객프 중간 정리

소프트웨어 특징

- 제조가 아닌 개발 :: 개인의 능력에 따라 차이가 큼
- 소모가 아닌 품질 저하 :: 닳지 않고 시간이 지나도 고장 빈도가 높지 않다. 사용 시작단계부터 사용자 요구가 발생

소프르퉤어 개발 생명주기

- 소프트웨어 만들기 위해 계획 단계에서 유지보수 단계에 이르기까지 일어나는 일련의 과정
- 계획 - 분석- 설계 - 구현 - 테스트 - 유지보수

프로그램 과 프로그래밍

- 프로그램
  - 컴퓨터에서 실행될 때 특정 작업을 수행하는 일련의 명령어모음
  - 0과 1로 고성된 이진 형식 파일로 저장
- 프로그래밍
  - 일련의 절차와 규칙을 프로그래밍언어를 통해 프로그램으로 구현하는 기술

프로그래밍 언어

- 저급언어일수록 기계어 가깝고 고급일 수록 사람이 사용하는 언어에 가까움

객체 지향 프로그래밍

- 여러개의 독립된 단위, 즉 '객체들의 모임'으로 파악하고자 하는 패러다임
- 추상화 클래스 상속

컴파일

- 기계어로 변환하는 과정
- **컴파일러** 라는 SW가 과정을 수행

JAVA

1. 단순하다
2. 객체지향언어
3. 함수형 코딩 지원
4. 플랫폼 독립적
5. 분산처리 지원

6. 견고하다
7. 안전하다
8. 이식성이 좋다
9. 멀티스레딩을 지원한다
10. 동적이다

- 어떤 버전이나 환경에서든 하위호환성 보장

Java 가상머신 (JVM)

- 바이트 코드로 컴파일된 프로그램을 실행할 수 있도록 하는 가상머신
- 플랫폼 종속성 문제 해결
- JAVA 위에서 실행
- 하드웨어나 운영체제달라도 JVM과 코드만 있으면 실행 가능
- 초기 모바일환경에서 강정

JAVA 바이트 코드

- java 가상 머신의 집삽
- JAVA 소스 코드 컴파일 시 결과물
- 다양한 플랫폼에서 일관되게 실행될 수 있도록설계

자바 프로그램 구조

- 클래스 파일 + 패키지(폴더)

문법 요소

1. 클래스
    1. 프로그램 개발하는 단위
2. 메서드
    1. 수행할 작업을 나열한 코드의 모임
3. 실행문
    1. 변순 선언, 값 저장, 메서드 호출 등의 작업 지시하는 코드
4. 주석문
    1. 프로그램에 덧붙이는 설명문
5. 식별자
    1. 클래스 변수 등 이름

식별자

- 클래스, 변수, 메서드 등을 구별 짓기 위한 이름
- 대소문자 구분, 한글 가능,

- 변수 메서드 :: 소문자, 두번째 단어 대문자
- 클래스 : 첫 자는 대문자 두번째 단어 대문자
- 상수 : 대문자 , _(언더바) 로 구분

상수

- 변경 불가능한 데이터 담는 변수
- **final** 사용
- 초기화 되면 덮어쓸 수 있음
- 선언만 하면 오류발생

리터럴

- 미리 정의된 값
- 변하지 않는 데이터 (PI 등)

```
double half = 0.5;
double half = 5E-1;
float pi = 3.141594F;
float pi = 3.141594; // 오류남
char c = 'A';
char c= 65;
char '\u0041;
char c = "A";  // 오류남
```

- 오류나는 것 파악

상수 vs 리터럴

- 상수 :: 프로그램 실행 중 항항 동일한 값을 가짐
- 리터럴 :: 프로그램에서 미리 정의한 값

강제 타입 변환

- 기억 공간 크기 차이, 정밀도 차이로 인해 타입 변환시 일부조건에서 데이터 손실발생
- 자료형 큰 것 에서 작은것으로 갈 때 **데이터 손실 발생** 주의
- ex : Integer.MAX_VALUE, Byte.MIN_VALUE

