

# 자바 소개

## ❖ 자바 (Java)

- 1995년 마이크로시스템즈(Sun Microsystems)에서 발표
- 현재 웹사이트 및 다양한 애플리케이션 개발의 핵심 언어
- 오라클 (<http://www.oracle.com>) 라이선스
- 특징
  - 모든 운영체제에서 실행 가능
  - 객체 지향 프로그래밍 (OOP : Object-Oriented Programming)
  - 메모리 자동 정리
  - 풍부한 무료 라이브러리

# 자바 개발 도구

## ❖ 자바 개발 도구 (JDK : Java Development Kit)

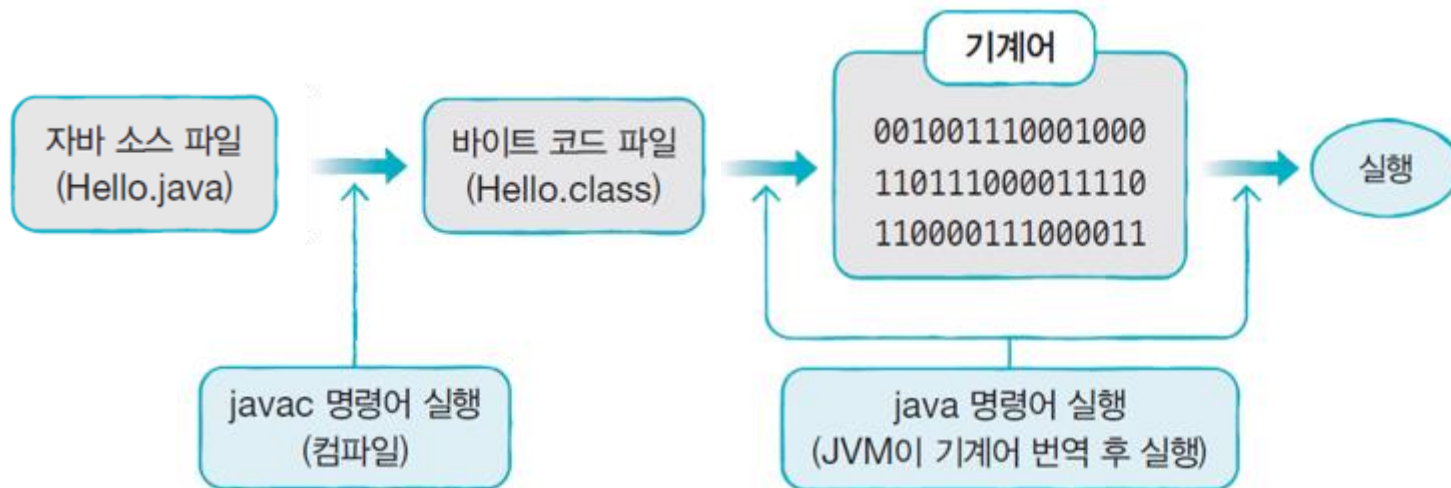
### ■ JDK 역할

- 자바 언어로 소프트웨어를 개발할 때 필요한 환경 및 도구를 제공하는 역할

### ■ JDK 종류

- Open JDK: <https://openjdk.java.net>
  - 개발, 학습용 및 상업용 모두 무료로 사용
- Oracle JDK: <https://www.oracle.com>
  - 개발, 학습용은 무료로 사용
  - 상업용 목적으로 사용할 경우 연간 사용료 지불
  - 장기 기술지원(LTS: Long Term Support) 및 업데이트 제공으로 안정적

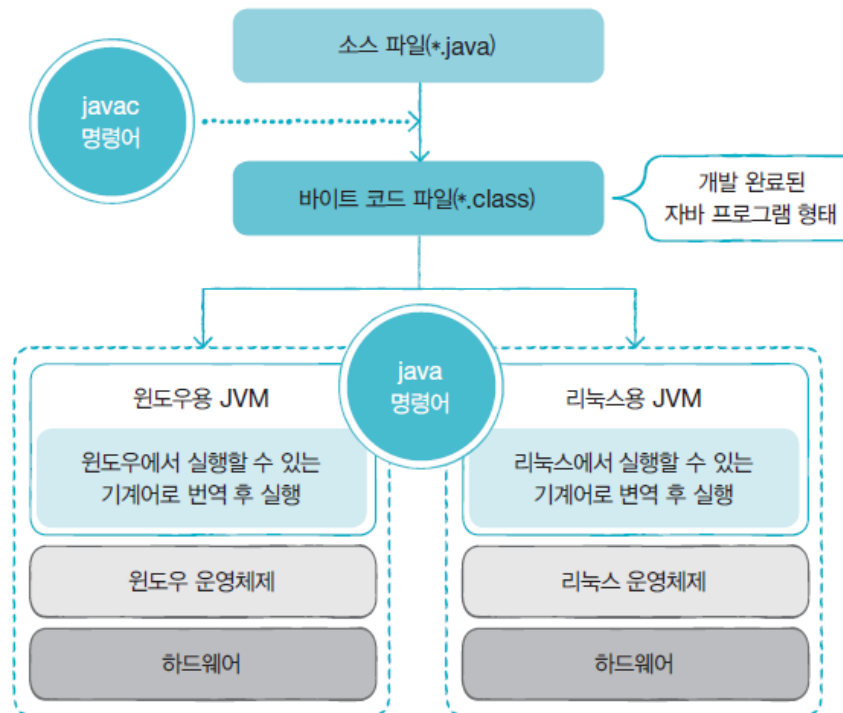
# 프로그램 개발 및 실행 프로세스



# 바이트 코드 파일과 자바 가상 기계

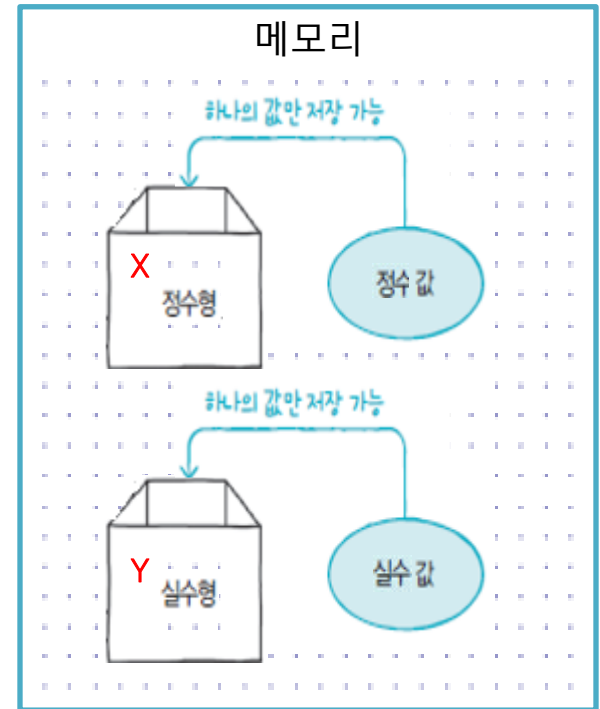
## ❖ 바이트 코드 파일과 자바 가상 기계

- 자바 프로그램은 완전한 기계어가 아닌, 바이트 코드(byte code) 파일(.class)로 구성
- 바이트 코드 파일은 운영체제에서 바로 실행할 수 없음
- 자바 가상 기계(JVM: Java Virtual Machine)가 완전한 기계어로 번역하고 실행



# 변수

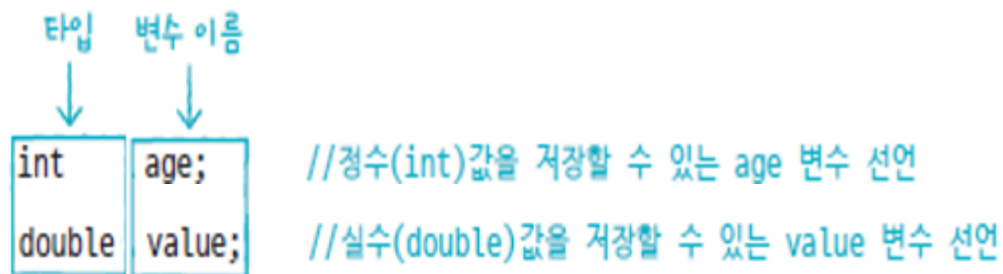
- ❖ 값을 저장할 수 있는 메모리의 특정 번지에 붙여진 이름
- ❖ 변수 통해 해당 메모리 번지에 하나의 값 저장하고 읽을 수 있음
- ❖ 변수는 정수, 실수 등 다양한 타입의 값을 저장할 수 있음



# 변수 선언

## ❖ 변수 사용 위해서 변수 선언 필요

- 변수에 어떤 타입의 데이터 저장할 것인지, 변수 이름 무엇인지 결정



- 같은 타입의 변수는 콤마 이용해 한꺼번에 선언할 수 있음




# 값 저장

## ❖ 값을 저장할 경우 대입 연산자 (=) 사용

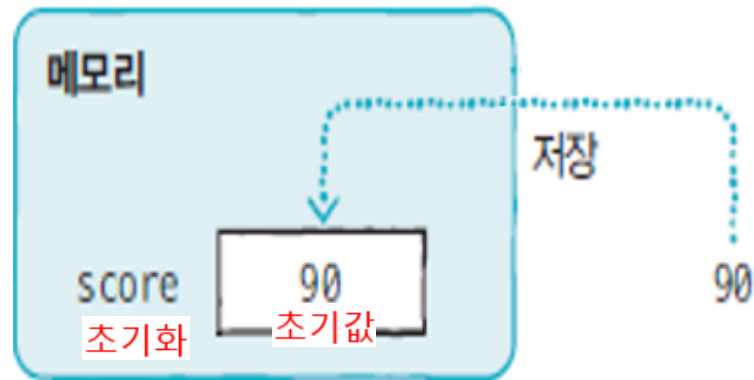
- 변수를 선언하고
- 대입 연산자를 사용해서 오른쪽의 값을 왼쪽의 변수에 저장

```
int score;           //변수 선언  
score = 90;          //값 저장
```



## ❖ 변수 초기화

- 변수에 최초로 값이 저장될 때
- 메모리에 변수가 생성
- 이것을 변수 초기화라 하고
- 이 때의 값을 초기값이라고 함



## 변수 사용

❖ 변수 사용: 변수의 값을 이용해서, 출력문이나 연산식을 수행하는 것

```
int hour = 3;  
int minute = 5;
```

```
System.out.println(hour + "시간 " + minute + "분"); //변수 hour와 minute 값을 출력: 3시간 5분
```

```
int totalMinute = (hour*60) + minute;
```



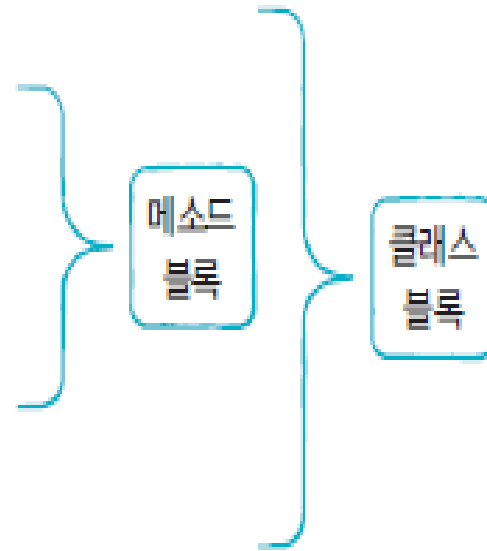
# 변수 사용 범위

## ❖ 로컬 변수 (Local Variable)

- 메소드 블록 내에서 선언된 변수를 로컬 변수라고 함

```
public class VariableExample {  
    public static void main(String[] args) {  
        int value = 10;           //로컬 변수 value  
        int sum = value + 20;      //로컬 변수 sum  
        System.out.println(sum);  
    }  
}
```

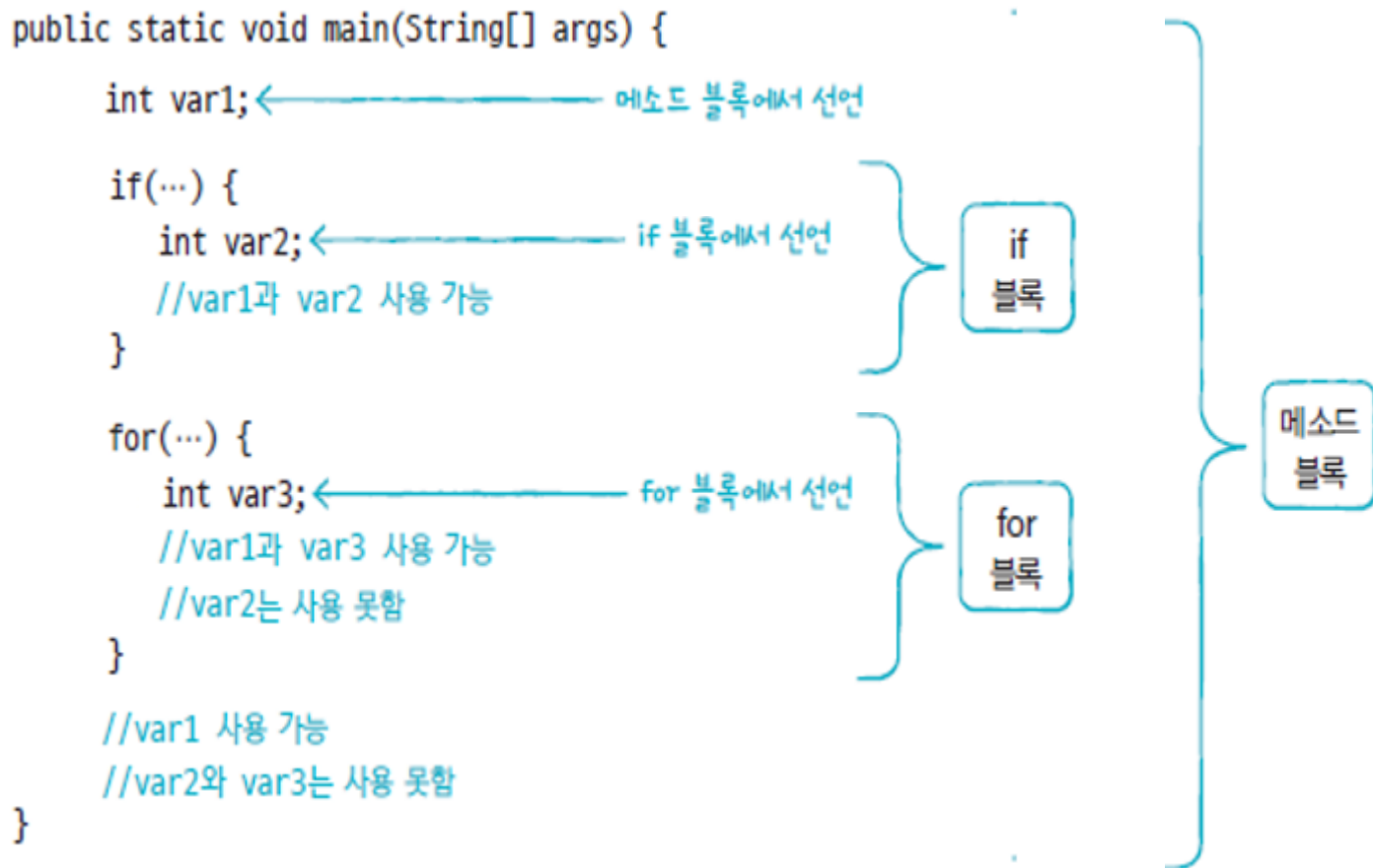
이 위치에서는 value와 sum 변수는 사용 못함



- 로컬 변수는 메소드 블록 내에서만 사용되고 메소드 실행이 끝나면 자동 삭제됨

# 변수 사용 범위

- 로컬 변수는 해당 중괄호 블록 내에서만 사용 가능



# 기본 타입

❖ 자바 언어는 정수, 실수, 논리값 저장하는 총 8 개의 기본 타입을 제공

구분	저장되는 값에 따른 분류	타입의 종류
기본 타입	정수 타입	byte, char, short, int, long
	실수 타입	float, double
	논리 타입	boolean

## 정수 타입

정수 타입

- 메모리 사용 크기와 저장되는 값의 허용 범위 각기 다름

타입	메모리 사용 크기		저장되는 값의 허용 범위	
byte	1byte	8bit	$-2^7 \sim (2^7-1)$	$-128 \sim 127$
short	2byte	16bit	$-2^{15} \sim (2^{15}-1)$	$-32,768 \sim 32,767$
char	2byte	16bit	$0 \sim (2^{16}-1)$	$0 \sim 65535$ (유니코드)
int	4byte	32bit	$-2^{31} \sim (2^{31}-1)$	$-2,147,483,648 \sim 2,147,483,647$
long	8byte	64bit	$-2^{63} \sim (2^{63}-1)$	$-9,223,372,036,854,775,808 \sim 9,223,372,036,854,775,807$

메모리

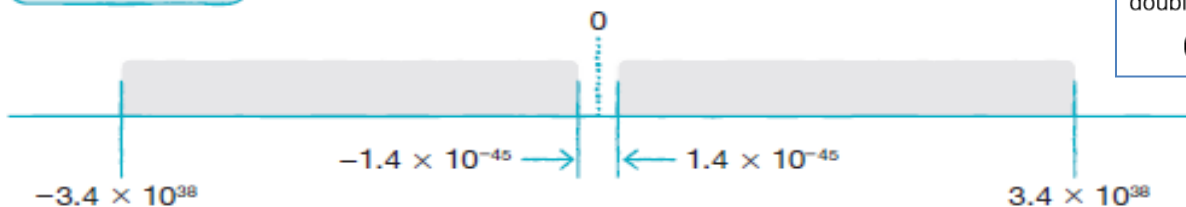
[illegible]

# 실수 타입

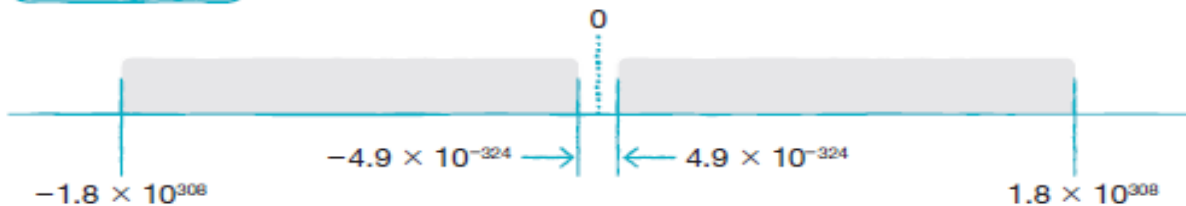
## ❖ 실수 타입

타입	메모리 사용 크기		저장되는 값의 허용 범위(양수 기준)	정밀도(소수점 이하 자리)
float	4byte	32bit	$(1.4 \times 10^{-45}) \sim (3.4 \times 10^{38})$	7자리
double	8byte	64bit	$(4.9 \times 10^{-324}) \sim (1.8 \times 10^{308})$	15자리

float 타입



double 타입



float: 소수점 이하자리 약 7 (6~9)

**0.12345679**

double: 소수점 이하자리 약 15 (15~18)

**0.1234567890123457**

# 논리 타입

## ❖ 논리 타입

- 참과 거짓에 해당하는 true와 false 리터럴을 저장하는 타입

```
boolean stop = true;  
boolean state = false;
```

- 두 가지 상태값에 따라 제어문의 실행 흐름을 변경하는데 사용

# 연산자의 종류

- ❖ 자바에서 제공하는 연산자
  - 산출되는 값의 타입이 연산자별로 다름

연산자 종류	연산자	피연산자 수	산출값	기능
산술	+, -, *, /, %	이항	숫자	사칙연산 및 나머지 계산
부호	+, -	단항	숫자	음수와 양수의 부호
문자열	+	이항	문자열	두 문자열을 연결
대입	=, +=, -=, *=, /=, %=	이항	다양	우변의 값을 좌변의 변수에 대입
증감	++, --	단항	숫자	1만큼 증가/감소
비교	==, !=, >, <, >=, <=, instanceof	이항	boolean	값의 비교
논리	!, &,  , &&,	단항 이항	boolean	논리 부정, 논리곱, 논리합
조건	(조건식) ? A : B	삼항	다양	조건식에 따라 A 또는 B 중 하나를 선택

# 연산의 방향과 우선순위

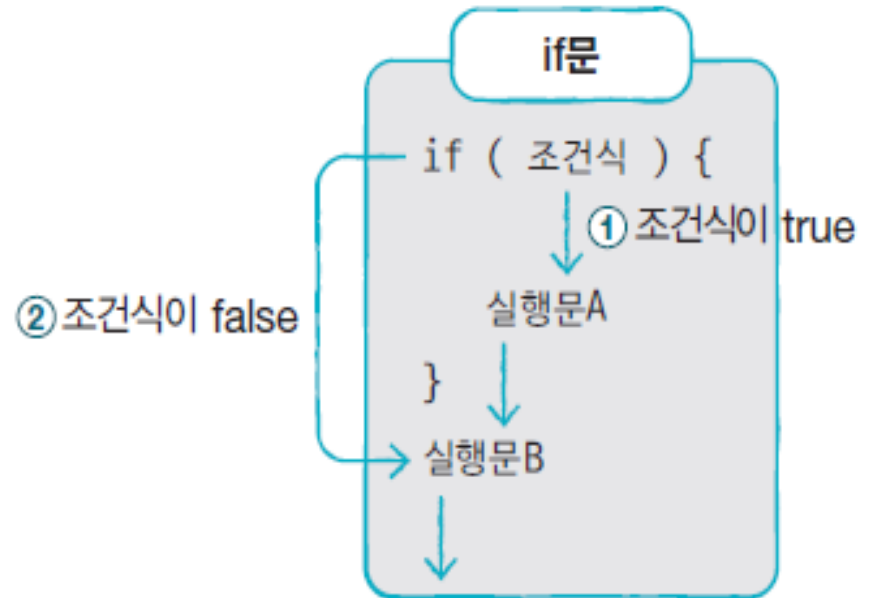
연산자	연산 방향	우선순위
증감(++), (--), 부호(+, -), 논리(!)	←	<div>높음</div> <div>↕</div> <div>낮음</div>
산술(*, /, %)	→	
산술(+, -)	→	
비교(<, >, <=, >=, instanceof)	→	
비교(==, !=)	→	
논리(&)	→	
논리(^)	→	
논리( )	→	
논리(&&)	→	
논리(!!)	→	
조건(?:)	→	
대입(=, +=, -=, *=, /=, %=)	←	



# if문

## ❖ if문

- 조건식 결과에 따라 블록 실행 여부가 결정
- 조건식에 올 수 있는 요소
  - true / false 값을 산출하는 연산식
  - boolean 타입 변수
- 중괄호 블록은 조건식이 true가 될 때 실행
  - 실행할 문장 하나뿐인 경우 생략 가능



