로 적용됩니다.
- ✓ Java 8부터 인터페이스에 static 메소드를 추가할 수 있으며 static 메소드의 사용은 일반 클래스와 동일합니다.
- ✓ Java 8부터 default 메소드가 추가되었으며 이 메소드는 그 자체로 완전한 메소드이며 구현 클래스는 선택적으로 재정의 할 수 있습니다.

```
public interface MyInterface {  
    생략 가능 → public static final String MESSAGE = "User define constant";  
  
    default void defaultMethod(){  
        System.out.println("Default Method~~");  
    }  
  
    → public static void staticMethod(){  
        System.out.println("Static Method~~");  
    }  
}
```

3.10 인터페이스의 활용 [1/2] – 다형성

- ✓ 자바의 상속 구조는 단일 상속의 원칙을 갖기 때문에 하나의 클래스가 여러 부모 클래스를 상속할 수 없습니다.
- ✓ 서로 다른 부모 클래스를 갖는 클래스간에도 같은 인터페이스를 구현할 수 있습니다.
- ✓ 같은 인터페이스를 구현하고 있는 클래스간에는 그 인터페이스로 하여금 대표성을 갖게 할 수 있습니다.



```
public class InterfaceAssist {
    public static void main(String[] args) {
        play(new Soccer());
        play(new BassGuitar());
    }

    public static void play(IBehavior ib){
        ib.play();
    }
}
```

```
public interface IBehavior {
    void play();
}

public class Soccer extends Sport implements IBehavior{
    @Override
    public void play() {
        System.out.println("Playing Soccer~~");
    }
}
```

```
public class BassGuitar extends Sport implements IBehavior{
    @Override
    public void play() {
        System.out.println("Playing BassGuitar~~");
    }
}
```

```
Playing Soccer~~
Playing BassGuitar~~
```

3.10 인터페이스의 활용 [2/2] – 인터페이스 역할

- ✓ 하나의 프로그램의 다수의 클래스들이 서로 관계를 형성하게 되고 각 클래스들은 역할에 따라 구분합니다.
- ✓ 클래스들간에 관계를 밀접하게 구성하게 되면 특정 클래스에서 변경이 일어날 경우 많은 클래스들이 영향을 받습니다.
- ✓ 따라서, 클래스 간에 관계를 구성할 때 그 관계를 느슨하게 관리하는 것이 중요합니다.
- ✓ 클래스와 클래스 사이에 인터페이스를 구성하면 직접적인 접근이 없는 느슨한 관계를 구성할 수 있습니다.



3.11 배열(1/4)

- ✓ 배열은 동일한 타입의 값들을 저장할 수 있는 자료구조입니다.
- ✓ 배열에 담기는 각각의 값들은 인덱스 번호를 통해 접근할 수 있습니다.
- ✓ 배열의 선언

- 문법 : 배열 요소의 타입[] 배열명;
- 선언 예 : int[] a; 또는 int a[];
- 선언된 배열은 초기화 후에 사용할 수 있습니다.

✓ 배열의 생성

- 문법 : 변수명 = new 배열 요소의 타입[요소 수];
- 배열 나타내는 변수에 새로운 배열을 생성하여 할당합니다.
 - int[] a;
 - a = new int[100];
- 배열 선언과 동시에 새로운 배열을 생성할 수도 있습니다.
 - int[] a = new int[100];

3.11 배열(2/4)

✓ 배열 초기화

- 자바는 배열 생성과 동시에 값을 초기화할 수 있는 문법을 제공합니다. (배열 선언 시에만 사용)

```
int[] months = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

```
int[] months;  
months = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}; (x)
```

✓ 배열 복사

- 배열 변수를 복사하는 경우, 참조변수만 복사되어 실제로는 두 변수가 동일한 배열을 참조합니다.

```
int[] a = new int[5];  
int[] b = a;  
b[0] = 5; // a[0]의 값도 변경됩니다.
```

- Arrays.copyOf() 메소드는 배열의 내용까지 복사합니다.

```
int[] copy = Arrays.copyOf(a, a.Length);
```

- 배열 복사와 동시에 배열의 크기를 늘릴 수 있습니다.

```
int[] copyDoubleSize = Arrays.copyOf(a, 2 * a.Length);
```

3.11 배열(3/4)

✓ 배열 정렬 (Sorting)

- Arrays.sort() 메소드는 배열 내부의 요소(element)를 정렬합니다.

```
int[] a = {2, 8, 6, 3, 1, 10};  
Arrays.sort(a); // a => {1, 2, 3, 6, 8, 10};
```

✓ 다차원 배열

```
double[][] balances;  
balances = new double[12][10];  
  
int[][] matrix = {  
    {1, 5, 2, 8},  
    {2, 7, 1, 9},  
    {7, 2, 6, 2},  
    {9, 1, 4, 8}  
};
```

Multi-dimensional Arrays

| | | | |
|---|---|---|---|
| 1 | 5 | 2 | 8 |
| 2 | 7 | 1 | 9 |
| 7 | 2 | 6 | 2 |
| 9 | 1 | 4 | 8 |

Ragged Arrays

| | | | |
|---|---|---|---|
| 1 | | | |
| 1 | 2 | | |
| 1 | 2 | 3 | |
| 1 | 2 | 3 | 4 |

✓ Ragged Array

- 사실 자바는 다차원 배열을 가지고 있지 않습니다. 단지 배열의 요소가 배열인 것 뿐입니다.
- 이러한 특성을 이용하면 가로의 길이가 일정하지 않은 배열을 생성할 수 있는데, 이것을 Ragged Array라 합니다.

3.11 배열(4/4)

- ✓ 배열과 같이 집합을 나타내는 자료형은 요소를 반복하기 위하여 반복문을 사용합니다.
- ✓ 배열 요소의 인덱스 값은 0부터 시작합니다.
- ✓ for each 문은 집합의 요소를 처음부터 끝까지 반복합니다.

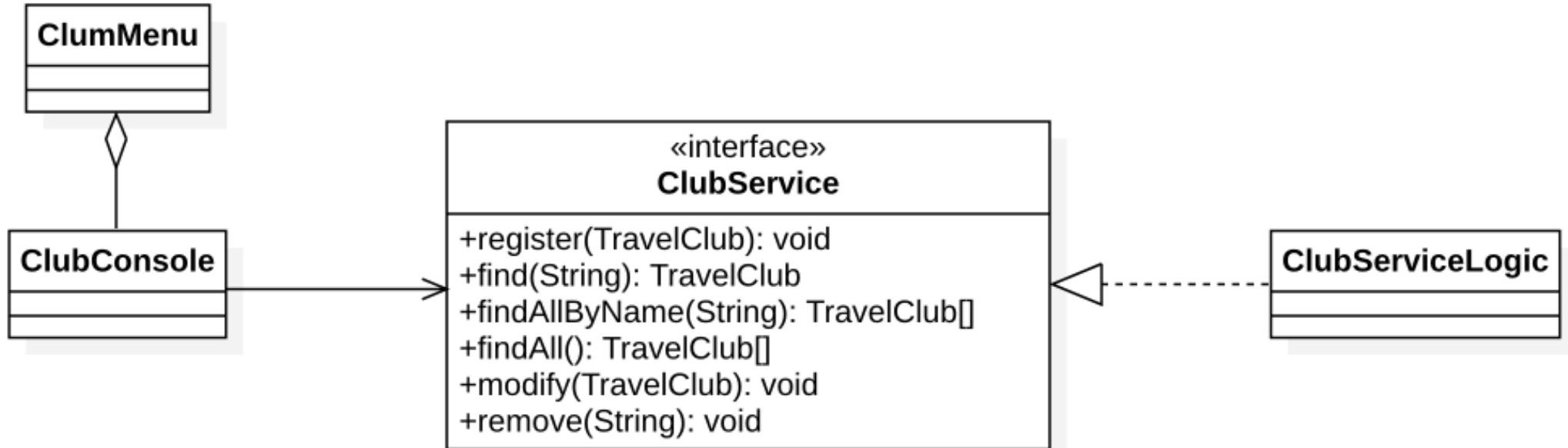
| | | | | |
|-----|---|---|---|---|
| arr | 0 | 1 | 2 | 3 |
| | 1 | 5 | 2 | 8 |

```
int arr[] = {1, 5, 2, 8};

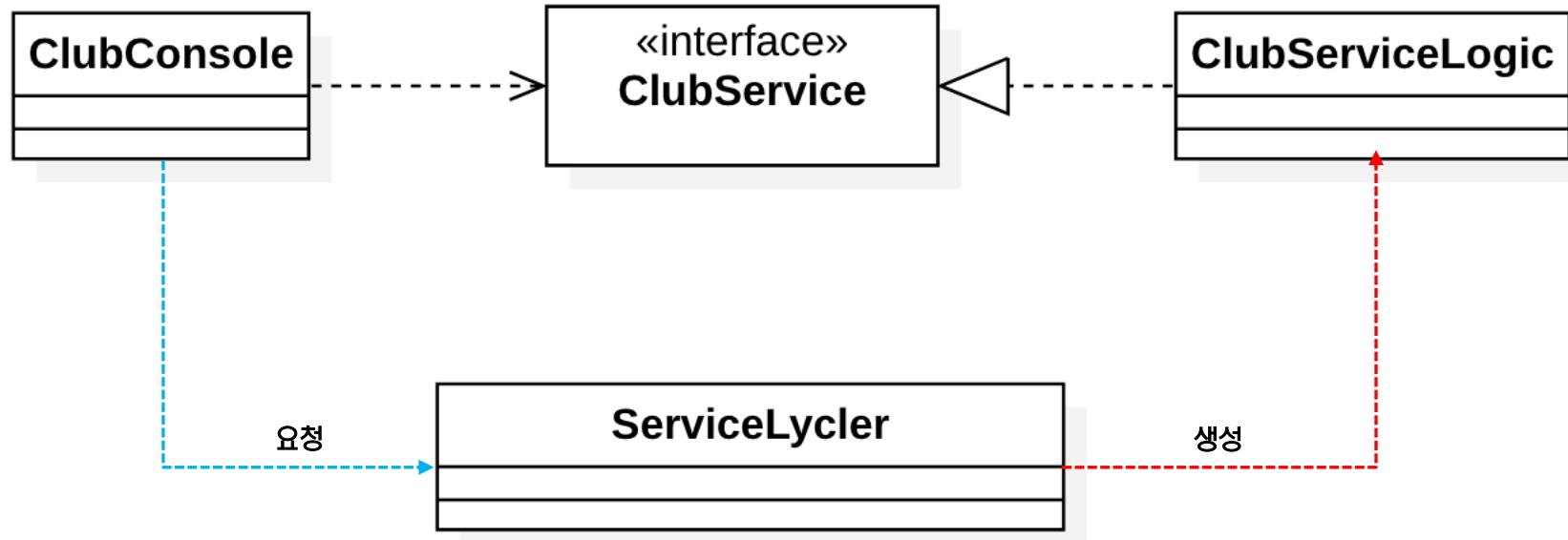
// for each문의 사용
for (int num : arr) {
    System.out.println(num);
}

// for 문을 사용하는 경우
for (int i = 0; i < arr.length; i++) {
    System.out.println(arr[i]);
}
```

[실습] TravelClub UI, Service Layer



[실습] TravelClub UI, Service Layer





4. Java API

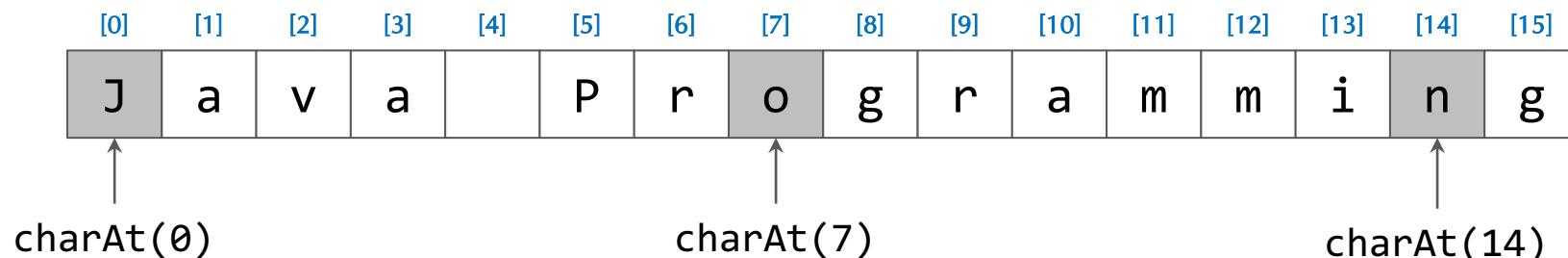
- 4.1 문자열 클래스
- 4.2 Wrapper 클래스
- 4.3 예외처리
- 4.4 Java Collection Framework의 이해
- 4.5 제네릭의 이해
- 4.6 Collection 인터페이스
- 4.7 List 인터페이스
- 4.8 Set 인터페이스
- 4.9 Map 인터페이스
- 4.10 Iterator 인터페이스

4.1 문자열 클래스[1/6] – String 클래스

- ✓ 자바는 문자열을 관리하기 위한 여러 클래스를 제공하고 있으며 대표적인 클래스가 바로 String 클래스입니다.
- ✓ String 클래스는 문자열 제어를 위한 다수의 메소드를 정의하고 있습니다.
- ✓ 프로그램내에서 빈번하게 발생하는 문자열 처리를 위해서는 String 클래스의 주요 기능들에 대한 이해와 특성에 대한 이해가 필요합니다.

```
// Creating Strings
String str1 = "Java Programming";
String str2 = new String("Java Programming");
char[] charAry = {'J', 'A', 'V', 'A'};
String str3 = new String(charAry);

// String's Methods
System.out.println(str1.length()); //16
System.out.println(str1.toUpperCase()); //JAVA PROGRAMMING
```



4.1 문자열 클래스[2/6] – String 클래스 문자 체계

- ✓ String 클래스는 내부적으로 byte[]을 이용해 문자열을 관리합니다. (Java 8까지 char[])
- ✓ 자바의 문자형 체계는 UTF-16 코드 체계를 채택하고 있으며 이는 2byte 기반의 문자 체계입니다.
- ✓ 1byte로 표현 가능한 문자의 경우 남은 1byte에 대한 메모리 낭비가 발생하게 되며 이를 개선하기 위해 Java 9부터 Compact Strings 개념이 도입되었습니다.

```
String eng = "ABCDEF";  
String kor = "자바";
```

```
eng = "ABCDEF"  
f value = {char[6]}@472 [A, B, C, D, E, F]  
0 0 = 'A' 65  
0 1 = 'B' 66  
0 2 = 'C' 67  
0 3 = 'D' 68  
0 4 = 'E' 69  
0 5 = 'F' 70  
f hash = 0  
kor = "자바"  
f value = {char[2]}@473 [자, 바]  
0 0 = '자' 51088  
0 1 = '바' 48148  
f hash = 0
```

Java 8

```
eng = "ABCDEF"  
f value = {byte[6]}@826 [65, 66, 67, 68, 69, 70]  
0 0 = 65  
0 1 = 66  
0 2 = 67  
0 3 = 68  
0 4 = 69  
0 5 = 70  
f coder = 0  
f hash = 0  
kor = "자바"  
f value = {byte[4]}@825 [-112, -57, 20, -68]  
0 0 = -112  
0 1 = -57  
0 2 = 20  
0 3 = -68  
f coder = 1  
f hash = 0
```

Java 9

4.1 문자열 클래스[3/6] – String 클래스 특징

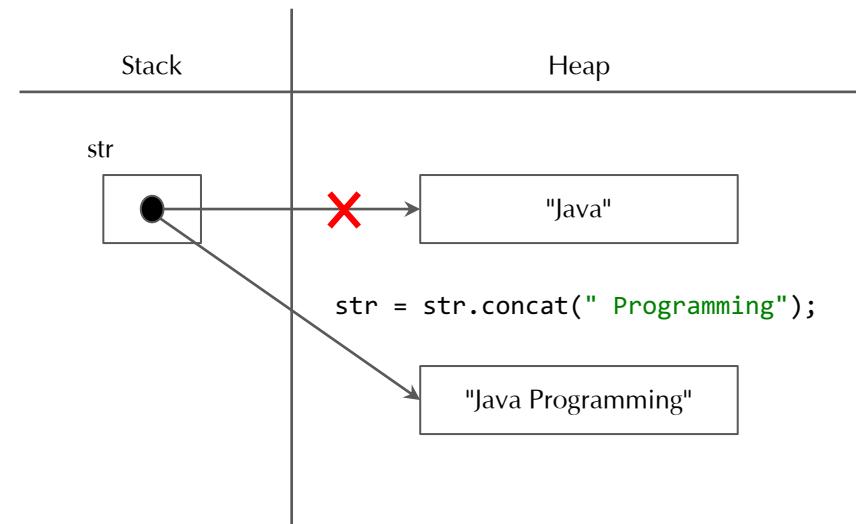
- ✓ String 클래스의 기능을 사용할 때 String 클래스의 특징을 이해하고 사용하는 것이 필요합니다.
- ✓ String 객체의 초기화 방식은 리터럴을 이용한 방식과 객체 생성을 통한 초기화 방식 2가지입니다.
- ✓ String은 불변(immutability)의 특성을 갖고 있으며 내부적으로 String Pool을 통해 문자열 상수를 관리합니다.
- ✗ 문자열의 변경이 빈번한 로직에서 String의 사용은 메모리 누수(memory leak)이 발생할 수 있기 때문에 주의합니다.