표준 입출력 :

- System.in : 입력
  - Scanner 사용
    - next() :: String
    - nextLine() :: String :: **문장으로 끊음**(나머지는 다 공백으로 끊음)
    - nextBye() :: byte

        - nextInt() :: int
        - nextShort() :: short
        - nextLong() :: long
        - nextFloat() :: float
        - nextDouble() :: double
- System.out : 출력

```
String a = in.next();
String b = in.nextLine();

// 안 녕하세요. 저는 강주영
```

- 출력
    - 안
    - 녕하세요. 저는 강주영

비교 연산자

- 값의 일치 및 대소비교
- boolean 타입 결과값
- != == < 등

논리연산자

- 두개의 boolean 피연산자 값 결합하여 boolean 반환
- && || a ^ b 등

비트 연산자

- & : 둘 다 1이여야 1
- | : 둘 다 0일 때만 0, 나머지는 1
- ^ : 두 비트 다를 때 1 동일할 때 0
- ~ : 1 은 0 으로 0 은 1로

시프트 연산자

- <<
-
-

연산자 우선순위

연산자	설명
[ ], ., ( ), ++, --	배열 접근, 객체 접근, 메서드 호출, 후위 증가, 후위 감소
+x, -x, ++x, --x, ~(비트), !(논리)	부호 +/-, 선위 증가, 선위 감소, 비트 부정, 논리 부정
( ), new	타입 변환, 객체 생성
*, /, %	곱셈, 나눗셈, 모듈로
+, -	덧셈, 뺄셈
>>, <<, <<<	시프트
>, <, >=, <=, instanceof	비교
==, !=	동등 여부
&	비트 AND
^	비트 XOR
|	비트 OR
&&	조건 AND
||	조건 OR
? :	조건 연산
=, +=, -=, *=, /=, %=, &=, ^=, !=, <<=, >>=, >>>=	대입

switch-case 업그레이드 버전

1. 다중 case 레이블
2. 화살표 case 레이블
3. yield 예약어

- 가독성을 높힘
- **변수에 담겨서 return 이 아닌 경우는 꼭 switch괄호에 ; (세미콜론 필수)**

1. 다중 case 레이블

```
switch(dya){
    case "mon", "tues", "wed" :
    result =1;
    break;
}
```

2. 화살표 연산자

- switch 변수의 모든 가능한 값에 대해 case 레이블을 사용하거나, default 사용해야함

```
String type = switch(day){
    case "mon", "tues", "wed" -> "weekday";
    default -> "invalid";
};
```

- 여러개 실행문 사용할 땐 블록 사용

```
switch (day){
    case "mon", "tue", "wed" -> typeofDay = "weekday"; // 기본
    case "sat", "sun" -> {
        typeofDay = "Weekend";
        System.out.println("문장문장들");
    }
    default -> {
        System.out.println("문장ㅜㄴ장);
        typeofDay = "unknown";
    }
};
```

3. yield 예약어 :: 값을 반환하면서 switch 연산 종료

- 반환할 값이 없는 경우 yield 사용

```
String typeofDay = switch(day){
case "mon" -> "weekday";
case "saturday" -> {
    System.out.pritn("출력출력);
    yield "weekend";
    }
default -> {
    System.out.println("unknown day");
    yield "unknown";
    }
};
```

메서드

- 특정 연산을 수행하기 위해 실행문 모아둔 블록

1. 반드시 클래스 내부에서만 선언 가능
2. 객체를 통해서만 호출 가능
3. 오버로딩 O
4. 중복코드 줄이고 코드 재사용할 수 있다.