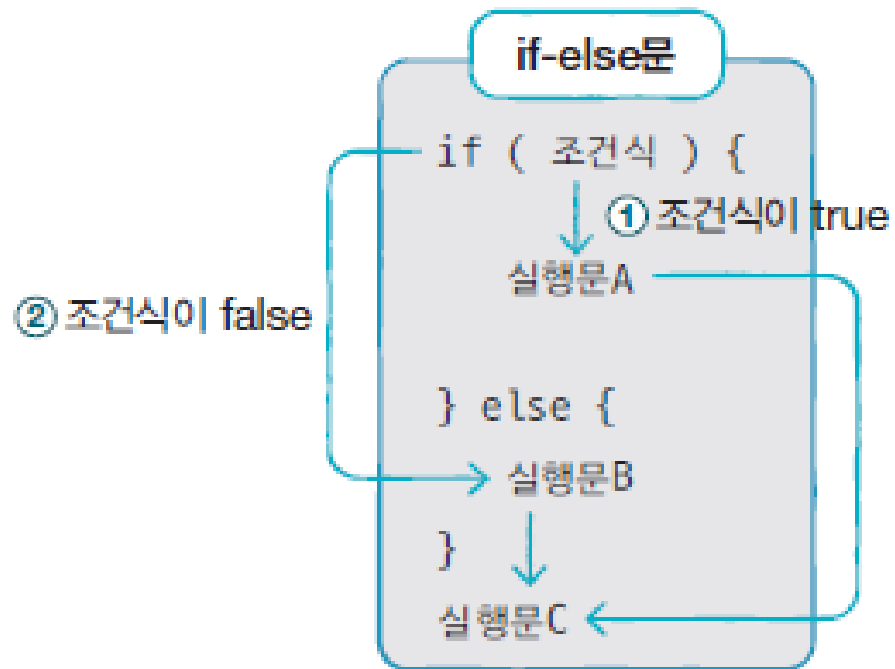
```
if ( 조건식 ) {  
    실행문;  
    실행문;  
    ...  
}
```

```
if ( 조건식 )  
    실행문;
```

# if-else문

## ❖ if-else문

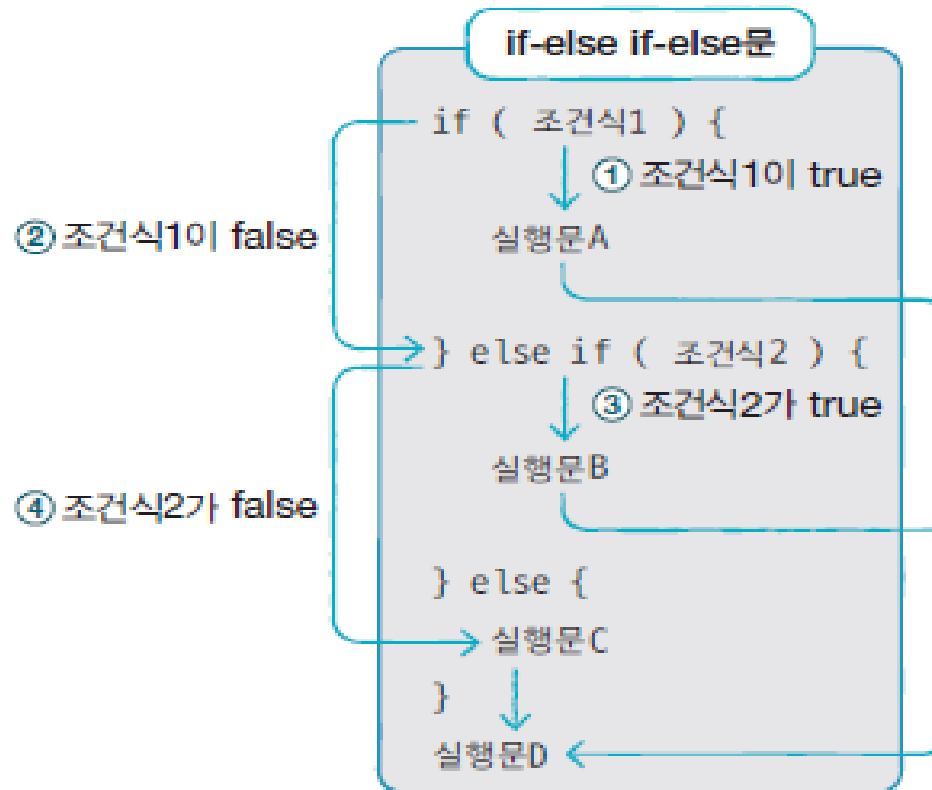
- if문을 else 블록과 함께 사용
- 조건식의 결과에 따라 실행 블록 선택
  - if문 조건식 true이면 if문 블록 실행
  - if문 조건식 false이면 else 블록 실행



# if-else if-else문

## ❖ if-else if-else문

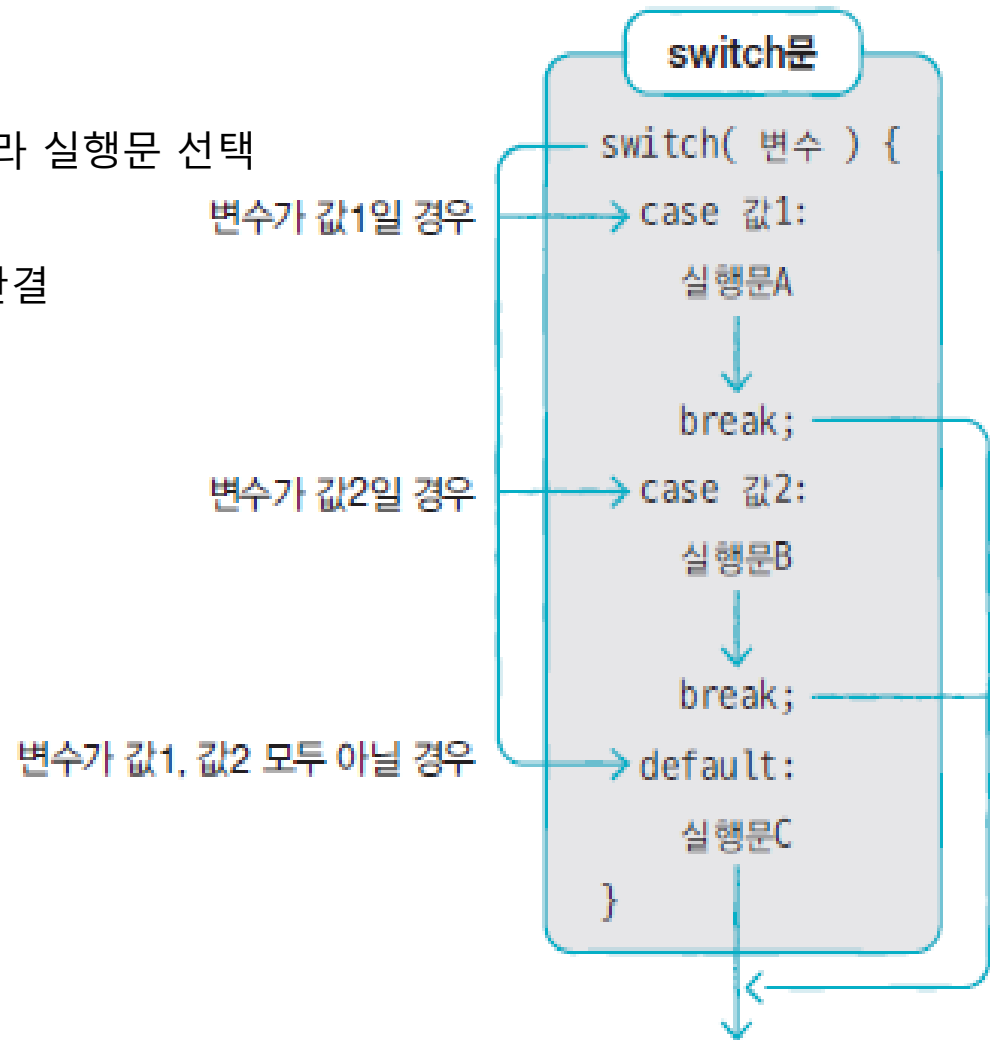
- 조건식이 여러 개인 if문
- 처음 if문의 조건식이 false일 경우 다른 조건식의 결과에 따라 실행 블록 선택
  - if 블록 끝에 else if문 추가
  - else if문 개수는 제한 없음



# swith문

## ❖ switch문

- 변수가 어떤 값을 갖는가에 따라 실행문 선택
- 같은 기능의 if문보다 코드가 간결



# for문

## ❖ for문

- 주어진 횟수만큼 반복하고 싶을 경우

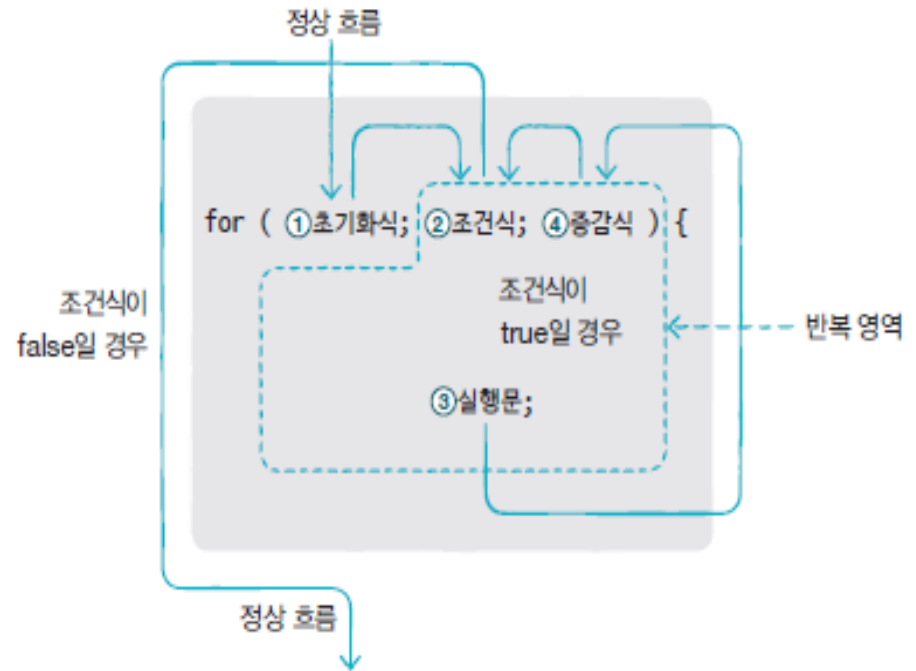
```
int sum = 0;
sum = sum + 1;
sum = sum + 2;
sum = sum + 3;
sum = sum + 4;
sum = sum + 5;
System.out.println("1~5의 합:" + sum);
```

5개의 실행문



```
int sum = 0;
for (int i=1; i<=100; i++) {
    sum = sum + i;
}
System.out.println("1~100의 합:" + sum);
```

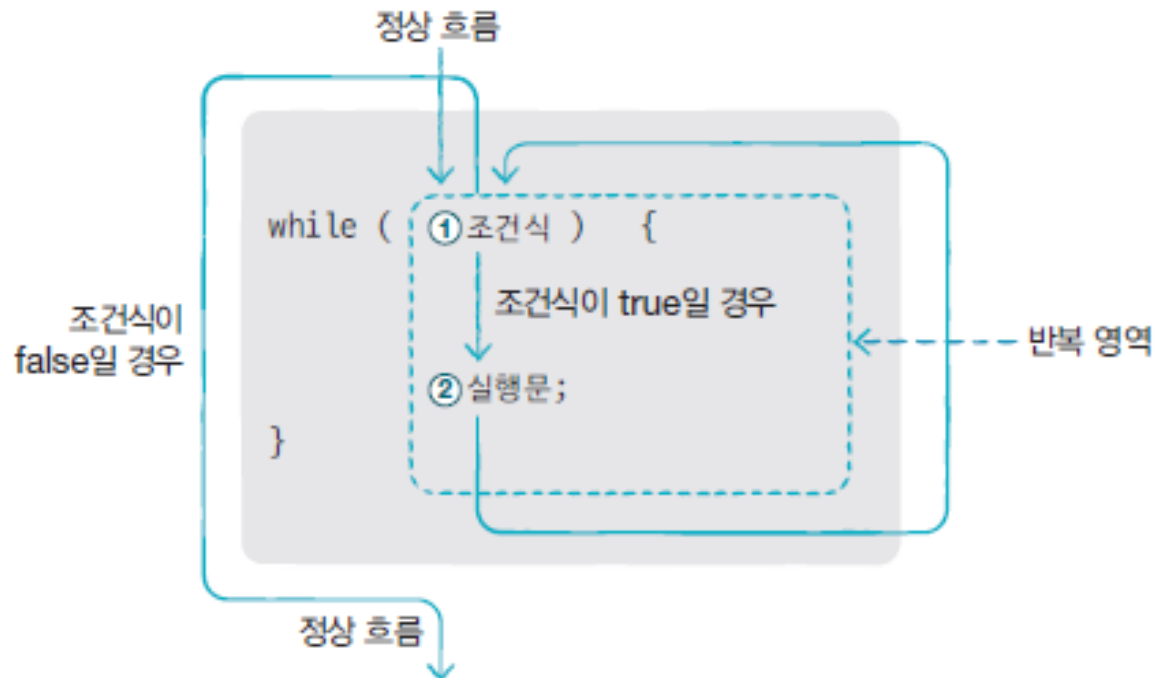
100번 반복



# while문

## ❖ while문

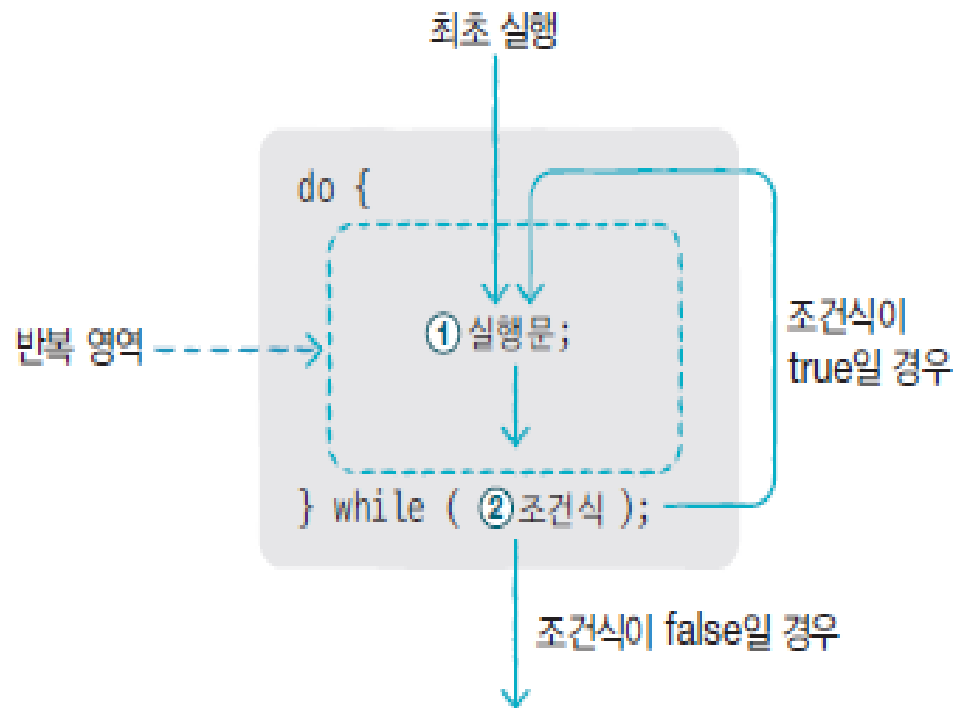
- 조건식에 따라 반복 여부를 결정할 경우
  - true일 경우 계속해서 반복
  - false일 경우 반복 종료
- 조건식에는 주로 비교 연산식, 논리 연산식



# do-while문

## ❖ do-while문

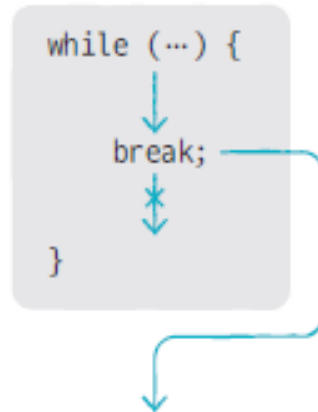
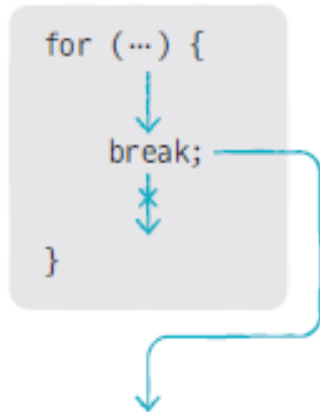
- 조건식에 의해 반복 실행하는 점에서 while문과 동일
- 블록 내부 실행문을 우선 실행하고 그 결과에 따라 반복 실행 여부를 결정



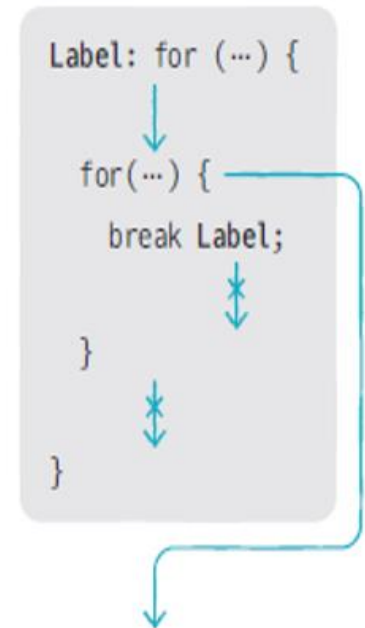
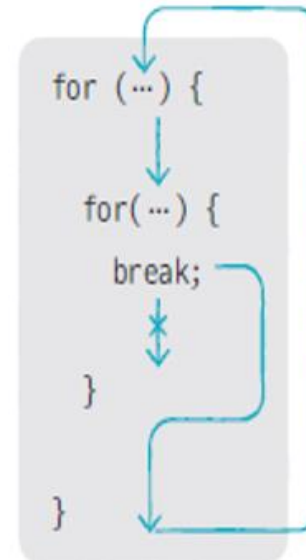
# break문

## ❖ break문

- for, while, do-while, switch문의 실행을 중지할 때 사용
- 주로 if문과 함께 사용



- 반복문이 중첩되어 있을 경우
  - Label을 이용해서 바깥 반복문을 빠져나감

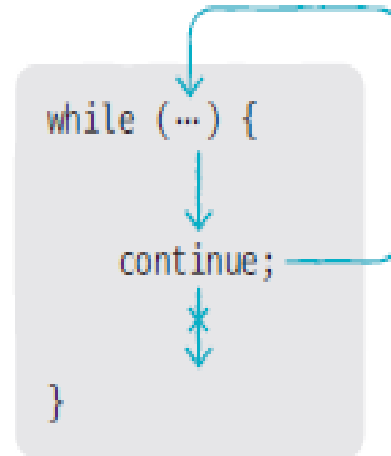
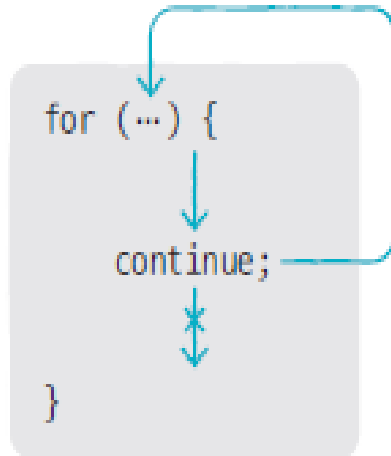




# continue문

## ❖ continue문

- for, while, do-while문에서만 사용
- for문의 증감식이나 while, do-while문의 조건식으로 이동
- 주로 if문과 함께 사용



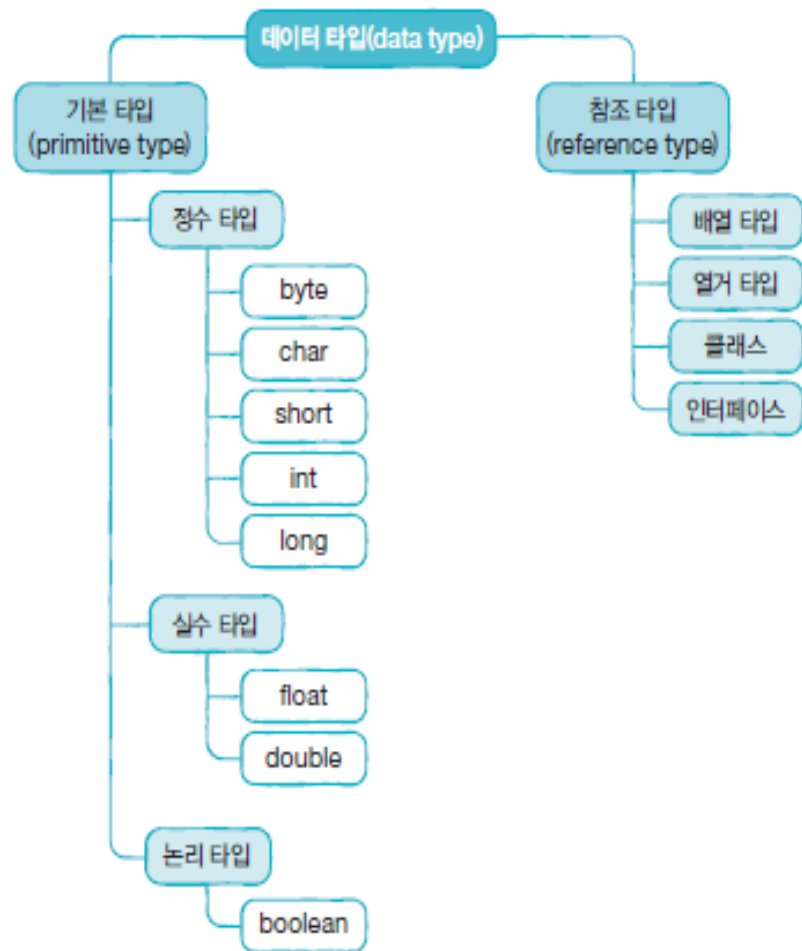
# 기본 타입과 참조 타입

## ❖ 기본 타입 (primitive type)

- 정수, 실수, 문자, 논리 리터럴 저장

## ❖ 참조 타입 (reference type)

- 객체(object)의 번지를 참조하는 타입
- 배열, 열거, 클래스, 인터페이스



# 기본 타입과 참조 타입

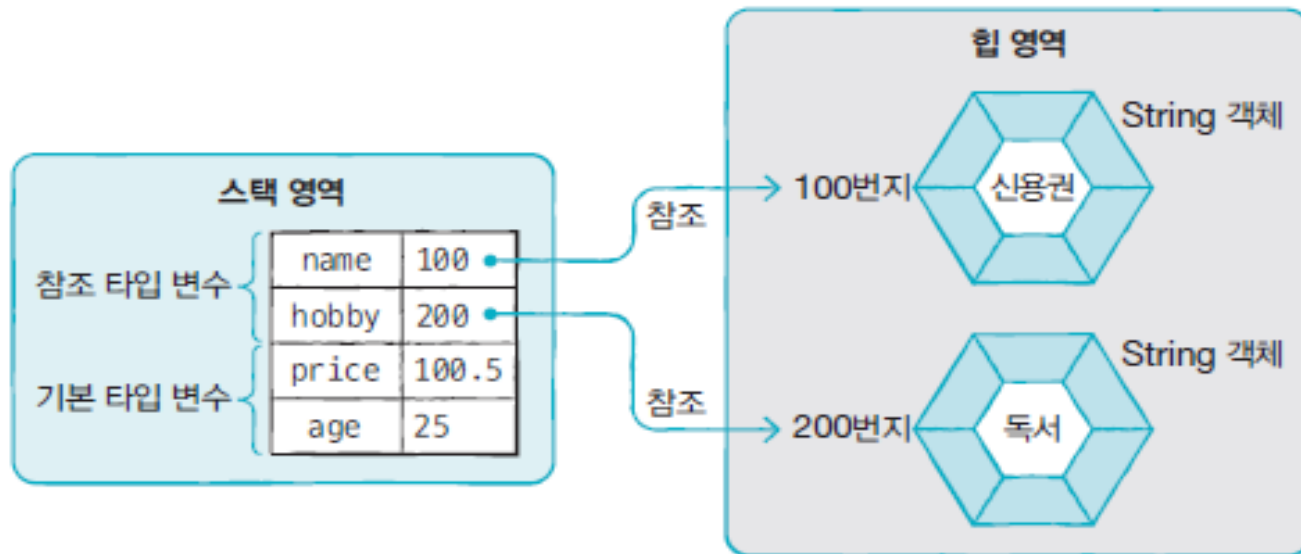
## ❖ 기본 타입 변수와 참조 타입 변수의 차이점

### 기본 타입 변수

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

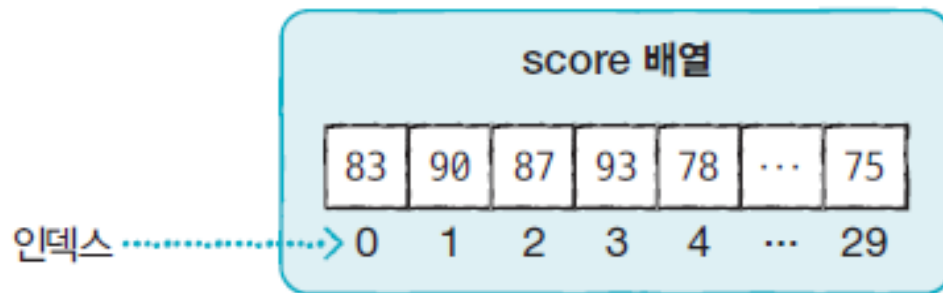
### 참조 타입 변수

```
String name = "신용권";  
String hobby = "독서";
```