```
String str1 = "Java Programming";
String str2 = new String("Java Programming");
String str3 = "Java Programming";

System.out.println(str1 == str2); // false
System.out.println(str1 == str3); // true
```

```
String str = "Java";
str = str.toUpperCase();
System.out.println(str.toString()); // JAVA
```

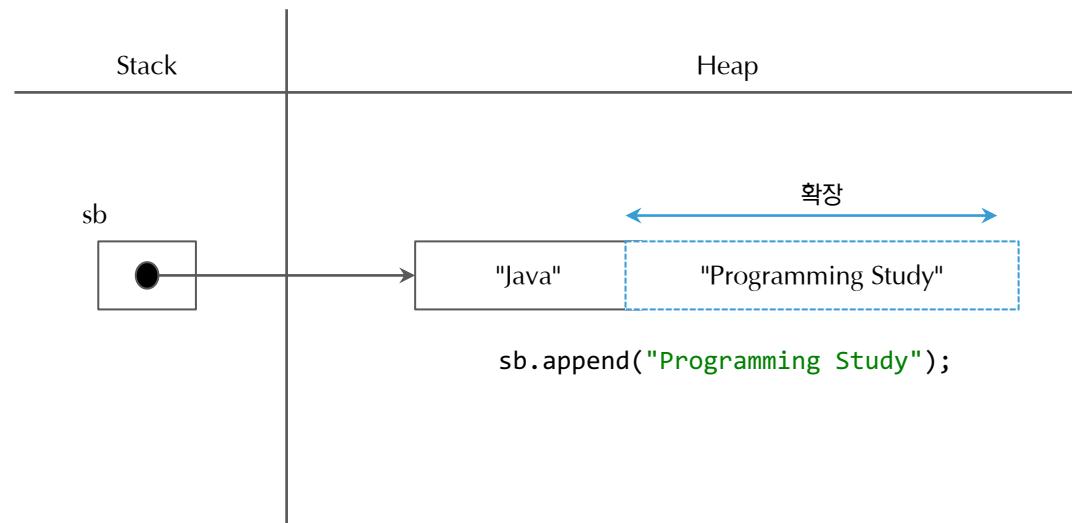
```
String str = "Java";
str = str.concat(" Programming");
System.out.println(str.toString());
// Java Programming
```



4.1 문자열 클래스[4/6] – StringBuilder, StringBuffer 클래스

- ✓ StringBuilder와 StringBuffer 클래스는 동기화(synchronized) 특성을 제외하고 모든 메소드의 기능이 동일합니다.
- ✓ 이 두 클래스는 String 클래스와 달리 가변(mutable)의 특성을 갖고 있습니다.
- ✓ StringBuilder, StringBuffer 두 클래스는 AbstractStringBuilder 추상 클래스를 상속하고 있으며 이 클래스는 내부적으로 문자열을 관리하기 위한 byte[]과 배열의 길이를 계산하기 위한 count 속성을 갖고 있습니다.

```
StringBuilder sb = new StringBuilder("Java");
System.out.println(sb.capacity()); // 20
sb.append("Programming Study"); // sb = sb.append("Programming"); X
System.out.println(sb.capacity()); // 42
```



4.1 문자열 클래스[5/6] – StringBuilder, StringBuffer 클래스의 차이

- ✓ StringBuilder 클래스와 StringBuffer 클래스의 유일한 차이는 동기화 처리 여부입니다.
- ✓ StringBuffer 클래스는 멀티스레드(Multi Thread) 프로그램에서 데이터에 대한 동기화 문제가 발생하지 않도록 대부분의 메소드에서 동기화 처리를 하고 있습니다.
- ✓ 이런 동기화 처리 과정은 성능에 영향을 주기 때문에 단일스레드(Single Thread) 프로그램에서는 StringBuilder 클래스를 사용합니다.

StringBuilder.java

```
public int compareTo(StringBuilder another) { ... }

public StringBuilder append(Object obj) { ... }
```

StringBuffer.java

```
public synchronized int compareTo(StringBuffer another) { ... }

public synchronized StringBuffer append(String str) { ... }
```

4.1 문자열 클래스[6/6] – + 연산자

- ✓ 자바의 + 연산자는 피연산자 문자열일 경우 두 피연산자를 문자열로 연결해 주는 기능을 갖고 있습니다.
- ✓ 문자열의 연결은 String 클래스의 concat(), StringBuilder 클래스의 append()와 동일한 기능입니다.
- ✓ Java 8까지 문자열에 대한 + 연산은 StringBuilder 클래스의 append() 기능을 이용해 수행했습니다.
- ✓ Java 9부터는 StringConcatFactory라는 새로운 클래스의 기능을 이용해 문자열에 대한 + 연산을 진행합니다.

```
String str = new String("Java");
str = str.concat("Programming");
str = str + "Programming";
```

Java 8 version Byte Code

```
L1
LINE NUMBER 20 L1
ALOAD 1
LDC "Programming"
INVOKE VIRTUAL java/lang/String.concat (Ljava/lang/String;)Ljava/lang/String;
ASTORE 1
L2
LINE NUMBER 22 L2
NEW java/lang/StringBuilder
DUP
INVOKE SPECIAL java/lang/StringBuilder.<init> ()V
ALOAD 1
INVOKE VIRTUAL java/lang/StringBuilder.append (Ljava/lang/String;)Ljava/lang/StringBuilder;
LDC "Programming"
INVOKE VIRTUAL java/lang/StringBuilder.append (Ljava/lang/String;)Ljava/lang/StringBuilder;
INVOKE VIRTUAL java/lang/StringBuilder.toString ()Ljava/lang/String;
ASTORE 1
```

Java 11 version Byte Code

```
L1
LINE NUMBER 20 L1
ALOAD 1
LDC "Programming"
INVOKE VIRTUAL java/lang/String.concat (Ljava/lang/String;)Ljava/lang/String;
ASTORE 1
L2
LINE NUMBER 21 L2
ALOAD 1
INVOKE DYNAMIC makeConcatWithConstants(Ljava/lang/String;)Ljava/lang/String; [
// handle kind 0x6 : INVOKESTATIC
java/lang/invoke/StringConcatFactory.makeConcatWithConstants( ...
...
...
...
ASTORE 1
```

4.2 Wrapper 클래스(1/2) – Wrapper 클래스의 이해

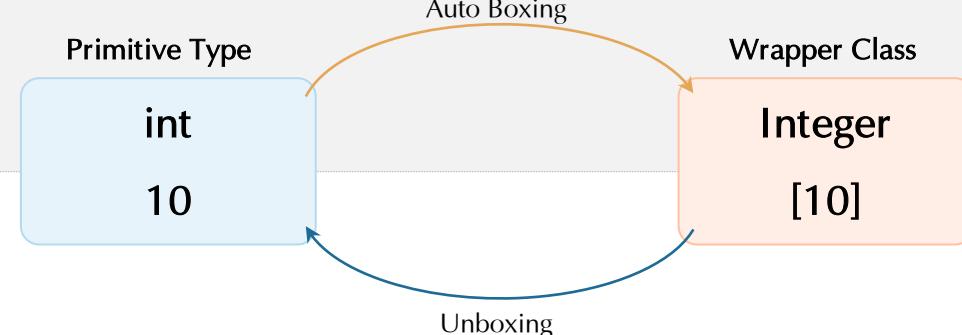
- ✓ 자바 프로그램에서 관리하는 데이터의 기본 단위는 객체입니다.
- ✓ 자바에서는 int, double 등과 같은 기본 데이터 타입(primitive data type)들을 객체로 관리 할 수 있도록 하는 클래스들을 제공하며 이 클래스들을 Wrapper 클래스라고 합니다.
- ✓ Wrapper 클래스들은 기본 데이터 타입에 대한 객체화와 함께 다양한 기능들을 정의하고 있습니다.

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

4.2 Wrapper 클래스(2/2) – Auto Boxing, Unboxing

- ✓ 기본 데이터 타입을 Wrapper 클래스로 감싸거나 Wrapper 클래스가 갖고 있는 기본 데이터 타입을 다시 꺼내는 과정은 Auto Boxing, Unboxing 기능을 통해 손쉽게 구현할 수 있습니다.

```
Integer intWrap = Integer.valueOf(10);  
int number = intWrap.intValue();  
  
Integer intWrap = 10; // Auto Boxing  
int number = intWrap; // Unboxing
```



- ✓ Wrapper 이외에도 실수 계산과 같은 오차가 발생할 수 있는 연산에는 BigInteger, BigDecimal과 같은 클래스의 기능을 사용합니다.

```
double da = 3.14;  
System.out.println(da + 1); // 4.140000000000001  
  
System.out.println(BigDecimal.valueOf(3.14).add(BigDecimal.valueOf(1))); // 4.14
```

4.3 예외처리(1/7) – 개요

- ✓ 예외란 프로그램 실행 도중에 발생하는 ‘예외적인 상황’이며, 이러한 상황을 처리하는 것이 예외처리입니다.
- ✓ Java에서는 예외처리를 위한 문법을 제공합니다. (**try ~ catch** 문)
- ✓ 예외적인 상황이란?
 - 파일을 읽으려고 하는데 해당 파일이 존재하지 않는 경우
 - 나눗셈을 하려고 두 수를 입력 받았는데, 제수(나누는 수)가 0인 경우
- ✓ **if** 문을 이용한 예외처리의 문제점
 - if 문을 이용하여 예외처리를 할 경우 프로그램의 주 흐름을 구성하는 코드와 예외처리 코드가 혼재되어 코드가 복잡해집니다.

```
...
if (num == 0) {
    // 예외 상황에 대한 처리 코드
    System.out.println("결과는 0이 될 수 없습니다.");
}
...
```

4.3 예외처리(2/7) – 예외 클래스의 종류

✓ 예외 클래스의 종류

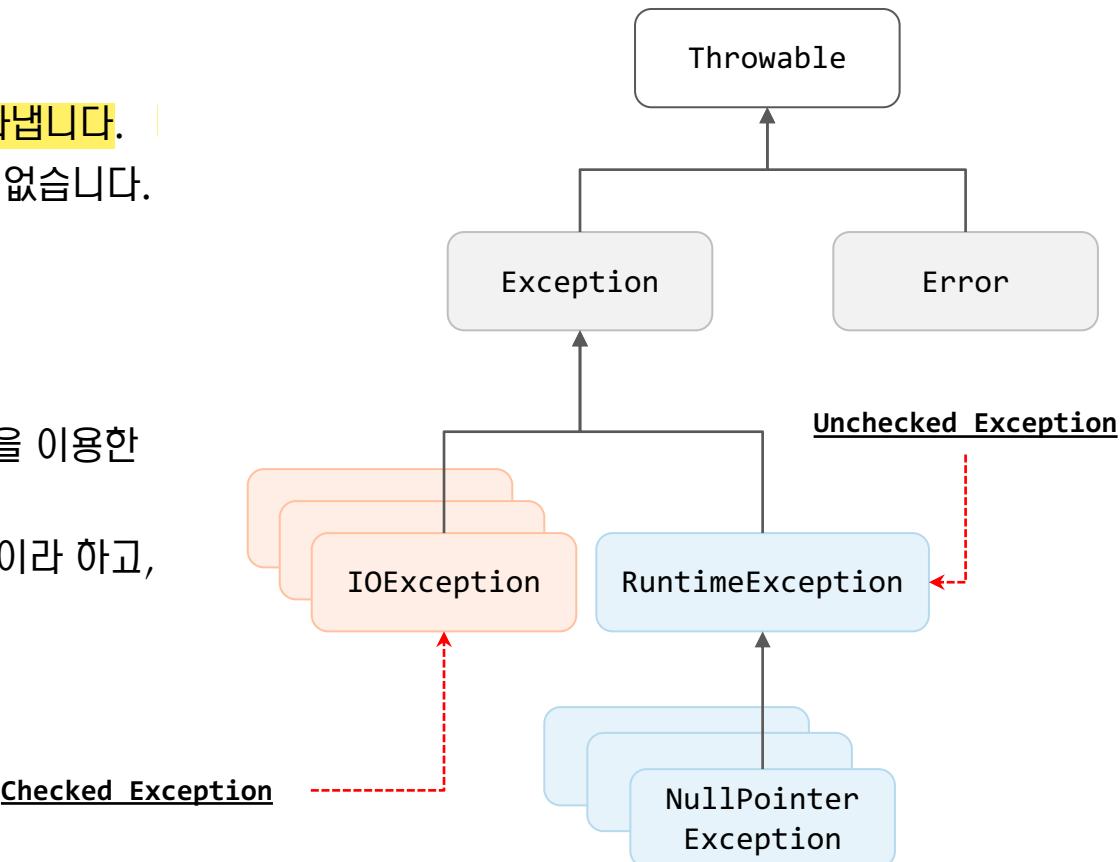
- 예외 클래스의 최상위 클래스는 Throwable 클래스입니다.
- Throwable의 하위 클래스는 Exception과 Error 클래스가 있습니다.

✓ Error 클래스

- Error 클래스를 상속한 클래스들은 매우 심각한 오류상황을 나타냅니다.
- 자바프로그램 외에서 발생한 오류로 프로그램 내에서 해결할 수 없습니다.
- 예) OutOfMemoryError 등

✓ RuntimeException

- Exception 클래스와는 달리, try ~ catch 문 또는 throws 절을 이용한 예외처리가 필요하지 않습니다.
- RuntimeException 계열의 예외를 Unchecked Exception이라 하고, 그 밖의 예외 클래스를 Checked Exception이라고 합니다.



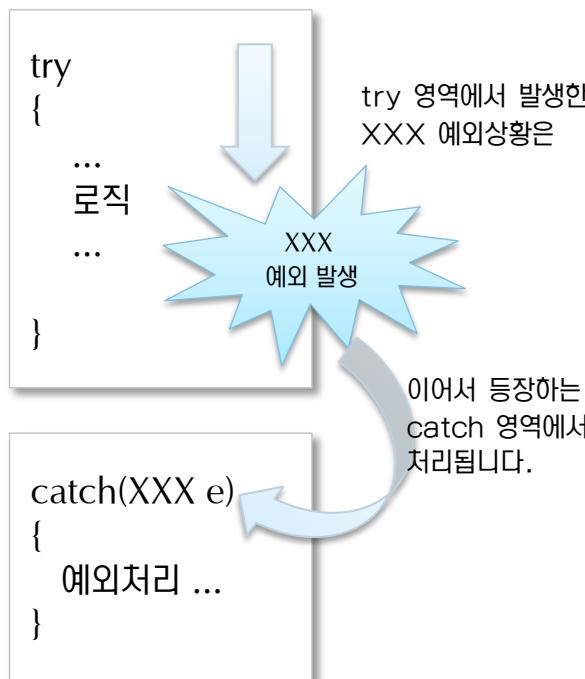
4.3 예외처리(3/7) – try ~ catch 구문(1/2)

✓ Java는 예외처리를 위하여 try ~ catch 문을 제공합니다.

- try 블록은 예외가 발생할 수 있는 영역을 감싸고, catch 블록에는 발생한 예외를 처리하는 코드를 작성합니다.

✓ Java에서 발생시키는 예외 클래스의 예

- `ArrayIndexOutOfBoundsException` : 배열 접근 시 잘못된 인덱스 값을 사용하는 경우
- `ClassCastException` : 허용할 수 없는 형변환 연산을 진행하는 경우
- `NullPointerException` : 참조변수가 null로 초기화 된 상황에서 메소드를 호출하는 경우