5. 코드 모듈화하여 가독성을 높여 프로그램 품질을 향상시킨다.
6. 인수 : 메서드 호출 시 **호출측**에서 제공하는 매개변수에 대응되는 값
7. **static 은 객체를 생성하지 않고 실행할 수있다**

메서드 시그니처

- JAVA에서 두 메서드의 이름이 동일하더라도 시그니처가 다르면 서로 다른 메서드 취급

메서드 오버로딩

- **메서드 이름이 동일하더라도, 매개변수 변수, 데이터 타입, 순서 중 하나라도 다르면 다름 메서드로 취급**

```
static int sum(int a, int b) {
  return a + b;
}

static int sum(int a, int b, int c) {
  return a + b + c;
}

static double sum(double a, double b) {
  return a + b;
}
```

객체

- SW에서의 객체 : 현실 세계의 객체를 **필드** 와 **메서드** 로 모델링한 것

객체 지향 프로그래밍

- 객체를 기반으로 하는 프로그래밍 패러다임
- 여러개의 독립된 단위, '객체들의 모임' 으로 파악하고자 하는 것
- 여러가지 객체 상호작용을 통해 문제 해결

객체와 클래스

- 클래스
  - 객체의 설계도
- 인스턴스
  - 클래스라는 틀로 만든 객체

- 인스턴스화
  - 클래스에서 객체를 생성하는 과정

객체

- 참조 변수가 없으면 두번 이상 사용 불가.
- 참조 되지 않은 변수는 가비지 컬렉터가 자동으로 수거한다.
- 객체 생성 :: 클래스이름 변수 = new 클래스이름();
  - new 연산자 :: 객체의 주소 반환

캡슐화

- 관련된 **필드**와 **메서드**를 하나의 캡슐처럼 포장하여 외부에서 알 수 없도록 감추는 것 --> 정보은닉
- 세부 구현을 숨겨 외부영향 줄인다.
- 외부로부터 특정 메서드 , 필드에 대한 접근 제한
- 중요한 정보만 외부에 공성

참조타입

```
Ball myBall = new Ball();
Ball yourBall = new Ball();
myBall = yourBall;
```

클래스 멤버 :: 필드와 + 메서드

클래스의 필드와 지역변수 차이

1. 필드 vv
   1. 기본 값이 존재
   2. 메서드 내부 제외한 클래스 전체에서 사용 가능
   3. 초기화 기본값이 있어초기화 안해도 가능
   4. static, final 가능
2. 지역변수
   1. 기본값 없어 반드시 초기화
   2. 선언된 블록 내부에서만 사용 가능
   3. 접근 제어자 사용 불가
   4. final만 가능

접근제어자

- 메서드, 필드, 클래스가 공개될 범위 지정
- private, public 키워드

- **캡슐화** 위하여 필요
-

접근자와 설정자

- private 으로 지정된 필드의 값과 반환하는 접근자와 값을 변경하는 설정자는 공개된 메서드
- 일반적으로 접근자 :: get, 설정자 :: set
- 필드 이름을 외부와 차단하여 독립시키기 때문에, 필드 이름 변경이나 데이터 검증 또한 가능

생성자

- 객체를 생성하는 시점에서 필드를 다양한 방식으로 초기화
- 클래스이름() {}

1. 생성자이름은 클래스 이름과 같다
2. 반환 타입이 없다
3. 생성자는 new 연산자와 함께 사용하며, 객체를 생성할 때 호출한다.
4. 생성자도 오버로딩 할 수 있다.

디폴트 생성자

- 모든 클래스는 최소한 하나의 생성자가 있다.
- 생성자를 선언하지 않으면, 컴파일러가 자동으로 디폴트 생성자 추가
- 생성자 오버로딩
  - 메서드와 같이 생성자 또한 오버로딩 가능
  - 다양한 매개변수를 갖는 여러개의 생성자 선언 가능

this 키워드

- 객체 자기 자신을 참조할 수 있도록 하는 키워드
- this() 키워드
  - 생성자에서 다른 생성자를 호출할 수 있도록 하는 키워드
  - 오버로딩 된 생성자에서 생기는 중복 코드 제거 가능
  - **다른 생성자 호출할 땐, 반드시 첫번째 줄에 작성해야한다.**
  - **부모생성자 호출 불가**

