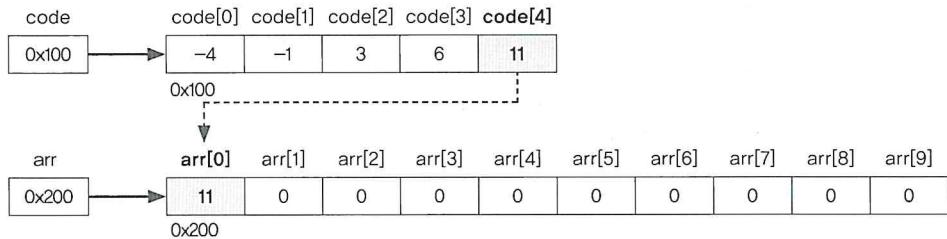


```

arr[i] = code[tmp];
→ arr[0] = code[4];      // code[4]는 배열 code의 5번째 요소이므로 11이다.
→ arr[0] = 11;           // arr[0]에 11이 저장된다.

```

위의 상황을 그림으로 그려보면 다음과 같다.



이와 같은 과정이 반복되면서 배열 arr에는 배열 code에 저장된 값들 중의 하나가 임의로 선택되어 저장된다.

▼ 예제 5-10/ch5/ArrayEx10.java

```

class ArrayEx10 {
    public static void main(String[] args) {
        int[] numArr = new int[10];

        for (int i=0; i < numArr.length ; i++ ) {
            System.out.print(numArr[i] = (int)(Math.random() * 10));
        }
        System.out.println();

        for (int i=0; i < numArr.length-1 ; i++ ) {
            boolean changed = false; // 자리바꿈이 발생했는지를 체크한다.

            for (int j=0; j < numArr.length-1-i ; j++) {
                if(numArr[j] > numArr[j+1]) { // 옆의 값이 작으면 서로 바꾼다.
                    int temp = numArr[j];
                    numArr[j] = numArr[j+1];
                    numArr[j+1] = temp;
                    changed = true; // 자리바꿈이 발생했으니 changed를 true로.
                }
            } // end for j

            if (!changed) break; // 자리바꿈이 없으면 반복문을 벗어난다.
        }

        for(int k=0; k<numArr.length;k++)
            System.out.print(numArr[k]); // 정렬된 결과를 출력한다.
        System.out.println();
    } // end for i
} // main의 끝
}

```

▼ 실행결과
1344213843 1342134438 1321344348 1213343448 1123334448

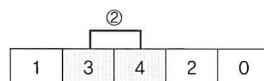
길이가 10인 배열에 0과 9사이의 임의의 값으로 채운 다음, 버블정렬 알고리즘을 통해서 크기순으로 정렬하는 예제이다. 이 알고리즘의 정렬방법은 아주 간단하다. 배열의 길이가 n 일 때, 배열의 첫 번째부터 $n-1$ 까지의 요소에 대해, 근접한 값과 크기를 비교하여 자리 바꿈을 반복하는 것이다.

```
for (int j = 0; j < numArr.length-1-i; j++) {
    // numArr[j]와 바로 옆의 요소 numArr[j+1]을 비교한다.
    if(numArr[j] > numArr[j+1]) {      // 왼쪽의 값이 크면 서로 바꾼다.
        int tmp      = numArr[j];
        numArr[j]    = numArr[j+1];
        numArr[j+1] = tmp;
    }
}
```

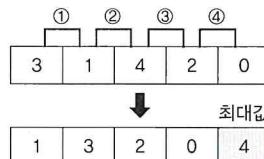
예를 들어 다음과 같이 길이가 5인 int배열이 있을 때, 첫 번째와 두 번째 요소의 값을 비교해서 왼쪽 요소의 값이 크면 두 값의 위치를 바꾸고, 그렇지 않으면 바꾸지 않는다.



위의 그림에서 왼쪽의 값이 크므로 두 값의 자리를 바꾼다.

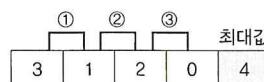


두 번째 비교에서는 왼쪽의 값이 작으므로 두 값의 자리를 바꾸지 않는다. 이러한 작업을 배열의 끝에 도달할 때 까지 반복하면 배열에서 제일 큰 값이 배열의 마지막 값이 된다.



비교횟수는 모두 4번이며, 이 값은 배열의 길이보다 1이 작은 값($\text{numArr.length}-1$)이다. 즉, 배열의 길이가 5라면, 4번만 비교하면 된다는 뜻이다. 나머지 값들이 아직 정렬되지 않았으므로 비교작업을 배열의 첫 번째 요소부터 다시 해야 한다.

그러나 처음과 달리 이번엔 세 번만 비교하면 된다. 배열의 마지막 요소는 최대값이므로 비교할 필요가 없기 때문이다.



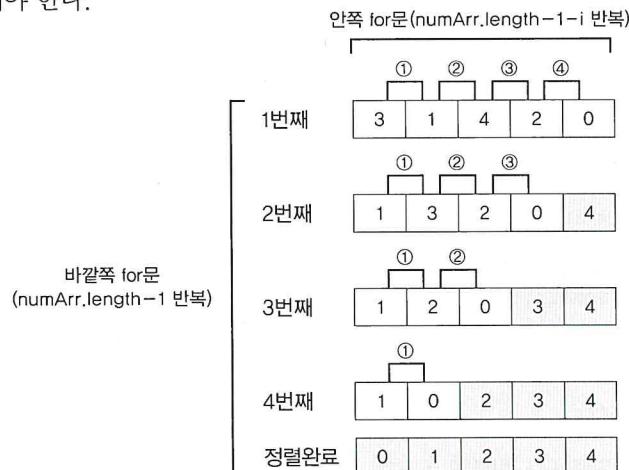
이처럼 비교작업(아래의 for문)을 반복할수록 비교해야하는 범위는 하나씩 줄어든다. 그래서 원래는 배열의 길이에서 1이 작은 ‘ $\text{numArr.length}-1$ ’번을 비교해야 하는데, 매 반복마다 비교횟수가 1씩 줄어들기 때문에 바깥쪽 for문의 제어변수 i 를 빼주는 것이다.

```

for (int j = 0; j < numArr.length-1-i; j++) {
    // numArr[j]와 바로 옆의 요소 numArr[j+1]을 비교한다.
    if (numArr[j] > numArr[j+1]) { // 왼쪽의 값이 크면 서로 바꾼다.
        int tmp      = numArr[j];
        numArr[j]    = numArr[j+1];
        numArr[j+1] = tmp;
    }
}

```

위의 작업이 한번 수행되는 것만으로는 정렬이 되지 않기 때문에 아래의 그림에서 알 수 있는 것처럼, 비교작업(위의 for문)을 모두 4번, 즉, ‘배열의 길이-1’번 만큼 반복해서 비교해야 한다.



그래서 바깥쪽 for문의 조건식이 ‘`numArr.length-1`’이어야 하는 것이다.

```

for (int i = 0; i < numArr.length-1; i++) {
    changed = false; // 매 반복마다 changed를 false로 초기화 한다.

    for (int j = 0; j < numArr.length-1-i; j++) {
        if (numArr[j] > numArr[j+1]) { // 옆의 값이 작으면 서로 바꾼다.
            int tmp      = numArr[j];
            numArr[j]    = numArr[j+1];
            numArr[j+1] = tmp;

            changed = true; // 자리바꿈이 발생했으니 changed를 true로 바꾼다.
        }
    } // end for j

    if (!changed) break; // 자리바꿈이 없으면 반복문을 벗어난다.

    for (int k = 0; k < numArr.length; k++)
        System.out.print(numArr[k]); // 정렬된 결과를 출력한다.
    System.out.println();
} // end of for i

```

보다 효율적인 작업을 위해 changed라는 boolean형 변수를 두어서 자리바꿈이 없으면 break문을 수행하여 정렬을 마치도록 했다. 자리바꿈이 없다는 것은 정렬이 완료되었음을 뜻하기 때문이다.

이 정렬 방법을 ‘버블 정렬(bubble sort)’라고 하는데, 비효율적이지만 가장 간단하다.

```
System.out.print(numArr[i] = (int)(Math.random() * 10));
```

그리고 위의 문장은 아래의 두 문장을 하나로 합친 것이다.

```
numArr[i] = (int)(Math.random() * 10);
System.out.print(numArr[i]);
```

▼ 예제 5-11/ch5/ArrayEx11.java

```
class ArrayEx11 {
    public static void main(String[] args) {
        int[] numArr = new int[10];
        int[] counter = new int[10];

        for (int i=0; i < numArr.length ; i++ ) {
            numArr[i] = (int)(Math.random() * 10); // 0~9의 임의의 수를 배열에 저장
            System.out.print(numArr[i]);
        }
        System.out.println();

        for (int i=0; i < numArr.length ; i++ ) {
            counter[numArr[i]]++;
        }

        for (int i=0; i < numArr.length ; i++ ) {
            System.out.println( i +"의 개수 :" + counter[i]);
        }
    } // main의 끝
}
```

▼ 실행결과

4446579753
0의 개수 :0
1의 개수 :0
2의 개수 :0
3의 개수 :1
4의 개수 :3
5의 개수 :2
6의 개수 :1
7의 개수 :2
8의 개수 :0
9의 개수 :1

길이가 10인 배열을 만들고 0과 9사이의 임의의 값으로 초기화 한다. 그리고 이 배열에 저장된 각 숫자가 몇 번 반복해서 나타나는지를 세어서 배열 counter에 담은 다음 화면에 출력한다.

간단한 예제라서 아래의 코드만 이해하면 나머지는 별 어려움이 없을 것이다.

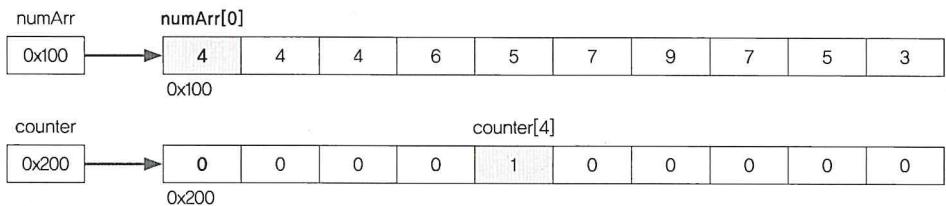
```
for (int i = 0; i < numArr.length ; i++ ) {
    counter[numArr[i]]++;
}
```

random()을 사용했기 때문에 실행할 때마다 결과가 달라지겠지만, 실행결과를 토대로 계산과정을 단계별로 살펴보면 다음과 같다.

```

counter[numArr[i]]++; // i의 값이 0인 경우를 가정하면,
→ counter[numArr[0]]++; // numArr[0]의 값은 4이다.
→ counter[4]++; // counter[4]의 값을 1증가시킨다.

```



배열 counter에서, 배열 numArr에 저장된 값과 일치하는 인덱스의 요소에 저장된 값을 1증가시킨다. 위의 그림에서는 numArr[0]에 4가 저장되어 있으므로 배열 counter의 인덱스가 4인 요소에 저장된 값이 0에서 1로 증가되었다. 이 과정이 반복되고 나면, 배열 counter의 각 요소에는 해당 인덱스의 값이 몇 번 나타났는지 알 수 있는 값이 저장된다.

| 플래시동영상 | 예제5-11에 대한 설명은 압축된 소스파일의 'flash/Array.exe'에서 자세히 볼 수 있다.

2. String배열

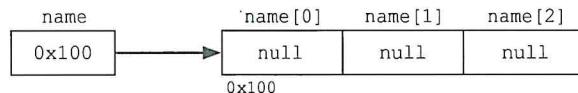
2.1 String배열의 선언과 생성

배열의 타입이 String인 경우에도 int배열의 선언과 생성방법은 다르지 않다. 예를 들어 3개의 문자열(String)을 담을 수 있는 배열을 생성하는 문장은 다음과 같다.

```
String[] name = new String[3]; // 3개의 문자열을 담을 수 있는 배열을 생성한다.
```

위의 문장을 수행한 결과를 그림으로 표현하면 다음과 같다. 3개의 String타입의 참조변수를 저장하기 위한 공간이 마련되고 참조형 변수의 기본값은 null이므로 각 요소의 값은 null로 초기화 된다.

| 참고 | null은 어떠한 객체도 가리키고 있지 않다는 뜻이다.



참고로 변수의 타입에 따른 기본값은 다음과 같다.

자료형	기본값
boolean	false
char	'\u0000'
byte, short, int	0
long	0L
float	0.0f
double	0.0d 또는 0.0
참조형 변수	null

▲ 표 5-2 타입에 따른 변수의 기본값(default value)

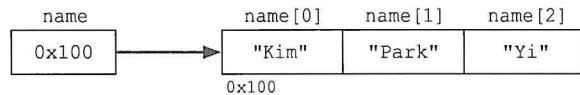
2.2 String배열의 초기화

초기화 역시 int배열과 동일한 방법으로 한다. 아래와 같이 배열의 각 요소에 문자열을 정하면 된다.

```
String[] name = new String[3]; // 길이가 3인 String배열을 생성
name[0] = "Kim";
name[1] = "Park";
name[2] = "Yi";
```

또는 팔호{}를 사용해서 다음과 같이 간단히 초기화 할 수도 있다.

```
String[] name = new String[]{"Kim", "Park", "Yi"};
String[] name = { "Kim", "Park", "Yi"}; // new String[]을 생략할 수 있음
```

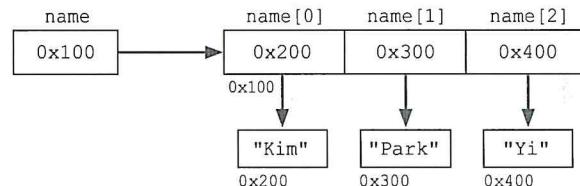


▲ 그림5-4 초기화된 String배열

특별히 String클래스만 “Kim”과 같이 큰따옴표만으로 간략히 표현하는 것이 허용되지만, 원래 String은 클래스이므로 아래의 왼쪽처럼 new연산자를 통해 객체를 생성해야한다.

String[] name = new String[3]; name[0] = new String("Kim"); name[1] = new String("Park"); name[2] = new String("Yi");	→	String[] name = new String[3]; name[0] = "Kim"; name[1] = "Park"; name[2] = "Yi";
--	---	--

그림5-4도 편의상 간략히 그린 것이며, 원래는 아래와 같이 그려야 더 정확한 그림이다.



배열에 실제 객체가 아닌 객체의 주소가 저장되어 있는 것을 볼 수 있다. 이처럼, 기본형 배열이 아닌 경우, 즉, 참조형 배열의 경우 배열에 저장되는 것은 객체의 주소이다. 참조형 배열을 객체 배열이라고도 하는데, 다음 장인 ‘6장 객체지향개념 1’에서 배울 것이다.

| 참고 | 참조형 변수를 간단히 참조변수라고도 하며, 모든 참조형 변수에는 객체가 메모리에 저장된 주소인 4 byte의 정수값(0x0~0xffffffff) 또는 null이 저장된다.

▼ 예제 5-12/ch5/ArrayEx12.java

```
class ArrayEx12 {
    public static void main(String[] args) {
        String[] names = {"Kim", "Park", "Yi"};

        for(int i=0; i < names.length;i++)
            System.out.println("names["+i+"]:"+names[i]);

        String tmp = names[2]; // 배열 names의 세 번째요소를 tmp에 저장
        System.out.println("tmp:"+tmp);
        names[0] = "Yu"; // 배열 names의 첫 번째 요소를 "Yu"로 변경

        for(String str : names) // 향상된 for문
            System.out.println(str);
    } // main
}
```

▼ 실행결과
names[0]:Kim names[1]:Park names[2]:Yi tmp:Yi Yu Park Yi

▼ 예제 5-13/ch5/ArrayEx13.java

```
class ArrayEx13 {
    public static void main(String[] args) {
        char[] hex = { 'C', 'A', 'F', 'E'};

        String[] binary = { "0000", "0001", "0010", "0011"
                            , "0100", "0101", "0110", "0111"
                            , "1000", "1001", "1010", "1011"
                            , "1100", "1101", "1110", "1111" };

        String result="";
        for (int i=0; i < hex.length ; i++) {
            if(hex[i] >='0' && hex[i] <='9') {
                result +=binary[hex[i]-'0']; // '8'-'0'의 결과는 8이다.
            } else { // A~F이면
                result +=binary[hex[i]-'A'+10]; // 'C'-'A'의 결과는 2
            }
        }
        // String(char[] value)
        System.out.println("hex:"+ new String(hex));
        System.out.println("binary:"+result);
    }
}
```

▼ 실행결과
hex:CAFE
binary:1100101011111110

16진수를 2진수로 변환하는 예제이다. 먼저 변환하고자 하는 16진수를 배열 hex에 나열한다. 16진수에는 A~F까지 6개의 문자가 포함되므로 char배열로 처리하였다. 그리고 문자열 배열 binary에는 이진수 '0000' 부터 '1111'(16진수로 0~F)까지 모두 16개의 값을 문자열로 저장하였다.

for문을 이용해서 배열 hex에 저장된 문자를 하나씩 읽어서 그에 해당하는 이진수 표현을 배열 binary에서 얻어 result에 덧붙이고 그 결과를 화면에 출력한다.

```
result +=binary[hex[i]-'A'+10];
```

i의 값이 0일 때, hex[0]의 값은 'C'이므로, 위의 문장은 다음과 같은 과정으로 계산된다.

```
→ result +=binary[hex[0]-'A'+10]; // hex[0]은 'C'  
→ result +=binary['C'-'A'+10]; // 'C'-'A' → 67-65 → 2  
→ result +=binary[2+10];  
→ result +=binary[12];  
→ result += "1100";
```

2.3 char배열과 String클래스

지금까지 여러 문자, 즉 문자열을 저장할 때 String타입의 변수를 사용했다. 사실 문자열이라는 용어는 ‘문자를 연이어 늘어놓은 것’을 의미하므로 문자배열인 char배열과 같은 뜻이다.

그런데 자바에서는 char배열이 아닌 String클래스를 이용해서 문자열을 처리하는 이유는 String클래스가 char배열에 여러 가지 기능을 추가하여 확장한 것이기 때문이다.

그래서 char배열을 사용하는 것보다 String클래스를 사용하는 것이 문자열을 다루기 더 편리하다.

String클래스는 char배열에 기능(메서드)을 추가한 것이다.

C언어에서는 문자열을 char배열로 다루지만, 객체지향언어인 자바에서는 char배열과 그에 관련된 기능들을 함께 묶어서 클래스에 정의한다. 객체지향개념이 나오기 이전의 언어들은 데이터와 기능을 따로 다루었지만, 객체지향언어에서는 데이터와 그에 관련된 기능을 하나의 클래스에 묶어서 다룰 수 있게 한다. 즉, 서로 관련된 것들끼리 데이터와 기능을 구분하지 않고 함께 묶는 것이다.

여기서 밀하는 ‘기능’은 함수를 의미하며, 메서드는 객체지향 언어에서 ‘함수’ 대신 사용하는 용어일 뿐 함수와 같은 뜻이다. 앞으로 ‘기능’이나 ‘함수’ 대신 ‘메서드’라는 용어를 사용할 것이다.

char배열과 String클래스의 한 가지 중요한 차이가 있는데, String객체(문자열)는 읽을 수만 있을 뿐 내용을 변경할 수 없다는 것이다.

```
String str = "Java";
str = str + "8";           // "Java8"이라는 새로운 문자열이 str에 저장된다.
System.out.println(str); // "Java8"
```

위의 문장에서 문자열 str의 내용이 변경되는 것 같지만, 문자열은 변경할 수 없으므로 새로운 내용의 문자열이 생성된다.

| 참고 | 변경 가능한 문자열을 다루려면, StringBuffer클래스를 사용하면 된다. 문자열에 대한 것은 9장에서 설명한다.

String클래스의 주요 메서드

String클래스는 상당히 많은 문자열 관련 메서드들을 제공하지만 지금은 가장 기본적인 몇 가지만 살펴보고 나머지는 9장에서 자세히 다를 것이다. 자세히 이해하려 하지 말고 원하는 결과를 얻으려면 어떻게 코드를 작성해야하는지 정도만 이해하자.

메서드	설명
char charAt(int index)	문자열에서 해당 위치(index)에 있는 문자를 반환한다.
int length()	문자열의 길이를 반환한다.
String substring(int from, int to)	문자열에서 해당 범위(from~to)에 있는 문자열을 반환한다. (to는 범위에 포함되지 않음)
boolean equals(Object obj)	문자열의 내용이 obj와 같은지 확인한다. 같으면 결과는 true, 다르면 false가 된다.
char[] toCharArray()	문자열을 문자배열(char[])로 변환해서 반환한다.

▲ 표5-3 String클래스의 주요 메서드

charAt메서드는 문자열에서 지정된 index에 있는 한 문자를 가져온다. 배열에서 ‘배열이 름[index]’로 index에 위치한 값을 가져오는 것과 같다고 생각하면 된다. 배열과 마찬가지로 charAt메서드의 index값은 0부터 시작한다.

```
String str = "ABCDE";
char ch = str.charAt(3); // 문자열 str의 4번째 문자 'D' 를 ch에 저장.
```

index	0	1	2	3	4
문자	A	B	C	D	E

substring()은 문자열의 일부를 뽑아낼 수 있다. 주의할 것은 범위의 끝은 포함되지 않는다는 것이다. 예를 들어, index의 범위가 1~4라면 4는 범위에 포함되지 않는다.

```
String str = "012345";
String tmp = str.substring(1,4); // str에서 index범위 1~4의 문자들을 반환
System.out.println(tmp); // "123"이 출력된다.
```

equals()는 이미 앞에서 간단히 배웠는데, 문자열의 내용이 같은지 다른지 확인하는데 사용한다. 기본형 변수의 값을 비교하는 경우 ‘==’연산자를 사용하지만, 문자열의 내용을 비교할 때는 equals()를 사용해야 한다. 그리고 이 메서드는 대소문자를 구분한다는 점에 주의하자. 대소문자를 구분하지 않고 비교하려면 equals()대신 equalsIgnoreCase()를 사용해야 한다.

```
String str = "abc";
if(str.equals("abc")) { // str와 "abc"가 내용이 같은지 확인한다.
    ...
}
```

char배열과 String클래스의 변환

가끔 char배열을 String클래스로 변환하거나, 또는 그 반대로 변환해야하는 경우가 있다. 그럴 때 다음의 코드를 사용하자.

```
char[] chArr = { 'A', 'B', 'C' };
String str = new String(chArr); // char배열 → String
char[] tmp = str.toCharArray(); // String → char배열
```

▼ 예제 5-14/ch5/ArrayEx14.java

```
class ArrayEx14 {
    public static void main(String[] args) {
        String src = "ABCDE";

        for(int i=0; i < src.length(); i++) {
            char ch = src.charAt(i); // src의 i번째 문자를 ch에 저장
            System.out.println("src.charAt(" + i + "): " + ch);
        }
    }
}
```

```

    // String을 char[]로 변환
    char[] chArr = src.toCharArray();

    // char배열(char[])을 출력
    System.out.println(chArr);
}
}

```

▼ 실행결과

```

src.charAt(0):A
src.charAt(1):B
src.charAt(2):C
src.charAt(3):D
src.charAt(4):E
ABCDE

```

String클래스의 `charAt(int idx)`을 사용하는 방법을 보여주는 예제이다. `charAt(int idx)`은 문자열 중에서 `idx`번째 위치에 있는 문자를 반환한다. `idx`의 값은 배열처럼 0부터 시작한다는 것을 확인하자.

그리고 `println()`로 문자배열을 출력하면 문자열 출력하듯이 문자배열의 모든 요소를 이어서 한 줄로 출력한다.

▼ 예제 5-15/ch5/ArrayEx15.java

```

class ArrayEx15 {
    public static void main(String[] args) {
        String source = "SOSHELP";
        String[] morse = {".-", "-...", "-.-.", "-..", ".",
            "....", "---", "...", "-.-", "-.-", "-.-", "-.-",
            "-.-.", "-.-.", "-.-", "...", "-",
            "...", "...-", "...", "-.-", "-.-", "-.-"};
        String result="";

        for (int i=0; i < source.length() ; i++ ) {
            result+=morse[source.charAt(i)-'A'];
        }
        System.out.println("source:"+ source);
        System.out.println("morse:"+result);
    }
}

```

▼ 실행결과

```

source:SOSHELP
morse:...-----...

```

문자열(String)을 모尔斯(morse)부호로 변환하는 예제이다. 이전의 16진수를 2진수로 변환하는 예제와 같지만, char배열 대신 이번엔 String을 사용했다.

String의 문자의 개수는 `length()`를 통해서 얻을 수 있고, `charAt(int i)`메서드는 String의 `i`번째 문자를 반환한다. 그래서 for문의 조건식에 `length()`를 사용하고 `charAt(i)`를 통해서 source에서 한 문자씩 차례대로 읽어 올 수 있다.

```

    result+=morse[source.charAt(i)-'A']; // i가 0일 때
→ result+=morse[source.charAt(0)-'A']; // source.charAt(0)는 첫 번째 문자
→ result+=morse['S'-'A'];           // 'S'-'A' → 83-65 → 18
→ result+=morse[18];
→ result+="...";                  // result = result + "...";와 같다.

```

2.4 커맨드 라인을 통해 입력받기

Scanner클래스의 nextLine()외에도 화면을 통해 사용자로부터 값을 입력받을 수 있는 간단한 방법이 있다. 바로 커맨드라인을 이용한 방법인데, 프로그램을 실행할 때 클래스 이름 뒤에 공백문자로 구분하여 여러 개의 문자열을 프로그램에 전달 할 수 있다.

만일 실행할 프로그램의 main메서드가 담긴 클래스의 이름이 MainTest라고 가정하면 다음과 같이 실행할 수 있을 것이다.

```
c:\jdk1.8\work\ch5>java MainTest abc 123
```

커맨드라인을 통해 입력된 두 문자열은 String배열에 담겨서 MainTest클래스의 main메서드의 매개변수(args)에 전달된다. 그리고는 main메서드 내에서 args[0], args[1]과 같은 방식으로 커맨드라인으로부터 전달받은 문자열에 접근할 수 있다. 여기서 args[0]은 “abc”이고 args[1]은 “123”이 된다.

▼ 예제 5-16/ch5/ArrayEx16.java

```
class ArrayEx16 {  
    public static void main(String[] args) {  
        System.out.println("매개변수의 개수:"+args.length);  
        for(int i=0;i< args.length;i++) {  
            System.out.println("args["+ i + "] = \""+ args[i] + "\"");  
        }  
    }  
}
```

▼ 실행결과

```
C:\jdk1.8\work\ch5>java ArrayEx16 abc 123 "Hello world"  
매개변수의 개수:3  
args[0] = "abc"  
args[1] = "123"  
args[2] = "Hello world"  
  
C:\jdk1.8\work\ch5>java ArrayEx16 ← 매개변수를 입력하지 않았다.  
매개변수의 개수:0
```

커맨드라인에 입력된 매개변수는 공백문자로 구분하기 때문에 입력될 값에 공백이 있는 경우 큰따옴표("")로 감싸주어야 한다. 그리고 커맨드라인에서 숫자를 입력해도 숫자가 아닌 문자열로 처리된다는 것에 주의해야한다. 문자열 “123”을 숫자 123으로 바꾸려면 다음과 같이 한다.

```
int num = Integer.parseInt("123"); // 변수 num에 숫자 123이 저장된다.
```

그리고 커맨드라인에 매개변수를 입력하지 않으면 크기가 0인 배열이 생성되어 args.length의 값은 0이 된다. 앞서 배운 것처럼 이렇게 크기가 0인 배열을 생성하는 것도 가능하다. 만일 입력된 매개변수가 없다고 해서 배열을 생성하지 않으면 참조변수 args의 값은 null이 될 것이고, 배열 args를 사용하는 모든 코드에서 에러가 발생할 것이다. 이러한 에러를 피하려면, 다음과 같이 main메서드에 if문을 추가해줘야 한다.

```

public static void main(String[] args) {
    if(args != null) { // args가 null이 아닐 때만 꽂호{}의 문장들을 수행
        System.out.println("매개변수의 개수:"+args.length);
        for(int i=0;i< args.length;i++) {
            System.out.println("args["+ i + "] = "+ args[i]);
        }
    }
}

```

그러나 JVM이 입력된 매개변수가 없을 때, null 대신 크기가 0인 배열을 생성해서 args에 전달하도록 구현되어 우리는 이러한 수고를 덜게 되었다.

▼ 예제 5-17/ch5/ArrayEx17.java

```

class ArrayEx17 {
    public static void main(String[] args) {
        if (args.length != 3) { // 입력된 값의 개수가 3개가 아니면,
            System.out.println("usage: java ArrayEx17 NUM1 OP NUM2");
            System.exit(0); // 프로그램을 종료한다.
        }

        int num1 = Integer.parseInt(args[0]); // 문자열을 숫자로 변환한다.
        char op = args[1].charAt(0); // 문자열을 문자(char)로 변환한다.
        int num2 = Integer.parseInt(args[2]);
        int result = 0;

        switch(op) { // switch문의 수식으로 char타입의 변수도 가능하다.
            case '+':
                result = num1 + num2;
                break;
            case '-':
                result = num1 - num2;
                break;
            case '*':
                result = num1 * num2;
                break;
            case '/':
                result = num1 / num2;
                break;
            default :
                System.out.println("지원되지 않는 연산입니다.");
        }
        System.out.println("결과:"+result);
    }
}

```

▼ 실행결과

```
C:\jdk1.8\work\ch5>java ArrayEx17
usage: java ArrayEx17 NUM1 OP NUM2

C:\jdk1.8\work\ch5>java ArrayEx17 10 + 3
결과:13

C:\jdk1.8\work\ch5>java ArrayEx17 10 x 3
결과:30
```

화면으로 부터 사칙연산을 수행하는 수식을 입력받아서 계산하여 그 결과를 보여주는 예제이다. 커맨드라인으로부터 입력받은 데이터는 모두 문자열이므로 숫자와 문자로 변환이 필요하며, Integer.parseInt()를 사용했다.

3. 다차원 배열

지금까지 우리가 배운 배열은 1차원 배열인데, 2차원 이상의 배열, 즉 다차원(multi-dimensional) 배열도 선언해서 사용할 수 있다. 메모리의 용량이 허용하는 한, 차원의 제한은 없지만, 주로 1, 2차원 배열이 사용되므로 2차원 배열만 잘 이해하고 나면 3차원 이상의 배열도 어렵지 않게 다룰 수 있으므로 2차원 배열에 대해서 중점적으로 배울 것이다.

3.1 2차원 배열의 선언과 인덱스

2차원 배열을 선언하는 방법은 1차원 배열과 같다. 다만 괄호[]가 하나 더 들어갈 뿐이다.

선언 방법	선언 예
타입[][] 변수이름;	int[][] score;
타입 변수이름[][];	int score[][];
타입[] 변수이름[];	int[] score[];

▲ 표5-4 2차원 배열의 선언

| 참고 | 3차원이상의 고차원 배열의 선언은 대괄호[]의 개수를 차원의 수만큼 추가해 주기만 하면 된다.

2차원 배열은 주로 테이블 형태의 데이터를 담는데 사용되며, 만일 4행 3열의 데이터를 담기 위한 배열을 생성하려면 다음과 같이한다.

```
int[][] score = new int[4][3]; // 4행 3열의 2차원 배열을 생성한다.
```

위 문장이 수행되면 아래의 그림처럼 4행 3열의 데이터, 모두 12개의 int값을 저장할 수 있는 공간이 마련된다.



위의 그림에서는 각 요소, 즉 저장공간의 타입을 적어놓은 것이고, 실제로는 배열요소의 타입인 int의 기본값인 0이 저장된다. 배열을 생성하면, 배열의 각 요소에는 배열요소타입의 기본값이 자동적으로 저장된다.

2차원 배열의 index

2차원 배열은 행(row)과 열(column)로 구성되어 있기 때문에 index도 행과 열에 각각 하나씩 존재한다. ‘행index’의 범위는 ‘0~행의 길이-1’이고 ‘열index’의 범위는 ‘0~열의 길이-1’이다. 그리고 2차원 배열의 각 요소에 접근하는 방법은 ‘배열이름[행index][열index]’이다.

만일 다음과 같이 배열 score를 생성하면, score[0][0]부터 score[3][2]까지 모두 12개 ($4 \times 3 = 12$)의 int값을 저장할 수 있는 공간이 마련되고, 각 배열 요소에 접근할 수 있는 방법은 아래의 그림과 같다.

```
int[][] score = new int[4][3]; // 4행 3열의 2차원 배열 score를 생성
```

		열 index(0 ~ 열의 길이-1)		
		0	1	2
행 index (0 ~ 행의 길이-1)	0	score[0][0]	score[0][1]	score[0][2]
	1	score[1][0]	score[1][1]	score[1][2]
	2	score[2][0]	score[2][1]	score[2][2]
	3	score[3][0]	score[3][1]	score[3][2]

배열 score의 1행 1열에 100을 저장하고, 이 값을 출력하려면 다음과 같이 하면 된다.

```
score[0][0] = 100; // 배열 score의 1행 1열에 100을 저장
System.out.println(score[0][0]); // 배열 score의 1행 1열의 값을 출력
```

3.2 2차원 배열의 초기화

2차원 배열도 괄호{}를 사용해서 생성과 초기화를 동시에 할 수 있다. 다만, 1차원 배열보다 괄호{}를 한번 더 써서 행별로 구분해 준다.

```
int[][] arr = new int[][]{ {1, 2, 3}, {4, 5, 6} };
int[][] arr = { {1, 2, 3}, {4, 5, 6} }; // new int[][]가 생략됨
```

크기가 작은 배열은 위와 같이 간단히 한 줄로 써주는 것도 좋지만, 가능하면 다음과 같이 행별로 줄 바꿈을 해주는 것이 보기도 좋고 이해하기 쉽다.