# 배열

- ❖ 데이터를 연속된 공간에 나열하고 각 데이터에 인덱스(Index) 부여한 자료구조
- ❖ 같은 타입의 데이터만 저장할 수 있음
- ❖ 한 번 생성된 배열은 길이를 늘리거나 줄일 수 없음



# 배열 선언

## ❖ 배열 변수 선언

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

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

- 참조할 배열 객체 없는 경우 배열 변수는 null 값으로 초기화

```
타입[] 변수 = null;
```

## ❖ 배열 생성

- 값 목록으로 배열 생성

```
타입[] 변수 = { 값0, 값1, 값2, 값3, ... };
```

- new 연산자를 이용해서 배열 생성

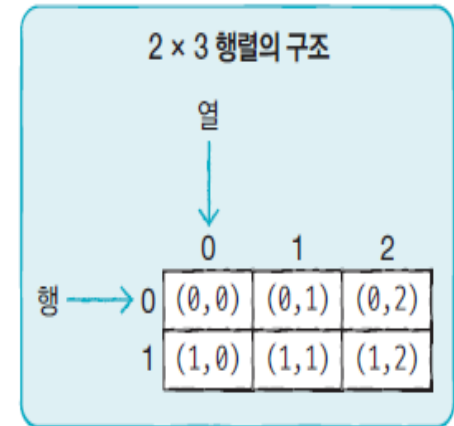
```
int[] scores = new int[30];
```

# 다차원 배열

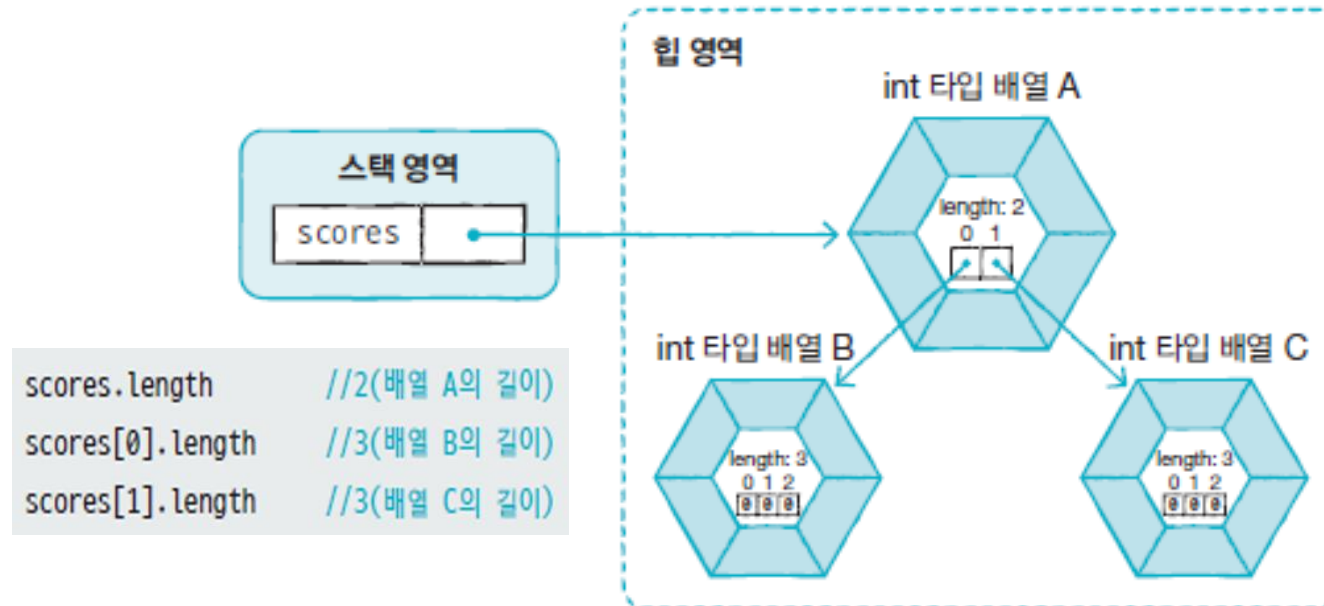
## ❖ 2차원 배열

### ■ 행렬 구조

```
int[][] scores = new int[2][3];
```



- 구현 방법: 1차원 배열이 다시 1차원 배열을 참조



# 다차원 배열

- 값 목록을 이용한 2차원 배열 생성

```
타입[ ][ ] 변수 = { {값1, 값2, ...}, {값1, 값2, ...}, ... };
```

↑                    ↑  
그룹 0 값 목록    그룹 1 값 목록

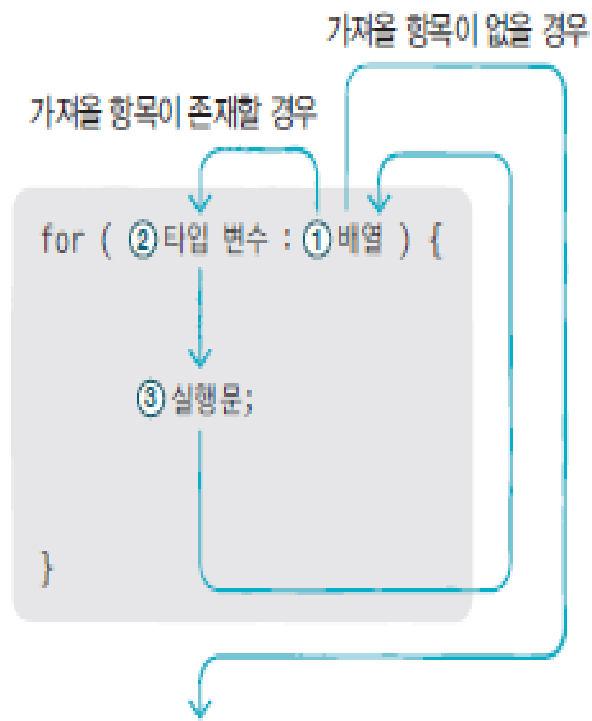
```
int[ ][ ] scores = { {95, 80}, {92, 96} };
```

```
int score = scores[0][0]; //95
```

```
int score = scores[1][1]; //96
```

# 향상된 for문

- ❖ 배열이나 컬렉션을 좀 더 쉽게 처리
- ❖ 반복 실행 위해 루프 카운터 변수나 증감식 사용하지 않음



```
int[] scores = { 95, 71, 84, 93, 87 };  
  
int sum = 0;  
for (int score : scores) {  
    sum = sum + score;  
}  
System.out.println("점수 총합 = " + sum);
```



# 열거 타입

- ❖ 소수의 한정된 값 목록만 저장할 수 있는 타입
- ❖ 열거 상수(한정된 값) 를 저장하는 타입

```
Week today;
```

```
today = Week.FRIDAY;
```

Week.java

```
public enum Week {
```

```
    MONDAY,
```

```
    TUESDAY,
```

```
    WEDNESDAY,
```

```
    THURSDAY,
```

```
    FRIDAY,
```

```
    SATURDAY,
```

```
    SUNDAY
```

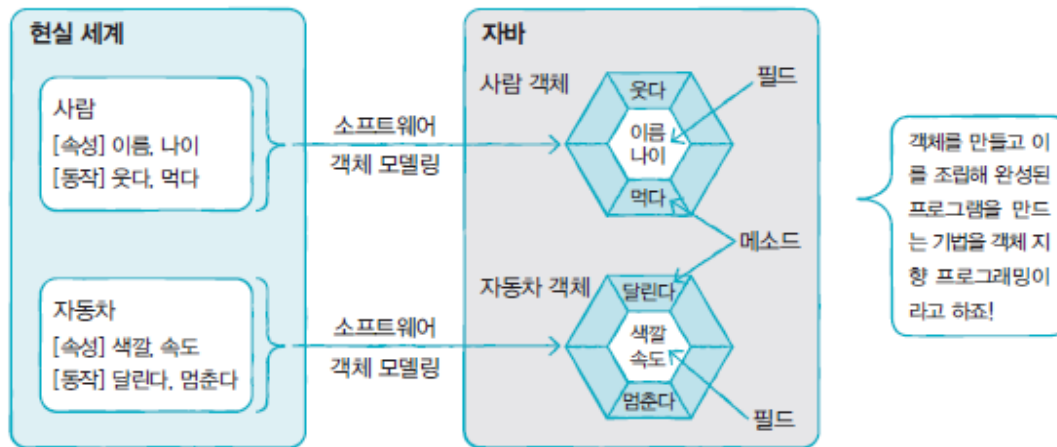
```
}
```

열거 타입 이름

열거 상수

# 객체

- ❖ 물리적으로 존재하거나 추상적으로 생각할 수 있는 것 중에서 자신의 속성을 가지며 식별 가능한 것
- ❖ 속성 (필드(field)) + 동작(메소드(method))로 구성



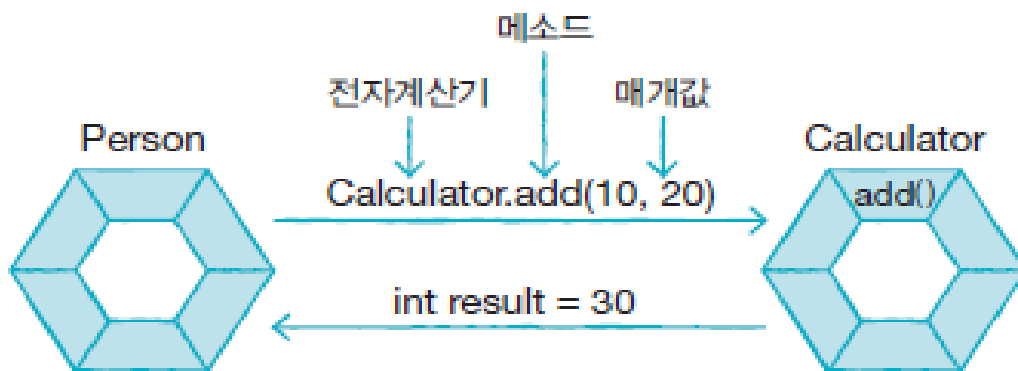
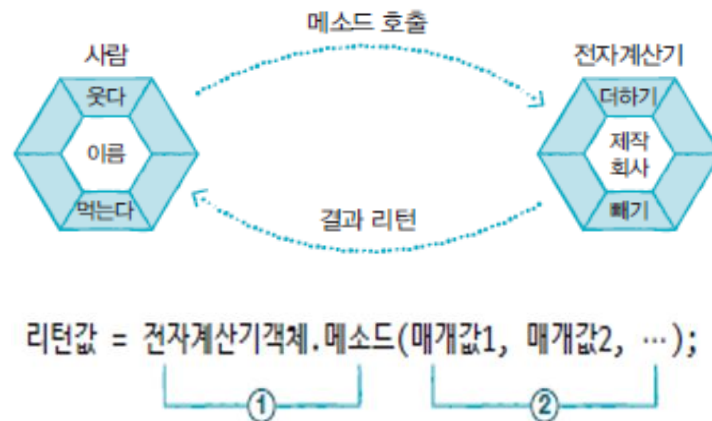
# 객체의 상호작용

## ❖ 객체와 객체 간의 상호작용

- 메소드를 통해 객체들이 상호작용
- 메소드 호출 : 객체가 다른 객체의 기능을 이용하는

```
int result = Calculator.add(10, 20);
```

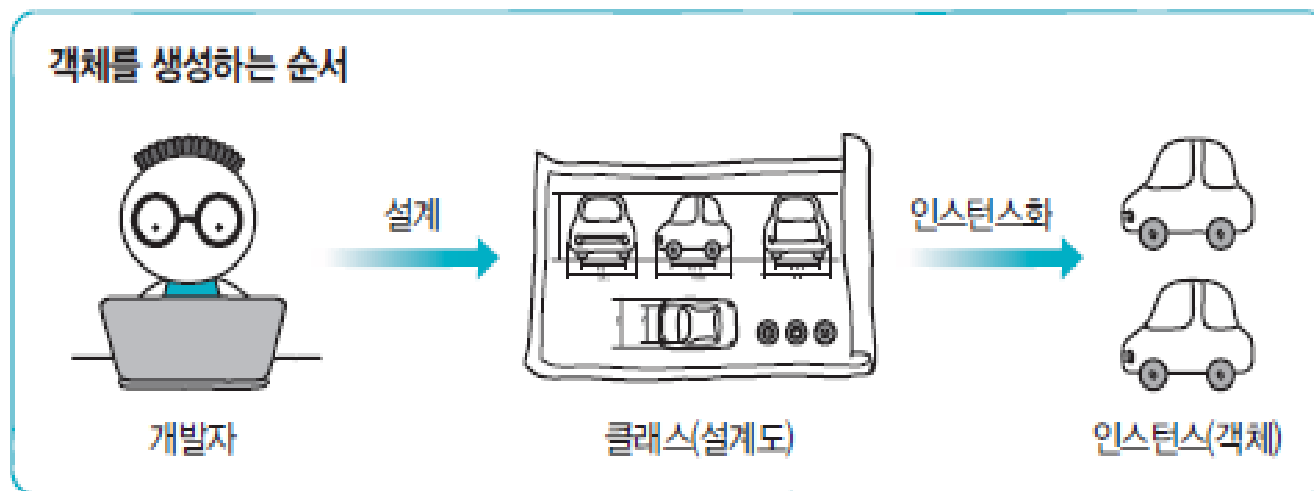
↑  
리턴한 값을 int 변수에 저장



# 객체와 클래스

## ❖ 클래스 (class)

- 자바의 설계도
- 인스턴스 (instance): 클래스로부터 만들어진 객체
- 객체지향 프로그래밍 단계
  - 클래스 설계 -> 설계된 클래스로 사용할 객체 생성 -> 객체 이용



# 클래스 선언

❖ '클래스 이름.java'로 소스 파일 생성

```
public class 클래스이름 {  
  
}
```

# 객체 생성과 클래스 변수

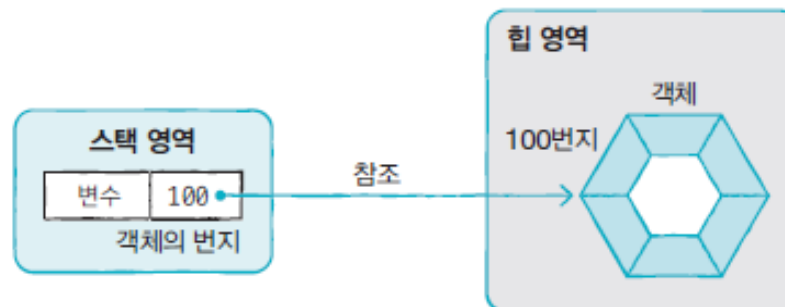
## ❖ 클래스로부터 객체를 생성

- new 클래스();
- new 연산자로 메모리 힙 영역에 객체 생성
- 객체 생성 후 객체 번지가 리턴
  - 클래스 변수에 저장하여 변수 통해 객체 사용 가능



```
클래스 변수;  
변수 = new 클래스();
```

```
클래스 변수 = new 클래스();
```



# 클래스의 구성 멤버

## ❖ 클래스 멤버

- 필드(Field)  
객체의 데이터가 저장되는 곳

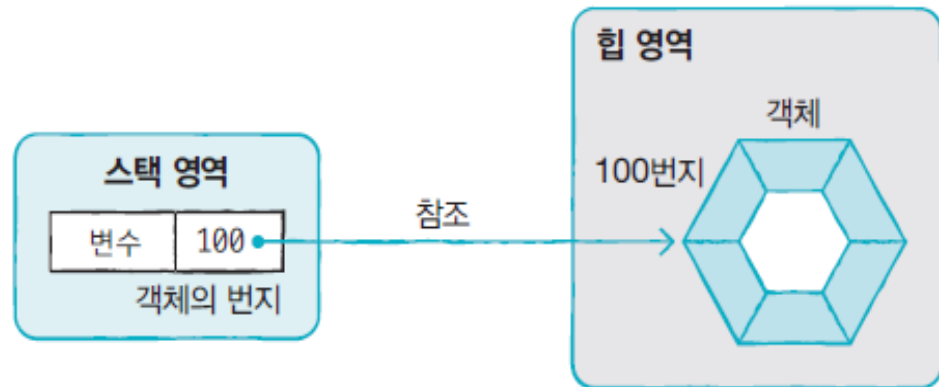
- 생성자(Constructor)  
객체 생성 시 초기화 역할 담당

- 메소드(Method)  
객체의 동작에 해당하는 실행 블록

```
public class ClassName {  
  
    //필드  
    int fieldname;  
  
    //생성자  
    ClassName() { ... }  
  
    //메소드  
    void methodName() { ... }  
  
}
```

# 생성자

- ❖ 클래스로부터 new 연산자로 객체를 생성할 때 호출되어 객체의 초기화를 담당
- ❖ 객체 초기화
  - 필드를 초기화하거나 메소드를 호출해서 객체를 사용할 준비를 하는 것
- ❖ 생성자가 성공적으로 실행
  - 힙 영역에 객체 생성되고 객체 번지가 리턴





# 기본 생성자

- ❖ 클래스 내부에 생성자 선언 생략할 경우 바이트 코드에 자동 추가

```
[public] 클래스() { }
```

- ❖ 클래스에 생성자 선언하지 않아도  
new 생성자()로 객체 생성 가능

```
Car myCar = new Car();
```

↑  
기본 생성자

소스 파일(Car.java)

```
public class Car {  
  
}
```

컴파일

바이트 코드 파일(Car.class)

```
public class Car {  
    public Car() { } //자동 추가  
}
```

↑  
기본 생성자

# 생성자 선언

## ❖ 생성자 선언

```
클래스( 매개변수선언, ... ) {  
    //객체의 초기화 코드  
}
```

} 생성자 블록

- 매개 변수 선언은 생략할 수도 있고 여러 개 선언할 수도 있음

```
public class Car {  
    //생성자  
    Car(String model, String color, int maxSpeed) { ... }  
}
```

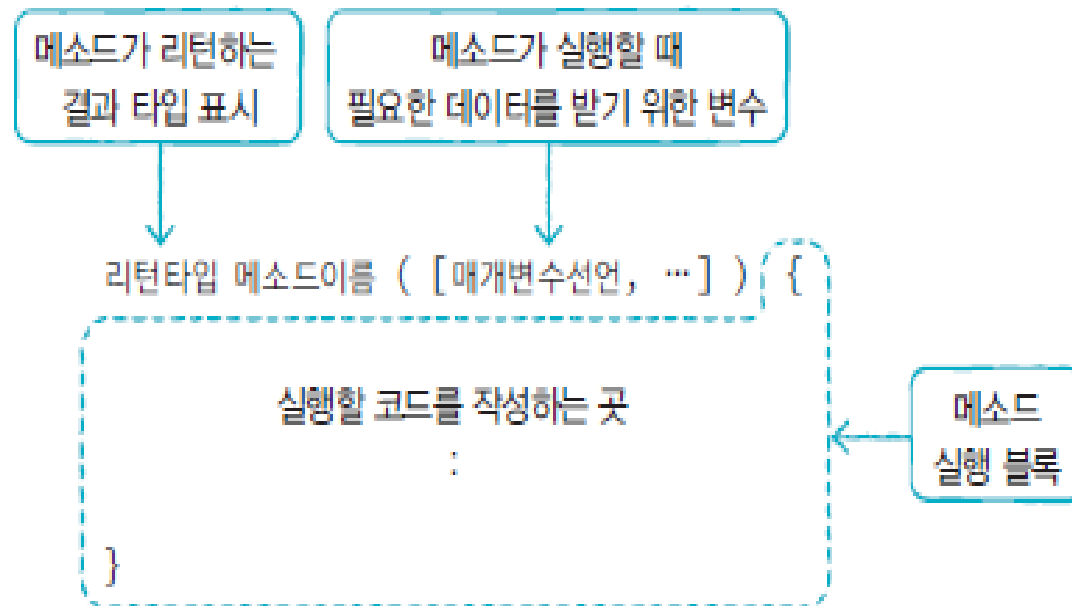
- 클래스에 생성자가 명시적으로 선언되었을 경우 반드시 선언된 생성자 호출하여 객체 생성

```
Car myCar = new Car("그랜저", "검정", 300);
```

# 메서드

## ❖ 메소드 선언부 (signature)

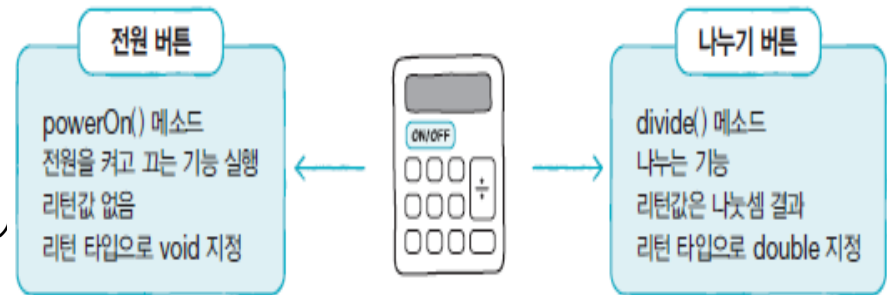
- 리턴 타입 : 메소드가 리턴하는 결과의 타입 표시
- 메소드 이름 : 메소드의 기능 드러나도록
- 식별자 규칙에 맞게 이름 지음
- 매개 변수 선언 : 메소드 실행할 때 필요한
- 데이터 받기 위한 변수 선언
- 메소드 실행 블록 : 실행할 코드 작성



# 메소드 선언

## ❖ 리턴 타입

- 메소드를 실행한 후의 결과값의 타입
- 리턴값 없을 수도 있음
- 리턴값 있는 경우 리턴 타입이 선언부에 명



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

- 리턴값 존재 여부에 따라 메소드 호출 방법 다름

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

```
int result = divide( 10, 20 ); //컴파일 에러
```