정적멤버 필요성

- 필드는 각 객체 간 독립적
- 하지만 **같은 클래스의 객체끼리 공유할 데이터가 필요해질 수 있음**

정적멤버와와 다른 변수들 비교

- static 키워드로 클래스 필드 공유할 수 있도록 지원 :: 모든 객체가 공유
- 인스턴스 변수 : static 키워드로 지정되지 않아, 공유되지 않은 필드로 인스턴스마다 자신의 필드 생성
- 정적변수 혹은 클래스 변수 : static 키워드로 지정하여 모든 인스턴스가 공유하는 필드

정적멤버 - 정적메서드

- 인스턴스와 무관하게 호출 가능
- **static** 키워드 사용
- **==객체 자신 가리키는 this 사용 불가==**
- 클래스이름.정적변수이름
- 클래스이름.정적메서드이름()
- **static 메서드에서 해당 메서드가 아닌 다른 메서드에서 static 메서드 아닌걸 호출 불가**

정적 블록

- static { . . . }
- 이는 한 번 실행된 후 재실행 하지 않는다. >> 객체를 생성하더라도 재실행 X

문자열

- 문자열 리터럴은 내부적으로 new String()을 호출하여 생성한 객체
- 내용이 같은 문자열 리터럴 이라면, 더이상 새로운 String 객체를 생성하지 않고 기존 리터럴 공유
    - ```
      String s1 = "안녕, 자바 !";
      String s2 = "안녕, 자바 !";
      ```
        - String 변수 s1과 s2의 객체 하나로 같이 씀

문자열 비교

1. == !=
    1. **문자열 내용 비교가 아닌, 동일한 객체인지 검사**
2. int compareTo(String s) :: 문자열을 사전 순으로 비교하여 정수 값 반환 :: 아스키코드
3. int compareToIgnoreCase(String s) :: 대소문자 무시하고, 문자열 사전순으로 비교
4. boolean equals(String s) :: 주어진 s와 현재 문자열 비교 후 T/F 반환
5. boolean equalsIgnoreCase(String s) :: 문자열 대소문자 구분 없이 비교하여 T/F 반환

문자열 조작 ⚠️(교재 확인)

- charAt(int dex) :: index가 지정한 문자 반환
- concat (String s) :: 문자열 s를 현재 문자열 뒤에 연결
- contains(String s) :: 문자열 s 포함하는지 조사
- length() :: 길이 반환
      ```

- int indexOf(String s) : 문자열 s가 있는 위치 반환
- substring(int index): index부터 시작하는 문자열 일부 반환
- toLowerCase() 모두 소문자 변환
- toUpperCase() 모두 대문자 변환
- isEmpty() :: 공백 포함하여 빈값인지 확인 :: 공백 있으면 false
- isBlank() :: 공백 포함 안하고 빈값인지 확인 ::공백 있어도 true
-
```
s1 = s1.concat(s2);
s1.toLowerCase();
s1.toUpperCase();
s1.length();
s1.charAt(1);
s1.isEmpty(); //Empty()는 공백도 값이라 판단.
s1.isBlank();

s1.repeat(10);
s1.trim().indexOf("v");
```
- s1.Empty()는 공백도 값이라 판단.

클래스의 정적 메서드

- format() :: 주어진 포맷에 맞춘 문자열 반환
- join() :: 주어진 구분자와 연결한 문자열을 반환
- valueOf() :: 기초타입이나 객체를 문자열로 반환

```
String version = String.format("%s %d", "JDK", 14);
System.out.println(version);//JDK 14
String fruits = String.join(",", "apple", "banana", "cherry");
System.out.println(fruits); // apple, banana, cherry
String pi = String.valueOf(3.14);
System.out.println(pi);//double 3.14 에서 String 3.14로 나옴옴
```

배열

- scores.length;
- int[] scores = { 100,200,300 };
- **int[] scores = new int[] {100,200,300};**
- int[] scores;
  scores = new int[] {100,200,300};

1. 잘못사용한 예시
   - int [] scores;

scores = {100,200,300};
- int [] scores[5]{12,3,4,5};

다차원 배열

- int[][] = {{100,200}, {300,400}, {500,600}};

