1.자바개요

◆ 자바소개

- 1995년 발표된 객체지향프로그램
- JDK도 같이 발표와 동시에 자바 언어를 이용하여 개발한 웹 브라우저인 핫자바를 발표
  ->핫자바 : 자바언어로 만든 애플릿을 실행할 수 있는 전용 인터넷 브라우저
  ->애플릿 : 자바로 만들어진 프로그램으로 인터넷 브라우저에서 실행되는 프로 그램
- 역사 : 1990년 초 그린 프로젝트 시작->C++언어를 사용->C++언어가 부족->C++언어를 기반으로 오크언어 개발->오크언어를 발전시켜 자바를 만들게 됨
- 특징
  - 단순하다->포인터를 이용하지 않고 메모리 관리가 편리함
  - 객체지형 언어->클래스와 클래스의 실례인 객체를 중심으로 프로 그램을 개발하는 언어
  - 시스템에 독립적->하나의 플랫폼에서 만든 자바 프로그램은 다른 플랫폼에서 별도의 작업 없이 실행 가능
  - 번역언어->자바 프로그램 소스는 중간 코드인 바이트코드로 변환되며 바이트코드는 자바 가상 기계에서 인터프리터의 도움으로 실행
- 자바의 다양한 기술
  - Java SE : 임베디드 환경과 개인용 컴퓨터 그리고 서버에서 활용될 자바 응용 프로그램을 개발하고 구현하는 기술
  - Java EE : 다중계층의 대규모 기업 응용 시스템을 개발하기 위한 표준 플랫폼 제공
  - Java ME : 모바일 전화기 및 PDA, TV 셋탑박스, 이동차량에 부착된 각종장치 및 여러 임베디드 장치를 위한 자바 플랫폼
- 스마트폰 앱 개발 언어인 자바
  - 안드로이드 : 자바를 만든 모바일 운영체제(구글)
  - iOS : 애플이 만든 운영체제

- 자바 프로그래밍
  - 자바소스 : 확장자는 java/대소문자 구분/소스에서 public인 클래스 이름과 동일
  - 컴파일러 : 소소 파일에서 실행파일을 생성하는 소프트웨어
  - 바이트코드 : 자바 소스를 컴파일하면 바이트코드 생성/확장자는 class/플랫폼에 독립적인 명령어로 구성된 이진파일/자바 플랫폼에서 인터프리터에 의해 실행/독립적으로 자바 플랫폼이 설치된 여러 플랫폼에서 실행
- 플랫폼에 독립적
  - 플랫폼 : 각종프로그램이 실행되는 하드웨어와 소프트웨어로 구성된 실행 환경/구성하는 주요 요소는 하드웨어인 CPU와 소프트웨어인 운영체제
  - 자바 플랫폼 : 여러 플랫폼에서 운영될 수 있는 소프트웨어로만 구성된 플랫폼
    ->자바가상기계와 자바 응용프로그래밍 인터페이스로 구성
  - 자바가상기계 : CPU와 같이 실행할 명령어 집합을 갖는 소프트웨어/.class인 바이트코드는 자바가상기계의 명령어인 기계어로 구성된 이진파일
  - 자바 API : 자바 프로그램을 실행하기 위한 각종 클래스 라이브러리
  - 플랫폼에 독립적 : 자바로 한번 작성된 프로그램은 어느 플랫폼에서도 실행될 수 있다
- 자바 개발 환경
  - JRE : 자바 API와 자바 가상 기계 그리고 자바프로그램을 실행하기 위한 여러 콤포넌트로 구성된 자바 실행 환경
  - JDK : 자바 언어를 이용하여 프로그램을 개발하기 위한 최소한의 환경
- 자바 통합 개발 환경
  - IDE : 컴파일러, 디버거, 링커, 에디터 등을 통합적으로 제공하는 개발 한경
  - 이클립스 : 이클립스 컨소시엄이 개발하는 자바 통합 개발 환경
  - 넷빈 : 넷빈 컨소시엄에서 개발하는 무료 통합개발환경으로 소스도 공개
- 자바 프로그램 개발 순서

  작업공간 지정->자바프로젝트 생성->자바 클래스 생성->자바 응용프로그램 실행

2.자바 프로그래밍 기초

◆ 자바 클래스와 패키지

- 클래스 : 자바의 프로그램 단위

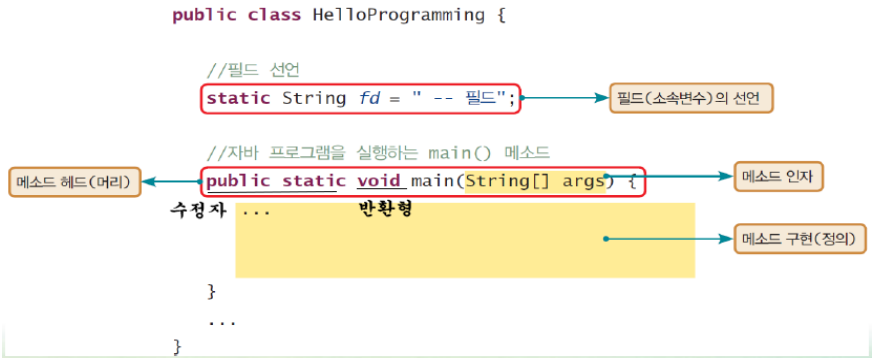
->하나의 소스로 구성/클래스 이름 이후에 중괄호의 블록으로 구성/이름은 대소문자 구분하며 public인 경우 반드시 파일 이름과 일치

- 패키지 : 관련된 다양한 자바 클래스가 모여있는 폴더

->package문장으로 패키지를 생성/자바 소스 맨 앞에 위치/마침표를 이용하여 하부 폴더 정의

◆ 자바 클래스의 구조

- 자바 클래스 내부는 필드(소속변수)와 메소드로 구성
- 메소드는 절차지향 언어에서 말하는 함수와 같은 기능 수행/main() 메소드는 특별한 메소드로 자바 프로그램이 실행되는 문장이 기술



◆ 키워드와 식별자

- 키워드(예약어) : 문법적으로 의미 있는 단어로 사용하기 위해 미리 정의해 놓은 단어
- 식별자 : 프로그래머가 정의하여 사용하는 단어

->대소문자 알파벳의 영문자, 숫자(0~9), 밑줄(_), $를 사용

1. 키워드는 식별자로 사용 못함
2. 식별자의 첫 문자로 숫자가 나올 수 없다
3. 대소문자를 구별하며 공백일 들어갈 수 없다
4. 유니코드를 지원하므로 한글로 이용가능

->실무 프로그램에서 한글 사용은 권장하지 않음

◆ 문장과 주석

- 문장 : 컴퓨터에게 명령을 내리는 최소 단위->세미콜론;으로 종료
- 블록 : 클래스 정의 또는 메소드 정의에 사용/사용자가 임의로 블록을 구성할 수 있다
- 인덴테이션 : 클래스 정의에서 필드나 메소드의 첫 글자는 탭만큼 들여쓰는 방식
- 주석 : 프로그램 내용에 전혀 영향을 미치지 않는 설명문 -> // : 한 줄 주석, /*...*/ : 블록 주석

◆ 자바의 자료형

- 기본형 : 변수의 저장공간에 값 자체가 저장
- 참조형 : 변수의 저장 공간에 참조 값이 저장

->참조 값 : 실제 값이 저장된 객체

구분	분류	키워드
기본형	논리형	boolean
	문자형	char
	정수형	byte, short, int, long
	실수형	float, double
참조형	배열	int [], float [] 등 다양
	클래스	String, Date 등 다양
	인터페이스	Runnable, Enumeration 등 다양

- 자료형의 크기
  - 정수와 실수를 표현하는 자료형마다 표현 범위가 다르다

분류	키워드	크기	상대적 크기 비교	최소 ~ 최대	지수형태 범위
논리형	boolean	1바이트		false, true	
문자형	char	2바이트		\u0000 ~ \uffff, [0 ~ 65,535]	$0 \sim 2^{16}-1$
정수형	byte	1바이트		-128 ~ 127	$-2^{7} \sim 2^{7}-1$
	short	2바이트		-32,768 ~ 32,767	$-2^{15} \sim 2^{15}-1$
	int	4바이트		-2,147,483,648 ~ 2,147,483,647	$-2^{31} \sim 2^{31}-1$
	long	8바이트		$-2^{63} \sim 2^{63}-1$	$-2^{63} \sim 2^{63}-1$
실수형	float	4바이트		(+, -)1.4E-45 ~ 3.4028235E38	
	double	8바이트		(+, -)4.9E-324 ~ 1.7976931348623157E308	

◆ 상수와 표현

- 상수 : 소스에 그대로 표현할 수 있는 다양한 자료 값

->숫자 앞에 0은 8진수, 0x or 0X는 16진수/01로만 구성된 수 앞에 0b는 이진수

- 숫자에 사용하는 밑줄
  ->숫자를 표현하는 중간에 밑줄(_)은 자릿수를 구분하는 구분자로 사용
- 특수문자
  - 인쇄할 수 없는 문자나 특수 문자를 표현하려면 역슬래시를 쓰고 문자나 숫자를 써서 나타냄

◆ **변수 선언과 초기화**

- 변수 선언 : 자료형과 변수 이름을 나열하여 표시 -> 변수 : 자료 값을 저장하는 공간
- 초기 값 지정 -> 변수의 초기화 : 변수를 선언한 이후에 는 반드시 값을 저장
- 여러 변수의 선언
  ->하나의 변수선언 문장으로 여러 개의 변수를 선언 가능
  ->ex) int a, b, c; =>여러 변수 선언/ int x, y=3, x=1; =>부분적으로 초기값 대입
  int num1=30, num2=20; =>모든 초기값 선언

◆ **소속변수와 지역변수, 변수의 기본 값**

- 소속변수(필드) : 클래스 내부에 소속된 변수이며 대부분의 메소드에서 사용가능
- 지역변수 : 메소드 내부에서 선언되는 변수로 선언된 메소드 내부에서만 사용
- 지역변수의 초기 값 미지정 오류

