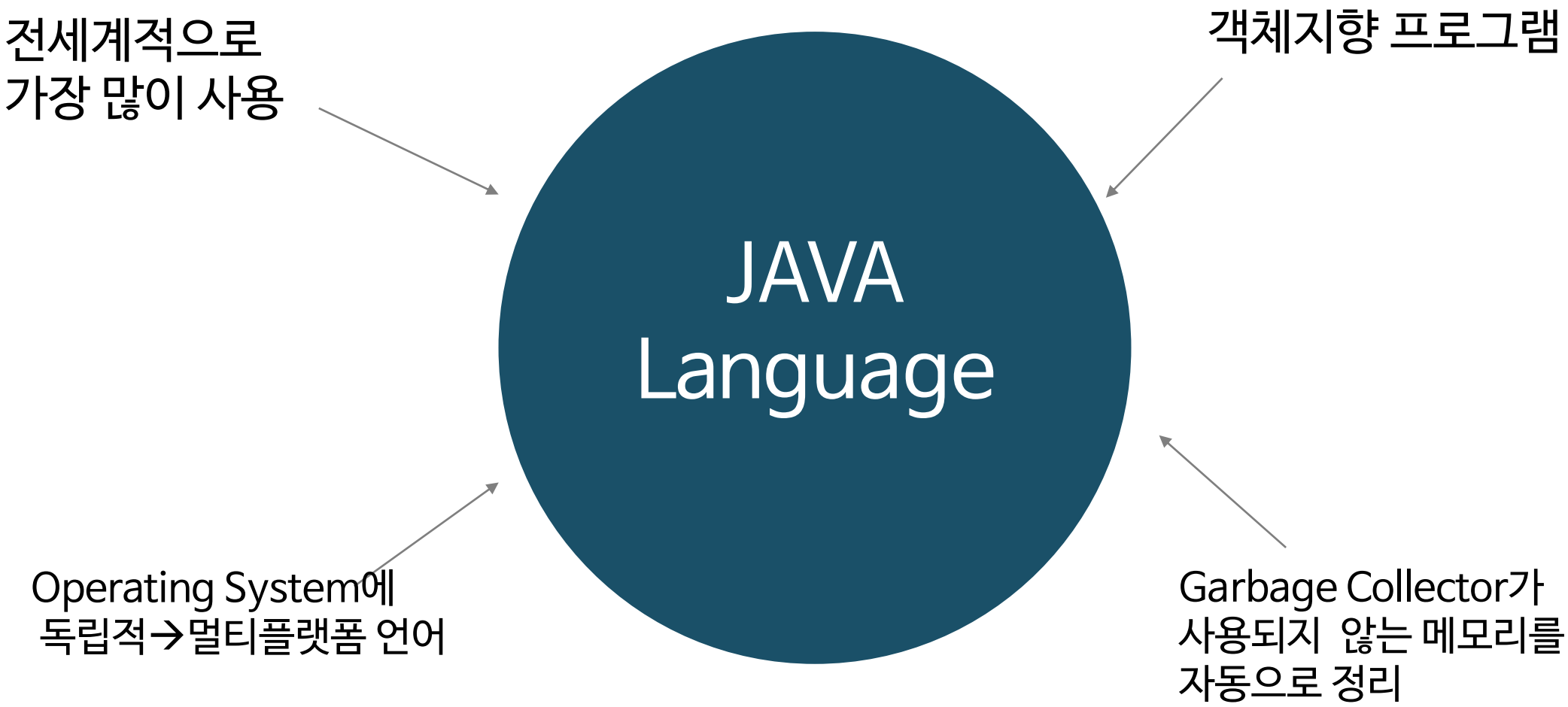
Java programming

1

자바프로그래밍 기본개요

1. 자바의 특징
2. 자바 개발환경 설치
3. 자바프로그램의 기본 구조

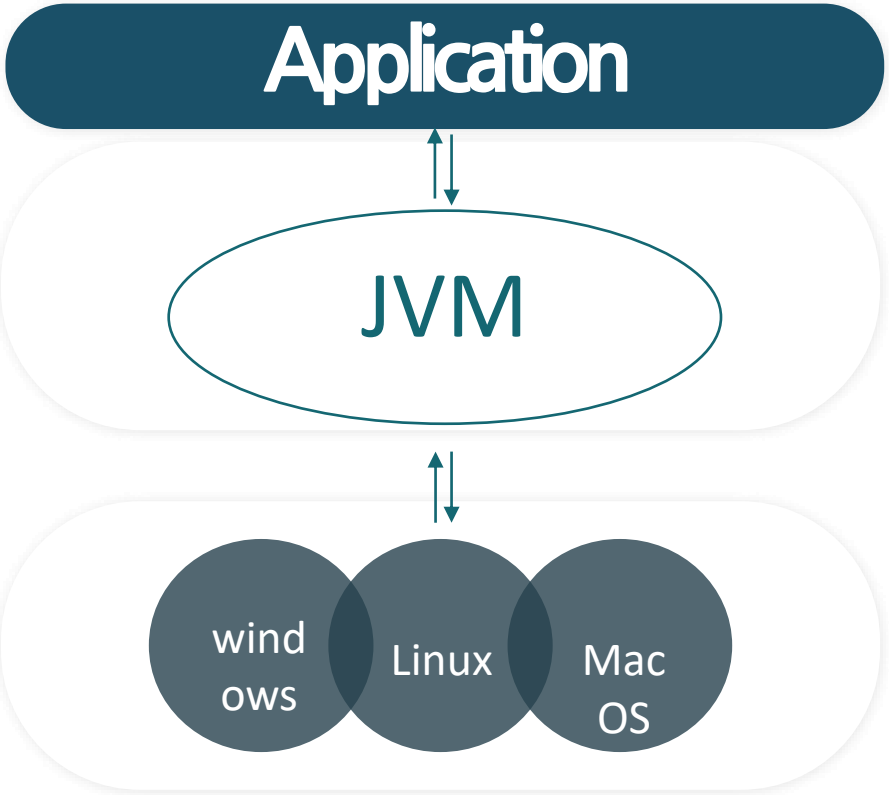
1.1. JAVA의 특징


전세계적으로
가장 많이 사용
객체지향 프로그램
JAVA
Language
Operating System에
독립적→멀티플랫폼 언어
Garbage Collector가
사용되지 않는 메모리를
자동으로 정리

1.1. JAVA의 특징

멀티 플랫폼 언어

자바는 멀티 플랫폼 언어다.
자바로 작성된 코드는 다양한 운영체제(UNIX, Window, MacOS, Android, etc..)에서 변경없이 동작한다..

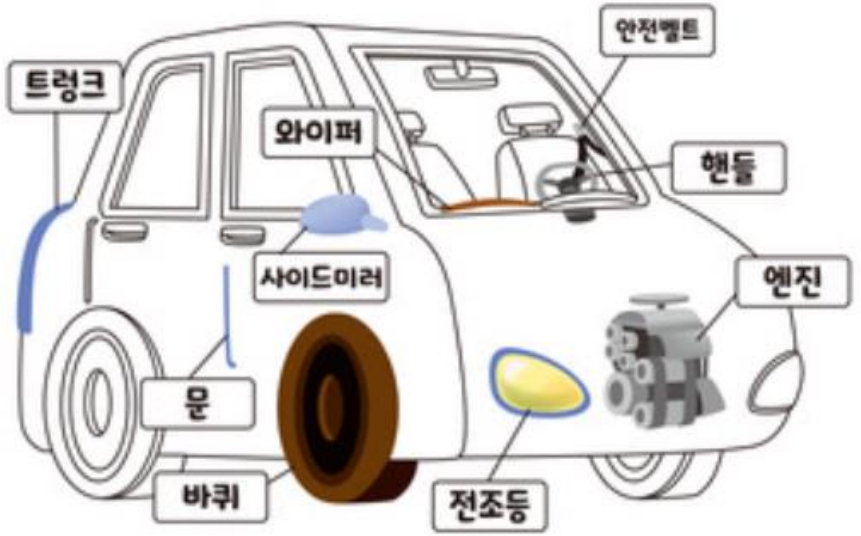


JAVA프로그램의 실행구조

1.1. JAVA의 특징

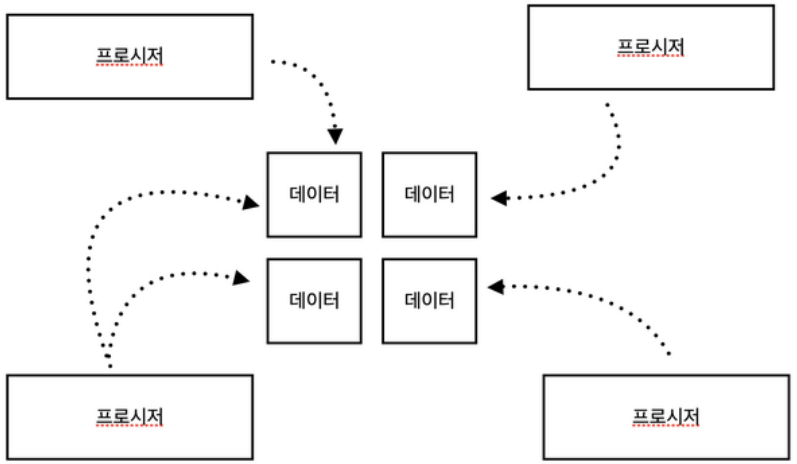
객체지향 언어

- 자바는 객체 지향 프로그래밍(Object-Oriented Programming) 언어다.
- 객체를 조립하여 전체 프로그램을 만드는 언어다.
- 자동차를 전체 프로그램, 각 부품을 객체라고 이해할 수 있겠다
- 객체 지향 프로그램은 여러 장점이 있다.
  - ✓ 수정이 필요한 경우, 해당 부분만 수정해 주면 된다.
  - ✓ 특정 기능들을 개선, 추가 또는 확장할 수 있다.



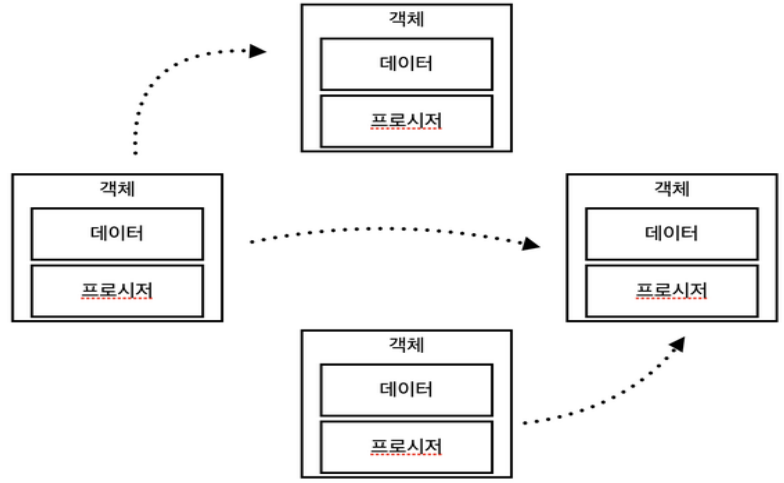
절차지향 & 객체지향

절차지향 프로그래밍



- 프로시져의 데이터 공유 – 데이터 중심 구현 방법
- 데이터 타입이나 의미를 변경해야 할 때, 이 데이터를 사용하는 모든 프로시저도 함께 수정해 주어야 한다

객체지향 프로그래밍

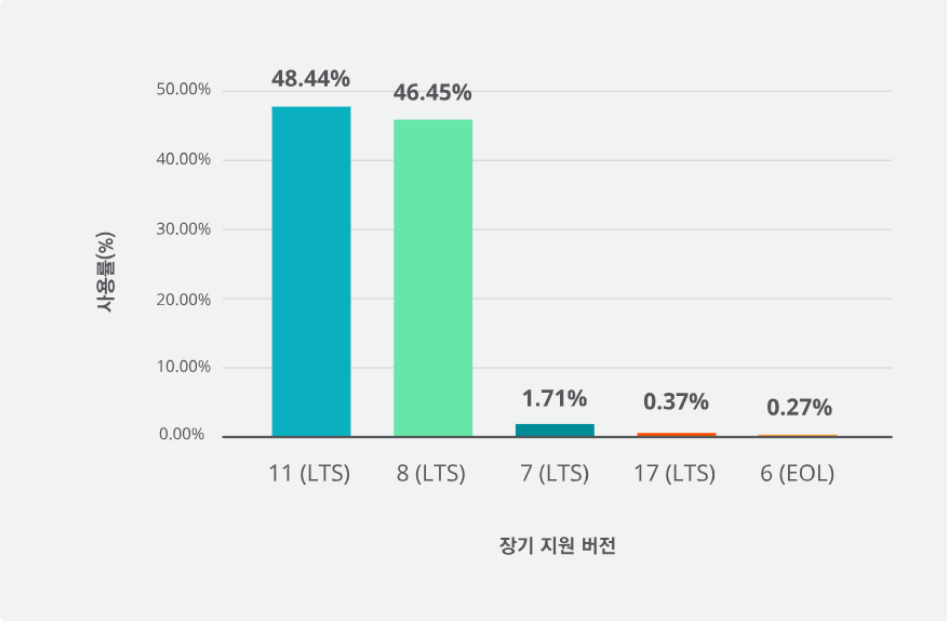


- 객체는 자신만의 데이터와 프로시저를 갖는다
- 객체의 데이터를 변경하더라도 다른 객체에 영향을 주지 않는다
- 프로그램의 확장과 유지보수 용이

1.2. 자바 개발 환경 설치

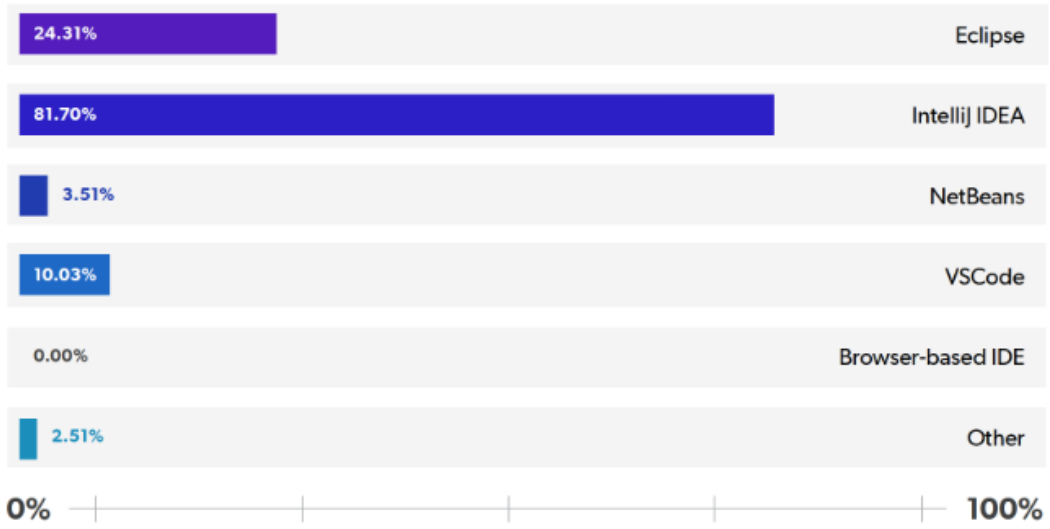
국내 JDK , IDE 사용현황

JDK



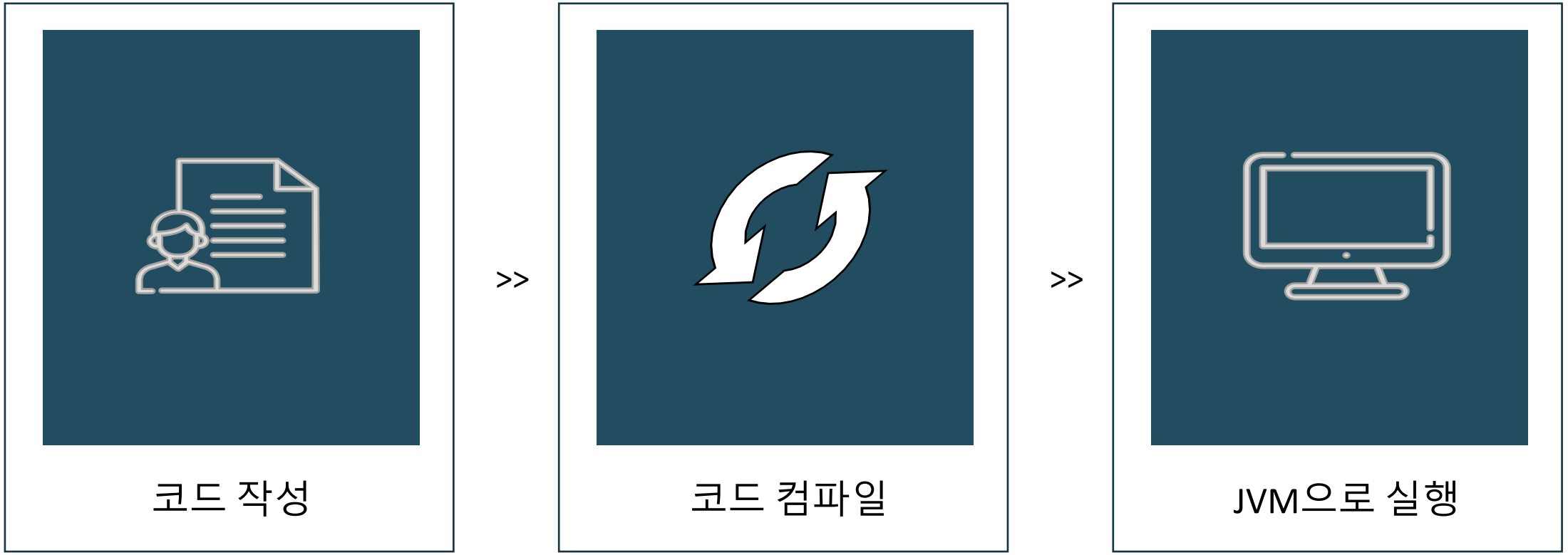
뉴렐릭 2022년 3월보고서

IDE



2020 Java Technology Report

1.3. 자바 프로그램 개발 순서



- 개발자는 컴퓨터에게 시킬 일들을 자바 언어로 작성한다. 이렇게 작성된 코드를 소스코드라 한다. 소스코드는 컴파일러(compiler)라는 번역기를 통해 기계어로 바뀐다. 비로소 컴퓨터는 기계어를 수행하게 된다. 특별히 자바 코드가 기계어로 번역되는 과정. 이를 컴파일링(compiling) 이라 한다.

1.4. 자바프로그램의 기본 구성요소

하나의 자바 프로그램에는 main() 메소드를 가지는 클래스가 반드시 하나는 존재해야 함

. main() 메소드는 반드시 public static void로 선언해야 함

Main Method

- 명령문--〉 자바 프로그램의 동작을 명시하고, 이러한 동작을 컴퓨터에 알려주는 데 사용되는 문장
- 자바의 모든 명령문은 반드시 세미콜론(;)으로 끝나야 함

명령문(statement)

- 주석 → 코드에 대한 이해를 돕는 설명을 적거나 디버깅을 위해 작성하는 일종의 메모
- 자바 컴파일러는 주석은 무시하고 컴파일하므로, 실제 실행 결과에는 아무런 영향을 주지 않음

주석(comment)

1.5. 자바프로그램의 기본 단위 'class'

- Java program의 기본 단위 → 'class'

class

- 객체의 속성과 기능을 정의하는 설계도

필드(Field)

- 객체의 속성
- 클래스의 객체들이 공유하는 데이터를 저장하는 역할

메소드(Method)

- 객체의 기능을 표현
- 특정 작업을 수행하기 위한 명령문의 집합

생성자(Constructor)

- 객체의 생성과 동시에 인스턴스 변수를 원하는 값으로 초기화할 수 있는 특수한 메소드

1.6. 자바프로그램의 기본 구조

```
class 클래스이름 {
   필드의 선언
   필드의 선언
   ...
   메소드의 선언
   메소드의 선언
   ...
}
```

예제

```
class Test {
   int field1;
   String field2;

   public void method1() {
      System.out.println("헬로 월드~!!");
   }
}
```

- 자바 프로그램은 한 개 이상의 클래스(class)로 구성
- 클래스는 한 개 이상의 필드(field)나 메소드(method)로 구성

2

변수와 데이터타입

1. 변수 개념
2. 변수 선언과 초기화
3. 자료형(Data type)
4. Wrapper 클래스
5. 콘솔출력과 키보드입력

2.1. 변수 개요[1/3]

데이터타입 변수명 = 초기값

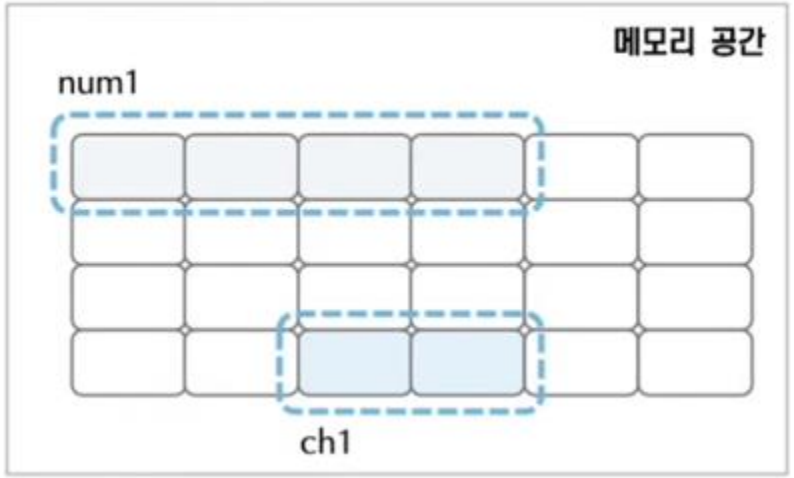
- 변수란 하나의 데이터(data)를 저장하기 위해
  프로그램에 의해 이름을 할당 받은 메모리 공간을 의미

- 변수는 데이터의 유형에 따라 다른 형태를 가지며, 프로그램의 상태를 유지하거나 조작하는 데 사용되며, 자바 프로그래밍에서 핵심적인 요소 중 하나

- Memory(RAM)에는 무수히 많은 번지가 있으나 자바프로그램은 프로그래머가 메모리를 직접 관리하지 않고 변수의 지정을 통해서 관리하면 된다

변수는 data type과 변수명을 사용하여 선언

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



2.1. 변수 개요[2/3] - 자바 변수의 유형

- 변수는 정의된 위치에 따라 4가지의 유형으로 구분하며 각 유형에 따라 각각의 특성이 있다
  - ✓ 지역변수(Local variable)
  - ✓ 매개변수(Parameter variable)
  - ✓ 인스턴스 변수(Instance variable)
  - ✓ 정적(클래스)변수(Class variable)

```
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;
    }
}
```

2.1. 변수 개요[3/3] - 변수 네이밍 규칙

1. 영문자(대소문자), 숫자, 언더스코어(_), 달러($)로만 구성할 수 있다.

2. 변수의 이름은 숫자로 시작할 수 없다.

3. 변수의 이름 사이에는 공백을 포함할 수 없다.

4. 변수의 이름으로 자바에서 미리 정의된 키워드(keyword)는 사용할 수 없다.

5. 변수의 이름은 해당 변수에 저장될 데이터의 의미를 잘 나타내도록 짓는 것이 좋다.

- **(참고) 자바 예약 키워드 : 변수명으로 사용 불가**

- abstract
- continue
- for
- New
- Switch
- assert
- Default
- goto
- Package
- synchronized
- boolean
- do
- if

- private
- this
- break
- double
- implements
- protected
- throw
- byte
- else
- import
- public
- throws
- case
- enum

- instanceof
- Return
- transient
- catch
- extends
- int
- short
- try
- char
- final
- interface
- static
- void
- class
- finally
- long

- strictfp
- volatile
- const
- float
- native
- super
- while

2.2.변수의 선언과 초기화 [1/3]

변수선언?

어떤 타입의 데이터를 저장할 것인지, 변수 이름이 무엇인지를 결정하는 것

언제 하나?

데이터를 보관할 필요가 있을 때

데이터를 보관 후 다음 코드에서 그 데이터를 사용하고 싶을 때

자바 변수 특징

자바에서는 변수 선언시의 타입에 맞는 데이터를 지정해줘야 한다

- 정수형 타입
  → 정수 값만 저장
- 실수형 타입
  → 실수 값만 저장

.

변수 선언 예시

- int Score = 98;
- String Title = " 들개";
- double weight = 52.3;
- boolean finished = true;

2.2. 변수 선언과 초기화[2/3]

- 변수는 사용되기 전에 반드시 초기화해주어야 한다.
- 초기화하지 않은 변수를 사용하면 자바 컴파일러는 에러를 발생시킨다
- 변수선언은 저장되는 data의 type과 이름만 결정한 것

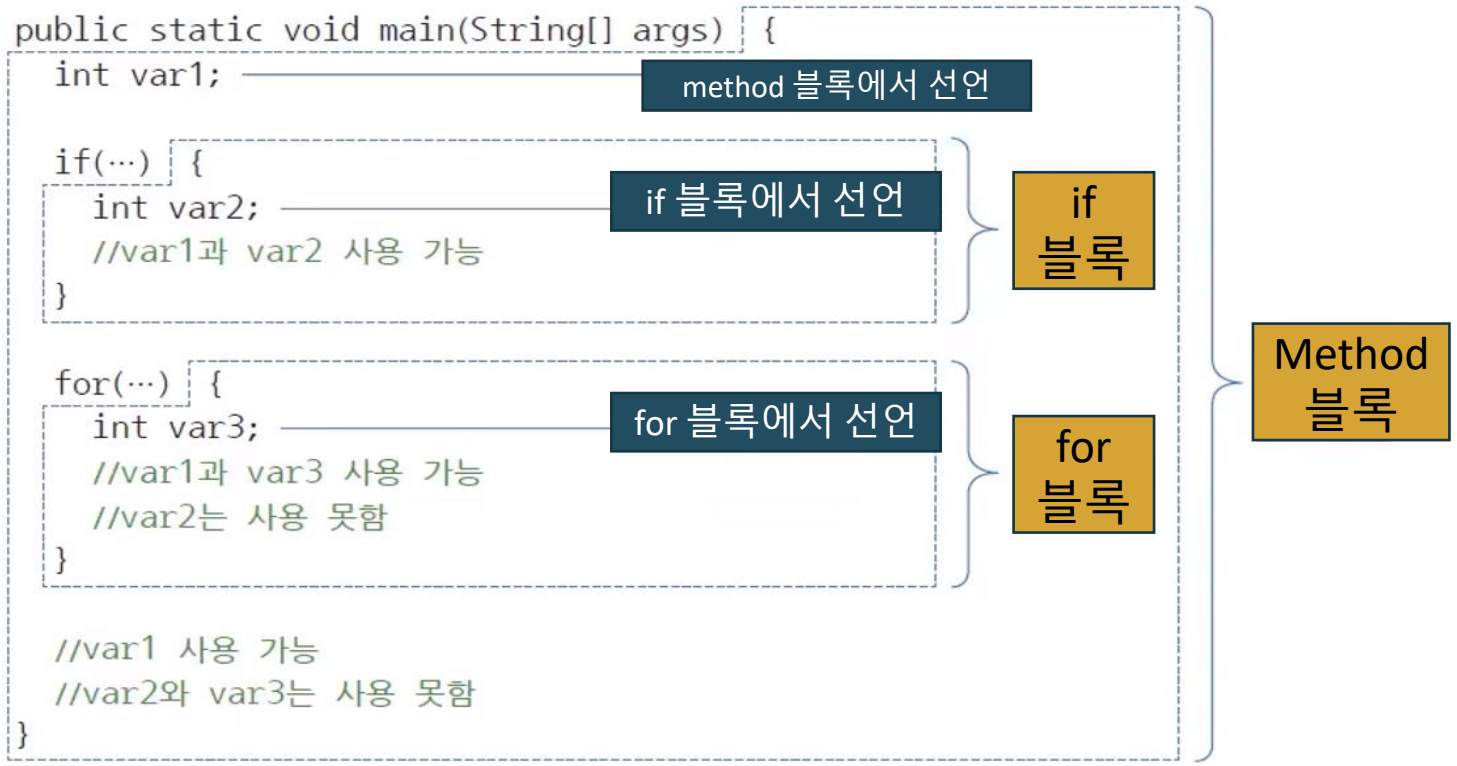
```
int number;   // int값을 갖는 number라는 변수
```

- 변수에 최초로 값이 대입되면 Memory에 할당되고 해당 메모리에 값이 저장됨.

```
int num1 = 55;  //선언과 동시에 초기화
int num2, num3;   // 같은 타입의 변수를 동시에 선언
double   num4 = 3.14 , num5 = 55.46; // 같은 타입의 변수를 동시에 선언 & 초기화
```

2.2. 변수 선언과 초기화[3/3] - 변수 사용 범위(Scope)

- 지역변수(local variable) : 해당 메소드 블록안에서만 유효한 변수
- 전역변수(class variable) : 클래스 전역에 접근할 수 있는 변수
- 인스턴스변수(instance variable) : 객체(인스턴스)가 생성될 때마다 각각의 객체에 속하는 변수



2.3. 자료형 (Data type) [1/12]

- 자바의 모든 변수는 Data type을 가진다
- 자료형에 따라 할당하는 메모리의 크기가 달라진다.

기본형 타입

- 정수형 Byte/ Short/ Int/ long
- 실수형 float/ double
- 문자형 char
- 논리형 Boolean

실제 값을 저장

참조형 타입

- 기본형을 제외한 나머지

String, Array 등

Data의 주소를 저장

2.3. 자료형(Data type) [2/12] - 기본형 타입 (Primitive type)

- 논리형(Boolean) - true와 false중 하나를 값으로 갖으며, 조건식과 논리적 계산에 사용.
- 문자형(char) - 문자를 저장하는데 사용되며, 변수 당 하나의 문자만 저장.
- 정수형(int, long, byte, short) - 정수 값의 저장에 사용.
- 실수형(float, double) - 실수 값의 저장에 사용.

종류 \ 크기	1byte	2byte	4byte	8byte
논리형	boolean			
문자형		char		
정수형	byte	short	int	long
실수형			float	double

2.3. 자료형 (Data type) [3/12] - 정수형 (Integer Type)

- 소수부가 없는 숫자 형
- 자바는 데이터의 표현범위에 따라 4가지의 정수형 타입을 제공한다

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	$-2^{63} \sim 2^{63}-1$

- 정수표현

  정수 리터럴은 기본적으로 int 형을 표현하므로, long 타입을 명시하기 위해서는 접미사 L을 사용

  필요에 따라 접두사(prefix)를 사용하여 8진수(0), 16진수(0x), 2진수(0b)를 표현

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

2.3. 자료형 (Data type) [4/12] - 실수형(Floating-Point Type)

- 소수부를 가진 숫자를 표현하는 자료형
- 자바는 데이터의 표현범위에 따라 2가지의 정수형 타입을 제공한다

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

- 실수의 표현

  실수 리터럴은 기본적으로 double 형을 표현하므로 float 타입 값을 표현하려면 접미사 F를 사용

  double 형을 나타내는 접미사는 D이나 생략할 수 있다

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

2.3. 자료형 (Data type) [5/12] - 문자형 (Character Type)

- 한 개의 문자를 작은 따옴표로 나타내거나 해당문자와 매핑되는 숫자로 나타내는 자료형
- Java는 Ascii code, Uni code 표준을 사용하여 2byte 문자를 숫자(양)로 매핑하여 표현하고 있다
- 문자 한 개는 실제 숫자로 저장된다

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);
```

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

2.3. 자료형 (Data type) [6/12] - 논리형

- 논리형은 True 와 false를 표현하는 자료형이다
  - 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);

    }
}
```

2.3. 자료형 (Data type) [7/12] - 상수 - Constants

- 상수는 값이 변하지 않는 수를 의미한다
  1. 리터럴 상수(Literals Constants) : 55, -8, 'a' "Hello world!"
  2. 사용자 정의 상수 : final double PI = 3.14;

- 리터럴 상수의 기본타입 :
  - 정수형 : int type
  - 실수형 : double type

- 사용자 정의 상수는 변수 선언 후 final 키워드를 붙이면 초기화 후 그 값을 변경할 수 없다

❖ final 키워드는 변수, 메소드 또는 클래스에 사용될 수 있으며 다음과 같은 역할을 한다

✓ 변수에 사용될 경우, final 키워드는 변수를 상수로 만들어 해당 변수의 값을 변경할 수 없게 한다.
✓ 메소드에 사용될 경우, 하위 클래스에서 해당 메소드를 Override(재정의)하지 못하게 하여 메소드의 동작을 변경할 수 없도록 한다.
✓ 클래스에 사용될 경우, 해당 클래스를 상속할 수 없도록 하여 확장되거나 수정되지 않도록 한다.