동적 배열

- ArrayList 객체 생성

```
ArrayList<참조타입> 변수이름 = new ArrayList<>();
```

1. 참조타임 :: Integer, Long, Short, Float, Double, Charater, Boolean
2. 변수.add(데이터)
3. 변수.remove(인덱스번호)
4. 변수.get(인덱스번호)
5. 변수.size()

for each 반복문

```
int [] = one2five = {0,1,2,3,4,5};
int sum = 0;

//1. 기존
for(int x= 0;x<one2five.length ;x++)
    one2five[x]++;

for (int x:one2five)
    sum+=x;
```

메서드 인수로 배열 전달

- 배열 변수는 배열 주소를 전달 (이름 그대로주면 됨)
- 배열 변수 인수로 전달하면 주소 복사되어 전달
- 매서드 매개변수는 동일한 배열 객체 가리킴

```
int[] x = {0};
mathod1(x);

public void mathod1(int[] x){
```

```
    int i x[0];
}
```

가변개수 매개변수

- 한개의 가변 개수 매개변수만 사용 가능하다
- 매개변수 마지막에 위치
- 가변개수 인수 가진 메서드 호출하면, 내부적으로 배열 생성하여 처리한다.

```
public static void print(itn... v){
    int sum = 0;
    for(itn i : v){
        sum+=i;
    }
    System.out.println(sum);
}

printf(1);
printf(1,2);
printf(1,2,3);
printf(1,2,3,4);

//호출
1
3
6
10
```

객체의 배열

- 객체 배열은 객체를 참조하는 **주소를 원소로 구성**
- **초기값 :: null**
- Ball[] balls = new Ball[5];
  - 5개의 Ball 객체를 생성하는 것이 아닌, **5개의 Ball 객체를 참조할 변수 준비**
  - for(int i=0;i<Ball.lengh;i++)
    balls[i] = new Ball();

열거 타입 필요성

1. 제한된 수의 일이나 사건등에 대하여 숫자로 표현
    1. 일이나 사건에 대한 경우의 수가 많다면, 개발자 관점에서 불편
    2. 부여되지 않은 의미없는 숫자 => 컴파일러 알 수 없다
    3. 출력값이 의미 없는 숫자로 표현됨
2. 제한된 사건에 대하여 숫자 대신 상수를 정의해서 부여
    1. 숫자에 부여된 의미를 개발자가 알 수 있지만, 여전히 나머지 문제가 미결

열거타입

- 서로 연관된 사건들 모아 상수로 정의한 **java.lang.Enum** 클래스의 자식클래스

1. 선언
    1. enum 열거타입 이름 {상수목록}
    2. enum Gender {MALE, FEMALE}

열거타입 응용

1. **일종의 클래스 타입인 열거타입 역시 생성자, 필드, 메서드 가질 수 있다.**
2. **열거타입 상수는 생성자에 의한 인스턴스이다.**
3. **이때 생성자, 필드, 메서드와 열거타입 상수를 구분하기 위하여 다음과 같이 열거타입 상수 뒤에 반드시 세미콜론 추가해야한다.**

```
enum 열거타입 이름{
    열거타입상수1, 열거타입 상수2;
    //필드
    //생성자
    //메서드
}
```

상속의 필요성

- 공통된 특성을 띄는 서로 다른 클래스가 존재
- 상속이 없다면, 클래스마다 중복된 내용 작성

1. 중복코드 제거
2. 유지보수 용이

상속

- 자식 클래스는 부모클래스에서 물려받은 멤버를 그대로 사용하거나 변경할 수 있고, 새로운 멤버도 추가할 수 있다.
- 자식클래스는 대체로 부모 클래스보다 속성이나 동작이 더 많다.
- extends 사용

```
class SuperClass { }//부모
class SubClass extends superClass{}//자식
```