# 메소드 선언

## ❖ 매개 변수 선언

- 메소드 실행에 필요한 데이터를 외부에서 받아 저장할 목적

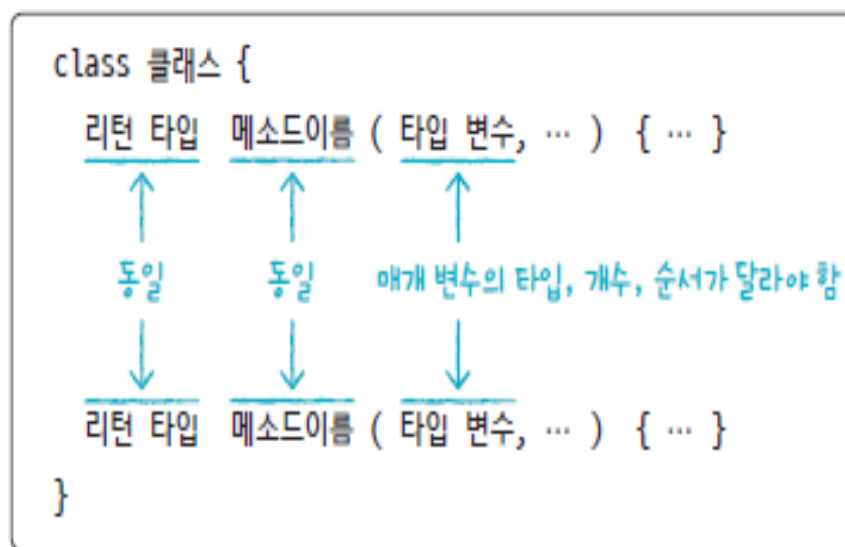
```
double divide( int x, int y ) { ... }
```

```
double result = divide( 10, 20 );
```

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

# 메소드 오버로딩

- ❖ 같은 이름의 메소드를 여러 개 선언
- ❖ 매개 값을 다양하게 받아 처리할 수 있도록 하기 위함
- ❖ 매개 변수의 타입, 개수, 순서 중 하나가 달라야



```
int plus(int x, int y) {  
    int result = x + y;  
    return result;  
}
```

```
double plus(double x, double y) {  
    double result = x + y;  
    return result;  
}
```

# 인스턴스 멤버와 정적 멤버

## ❖ 인스턴스 멤버

- 객체 마다 가지고 있는 멤버
  - 인스턴스 필드: 힙 영역의 객체 마다 가지고 있는 멤버, 객체마다 다른 데이터를 저장
  - 인스턴스 메소드: 객체가 있어야 호출 가능한 메소드
  - 클래스 코드(메소드 영역)에 위치하지만, 이해하기 쉽도록 객체 마다 가지고 있는 메소드라고 생각해도 됨

## ❖ 정적 멤버

- 객체와 상관없는 멤버, 클래스 코드(메소드 영역)에 위치
  - 정적 필드 및 상수: 객체 없이 클래스만으로도 사용 가능한 필드
  - 정적 메소드: 객체가 없이 클래스만으로도 호출 가능한 메소드

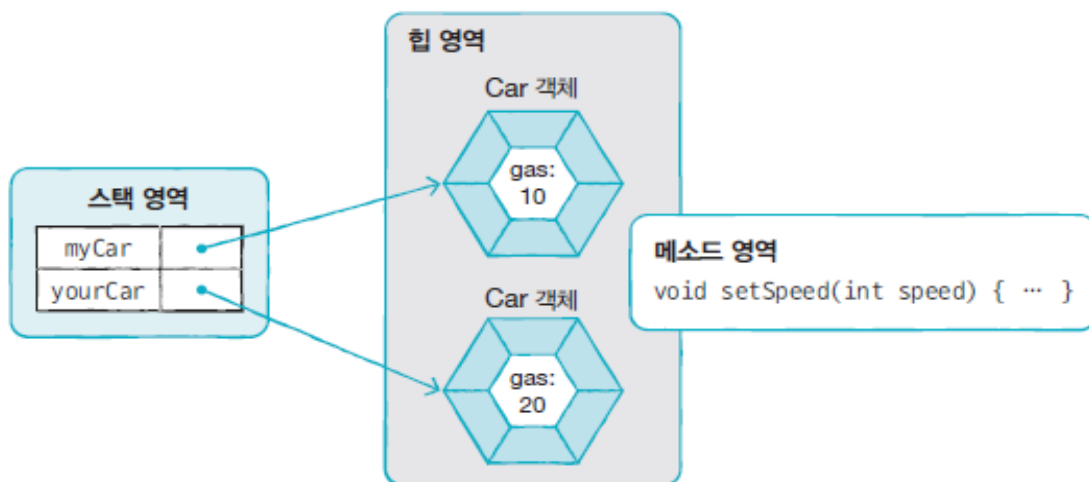
# 인스턴스 멤버와 this

## ❖ 인스턴스 (instance) 멤버:

- 객체를 생성한 후 사용할 수 있는 필드와 메소드

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

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





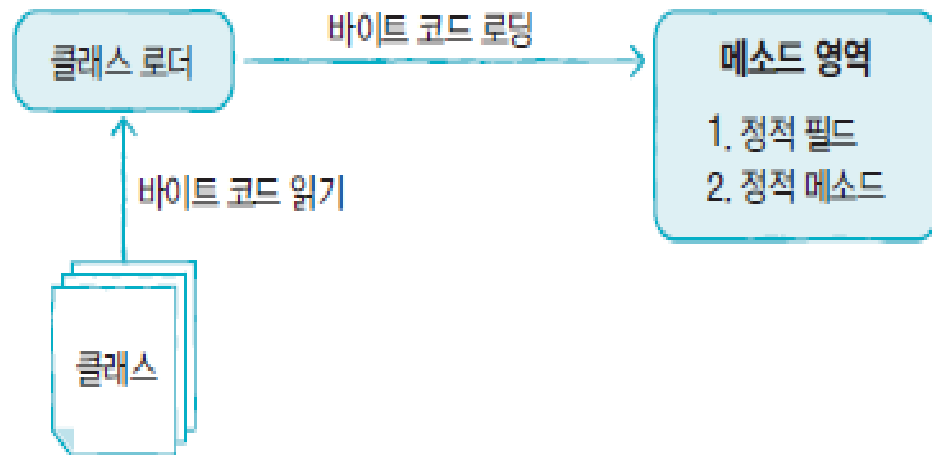
# 정적 멤버와 static

## ❖ 정적 (static) 멤버

- 클래스에 고정된 멤버로서 객체 생성하지 않고 사용할 수 있는 필드와 메소드

## ❖ 정적 멤버 선언

```
public class 클래스 {  
    //정적 필드  
    static 타입 필드 [= 초기값];  
  
    //정적 메소드  
    static 리턴 타입 메소드( 매개변수선언, ... ) { ... }  
}
```



# final 필드와 상수

## ❖ final 필드

- 초기값이 저장되면 최종값이 되어 프로그램 실행 도중 수정 불가
- final 필드의 초기값 주는 방법
  - 단순 값일 경우 필드 선언 시 초기화(주로 정적 필드(상수)일 경우)
  - 객체 생성 시 외부 데이터로 초기화 필요한 경우 생성자에서 초기화(주로 인스턴스 필드일 경우)
- 인스턴스 final 필드
  - 객체에 한번 초기화된 데이터를 변경 불가로 만들 경우: ex) 주민 번호

```
final 타입 필드 [= 초기값];
```

```
final String ssn; //생성자에서 초기화
```

- 정적 final 필드 (관례적으로 모두 대문자로 작성)
  - 불변의 값인 상수를 만들 경우: ex)

```
static final 타입 상수 = 초기값;
```

```
static final double PI = 3.14159;
```

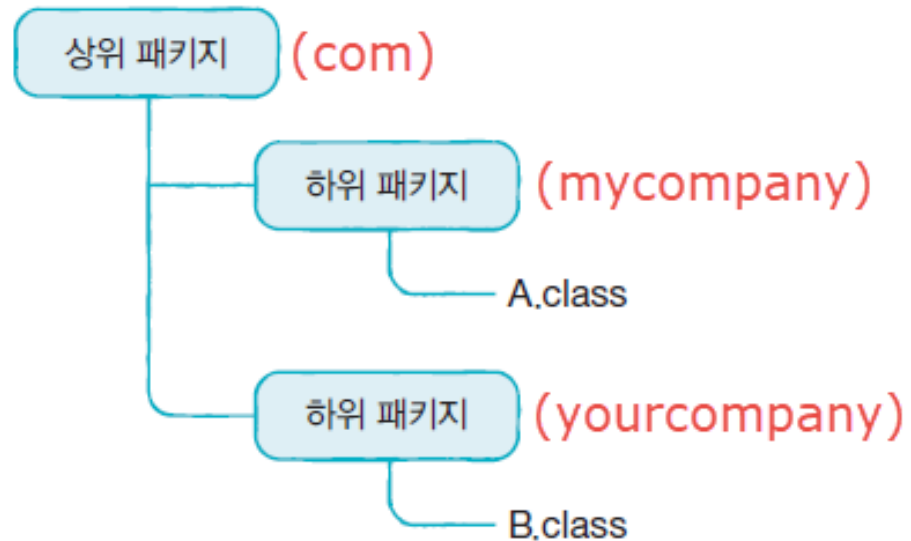
```
static final double EARTH_RADIUS = 6400;
```

```
static final double EARTH_AREA = 4 * Math.PI * EARTH_RADIUS * EARTH_RADIUS;
```

# 패키지

## ❖ 패키지

- 패키지의 물리적인 형태는 파일 시스템의 폴더
- 패키지는 클래스의 일부분으로, 클래스를 유일하게 만들어주는 식별자 역할
- 클래스 이름이 동일하더라도 패키지가 다르면 다른 클래스로 인식
- 클래스의 전체 이름은 패키지+클래스 사용해서 다음과 같이 표현
  - 상위패키지.하위패키지.클래스
  - com.mycompany.A
  - com.yourcompany.B



# 패키지 선언

## ❖ 패키지 선언

- 클래스 작성 시 해당 클래스가 어떤 패키지에 속할 것인지를 선언

```
package 상위패키지.하위패키지;
```

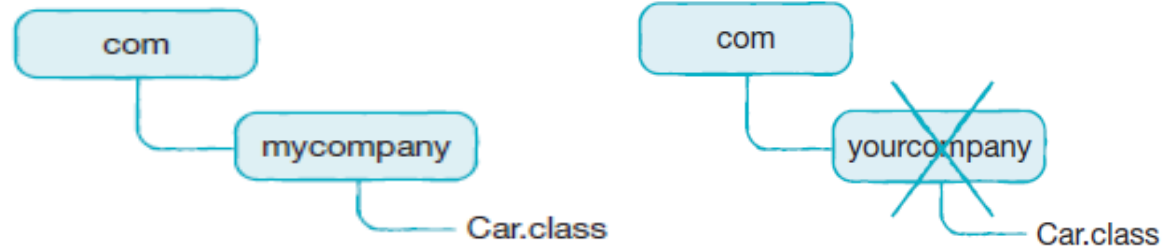
```
public class ClassName { ... }
```

```
package com.mycompany;
```

```
public class Car { ... }
```

### ■ 패키지 이름 규칙

- - 숫자로 시작 불가
- - \_ 및 \$ 제외한 특수문자 사용 불가
- - java로 시작하는 패키지는 자바 표준 API 에서만 사용하므로 사용 불가
- - 모두 소문자로 작성하는 것이 관례



# 패키지 선언

## ❖ import문

- 사용하고자 하는 클래스 또는 인터페이스가 다른 패키지에 소속된 경우
- 해당 패키지 클래스 또는 인터페이스 가져와 사용할 것임을 컴파일러에 통지

```
import 상위패키지.하위패키지.클래스이름;  
import 상위패키지.하위패키지.*;
```

- 패키지 선언과 클래스 선언 사이에 작성
- 하위 패키지는 별도로 import를 해야함

```
import com.hankook.*;  
import com.hankook.project.*;
```

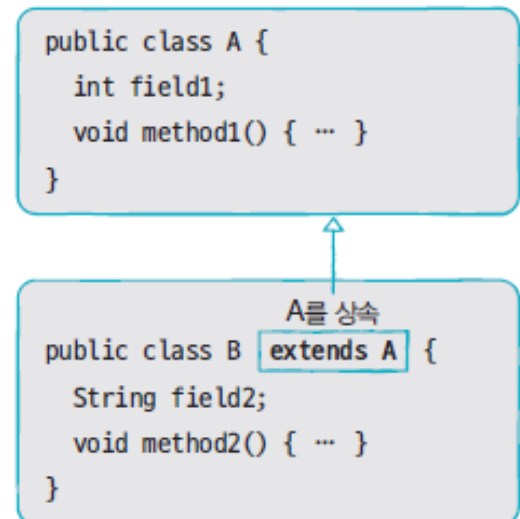
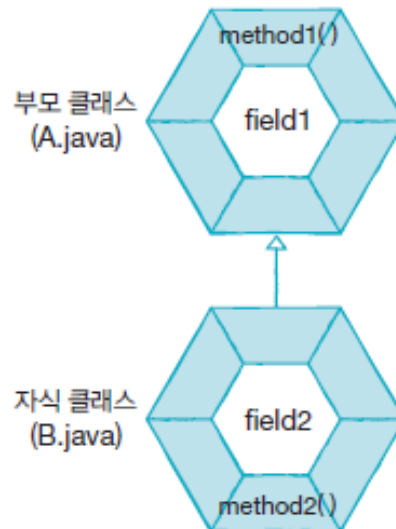
```
package com.mycompany;  
  
import com.hankook.Tire;  
[ 또는 import com.hankook.*; ]  
  
public class Car {  
    Tire tire = new Tire();  
}
```

- 다른 패키지에 동일한 이름의 클래스가 있을 경우  
import와 상관없이 클래스 전체 이름을 기술

# 상속

## ❖ 상속

- 이미 개발된 클래스를 재사용하여 새로운 클래스를 만들기에 중복되는 코드를 줄임
- 부모 클래스의 한번의 수정으로 모든 자식 클래스까지 수정되는 효과가 있어 유지보수 시간이 줄어듦



# 클래스 상속

## ❖ 클래스 상속

- 자식 클래스 선언 시 부모 클래스 선택
- extends 뒤에 부모 클래스 기술

```
class 자식클래스 extends 부모클래스 {  
    //필드  
    //생성자  
    //메소드  
}
```

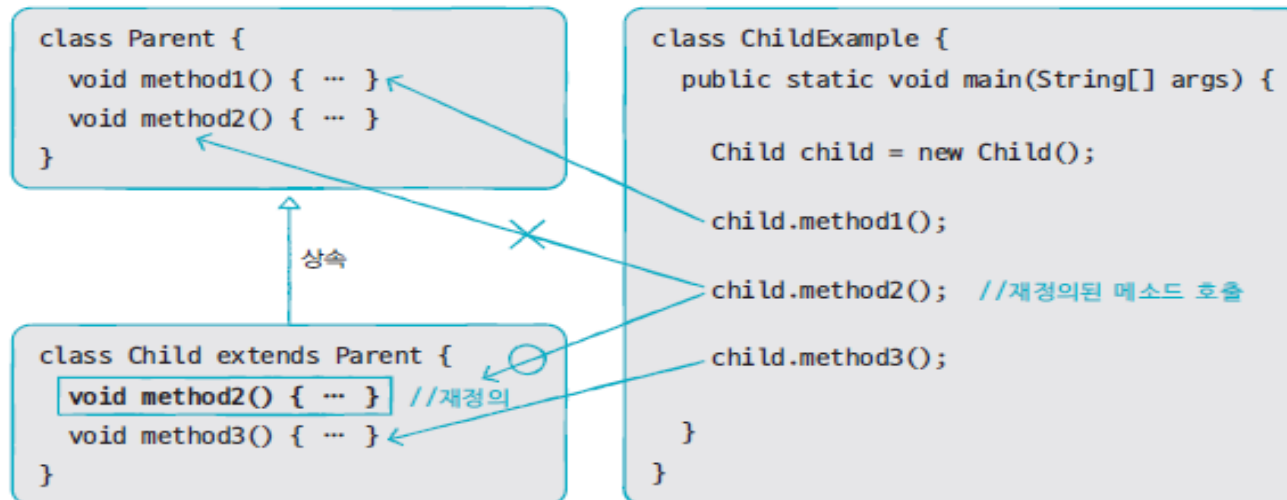
```
class SportsCar extends Car {  
}
```

- 여러 개의 부모 클래스 상속할 수 없음
- 부모 클래스에서 private 접근 제한 갖는 필드와 메소드는 상속 대상에서 제외
- 부모와 자식 클래스가 다른 패키지에 존재할 경우 default 접근 제한된 필드와 메소드 역시 제외

# 메소드 재정의

## ❖ 메소드 재정의 (오버라이딩 / Overriding)

- 부모 클래스의 메소드가 자식 클래스에서 사용하기에 부적합할 경우 자식 클래스에서 수정하여 사용
- 메소드 재정의 방법
  - 부모 메소드와 동일한 시그니처 가져야 함
  - 접근 제한 더 강하게 재정의할 수 없음
  - 새로운 예외를 throws 할 수 없음
- 메소드가 재정의될 경우 부모 객체 메소드가 숨겨지며,  
자식 객체에서 메소드 호출하면 재정의된 자식 메소드가 호출됨





# final 클래스와 final 메소드

## ❖ final 키워드

- 해당 선언이 최종 상태이며 수정될 수 없음을 의미
- 클래스 및 메소드 선언 시 final 키워드를 사용하면 상속과 관련됨

## ❖ 상속할 수 없는 final 클래스

- 부모 클래스가 될 수 없어 자식 클래스 만들 수 없음을 의미

```
public final class 클래스 { ... }
```

```
public final class String { ... }
```

```
public class NewString extends String { ... }
```

## ❖ 재정의할 수 없는 final 메소드

- 부모 클래스에 선언된 final 메소드는 자식 클래스에서 재정의 할 수 없음

```
public final 리턴타입 메소드( [매개변수, ...] ) { ... }
```

# 다형성

## ❖ 다형성

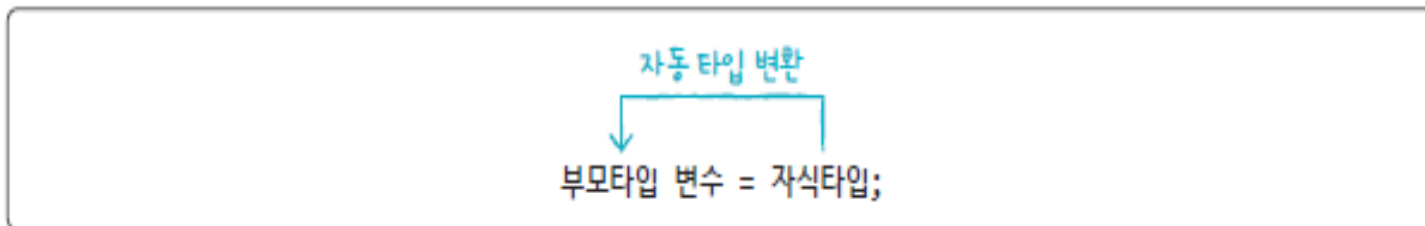
- 사용 방법은 동일하지만 다양한 객체 활용해 여러 실행결과가 나오도록 하는 성질
- 메소드 재정의와 타입 변환으로 구현



# 자동 타입 변환

## ❖ 자동 타입 변환 (promotion)

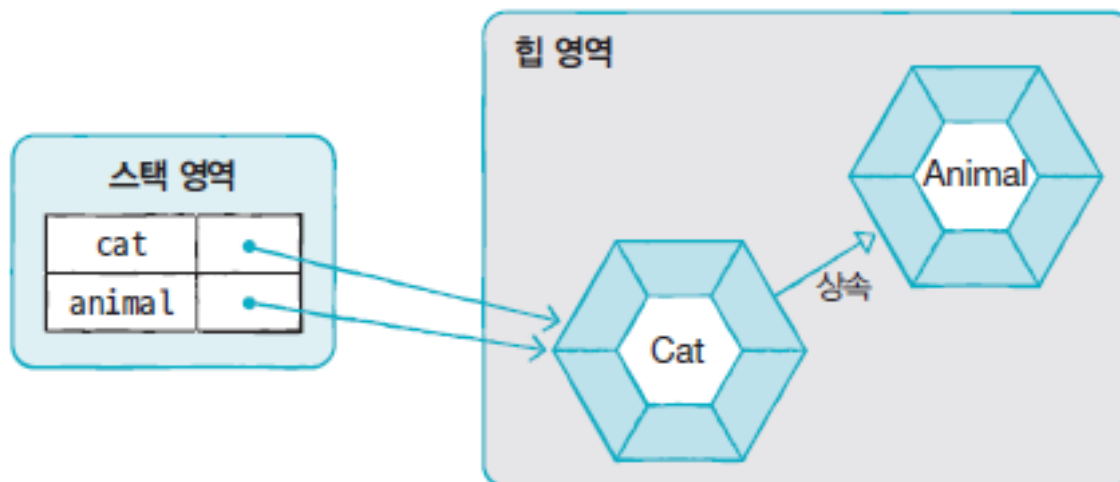
- 프로그램 실행 도중 자동으로 타입 변환 일어나는 것



```
Cat cat = new Cat();  
Animal animal = cat;
```

← Animal animal = new Cat(); 도 가능

This code block shows two lines of Java code. The first line creates a Cat object and assigns it to a Cat variable. The second line assigns the same Cat object to an Animal variable. A blue arrow points from the text 'Animal animal = new Cat(); 도 가능' (Animal animal = new Cat(); is also possible) to the second line of code, indicating that this is a valid operation due to automatic type promotion.