->지역변수는 변수 선언 시 초기 값을 저장하지 않고 사용하면 오류 발생

◆ **클래스 Scanner를 이용한 자료형의 입력**

- 콘솔에 입력하는 다양한 자료 값을 입력받으려면 클래스 java.util.Scanner를 사용

->input은 자료형을 java.util.Scanner로 선언하여 객체를 저장

->생성된 객체 input을 이용하여 입력받으려면 input.next()를 호출하여 반환 값을 문자열 변수에 저장

- Import 문장
  - Java.lang을 제외한 모든 패키지는 클래스 이름 앞에 모두 패키지 이름을 기술해야하는 번거로움을 피하기 위해 import 문장을 사용한다

3.연산자와 조건

◆ 표현식과 연산자 종류

- 연산자 : +, -, *기호와 같이 이미 정의된 연산을 수행하는 문자 또는 문자 조합기호
- 피연산자 : 연산에 참여하는 변수나 상수
- 표현식 : 연산자와 피연산자로 구성된 연산식

  ->항상 하나의 결과 값을 반드시 값는다
- 종류 : 피연산자의 수에 따라 단항, 이항, 삼항이 존재
- 우선순위

우선순위	연산자 이름	연산자	연산방향
1	후위 단항	var++ var--	왼쪽에서 오른쪽으로 ⟶
2	전위 단항	++var --var +expr -expr ~ !	
3	곱셈 부류	* / %	
4	덧셈 부류	+ -	
5	비트 이동	<< >> >>>	
6	관계	< > <= >= instanceof	
7	동등	== !=	
8	비트 AND	&	
9	비트 배타적 OR	^	
10	비트 OR	|	
11	논리 AND	&&	
12	논리 OR	||	
13	조건 삼항	expr ? x : y	
14	대입	= += -= *= /= %= &= ^= |= <<= >>= >>>=	⟵

- 대입 연산자
  - 연산자의 오른쪽 값을 왼쪽 변수에 저장하는 연산자

    ->왼쪽은 반드시 값을 저장할 수 있는 변수
- 축약 대입 연산자 ⟶
  - 복합 대입 연산자라고 부른다
  - 결과 값은 왼쪽 변수에 저장

연산자	연산 예	연산자 의미
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

- 증감 연산자
  - 모두 단항 연산자이며 변수만을 피연산자로 사용
  - 상수나 일반 수식을 피연산자로 사용 못함

연산자 위치	연산자	연산 결과	연산 후 n의 값
전위	++n	n+1	1증가
	--n	n-1	1감소
후위	n++	n	1증가
	n--	n	1감소

- 조건 연산자
  - 조건의 논리 값에 따라 2개의 피연산자 중 하나가 결과 값이 되는 연산자
  - 유일한 삼항 연산자
  - 연산 결과는 true 또는 false

  ->x ? a : b =>x가 true이면 a, false이면 b
- 관계 연산자
  - 2개의 피연산자의 크기를 비교하는 연산자
  - 연산 결과는 true 또는 false
  - 종류 : >, >=, <, <=, ==, !=(같지 않다)
- 논리 연산자
  - 피연산자는 boolean형이어야 하며 결과도 true 또는 false
  - &&와 ||는 피연산자 두 개 중에서 왼쪽 피연산자만으로 전체 결과가 결정된다면 오른쪽 피연산자는 평가하지 않음

연산자	이름	연산자의 의미
&&	AND	모두 true이면 true
||	OR	하나라도 true이면 true
^	XOR	서로 다르면true
!	NOT	!x에서 x의 논리 값과 반대

- 비트 연산자
  - 피연산자 정수 값으 비트 단위로 논리 연산을 수행
  - ~ 단항 연산자이며 나머지는 이항 연산자
  - 각 피연산자를 int형으로 변환하여 연산하며 결과도 int형

연산자	연산자 이름	사용	의미
&	비트 AND	op1&op2	비트가 모두 1이면 1
|	비트 OR	op1|op2	비트가 적어도 하나가 1이면 1
^	비트 베타적 OR(XOR)	op1^op2	비트가 서로 다르면1
~	보수	~op1	비트가 0이면1, 1이면 0

- 비트 이동 연산자

연산자	이름	사용	연산 방법	새로 채워지는 비트
>>	Signed left shift	op1>>op2	op1을 오른쪽으로 op2 비트만큼 이동	가장 왼쪽 비트인 부호는 원래의 비트로
<<	Signed right shift	op1<<op2	Op1을 왼쪽으로 op2 비트만큼 이동	가장 오른쪽 비트를 모두 0으로 채움
>>>	Unsigned left shift	op1>>>op2	Op1을 오른쪽으로 op2 비트만큼 이동	가장 왼쪽 비트인 부호 비트는 모두 0으로 채움

- 비트 축약 대입 연산자
  - (x^y)^y==x가 성립

연산자	연산 예	연산자 의미
<<=	x<<=y	x=x<<y
>>=	x>>=y	x=x>>y
>>>=	x>>>=y	x=x>>>y
&=	x&=y	x=x&y
|=	x|=y	x=x|y
^=	x^=y	x=x^y

◆ 형 변환

- 명시적 형변환(큰->작)
  - 실수를 정수로 변환하거나 범위가 큰 정수형에서 더 작은 정수형으로 변환하는 것
- 자동 형변환(작->큰)
  - 컴파일러에 의해 표현 범위가 넓은 자료형으로 변환되는 것

◆ 조건문

- If
  - If는 조건의 논리 값에 따라 선택을 지원하는 구문
  - 형태 : if (cond) stmt;
    ->조건 cond가 true이면 stmt를 실행/false이면 stmt 실행하지 않음
- If else
  - 조건이 만족되면 문장을 실행하는 구문
  - 형태 : if (cond) stmt1; else stmt2;

  ->조건 cond가 true이면 stmt1을 실행하고 false이면 stmt2를 실행
- 계속된 조건 If else if else/중첩된 if ->이미지 참조

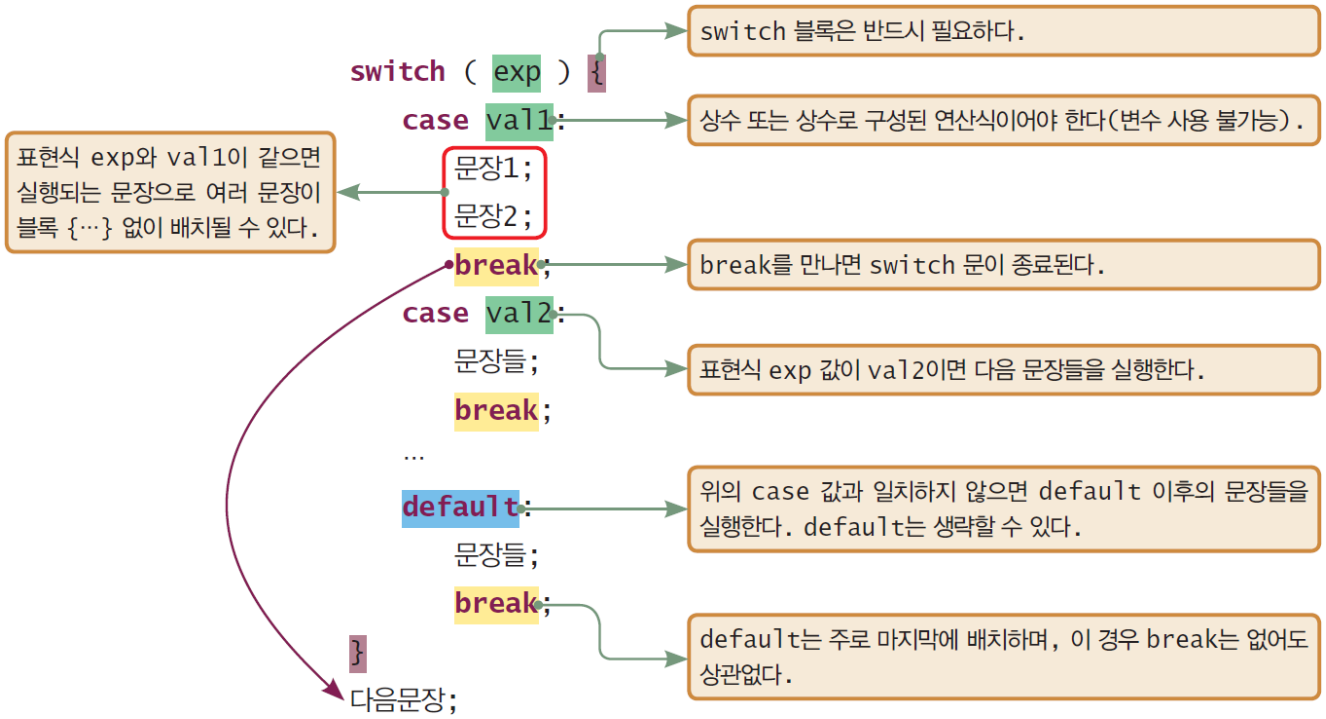
```java
package control.ifcondition;

import java.util.Scanner;

public class NestedIf {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("면허시험 종류선택 (1[1종] 또는 2[2종] 입력) >> ");
        int type = in.nextInt();
        System.out.print("필기 면허시험 점수 입력 >> ");
        int score = in.nextInt();

        if (type == 1) {
            if (score >= 70)
                System.out.println("1종 면허 시험 합격");
            else
                System.out.println("1종 면허 시험 불합격");
        } else if (type == 2) {
            if (score >= 60)
                System.out.println("2종 면허 시험 합격");
            else
                System.out.println("2종 면허 시험 불합격");
        }
    }

}

```

```java
package control.ifcondition;

import java.util.Scanner;

public class Cradeifelse {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("성적 입력:");
        int point = input.nextInt();
        char grade;

        if (90 <= point)
            grade = 'A';

        else if (80 <= point)
            grade = 'B';

        else if (70 <= point)
            grade = 'C';
        else
            grade = 'F';
        System.out.println("학점:" + grade);
    }

}

```

- Switch(exp)
  - 주어진 연산식의 결과 값에 따라 여러 개의 실행 경로 중 하나를 실행하는 문장
  - switch(exp) {...}는 표현식 exp 결과 값 중에서 case의 값과 일치하는 내부 문장을 실행