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

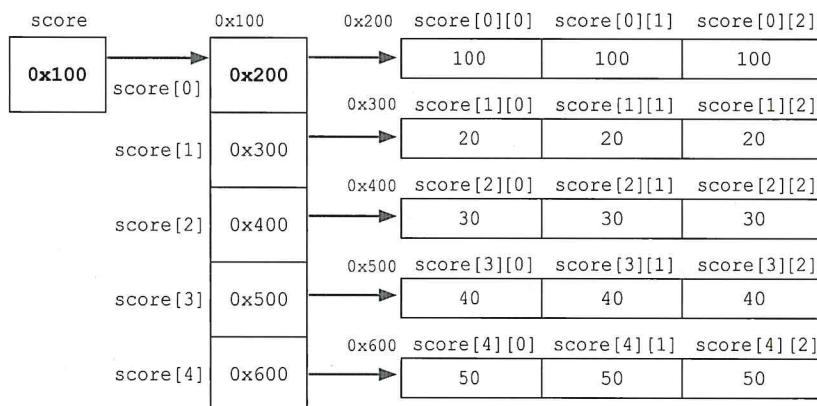
만일 아래와 같은 테이블형태의 데이터를 배열에 저장하려면,

	국어	영어	수학
1	100	100	100
2	20	20	20
3	30	30	30
4	40	40	40
5	50	50	50

다음과 같이 하면 된다.

```
int[][] score = {
    {100, 100, 100}
, {20, 20, 20}
, {30, 30, 30}
, {40, 40, 40}
, {50, 50, 50}
};
```

위 문장이 수행된 후에, 2차원 배열 score가 메모리에 어떤 형태로 만들어지는지 그려보면 다음과 같다.



▲ 그림5-5 2차원 배열

그림5-5에서 알 수 있듯이 2차원 배열은 ‘배열의 배열’로 구성되어 있다. 즉, 여러 개의 1차원 배열을 묶어서 또 하나의 배열로 만든 것이다.

그러면, 여기서 score.length의 값은 얼마일까?

배열 참조변수 score가 참조하고 있는 배열의 길이가 얼마인가를 세어보면 될 것이다. 정답은 5이다. 그리고 score[0].length은 배열 참조변수 score[0]이 참조하고 있는 배열의 길이이므로 3이다.

같은 이유로 score[1].length, score[2].length, score[3].length, score[4].length의 값 역시 모두 3이다.

만일 for문을 이용해서 2차원 배열을 초기화한다면 다음과 같을 것이다.

```
for (int i = 0; i < score.length; i++) {
    for (int j = 0; j < score[i].length; j++) {
        score[i][j] = 10;
    }
}
```

위의 코드는 2차원 배열 score의 모든 요소를 10으로 초기화한다.

▼ 예제 5-18/ch5/ArrayEx18.java

```

class ArrayEx18 {
    public static void main(String[] args) {
        int[][] score = {
            { 100, 100, 100}
            , { 20, 20, 20}
            , { 30, 30, 30}
            , { 40, 40, 40}
        };
        int sum = 0;

        for(int i=0;i < score.length;i++) {
            for(int j=0;j < score[i].length;j++) {
                System.out.printf("score[%d][%d]=%d%n", i, j, score[i][j]);
            }
        }

        for (int[] tmp : score) {
            for (int i : tmp) {
                sum += i;
            }
        }

        System.out.println("sum="+sum);
    }
}

```

▼ 실행결과

```

score[0][0]=100
score[0][1]=100
score[0][2]=100
score[1][0]=20
score[1][1]=20
score[1][2]=20
score[2][0]=30
score[2][1]=30
score[2][2]=30
score[3][0]=40
score[3][1]=40
score[3][2]=40
sum=570

```

2차원 배열 score의 모든 요소의 합을 구하고, 출력하는 예제이다. 하나의 이중 for문으로 처리가 가능한 작업이지만, 향상된 for문으로 2차원 배열의 모든 요소를 읽어오는 방법을 보여주기 위해 출력과 합계를 따로 처리하였다.

```

for (int i : score) {      // 예전. 2차원 배열 score의 각 요소는 1차원 배열
    sum += i;
}

```

이렇게 간단히 되면 좋겠지만, 2차원 배열 score의 각 요소는 1차원 배열이므로 아래와 같이 for문을 하나 더 추가해야 한다.

```

for (int[] tmp : score) { // score의 각 요소(1차원 배열 주소)를 tmp에 저장
    for (int i : tmp) { // tmp는 1차원 배열을 가리키는 참조변수
        sum += i;
    }
}

```

향상된 for문으로 배열의 각 요소에 저장된 값들을 순차적으로 읽어올 수는 있지만, 배열에 저장된 값을 변경할 수는 없다.

▼ 예제 5-19/ch5/ArrayEx19.java

```
class ArrayEx19 {
    public static void main(String[] args) {
        int[][] score = {
            { 100, 100, 100}
            , { 20, 20, 20}
            , { 30, 30, 30}
            , { 40, 40, 40}
            , { 50, 50, 50}
        };
        // 과목별 총점
        int korTotal = 0, engTotal = 0, mathTotal = 0;

        System.out.println("번호 국어 영어 수학 총점 평균 ");
        System.out.println("===== ");

        for(int i=0;i < score.length;i++) {
            int sum = 0;      // 개인별 총점
            float avg = 0.0f; // 개인별 평균

            korTotal += score[i][0];
            engTotal += score[i][1];
            mathTotal += score[i][2];
            System.out.printf("%3d", i+1);

            for(int j=0;j < score[i].length;j++) {
                sum += score[i][j];
                System.out.printf("%5d", score[i][j]);
            }

            avg = sum/(float)score[i].length; // 평균계산
            System.out.printf("%5d %5.1f\n", sum, avg);
        }

        System.out.println("===== ");
        System.out.printf("총점:%3d %4d %4d\n", korTotal,engTotal,mathTotal);
    }
}
```

▼ 실행결과

```
번호 국어 영어 수학 총점 평균
=====
1 100 100 100 300 100.0
2 20 20 20 60 20.0
3 30 30 30 90 30.0
4 40 40 40 120 40.0
5 50 50 50 150 50.0
=====
총점: 240 240 240
```

5명의 학생의 세 과목 점수를 더해서 각 학생의 총점과 평균을 계산하고, 과목별 총점을 계산하는 예제이다. 간단한 예제이므로 자세한 설명은 생략한다.

3.3 가변 배열

자바에서는 2차원 이상의 배열을 ‘배열의 배열’의 형태로 처리한다는 사실을 이용하면 보다 자유로운 형태의 배열을 구성할 수 있다.

2차원 이상의 다차원 배열을 생성할 때 전체 배열 차수 중 마지막 차수의 길이를 지정하지 않고, 추후에 각기 다른 길이의 배열을 생성함으로써 고정된 형태가 아닌 보다 유동적인 가변 배열을 구성할 수 있다.

만일 다음과 같이 ‘ 5×3 ’길이의 2차원 배열 score를 생성하는 코드가 있을 때,

```
int[][] score = new int[5][3]; // 5행 3열의 2차원 배열 생성
```

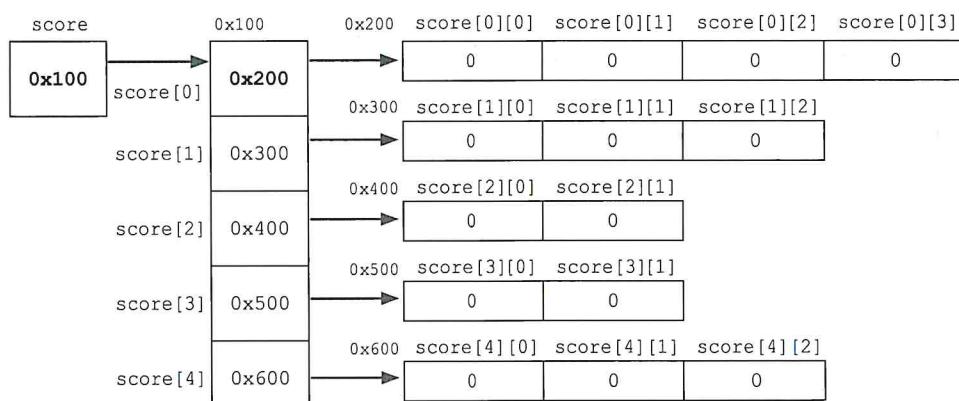
위 코드를 다음과 같이 표현 할 수 있다.

```
int[][] score = new int[5][]; // 두 번째 차원의 길이는 지정하지 않는다.
score[0] = new int[3];
score[1] = new int[3];
score[2] = new int[3];
score[3] = new int[3];
score[4] = new int[3];
```

첫 번째 코드와 같이 2차원 배열을 생성하면 직사각형 테이블 형태의 고정적인 배열만 생성할 수 있지만, 두 번째 코드와 같이 2차원 배열을 생성하면 다음과 같이 각 행마다 다른 길이의 배열을 생성하는 것이 가능하다.

```
int[][] score = new int[5][];
score[0] = new int[4];
score[1] = new int[3];
score[2] = new int[2];
score[3] = new int[2];
score[4] = new int[3];
```

위의 코드에 의해서 생성된 2차원 배열을 그림으로 표현하면 다음과 같다.



▲ 그림5-6 가변 배열

score.length의 값은 여전히 5지만, 일반적인 2차원 배열과 달리 score[0].length의 값은 4이고 score[1].length의 값은 3으로 서로 다르다.

가변배열 역시 중괄호{}를 이용해서 다음과 같이 생성과 초기화를 동시에 하는 것이 가능하다.

```
int[][] score = {
    {100, 100, 100, 100}
, {20, 20, 20}
, {30, 30}
, {40, 40}
, {50, 50, 50}
};
```

| 플래시동영상 | MultiDim.exe을 보면 가변배열의 생성과정을 자세히 볼 수 있다.

3.4 다차원 배열의 활용

다차원 배열의 대표적인 예제들을 몇 가지 골라보았다. 이 예제들만 잘 이해해도 다차원 배열을 활용하는데 별 어려움이 없을 것이다.

[예제5-20] 좌표에 X표하기 입력한 2차원 좌표의 위치에 X를 표시

[예제5-21] 빙고 빙고판을 만들고 입력받은 숫자를 빙고판에서 지운다.

[예제5-22] 행렬의 곱셈 두 행렬(matrix)을 곱한 결과를 출력

[예제5-23] 단어 맞추기 영어 단어를 보여주고, 뜻을 맞추는 게임

이 예제에 추가하면 좋을만한 기능은 없는지 고민해보고, 조금씩 단계별로 발전시켜보면 좋은 공부가 될 것이다.

▼ 예제 5-20/ch5/MultiArrEx1.java

```
import java.util.*;

class MultiArrEx1 {
    public static void main(String[] args) {
        final int SIZE = 10;
        int x = 0, y = 0;

        char[][] board = new char[SIZE][SIZE];
        byte[][] shipBoard = {
            // 1 2 3 4 5 6 7 8 9
            { 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 }, // 1
            { 1, 1, 1, 1, 0, 0, 1, 0, 0, 0 }, // 2
            { 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 }, // 3
            { 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 }, // 4
            { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }, // 5
            { 1, 1, 0, 1, 0, 0, 0, 0, 0, 0 }, // 6
            { 0, 0, 0, 1, 0, 0, 0, 0, 0, 0 }, // 7
            { 0, 0, 0, 1, 0, 0, 0, 0, 0, 0 }, // 8
            { 0, 0, 0, 0, 0, 1, 1, 1, 0, 0 } // 9
        };
    }
}
```

```

// 1행에 행번호를, 1열에 열번호를 저장한다.
for(int i=1;i<SIZE;i++)
    board[0][i] = board[i][0] = (char)(i+'0');

Scanner scanner = new Scanner(System.in);

while(true) {
    System.out.printf("좌표를 입력하세요. (종료는 00)>");
    String input = scanner.nextLine(); // 화면입력받은 내용을 input에 저장

    if(input.length()==2) { // 두 글자를 입력한 경우
        x = input.charAt(0) - '0'; // 문자를 숫자로 변환
        y = input.charAt(1) - '0';

        if(x==0 && y==0) // x와 y가 모두 0인 경우 종료
            break;
    }

    if(input.length()!=2 || x<=0 || x>=SIZE || y<=0 || y>=SIZE) {
        System.out.println("잘못된 입력입니다. 다시 입력해주세요.");
        continue;
    }

    // shipBoard[x-1][y-1]의 값이 1이면, 'O'를 board[x][y]에 저장한다.
    board[x][y] = shipBoard[x-1][y-1]==1 ? 'O' : 'X';

    // 배열 board의 내용을 화면에 출력한다.
    for(int i=0;i<SIZE;i++)
        System.out.println(board[i]); // board[i]는 1차원 배열
    System.out.println();
}
} // main의 끝
}

```

▼ 실행결과

좌표를 입력하세요. (종료는 00)>1010

잘못된 입력입니다. 다시 입력해주세요.

좌표를 입력하세요. (종료는 00)>33

123456789

1
2
3 X
4
5
6
7
8
9

좌표를 입력하세요. (종료는 00)>00

들이 마주 앉아 다양한 크기의 배를 상대방이 알지 못하게 배치한 다음, 번갈아가며 좌표를 불러서 상대방의 배의 위치를 알아내는 게임을 간단히 하여 예제로 만들어 보았다.

2차원 char배열 board는 입력한 좌표를 표시하기 위한 것이고, 2차원 byte배열 shipBoard에는 상대방의 배의 위치를 저장한다. 0은 바다이고, 1은 배가 있는 것이다.

```

char[][] board = new char[SIZE][SIZE];
byte[][] shipBoard = {
    // 1 2 3 4 5 6 7 8 9
    { 0, 0, 0, 0, 0, 1, 0, 0 }, // 1
    { 1, 1, 1, 1, 0, 0, 1, 0, 0 }, // 2
    { 0, 0, 0, 0, 0, 0, 1, 0, 0 }, // 3
    { 0, 0, 0, 0, 0, 0, 1, 0, 0 }, // 4
    { 0, 0, 0, 0, 0, 0, 0, 0, 0 }, // 5
    { 1, 1, 0, 1, 0, 0, 0, 0, 0 }, // 6
    { 0, 0, 0, 1, 0, 0, 0, 0, 0 }, // 7
    { 0, 0, 0, 1, 0, 0, 0, 0, 0 }, // 8
    { 0, 0, 0, 0, 0, 1, 1, 1, 0 } // 9
};

```

배열 board는 좌표를 쉽게 입력하기 위한 행번호와 열번호가 필요하다. 그래서 배열 board가 배열 shipBoard보다 행과 열의 길이가 1씩 큰 것이다.

```

// 1행에 행번호를, 1열에 열번호를 저장한다.
for(int i=1;i<SIZE;i++) {
    board[0][i] = board[i][0] = (char)(i+'0'); // (char)(1+'0') → '1'
} // (char)(2+'0') → '2'

```

board는 char배열이므로, 숫자를 문자로 변환하여 저장해야한다. 그래서 변수 i에 문자 '0'을 더한다. 숫자 1에 문자'0'을 더하면 문자 '1'이 된다. 그 다음엔 무한반복문으로 좌표를 반복해서 입력받는다. 입력받은 좌표 x, y에 저장된 값이 1이면, board[x][y]에 'O'를 저장하고, 1이 아니면 'X'를 저장한다. 배열 board와 shipBoard간에는 좌표가 가로, 세로로 각각 1씩 차이가 난다는 점을 잊지 말자.

```

while(true) {
    ...
    board[x][y] = shipBoard[x-1][y-1] == 1 ? 'O':'X';
}

```

그리고 2차원 char배열의 각 요소는 1차원 배열이므로 아래의 왼쪽과 같이 간단히 출력할 수 있다. 원래는 오른쪽과 같이 배열의 각 요소를 하나씩 출력해야하는데, println메서드로 1차원 char배열의 참조변수를 출력하면, 배열의 모든 요소를 한 줄로 출력한다.

| 참고 | 그림5-5에서 알 수 있듯이 2차원 배열의 각 요소는 1차원 배열의 참조변수 역할을 한다.

<pre> for(int i=0;i<SIZE;i++) { System.out.println(board[i]); } </pre>	<pre> for(int i=0;i<SIZE;i++) { for(int j=0;j<SIZE;j++) System.out.print(board[i][j]); System.out.println(); } </pre>
---	---

println메서드가 모든 1차원 배열을 이렇게 출력할 수 있는 것은 아니고, char배열인 경우만 가능하다.

▼ 예제 5-21/ch5/MultiArrEx2.java

```

import java.util.*;

class MultiArrEx2 {
    public static void main(String[] args) {
        final int SIZE = 5;
        int x = 0, y = 0, num = 0;

        int[][] bingo = new int[SIZE][SIZE];
        Scanner scanner = new Scanner(System.in);

        // 배열의 모든 요소를 1부터 SIZE*SIZE까지의 숫자로 초기화
        for(int i=0;i<SIZE;i++) {
            for(int j=0;j<SIZE;j++)
                bingo[i][j] = i*SIZE + j + 1;

        // 배열에 저장된 값을 뒤섞는다. (shuffle)
        for(int i=0;i<SIZE;i++) {
            for(int j=0;j<SIZE;j++) {
                x = (int)(Math.random() * SIZE);
                y = (int)(Math.random() * SIZE);

                // bingo[i][j]와 임의로 선택된 값(bingo[x][y])을 바꾼다.
                int tmp = bingo[i][j];
                bingo[i][j] = bingo[x][y];
                bingo[x][y] = tmp;
            }
        }

        do {
            for(int i=0;i<SIZE;i++) {
                for(int j=0;j<SIZE;j++)
                    System.out.printf("%2d ", bingo[i][j]);
                System.out.println();
            }
            System.out.println();

            System.out.printf("1~%d의 숫자를 입력하세요. (종료:0) >", SIZE*SIZE);
            String tmp = scanner.nextLine(); // 화면에서 입력받은 내용을 tmp에 저장
            num = Integer.parseInt(tmp); // 입력받은 문자열(tmp)를 숫자로 변환

            // 입력받은 숫자와 같은 숫자가 저장된 요소를 찾아서 0을 저장
            outer:
            for(int i=0;i<SIZE;i++) {
                for(int j=0;j<SIZE;j++) {
                    if(bingo[i][j]==num) {
                        bingo[i][j] = 0;
                        break outer; // 2중 반복문을 벗어난다.
                    }
                }
            }

        } while(num!=0);
    } // main의 끝
}

```

▼ 실행결과

```
9 22 2 12 17  
6 1 25 3 5  
16 7 11 19 23  
14 10 21 13 8  
4 15 20 24 18
```

1~25의 숫자를 입력하세요. (종료: 0) >1

```
9 22 2 12 17  
6 0 25 3 5  
16 7 11 19 23  
14 10 21 13 8  
4 15 20 24 18
```

1~25의 숫자를 입력하세요. (종료: 0) >9

```
0 22 2 12 17  
6 0 25 3 5  
16 7 11 19 23  
14 10 21 13 8  
4 15 20 24 18
```

1~25의 숫자를 입력하세요. (종료: 0) >0

5×5크기의 빙고판에 1~25의 숫자를 차례로 저장한 다음에, Math.random()을 이용해서 저장된 값의 위치를 섞는다. 여기까지의 과정은 지금까지 여러 번 반복된 것이므로 설명을 생략한다.

그 다음에 사용자로부터 숫자를 입력받아서 일치하는 숫자가 빙고판에 있으면 해당 숫자를 0으로 바꾼다.

```
// 입력받은 숫자와 같은 숫자가 저장된 요소를 찾아서 0을 저장  
outer:  
for(int i = 0; i < SIZE; i++) {  
    for(int j = 0; j < SIZE; j++) {  
        if(bingo[i][j] == num) {  
            bingo[i][j] = 0;           // 일치하는 숫자를 찾으면 0으로 변경  
            break outer;          // 2중 반복문을 벗어난다.  
        }  
    }  
}
```

입력받은 숫자와 일치하는 숫자를 빙고판에서 찾는 방법은 간단하다. 배열의 첫 번째 요소부터 순서대로 하나씩 비교하다 일치하는 숫자를 찾으면, 값을 0으로 바꾸고 break문으로 반복문을 빠져나오면 된다. 2중 반복문이므로, 이름 붙은 break문을 사용해야 한다.

▼ 예제 5-22/ch5/MultiArrEx3.java

```

class MultiArrEx3 {
    public static void main(String[] args) {
        int[][] m1 = {
            {1, 2, 3},
            {4, 5, 6}
        };

        int[][] m2 = {
            {1, 2},
            {3, 4},
            {5, 6}
        };

        final int ROW      = m1.length;      // m1의 행 길이
        final int COL      = m2[0].length;   // m2의 열 길이
        final int M2_ROW   = m2.length;     // m2의 행 길이

        int[][] m3 = new int[ROW][COL];

        // 행렬곱 m1 x m2의 결과를 m3에 저장
        for(int i=0;i<ROW;i++)
            for(int j=0;j<COL;j++)
                for(int k=0;k<M2_ROW;k++)
                    m3[i][j] += m1[i][k] * m2[k][j];

        // 행렬 m3을 출력
        for(int i=0;i<ROW;i++) {
            for(int j=0;j<COL;j++) {
                System.out.printf("%3d ", m3[i][j]);
            }
            System.out.println();
        }
    } // main의 끝
}

```

▼ 실행결과
22 28
49 64

수학에서 두 개의 행렬(matrix) m1과 m2가 있을 때, 이 두 행렬을 곱한 결과인 행렬 m3는 아래와 같이 정의된다.

$$\begin{matrix} m1 & & m2 & & m3 \\ \left(\begin{matrix} A0 & A1 & A2 \\ B0 & B1 & B2 \end{matrix} \right) & \times & \left(\begin{matrix} a0 & a1 \\ b0 & b1 \\ c0 & c1 \end{matrix} \right) & = & \left(\begin{matrix} A0 \times a0 + A1 \times b0 + A2 \times c0 & A0 \times a1 + A1 \times b1 + A2 \times c1 \\ B0 \times a0 + B1 \times b0 + B2 \times c0 & B0 \times a1 + B1 \times b1 + B2 \times c1 \end{matrix} \right) \end{matrix}$$

두 행렬의 곱셈이 가능하려면, m1의 열의 길이와 m2의 행의 길이가 같아야 한다는 조건이 있다. 위의 경우에는 m1이 2행 3열이고, m2가 3행 2열이므로 곱셈이 가능하다.

그리고 곱셈연산의 결과인 행렬 m3의 행의 길이는 m1의 행의 길이와 같고, 열의 길이는 m2의 열의 길이와 같다. 2행 3열인 행렬과 3행 2열인 행렬을 곱하면 결과는 2행 2열의 행렬이 되는 것이다.

지금까지의 내용은 수학에서 정의된 행렬의 곱셈에 대한 정의와 규칙일 뿐, 왜 그렇게 되는지 따질 필요는 없다. 이 내용에 맞게 코드를 작성하는 것에 중점을 두자. 위의 행렬의 곱셈공식을 2차원 배열로 표현하면 아래와 같다.

$$\begin{array}{c}
 \text{m1} \\
 \begin{array}{|c|c|c|} \hline
 m1[0][0] & m1[0][1] & m1[0][2] \\ \hline
 m1[1][0] & m1[1][1] & m1[1][2] \\ \hline
 \end{array}
 \end{array}
 \times
 \begin{array}{c}
 \text{m2} \\
 \begin{array}{|c|c|} \hline
 m2[0][0] & m2[0][1] \\ \hline
 m2[1][0] & m2[1][1] \\ \hline
 m2[2][0] & m2[2][1] \\ \hline
 \end{array}
 \end{array}
 =
 \begin{array}{c}
 \text{m3} \\
 \begin{array}{|c|c|} \hline
 m3[0][0] & m3[0][1] \\ \hline
 m3[1][0] & m3[1][1] \\ \hline
 \end{array}
 \end{array}$$

그리고 행렬 곱셈의 결과인 행렬 m3의 각 요소들은 아래와 같이 계산된다.

$$\begin{aligned}
 m3[0][0] &= m1[0][0] * m2[0][0] \\
 &\quad + m1[0][1] * m2[1][0] \\
 &\quad + m1[0][2] * m2[2][0]; \\
 m3[0][1] &= m1[0][0] * m2[0][1] \\
 &\quad + m1[0][1] * m2[1][1] \\
 &\quad + m1[0][2] * m2[2][1]; \\
 m3[1][0] &= m1[1][0] * m2[0][0] \\
 &\quad + m1[1][1] * m2[1][0] \\
 &\quad + m1[1][2] * m2[2][0]; \\
 m3[1][1] &= m1[1][0] * m2[0][1] \\
 &\quad + m1[1][1] * m2[1][1] \\
 &\quad + m1[1][2] * m2[2][1];
 \end{aligned}$$

위의 문장들을 자세히 들여다보면, 행렬 m3의 행index가 행렬 m1의 행index와 일치하고, m3의 열index가 m2의 열index와 일치한다는 것을 알 수 있다. 그래서 위의 문장들은 다음과 같이 2중 for문으로 대체할 수 있다.

```

for(int i=0;i<2;i++) { // i = 0, 1
    for(int j=0;j<2;j++) { // j = 0, 1
        m3[i][j] = m1[i][0] * m2[0][j]
                    + m1[i][1] * m2[1][j]
                    + m1[i][2] * m2[2][j];
    }
}
    
```

위의 문장을 잘 보면 여전히 반복되는 부분이 있다. m1의 열index와 m2의 행index가 동일하게 0부터 2까지 1씩 증가한다. 이 부분을 또 하나의 for문으로 바꾸면 다음과 같이 된다.

```

for(int i=0;i<2;i++) {
    for(int j=0;j<2;j++) {
        for(int k=0;k<3;k++) { // k = 0, 1, 2
            m3[i][j] += m1[i][k] * m2[k][j];
        }
    }
}
    
```

행렬 m1과 m2의 길이가 달라져도 행렬 m3가 계산될 수 있도록, 배열 m3의 크기와 for 문의 조건식이 동적으로 계산되게 하였다.

```
final int ROW      = m1.length;      // m1의 행 길이(m3의 행 길이)
final int COL      = m2[0].length;    // m2의 열 길이(m3의 열 길이)
final int M2_ROW   = m2.length;      // m2의 행 길이

int[][] m3 = new int[ROW][COL];
```

두 행렬의 곱셈이 가능하려면, 배열 m1의 열의 길이와 배열 m2의 행의 길이가 일치해야 한다는 것에 주의하자.

▼ 예제 5-23/ch5/MultiArrEx4.java

```
import java.util.*;

class MultiArrEx4{
    public static void main(String[] args) {
        String[][] words = {
            {"chair", "의자"},      // words[0][0], words[0][1]
            {"computer", "컴퓨터"}, // words[1][0], words[1][1]
            {"integer", "정수"}     // words[2][0], words[2][1]
        };

        Scanner scanner = new Scanner(System.in);

        for(int i=0;i<words.length;i++) {
            System.out.printf("Q%d. %s의 뜻은?", i+1, words[i][0]);

            String tmp = scanner.nextLine();

            if(tmp.equals(words[i][1])) {
                System.out.printf("정답입니다.%n%n");
            } else {
                System.out.printf("틀렸습니다. 정답은 %s입니다.%n%n", words[i][1]);
            }
        } // for
    } // main의 끝
}
```

▼ 실행결과

Q1. chair의 뜻은?dmlwk
틀렸습니다. 정답은 의자입니다.

Q2. computer의 뜻은?컴퓨터
정답입니다.

Q3. integer의 뜻은?정수
정답입니다.

영단어를 보여주고 단어의 뜻을 맞추는 예제이다. words[i][0]은 문제이고, words[i][1]은 답이다. words[i][0]을 화면에 보여주고, 입력받은 답은 tmp에 저장한다.

```
System.out.printf("Q%d. %s의 뜻은?", i+1, words[i][0]);
String tmp = scanner.nextLine();
```

그 다음엔 equals()로 tmp와 words[i][1]을 비교해서 정답인지 확인한다.

```
if(tmp.equals(words[i][1])) {
    System.out.printf("정답입니다.%n%n");
} else {
    System.out.printf("틀렸습니다. 정답은 %s입니다.%n%n",words[i][1]);
}
```

Java Programming Language

Chapter 06

객체지향 프로그래밍 I

Object-oriented Programming I

1. 객체지향언어

1.1 객체지향언어의 역사

요즘은 컴퓨터의 눈부신 발전으로 활용 폭이 넓고 다양해져서 컴퓨터가 사용되지 않는 분야가 없을 정도지만, 초창기에는 주로 과학실험이나 미사일 발사실험과 같은 모의실험(simulation)을 목적으로 사용되었다. 이 시절의 과학자들은 모의실험을 위해 실제 세계와 유사한 가상 세계를 컴퓨터 속에 구현하고자 노력하였으며 이러한 노력은 객체지향이론을 탄생시켰다.

객체지향이론의 기본 개념은 '실제 세계는 사물(객체)로 이루어져 있으며, 발생하는 모든 사건들은 사물간의 상호작용이다.'라는 것이다. 실제 사물의 속성과 기능을 분석한 다음, 데이터(변수)와 함수로 정의함으로써 실제 세계를 컴퓨터 속에 옮겨 놓은 것과 같은 가상 세계를 구현하고 이 가상세계에서 모의실험을 함으로써 많은 시간과 비용을 절약할 수 있었다. 객체지향이론은 상속, 캡슐화, 추상화 개념을 중심으로 점차 구체적으로 발전되었으며 1960년대 중반에 객체지향이론을 프로그래밍언어에 적용한 시뮬라(Simula)라는 최초의 객체지향언어가 탄생하였다.

그 당시에는 FORTRAN이나 COBOL과 같은 절차적 언어들이 주류를 이루었으며, 객체지향언어는 널리 사용되지 못하고 있었다. 1980년대 중반에 C++을 비롯하여 여러 객체지향언어가 발표되면서 객체지향언어가 본격적으로 개발자들의 관심을 끌기 시작하였지만 여전히 사용자층이 넓지 못했다.

그러나 프로그램의 규모가 점점 커지고 사용자들의 요구가 빠르게 변화해가는 상황을 절차적 언어로는 극복하기 어렵다는 한계를 느끼고 객체지향언어를 이용한 개발방법론이 대안으로 떠오르게 되면서 조금씩 입지를 넓혀가고 있었다.

자바가 1995년에 발표되고 1990년대 말에 인터넷의 발전과 함께 크게 유행하면서 객체지향언어는 이제 프로그래밍언어의 주류로 자리 잡았다.

1.2 객체지향언어

객체지향언어는 기존의 프로그래밍언어와 다른 전혀 새로운 것이 아니라, 기존의 프로그래밍 언어에 몇 가지 새로운 규칙을 추가한 보다 발전된 형태의 것이다. 이러한 규칙들을 이용해서 코드 간에 서로 관계를 맺어 줌으로써 보다 유기적으로 프로그램을 구성하는 것이 가능해졌다. 기존의 프로그래밍 언어에 익숙한 사람이라면 자바의 객체지향적인 부분만 새로 배우면 된다. 다만 절차적 언어에 익숙한 프로그래밍 습관을 객체지향적으로 바꾸도록 노력해야 할 것이다. 객체지향언어의 주요특징은 다음과 같다.

1. 코드의 재사용성이 높다.

새로운 코드를 작성할 때 기존의 코드를 이용하여 쉽게 작성할 수 있다.

2. 코드의 관리가 용이하다.

코드간의 관계를 이용해서 적은 노력으로 쉽게 코드를 변경할 수 있다.

3. 신뢰성이 높은 프로그래밍을 가능하게 한다.

제어자와 메서드를 이용해서 데이터를 보호하고 올바른 값을 유지하도록 하며, 코드의 중복을 제거하여 코드의 불일치로 인한 오동작을 방지할 수 있다.

객체지향언어의 가장 큰 장점은 '코드의 재사용성이 높고 유지보수가 용이하다.'는 것이다. 이러한 객체지향언어의 장점은 프로그램의 개발과 유지보수에 드는 시간과 비용을 획기적으로 개선하였다.

앞으로 상속, 다형성과 같은 객체지향개념을 학습할 때 재사용성과 유지보수 그리고 중복된 코드의 제거, 이 세 가지 관점에서 보면 보다 쉽게 이해할 수 있을 것이다.

객체지향 프로그래밍은 프로그래머에게 거시적 관점에서 설계할 수 있는 능력을 요구하기 때문에 객체지향개념을 이해했다 하더라도 자바의 객체지향적 장점들을 충분히 활용한 프로그램을 작성하기란 쉽지 않을 것이다.

너무 객체지향개념에 얹매여서 고민하기보다는 일단 프로그램을 기능적으로 완성한 다음 어떻게 하면 보다 객체지향적으로 코드를 개선할 수 있을지를 고민하여 점차 개선해 나가는 것이 좋다.

이러한 경험들이 축적되어야 프로그램을 객체지향적으로 설계할 수 있는 능력이 길러지는 것인지 처음부터 이론을 많이 안다고 해서 좋은 설계를 할 수 있는 것은 아니다.

2. 클래스와 객체

2.1 클래스와 객체의 정의와 용도

클래스란 '객체를 정의해놓은 것.' 또는 클래스는 '객체의 설계도 또는 틀'이라고 정의할 수 있다. 클래스는 객체를 생성하는데 사용되며, 객체는 클래스에 정의된 대로 생성된다.

클래스의 정의 클래스란 객체를 정의해 놓은 것이다.

클래스의 용도 클래스는 객체를 생성하는데 사용된다.

객체의 사전적인 정의는, '실제로 존재하는 것'이다. 우리가 주변에서 볼 수 있는 책상, 의자, 자동차와 같은 사물들이 곧 객체이다. 객체지향이론에서는 사물과 같은 유형적인 것뿐만 아니라, 개념이나 논리와 같은 무형적인 것들도 객체로 간주한다.

프로그래밍에서의 객체는 클래스에 정의된 내용대로 메모리에 생성된 것을 뜻한다.

객체의 정의 실제로 존재하는 것. 사물 또는 개념

객체의 용도 객체가 가지고 있는 기능과 속성에 따라 다름

유형의 객체 책상, 의자, 자동차, TV와 같은 사물

무형의 객체 수학공식, 프로그램 애러와 같은 논리나 개념

클래스와 객체의 관계를 우리가 살고 있는 실생활에서 예를 들면, 제품 설계도와 제품과의 관계라고 할 수 있다. 예를 들면, TV설계도(클래스)는 TV라는 제품(객체)을 정의한 것이며, TV(객체)를 만드는데 사용된다.