2.3. 자료형 (Data type) [8/12] - 변수& 상수

변수

- 변수 생성(네이밍) 규칙에 의하여 선언
- 메모리에 저장된 데이터는 변경할 수 있다

```
int num;            // 변수의 선언
num = 20;           // 변수의 초기화
System.out.println(num); // 20
num = 550;          // 변수의 값 변경
System.out.println(num); // 550
```

상수

- final 키워드를 사용하여 선언
- 메모리에 저장된 데이터를 변경할 수 없다

```
final int MAX = 100;
MAX = 200; // 에러
```

2.3. 자료형 (Data type) [9/12] - 리터럴 상수와 접미사

1. 정수형 리터럴 → 아라비아 숫자와 부호(123, -456)
2. 실수형 리터럴 → 소수 부분을 가지는 아라비아 숫자(3.14, -55.8)
3. 논리형 리터럴 → true 또는 false
4. 문자형 리터럴 → 작은따옴표('')로 감싸진 문자로 표현('a', 'Z')
5. 문자열 리터럴 → 큰따옴표("")로 감싸진 문자열로 표현(" 리터럴", "홍길동“)
6. null 리터럴 → 단 하나의 값 (null)

리터럴 상수

타입 접미사	리터럴 타입	예제
L 또는 l	long형	123456789L
F 또는 f	float형	1.234567F
D 또는 d (생략 가능)	double형	1.2345D

접미사 예시

2.3. 자료형 (Data type) [10/12] - 변수의 기본값

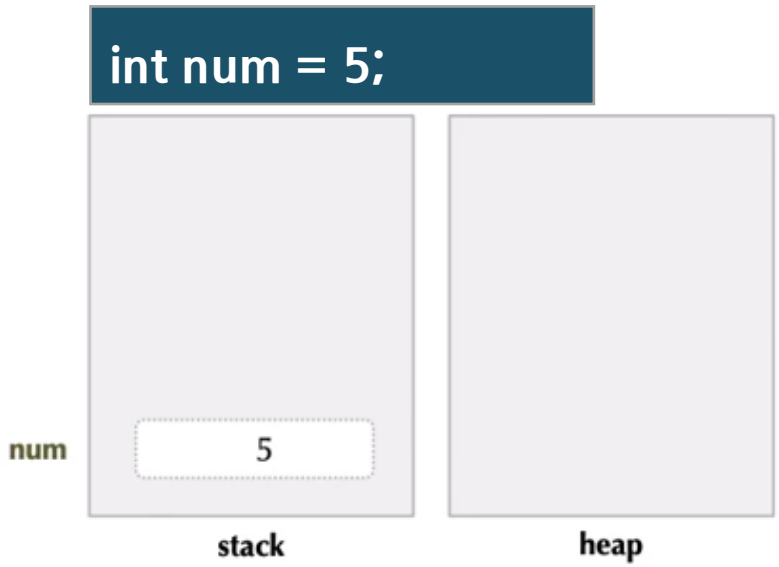
변수는 Data type에 따라 다음과 같은 기본 값을 갖는다

Data type	기본값
boolean	false
char	'₩u0000'
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d 또는 0.0
참조형 변수	null

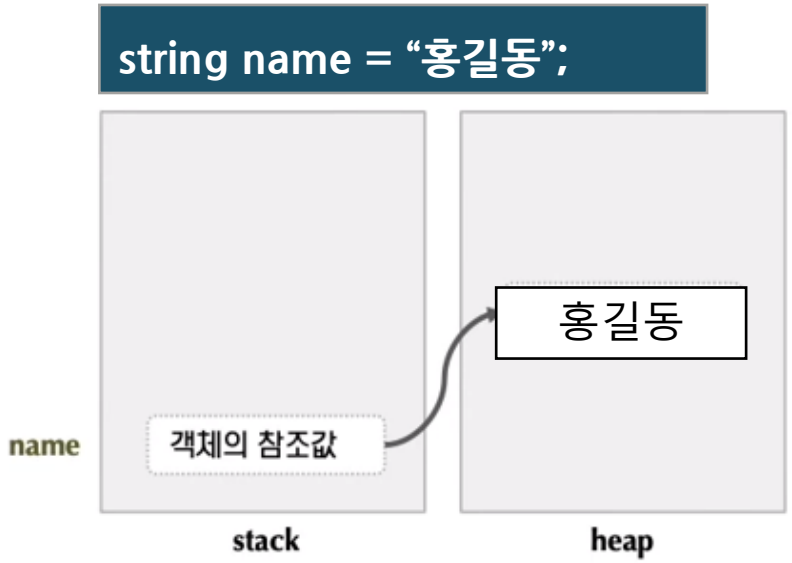
```
boolean isGood = false;
char grade = ' ';     // 공백
byte b = 0;
short s = 0;
int i = 0;
long l = 0;       // 0L로 자동변환
float f = 0;      // 0.0f로 자동변환
double d = 0;     // 0.0로 자동변환
String s1 = null;
String s2 = "";          // 빈 문자열
```

2.3. 자료형(Data type) [11/12] - 참조 형(Reference Type)

- 참조타입(Reference Type)이란 객체의 번지를 참조하는 타입이다
  - ✓ String type, 배열 , 열거(Enum), Class, Interface등
- 참조자료형 변수는 4byte의 크기를 갖으며 instence객체에 접근할 수 있는 정보(번지수)를 갖는다
- 객체가 더 이상 사용되지 않을 때는 Garbage Collector에 의해 자동 제거된다



기본형타입



참조형타입

2.3. 자료형 (Data type) [12/12] - 기본타입 변수 & 참조타입 변수

- 변수들은 모두 Stack 영역에 생성된다
- 기본타입 변수는 Stack 영역에 직접 해당 값을 저장한다
- 참조타입변수는 Heap 영역에 있는 객체가 생성된 메모리 번지수를 저장한다

- 참조타입변수는 아직 번지를 저장하고 있지 않다는 뜻으로 null값으로 초기화할 수 있다
  - ✓ string va1 = “Hello”;
  - ✓ string va2 = null;
- 기본타입변수는 null로 초기화 할 수 없다
  - ✓ int score = null ; // error 발생
  - ✓ 기본타입 변수의 기본값 참조

2.4. Wrapper 클래스[1/2]

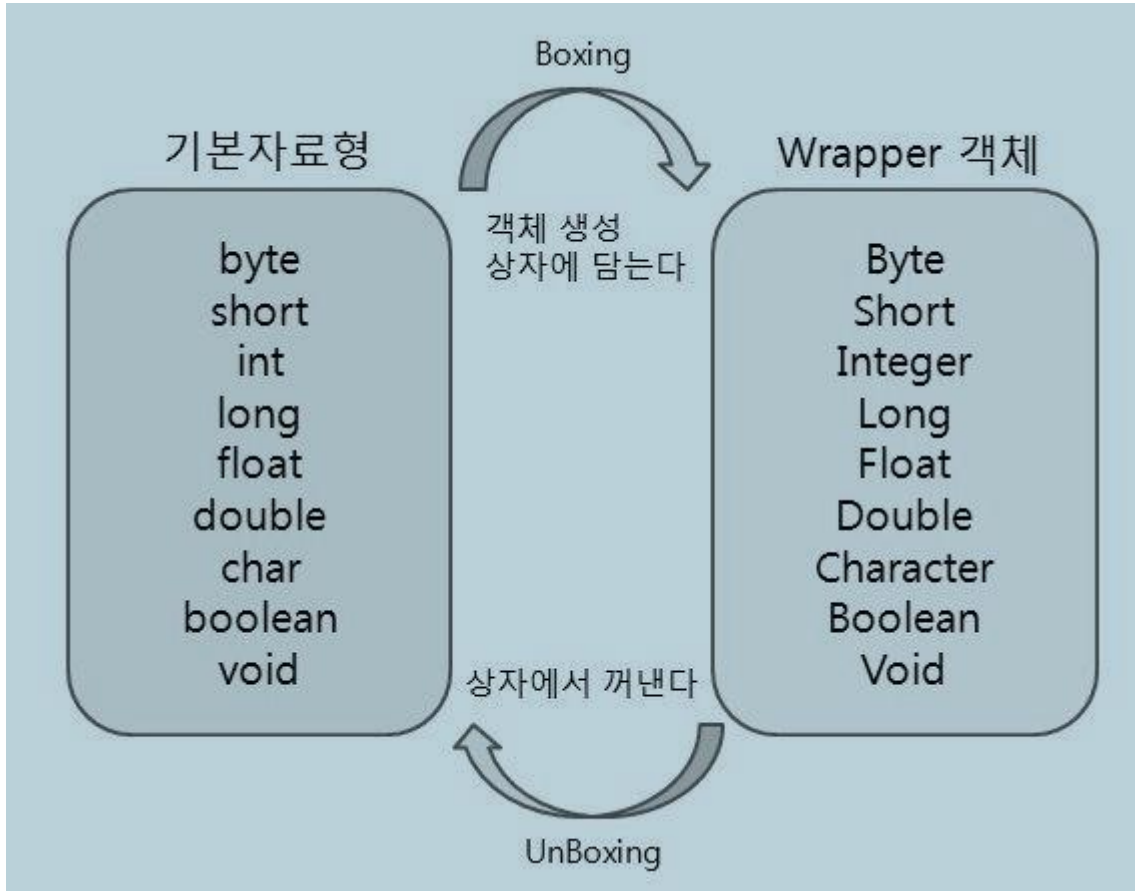
- Wrapper 클래스는 기본 데이터 타입(primitive data type)을 객체로 다룰 수 있게 해주는 클래스이다
- Wrappe 클래스들은 기본데이터타입에 대한 객체화와 함께 다양한 기능들을 제공하고 있다

Wrapper 클래스가 있는 이유와 장점

- 객체 지향 프로그래밍과 상호작용
  - 기본 데이터 타입은 값 자체만 저장하고 메소드를 가질 수 없기 때문에, 객체로 다루기 어렵다. Wrapper 클래스를 사용하면 기본 데이터 타입을 객체로 감싸서 객체처럼 다룰 수 있어 객체 지향적인 프로그래밍에 도움을 준다.
- 컬렉션과 제네릭 활용
  - 많은 자바 컬렉션 프레임워크는 객체를 다루기 때문에 기본 데이터 타입을 사용할 수 없다 Wrapper 클래스를 사용하여 기본 데이터 타입을 컬렉션에 담을 수 있습니다
- Null 값 다루기
  - 기본 데이터 타입은 null 값을 가질 수 없지만 Wrapper 클래스는 null 값을 가질 수 있어서 객체가 없는 상황을 표현할 수 있다
- 메소드 활용
  - Wrapper 클래스는 기본 데이터 타입을 감싸기 때문에 객체로써 메소드를 호출할 수 있다

2.4. Wrapper 클래스[2/2] - Boxing과 UnBoxing

- Wrapper Class는 산술연산을 위해 정의된 클래스가 아니기 때문에, 이 클래스의 객체 인스턴스에 저장된 값을 변경이 불가능하며 값을 저장하는 새로운 객체의 생성 및 참조만 가능하다




```
int pint = 42;
Integer wint = Integer.valueOf(pint);
            // int를 Integer 객체로 변환

int eint = wint.intValue();
            // Integer를 다시 int로 변환
```

2.5. 콘솔 출력과 키보드입력 [1/3] - 콘솔 출력

- 출력방법
  - ✓ print(), printf(), println()을 사용한다

메소드	실행
println(내용);	괄호 안의 내용을 출력하고 행을 바꿔라.
print(내용);	괄호 안의 내용을 출력하고 행은 바꾸지 말아라.
printf("형식문자열", 값1, 값2, …);	형식 문자열에 맞추어 뒤의 값을 출력해라.

println()

- 문자열을 출력하기 위해서는 큰따옴표로 묶어줘야 한다.
- 큰 따옴표 없는 것은 모두 변수명으로 인식한다. (입력한 변수명이 없을 경우 에러발생)
- 문자열과 변수명을 함께 사용할 수 있으나 반드시 '+'로 연결시켜줘야 한다.
- 가로안의 내용을 출력한 후 자동으로 줄바꾸는 기능이 있다.

2.5. 콘솔 출력과 키보드입력 [2/3] - 형식화된 출력 printf()

- printf()는 변수의 값을 지시자를 통해서 여러 가지 형식으로 변환하여 출력할 수 있다.
  - ✓ 출력 후 줄바꿈을 하지 않는다. 줄바꿈을 하려면 지시자 '%n'을 넣어줘야 한다.
  - ✓ 출력하려는 값의 수만큼 지시자도 사용해야 한다.
  - ✓ 출력될 값과 지시자의 순서는 일치해야 한다.
  - ✓ 지시자를 제외한 문자는 입력한 그대로 출력된다.

- system.out.printf("출력 서식",출력할 내용);

- 출력서식
  - ✓ % [-] [0] [n] [.m] 지시자

지시자	실행
%b	boolean 형식으로 출력
%d	정수 형식으로 출력
%o	8진수 정수의 형식으로 출력
%x 또는 %X	16진수 정수의 형식으로 출력
%f	소수점 형식으로 출력
%c	문자형식으로 출력
%s	문자열 형식으로 출력
%n	줄바꿈 기능
%e 또는 %E	지수 표현식의 형식으로 출력

2.5. 콘솔 출력과 키보드입력 [3/3] - 키보드 입력 처리

- 자바에서 키보드 입력을 처리하는 가장 일반적인 방법은 Scanner 클래스를 사용하는 것이다
  - ✓ Scanner 객체명 = new Scanner(system.in);

예제코드

```
import java.util.Scanner;
public class KeyboardInputExample {
        public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("이름을 입력하세요: ");
        String name = scanner.nextLine(); System.out.print("나이를 입력하세요: ");
        int age = scanner.nextInt(); System.out.println("입력한 이름: " + name);
        System.out.println("입력한 나이: " + age);
         scanner.close();
        }
}
```

3

연산자

1. 연산자 개요
2. 연산자의 종류
3. 타입변환
4. 문자열 결합 연산
5. Math

3.1. 연산자 개요

피연산자를 대상으로 특정기능을 수행하고 결과를 반환하는 특수 기호

산술연산자	대입연산자	증감연산자	비교연산자
사칙연산 +, -, *, /, %	변수에 값을 대입할 때 사용	피연산자를 1씩 증가 또는 감소시킬 때 사용	피연산자 사이의 상대적인 크기를 판단

논리연산자	비트연산자	삼항연산자	Instanceof 연산자
주어진 논리식을 판단하여, 참과 거짓을 결정	비트(bit) 단위로 논리 연산을 할 때 사용	? 앞의 조건식에 따라 결과가 참이면 1 반환, 결과가 거짓이면 2 반환 Ex) 조건식 ? 1 :2	참조 변수가 참조하고 있는 인스턴스의 실제 타입 반환

3.2. 연산자의 종류[1/8]

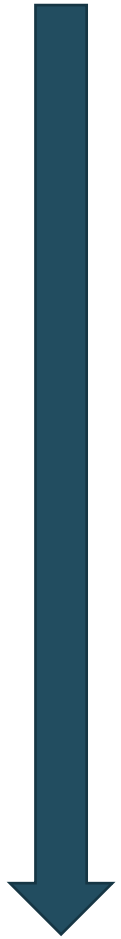
산술연산자	+	a+b	a와 b의 합
	-	a-b	a와 b의 차
	*	a*b	a와 b의 곱
	/	a/b	a를 b로 나눈 몫
	%	a%b	a를 b로 나눈 나머지
대입연산자	+=	a+=b	a= a+b
	-=	a-=b	a= a-b
	*=	a*=b	a = a*b
	/=	a/=b	a= a/b
	%=	a%=b	a= a%b
	&=	a&=b	a= a&b
	|=	a|=b	a= a|b
	^=	a^=b	a= a^b
증감연산자	++	i++	i=i+1
	--	i--	i=i-1

비교연산자	>	a > b	a가 b보다 큰 경우 true
	>=	a>=b	a가 b보다 크거나 같을 경우 true
	<	a<b	a가 b보다 작을 경우 true
	<=	a<=b	a가 b보다 작거나 같을 경우 true
	==	a==b	a와 b가 같을 경우 true
	!=	a!=b	a와 b가 같지 않을 경우 true
비트연산자	&	a&b	비트단위의 AND
	|	a|b	비트단위의 OR
	^	a^b	비트단위의 XOR
	~	a~b	비트단위의 보수
	>>	a>>b	a를 b만큼 오른쪽으로 이동 (빈 자리는 양수는 0 음수는 1)
	>>>	a>>>b	a를 b만큼 오른쪽으로 이동 (빈자리는 항상 0)
	<<	a<<b	a를 b만큼 왼쪽으로 이동
논리연산자	&&	a && b	a와 b가 모두 true 인 경우 true
	||	a||b	a 또는 b가 true 인 경우 true
	!	!a	a가 true 면 false , false 면 true

3.2. 연산자의 종류[2/8] - 연산 우선순위

연산자	연산 방향
증감(++, --), 부호(+, -), 비트(~), 논리(!)	←
산술(*, /, %)	→
산술(+, -)	→
쉬프트(<<, >>, >>>)	→
비교(<, >, <=, >=, instanceof)	→
비교(==, !=)	→
논리(&)	→
논리(^)	→
논리(|)	→
논리(&&)	→
논리(||)	→
조건(?:)	→
대입(=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=)	←

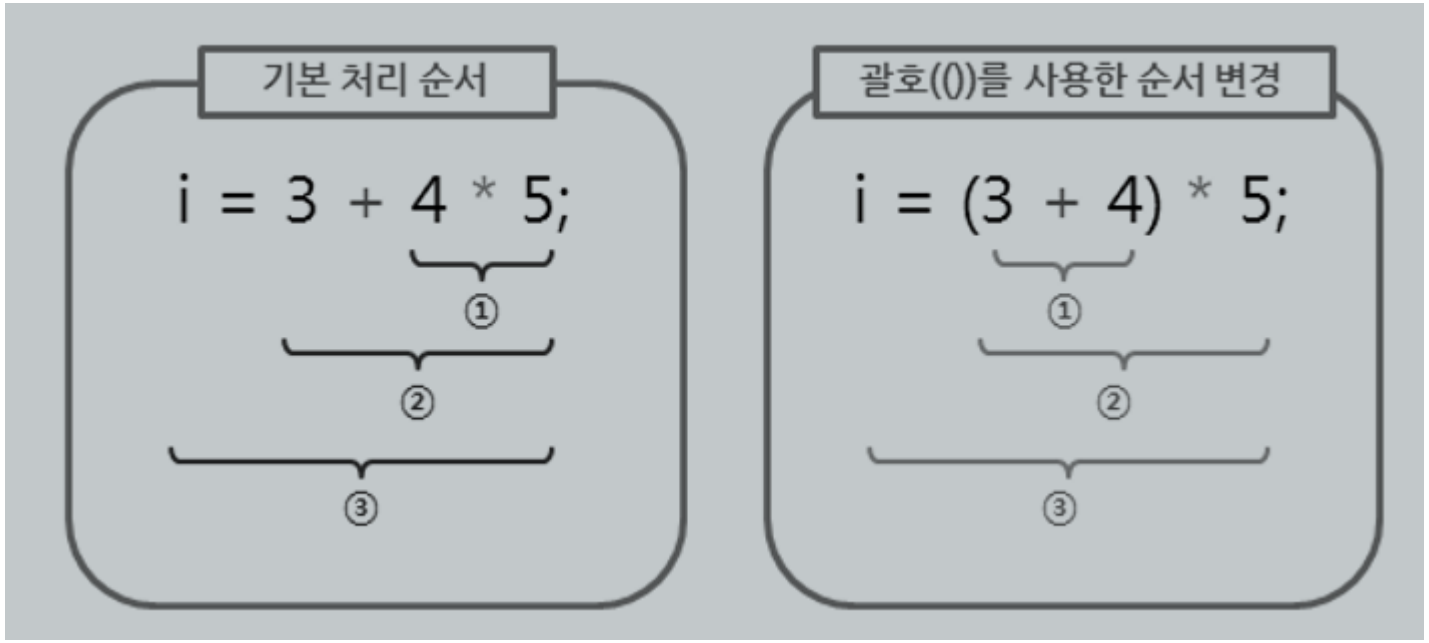
우선순위



괄호의
우선순위가
가장 높다

3.2. 연산자의 종류[3/8] - 연산자의 우선순위와 결합방향

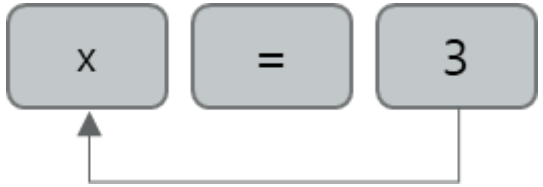
- 자바에서는 수식 내에 여러 연산자가 함께 등장할 때, 어느 연산자가 우선인지를 정함
- 필요시 가장 우선순위가 높은 괄호를 이용해 연산처리순서 변경 가능



3.2. 연산자의 종류[4/8] - 대입 연산자

- 대입연산자는 이항연산자로 = 기호를 중심으로 오른쪽(우항) 값을 왼쪽(좌항)에 저장하는 기능
- 좌항에는 변수만 있을 수 있다
- L value 와 R value 의 타입이 같아야 한다 → 타입 다른 경우 기준은 L value 이다

L value = **R** value



- 대입연산자는 산술연산자와 함께 복합 연산자로 사용할 수 있다

연 산 자	예	의 미
+=	x += 10	x = x + 10
-=	x -= 10	x = x - 10
*=	x *= 10	x = x * 10
/=	x /= 10	x = x / 10
%=	x %= 10	x = x % 10

3.2. 연산자의 종류[5/8] - 증감 연산자

증가연산자(++)

피연산자의 값을 1 증가

감소 연산자(--)

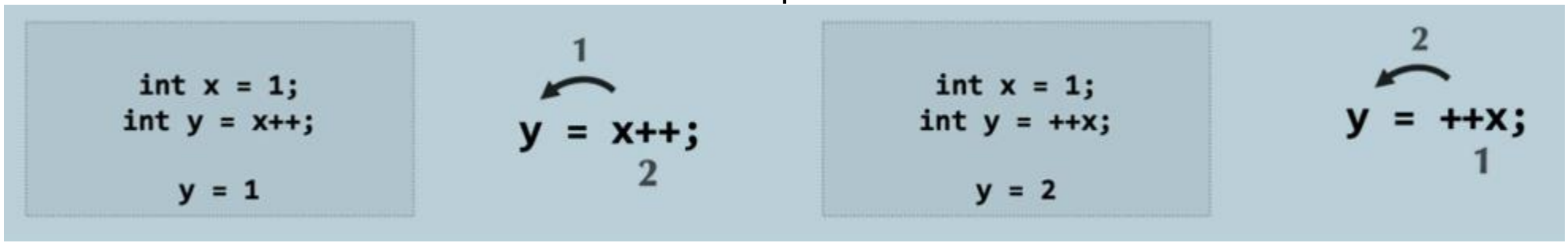
피연산자의 값을 1 감소

전위(++x , --x)

값이 참조되기 전 증가(감소)

후위 (x++ , x--)

값이 참조된 후 증가(감소)



3.2. 연산자의 종류[6/8] - 비교 연산자

피연산자를 같은 타입으로
변환 후 비교

↓

기본형 & 참조형에 사용 가능

↓

Boolean 형에는 사용 불가

참조형에는 ==와 !=만 사용

↓

결과 값은 true 또는 false

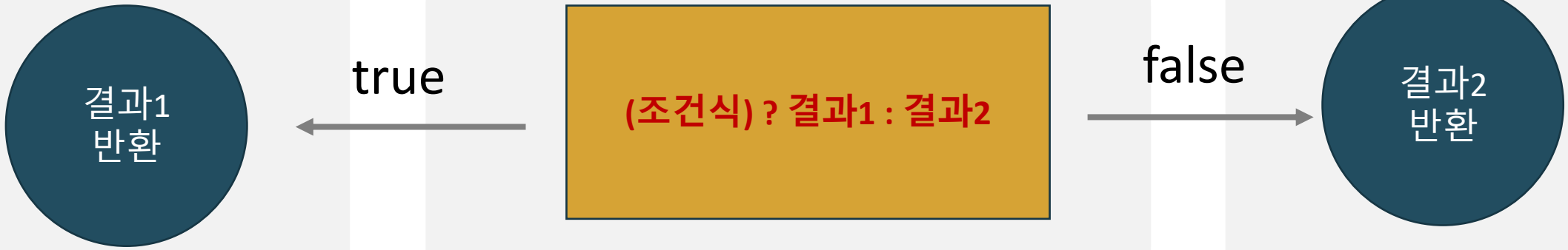
비교 연산자 연산 결과

수 식	연 산 결 과
x > y	x가 y보다 클 때 true, 그 외에는 false
x < y	x가 y보다 작을 때 true, 그 외에는 false
x >= y	x가 y보다 크거나 같을 때 true, 그 외에는 false
x <= y	x가 y보다 작거나 같을 때 true, 그 외에는 false
x == y	x와 y가 같을 때 true, 그 외에는 false
x != y	x와 y가 다를 때 true, 그 외에는 false

3.2. 연산자의 종류[7/8] - 논리 연산자

논리 연산자	설명
&&	논리식이 모두 참이면 참을 반환 (논리 AND 연산)
||	논리식 중에서 하나라도 참이면 참을 반환 (논리 OR 연산)
!	논리식의 결과가 참이면 거짓을, 거짓이면 참을 반환 (논리 NOT 연산)

3.2. 연산자의 종류[8/8] - 삼항연산자



- 조건식의 연산결과가 true이면 '결과1' 값을 반환하고 false이면 '결과2'의 값을 반환한다.

```
int age = 27;
String adult = (age >= 20) ? "adult" : "child";
```

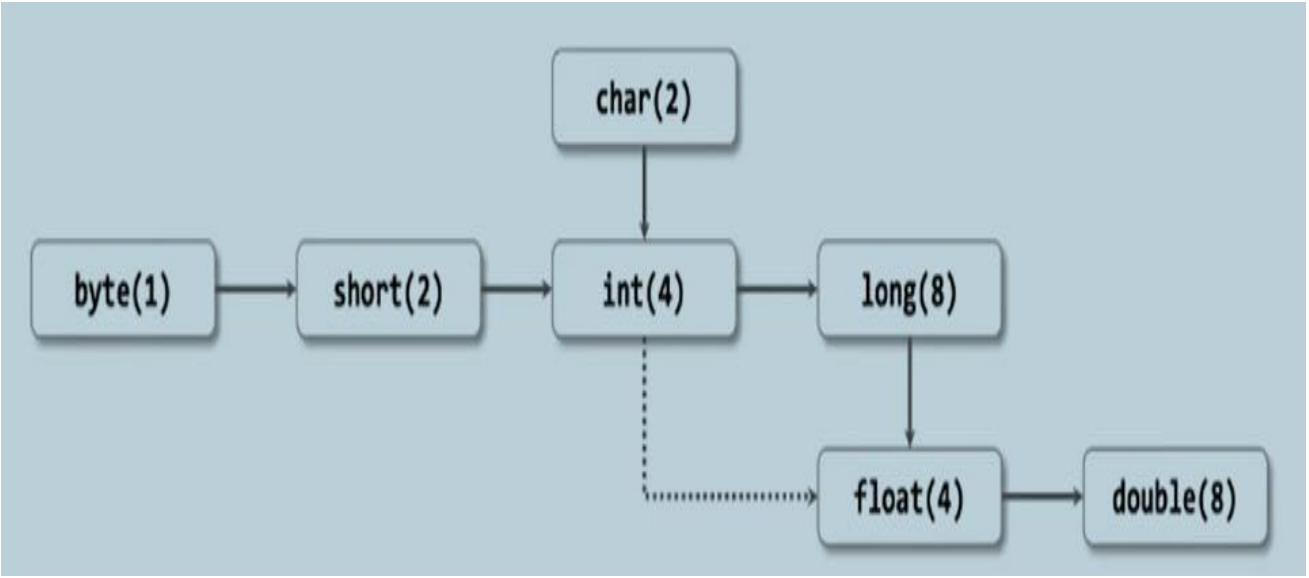
3.3. 타입변환[1/3] - 자동 타입 변환

- 대입 연산이나 산술 연산에서 컴파일러가 자동으로 수행해주는 타입 변환

- 자바에서는 데이터의 손실이 발생하는 대입 연산은 허용하지 않고, 손실이 최소화되는 방향으로 자동으로 타입을 변환

- Data Type이 다른 변수에 값을 할당하는 경우
- Data Type이 다른 값을 연산하는 경우

✓ 1) double d = 2; // 2 => 2.0
✓ 2) int 1 = 5 / 2; // 2
✓ 3) double d = 1; // 2 => 2.0



3.3. 타입변환[2/3] - 강제타입 변환(Casting)

- 사용자가 타입 캐스트 연산자(())를 사용하여 강제적으로 수행하는 타입 변환

- 예) int n=1 , n2=4;
  1) double var1= n1/n2;
  2) double var2= (double)n1/n2;
  ✓ 1) 의 실행 결과 → 0.0
  ✓ 2) 의 실행 결과 → 0.25