  ->exp : byte, short, char, int의 정수형 허용/실수형 허용X

  열거형과 문자열을 표현하는 String클래스, 기본형의 랩퍼 클래스인 Character, Byte, Short, Integer클래스를 허용



- Switch의 break
  - Switch에서 일치하는 case 문 내부를 실행한 후 break 문이 없다면 break 문을 만나기 전까지 무조건 다음 case 내부문장을 모두 실행
  - Switch문에서 하나의 case에 여러 개의 정수를 콤마로 나열하지 못함->case 4, 5(X)
- 문자와 문자열을 지원하는 switch
  - Switch의 연산식 결과 : 문자, 클래스 String 객체와 Byte, Short, Integer, Character 등의 랩퍼 클래스 객체 지원

```
char op = '+';
double x = 3.45, y = 9.83;
switch (op) {
    case '+':
       System.out.printf("%f + %f = %f\n", x, y, x+y);
       break;
    case '-':
       System.out.printf("%f - %f = %f\n", x, y, x-y);
       break;
    ...
}
```

그림 3-20 • switch에서 문자 지원

```
String nation = "한국";
switch ( nation ) {
    case "한국" :
    case "일본" :
    case "중국" :
       System.out.printf("%s은(는) 아시아입니다.\n", nation);
       break;
    ...
}
```

그림 3-21 • switch에서 문자열 지원

4.반복과 배열

◆ **제어문**

- 실행 흐름을 변형하여 조건에 따라 실행하거나 지정된 블록을 반복하거나 또는 다른 곳으로 이동하여 실행
- 종류 : 순차, 반복(for, do while, while), 조건(if, if else, if else if, nested if, switch), 분기(break, continue, return)

◆ **반복문**

- While
  - 형태 : while (cond) stmt;

    ->반복 조건식 cond를 평가하여 false이면 while 문장 종료, true이면 반복몸체인 stmt를 실행하고 다시 반복조건 cond를 평가하여 while문 종료 시까지 반복
- Do while
  - 반복몸체 수행 후 반복조건을 검사
  - 반복몸체에 특별한 구문이 없는 경우 반복문 do while의 몸체는 적어도 한 번은 실행
- For
  - 형태 : for (init;cond;inc) stmt;

  ->init : 초기화가 이루어짐, cond : 반복 검사, inc : 증감연산자
  - Cond를 생략하면 반복이 계속 진행
  - 반복 조건에 이용되는 변수i를 반복 제어변수라 함
  - 예외 : for (init;cond;inc); 반복몸체;

  ->반복몸체가 for에 의해 반복이 실행되지 않으며 1회만 실행
- 중첩된 반복문
  - 반복문 내부에 반복문이 다시 있는 구문
- For와 while 문의 비교
  - For 문장 Pre ; for (A;B;C) body; -> while 문장 pre ; A; while(B); {body; C;}
- 반복을 종료하는 break
  - 반복 내부에서 강제로 반복을 종료하려면 break문을 사용

- 반복을 계속하는 continue
  - 반복몸체의 나머지 부분을 실행하지 않고 다음 반복을 계속 유지하는 문장

◆ 배열

- 동일한 자료형을 정해진 수만큼 저장하는 객체
- 원소 : 동일한 자료형으로 배열을 구성하는 항목 -> 0으로 시작하는 수의 첨자에 의해 참조
- 배열 크기 : 배열 원소의 수
- 참조형으로 배열원소를 위한 공간과 함께 배열 크기가 저장되는 공간 필드 length의 객체를 가리킴
- 배열의 이름을 array라 하면 array.length로 배열 크기를 참조하며 배열 array의 배열 크기가 4이면 array.length는 4를 나타낸다

◆ 배열 선언과 객체 생성

- Float 형 배열은 float[]/double 형 배열은 double[]로 표현
- 배열형과 배열 이름으로 선언
- 배열 선언에서 절대로 배열 크기를 표시 할 수 없다

int month[];
double values[];

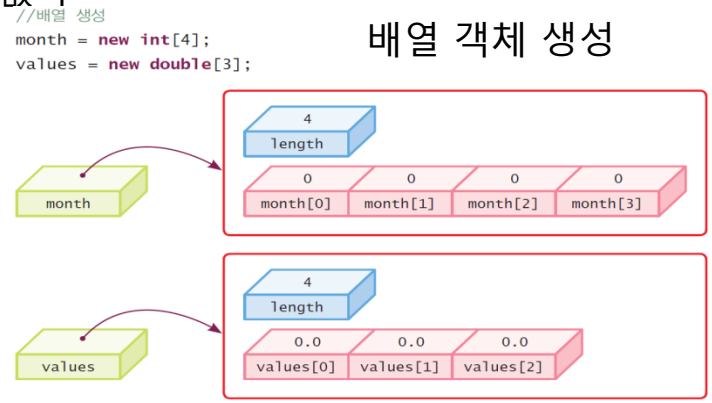
int []month;
double []values;

잘못된 경우
Int month[5];
Double values[10];

int[] ary1, ary2; - > 모두 int[]배열
int ary3[], i; -> ary3은 int[] 배열이나 i는 int

- 배열 선언과 생성을 한 문장으로
  - Int[] month = new int[4];
    -> int month[]; month = new int[4]
  - Double[] values = new double[3];
    -> double values[]; values = new double[3];

배열 객체 생성



기본 저장 값
정수형 : 0
실수형 : 0.0
논리형 : false
참조형 : null

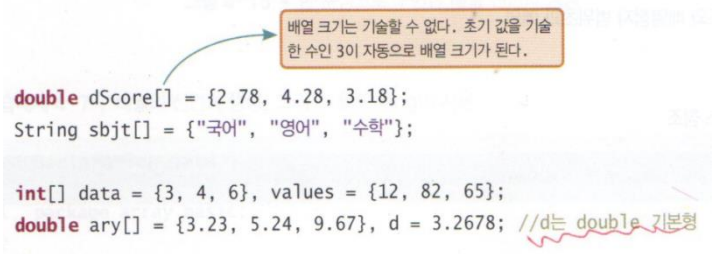
- 배열 원소 참조
  - 배열 이름[첨자] 형식을 사용
  - 배열첨자 범위를 벗어나면 오류
  - 첨자는 0에서 [배열크기-1]까지 유효

Double[] points = new double[3];

Points[0]=25; -> 유효범위
Points[3]=47; -> 유효범위를 벗어나므로 오류

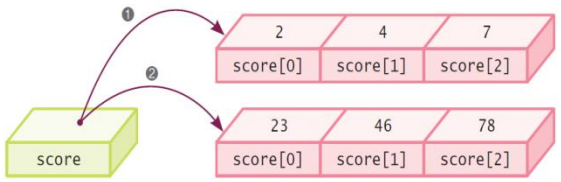
◆ **배열 초기화**

- 배열 선언 초기화 : 배열을 선언하면서 동시에 원소 값을 손쉽게 저장
- 중괄호 사이에 여러 원소 값을 쉼표로 구분하여 기술
- 배열크기는 기술하지 않으며 중괄호 속의 원소 수가 자동으로 배열크기



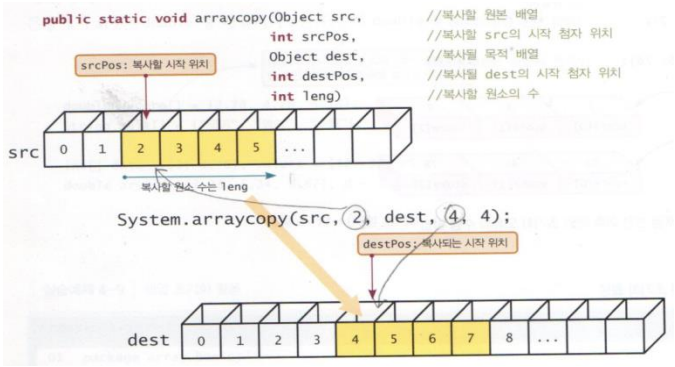
- 배열 선언 이후의 초기화와 다른 배열의 대입
  - 배열선언 이후 배열 이름으로 초기 값을 저장하면 오류 발생
  - 초기 값을 위한 중괄호 앞에 new 자료형[]을 기술

```
int score[];
score = {2, 4, 7};                  //오류발생
❶ score = new int[] {2, 4, 7};      //배열 선언 이후 초기화 방법
...
❷ score = new int[] {23, 46, 78};   //다른 배열을 생성하여 대입
```



◆ 배열 복사와 원소 출력

- 자바의 System 클래스는 배열을 복사하기 위해 메소드 arraycopy()를 제공



- For, for each 문을 이용하여 배열 원소 출력
  - 반복은 배열 원소 순서로 처리되며 각 반복 시 변수 value에 원소 값이 저장

```
int[] copyFrom = {1, 2, 3, 4, 5, 6, 7};

for (int i = 0; i < copyFrom.length; i++)
    System.out.print(copyFrom[i] + " ");
```

For문 이용

```
int[] copyTo = {10, 20, 30, 40, 50, 60, 70, 80};

for (int value : copyTo)
    System.out.print(value + " ");
```

```
for (배열 원소자료형 배열 원소가저장되는변수이름 : 배열 이름) {
    ...  value ...
}
```

For each문 이용

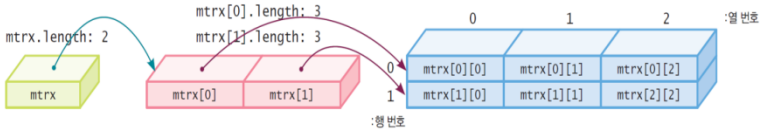
◆ 다차원 배열

- 이차원 배열
  - 행과 열의 구조로 표현할 수 있어 테이블 형태의 구조를 표현하는데 편리
  - 첫 번째 대괄호에는 배열의 행 크기, 두 번째는 배열의 열 크기를 저장