또한 클래스는 단지 객체를 생성하는데 사용될 뿐, 객체 그 자체는 아니다. 우리가 원하는 기능의 객체를 사용하기 위해서는 먼저 클래스로부터 객체를 생성하는 과정이 선행되어야 한다.

우리가 TV를 보기 위해서는, TV(객체)가 필요한 것이지 TV설계도(클래스)가 필요한 것은 아니며, TV설계도(클래스)는 단지 TV라는 제품(객체)을 만드는 데만 사용될 뿐이다.

그리고 TV설계도를 통해 TV가 만들어진 후에야 사용할 수 있다. 프로그래밍에서는 먼저 클래스를 작성한 다음, 클래스로부터 객체를 생성하여 사용한다.

| 참고 | 객체를 사용한다는 것은 객체가 가지고 있는 속성과 기능을 사용한다는 뜻이다.

클래스	객체
제품 설계도	제품
TV 설계도	TV
붕어빵 기계	붕어빵

▲ 표 6-1 클래스와 객체의 예

클래스를 정의하고 클래스를 통해 객체를 생성하는 이유는 설계도를 통해서 제품을 만드는 이유와 같다. 하나의 설계도만 잘 만들어 놓으면 제품을 만드는 일이 쉬워진다. 제품을 만들 때마다 매번 고민할 필요없이 설계도대로만 만들면 되기 때문이다.

설계도 없이 제품을 만든다고 생각해보라. 복잡한 제품일수록 설계도 없이 제품을 만든다는 것은 상상할 수도 없을 것이다.

이와 마찬가지로 클래스를 한번만 잘 만들어 놓기만 하면, 매번 객체를 생성할 때마다 어떻게 객체를 만들어야 할지를 고민하지 않아도 된다. 그냥 클래스로부터 객체를 생성해서 사용하기만 하면 되는 것이다.

JDK(Java Development Kit)에서는 프로그래밍을 위해 많은 수의 유용한 클래스(Java API)를 기본적으로 제공하고 있으며, 우리는 이 클래스들을 이용해서 원하는 기능의 프로그램을 보다 쉽게 작성할 수 있다.

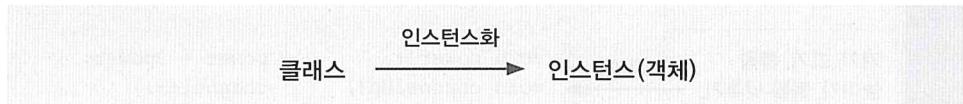
2.2 객체와 인스턴스

클래스로부터 객체를 만드는 과정을 클래스의 인스턴스화(instantiate)라고 하며, 어떤 클래스로부터 만들어진 객체를 그 클래스의 인스턴스(instance)라고 한다.

예를 들면, Tv클래스로부터 만들어진 객체를 Tv클래스의 인스턴스라고 한다. 결국 인스턴스는 객체와 같은 의미이지만, 객체는 모든 인스턴스를 대표하는 포괄적인 의미를 갖고 있으며, 인스턴스는 어떤 클래스로부터 만들어진 것인지를 강조하는 보다 구체적인 의미를 갖고 있다.

예를 들면, ‘책상은 인스턴스다.’라고 하기 보다는 ‘책상은 객체다.’라는 쪽이, ‘책상은 책상 클래스의 객체이다.’라고 하기 보다는 ‘책상은 책상 클래스의 인스턴스다.’라고 하는 것이 더 자연스럽다.

인스턴스와 객체는 같은 의미이므로 두 용어의 사용을 엄격히 구분할 필요는 없지만, 위의 예에서 본 것과 같이 문맥에 따라 구별하여 사용하는 것이 좋다.



2.3 객체의 구성요소 – 속성과 기능

객체는 속성과 기능, 두 종류의 구성요소로 이루어져 있으며, 일반적으로 객체는 다수의 속성과 다수의 기능을 갖는다. 즉, 객체는 속성과 기능의 집합이라고 할 수 있다. 그리고 객체가 가지고 있는 속성과 기능을 그 객체의 멤버(구성원, member)라 한다.

클래스란 객체를 정의한 것이므로 클래스에는 객체의 모든 속성과 기능이 정의되어 있다. 클래스로부터 객체를 생성하면, 클래스에 정의된 속성과 기능을 가진 객체가 만들어지는 것이다.

속성과 기능은 아래와 같이 같은 뜻의 여러 가지 용어가 있으며, 앞으로 이 중에서도 ‘속성’보다는 ‘멤버변수’를, ‘기능’보다는 ‘메서드’를 주로 사용할 것이다.

속성(property)	멤버변수(member variable), 특성(attribute), 필드(field), 상태(state)
기능(function)	메서드(method), 함수(function), 행위(behavior)

보다 쉽게 이해할 수 있도록 TV를 예로 들어보자. TV의 속성으로는 전원상태, 크기, 길이, 높이, 색상, 볼륨, 채널과 같은 것들이 있으며, 기능으로는 켜기, 끄기, 볼륨 높이기, 채널 변경하기 등이 있다.

속성	크기, 길이, 높이, 색상, 볼륨, 채널 등
기능	켜기, 끄기, 볼륨 높이기, 볼륨 낮추기, 채널 변경하기 등

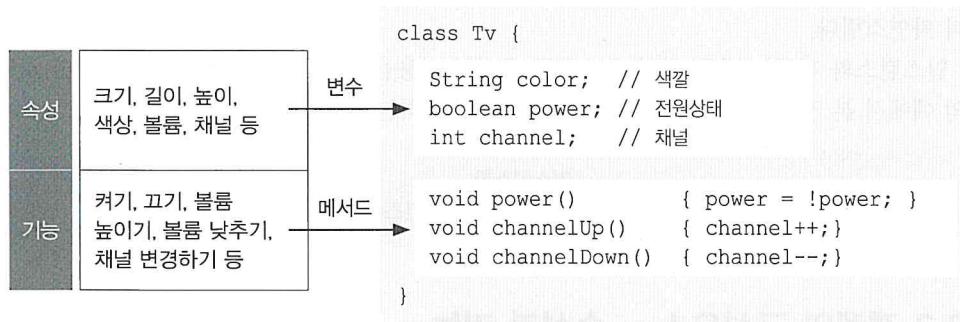
▲ 그림 6-1 TV의 속성과 기능

객체지향 프로그래밍에서는 속성과 기능을 각각 변수와 메서드로 표현한다.

속성(property) → 멤버변수(variable)
기능(function) → 메서드(method)

채널 → int channel
채널 높이기 → channelUp() { ... }

위에서 분석한 내용을 토대로 Tv클래스를 만들어 보면 다음과 같다.



참고! 멤버변수와 메서드를 선언하는데 있어서 순서는 관계없지만, 일반적으로 메서드보다는 멤버변수를 먼저 선언하고 멤버변수는 멤버변수끼리 메서드끼리 모아 놓는 것이 일반적이다.

실제 TV가 갖는 기능과 속성은 이 외에도 더 있지만, 프로그래밍에 필요한 속성과 기능만을 선택하여 클래스를 작성하면 된다.

각 변수의 자료형은 속성의 값에 알맞은 것을 선택해야한다. 전원상태(power)의 경우, on과 off 두 가지 값을 가질 수 있으므로 boolean형으로 선언했다.

power()의 ‘power = !power;’ 이 문장에서 power의 값이 true면 false로, false면 true로 변경하는 일을 한다. power의 값에 관계없이 항상 반대의 값으로 변경해주면 되므로 굳이 if문을 사용할 필요가 없다. 참고로 if문을 사용하여 코드를 작성하면 다음과 같다.

```
if (power) // if (power == true)
    power = false;
else
    power = true;
```

2.4 인스턴스의 생성과 사용

Tv클래스를 선언한 것은 Tv설계도를 작성한 것에 불과하므로, Tv인스턴스를 생성해야 제품(Tv)을 사용할 수 있다. 클래스로부터 인스턴스를 생성하는 방법은 여러 가지가 있지만 일반적으로는 다음과 같이 한다.

```
클래스명 변수명; // 클래스의 객체를 참조하기 위한 참조변수를 선언
변수명 = new 클래스명(); // 클래스의 객체를 생성 후, 객체의 주소를 참조변수에 저장

Tv t; // Tv클래스 타입의 참조변수 t를 선언
t = new Tv(); // Tv인스턴스를 생성한 후, 생성된 Tv인스턴스의 주소를 t에 저장
```

▼ 예제 6-1/ch6/TvTest.java

```

class Tv {
    // Tv의 속성(멤버변수)
    String color;           // 색상
    boolean power;          // 전원상태(on/off)
    int channel;            // 채널

    // Tv의 기능(메서드)
    void power() { power = !power; } // TV를 켜거나 끄는 기능을 하는 메서드
    void channelUp() { ++channel; } // TV의 채널을 높이는 기능을 하는 메서드
    void channelDown() { --channel; } // TV의 채널을 낮추는 기능을 하는 메서드
}

class TvTest {
    public static void main(String args[]) {
        Tv t;                  // Tv인스턴스를 참조하기 위한 변수 t를 선언
        t = new Tv();           // Tv인스턴스를 생성한다.
        t.channel = 7;          // Tv인스턴스의 멤버변수 channel의 값을 7로 한다.
        t.channelDown();        // Tv인스턴스의 메서드 channelDown()을 호출한다.
        System.out.println("현재 채널은 " + t.channel + " 입니다.");
    }
}

```

▼ 실행결과
현재 채널은 6 입니다.

이 예제는 `Tv` 클래스로부터 인스턴스를 생성하고 인스턴스의 속성(`channel`)과 메서드(`channelDown()`)를 사용하는 방법을 보여 주는 것이다. 이 예제를 그림과 함께 단계별로 자세히 살펴보도록 하자.

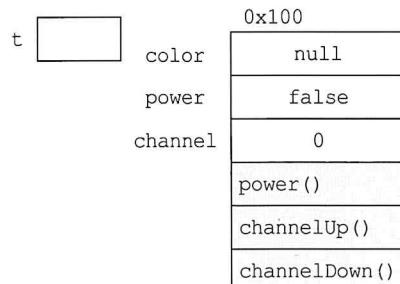
1. `Tv t;`

`Tv` 클래스 타입의 참조변수 `t`를 선언한다. 메모리에 참조변수 `t`를 위한 공간이 마련된다. 아직 인스턴스가 생성되지 않았으므로 참조변수로 아무것도 할 수 없다.

2. `t = new Tv();`

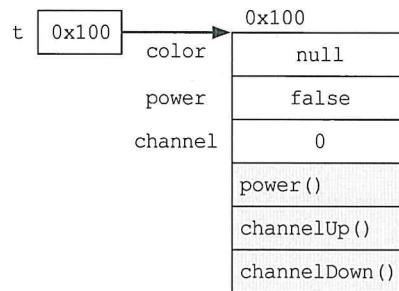
연산자 `new`에 의해 `Tv` 클래스의 인스턴스가 메모리의 빈 공간에 생성된다. 주소가 `0x100`인 곳에 생성되었다고 가정하자. 이 때, 멤버변수는 각 자료형에 해당하는 기본값으로 초기화 된다.

`color`은 참조형이므로 `null`로, `power`는 `boolean`이므로 `false`로, 그리고 `channel`은 `int`이므로 0으로 초기화 된다.



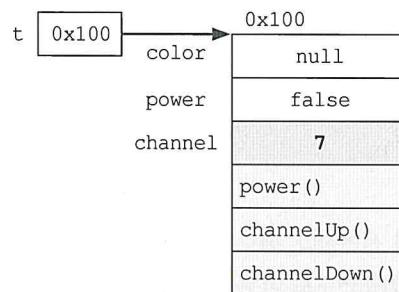
그 다음에는 대입연산자(=)에 의해서 생성된 객체의 주소값이 참조변수 t에 저장된다. 이제는 참조변수 t를 통해 Tv인스턴스에 접근할 수 있다. 인스턴스를 다루기 위해서는 참조변수가 반드시 필요하다.

| 참고 | 아래 그림에서의 화살표는 참조변수 t가 Tv인스턴스를 참조하고 있다는 것을 알기 쉽게 하기 위해 추가한 상징적인 것이다. 이 때, 참조변수 t가 Tv인스턴스를 '가리키고 있다' 또는 '참조하고 있다'라고 한다.



3. t.channel = 7 ;

참조변수 t에 저장된 주소에 있는 인스턴스의 멤버변수 channel에 7을 저장한다. 여기서 알 수 있는 것처럼, 인스턴스의 멤버변수(속성)를 사용하려면 '참조변수.멤버변수'와 같이 하면 된다.

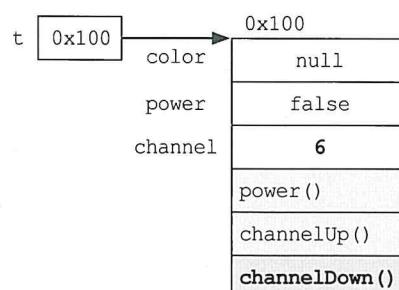


4. t.channelDown();

참조변수 t가 참조하고 있는 Tv인스턴스의 channelDown메서드를 호출한다. channel Down메서드는 멤버변수 channel에 저장되어 있는 값을 1 감소시킨다.

```
void channelDown() { --channel; }
```

channelDown()에 의해서 channel의 값은 7에서 6이 된다.



5. `System.out.println("현재 채널은 " + t.channel + " 입니다.");`

참조변수 t가 참조하고 있는 Tv인스턴스의 멤버변수 channel에 저장되어 있는 값을 출력한다. 현재 channel의 값은 60이므로 '현재 채널은 6 입니다.'가 화면에 출력된다.

인스턴스와 참조변수의 관계는 마치 우리가 일상생활에서 사용하는 TV와 TV리모콘의 관계와 같다. TV리모콘(참조변수)을 사용하여 TV(인스턴스)를 다루기 때문이다. 다른 점이라면, 인스턴스는 오직 참조변수를 통해서만 다룰 수 있다는 것이다.

그리고 TV를 사용하려면 TV 리모콘을 사용해야하고, 에어콘을 사용하려면, 에어콘 리모콘을 사용해야하는 것처럼 Tv인스턴스를 사용하려면, Tv클래스 타입의 참조변수가 필요한 것이다.

인스턴스는 참조변수를 통해서만 다룰 수 있으며,

참조변수의 타입은 인스턴스의 타입과 일치해야한다.

▼ 예제 6-2/ch6/TvTest2.java

```

class Tv {
    // Tv의 속성(멤버변수)
    String color;           // 색상
    boolean power;          // 전원상태 (on/off)
    int channel;            // 채널

    // Tv의 기능(메서드)
    void power() { power = !power; } // TV를 켜거나 끄는 기능을 하는 메서드
    void channelUp() { ++channel; } // TV의 채널을 높이는 기능을 하는 메서드
    void channelDown() { --channel; } // TV의 채널을 낮추는 기능을 하는 메서드
}

class TvTest2 {
    public static void main(String args[]) {
        Tv t1 = new Tv(); // Tv t1; t1 = new Tv();를 한 문장으로 가능
        Tv t2 = new Tv();
        System.out.println("t1의 channel값은 " + t1.channel + "입니다.");
        System.out.println("t2의 channel값은 " + t2.channel + "입니다.");

        t1.channel = 7; // channel 값을 7으로 한다.
        System.out.println("t1의 channel값을 7로 변경하였습니다.");

        System.out.println("t1의 channel값은 " + t1.channel + "입니다.");
        System.out.println("t2의 channel값은 " + t2.channel + "입니다.");
    }
}

```

▼ 실행결과

```

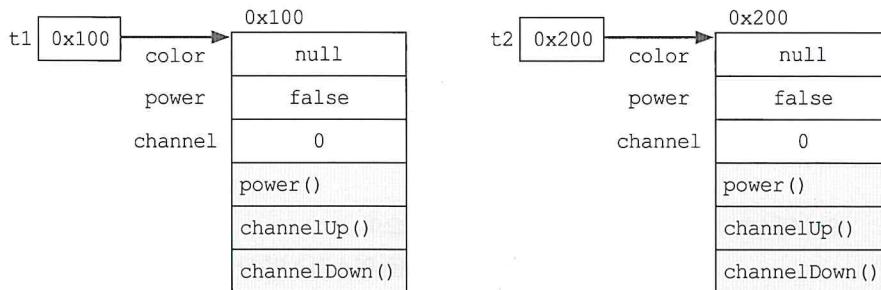
t1의 channel값은 0입니다.
t2의 channel값은 0입니다.
t1의 channel값을 7로 변경하였습니다.
t1의 channel값은 7입니다.
t2의 channel값은 0입니다.

```

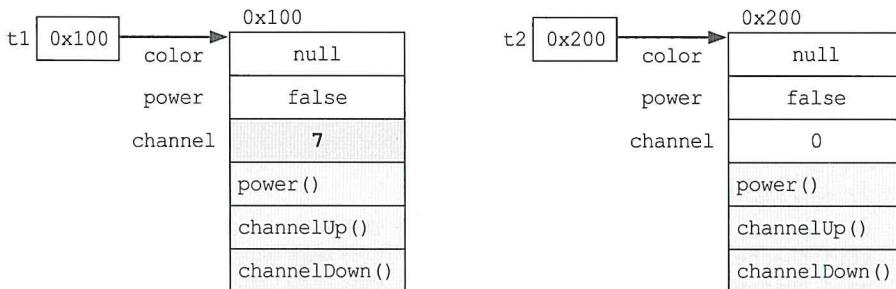
위의 예제는 Tv클래스의 인스턴스 t1과 t2를 생성한 후에, 인스턴스 t1의 멤버변수인 channel의 값을 변경하였다.

| 참고 | 참조변수 t1이 가리키고(참조하고) 있는 인스턴스를 간단히 인스턴스 t1이라고 했다.

1. `Tv t1 = new Tv();
Tv t2 = new Tv();`



2. `t1.channel = 7; // t1이 가리키고 있는 인스턴스의 멤버변수 channel의 값을 7로 변경한다.`



같은 클래스로부터 생성되었을지라도 각 인스턴스의 속성(멤버변수)은 서로 다른 값을 유지할 수 있으며, 메서드의 내용은 모든 인스턴스에 대해 동일하다.

▼ 예제 6-3/ch6/TvTest3.java

```
class Tv {
    // Tv의 속성(멤버변수)
    String color;           // 색상
    boolean power;          // 전원상태(on/off)
    int channel;            // 채널

    // Tv의 기능(메서드)
    void power() { power = !power; } // TV를 켜거나 끄는 기능을 하는 메서드
    void channelUp() { ++channel; } // TV의 채널을 높이는 기능을 하는 메서드
    void channelDown() { --channel; } // TV의 채널을 낮추는 기능을 하는 메서드
}

class TvTest3 {
    public static void main(String args[]) {
        Tv t1 = new Tv();
        Tv t2 = new Tv();
        System.out.println("t1의 channel값은 " + t1.channel + "입니다.");
        System.out.println("t2의 channel값은 " + t2.channel + "입니다.");
    }
}
```

```

        t2 = t1;           // t1이 저장하고 있는 값(주소)을 t2에 저장한다.
        t1.channel = 7; // channel 값을 7로 한다.
        System.out.println("t1의 channel값을 7로 변경하였습니다.");
        System.out.println("t1의 channel값은 " + t1.channel + "입니다.");
        System.out.println("t2의 channel값은 " + t2.channel + "입니다.");
    }
}

```

▼ 실행결과

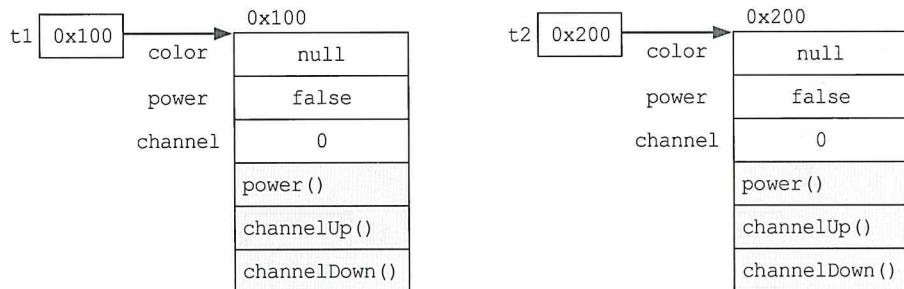
```

t1의 channel값은 0입니다.
t2의 channel값은 0입니다.
t1의 channel값을 7로 변경하였습니다.
t1의 channel값은 7입니다.
t2의 channel값은 7입니다.

```

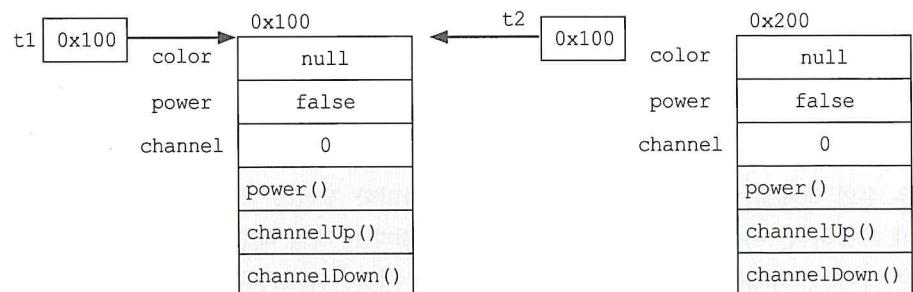
이 예제의 실행과정을 그림과 함께 설명하면 다음과 같다.

1. `Tv t1 = new Tv();`
`Tv t2 = new Tv();`



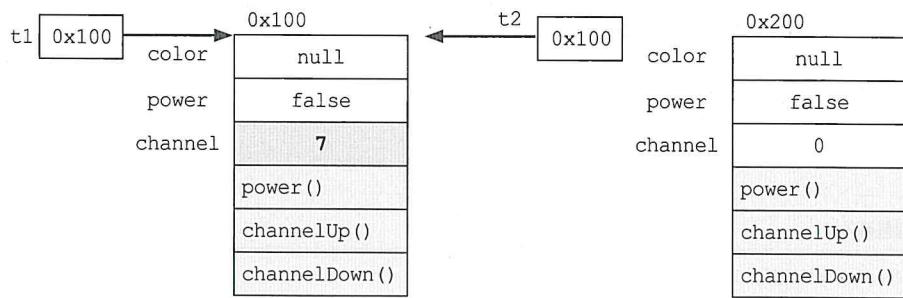
2. `t2 = t1;` // t1이 저장하고 있는 값(주소)을 t2에 저장한다.

t1은 참조변수이므로, 인스턴스의 주소를 저장하고 있다. 이 문장이 수행되면, t2가 가지고 있던 값은 잃어버리게 되고 t1에 저장되어 있던 값이 t2에 저장되게 된다. 그렇게 되면 t2 역시 t1이 참조하고 있던 인스턴스를 같이 참조하게 되고, t2가 원래 참조하고 있던 인스턴스는 더 이상 사용할 수 없게 된다.



| 참고 | 자신을 참조하고 있는 참조변수가 하나도 없는 인스턴스는 더 이상 사용되어질 수 없으므로 '가비지 컬렉터(Garbage Collector)'에 의해서 자동적으로 메모리에서 제거된다.

3. `t1.channel = 7;` // channel 값을 7로 한다.



4. `System.out.println("t1의 channel값은 "+t1.channel+"입니다.");`

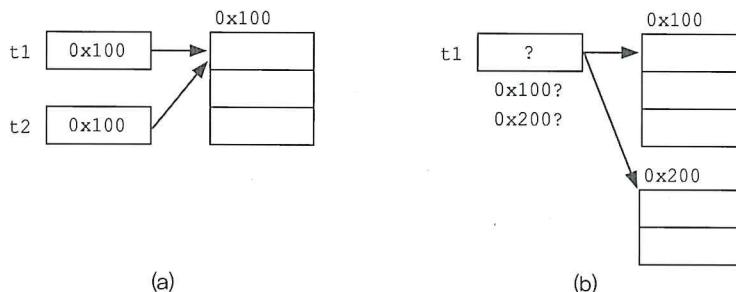
`System.out.println("t2의 channel값은 "+t2.channel+"입니다.");`

이제 t1, t2 모두 같은 Tv클래스의 인스턴스를 가리키고 있기 때문에 t1.channel과 t2.channel의 값은 7이며 다음과 같은 결과가 화면에 출력된다.

t1의 channel값은 7입니다.

t2의 channel값은 7입니다.

이 예제에서 알 수 있듯이, 참조변수에는 하나의 값(주소)만이 저장될 수 있으므로 둘 이상의 참조변수가 하나의 인스턴스를 가리키는(참조하는) 것은 가능하지만 하나의 참조변수로 여러 개의 인스턴스를 가리키는 것은 가능하지 않다.



(a) 하나의 인스턴스를 여러 개의 참조변수가 가리키는 경우(가능)

(b) 여러 인스턴스를 하나의 참조변수가 가리키는 경우(불가능)

▲ 그림 6-2 참조변수와 인스턴스의 관계

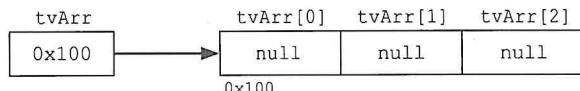
2.5 객체 배열

많은 수의 객체를 다뤄야 할 때, 배열로 다루면 편리할 것이다. 객체 역시 배열로 다루는 것이 가능하며, 이를 ‘객체 배열’이라고 한다. 그렇다고 객체 배열 안에 객체가 저장되는 것은 아니고, 객체의 주소가 저장된다. 사실 객체 배열은 참조변수들을 하나로 묶은 참조 변수 배열인 것이다.



길이가 3인 객체 배열 tvArr을 아래와 같이 생성하면, 각 요소는 참조변수의 기본값인 null로 자동 초기화 된다. 그리고 이 객체 배열은 3개의 객체, 정확히는 객체의 주소,를 저장할 수 있다.

```
Tv[] tvArr = new Tv[3]; // 길이가 3인 Tv타입의 참조변수 배열
```



위의 그림에서 알 수 있듯이 객체 배열을 생성하는 것은, 그저 객체를 다루기 위한 참조 변수들이 만들어진 것일 뿐, 아직 객체가 저장되지 않았다. 객체를 생성해서 객체 배열의 각 요소에 저장하는 것을 잊으면 안 된다. 지금은 이런 실수를 안 할 것 같지만, 객체 배열에서 제일 많이 받는 질문이 객체 배열만 생성해 놓고 ‘분명히 객체를 생성했는데, 에러가 발생한다.’는 것이다.

```
Tv[] tvArr = new Tv[3]; // 참조변수 배열(객체 배열)을 생성
```

```
// 객체를 생성해서 배열의 각 요소에 저장  
tvArr[0] = new Tv();  
tvArr[1] = new Tv();  
tvArr[2] = new Tv();
```

배열의 초기화 블럭을 사용하면, 다음과 같이 한 줄로 간단히 할 수 있다.

```
Tv[] tvArr = { new Tv(), new Tv(), new Tv() };
```

다뤄야 할 객체의 수가 많을 때는 for문을 사용하면 된다.

```
Tv[] tvArr = new Tv[100];
```

```
for(int i=0;i<tvArr.length;i++) {  
    tvArr[i] = new Tv();  
}
```

모든 배열이 그렇듯이 객체 배열도 같은 타입의 객체만 저장할 수 있다. 그러면, 여러 종류의 객체를 하나의 배열에 저장할 수 있는 방법은 없을까? 다음 장에 나오는 ‘다형성 (polymorphism)’을 배우고 나면, 하나의 배열로 여러 종류의 객체를 다룰 수 있게 된다.

▼ 예제 6-4/ch6/TvTest4.java

```
class TvTest4 {
    public static void main(String args[]) {
        Tv[] tvArr = new Tv[3]; // 길이가 3인 Tv객체 배열

        // Tv객체를 생성해서 Tv객체 배열의 각 요소에 저장
        for(int i=0; i < tvArr.length;i++) {
            tvArr[i] = new Tv();
            tvArr[i].channel = i+10; // tvArr[i]의 channel에 i+10을 저장
        }

        for(int i=0; i < tvArr.length;i++) {
            tvArr[i].channelUp(); // tvArr[i]의 메서드를 호출. 채널이 1증가
            System.out.printf("tvArr[%d].channel=%d%n",i,
                               tvArr[i].channel);
        }
    } // main의 끝
}

class Tv {
    String color;           // 색상
    boolean power;          // 전원상태(on/off)
    int channel;            // 채널

    void power() { power = !power; }
    void channelUp() { ++channel; }
    void channelDown() { --channel; }
}
```

▼ 실행결과

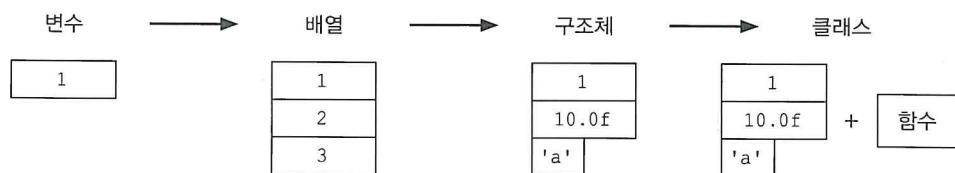
tvArr[0].channel=11
tvArr[1].channel=12
tvArr[2].channel=13

2.6 클래스의 또 다른 정의

클래스는 ‘객체를 생성하기 위한 틀’이며 ‘클래스는 속성과 기능으로 정의되어있다.’고 했다. 이것은 객체지향이론의 관점에서 내린 정의이고, 이번엔 프로그래밍적인 관점에서 클래스의 정의와 의미를 살펴보도록 하자.

1. 클래스 – 데이터와 함수의 결합

프로그래밍언어에서 데이터 처리를 위한 데이터 저장형태의 발전과정은 다음과 같다.



▲ 그림6-3 데이터 저장개념의 발전과정

1. **변수** 하나의 데이터를 저장할 수 있는 공간
2. **배열** 같은 종류의 여러 데이터를 하나의 집합으로 저장할 수 있는 공간
3. **구조체** 서로 관련된 여러 데이터를 종류에 관계없이 하나의 집합으로 저장할 수 있는 공간
4. **클래스** 데이터와 함수의 결합(구조체 + 함수)

하나의 데이터를 저장하기 위해 변수를, 그리고 같은 종류의 데이터를 보다 효율적으로 다루기 위해서 배열이라는 개념을 도입했으며, 후에는 구조체(structure)가 등장하여 자료형의 종류에 상관없이 서로 관계가 깊은 변수들을 하나로 묶어서 다룰 수 있도록 했다.

그동안 데이터와 함수가 서로 관계가 없는 것처럼 데이터는 데이터끼리, 함수는 함수끼리 따로 다루어져 왔지만, 사실 함수는 주로 데이터를 가지고 작업을 하기 때문에 많은 경우에 있어서 데이터와 함수는 관계가 깊다.

그래서 자바와 같은 객체지향언어에서는 변수(데이터)와 함수를 하나의 클래스에 정의하여 서로 관계가 깊은 변수와 함수들을 함께 다룰 수 있게 했다.

서로 관련된 변수들을 정의하고 이들에 대한 작업을 수행하는 함수들을 함께 정의한 것이 바로 클래스이다. C언어에서는 문자열을 문자의 배열로 다루지만, 자바에서는 String이라는 클래스로 문자열을 다룬다. 문자열을 단순히 문자의 배열로 정의하지 않고 클래스로 정의한 이유는 문자열과 문자열을 다루는데 필요한 함수들을 함께 묶기 위해서이다.

```
public final class String implements java.io.Serializable, Comparable {
    private char[] value;           // 문자열을 저장하기 위한 공간

    public String replace(char oldChar, char newChar) {
        ...
        char[] val = value;       // 같은 클래스 내의 변수를 사용해서 작업을 한다.
        ...
    }
}
```

위 코드는 String클래스의 실제 소스의 일부이다. 클래스 내부에 value라는 문자형 배열이 선언되어 있고, 문자열을 다루는 데 필요한 함수들을 함께 정의해 놓았다. 문자열의 일부를 뽑아내는 함수나 문자열의 길이를 알아내는 함수들은 항상 문자열을 작업대상으로 필요로 하기 때문에 문자열과 깊은 관계에 있으므로 함께 정의하였다.

이렇게 하면 변수와 함수가 서로 유기적으로 연결되어 작업이 간단하고 명료해진다.

2. 클래스 – 사용자정의 타입(user-defined type)

프로그래밍언어에서 제공하는 자료형(primitive type)외에 프로그래머가 서로 관련된 변수들을 묶어서 하나의 타입으로 새로 추가하는 것을 사용자정의 타입(user-defined type)이라고 한다.

다른 프로그래밍언어에서도 사용자정의 타입을 정의할 수 있는 방법을 제공하고 있으며 자바와 같은 객체지향언어에서는 클래스가 곧 사용자 정의 타입이다. 기본형의 개수는 8

개로 정해져 있지만 참조형의 개수가 정해져 있지 않은 이유는 이처럼 프로그래머가 새로운 타입을 추가할 수 있기 때문이다.

```
int hour;           // 시간을 표현하기 위한 변수
int minute;        // 분을 표현하기 위한 변수
float second;      // 초를 표현하기 위한 변수, 1/100초까지 표현하기 위해 float로 했다.
```

시간을 표현하기 위해서 위와 같이 3개의 변수를 선언하였다. 만일 3개의 시간을 다뤄야 한다면 다음과 같이 해야 할 것이다.

```
int     hour1, hour2, hour3;
int     minutel, minute2, minute3;
float   second1, second2, second3;
```

이처럼 다뤄야 하는 시간의 개수가 늘어날 때마다 시, 분, 초를 위한 변수를 추가해줘야 하는데 데이터의 개수가 많으면 이런 식으로는 곤란하다.

```
int[]   hour    = new int[3];
int[]   minute  = new int[3];
float[] second  = new float[3];
```

위와 같이 배열로 처리하면 다뤄야 하는 시간 데이터의 개수가 늘어나더라도 배열의 크기만 변경해주면 되므로, 변수를 매번 새로 선언해줘야 하는 불편함과 복잡함은 없어졌다. 그러나 하나의 시간을 구성하는 시, 분, 초가 서로 분리되어 있기 때문에 프로그램 수행과정에서 시, 분, 초가 따로 뒤섞여서 올바르지 않은 데이터가 될 가능성이 있다. 이런 경우 시, 분, 초를 하나로 묶는 사용자정의 타입, 즉 클래스를 정의하여 사용해야한다.

```
class Time {
    int    hour;
    int    minute;
    float second;
}
```

위의 코드는 시, 분, 초를 저장하기 위한 세 변수를 멤버변수로 갖는 Time클래스를 정의한 것이다. 이제 새로 작성된 사용자정의 타입인 Time클래스를 사용해서 코드를 변경하면 아래와 같다.