```
try
{
    System.out.println("나눗셈 결과: " + (num1/num2));
}
catch(ArithmeticException e)
{
    System.out.println("나눗셈 연산오류");
    System.out.println(e.getMessage());
}
```

Java는 0으로 나눌 때
ArithemticException
예외를 발생시킵니다.

Calculator.java

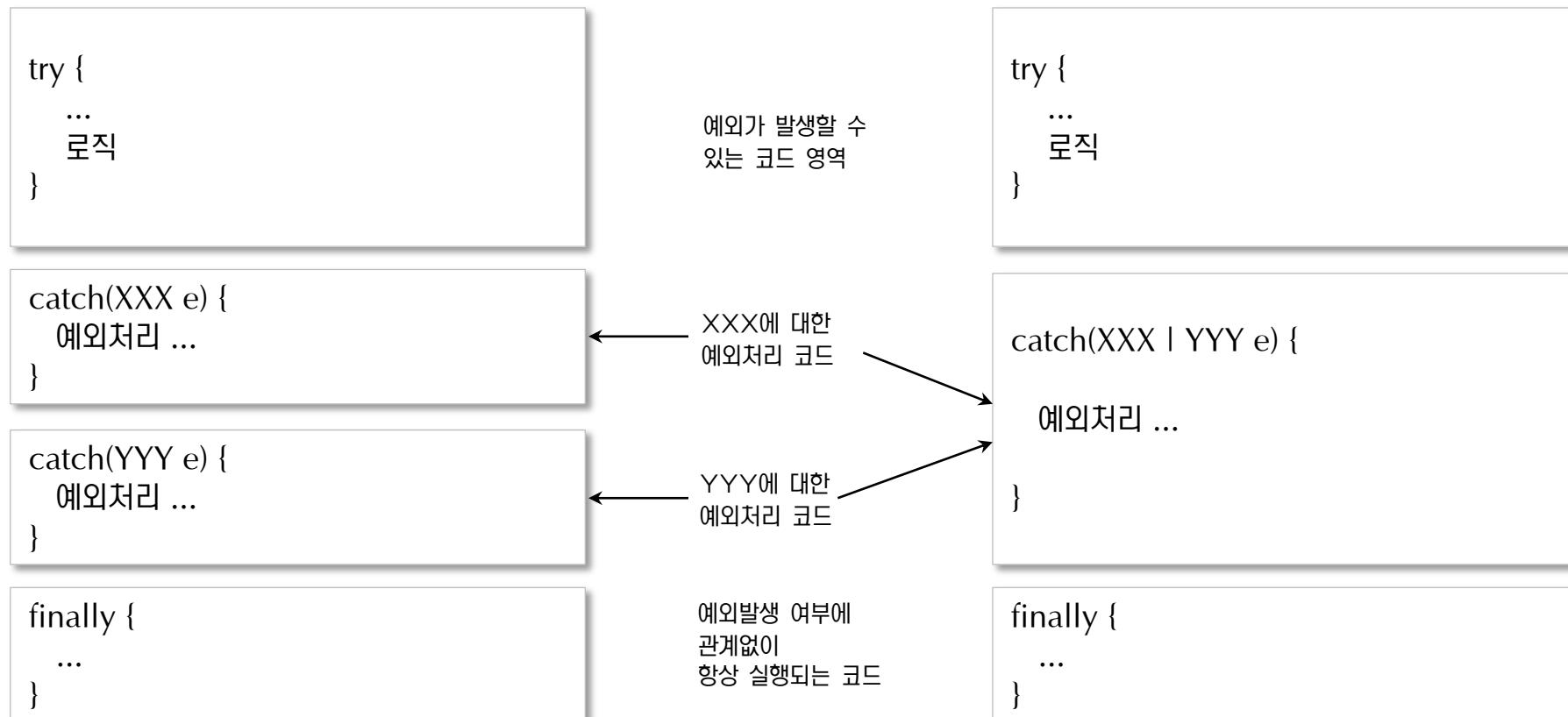
다음은, 제수(num2)가 0인 경우 실행결과입니다.

나눗셈 연산오류
/ by zero

실행결과

4.3 예외처리(4/7) – try ~ catch 구문(2/2)

- ✓ try 영역에서 발생하는 예외가 여러 개인 경우 다수의 catch문을 작성할 수 있습니다.
- ✓ 예외 발생여부와 상관없이 항상 실행해야 할 코드가 있는 경우 finally 구문을 사용합니다.
- ✓ Java7 부터는 다중 예외 처리를 위한 파이프(|)를 제공합니다.



< JAVA 7 >

4.3 예외처리(5/7) – 사용자 정의 예외 클래스(1/2)

- ✓ 문법적으로 문제가 되는 상황에서는 JVM이 적절한 예외를 발생시킵니다.
- ✓ 그러나, 논리적으로 문제가 되는 경우에는 상황에 맞는 예외클래스를 직접 작성해야 합니다.
- ✓ 예외 클래스는 다음과 같이 작성합니다.
 - Exception 클래스를 상속합니다.
 - 예외클래스의 생성자에 메시지를 추가합니다.

```
public class NotAvailableAgeException extends Exception {  
  
    public NotAvailableAgeException() {  
        super("유효하지 않은 나이가 입력되었습니다.");  
    }  
}
```

✓ throw 키워드

- 예외가 발생했음을 JVM에 알리기 위해 사용합니다.
- 문법 : throw 예외클래스의 인스턴스;

✓ 예외의 전파 (throws)

- 발생된 예외상황을 메소드를 호출한 곳으로 전달하기 위해 사용합니다.

```
public class AgeInputExample {  
    public static void main(String[] args) {  
        System.out.print("나이를 입력하세요. : ");  
        try {  
            int age = readAge();  
            System.out.println("당신의 나이는 " + age + "세입니다.");  
        } catch (NotAvailableAgeException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
  
    private static int readAge() throws NotAvailableAgeException {  
        Scanner scanner = new Scanner(System.in);  
        int age = scanner.nextInt();  
  
        if (age < 0) {  
            throw new NotAvailableAgeException();  
        }  
        return 0;  
    }  
}
```

AgeInputExample.java

나이를 입력하세요. : -1
유효하지 않은 나이가 입력되었습니다.

실행결과

4.3 예외처리(6/7) – 사용자 정의 예외 클래스(2/2)

- ✓ Exception 클래스는 printStackTrace() 메소드를 제공합니다.
- ✓ printStackTrace() 메소드는 예외가 발생한 클래스를 포함한 상세한 정보를 보여줍니다.
- ✓ printStackTrace() 메소드는 개발 시에만 사용하는 것이 좋습니다.

```
System.out.print("나이를 입력하세요. : ");

try {
    int age = readAge();
    System.out.println("당신의 나이는 " + age + "세 입니다.");
} catch (NotAvailableAgeException e) {
    System.out.println(e.getMessage());
    e.printStackTrace();
}
```

나이를 입력하세요. : -1
유효하지 않은 나이가 입력되었습니다.
chapter4.NotAvailableAgeException: 유효하지 않은 나이가 입력되었습니다.
 at chapter4.AgeInputExample.readAge(AgeInputExample.java:26)
 at chapter4.AgeInputExample.main(AgeInputExample.java:12)

실행결과

➔ printStackTrace() 메소드를 사용하면 예외를 발생시킨 코드정보를 보여주기 때문에 오류상황을 분석하는데 도움이 됩니다.

4.3 예외처리(7/7) – 예외를 처리하는 방법

✓ 예외가 발생했을 때 이를 처리하는 방식은 3가지가 있습니다.

- 예외 복구 : 특정 기능에서 예외가 발생했을 때 재시도를 통해 예외를 복구하는 방법.
- 예외 회피 : 해당 기능에서 예외처리를 수행하지 않고 기능을 요청한 곳으로 예외를 보내는 방법.
- 예외 전환 : 발생한 예외를 다른 종류의 예외로 전환하여 보내는 방법.

```
int maxRetry = MAX_RETRY;
while(maxRetry-- > 0){
    try{
        // 예외 발생 가능 코드;
        return; // 이상없는 완료
    }catch(Exception e){
        // 예외 로그 출력 및 재시도를 위한 대기
    }finally{
        // 리소스 반납
    }
}
throw new RetryFailedException() // MAX_RETRY 초과시 예외 발생
```

예외 복구

```
public void add() throws SQLException {
    ...
}
```

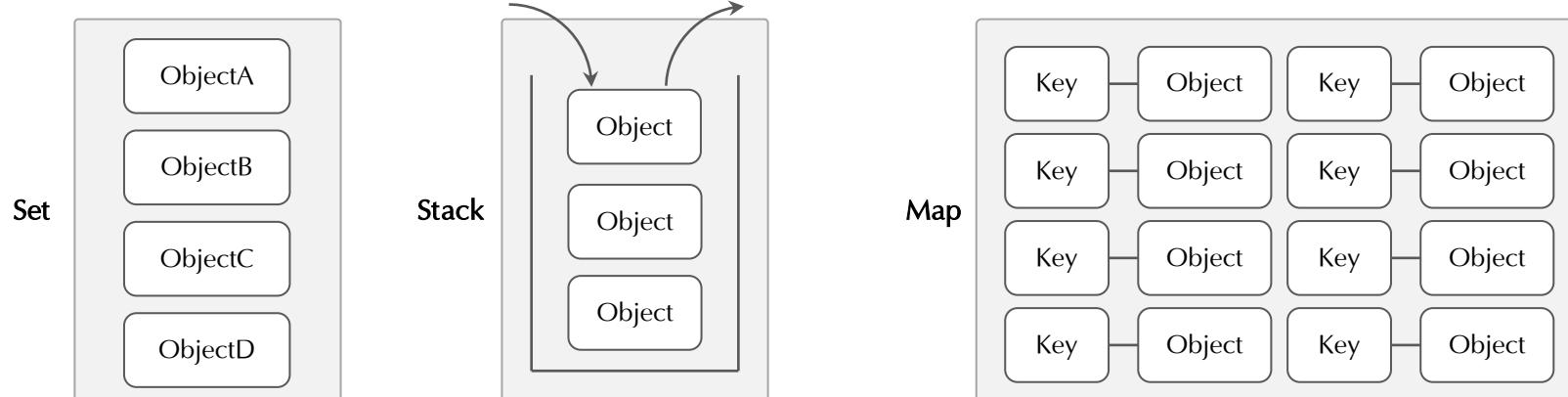
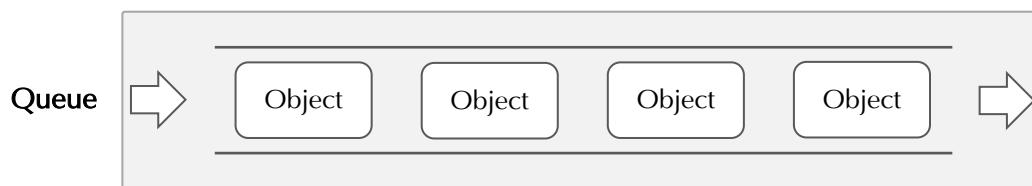
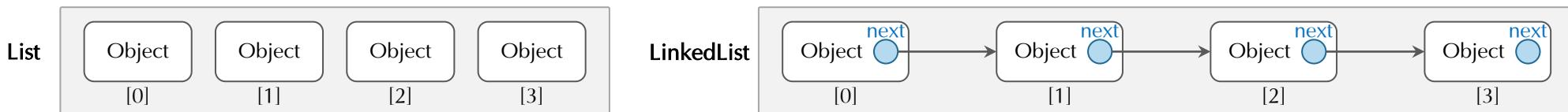
예외 회피

```
try{
    ...
}catch(SQLException e){
    ...
    throw RuntimeException("Message~");
}
```

예외 전환

4.4 Java Collection Framework의 이해(1/3)

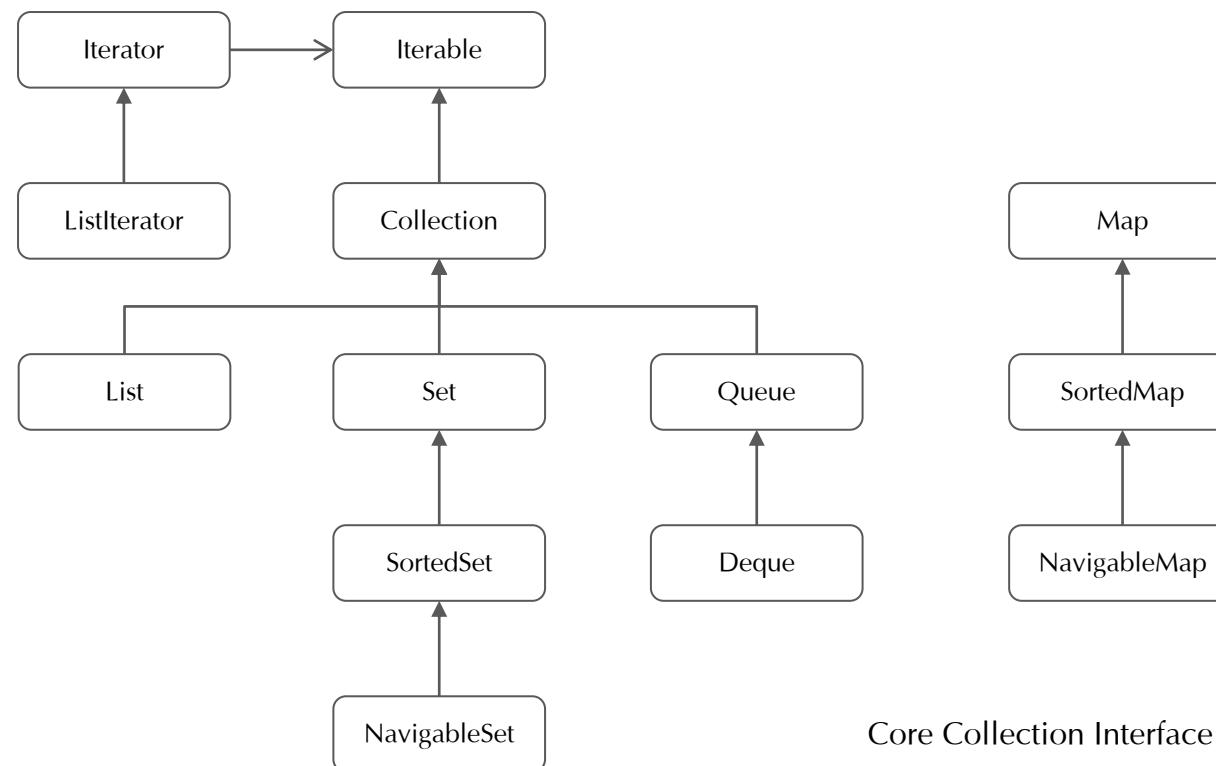
- ✓ Java Collection Framework는 객체들을 관리하기 위해 사용하는 컨테이너 클래스들의 집합입니다.
- ✓ 컨테이너 클래스들의 종류는 크게 List, Set, Queue, Map 계열로 구분합니다.
- ✓ 각 계열에 따라 객체들을 저장하고 관리하는 방식에 차이가 있습니다.
- ✓ 컬렉션 프레임워크 클래스들을 잘 활용하기 위해서는 각 클래스들의 특징과 저장 방식에 대한 이해가 필요합니다.



4.4 Java Collection Framework의 이해(2/3)

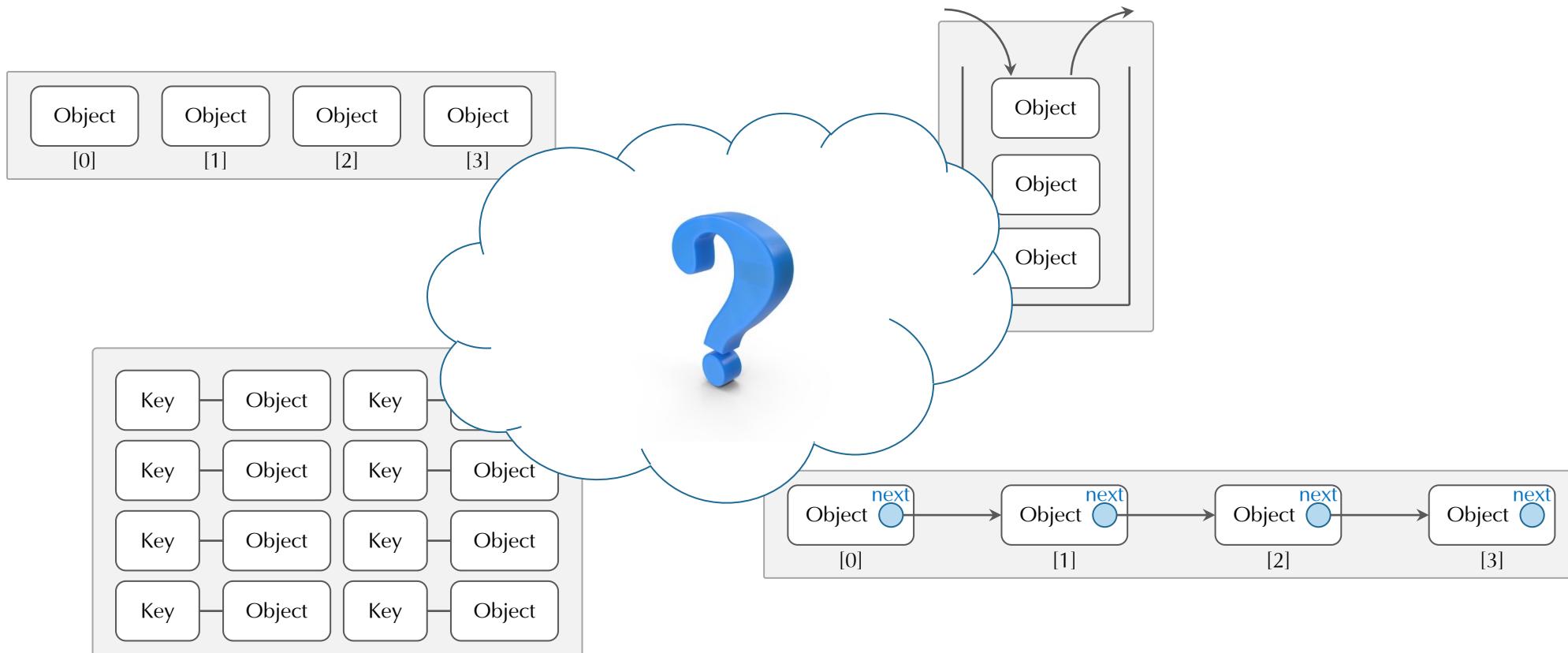
✓ 컬렉션 관련 인터페이스들과 클래스들은 `java.util.*` 패키지(package)에 포함되어 있습니다.

- Collection interface : 순서 없는 객체들의 집합입니다.
- List interface : 순차적 나열로 순서 지정이 가능한 객체들의 집합입니다.
- Set interface : 중복을 허용하지 않는 객체들의 집합입니다.
- Queue interface : FIFO(First In First Out) 처음 저장하는 객체가 가장 먼저 나오는 구조의 집합입니다.
- Map : 키와 그 키에 대응하는 객체들로 이루어진 집합입니다.



4.4 Java Collection Framework의 이해(3/3)

- ✓ 저장 관리하고자 하는 객체들의 특성에 따라 사용할 컬렉션 클래스를 선택합니다.
- ✓ 컬렉션 클래스를 선택할 때 필요한 대표적인 고려사항은 다음과 같습니다.
 - 저장 객체의 추가, 삭제와 같은 데이터의 변경이 자주 발생하는지
 - 저장 객체의 빠른 탐색이 최우선인지 필요한지
 - 저장 메모리를 최대한 효율적으로 사용하고자 하는지



4.5 제네릭의 이해(1/2) – 개요

- ✓ 제네릭(Generic)은 Java 5에서 추가된 기능으로 특히 객체를 수집, 관리하는 컬렉션을 이용할 때 반드시 사용합니다.
- ✓ 제네릭을 사용하면 데이터를 저장하는 시점에 어떤 데이터를 저장할 것인지 명시할 수 있습니다.
- ✓ 이를 통해 사용하고자 하는 데이터의 타입을 명확히 선언할 수 있고, 정확한 데이터의 사용 여부를 컴파일 시점에 확인할 수 있습니다.