3.3. 타입변환[3/3] - 자바_타입변환 주의사항

type error

원인과

타입관련

주의사항

나눗셈 연산

- int와 int의 연산 결과 → int
- double과 int의 연산결과 → double
- 숫자간 연산은 더 큰 타입을 따른다

타입 불일치

변수에 값을 대입할 때, 값과 변수의 타입이 같아야 함

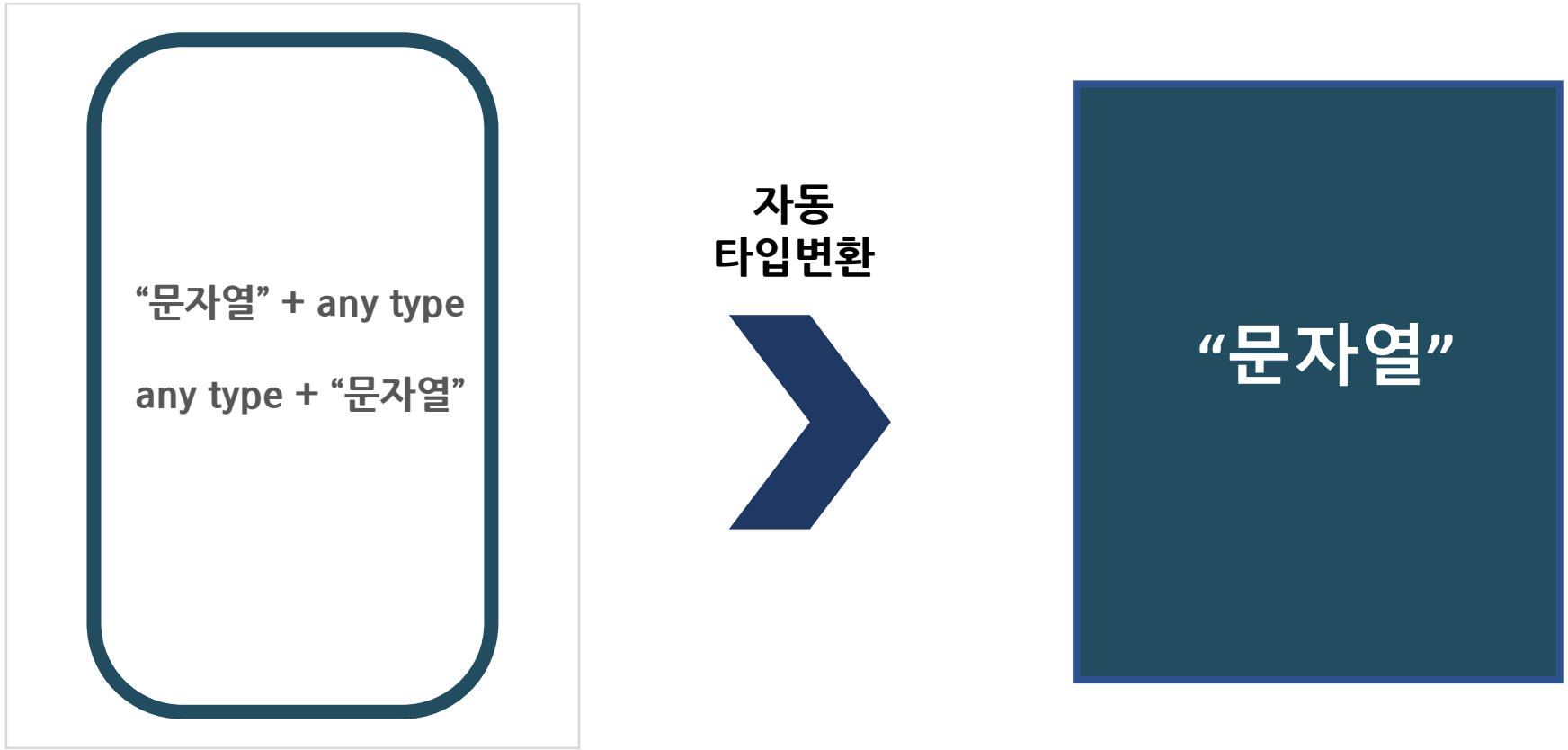
자동 타입변환

데이터타입의 표현 범위에 따라 더 큰 타입으로 자동 타입 변환

byte → short → int → long → float → double

3.4. 문자열 결합 연산[1/2] - 자동 타입 변환

- 자바의 + 연산자는 피연산자가 모두 숫자일 경우 덧셈 연산을 수행하고 피연산자중 하나가 문자열인 경우 나머지 피연산자도 자동 변환되어 문자열 결합연산을 수행한다
  - string s = “3” + 7 ; → string s = “37” ;



3.4. 문자열 결합 연산[2/2] - 문자열 타입을 기본타입으로 변환

- 프로그래밍시 문자열을 숫자타입으로 변환하는 경우가 많다
- 자바에서 문자열을 기본타입으로 변환하는 방법
- 기본타입을 문자열로 변환하는 방법

변환타입	예제
String → int	String n = “100”; int var = Integer.parseInt(n);
String → double	String n = “100.01”; double var = Double.parseDouble(n);
String → boolean	String n = “true”; int var = Boolean.parseBoolean(n);
기본타입 → String	String str = String.valueOf(기본타입값)

3.5. Java Math

- Java Math class에는 수학 계산을 간단하게 처리할 수 있는 메소드가 포함되어 있다
  - ✓ Math.max(x,y)

```
Math.max(5, 10);
```

  - ✓ Math.min(x,y)

```
Math.min(5, 10);
```

  - ✓ Math.sqrt(x)

```
Math.sqrt(64);  // 제곱근
```

  - ✓ Math.abs(x)

```
Math.abs(-4.7); // 절대값
```

  - ✓ Math.random()

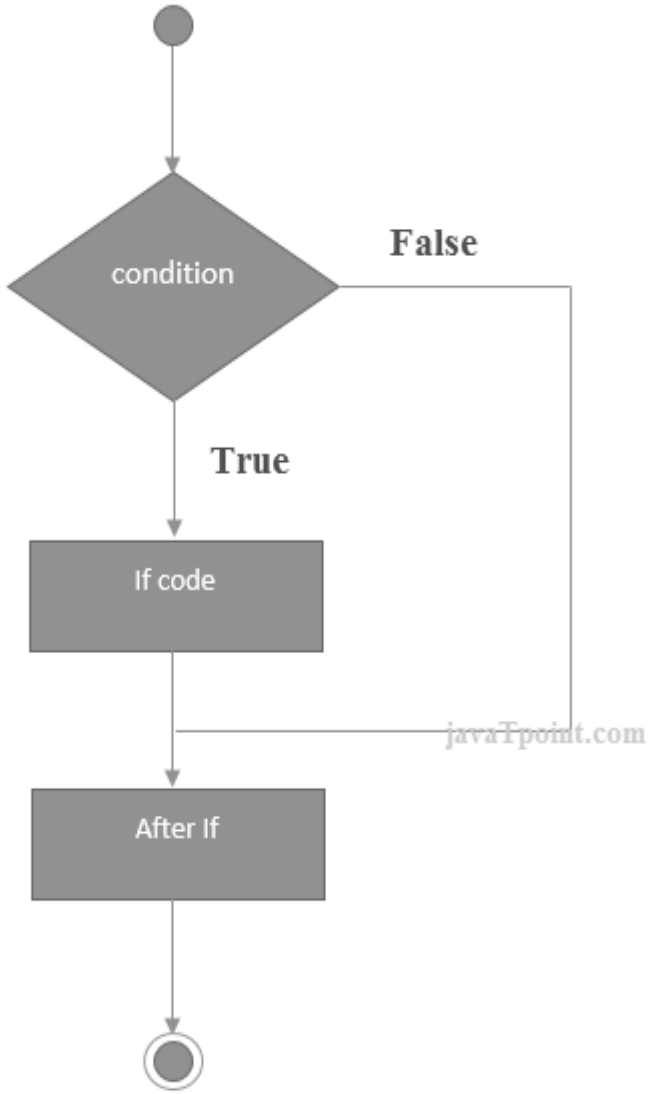
```
Math.random();  // 0.0~1.0사이의 난수
int randomNum = (int)(Math.random() * 101);  // 0 to 100 사이의 (int) 난수
```

4

제어문

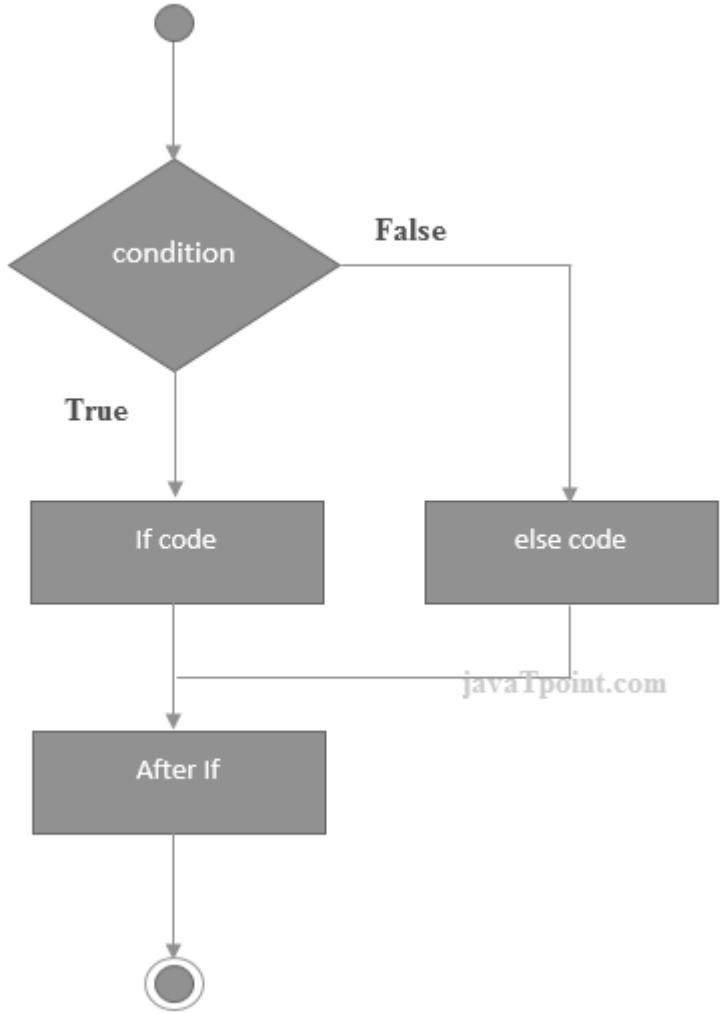
1. 조건문
2. 반복문
3. Loop 제어문

4.1. 조건문조건문[1/4] - If 문



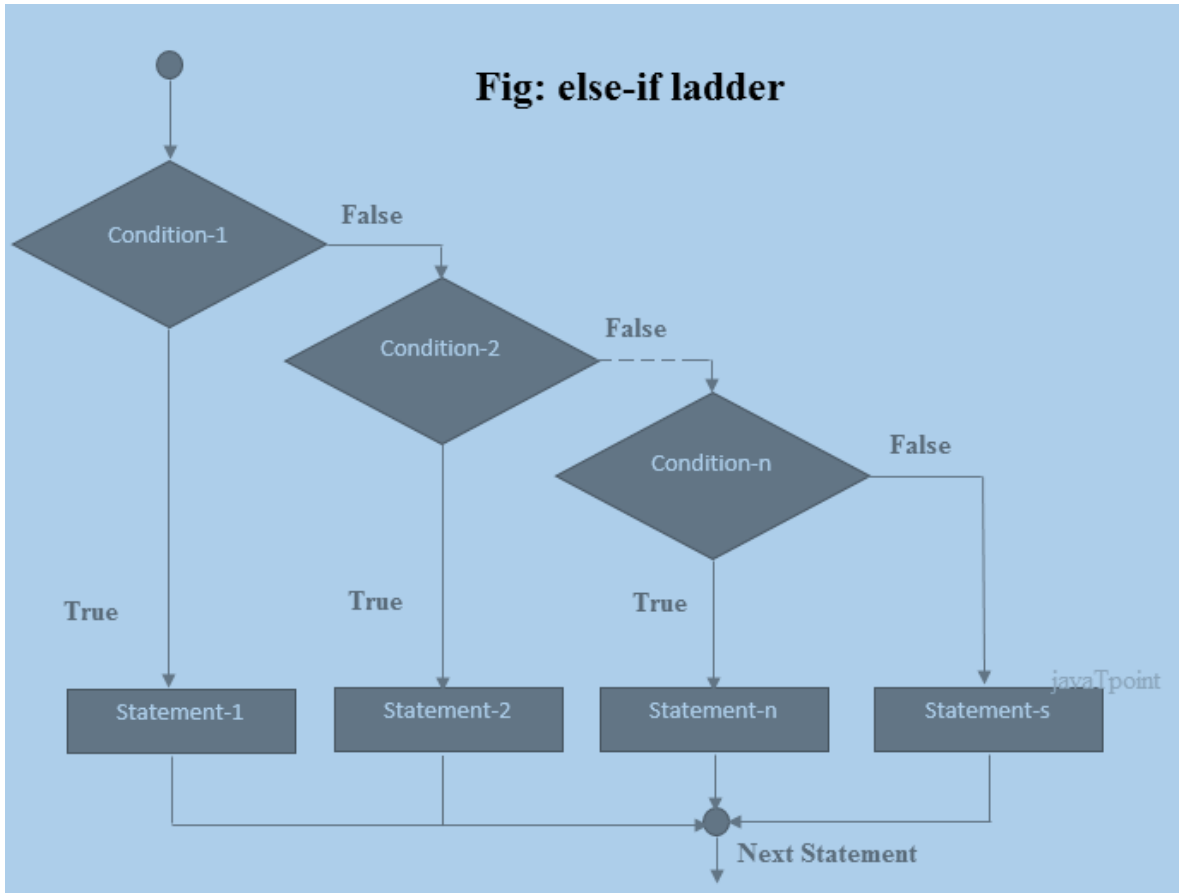

```
if (condition) {

    code                    //조건식이 참일 때

}
```

4.1. 조건문조건문[2/4] - If - else



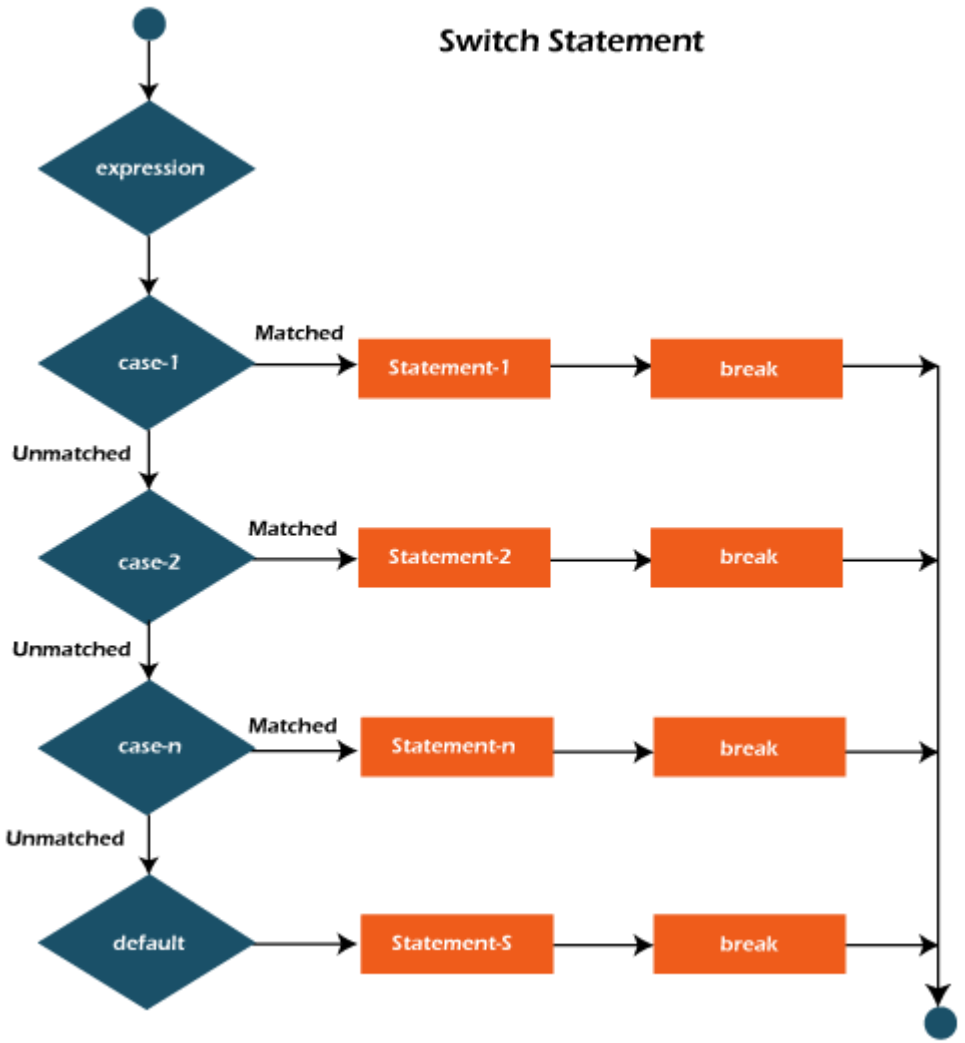

```
if (condition) {

   If code              //조건식이 참일 때
}
else {
    ese code            //조건식이 참이 아닐 때
}
```

4.1. 조건문[3/4] - If - else if




```
if (condition 1) {
실행 내용1   //조건식1이 참일 때
} else if (condition 2){
실행 내용2   //조건식2가 참일 때
} else if (condition n){
실행 내용n   //조건식n이 참일 때
} else {
실행 내용s ...   //앞 조건들이 모두 참이 아닐 때
}
```

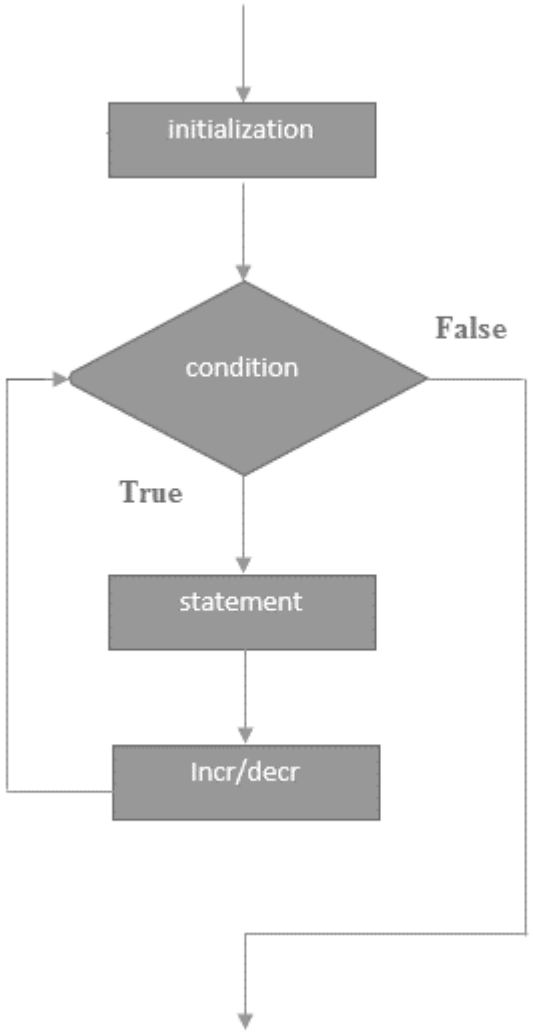
4.1. 조건문[4/4] - switch




```
switch (조건식) {
case 값1 :
실행내용 ;
break;
case 값2 :
실행내용 ;
break;
…
case 값  n :
실행내용 ;
break;
default :
실행내용 s;
break;
}
```

- switch 문은 if / else 문과 마찬가지로 주어진 조건 값의 결과에 따라 프로그램이 다른 명령을 수행하도록 하는 조건문
- switch 문의 조건 값으로는 int형과 string형 가능
- default 절은 조건 값이 위에 나열된 어떠한 case 절에도 해당하지 않을 때만 실행.
- default 절은 필수요소는 아니며 필요할 때만 선언.

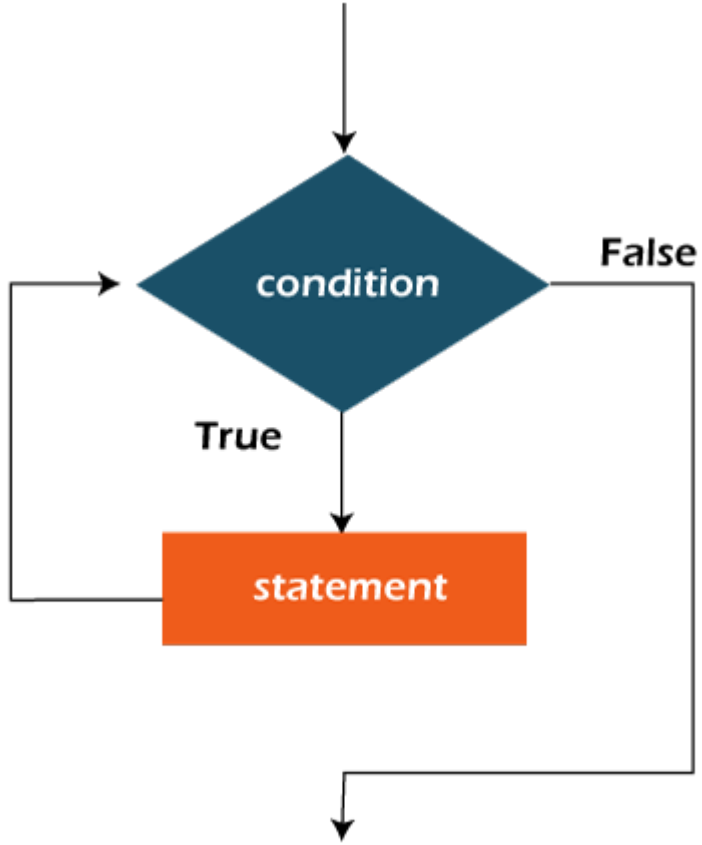
4.2. 반복문[1/3] - for문




```
1.public class ForExample {
2.public static void main(String[] args) {
3.
4.    for(int i=1; i<=10; i++){
5.        System.out.println(i);
6.    }
7.}
8.}
```

- for 문은 while 문과는 달리 자체적으로 초기식, 조건식, 증감식을 모두 포함하고 있는 반복문
- for 문에서 직접 선언된 변수는 for 문이 종료되면 같이 소멸

4.2. 반복문[2/3] - while문

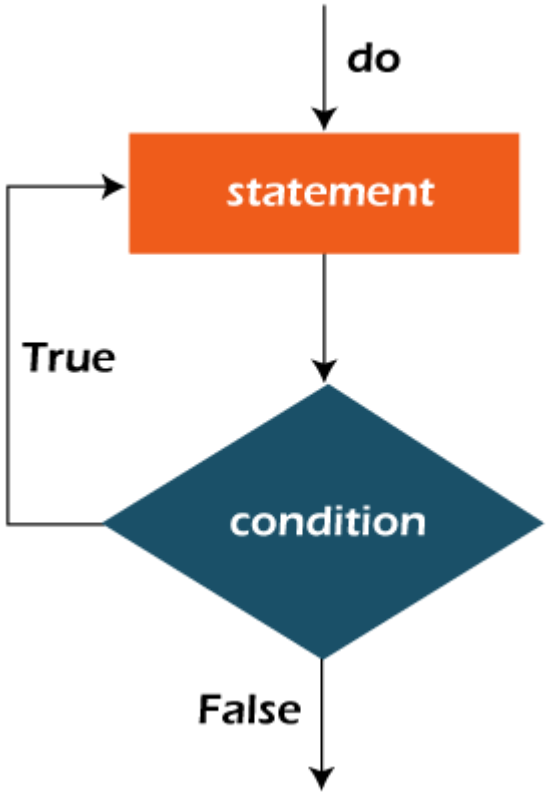



```
1.public class WhileExample {
2.public static void main(String[] args) {
3.    int i=1;
4.    while(i<=10){
5.        System.out.println(i);
6.        i++;          //이 부분 없으면 무한 루프에 빠짐
7.    }
8.}
9.}
```

1. 조건식이 참(true)인지를 판단하여, 참이면 내부의 명령문을 실행
2. 내부의 명령문을 전부 실행하고 나면, 다시 조건식으로 돌아와 또 한 번 참인지를 판단

** 조건식의 검사를 통해 반복해서 실행되는 반복문을 루프(loop)라고 한다.

4.2. 반복문[3/3] - do-while




```
1.public class DoWhileExample {
2.public static void main(String[] args) {
3.    int i=1;
4.    do{
5.        System.out.println(i);
6.    i++;
7.    }while(i<=10);
8.}
9.}
```

- while 문은 루프에 진입하기 전에 먼저 조건식부터 검사
- do / while 문은 먼저 루프를 한 번 실행한 후에 조건식을 검사
- 즉, do / while 문은 조건식의 결과와 상관없이 무조건 한 번은 루프 실행

4.3. Loop 제어문[1/3] - continue

Loop의 제어

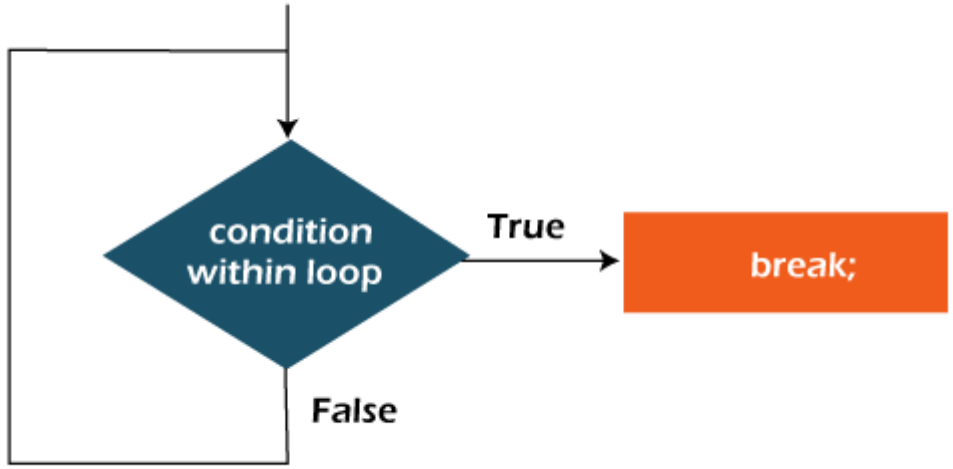
- 자신이 포함된 반복문의 끝으로 이동
- Continue문 이후의 문장은 수행되지 않음
- continue 문은 루프 내에서 사용하여 해당 루프의 나머지 부분을 건너뛰고, 바로 다음 조건식의 판단으로 넘어가게 해 줌
- 보통 반복문 내에서 특정 조건에 대한 예외 처리를 할 때 사용

```
1.public class ContinueWhileExample {
2.public static void main(String[] args) {
3.    //while loop
4.    int i=1;
5.    while(i<=10){
6.       if(i==5){
7.          //using continue statement
8.          i++;
9.          continue;//it will skip the rest statement
10.       }
11.       System.out.println(i);
12.       i++;
13.   }
```

실행결과

```
1
2
3
4
6
7
8
9
10
```

4.3. Loop 제어문[2/3] - break




```
int num = 1, sum = 0;

while (true) {          // 무한 루프
   sum += num;
   if (num == 100) {
      break;
   }
   num++;
}
System.out.println(sum);
```

- break 문은 루프 내에서 사용하여 해당 반복문을 완전히 종료시킨 뒤, 반복문 바로 다음에 위치한 명령문 실행
- 루프 내에서 조건식의 판단 결과와 상관없이 반복문을 완전히 빠져나가고 싶을 때 사용

4.3. Loop 제어문[3/3] - break with label

여러 반복문이 중첩된 상황에서 한 번에 모든 반복문을 빠져나가거나, 특정 반복문 까지만 빠져나가고 싶을 때 사용

현재 반복문이 아닌 해당 이름의 반복문 바로 다음으로 프로그램의 실행을 옮겨 준다

Break with Label

자바에서 반복문을 가리키는 이름(label)은
break 문이나 continue 문에만 사용

Label의 사용

```
allLoop:
for (int i = 2; i < 10; i++) {
   for (int j = 2; j < 10; j++) {
      if (i == 5) {
         break allLoop;
      }
      System.out.println(i + " * " + j +
                                 " =" + (i * j));
   }
}
```

예제-9*9단 4단까지

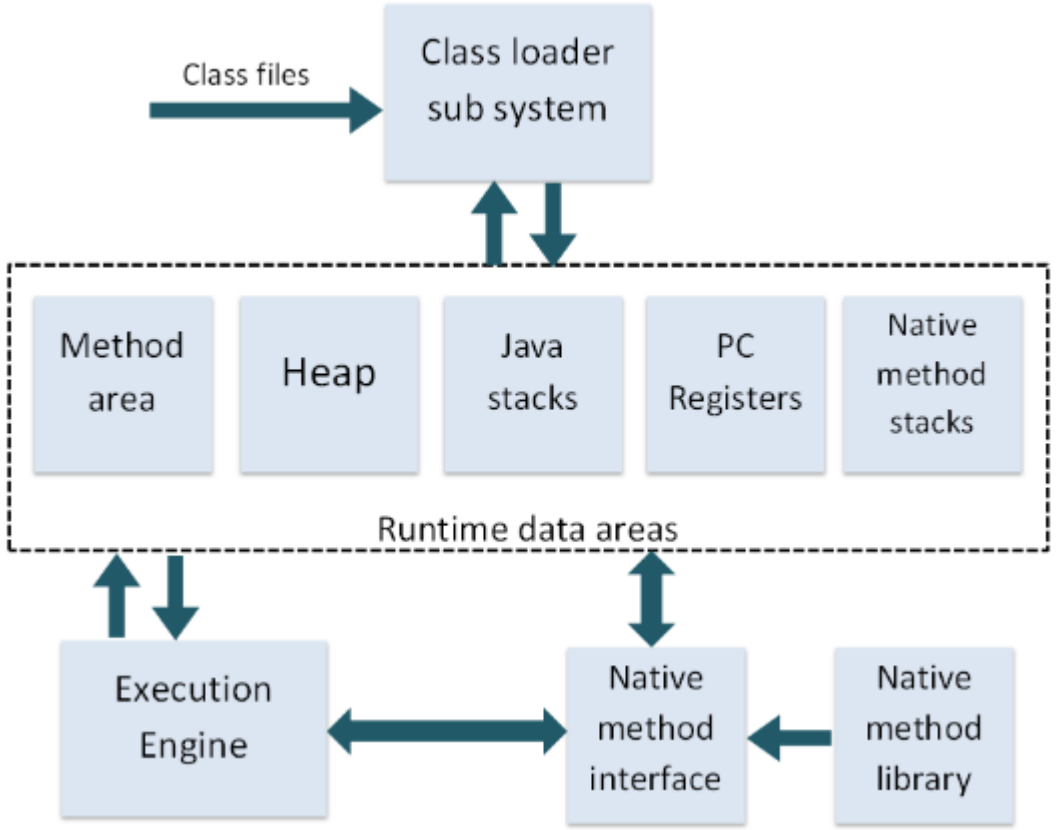
5

배열

1. 메모리 구조 이해
2. 배열(Array) 개요
3. 배열의 생성과 초기화
4. 배열요소 사용법
5. 배열과 반복문
6. 다차원 배열

5.1. 메모리 구조 이해 - JVM Memory Structure

- 프로그램이 실행되면 JVM은 OS로부터 메모리를 할당받고, 그 메모리를 용도에 따라 여러 영역으로 나누어 관리한다
- JVM의 메모리는 크게 쓰레드별로 생성되는 데이터 영역과 전체 쓰레드가 공유하는 데이터 영역으로 나뉘며, Run-Time Data Areas 라고 부른다



1.메소드 영역 (Method Area):

- 클래스 정보, 정적 변수, 메소드 코드 등이 저장되는 영역
- JVM이 시작될 때 생성되며, 모든 스레드가 공유한다.

2.힙 (Heap):

- 동적으로 생성된 객체 인스턴스가 저장되는 영역
- 가비지 컬렉터에 의해 관리되며, 메모리 할당 및 해제가 이루어진다.