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

다음과 같은 구조를 표현하는 배열 생성 방법

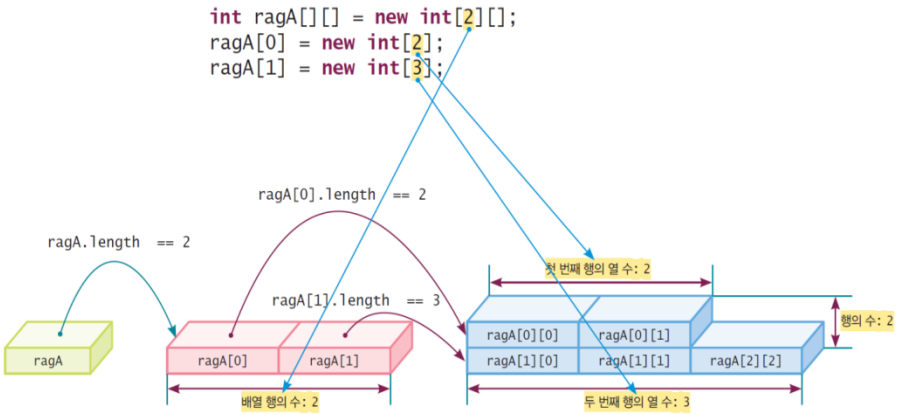
```
int mtrx[][];
mtrx = new int[2][];
mtrx[0] = new int[3];
mtrx[1] = new int[3];
```




```
int [][]mtrx;   //int mtrx[][]; 와 동일
int []mtrx[];   //int mtrx[][]; 와 동일
```

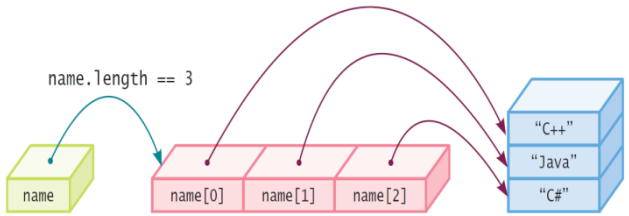
- 이차원 배열 원소 참조와 출력
  - Int mtrx[] = new int[2][3];
    mtrx[0][0]=3; mtrx[0][1]=5; mtrx[0][2]=7;
    mtrx[1][0]=7; mtrx[1][1]=2; mtrx[1][2]=8;
  - For문을 이용하여 이차원 배열 원소 출력은 행을 나타내는 제어변수 i로 외부 반복을 i<mrtx.length 조건으로 반복하고 열을 나타내는 제어변수 j로 내부 반복을 j<mrtx[i].length조건으로 원소 mrtx[i][j]를 참조

- 레기드 배열
  - 행마다 열의 수가 들쑥날쑥한 모양의 배열

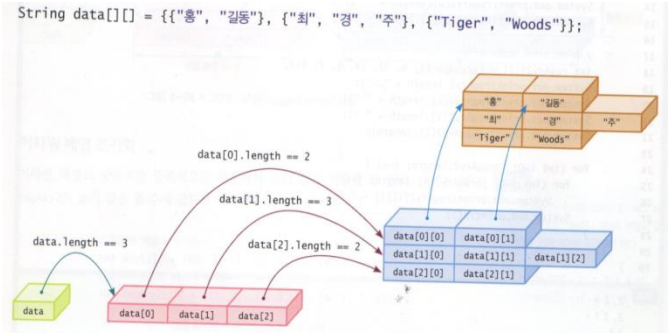


- 이차원 배열의 초기화
  - Int ary[][] = {{2, 3, 5}, {2, 6, 7}, {4, 2, 9}};->이차원 배열 초기화
  - Int ragAry2[][] = {{2, 3}, {3, 6, 9}, {4, 5, 7, 8}};->레기드 이차원 배열 초기화
  - Int ragAry2[][]; ragAry2 = new int[][] {{2, 3}, {3, 6, 9}, {4, 5, 7, 8}}->이차원 배열 선언 이후 초기화
- 참조형 원소를 위한 배열
  - 배열의 원소는 모든 자료형이 가능하므로
  - 참조형 배열도 가능

```
String name[] = {"C++", "Java", "C#"};
```



- 문자열을 위한 이차원 배열, 이차원 배열을 위한 for each




```
String data[][] = {{"홍", "길동"}, {"최", "경", "주"}, {"Tiger", "Woods"}};
for (String ary[] : data) {
    for (String value : ary)
        System.out.print(value + " ");
    System.out.println();
}
```

반복 시 저장될 일차원 배열을 위한 변수

반복 시 저장될 배열 원소를 위한 변수

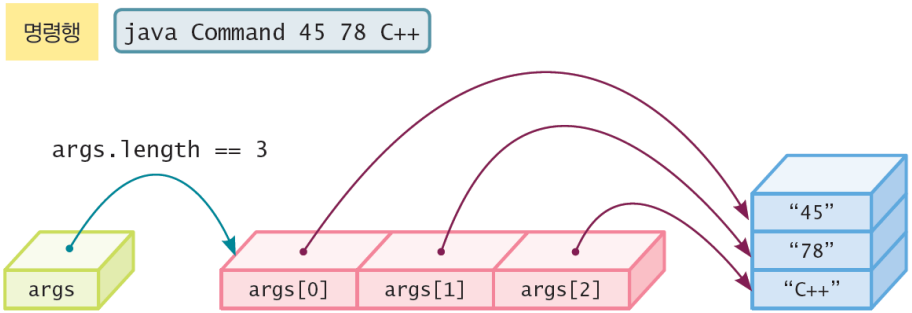
◆ **배열 크기 지정**

Ex) Int size=4;
Int score[]=new int[size];

- 자바에서 배열 크기를 상수뿐 아니라 변수로도 저장 가능

◆ **명령형 인자**

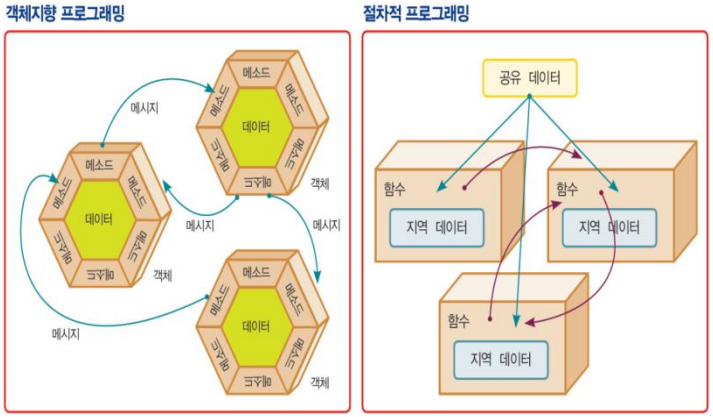
- 자바 프로그램 실행 시 인자를 받아 처리하는 것
- 프로그램 Command를 실행하면서 뒤에 여러 개의 인자를 입력하면 문자열 배열 arge에 저장되어 프로그램에 전달
- 쉼표나 탭과 같은 구분자로 구분/정수나 실수의 숫자 형태도 모두 문자열로 전달

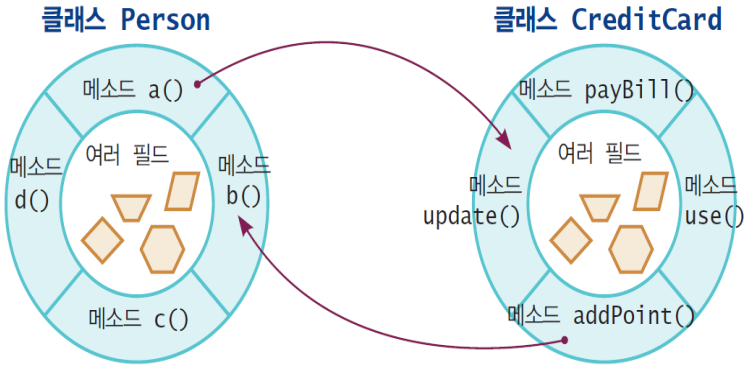


5.객체지향과 클래스

◆ 객체지향 프로그래밍의 이해

- 객체지향 프로그래밍 방식
  - 객체지형 프로그래밍 언어의 원조는 시뮬라
  - 클래스라는 개념을 처음으로 도입
  - 클래스를 생성하고 클래스로부터 객체를 만들어 객체간의 상호작용을 이용하여 주어진 문제를 해결
  - 데이터인 필드와 절차인 메소드를 하나로 묶은 클래스 단위의 프로그램
- 절차적 프로그래밍 방식
  - 데이터를 정의하고 데이터를 처리하는 함수로 구현하는 방식
- 필드와 메소드
  - 객체 : 현실세계의 사물이나 시스템에서 이용하기 위해 현실 세계를 자연스럽게 표현하여 이용할 수 있도록 만든 소프트웨어 모델
  - 객체는 속성과 행동으로 구성
  - 속성(필드) : 객체의 특성을 표현하는 정적인 성질
  - 행동(메소드) : 객체 내부의 일을 처리하거나 객체들간의 서로 영향을 주고 받는 동적인 일을 처리하는 단위





- 객체와 클래스
  - 클래스 : 객체를 만들기 위한 모형이자 틀
  - 객체는 클래스의 구체적인 하나의 실례
    -> ex) 붕어빵 틀(클래스), 붕어빵(객체)

◆ **추상화과 캡슐화**

- 추상화
  - 현실세계의 사실에서 주어진 문제의 중요한 측면을 주목하여 설명하는 방식
  - 객체지향 언어에서 클래스를 이용함으로써 속성과 행동을 함께 추상화의 구조에 넣어 보다 완벽한 추상화를 실현
  - 객체지향 언어는 추상화 과정을 통해 클래스를 생성
  - 추상화 과정 : 실세계의 객체에서 불필요한 부분을 제거하여 필요한 부분만을 간결하고 이해하기 쉬운 클래스로 만드는 작업
- 캡슐화
  - 객체와 객체간의 의사 소통을 위한 정보만을 노출시키고 실제 내부 구현 정보는 숨기는 원리
  - 캡슐화 과정에서 클래스 내부 구현을 외부에 숨기는 정보 은닉이 발생
  - 클래스의 단위의 내부 기능 중 일부는 외부에 공개되어 다른 객체와 메시지 전달과 수신을 하며 외부와의 통로역할

◆ **상속과 다형성**

- 상속
  - 객체지향의 가장 핵심이 되는 개념으로 프로그램을 쉽게 확장할 수 있도록 해주는 강력한 수단
  - 객체지향만의 고유의 특징
  - 상위클래스와 하위클래스 간의 관계가 객체지향의 상속의 개념
  - 중복적인 코드를 줄이고 보다 간편히 나머지 클래스를 구현
- 상속의 장점
  - 클래스를 계층적으로 체계화할 수 있으며 기존의 클래스로부터 확장이 쉽다
  - 공통의 특성을 하위 클래스마다 반복적으로 기술하지 않고 한번만 기술하기 때문에 중복을 줄여 재사용성의 효과