```
public class Box {  
    private Object item;  
  
    public Box(Object item){  
        this.item = item;  
    }  
  
    public Object getItem(){  
        return item;  
    }  
}
```

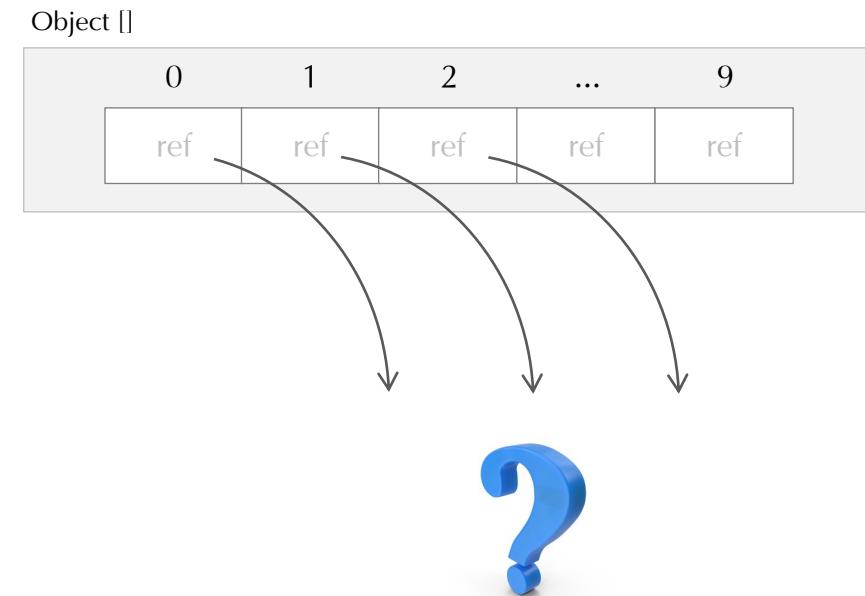
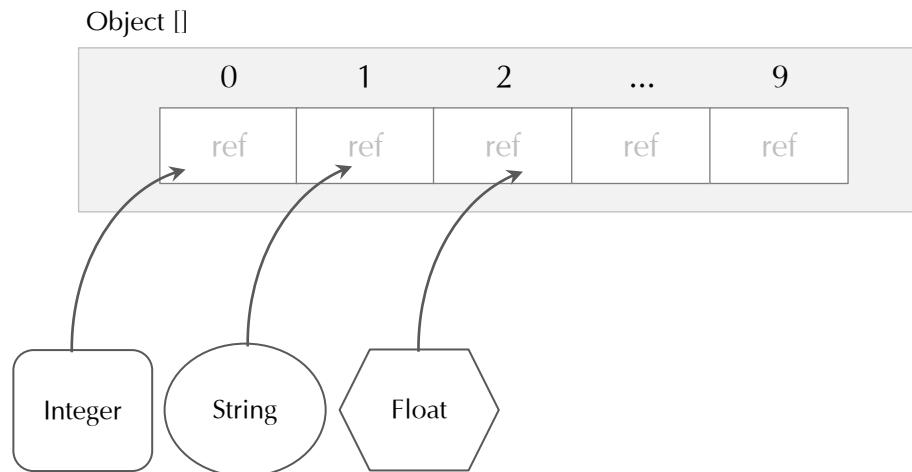
```
Box box = new Box(new Apple(10));  
Apple apple = (Apple)box.getItem();  
System.out.println(apple.getSugarContent());
```

```
public class Box<T> {  
    private T item;  
  