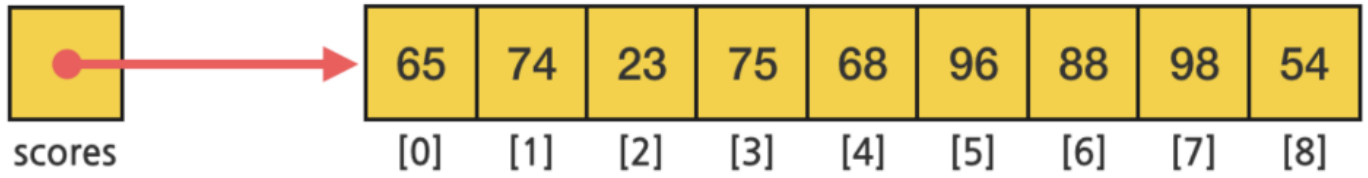
3.스택 (Stack):

- 각 스레드마다 별도로 생성되는 영역으로, 메소드 호출 시 지역 변수와 메소드 호출 정보가 저장된다.

5.2. 배열(Array) 개요

- **Java 배열은** 같은 타입의 요소를 저장하는 데이터 구조를 말한다
- 배열의 요소(element) : 배열을 구성하는 각각의 값
- 인덱스(Index) : 배열에서의 위치를 가리키는 숫자 (0부터 양의 정수)
- length 멤버를 사용하여 배열의 길이를 얻을 수 있음

- **int[] scores = { 65, 74, 23, 75, 68, 96, 88, 98, 54 };**
- **특징**
- **배열은 인덱스(index)라는 순서를 가지며, 모든 값의 타입이 같다.**



5.3. 1차원배열 생성과 초기화

생성과 동시에 초기화

자료형[] 변수 = {데이터1, 데이터2, 데이터3, ... };

```
public static void main(String[] args)
{ String[] beer = {"Kloud", "Cass", "Asahi", "Guinness", "Heineken"};
// 인덱스 번호 : 0 , 1 , 2 , 3 , 4
System.out.println(beer[0]); // Kloud
System.out.println(beer[1]); // Cass
System.out.println(beer[2]); // Asahi
System.out.println(beer[3]); // Guinness
System.out.println(beer[4]); // Heineken }
}
```

공간 할당 후 값 대입

자료형[] 변수 = new 자료형[배열 크기];

변수[0] = 데이터 값;

변수[1] = 데이터 값;

```
public static void main(String[] args) {
        int[] num = new int[3]; // 크기가 3인 배열 생성
        num[0] = 10; // 0번 index에 값 할당
        num[1] = 15; // 1번 index에 값 할당
        num[2] = 13; // 2번 index에 값 할당
        for (int i = 0; i < num.length; i++) {
                System.out.println(num[i]);
        }
}
```

5.4. 배열 요소(element) 사용법

- 배열의 요소는 인덱스를 통해 사용한다.

- 배열의 값 읽기

```
int[] scores = {99, 88, 77};
System.out.println(scores[0]); // 99
System.out.println(scores[1]); // 88
System.out.println(scores[2]); // 77
```

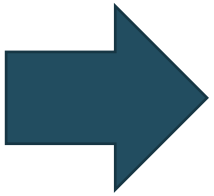
- 배열의 값 변경

```
int[] scores = {99, 88, 77};
System.out.println(scores[0]); // 99
scores[0] = 0; // 0번 인덱스 값 변경
System.out.println(scores[0]); // 0
```

5.5. 배열과 반복문

- 배열은 보통 반복문과 함께 활용된다

```
int[] arr = { 7, 9, 31, 2, 6 };
int sum = arr[0];
sum += arr[1];
sum += arr[2];
sum += arr[3];
sum += arr[4];
```



```
int[] arr = { 7, 9, 31, 2, 6 };
sum = 0;
    for (int i = 0; i < 5; i++) {
     sum += arr[i]
}
```

```
int[] arr = { 7, 9, 31, 2, 6 };
sum = 0;
    for (int i = 0; i < arr.lenth; i++) {
     sum += arr[i]
}
```

- 배열이름.length 키워드로 배열의 길이를 얻을 수 있음

5.6. 다차원 배열

- 2차원 이상의 배열을 의미하며, 배열 요소로 또 다른 배열을 가지는 배열
  - ✓ 일반적으로 2차원배열을 주로 사용한다
- 2차원 배열은 배열 요소로 1차원 배열을 가지는 배열
- [] 의 개수가 차원의 수를 의미한다

선언방법	선언예
타입[][] 변수이름;	int[][] score;
타입 변수이름[][];	int score[][];
타입[] 변수이름[];	int[] score[];

```
Int [] [] score = new int [3] [2]; //3행2열의 2차원 배열 생성
```

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)

6

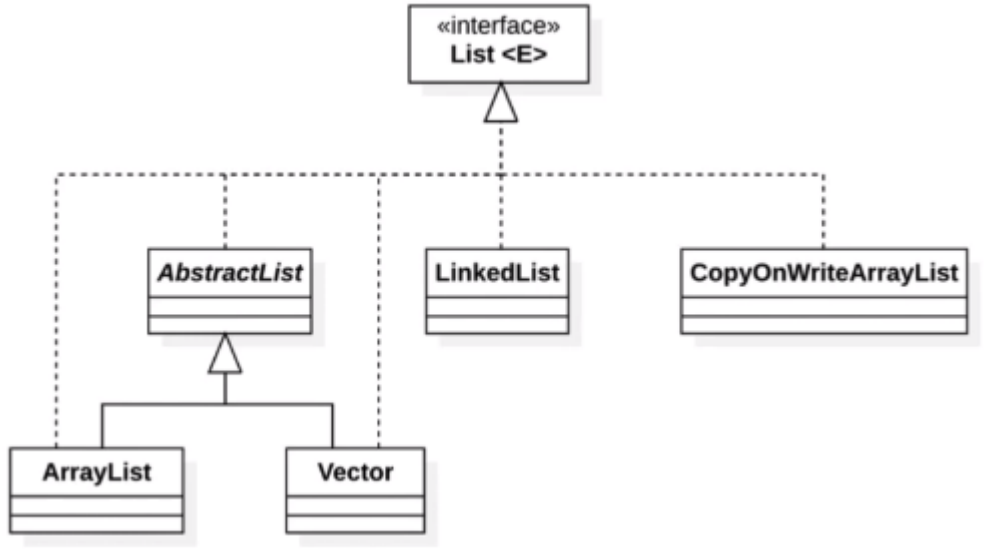
컬렉션 프레임워크와 배열

1. List 인터페이스
2. ArrayList와 배열

6.1. List 인터페이스

- List 인터페이스는 Collection의 다른 인터페이스들과 가장 큰 차이는 배열처럼 순서가 있다는 것이다
- List 계열의 컬렉션은 저장요소를 순차적으로 관리하며 요소로 중복 값과 null값을 가질수 있다
- 요소에 대한 접근은 인덱스를 통해서 한다
- List 계열의 대표 클래스는 ArrayList, LinkedList가 있다

List Interface 구현 클래스들



List

+ add(Object) : boolean
+ get(int) : Object
+ set(int, Object) : Object
+ listIterator() : ListIterator
+ remove(int) : Object
+ subList(int, int) : List

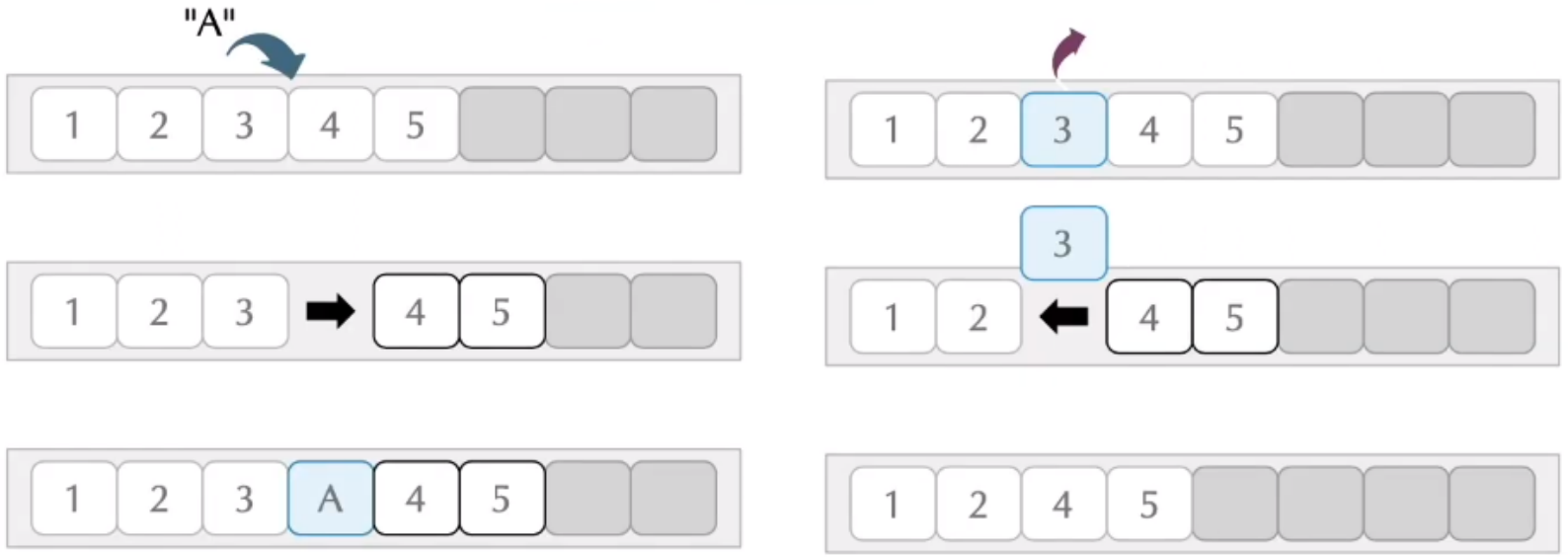
6.2. ArrayList와 배열(1)

- ArrayList 클래스는 배열을 이용하기 때문에 인덱스를 이용해 배열 요소에 빠르게 접근할 수 있다.

- 배열은 크기를 변경할 수 없는 인스턴스이므로, 내부배열의 용량(Capacity)을 넘어 요소를 저장할 경우 내부적으로 용량을 늘린 새로운 배열을 만들어 요소를 옮겨야 한다
  물론 이 과정은 자동으로 수행되지만, 요소의 추가 및 삭제 작업에 시간이 걸리는 단점을 가지게 된다.

6.2. ArrayList와 배열(2)

- ArrayList는 순차적으로 요소를 저장할 수 있는 메소드와 인덱스를 이용하여 저장할 수 있는 add()메소드를 제공한다
- 특정 인덱스 위치에 요소를 저장할 경우 내부적으로 기존에 저장된 요소들의 이동이 발생한다
- 특정요소의 삭제에도 해당 요소의 인덱스가 필요하며 이때도 기존 요소들의 이동이 이루어진다
- 이런 점 때문에 요소들의 추가나 삭제가 빈번하게 이루어지는 데이터 관리에는부하가 많이 걸려 퍼포먼스가 떨어질 수 있으므로 적합하지 않다



7

객체지향 프로그래밍 기본 개념

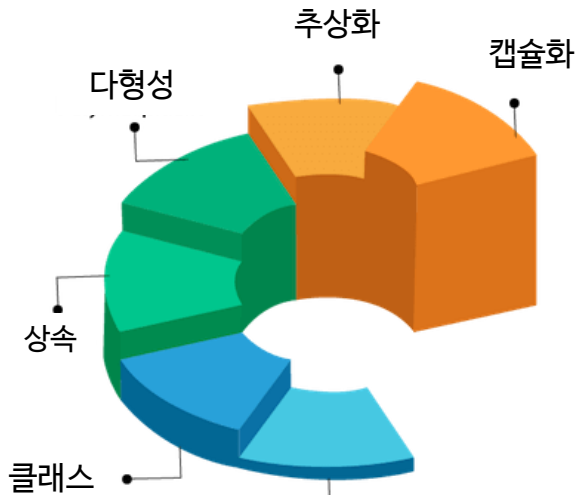
1. 객체지향 프로그래밍 개요
2. 클래스와 객체
3. 메소드와 함수
4. 생성자
5. 메소드 오버로딩과 오버라이딩
5. this 키워드
6. 접근 제한자
7. static 키워드와 final키워드

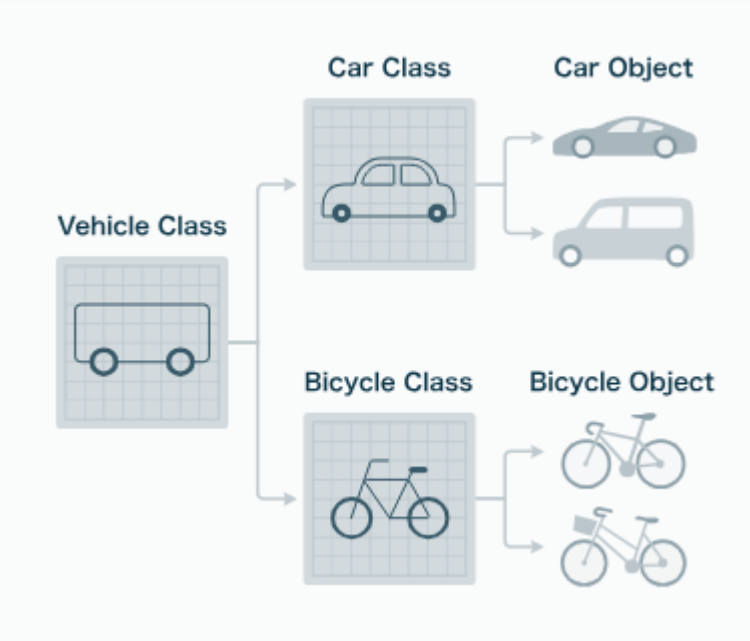
7.1. 객체지향 프로그래밍 개요

- 객체지향 프로그래밍은 코드를 보다 모듈화하고 유지보수하기 쉽게 만들어주며, 프로그램의 구조를 더 잘 이해하고 설계할 수 있게 도와준다.

- 자바에서는 클래스를 통해 객체의 특성과 동작을 정의하며, 이러한 객체들 간에는 상속과 인터페이스를 통해 관계를 형성한다.

- 다형성은 같은 이름의 메소드나 클래스가 다양한 상황에서 다르게 동작할 수 있는 기능을 의미하며,
- 추상화는 현실세계에서 존재하는 모든 사물의 공통점과 공통 기능을 유추해서 클래스를 만들어내는 일련의 과정을 의미한다





7.2. 클래스와 객체(Object)[1/6]

- 객체(Object) 란?
  - ✓ 구체적 추상적 데이터의 단위를 객체(object)라고 표현한다
  - ✓ 객체는 크게 속성(필드), 동작(메소드)로 구성된다
  - ✓ 예) 회사원의 속성 : 이름, 나이, 소속사등
    - ✓ 회사원의 동작 : 출근하다, 점심먹다, 근무하다, 퇴근하다 등

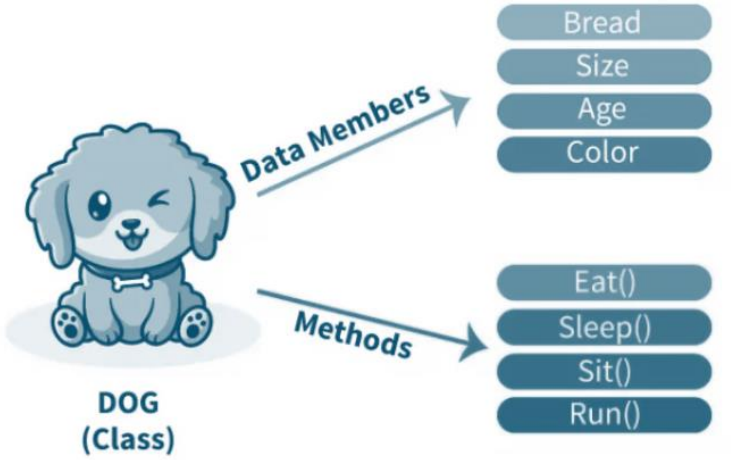
- 클래스를 정의한다는 것은 객체를 만들기 위한 과정이며 클래스는 객체에 대한 설계도라 할 수 있다
- 클래스란 ? 객체를 코드로 구현한 것, 객체지향 프로그래밍의 가장 기본적인 요소이다

- 인스턴스란 ? 클래스로 객체를 만드는 것을 “인스턴스화”라고 한다
  - ✓ 객체의 또 다른 표현으로는 인스턴스(instance) 또는 인스턴스 객체 (instance object)가 있다

7.2. 클래스와 객체(Object)[2/6]

클래스란?

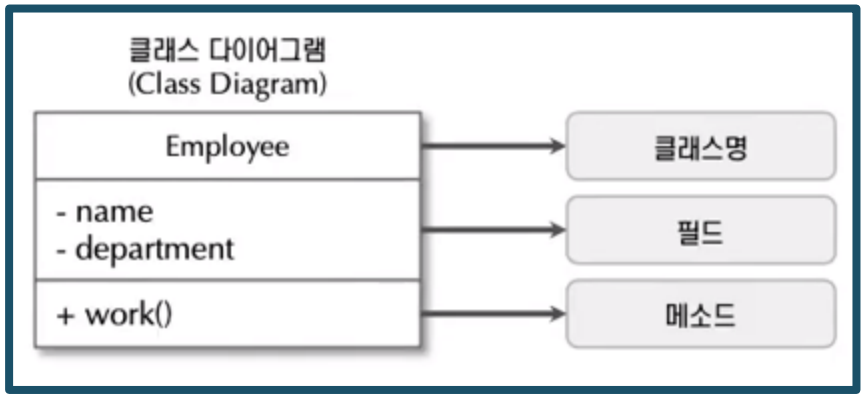
- 객체에 대한 설계도
- 객체의 속성과 동작(기능)을 코드로 구현한 것
- 클래스로 객체를 만드는것
  → “인스턴스화”



- 객체(Object)
- 인스턴스(instance)
- 인스턴스 객체 (instance object)

7.2. 클래스와 객체(Object)[3/6] - 클래스 선언

- 필드(Fields): 클래스 내부에서 데이터를 저장하는 변수를 의미한다
- 메소드 (Methods): 클래스 내부에서 함수 또는 동작을 정의하는 부분이다
- 생성자 (Constructors): 객체를 생성할 때 호출되는 특별한 메소드로, 초기화 작업을 수행한다.




```
public class Employee {

    private String name;
    private String department;                          Fields

    public Employee(String name, String department){
        this.name = name;
        this.department = department;                   Constructor
    }

    public void work(){
        ...                                             Method
    }

}
```

7.2. 클래스와 객체(Object)[4/6] - 필드(field) 개요

- 필드는 클래스에 포함된 변수를 의미한다. 즉, 객체의 속성을 정의하는 공간이다.
- 클래스 내에서 필드는 선언된 위치에 따라 다음과 같이 구분할 수 있다.
  - ✓ 클래스 변수(static variable)
  - ✓ 인스턴스 변수(instance variable)
  - ✓ 지역 변수(local variable)
- 필드는 클래스 내부에 선언되며, 해당 클래스의 인스턴스가 생성될 때마다 각각 다른 값을 가질 수 있다
- 필드의 접근 제어자를 통해 외부에서의 접근을 제어할 수 있고, private, protected, public 등의 제어자를 사용하여 필드의 가시성을 설정한다.

```
class Variable {
        static int classVariable;
        int instanceVariable;
        void variableMethod() {
                int localVariable;
        }
}
```

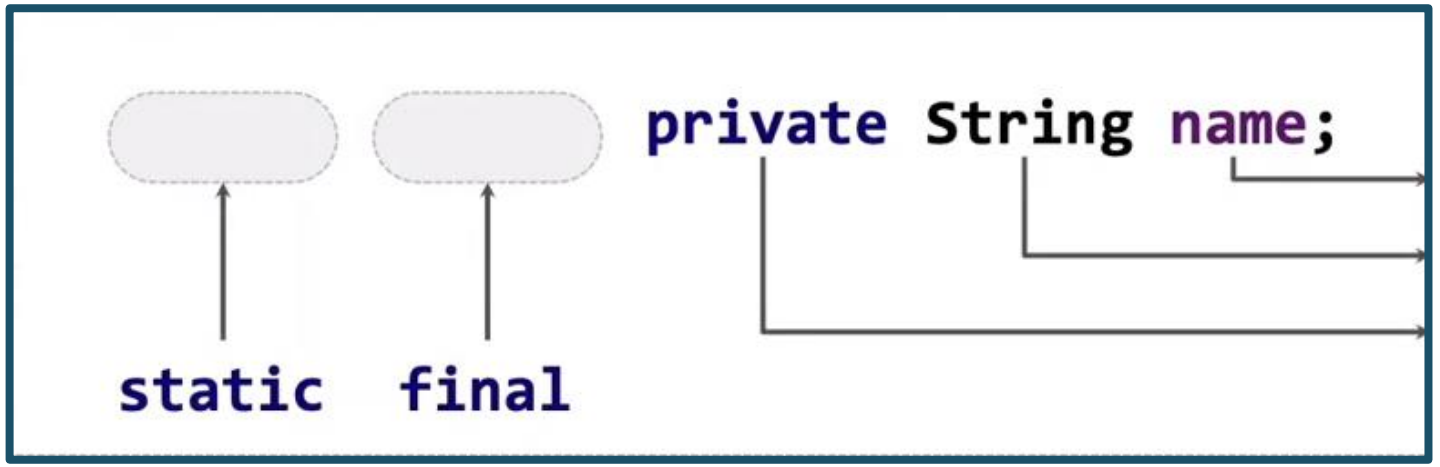
클래스 변수(static 변수 또는 공유변수)

인스턴스 변수(static 없음)

지역변수(메소드 내에서만 사용 가능)

7.2. 클래스와 객체(Object)[5/6] - 필드 정의(선언)

- 필드를 정의할 때는 정의하고자 하는 클래스의 특성을 정확하게 이해하고 해당 클래스의 핵심 속성들을 정의한다
- 필드를 정의할 때는 접근제한자, 타입, 필드명으로 선언한다



- 필드명
- 타입
- 접근제한자

7.2. 클래스와 객체(Object)[6/6] - 각 변수들의 특징

- 클래스 변수는 인스턴스를 생성하지 않고도 바로 사용할 수 있다
- 클래스 변수는 해당 클래스의 모든 인스턴스가 공유해야 하는 값을 유지하기 위해 사용된다
- 반면에 인스턴스 변수는 인스턴스마다 가져야 하는 고유한 값을 가진다
- 클래스 변수와 인스턴스 변수는 초기화를 하지 않아도 변수의 타입에 맞게 자동으로 초기화된다
- 지역 변수는 사용하기 전에 초기화하지 않으면, 자바 컴파일러가 오류를 발생시킨다

변수	생성 시기	소멸 시기	저장 메모리	사용 방법
클래스 변수	클래스가 메모리에 올라갈 때	프로그램이 종료될 때	메소드 영역	클래스이름.변수이름
인스턴스 변수	인스턴스가 생성될 때	인스턴스가 소멸할 때	힙 영역	인스턴스이름.변수이름
지역 변수	블록 내에서 변수의 선언문이 실행될 때	블록을 벗어날 때	스택 영역	변수이름

7.3. 메소드[1/7] - 메소드와 함수 개요

- 메소드(method)란 어떠한 특정 작업을 수행하기 위한 명령문의 집합이라 할 수 있다
- 객체의 기능(행위)에 해당하는 부분이다.
  - ✓ 메소드의 역할은 해당 클래스의 데이터를 제어하는 일이다

- 자주 사용할 기능을 코드로 미리 정의해두고 필요할 때 호출해서 사용
  - ✓ 중복되는 코드의 반복적인 프로그래밍을 피할 수 있다
  - ✓ 모듈화로 인한 코드의 가독성 향상
  - ✓ 프로그램에 문제가 발생하거나 기능의 변경이 필요할 때도 손쉽게 유지보수를 할 수 있다

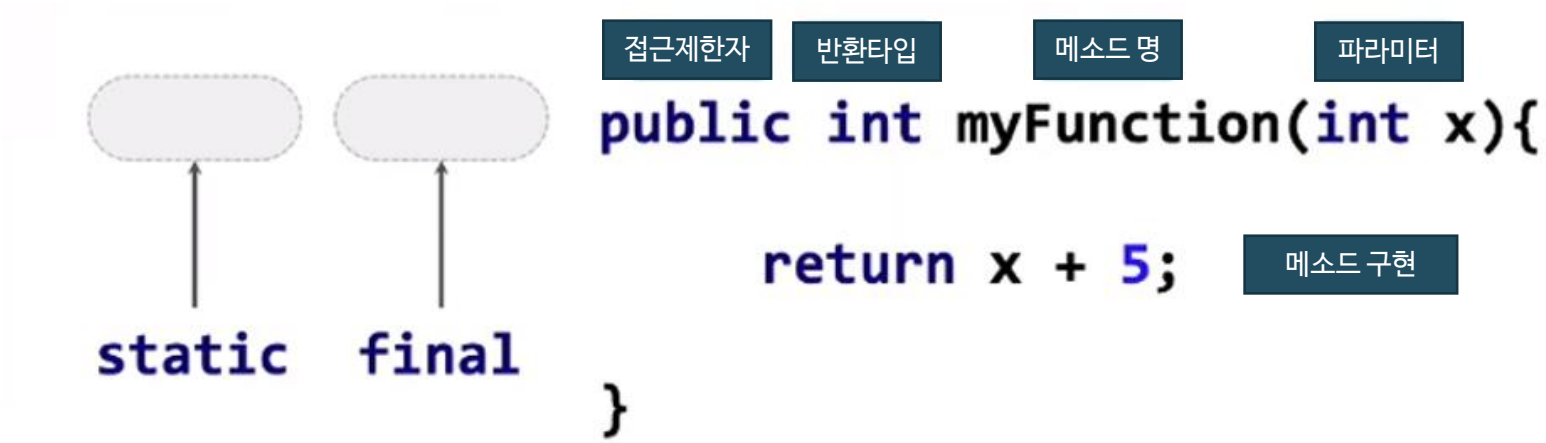
함수 & 메소드

- 함수 : 하나의 기능을 수행하는 일련의 코드이며, 호출하여 사용 및 함수가 실행된 후 값을 반환할 수 있다.
  함수가 선언되면 여러 곳에서 호출되어 사용될 수 있다.

- 함수와 메소드의 차이 :
  - ✓ 특정 클래스내부에서 구현되는 함수를 메소드라 하고, 그렇지 않으면 함수라고 한다
  - ✓ 함수는 호출이 되면 Stack 메모리에 저장되며 함수의 동작이 끝나면 메모리에서 사라진다.

7.3. 메소드[2/7] - 메소드의 정의

- 클래스에서 메소드를 정의하는 방법은 일반 함수를 정의하는 방법과 크게 다르지 않다



- 접근 제한자 : 해당 메소드에 접근할 수 있는 범위
- 반환 타입(return type) : 메소드가 모든 작업을 마치고 반환하는 데이터의 타입 / 리턴값이 없으면 void로 표현한다
- 메소드 명 : 메소드를 호출하기 위한 이름
- 매개변수 목록(parameters) : 메소드 호출 시에 전달되는 자료형과 매개변수들
- 구현부 : 메소드의 고유 기능을 수행하는 명령문들

7.3. 메소드[3/7] - 메소드 선언

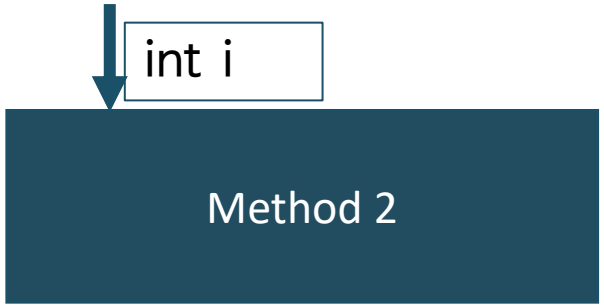
- method는 입력되는 값의 유무, 리턴되는 값의 유무에 따라 다르게 선언한다
  - ✓ 매개변수(parameter) : 전달된 인자를 받아들이는 변수
  - ✓ 전달인자(argument) : 메소드 또는 함수 호출시 전달되는 값 /리턴값

메소드의 분류

- 입력과 출력이 모두 있는 메소드
- 입력과 출력이 모두 없는 메소드
- 입력은 없고 출력은 있는 메소드
- 입력은 있고 출력은 없는 메소드

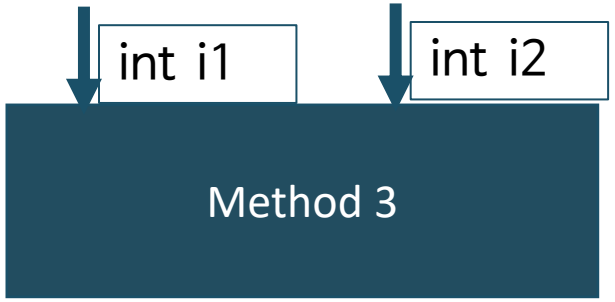
- 입력되는 매개변수도 없고 돌려줄 리턴 값도 없는(void) 메소드

```
public void method1 () {
        ..................실행문;
}
```



- 매개변수는 1개 있고 리턴 값은 없는(void) 메소드

```
public void method2(int i) {
..................실행문;
}
```

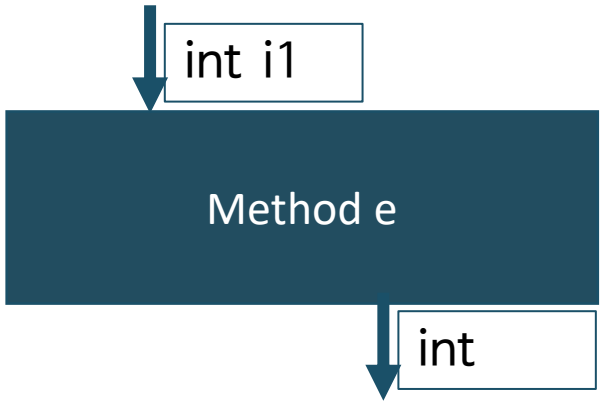


- 매개변수는 2개 있고 리턴 값은 없는(void) 메소드

```
public void method3(int i1, int i2) {
.....실행문;
}
```

- 매개변수는 없고 리턴 값이 (int)인 메소드

```
public void method4(int i1, int i2) {
.....return할 부분을실행;
        return  34;   // 정수값
}
```



- 매개변수는 정수1개, 리턴 값(int)인 메소드

```
public void method5(int i1) {
.....return할 부분을실행;
        return  55;   // 정수값
}
```

7.3. 메소드[4/7] - 메소드 호출

- 메소드는 멤버 참조 연산자(.)를 사용하여 호출할 수 있다

```
객체참조변수이름.메소드이름();                          // 매개변수가 없는 메소드의 호출
객체참조변수이름.메소드이름(파라미터1, 파라미터2, ...);    // 매개변수가 있는 메소드의 호출
```

```
public class Ex01 {
        int sum(int a, int b) {                          // a, b 는 매개변수(파라미터)
                return a+b;
        }

public static void main(String[] args) {
        Ex01 ex01 = new Ex01();
        int num = ex01.sum(2, 5);                        // 2, 5는 인수

        System.out.println(num);                         // 7 출력
        }
}
```

7.3. 메소드[5/7] - 메소드의 Overloading

- 한 클래스 내에 이름이 같지만 매개변수의 타입이나 개수가 서로 다른 여러 개의 메소드를 중복 작성하는 것을 메소드 오버로딩이라고 한다.
- 메소드 오버로딩은 객체 지향 프로그래밍의 특징 중 하나인 다형성(polymorphism)을 구현하는 방법 중 하나이다.