- 다형성
  - 외부에 보이는 모습은 한가지 형태이지만 실질적으로 쓰이는 기능은 여러 가지 역할을 수행한다는 의미
  - 오버로딩이나 오버라이딩

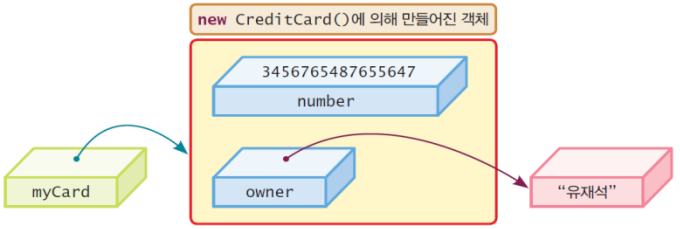
◆ 클래스의 필드 구현

- 필드 선언 시 자료형 앞에는 필드 특성을 표현하는 지정자인 키워드를 위치시킴
- 클래스의 정의에서 초기값이 없는 경우 필드 기본 값 저장됨

특성 종류	키워드	의미
상수	final	필드를 수정될 수 없는 상수로 한정
접근 지정자	public protected private	필드의 참조권한의 종류를 지정
정적	static	필드의 소속을 클래스로 한정

- 객체 생성과 필드 참조
  - 구현된 클래스를 이용하여 객체를 생성하기 위해서는 키워드 new이후에 CreditCard()와 같은 생성자를 호출
  - 참조형 변수인 myCard를 사용하여 참조 가능한 필드 owner와 number를 참조하려면 myCard.owner와 같이 참조연산자.을 사용

```
CreditCard myCard = new CreditCard();  자료형(클래스 이름)  변수  =  new  클래스 이름();
myCard.number = 3456_7654_8765_5647L;  변수이름.필드이름
myCard.owner = "유 재석";
```



◆ 클래스의 메소드 구현

- 메소드 구현
  - 신용카드를 사용하는 메소드 use()
  - 매월 또는 수시로 카드 비용을 지불하는 메소드 payBill()

```
public void use(int amount) {
    balance += amount;
}
public void payBill(int amount) {
    balance -= amount;
    addPoint(amount);
}
private void addPoint(int amount) {
    point += amount/1000;
}
```

지정자 반환형 메소드이름(인자목록) {
    구현;
}

addPoint(amount); → 구현한 메소드 addPoint()를 호출

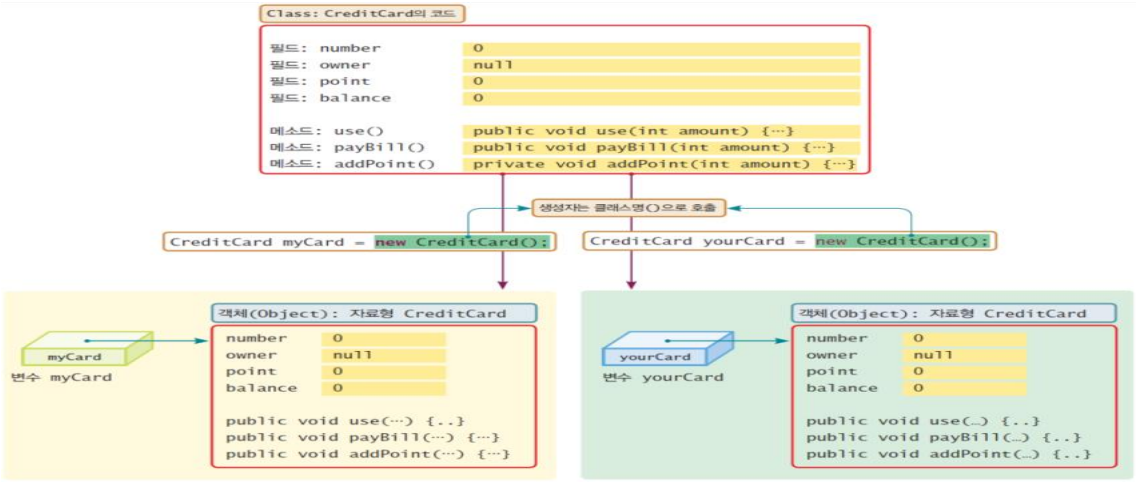
amount/1000; → 1000원에 1포인트를 추가하기 위한 연산식

- 메소드 지정자
  - 메소드 구현 시 반환형 앞에는 메소드 특성을 표현하는 지정자인 다양한 키워드를 위치시킴

특성 종류	키워드	의미
메소드 재정의 제한	final	하위 클래스에서 메소드를 더 이상 재정의할 수 없도록 한정
접근 지정자	public protected private	메소드의 참조권한의 종류를 지정
정적	static	메소드의 소속을 클래스로 한정
추상	abstract	구현이 없는 추상 메소드 지정
동기	synchronized	다중 스레드에서 메소드 동기화 지정

◆ 객체 생성과 참조

- 클래스로 부터 객체 생성



- 객체의 필드와 메소드 참조
  - 참조 연산자.를 사용하여 참조 변수의 필드와 메소드를 사용
    ->객체변수.필드 이용하여 객체의 필드에 접근
  - 구현된 클래스가 참조 가능하면 어느 클래스에서나 자료형으로 사용될 수 있으며 객체 생성도 가능
  - 접근 지정자가 private한 필드는 참조X/ 접근 지정자가 public한 필드는 참조

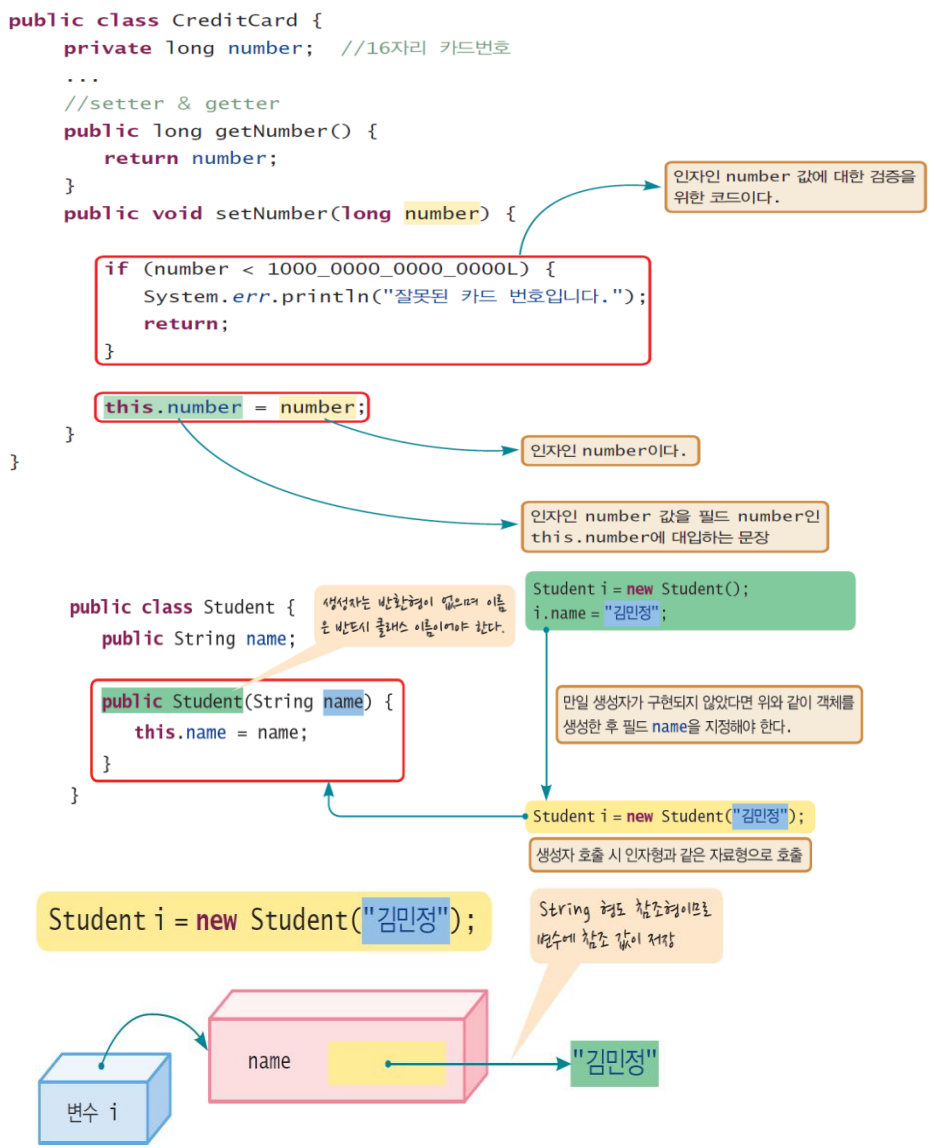
◆ 필드 참조 메소드 getter와 setter 구현

- 필드의 값 저장하고 반환하는 메소드를 각각 setter와 getter
- 접근 지정자가 private과 같이 외부에서 바로 참조 할 수 없는 필드에 대해 getter와 setter를 생성
- getter와 setter의 접근 지정자는 주로 public
- Setter의 구현 문장 this.number=number; 에서 this는 객체 자신을 의미하는 키워드

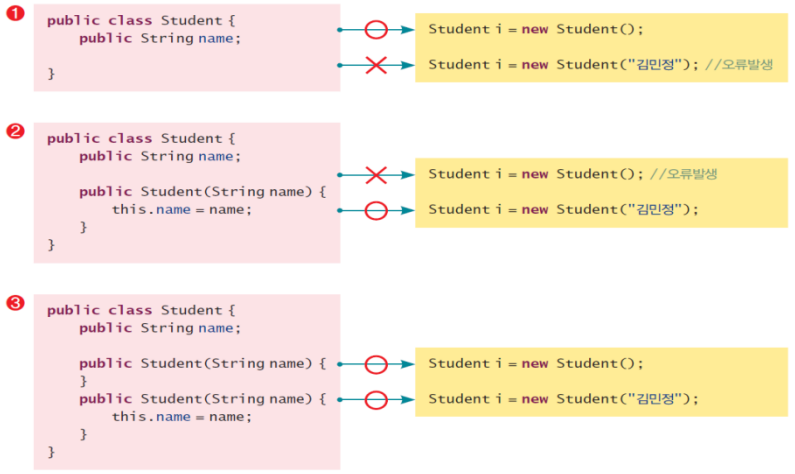
◆ 생성자

- 객체가 생성될 때 필요한 작업을 수행하는 특별한 메소드
- 일반 메소드와는 달리 반환형을 기술하지 않으며 이름은 반드시 클래스 이름과 같아야 한다
- 주로 접근 지정자 public을 사용