    public Box(T item){  
        this.item = item;  
    }  
  
    public T getItem(){  
        return item;  
    }  
}
```

```
Box<Apple> box = new Box<Apple>(new Apple(10));  
Apple apple = box.getItem();  
System.out.println(apple.getSugarContent());
```

4.5 제네릭의 이해[2/2] – 제네릭과 컬렉션

- ✓ Object 클래스는 최상위 클래스로서 Java의 모든 클래스를 참조 할 수 있습니다.
- ✓ 모든 클래스를 참조 할 수 있다는 것은 분명 편리할 수 있지만 오류를 발생시킬 수 있는 여지 또한 매우 큽니다.
- ✓ 예를 들어, Object 배열에 다양한 객체의 참조를 넣었을 때를 생각해 볼 수 있습니다.
- ✓ 객체의 구분없이 배열에 담을 수 있다는 것은 담을 때의 편리성은 있지만 다시 꺼낼 때는 문제가 발생합니다.



4.6 Collection 인터페이스[1/2]

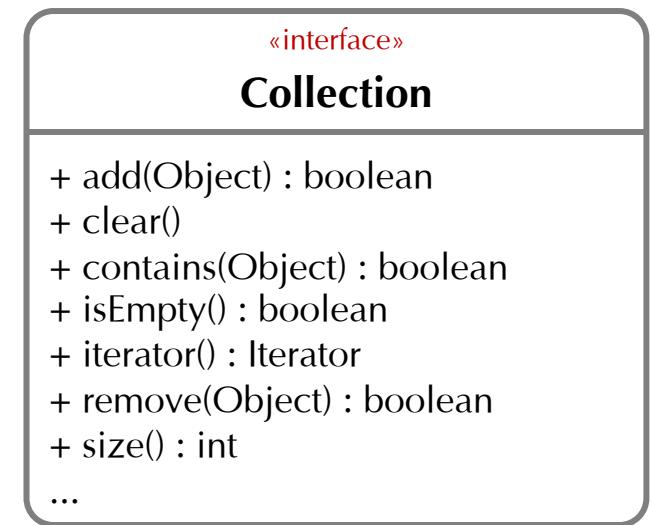
✓ java.util.Collection 인터페이스

- 컬렉션 프레임워크의 최상위 인터페이스입니다.
- 요소(객체)에 대한 삽입, 삭제, 탐색의 기능을 정의합니다.

✓ 주요 메소드

- add() : 새로운 요소를 삽입합니다. 중복 요소를 허용하지 않는 경우 false를 반환합니다.
- clear() : 모든 요소를 제거합니다.
- contains() : 파라미터로 전달되는 객체가 요소로 존재하는지를 반환합니다.
- isEmpty() : 해당 컬렉션이 포함하고 있는 요소가 0인지를 반환합니다.
- remove() : 파라미터로 전달되는 객체를 제거합니다. 절달되는 객체가 요소로 존재하지 않다면 false를 반환합니다.
- size() : 현재 포함하고 있는 요소의 개수를 반환합니다.
- iterator() : 해당 컬렉션이 포함하고 있는 요소들을 순회하기 위한 iterator 객체를 반환합니다.

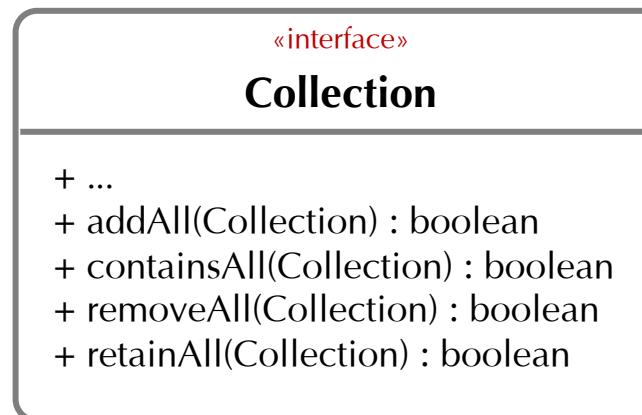
✓ Java 8 버전 이후 Stream 관련 디폴트 메소드들이 추가 되었습니다.



4.6 Collection 인터페이스[2/2]

✓ **java.util.Collection** 추가 기능과 같은 주요 메소드는 다음과 같습니다.

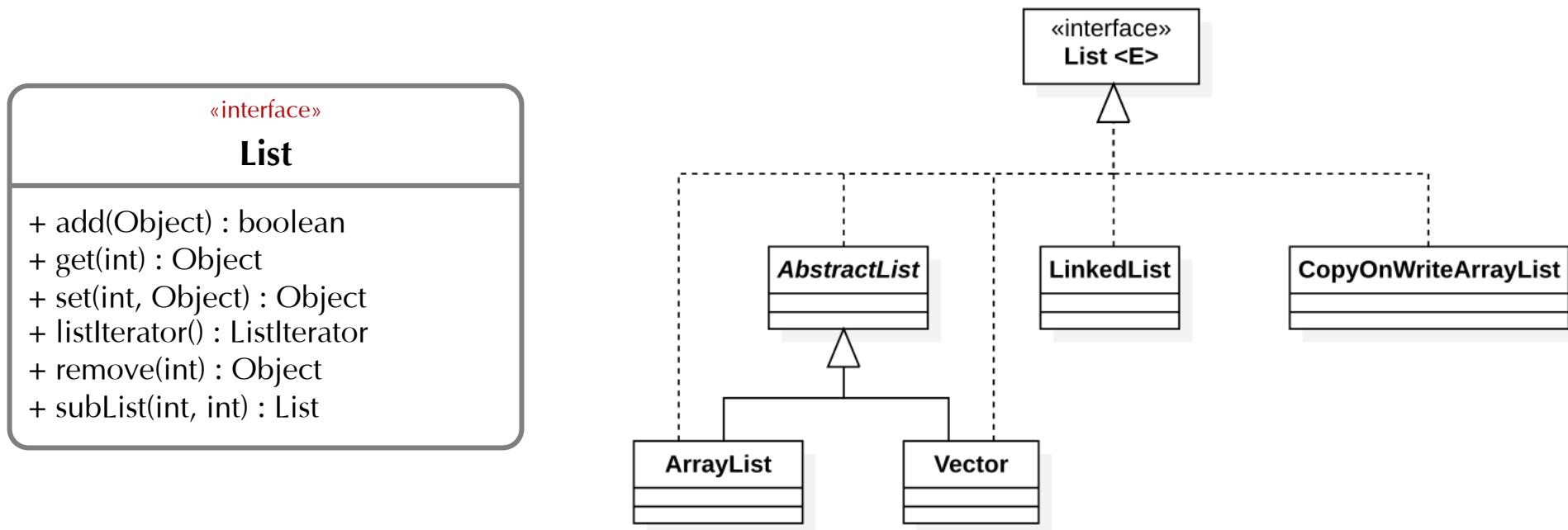
- `addAll()` : 파라미터로 전달되는 컬렉션의 모든 요소를 추가합니다.
- `containsAll()` : 파라미터로 전달되는 컬렉션의 모든 요소를 현재 포함하고 있다면 `true`를 반환합니다.
- `removeAll()` : 파라미터로 전달되는 컬렉션의 요소들과 일치하는 요소들을 모두 제거합니다.
- `retainAll()` : 파라미터로 전달되는 컬렉션의 요소들과 일치하지 않는 요소들을 모두 제거합니다.



| 메소드 | 집합연산 | 기호 | $x = \{1, 2, 3, 4, 5\}$ $y = \{1, 4, 5, 7, 9\}$ |
|-------------------------------|------|-----------------|--|
| <code>x.addAll(y)</code> | 합집합 | $x = x \cup y$ | $x = \{1, 2, 3, 4, 5, 1, 4, 5, 7, 9\}$ |
| <code>x.containsAll(y)</code> | 부분집합 | $x \supseteq y$ | false 를 리턴 |
| <code>x.removeAll(y)</code> | 상대보수 | $x = x - y$ | $x = \{2, 3, 4, 7, 9\}$ |
| <code>x.retainAll(y)</code> | 교집합 | $x = x \cap y$ | $x = \{1, 4, 5\}$ |

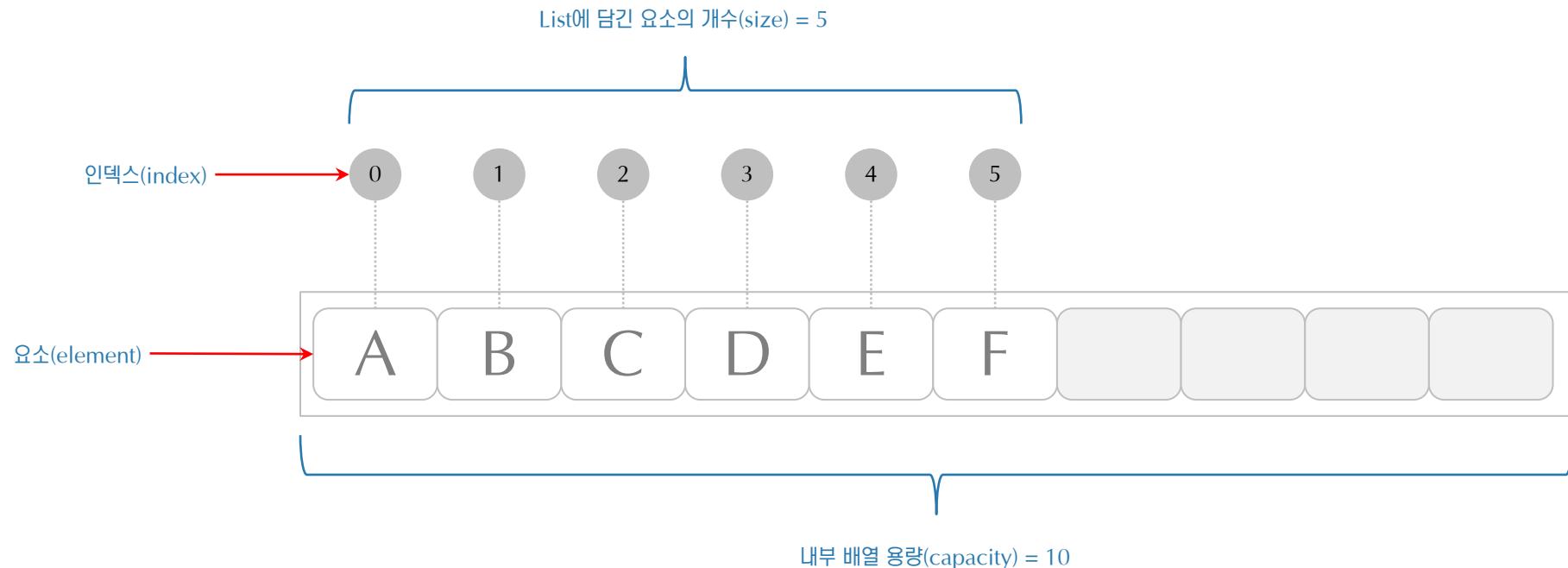
4.7 List 인터페이스(1/5) – 개요

- ✓ List 계열의 컬렉션은 저장 요소를 순차적으로 관리하며 중복된 값과 null 값을 요소로 가질 수 있습니다.
- ✓ 요소에 대한 접근은 배열과 마찬가지로 인덱스(index)를 통해 접근합니다.
- ✓ 배열 자료구조 형태로 데이터를 저장할 때 저장 데이터의 특성에 따라 적절한 List 계열의 클래스들을 활용합니다.
- ✓ List 인터페이스를 구현한 대표클래스는 **ArrayList**, **LinkedList**, **Vector** 클래스가 있습니다.



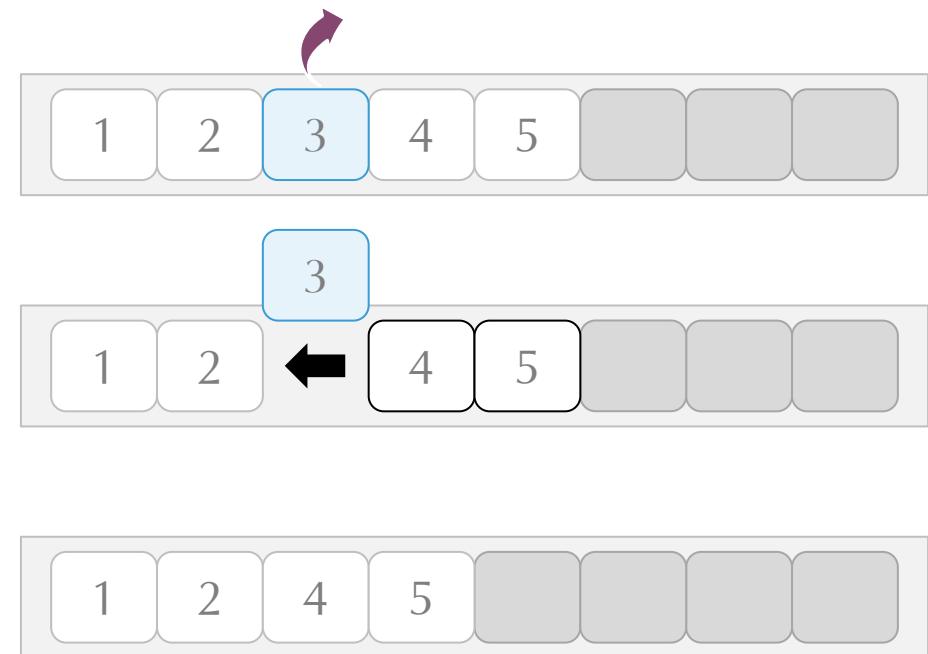
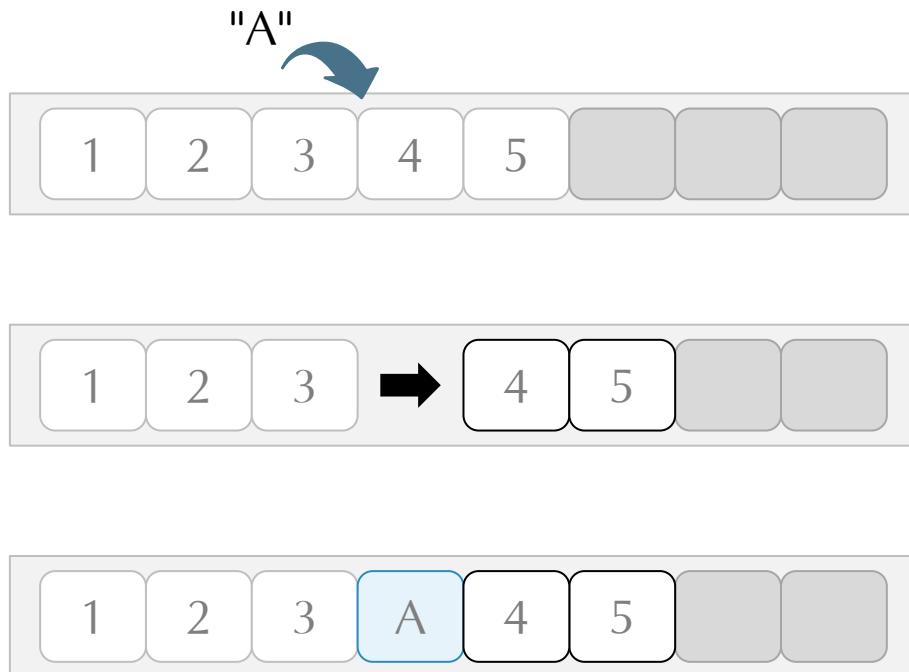
4.7 List 인터페이스(2/5) – ArrayList 클래스(1/2)

- ✓ ArrayList 클래스는 내부에 배열을 갖고 있습니다. 따라서 고정 길이 저장 공간으로 요소들을 관리합니다.
- ✓ ArrayList 클래스의 내부 배열이 요소를 담을 수 있는 용량을 capacity라고 합니다.
- ✓ 내부 배열의 용량(capacity)을 넘어 요소를 저장할 경우 내부적으로 용량을 늘린 새로운 배열을 만들어 요소를 담습니다.
- ✓ 요소에 대한 접근은 배열과 마찬가지로 인덱스(index)를 통해 접근합니다.



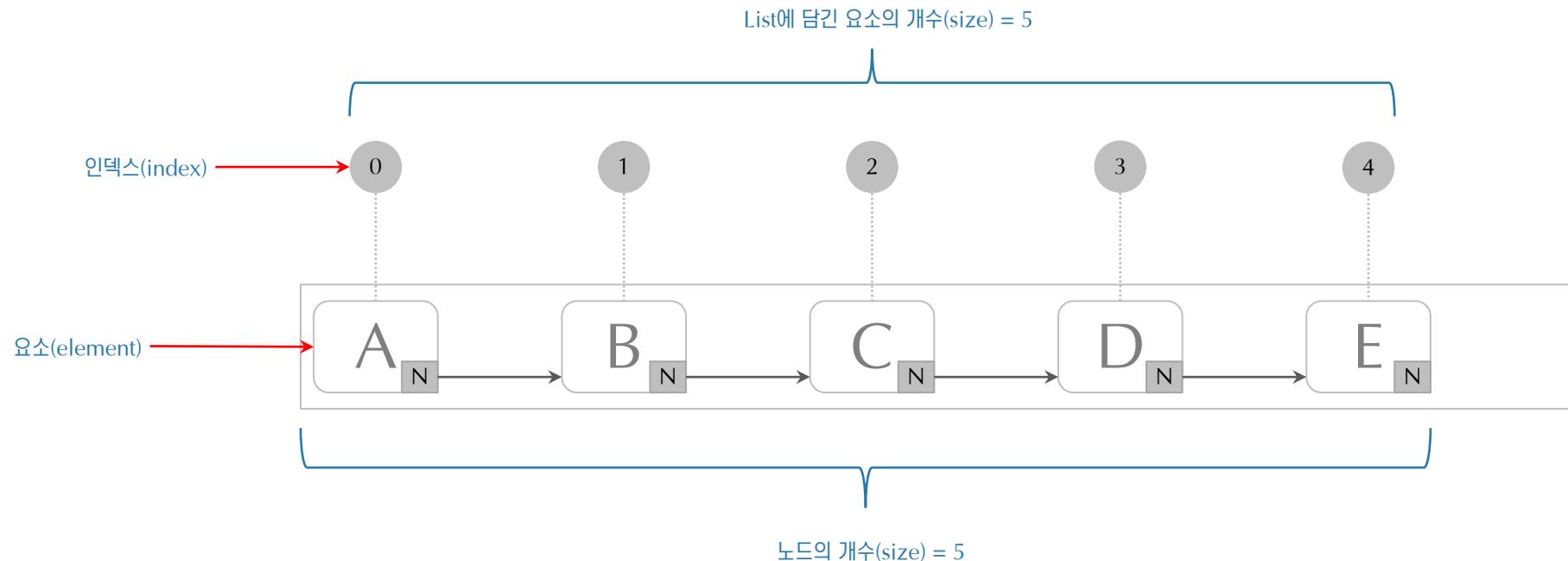
4.7 List 인터페이스(3/5) – ArrayList 클래스(2/2)

- ✓ ArrayList는 순차적으로 요소를 저장할 수 있는 메소드와 인덱스를 통해 저장할 수 있는 add() 메소드를 제공합니다.
- ✓ 특정 인덱스에 요소를 저장할 경우 기존에 저장된 요소들의 이동이 내부적으로 이루어집니다.
- ✓ 특정 요소를 삭제하기 위해서는 해당 요소의 인덱스가 필요하며 이 경우에도 저장된 요소들의 이동이 발생합니다.
- ✓ ArrayList가 갖는 이와 같은 특징(용량, 요소의 이동)은 추가, 삭제가 빈번한 데이터의 관리에는 적합하지 않습니다.



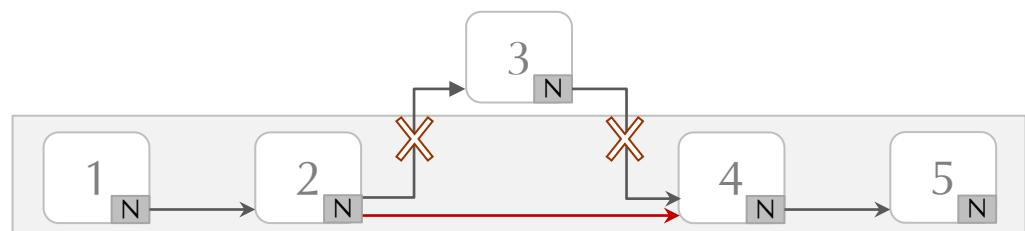
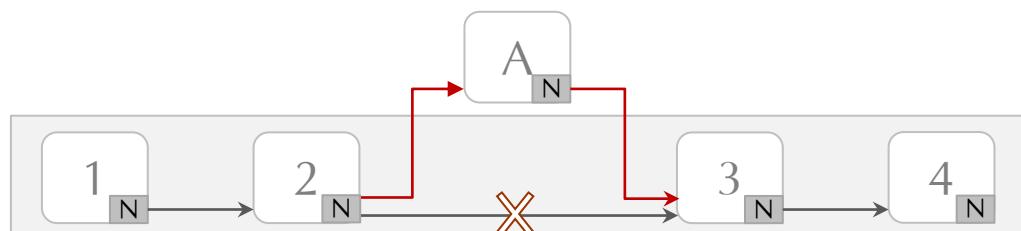
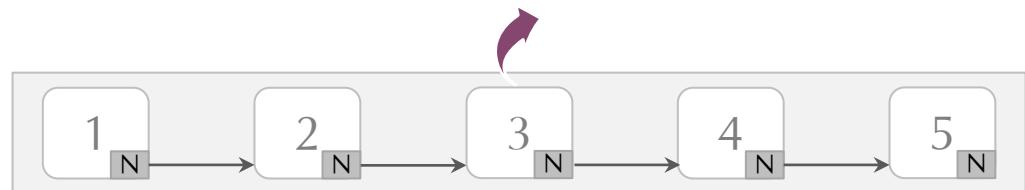
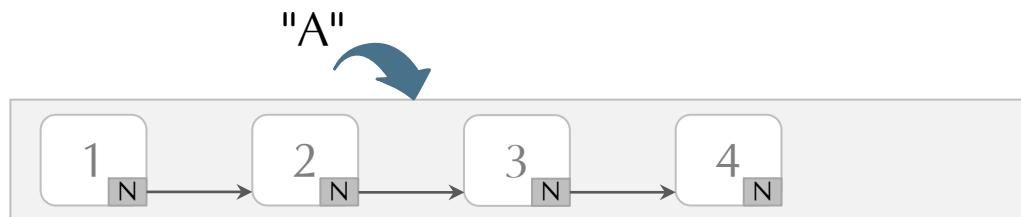
4.7 List 인터페이스(4/5) – LinkedList 클래스(1/2)

- ✓ LinkedList 클래스는 집합하는 각 요소들을 노드(node)로 표현하고, 각 노드들을 서로 연결하여 리스트를 구성합니다.
- ✓ 요소를 갖는 각 노드들은 다음 노드에 대한 참조 정보를 통해 접근합니다.
- ✓ 노드는 필요할 경우 만들어서 연결하여 사용하기 때문에, 미리 정의된 용량(capacity)의 개념이 없습니다.
- ✓ List 인터페이스를 구현한 다른 클래스들과 마찬가지로 인덱스를 통해 요소에 접근합니다.



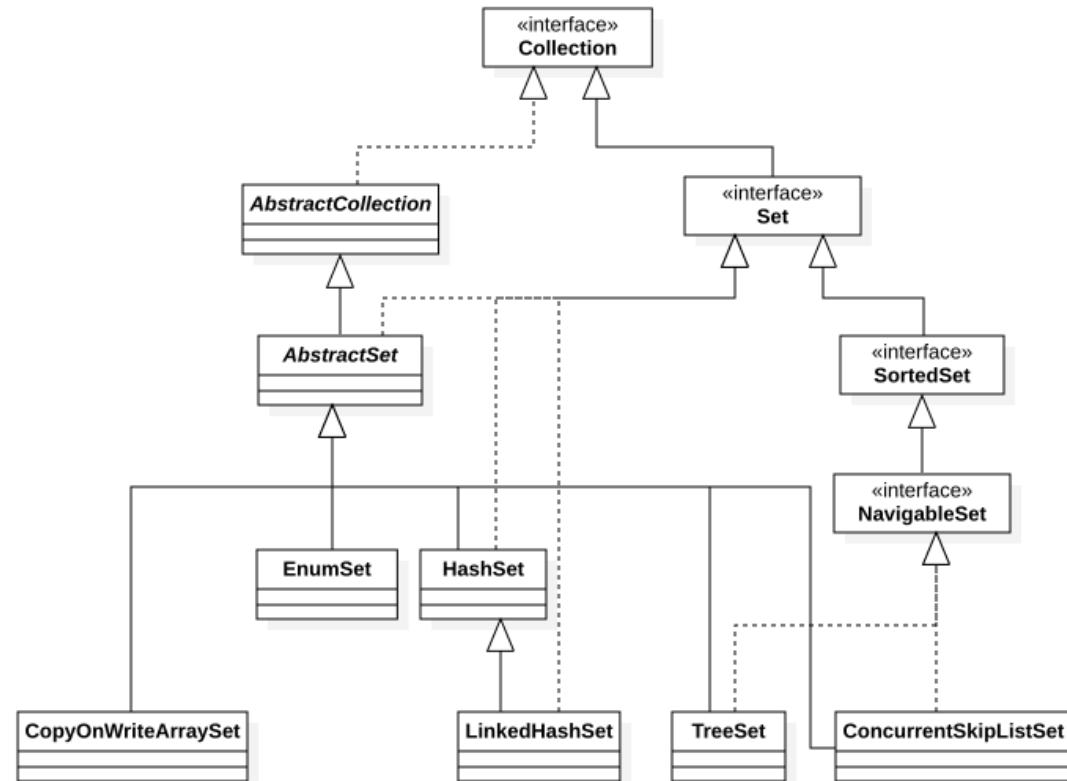
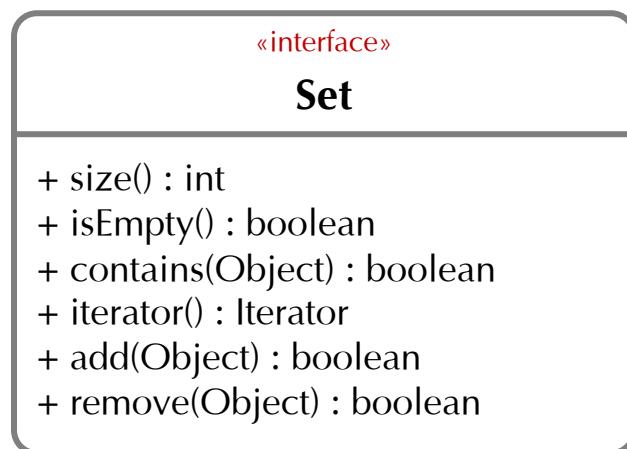
4.7 List 인터페이스(5/5) – LinkedList 클래스(2/2)

- ✓ 요소의 추가는 끈 노드의 추가를 의미하며, 일반적인 추가(add)는 순차적으로 링크를 연결하여 추가합니다.
- ✓ 인덱스를 지정해 특정 지점에 요소를 추가할 경우에는 기존 노드의 연결을 끊고 새로운 노드를 연결하여 추가합니다.
- ✓ 요소의 삭제는 삭제할 노드가 갖고 있는 다음 노드에 대한 포인트를 이전 노드가 포인트 할 수 있도록 하여 삭제합니다.
- ✓ ArrayList와 달리 요소의 추가, 삭제에 대한 부하가 적은 반면 메모리의 사용은 더 많다는 특징을 갖습니다.



4.8 Set 인터페이스(1/2) – 개요

- ✓ Set 인터페이스를 구현한 컬렉션 클래스들의 가장 큰 특징은 저장하는 요소의 중복을 허용하지 않는다는 것입니다.
- ✓ Set 인터페이스의 구현 클래스들은 equals() 메소드를 이용해 저장 요소의 중복을 검사합니다.
- ✓ Set 인터페이스가 정의하고 있는 추상 메소드 중에는 단일 요소를 꺼내기 위한 get() 메소드가 존재하지 않습니다.
- ✓ Set 인터페이스를 구현한 주요 클래스는 HashSet, LinkedHashSet, TreeSet 등이 있습니다.



4.8 Set 인터페이스(2/2) – 주요 클래스

- ✓ Set 인터페이스의 구현체 HashSet은 집합 하는 요소의 중복을 허용하지 않고 입력 순서를 유지하지 않습니다.
- ✓ HashSet 클래스는 순서에 상관없이 어떤 데이터가 존재하는지의 여부를 확인하는 용도로 많이 사용됩니다.
- ✓ 집합 요소의 정렬이 필요할 경우 TreeSet이나 LinkedHashSet 클래스를 사용합니다.
- ✓ Set 계열의 클래스에서 전체 집합 요소를 순회하고자 할 경우 Iterator를 이용합니다.

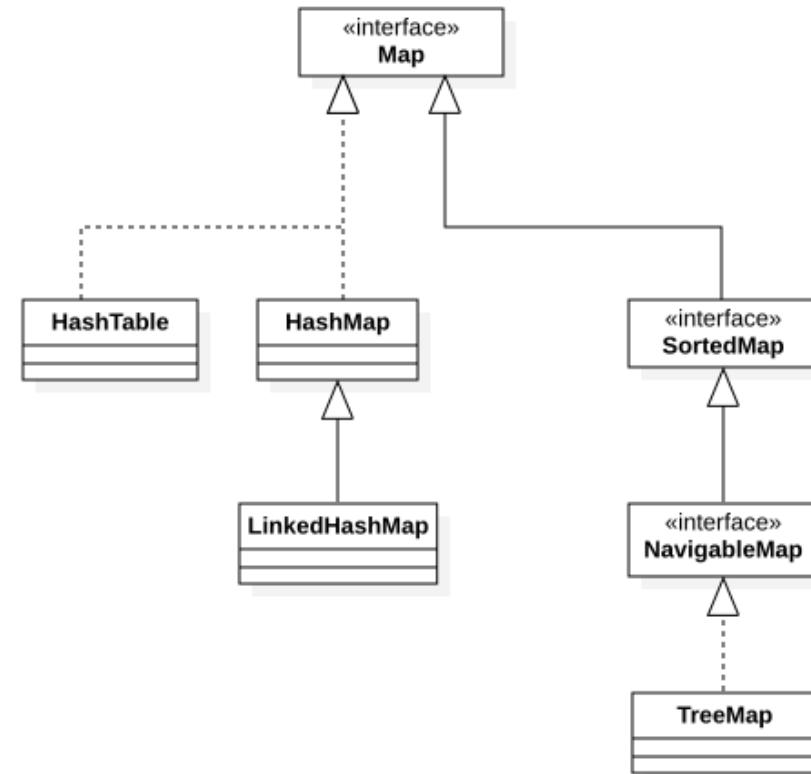
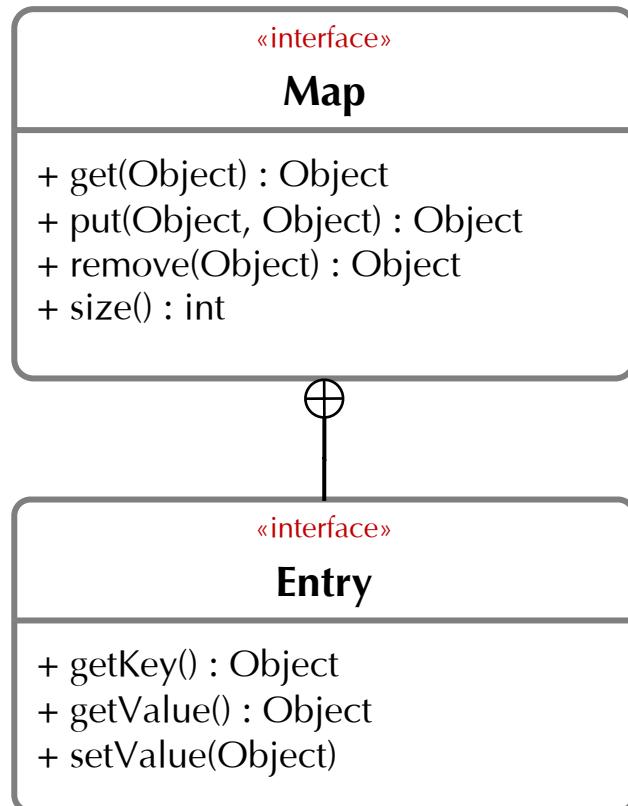
```
Random rand = new Random(47);
Set<Integer> intSet = new HashSet<Integer>();