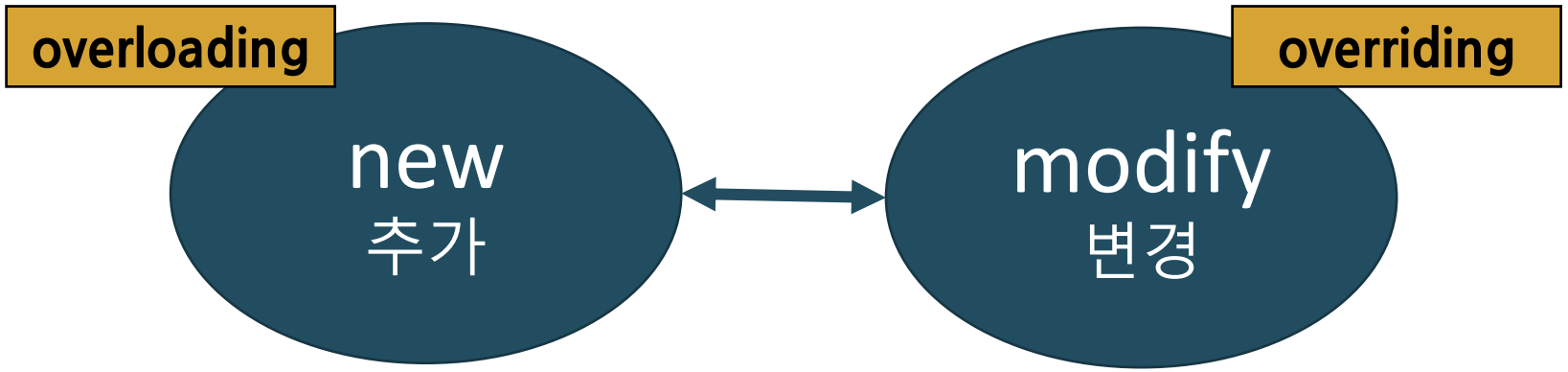
- 메소드 오버로딩은 서로 다른 시그니처를 갖는 여러 메소드를 같은 이름으로 정의하는 것이다.
- 메소드의 반환타입은 메소드 시그니처에 포함되지 않으므로 오버로딩과는 관련이 없다

- **메소드시그니처**

- 메소드의 선언부에 명시되는 매개변수의 리스트
- 두 메소드가 매개변수의 개수와 타입, 그 순서까지 모두 같다면, 이 두 메소드의 시그니처는 같다고 할 수 있다

7.3. 메소드[6/7] - 메소드 오버로딩과 오버라이딩

- Method overloading
  - ✓ 이름은 동일하지만 파라미터가 다르도록 메소드를 재정의하는 것이다
- Method overriding
  - ✓ 상위 클래스의 메소드를 하위 클래스가 재정의 하는 것이다
  - ✓ 자식 클래스가 부모 클래스에 존재하는 메소드와 동일한 파라미터를 갖는 메소드를 재정의하여 부모 클래스의 메소드를 감추는 현상이다.
  - ✓ 메소드의 이름은 물론 매개변수의 갯수나 타입도 동일해야 하며, 주로 상위 클래스의 동작을 상속받은 하위 클래스에서 변경하기 위해 사용된다.
  - ✓ 접근제한자를 좁은 범위로 변경할 수 없다(ex : public → private)



7.3. 메소드[7/7] - return문의 다른 활용

- return은 메소드를 호출한 곳에 리턴값을 돌려주는 일도 하지만 메소드를 즉시 빠져나가게도 한다
  - ✓ 리턴 자료형이 void인 메소드만 가능하다

```
public class Sample {
    void sayName(String name) {
            if ( " 홍길동".equals(name)) {
                return;
            }
            System.out.println("나의 이름은 "+ name +" 입니다.");

}
    public static void main(String[] args) {
             Sample sample = new Sample();
            sample.sayName( " 박찬호");
            sample.sayName( " 홍길동");                    // 출력되지 않는다.
    }
}
```

7.4. 생성자(constructor) [1/4] 와 인스턴스(Instance) 생성

- 생성자는 객체가 생성될 때 객체의 초기화를 위해 실행되는 메소드이다
- 모든 클래스에는 반드시 하나 이상의 생성자가 있어야 하며, 생성자명은 클래스명과 같아야 한다
- 생성자는 new를 통해 객체를 생성할 때 한번만 호출된다.
- 생성자는 어떤 값도 리턴하지 않기 때문에 리턴 타입을 지정할 수 없다.
- 생성자에도 접근제한자를 적용할 수 있으며 일반적으로 public이 적용된다

생성자 선언 방법

```
public 클래스(매개변수) {
        ....
}
```

생성자와 메소드 차이

- 생성자는 반드시 클래스명과 동일하게 정의하여야 한다
- 생성자 앞에는 접근 제어자만 올수 있다
  - ✓ 메소드는 static과 같은 수식어를 작성할 수 있다
- 반환값이 없으므로 void나 자료형을 작성할 수 없다
  - ✓ 메소드는 void나 자료형이 있어야 한다

7.4. 생성자(constructor) [2/4] - 디폴트(Default)생성자

- 매개변수 없이 클래스가 생성될 때 자동으로 호출되는 생성자를 말한다
- default 생성자는 주로 매개변수를 필요로하지 않는 단순한 초기화 작업을 수행하는데 사용된다.
  - ✓ 클래스 멤버 변수를 기본 값으로 초기화하거나, 초기화 블록을 통해 특정 작업을 수행하는 등의 역할
- default 생성자는 클래스 내부에 명시적으로 정의하지 않았을 경우에만 자동으로 생성된다
  - ✓ 클래스에 다른 생성자가 정의되어 있는 경우에는 default 생성자가 자동으로 생성되지 않는다
- 생성자가 필드의 값을 명시하지 않으면 디폴트값으로 초기화된다
  - ✓ 숫자타입 : 0 , 논리타입 : false, 참조타입 : null

생성자를 생략하면 주석 처리된 부분이 자동으로 생성된다

```
public class MyClass {
   //public MyClass() {}                    // 컴파일러에 의해 자동으로 default 생성자가 추가됨

 public static void main(String[] args) {
            MyClass mc = new MyClass();     // 인스턴스 생성 및 생성자 호출
}
```

7.4. 생성자(constructor)[3/4]- 생성자 Overloading

- **생성자의 매개변수를 다르게 지정하여 여러개를 정의하는 것을 생성자 오버로딩**이라고 한다
- 오버로딩 성립조건
  - ✓ 메소드 이름이 같아야 한다.
  - ✓ 매개변수의 개수 or 타입이 달라야 한다.
  - ✓ 반환 타입은 영향 없다.
- 생성자의 이름은 클래스와 동일해야하기 때문에 다수의 생성자 정의는 생성자 오버로딩으로 정의한다

- 객체의 인스턴스과정에서 생성자를 호출할 때 전달인자를 다르게 하여 필요한 생성자를 호출한다

```
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");
 }
}
```

```
// 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
```

7.4. 생성자(constructor)[4/4] - 생성자 호출 this()메소드

- this()는 같은 클래스 내에서 생성자가 다른 생성자를 호출할 때 사용한다.
- 하나의 클래스에 정의된 2개 이상의 생성자 사이에는 this()생성자를 통해 호출할 수 있다
- this()생성자는 중복되는 코드를 제거하고 생성자를 재사용할 수 있게 해준다
- this()생성자의 호출은 반드시 생성자 첫 줄에서만 호출할 수 있다.

```
public class MyClass {
   private int value;

   public MyClass() {
      this(0);                  // 파라미터가 있는 생성자 호출
   }
   public MyClass(int value) {
      this.value = value;
   }
}
```

this() 사용시 주의

- this()는 반드시 생성자에서만 호출할 수 있다.
- this()는 반드시 같은 클래스 내 다른 생성자를 호출할 때 사용한다.
- this()는 반드시 생성자의 첫번째 문장이어야 한다.

7.5. this 키워드

- this 키워드는 자바에서 사용되는 특별한 키워드로서, 현재 객체를 가리키는 레퍼런스를 나타낸다
- 인스턴스화 되었을 때 자기자신의 메모리 주소를 담고있는 키워드이다.
- 주로 클래스의 멤버 변수와 메소드의 매개변수 이름이 동일할 때 혼동을 방지하고자 사용

인스턴스 변수 초기화

생성자 내에서 인스턴스 변수를 초기화할 때, 생성자의 매개변수와 인스턴스 변수의 이름이 같은 경우 this 키워드를 사용하여 인스턴스 변수를 구분

```
public class MyClass {
    private int value;

    public MyClass(int value) {
        this.value = value;                    // 인스턴스 변수 value에 매개변수 value 할당
    }
}
```

다른 생성자호출

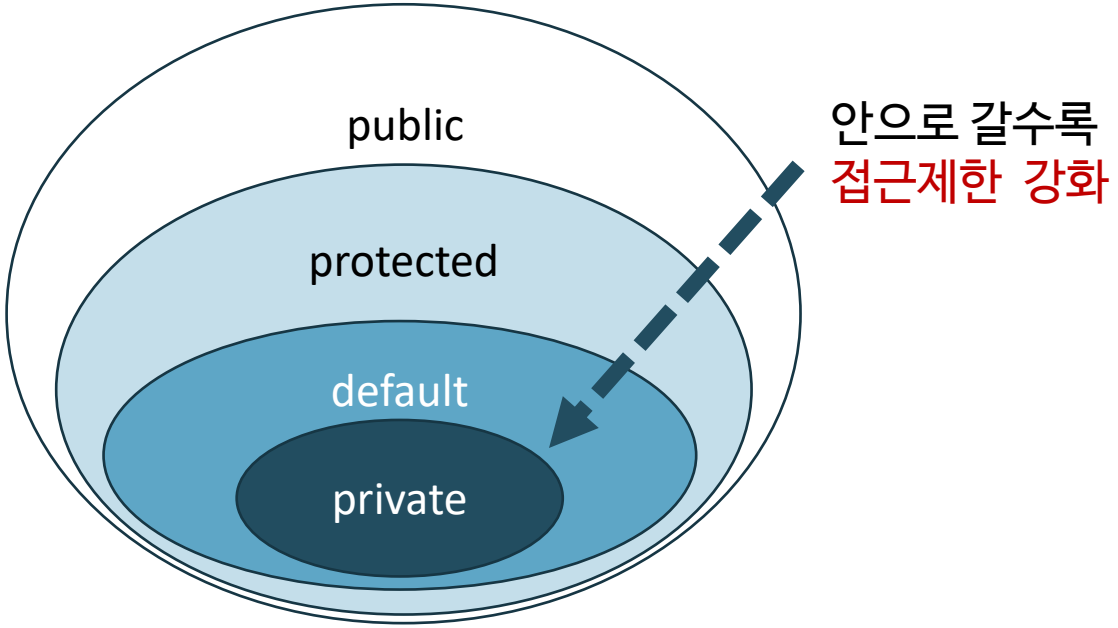
하나의 클래스 내에서 여러 개의 생성자를 오버로딩할 때, 중복되는 코드를 피하기 위해 this 키워드를 사용하여 다른 생성자를 호출

```
public class MyClass {
    private int value;

    public MyClass() {
        this(0);                               // 파라미터가 있는 생성자 호출
    }

    public MyClass(int value) {
        this.value = value;
    }
}
```

7.6. 접근제한자(Access Modifier)[1/5]

- 접근 제한자(Access Modifier)는 클래스, 필드, 생성자, 메소드등의 접근(생성 또는 호출)을 제한하기 위해 사용된다

접근 제한	적용 대상	접근할 수 없는 클래스
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	자식 클래스가 아닌 다른 패키지에 소속된 클래스
default	클래스, 필드, 생성자, 메소드	다른 패키지에 소속된 클래스
private	필드, 생성자, 메소드	모든 외부 클래스



7.6. 접근제한자(Access Modifier)[2/5] - 클래스의 접근제한

- 클래스 선언시 해당 클래스를 다른 패키지에서도 사용할 수 있도록 할 것인지 여부를 선언
  - ✓ 클래스는 public, default 접근 제한을 가진다

- Default 접근제한

  - ✓ 클래스를 선언할 때 public을 생략→ default 접근 제한
  - ✓ 클래스가 default 접근 제한을 가지면 같은 패키지에서는 사용할 수 있지만 다른 패키지에서는 사용할 수 없다

- public 접근제한

  - ✓ 클래스를 선언할 때 public을 붙이는 경우
  - ✓ 클래스가 public 접근 제한을 가지면 같은 패키지에서뿐 아니라 다른 패키지에서는 사용할 수 있다

7.6. 접근제한자(Access Modifier)[3/5] - 생성자의 접근제한

- 객체를 생성하기 위해서 new 연산자로 생성자를 호출하지만, 생성자가 어떤 접근 제한을 갖느냐에 따라 호출 가능 여부가 결정된다

- **public 접근제한**
  - ✓ 모든 패키지에서 아무런 제한 없이 생성자를 호출할 수 있다.

- **protected 접근제한**
  - ✓ 같은 패키지에 속하는 클래스에서 생성자를 호출할 수 있다
  - ✓ 해당 클래스의 자식(child) 클래스애서 생성자를 호출할 수 있다

- **default 접근제한**
  - ✓ 같은 패키지에 속하는 클래스에서만 생성자를 호출할 수 있다

- **private 접근제한**
  - ✓ 같은 패키지에 속하는 클래스에서도 생성자를 호출할 수 없다
  - ✓ 오직 해당 클래스 내부에서만 생성자를 호출할 수 있다

7.6. 접근제한자(Access Modifier)[4/5] - 필드, 메소드의 접근제한

- 필드와 메소드를 선언할 때 해당 필드와 메소드는 public, protected, default, private 접근 제한을 가질 수 있다

- **public 접근제한**
  - ✓ 모든 패키지에서 아무런 제한 없이 필드와 메소드를 사용할 수 있다

- **protected 접근제한**
  - ✓ 같은 패키지에 속하는 클래스에서 필드와 메소드를 사용할 수 있다
  - ✓ 해당 클래스의 자식(child) 클래스애서 필드와 메소드를 사용할 수 있다

- **default 접근제한**
  - ✓ 같은 패키지에 속하는 클래스에서만 필드와 메소드를 사용할 수 있다

- **private 접근제한**
  - ✓ 같은 패키지에 속하는 클래스에서도 필드와 메소드를 사용할 수 있다
  - ✓ 오직 해당 클래스 내부에서만 필드와 메소드를 사용할 수 있다

7.6. 접근제한자(Access Modifier)[5/5] - getter & setter

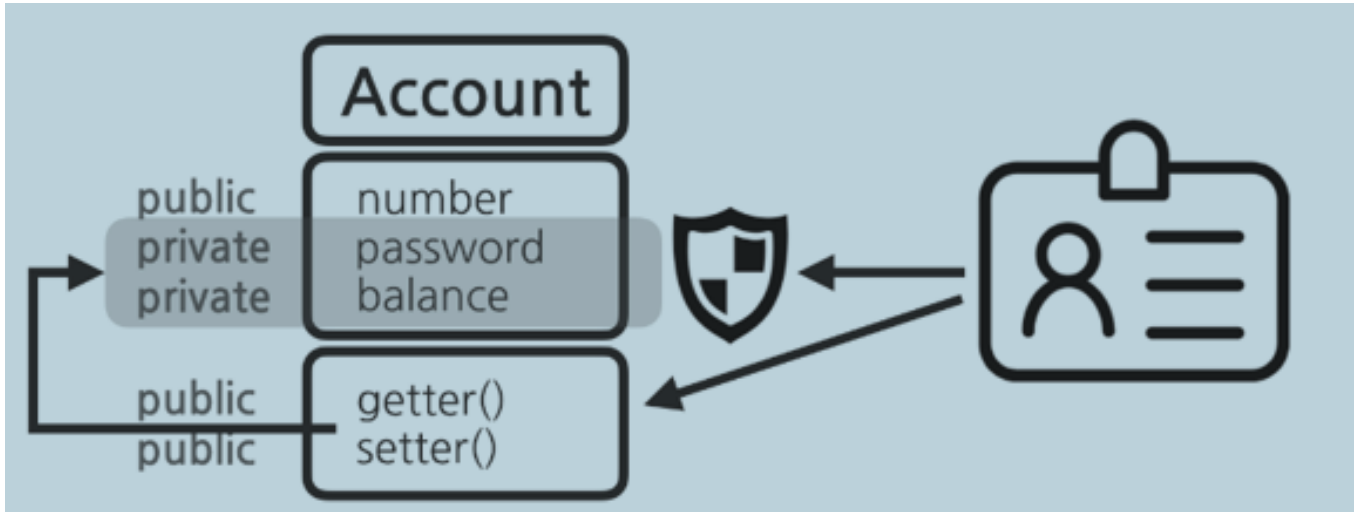
- private으로 선언된 필드는 외부 접근이 불가능하다.
- getter 와 setter를 사용하면, private 필드를 우회하여 가져오거나 변경할 수 있다.

게터 메소드
(getter methods)

=> private 필드를 반환(get)

세터 메소드
(setter methods)

=> private 필드를 변경(set)



7.7. final 키워드

- final 키워드는 변수(variable), 메소드(method), 또는 클래스(class)에 사용될 수 있다
- final 키워드는 어떤 곳에 사용되냐에 따라 다른 의미를 가지고 있지만, final 키워드를 붙이면 무언가를 제한한다는 의미를 가지는 것은 공통적이다

• 변수

✓ **변수에 final을 붙이면 이 변수는 수정할 수 없다.** 수정될 수 없기 때문에 초기화 값은 필수적이다
✓ 기본형 변수라면 값을 변경하지 못하고 참조형 변수라면 가리키는 객체를 변경하지 못하는 것이다

• 메소드

✓ **메소드에 final을 붙이면 상속 받은 클래스에서 해당 메소드를 수정해서 사용하지 못하도록 override를 제한한다**

• 클래스

✓ final 키워드를 클래스에 붙이면 상속 불가능 클래스가 된다. 즉, 다른 클래스에서 상속하여 재정의를 할 수 없는 것이다
대표적인 클래스로 Integer와 같은 랩퍼(Wrapper) 클래스가 있다.
✓ 클래스 설계시 추후 유지보수차원에서 재정의 불가능하게 사용하고 싶다면 final로 등록하면 될 것이다

7.8. static 키워드

- static 키워드는 변수에 사용하면 해당 변수를 클래스 변수로 만들어 준다
- static 키워드를 메소드에 사용하면 해당 메소드를 클래스 메소드로 만들어 준다
- static 제어자는 초기화 블록에도 사용할 수 있다
- 자바에서 static 제어자를 사용할 수 있는 대상은 3가지이다
  - ✓ 메소드, 필드, 초기화 블록

```
public class MyClass {
    static int staticVariable = 10;

    static void staticMethod() {
        System.out.println("This is a static method.");
    }
}
```

staticVariable 클래스의 모든 인스턴스에서 공유되며, staticMethod는 객체 생성 없이 MyClass.staticMethod() 형식으로 호출할 수 있다

7.9. 패키지[1/2]

- 자바에서 패키지(Package)는 관련된 클래스와 인터페이스를 그룹화하고, 이름 충돌을 방지하며, 코드의 구조를 조직화하는 데 사용되는 기능
- **클래스를 유일하게 만들어주는 식별자**
  - ✓클래스 이름이 동일해도 패키지가 다르면 다른 클래스로 인식한다. 또한 패키지 내부에 패키지를 둘 수도 있다.
- 패키지는 프로그램을 모듈화하여 코드의 유지보수와 개발을 더 효율적으로 관리할 수 있도록 해준다.

패키지명 사용규칙

- 1. 숫자로 시작하거나, '_' 과 '$'를 제외한 특수 문자를 사용 금지
- 2. java로 시작하는 패키지 금지(자바 표준 API에서만 사용)
- 3. int, static 등 자바 예약어 금지
- 4. 모두 소문자로 작성하는 것이 관례

7.9. 패키지[2/2] - 다른패키지의 클래스를 사용하는 방법

1. 패키지를 포함한 클래스명을 사용

```
java.util.Random ran = new java.util.Random();
```

2. import문 사용

```
import java.util.Random;
.............
Random ran = new Random();
```

- **import 문**

- import문은 소스파일의 최상단에 위치하며 다른 패키지에 있는 클래스나 인터페이스를 현재의 소스 코드로 가져오는 역할을 한다

8

상속과 다형성

1. 상속의 개념과 필요성
2. 메소드 재정의와 다형성
3. 추상 클래스와 추상 메소드
4. 인터페이스의 개념과 활용
5. 다형성과 인터페이스 활용

상속

확장성 + 코드 재사용

8.1. 상속의 개념[1/2]

- 상속이란 기존 클래스의 변수와 메소드를 물려 받아 새로운 클래스를 구성하는 것을 의미
- 이러한 상속은 캡슐화, 추상화, 다형성과 더불어 객체지향프로그래밍을 구성하는 특징 중 하나입니다.상속을 구현하기 위해 extends 키워드를 사용하며, 부모 클래스의 멤버들을 자식 클래스에서 사용할 수 있다
  - ✓ 상속받고자 하는 자식 클래스명 옆에 extends 키워드를 붙이고, 상속할 부모 클래스명을 명시한다

- 자식 클래스에서 필요한 경우 메소드를 오버라이딩하여 자식 클래스의 독특한 동작(기능)을 구현할 수 있다.

- 자바에서 계층구조의 최상위에 java.lang.Object 클래스가 있다.
  - ✓ 모든 클래스는 Object클래스를 자동으로 상속받도록 되어있다.
  - ✓ toString(), equals()와 같은 메소드를 바로 사용할 수 있다.

상속의 장점

- 기존 클래스의 변수와 코드를 재사용할 수 있어 개발 시간이 단축된다.
- 먼저 작성된 검증된 프로그램을 재사용하기 때문에 신뢰성 있는 프로그램을 개발할 수 있다
- 클래스 간 계층적 분류 및 관리가 가능하여 유지보수가 용이하다

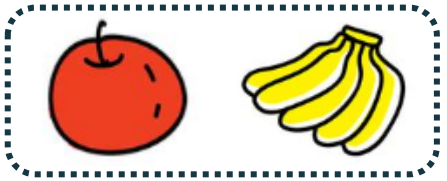
상속의 특이점

- 단일 상속만을 지원하므로 하나의 클래스는 하나의 클래스만 상속할 수 있다
  - ✓ extends 다음에는 클래스 이름 하나만 지정할 수 있다
  - ✓ 상속의 횟수에 제한을 두지 않는다.
- 접근 제한자에 따라 접근이 제한될 수 있다
  - ✓ 부모클래스의 private접근 제한을 갖는 필드 및 메소드는 자식이 물려받을 수 없다

8.1. 상속의 개념 [2/2] - 클래스 상속 관계

- 클래스 상속을 위해서는 **extends**라는 키워드를 사용한다
  - ✓ 자바는 다중 상속을 허용하지 않으므로, extends 뒤에는 하나의 부모 클래스만 와야 한다

```
public Class Fruit{ .... };
public Class Apple extends Fruit { .... };
```



과일이 아닌것은?

- 자식 클래스(sub class)가 변경되는 건 부모 클래스(super class)에게 영향을 주지 못한다.
- 생성자와 초기화 블럭은 상속되지 않는다. (멤버(필드, 메소드)만 상속된다)

```
class Fruit {
string name;
}

class Child extends Fruit {
void taste() {
System.out.println("아삭아삭 사과");
                }
}
```

Class	Member
Friut	name
Apple	name, taste()

Apple(자식)클래스는
부모인 Fruit클래스가 가지고 있는
name변수를 가지고 있으며,
Apple클래스에서 따로 추가한
taste()메소드만 작성하면 된다

8.2. super, super()

- super는 자바에서 사용되는 예약어로, 자식 클래스가 부모 클래스의 멤버에 접근할 때 사용
- super 키워드를 사용하면 부모 클래스의 생성자, 필드, 메소드 등에 접근할 수 있다
- 부모 클래스의 멤버와 자식클래스의 멤버 이름이 같을 경우 super를 통해 부모 클래스의 멤버에 접근 할 수 있다.
- 인스턴스 메소드에만 사용 가능하며 클래스 메소드에는 사용할 수 없다.

자식 클래스가 부모 클래스의 멤버(필드) 접근

```
class Animal {
    String sound = "Animal sound";

    void makeSound() {
        System.out.println(sound);
    }
}

class Dog extends Animal {
    String sound = "Woof!";

    void makeSound() {
        System.out.println(sound);          // 자식 클래스의 sound 출력
        System.out.println(super.sound);    // 부모 클래스의 sound 출력
    }
```

- super()는 부모 클래스의 생성자를 호출하거나 부모 클래스의 생성자에 전달할 인수를 지정하는 데 사용된다
- 부모 클래스의 멤버를 초기화하기 위해서 부모클래스의 생성자를 자식클래스의 생성자 첫줄에서 호출해야 한다.
- 자바 컴파일러는 부모 클래스의 생성자를 명시적으로 호출하지 않은 자식 클래스의 생성자 첫 줄에 자동으로 super()를 추가하여 부모클래스의 멤버를 초기화하도록 한다.
  단 컴파일 시 클래스에 생성자가 하나도 정의되어 있지 않아야 자동으로 기본 생성자를 추가해준다.

부모 클래스의 생성자에 접근

```
class Parent {
   int value;

   Parent(int value) {
      this.value = value;
   }
}
class Child extends Parent {
   Child(int value) {
      super(value);                    // 부모 클래스 생성자 호출
   }
}
```

8.3. 메소드 재정의(overriding) [1/7] - 개요

- 메소드 재정의(Overriding): 부모 클래스의 메소드를 자식 클래스에서 동일한 이름과 시그니처로 재정의하는 것.
- 서브 클래스가 슈퍼 클래스의 특정 메소드를 상속받아 동작을 변경하거나 확장할 수 있게 함.
- 다형성(Polymorphism)의 핵심 개념이다

메소드 재정의 필요성과 장점

- 부모 클래스에서 이미 구현된 메소드를 자식 클래스에서 필요에 따라 재정의하여 커스터마이징 가능
- 클래스 간의 강한 결합도를 줄이고, 유연하고 확장 가능한 코드 작성 가능
- 코드 중복 최소화: 공통 로직을 부모 클래스에서 구현하고 자식 클래스에서만 필요한 부분을 재정의.
- 유연한 확장성: 새로운 자식 클래스를 추가하거나 기존 클래스를 수정하지 않고도 동작 확장 가능.
- 다형성 지원: 부모 클래스 타입으로 자식 클래스 객체를 다루게 함으로서 객체지향프로그래밍의 핵심개념인 다형성을 지원함

8.3. 메소드 재정의(overriding) [2/7] - @override 어노테이션

- @Override 어노테이션은 컴파일러에게 메소드 재정의 의도를 알려주는 역할을 한다
- 이 어노테이션을 사용하면 컴파일러는 해당 메소드가 부모 클래스의 메소드를 재정의하려는 것임을 확인하고 규칙을 체크
- 재정의 규칙에 어긋나면 컴파일 오류가 발생한다

```
class Animal {
    public void makeSound() {
        System.out.println("동물이 소리를 내고 있습니다.");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("개가 짖고 있습니다.");
    }
}

public class OverrideExample {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.makeSound();
    }
}
```

- Dog 클래스는 Animal 클래스의 makeSound 메소드를 재정의하고 있다
- @Override 어노테이션을 사용, 컴파일러에게 재정의 의도를 알린다

- 어노테이션을 사용하지 않으면 메소드 이름, 매개변수의 오타를 놓칠 수 있음
- @Override 어노테이션을 사용하여 오류를 방지할 수 있음

개발시 실수를 줄이고
코드의 안정성을 높일 수 있다

8.3. 메소드 재정의(overriding) [3/7] - 규칙

- 메소드 재정의 시 규칙을 준수하여야 함.
- 접근 제어자, 반환형, 메소드 이름, 매개변수가 일치해야 함.
- 재정의된 메소드는 부모 클래스 메소드의 동작을 대체하거나 확장.

접근 제어자

- 재정의된 메소드의 접근 제어자는 부모 클래스 메소드의 접근 제어자보다 더 강하거나 같아야 함.
- 부모 클래스 메소드가 public이면 재정의된 메소드도 public이나 protected로 변경 가능.

```
class Animal {
    protected void makeSound() {
        System.out.println("동물이 소리를 내고 있습니다.");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("개가 짖고 있습니다.");
    }
}
```

반환(return)형

- 재정의된 메소드의 반환형은 부모 클래스 메소드의 반환형과 일치해야 함.
- 반환형이 서로 다르면 컴파일 오류 발생

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
}

class ScientificCalculator extends Calculator {
    // 오류: 반환형이 다르므로 컴파일 오류 발생
    @Override
    double add(int a, int b) {
        return a + b;
    }
}
```

메소드 명과 매개변수

- 재정의된 메소드의 이름과 매개변수는 부모 클래스 메소드와 동일해야 함.
- 메소드 시그니처(signature)가 일치하지 않으면 오류 발생.

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

class Circle extends Shape {
    // 오류: 메소드 이름이 다르므로 컴파일 오류 발생
    @Override
    void drawCircle() {
        System.out.println("원을 그립니다.");
    }
}
```

8.3. 메소드 재정의에 의한 다형성 구현[4/7]

- 하나의 객체가 여러 자료형 타입을 가질수 있는 것을 다형성(polymorphism)이라고 한다.
- 부모클래스의 참조변수로 자식클래스 타입의 인스턴스를 참조할 수 있다.
- 오버라이딩은 상속을 통해 같은 이름의 메소드를 서로 다른 내용으로 구현한다는 점에서 객체 지향의 다형성을 실현한다고 볼 수 있다.

참조변수의 다형성

- 자바에서는 부모클래스의 참조변수로 자식클래스 타입의 인스턴스를 참조할수 있다
  - ✓ 참조 변수가 사용할 수 있는 멤버의 개수가 실제 인스턴스의 멤버 개수보다 같거나 적어야 가능하다.
- 자식클래스의 참조변수로 부모클래스의 인스턴스를 참조할 수 없다.

참조변수의 타입변환

- 서로 상속 관계에 있는 클래스 사이에서만 타입변환을 할 수 있다.
- 자식클래스에서 부모클래스로 타입변환(업캐스팅)은 생략가능하다.
- 부모클래스에서 자식클래스로 타입변환(다운캐스팅)은 반드시 명시해야한다.

8.3. 메소드 재정의 [5/7] - 업캐스팅

- 업캐스팅은 자식 클래스의 인스턴스를 부모 클래스 타입으로 변환하는 것
- 부모 클래스 변수를 사용하여 자식 클래스 객체를 다룰 수 있다

```
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();    // 업캐스팅
        animal.makeSound();           // "Dog barks" 출력
    }
}
```

- 상속관계에서 객체간 타입의 변환은 객체의 관리에 있어 큰 장점으로 작용한다
- 자식클래스가 부모 클래스 타입의 변수로 참조 가능하므로 같은 부모클래스를 둔 모든 자식 클래스들을 하나의 타입으로 관리할 수 있다

8.3. 메소드 재정의 [6/7] - 다운캐스팅

- 다운캐스팅은 업캐스팅된 객체를 다시 원래 자식 클래스 타입으로 변환하는 것이다.
- 다운캐스팅은 런타임에 검사를 수행하여 올바른 타입 변환인지 확인해야한다

```
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();    // 업캐스팅

        // 다운캐스팅 시 ClassCastException 주의
        Dog dog = (Dog) animal;       // 다운캐스팅
        dog.makeSound();              // "Dog barks" 출력
    }
}
```

올바른 타입변환인지 확인
다운캐스팅 시
ClassCastException 주의

8.3. 메소드 재정의 [7/7] - instanceof 연산자

- instanceof 연산자는 객체가 특정 클래스나 인터페이스의 인스턴스인지 확인하는 데 사용
- instanceof 연산자 를 사용하여 객체의 타입을 검사하고, 해당 타입으로 안전하게 다운캐스팅할 수 있는지 판단할 수 있다

```
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    void makeSound() {
        System.out.println("Cat meows");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();                  // 업캐스팅

        if (animal instanceof Dog) {
            Dog dog = (Dog) animal;                 // 안전한 다운캐스팅
            dog.makeSound();                        // "Dog barks" 출력
        } else if (animal instanceof Cat) {
            Cat cat = (Cat) animal;                 // 안전한 다운캐스팅
            cat.makeSound();                        // "Cat meows" 출력
        } else {
            System.out.println("Unknown animal");
        }
    }
}
```

8.4. 추상 클래스 [1/2] - 추상화란?