# 강제 타입 변환

## ❖ 강제 타입 변환 (casting)

- 부모 타입을 자식 타입으로 변환
  - 조건: 자식 타입이 부모 타입으로 자동 타입 변환한 후 다시 반대로 변환할 때 사용

자식타입 변수 = (자식타입) 부모타입;  
부모 타입을 자식 타입으로 변환

```
Parent parent = new Child(); //자동 타입 변환  
Child child = (Child) parent; //강제 타입 변환
```

```
class Parent {  
    String field1;  
    void method1() { ... }  
    void method2() { ... }  
}
```

상속

```
class Child extends Parent {  
    String field2;  
    void method3() { ... }  
}
```

```
class ChildExample {  
    public static void main(String[] args) {  
        Parent parent = new Child();  
        parent.field1 = "xxx";  
        parent.method1();  
        parent.method2();  
        parent.field2 = "yyy"; //불가능  
        parent.method3();      //불가능  
  
        Child child = (Child) parent;  
        child.field2 = "yyy"; //가능  
        child.method3();      //가능  
    }  
}
```

# 객체 타입 확인

## ❖ instanceof 연산자

- 어떤 객체가 어느 클래스의 인스턴스인지 확인
- 메소드 내 강제 타입 변환 필요한 경우
  - 타입 확인하지 않고 강제 타입 변환 시도 시 ClassCastException 발생할 수 있음
  - instanceof 연산자 통해 확인 후 안전하게 실행

```
boolean result = 좌항(객체) instanceof 우항(타입)
```

```
Parent parent = new Parent();
```

```
Child child = (Child) parent;    //강제 타입 변환을 할 수 없음
```

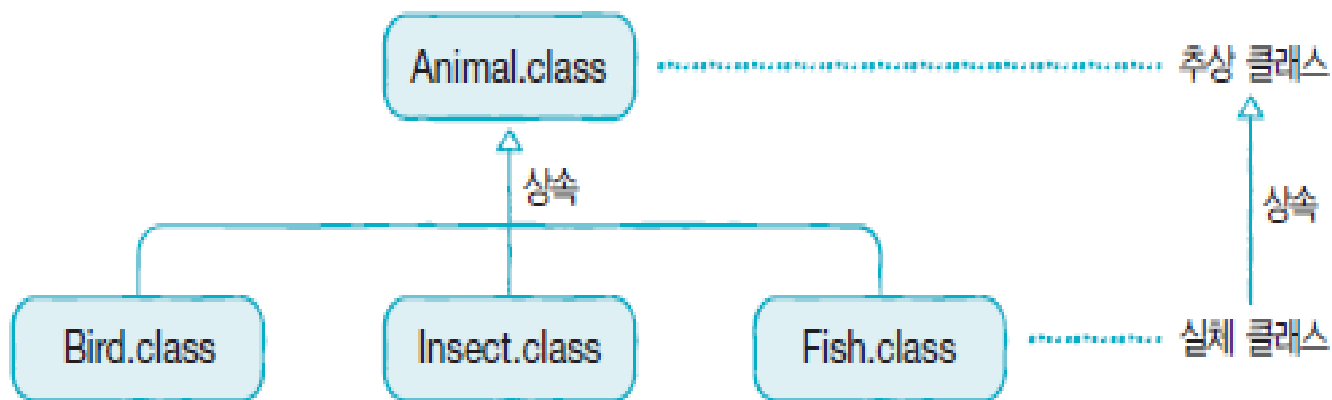
The diagram illustrates the instanceof operator. At the top, 'Parent 객체' and 'Child 객체' are shown with arrows pointing to the 'parent' parameter in the code below. The code is a method signature: `public void method(Parent parent) {`. Inside the method, there is an `if(parent instanceof Child) {` line. A blue arrow points from the `Child` in the instanceof check to the text 'Parent 매개 변수가 참조하는 객체가 Child인지 조사' (Check if the object referenced by the Parent parameter is a Child). Below the if statement is `Child child = (Child) parent;`, followed by closing braces `}` and `}`.

```
public void method(Parent parent) {  
    if(parent instanceof Child) {  
        Child child = (Child) parent;  
    }  
}
```

# 추상 클래스

## ❖ 추상 클래스

- 실체 클래스(객체 생성용 클래스)들의 공통적인 특성(필드, 메소드)을 추출하여 선언한 것
- 추상 클래스와 실체 클래스는 부모, 자식 클래스로서 상속 관계를 가짐



# 추상 클래스 선언

## ❖ 추상 클래스 선언

- abstract 키워드

- 상속 통해 자식 클래스만 만들 수 있게 만듦(부모로서의 역할만 수행)

```
public abstract class 클래스 {  
    //필드  
    //생성자  
    //메소드  
}
```

- 추상 클래스도 일반 클래스와 마찬가지로 필드, 생성자, 메소드 선언 할 수 있음
- 직접 객체를 생성할 수 없지만 자식 객체 생성될 때 객체화 됨.
  - 자식 생성자에서 super(...) 형태로 추상 클래스의 생성자 호출

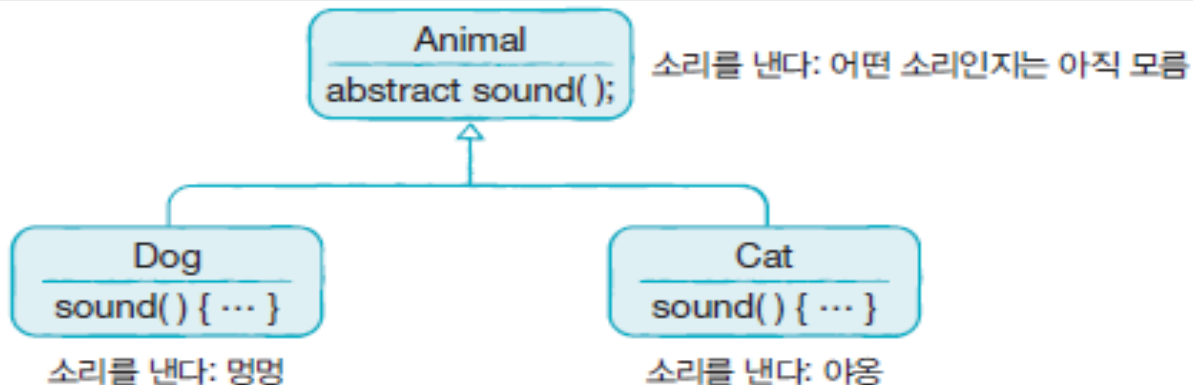
# 추상 메소드와 재정의

## ❖ 추상 메소드

- 메소드 선언만 통일하고 실행 내용은 실제 클래스마다 달라야 하는 경우
- abstract 키워드로 선언되고 중괄호가 없는 메소드
- 하위 클래스는 반드시 재정의해서 실행 내용을 채워야 함.

```
[public | protected] abstract 리턴타입 메소드이름(매개변수, ...);
```

```
public abstract class Animal {  
    public abstract void sound();  
}
```

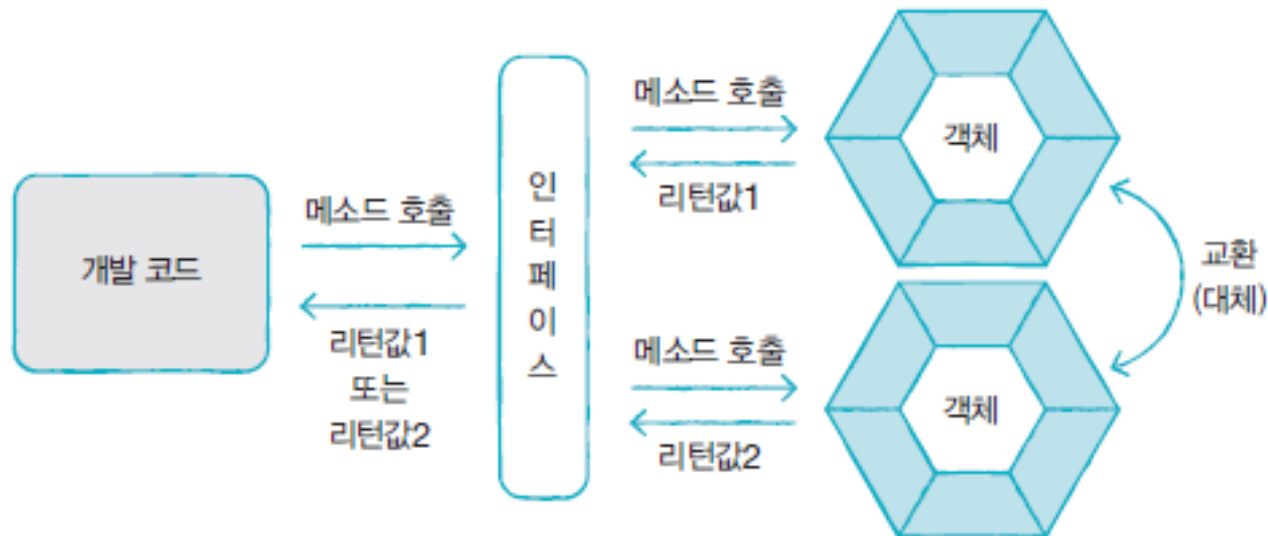




# 인터페이스

## ❖ 인터페이스 (interface)

- 개발 코드는 인터페이스를 통해서 객체와 서로 통신한다.
- 인터페이스의 메소드 호출하면 객체의 메소드가 호출된다.
- 개발 코드를 수정하지 않으면서 객체 교환이 가능하다.



# 인터페이스 선언

## ❖ 인터페이스 선언

- ~.java 형태 소스 파일로 작성 및 컴파일러 통해 ~.class 형태로 컴파일된다.
- 클래스와 물리적 파일 형태는 같으나 소스 작성 내용이 다르다.

```
[public] interface 인터페이스이름 { ... }
```

- 인터페이스는 객체로 생성할 수 없으므로 생성자 가질 수 없다.

```
interface 인터페이스이름 {  
    //상수  
    타입 상수이름 = 값;  
    //추상 메소드  
    타입 메소드이름(매개변수, ...);  
}
```

# 인터페이스 선언

## ❖ 상수 필드 (constant field) 선언

- 데이터를 저장할 인스턴스 혹은 정적 필드 선언 불가
- 상수 필드만 선언 가능

```
[public static final] 타입 상수이름 = 값;
```

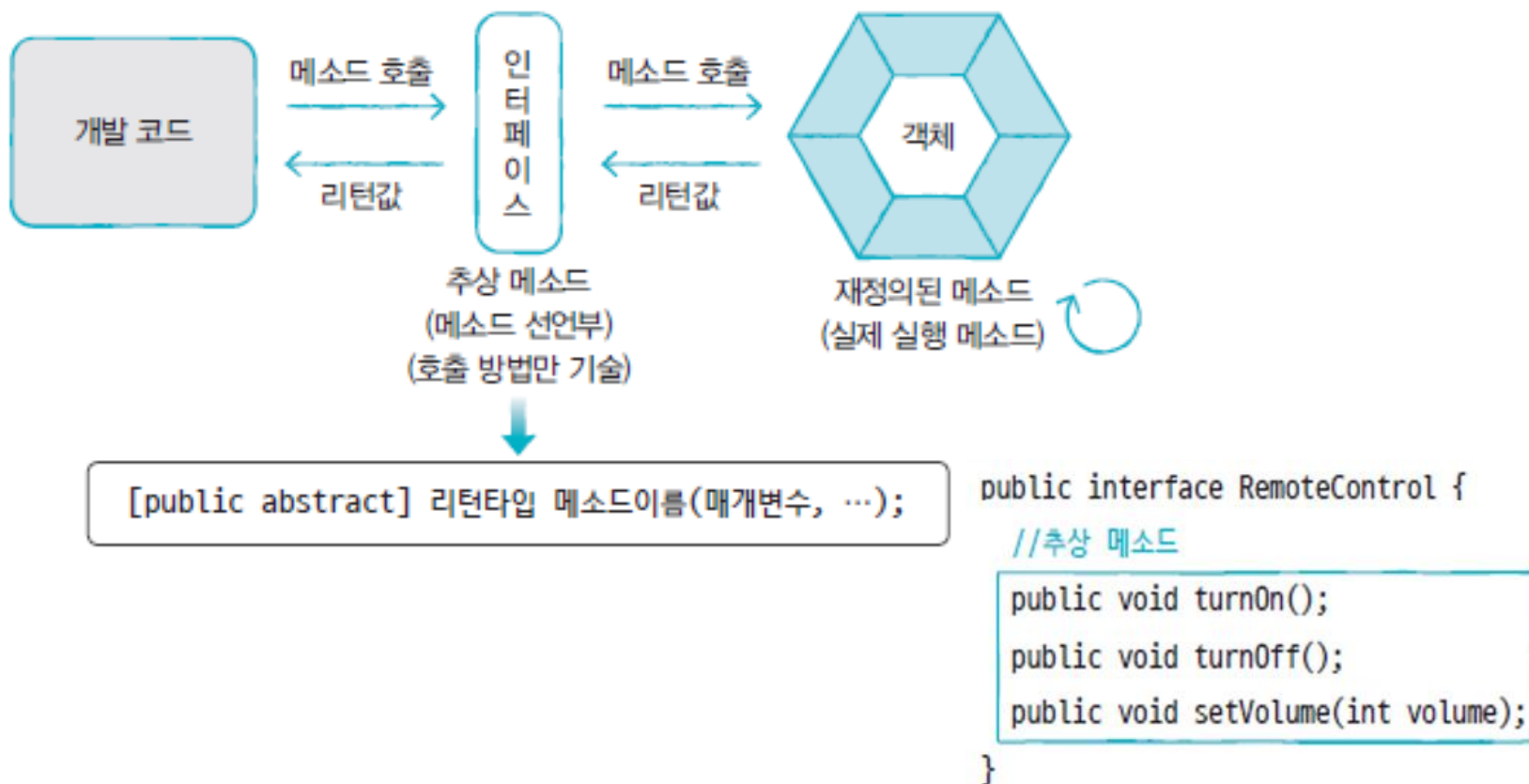
- 상수 이름은 대문자로 작성하되 서로 다른 단어로 구성되어 있을 경우 언더바(\_)로 연결

```
public interface RemoteControl {  
    public int MAX_VOLUME = 10;  
    public int MIN_VOLUME = 0;  
}
```

# 인터페이스 선언

## ❖ 추상 메소드 선언

- 인터페이스 통해 호출된 메소드는 최종적으로 객체에서 실행
- 인터페이스의 메소드는 실행 블록 필요 없는 추상 메소드로 선언



# 인터페이스 구현

## ❖ 구현 (implement) 클래스

- 인터페이스에서 정의된 추상 메소드를 재정의해서 실행내용을 가지고 있는 클래스
- 클래스 선언부에 implements 키워드 추가하고 인터페이스 이름 명시

```
public class 구현클래스이름 implements 인터페이스이름 {  
    //인터페이스에 선언된 추상 메소드의 실제 메소드 선언  
}
```

```
public class Television implements RemoteControl {
```

```
    //turnOn() 추상 메소드의 실제 메소드
```

```
    public void turnOn() {  
        System.out.println("TV를 켭니다.");  
    }
```

```
    //turnOff() 추상 메소드의 실제 메소드
```

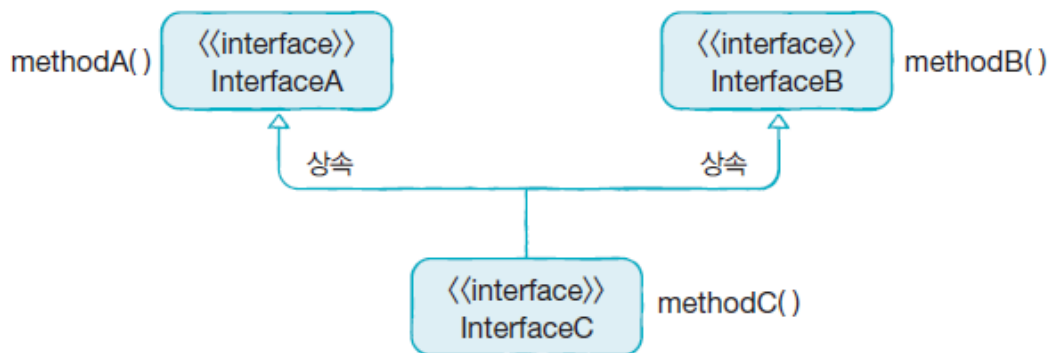
```
    public void turnOff() {  
        System.out.println("TV를 끕니다.");  
    }
```

# 인터페이스 상속

## ❖ 인터페이스 상속

- 인터페이스는 다중 상속을 할 수 있다.

```
public interface 하위인터페이스 extends 상위인터페이스1, 상위인터페이스2 { ... }
```



```
public interface InterfaceC extends InterfaceA, InterfaceB {
```

```
하위인터페이스 변수 = new 구현클래스(...);  
상위인터페이스1 변수 = new 구현클래스(...);  
상위인터페이스2 변수 = new 구현클래스(...);
```

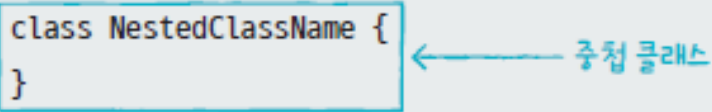
```
public class ImplementationC implements InterfaceC {  
  
    ImplementationC impl = new ImplementationC();  
  
    InterfaceC ic = impl;  
    InterfaceA ia = impl;  
    InterfaceB ib = impl;
```

# 중첩 타입

## ❖ 중첩 클래스 (nested class)

- 클래스 내부에 선언한 클래스
- 두 클래스의 멤버들을 서로 쉽게 접근하게 하고, 외부에는 불필요한 관계 클래스 감춤
- 코드 복잡성 줄임

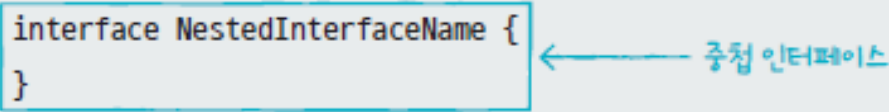
```
class ClassName {  
    class NestedClassName {  
    }  
}
```



## ❖ 중첩 인터페이스 (nested interface)

- 인터페이스 역시 클래스 내부에 선언 가능
- 해당 클래스와 긴밀한 관계 갖는 구현 클래스 만들기 위함

```
class ClassName {  
    interface NestedInterfaceName {  
    }  
}
```



# 익명 객체

## ❖ 익명 (anonymous) 객체

- 이름이 없는 객체
- 어떤 클래스를 상속하거나 인터페이스를 구현하여야 함

[상속]

```
class 클래스이름1 extends 부모클래스 { ... }  
부모클래스 변수 = new 클래스이름1();
```

[구현]

```
class 클래스이름2 implements 인터페이스 { ... }  
인터페이스 변수 = new 클래스이름2();
```

[상속]

```
부모클래스 변수 = new 부모클래스() { ... };
```

[구현]

```
인터페이스 변수 = new 인터페이스() { ... };
```



# 예외

## ❖ 예외 (Exception)

- 사용자의 잘못된 조작 또는 개발자의 잘못된 코딩으로 인해 발생하는 프로그램 오류
- 예외 처리 프로그램 통해 정상 실행상태 유지 가능
- 예외 발생 가능성이 높은 코드 컴파일할 때 예외 처리 유무 확인

# 예외와 예외 클래스

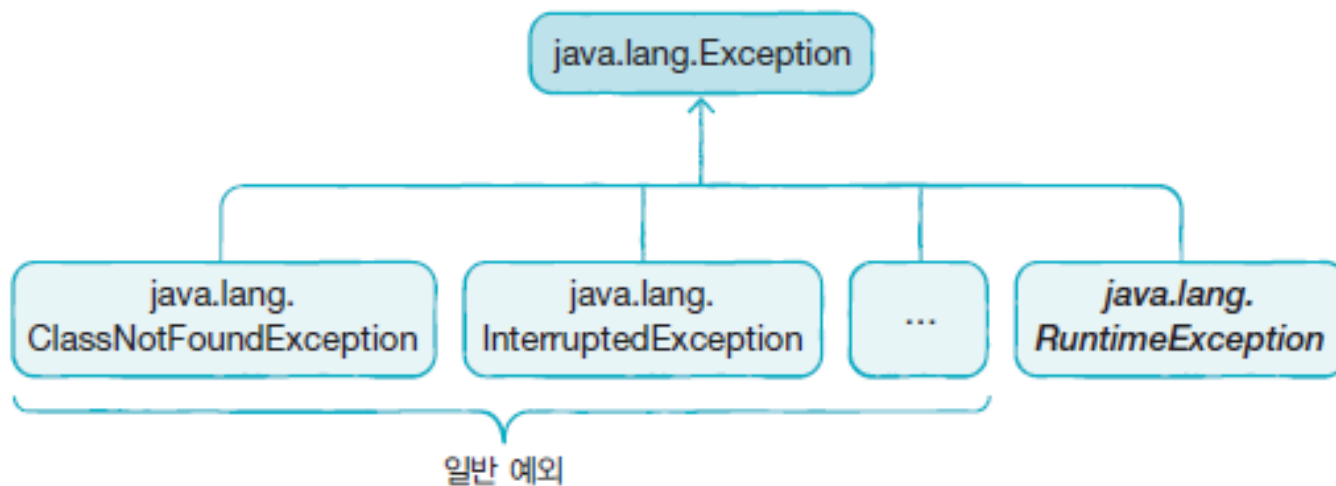
## ❖ 일반 예외 (exception)

- 컴파일러 체크 예외
- 자바 소스 컴파일 과정에서 해당 예외 처리 코드 있는지 검사하게 됨

## ❖ 실행 예외 (runtime exception)

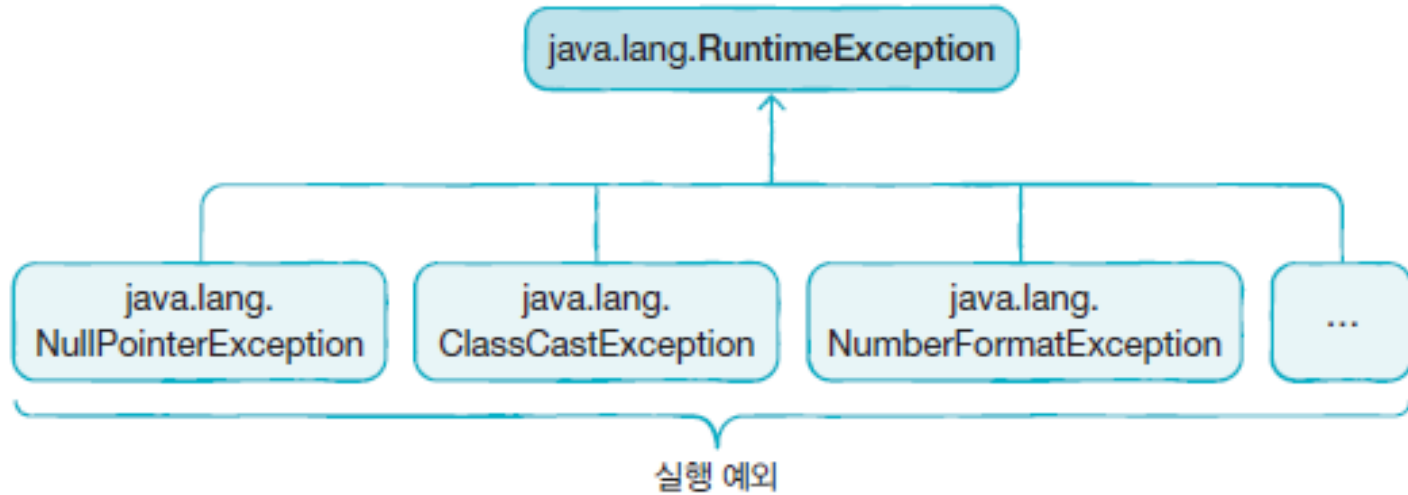
- 컴파일러 런 체크 예외
- 실행 시 예측할 수 없이 갑자기 발생하기에 컴파일 과정에서 예외처리코드 검사하지 않음

## ❖ 자바에서는 예외를 클래스로 관리



## 예외와 예외 클래스

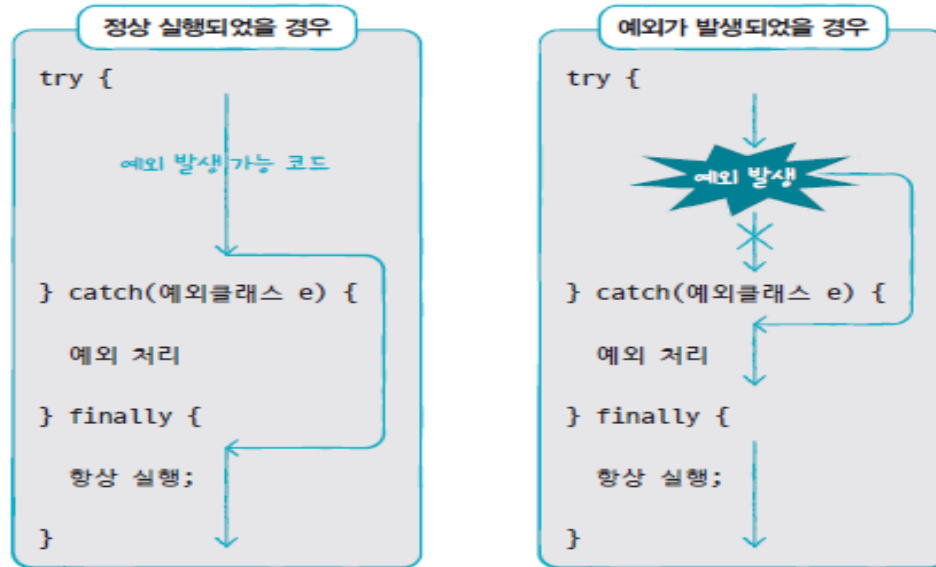
- RuntimeException 클래스 기준으로 일반 및 실행 예외 클래스 구분



# 예외 처리 코드

## ❖ try-catch-finally 블록

- 생성자 및 메소드 내부에서 작성되어 일반예외와 실행예외가 발생할 경우 예외 처리 가능하게 함



- try 블록에는 예외 발생 가능 코드가 위치
- try 블록 코드가 예외발생 없이 정상실행되면 catch 블록의 코드는 실행되지 않고 finally 블록의 코드를 실행. try 블록의 코드에서 예외가 발생한다면 실행 멈추고 catch 블록으로 이동하여 예외 처리 코드 실행. 이후 finally 블록 코드 실행
- finally 블록은 생략 가능하며, 예외와 무관하게 항상 실행할 내용이 있을 경우에만 작성.