비객체지향적 코드	객체지향적 코드
<pre>int hour1, hour2, hour3; int minutel, minute2, minute3; float second1, second2, second3;</pre>	<pre>Time t1 = new Time(); Time t2 = new Time(); Time t3 = new Time();</pre>
<pre>int[] hour = new int[3]; int[] minute = new int[3]; float[] second = new float[3];</pre>	<pre>Time[] t = new Time[3]; t[0] = new Time(); t[1] = new Time(); t[2] = new Time();</pre>

▲ 표6-2 비객체지향적 코드와 객체지향적 코드의 비교

이제 시, 분, 초가 하나의 단위로 묶여서 다루어지기 때문에 다른 시간 데이터와 섞이는 일은 없겠지만, 시간 데이터에는 다음과 같은 추가적인 제약조건이 있다.

1. 시, 분, 초는 모두 0보다 크거나 같아야 한다.
2. 시의 범위는 0~23, 분과 초의 범위는 0~59 이다.

이러한 조건들이 모두 프로그램 코드에 반영될 때, 보다 정확한 데이터를 유지할 수 있을 것이다. 객체지향언어가 아닌 언어에서는 이러한 추가적인 조건들을 반영하기가 어렵다.

그러나 객체지향언어에서는 제어자와 메서드를 이용해서 이러한 조건들을 코드에 쉽게 반영할 수 있다.

아직 제어자에 대해서 배우지는 않았지만, 위의 조건들을 반영하여 Time클래스를 작성해 보았다. 가볍게 참고만 하기 바란다.

```
public class Time {
    private int hour;
    private int minute;
    private float second;

    public int getHour() { return hour; }
    public int getMinute() { return minute; }
    public float getSecond() { return second; }

    public void setHour(int h) {
        if (h < 0 || h > 23) return;
        hour = h;
    }

    public void setMinute(int m) {
        if (m < 0 || m > 59) return;
        minute = m;
    }

    public void setSecond(float s) {
        if (s < 0.0f || s > 59.99f) return;
        second = s;
    }
}
```

제어자를 이용해서 변수의 값을 직접 변경하지 못하도록 하고 대신 메서드를 통해서 값을 변경하도록 작성하였다. 값을 변경할 때 지정된 값의 유효성을 조건문으로 점검한 다음에 유효한 값일 경우에만 변경한다.

이 외에도 시간의 차를 구하는 메서드와 같이 시간과 관련된 메서드를 추가로 정의하여 Time클래스를 향상시켜 보는 것도 좋은 프로그래밍 공부거리가 될 것이다.

3. 변수와 메서드

3.1 선언위치에 따른 변수의 종류

변수는 클래스변수, 인스턴스변수, 지역변수 모두 세 종류가 있다. 변수의 종류를 결정짓는 중요한 요소는 '변수의 선언된 위치'이므로 변수의 종류를 파악하기 위해서는 변수가 어느 영역에 선언되었는지를 확인하는 것이 중요하다. 멤버변수를 제외한 나머지 변수들은 모두 지역변수이며, 멤버변수 중 static이 붙은 것은 클래스변수, 붙지 않은 것은 인스턴스변수이다.

아래의 그림에는 모두 3개의 int형 변수가 선언되어 있는데, iv와 cv는 클래스 영역에 선언되어있으므로 멤버변수이다. 그 중 cv는 키워드 static과 함께 선언되어 있으므로 클래스 변수이며, iv는 인스턴스변수이다. 그리고 lv는 메서드인 method()의 내부, 즉 '메서드 영역'에 선언되어 있으므로 지역변수이다.

```
class Variables
{
    int iv;           // 인스턴스변수
    static int cv;   // 클래스변수(static변수, 공유변수)
}

void method()
{
    int lv = 0;     // 지역변수
}
```

변수의 종류	선언위치	생성시기
클래스변수 (class variable)	클래스 영역	클래스가 메모리에 올라갈 때
인스턴스변수 (instance variable)		인스턴스가 생성되었을 때
지역변수 (local variable)	클래스 영역 이외의 영역 (메서드, 생성자, 초기화 블럭 내부)	변수 선언문이 수행되었을 때

▲ 표 6-3 변수의 종류와 특징

1. 인스턴스변수(instance variable)

클래스 영역에 선언되며, 클래스의 인스턴스를 생성할 때 만들어진다. 그렇기 때문에 인스턴스 변수의 값을 읽어 오거나 저장하기 위해서는 먼저 인스턴스를 생성해야한다.

인스턴스는 독립적인 저장공간을 가지므로 서로 다른 값을 가질 수 있다. 인스턴스마다 고유한 상태를 유지해야하는 속성의 경우, 인스턴스변수로 선언한다.

2. 클래스변수(class variable)

클래스 변수를 선언하는 방법은 인스턴스변수 앞에 static을 붙이기만 하면 된다. 인스턴스마다 독립적인 저장공간을 갖는 인스턴스변수와는 달리, 클래스변수는 모든 인스턴스

가 공통된 저장공간(변수)을 공유하게 된다. 한 클래스의 모든 인스턴스들이 공통적인 값을 유지해야하는 속성의 경우, 클래스변수로 선언해야 한다.

클래스 변수는 인스턴스변수와 달리 인스턴스를 생성하지 않고도 언제라도 바로 사용할 수 있다는 특징이 있으며, ‘클래스이름.클래스변수’와 같은 형식으로 사용한다. 예를 들어 Variables클래스의 클래스변수 cv를 사용하려면 ‘Variables.cv’와 같이 하면 된다.

클래스가 메모리에 ‘로딩/loading)’될 때 생성되어 프로그램이 종료될 때 까지 유지되며, public을 앞에 붙이면 같은 프로그램 내에서 어디서나 접근할 수 있는 ‘전역변수(global variable)’의 성격을 갖는다.

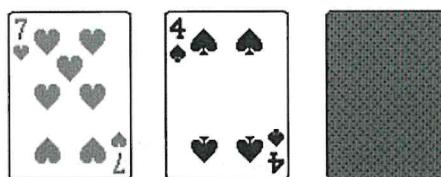
| 참고 | 참조변수의 선언이나 객체의 생성과 같이 클래스의 정보가 필요할 때, 클래스는 메모리에 로딩된다.

3. 지역변수(local variable)

메서드 내에 선언되어 메서드 내에서만 사용 가능하며, 메서드가 종료되면 소멸되어 사용할 수 없게 된다. for문 또는 while문의 블럭 내에 선언된 지역변수는, 지역변수가 선언된 블럭{} 내에서만 사용 가능하며, 블럭{}을 벗어나면 소멸되어 사용할 수 없게 된다. 우리가 6장 이전에 선언한 변수들은 모두 지역변수이다.

3.2 클래스변수와 인스턴스변수

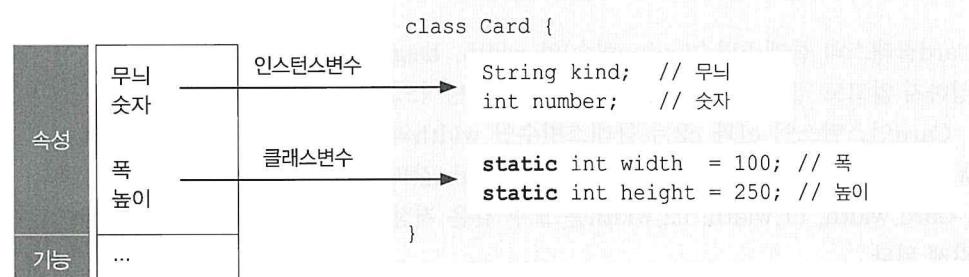
클래스변수와 인스턴스변수의 차이를 이해하기 위한 예로 카드 게임에 사용되는 카드를 클래스로 정의해보자.



▲ 그림6-4 게임카드

카드 클래스를 작성하기 위해서는 먼저 카드를 분석해서 속성과 기능을 알아 내야한다. 속성으로는 카드의 무늬, 숫자, 폭, 높이 정도를 생각할 수 있을 것이다.

이 중에서 어떤 속성을 클래스 변수로 선언할 것이며, 또 어떤 속성들을 인스턴스변수로 선언할 것인지 한번 생각해보자.



각 Card인스턴스는 자신만의 무늬(kind)와 숫자(number)를 유지하고 있어야 하므로 이들을 인스턴스변수로 선언하였고, 각 카드의 폭(width)과 높이(height)는 모든 인스턴스가 공통적으로 같은 값을 유지해야하므로 클래스변수로 선언하였다.

카드의 폭을 변경해야할 필요가 있을 경우, 모든 카드의 width값을 변경하지 않고 한 카드의 width값만 변경해도 모든 카드의 width값이 변경되는 셈이다.

▼ 예제 6-5/ch6/CardTest.java

```
class CardTest{
    public static void main(String args[]) {
        System.out.println("Card.width = " + Card.width);
        System.out.println("Card.height = " + Card.height);

        Card c1 = new Card();
        c1.kind = "Heart";
        c1.number = 7;
    }

    Card c2 = new Card();
    c2.kind = "Spade";
    c2.number = 4;
}

System.out.println("c1은 " + c1.kind + ", " + c1.number
    + "이며, 크기는 (" + c1.width + ", " + c1.height + ")" );
System.out.println("c2는 " + c2.kind + ", " + c2.number
    + "이며, 크기는 (" + c2.width + ", " + c2.height + ")" );
System.out.println("c1의 width와 height를 각각 50, 80으로 변경합니다.");
c1.width = 50;
c1.height = 80;

System.out.println("c1은 " + c1.kind + ", " + c1.number
    + "이며, 크기는 (" + c1.width + ", " + c1.height + ")" );
System.out.println("c2는 " + c2.kind + ", " + c2.number
    + "이며, 크기는 (" + c2.width + ", " + c2.height + ")" );
}
```

▼ 실행결과

```
Card.width = 100
Card.height = 250
c1은 Heart, 70이며, 크기는 (100, 250)
c2는 Spade, 40이며, 크기는 (100, 250)
c1의 width와 height를 각각 50, 80으로 변경합니다.
c1은 Heart, 70이며, 크기는 (50, 80)
c2는 Spade, 40이며, 크기는 (50, 80)
```

| 플래시동영상 | MemberVar.exe는 예제6-5의 실행과정을 설명과 함께 보여준다.

Card클래스의 클래스변수(static변수)인 width, height는 Card클래스의 인스턴스를 생성하지 않고도 '클래스이름.클래스변수'와 같은 방식으로 사용할 수 있다.

Card인스턴스인 c1과 c2는 클래스변수인 width와 height를 공유하기 때문에, c1의 width와 height를 변경하면 c2의 width와 height값도 바뀐 것과 같은 결과를 얻는다.

Card.width, c1.width, c2.width는 모두 같은 저장공간을 참조하므로 항상 같은 값을 갖게 된다.

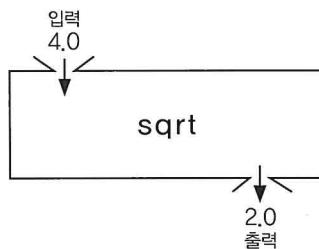
클래스변수를 사용할 때는 Card.width와 같이 ‘클래스이름.클래스변수’의 형태로 하는 것이 좋다. 참조변수 c1, c2를 통해서도 클래스변수를 사용할 수 있지만 이렇게 하면 클래스 변수를 인스턴스변수로 오해하기 쉽기 때문이다.

인스턴스변수는 인스턴스가 생성될 때마다 생성되므로 인스턴스마다 각기 다른 값을 유지할 수 있지만, 클래스 변수는 모든 인스턴스가 하나의 저장공간을 공유하므로, 항상 공통된 값을 갖는다.

3.3 메서드

‘메서드(method)’는 특정 작업을 수행하는 일련의 문장들을 하나로 묶은 것이다. 기본적으로 수학의 함수와 유사하며, 어떤 값을 입력하면 이 값으로 작업을 수행해서 결과를 반환한다. 예를 들어 제곱근을 구하는 메서드 ‘Math.sqrt()’는 4.0을 입력하면, 2.0을 결과로 반환한다.

! 참고 ! 수학의 함수와 달리 메서드는 입력값 또는 출력값(결과값)이 없을 수도 있으며, 심지어는 입력값과 출력값이 모두 없을 수도 있다.



그저 메서드가 작업을 수행하는데 필요한 값만 넣고 원하는 결과를 얻으면 될 뿐, 이 메서드가 내부적으로 어떤 과정을 거쳐 결과를 만들어내는지 전혀 몰라도 된다. 즉, 메서드에 넣을 값(입력)과 반환하는 결과(출력)만 알면 되는 것이다. 그래서 메서드를 내부가 보이지 않는 ‘블랙박스(black box)’라고도 한다.

`sqrt()` 외에도 지금까지 빈번히 사용해온 `println()`이나 `random()`과 같은 메서드들 역시 내부적으로 어떻게 동작하는지 몰라도 사용하는데 아무런 어려움이 없었다.

메서드를 사용하는 이유

메서드를 통해서 얻는 이점은 여러 가지가 있지만 그중에서 대표적인 세 가지를 소개하려 한다. 아직은 메서드를 배우기 전이므로 여기서 나열하는 장점들이 잘 와닿지 않을 수도 있지만, 이 장점들을 염두에 두고 있으면 메서드를 더 깊게 이해하는데 도움이 될 것이다.

1. 높은 재사용성(reusability)

이미 Java API에서 제공하는 메서드들을 사용하면서 경험한 것처럼 한번 만들어 놓은 메서드는 몇 번이고 호출할 수 있으며, 다른 프로그램에서도 사용이 가능하다.

2. 중복된 코드의 제거

프로그램을 작성하다보면, 같은 내용의 문장들이 여러 곳에 반복해서 나타나곤 한다. 이렇게 반복되는 문장들을 끌어서 하나의 메서드로 작성해 놓으면, 반복되는 문장들 대신 메서드를 호출하는 한 문장으로 대체할 수 있다. 그러면, 전체 소스 코드의 길이도 짧아지고, 변경사항이 발생했을 때 수정해야 할 코드의 양이 줄어들어 오류가 발생할 가능성도 함께 줄어든다. 아래의 코드는 예제5-10을 약간 변경한 것인데, 배열을 출력하는 같은 내용의 문장이 두 번 반복된다.

```
public static void main(String args[]) {
    ...
    for (int i = 0; i < 10; i++)
        numArr[i] = (int) (Math.random() * 10);

    for (int i = 0; i < 10; i++)
        System.out.printf("%d", numArr[i]);
    System.out.println();
    ...
    ...중간 생략...
    for (int i = 0; i < 10; i++)
        System.out.printf("%d", numArr[i]);
    System.out.println();
}
```

같은 내용의 코드가 반복됨.
변경할 때도 두 곳을 모두 수정해야 함.

이처럼 같은 내용의 문장들이 반복되면, 소스코드도 길어지고 해당 문장에서 변경사항이 발생했을 때 여러 곳을 고쳐야 하므로 일도 많아지고 오류를 만들어낼 가능성도 높다.

아래와 같이 printArr이라는 메서드를 만들어서 이전의 코드에 적용하면, 다음과 같다.

```
static void printArr(int[] numArr) {
    for (int i = 0; i < 10; i++)
        System.out.printf("%d", numArr[i]);
    System.out.println();
}

public static void main(String args[]) {
    ...
    for (int i = 0; i < 10; i++)
        numArr[i] = (int) (Math.random() * 10);

    printArr(numArr); // 배열을 출력
    ...
    ...중간 생략...
    printArr(numArr); // 배열을 출력
}
```

이제 여기만 변경하면 된다.

반복되는 코드 대신 메서드를 호출

이처럼 반복적으로 나타나는 문장을 메서드로 만들어서 사용하면 코드의 중복이 제거되고, 변경사항이 발생했을 때 이 메서드만 수정하면 되므로 관리도 쉽고 오류의 발생 가능성도 낮아진다.

메서드의 장점을 글로만 간단히 설명하면 와 닿지 않을 것 같아서 간략한 예를 들었는데, 아직 메서드에 대해 배우지 않아서 이해가지 않는 부분이 있을 것이다. 지금은 메서드의 장점을 느낄 수 있는 정도만 이해하자. 곧 자세히 배우게 될 것이다.

3. 프로그램의 구조화

지금까지는 main메서드 안에 모든 문장을 넣는 식으로 프로그램을 작성해왔다. 길어야 100줄 정도 밖에 안 되는 작은 프로그램을 작성할 때는 이렇게 해도 별 문제가 없지만, 몇 천줄, 몇 만 줄이 넘는 프로그램도 이런 식으로 작성할 수는 없다. 큰 규모의 프로그램에서는 문장들을 작업단위로 나눠서 여러 개의 메서드에 담아 프로그램의 구조를 단순화시키는 것이 필수적이다. 예를 들어서 예제5-10를 작업단위로 나누어서 메서드로 만들면, main메서드가 아래와 같이 간단해 진다.

```
public static void main(String args[]) {
    int[] numArr = new int[10];

    initArr(numArr); // 1. 배열을 초기화
    printArr(numArr); // 2. 배열을 출력
    sortArr(numArr); // 3. 배열을 정렬
    printArr(numArr); // 4. 배열을 출력
}
```

이처럼 main메서드는 프로그램의 전체 흐름이 한눈에 들어올 정도로 단순하게 구조화하는 것이 좋다. 그래야 나중에 프로그램에 문제가 발생해도 해당 부분을 쉽게 찾아서 해결할 수 있다.

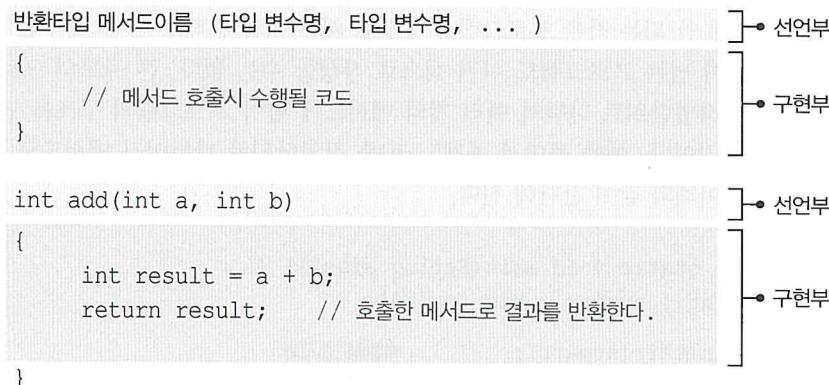
처음에 프로그램을 설계할 때 내용이 없는 메서드를 작업단위로 만들어 놓고, 하나씩 완성해가는 것도 프로그램을 구조화하는 좋은 방법이다. 아래의 코드는 성적처리 프로그램을 설계한 것인데, 처음에 showMenu메서드를 호출해서 메뉴를 보여주고 선택한 메뉴에 따라 각기 다른 작업을 하도록 하였다.

```
static int showMenu() { /* 나중에 내용을 완성한다. */ }
static void inputRecord() { /* 나중에 내용을 완성한다. */ }
static void changeRecord() { /* 나중에 내용을 완성한다. */ }
static void deleteRecord() { /* 나중에 내용을 완성한다. */ }
static void searchRecord() { /* 나중에 내용을 완성한다. */ }
static void showRecordList() { /* 나중에 내용을 완성한다. */ }

public static void main(String args[]) {
    ...
    switch(showMenu()) {
        case 1: inputRecord(); break; // 데이터를 입력받는 메서드
        case 2: changeRecord(); break; // 데이터를 변경하는 메서드
        case 3: deleteRecord(); break; // 데이터를 삭제하는 메서드
        case 4: searchRecord(); break; // 데이터를 검색하는 메서드
        default: showRecordList(); // 데이터의 목록을 보여주는 메서드
    }
}
```

3.4 메서드의 선언과 구현

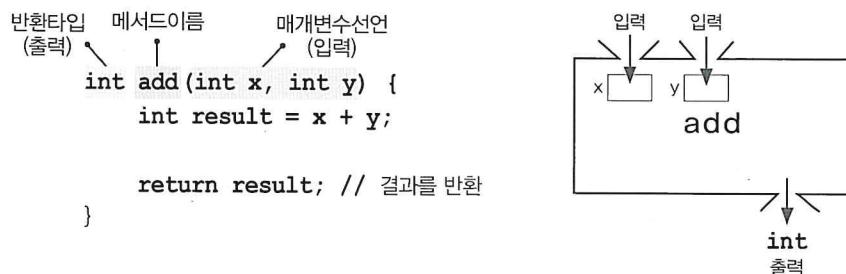
메서드는 크게 두 부분, ‘선언부(header, 머리)’와 ‘구현부(body, 몸통)’로 이루어져 있다. 메서드를 정의한다는 것은 선언부와 구현부를 작성하는 것을 뜻하며 다음과 같은 형식으로 메서드를 정의한다.



메서드 선언부(method declaration, method header)

메서드 선언부는 ‘메서드의 이름’과 ‘매개변수 선언’, 그리고 ‘반환타입’으로 구성되어 있으며, 메서드가 작업을 수행하기 위해 어떤 값을 필요로 하고 작업의 결과로 어떤 타입의 값을 반환하는지에 대한 정보를 제공한다.

예를 들어 아래에 정의된 메서드 add는 두 개의 정수를 입력받아서, 두 값을 더한 결과(int 타입의 값)를 반환한다.



메서드의 선언부는 후에 변경사항이 발생하지 않도록 신중히 작성해야 한다. 메서드의 선언부를 변경하게 되면, 그 메서드가 호출되는 모든 곳도 같이 변경해야 하기 때문이다.

매개변수 선언(parameter declaration)

매개변수는 메서드가 작업을 수행하는데 필요한 값을(입력)을 제공받기 위한 것이며, 필요한 값의 개수만큼 변수를 선언하며 각 변수 간의 구분은 쉼표','를 사용한다. 한 가지 주의할 점은 일반적인 변수선언과 달리 두 변수의 타입이 같아도 변수의 타입을 생략할 수 없다는 것이다.

```
int add(int x, int y) { ... } // OK.
int add(int x, y) { ... } // 에러. 매개변수 y의 타입이 없다.
```

선언할 수 있는 매개변수의 개수는 거의 제한이 없지만, 만일 입력해야 할 값의 개수가 많은 경우에는 배열이나 참조변수를 사용하면 된다. 만일 값을 전혀 입력받을 필요가 없다면 괄호() 안에 아무 것도 적지 않는다.

| 참고 | 매개변수도 메서드 내에 선언된 것으로 간주되므로 '지역변수(local variable)'이다.

메서드의 이름(method name)

메서드의 이름도 앞서 배운 변수의 명명규칙대로 작성하면 된다. 메서드는 특정 작업을 수행하므로 메서드의 이름은 'add'처럼 동사인 경우가 많으며, 이름만으로도 메서드의 기능을 쉽게 알 수 있도록 함축적이면서도 의미있는 이름을 짓도록 노력해야 한다.

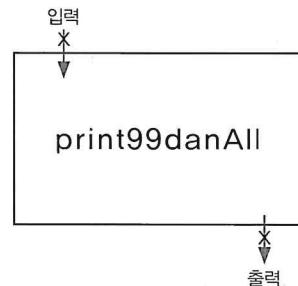
반환타입(return type)

메서드의 작업수행 결과(출력)인 '반환값(return value)'의 타입을 적는다. 반환값이 없는 경우 반환타입으로 'void'를 적어야한다.

아래에 정의된 메서드 'print99danAll()'은 구구단 전체를 출력하는데, 작업을 수행하는데 필요한 값도, 작업수행의 결과인 반환값도 없다. 그래서 반환타입이 'void'이다.

| 참고 | 'void'는 '아무 것도 없음'을 의미한다.

```
void print99danAll() {
    for(int i=1;i<=9;i++) {
        for(int j=2;j<=9;j++) {
            System.out.print(j+"*" + i + " = " + (j*i) + " ");
        }
        System.out.println();
    }
}
```



메서드의 구현부(method body, 메서드 몸통)

메서드의 선언부 다음에 오는 괄호{}를 '메서드의 구현부'라고 하는데, 여기에 메서드를 호출했을 때 수행될 문장들을 넣는다. 우리가 그동안 작성해온 문장들은 모두 main 메서드의 구현부{}에 속한 것들이었으므로 지금까지 하던 대로만 하면 된다.

return문

메서드의 반환타입이 'void'가 아닌 경우, 구현부{} 안에 'return 반환값;'이 반드시 포함되어 있어야 한다. 이 문장은 작업을 수행한 결과인 반환값을 호출한 메서드로 전달하는데, 이 값의 타입은 반환타입과 일치하거나 적어도 자동 형변환이 가능한 것이어야 한다.

여러 개의 변수를 선언할 수 있는 매개변수와 달리 return문은 단 하나의 값만 반환할 수 있는데, 메서드로의 입력(매개변수)은 여러 개일 수 있어도 출력(반환값)은 최대 하나만 허용하는 것이다.

```

    → int add(int x, int y)
    {
        int result = x + y;
        return result; // 작업 결과(반환값)를 반환한다.
    }

```

타입이 일치해야 한다.

위의 코드에서 ‘return result;’는 변수 result에 저장된 값을 호출한 메서드로 반환한다. 변수 result의 타입이 int이므로 메서드 add의 반환타입이 일치하는 것을 알 수 있다.

지역변수(local variable)

메서드 내에 선언된 변수들은 그 메서드 내에서만 사용할 수 있으므로 서로 다른 메서드라면 같은 이름의 변수를 선언해도 된다. 이처럼 메서드 내에 선언된 변수를 ‘지역변수(local variable)’라고 한다.

| 참고 | 매개변수도 메서드 내에 선언된 것으로 간주되므로 지역변수이다.

```

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

int multiply(int x, int y) {
    int result = x * y;
    return result;
}

```

위에 정의된 메서드 add와 multiply에 각기 선언된 변수, x, y, result는 이름만 같을 뿐 서로 다른 변수이다.

3.5 메서드의 호출

메서드를 정의했어도 호출되지 않으면 아무 일도 일어나지 않는다. 메서드를 호출해야만 구현부{}의 문장들이 수행된다. 메서드를 호출하는 방법은 다음과 같다.

| 참고 | main메서드는 프로그램 실행 시 OS에 의해 자동적으로 호출된다.

메서드이름(값1, 값2, ...); // 메서드를 호출하는 방법

```

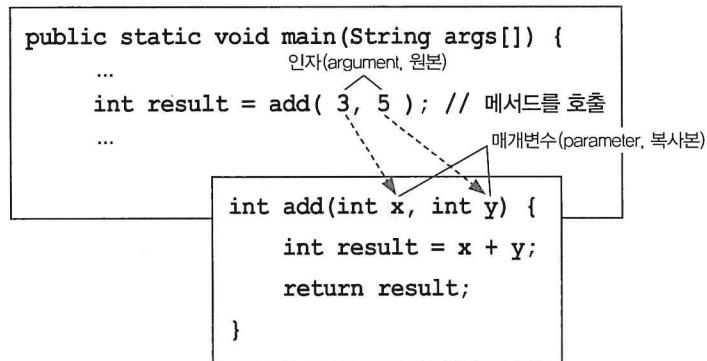
print99danAll();           // void print99danAll()을 호출
int result = add(3, 5);   // int add(int x, int y)를 호출하고, 결과를 result에 저장

```

인자(argument)와 매개변수(parameter)

메서드를 호출할 때 괄호()안에 지정해준 값을 ‘인자(argument)’ 또는 ‘인수’라고 하는데, 인자의 개수와 순서는 호출된 메서드에 선언된 매개변수와 일치해야 한다.

그리고 인자는 메서드가 호출되면서 매개변수에 대입되므로, 인자의 타입은 매개변수의 타입과 일치하거나 자동 형변환이 가능한 것이어야 한다.



만일 아래와 같이 메서드에 선언된 매개변수의 개수보다 많은 값을 괄호()에 넣거나 타입이 다른 값을 넣으면 컴파일러가 에러를 발생시킨다.

```

int result = add(1, 2, 3); // 에러. 메서드에 선언된 매개변수의 개수가 다름
int result = add(1.0, 2.0); // 에러. 메서드에 선언된 매개변수의 타입이 다름

```

반환타입이 void가 아닌 경우, 메서드가 작업을 수행하고 반환한 값을 대입연산자로 변수에 저장하는 것이 보통이지만, 저장하지 않아도 문제가 되지 않는다.

```

int result = add(3, 5); // int add(int x, int y)의 호출결과를 result에 저장
                        // OK. 메서드 add가 반환한 결과를 사용하지 않아도 된다.

```

메서드의 실행흐름

같은 클래스 내의 메서드끼리는 참조변수를 사용하지 않고도 서로 호출이 가능하지만 static메서드는 같은 클래스 내의 인스턴스 메서드를 호출할 수 없다.

다음은 두 개의 값을 매개변수로 받아서 사칙연산을 수행하는 4개의 메서드를 가진 MyMath클래스를 정의한 것이다.

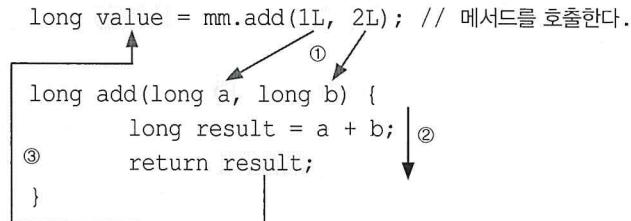
```

class MyMath {
    long add(long a, long b) {
        long result = a+b;
        return result;
        // return a + b; // 위의 두 줄을 이와 같이 한 줄로 간단히 할 수 있다.
    }
    long subtract(long a, long b) { return a - b; }
    long multiply(long a, long b) { return a * b; }
    double divide(double a, double b) { return a / b; }
}

```

MyMath 클래스의 ‘add(long a, long b)’를 호출하기 위해서는 먼저 ‘MyMath mm = new MyMath();’와 같이 해서, MyMath 클래스의 인스턴스를 생성한 다음 참조변수 mm을 통해야한다.

```
MyMath mm = new MyMath(); // 먼저 인스턴스를 생성한다.
```



- ① main 메서드에서 메서드 add를 호출한다. 호출 시 지정한 1L과 2L이 메서드 add의 매개변수 a, b에 각각 복사(대입)된다.
- ② 메서드 add의 괄호[]안에 있는 문장들이 순서대로 수행된다.
- ③ 메서드 add의 모든 문장이 실행되거나 return문을 만나면, 호출한 메서드(main 메서드)로 되돌아와서 이후의 문장들을 실행한다.

메서드가 호출되면 지금까지 실행 중이던 메서드는 실행을 잠시 멈추고 호출된 메서드의 문장들이 실행된다. 호출된 메서드의 작업이 모두 끝나면, 다시 호출한 메서드로 돌아와 이후의 문장들을 실행한다.

위의 그림에서는 편의상 메서드 add의 호출 결과가 바로 value에 저장되는 것처럼 그렸지만, 사실은 호출한 자리를 반환값이 대신하고 대입연산자에 의해 이 값이 변수 value에 저장된다.

```

long value = add(1L, 2L);
→ long value = 3L;

```

add 메서드의 매개변수의 타입이 long이므로 long 또는 long으로 자동 형변환이 가능한 값을 지정해야 한다. 호출 시 매개변수로 지정된 값은 메서드의 매개변수로 복사된다. 위의 코드에서는 1L과 2L의 값이 long 타입의 매개변수 a와 b에 각각 복사된다.

메서드는 호출 시 넘겨받은 값으로 연산을 수행하고 그 결과를 반환하면서 종료된다. 반환된 값은 대입연산자에 의해서 변수 value에 저장된다. 메서드의 결과를 저장하기 위한 변수 value 역시 반환값과 같은 타입이거나 반환값이 자동 형변환되어 저장될 수 있는 타입이어야 한다.

▼ 예제 6-6/ch6/MyMathTest.java

```

class MyMathTest {
    public static void main(String args[]) {
        MyMath mm = new MyMath();
        long result1 = mm.add(5L, 3L);
    }
}

```

```

long result2 = mm.subtract(5L, 3L);
long result3 = mm.multiply(5L, 3L);
double result4 = mm.divide(5L, 3L); •—————
                                         double 대신 long값으로 호출하였다. 이 값은 double로 자동형변환된다.

System.out.println("add(5L, 3L) = " + result1);
System.out.println("subtract(5L, 3L) = " + result2);
System.out.println("multiply(5L, 3L) = " + result3);
System.out.println("divide(5L, 3L) = " + result4);
}

class MyMath {
    long add(long a, long b) {
        long result = a+b;
        return result;
        // return a + b; // 위의 두 줄을 이와 같이 한 줄로 간단히 할 수 있다.
    }

    long subtract(long a, long b) { return a - b; }
    long multiply(long a, long b) { return a * b; }
    double divide(double a, double b) {
        return a / b;
    }
}

```

▼ 실행결과

```

add(5L, 3L) = 8
subtract(5L, 3L) = 2
multiply(5L, 3L) = 15
divide(5L, 3L) = 1.6666666666666667

```

사칙연산을 위한 4개의 메서드가 정의되어 있는 MyMath클래스를 이용한 예제이다. 이 예제를 통해서 클래스에 선언된 메서드를 어떻게 호출하는지 알 수 있을 것이다.

여기서 눈여겨봐야 할 곳은 divide(double a, double b)를 호출하는 부분이다. divide메서드에 선언된 매개변수 타입은 double형인데, 이와 다른 long형의 값인 5L과 3L을 사용해서 호출하는 것이 가능하다.

```

double result4 = mm.divide( 5L , 3L );

double divide(double a, double b) {
    return a / b;
}

```

호출 시에 입력된 값은 메서드의 매개변수에 대입되는 값이므로, long형의 값을 double형 변수에 저장하는 것과 같아서 'double a = 5L;'을 수행 했을 때와 같이 long형의 값인 5L은 double형 값인 5.0으로 자동 형변환되어 divide의 매개변수 a에 저장된다.