다중상속

- JAVA에서 금지됨

메서드 오버라이딩

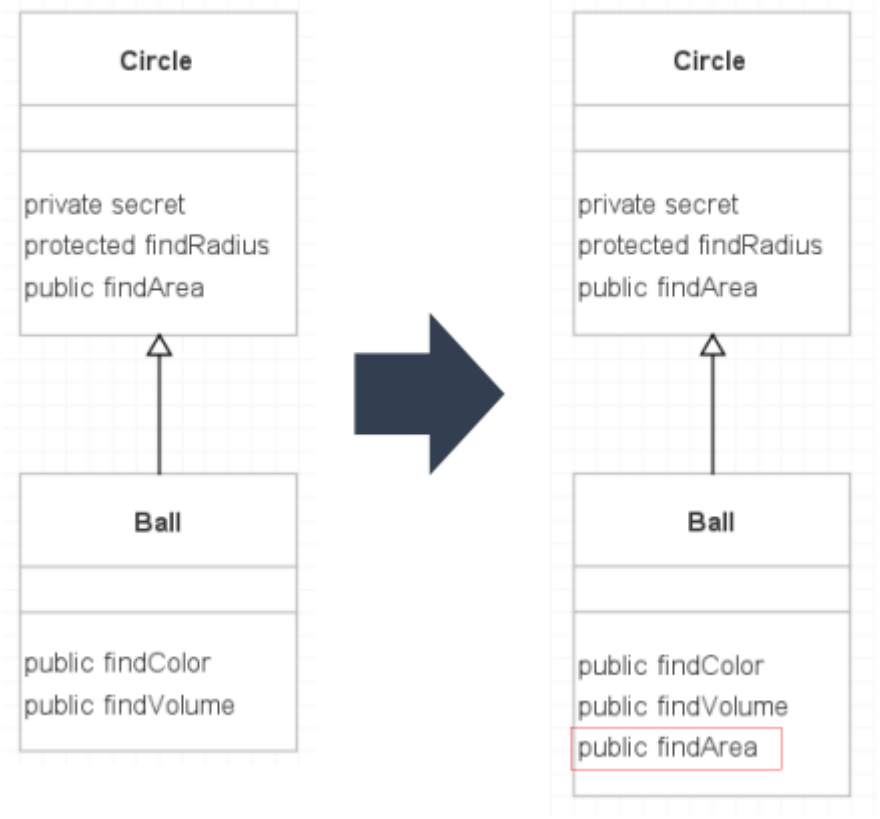
- 상속 관계에서 부모 메서드를 자식 클래스에 맞게 수정하는것

메서드 오버라이딩 규칙

- 부모클래스의 메서드와 동일한 시그니처 사용
- 반환타입까지 동일할 것.
- 부모클래스의 메서드보다 접근 범위를 더 좁게 수정할 수 없다.
- 추가적인 예외가 발생할 수 있다.

메서드 오버라이딩 불가능한 경우

1. **private 메서드** :: 부모클래스 전용이기에 자식클래스에 상속 안됨
2. **정적메서드 :: 클래스 소속이므로 자식 클래스에 상속되지 않는다.**
3. **final 메서드** :: final메서드는 더이상 수정할 수 없으므로 자식클래스가 오버라이딩 할 수 없다.



부모 클래스의 멤버 접근

- 자식 클래스가 메서드 오버라이딩 하면 자식 객체는 부모 클래스의 오버라이딩 된 메서드를 숨긴다.
- 숨겨진 메서드 호출하려면 super 키워드 사용한다.
- **super : 현재 객체에서 부모 클래스의 참조를 의미**

메서드 오버라이딩과 오버로딩

비교 요소	메서드 오버라이딩	메서드 오버로딩
메서드 이름	동일하다.	동일하다.
매개변수	동일하다.	다르다.
반환 타입	동일하다.	관계없다.
상속 관계	필요하다.	필요 없다.
예외와 접근 범위	제약이 있다.	제약이 없다.
바인딩	호출할 메서드를 실행 중 결정하는 동적 바인딩이다.	호출할 메서드를 컴파일할 때 결정하는 정적 바인딩이다.