- 추상(사전) : 사물이나 표상을 어떤 성질, 공통성, 본질에 착안하여 그것을 추출하여 파악하는 것
- 추상화는 **객체의 공통적인 속성(필드)과 기능(메소드)을 추출하여 정의하는것**

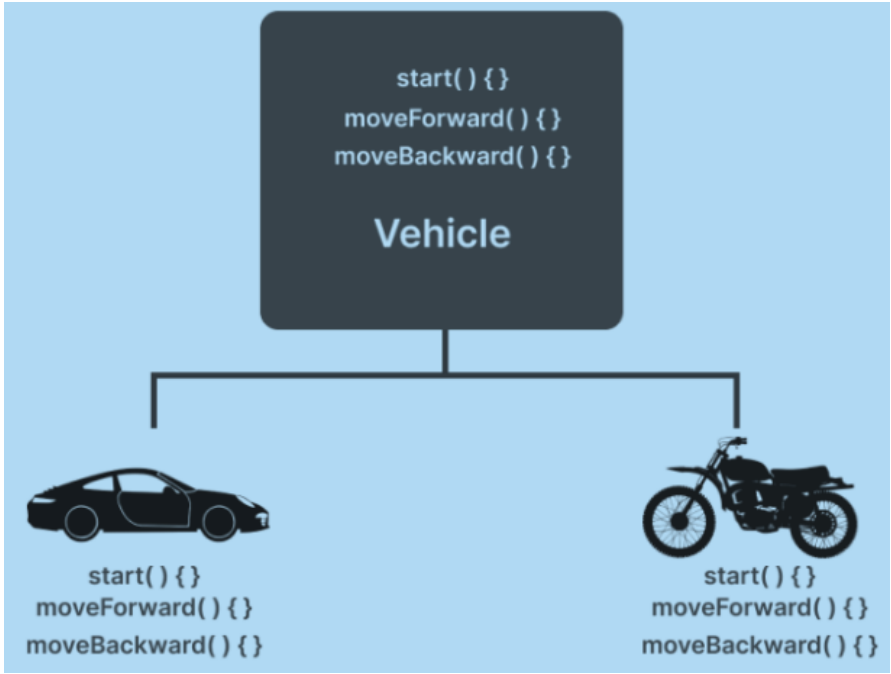
- 자동차와 오토바이는 모두 이동 수단이며 모든 Vehicle 은 전진과 후진을 할 수 있다는 공통점

⬇

- 하위 클래스들의 공통적인 기능 추출
  - 전진과 후진

⬇

- Vehicle 이라는 상위 클래스에 정의



8.4. 추상 클래스[2/2] - 추상클래스와 추상메소드

- 추상메소드를 가지고 있으면 반드시 추상 클래스로 선언해야 한다.
- 추상클래스 선언은 abstract 키워드로 한다.
- 추상 클래스는 객체를 생성할 수 없다
  - ✓ 추상메소드를 가진 추상클래스는 동작이 정의되어 있지 않으므로 인스턴스를 생성할 수 없다.
  - ✓ 추상클래스는 먼저 상속을 통해 자식클래스를 만들고 자식클래스에서 추상메소드를 오버라이딩해야 인스턴스를 생성할 수 있다.
  - ✓ 추상클래스의 객체를 생성하는 코드는 컴파일 오류가 발생한다.

추상클래스의 상속

- 추상클래스를 상속받는 자식클래스는 추상클래스가 된다.
- 추상클래스의 추상메소드를 그대로 상속받기 때문에 오버라이딩을 하지 않으면 자식클래스에 abstract를 붙여 추상클래스임을 명시해야 컴파일 오류가 발생하지 않는다.
- 추상메소드가 포함된 추상클래스를 상속받은 자식클래스는 추상메소드를 오버라이딩 해야만 인스턴스를 생성할 수 있다.

추상 메소드

- 추상메소드란 선언은 되어 있으나 코드가 구현되어 있지 않은 메소드이다.
- 추상메소드는 abstract키워드를 선언하는 선언부만이 존재하고, 구현부는 작성하지 않고 사용한다.
- 작성되어 있지 않은 구현부는 자식클래스에서 오버라이딩하여 사용한다.

```
public abstract String getSubject();
public abstract void setSubject(String s);
public abstract fail() {
        return "Hi";                                    // 컴파일오류
}
```

추상클래스의 구현 목적

- 추상메소드가 포함된 클래스를 **상속받는 자식클래스가 반드시 오버라이딩하여 구현하기 위함**이다.
- 추상클래스는 추상메소드를 통해 자식클래스가 구현할 메소드를 명료하게 알려주는 **인터페이스 역할**을 한다.
- 자식클래스는 추상메소드를 목적에 맞게 구현하는 **다형성을 실현**한다.

8.4. 추상메소드(abstract method) 상속시 오버라이딩 구현

```
abstract class Animal {                    // 추상클래스 선언
        public void Sound() {}

        public void action() {
        eat();
        }

        abstract public void eat(); // 추상 메소드 선언
}
```

```
class dog extends Animal {

        @Override
        public void eat() {
                System.out.println("강아지가 밥먹는다");

        }
}


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

                dog a = new dog();      //인스턴스 생성 가능
                a.eat();                // 강아지가 밥먹는다

        }

}
```

자식클래스에서 추상메소드를 오버라이딩 → 인스턴스 생성 가능

8.5. 인터페이스 개요

- 인터페이스는 추상 메소드만을 가질 수 있고 구현메소드는 가질 수 없으며, interface키워드로 정의한다
- 클래스가 인터페이스를 구현하기 위해서는 implements키워드를 사용해야한다
- 상속과 달리 인터페이스는 하나의 자식클래스가 2개이상의 인터페이스를 동시에 구현할 수 있어 다중 상속이 가능하다는 특징을 가진다

```
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~~");
    }
}
```

8.5. 인터페이스 특징

- 인터페이스에 정의하는 모든 필드는 public static final(사용자정의 상수)가 자동 적용된다
- 인터페이스에는 필드, 추상메소드, static method, default method를 정의할 수 있다
- 밀접한 관계를 갖는 두개 이상의 클래스들 사이에 인터페이스를 구성하면 특정 클래스의 변경에도 영향을 받지 않게 느슨한 관계를 구성할 수 있다
- 추상클래스와의 차이점
  - ✓ 추상클래스는 서로 비슷한 기능을 하는 클래스들을 묶을 때 사용
  - ✓ 인터페이스는 서로 다른 기능을 하는 클래스들을 묶을 때 사용

1. 인터페이스 정의

```
interface Vehicle {
    int maxSpeed = 100;    // 인터페이스 필드 (public static final 생략 가능)


    void start();
    void stop();
}
```

2. 인터페이스 구현

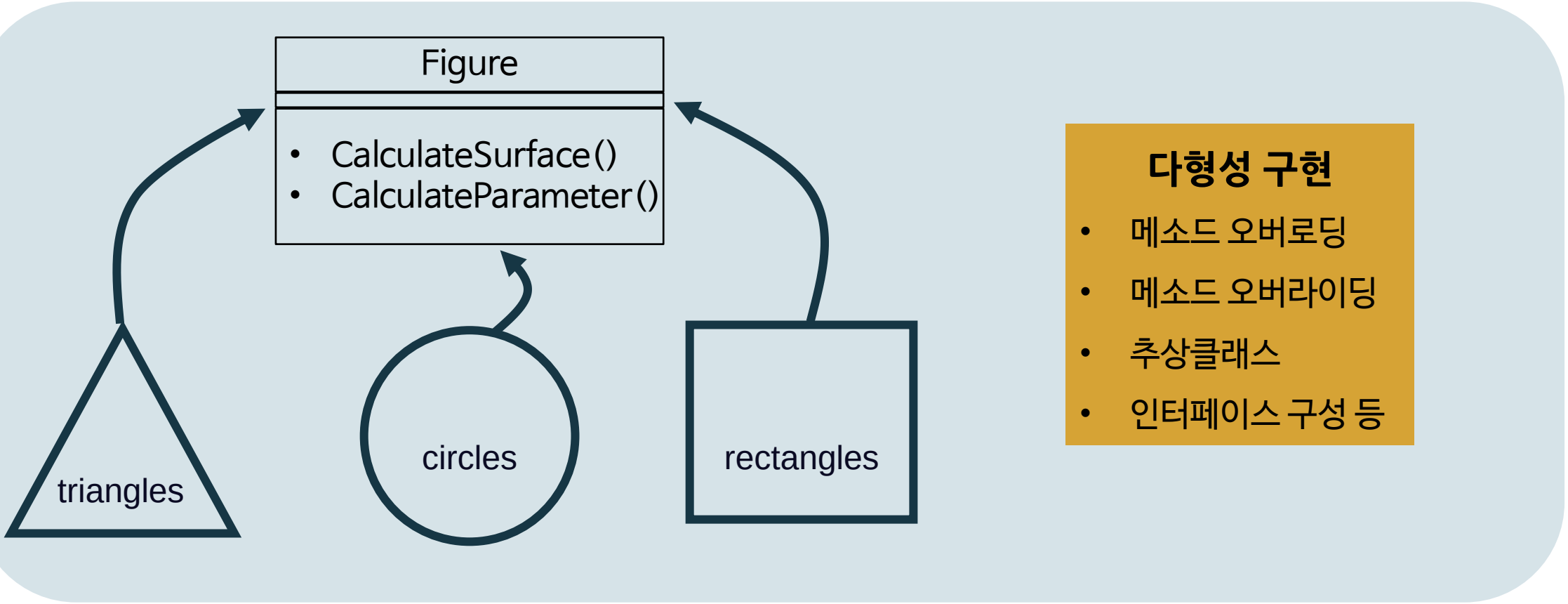
```
class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car is starting.");
    }

    @Override
    public void stop() {
        System.out.println("Car is stopping.");
    }
}
```

3. main class

```
public class InterfaceWithFieldExample {
    public static void main(String[] args) {
        Car car = new Car();
         car.start();
        car.stop();
        System.out.println("Max speed of car: " +
Vehicle.maxSpeed);
}
```

8.6. 다형성의 활용

- 다형성은 하나의 객체가 다양한 형태로 처리될 수 있는 특성을 말한다
- 여러개(poly) + 형태(morphism) → polymorphism
- **상위 클래스 타입의 참조변수로 하위/자손 클래스의 인스턴스를 참조**할 수 있도록 하는 것
- 다형성을 구현하기 위한 조건으로는 상속, 메소드 재정의와 객체간의 형변환이 있다



다형성 구현

- 메소드 오버로딩
- 메소드 오버라이딩
- 추상클래스
- 인터페이스 구성 등

9

캡슐화

1. 캡슐화와 정보 은닉
2. 접근자(getters)와 설정자(setters) 메소드
3. final 클래스 생성

9.1. 캡슐화와 정보은닉 - 접근제어자 활용

- 연관된 목적을 가지는 변수와 함수를 하나의 클래스로 묶어 외부에서 쉽게 접근하지 못하도록 은닉하는 것이다. 캡슐화는 중요한 데이터를 쉽게 바꾸지 못하도록 할 때 사용한다.
- 접근 제어자(private, protected, public)를 사용하여 캡슐화를 구현하며, 이를 통해 데이터 은닉과 코드 재사용성을 증가시킬 수 있다

```
public class Person {
    private String name;
    private int age;

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        if (name != null && !name.isEmpty()) {
            this.name = name;
        }
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if (age >= 0 && age <= 150) {
            this.age = age;
        }
    }
}
```

- name과 age → private 멤버 변수
  - ✓ 외부에서 직접 접근할 수 없다
  - ✓ name, age 변경 : setName과 setAge 메소드
  - ✓ name, age 조회 : getName과 getAge 메소드

9.2. getter & setter 메소드

- private 멤버 변수에 접근하기 위해 public 메소드를 사용하는 방식
- Getter 메소드는 private 변수의 값을 반환하고, Setter 메소드는 private 변수의 값을 설정(변경)한다

```
public class MyClass {
    private int privateVariable;

    public int getPrivateVariable() {
        return privateVariable;
    }

    public void setPrivateVariable(int newValue) {
        privateVariable = newValue;
    }
}
```

외부에서 private 변수에 접근할 때
Getter와 Setter 메소드를 통해
간접적으로 접근

9.3. 캡슐화와 정보은닉 - final 클래스 생성

- 클래스를 불변하게 만들어 내부 상태가 변경되지 않도록 설계하는 방법도 정보 은닉화의 한 형태이다
- final 클래스는 내부 데이터를 수정할 수 없도록 하여 안정성을 보장한다

```
public final class ImmutableClass {
    private final int value;

    public ImmutableClass(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

value 변수는 생성된 이후
변경되지 않는다

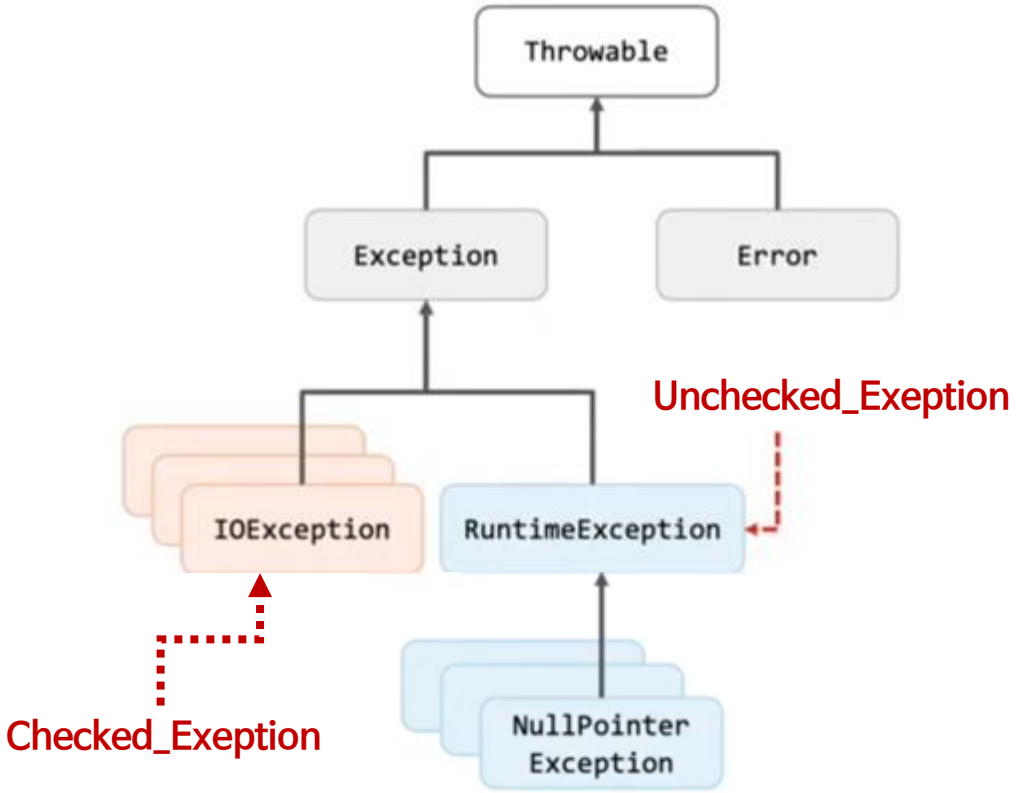
10

예외처리

1. 예외처리 개요와 종류
2. 예외처리 방법
3. 사용자 정의 예외 클래스

10.1. 예외처리 개요와 종류

- 예외처리란 프로그램 실행중에 발생하는 예외적인 상황이 발생할 때 프로그램이 비정상적으로 종료 됨을 방지하기 위해 처리하는 것이다
- 자바에서는 (tyr ~ catch)를 예외처리를 위한 도구로 제공한다



- 예외 클래스의 종류

- Throwable : 예외클래스의 최상위클래스
  1. Exception 클래스
  2. Error 클래스

- Error class

- 매우 심각한 오류상황을 나타냄
- 자바프로그램 밖에서 발생한 오류
- OutOfMemoryError 등

Checked 예외 (Checked Exception)

- 컴파일러가 체크하는 예외로, 반드시 예외 처리 코드가 작성되어야 한다
- 주로 파일 입출력, 네트워크 통신 등 외부 리소스와 관련된 상황에서 발생
- 이러한 예외는 try-catch 블록을 사용하여 처리할 수 있다.

UnChecked 예외 (또는 RuntimeException)

- 컴파일러가 체크하지 않는 예외로, 예외 처리 코드가 필요하지 않다
- 주로 프로그래머의 실수나 프로그램 논리상의 오류로 인해 발생
- 배열 인덱스 범위 초과, 널 포인터 참조 등
- 이러한 예외는 필요에 따라 처리하거나, 발생한 예외를 상위 호출자에게 전달할 수 있다

10.2. 예외처리 방법[1/2]

- Java에서 발생시키는 예외클래스 들
  - ArrayIndexOutOfBoundsException(배열접근시 잘못된 인덱스 값 사용)
  - ClassCastException : 잘못된 형변환 연산 진행
  - NullPointerException : 참조변수가 null인 상황에서 메소드 호출

메소드 내부에 try~catch 선언

```
public void MethodA() {
  try {
      ……로직
  } catch (Exception e) {
  }
}
```

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

Calculator.java

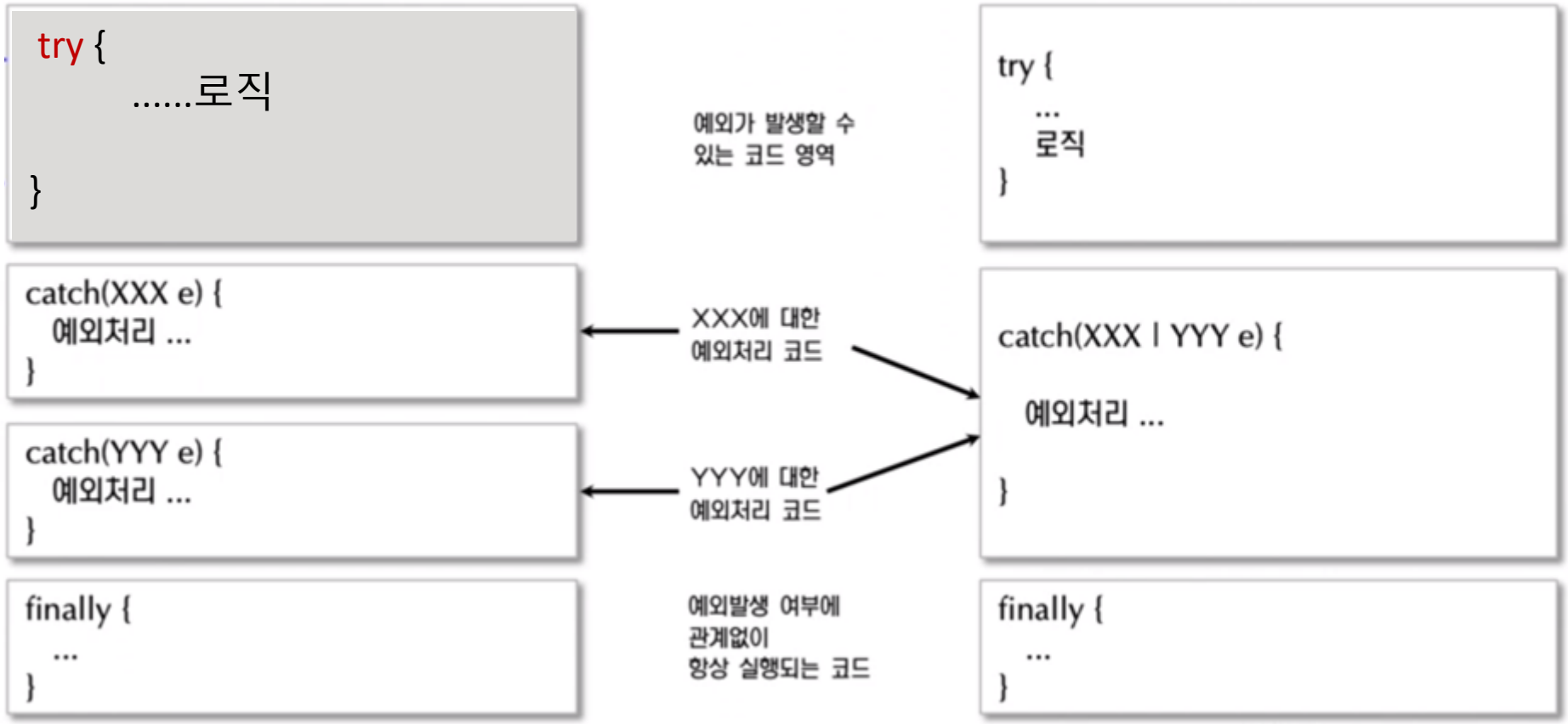
Java는 0으로 나눌 때 ArithmeticException 예외를 발생시킵니다.

다음은, 제수(num2)가 0인 경우 실행결과입니다.

실행결과

```
나눗셈 연산오류
/ by zero
```

10.2. 예외처리 방법 [2/2] - 다중 try~catch



10.3. 사용자 정의 예외 클래스

- 문법적으로 문제가 되는 상황에는 JVM이 적절한 예외를 발생시키지만 논리적인 문제인 경우 개발자가 직접 상황에 맞는 예외클래스를 작성해서 프로그램이 비정상적으로 종료되는 것을 방지한다

- 예외 클래스 작성

  - Exception 클래스 상속
  - 예외클래스 생성자에 메시지 추가

```
class NegativeNumberException extends Exception {
    public NegativeNumberException(String message) {
        super(message);
    }
}
```

- throw 키워드

  - 예외 발생을 JVM에 알림
  - 발생된 예외상황을 메소드 호출한곳으로 전달함
  - 문법 : throw 예외클래스 인스턴스

```
public class UserDefinedExceptionExample {
   public static void main(String[] args) {
      try {
         int number = getPositiveNumber();
         System.out.println("입력된 양수: " + number);
      } catch (NegativeNumberException e) {
         System.out.println("예외가 발생했습니다: "
                                    + e.getMessage());
      }
   }
```

```
public static int getPositiveNumber() throws
NegativeNumberException {
      java.util.Scanner scanner = new
java.util.Scanner(System.in);
      System.out.print("양수를 입력하세요: ");
      int num = scanner.nextInt();

      if (num < 0) {
         try {
            throw new NegativeNumberException("음수는
입력할 수 없습니다.");
         } catch (NegativeNumberException e) {
            scanner.close();
            throw e;
         }
      }

      scanner.close();
      return num;
   }
```

11

자바의 입출력

1. 자바 입출력 개요
2. 파일입출력
3. 파일 직렬화와 역직렬화
4. Stream API 개요
5. Stream API 중간연산과 최종연산

11.1. 자바 입출력 개요[1/5] -

- 입출력(Input/Output)은 컴퓨터 프로그램이 데이터를 주고받는 과정을 나타낸다
- 입출력은 프로그램이 실제로 활용되는 환경과 상호작용하며 데이터를 교환하는 중요한 부분이다

- **입출력의 중요성**

- **사용자와 상호작용**
  - ✓ 프로그램이 사용자와 효율적으로 상호작용하려면 입력을 받아야 하고 키보드나 마우스를 통해 사용자가 입력한 명령이나 데이터를 처리하여 원하는 결과를 출력하게 된다.
- **데이터 처리**
  - ✓ 프로그램은 입력된 데이터를 처리하고 결과를 출력하는 역할을 하는데 DB에서 데이터를 가져와 분석하거나, 파일에서 정보를 읽어와 가공하는 등의 작업을 수행할 때 입출력이 필요하다
- **외부 시스템과 연동**
  - ✓ 프로그램이 네트워크를 통해 원격 시스템과 데이터를 주고받거나, 파일을 읽고 쓰면서 다른 애플리케이션과 연동하는 경우에도 입출력이 필요
- **사용자 피드백**
  - ✓ 입출력을 통해 사용자에게 필요한 정보나 결과를 제공하므로, 사용자는 프로그램이 동작하는 상태를 파악하고 이해할 수 있다.
- **데이터 저장 및 유지**
  - ✓ 프로그램이 처리한 결과를 파일이나 데이터베이스에 저장하고, 조회할 경우 입출력이 필수이다

11.1. 자바 입출력 개요[2/5] - InputStream 클래스

- InputStream은 자바에서 입력 스트림을 다루기 위한 추상 클래스이다.
- InputStream 클래스는 바이트 단위로 데이터를 읽어오는 기능을 제공하며, 파일, 네트워크 연결, 시스템 입력 등 다양한 소스로부터 데이터를 읽어올 수 있다

- **InputStream의 주요 메소드**

- int read(): 입력 스트림으로부터 바이트를 읽고, 읽은 바이트를 반환 더 이상 읽을 데이터가 없으면 -1을 반환
- int read(byte[] b): 입력 스트림으로부터 바이트 배열 b에 최대한의 데이터를 읽어옴 읽은 바이트 수를 반환
- int read(byte[] b, int off, int len): 입력 스트림으로부터 바이트 배열 b의 지정된 위치 off부터 최대 길이 len까지의 데이터를 읽어옴
- void close(): 입력 스트림을 닫아 관련 리소스를 해제.

11.1. 자바 입출력 개요[3/5] - OutputStream 클래스

- OutputStream은 자바에서 출력 스트림을 다루기 위한 추상 클래스이다.
- OutputStream 클래스는 바이트 단위로 데이터를 출력하는 기능을 제공하며, 파일, 네트워크 연결, 시스템 출력 등 다양한 대상에 데이터를 쓸 수 있다

- **InputStream의 주요 메소드**

- void write(int b): 출력 스트림에 바이트 b를 출력한다
- void write(byte[] b): 출력 스트림에 바이트 배열 b의 모든 데이터를 출력한다
- void write(byte[] b, int off, int len): 출력 스트림에 바이트 배열 b의 지정된 위치 off부터 최대 길이 len까지의 데이터를 출력한다

- void flush(): 버퍼에 잔류하는 데이터를 강제로 출력한다
- void close(): 출력 스트림을 닫아 관련 리소스를 해제한다

11.1. 자바 입출력 개요[4/5] - 표준 입출력 (Standard I/O)

• System.out과 System.in

- System.out: 표준 출력 스트림으로, 화면에 데이터를 출력하는 데 사용
  - ✓ 대부분의 경우 텍스트 데이터를 출력할 때 주로 사용
- System.in: 표준 입력 스트림으로, 사용자로부터 데이터를 입력받는 데 사용
  - ✓ 주로 키보드 입력을 읽어올 때 사용

• PrintStream을 이용한 표준 출력

- PrintStream 클래스는 자바의 출력 스트림 중 하나로, System.out과 같이 텍스트 데이터를 화면에 출력하는 데 사용

• Scanner를 이용한 표준 입력

- Scanner 클래스는 키보드 입력과 같이 표준 입력 스트림인 System.in을 편리하게 다루기 위해 사용
  Scanner를 이용하면 다양한 타입의 입력을 처리할 수 있다

11.1. 자바 입출력 개요[5/5] - 표준 입출력 (Standard I/O)

- **Scanner 클래스**
  - ✓ Scanner 클래스는 키보드로부터 데이터를 읽어오는데 사용되며, java.util 패키지에 포함되어 있다
  - ✓ Scanner를 이용하면 키보드나 파일 등 다양한 소스에서 데이터를 읽어와서 자바 프로그램에서 처리할 수 있다. 주로 사용자의 입력을 받거나 파일의 내용을 읽는 데 활용한다

- **Scanner 클래스의 주요 기능**

- 생성자로 입력 소스 지정
  - ✓ Scanner 객체를 생성할 때, 읽어올 데이터의 소스를 지정한다 주로 System.in을 사용하여 표준 입력(사용자의 키보드 입력)을 읽어올 수 있다. 또한 파일이나 문자열 등을 입력 소스로 지정할 수도 있다.
- 다양한 메소드로 데이터 읽기
  - ✓ Scanner 클래스는 다양한 메소드를 제공하여 입력 데이터를 읽을 수 있다.
  - ✓ 가장 일반적인 메소드로는 next(), nextInt(), nextLine() 등이 있다 각각 단어, 정수, 한 줄의 문자열을 읽어온다.
- 타입별 데이터 변환
- Scanner는 입력된 문자열 데이터를 다양한 데이터 타입으로 변환하여 사용할 수 있는 메소드를 제공한다
  - ✓ ex) nextInt() 메소드는 읽은 문자열을 정수로 변환하여 반환한다
- 4. 자원 관리
  - ✓ Scanner 객체를 사용한 후에는 .close() 메소드를 호출하여 자원을 정리해야 한다 더 나은 방법으로는 try-with-resources 문을 사용하여 자동으로 자원을 닫을 수 있다

11.2. 파일 입출력[1/3] - 개념과 종류

- 파일 입출력은 컴퓨터 시스템의 파일 시스템과 상호 작용하여 파일을 읽고 쓰는 작업을 의미한다
- 파일 입출력은 데이터를 파일에 저장하거나 파일에서 데이터를 읽어오는 과정을 포함하며, 자바에서는 InputStream과 OutputStream 클래스를 사용하여 파일 입출력을 처리한다
- 파일입출력은 바이트기반 입출력과 **문자 기반 파일 입출력으로 나눌수 있다**

• 바이트 기반 파일 입출력 (Byte Streams)

- 바이트 기반 파일 입출력은 파일을 바이트 단위로 읽고 쓰는 작업을 다룬다
- 바이트 기반 파일 입출력은 텍스트 파일 뿐만 아니라 모든 종류의 파일을 다룰 수 있다.
- FileInputStream
  - ✓ FileInputStream 클래스는 파일로부터 바이트 단위로 데이터를 읽어오는 데 사용한다
  - ✓ FileOutputStream 클래스는 파일에 바이트 단위로 데이터를 쓰는 데 활용한다

• 문자 기반 파일 입출력 (Character Streams)

- 문자 기반 파일 입출력은 파일을 문자 단위로 읽고 쓰는 작업을 다루며 주로 텍스트 파일의 내용을 읽고 쓸 때 사용된다
- **FileReader**
  - ✓ FileReader 클래스는 파일로부터 문자 단위로 데이터를 읽어오는 데 사용
- **FileWriter**
  - ✓ FileWriter 클래스는 파일에 문자 단위로 데이터를 쓰는 데 사용

11.2. 파일 입출력[2/3] - 파일읽기

```
import java.io.FileInputStream;
import java.io.IOException;
public class FileReadingExample {
   public static void main(String[] args) {
      try {
         FileInputStream inputStream = new FileInputStream("input.txt");
          int data;
         while ((data = inputStream.read()) != -1) {
            System.out.print((char) data);
         }
        inputStream.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

input.txt 파일의 내용을 읽어서 콘솔에 출력

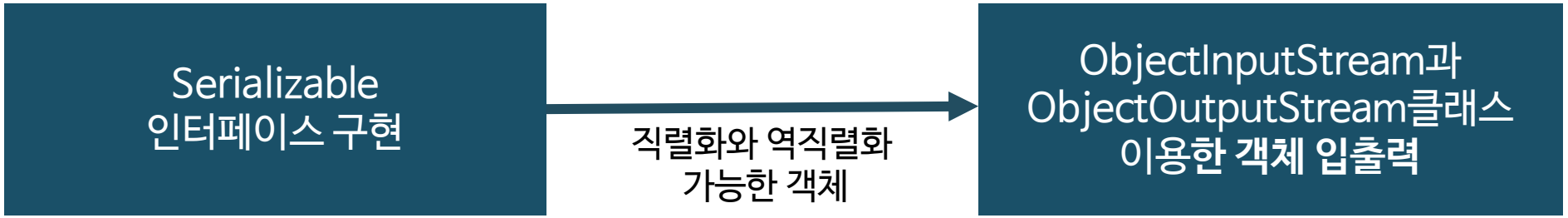
11.2. 파일 입출력[3/3] - 파일쓰기

```
import java.io.FileOutputStream;
import java.io.IOException;
public class FileWritingExample {
    public static void main(String[] args) {
        String content = "Hello, World!";
        try {
            FileOutputStream outputStream = new FileOutputStream("output.txt");
            byte[] data = content.getBytes();
            outputStream.write(data);
            outputStream.close();
            System.out.println("Data has been written to the file.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

문자열을 output.txt파일에 입력

11.3. 객체 직렬화와 역직렬화[1/4]

- 객체 직렬화 (Object Serialization):
  - ✓ 자바 객체를 바이트 스트림으로 변환하는 과정을 의미. 객체를 파일에 저장하거나 네트워크를 통해 전송할 때, 객체의 상태와 데이터를 바이트로 변환하여 저장하고 나중에 객체로 복원할 수 있게 해준다.
- 객체 역직렬화 (Object Deserialization):
  - ✓ 바이트 스트림으로 저장된 객체를 원래의 객체로 복원하는 과정을 의미
- 객체를 직렬화하기 위해서는 해당 클래스가 Serializable 인터페이스를 구현해야 한다.
- Serializable 인터페이스를 구현하면 자동으로 직렬화와 역직렬화가 가능한 객체가 된다



- 객체를 직렬화하기 위해서는 해당 클래스가 Serializable 인터페이스를 구현해야 한다.
- Serializable 인터페이스를 구현하면 자동으로 직렬화와 역직렬화가 가능한 객체가 된다.