for (int i = 0; i < 10000; i++) {
    intSet.add(rand.nextInt(30));
}
System.out.println(intSet);
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 16, 19,
18, 21, 20, 23, 22, 25, 24, 27, 26, 29, 28]
```

4.9 Map 인터페이스(1/2) – 개요

- ✓ Map 인터페이스가 갖는 대표적인 특성은 요소를 저장하기 위해서는 유일한 키(key)와 함께 저장해야 한다는 것입니다.
- ✓ List, Set, Queue와 달리 Map 인터페이스는 Collection 인터페이스를 상속하지 않습니다.
- ✓ Map 인터페이스는 내부에 Entry 인터페이스를 가지고 있으며 Entry는 키와 값을 가진 객체의 순서 쌍입니다.
- ✓ Map 인터페이스의 주요 구현 클래스는 HashMap, LinkedHashMap, TreeMap 등이 있습니다.



4.9 Map 인터페이스[2/2] – 주요 클래스

- ✓ HashMap에 요소를 저장하기 위해서는 키(key)와 값(value)이 필요하며 값의 경우 중복이 가능합니다.
- ✓ HashMap 클래스는 저장되는 요소의 순서를 보장하지 않습니다.
- ✓ LinkedHashMap 클래스는 요소를 추가한 순서를 보장합니다.
- ✓ TreeMap 클래스는 요소의 키에 대한 정렬된 순서를 보장하며 이를 위해서는 키 객체가 Comparable 인터페이스를 구현하고 있어야 합니다.

```
Map<String, Customer> map = new HashMap<>();  
  
Customer kim = new Customer("kim@namoosori.io", "Kim");  
Customer lee = new Customer("lee@namoosori.io", "Lee");  
Customer park = new Customer("park@namoosori.io", "Park");  
  
map.put(kim.getEmail(), kim);  
map.put(lee.getEmail(), lee);  
map.put(park.getEmail(), park);  
  
System.out.println(map);
```

```
{lee@namoosori.io=Email:lee@namoosori.io, NickName:Lee  
, park@namoosori.io=Email:park@namoosori.io, NickName:Park  
, kim@namoosori.io=Email:kim@namoosori.io, NickName:Kim}
```

4.10 Iterator 인터페이스

- ✓ Iterator 인터페이스는 컬렉션 객체가 가지고 있는 요소들을 순회할 수 있는 기능들을 명세한 인터페이스입니다.
- ✓ Collection 인터페이스를 구현한 클래스들은 iterator() 메소드를 통해 Iterator 구현 객체를 반환 합니다.
- ✓ Map 구현 클래스의 경우 Map.values() 메소드를 통해 값(요소)들을 Collection 타입으로 객체를 반환 받은 이후에 다시 Iterator 구현 객체를 이용하여 순회할 수 있습니다.

```
List<Customer> list = new ArrayList<>();
list.add(new Customer("kim@namoosori.io", "Kim"));
list.add(new Customer("lee@namoosori.io", "Lee"));
list.add(new Customer("park@namoosori.io", "Park"));

Iterator<Customer> iterator = list.iterator();
while(iterator.hasNext()){
    Customer customer = iterator.next();
    // something to do
}
```

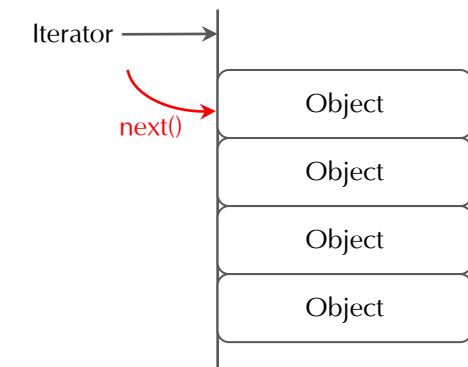
```
Map<String, Customer> map = new HashMap();
map.put("kim@namoosori.io", new Customer("kim@namoosori.io", "Kim"));
map.put("lee@namoosori.io", new Customer("lee@namoosori.io", "Lee"));
map.put("park@namoosori.io", new Customer("park@namoosori.io", "Park"));

Collection<Customer> coll = map.values();
Iterator<Customer> iterator = coll.iterator();
while(iterator.hasNext()){
    Customer customer = iterator.next();
    //something to do
}
```

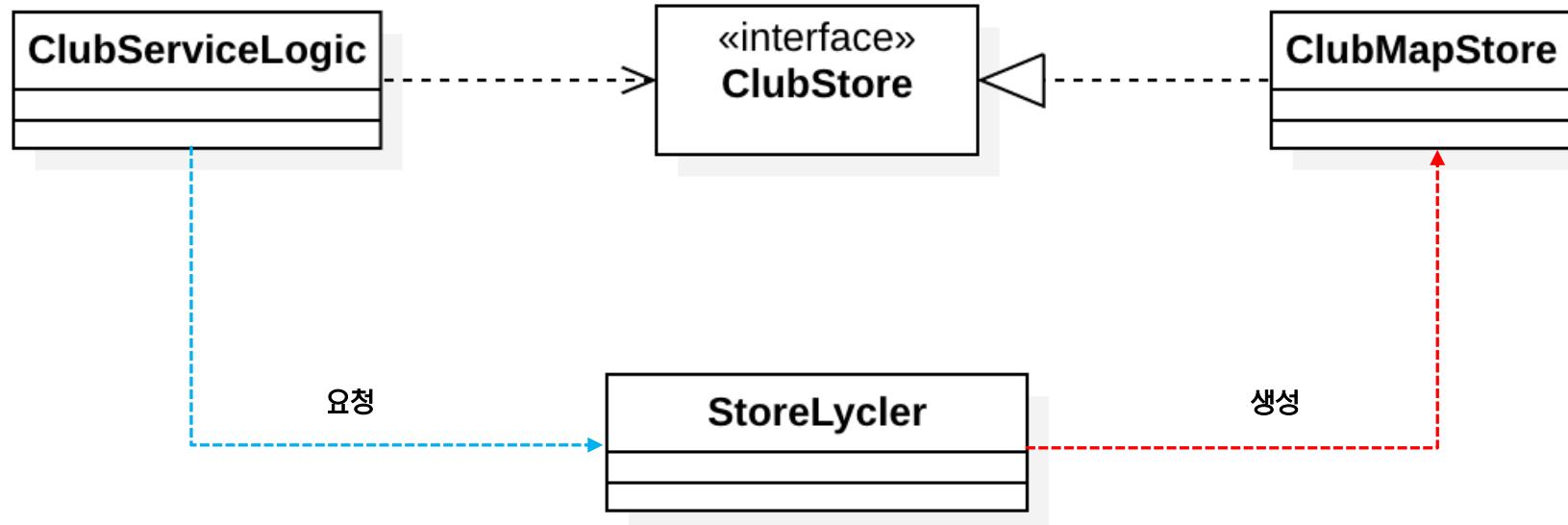
«interface»

Iterator

+ hasNext() : boolean
+ next() : Object
+ remove()



[실습] Store Layer





5. Java 8

- 5.1 Java 8 개요
- 5.2 인터페이스 다시보기
- 5.3 함수형 프로그래밍의 이해
- 5.4 람다(Lambda)의 이해와 활용
- 5.5 Stream API의 이해와 활용

5.1 Java 8 개요(1/2)

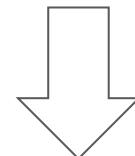
✓ Java 8(JDK 1.8)은 2014년 3월에 정식으로 릴리즈 되었습니다.

✓ Java 8을 통한 대표적인 변화는 다음과 같습니다.

- 인터페이스의 디폴트 메서드(default method)
- 람다식(lambda)를 통한 함수형 프로그래밍
- 스트림(Stream) API의 추가

```
Collections.sort(customers, new Comparator<Customer>() {  
    public int compare(Customer o1, Customer o2) {  
        return o1.getName().compareTo(o2.getName());  
    };  
});
```

Java 7

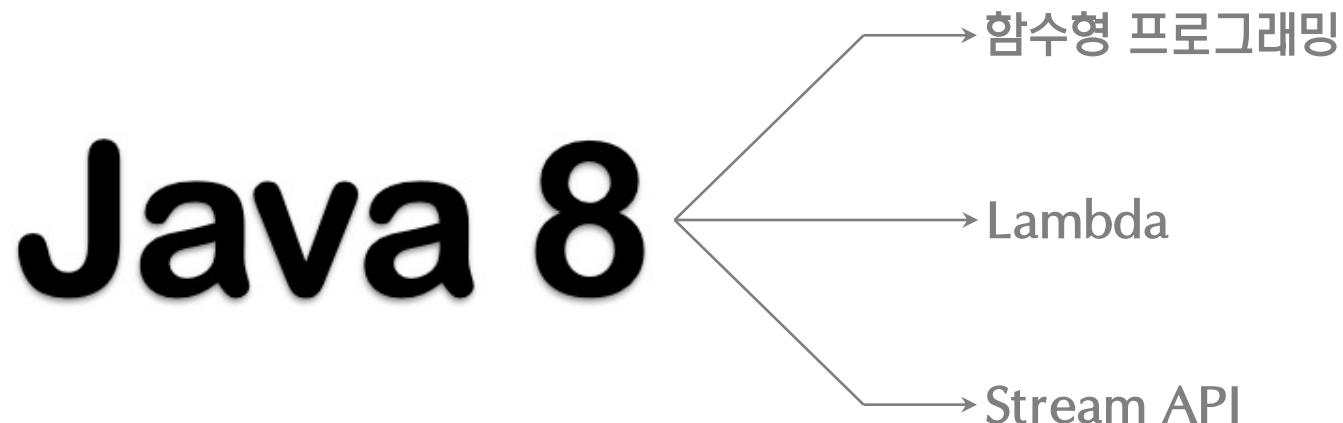