패키지

- 클래스 파일을 묶어서 관리하기 위한 것으로 파일시스템의 폴더를 이용
- 장점
  1. 패키지마다 별도의 이름 공간이 생기기 때문에 클래스이름의 유일성 보장
  2. 클래스를 패키지 단위로 제어할 수 있기 때문에 세밀하게 접근 제어 가능
  3. **java.lnag** :: import 하지 않아도 자동 임포트 되는 자바의 기본 클래스
  4. **java.awt** :: 그래픽 프로그래밍
  5. **java.io** :: 입출력 클래스

패키지 선언

- 주석문 제외하고 반드시 첫 라인에 위치
- 패키지 이름은 모두 소문자로 하는 것이 관례
- 일반적으로 패키지이름이 중복되지 않도록 회사 도메인 이름 역순으로 사용

패키지 사용

- 다른 패키지에 공개된 클래스를 사용하려면 패키지 경로를 적어줘야한다.

패키지 import 문

- package 문과 첫번째 클래스 사이에 위치

정적 import 문

- 정적 클래스 멤버를 import
- import 문이 있는 경우는 클래스 이름과 함께 필드와 메서드 사용
  - Calendar.JANUARY
  - Calendar.getInstance();

```
package chap06;

import static java.util.Arrays.sort;

import java.util.Calendar;

public class StaticImportDemo {
    public static void main(String[] args) {
        int[] data = { 3, 5, 1, 7 };

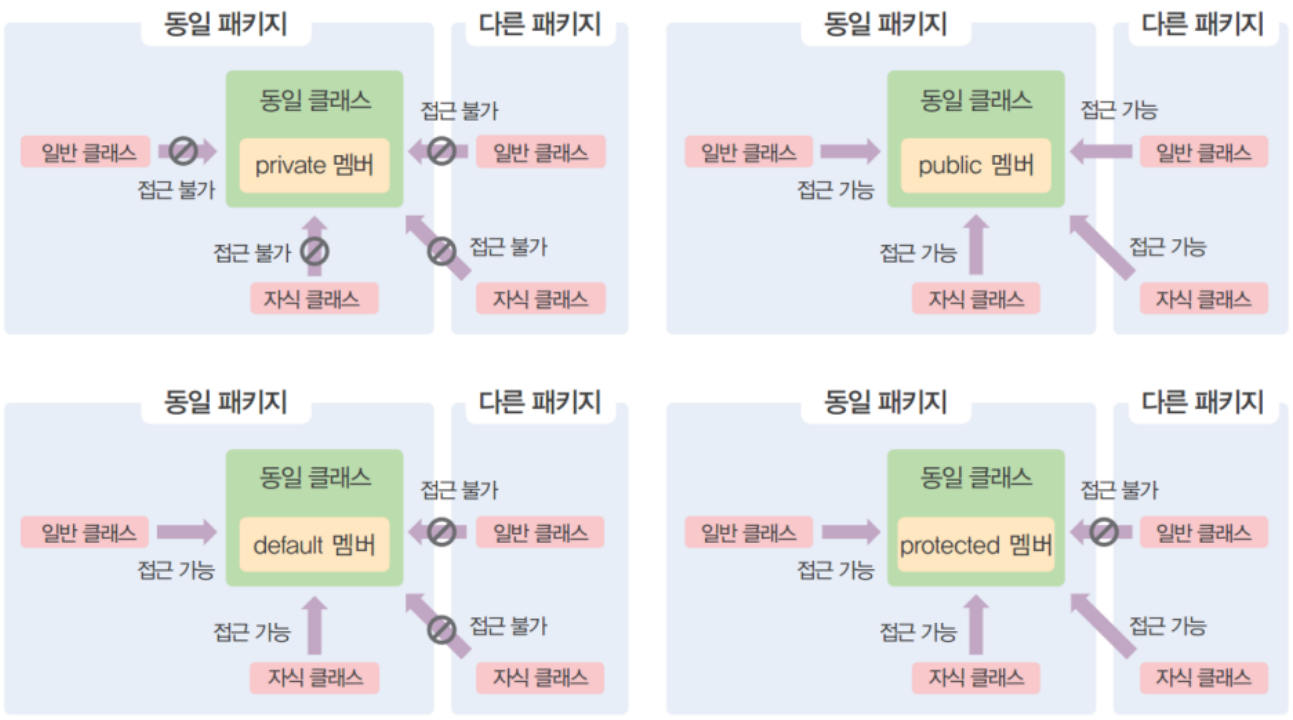
        sort(data);
        System.out.println(Calendar.JANUARY);
        Calendar.getInstance( );
    }
}
```