- 객체 생성을 위한 생성자 호출
  - 객체를 만들기 위해서는 new 이후에 생성자의 호출이 필요



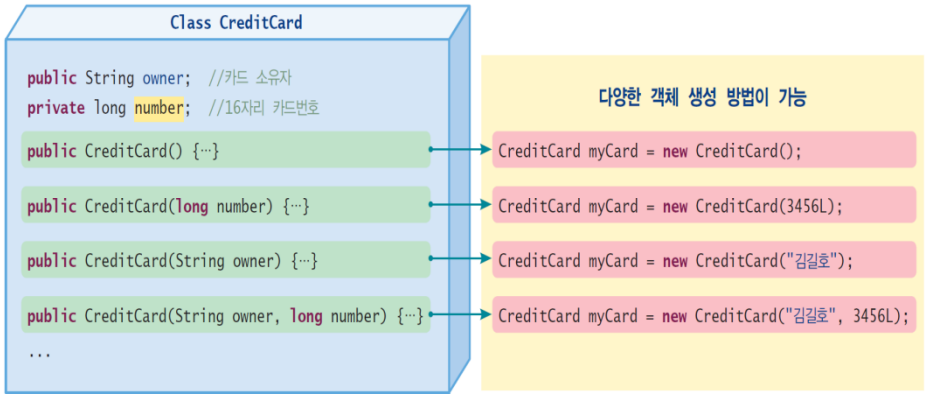
- 기본 생성자
  - 인자가 없는 생성자
  - 생성자가 전혀 구현되지 않은 클래스는 기본 생성자를 호출하여 객체를 생성



- 기본생성자의 구현
  - 클래스에서 인자가 있는 생성자가 적어도 하나 구현되었다면 더 이상 기본 생성자는 자동으로 사용할 수 없다
  - 인자가 있는 다른 생성자가 구현된 클래스에서 기본 생성자를 사용하려면 기본 생성자도 직접 구현

◆ **필요한 여러 생성자 구현**

- 생성자 오버로딩 : 하나의 클래스에서 인자가 다르면 생성자를 여러 개 만들 수 있다
  ->생성자에서 인자가 다르다는 것은 인자의 수가 다르거나 인자 수가 같더라도 인자의 자료형 순서가 다른 것을 의미



◆ 자기 자신의 다른 생성자 호출

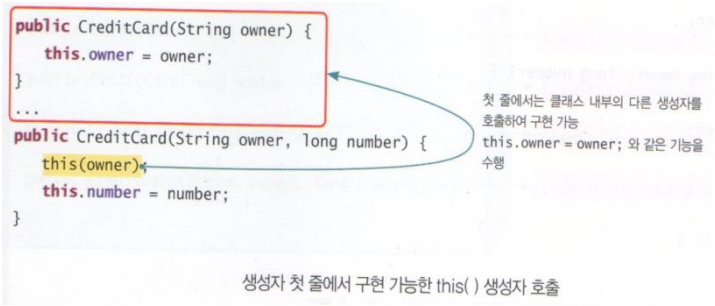
- this(...)는 구현된 자기 자신의 다른 생성자를 호출
- 생성자 구현에서 첫 줄에는 this(인자)
- this(...)는 두 번째 줄 이상에서는 절대 사용할 수 없다

```
public Account(String owner) {
    this.owner = owner;
}
public Account(String owner, long balance) {
    this(owner);
    this(balance); //오류 발생
}
```

다음에서 첫 줄이 아닌 this(…)는 모두 문법 오류 발생

```
public Account(String owner, long balance) {
    this(owner);
    this(balance); //오류 발생
}
```

```
public Account(String owner, long balance) {
    this.owner = owner;
    this(balance); //오류 발생
}
```



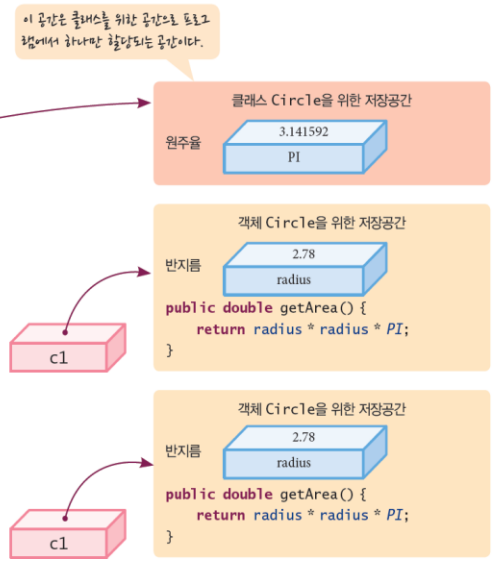
◆ 정적 필드와 메소드를 위한 키워드 static

- 키워드 static은 필드나 메소드의 소속을 클래스로 제한하는 키워드
- Static을 사용한 정적 변수나 정적 메소드는 클래스 변수와 클래스 메소드라 한다
- Static이 없는 변수와 메소드는 비정적으로 객체 변수, 객체 메소드라한다
- 정적 필드 PI는 클래스에 소속된 저장공간이 하나만 존재하는 변수로 할당되는 객체에는 저장공간이 없다

```
public class OldCircle {
    public double radius;

    //현재 반지름을 사용하여 원의 면적을 구하는 메소드
    public double getArea() {
        return radius * radius * 3.14;
    }
}
```

```
public class Circle {
    public double radius;
    public static double PI = 3.141592;

    //생성자 구현
    public Circle(double radius) {
        this.radius = radius;
    }

    public double getArea() {
        return radius * radius * PI;
    }
}
```

```
Circle c1 = new Circle(2.78);
System.out.println(c1.getArea());

Circle c2 = new Circle(5.25);
System.out.println(c2.getArea());
```



- 정적 필드 참조 방법
  - 정적 필드는 Circle.PI와 같이 클래스이름.정적필드로 참조하는 방법이 원칙이지만 c1.PI나 c2.PI와 같이 객체이름으로도 참조 가능

◆ **지역변수의 상수**

- 변수 선언 시 저장된 값을 더 이상 수정할 수 없도록 하려면 변수 선언 시 자료형 앞에 키워드 final을 명시

```
public static void main(String[] args) {
final int maxSize = 5;
//maxSize = 8; ->오류 발생
}
```

지역 변수의 상수

- 필드의 상수
  - 소속 변수인 필드를 상수로 선언하기 위해 final을 사용 할 경우, static도 함께 사용하여 정적으로 하는 편이 좋다

```
public class Circle {
public double radius;
public static final double PI = 3.141592;
....
}
```

->필드를 final로 상수 만들기

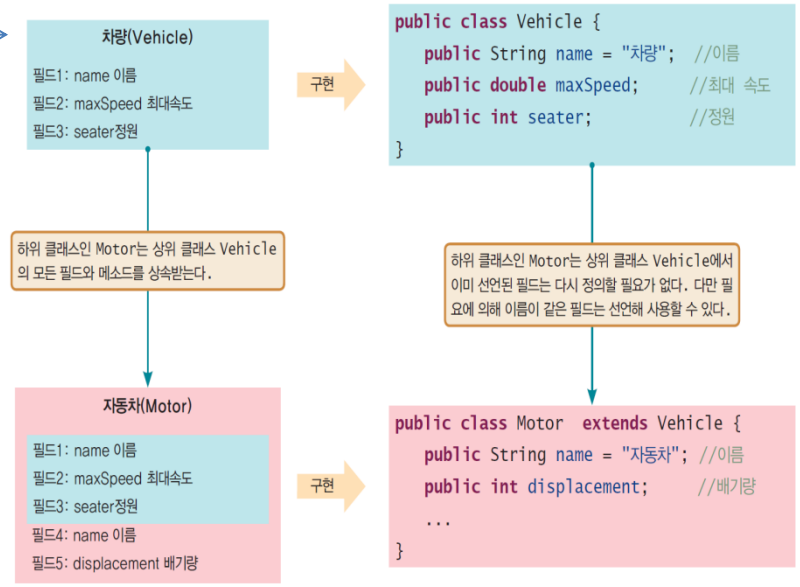
6.상속과 다형성

◆ 상속의 정의

- 하위클래스는 상위 클래스의 특징인 필드와 메소드를 그대로 물려받을 수 있는 특성
- 키워드 extends
  - 두 클래스 A와B에서 [A는 B이다]와 같이[이다 관계]가 성립하면 B는 상위 클래스, A는 하위 클래스 관계로 규정
  - 상위 클래스는 슈퍼 클래스 또는 부모 클래스, 기본 클래스라고도 부르며 하위 클래스는 서브 클래스 또는 자식 클래스,유도 클래스라고 부른다
  - 하위 클래스 정의에서 키워드 extends를 사용하여 ***하위 클래스이름 extends 상위 클래스이름***으로 상위와 하위 클래스의 관계를 규정
  - 최상위 클래스는 무조건 자바가 제공하는 클래스 Object를 부모 클래스로 갖음