```
customers.sort(comapring(Customer::getName));
```

Java 8

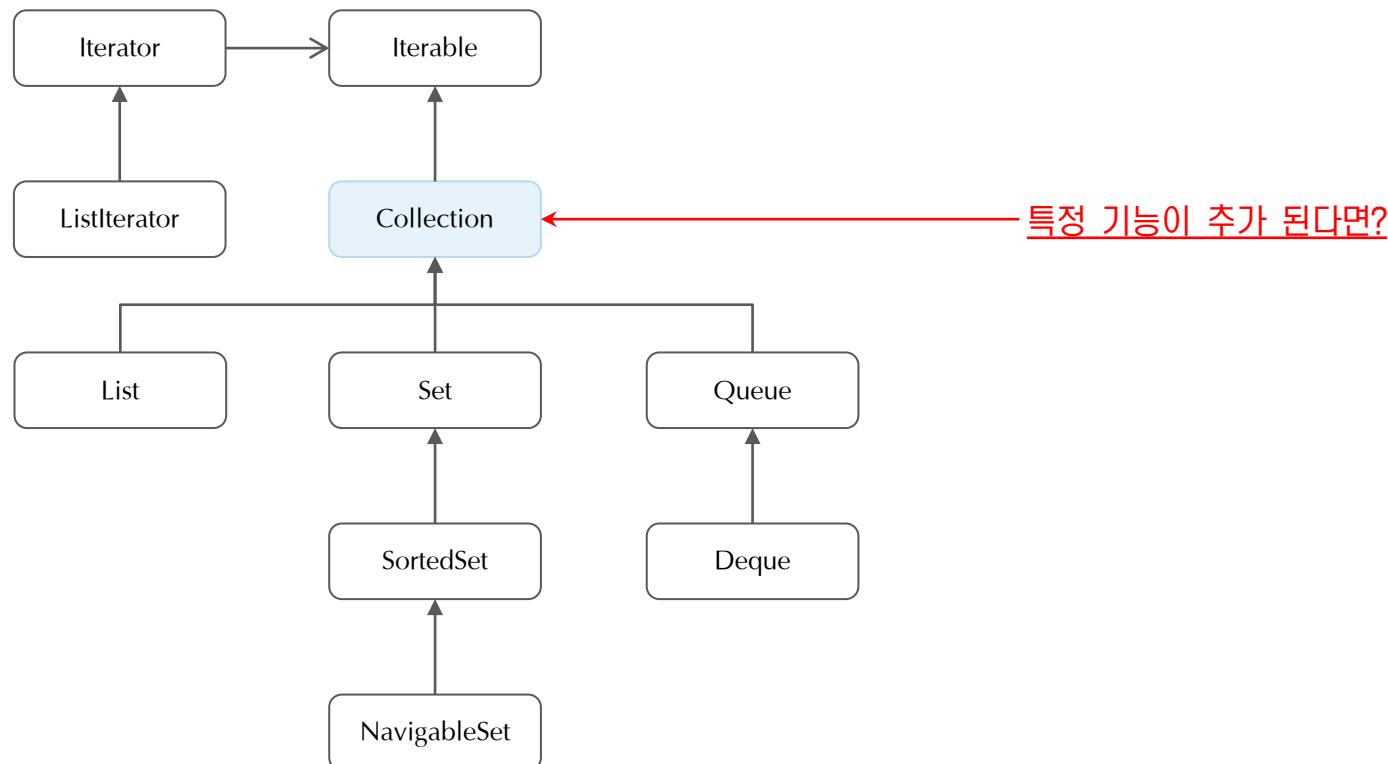
5.1 Java 8 개요(2/2)

- ✓ Java 8의 가장 큰 변화는 함수형 프로그래밍의 적용과 람다(Lambda), 스트림(stream)의 등장이라 할 수 있습니다.
- ✓ 이런 변화의 중심에는 인터페이스의 변경이 있으며 이를 토대로 함수형 프로그래밍과 스트림 등이 등장할 수 있었습니다.
- ✓ 인터페이스의 변경은 Java 7부터 시작 되었고 Java 8에 이르러 대대적으로 변경됐습니다.
- ✓ 함수형 프로그래밍의 적용과 람다, 스트림의 등장은 컬렉션 프레임워크를 개선하는데 바탕이 되었습니다.



5.2 인터페이스 다시보기(1/3) – 인터페이스의 한계와 변화

- ✓ 인터페이스의 주요 사용 목적은 다수의 구현체를 통일화된 명세로 정의하는 것입니다.
- ✓ 다수의 기능을 정의한 특정 인터페이스를 구현한 구현 클래스들은 해당 인터페이스를 통해 구현 방법에 상관없이 명세된 기능들을 사용할 수 있습니다.
- ✓ 이런 인터페이스가 가지고 있는 가장 큰 단점은 **한번 배포된 인터페이스는 이후 수정이 어렵다는 것입니다.**
- ✓ 인터페이스 변경의 어려움은 Java API의 오랜 고민이었으며 이는 **Collections 클래스를 통해 확인할 수 있습니다.**



5.2 인터페이스 다시보기(2/3) – 진화한 인터페이스(1/2)

- ✓ 인터페이스는 Java가 발전되어 오면서 크고 작은 변화가 있었으며 Java 8에 이르러 큰 변화를 갖습니다.
- ✓ Java 2, 5에서도 인터페이스를 정의하는 규칙에 일부의 변화가 있었지만 가장 큰 변화는 Java 8에 있습니다.
- ✓ Java 8 부터는 인터페이스에 정적 메소드와 **default** 키워드를 적용한 디폴트 메소드를 추가할 수 있습니다.
- ✓ 인터페이스에서 정의한 정적 메소드와 디폴트 메소드는 그 자체로 완전한 메소드이기 때문에 구현 클래스에서 재정의 하지 않습니다.

| Java 1.x | Java 1.2 | Java 1.5 |
|---|----------------------------|--------------------------------|
| 상수만 선언할 수 있다. (final static) | 중첩 클래스를 선언할 수 있다. (Nested) | 중첩 열거형을 선언할 수 있다.(Nested enum) |
| 추상 메소드만 정의할 수 있다. (abstract) | 중첩 인터페이스를 선언할 수 있다. | 중첩 어노테이션을 선언할 수 있다. |
| 인터페이스를 구현할 클래스는 해당 메소드를 재정의하거나 추상 클래스여야 한다. | - | - |
| 모든 필드와 메소드는 public 이다. | - | - |

| Java 1.8 | Java 1.9 |
|------------------------------|------------------------|
| static 메소드를 선언할 수 있다. | private 메소드를 선언할 수 있다. |
| 디폴트 메소드를 선언할 수 있다. (default) | - |
| - | - |
| - | - |

5.2 인터페이스 다시보기(3/3) – 진화한 인터페이스(2/2)

- ✓ **default** 메소드는 기존에 사용중인 인터페이스에 새로운 기능을 추가할 경우 적용할 수 있습니다.
- ✓ **default** 키워드는 **public** 접근 지정자를 포함하고 있으며 구현 클래스에서 재정의 가능합니다.
- ✓ 인터페이스에 **static**, **default**, **private**(Java 9) 메소드를 정의할 수 있게 되면서 클래스나 추상 클래스를 상속하는 것과 차이가 없는 것으로 생각할 수 있지만 인터페이스가 필드를 가질 수 없는 것과 인스턴스화가 될 수 없는 차이를 갖습니다.

```
public interface Message {  
    String LANGUAGE = "KOREAN"; // public static final  
  
    String showMessage(); // public abstract  
  
    default String getLanguage(){ // public  
        return Message.LANGUAGE;  
    }  
}
```

```
public class MyMessage implements Message{  
    @Override  
    public String showMessage() {  
        return null;  
    }  
    @Override  
    public String getLanguage() {  
        return Message.super.getLanguage();  
    }  
}
```

5.3 함수형 프로그래밍의 이해(1/2)

- ✓ 함수형 프로그래밍을 이해하기 위해서는 우선 명령형(imperative) 프로그래밍과 선언형(declarative) 프로그래밍에 대한 이해가 필요합니다.
- ✓ 명령형 프로그래밍은 특정 기능을 수행하기 위해 어떻게(how)에 집중하는 방식을 의미합니다.
- ✓ 선언형 프로그래밍은 특정 기능을 수행하기 위해 무엇(what)에 집중하는 방식을 의미합니다.
- ✓ 함수형 프로그래밍은 선언형 프로그래밍을 따르는 대표적인 프로그래밍 패러다임입니다.

```
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9,10);

int result = 0;

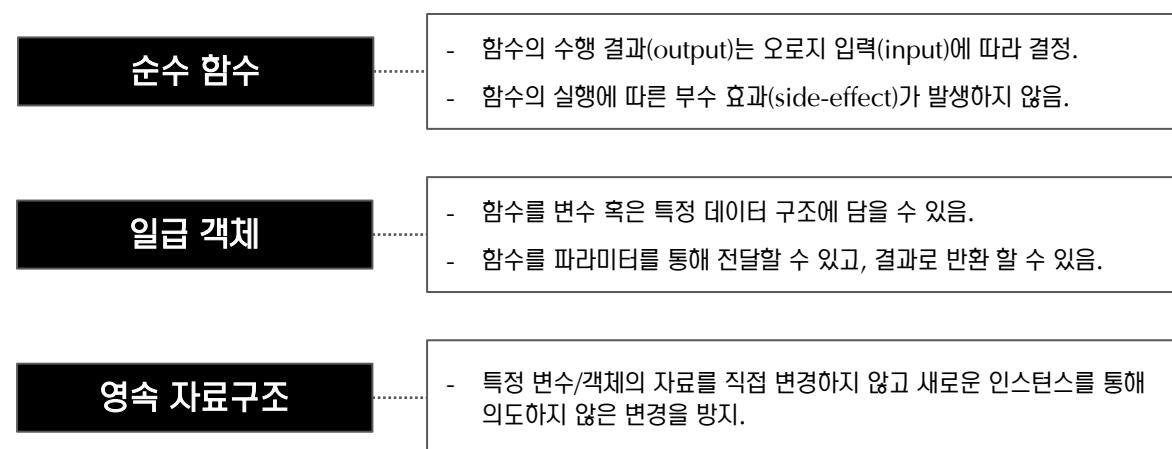
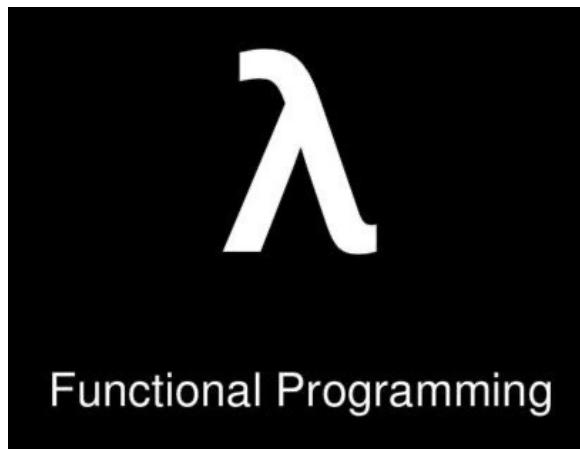
for(int n : list){
    if(n % 2 == 0){
        result += n;
    }
}
```

```
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9,10);

int result = 0;
result = list.stream()
    .filter(n -> n % 2 == 0)
    .mapToInt(Integer::intValue)
    .sum();
```

5.3 함수형 프로그래밍의 이해(2/2)

- ✓ 함수형 프로그래밍은 함수들의 집합으로 프로그램을 구성하는 것을 의미합니다.
- ✓ 자바에 함수형 프로그래밍 도입은 프로그램 구현 방식에 큰 변화를 가져 왔습니다.
- ✓ 함수형 프로그래밍의 함수는 순수 함수(Pure function), 일급 객체(First-class), 불변의 자료구조 혹은 영속 자료구조 등과 같은 특성을 갖습니다.



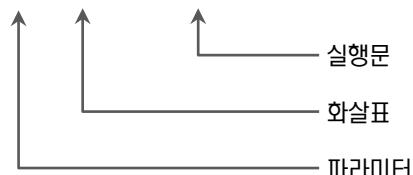
5.4 람다(Lambda)의 이해와 활용 (1/5) – 개요(1/2)

- ✓ Java 8에서 가장 중요한 변화라 할 수 있는 램다 표현식의 등장은 불필요한 코드를 줄이고, 코드의 이해를 돋습니다.
- ✓ 램다 표현식은 메소드로 전달할 수 있는 익명 함수를 단순화한 코드의 블록입니다.
- ✓ 램다 표현식은 특정 클래스에 종속되지 않으며 함수라는 이름으로 명명 합니다.
- ✓ 램다 표현식은 함수 자체를 전달 인자로 보내거나 변수에 저장하는 것이 가능합니다.

```
Runnable runnable = new Runnable() {  
    @Override  
    public void run() {  
        //TODO Auto-generated method stub  
    }  
};
```

```
Runnable runnable = () -> {      }; // Lambda
```

```
() -> { ... };
```



5.4 람다(Lambda)의 이해와 활용 (2/5) – 개요(2/2)

- ✓ 램다 표현식은 익명 구현 클래스를 생성하고 객체화 합니다.
- ✓ 익명 구현 클래스로 생성된 램다 표현식은 인터페이스로 대입 가능하며 이 인터페이스를 함수형 인터페이스라고 합니다.
- ✓ 하나의 추상 메소드를 갖는 인터페이스는 모두 함수형 인터페이스가 될 수 있습니다.
- ✓ 다수의 디폴트 메소드를 갖는 인터페이스라도 추상 메소드가 하나라면 함수형 인터페이스입니다.
- ✓ 함수형 인터페이스를 정의 할 때 `@FunctionalInterface` 어노테이션을 이용해 컴파일 검사를 진행할 수 있습니다.
- ✓ 함수형 인터페이스의 추상메소드 시그니처를 함수 디스크립터(function descriptor)라고 합니다.

```
@FunctionalInterface  
public interface FunctionalInterfaceSample {  
    void testMethod();  
    void errMethod(); // error  
}
```



Invalid '@FunctionalInterface' annotation; FunctionalInterfaceSample is not a functional interface

5.4 람다(Lambda)의 이해와 활용 (3/5) – 함수형 인터페이스

- ✓ `java.util.function` 패키지를 통해 다양한 함수형 인터페이스를 제공하고 있습니다.
- ✓ 기본형 특화는 Java의 Auto-Boxing 동작을 제한한 함수형 인터페이스입니다.

| 함수형 인터페이스 | 함수 디스크립터 | 기본형 특화 |
|--------------------------------------|--------------------------------|--|
| <code>Predicate<T></code> | <code>T -> boolean</code> | <code>IntPredicate, LongPredicate, DoublePredicate</code> |
| <code>Consumer<T></code> | <code>T -> void</code> | <code>IntConsumer, LongConsumer, DoubleConsumer</code> |
| <code>Function<T, R></code> | <code>T -> R</code> | <code>IntFunction<T>, IntToDoubleFunction, ...</code> |
| <code>Supplier<T></code> | <code>() -> T</code> | <code>BooleanSupplier, IntSupplier, ...</code> |
| <code>BinaryOperator<T></code> | <code>(T, T) -> T</code> | <code>IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator</code> |
| <code>UnaryOperator<T></code> | <code>T -> T</code> | <code>IntUnaryOperator, LongUnaryOperator...</code> |
| <code>BiConsumer<T, U></code> | <code>(T, U) -> void</code> | <code>ObjIntConsumer<T>, ObjectLongConsumer<T>, ...</code> |
| <code>BinaryOperator<T></code> | <code>(T, T) -> T</code> | <code>IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator</code> |
| ... | ... | ... |

5.4 람다(Lambda)의 이해와 활용 (4/5) – 람다의 사용

- ✓ 람다 표현식의 사용은 메소드 내부에서 주로 이루어지기 때문에 지역변수 사용시에 제약이 존재합니다.
- ✓ 메소드 내부에서의 람다 표현식은 곧 익명 객체를 메소드 내부에서 생성하는 것과 같습니다.
- ✓ 람다 표현식에서 접근하는 해당 메소드의 지역 변수와 매개 변수는 `final` 특성을 적용합니다.
- ✓ 람다 표현식에서 해당 메소드의 지역 변수, 매개 변수를 참조하는 것을 람다 캡쳐링(lambda capturing)이라 합니다.

```
int number = 10;  
  
Runnable runnable = () -> System.out.println(number); // (0)
```

```
int number = 10;  
  
Runnable runnable = () -> System.out.println(number); // (X)  
number = 20;
```

Local variable `number` defined in an enclosing scope must be final or effectively final

5.4 람다(Lambda)의 이해와 활용 (5/5) – 메소드 레퍼런스

- ✓ 기존 클래스의 메소드를 람다 표현식을 통해 호출하는 것을 메소드 레퍼런스라고 합니다.
- ✓ 메소드 레퍼런스는 특정 메소드만을 호출하는 람다 표현식의 축약형입니다.
- ✓ 람다의 메소드 레퍼런스는 :: 구분자를 통해 활용합니다.