그래서, divide메서드에 두 개의 정수값(5L, 3L)을 입력하여 호출하였음에도 불구하고 연산결과가 double형의 값이 된다.

이와 마찬가지로 add(long a, long b)메서드에도 매개변수 a, b에 int형의 값을 넣어 add(5,3)과 같이 호출하는 것이 가능하다.

3.6 return문

return문은 현재 실행중인 메서드를 종료하고 호출한 메서드로 되돌아간다. 지금까지 반환값이 있을 때만 return문을 썼지만, 원래는 반환값의 유무에 관계없이 모든 메서드에는 적어도 하나의 return문이 있어야 한다. 그런데도 반환타입이 void인 경우, return문 없이도 아무런 문제가 없었던 이유는 컴파일러가 메서드의 마지막에 'return;'을 자동적으로 추가해주었기 때문이다.

```
void printGugudan(int dan) {
    for(int i = 1; i <= 9; i++) {
        System.out.printf("%d * %d = %d\n", dan, i, dan * i);
    }
    // return; // 반환 타입이 void이므로 생략 가능. 컴파일러가 자동추가
}
```

그러나 반환타입이 void가 아닌 경우, 즉 반환값이 있는 경우, 반드시 return문이 있어야 한다. return문이 없으면 컴파일 에러(error: missing return statement)가 발생한다.

```
int multiply(int x, int y) {
    int result = x * y;

    return result; // 반환 타입이 void가 아니므로 생략불가
}
```

아래의 코드는 두 값 중에서 큰 값을 반환하는 메서드이다. 이 메서드의 리턴타입이 int이 고 int타입의 값을 반환하는 return문이 있지만, return문이 없다는 에러가 발생한다. 왜냐하면 if문 조건식의 결과에 따라 return문이 실행되지 않을 수도 있기 때문이다.

```
int max(int a, int b) {
    if(a > b)
        return a; // 조건식이 참일 때만 실행된다.
}
```

그래서 이런 경우 다음과 같이 if문의 else블럭에 return문을 추가해서, 항상 결과값이 반환되도록 해야 한다.

```
int max(int a, int b) {
    if(a > b)
        return a; // 조건식이 참일 때 실행된다.
    else
        return b; // 조건식이 거짓일 때 실행된다.
}
```

반환값(return value)

return문의 반환값으로 주로 변수가 오긴 하지만 항상 그런 것은 아니다. 아래 왼쪽의 코드는 오른쪽과 같이 간략히 할 수 있는데, 오른쪽의 코드는 return문의 반환값으로 ‘ $x+y$ ’라는 수식이 적혀있다. 그렇다고 해서 수식이 반환되는 것은 아니고, 이 수식을 계산한 결과가 반환된다.

```
int add(int x, int y) {
    int result = x + y;
    return result;
} ←→ int add(int x, int y) {
        return x + y ;
    }
```

| 참고 | 수학에서처럼, result의 값이 ' $x+y$ '와 같으므로 result 대신 ' $x+y$ '를 쓸 수 있다고 생각하면 이해하기 쉽다.

예를 들어 매개변수 x 와 y 의 값이 각각 3과 5라면, ‘return $x+y$;’는 다음과 같은 계산과정을 거쳐서 반환값은 8이 된다.

```
return x + y;
→ return 3 + 5;
→ return 8;
```

아래의 diff메서드는 두 개의 정수를 받아서 그 차이를 절대값으로 반환한다. 오른쪽 코드 역시 메서드를 반환하는 것이 아니라 메서드 abs를 호출하고, 그 결과를 받아서 반환한다. 메서드 abs의 반환타입이 메서드 diff의 반환타입과 일치하기 때문에 이렇게 하는 것이 가능하다는 것에 주의하자.

```
int diff(int x, int y) {
    int result = abs(x-y);
    return result;
} ←→ int diff(int x, int y) {
        return abs(x-y);
    }
```

간단한 메서드의 경우 if문 대신 조건연산자를 사용하기도 한다. 메서드 abs는 입력받은 정수의 부호를 판단해서 음수일 경우 부호연산자(-)를 사용해서 양수로 반환한다.

```
int abs(int x) {
    if(x>=0) {
        return x;
    } else {
        return -x;
    }
} ←→ int abs(int x) {
        return x>=0 ? x : -x;
    }
```

매개변수의 유효성 검사

메서드의 구현부 {}를 작성할 때, 제일 먼저 해야 하는 일이 매개변수의 값이 적절한 것인지 확인하는 것이다. 메서드를 작성하는 사람은 ‘호출하는 쪽에서 알아서 적절한 값을 넘겨주겠지.’라는 생각을 절대로 가져서는 안 된다. 타입만 맞으면 어떤 값도 매개변수를 통해 넘어올 수 있기 때문에, 가능한 모든 경우의 수에 대해 고민하고 그에 대비한 코드를 작성해야 한다.

아래에 정의된 메서드 divide는 매개변수 x를 y로 나눈 결과를 실수(float타입)로 반환하는데, 0으로 나누는 것은 금지되어 있기 때문에 계산 전에 y의 값이 0인지 확인해야 한다. 그래서 y의 값이 0이면, 나누기를 계산할 수 없으므로 return문을 이용해서 작업을 중단하고 메서드를 빠져나와야 한다. 그렇지 않으면, 나누기를 하는 문장에서 프로그램이 비정상적으로 종료된다.

```
float divide(int x, int y) {  
    // 작업을 하기 전에 나누는 수(y)가 0인지 확인한다.  
    if(y == 0) {  
        System.out.println("0으로 나눌 수 없습니다.");  
        return 0; // 매개변수가 유효하지 않으므로 메서드를 종료한다.  
    }  
  
    return x / (float)y;  
}
```

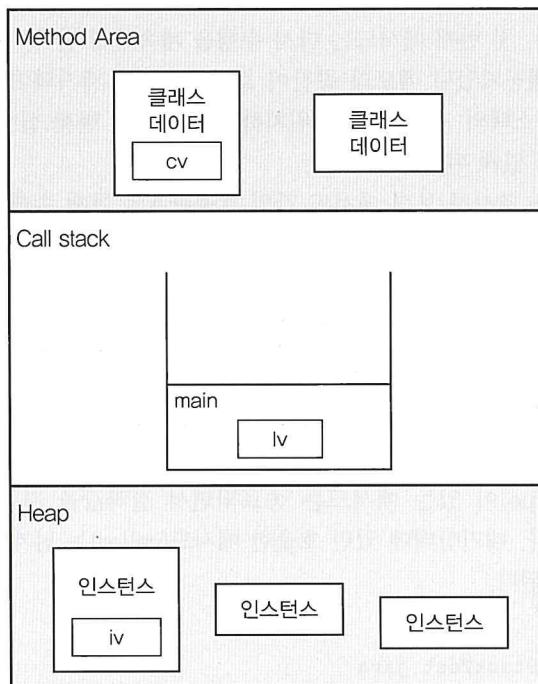
적절하지 않은 값이 매개변수를 통해 넘어온다면 매개변수의 값을 보정하던가, 보정하는 것이 불가능하다면 return문을 사용해서 작업을 중단하고 호출한 메서드로 되돌아가야 한다.

이 책의 많은 예제에서 코드를 단순화하기 위해서 유효성 검사를 생략한 경우가 많지만 여러분들이 메서드를 작성할 때는 매개변수의 유효성 검사하는 코드를 반드시 넣길 바란다. 매개변수의 유효성 검사는 메서드의 작성에 있어서 간과하기 쉬운 중요한 부분이다.

3.7 JVM의 메모리 구조

응용프로그램이 실행되면, JVM은 시스템으로부터 프로그램을 수행하는데 필요한 메모리를 할당받고 JVM은 이 메모리를 용도에 따라 여러 영역으로 나누어 관리한다.

그 중 3가지 주요 영역(method area, call stack, heap)에 대해서 알아보자.



▲ 그림 6-5 JVM의 메모리 구조

| 참고 | cv는 클래스변수, lv는 지역변수, iv는 인스턴스변수를 뜻한다.

1. 메서드 영역(method area)

- 프로그램 실행 중 어떤 클래스가 사용되면, JVM은 해당 클래스의 클래스파일(*.class)을 읽어서 분석하여 클래스에 대한 정보(클래스 데이터)를 이곳에 저장한다. 이 때, 그 클래스의 클래스변수(class variable)도 이 영역에 함께 생성된다.

2. 힙(heap)

- 인스턴스가 생성되는 공간. 프로그램 실행 중 생성되는 인스턴스는 모두 이곳에 생성된다. 즉, 인스턴스변수(instance variable)들이 생성되는 공간이다.

3. 호출스택(call stack 또는 execution stack)

- 호출스택은 메서드의 작업에 필요한 메모리 공간을 제공한다. 메서드가 호출되면, 호출스택에 호출된 메서드를 위한 메모리가 할당되며, 이 메모리는 메서드가 작업을 수행하는 동안 지역변수(매개변수 포함)들과 연산의 중간결과 등을 저장하는데 사용된다. 그리고 메서드가 작업을 마치면 할당되었던 메모리공간은 반환되어 비워진다.

각 메서드를 위한 메모리상의 작업공간은 서로 구별되며, 첫 번째로 호출된 메서드를 위한 작업공간이 호출스택의 맨 밑에 마련되고, 첫 번째 메서드 수행 중에 다른 메서드를 호출하면, 첫 번째 메서드의 바로 위에 두 번째로 호출된 메서드를 위한 공간이 마련된다.

이 때 첫 번째 메서드는 수행을 멈추고, 두 번째 메서드가 수행되기 시작한다. 두 번째로 호출된 메서드가 수행을 마치게 되면, 두 번째 메서드를 위해 제공되었던 호출스택의 메모리공간이 반환되며, 첫 번째 메서드는 다시 수행을 계속하게 된다. 첫 번째 메서드가 수행을 마치면, 역시 제공되었던 메모리 공간이 호출스택에서 제거되며 호출스택은 완전히 비워지게 된다. 호출스택의 제일 상위에 위치하는 메서드가 현재 실행 중인 메서드이며, 나머지는 대기상태에 있게 된다.

따라서, 호출스택을 조사해 보면 메서드 간의 호출관계와 현재 수행중인 메서드가 어느 것인지 알 수 있다. 호출스택의 특징을 정리해보면 다음과 같다.

- 메서드가 호출되면 수행에 필요한 만큼의 메모리를 스택에 할당받는다.
- 메서드가 수행을 마치고나면 사용했던 메모리를 반환하고 스택에서 제거된다.
- 호출스택의 제일 위에 있는 메서드가 현재 실행 중인 메서드이다.
- 아래에 있는 메서드가 바로 위의 메서드를 호출한 메서드이다.

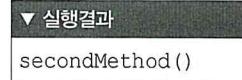
반환타입(return type)이 있는 메서드는 종료되면서 결과값을 자신을 호출한 메서드(caller)에게 반환한다. 대기상태에 있던 호출한 메서드(caller)는 넘겨받은 반환값으로 수행을 계속 진행하게 된다.

▼ 예제 6-7/ch6/CallStackTest.java

```
class CallStackTest {
    public static void main(String[] args) {
        firstMethod(); // static메서드는 객체 생성없이 호출 가능하다.
    }

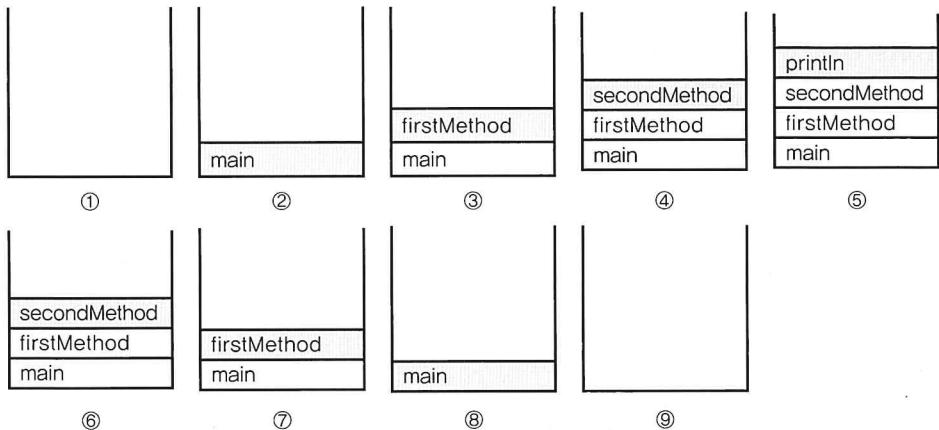
    static void firstMethod() {
        secondMethod();
    }

    static void secondMethod() {
        System.out.println("secondMethod()");
    }
}
```



main()이 firstMethod()를 호출하고 firstMethod()는 secondMethod()를 호출한다. 객체를 생성하지 않고도 메서드를 호출할 수 있으려면, 메서드 앞에 'static'을 붙여야 한다. 잠시 후에 자세히 설명할 것이므로 지금은 이정도만 알아두자.

위의 예제를 실행시켰을 때, 프로그램이 수행되는 동안 호출스택의 변화를 그림과 함께 살펴보도록 하자.



▲ 그림 6-6 예제6-7의 실행 시 호출스택의 변화

- ①~② 위의 예제를 실행시키면, JVM에 의해서 main메서드가 호출됨으로써 프로그램이 시작된다. 이 때, 호출스택에는 main메서드를 위한 메모리공간이 할당되고 main메서드의 코드가 수행되기 시작한다.
- ③ main메서드에서 firstMethod()를 호출한 상태이다. 아직 main메서드가 끝난 것은 아니므로 main메서드는 호출스택에 대기상태로 남아있고 firstMethod()의 수행이 시작된다.
- ④ firstMethod()에서 다시 secondMethod()를 호출했다. firstMethod()는 secondMethod()가 수행을 마칠 때까지 대기상태에 있게 된다. secondMethod()가 수행을 마쳐야 firstMethod()의 나머지 문장들을 수행 할 수 있기 때문이다.
- ⑤ secondMethod()에서 println()을 호출했다. println메서드에 의해 'secondMethod()'가 화면에 출력된다.
- ⑥ println메서드의 수행이 완료되어 호출스택에서 사라지고 자신을 호출한 second Method()로 돌아간다. 대기 중이던 secondMethod()는 println()을 호출한 이후부터 수행을 재개한다.
- ⑦ secondMethod()에 더 이상 수행할 코드가 없으므로 종료되고, 자신을 호출한 firstMethod()로 돌아간다.
- ⑧ firstMethod()에도 더 이상 수행할 코드가 없으므로 종료되고, 자신을 호출한 main메서드로 돌아간다.
- ⑨ main메서드에도 더 이상 수행할 코드가 없으므로 종료되어, 호출스택은 완전히 비워지게 되고 프로그램은 종료된다.

▼ 예제 6-8/ch6/CallStackTest2.java

```
class CallStackTest2 {
    public static void main(String[] args) {
        System.out.println("main(String[] args)이 시작되었음.");
        firstMethod();
        System.out.println("main(String[] args)이 끝났음.");
    }

    static void firstMethod() {
        System.out.println("firstMethod()이 시작되었음.");
        secondMethod();
        System.out.println("firstMethod()이 끝났음.");
    }

    static void secondMethod() {
        System.out.println("secondMethod()이 시작되었음.");
        System.out.println("secondMethod()이 끝났음.");
    }
}
```

▼ 실행결과

```
main(String[] args)이 시작되었음.
firstMethod()이 시작되었음.
secondMethod()이 시작되었음.
secondMethod()이 끝났음.
firstMethod()이 끝났음.
main(String[] args)이 끝났음.
```

예제6-7에 출력문을 추가해서 각 메서드의 시작과 종료의 순서를 확인하는 예제이다. 그림6-6과 함께 다시 한 번 호출과정을 확인해보자.

3.8 기본형 매개변수와 참조형 매개변수

자바에서는 메서드를 호출할 때 매개변수로 지정한 값을 메서드의 매개변수에 복사해서 넘겨준다. 매개변수의 타입이 기본형(primitive type)일 때는 기본형 값이 복사되겠지만, 참조형(reference type)이면 인스턴스의 주소가 복사된다.

메서드의 매개변수를 기본형으로 선언하면 단순히 저장된 값만 얻지만, 참조형으로 선언하면 값이 저장된 곳의 주소를 알 수 있기 때문에 값을 읽어 오는 것은 물론 값을 변경하는 것도 가능하다.

기본형 매개변수 변수의 값을 읽기만 할 수 있다.(read only)

참조형 매개변수 변수의 값을 읽고 변경할 수 있다.(read & write)

▼ 예제 6-9/ch6/PrimitiveParamEx.java

```
class Data { int x; }

class PrimitiveParamEx {
    public static void main(String[] args) {
        Data d = new Data();
        d.x = 10;
        System.out.println("main() : x = " + d.x);

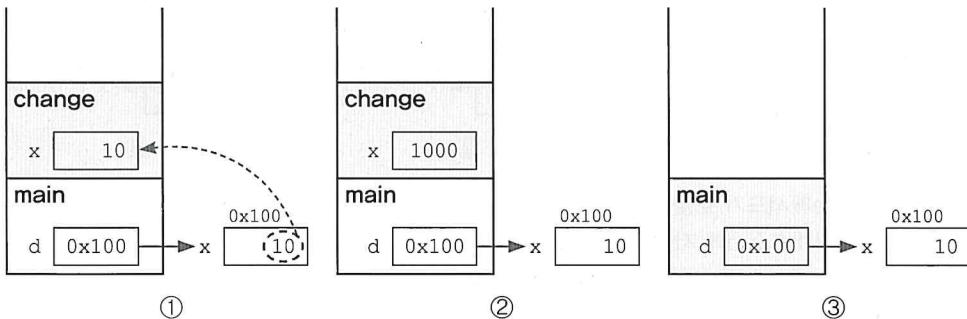
        change(d.x);
        System.out.println("After change(d.x)");
        System.out.println("main() : x = " + d.x);
    }

    static void change(int x) { // 기본형 매개변수
        x = 1000;
        System.out.println("change() : x = " + x);
    }
}
```

▼ 실행결과

```
main() : x = 10
change() : x = 1000
After change(d.x)
main() : x = 10
```

change메서드에서 main메서드로부터 넘겨받은 d.x의 값을 1000으로 변경했는데도 main메서드에서는 d.x의 값이 그대로이다. 왜 이런 결과가 나오는지 단계별로 그림을 그려보면 아래와 같다.



- ① change메서드가 호출되면서 'd.x'가 change메서드의 매개변수 x에 복사됨
- ② change메서드에서 x의 값을 1000으로 변경
- ③ change메서드가 종료되면서 매개변수 x는 스택에서 제거됨

'd.x'의 값이 변경된 것이 아니라, change메서드의 매개변수 x의 값이 변경된 것이다. 즉, 원본이 아닌 복사본이 변경된 것이라 원본에는 아무런 영향을 미치지 못한다. 이처럼 기본형 매개변수는 변수에 저장된 값만 읽을 수만 있을 뿐 변경할 수는 없다.

▼ 예제 6-10/ch6/ReferenceParamEx.java

```
class Data { int x; }

class ReferenceParamEx {
    public static void main(String[] args) {
        Data d = new Data();
        d.x = 10;
        System.out.println("main() : x = " + d.x);

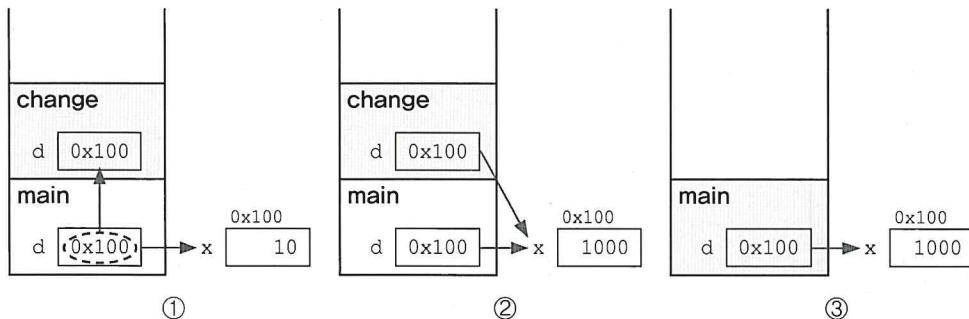
        change(d);
        System.out.println("After change(d)");
        System.out.println("main() : x = " + d.x);
    }

    static void change(Data d) { // 참조형 매개변수
        d.x = 1000;
        System.out.println("change() : x = " + d.x);
    }
}
```

▼ 실행결과
main() : x = 10 change() : x = 1000 After change(d) main() : x = 1000

| 플래시동영상 | ReferenceParam.exe를 보면 예제6-10의 실행과정을 자세히 볼 수 있다.

이전 예제와 달리 change메서드를 호출한 후에 d.x의 값이 변경되었다. change메서드의 매개변수가 참조형이라서 값이 아니라 '값이 저장된 주소'를 change메서드에게 넘겨주었기 때문에 값을 읽어오는 것뿐만 아니라 변경하는 것도 가능하다.



- ① change메서드가 호출되면서 참조변수 d의 값(주소)이 매개변수 d에 복사됨.
이제 매개변수 d에 저장된 주소값으로 x에 접근이 가능
- ② change메서드에서 매개변수 d로 x의 값을 1000으로 변경
- ③ change메서드가 종료되면서 매개변수 d는 스택에서 제거됨

이전 예제와 달리, change메서드의 매개변수를 참조형으로 선언했기 때문에, x의 값이 아닌 주소가 매개변수 d에 복사되었다. 이제 main메서드의 참조변수 d와 change메서드의 참조변수 d는 같은 객체를 가리키게 된다. 그래서 매개변수 d로 x의 값을 읽는 것과 변경하는 것이 모두 가능한 것이다. 이 두 예제를 잘 비교해서 차이를 완전히 이해해야 한다.

▼ 예제 6-11/ch6/ReferenceParamEx2.java

```
class ReferenceParamEx2 {
    public static void main(String[] args)
    {
        int[] x = {10}; // 크기가 1인 배열. x[0] = 10;
        System.out.println("main() : x = " + x[0]);

        change(x);
        System.out.println("After change(x)");
        System.out.println("main() : x = " + x[0]);
    }

    static void change(int[] x) { // 참조형 매개변수
        x[0] = 1000;
        System.out.println("change() : x = " + x[0]);
    }
}
```

▼ 실행결과
main() : x = 10 change() : x = 1000 After change(x) main() : x = 1000

이전의 참조형 매개변수 예제를 Data클래스의 인스턴스 대신 길이가 1인 배열 x를 사용하도록 변경한 것이다. 배열도 객체와 같이 참조변수를 통해 데이터가 저장된 공간에 접근한다는 것을 이미 배웠다. 이전 예제의 Data클래스 타입의 참조변수 d와 같이 변수 x도 int배열타입의 참조변수이기 때문에 같은 결과를 얻는다.

임시적으로 간단히 처리할 때는 별도의 클래스를 선언하는 것보다 이처럼 배열을 이용 할 수도 있다.

▼ 예제 6-12/ch6/ReferenceParamEx3.java

```

class ReferenceParamEx3 {
    public static void main(String[] args) {
        int[] arr = new int[] {3,2,1,6,5,4};

        printArr(arr); // 배열의 모든 요소를 출력
        sortArr(arr); // 배열을 정렬
        printArr(arr); // 정렬후 결과를 출력
        System.out.println("sum="+sumArr(arr)); // 배열의 총합을 출력
    }

    static void printArr(int[] arr) { // 배열의 모든 요소를 출력
        System.out.print("[");
        for(int i : arr) // 향상된 for문
            System.out.print(i+",");
        System.out.println("]");
    }

    static int sumArr(int[] arr) { // 배열의 모든 요소의 합을 반환
        int sum = 0;
        for(int i=0;i<arr.length;i++)
            sum += arr[i];
        return sum;
    }

    static void sortArr(int[] arr) { // 배열을 오름차순으로 정렬
        for(int i=0;i<arr.length-1;i++)
            for(int j=0;j<arr.length-1-i;j++)
                if(arr[j] > arr[j+1]) {
                    int tmp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = tmp;
                }
    } // sortArr(int[] arr)
}

```

▼ 실행결과
[3,2,1,6,5,4]
[1,2,3,4,5,6]
sum=21

메서드로 배열을 다루는 여러 가지 방법을 보여주는 예제이다. 매개변수의 타입이 배열이니까, 참조형 매개변수이다. 그래서 sortArr메서드에서 정렬한 것이 원래의 배열에 영향을 미친다. 그 외에는 따로 설명하지 않아도 충분히 이해가 될 것이다.

▼ 예제 6-13/ch6/ReturnTest.java

```

class ReturnTest {
    public static void main(String[] args) {
        ReturnTest r = new ReturnTest();

        int result = r.add(3,5);
        System.out.println(result);

        int[] result2 = {0}; // 배열을 생성하고 result2[0]의 값을 0으로 초기화
        r.add(3,5,result2); // 배열을 add메서드의 매개변수로 전달
        System.out.println(result2[0]);
    }
}

```

```

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

void add(int a, int b, int[] result) {
    result[0] = a + b; // 매개변수로 넘겨받은 배열에 연산결과를 저장
}
}

```

▼ 실행결과
8
8

이 예제는 반환값이 있는 메서드를 반환값이 없는 메서드로 바꾸는 방법을 보여준다. 앞서 배운 참조형 매개변수를 활용하면 반환값이 없어도 메서드의 실행결과를 얻어 옮을 수 있다.

```

int add(int a, int b) {           void add(int a, int b, int[] result) {
    return a + b;               result[0] = a + b;
}                                }

```

메서드는 단 하나의 값만을 반환할 수 있지만 이것을 응용하면 여러 개의 값을 반환받는 것과 같은 효과를 얻을 수 있다.

3.9 참조형 반환타입

매개변수뿐만 아니라 반환타입도 참조형이 될 수 있다. 반환타입이 참조형이라는 것은 반환하는 값의 타입이 참조형이라는 얘긴데, 모든 참조형 타입의 값은 '객체의 주소'이므로 그저 정수값이 반환되는 것일 뿐 특별할 것이 없다. 일단 예제부터 먼저 살펴보자.

▼ 예제 6-14/ch6/ReferenceReturnEx.java

```

class Data { int x; }

class ReferenceReturnEx {
    public static void main(String[] args) {
        Data d = new Data();
        d.x = 10;

        Data d2 = copy(d);
        System.out.println("d.x =" + d.x);
        System.out.println("d2.x=" + d2.x);
    }

    static Data copy(Data d) {
        Data tmp = new Data();
        tmp.x = d.x;

        return tmp;
    }
}

```

▼ 실행결과
d.x =10
d2.x=10

copy메서드는 새로운 객체를 생성한 다음에, 매개변수로 넘겨받은 객체에 저장된 값을 복사해서 반환한다. 반환하는 값이 Data객체의 주소이므로 반환 타입이 'Data'인 것이다.

```

static Data copy(Data d) {
    Data tmp = new Data(); // 새로운 객체 tmp를 생성한다.
    tmp.x = d.x;           // d.x의 값을 tmp.x에 복사한다.

    return tmp; // 복사한 객체의 주소를 반환한다.
}

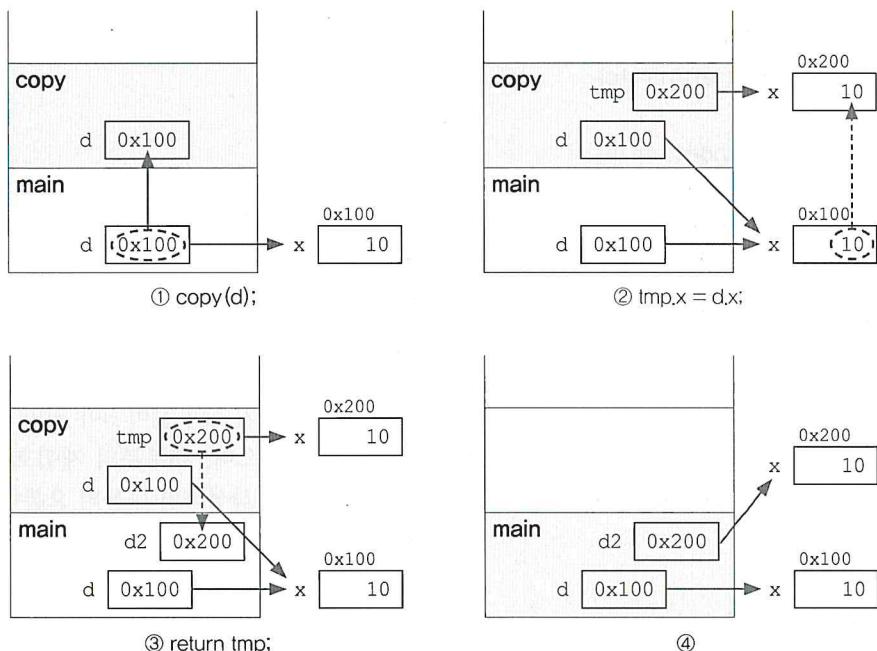
```

이 메서드의 반환타입이 'Data'이므로, 호출결과를 저장하는 변수의 타입 역시 'Data'타입의 참조변수이어야 한다.

```
Data d2 = copy(d); // static Data copy(Data d)
```

copy메서드 내에서 생성한 객체를 main메서드에서 사용할 수 있으면, 이렇게 새로운 객체의 주소를 반환해줘야 한다. 그렇지 않으면, copy메서드가 종료되면서 새로운 객체의 참조가 사라지기 때문에 더 이상 이 객체를 사용할 방법이 없다.

copy메서드가 호출된 직후부터 종료까지의 과정을 단계별로 살펴보면 다음과 같다.



- ① copy메서드를 호출하면서 참조변수 d의 값이 매개변수 d에 복사된다.
- ② 새로운 객체를 생성한 다음, d.x에 저장된 값을 tmp.x에 복사한다.
- ③ copy메서드가 종료되면서 반환한 tmp의 값은 참조변수 d2에 저장된다.
- ④ copy메서드가 종료되어 tmp가 사라졌지만, d2로 새로운 객체를 다룰 수 있다.

“반환타입이 ‘참조형’이라는 것은

메서드가 ‘객체의 주소’를 반환한다는 것을 의미한다.”

3.10 재귀호출(recursive call)

메서드의 내부에서 메서드 자신을 다시 호출하는 것을 ‘재귀호출(recursive call)’이라 하고, 재귀호출을 하는 메서드를 ‘재귀 메서드’라 한다.

```
void method() {
    method(); // 재귀호출. 메서드 자신을 호출한다.
}
```

어떻게 메서드가 자기자신을 호출할 수 있는지 의아하겠지만, 메서드 입장에서는 자기 자신을 호출하는 것과 다른 메서드를 호출하는 것은 차이가 없다. ‘메서드 호출’이라는 것이 그저 특정 위치에 저장되어 있는 명령들을 수행하는 것일 뿐이기 때문이다.

호출된 메서드는 ‘값에 의한 호출(call by value)’을 통해, 원래의 값이 아닌 복사된 값으로 작업하기 때문에 호출한 메서드와 관계없이 독립적인 작업수행이 가능하다.

그런데 위의 코드처럼 오로지 재귀호출뿐이면, 무한히 자기 자신을 호출하기 때문에 무한 반복에 빠지게 된다. 무한반복문이 조건문과 함께 사용되어야하는 것처럼, 재귀호출도 조건문이 필수적으로 따라다닌다.

```
void method(int n) {
    if(n == 0)
        return; // n의 값이 0일 때, 메서드를 종료한다.
    System.out.println(n);

    method(--n); // 재귀호출. method(int n)을 호출
}
```

이 코드는 매개변수 n을 1씩 감소시켜가면서 재귀호출을 하다가 n의 값이 0이 되면 재귀호출을 중단하게 된다. 재귀호출은 반복문과 유사한 점이 많으며, 대부분의 재귀호출은 반복문으로 작성하는 것이 가능하다. 위의 코드를 반복문으로 작성하면 다음의 오른쪽 코드와 같다.

```
void method(int n) {
    if(n==0) return;
    System.out.println(n);
    method(--n); // 재귀호출
}
```



```
void method(int n) {
    while(n!=0) {
        System.out.println(n--);
    }
}
```

반복문은 그저 같은 문장을 반복해서 수행하는 것이지만, 메서드를 호출하는 것은 반복문 보다 몇 가지 과정, 예를 들면 매개변수 복사와 종료 후 복귀할 주소저장 등, 이 추가로 필요하기 때문에 반복문보다 재귀호출의 수행시간이 더 오래 걸린다.

그렇다면 ‘왜? 굳이 반복문대신 재귀호출을 사용할까’. 그 이유는 바로 재귀호출이 주는 논리적 간결함 때문이다. 몇 겹의 반복문과 조건문으로 복잡하게 작성된 코드가 재귀호출로 작성하면 보다 단순한 구조로 바뀔 수도 있다. 아무리 효율적이라도 알아보기 힘들게 작성하는 것보다 다소 비효율적이라도 알아보기 쉽게 작성하는 것이 논리적 오류가 발생할 확률도 줄어들고 나중에 수정하기도 좋다.

어떤 작업을 반복적으로 처리해야 한다면, 먼저 반복문으로 작성해보고 너무 복잡하면 재귀호출로 간단히 할 수 없는지 고민해볼 필요가 있다. 재귀호출은 비효율적이므로 재귀호출에 드는 비용보다 재귀호출의 간결함이 주는 이득이 충분히 큰 경우에만 사용해야 한다는 것도 잊지 말자.

대표적인 재귀호출의 예는 팩토리얼(factorial)을 구하는 것이다. 팩토리얼은 한 숫자가 1이 될 때까지 1씩 감소시켜가면서 계속해서 곱해 나가는 것인데, $n!(n은 양의 정수)$ 과 같이 표현한다. 예를 들면, ' $5! = 5 * 4 * 3 * 2 * 1 = 120$ '이다.