```
import java.io.*;
public class SerializationExample {
   public static void main(String[] args) {
      // 직렬화
      try {
         ObjectOutputStream outputStream =
                  new ObjectOutputStream(new FileOutputStream("data.ser"));
         MyClass objectToSerialize = new MyClass(42, "Hello");
         outputStream.writeObject(objectToSerialize);
         outputStream.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
      // 역직렬화
      try {
         ObjectInputStream inputStream = new ObjectInputStream(new
FileInputStream("data.ser"));
         MyClass deserializedObject = (MyClass) inputStream.readObject();
         inputStream.close();

         System.out.println("Int value: " + deserializedObject.getIntValue());
         System.out.println("String value: " +
deserializedObject.getStringValue());
      } catch (IOException | ClassNotFoundException e) {
         e.printStackTrace();
      }
```

```
class MyClass implements Serializable {
    private int intValue;
    private String stringValue;

    public MyClass(int intValue, String stringValue) {
        this.intValue = intValue;
        this.stringValue = stringValue;
    }
}
```

1. Serializable 인터페이스 구현
2. 객체를 직렬화하여 파일에 저장
3. 역직렬화를 통해 객체를 다시 복원

11.3. 객체 직렬화와 역직렬화[3/4]

```
import java.io.*;
class Student implements Serializable {
   private static final long serialVersionUID = 1L;
   private String name;
   private int age;

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

   public String getName() {
      return name;
   }

   public int getAge() {
      return age;
   }
}
```

- Student 클래스
→ Serializable 인터페이스 구현
→ 직렬화 가능

```
public class SerializationExample {
   public static void main(String[] args) {
      // 객체 생성
      Student student = new Student("Alice", 20);

      // 객체를 파일에 직렬화하여 저장
      try {
         FileOutputStream fileOut = new FileOutputStream("student.ser");
         ObjectOutputStream out = new ObjectOutputStream(fileOut);
         out.writeObject(student);
         out.close();
         fileOut.close();
         System.out.println("Student object has been serialized.");
      } catch (IOException e) {
         e.printStackTrace();
      }
```

- 객체를 직렬화 → 파일에 저장

11.3. 객체 직렬화와 역직렬화[4/4]

```
// 파일에서 객체를 역직렬화하여 읽어옴
    try {
        FileInputStream fileIn = new FileInputStream("student.ser");
        ObjectInputStream in = new ObjectInputStream(fileIn);
        Student serializedStudent = (Student) in.readObject();
        in.close();
        fileIn.close();

        System.out.println("Deserialized Student:");
        System.out.println("Name: " + serializedStudent.getName());
        System.out.println("Age: " + serializedStudent.getAge());
    } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
    }
  }
}
```

- 역직렬화 → 다시 객체를 읽어옴

11.4. Stream API의 개요와 목적[1/6]

- Stream API는 자바 8부터 도입된 기능으로, 데이터 처리를 편리하고 간결하게 수행할 수 있도록 설계된 기능
- 입력과 출력 내용에서 살펴본 스트림과는 전혀 다른 개념이다
- Stream API는 컬렉션, 배열, 파일 등의 데이터 소스를 처리하는 강력한 도구로, 데이터를 다루는 작업을 더욱 효율적으로 처리할 수 있게 제공하고 있다

- **Stream API의 장점**

- 간결한 코드 작성:
  - Stream API는 데이터 처리 과정을 더욱 간결하게 표현할 수 있어서반복문과 조건문을 줄일 수 있어 코드가 간결해진다
- 병렬 처리 지원:
  - Stream API는 데이터 처리를 병렬로 수행할 수 있는 기능을 제공해 멀티코어 CPU를 활용하여 데이터 처리 속도를 향상시킬 수 있다.
- 데이터 스트림 처리:
  - ✓ Stream API는 데이터를 스트림의 형태로 처리한다.
  - ✓ 스트림은 데이터가 흐르는 통로 역할을 하며, 데이터 변환, 필터링, 집계 등 다양한 작업을 수행할 수 있다.

11.4. Stream API 특징[2/6]

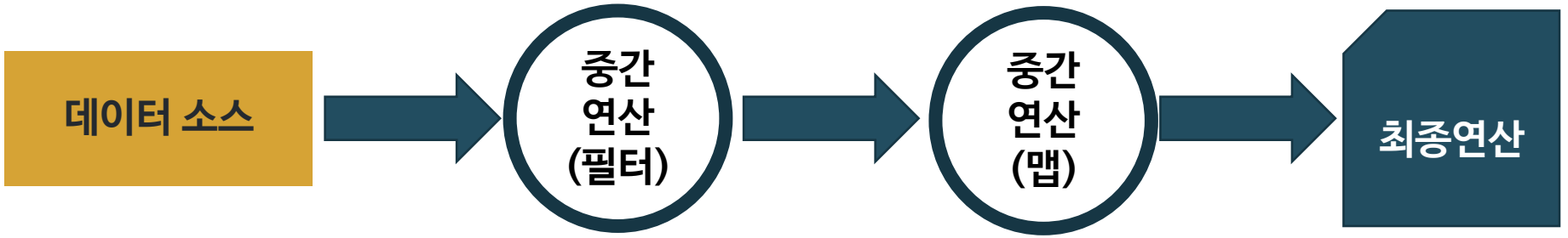
- Stream API는 컬렉션과 배열의 데이터를 처리하는데만 국한되지 않고, 파일, 네트워크 연결 등 다양한 데이터 소스를 처리하는 강력한 기능을 제공한다
- Stream API를 활용하여 데이터 처리 작업을 더욱 효율적으로 수행할 수 있으며, 현대적인 자바 프로그래밍에 필수적인 요소 중 하나다.

- **Stream API의 특징**

- **스트림은 외부 반복을 통해 작업하는 컬렉션과는 달리 내부 반복을 통해 작업을 수행한다**
- **스트림은 재사용이 가능한 컬렉션과는 달리 단 한 번만 사용할 수 있다**
- **스트림은 원본 데이터를 변경하지 않는다**
- **스트림의 연산은 필터-맵 기반의 API를 사용하여 지연(lazy) 연산을 통해 성능을 최적화한다**
- **스트림은 parallelStream() 메소드를 통한 손쉬운 병렬 처리를 지원한다**

11.4. 스트림 API의 동작 흐름 [3/6]

- 스트림 API는 다음과 같이 세 가지 단계에 걸쳐서 동작한다
  1. 스트림의 생성
  2. 스트림의 중개 연산 (스트림의 변환)
  3. 스트림의 최종 연산 (스트림의 사용)



자바 스트림 API의 동작 흐름

11.4. Stream 생성방법 [4/6]

- Stream API는 다양한 데이터 소스로부터 스트림을 생성하는 방법을 제공한다.
- 주로 컬렉션, 배열, 파일 등의 데이터 소스로부터 스트림을 생성할 수 있다

- 컬렉션으로부터 Stream 생성

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
Stream<String> stream = names.stream();
```

- 배열로부터 Stream 생성

```
int[] numbers = {1, 2, 3, 4, 5};
IntStream stream = Arrays.stream(numbers);
```

- 파일로부터 Stream 생성

```
try (Stream<String> lines = Files.lines(Paths.get("data.txt"))) {
// lines 스트림을 활용하여 파일의 각 줄을 처리
} catch (IOException e) {
   e.printStackTrace();
}
```

11.4. Stream 중간연산 [5/6]

- 스트림 API에 의해 생성된 초기 스트림은 중간 연산을 통해 또 다른 스트림으로 변환된다.
- 중간 연산은 스트림을 전달받아 스트림을 반환하므로, 중간 연산은 연속으로 연결해서 사용할 수 있다
- 스트림의 중간 연산은 필터-맵(filter-map) 기반의 API를 사용함으로 지연(lazy) 연산을 통해 성능을 최적화할 수 있다

메소드	설명
Stream<T> filter(Predicate<? super T> predicate)	해당 스트림에서 주어진 조건(predicate)에 맞는 요소만으로 구성된 새로운 스트림을 반환함.
<R> Stream<R> map(Functoin<? super T, ? extends R> mapper)	해당 스트림의 요소들을 주어진 함수에 인수로 전달하여, 그 반환값으로 이루어진 새로운 스트림을 반환함.
<R> Stream<R> flatMap(Functoin<? super T, ? extends Stream<? extends R>> mapper)	해당 스트림의 요소가 배열일 경우, 배열의 각 요소를 주어진 함수에 인수로 전달하여, 그 반환값으로 이루어진 새로운 스트림을 반환함.
Stream<T> distinct()	해당 스트림에서 중복된 요소가 제거된 새로운 스트림을 반환함. 내부적으로 Object 클래스의 equals() 메소드를 사용함.
Stream<T> limit(long maxSize)	해당 스트림에서 전달된 개수만큼의 요소만으로 이루어진 새로운 스트림을 반환함.
Stream<T> peek(Consumer<? super T> action)	결과 스트림으로부터 각 요소를 소모하여 추가로 명시된 동작(action)을 수행하여 새로운 스트림을 생성하여 반환함.
Stream<T> skip(long n)	해당 스트림의 첫 번째 요소부터 전달된 개수만큼의 요소를 제외한 나머지 요소만으로 이루어진 새로운 스트림을 반환함.
Stream<T> sorted() Stream<T> sorted(Comparator<? super T> comparator)	해당 스트림을 주어진 비교자(comparator)를 이용하여 정렬함. 비교자를 전달하지 않으면 영문사전 순(natural order)으로 정렬함.

11.4. Stream 최종 연산 [6/6]

- 스트림 API에서 중간 연산을 통해 변환된 스트림은 마지막으로 최종 연산을 통해 각 요소를 소모하여 결과를 표시한다. 지연(lazy)되었던 모든 중간 연산들이 최종 연산 시에 모두 수행되는 것이다
- 최종 연산 시에 모든 요소를 소모한 해당 스트림은 더는 사용할 수 없게 된다.

메소드	설명
void forEach(Consumer<? super T> action)	스트림의 각 요소에 대해 해당 요소를 소모하여 명시된 동작을 수행함.
Optional<T> reduce(BinaryOperator<T> accumulator) T reduce(T identity, BinaryOperator<T> accumulator)	처음 두 요소를 가지고 연산을 수행한 뒤, 그 결과와 다음 요소를 가지고 또다시 연산을 수행함. 이런 식으로 해당 스트림의 모든 요소를 소모하여 연산을 수행하고, 그 결과를 반환함.
Optional<T> findFirst() Optional<T> findAny()	해당 스트림에서 첫 번째 요소를 참조하는 Optional 객체를 반환함. (findAny() 메소드는 병렬 스트림일 때 사용함)
boolean anyMatch(Predicate<? super T> predicate)	해당 스트림의 일부 요소가 특정 조건을 만족할 경우에 true를 반환함.
boolean allMatch(Predicate<? super T> predicate)	해당 스트림의 모든 요소가 특정 조건을 만족할 경우에 true를 반환함.
boolean noneMatch(Predicate<? super T> predicate)	해당 스트림의 모든 요소가 특정 조건을 만족하지 않을 경우에 true를 반환함.
long count()	해당 스트림의 요소의 개수를 반환함.
Optional<T> max(Comparator<? super T> comparator)	해당 스트림의 요소 중에서 가장 큰 값을 가지는 요소를 참조하는 Optional 객체를 반환함.
Optional<T> min(Comparator<? super T> comparator)	해당 스트림의 요소 중에서 가장 작은 값을 가지는 요소를 참조하는 Optional 객체를 반환함.
T sum()	해당 스트림의 모든 요소에 대해 합을 구하여 반환함.
Optional<T> average()	해당 스트림의 모든 요소에 대해 평균값을 구하여 반환함.
<R,A> R collect(Collector<? super T,A,R> collector)	인수로 전달되는 Collectors 객체에 구현된 방법대로 스트림의 요소를 수집함.

12

스레드 프로그래밍

1. 스레드 개요
2. 스레드 생성 방법
3. 스레드의 상태
4. 스레드의 동기화
5. 스레드 프로그래밍시 주의점

12.1. 스레드 개요

- 프로세스란?

- 프로세스는 실행 중인 프로그램의 인스턴스를 의미한다.
- 프로세스는 운영체제에서 독립적으로 실행되는 작업 단위로, 자체의 메모리 공간, 실행 상태, 리소스 등을 가진다
- 프로세스 간에는 각각 독립적인 메모리 영역을 가지기 때문에 하나의 프로세스에서 다른 프로세스의 메모리에 직접 접근할 수 없다.

- 스레드란?

- 스레드는 프로세스 내에서 실행되는 작업의 단위이다.
- 한 프로세스 안에서 여러 개의 스레드가 동시에 실행될 수 있으며, 이들 스레드는 같은 프로세스의 자원을 공유
- 스레드는 프로세스의 메모리 공간을 공유하면서 각각 독립적인 실행 흐름을 가짐

- 프로세스와 스레드의 차이점

- 프로세스는 각자 독립적인 메모리 공간과 자원을 가지며, 스레드는 같은 프로세스의 메모리와 자원을 공유
- 프로세스를 생성할 때는 독립적인 메모리 공간을 할당, 스레드는 프로세스의 메모리를 공유하기 때문에 생성 비용이 낮다.
- 프로세스 간에는 통신을 위해 IPC(Inter-Process Communication) 메커니즘을 사용해야 하지만, 스레드는 같은 프로세스의 메모리를 공유하기 때문에 데이터 공유가 간단
- 다수의 프로세스는 여러 CPU 코어에서 병렬적으로 실행될 수 있다. 스레드도 병렬 실행이 가능하지만, 하나의 프로세스 내에서 스레드 간에는 스케줄링과 동기화 관련 이슈가 발생할 수 있다.

12.2. 스레드 생성 방법 [1/2] -Thread클래스 상속방법

- 자바에서는 Thread 클래스를 상속받거나 Runnable 인터페이스를 구현하여 스레드를 생성할 수 있다
- Thread 클래스를 상속받은 경우에는 run() 메소드를 오버라이딩하여 스레드가 실행할 작업을 정의한다
-

```
// Thread 클래스 상속
class MyThread extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread A: " + i);
        }
    }
}

public class ThreadExample1 {
    public static void main(String[] args) {
        MyThread threadA = new MyThread();
        threadA.start();          // 스레드 시작

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

12.2. 스레드 생성 방법 [2/2] - Runnable 인터페이스 구현 방법

- Runnable 인터페이스를 구현한 경우에는 run() 메소드를 구현하여 작업 내용을 정의하고, 이를 스레드로 실행하는 Thread 객체를 생성하여 실행한다

```
// Runnable 인터페이스 구현
class MyRunnable implements Runnable {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread B: " + i);
        }
    }
}

public class ThreadExample2 {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread threadB = new Thread(myRunnable);
        threadB.start(); // 스레드 시작

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

- 자바는 단일 상속만을 지원하므로 Thread 클래스를 상속하는 경우 다른 클래스를 상속할 수 없는 단점이 있기 때문에 Runnable 인터페이스를 구현하는 방법이 자바에서 더 권장되는 방법이다.

12.3. 스레드의 상태

- 자바 스레드는 다양한 상태를 가질 수 있으며, 자바 스레드는 이러한 상태들 사이를 이동하면서 작업을 수행한다
- 스레드 스케줄링에 의해 각 상태로 전환되며, 상태 전환은 스레드가 특정 이벤트에 응답하거나 특정 시간이 경과했을 때 발생한다
- 주의해야 할 점은 상태 전환과 관련된 메소드들을 올바르게 사용해야 스레드의 동작을 제어할 수 있다

- 스레드의 주요 상태

- NEW (새로운 상태): 스레드 객체가 생성되었지만 start() 메소드가 호출되지 않은 초기 상태
- RUNNABLE (실행 가능한 상태): start() 메소드가 호출되어 스레드가 실행될 준비가 된 상태
  운영체제의 스케줄러에 의해 선택되어 CPU를 할당받아 실행될 수 있다.
- BLOCKED (차단 상태): 스레드가 락(lock)을 획득하지 못해서 다른 스레드가 해당 락을 사용 중일 때 발생하는 상태. 락이 해제되면 다시 RUNNABLE 상태로 돌아간다.
- WAITING (대기 상태): 스레드가 특정 이벤트를 기다리는 상태로, 다른 스레드가 해당 스레드를 깨워줄 때까지 기다리는 상태다. 예를 들어 wait() 메소드가 호출된 경우 해당 상태로 들어간다.
- TIMED_WAITING (시간 지정 대기 상태): 특정 시간 동안 대기하는 상태로, 예를 들어 sleep() 메소드나 특정 시간을 가진 wait() 메소드가 호출된 경우에 해당한다. 주어진 시간이 지나면 다시 RUNNABLE 상태로 돌아간다.
- TERMINATED (종료 상태): 스레드의 실행이 완료되거나, run() 메소드 내에서 예외가 발생하여 스레드가 종료된 상태

12.4. 스레드 동기화와 비동기화

- 스레드 동기화는 여러 스레드가 공유된 데이터에 동시에 접근하는 것을 조절하여 데이터 불일치나 예기치 않은 동작을 방지하는 메커니즘이다
- 스레드가 동시에 공유 데이터를 읽고 쓰는 경우 문제가 발생할 수 있으므로 이를 해결하기 위해 동기화 메커니즘을 사용하여 스레드 간의 순차적 접근을 보장한다
- 자바에서는 synchronized 키워드나 Lock 인터페이스를 활용하여 동기화를 구현할 수 있다
- 동기화 블록을 사용하면 특정 영역에 대한 접근을 오직 하나의 스레드만 허용하고, 다른 스레드들은 대기해야 한다

```
public class SynchronizedExample {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public synchronized int getCount() {
        return count;
    }
}
```

스레드 동기화

여러 스레드가 공유 데이터에 접근할 때
일어날 수 있는 문제를
방지하기 위한 메커니즘

스레드 비동기화

각 스레드가 독립적으로 작업을 수행하여
상호간섭 없이 실행되는 개념

12.5. 주의할 점

- 스레드 프로그래밍은 효율적이고 성능을 높일 수 있는 방법이지만, 복잡성과 동시성 문제를 야기할 수 있으므로 주의해야한다

- **경쟁 상태 (Race Condition) 주의**

여러 스레드가 공유 데이터에 동시에 접근할 때 데이터의 일관성을 보장하지 않는 상황을 경쟁 상태라고 하는데 이를 피하기 위해 동기화 메커니즘을 사용하거나, 스레드 안전한 자료구조를 활용하여 데이터 접근을 보호해야 한다

- **데드락 (Deadlock) 피하기:**

두 개 이상의 스레드가 서로가 가진 자원을 기다리며 무한히 대기하는 상태인 데드락이 발생할 수 있다.
이를 피하기 위해 락을 순서대로 획득하거나, 타임아웃을 설정하여 자원을 반납하는 등의 방법을 사용해야 합니다

- **스레드 안전성 보장:**

여러 스레드가 동시에 메소드를 호출하거나 데이터를 조작할 때 원치 않은 결과가 발생하지 않도록 스레드 안전한 코드를 작성해야 한다
동기화 메커니즘을 적절하게 사용하거나 스레드 안전한 자료구조를 선택하는 것이 중요

- **메모리 가시성 이슈 해결:**

스레드는 각각 독립적인 캐시를 가지고 있기 때문에 다른 스레드에서 변경한 데이터가 제대로 반영되지 않을 수 있다
volatile 키워드를 사용하여 변수의 변경 사항을 스레드에게 즉시 반영하도록 할 수 있다

- **스레드 우선순위 조절 주의:**

스레드의 우선순위를 변경하는 것은 플랫폼에 따라 동작이 다르며, 과도한 우선순위 조절은 시스템의 전반적인 성능을 저하시킬 수 있다.

- **스레드 풀 사용 권장 :** 스레드 생성과 제거보다는 스레드 풀을 사용하여 스레드를 재활용하면 성능을 향상시킬 수 있다.
- **무분별한 병렬화 주의 :** 작업의 복잡성과 분할 가능성을 고려하여 적절한 수의 스레드를 사용해야 한다.

13

Java 주요 API

1. Object class
2. Wrapper클래스
3. 문자열클래스

13.1. Object 클래스

- java.lang 패키지
  - ✓ java.lang 패키지는 자바에서 가장 기본적인 동작을 수행하는 클래스들의 집합이다
  - ✓ java.lang 패키지의 클래스들은 import 문을 사용하지 않아도 클래스 이름만으로 바로 사용할 수 있다

- java.lang.Object
  - ✓ 자바의 모든 클래스의 최상위 부모 클래스
  - ✓ 모든 클래스는 암시적으로 또는 명시적으로 Object 클래스를 상속한다
  - ✓ Object 클래스는 자바에서 모든 객체가 가지는 공통적인 메소드와 기능을 제공한다
  - ✓ Object 클래스는 필드를 가지지 않으며, 총 11개의 메소드만으로 구성

13.1. Object 클래스의 메소드

메소드	설명
protected Object clone()	해당 객체의 복제본을 생성하여 반환함
boolean equals(Object obj)	해당 객체와 전달받은 객체가 같은지 여부를 반환함
protected void finalize()	해당 객체를 더는 아무도 참조하지 않아 가비지 컬렉터가 객체의 리소스를 정리하기 위해 호출함
Class<T> getClass()	해당 객체의 클래스 타입을 반환함
int hashCode()	해당 객체의 해시 코드값을 반환함
void notify()	해당 객체의 대기(wait)하고 있는 하나의 스레드를 다시 실행할 때 호출함
void notifyAll()	해당 객체의 대기(wait)하고 있는 모든 스레드를 다시 실행할 때 호출함
String toString()	해당 객체의 정보를 문자열로 반환함
void wait()	해당 객체의 다른 스레드가 notify()나 notifyAll() 메소드를 실행할 때까지 현재 스레드를 일시적으로 대기(wait)시킬 때 호출함
void wait(long timeout)	해당 객체의 다른 스레드가 notify()나 notifyAll() 메소드를 실행하거나 전달받은 시간이 지날 때까지 현재 스레드를 일시적으로 대기(wait)시킬 때 호출함
void wait(long timeout, int nanos)	해당 객체의 다른 스레드가 notify()나 notifyAll() 메소드를 실행하거나 전달받은 시간이 지나거나 다른 스레드가 현재 스레드를 인터럽트(interrupt) 할 때까지 현재 스레드를 일시적으로 대기(wait)시킬 때 호출함

13.2. Wrapper 클래스[1/2]

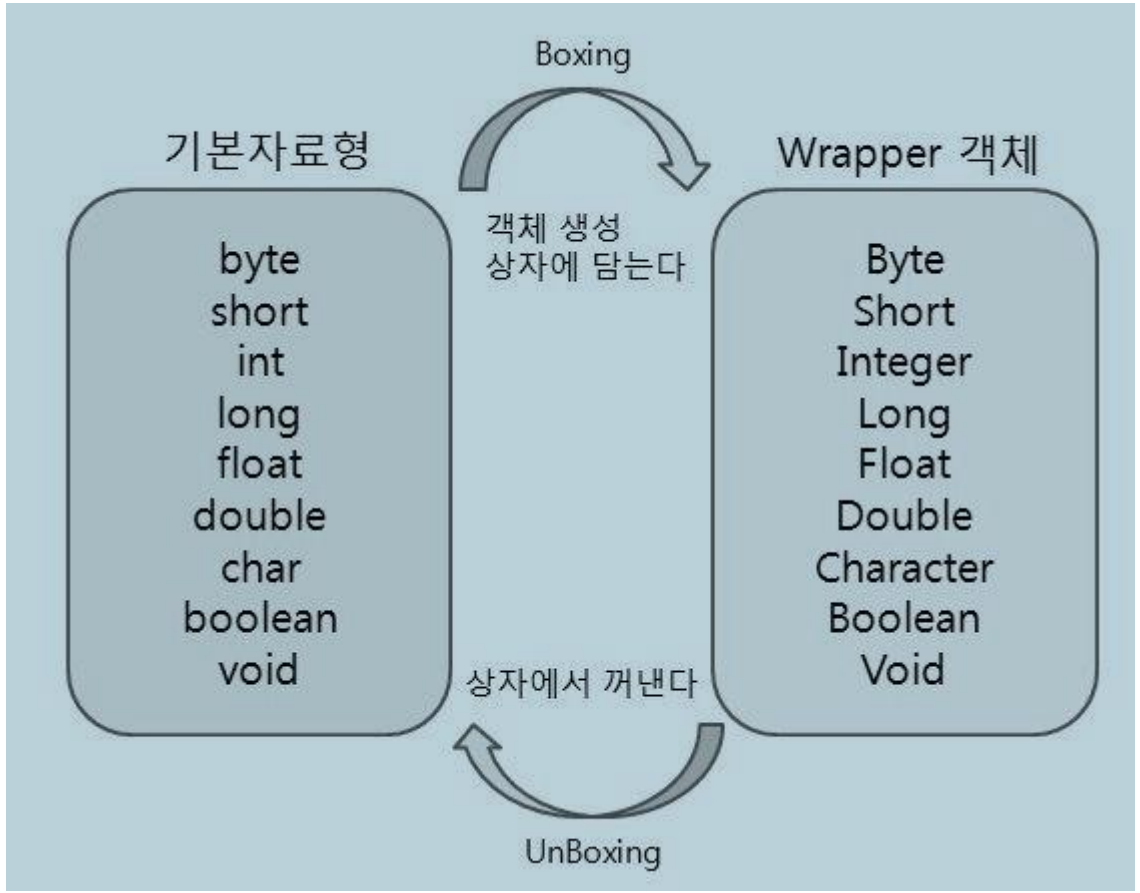
- Wrapper 클래스는 기본 데이터 타입(primitive data type)을 객체로 다룰 수 있게 해주는 클래스이다
- Wrappe 클래스들은 기본데이터타입에 대한 객체화와 함께 다양한 기능들을 제공하고 있다

Wrapper 클래스가 있는 이유와 장점

- 객체 지향 프로그래밍과 상호작용
  - 기본 데이터 타입은 값 자체만 저장하고 메소드를 가질 수 없기 때문에, 객체로 다루기 어렵다. Wrapper 클래스를 사용하면 기본 데이터 타입을 객체로 감싸서 객체처럼 다룰 수 있어 객체 지향적인 프로그래밍에 도움을 준다.
- 컬렉션과 제네릭 활용
  - 많은 자바 컬렉션 프레임워크는 객체를 다루기 때문에 기본 데이터 타입을 사용할 수 없다 Wrapper 클래스를 사용하여 기본 데이터 타입을 컬렉션에 담을 수 있습니다
- Null 값 다루기
  - 기본 데이터 타입은 null 값을 가질 수 없지만 Wrapper 클래스는 null 값을 가질 수 있어서 객체가 없는 상황을 표현할 수 있다
- 메소드 활용
  - Wrapper 클래스는 기본 데이터 타입을 감싸기 때문에 객체로써 메소드를 호출할 수 있다

13.2. Wrapper 클래스[2/2] - Boxing과 UnBoxing

- Wrapper Class는 산술연산을 위해 정의된 클래스가 아니기 때문에, 이 클래스의 객체 인스턴스에 저장된 값을 변경이 불가능하며 값을 저장하는 새로운 객체의 생성 및 참조만 가능하다




```
int pint = 42;
Integer wint = Integer.valueOf(pint);
            // int를 Integer 객체로 변환

int eint = wint.intValue();
            // Integer를 다시 int로 변환
```

13.3. String 클래스

- **java.lang.String 클래스**
- **String 클래스는 문자열을 다루기 위한 기본 클래스로, 자바에서는 문자열의 생성, 조작, 비교 등 다양한 작업을 수행할 수 있는 기능을 제공한다.**
- **String 클래스는 자바의 내장 클래스로서 매우 중요하며, 자주 사용되는 클래스 중 하나이다.**

- **String 클래스의 주요 특징**

- 불변성 (Immutable): String 객체는 생성된 후에 내용을 변경할 수 없다. 즉, 한 번 생성된 문자열은 변경되지 않는다. 이는 문자열의 안전성과 일관성을 보장하며 멀티스레드 환경에서도 안정적으로 사용될 수 있도록 합니다.

- 문자열 연결 최적화: String 객체의 불변성으로 인해 문자열 연결(Concatenation) 시 새로운 문자열 객체를 생성하는 것이 아니라 기존의 문자열 객체에 새로운 문자열을 추가하는 방식으로 최적화되어 있다. 문자열 연결이 많은 경우 StringBuilder나 StringBuffer를 사용하면 성능 향상을 가져온다

- 문자열 리터럴: "Hello World"와 같은 문자열 리터럴은 자동으로 String 객체로 생성된다. 이는 자바 컴파일러에 의해 처리되며, 리터럴이 동일한 경우 동일한 String 객체를 공유한다.