```
Consumer<String> con = s -> System.out.println(s);
con.accept(str);

con = System.out::println; // 메소드 레퍼런스
con.accept(str);
```

- ✓ 정적 메소드 레퍼런스는 [클래스 :: 메소드] 형태로 사용합니다.

```
IntBinaryOperator operator;

operator = MethodReferenceSample::staticAdd;
System.out.println(operator.applyAsInt(10, 20));
```

- ✓ 인스턴스 메소드 레퍼런스는 [생성객체 :: 메소드] 형태로 사용합니다.

```
IntBinaryOperator operator;
MethodReferenceSample mf = new MethodReferenceSample();

operator = mf::instanceAdd;
System.out.println(operator.applyAsInt(100, 200));
```

5.5 Stream API의 이해와 활용 (1/11) – 개요

- ✓ Java 8에 추가된 Stream API를 활용하면 다양한 데이터 소스를 표준화된 방법으로 다룰 수 있습니다.
- ✓ 따라서, Collection F/W를 통해 관리하는 데이터를 처리하기 위해 주로 사용합니다.
- ✓ Stream API의 활용을 통해 수집된 다양한 데이터를 활용하는데 있어서 간결하고 가독성 있는 처리가 가능합니다.
- ✓ Stream API의 다양한 기능들은 대부분 람다를 필요로 하기 때문에 람다를 이해하고 사용할 수 있어야 합니다.

```
List<String> list = Arrays.asList("Lee", "Park", "Kim");

// 기존
Iterator<String> it = list.iterator();
while(it.hasNext()) {
    System.out.println(it.next());
}
```



```
List<String> list = Arrays.asList("Lee", "Park", "Kim");

// Stream 활용
list.stream().forEach(name -> System.out.println(name));
```

5.5 Stream API의 이해와 활용 (2/11) – Stream interface

- ✓ Stream API의 최상위 인터페이스는 BaseStream 인터페이스이지만 직접 사용하는 경우는 없습니다.
- ✓ 주로 사용하는 인터페이스는 Stream 인터페이스이며 BaseStream을 상속하는 인터페이스입니다.
- ✓ Stream 인터페이스는 여러 메소드들을 정의하고 있으며 많은 메소드들의 파라미터에 람다와 메소드 참조가 필요합니다.
- ✓ Stream을 구현한 객체의 주요 특징은 불변성이며 Stream을 통해 얻은 결과는 새롭게 생성된 데이터입니다.

| 메소드 | 기능 |
|---------------------------------|--|
| long count() | 해당 스트림에 포함된 항목의 수를 반환. |
| Stream concat(Stream, Stream) | 파라미터로 전달되는 두 개의 스트림을 하나의 스트림으로 반환. |
| R collect(Collector) | 스트림의 항목들을 컬렉션 타입의 객체로 반환. |
| Stream filter(Predicate) | 스트림의 항목들을 파라미터의 조건에 따라 필터링하고 결과 항목들을 스트림 형태로 반환. |
| void forEach(Consumer) | 스트림 항목들에 대한 순회.(최종 연산) |
| Optional reduce(BinaryOperator) | 람다 표현식을 기반으로 데이터를 소모하고 그 결과를 반환.(최종 연산) |
| Object[] toArray() | 스트림 항목들을 배열 객체로 반환. |
| Stream sorted() | 스트림 항목들에 대해 정렬하고 이를 스트림으로 반환. |
| ... | ... |

5.5 Stream API의 이해와 활용 (3/11) – Stream 객체 생성(1/2)

- ✓ Stream 객체를 생성하는 방법은 Collection 객체를 통한 방법과 스트림 빌더를 통한 방식 두 가지가 있습니다.
- ✓ Collection 인터페이스는 stream() 메소드를 default 메소드로 정의하고 있습니다.
- ✓ 이 메소드는 해당 컬렉션이 가지고 있는 항목들에 대해 스트림 처리가 가능한 Stream 객체를 반환합니다.
- ✓ 한번 생성한 스트림은 사용 후 다시 사용할 수 없으며 전체 데이터에 대한 처리가 이루어지면 종료됩니다.

```
List<String> list =  
    Arrays.asList("Lee", "Kim", "Park", "Hong", "Choi", "Song");  
  
Stream<String> stream = list.stream(); // Stream 생성  
//stream.forEach( name -> System.out.println(name));
```

```
List<String> list =  
    Arrays.asList("Lee", "Kim", "Park", "Hong", "Choi", "Song");  
  
Stream<String> stream = list.stream(); // Stream 생성  
System.out.println(stream.count()); // Stream 사용  
stream.forEach(System.out::println); // Exception 발생
```



stream has already been operated upon or closed

5.5 Stream API의 이해와 활용 [4/11] – Stream 객체 생성[2/2]

- ✓ Collection 객체를 통한 생성은 처리할 이미 데이터가 존재하고 이를 처리하기 위한 일반적인 생성 방식입니다.
- ✓ Stream.Builder를 이용한 Stream 객체의 생성 방식은 스트림 자체적으로 데이터를 생성하고 처리할 수 있습니다.
- ✓ Stream.Builder 인터페이스는 Consumer 인터페이스를 상속하고 있으며 데이터를 추가하는 accept(), add() 메소드와 데이터의 추가 작업을 완료하고 Stream을 반환하는 build() 메소드를 정의하고 있습니다.

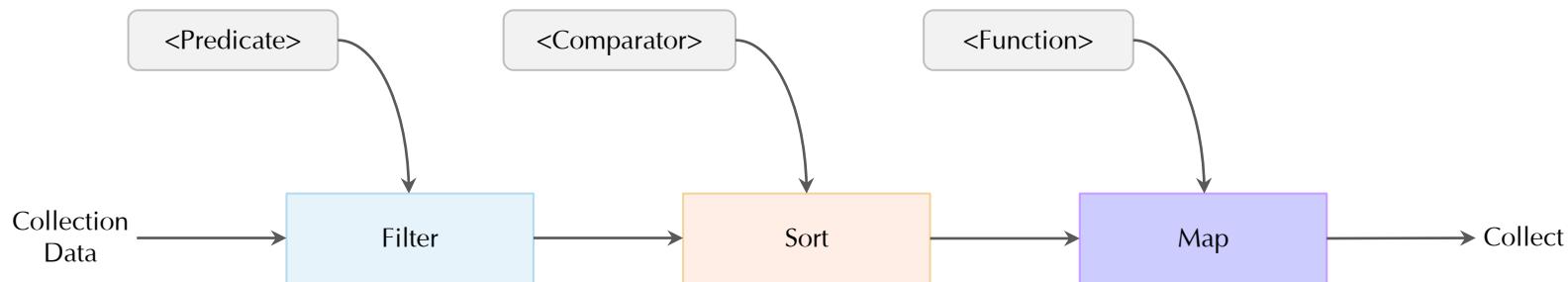
| 메소드 | 기능 |
|--------------------------|------------------------------|
| void accept(T) | 스트림 빌더에 데이터 추가. |
| Stream.Builder<T> add(T) | 스트림 빌더에 데이터를 추가하고, 스트림 반환. |
| Stream<T> build() | 스트림 빌더에 데이터 추가를 종료하고 스트림 반환. |

```
Stream.Builder<String> builder = Stream.builder();
builder.accept("Kim");
builder.accept("Lee");
builder.accept("Song");
builder.accept("Park");
builder.accept("Lee");

Stream<String> stream = builder.build();
stream.forEach(System.out::println);
```

5.5 Stream API의 이해와 활용 (5/11) – Stream 연산(1/3)

- ✓ 스트림을 이용한 연산은 각 연산의 연결을 통해 파이프라인을 구성할 수 있습니다.
- ✓ 파이프라인을 구성할 수 있다는 것은 스트림 대상 데이터에 대한 다양한 연산을 조합할 수 있다는 것을 의미합니다.
- ✓ 스트림을 이용한 연산 처리는 스트림 객체의 생성부터 중간 연산, 그리고 최종 연산 단계로 구분할 수 있습니다.
- ✓ 스트림 객체가 제공하는 다양한 연산을 이해하고 연산에 필요한 람다표현식을 이해하고 적용하는 것이 중요합니다.



5.5 Stream API의 이해와 활용 (6/11) – Stream 연산(2/3)

- ✓ 스트림의 연산은 중간 연산(intermediate operation)과 최종 연산(terminal operation)이 있습니다.
- ✓ 중간 연산은 filter, map과 같은 연산으로 Stream을 반환합니다.
- ✓ 중간 연산은 연속해서 호출하는 메소드 체이닝(Method Chaining)으로 구현 가능합니다.
- ✓ 최종연산이 실행되어야 중간연산이 처리되므로 중간연산들로만 구성된 메소드 체인은 실행되지 않습니다.

| 연산 | 반환 형식 | 연산 인수 |
|----------|-----------|----------------|
| filter | Stream<T> | Predicate<T> |
| map | Stream<T> | Function<T, R> |
| limit | Stream<T> | |
| sorted | Stream<T> | Comparator<T> |
| distinct | Stream<T> | |
| peek | Stream<T> | Consumer<T> |
| skip | Stream<T> | |

5.5 Stream API의 이해와 활용 (7/11) – Stream 연산(3/3)

- ✓ 최종 연산은 `forEach`, `collect`와 같은 연산으로 `void`를 반환하거나 컬렉션 타입을 반환합니다.
- ✓ 중간 연산을 통해 가공된 스트림은 마지막으로 최종연산을 통해 각 요소를 소모하여 결과를 출력합니다.
- ✓ 즉, 지연(lazy)되었던 모든 중간 연산들이 최종연산시 모두 수행되는 것입니다.
- ✓ 최종 연산 후에는 한번 생성해서 소모한 스트림은 닫히게 되고 재사용이 불가능합니다.

| 연산 | 반환 형식 |
|----------------------|---|
| <code>forEach</code> | 스트림의 각 요소를 소비하며 람다 적용. <code>void</code> 반환 |
| <code>count</code> | 스트림 요소의 수를 반환. <code>Long</code> 반환 |
| <code>collect</code> | <code>List</code> , <code>Map</code> 형태의 컬렉션 반환 |
| <code>sum</code> | 스트림의 모든 요소에 대한 합을 반환 |
| <code>reduce</code> | 스트림의 요소를 하나씩 줄여가며 연산 수행 후 결과 반환. <code>Optional</code> 반환 |

5.5 Stream API의 이해와 활용 (8/11) – 필터링

- ✓ 필터링은 전체 데이터에서 불필요한 데이터를 없애고 원하는 데이터를 정확히 추출하기 위한 과정입니다.
- ✓ Stream API의 `filter()`, `distinct()`와 같은 메소드를 이용해 데이터 추출이나 중복제거를 구현합니다.
- ✓ 필터링 연산은 스트림의 중간 연산으로 필터링 결과는 Stream 객체로 반환하며 연산 완료를 위한 최종 연산이 필요합니다.
- ✓ 데이터의 중복을 제거하는 `distinct()`는 병렬 스트림의 경우 성능에 대한 고려가 필요하며 중복 객체의 비교는 `equals()` 메소드를 이용하기 때문에 이에 대한 고려 또한 필요합니다.

```
List<Customer> customers = new ArrayList<>();  
customers.add(new Customer("Kim", 33));  
customers.add(new Customer("Park", 21));  
customers.add(new Customer("Song", 45));  
customers.add(new Customer("Lee", 67));  
customers.add(new Customer("Choi", 19));  
customers.add(new Customer("Kim", 33)); // 중복 데이터
```

```
Stream<Customer> stream = customers.stream();  
stream.filter( customer -> customer.getAge() > 30)  
.forEach(System.out::println);
```

```
[Name : Kim, Age : 33]  
[Name : Song, Age : 45]  
[Name : Lee, Age : 67]  
[Name : Kim, Age : 33]
```

```
Stream<Customer> stream = customers.stream();  
stream.filter( customer -> customer.getAge() > 30)  
.distinct()  
.forEach(System.out::println);
```

```
[Name : Kim, Age : 33]  
[Name : Song, Age : 45]  
[Name : Lee, Age : 67]
```

5.5 Stream API의 이해와 활용 [9/11] – 정렬

- ✓ Stream API의 sorted() 메소드는 특정 조건에 따라 데이터를 정렬하고 이를 다시 Stream으로 반환합니다.
- ✓ sorted()를 이용한 정렬을 위해서는 반드시 대상 객체들이 Comparable 인터페이스를 구현한 클래스, 즉 비교 가능한 객체여야 합니다.
- ✓ Comparable 객체가 아닐 경우나 역순 정렬, 혹은 다른 조건의 정렬에는 Comparator 인터페이스가 제공하는 여러 default, static 메소드를 이용해 정렬을 구현합니다.

```
public class Customer implements Comparable<Customer>{  
    ...  
    @Override  
    public int compareTo(Customer customer) {  
        if(this.age > customer.getAge()){  
            return 1;  
        }else if( this.age == customer.getAge()){  
            return 0;  
        }else {  
            return -1;  
        }  
    }  
}
```

```
List<Customer> customers = new ArrayList<>();  
customers.add(new Customer("Kim", 33));  
customers.add(new Customer("Park", 21));  
customers.add(new Customer("Song", 45));  
customers.add(new Customer("Lee", 67));  
customers.add(new Customer("Choi", 19));
```

```
customers.stream()  
    .sorted()  
    .forEach(System.out::println);
```

```
customers.stream()  
    .sorted(Comparator.comparing(Customer::getName))  
    .forEach(System.out::println);
```

5.5 Stream API의 이해와 활용 (10/11) – 맵핑

- ✓ 스트림의 맵핑(map) 연산은 스트림이 관리하는 데이터를 다른 형태의 데이터로 변환할 수 있도록 합니다.
- ✓ 맵핑 연산의 메소드는 `map()`, `mapToInt()`, `mapToDouble()`, `mapToLong()`이 있습니다.
- ✓ 주로 사용하는 메소드는 `map()` 메소드이며 파라미터는 `Function` 함수형 인터페이스입니다.
- ✓ `double`, `int`, `long` 기본형 데이터 타입의 데이터를 처리하기 위한 메소드들은 맵핑된 값의 결과가 기본형 데이터 타입일 경우 적용하여 사용합니다.

```
List<Customer> customers = new ArrayList<>();  
customers.add(new Customer("Kim", 33));  
customers.add(new Customer("Park", 21));  
customers.add(new Customer("Song", 45));  
customers.add(new Customer("Lee", 67));  
customers.add(new Customer("Choi", 19));
```

```
List<String> names = customers.stream()  
    .map(Customer::getName)  
    .collect(Collectors.toList());  
  
names.stream().forEach(System.out::println);
```

```
customers.stream()  
    .map(Customer::getName)  
    .forEach(System.out::println);
```

```
Kim  
Park  
Song  
Lee  
Choi
```

5.5 Stream API의 이해와 활용 (11/11) – 최종 연산

- ✓ 스트림이 관리하는 전체 데이터에 대한 순회 작업은 최종 연산인 `forEach()` 메소드를 이용합니다.
- ✓ `collect()` 메소드는 스트림 처리 이후 처리된 데이터에 대해 Collection 객체로 반환하는 메소드입니다.
- ✓ 스트림의 최종 연산은 `forEach()`와 같은 스트림 처리 결과를 바로 확인할 수 있는 연산이 있고, 데이터를 모두 소모한 이후에 그 결과를 알 수 있는 `count()`와 같은 연산이 있습니다.
- ✓ 이외에도 특정 데이터를 검색할 수 있는 `allMatch()`, `anyMatch()`등과 같은 다양한 메소드들을 제공하고 있습니다.

| 연산 | 반환 형식 |
|------------------------|---|
| <code>allMatch</code> | 파라미터로 전달되는 람다식 기준으로 스트림 데이터가 모두 일치하는지를 확인. |
| <code>anyMatch</code> | 파라미터로 전달되는 람다식 기준으로 스트림 데이터가 하나라도 일치하는지를 확인. |
| <code>noneMatch</code> | 파라미터로 전달되는 람다식 기준으로 스트림 데이터가 모두 일치하지 않는지를 확인. |
| <code>findFirst</code> | 스트림 데이터 중에서 가장 첫번째 데이터를 반환. |
| <code>reduce</code> | 스트림 데이터 중에서 임의의 데이터를 반환. |

✓ 토론

감사합니다...

- ❖ 넥스트리(주), 나무소리
- ❖ www.nextree.io