자식 생성자와 부모 생성자

- 자식 생성자를 호출하면 부모 생성자 호출됨
- 자식생성자의첫 행에서 부모의 기본 생성자 호출
- **부모클래스의 디폴트 생성자가 정의 안된 경우, 자식 생성자의 첫 행에 명시적 부모 생성자 호출이 필요**
- super()로 호출하려면, 가장 첫 줄에 있어야한다.
- super(s); 를 했는데, 부모 생성자에 해당하는 생성자가 없다면 , 오류 발생한다.
- 자식 객체를 생성했을 때, 부모에 생성자가 하나도 없으면, 컴파일러가 새로 만들어준다.
  - 그러나 여기서 부모에 매개변수가있는 생성자(기본생성자가 아닌)가 있다면, 컴파일러가 기본생성자를 호출하지 못하기 때문에 오류발생한다.
  - 자식생성자에 기본생성자가 있는 상태, 부모에는 기본생성자가 아닌 생성자가 있는 상태

상속과 접근제어 범위

접근 지정자의 접근 범위



접근 지정자 사용 시 주의 사항

1. private 멤버는 자식 클래스에 상속되지 않는다.
2. 클래스 멤버는 어떤 접근 지정자로도 지정할 수 있지만, **클래스 자체는 protected와 private로 지정 불가능하다.**
3. 메서드를 오버라이딩 할 때 부모 클래스의 메서드보다 가시성을 더 좁게 할 수 없다.

final 클래스

- 더 이상 상속 될 수 없는 클래스
- ex : String 클래스

final 메서드

- 자식 클래스에서의 오버라이딩 금지

객체의 타입 변환

- **상속 관계일 경우만 가능**
- 강제 타입 변환 가능

강제 타입 변환

- 자동 :: 자식 -> 부모

- **강제 :: 부모 -> 자식(언더)**

```
Person  p = new Person();
Student s = (Student) p;      // 강제로 타입 변환을 하면 오류가 발생한다.
```

```
Student s1 = new Student();
Person  p  = s1;
Student s2 = (Student) p;     // 강제로 타입 변환을 할 수 있다.
```

부모 타입 변수이지만 자식 객체를 가리킨다.

1. 부모가 바로 자식으로 강제 타입 변환하면 오류 발생하는데, 자식 객체를 부모가 참조하고 잇따면, 자식타입으로 반환 가능하다.

instanceof 연산자

- 변수 instanceof 타입
  - 변수가 해당 타입이나 자식 타입이라면 true 반환
  - 그렇지 않으면 false

개선된 instanceof 연산자

- 타입확인과 캐스팅 동시 수행
- 변수 instanceof 타입 새로 선언할 변수
- **s instanceof Person; // 앞(s)가 자식이면 true**
  ``` java
  //기존
  if(p instanceof Student){//p가 Student 타입이거나 Student의 자식 클래스라면
  Student s = (Student)p;
  s.work();
  }

```
``` java
//개선
    if(p instanceof Student s){
        s.work()
    }
```

다형성

- **하나의 참조 변수에 여러 객체를 대입**하여 동일한 명령어를 적용하더라도 **객체의 종류에 따라 다양한 동작을 수행할 수 있는 성질**

- 상속 관계에서 Person p = new Student();를 했을 때, 필드 값과 static 메서드(정적메서드)는 오버라이딩 될 수 없어서 부모꺼를 따른다.

동적 바인딩

- 실행 도중에 객체의 실제 타입을 기준으로 오버라이딩 된 메서드를 연결하고 호출하는 것