팩토리얼을 수학적 메서드로 표현하면 아래와 같이 표현할 수 있다.

$$f(n) = n * f(n-1), \text{ 단 } f(1) = 1$$

다음 예제는 위의 메서드를 자바로 구현한 것이다.

▼ 예제 6-15/ch6/FactorialTest.java

```
class FactorialTest {
    public static void main(String args[]) {
        int result = factorial(4);
        System.out.println(result);
    }

    static int factorial(int n) {
        int result=0;
        if ( n == 1)
            result = 1;
        else
            result = n * factorial(n-1); // 다시 메서드 자신을 호출한다.
        return result;
    }
}
```

▼ 실행결과
24

| 플래시동영상 | RecursiveCall.exe를 보면 예제6-15의 실행과정을 자세히 볼 수 있다.

위 예제는 팩토리얼을 계산하는 메서드를 구현하고 테스트하는 것이다. factorial메서드가 static메서드이므로 인스턴스를 생성하지 않고 직접 호출할 수 있다. 그리고 main메서드와 같은 클래스에 있기 때문에 static메서드를 호출할 때 클래스이름을 생략하는 것이 가능하다. 그래서 ‘FactorialTest.factorial(4)’ 대신 ‘factorial(4)’와 같이 하였다.

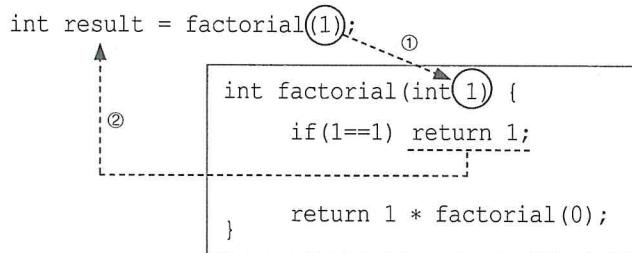
예제6-15는 실행과정을 플래시 동영상으로 보여주기 위해 작성한 것이라 코드가 길어졌는데, 좀 더 간단히 하면 다음과 같이 쓸 수 있다.

```
static int factorial(int n) {
    if(n == 1) return 1;
    return n * factorial(n-1);
}
```

이해하기 어려운 코드는 변수에 직접 값을 대입해보면 알기 쉬워진다. 만일 매개변수 n의 값이 3이라면, n대신 3을 직접 대입해보자. 오른쪽의 코드처럼 된다.

<pre>int factorial(int n) { if(n==1) return 1; return n * factorial(n-1); }</pre>	→	<pre>int factorial(int 3) { if(3==1) return 1; return 3 * factorial(3-1); }</pre>
---	---	---

이 방법을 이용해서 main메서드에서 factorial(1)을 호출했을 때의 실행과정을 살펴보자. 아래의 그림은 매개변수 n 대신 1을 대입한 것이다.



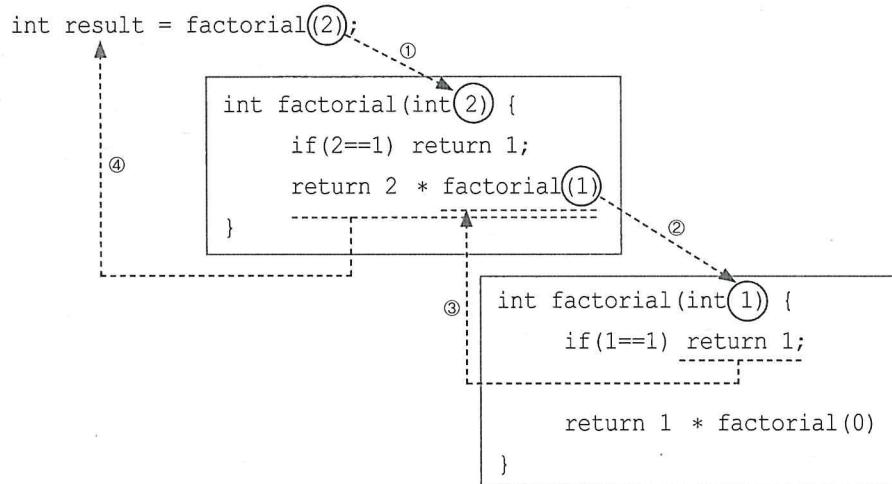
if문의 조건식이 참이 되어 1을 반환하기 때문에, 재귀호출이 일어나기도 전에 factorial메서드는 종료된다. 그리고 result에는 'factorial(1)'의 반환값인 1이 저장된다. 위의 그림에서는 반환값이 바로 result에 저장되는 것처럼 되어 있지만, 정확히는 아래와 같이 반환값이 메서드의 호출을 대신한다.

```
int result = factorial(1);
→ int result = 1;
```

이번엔 'factorial(2)'를 호출했을 때의 실행과정을 살펴보자. 매개변수의 값이 1이 아니므로 조건식이 거짓이 되어 그 다음 문장인 'return 2 * factorial(1);'이 수행되고, 이 식을 계산하는 과정에서 다시 factorial(1)이 호출된다.

- ① factorial(2)를 호출하면서 매개변수 n에 2가 복사된다.
- ② 'return 2 * factorial(1);'을 계산하려면 factorial(1)을 호출한 결과가 필요하다.
그래서 factorial(1)이 호출되고 매개변수 n에 1이 복사된다.
- ③ if문의 조건식이 참이므로 1을 반환하면서 메서드는 종료된다. 그리고 factorial(1)을 호출한 곳으로 되돌아간다.
- ④ 이제 factorial(1)의 결과값인 1을 얻었으므로, return문이 다음의 과정으로 계산된다.

```
return 2 * factorial(1);
→ return 2 * 1;
→ return 2;
```

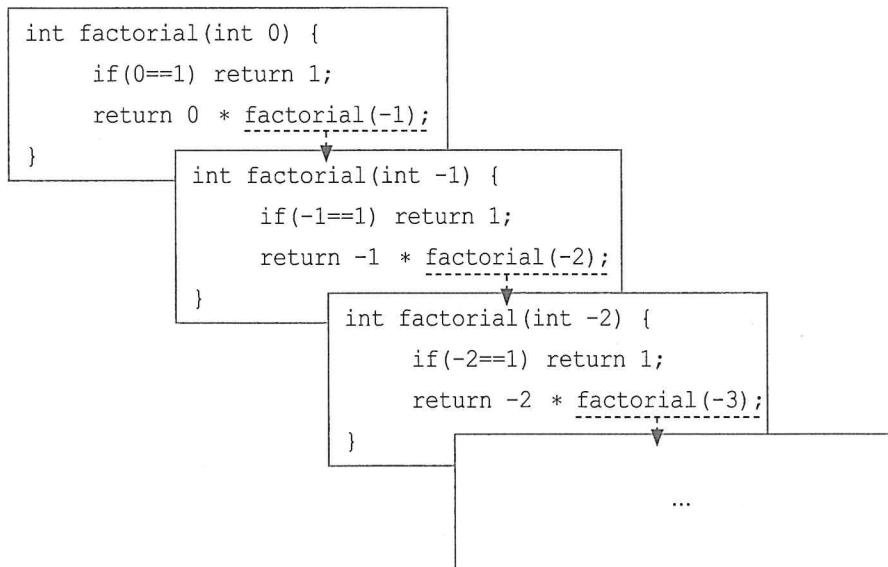


`factorial(2)`는 종료되면서 결과값 2를 반환하고, 이 값은 변수 `result`에 저장된다.

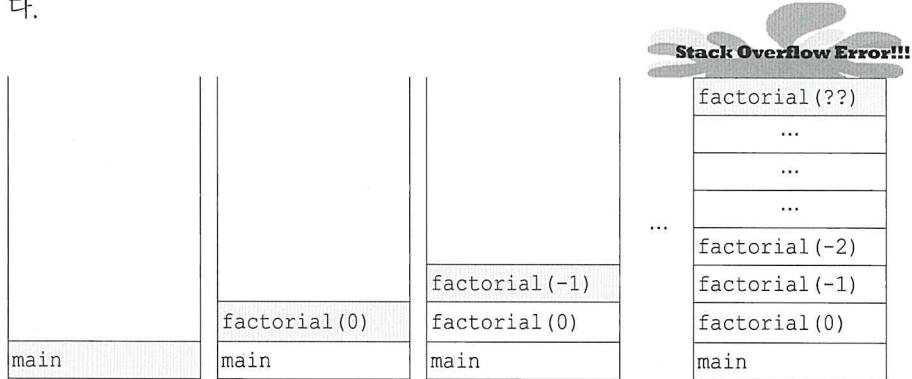
```

int result = factorial(2);
→ int result = 2;
  
```

이제 매개변수 `n`의 값이 커도 어떤 과정으로 계산되는지 충분히 이해할 수 있으리라 생각한다. 그런데 만일 `factorial`메서드의 매개변수 `n`의 값이 0이면 어떻게 될까? 또는 100,000과 같이 큰 수이면 어떻게 될까? 예제를 변경해서 실행해보면 쉽게 알 수 있겠지만 그 전에 잠시 시간을 갖고 어떻게 될 것인지 예측해보자. 먼저 매개변수 `n`의 값이 0인 경우는 `if`문의 조건식이 절대 참이 될 수 없기 때문에 계속해서 재귀호출만 일어날 뿐 메서드가 종료되지 않으므로 스택에 계속 데이터가 쌓여만 간다.



어느 시점에 이르러서는 결국 스택의 저장한계를 넘게 되고, ‘스택오버플로우 에러(Stack Overflow Error)’가 발생한다. 매개변수 n의 값이 100,000과 같이 큰 경우에도 마찬가지다.



이처럼 우리가 메서드를 작성할 때, ‘호출하는 사람이 당연히 알아서 적절한 값을 인자로 주겠지.’라는 막연한 믿음을 가져서는 안 되고, 어떤 값이 들어와도 에러없이 처리되는 견고한 코드를 작성해야 한다. 그래서 ‘매개변수의 유효성검사’가 중요한 것이다.

```
static int factorial(int n) {
    if(n <= 0 || n > 12) return -1; // 매개변수 n의 유효성 검사를 추가
    if(n == 1) return 1;

    return n * factorial(n-1);
}
```

매개변수 n의 상한값을 12로 정한 이유는 $13!$ 의 값이 메서드 factorial의 반환타입인 int의 최대값(약20억)보다 크기 때문이다. 만일 그 이상의 값을 구하고 싶으면 반환타입을 int보다 큰 타입으로 변경하면 된다.

참고로 재귀메서드 factorial을 반복문으로 작성하면 다음과 같다.

<pre>int factorial(int n) { if(n==1) return 1; return n * factorial(n-1); }</pre>		<pre>int factorial(int n) { int result = 1; while(n!=0) result *= n--; return result; }</pre>
---	--	---

while문으로 작성한 오른쪽의 코드는 왼쪽의 재귀호출과 달리 많은 수의 반복에도 ‘스택 오버플로우 에러’와 같은 메모리 부족문제를 겪지 않을 뿐만 아니라 속도도 빠르다.

▼ 예제 6-16/ch6/FactorialTest2.java

```

class FactorialTest2 {
    static long factorial(int n) {
        if(n<=0 || n>20) return -1; // 매개변수의 유효성 검사.
        if(n<=1) return 1;
        return n * factorial(n-1);
    }

    public static void main(String args[]) {
        int n = 21;
        long result = 0;

        for(int i = 1; i <= n; i++) {
            result = factorial(i);

            if(result===-1) {
                System.out.printf("유효하지 않은 값입니다. (0<n<=20) :%d%n", n);
                break;
            }

            System.out.printf("%2d!=%20d%n", i, result);
        }
    } // main의 끝
}

```

▼ 실행결과

1!=	1
2!=	2
3!=	6
4!=	24
5!=	120
6!=	720
7!=	5040
8!=	40320
9!=	362880
10!=	3628800
11!=	39916800
12!=	479001600
13!=	6227020800
14!=	87178291200
15!=	1307674368000
16!=	20922789888000
17!=	355687428096000
18!=	6402373705728000
19!=	121645100408832000
20!=	2432902008176640000
유효하지 않은 값입니다. (0<n<=20) :21	

이전 예제6-15에 매개변수의 유효성을 검사하는 코드를 추가해서 메서드 factorial의 매개변수 n이 음수거나 20보다 크면 -1을 반환하도록 하였다.

▼ 예제 6-17/ch6/MainTest.java

```
class MainTest {
    public static void main(String args[]) {
        main(null);      // 재귀호출. 자기 자신을 다시 호출한다.
    }
}
```

▼ 실행결과

```
java.lang.StackOverflowError
at MainTest.main(MainTest.java:3)
at MainTest.main(MainTest.java:3)
...
at MainTest.main(MainTest.java:3)
at MainTest.main(MainTest.java:3)
```

main메서드 역시 자기 자신을 호출하는 것이 가능하며 아무런 조건도 없이 계속해서 자기 자신을 다시 호출하기 때문에 무한호출에 빠지게 된다.

main메서드가 종료되지 않고 호출스택에 계속해서 쌓이게 되므로 결국 호출스택의 메모리 한계를 넘게 되고 StackOverflowError가 발생하여 프로그램은 비정상적으로 종료된다.

▼ 예제 6-18/ch6/PowerTest.java

```
class PowerTest {
    public static void main(String[] args) {
        int x = 2;
        int n = 5;
        long result = 0;

        for(int i=1; i<=n; i++) {
            result += power(x, i);
        }

        System.out.println(result);
    }

    static long power(int x, int n) {
        if(n==1) return x;
        return x * power(x, n-1);
    }
}
```

▼ 실행결과
62

x^1 부터 x^n 까지의 합을 구하는 예제이다. 재귀호출을 사용하여 x^n 을 구하는 power()를 작성하였다. x는 2, n은 5로 계산했기 때문에 $2^1+2^2+2^3+2^4+2^5$ 의 결과인 62가 출력되었다.

x의 n제곱을 계산하는 메서드는 다음과 같이 정의할 수 있는데, 이 메서드 역시 메서드의 정의에 자신을 포함하는 재귀 메서드이다.

$$f(x, n) = x * f(x, n-1), \text{ 단 } f(x, 1) = x$$

예를 들어 2의 4제곱을 구해보자. x는 2이고, n은 4이므로 각각을 메서드에 대입하면 다음과 같다.

$$f(2, 4) = 2 * f(2, 3)$$

$f(2, 3)$ 은 ' $2 * f(2, 2)$ '이므로, 위의 식에 $f(2, 3)$ 대신 ' $2 * f(2, 2)$ '를 대입하면 다음과 같다.

$$\begin{aligned} f(2, 4) &= 2 * f(2, 3) \\ f(2, 4) &= 2 * 2 * f(2, 2) \end{aligned}$$

같은 방식으로 반복해서 계산하면 다음과 같다. $f(2, 1)$ 은 2라는 것에 주의하자.

$$\begin{aligned} f(2, 4) &= 2 * f(2, 3) \\ \rightarrow f(2, 4) &= 2 * 2 * f(2, 2) \\ \rightarrow f(2, 4) &= 2 * 2 * 2 * f(2, 1) \\ \rightarrow f(2, 4) &= 2 * 2 * 2 * 2 \end{aligned}$$

이 메서드도 재귀호출이 아닌 반복문으로 처리하는 것이 가능하다. 재귀호출의 예를 보여주기 위해 재귀메서드로 작성한 것일 뿐이다.

3.11 클래스 메서드(static메서드)와 인스턴스 메서드

변수에서 그랬던 것과 같이, 메서드 앞에 static이 붙어 있으면 클래스메서드이고 붙어 있지 않으면 인스턴스 메서드이다.

클래스 메서드도 클래스변수처럼, 객체를 생성하지 않고도 '클래스이름.메서드이름(매개변수)'와 같은 식으로 호출이 가능하다. 반면에 인스턴스 메서드는 반드시 객체를 생성해야만 호출할 수 있다.

그렇다면 클래스를 정의할 때, 어느 경우에 static을 사용해서 클래스 메서드로 정의해야하는 것일까?

클래스는 '데이터(변수)와 데이터에 관련된 메서드의 집합'이므로, 같은 클래스 내에 있는 메서드와 멤버변수는 아주 밀접한 관계가 있다.

인스턴스 메서드는 인스턴스 변수와 관련된 작업을 하는, 즉 메서드의 작업을 수행하는데 인스턴스 변수를 필요로 하는 메서드이다. 그런데 인스턴스 변수는 인스턴스(객체)를 생성해야만 만들어지므로 인스턴스 메서드 역시 인스턴스를 생성해야만 호출할 수 있는 것이다.

반면에 메서드 중에서 인스턴스와 관계없는(인스턴스 변수나 인스턴스 메서드를 사용하지 않는) 메서드를 클래스 메서드(static메서드)로 정의한다.

물론 인스턴스 변수를 사용하지 않는다고 해서 반드시 클래스 메서드로 정의해야하는 것은 아니지만 특별한 이유가 없는 한 그렇게 하는 것이 일반적이다.

| 참고 | 클래스 영역에 선언된 변수를 멤버변수라 한다. 멤버변수 중에 static이 붙은 것은 클래스변수(static변수), static이 붙지 않은 것은 인스턴스변수라 한다. 멤버변수는 인스턴스변수와 static변수를 모두 통칭하는 말이다.

1. 클래스를 설계할 때, 멤버변수 중 모든 인스턴스에 공통으로 사용하는 것에 static을 붙인다.
 - 생성된 각 인스턴스는 서로 독립적이기 때문에 각 인스턴스의 변수(iv)는 서로 다른 값을 유지한다. 그러나 모든 인스턴스에서 같은 값이 유지되어야 하는 변수는 static을 붙여서 클래스변수로 정의해야 한다.
2. 클래스 변수(static변수)는 인스턴스를 생성하지 않아도 사용할 수 있다.
 - static이 붙은 변수(클래스변수)는 클래스가 메모리에 올라갈 때 이미 자동적으로 생성되기 때문이다.
3. 클래스 메서드(static메서드)는 인스턴스 변수를 사용할 수 없다.
 - 인스턴스변수는 인스턴스가 반드시 존재해야만 사용할 수 있는데, 클래스메서드(static이 붙은 메서드)는 인스턴스 생성 없이 호출가능하므로 클래스 메서드가 호출되었을 때 인스턴스가 존재하지 않을 수도 있다. 그래서 클래스 메서드에서 인스턴스변수의 사용을 금지한다. 반면에 인스턴스변수나 인스턴스메서드에서는 static이 붙은 멤버들을 사용하는 것이 언제나 가능하다. 인스턴스 변수가 존재한다는 것은 static변수가 이미 메모리에 존재한다는 것을 의미하기 때문이다.
4. 메서드 내에서 인스턴스 변수를 사용하지 않는다면, static을 붙이는 것을 고려한다.
 - 메서드의 작업내용 중에서 인스턴스변수를 필요로 한다면, static을 붙일 수 없다. 반대로 인스턴스변수를 필요로 하지 않는다면 static을 붙이자. 메서드 호출시간이 짧아지므로 성능이 향상된다. static을 안 붙인 메서드(인스턴스메서드)는 실행 시 호출되어야 할 메서드를 찾는 과정이 추가적으로 필요하기 때문에 시간이 더 걸린다.

- 클래스의 멤버변수 중 모든 인스턴스에 공통된 값을 유지해야하는 것이 있는지 살펴보고 있으면, static을 붙여준다.
- 작성한 메서드 중에서 인스턴스 변수나 인스턴스 메서드를 사용하지 않는 메서드에 static을 붙일 것을 고려한다.

| 참고 | 참고로 random()과 같은 Math클래스의 메서드는 모두 클래스 메서드이다. Math클래스에는 인스턴스변수가 하나도 없거니와 작업을 수행하는데 필요한 값을 모두 매개변수로 받아서 처리하기 때문이다.

▼ 예제 6-19/ch6/MyMathTest2.java

```

class MyMath2 {
    long a, b;

    // 인스턴스변수 a, b만을 이용해서 작업하므로 매개변수가 필요없다.
    long add() { return a + b; } // a, b는 인스턴스변수
    long subtract() { return a - b; }
    long multiply() { return a * b; }
    double divide() { return a / b; }

    // 인스턴스변수와 관계없이 매개변수만으로 작업이 가능하다.
    static long add(long a, long b) { return a + b; } // a, b는 지역변수
    static long subtract(long a, long b) { return a - b; }
    static long multiply(long a, long b) { return a * b; }
    static double divide(double a, double b) { return a / b; }
}

class MyMathTest2 {
    public static void main(String args[]) {
        // 클래스메서드 호출. 인스턴스 생성없이 호출가능
        System.out.println(MyMath2.add(200L, 100L));
        System.out.println(MyMath2.subtract(200L, 100L));
        System.out.println(MyMath2.multiply(200L, 100L));
        System.out.println(MyMath2.divide(200.0, 100.0));

        MyMath2 mm = new MyMath2(); // 인스턴스를 생성
        mm.a = 200L;
        mm.b = 100L;
        // 인스턴스메서드는 객체생성 후에만 호출이 가능함.
        System.out.println(mm.add());
        System.out.println(mm.subtract());
        System.out.println(mm.multiply());
        System.out.println(mm.divide());
    }
}

```

▼ 실행결과
300
100
20000
2.0
300
100
20000
2.0

인스턴스메서드인 add(), subtract(), multiply(), divide()는 인스턴스변수인 a와 b만으로도 충분히 작업이 가능하기 때문에, 매개변수를 필요로 하지 않으므로 괄호()에 매개변수를 선언하지 않았다.

반면에 add(long a, long b), subtract(long a, long b) 등은 인스턴스변수 없이 매개변수만으로 작업을 수행하기 때문에 static을 붙여서 클래스메서드로 선언하였다.

그래서 MyMathTest2의 main메서드에서 보면, 클래스메서드는 객체생성없이 바로 호출이 가능했고, 인스턴스메서드는 MyMath2클래스의 인스턴스를 생성한 후에야 호출이 가능했다.

이 예제를 통해서 어떤 경우 인스턴스메서드로, 또는 클래스메서드로 선언해야하는지, 그리고 그 차이를 이해하는 것은 매우 중요하다.

3.12 클래스 멤버와 인스턴스 멤버간의 참조와 호출

같은 클래스에 속한 멤버들 간에는 별도의 인스턴스를 생성하지 않고도 서로 참조 또는 호출이 가능하다. 단, 클래스멤버가 인스턴스 멤버를 참조 또는 호출하고자 하는 경우에 인스턴스를 생성해야 한다.

그 이유는 인스턴스 멤버가 존재하는 시점에 클래스 멤버는 항상 존재하지만, 클래스멤버가 존재하는 시점에 인스턴스 멤버가 존재하지 않을 수도 있기 때문이다.

| 참고 | 인스턴스 멤버란 인스턴스 변수와 인스턴스 메서드를 의미한다.

```
class TestClass {
    void instanceMethod() {}           // 인스턴스메서드
    static void staticMethod() {}       // static메서드

    void instanceMethod2() {           // 인스턴스메서드
        instanceMethod();             // 다른 인스턴스메서드를 호출한다.
        staticMethod();               // static메서드를 호출한다.
    }

    static void staticMethod2() { // static메서드
        instanceMethod();          // 애러!!! 인스턴스메서드를 호출할 수 없다.
        staticMethod();             // static메서드는 호출 할 수 있다.
    }
} // end of class
```

위의 코드는 같은 클래스 내의 인스턴스 메서드와 static메서드 간의 호출에 대해서 설명하고 있다. 같은 클래스 내의 메서드는 서로 객체의 생성이나 참조변수 없이 직접 호출이 가능하지만 static메서드는 인스턴스 메서드를 호출할 수 없다.

```
class TestClass2 {
    int iv;                      // 인스턴스 변수
    static int cv;                // 클래스 변수

    void instanceMethod() {        // 인스턴스 메서드
        System.out.println(iv);   // 인스턴스 변수를 사용할 수 있다.
        System.out.println(cv);   // 클래스 변수를 사용할 수 있다.
    }

    static void staticMethod() { // static메서드
        System.out.println(iv); // 애러!!! 인스턴스 변수를 사용할 수 없다.
        System.out.println(cv); // 클래스 변수는 사용할 수 있다.
    }
} // end of class
```

이번엔 변수와 메서드간의 호출에 대해서 살펴보자. 메서드간의 호출과 마찬가지로 인스턴스메서드는 인스턴스변수를 사용할 수 있지만, static메서드는 인스턴스변수를 사용할 수 없다.

▼ 예제 6-20/ch6/MemberCall.java

```

class MemberCall {
    int iv = 10;
    static int cv = 20;

    int iv2 = cv;
    // static int cv2 = iv;           // 에러. 클래스변수는 인스턴스 변수를 사용할 수 없음.
    static int cv2 = new MemberCall().iv; // 이처럼 객체를 생성해야 사용가능.

    static void staticMethod1() {
        System.out.println(cv);
    }
    // System.out.println(iv); // 에러. 클래스메서드에서 인스턴스변수를 사용불가.
    MemberCall c = new MemberCall();
    System.out.println(c.iv); // 객체를 생성한 후에야 인스턴스변수의 참조가능.
}

void instanceMethod1() {
    System.out.println(cv);
    System.out.println(iv); // 인스턴스메서드에서는 인스턴스변수를 바로 사용가능.
}

static void staticMethod2() {
    staticMethod1();
}
// instanceMethod1(); // 에러. 클래스메서드에서는 인스턴스메서드를 호출할 수 없음.
MemberCall c = new MemberCall();
c.instanceMethod1(); // 인스턴스를 생성한 후에야 호출할 수 있음.

void instanceMethod2() { // 인스턴스메서드에서는 인스턴스메서드와 클래스메서드
    staticMethod1(); // 모두 인스턴스 생성없이 바로 호출이 가능하다.
    instanceMethod1();
}
}

```

클래스멤버(클래스변수와 클래스메서드)는 언제나 참조 또는 호출이 가능하기 때문에 인스턴스멤버가 클래스멤버를 사용하는 것은 아무런 문제가 안 된다. 클래스멤버간의 참조 또는 호출 역시 아무런 문제가 없다.

그러나, 인스턴스멤버(인스턴스변수와 인스턴스메서드)는 반드시 객체를 생성한 후에만 참조 또는 호출이 가능하기 때문에 클래스멤버가 인스턴스멤버를 참조, 호출하기 위해서는 객체를 생성하여야 한다.

하지만, 인스턴스멤버간의 호출에는 아무런 문제가 없다. 하나의 인스턴스멤버가 존재한다는 것은 인스턴스가 이미 생성되어있다는 것을 의미하며, 즉 다른 인스턴스멤버들도 모두 존재하기 때문이다.

실제로는 같은 클래스 내에서 클래스멤버가 인스턴스멤버를 참조 또는 호출해야하는 경우는 드물다. 만일 그런 경우가 발생한다면, 인스턴스메서드로 작성해야 할 메서드를 클래스메서드로 한 것은 아닌지 한 번 더 생각해보야 한다.

■ 알아두면 좋아요!

수학에서의 대입법처럼, `c = new MemberCall()`이므로 `c.instanceMethod1();`에서 `c` 대신 `new MemberCall()`을 대입하여 사용할 수 있다.

```
MemberCall c = new MemberCall();
int result = c.instanceMethod1();
```

그래서 위의 두 줄을 다음과 같이 한 줄로 할 수 있다.

```
int result = new MemberCall().instanceMethod1();
```

대신 참조변수를 선언하지 않았기 때문에 생성된 MemberCall인스턴스는 더 이상 사용할 수 없다.

4. 오버로딩(overloading)

4.1 오버로딩이란?

메서드도 변수와 마찬가지로 같은 클래스 내에서 서로 구별될 수 있어야 하기 때문에 각기 다른 이름을 가져야 한다. 그러나 자바에서는 한 클래스 내에 이미 사용하려는 이름과 같은 이름을 가진 메서드가 있더라도 매개변수의 개수 또는 타입이 다르면, 같은 이름을 사용해서 메서드를 정의할 수 있다.

이처럼, 한 클래스 내에 같은 이름의 메서드를 여러 개 정의하는 것을 ‘메서드 오버로딩(method overloading)’ 또는 간단히 ‘오버로딩(overloading)’이라 한다.

오버로딩(overloading)의 사전적 의미는 ‘과적하다.’ 즉, 많이 실는 것을 뜻한다. 보통 하나의 메서드 이름에 하나의 기능만을 구현해야하는데, 하나의 메서드 이름으로 여러 기능을 구현하기 때문에 붙여진 이름이라 생각할 수 있다. 앞으로는 ‘메서드 오버로딩’을 간단히 ‘오버로딩’이라고 하겠다.

4.2 오버로딩의 조건

같은 이름의 메서드를 정의한다고 해서 무조건 오버로딩인 것은 아니다. 오버로딩이 성립하기 위해서는 다음과 같은 조건을 만족해야한다.

1. 메서드 이름이 같아야 한다.
2. 매개변수의 개수 또는 타입이 달라야 한다.

비록 메서드의 이름이 같다 하더라도 매개변수가 다르면 서로 구별될 수 있기 때문에 오버로딩이 가능한 것이다. 위의 조건을 만족시키지 못하는 메서드는 중복 정의로 간주되어 컴파일 시에 에러가 발생한다. 그리고 오버로딩된 메서드들은 매개변수에 의해서만 구별될 수 있으므로 반환 타입은 오버로딩을 구현하는데 아무런 영향을 주지 못한다는 것에 주의하자.

4.3 오버로딩의 예

오버로딩의 예로 가장 대표적인 것은 `println`메서드이다. 지금까지 여러분은 `println`메서드에 괄호 안에 값만 지정해주면 화면에 출력하는데 아무런 어려움이 없었다.

하지만, 실제로는 `println`메서드를 호출할 때 매개변수로 지정하는 값의 타입에 따라서 호출되는 `println`메서드가 달라진다.

`PrintStream`클래스에는 어떤 종류의 매개변수를 지정해도 출력할 수 있도록 아래와 같이 10개의 오버로딩된 `println`메서드를 정의해놓고 있다.

```
void println()
void println(boolean x)
void println(char x)
void println(char[] x)
void println(double x)
void println(float x)
void println(int x)
void println(long x)
void println(Object x)
void println(String x)
```

println메서드를 호출할 때 매개변수로 넘겨주는 값의 타입에 따라서 위의 오버로딩된 메서드들 중의 하나가 선택되어 실행되는 것이다.

몇 가지 예를 들어 오버로딩에 대해 자세히 설명하고자 한다.

[보기 1]

```
int add(int a, int b) { return a+b; }
int add(int x, int y) { return x+y; }
```

위의 두 메서드는 매개변수의 이름만 다를 뿐 매개변수의 타입이 같기 때문에 오버로딩이 성립하지 않는다. 매개변수의 이름이 다르면 메서드 내에서 사용되는 변수의 이름이 달라질 뿐, 아무런 의미가 없다. 그래서 이 두 메서드는 정확히 같은 것이다. 마치 수학에서 ' $f(x) = x + 1$ '과 ' $f(a) = a + 1$ '이 같은 표현인 것과 같다.

컴파일하면, 'add(int,int) is already defined(이미 같은 메서드가 정의되었다.)'라는 메시지가 나타날 것이다.

[보기 2]

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

이번 경우는 리턴타입만 다른 경우이다. 매개변수의 타입과 개수가 일치하기 때문에 add(3,3)과 같이 호출하였을 때 어떤 메서드가 호출된 것인지 결정할 수 없기 때문에 오버로딩으로 간주되지 않는다.

이 경우 역시 컴파일하면, 'add(int,int) is already defined(이미 같은 메서드가 정의되었다.)'라는 메시지가 나타날 것이다.

[보기 3]

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

두 메서드 모두 int형과 long형 매개변수가 하나씩 선언되어 있지만, 서로 순서가 다른 경우이다. 이 경우에는 호출 시 매개변수의 값에 의해 호출될 메서드가 구분될 수 있으므로 중복된 메서드 정의가 아닌, 오버로딩으로 간주한다.

이처럼 단지 매개변수의 순서만을 다르게 하여 오버로딩을 구현하면, 사용자가 매개변수의 순서를 외우지 않아도 되는 장점이 있지만, 오히려 단점이 될 수도 있기 때문에 주의해야한다.

예를 들어 add(3,3L)과 같이 호출되면 첫 번째 메서드가, add(3L, 3)과 같이 호출되면 두 번째 메서드가 호출된다. 단, 이 경우에는 add(3,3)과 같이 호출할 수 없다. 이와 같이 호출할 경우, 두 메서드 중 어느 메서드가 호출된 것인지 알 수 없기 때문에 메서드를 호출하는 곳에서 컴파일 에러가 발생한다.

[보기 4]

```
int add(int a, int b) { return a+b; }
long add(long a, long b) { return a+b; }
long add(int[] a) { // 배열의 모든 요소의 합을 반환한다.
    long result = 0;
    for(int i=0; i < a.length; i++) {
        result += a[i];
    }
    return result;
}
```

위 메서드들은 모두 바르게 오버로딩되어 있다. 정의된 매개변수가 서로 다른 해도, 세 메서드 모두 매개변수로 넘겨받은 값을 더해서 그 결과를 돌려주는 일을 한다.

같은 일을 하지만 매개변수를 달리해야하는 경우에, 이와 같이 이름은 같고 매개변수를 다르게 하여 오버로딩을 구현한다.

4.4 오버로딩의 장점

지금까지 오버로딩의 정의와 성립하기 위한 조건을 알아보았다. 그렇다면 오버로딩을 구현함으로써 얻는 이득은 무엇인가에 대해서 생각해보도록 하자.

만일 메서드도 변수처럼 단지 이름만으로 구별된다면, 한 클래스내의 모든 메서드들은 이름이 달라야한다. 그렇다면, 이전에 예로 들었던 10가지의 println메서드들은 각기 다른 이름을 가져야 한다.

예를 들면, 아래와 같은 방식으로 메서드 이름이 변경되어야 할 것이다.

```
void println()
void printlnBoolean(boolean x)
void printlnChar(char x)
void printlnDouble(double x)
void printlnString(String x)
```

모두 근본적으로는 같은 기능을 하는 메서드들이지만, 서로 다른 이름을 가져야 하기 때문에 메서드를 작성하는 쪽에서는 이름을 짓기도 어렵고, 메서드를 사용하는 쪽에서는 이름을 일일이 구분해서 기억해야하기 때문에 서로 부담이 된다.