# 예외 떠넘기기

## ❖ throws 키워드

- 메소드 선언부 끝에 작성되어 메소드에서 처리하지 않은 예외를 호출한 곳으로 넘기는 역할
- throws 키워드 뒤에는 떠넘길 예외 클래스를 쉼표로 구분하여 나열

```
리턴타입 메소드이름(매개변수,...) throws 예외클래스1, 예외클래스2, ... {  
}
```

```
리턴타입 메소드이름(매개변수,...) throws Exception {  
}
```

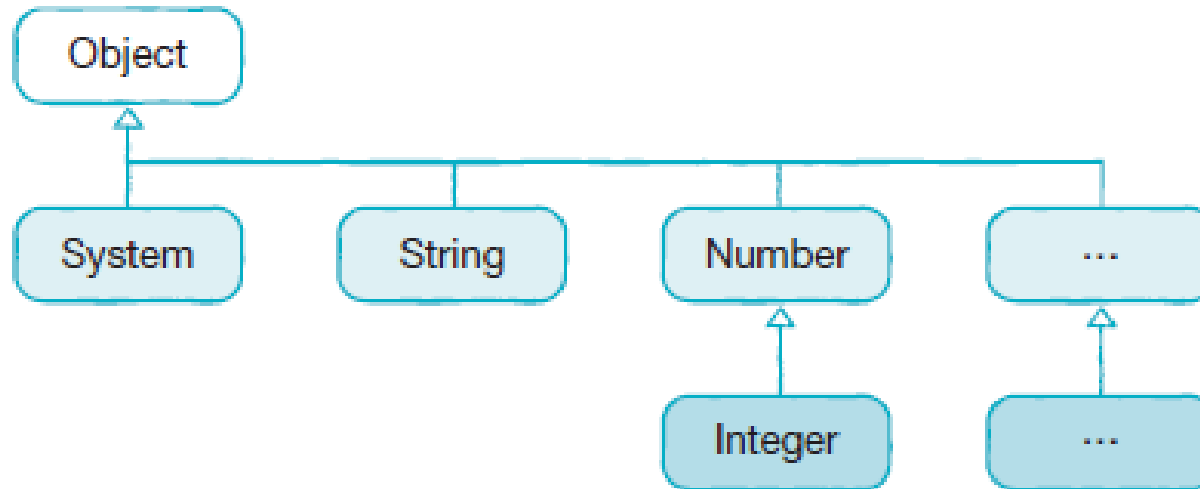
# java.lang 패키지의 클래스

## ❖ java.lang 패키지의 주요 클래스와 용도

클래스		용도
Object		- 자바 클래스의 최상위 클래스로 사용
System		- 표준 입력 장치(키보드)로부터 데이터를 입력받을 때 사용 - 표준 출력 장치(모니터)로 출력하기 위해 사용 - 자바 가상 기계를 종료할 때 사용 - 쓰레기 수집기를 실행 요청할 때 사용
Class		- 클래스를 메모리로 로딩할 때 사용
String		- 문자열을 저장하고 여러 가지 정보를 얻을 때 사용
Wrapper	Byte, Short, Character Integer, Float, Double Boolean, Long	- 기본 타입의 데이터를 갖는 객체를 만들 때 사용 - 문자열을 기본 타입으로 변환할 때 사용 - 입력값 검사에 사용
Math		- 수학 함수를 이용할 때 사용

# Object 클래스

❖ 모든 클래스는 Object 클래스의 자식이거나 자손 클래스



# System 클래스

- ❖ System 클래스의 모든 필드와 메소드는 정적 필드 및 메소드로 구성
- ❖ 프로그램 종료 (exit())
  - exit() 메소드 호출하여 JVM을 강제 종료
  - exit() 메소드가 지정하는 int 매개값을 종료 상태값이라 함
- ❖ 현재 시각 읽기 ( currentTimeMillis(), nanoTime() )
  - System 클래스의 currentTimeMillis() 및 nanoTime() 메서드로 각각 1/1000초 및  $10/10^{10}$  단위 long 값 반환

```
long time = System.currentTimeMillis();  
long time = System.nanoTime();
```



# Class 클래스

- ❖ 자바는 클래스와 인터페이스의 메타 데이터를 Class 클래스로 관리
- ❖ Class 객체 얻기 (getClass(), forName())

## 클래스로부터 얻는 방법

- ① `Class clazz = 클래스이름.class`
- ② `Class clazz = Class.forName("패키지...클래스이름")`

## 객체로부터 얻는 방법

- ③ `Class clazz = 참조변수.getClass( );`

# String 클래스

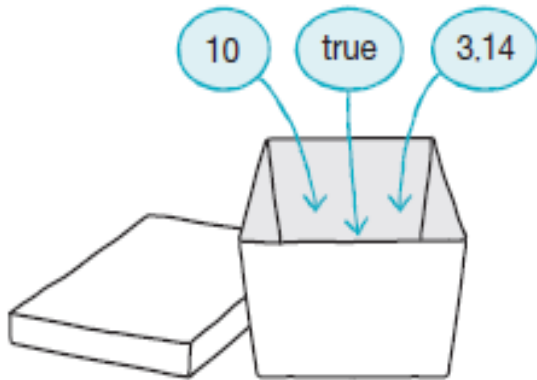
## ❖ String 메소드

리턴 타입	메소드 이름(매개 변수)	설명
char	charAt(int index)	특정 위치의 문자를 리턴합니다.
boolean	equals(Object anObject)	두 문자열을 비교합니다.
byte[]	getBytes()	byte[]로 리턴합니다.
byte[]	getBytes(Charset charset)	주어진 문자셋으로 인코딩한 byte[]로 리턴합니다.
int	indexOf(String str)	문자열 내에서 주어진 문자열의 위치를 리턴합니다.
int	length()	총 문자의 수를 리턴합니다.
String	replace(CharSequence target, CharSequence replacement)	target 부분을 replacement로 대치한 새로운 문자열을 리턴합니다.
String	substring(int beginIndex)	beginIndex 위치에서 끝까지 잘라낸 새로운 문자열을 리턴합니다.
String	substring(int beginIndex, int endIndex)	beginIndex 위치에서 endIndex 전까지 잘라낸 새로운 문자열을 리턴합니다.
String	toLowerCase()	알파벳 소문자로 변환한 새로운 문자열을 리턴합니다.
String	toUpperCase()	알파벳 대문자로 변환한 새로운 문자열을 리턴합니다.
String	trim()	앞뒤 공백을 제거한 새로운 문자열을 리턴합니다.
String	valueOf(int i) valueOf(double d)	기본 타입 값을 문자열로 리턴합니다.

# Wrapper(포장) 클래스

## ❖ 포장 객체

- 기본 타입의 값을 내부에 두고 포장
- 포장하고 있는 기본 타입 값은 외부에서 변경할 수 없음
- byte, char, short, int, long, float, double, boolean 기본 타입 값 갖는 객체



기본 타입	포장 클래스
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

# Wrapper(포장) 클래스

## ❖ Boxing과 Unboxing

- 박싱 : 기본 타입의 값을 포장 객체로 만드는 과정
- 언박싱 : 포장 객체에서 기본 타입의 값을 얻어내는 과정

기본 타입의 값을 줄 경우	문자열을 줄 경우
Byte obj = new Byte(10);	Byte obj = new Byte("10");
Character obj = new Character('가');	없음
Short obj = new Short(100);	Short obj = new Short("100");
Integer obj = new Integer(1000);	Integer obj = new Integer("1000");
Long obj = new Long(10000);	Long obj = new Long("10000");
Float obj = new Float(2.5F);	Float obj = new Float("2.5F");
Double obj = new Double(3.5);	Double obj = new Double("3.5");
Boolean obj = new Boolean(true);	Boolean obj = new Boolean("true");

# Math 클래스

## ❖ Math 클래스

- 수학 계산에 사용

메소드	설명	예제 코드	리턴값
<code>int abs(int a)</code> <code>double abs(double a)</code>	절대값	<code>int v1 = Math.abs(-5);</code> <code>double v2 = Math.abs(-3.14);</code>	<code>v1 = 5</code> <code>v2 = 3.14</code>
<code>double ceil(double a)</code>	올림값	<code>double v3 = Math.ceil(5.3);</code> <code>double v4 = Math.ceil(-5.3);</code>	<code>v3 = 6.0</code> <code>v4 = -5.0</code>
<code>double floor(double a)</code>	버림값	<code>double v5 = Math.floor(5.3);</code> <code>double v6 = Math.floor(-5.3);</code>	<code>v5 = 5.0</code> <code>v6 = -6.0</code>
<code>int max(int a, int b)</code> <code>double max(double a, double b)</code>	최대값	<code>int v7 = Math.max(5, 9);</code> <code>double v8 = Math.max(5.3, 2.5);</code>	<code>v7 = 9</code> <code>v8 = 5.3</code>
<code>int min(int a, int b)</code> <code>double min(double a, double b)</code>	최소값	<code>int v9 = Math.min(5, 9);</code> <code>double v10 = Math.min(5.3, 2.5);</code>	<code>v9 = 5</code> <code>v10 = 2.5</code>
<code>double random()</code>	랜덤값	<code>double v11 = Math.random();</code>	<code>0.0 &lt;= v11 &lt; 1.0</code>
<code>double rint(double a)</code>	가까운 정수의 실수값	<code>double v12 = Math.rint(5.3);</code> <code>double v13 = Math.rint(5.7);</code>	<code>v12 = 5.0</code> <code>v13 = 6.0</code>
<code>long round(double a)</code>	반올림값	<code>long v14 = Math.round(5.3);</code> <code>long v15 = Math.round(5.7);</code>	<code>v14 = 5</code> <code>v15 = 6</code>

# Date 클래스

## ❖ Date 클래스

- 날짜를 표현하는 클래스
- Date는 객체 간 날짜 정보 주고받을 때 매개 변수나 리턴 타입으로 주로 사용

```
Date now = new Date();
```

- 원하는 날짜 형식의 문자열 얻기 위해 java.text 패키지의 SimpleDateFormat 클래스와 함께 사용

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy년 MM월 dd일 hh시 mm분 ss초");
```

- format() 메소드 호출

```
String strNow = sdf.format(now);
```

# Calendar 클래스

## ❖ Calendar 클래스

- 추상 클래스이므로 new 연산자 사용하여 인스턴스 생성 불가
- getInstance() 메소드 이용하여 현재 운영체제에 설정된 시간대 기준으로 한 Calendar 하위 객체 얻을 수 있음

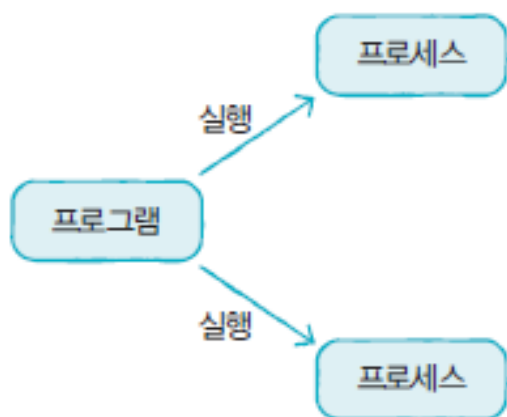
```
Calendar now = Calendar.getInstance();
```

```
int year    = now.get(Calendar.YEAR);        //연도를 리턴
int month   = now.get(Calendar.MONTH) + 1;   //월을 리턴
int day     = now.get(Calendar.DAY_OF_MONTH); //일을 리턴
int week    = now.get(Calendar.DAY_OF_WEEK); //요일을 리턴
int amPm    = now.get(Calendar.AM_PM);       //오전/오후를 리턴
int hour    = now.get(Calendar.HOUR);        //시를 리턴
int minute  = now.get(Calendar.MINUTE);      //분을 리턴
int second  = now.get(Calendar.SECOND);      //초를 리턴
```

# 시작하기 전에

## ❖ 프로세스 (process)

- 실행 중인 하나의 애플리케이션
- 애플리케이션이 실행되면 운영체제로부터 실행에 필요한 메모리 할당 받아 코드를 실행함
- 멀티 프로세스 역시 가능함



작업 관리자 (Task Manager) - 탭: 프로세스

이름	상태	9% CPU	24% 메모리
애플리케이션 (3)			
메모장		0.4%	2.2MB
메모장		0.1%	2.2MB
작업 관리자		1.3%	34.2MB
백그라운드 프로세스 (84)			
AcroTray(32비트)		0%	1.3MB
Activation Licensing Service(32...		0%	1.6MB
Adobe Acrobat Update Service(...		0%	0.9MB

간단히(D) | 작업 끝내기(E)



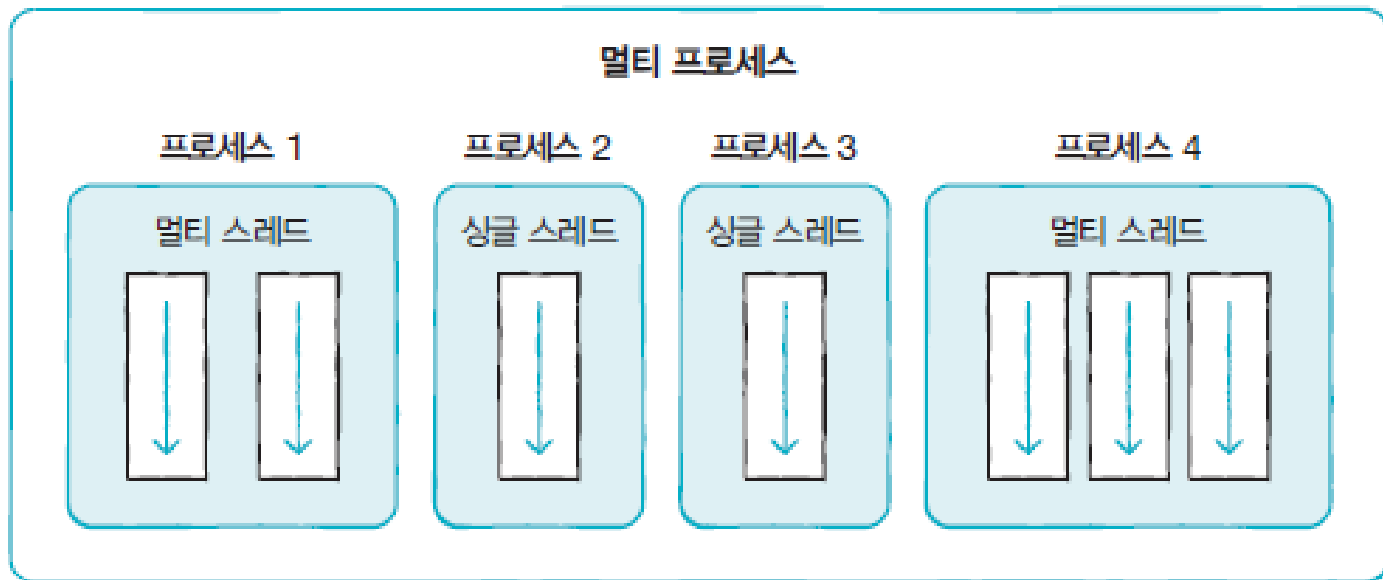
# 스레드

## ❖ 스레드 (thread)

- 한 가지 작업을 실행하기 위해 순차적으로 실행할 코드를 이어놓은 것
- 하나의 스레드는 하나의 코드 실행 이름

## ❖ 멀티 스레드 (multi thread)

- 하나의 프로세스로 두 가지 이상의 작업을 처리
- 데이터 분할하여 병렬로 처리하거나 다수 클라이언트 요청 처리하는 서버 개발하는 등의 용도
- 한 스레드가 예외 발생시킬 경우 프로세스 자체가 종료될 수 있음



# 메인 스레드

## ❖ 메인 스레드 (main thread)

- 모든 자바 애플리케이션은 메인 스레드가 main() 메소드 실행하면서 시작됨
- main() 메소드의 첫 코드부터 아래로 순차적으로 실행

```
public static void main(String[] args) {  
    String data = null;  
    if(...) {  
    }  
    while(...) {  
    }  
    System.out.println("...");  
}
```

코드의 실행 흐름 → 스레드

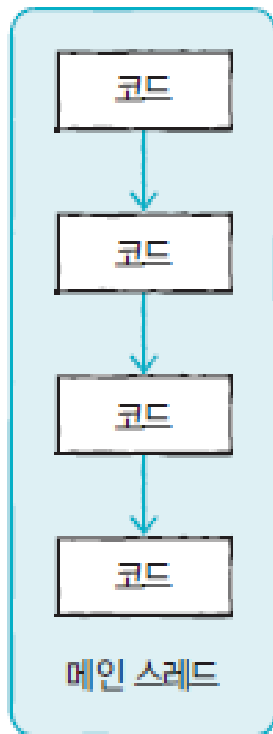


- 필요에 따라 작업 스레드들 만들어 병렬로 코드 실행 가능
- 멀티 스레드 애플리케이션에서는 실행 중인 스레드 하나라도 있으면 프로세스 종료되지 않음

# 메인 스레드

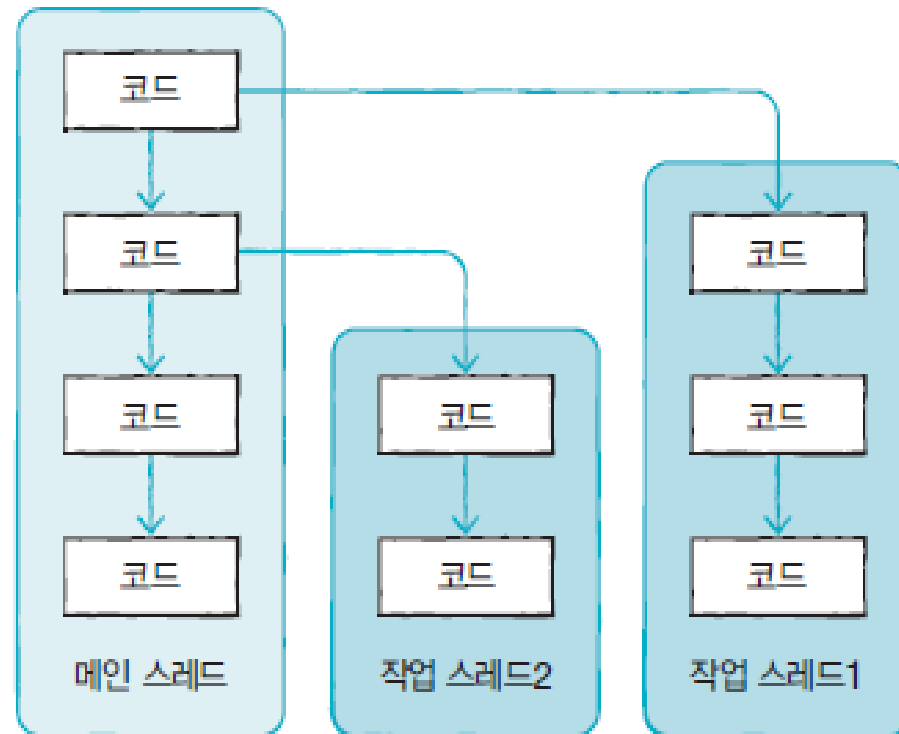
싱글 스레드 애플리케이션

프로세스



멀티 스레드 애플리케이션

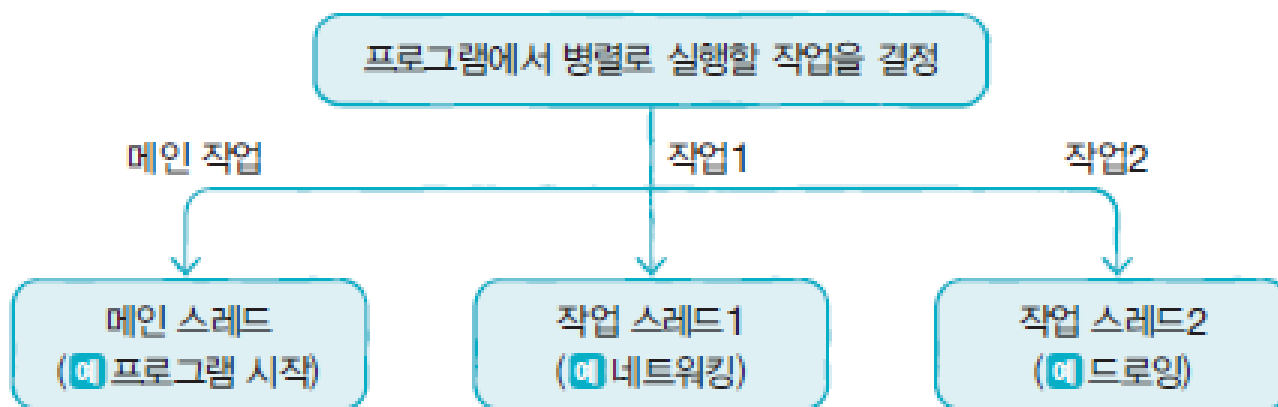
프로세스



# 작업 스레드 생성과 실행

## ❖ 작업 스레드

- 멀티 스레드로 실행하는 애플리케이션 개발하려면 몇 개의 작업을 병렬로 실행할지 우선 결정한 뒤 각 작업별로 스레드 생성해야
- 작업 스레드 역시 객체로 생성되므로 클래스 필요
  - Thread 클래스 상속하여 하위 클래스 만들어 사용할 수 있음



# 작업 스레드 생성과 실행

## ❖ Thread 클래스로부터 직접 생성

- Runnable을 매개값으로 갖는 생성자 호출


```
Thread thread = new Thread(Runnable target);
```

- 구현 객체 만들어 대입 필요

```
class Task implements Runnable {  
    public void run() {  
        스레드가 실행할 코드;  
    }  
}
```

- 구현 객체 매개값으로 Thread 생성자 호출하면 작업 스레드 생성됨