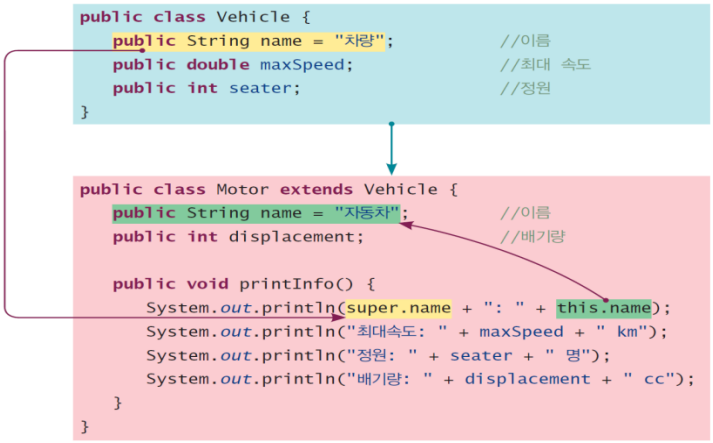
◆ 상속과 구현

- 하위 클래스에서는 멤버인 필드와 메소드를 다시 구현 없이 사용



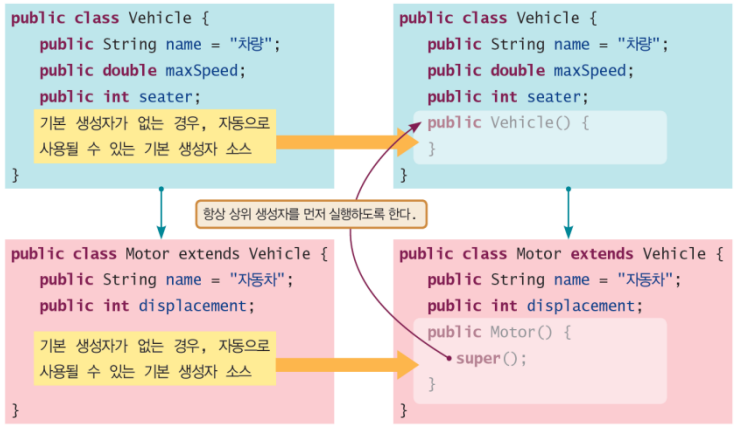
◆ this와 super

- this는 객체 자신을 의미하는 키워드
- super는 상위 객체를 의미하는 키워드

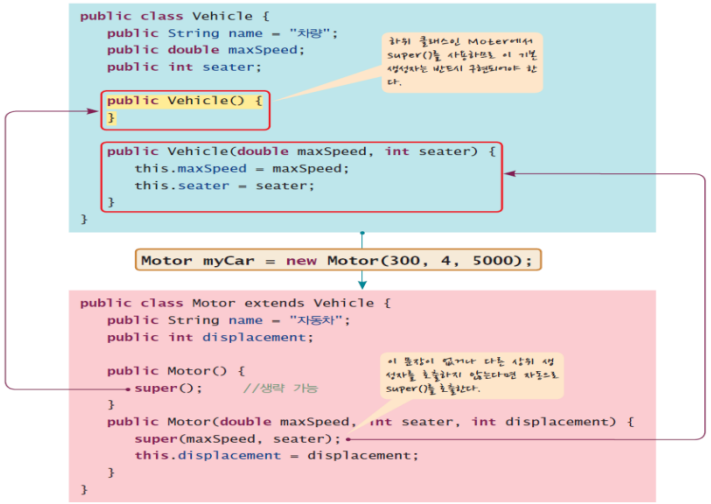


◆ 상위 생성자 호출

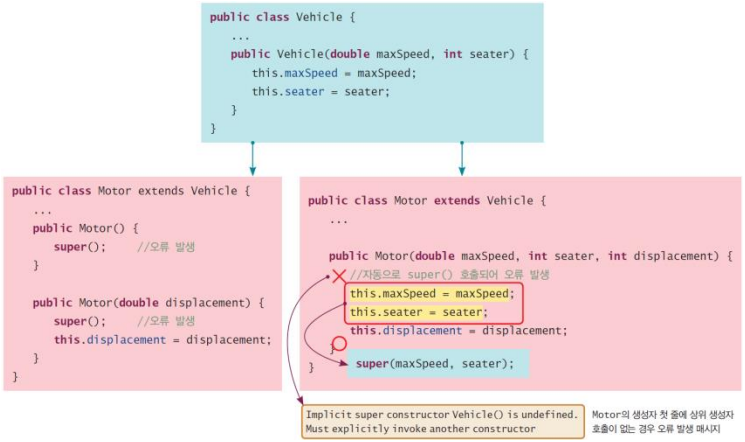
- 생성자 구현 첫 줄에서의 super()
  - 성생자의 첫 줄에서 상위 생성자를 호출하는 super() 또는 super(..)를 명시적으로 호출하지 않는다면 첫 줄에서 무조건 자동으로 super()를 호출
  - super()는 상위 클래스의 기본 생성자를 호출하는 문장
  - 상위 객체를 위한 필드와 메소드가 먼저 생성된 후 하위 객체가 생성



- 상위 클래스에서 기본 생성자 구현의 중요성
  - 사용자가 직접 구현하는 생성자에서 첫 줄이 상위 생성자의 호출인 super() 또는 super(인자)가 아니면 자동으로 기본 생성자 super()를 호출



- 상위 클래스에서 기본 생성자가 없는 경우의 문제
  - 인자가 있는 생성자를 구현한다면 더 이상 기본 생 성자는 구현 없이 자동으로 사용 될 수 없다



◆ 접근지정자

- 클래스 접근 지정자
  - 클래스 접근 지정자는 public과 [default] 방식

    ->접근 지정자를 기술하지 않는 default 클래스는 동일한 패키지의 다른 클래스에서만 사용 가능 따라서 default 클래스는 package 클래스라고 부른
- 필드와 메소드의 접근 지정자

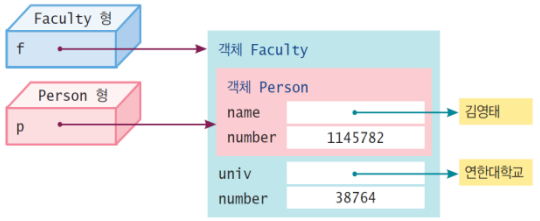
공개적 순위	키워드	클래스 내부	동일 패키지		다른 패키지	
			하위 클래스	일반클래스	하위 클래스	일반클래스
1	public	O	O	O	O	O
2	protected	O	O	O	O	X
3	default	O	O	O	X	X
4	private	O	X	X	X	X

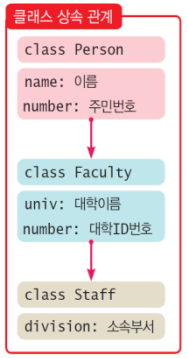
◆ 다형성

- 업 캐스팅
  - 하위 객체는 상위 클래스형 변수에 대입 가능
  - 상위로의 자료형 변환이며 자동으로 수행
  - 변수로는 하위 객체의 멤버를 참조할 수 없는 제약이 따름

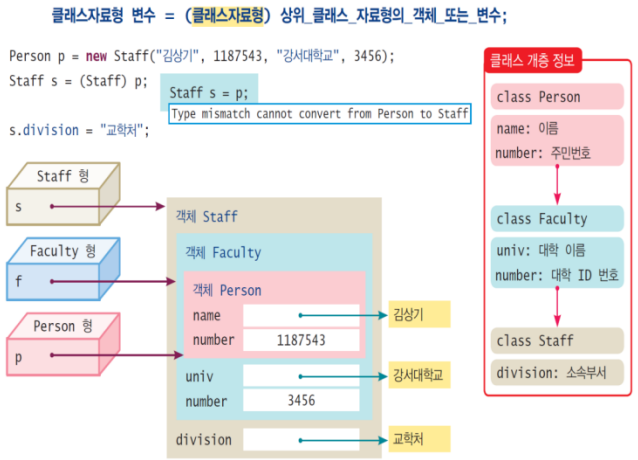
```
클래스자료형 변수 = 하위_클래스_자료형의_객체_또는_변수;
```

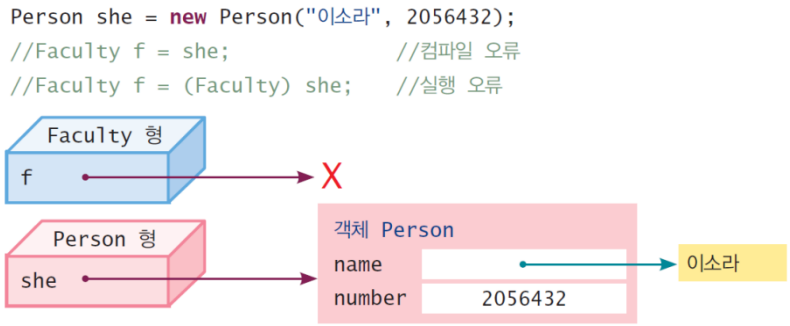
```
Faculty f = new Faculty("김영태", 1145782, "연한대학교", 38764);
Person p = f;
```





- 다운 캐스팅
  - 상위 클래스 형을 하위 클래스 형으로 변환하는 다운 캐스팅은 반드시 명시적인 형변환 연산자(하위 클래스)가 필요
  - 컴파일 시간에 상속 관계만 성립하면 다운 캐스팅은 가능하나 실제 객체가 할당되지 않았다면 실행 시간에 오류 발생





◆ **연산자 instanceof(객체 확인 연산자)**

- 첫 번째 피연산자 객체변수가 참조하는 객체가 실제 두 번째 피연산자 클래스 이름이면 true, 아니면 false를 반환

```
Person she = new Person("이소라", 2056432);
if (she instanceof Staff){
    Staff st1 = (Staff) she;
} else {
    System.out.print("she는 Staff 객체가 아닙니다. ");
}
```

사용법: **객체변수 instanceof 클래스이름**

◆ **오버라이딩**

- 메소드 오버라이딩
  - 상위 클래스의 동일한 메소드를 하위 클래스에서 다시 정의하는 것
  - 메소드의 재정의또는 메소드대체라고 표현

- 목적 : 상위 클래스에서 이미 정의한 메소드를 다시 수정하지 않고 하위 클래스에서 좀 더 보완 수정하거나 완전히 새로운 것으로 대체하기 위한 방법

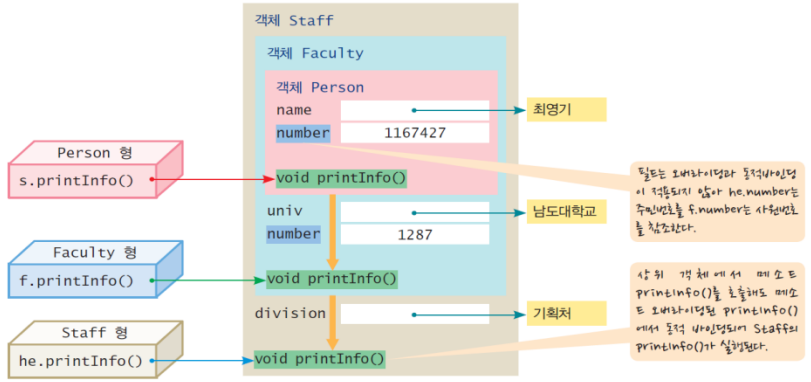
- 오버라이딩 조건
  - 메소드의 반환 값과 메소드 이름, 매개변수는 반드시 같아야 한다
  - 접근 지정자는 하위 클래스의 메소드가 보다 공개적이어야 한다