하지만 오버로딩을 통해 여러 메서드들이 `println`이라는 하나의 이름으로 정의될 수 있다면, `println`이라는 이름만 기억하면 되므로 기억하기도 쉽고 이름도 짧게 할 수 있어서 오류의 가능성을 많이 줄일 수 있다. 그리고 메서드의 이름만 보고도 ‘이 메서드들은 이름이 같으니, 같은 기능을 하겠구나.’라고 쉽게 예측할 수 있게 된다.

또 하나의 장점은 메서드의 이름을 절약할 수 있다는 것이다. 하나의 이름으로 여러 개의 메서드를 정의할 수 있으니, 메서드의 이름을 짓는데 고민을 덜 수 있는 동시에 사용되어 있어야 할 메서드 이름을 다른 메서드의 이름으로 사용할 수 있기 때문이다.

▼ 예제 6-21/ch6/OverloadingTest.java

```
class OverloadingTest {
    public static void main(String args[]) {
        MyMath3 mm = new MyMath3();
        System.out.println("mm.add(3, 3) 결과: " + mm.add(3, 3));
        System.out.println("mm.add(3L, 3) 결과: " + mm.add(3L, 3));
        System.out.println("mm.add(3, 3L) 결과: " + mm.add(3, 3L));
        System.out.println("mm.add(3L, 3L) 결과: " + mm.add(3L, 3L));

        int[] a = {100, 200, 300};
        System.out.println("mm.add(a) 결과: " + mm.add(a));
    }
}

class MyMath3 {
    int add(int a, int b) {
        System.out.print("int add(int a, int b) - ");
        return a+b;
    }

    long add(int a, long b) {
        System.out.print("long add(int a, long b) - ");
        return a+b;
    }

    long add(long a, int b) {
        System.out.print("long add(long a, int b) - ");
        return a+b;
    }

    long add(long a, long b) {
        System.out.print("long add(long a, long b) - ");
        return a+b;
    }

    int add(int[] a) { // 배열의 모든 요소의 합을 결과로 돌려준다.
        System.out.print("int add(int[] a) - ");
        int result = 0;
        for(int i=0; i < a.length; i++) {
            result += a[i];
        }
        return result;
    }
}
```

▼ 실행결과

```
int add(int a, int b) - mm.add(3, 3) 결과: 6
long add(long a, int b) - mm.add(3L, 3) 결과: 6
long add(int a, long b) - mm.add(3, 3L) 결과: 6
long add(long a, long b) - mm.add(3L, 3L) 결과: 6
int add(int[] a) - mm.add(a) 결과: 600
```

| 참고 | add(3L, 3), add(3, 3L), add(3L, 3L)의 결과는 모두 6L이지만, System.out.println(6L);을 수행하면 6이 출력된다. 리터럴의 접미사는 출력되지 않는다.

실행결과의 출력순서를 보고 의아할 수도 있다. 어떻게 add메서드가 println메서드보다 먼저 출력될 수 있는가?

```
System.out.println("mm.add(3, 3) 결과:" + mm.add(3, 3));
```

println메서드가 결과를 출력하려면, add메서드의 결과가 먼저 계산되어야 하기 때문이다. 간단히 위의 문장이 아래의 두 문장을 하나로 합친 것이라고 생각하면 이해가 쉬울 것이다.

```
int result = mm.add(3, 3);
System.out.println("mm.add(3, 3) 결과:" + result);
```

4.5 가변인자(varargs)와 오버로딩

기존에는 메서드의 매개변수 개수가 고정적이었으나 JDK1.5부터 동적으로 지정해 줄 수 있게 되었으며, 이 기능을 ‘가변인자(variable arguments)’라고 한다.

가변인자는 ‘타입... 변수명’과 같은 형식으로 선언하며, PrintStream클래스의 printf()가 대표적인 예이다.

```
public PrintStream printf(String format, Object... args) { ... }
```

위와 같이 가변인자 외에도 매개변수가 더 있다면, 가변인자를 매개변수 중에서 제일 마지막에 선언해야 한다. 그렇지 않으면, 컴파일 에러가 발생한다. 가변인자인지 아닌지를 구별할 방법이 없기 때문에 허용하지 않는 것이다.

```
// 컴파일 에러 발생 - 가변인자는 항상 마지막 매개변수이어야 한다.
public PrintStream printf(Object... args, String format) {
    ...
}
```

만일 여러 문자열을 하나로 결합하여 반환하는 concatenate메서드를 작성한다면, 아래와 같이 매개변수의 개수를 다르게 해서 여러 개의 메서드를 작성해야할 것이다.

```
String concatenate(String s1, String s2) { ... }
String concatenate(String s1, String s2, String s3) { ... }
String concatenate(String s1, String s2, String s3, String s4){ ... }
```

이럴 때, 가변인자를 사용하면 메서드 하나로 간단히 대체할 수 있다.

```
String concatenate(String... str) { ... }
```

이 메서드를 호출할 때는 아래와 같이 인자의 개수를 가변적으로 할 수 있다. 심지어는 인자가 아예 없어도 되고 배열도 인자가 될 수 있다.

```
System.out.println(concatenate());           // 인자가 없음
System.out.println(concatenate("a"));          // 인자가 하나
System.out.println(concatenate("a", "b"));       // 인자가 둘
System.out.println(concatenate(new String[]{"A", "B"})); // 배열도 가능
```

이쯤에서 아마도 눈치를 챘을 것이다. 그렇다. 가변인자는 내부적으로 배열을 이용하는 것이다. 그래서 가변인자가 선언된 메서드를 호출할 때마다 배열이 새로 생성된다. 가변인자가 편리하지만, 이런 비효율이 숨어있으므로 꼭 필요한 경우에만 가변인자를 사용하자.

그러면 가변인자는 아래와 같이 매개변수의 타입을 배열로 하는 것과 어떤 차이가 있는가?

```
String concatenate(String[] str) { ... }

String result = concatenate(new String[0]); // 인자로 배열을 지정
String result = concatenate(null);         // 인자로 null을 지정
String result = concatenate();             // 예러. 인자가 필요함.
```

매개변수의 타입을 배열로 하면, 반드시 인자를 지정해 줘야하기 때문에, 위의 코드에서처럼 인자를 생략할 수 없다. 그래서 null이나 길이가 0인 배열을 인자로 지정해줘야 하는 불편함이 있다.

| 참고 | C언어와 달리 자바에서는 길이가 0인 배열을 생성하는 것이 허용된다.

가변인자를 오버로딩할 때 한 가지 주의해야할 점이 있는데, 먼저 예제부터 살펴보자.

▼ 예제 6-22/ch6/VarArgsEx.java

```

class VarArgsEx {
    public static void main(String[] args) {
        String[] strArr = { "100", "200", "300" };

        System.out.println(concatenate("", "100", "200", "300"));
        System.out.println(concatenate("-", strArr));
        System.out.println(concatenate("", new String[]{"1", "2", "3"}));
        System.out.println("[ "+concatenate("", new String[0])+" ]");
        System.out.println("[ "+concatenate(",")+" ]");
    }

    static String concatenate(String delim, String... args) {
        String result = "";

        for(String str : args) {
            result += str + delim;
        }

        return result;
    }

    /*
     * static String concatenate(String... args) {
     *     return concatenate("", args);
     * }
    */
} // class

```

▼ 실행결과
100200300
100-200-300-
1,2,3,
[]
[]

concatenate메서드는 매개변수로 입력된 문자열에 구분자를 사이에 포함시켜 결합해서 반환한다. 가변인자로 매개변수를 선언했기 때문에 문자열을 개수의 제약없이 매개변수로 지정할 수 있다:

```

String[] strArr = new String[]{"100", "200", "300"};
System.out.println(concatenate("-", strArr));

```

위의 두 문장을 하나로 합치면 아래와 같이 쓸 수 있다.

```
System.out.println(concatenate("-", new String[]{"100", "200", "300"}));
```

그러나 아래와 같은 문장은 허용되지 않는다는 것에 주의하자.

```
System.out.println(concatenate("-", {"100", "200", "300"}));
```

위의 예제에서는 주석처리하였지만, concatenate메서드의 또 다른 오버로딩된 메서드가 있다.

```
static String concatenate(String delim, String... args) {
    String result = "";
    for(String str : args) {
        result += str + delim;
    }
    return result;
}

static String concatenate(String... args) {
    return concatenate("",args);
}
```

이 두 메서드는 별 문제가 없어 보이지만 위의 예제에서 주석을 풀고 컴파일을 하면 아래와 같이 컴파일에러가 발생한다.

```
VarArgsEx.java:5: error: reference to concatenate is ambiguous
    System.out.println(concatenate("-", "100", "200", "300"));
                                         ^
both method concatenate(String, String...) in VarArgsEx and method
concatenate(String...) in VarArgsEx match
1 error
```

에러의 내용을 살펴보면 두 오버로딩된 메서드가 구분되지 않아서 발생하는 것임을 알 수 있다. 가변인자를 선언한 메서드를 오버로딩하면, 메서드를 호출했을 때 이와 같이 구별되지 못하는 경우가 발생하기 쉽기 때문에 주의해야 한다. 가능하면 가변인자를 사용한 메서드는 오버로딩하지 않는 것이 좋다.

5. 생성자(Constructor)

5.1 생성자란?

생성자는 인스턴스가 생성될 때 호출되는 ‘인스턴스 초기화 메서드’이다. 따라서 인스턴스 변수의 초기화 작업에 주로 사용되며, 인스턴스 생성 시에 실행되어야 할 작업을 위해서도 사용된다.

| 참고 | 인스턴스 초기화란, 인스턴스변수들을 초기화하는 것을 뜻한다.

생성자 역시 메서드처럼 클래스 내에 선언되며, 구조도 메서드와 유사하지만 리턴값이 없다는 점이 다르다. 그렇다고 해서 생성자 앞에 리턴값이 없음을 뜻하는 키워드 void를 사용하지는 않고, 단지 아무 것도 적지 않는다. 생성자의 조건은 다음과 같다.

1. 생성자의 이름은 클래스의 이름과 같아야 한다.
2. 생성자는 리턴 값이 없다.

| 참고 | 생성자도 메서드이기 때문에 리턴값이 없다는 의미의 void를 붙여야 하지만, 모든 생성자가 리턴값이 없으므로 void를 생략할 수 있게 한 것이다.

생성자는 다음과 같이 정의한다. 생성자도 오버로딩이 가능하므로 하나의 클래스에 여러 개의 생성자가 존재할 수 있다.

```
클래스이름(타입 변수명, 타입 변수명, ...) {
    // 인스턴스 생성 시 수행될 코드,
    // 주로 인스턴스 변수의 초기화 코드를 적는다.
}

class Card {
    Card() {           // 매개변수가 없는 생성자.
        ...
    }

    Card(String k, int num) {      // 매개변수가 있는 생성자.
        ...
    }
}
```

연산자 new가 인스턴스를 생성하는 것이지 생성자가 인스턴스를 생성하는 것이 아니다. 생성자라는 용어 때문에 오해하기 쉬운데, 생성자는 단순히 인스턴스변수들의 초기화에 사용되는 조금 특별한 메서드일 뿐이다. 생성자가 갖는 몇 가지 특징만 제외하면 메서드와 다르지 않다.

Card클래스의 인스턴스를 생성하는 코드를 예로 들어, 수행되는 과정을 단계별로 나누어 보면 다음과 같다.

```
Card c = new Card();
```

1. 연산자 new에 의해서 메모리(heap)에 Card클래스의 인스턴스가 생성된다.
2. 생성자 Card()가 호출되어 수행된다.
3. 연산자 new의 결과로, 생성된 Card인스턴스의 주소가 반환되어 참조변수 c에 저장된다.

지금까지 인스턴스를 생성하기 위해 사용해왔던 ‘클래스이름()’이 바로 생성자였던 것이다. 인스턴스를 생성할 때는 반드시 클래스 내에 정의된 생성자 중의 하나를 선택하여 지정해주어야 한다.

5.2 기본 생성자(default constructor)

지금까지는 생성자를 모르고도 프로그래밍을 해 왔지만, 사실 모든 클래스에는 반드시 하나 이상의 생성자가 정의되어 있어야 한다.

그러나 지금까지 클래스에 생성자를 정의하지 않고도 인스턴스를 생성할 수 있었던 이유는 컴파일러가 제공하는 ‘기본 생성자(default constructor)’ 덕분이었다.

컴파일 할 때, 소스파일(*.java)의 클래스에 생성자가 하나도 정의되지 않은 경우 컴파일러는 자동적으로 아래와 같은 내용의 기본 생성자를 추가하여 컴파일 한다.

```
클래스이름() { }  
Card() { }
```

컴파일러가 자동적으로 추가해주는 기본 생성자는 이와 같이 매개변수도 없고 아무런 내용도 없는 아주 간단한 것이다.

그동안 우리는 인스턴스를 생성할 때 컴파일러가 제공한 기본 생성자를 사용해왔던 것이다. 특별히 인스턴스 초기화 작업이 요구되어지지 않는다면 생성자를 정의하지 않고 컴파일러가 제공하는 기본 생성자를 사용하는 것도 좋다.

| 참고 | 클래스의 '접근 제어자(Access Modifier)'가 public인 경우에는 기본 생성자로 'public 클래스이름() {}'이 추가된다.

▼ 예제 6-23/ch6/ConstructorTest.java

```
class Data1 {  
    int value;  
}
```

```

class Data2 {
    int value;

    Data2(int x) { // 매개변수가 있는 생성자.
        value = x;
    }
}

class ConstructorTest {
    public static void main(String[] args) {
        Data1 d1 = new Data1();
        Data2 d2 = new Data2();           // compile error 발생
    }
}

```

▼ 실행결과

```

ConstructorTest.java:15: cannot resolve symbol
symbol : constructor Data2 ()
location: class Data2
        Data2 d2 = new Data2();           // compile error 발생
^
1 error

```

이 예제를 컴파일 하면 위와 같은 에러메시지가 나타난다. 이것은 Data2에서 Data2()라는 생성자를 찾을 수 없다는 내용의 에러메시지인데, Data2에 생성자 Data2()가 정의되어 있지 않기 때문에 에러가 발생한 것이다.

Data1의 인스턴스를 생성하는 코드에는 에러가 없는데, Data2의 인스턴스를 생성하는 코드에서 에러가 발생하는 이유는 무엇일까?

그 이유는 Data1에는 정의되어 있는 생성자가 하나도 없으므로 컴파일러가 기본 생성자를 추가해주었지만, Data2에는 이미 생성자 Data2(int x)가 정의되어 있으므로 기본 생성자가 추가되지 않았기 때문이다.

컴파일러가 자동적으로 기본 생성자를 추가해주는 경우는 ‘클래스 내에 생성자가 하나도 없을 때’뿐이라는 것을 명심해야한다.

```

Data1 d1 = new Data1();
Data2 d2 = new Data2(); // 에러

```

```

Data1 d1 = new Data1();
Data2 d2 = new Data2(10); // OK

```

이 예제에서 컴파일 에러가 발생하지 않도록 하기 위해서는 위의 오른쪽 코드와 같이 Data2의 인스턴스를 생성할 때 생성자 Data2(int x)를 사용하던가, 아니면 클래스 Data2에 생성자 Data2()를 추가로 정의해주면 된다.

기본 생성자가 컴파일러에 의해서 추가되는 경우는

클래스에 정의된 생성자가 하나도 없을 때 뿐이다.

5.3 매개변수가 있는 생성자

생성자도 메서드처럼 매개변수를 선언하여 호출 시 값을 넘겨받아서 인스턴스의 초기화 작업에 사용할 수 있다. 인스턴스마다 각기 다른 값으로 초기화되어야하는 경우가 많기 때문에 매개변수를 사용한 초기화는 매우 유용하다.

아래의 코드는 자동차를 클래스로 정의한 것인데, 단순히 color, gearType, door 세 개의 인스턴스변수와 두 개의 생성자만을 가지고 있다.

```
class Car {  
    String color;           // 색상  
    String gearType;       // 변속기 종류 - auto(자동), manual(수동)  
    int door;              // 문의 개수  
  
    Car() {} // 생성자  
    Car(String c, String g, int d) { // 생성자  
        color = c;  
        gearType = g;  
        door = d;  
    }  
}
```

Car인스턴스를 생성할 때, 생성자 Car()를 사용한다면, 인스턴스를 생성한 다음에 인스턴스변수들을 따로 초기화해주어야 하지만, 매개변수가 있는 생성자 Car(String color, String gearType, int door)를 사용한다면 인스턴스를 생성하는 동시에 원하는 값으로 초기화를 할 수 있게 된다.

인스턴스를 생성한 다음에 인스턴스변수의 값을 변경하는 것보다 매개변수를 갖는 생성자를 사용하는 것이 코드를 보다 간결하고 직관적으로 만든다.

```
Car c = new Car();  
c.color = "white";  
c.gearType = "auto";  
c.door = 4;  
—————> Car c = new Car("white", "auto", 4);
```

위의 양쪽 코드 모두 같은 내용이지만, 오른쪽의 코드가 더 간결하고 직관적이다. 이처럼 클래스를 작성할 때 다양한 생성자를 제공함으로써 인스턴스 생성 후에 별도로 초기화를 하지 않아도 되도록 하는 것이 바람직하다.

▼ 예제 6-24/ch6/CarTest.java

```
class Car {  
    String color;           // 색상  
    String gearType;       // 변속기 종류 - auto(자동), manual(수동)  
    int door;              // 문의 개수  
  
    Car() {}
```

```

Car(String c, String g, int d) {
    color = c;
    gearType = g;
    door = d;
}
}

class CarTest {
    public static void main(String[] args) {
        Car c1 = new Car();
        c1.color      = "white";
        c1.gearType   = "auto";
        c1.door       = 4;

        Car c2 = new Car("white", "auto", 4);

        System.out.println("c1의 color=" + c1.color + ", gearType="
                           + c1.gearType + ", door=" + c1.door);
        System.out.println("c2의 color=" + c2.color + ", gearType="
                           + c2.gearType + ", door=" + c2.door);
    }
}

```

▼ 실행결과

c1의 color=white, gearType=auto, door=4
 c2의 color=white, gearType=auto, door=4

5.4 생성자에서 다른 생성자 호출하기 – this(), this

같은 클래스의 멤버들 간에 서로 호출할 수 있는 것처럼 생성자 간에도 서로 호출이 가능하다. 단, 다음의 두 조건을 만족시켜야 한다.

- 생성자의 이름으로 클래스이름 대신 this를 사용한다.
- 한 생성자에서 다른 생성자를 호출할 때는 반드시 첫 줄에서만 호출이 가능하다.

다음의 코드는 생성자를 작성할 때 지켜야하는 두 조건을 모두 만족시키지 못했기 때문에 에러가 발생한다.

```

Car(String color) {
    door = 5;           // 첫 번째 줄
    Car(color, "auto", 4); // 에러1. 생성자의 두 번째 줄에서 다른 생성자 호출
}                         // 에러2. this(color, "auto", 4);로 해야함

```

생성자 내에서 다른 생성자를 호출할 때는 클래스이름인 ‘Car’대신 ‘this’를 사용해야하는데 그러지 않아서 에러이고, 또 다른 에러는 생성자 호출이 첫 번째 줄이 아닌 두 번째 줄이기 때문에 에러이다.

생성자에서 다른 생성자를 첫 줄에서만 호출이 가능하도록 한 이유는 생성자 내에서 초기화 작업도중에 다른 생성자를 호출하게 되면, 호출된 다른 생성자 내에서도 멤버변수들의 값을 초기화를 할 것이므로 다른 생성자를 호출하기 이전의 초기화 작업이 무의미해질 수 있기 때문이다. 이에 대해서는 7장에서 좀 더 자세히 배우게 된다.

▼ 예제 6-25/ch6/CarTest2.java

```

class Car {
    String color;          // 색상
    String gearType;       // 변속기 종류 - auto(자동), manual(수동)
    int door;              // 문의 개수

    Car() {
        this("white", "auto", 4); •—————
    }

    Car(String color) {
        this(color, "auto", 4);
    }
    Car(String color, String gearType, int door) {
        this.color = color;
        this.gearType = gearType;
        this.door = door;
    }
}

class CarTest2 {
    public static void main(String[] args) {
        Car c1 = new Car();
        Car c2 = new Car("blue");

        System.out.println("c1의 color=" + c1.color + ", gearType="
                           + c1.gearType + ", door=" + c1.door);
        System.out.println("c2의 color=" + c2.color + ", gearType="
                           + c2.gearType + ", door=" + c2.door);
    }
}

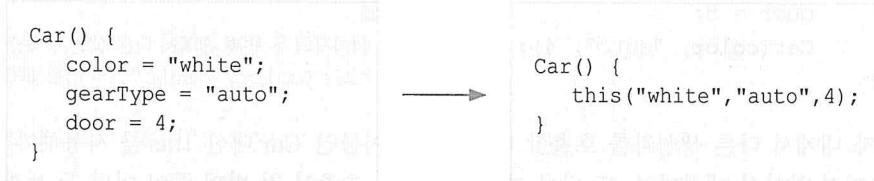
```

Car(String color, String gearType, int door)를 호출

▼ 실행결과

c1의 color=white, gearType=auto, door=4
c2의 color=blue, gearType=auto, door=4

생성자 Car()에서 또 다른 생성자 Car(String color, String gearType, int door)를 호출하였다. 이처럼 생성자간의 호출에는 생성자의 이름 대신 this를 사용해야만 하므로 ‘Car’ 대신 ‘this’를 사용했다. 그리고 생성자 Car()의 첫째 줄에서 호출하였다는 점을 눈여겨보기 바란다.



위 코드는 양쪽 모두 같은 일을 하지만 오른쪽의 코드는 생성자 Car(String color, String gearType, int door)를 활용해서 더 간략히 한 것이다. Car c1 = new Car();와 같이 생성자Car()를 사용해서 Car인스턴스를 생성한 경우에, 인스턴스변수 color는 “white”, gearType은 “auto”, door는 4로 초기화 되도록 하였다.

이것은 마치 실생활에서 자동차(Car인스턴스)를 생산할 때, 아무런 옵션도 주지 않으면, 기본적으로 흰색(white)에 자동변속기어(auto) 그리고 문의 개수가 4개인 자동차가 생산되도록 하는 것에 비유할 수 있다.

같은 클래스 내의 생성자들은 일반적으로 서로 관계가 깊은 경우가 많아서 이처럼 서로 호출하도록 하여 유기적으로 연결해주면 더 좋은 코드를 얻을 수 있다. 그리고 수정이 필요한 경우에도 보다 적은 코드만을 변경하면 되므로 유지보수가 쉬워진다.

```
Car(String c, String g, int d) {
    color = c;
    gearType = g;
    door = d;
}

Car(String color, String gearType,
     int door) {
    this.color = color;
    this.gearType = gearType;
    this.door = door;
}
```

왼쪽 코드의 ‘color = c;’는 생성자의 매개변수로 선언된 지역변수 c의 값을 인스턴스 변수 color에 저장한다. 이 때 변수 color와 c는 이름만으로도 서로 구별되므로 아무런 문제가 없다.

하지만, 오른쪽 코드에서처럼 생성자의 매개변수로 선언된 변수의 이름이 color로 인스턴스 변수 color와 같을 경우에는 이름만으로는 두 변수가 서로 구별이 안 된다. 이런 경우에는 인스턴스 변수 앞에 ‘this’를 사용하면 된다.

이렇게 하면 this.color는 인스턴스 변수이고, color는 생성자의 매개변수로 정의된 지역변수로 서로 구별이 가능하다. 만일 오른쪽 코드에서 ‘this.color = color’ 대신 ‘color = color’와 같이 하면 둘 다 지역변수로 간주된다.

이처럼 생성자의 매개변수로 인스턴스 변수들의 초기값을 제공받는 경우가 많기 때문에 매개변수와 인스턴스 변수의 이름이 일치하는 경우가 자주 있다. 이때는 왼쪽 코드와 같이 매개변수 이름을 다르게 하는 것보다 ‘this’를 사용해서 구별되도록 하는 것이 의미가 더 명확하고 이해하기 쉽다.

‘this’는 참조변수로 인스턴스 자신을 가리킨다. 참조변수를 통해 인스턴스의 멤버에 접근할 수 있는 것처럼, ‘this’로 인스턴스 변수에 접근할 수 있는 것이다.

하지만, ‘this’를 사용할 수 있는 것은 인스턴스 멤버뿐이다. static메서드(클래스 메서드)에서는 인스턴스 멤버들을 사용할 수 없는 것처럼, ‘this’ 역시 사용할 수 없다. 왜냐하면, static메서드는 인스턴스를 생성하지 않고도 호출될 수 있으므로 static메서드가 호출된 시점에 인스턴스가 존재하지 않을 수도 있기 때문이다.

사실 생성자를 포함한 모든 인스턴스메서드에는 자신이 관련된 인스턴스를 가리키는 참조변수 ‘this’가 지역변수로 숨겨진 채로 존재한다.

일반적으로 인스턴스메서드는 특정 인스턴스와 관련된 작업을 하기 때문에 자신과 관련된 인스턴스의 정보가 필요하지만, static메서드는 인스턴스와 관련 없는 작업을 하기 때문에 인스턴스에 대한 정보가 필요 없기 때문이다. 인스턴스메서드와 static메서드의 차이를 다시 한 번 되새겨 보도록 하자.

this 인스턴스 자신을 가리키는 참조변수, 인스턴스의 주소가 저장되어 있다.

모든 인스턴스메서드에 지역변수로 숨겨진 채로 존재한다.

this(), this(매개변수) 생성자, 같은 클래스의 다른 생성자를 호출할 때 사용한다.

| 참고 | **this와 this()**는 비슷하게 생겼을 뿐 완전히 다른 것이다. **this**는 '참조 변수'이고, **this()**는 '생성자'이다.

5.5 생성자를 이용한 인스턴스의 복사

현재 사용하고 있는 인스턴스와 같은 상태를 갖는 인스턴스를 하나 더 만들고자 할 때 생성자를 이용할 수 있다. 두 인스턴스가 같은 상태를 갖는다는 것은 두 인스턴스의 모든 인스턴스 변수(상태)가 동일한 값을 갖고 있다는 것을 뜻한다.

하나의 클래스로부터 생성된 모든 인스턴스의 메서드와 클래스변수는 서로 동일하기 때문에 인스턴스간의 차이는, 인스턴스마다 각기 다른 값을 가질 수 있는 인스턴스변수 뿐이다.

```
Car(Car c) {
    color      = c.color;
    gearType  = c.gearType;
    door       = c.door;
}
```

위의 코드는 Car클래스의 참조변수를 매개변수로 선언한 생성자이다. 매개변수로 넘겨진 참조변수가 가리키는 Car인스턴스의 인스턴스변수인 color, gearType, door의 값을 인스턴스 자신으로 복사하는 것이다.

어떤 인스턴스의 상태를 전혀 알지 못해도 똑같은 상태의 인스턴스를 추가로 생성할 수 있다. Java API의 많은 클래스들이 인스턴스의 복사를 위한 생성자를 정의해놓고 있으니 참고하기 바란다.

| 참고 | Object클래스에 정의된 clone메서드를 이용하면 간단히 인스턴스를 복사할 수 있다. p.456

▼ 예제 6-26/ch6/CarTest3.java

```
class Car {
    String color;        // 색상
    String gearType;    // 변속기 종류 - auto(자동), manual(수동)
    int door;           // 문의 개수

    Car() {
        this("white", "auto", 4);
    }

    Car(Car c) {    // 인스턴스의 복사를 위한 생성자.
        color      = c.color;
        gearType  = c.gearType;
        door       = c.door;
    }
}
```

```

Car(String color, String gearType, int door) {
    this.color      = color;
    this.gearType   = gearType;
    this.door       = door;
}
}

class CarTest3 {
    public static void main(String[] args) {
        Car c1 = new Car();
        Car c2 = new Car(c1); // c1의 복사본 c2를 생성한다.
        System.out.println("c1의 color=" + c1.color + ", gearType="
                           + c1.gearType + ", door=" + c1.door);
        System.out.println("c2의 color=" + c2.color + ", gearType="
                           + c2.gearType + ", door=" + c2.door);
        c1.door=100; // c1의 인스턴스변수 door의 값을 변경한다.
        System.out.println("c1.door=100; 수행 후");
        System.out.println("c1의 color=" + c1.color + ", gearType="
                           + c1.gearType + ", door=" + c1.door);
        System.out.println("c2의 color=" + c2.color + ", gearType="
                           + c2.gearType + ", door=" + c2.door);
    }
}

```

▼ 실행결과

```

c1의 color=white, gearType=auto, door=4
c2의 color=white, gearType=auto, door=4
c1.door=100; 수행 후
c1의 color=white, gearType=auto, door=100
c2의 color=white, gearType=auto, door=4

```

인스턴스 c2는 c1을 복사하여 생성된 것이므로 서로 같은 상태를 갖지만, 서로 독립적으로 메모리공간에 존재하는 별도의 인스턴스이므로 c1의 값들이 변경되어도 c2는 영향을 받지 않는다.

생성자 'Car(Car c)'는 아래와 같이 다른 생성자인 'Car(String color, String gearType, int door)'를 호출하는 것이 바람직하다. 무작정 새로 코드를 작성하는 것보다 기존의 코드를 활용할 수 없는지 고민해야한다.

<pre> Car(Car c) { color = c.color; gearType = c.gearType; → door = c.door; } </pre>	<pre> Car(Car c) { // Car(String color, String gearType, int door) this(c.color, c.gearType, c.door); } </pre>
---	--

지금까지 생성자에 대해서 모르고도 자바프로그래밍이 가능했던 것을 생각한다면, 생성자는 그리 중요하지 않은 것으로 생각될지도 모른다. 하지만, 지금까지 본 것처럼 생성자를 잘 활용하면 보다 간결하고 직관적인, 객체지향적인 코드를 작성할 수 있을 것이다.

인스턴스를 생성할 때는 다음의 2가지 사항을 결정해야한다.

1. 클래스 – 어떤 클래스의 인스턴스를 생성할 것인가?
2. 생성자 – 선택한 클래스의 어떤 생성자로 인스턴스를 생성할 것인가?

6 변수의 초기화

6.1 변수의 초기화

변수를 선언하고 처음으로 값을 저장하는 것을 ‘변수의 초기화’라고 한다. 변수의 초기화는 경우에 따라서 필수적이기도 하고 선택적이기도 하지만, 가능하면 선언과 동시에 적절한 값으로 초기화 하는 것이 바람직하다.

멤버변수는 초기화를 하지 않아도 자동적으로 변수의 자료형에 맞는 기본값으로 초기화가 이루어지므로 초기화하지 않고 사용해도 되지만, 지역변수는 사용하기 전에 반드시 초기화해야 한다.

```
class InitTest {
    int x;           // 인스턴스변수
    int y = x;       // 인스턴스변수

    void method1() {
        int i;         // 지역변수
        int j = i;     // 에러. 지역변수를 초기화하지 않고 사용
    }
}
```

위의 코드에서 x, y는 인스턴스 변수이고, i, j는 지역변수이다. 그 중 x와 i는 선언만 하고 초기화를 하지 않았다. 그리고 y를 초기화 하는데 x를 사용하였고, j를 초기화 하는데 i를 사용하였다.

인스턴스 변수 x는 초기화를 해주지 않아도 자동적으로 int형의 기본값인 0으로 초기화되므로, ‘int y = x;’와 같이 할 수 있다. x의 값이 0이므로 y역시 0이 저장된다.

하지만, method1()의 지역변수 i는 자동적으로 초기화되지 않으므로, 초기화 되지 않은 상태에서 변수 j를 초기화 하는데 사용될 수 없다. 컴파일하면, 에러가 발생한다.

멤버변수(클래스변수와 인스턴스변수)와 배열의 초기화는 선택적이지만,
지역변수의 초기화는 필수적이다.

참고로 각 타입의 기본값(default value)은 다음과 같다.

자료형	기본값
boolean	false
char	'\u0000'
byte, short, int	0
long	0L
float	0.0f
double	0.0d 또는 0.0
참조형 변수	null

변수의 초기화에 대한 예를 몇 가지 더 살펴보자.

선언예	설명
int i=10; int j=10;	int형 변수 i를 선언하고 10으로 초기화 한다. int형 변수 j를 선언하고 10으로 초기화 한다.
int i=10, j=10;	같은 타입의 변수는 콤마(,)를 사용해서 함께 선언하거나 초기화 할 수 있다.
int i=10, long j=0;	에러. 타입이 다른 변수는 함께 선언하거나 초기화할 수 없다.
int i=10; int j=i;	변수 i에 저장된 값으로 변수 j를 초기화 한다. 변수 j는 i의 값인 10으로 초기화 된다.
int j=i; int i=10;	에러. 변수 i가 선언되기 전에 i를 사용할 수 없다.

▲ 표 6-4 다양한 초기화 방법

멤버변수의 초기화는 지역변수와 달리 여러 가지 방법이 있는데 앞으로 멤버변수의 초기화에 대한 모든 방법에 대해 비교, 정리할 것이다.

▶ 멤버변수의 초기화 방법

1. 명시적 초기화(explicit initialization)
2. 생성자(constructor)
3. 초기화 블럭(initialization block)
 - 인스턴스 초기화 블럭 : 인스턴스변수를 초기화하는데 사용.
 - 클래스 초기화 블럭 : 클래스변수를 초기화하는데 사용.

6.2 명시적 초기화(explicit initialization)

변수를 선언과 동시에 초기화하는 것을 명시적 초기화라고 한다. 가장 기본적이면서도 간단한 초기화 방법이므로 여러 초기화 방법 중에서 가장 우선적으로 고려되어야 한다.

```
class Car {
    int door = 4; // 기본형(primitive type) 변수의 초기화
    Engine e = new Engine(); // 참조형(reference type) 변수의 초기화

    ...
}
```

명시적 초기화가 간단하고 명료하긴 하지만, 보다 복잡한 초기화 작업이 필요할 때는 ‘초기화 블럭(initialization block)’ 또는 생성자를 사용해야 한다.