```
Runnable task= new Task();  
Thread thread = new Thread(task);
```



# 작업 스레드 생성과 실행

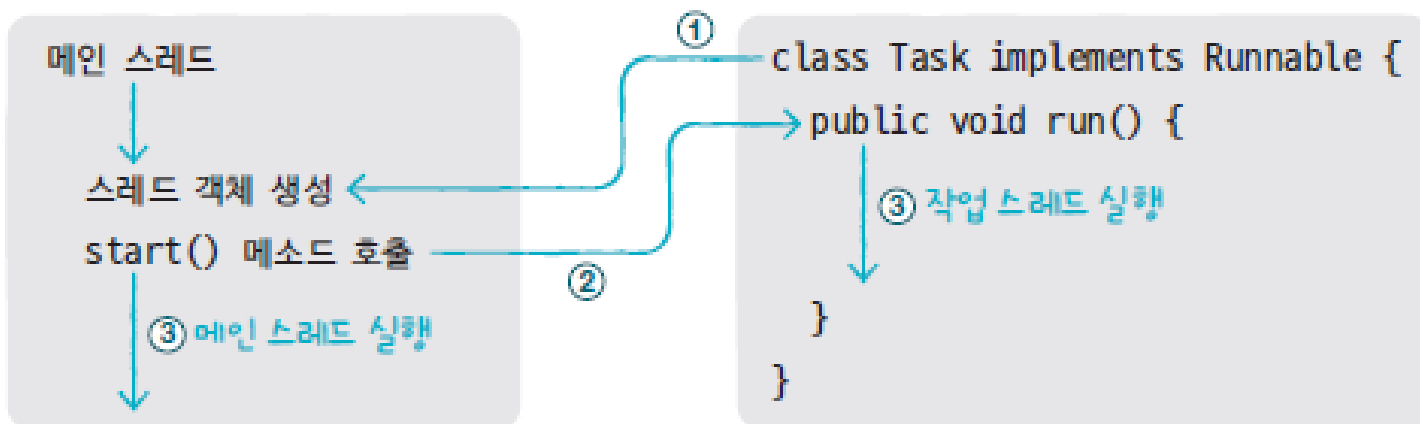
- Runnable 익명 객체를 매개값으로 사용하여 Thread 생성자 호출할 수도 있음

```
Thread thread = new Thread( new Runnable() {  
    public void run() {  
        스레드가 실행할 코드;  
    }  
});
```

← 익명 구현 객체

- start() 메소드 호출하면 작업 스레드 실행

```
thread.start();
```



# 작업 스레드 생성과 실행

## ❖ Thread 하위 클래스로부터 생성

- Thread의 하위 클래스로 작업 스레드를 정의하면서 작업 내용을 포함
- 작업 스레드 클래스 정의하는 법

```
public class WorkerThread extends Thread {  
    @Override  
    public void run() {  
        스레드가 실행할 코드;  
    }  
}  
Thread thread = new WorkerThread();
```

← run() 메소드 재정의

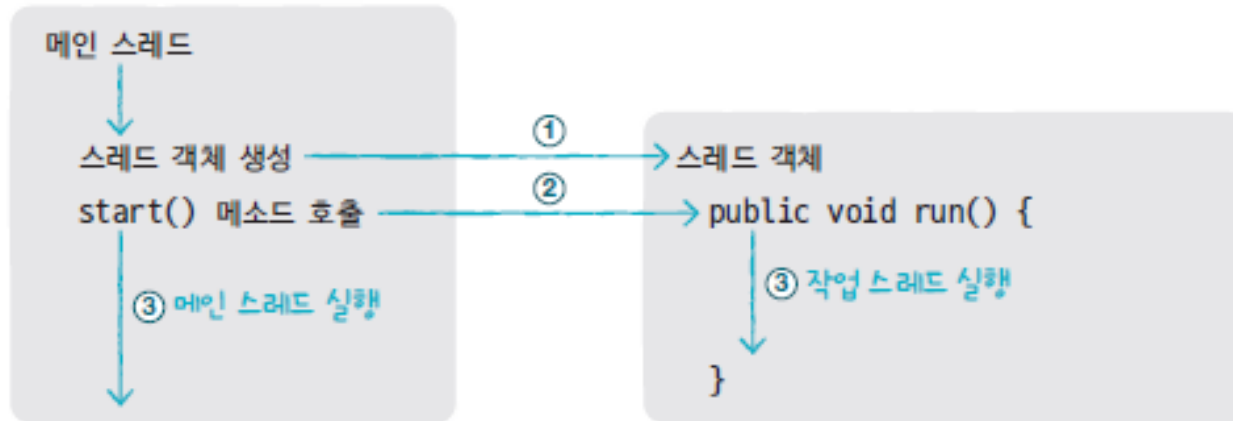
```
Thread thread = new Thread() {  
    public void run() {  
        스레드가 실행할 코드;  
    }  
};
```

← 익명 자식 객체

# 작업 스레드 생성과 실행

- 작업 스레드 객체 생성 후 start() 메소드 호출하면 run() 메소드가 실행

```
thread.start();
```

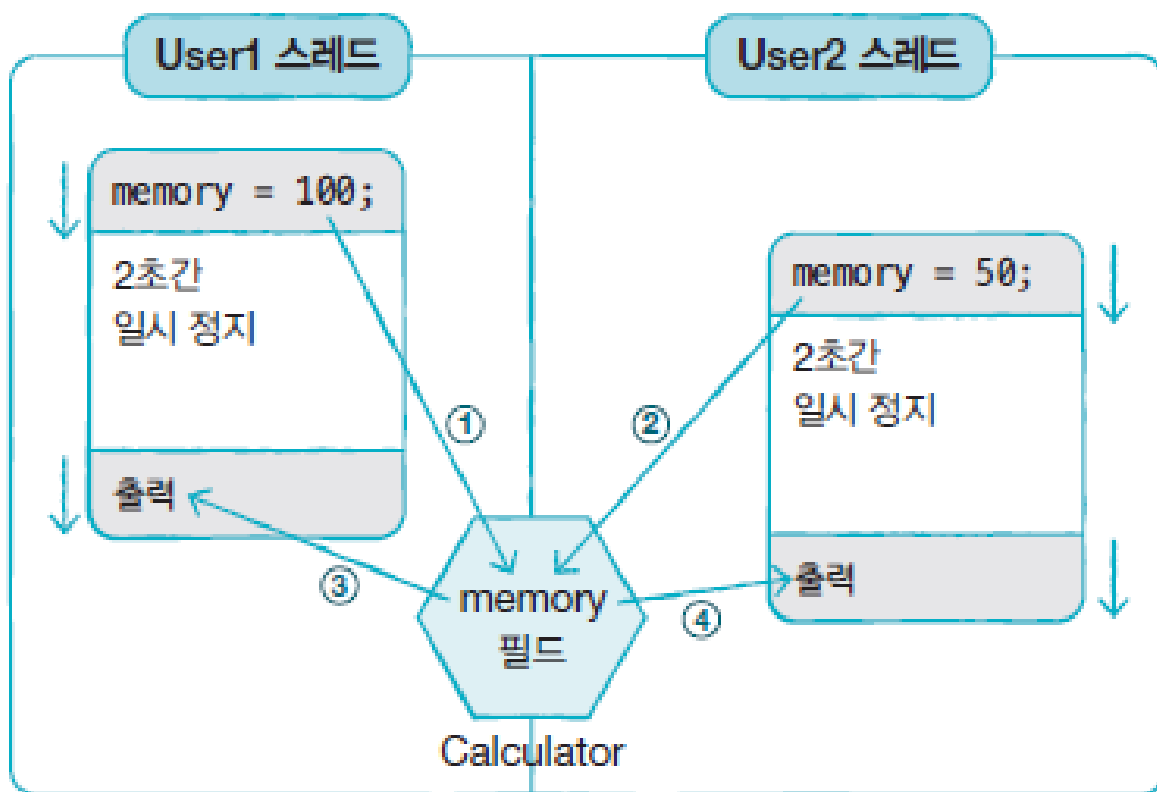




# 동기화 메소드

## ❖ 공유 객체를 사용할 때 주의할 점

- 멀티 스레드 프로그램에서 스레드들이 객체 공유해서 작업해야 하는 경우 의도했던 것과 다른 결과 나올 수 있음

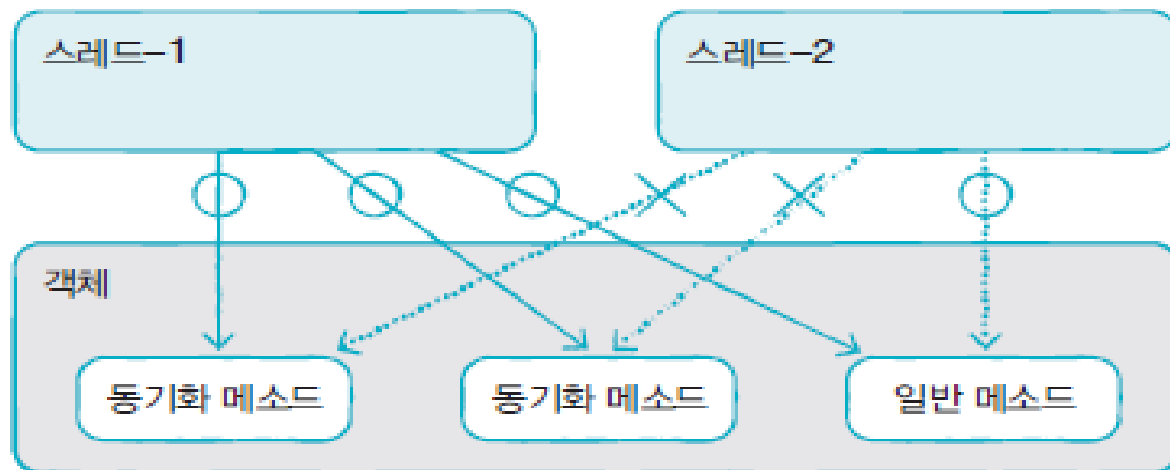


# 동기화 메소드

## ❖ 동기화 메소드

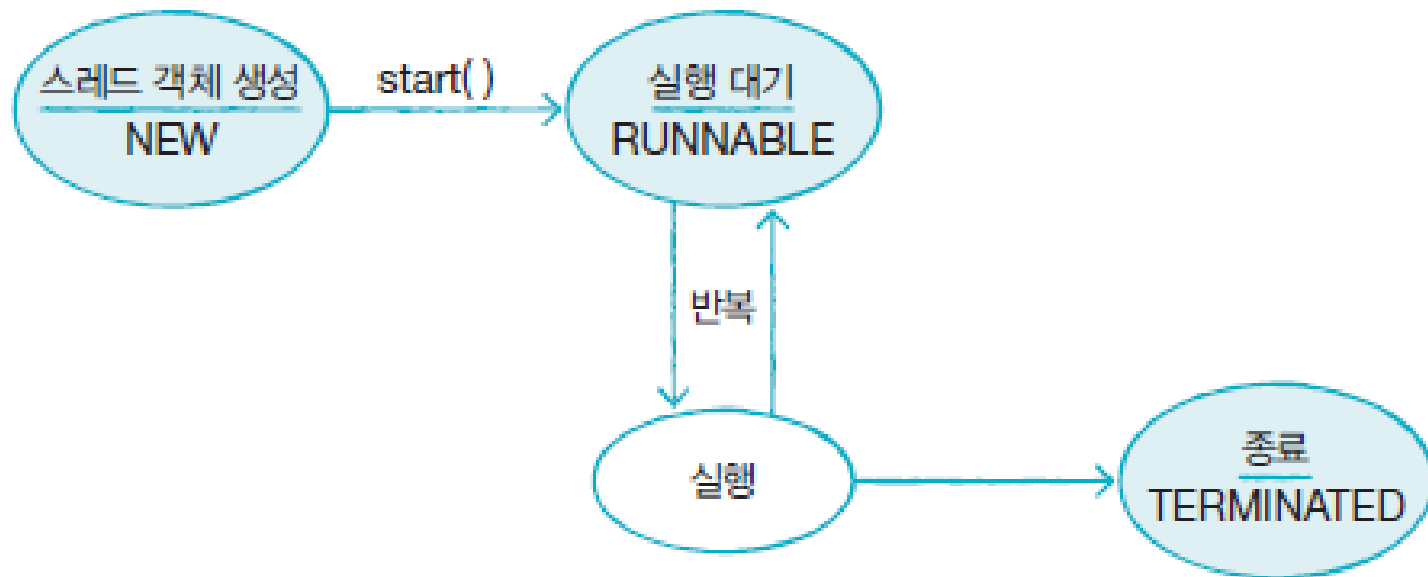
- 스레드가 사용 중인 객체를 다른 스레드가 변경할 수 없게 하려면 스레드 작업 끝날 때까지 객체에 잠금 걸어야 함
- 임계 영역 (critical section) : 단 하나의 스레드만 실행할 수 있는 코드 영역
- 동기화 (synchronized) 메소드 : 스레드가 객체 내부의 동기화 메소드 실행하면 즉시 객체에 잠금 걸림

```
public synchronized void method() {  
    임계 영역; //단 하나의 스레드만 실행  
}
```



# 쓰레드 수명 주기

- 실행 상태 스레드는 run() 메소드를 모두 실행하기 전 다시 실행 대기 상태로 돌아갈 수 있음
- 실행 대기 상태에 있는 다른 스레드가 선택되어 실행 상태가 되기도 함
- 실행 상태에서 run() 메소드의 내용이 모두 실행되면 스레드 실행이 멈추고 종료 상태가 됨



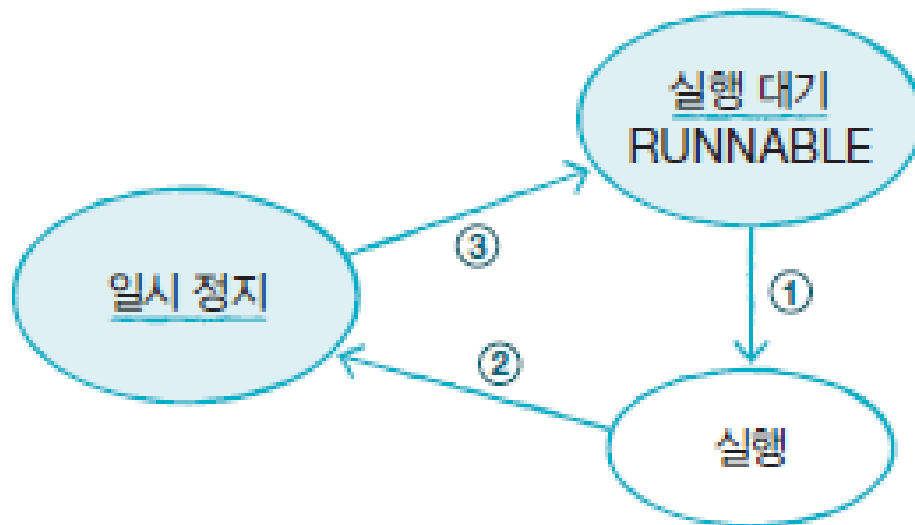
# 스레드 상태

## ❖ 실행 (running) 상태

- 실행 대기 상태의 스레드 중에서 운영체제가 하나를 선택하여 CPU가 run() 메소드를 실행하도록 하
- run() 메소드 모두 실행하기 전에 다시 실행 대기 상태로 돌아갈 수 있음

## ❖ 종료 (terminated) 상태

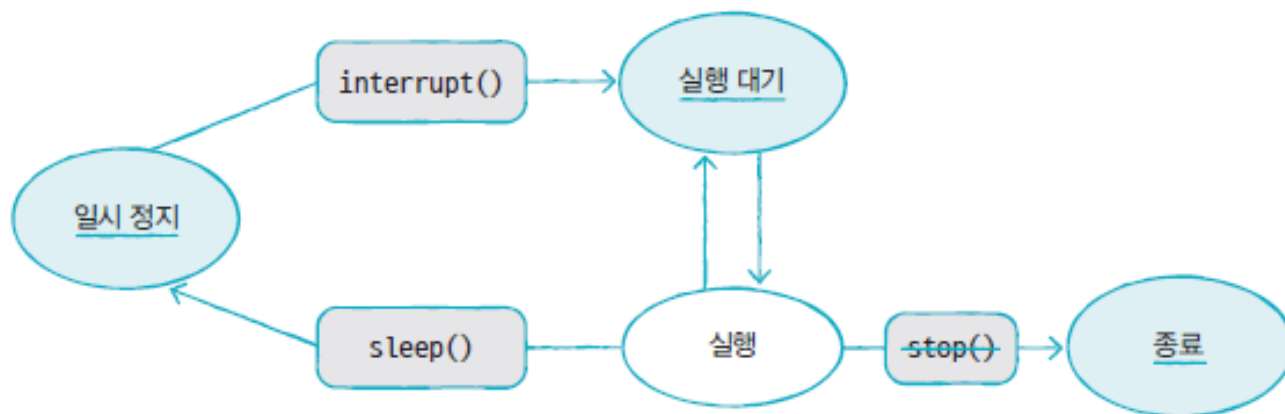
- 실행 상태에서 run() 메소드 종료되면 더 이상 실행할 코드 없기 때문에 스레드 실행이 정지됨



# 스레드 상태 제어

## ❖ 스레드 상태 제어

- 실행 중인 스레드의 상태를 변경
- 스레드 상태 변화에 필요한 메소드를 정확히 파악해야

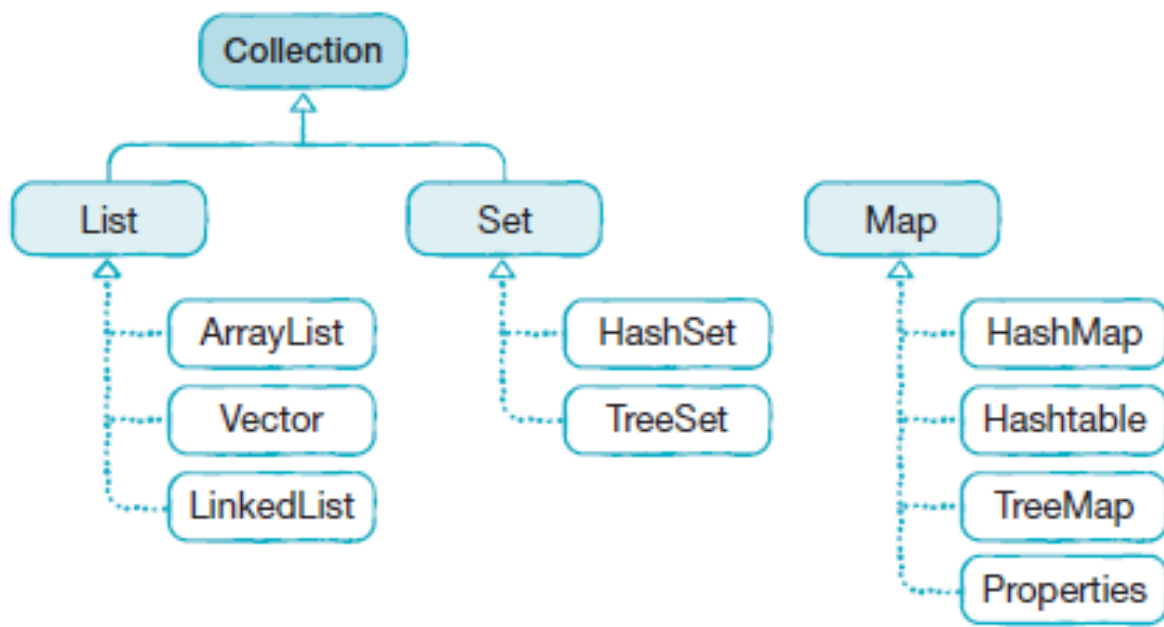


메소드	설명
interrupt()	일시 정지 상태의 스레드에서 InterruptedException을 발생시켜, 예외 처리 코드(catch)에서 실행 대기 상태로 가거나 종료 상태로 갈 수 있도록 합니다.
sleep(long millis)	주어진 시간 동안 스레드를 일시 정지 상태로 만듭니다. 주어진 시간이 지나면 자동적으로 실행 대기 상태가 됩니다.
stop()	스레드를 즉시 종료합니다. 불안정한 종료를 유발하므로 사용하지 않는 것이 좋습니다.

# 컬렉션 프레임워크

## ❖ 컬렉션 프레임워크 (Collection Framework)

- 자료구조를 사용해서 객체들을 효율적으로 관리할 수 있도록 인터페이스와 구현 클래스를 java.util 패키지에서 제공함
- 프레임워크 : 사용 방법을 정해놓은 라이브러리
- 주요 인터페이스로 List, Set, Map이 있음



# List 컬렉션

## ❖ List 컬렉션

- 객체를 인덱스로 관리
- 저장용량이 자동으로 증가하며 객체 저장 시 자동 인덱스가 부여
- 추가, 삭제, 검색 위한 다양한 메소드 제공
- 객체 자체를 저장하는 것이 아닌 객체 번지 참조
  - null도 저장 가능

### 힙 영역

#### List 컬렉션

0	1	2	...	n-1
번지	번지	번지	...	번지



# List 컬렉션

기능	메소드	설명
객체 추가	<code>boolean add(E e)</code>	주어진 객체를 맨 끝에 추가합니다.
	<code>void add(int index, E element)</code>	주어진 인덱스에 객체를 추가합니다.
	<code>E set(int index, E element)</code>	주어진 인덱스에 저장된 객체를 주어진 객체로 바꿉니다.
객체 검색	<code>boolean contains(Object o)</code>	주어진 객체가 저장되어 있는지 조사합니다.
	<code>E get(int index)</code>	주어진 인덱스에 저장된 객체를 리턴합니다.
	<code>boolean isEmpty()</code>	컬렉션이 비어 있는지 조사합니다.
	<code>int size()</code>	저장되어 있는 전체 객체 수를 리턴합니다.
객체 삭제	<code>void clear()</code>	저장된 모든 객체를 삭제합니다.
	<code>E remove(int index)</code>	주어진 인덱스에 저장된 객체를 삭제합니다.
	<code>boolean remove(Object o)</code>	주어진 객체를 삭제합니다.



# List 컬렉션

## ❖ ArrayList

- List 인터페이스의 대표적 구현 클래스
- ArrayList 객체 생성



```
List<String> list = new ArrayList<String>();
```

```
List<String> list = new ArrayList<>();
```

# List 컬렉션

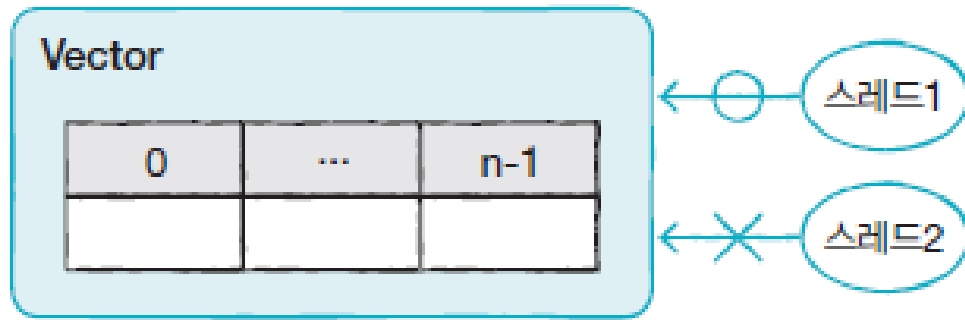
## ❖ Vector

- 저장할 객체 타입을 타입 파라미터로 표기하고 기본 생성자 호출하여 생성