13.3. 대표적인 String메소드

메소드	설명
char charAt(int index)	해당 문자열의 특정 인덱스에 해당하는 문자를 반환함.
int compareTo(String str)	해당 문자열을 인수로 전달된 문자열과 사전 편찬 순으로 비교함.
int compareToIgnoreCase(String str)	해당 문자열을 인수로 전달된 문자열과 대소문자를 구분하지 않고 사전 편찬 순으로 비교함.
String concat(String str)	해당 문자열의 뒤에 인수로 전달된 문자열을 추가한 새로운 문자열을 반환함.
int indexOf(int ch) int indexOf(String str)	해당 문자열에서 특정 문자나 문자열이 처음으로 등장하는 위치의 인덱스를 반환함.
int indexOf(int ch, int fromIndex) int indexOf(String str, int fromIndex)	해당 문자열에서 특정 문자나 문자열이 전달된 인덱스 이후에 처음으로 등장하는 위치의 인덱스를 반환함.
int lastIndexOf(int ch)	해당 문자열에서 특정 문자가 마지막으로 등장하는 위치의 인덱스를 반환함.
int lastIndexOf(int ch, int fromIndex)	해당 문자열에서 특정 문자가 전달된 인덱스 이후에 마지막으로 등장하는 위치의 인덱스를 반환함.
String [] split(String regex)	해당 문자열을 전달된 정규 표현식(regular expression)에 따라 나눠서 반환함.
String substring(int beginIndex)	해당 문자열의 전달된 인덱스부터 끝까지를 새로운 문자열로 반환함.
String substring(int begin, int end)	해당 문자열의 전달된 시작 인덱스부터 마지막 인덱스까지를 새로운 문자열로 반환함.
String toLowerCase()	해당 문자열의 모든 문자를 소문자로 변환함.
String toUpperCase()	해당 문자열의 모든 문자를 대문자로 변환함.
String trim()	해당 문자열의 맨 앞과 맨 뒤에 포함된 모든 공백 문자를 제거함.
length()	해당 문자열의 길이를 반환함.
isEmpty()	해당 문자열의 길이가 0이면 true를 반환하고, 아니면 false를 반환함.

14

Java의 현대적 프로그래밍 패턴

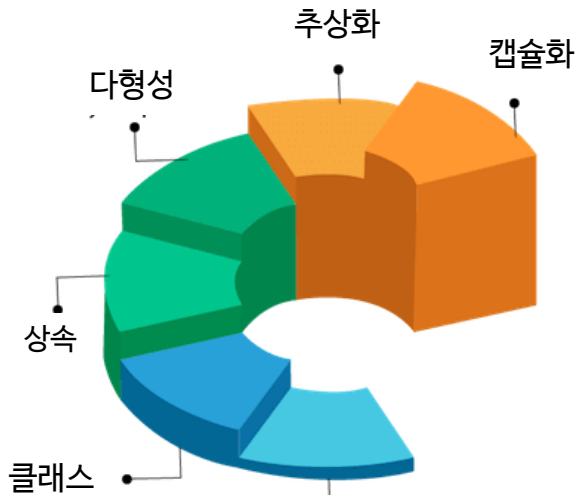
1. 객체 지향 프로그래밍(OOP)
2. MVC 패턴(Model-View-Controller
3. 싱글톤(Singleton) 패턴
4. 컬렉션프레임워크

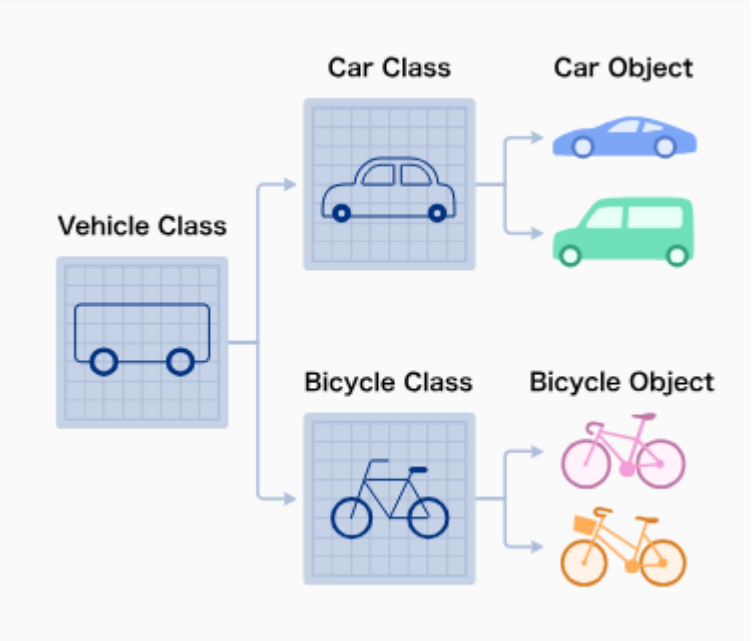
14.1. 객체지향프로그래밍

- 객체지향 프로그래밍은 코드를 보다 모듈화하고 유지보수하기 쉽게 만들어주며, 프로그램의 구조를 더 잘 이해하고 설계할 수 있게 도와준다.

- 자바에서는 클래스를 통해 객체의 특성과 동작을 정의하며, 이러한 객체들 간에는 상속과 인터페이스를 통해 관계를 형성한다.

- 다형성은 같은 이름의 메소드나 클래스가 다양한 상황에서 다르게 동작할 수 있는 기능을 의미하며,
- 추상화는 현실세계에서 존재하는 모든 사물의 공통점과 공통 기능을 유추해서 클래스를 만들어내는 일련의 과정을 의미한다





14.2. MVC패턴

- MVC 패턴은 소프트웨어 디자인 패턴 중 하나로, 애플리케이션을 세 가지 주요 구성 요소로 나누어 관리하는 아키텍처 패턴이다.
- 구성 요소는 모델(Model), 뷰(View), 컨트롤러(Controller)다.
- 각 구성 요소는 특정한 역할을 수행하며, 이를 분리함으로써 애플리케이션의 유지보수성과 확장성을 향상시키는데 목적이 있다
- **모델 (Model)**:
  - ✓ 모델은 애플리케이션의 데이터와 비즈니스 로직을 담당
  - ✓ 데이터를 유지하고 조작하는데 필요한 작업을 수행하며, 데이터의 변경을 감지하고 뷰 및 컨트롤러에 변경 사항을 알림.
  - ✓ 모델은 사용자 인터페이스나 데이터 표현과 독립적이며, 데이터의 일관성과 유효성을 유지하며 관리한다
- **뷰 (View)**:
  - ✓ 뷰는 사용자에게 데이터의 시각적 표현을 제공
  - ✓ 모델의 데이터를 이해 가능한 형태로 표현하고, 사용자가 인터랙션할 수 있는 인터페이스를 구성한다
  - ✓ 뷰는 모델의 데이터 변화를 감지하여 자동으로 업데이트되어 사용자에게 최신 정보를 제공한다
- **컨트롤러 (Controller)**:
  - ✓ 컨트롤러는 사용자의 입력을 처리하고, 해당하는 작업을 모델과 뷰에 전달한다
  - ✓ 사용자의 요청을 해석하고, 이에 따라 모델을 업데이트하거나 뷰를 선택적으로 변경.
  - ✓ 컨트롤러는 모델과 뷰 간의 상호작용을 관리하면서 뷰의 변경 사항에 따라 모델의 업데이트를 조정한다

14.2. MVC 패턴의 주요 이점

- 분리된 역할:
  - ✓ 각 구성 요소가 독립적으로 역할을 수행하므로, 애플리케이션의 다양한 측면을 더 쉽게 관리하고 수정할 수 있다
- 유연성과 확장성:
  - ✓ 각 구성 요소를 개별적으로 확장하거나 교체할 수 있으므로 애플리케이션을 더 유연하게 개발하고 변경할 수 있다
- 재사용성:
  - ✓ 뷰와 컨트롤러는 모델에 의존하지 않으므로, 다양한 뷰를 재사용하거나 동일한 모델을 다양한 방식으로 표현할 수 있다
- 테스트 용이성:
  - ✓ 모델과 뷰, 컨트롤러를 분리하여 단위 테스트와 통합 테스트를 보다 용이하게 수행할 수 있다

14.3. 싱글톤 패턴(Singleton Pattern)

- 싱글톤 패턴(Singleton Pattern)은 객체 지향 프로그래밍에서 사용되는 디자인 패턴 중 하나
- 특정 클래스의 인스턴스가 오직 하나만 생성되도록 보장하는 패턴
- 이 패턴을 사용하면 어떤 클래스의 인스턴스가 항상 하나만 존재하게 되므로, 해당 객체를 여러 곳에서 공유하거나 중복 생성하는 것을 방지할 수 있다

• 싱글톤패턴의 활용

- **공유 리소스 관리**:
  - ✓ 여러 곳에서 하나의 리소스에 접근해야 하는 상황에서 싱글톤 패턴을 사용하여 해당 리소스를 공유하고 중복 생성을 방지할 수 있다
- **데이터베이스 연결, 로깅**:
  - ✓ 데이터베이스 연결과 같은 고비용 작업이 필요한 경우, 싱글톤 패턴을 사용하여 불필요한 연결 생성을 피하고 연결을 재사용할 수 있다 또한 로깅과 같이 모든 곳에서 동일한 로그 인스턴스를 사용하는 경우에도 유용하다
- **환경 설정**:
  - ✓ 애플리케이션의 전역적인 설정 정보를 담는 싱글톤 인스턴스를 사용하여 애플리케이션 전체에서 일관된 설정을 사용할 수 있다

14.3. 싱글톤 패턴의 구현 방법_특징

- 싱글톤 패턴의 구현 방법은 다양하지만, 주로 다음과 같은 특징을 갖는다

- **프라이빗 생성자 (Private Constructor)**:
  - ✓ 클래스 내부에 생성자를 private로 선언하여 외부에서 직접 객체 생성을 막는다
- **정적 메소드로 유일한 인스턴스 생성 (Static Method for Singleton Instance)**:
  - ✓ 클래스 내부에 유일한 인스턴스를 저장하고 반환하는 정적 메소드를 구현한다
- **지연 초기화 (Lazy Initialization)**:
  - ✓ 인스턴스가 처음 사용될 때 생성하도록 구현하여 애플리케이션 시작 시 부하를 줄인다
- **스레드 안전 (Thread Safety)**:
  - ✓ 멀티스레드 환경에서도 유일한 인스턴스가 보장되도록 스레드 안전한 방식으로 구현한다
- **직렬화 (Serialization)**:
  - ✓ 싱글톤 클래스가 직렬화되고 역직렬화될 때 유일한 인스턴스를 유지하도록 메소드를 추가한다

14.3 싱글톤 패턴 예제

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        // private constructor
    }

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

- 정적 메소드 getInstance()를 통해
  - ✓ 유일한 인스턴스 반환
- 생성자 private로 선언
  - ✓ 외부에서 직접 객체 생성을 막음
- synchronized 키워드를 사용
  - ✓ 스레드를 안전하게 처리

14.6.1. 컬렉션프레임워크 개요

- 자바에서 제공하는 데이터를 저장, 관리, 조작하는 데 사용되는 인터페이스와 클래스의 집합
- 데이터 구조를 추상화하고 다양한 유형의 데이터를 효율적으로 다루기 위해 설계됨
- 이를 통해 데이터의 추가, 삭제, 검색, 정렬 등을 표준화된 방식으로 수행할 수 있다

주요 컬렉션인터페이스

- List:
  - ✓ 순서가 있는 데이터의 리스트를 표현. 중복가능
- Set:
  - ✓ 순서가 없는 고유한 데이터의 집합. 중복불가
- Queue:
  - ✓ 데이터를 큐 형태로 관리하는 인터페이스
  - ✓ FIFO(First-In-First-Out) 원칙을 따른다
- Map:
  - ✓ 키와 값으로 구성된 데이터의 매핑구조

주요 컬렉션 구현 클래스

- List:
  - ✓ ArrayList, LinkedList
- Set:
  - ✓ HashSet, TreeSet
- Map:
  - ✓ HashMap, TreeMap

14.6.2. 컬렉션 인터페이스(collection Interface)

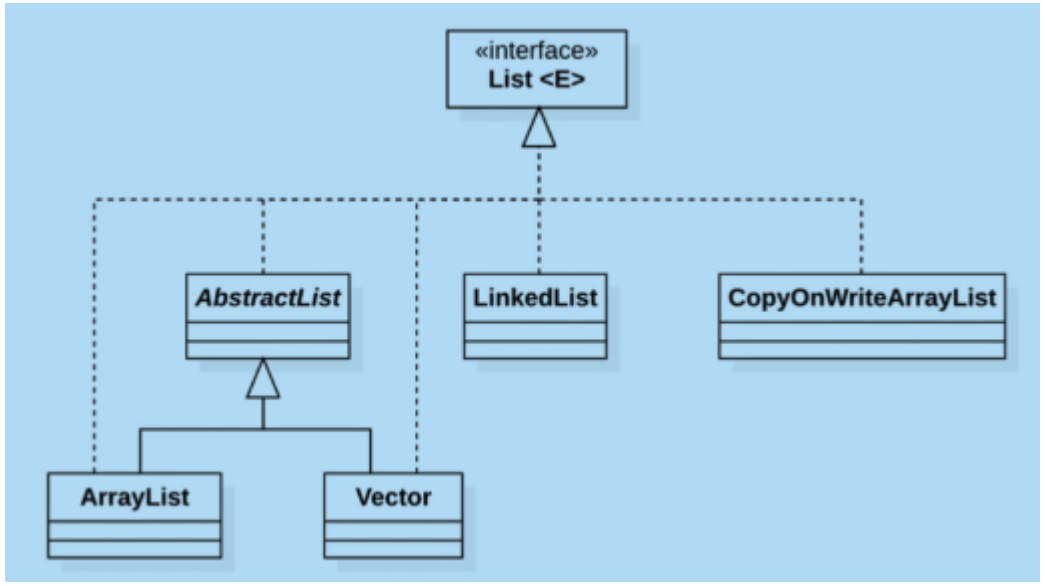
- Collection 인터페이스에서는 List와 Set 인터페이스의 많은 공통된 부분을 정의하고 있다

메소드	설명
boolean add(E e)	해당 컬렉션(collection)에 전달된 요소를 추가함. (선택적 기능)
void clear()	해당 컬렉션의 모든 요소를 제거함. (선택적 기능)
boolean contains(Object o)	해당 컬렉션이 전달된 객체를 포함하고 있는지를 확인함.
boolean equals(Object o)	해당 컬렉션과 전달된 객체가 같은지를 확인함.
boolean isEmpty()	해당 컬렉션이 비어있는지를 확인함.
Iterator<E> iterator()	해당 컬렉션의 반복자(iterator)를 반환함.
boolean remove(Object o)	해당 컬렉션에서 전달된 객체를 제거함. (선택적 기능)
int size()	해당 컬렉션의 요소의 총 개수를 반환함.
Object[] toArray()	해당 컬렉션의 모든 요소를 Object 타입의 배열로 반환함.

14.6.3. List 인터페이스[1/5]

- List 인터페이스는 Collection의 다른 인터페이스들과 가장 큰 차이는 배열처럼 순서가 있다는 것이다
- List 계열의 컬렉션은 저장요소를 순차적으로 관리하며 요소로 중복 값과 null값을 가질수 있다
- 요소에 대한 접근은 인덱스를 통해서 한다
- List 계열의 대표 클래스는 ArrayList, LinkedList가 있다

List Interface 구현 클래스들



List

+ add(Object) : boolean
+ get(int) : Object
+ set(int, Object) : Object
+ listIterator() : ListIterator
+ remove(int) : Object
+ subList(int, int) : List

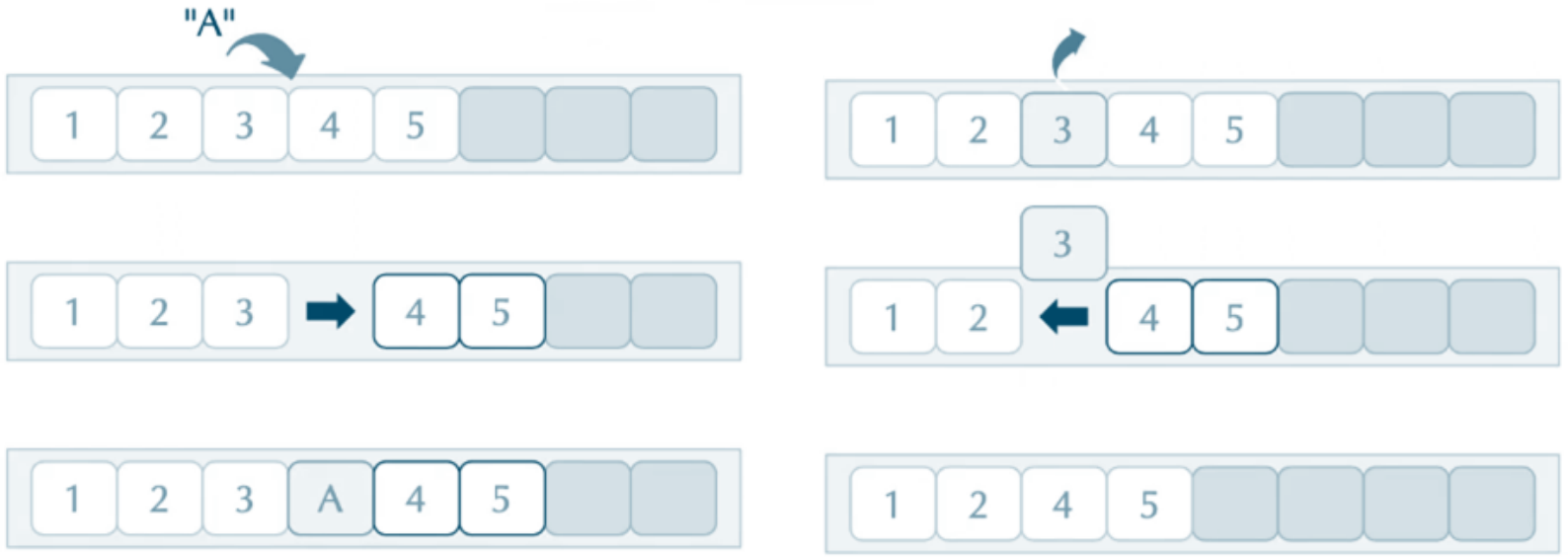
14.6.3. List 인터페이스[2/5] - ArrayList 클래스(1)

- ArrayList 클래스는 배열을 이용하기 때문에 인덱스를 이용해 배열 요소에 빠르게 접근할 수 있다.

- 배열은 크기를 변경할 수 없는 인스턴스이므로, 내부배열의 용량(Capacity)을 넘어 요소를 저장할 경우 내부적으로 용량을 늘린 새로운 배열을 만들어 요소를 옮겨야 한다
  물론 이 과정은 자동으로 수행되지만, 요소의 추가 및 삭제 작업에 시간이 걸리는 단점을 가지게 된다.

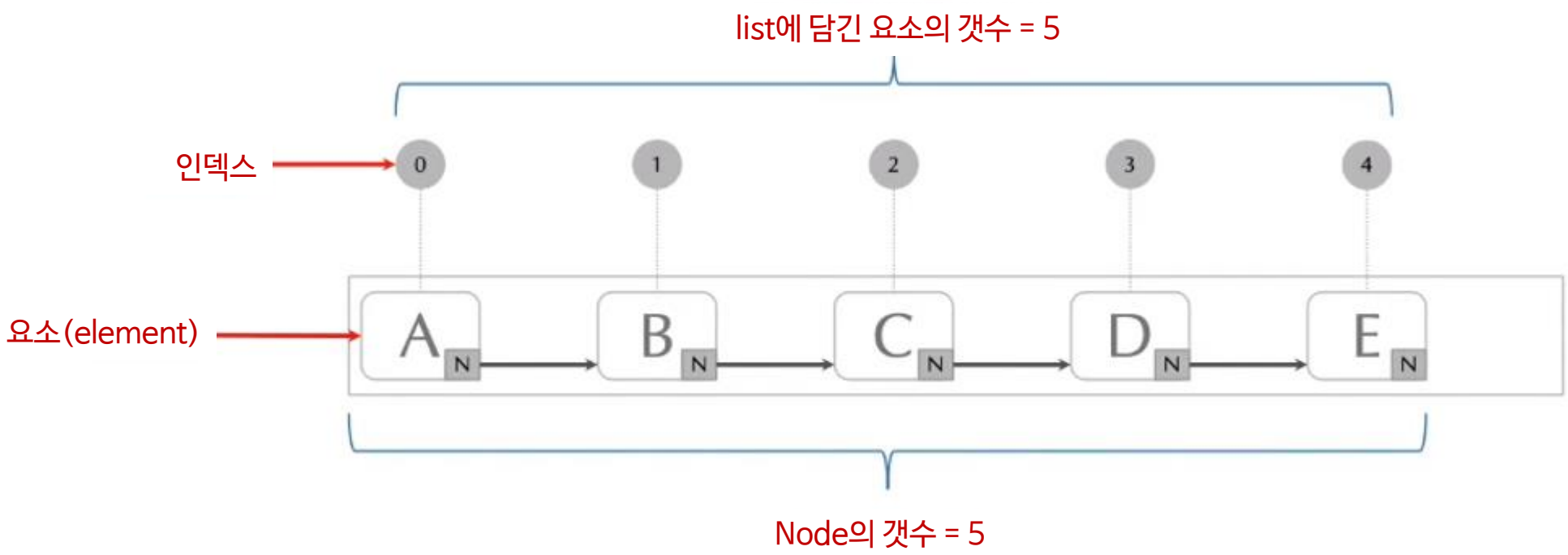
14.6.3. List 인터페이스[3/5] - ArrayList 클래스(2)

- ArrayList는 순차적으로 요소를 저장할 수 있는 메소드와 인덱스를 이용하여 저장할 수 있는 add()메소드를 제공한다
- 특정 인덱스 위치에 요소를 저장할 경우 내부적으로 기존에 저장된 요소들의 이동이 발생한다
- 특정요소의 삭제에도 해당 요소의 인덱스가 필요하며 이때도 기존 요소들의 이동이 이루어진다
- 이런 점 때문에 요소들의 추가나 삭제가 빈번하게 이루어지는 데이터 관리에는부하가 많이 걸려 퍼포먼스가 떨어질 수 있으므로 적합하지 않다



14.6.3. List 인터페이스[4/5] - LinkedList 클래스(1)

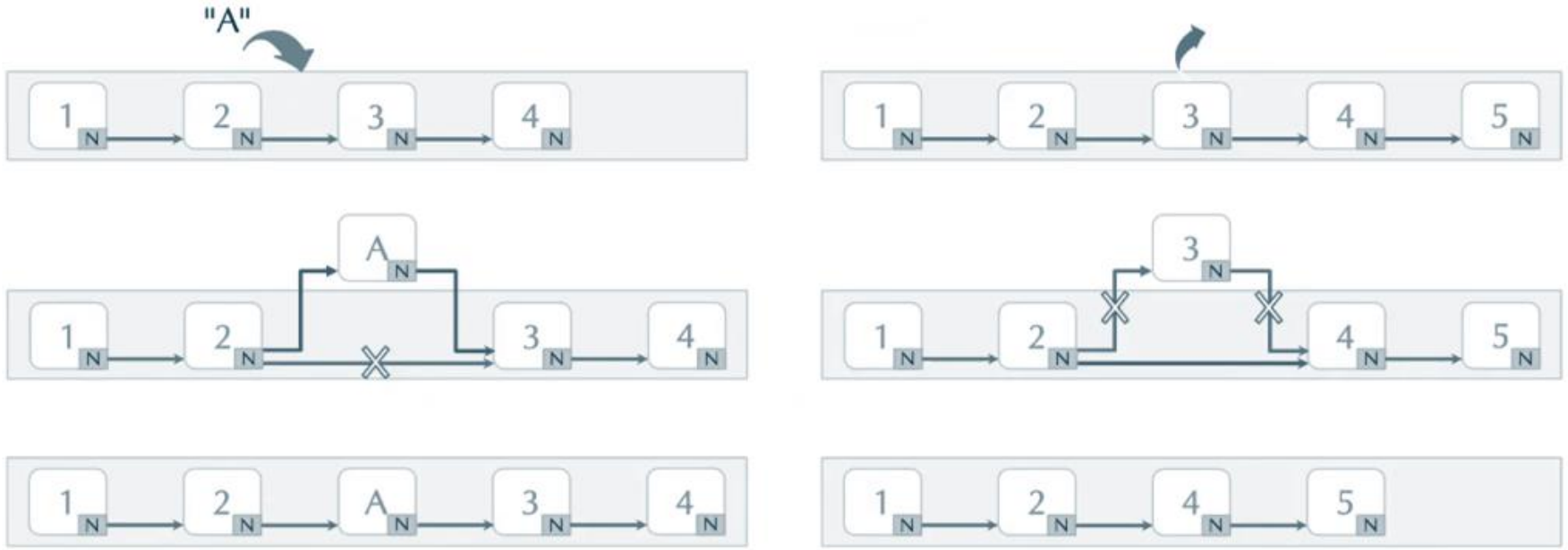
- LinkedList 는 ArrayList 가 배열을 이용하여 요소를 저장함으로써 발생하는 단점을 극복하기 위해 나옴
- 내부적으로 연결 리스트(linked list)를 이용하여 요소를 저장
- 연결 리스트는 저장된 요소가 비순차적으로 분포되며, 이러한 요소들 사이를 링크(link)로 연결하여 구성
- LinkedList 역시 List 인터페이스를 구현하므로, ArrayList 와 사용할 수 있는 메소드가 거의 같다



14.6.3. List 인터페이스[5/5] - LinkedList 클래스(2)

- LinkedList에서 요소를 add()하면 노드도 함께 추가된다
- 일반적인 add()는 순차적으로 링크를 연결하며 추가한다

- 특정 인덱스를 지정해서 요소를 추가하는 경우 기존 노드의 연결을 끊고 새로운 노드를 연결하여 추가한다
- 요소의 삭제 역시 특정 인덱스를 지정해야 하므로 삭제할 노드의 다음 노드에 대한 포인트를 이전 노드가 가리킬 수 있도록 하여 삭제한다
- ArrayList보다 요소의 추가 삭제에 대한 부하는 적지만 메모리 사용량이 늘어난다는 단점이 있다

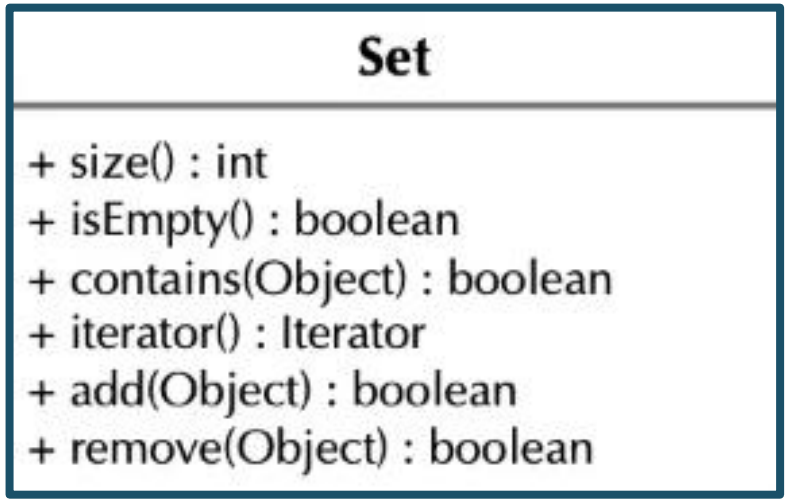


14.6.4. Set 인터페이스

- Set 인터페이스를 구현한 Set 컬렉션 클래스는 다음과 같은 특징을 가진다
  - ✓ 요소의 저장 순서를 유지하지 않는다.
  - ✓ 같은 요소의 중복 저장을 허용하지 않는다.

- 단일 요소를 꺼내기 위한 get() 메소드가 존재하지 않는다
- Set 인터페이스는 Collection 인터페이스를 상속받으므로, Collection 인터페이스에서 정의한 메소드도 모두 사용할 수 있다.

- Set 인터페이스를 구현한 주요 클래스는 HashSet, LinkedSet, TreeSet 등이 있다

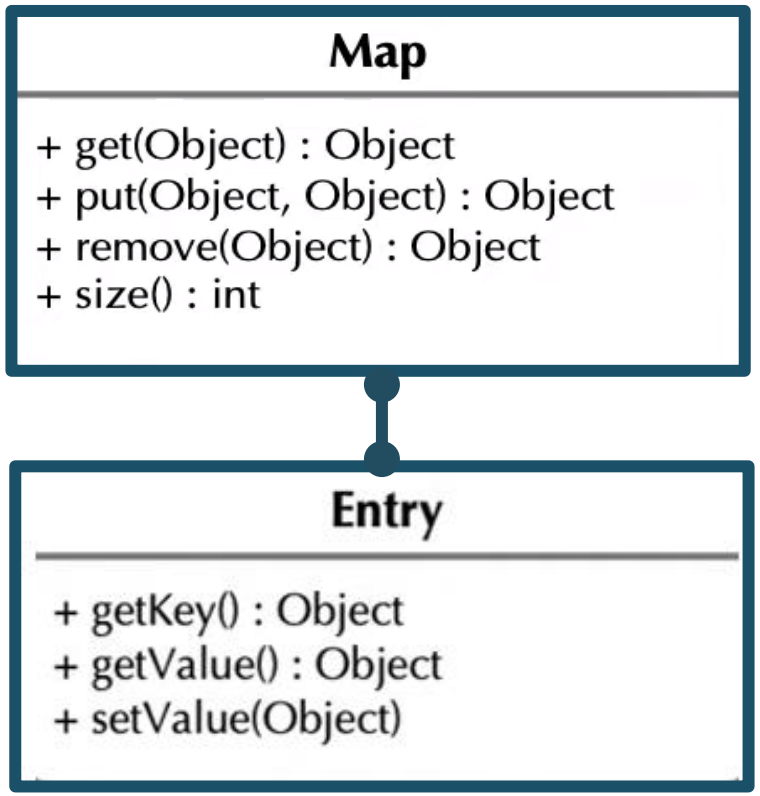


14.6.5. Map 인터페이스

- Map 인터페이스의 특징은 요소를 저장하기 위해 유니크한 key와 함께 저장해야 한다
- Map 인터페이스는 Collection 인터페이스를 상속하지 않는다
- Map 인터페이스는 내부에 (key, value)의 쌍으로된 Entry인터페이스를 가지고 있다
- Map 인터페이스의 주요 구현 클래스는 HashMap, LinkedMap, TreeMap이 있다

Map의 주요특징

- 모든 데이터는 키와 값이 존재한다.
- 키가 없이 값만 저장할 수는 없다.
- 값이 없이 키만 저장할 수 없다.
- 키는 해당 Map에서 고유해야만 한다.
- 값은 Map에서 중복되어도 된다.
- 데이터 추가 순서는 중요하지 않다



14.6.6. Iterator 인터페이스

- 자바의 컬렉션 프레임워크는 컬렉션에 저장된 요소를 읽어오는 방법을 Iterator 인터페이스로 표준화 하고 있다
- Collection 인터페이스에서는 Iterator 인터페이스를 구현한 클래스의 인스턴스를 반환하는 iterator() 메소드를 정의하여 각 요소에 접근하도록 하고 있다
- Collection 인터페이스를 상속받는 List와 Set 인터페이스에서도 iterator() 메소드를 사용할 수 있다
- Map 구현 클래스의 경우 Map.value()메소드를 통해 요소들을 컬렉션타입으로 객체를 반환 받은 후 Iterator를 이용하여 순회할 수 있다

메소드	실행
boolean hasNext()	해당 이터레이션(iteration)이 다음 요소를 가지고 있으면 true를 반환하고, 더 이상 다음 요소를 가지고 있지 않으면 false를 반환함.
E next()	이터레이션(iteration)의 다음 요소를 반환함.
default void remove()	해당 반복자로 반환되는 마지막 요소를 현재 컬렉션에서 제거함. (선택적 기능)