    ->상위 메소드가 default이면 하위 메소드는 public,protected,default만 가능
  - 메소드 수정자 final, private인 메소드는 오버라이딩될 수 없다
- final 클래스와 final 메소드
  - 클래스 앞의 final은 클래스가 더 이상 상속되지 못한다는 의미의 키워드
  - 메소드 반환형 앞의 지정자 final은 더 이상 하위 클래스에서 메소드 오버라이딩이 허용되지 않도록 하는 키워드

◆ 동적 바인딩

- 오버라이딩 메소드 호출
  - 가장 하위 객체의 메소드 실행
  - 실행 단계에서 메소드 호출시 객체의 type을 보고 적절한 메소드를 호출

```
Person he = new Staff("최영기", 1167429, "남도대학교", 1287, "기획처");
he.printInfo();
Faculty f = (Faculty) he;
f.printInfo();
Staff s = (Staff) he;
s.printInfo();
```



◆ 메소드 오버로딩

- 클래스 내부에서 인자가 다르나 이름이 같은 메소드가 여러 개 정의 될 수 있는 특징
- 반환값이나 지정자가 다르더라도 인자가 같으면 더 이상 동일한 이름으로 메소드를 만들 수 없다

- 정적메소드와 비정적 메소드의 오버로딩
  - 정적과 비정적과 무관
  - 메소드의 이름이 같으며 인자가 다르면 가능
  - 클래스에 소속된 정적 메소드 내부에서는 객체에 소속된 변수와 메소드를 참조 할 수 없다
  - 정적 메소드 내부에서는 비정적 필드와 비정적 메소드를 참조 할 수 없다
  - 정적 메소드 내부에서는 this와 super를 사용할 수 없다

◆ 추상클래스

- 클래스 간의 계층구조에서 상위에 존재하여 하위 클래스를 대표하는 클래스

1. 직접 홀로 객체화될 수 있다 ->생성자를 사용하여 객체를 생성X
2. 다른 클래스에 의하여 상속되어야 한다 ->하위 클래스가 없는 추상클래스는 의미가 없다
3. 하위 클래스가 있어야 하므로 추상 클래스 구현 시 클래스 앞에 키워드 final이 올 수 없다

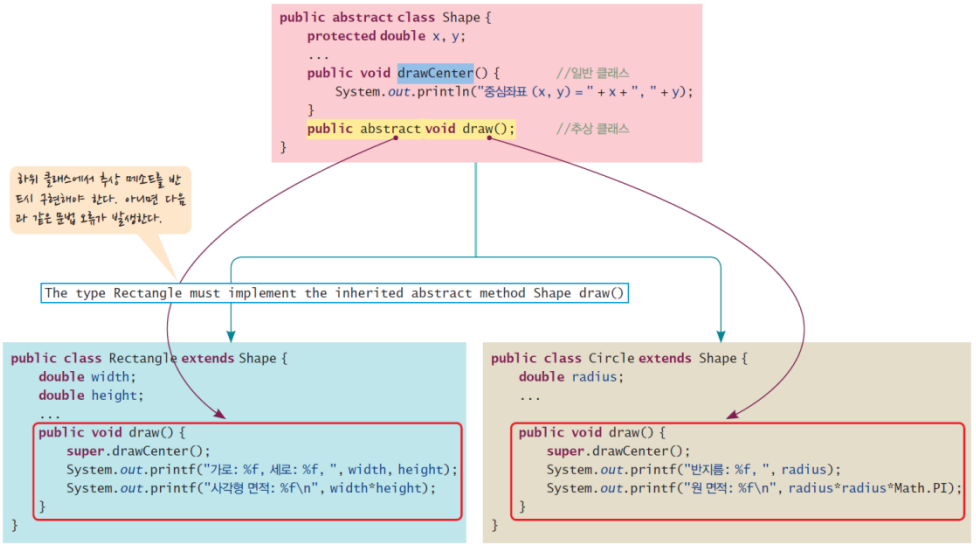
- 추상화 클래스 지정자 abstract
  - 클래스 정의 시 키워드 class 앞에 abstract 키워드를 기술하여 구현

```
public abstract class Shape {
   protected double x, y;

   public Shape(double x, double y) {
      this.x = x;
      this.y = y;
   }
   ...
}
```

```
Shape s = new Shape(3.1, 4.5); //객체화 오류
cannot instantiate the type Shape
```

- 추상 메소드와 추상 클래스
  - 추상 메소드
    - 메소드 몸체가 없는 메소드
    - 메소드 오버라이딩에 지정할 수 없는 접근 지정자 private와 오버라이딩 제한 지정자 final이 사용될 수 없다
    - 적어도 하나 이상의 추상 메소드를 가진 클래스는 반드시 추상이어야 한다

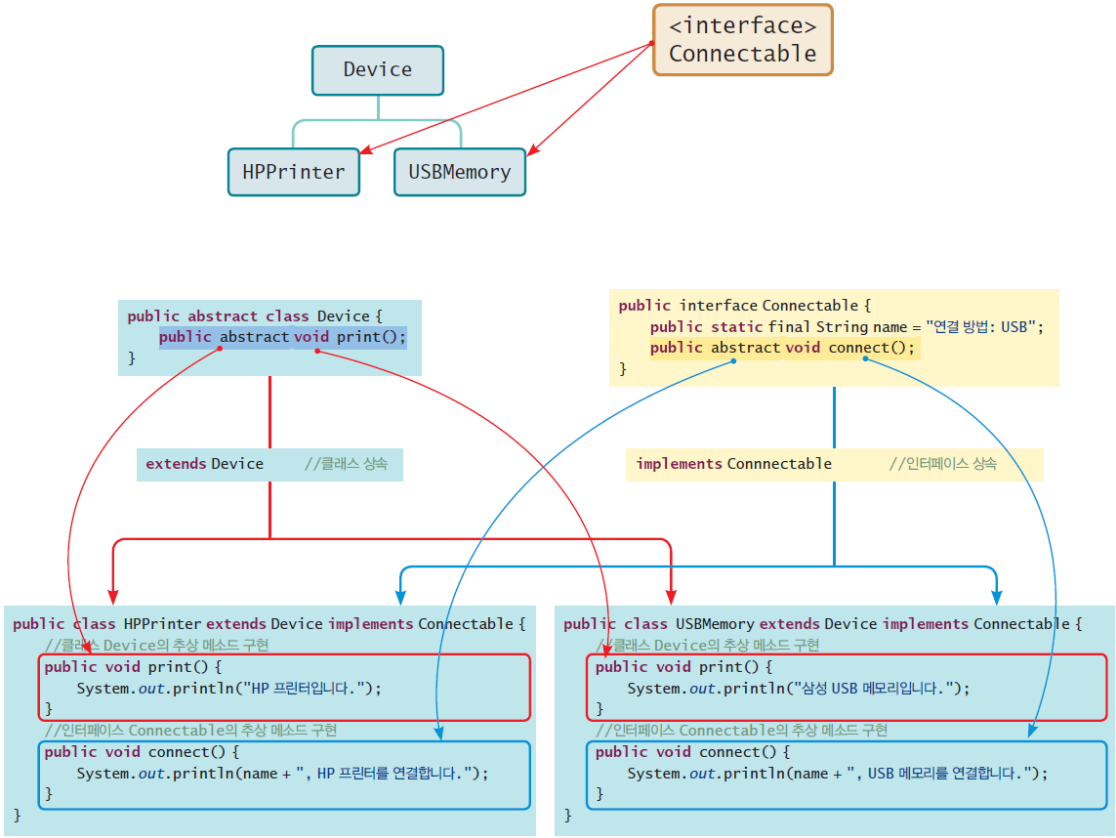
- 하위 클래스에서 추상 메소드 구현



◆ 인터페이스

- 해야 할 작업의 구체적 구현 없이 기능만 선언한 클래스
- 하위 클래스가 수행해야 하는 메소드와 필요한 상수만을 미리 추상적으로 정의해 놓은 특별한 메소드
- 추상 클래스보다 더 추상적인 클래스로 여러 인터페이스를 상속받는 다중 상속을 지원
- 일반 필드의 선언을 허용하지 않으며 public abstract final을 사용한 상수만 정의
- 인터페이스 키워드 interface
  - 인터페이스의 구현에서 class 대신 키워드 interface를 사용하며 구현 없이 기능만 정의되는 메소드 public abstract의 추상 메소드로만 정의
- 인터페이스의 상속
  - 인터페이스 구현에서 인터페이스들 간의 상속은 키워드 extends를 사용하며 다중 상속인 경우 상위 인터페이스 여러 개를 쉼표로 구분하여 나열
  - 인터페이스를 상속 받는 하위 클래스는 정의라 때는 키워드 implements를 사용
  - 인터페이스를 상속받은 클래스는 상위 인터페이스에서 정의한 모든 추상 메소드를 구현

- 인터페이스 설계


<interface>
Connectable
Device
HPPrinter
USBMemory
public abstract class Device {
public abstract void print();
}
public interface Connectable {
public static final String name = "연결 방법: USB";
public abstract void connect();
}
extends Device //클래스 상속
implements Connnectable //인터페이스 상속
public class HPPrinter extends Device implements Connectable {
//클래스 Device의 추상 메소드 구현
public void print() {
System.out.println("HP 프린터입니다.");
}
//인터페이스 Connectable의 추상 메소드 구현
public void connect() {
System.out.println(name + ", HP 프린터를 연결합니다.");
}
}
public class USBMemory extends Device implements Connectable {
//클래스 Device의 추상 메소드 구현
public void print() {
System.out.println("삼성 USB 메모리입니다.");
}
//인터페이스 Connectable의 추상 메소드 구현
public void connect() {
System.out.println(name + ", USB 메모리를 연결합니다.");
}
}