```
List<E> list = new Vector<E>();  
List<E> list = new Vector<>();
```

Vector의 E 타입 파라미터를 생각하면  
← 왼쪽 List에 지정된 타입을 따라 감

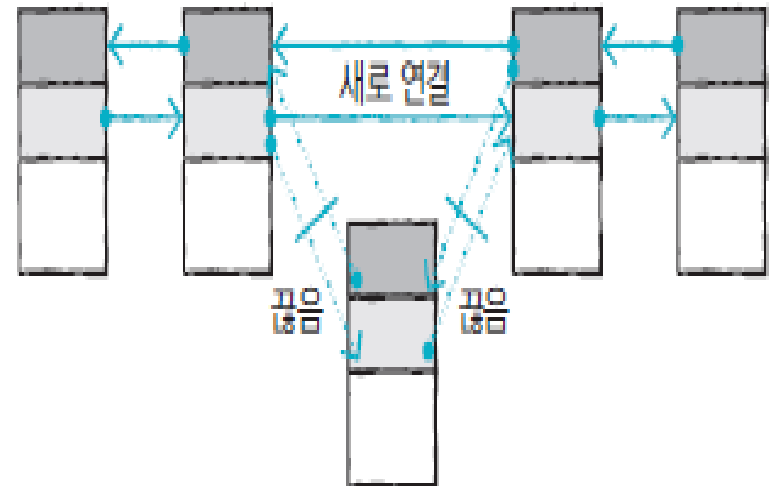
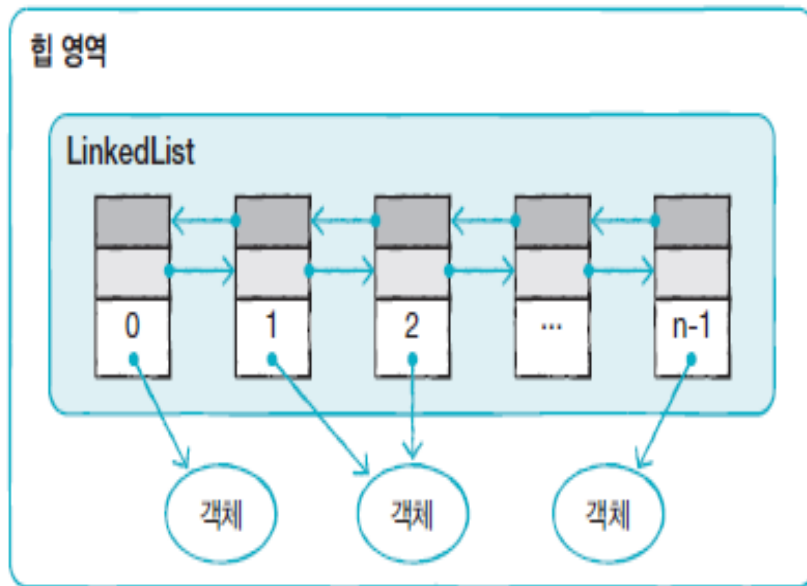
- 동기화된 메소드로 구성되어 멀티 스레드가 동시에 Vector의 메소드들 실행할 수 없고, 하나의 스레드가 메소드 실행 완료해야만 다른 스레드가 메소드 실행할 수 있음
- 멀티 스레드 환경에서 안전하게 객체 추가 및 삭제할 수 있음
  - 스레드에 안전 (thread safe)



# List 컬렉션

## ❖ LinkedList

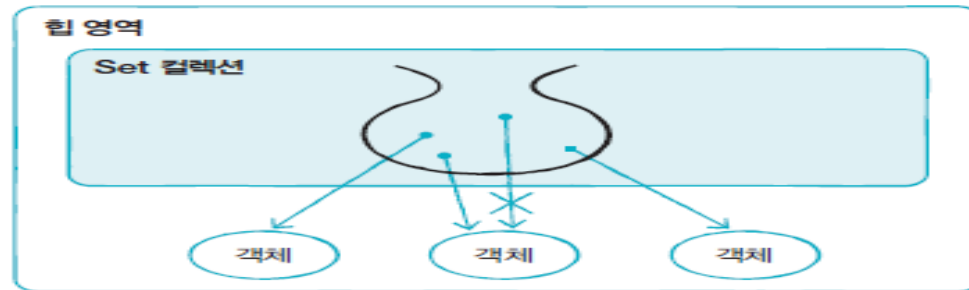
- ArrayList와 사용 방법은 같으나 내부 구조가 다름
- 인접 참조를 링크하여 체인처럼 객체를 관리
- 특정 인덱스 객체 제거하거나 삽입하면 앞뒤 링크만 변경되고 나머지 링크는 변경되지 않음



# Set 컬렉션

## ❖ Set 컬렉션

- 저장 순서 유지되지 않으며, 객체 중복하여 저장할 수 없고 하나의 null만 저장할 수 있다.



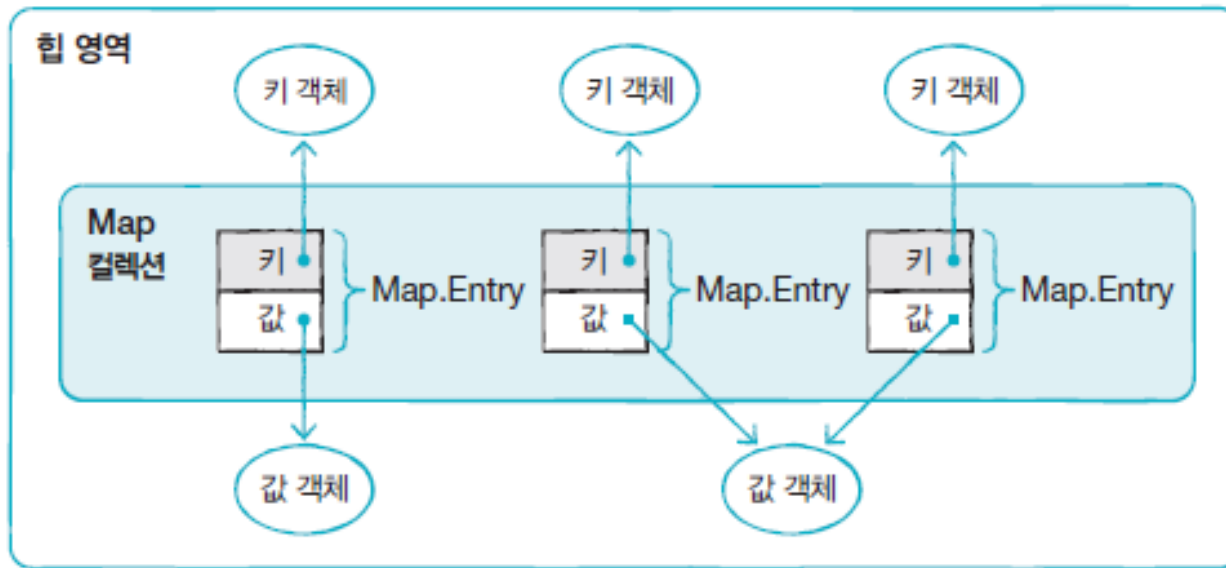
- HashSet, LinkedHashSet, TreeSet 등

기능	메소드	설명
객체 추가	<code>boolean add(E e)</code>	주어진 객체를 저장합니다. 객체가 성공적으로 저장되면 <code>true</code> 를 리턴하고 중복 객체면 <code>false</code> 를 리턴합니다.
객체 검색	<code>boolean contains(Object o)</code>	주어진 객체가 저장되어 있는지 조사합니다.
	<code>boolean isEmpty()</code>	컬렉션이 비어 있는지 조사합니다.
	<code>Iterator&lt;E&gt; iterator()</code>	저장된 객체를 한 번씩 가져오는 반복자를 리턴합니다.
	<code>int size()</code>	저장되어 있는 전체 객체 수를 리턴합니다.
객체 삭제	<code>void clear()</code>	저장된 모든 객체를 삭제합니다.
	<code>boolean remove(Object o)</code>	주어진 객체를 삭제합니다.

# Map 컬렉션

## ❖ Map 컬렉션

- 키와 값으로 구성된 Map.Entry 객체 저장하는 구조 가짐
- 키는 중복 저장될 수 없으나 값은 중복 저장될 수 있음
  - 기존 저장된 키와 동일한 키로 값을 저장하면 기존 값 없어지고 새로운 값으로 대체



- HashMap, Hashtable, LinkedHashMap, Properties, TreeMap 등

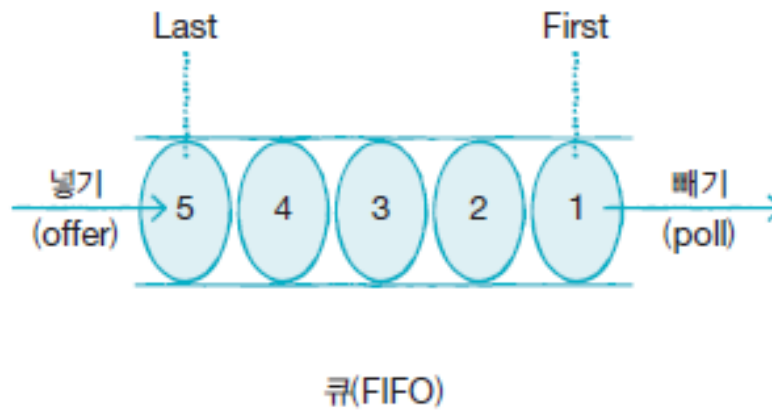
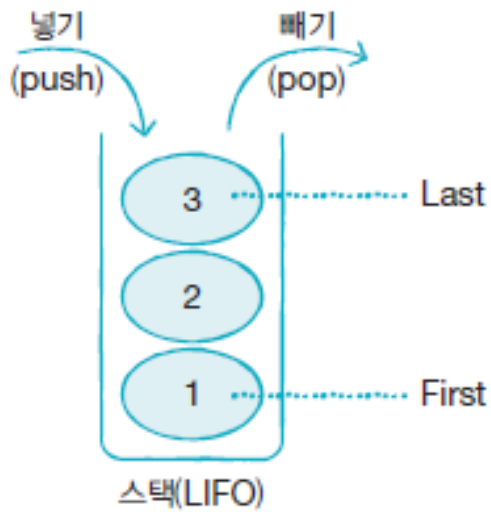
# Map 컬렉션

## ■ Map 인터페이스 메소드

기능	메소드	설명
객체 추가	<code>V put(K key, V value)</code>	주어진 키로 값을 저장합니다. 새로운 키일 경우 null을 리턴하고 동일한 키가 있을 경우 값을 대체하고 이전 값을 리턴합니다
객체 검색	<code>boolean containsKey(Object key)</code>	주어진 키가 있는지 여부를 확인합니다.
	<code>boolean containsValue(Object value)</code>	주어진 값이 있는지 여부를 확인합니다.
	<code>Set&lt;Map.Entry&lt;K,V&gt;&gt; entrySet()</code>	키와 값의 쌍으로 구성된 모든 Map.Entry 객체를 Set에 담아서 리턴합니다.
	<code>V get(Object key)</code>	주어진 키가 있는 값을 리턴합니다.
	<code>boolean isEmpty()</code>	컬렉션이 비어 있는지 여부를 확인합니다.
	<code>Set&lt;K&gt; keySet()</code>	모든 키를 Set 객체에 담아서 리턴합니다.
	<code>int size()</code>	저장된 키의 총 수를 리턴합니다.
	<code>Collection&lt;V&gt; values()</code>	저장된 모든 값을 Collection에 담아서 리턴합니다.
객체 삭제	<code>void clear()</code>	모든 Map.Entry(키와 값)를 삭제합니다.
	<code>V remove(Object key)</code>	주어진 키와 일치하는 Map.Entry를 삭제하고 값을 리턴합니다.

# 입출력 제어 컬렉션 클래스

- ❖ 후입선출 (LIFO : List In First Out)
  - 나중에 넣은 객체가 먼저 빠져나가는 자료구조
- ❖ 선입선출 (FIFO : First In First Out)
  - 먼저 넣은 객체가 먼저 빠져나가는 자료구조
- ❖ 컬렉션 프레임워크에는 LIFO 자료구조 제공하는 Stack 클래스와 FIFO 자료구조 제공하는 Queue 인터페이스 제공됨



# Stack

## ❖ Stack

- LIFO 자료구조 구현한 클래스

리턴 타입	메소드	설명
E	push(E item)	주어진 객체를 스택에 넣습니다.
E	peek()	스택의 맨 위 객체를 가져옵니다. 객체를 스택에서 제거하지 않습니다.
E	pop()	스택의 맨 위 객체를 가져옵니다. 객체를 스택에서 제거합니다.

- Stack 객체 생성하려면 저장할 객체 타입을 E 타입 파라미터 자리에 표기하고 기본 생성자를 호출

```
Stack<E> stack = new Stack<E>();
```

```
Stack<E> stack = new Stack<>();
```

Stack의 E 타입 파라미터를 생략하면

왼쪽 Stack에 지정된 타입을 따라 감



# Queue

## ❖ Queue

- FIFO 자료구조에서 사용되는 메소드 정의

리턴 타입	메소드	설명
boolean	offer(E e)	주어진 객체를 넣습니다.
E	peek()	객체 하나를 가져옵니다. 객체를 큐에서 제거하지 않습니다.
E	poll()	객체 하나를 가져옵니다. 객체를 큐에서 제거합니다.

- LinkedList 클래스

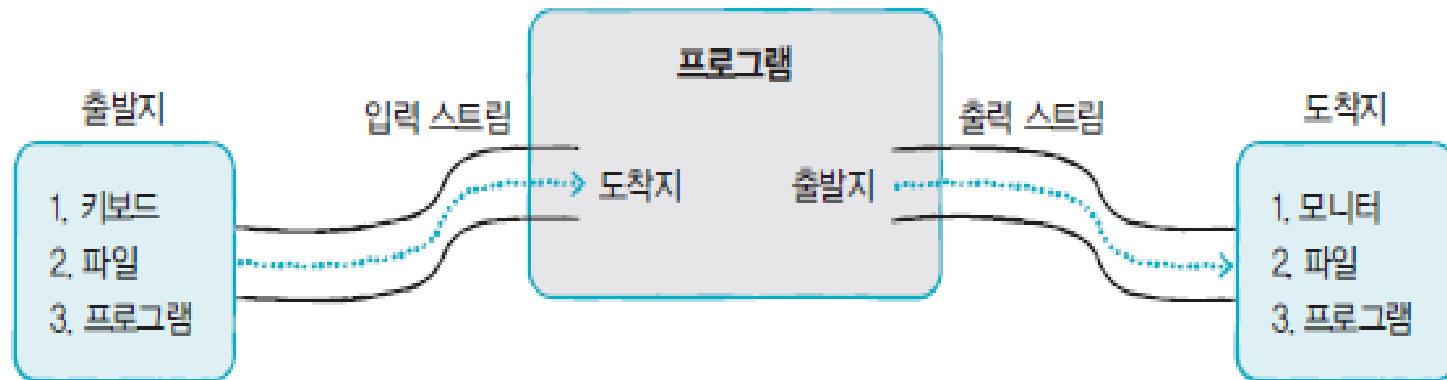
```
Queue<E> queue = new LinkedList<E>();  
Queue<E> queue = new LinkedList<>();
```

LinkedList의 E 타입 파라미터를 생략하면  
왼쪽 Queue에 지정된 타입을 따라 감

# 스트림

## ❖ 스트림 (Stream)

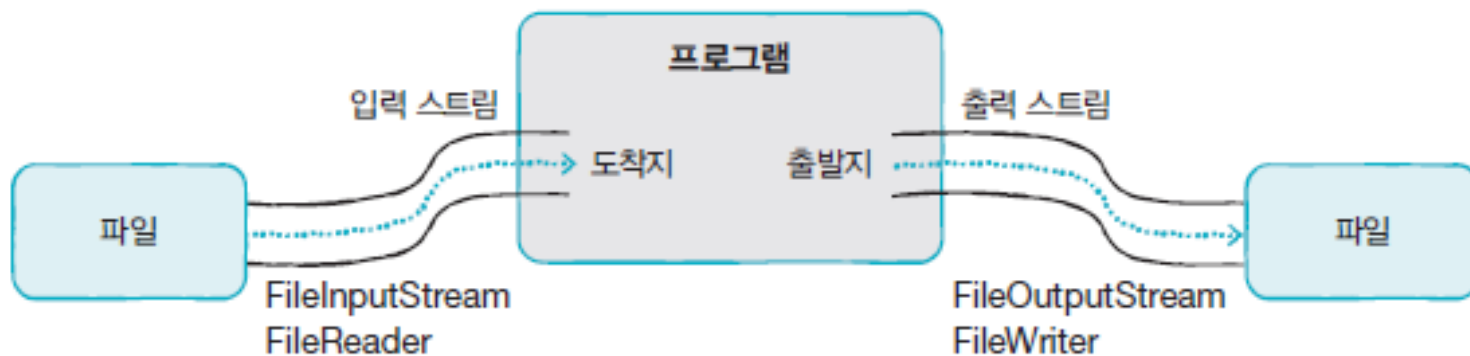
- 자바에서 데이터는 스트림을 통해 입출력됨
- 프로그램이 데이터의 출발지인지 도착지인지의 여부에 따라 사용하는 스트림의 종류가 결정



# 입출력 스트림의 종류

- ❖ 바이트 기반 스트림
  - 그림, 멀티미디어 등의 바이너리 데이터를 읽고 출력
- ❖ 문자 기반 스트림
  - 문자 데이터를 읽고 출력할 때 사용
- ❖ 최상위 클래스로 스트림 클래스의 바이트 / 문자 기반 판단

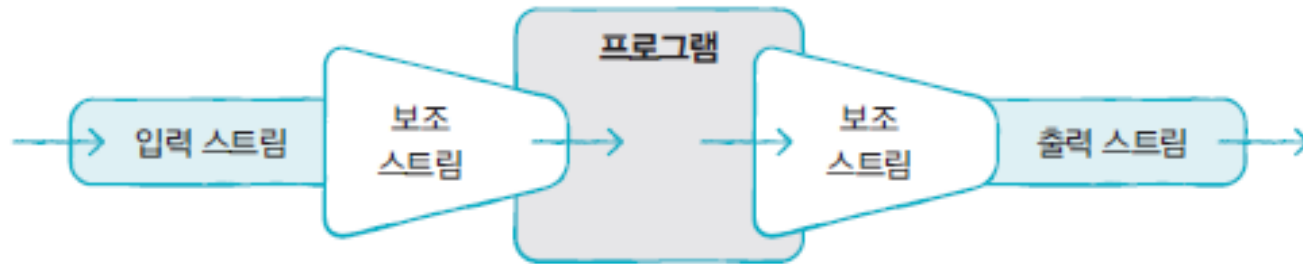
구분	바이트 기반 스트림		문자 기반 스트림	
	입력 스트림	출력 스트림	입력 스트림	출력 스트림
최상위 클래스	InputStream	OutputStream	Reader	Writer
하위 클래스 (예)	XXXInputStream (FileInputStream)	XXXOutputStream (FileOutputStream)	XXXReader (FileReader)	XXXWriter (FileWriter)



# 시작하기 전에

## ❖ 보조 스트림

- 다른 스트림과 연결되어 여러가지 편리한 기능을 제공하는 스트림
- 자체적으로 입출력 수행할 수 없기 때문에 입출력 소스와 바로 연결되는 InputStream, OutputStream, Reader, Writer 등에 연결하여 입출력 수행



- 프로그램은 입력 스트림으로부터 직접 데이터 읽지 않고, 보조 스트림에서 제공하는 기능 이용하여 데이터 읽음. 또한 출력 스트림으로 직접 데이터 보내지 않고 보조 스트림에서 제공하는 기능 이용하여 데이터 보냄

# 보조 스트림 연결하기

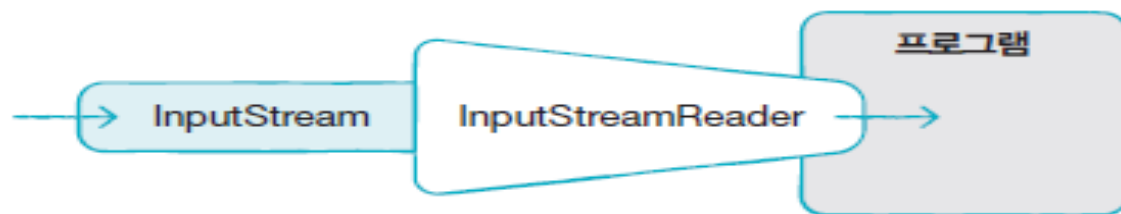
## ❖ 보조 스트림 연결하기

- 보조 스트림 생성 시 자신이 연결될 스트림을 생성자의 매개값으로 제공

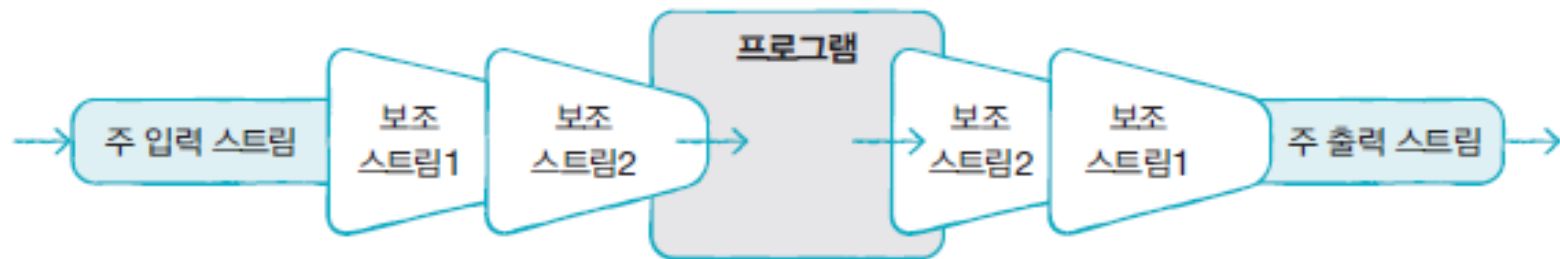
```
보조스트림 변수 = new 보조스트림(연결스트림)
```

```
InputStream is = ...;
```

```
InputStreamReader reader = new InputStreamReader(is);
```



- 보조 스트림을 연속적으로 연결할 수도 있음



# File 클래스

❖ java.io 패키지에서 제공하는 File 클래스는 파일 및 폴더 정보 제공 역할 함

```
File file = new File("C:/Temp/file.txt");  
File file = new File("C:\\Temp\\file.txt");
```

- 경로 구분자는 운영체제마다 조금씩 다름
  - 윈도우에서는 \ 및 / 모두 사용 가능
- 해당 경로에 실제로 파일이나 폴더 있는지 확인하려면 File 객체 생성후 exists() 메소드 호출

```
boolean isExist = file.exists();
```

- exists() 메소드의 리턴값이 false일 경우 다음 메소드로 파일 혹은 폴더 생성

리턴 타입	메소드	설명
boolean	createNewFile()	새로운 파일을 생성합니다.
boolean	mkdir()	새로운 폴더를 생성합니다.
boolean	mkdirs()	경로상에 없는 모든 폴더를 생성합니다.

# File 클래스

- exists() 메소드의 리턴값이 true라면 다음 메소드 사용 가능

리턴 타입	메소드	설명
boolean	delete()	파일 또는 폴더를 삭제합니다.
boolean	canExecute()	실행할 수 있는 파일인지 여부를 확인합니다.
boolean	canRead()	읽을 수 있는 파일인지 여부를 확인합니다.
boolean	canWrite()	수정 및 저장할 수 있는 파일인지 여부를 확인합니다.
String	getName()	파일의 이름을 리턴합니다.
String	getParent()	부모 폴더를 리턴합니다.
File	getParentFile()	부모 폴더를 File 객체로 생성 후 리턴합니다.
String	getPath()	전체 경로를 리턴합니다.
boolean	isDirectory()	폴더인지 여부를 확인합니다.
boolean	isFile()	파일인지 여부를 확인합니다.
boolean	isHidden()	숨김 파일인지 여부를 확인합니다.
long	lastModified()	마지막 수정 날짜 및 시간을 리턴합니다.
long	length()	파일의 크기를 리턴합니다.
String[]	list()	폴더에 포함된 파일 및 서브 폴더 목록 전체를 String 배열로 리턴합니다.
String[]	list(FilenameFilter filter)	폴더에 포함된 파일 및 서브 폴더 목록 중에 FilenameFilter에 맞는 것만 String 배열로 리턴합니다.
File[]	listFiles()	폴더에 포함된 파일 및 서브 폴더 목록 전체를 File 배열로 리턴합니다.
File[]	listFiles(FilenameFilter filter)	폴더에 포함된 파일 및 서브 폴더 목록 중에 FilenameFilter에 맞는 것만 File 배열로 리턴합니다.