6.3 초기화 블럭(initialization block)

초기화 블럭에는 ‘클래스 초기화 블럭’과 ‘인스턴스 초기화 블럭’ 두 가지 종류가 있다. 클래스 초기화 블럭은 클래스변수의 초기화에 사용되고, 인스턴스 초기화 블럭은 인스턴스 변수의 초기화에 사용된다.

클래스 초기화 블럭 클래스변수의 복잡한 초기화에 사용된다.

인스턴스 초기화 블럭 인스턴스변수의 복잡한 초기화에 사용된다.

초기화 블럭을 작성하려면, 인스턴스 초기화 블럭은 단순히 클래스 내에 블럭{}만들고 그 안에 코드를 작성하기만 하면 된다. 그리고 클래스 초기화 블럭은 인스턴스 초기화 블럭 앞에 단순히 static을 덧붙이기만 하면 된다.

초기화 블럭 내에는 메서드 내에서와 같이 조건문, 반복문, 예외처리구문 등을 자유롭게 사용할 수 있으므로, 초기화 작업이 복잡하여 명시적 초기화만으로는 부족한 경우 초기화 블럭을 사용한다.

```
class InitBlock {
    static { /* 클래스 초기화블럭 입니다. */ }

    { /* 인스턴스 초기화블럭 입니다. */ }

    // ...
}
```

클래스 초기화 블럭은 클래스가 메모리에 처음 로딩될 때 한번만 수행되며, 인스턴스 초기화 블럭은 생성자와 같이 인스턴스를 생성할 때마다 수행된다.

그리고 생성자 보다 인스턴스 초기화 블럭이 먼저 수행된다는 사실도 기억해두자.

| 참고 | 클래스가 처음 로딩될 때 클래스변수들이 자동적으로 메모리에 만들어지고, 곧바로 클래스 초기화블럭이 클래스변수들을 초기화하게 되는 것이다.

인스턴스 변수의 초기화는 주로 생성자를 사용하고, 인스턴스 초기화 블럭은 모든 생성자에서 공통으로 수행돼야 하는 코드를 넣는데 사용한다.

```
Car() {
    count++;
    serialNo = count;
    color = "White";
    gearType = "Auto";
}

Car(String color, String gearType) {
    count++;
    serialNo = count;
    this.color = color;
    this.gearType = gearType;
}
```

같은 코드가 중복되었다.

예를 들면, 위와 같이 클래스의 모든 생성자에 공통으로 수행되어야 하는 문장들이 있을 때, 이 문장들을 각 생성자마다 써주기 보다는 아래와 같이 인스턴스 블럭에 넣어주면 코드가 보다 간결해진다.

```
{
    count++;
    serialNo = count;
}

Car() {
    color = "White";
    gearType = "Auto";
}

Car(String color, String gearType) {
    this.color = color;
    this.gearType = gearType;
}
```

이처럼 코드의 중복을 제거하는 것은 코드의 신뢰성을 높여 주고, 오류의 발생 가능성을 줄여 준다는 장점이 있다. 즉, 재사용성을 높이고 중복을 제거하는 것, 이것이 바로 객체지향 프로그래밍이 추구하는 궁극적인 목표이다.

프로그래머는 이와 같은 객체지향언어의 요소들을 잘 이해하고 활용하여 코드의 중복을 최대한 제거하기 위해서 노력해야 한다.

▼ 예제 6-27/ch6/BlockTest.java

```
class BlockTest {
    static {
        System.out.println("static { }");
    }

    {
        System.out.println("{ }");
    }

    public BlockTest() {
        System.out.println("생성자");
    }

    public static void main(String args[]) {
        System.out.println("BlockTest bt = new BlockTest(); ");
        BlockTest bt = new BlockTest();

        System.out.println("BlockTest bt2 = new BlockTest(); ");
        BlockTest bt2 = new BlockTest();
    }
}
```

▼ 실행결과

```
static { }
BlockTest bt = new BlockTest();
{
}
생성자
BlockTest bt2 = new BlockTest();
{
}
생성자
```

예제가 실행되면서 BlockTest가 메모리에 로딩될 때, 클래스 초기화 블럭이 가장 먼저 수행되어 'static { }'이 화면에 출력된다. 그 다음에 main메서드가 수행되어 BlockTest인스턴스가 생성되면서 인스턴스 초기화 블럭이 먼저 수행되고, 끝으로 생성자가 수행된다.

위의 실행결과에서도 알 수 있듯이 클래스 초기화 블럭은 처음 메모리에 로딩될 때 한번만 수행되었지만, 인스턴스 초기화 블럭은 인스턴스가 생성될 때마다 수행되었다.

▼ 예제 6-28/ch6/StaticBlockTest.java

```
class StaticBlockTest {
    static int[] arr = new int[10];

    static {
        for(int i=0;i<arr.length;i++) {
            // 1과 10사이의 임의의 값을 배열 arr에 저장한다.
            arr[i] = (int)(Math.random()*10) + 1;
        }
    }

    public static void main(String args[]) {
        for(int i=0; i<arr.length;i++)
            System.out.println("arr["+i+"] :" + arr[i]);
    }
}
```

▼ 실행결과	
arr[0]	: 4
arr[1]	: 8
arr[2]	: 7
arr[3]	: 2
arr[4]	: 2
arr[5]	: 10
arr[6]	: 7
arr[7]	: 10
arr[8]	: 1
arr[9]	: 7

명시적 초기화를 통해 배열 arr을 생성하고, 클래스 초기화 블럭을 이용해서 배열의 각 요소들을 random()을 사용해서 임의의 값으로 채우도록 했다.

이처럼 배열이나 예외처리가 필요한 초기화에서는 명시적 초기화만으로는 복잡한 초기화 작업을 할 수 없다. 이런 경우에 추가적으로 클래스 초기화 블럭을 사용하도록 한다.

| 참고 | 인스턴스변수의 복잡한 초기화는 생성자 또는 인스턴스 초기화 블럭을 사용한다.

6.4 멤버변수의 초기화 시기와 순서

지금까지 멤버변수를 초기화하는 방법에 대해서 알아봤다. 이제는 초기화가 수행되는 시기와 순서에 대해서 정리해보도록 하자.

클래스변수의 초기화시점 클래스가 처음 로딩될 때 단 한번 초기화 된다.

인스턴스변수의 초기화시점 인스턴스가 생성될 때마다 각 인스턴스별로 초기화가 이루어진다.

클래스변수의 초기화순서 기본값 → 명시적초기화 → 클래스 초기화 블럭

인스턴스변수의 초기화순서 기본값 → 명시적초기화 → 인스턴스 초기화 블럭 → 생성자

프로그램 실행도중 클래스에 대한 정보가 요구될 때, 클래스는 메모리에 로딩된다. 예를 들면, 클래스 멤버를 사용했을 때, 인스턴스를 생성할 때 등이 이에 해당한다.

하지만, 해당 클래스가 이미 메모리에 로딩되어 있다면, 또다시 로딩하지 않는다. 물론 초기화도 다시 수행되지 않는다.

| 참고 | 클래스의 로딩 시기는 JVM의 종류에 따라 좀 다를 수 있는데, 클래스가 필요할 때 바로 메모리에 로딩하도록 설계가 되어있는 것도 있고, 실행효율을 높이기 위해서 사용될 클래스들을 프로그램이 시작될 때 미리 로딩하도록 되어있는 것도 있다.

```
class InitTest {
    static int cv = 1;           } └─ 명시적 초기화  

    int iv = 1;                  └─ (explicit initialization)  

  
    static {          cv = 2;      }      // 클래스 초기화 블럭  

    {          iv = 2;      }      // 인스턴스 초기화 블럭  

  
    InitTest () {  
        iv = 3;  
    }  
}
```

| 플래시동영상 | Initialization.exe에 초기화 과정에 대한 보다 자세한 설명이 있다.

위의 InitTest클래스는 클래스변수(cv)와 인스턴스변수(iv)를 각각 하나씩 가지고 있다. ‘new InitTest();’와 같이 하여 인스턴스를 생성했을 때, cv와 iv가 초기화되어가는 과정을 단계별로 자세히 살펴보도록 하자.

클래스 초기화			인스턴스 초기화				
기본값	명시적 초기화	클래스 초기화블럭	기본값	명시적 초기화	인스턴스 초기화블럭	생성자	
cv <input type="text" value="0"/>	cv <input type="text" value="1"/>	cv <input type="text" value="2"/>	cv <input type="text" value="2"/> iv <input type="text" value="0"/>	cv <input type="text" value="2"/> iv <input type="text" value="1"/>	cv <input type="text" value="2"/> iv <input type="text" value="2"/>	cv <input type="text" value="2"/>	iv <input type="text" value="3"/>
1	2	3	4	5	6	7	

▶ 클래스변수 초기화 (1~3) : 클래스가 처음 메모리에 로딩될 때 차례대로 수행됨.

▶ 인스턴스변수 초기화(4~7) : 인스턴스를 생성할 때 차례대로 수행됨

| 중요 | 클래스변수는 항상 인스턴스변수보다 항상 먼저 생성되고 초기화 된다.

1. cv가 메모리(method area)에 생성되고, cv에는 int형의 기본값인 0이 cv에 저장된다.
2. 그 다음에는 명시적 초기화(int cv=1)에 의해서 cv에 1이 저장된다.
3. 마지막으로 클래스 초기화 블럭(cv=2)이 수행되어 cv에는 2가 저장된다.
4. InitTest클래스의 인스턴스가 생성되면서 iv가 메모리(heap)에 존재하게 된다.
iv 역시 int형 변수이므로 기본값 0이 저장된다.
5. 명시적 초기화에 의해서 iv에 1이 저장되고
6. 인스턴스 초기화 블럭이 수행되어 iv에 2가 저장된다.
7. 마지막으로 생성자가 수행되어 iv에는 3이 저장된다.

▼ 예제 6-29/ch6/ProductTest.java

```
class Product {  
    static int count = 0; // 생성된 인스턴스의 수를 저장하기 위한 변수  
    int serialNo; // 인스턴스 고유의 번호  
  
    {  
        ++count;  
        serialNo = count; } } // Product인스턴스가 생성될 때마다 count의  
// 값을 1씩 증가시켜서 serialNo에 저장한다.  
public Product() {} // 기본생성자, 생략가능  
  
class ProductTest {  
    public static void main(String args[]) {  
        Product p1 = new Product();  
        Product p2 = new Product();  
        Product p3 = new Product();  
  
        System.out.println("p1의 제품번호(serial no)는 " + p1.serialNo);  
        System.out.println("p2의 제품번호(serial no)는 " + p2.serialNo);  
        System.out.println("p3의 제품번호(serial no)는 " + p3.serialNo);  
        System.out.println("생산된 제품의 수는 모두 "+Product.count+"개 입니다.");  
    }  
}
```

▼ 실행결과

```
p1의 제품번호(serial no)는 1  
p2의 제품번호(serial no)는 2  
p3의 제품번호(serial no)는 3  
생산된 제품의 수는 모두 3개 입니다.
```

공장에서 제품을 생산할 때 제품마다 생산일련번호(serial no)를 부여하는 것과 같이 Product클래스의 인스턴스가 고유의 일련번호(serialNo)를 갖도록 하였다.

Product클래스의 인스턴스를 생성할 때마다 인스턴스 블럭이 수행되어, 클래스변수 count의 값을 1증가시킨 다음, 그 값을 인스턴스변수 serialNo에 저장한다.

이렇게 함으로써 새로 생성되는 인스턴스는 이전에 생성된 인스턴스보다 1이 증가된 serialNo값을 갖게 된다.

생성자가 하나 밖에 없기 때문에 인스턴스 블럭 대신, Product클래스의 생성자를 사용해도 결과는 같지만, 코드의 의미상 모든 생성자에서 공통으로 수행되어야하는 내용이기 때문에 인스턴스 블럭을 사용하였다.

만일 count를 인스턴스 변수로 선언했다면, 인스턴스가 생성될 때마다 0으로 초기화 될 것이므로 모든 Product인스턴스의 serialNo값은 항상 1이 될 것이다.

▼ 예제 6-30/ch6/DocumentTest.java

```

class Document {
    static int count = 0;
    String name;           // 문서명(Document name)

    Document() {          // 문서 생성 시 문서명을 지정하지 않았을 때
        this("제목없음" + ++count);
    }

    Document(String name) {
        this.name = name;
        System.out.println("문서 " + this.name + "가 생성되었습니다.");
    }
}

class DocumentTest {
    public static void main(String args[]) {
        Document d1 = new Document();
        Document d2 = new Document("자바.txt");
        Document d3 = new Document();
        Document d4 = new Document();
    }
}

```

▼ 실행결과

문서 제목없음1가 생성되었습니다.

문서 자바.txt가 생성되었습니다.

문서 제목없음2가 생성되었습니다.

문서 제목없음3가 생성되었습니다.

바로 이전의 일련번호 예제를 응용한 것으로, 워드프로세서나 문서편집기에 이와 유사한 코드가 사용된다. 문서(Document)를 생성할 때, 문서의 이름을 지정하면 그 이름의 문서가 생성되지만, 문서의 이름을 지정하지 않으면 프로그램이 일정한 규칙을 적용해서 자동으로 이름을 결정한다.

예를 들면, ‘제목없음1’, ‘제목없음2’, ‘제목없음3’ ... 과 같은 식으로 문서의 이름이 결정된다. 문서의 이름은 서로 구별될 수 있어야 하기 때문이다.

| 참고 | 연습문제는 코드초보스터디 (<http://cafe.naver.com/javachobostudy.cafe>)에서 PDF파일로 제공

Memo

Java Programming Language

Chapter 07

객체지향 프로그래밍 II

Object-oriented Programming II

1. 상속(inheritance)

1.1 상속의 정의와 장점

상속이란, 기존의 클래스를 재사용하여 새로운 클래스를 작성하는 것이다. 상속을 통해서 클래스를 작성하면 보다 적은 양의 코드로 새로운 클래스를 작성할 수 있고 코드를 공통적으로 관리할 수 있기 때문에 코드의 추가 및 변경이 매우 용이하다.

이러한 특징은 코드의 재사용성을 높이고 코드의 중복을 제거하여 프로그램의 생산성과 유지보수에 크게 기여한다.

자바에서 상속을 구현하는 방법은 아주 간단하다. 새로 작성하고자 하는 클래스의 이름 뒤에 상속받고자 하는 클래스의 이름을 키워드 ‘extends’와 함께 써 주기만 하면 된다.

예를 들어 새로 작성하려는 클래스의 이름이 Child이고 상속받고자 하는 기존 클래스의 이름이 Parent라면 다음과 같이 하면 된다.

```
class Child extends Parent {  
    // ...  
}
```

이 두 클래스는 서로 상속 관계에 있다고 하며, 상속해주는 클래스를 ‘조상 클래스’라 하고 상속 받는 클래스를 ‘자손 클래스’라 한다.

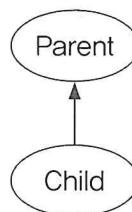
| 참고 | 서로 상속관계에 있는 두 클래스를 아래와 같은 용어를 사용해서 표현하기도 한다.

조상 클래스 부모(parent)클래스, 상위(super)클래스, 기반(base)클래스

자손 클래스 자식(child)클래스, 하위(sub)클래스, 파생된(derived) 클래스

아래와 같이 서로 상속관계에 있는 두 클래스를 그림으로 표현하면 다음과 같다.

```
class Parent { }  
class Child extends Parent { }
```

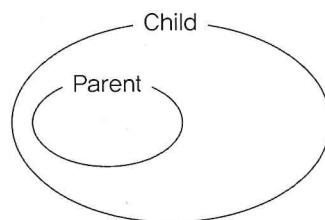


▲ 그림 7-1 클래스 Parent와 Child의 상속 관계도

클래스는 타원으로 표현했고 클래스간의 상속 관계는 화살표로 표시했다. 이와 같이 클래스 간의 상속관계를 그림으로 표현한 것을 상속계층도(class hierarchy)라고 한다.

프로그램이 커질수록 클래스간의 관계가 복잡해지는데, 이 때 그림7-1과 같이 그림으로 표현하면 클래스간의 관계를 보다 쉽게 이해할 수 있다.

자손 클래스는 조상 클래스의 모든 멤버를 상속받기 때문에, Child클래스는 Parent클래스의 멤버들을 포함한다고 할 수 있다. 클래스는 멤버들의 집합이므로 클래스 Parent와 Child의 관계를 다음과 같이 표현할 수도 있다.



▲ 그림7-2 클래스 Parent와 Child의 다이어그램

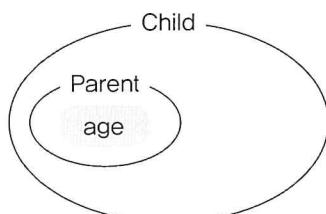
만일 Parent클래스에 age라는 정수형 변수를 멤버변수로 추가하면, 자손 클래스는 조상의 멤버를 모두 상속받기 때문에, Child클래스는 자동적으로 age라는 멤버변수가 추가된 것과 같은 효과를 얻는다.

```

class Parent {
    int age;
}

class Child extends Parent { }

```

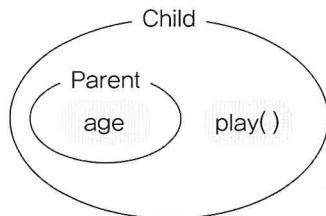


클래스	클래스의 멤버
Parent	age
Child	age

이번엔 반대로 자손인 Child클래스에 새로운 멤버로 play() 메서드를 추가해보자.

```
class Parent {
    int age;
}

class Child extends Parent {
    void play() {
        System.out.println("놀자~");
    }
}
```



클래스	클래스의 멤버
Parent	age
Child	age, play()

Child클래스에 새로운 코드가 추가되어도 조상인 Parent클래스는 아무런 영향도 받지 않는다. 여기서 알 수 있는 것처럼, 조상 클래스가 변경되면 자손 클래스는 자동적으로 영향을 받게 되지만, 자손 클래스가 변경되는 것은 조상 클래스에 아무런 영향을 주지 못한다.

자손 클래스는 조상 클래스의 모든 멤버를 상속 받으므로 항상 조상 클래스보다 같거나 많은 멤버를 갖는다. 즉, 상속에 상속을 거듭할수록 상속받는 클래스의 멤버 개수는 점점 늘어나게 된다.

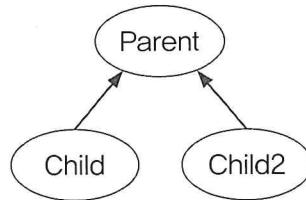
그래서 상속을 받는다는 것은 조상 클래스를 확장(extend)한다는 의미로 해석할 수도 있으며 이것이 상속에 사용되는 키워드가 ‘extends’인 이유이기도 하다.

- 생성자와 초기화 블럭은 상속되지 않는다. 멤버만 상속된다.
- 자손 클래스의 멤버 개수는 조상 클래스보다 항상 같거나 많다.

| 참고 | 접근 제어자(access modifier)가 private 또는 default인 멤버들은 상속되지 않는다기보다 상속은 받지만 자손 클래스로부터의 접근이 제한되는 것이다.

이번엔 Parent클래스로부터 상속받는 Child2클래스를 새로 작성해보자. Child2클래스를 포함한 세 클래스간의 상속계층도는 다음과 같다.

```
class Parent { }
class Child extends Parent { }
class Child2 extends Parent { }
```



클래스 Child와 Child2가 모두 Parent클래스를 상속받고 있으므로 Parent클래스와 Child클래스, 그리고 Parent클래스와 Child2클래스는 서로 상속관계에 있지만 클래스 Child와 Child2간에는 서로 아무런 관계도 성립되지 않는다. 클래스 간의 관계에서 형제 관계와 같은 것은 없다. 부모와 자식의 관계(상속관계)만이 존재할 뿐이다.

만일 Child클래스와 Child2클래스에 공통적으로 추가되어야 하는 멤버(멤버변수나 메서드)가 있다면, 이 두 클래스에 각각 따로 추가해주는 것보다는 이들의 공통조상인 Parent 클래스에 추가하는 것이 좋다.

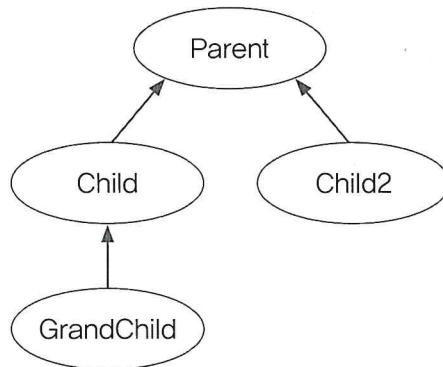
Parent클래스의 자손인 Child클래스와 Child2클래스는 조상의 멤버를 상속받기 때문에, Parent클래스에 새로운 멤버를 추가해주는 것은 Child클래스와 Child2클래스에 새로운 멤버를 추가해주는 것과 같은 효과를 얻는다.

이제는 Parent클래스 하나만 변경하면 되므로 작업이 간단해진다. 이보다 더 중요한 사실은 같은 내용의 코드를 한 곳에서 관리함으로써 코드의 중복이 줄어든다는 것이다. 코드의 중복이 많아지면 유지보수가 어려워지고 일관성을 유지하기 어렵다.

이처럼 같은 내용의 코드를 하나 이상의 클래스에 중복적으로 추가해야하는 경우에는 상속관계를 이용해서 코드의 중복을 최소화해야한다. 프로그램이 어떤 때는 잘 동작하지만 어떤 때는 오동작을 하는 이유는 중복된 코드 중에서 바르게 변경되지 않은 곳이 있기 때문이다.

여기에 또다시 Child클래스로부터 상속받는 GrandChild라는 새로운 클래스를 추가한다면 상속계층도는 다음과 같을 것이다.

```
class Parent { }
class Child extends Parent { }
class Child2 extends Parent { }
class GrandChild extends Child { }
```



자손 클래스는 조상 클래스의 모든 멤버를 물려받으므로 GrandChild클래스는 Child클래스의 모든 멤버, Child클래스의 조상인 Parent클래스로부터 상속받은 멤버까지 상속받게 된다. 그래서 GrandChild클래스는 Child클래스의 자손이면서 Parent클래스의 자손이기도 하다. 좀 더 정확히 말하자면, Child클래스는 GrandChild클래스의 직접 조상이고, Parent클래스는 GrandChild클래스의 간접 조상이 된다. 그래서 GrandChild클래스는 Parent클래스와 간접적인 상속관계에 있다고 할 수 있다.

이제 Parent클래스에 전과 같이 정수형 변수인 age를 멤버변수로 추가해 보자.

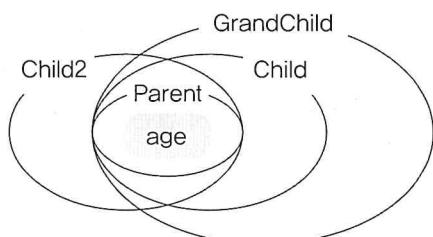
```

class Parent {
    int age;
}

class Child extends Parent { }

class Child2 extends Parent { }

class GrandChild extends Child { }
  
```



클래스	클래스의 멤버
Parent	age
Child	age
Child2	age
GrandChild	age

Parent클래스는 클래스 Child, Child2, GrandChild의 조상이므로 Parent클래스에 추가된 멤버변수 age는 Parent클래스의 모든 자손에 추가된다. 반대로 Parent클래스에서 멤버변수 age를 제거 한다면, Parent의 자손클래스인 Child, Child2, GrandChild에서도 제거된다.

이처럼 조상 클래스만 변경해도 모든 자손 클래스에, 자손의 자손 클래스에까지 영향을 미치기 때문에, 클래스간의 상속관계를 맺어 주면 자손 클래스들의 공통적인 부분은 조상 클래스에서 관리하고 자손 클래스는 자신에 정의된 멤버들만 관리하면 되므로 각 클래스의 코드가 적어져서 관리가 쉬워진다.

전체 프로그램을 구성하는 클래스들을 면밀히 설계 분석하여, 클래스간의 상속관계를 적절히 맺어 주는 것이 객체지향 프로그래밍에서 가장 중요한 부분이다.

▼ 예제 7-1/ch7/CaptionTvTest.java

```
class Tv {
    boolean power; // 전원상태(on/off)
    int channel; // 채널

    void power() { power = !power; }
    void channelUp() { ++channel; }
    void channelDown() { --channel; }
}

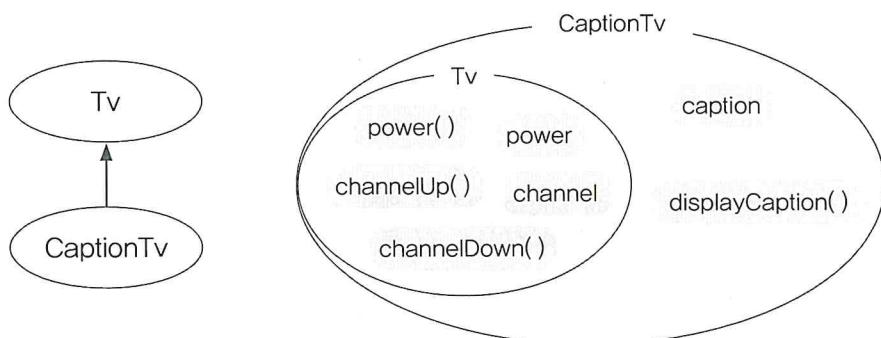
class CaptionTv extends Tv {
    boolean caption; // 캡션상태(on/off)
    void displayCaption(String text) {
        if (caption) { // 캡션 상태가 on(true) 일 때만 text를 보여 준다.
            System.out.println(text);
        }
    }
}

class CaptionTvTest {
    public static void main(String args[]) {
        CaptionTv ctv = new CaptionTv();
        ctv.channel = 10; // 조상 클래스로부터 상속받은 멤버
        ctv.channelUp(); // 조상 클래스로부터 상속받은 멤버
        System.out.println(ctv.channel);
        ctv.displayCaption("Hello, World");
        ctv.caption = true; // 캡션(자막) 기능을 켠다.
        ctv.displayCaption("Hello, World");
    }
}
```

▼ 실행결과
11
Hello, World

Tv클래스로부터 상속받고 기능을 추가하여 CaptionTv클래스를 작성하였다. 멤버변수 caption은 캡션기능의 상태를 저장하기 위한 boolean형 변수이고, displayCaption(String text)은 매개변수로 넘겨받은 문자열(text)을 캡션이 켜져 있는 경우(caption의 값이 true인 경우)에만 화면에 출력한다.

자손 클래스의 인스턴스를 생성하면 조상 클래스의 멤버도 함께 생성되기 때문에 따로 조상 클래스의 인스턴스를 생성하지 않고도 조상 클래스의 멤버들을 사용할 수 있다.



자손 클래스의 인스턴스를 생성하면 조상 클래스의 멤버와 자손 클래스의 멤버가 합쳐진 하나의 인스턴스로 생성된다.

1.2 클래스간의 관계 – 포함관계

지금까지 상속을 통해 클래스 간에 관계를 맺어 주고 클래스를 재사용하는 방법에 대해서 알아보았다. 상속이외에도 클래스를 재사용하는 또 다른 방법이 있는데, 그것은 클래스 간에 ‘포함(Composite)’관계를 맺어 주는 것이다. 클래스 간의 포함관계를 맺어 주는 것은 한 클래스의 멤버변수로 다른 클래스 타입의 참조변수를 선언하는 것을 뜻한다.

원(Circle)을 표현하기 위한 Circle이라는 클래스를 다음과 같이 작성하였다고 가정하자.

```
class Circle {
    int x;           // 원점의 x좌표
    int y;           // 원점의 y좌표
    int r;           // 반지름(radius)
}
```

그리고 좌표상의 한 점을 다루기 위한 Point클래스가 다음과 같이 작성되어 있다고 하자.

```
class Point {
    int x;           // x좌표
    int y;           // y좌표
}
```

Point클래스를 재사용해서 Circle클래스를 작성한다면 다음과 같이 할 수 있을 것이다.

<pre>class Circle { int x; // 원점의 x좌표 int y; // 원점의 y좌표 int r; // 반지름(radius) }</pre>		<pre>class Circle { Point c = new Point(); // 원점 int r; }</pre>
---	--	---

이와 같이 한 클래스를 작성하는 데 다른 클래스를 멤버변수로 선언하여 포함시키는 것은 좋은 생각이다. 하나의 거대한 클래스를 작성하는 것보다 단위별로 여러 개의 클래스를 작성한 다음, 이 단위 클래스들을 포함관계로 재사용하면 보다 간결하고 손쉽게 클래스를 작성할 수 있다. 또한 작성된 단위 클래스들은 다른 클래스를 작성하는데 재사용될 수 있을 것이다.

```
class Car {
    Engine e = new Engine(); // 엔진
    Door[] d = new Door[4]; // 문, 문의 개수를 넷으로 가정하고 배열로 처리했다.
    //...
}
```

위와 같은 Car클래스를 작성할 때, Car클래스의 단위구성요소인 Engine, Door와 같은 클래스를 미리 작성해 놓고 이 들을 Car클래스의 멤버변수로 선언하여 포함관계를 맺어 주면, 클래스를 작성하는 것도 쉽고 코드도 간결해서 이해하기 쉽다. 그리고 단위클래스 별로 코드가 작게 나뉘어 작성되어 있기 때문에 코드를 관리하는데도 수월하다.

1.3 클래스간의 관계 결정하기

클래스를 작성하는데 있어서 상속관계를 맺어 줄 것인지 포함관계를 맺어 줄 것인지 결정하는 것은 때때로 혼돈스러울 수 있다.

전에 예를 든 Circle클래스의 경우, Point클래스를 포함시키는 대신 상속관계를 맺어 주었다면 다음과 같을 것이다.

```
class Circle {  
    Point c = new Point();  
    int r;  
}  
class Circle extends Point {  
    int r;  
}
```

두 경우를 비교해 보면 Circle클래스를 작성하는데 있어서 Point클래스를 포함시키거나 상속받도록 하는 것은 결과적으로 별 차이가 없어 보인다.

그럴 때는 ‘~은 ~이다(is-a)’와 ‘~은 ~을 가지고 있다(has-a)’를 넣어서 문장을 만들어 보면 클래스 간의 관계가 보다 명확해 진다.

원(Circle)은 점(Point)이다. – Circle is a Point.

원(Circle)은 점(Point)을 가지고 있다. – Circle has a Point.

원은 원점(Point)과 반지름으로 구성되므로 위의 두 문장을 비교해 보면 첫 번째 문장보다 두 번째 문장이 더 옳다는 것을 알 수 있을 것이다.

이처럼 클래스를 가지고 문장을 만들었을 때 ‘~은 ~이다.’라는 문장이 성립한다면, 서로 상속관계를 맺어 주고, ‘~은 ~을 가지고 있다.’는 문장이 성립한다면 포함관계를 맺어 주면 된다. 그래서 Circle클래스와 Point클래스 간의 관계는 상속관계 보다 포함관계를 맺어 주는 것이 더 옳다.

몇 가지 더 예를 들면, Car클래스와 SportsCar클래스는 ‘SportsCar는 Car이다.’와 같이 문장을 만드는 것이 더 옳기 때문에 이 두 클래스는 Car클래스를 조상으로 하는 상속관계를 맺어 주어야 한다.

Card클래스와 Deck클래스는 ‘Deck은 Card를 가지고 있다.’와 같이 문장을 만드는 것이 더 옳기 때문에 Deck클래스에 Card클래스를 포함시켜야 한다.

| 참고 | Deck은 카드 한 벌을 뜻한다.

상속관계 ‘~은 ~이다.(is-a)’

포함관계 ‘~은 ~을 가지고 있다.(has-a)’

| 참고 | 프로그램의 모든 클래스를 분석하여 가능한 많은 관계를 맷도록 노력해서 코드의 재사용성을 높여야 한다.

▼ 예제 7-2/ch7/DrawShape.java

```
class DrawShape {
    public static void main(String[] args) {
        Point[] p = { new Point(100, 100),
                      new Point(140, 50),
                      new Point(200, 100)
                  };

        Triangle t = new Triangle(p);
        Circle c = new Circle(new Point(150, 150), 50);

        t.draw(); // 삼각형을 그린다.
        c.draw(); // 원을 그린다.
    }
}

class Shape {
    String color = "black";
    void draw() {
        System.out.printf("[color=%s]%n", color);
    }
}

class Point {
    int x;
    int y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    Point() {
        this(0,0);
    }

    String getXY() {
        return "("+x+","+y+")"; // x와 y의 값을 문자열로 반환
    }
}

class Circle extends Shape {
    Point center; // 원의 원점좌표
    int r; // 반지름

    Circle() {
        this(new Point(0, 0), 100); // Circle(Point center, int r)를 호출
    }
}
```

```

Circle(Point center, int r) {
    this.center = center;
    this.r = r;
}

void draw() { // 원을 그리는 대신에 원의 정보를 출력하도록 했다.
    System.out.printf("[center=(%d, %d), r=%d, color=%s]%n",
                      center.x, center.y, r, color);
}
}

class Triangle extends Shape {
    Point[] p = new Point[3]; •————— 3개의 Point인스턴스를 담을
                                배열을 생성한다.

    Triangle(Point[] p) {
        this.p = p;
    }

    void draw() {
        System.out.printf("[p1=%s, p2=%s, p3=%s, color=%s]%n",
                          p[0].getXY(), p[1].getXY(), p[2].getXY(), color);
    }
}

```

▼ 실행결과

```
[p1=(100,100), p2=(140,50), p3=(200,100), color=black]
[center=(150, 150), r=50, color=black]
```

도형을 의미하는 Shape클래스를 정의하고, 2차원 좌표에서의 점을 의미하는 Point클래스를 정의한 다음, 이 두 클래스를 재활용해서 Circle클래스와 Triangle클래스를 정의하였다. 앞서 배운 ‘is-a’와 ‘has-a’로 클래스간의 관계를 어떻게 맺어야하는지 확인해보자.

```

A Circle is a Shape. // 1. 원은 도형이다.
A Circle is a Point. // 2. 원은 점이다?

A Circle has a Shape. // 3. 원은 도형을 가지고 있다?
A Circle has a Point. // 4. 원은 점을 가지고 있다.

```

네 문장 중에서 첫 번째와 네 번째 문장이 자연스럽다는 것을 쉽게 알 수 있다. 클래스 간의 관계를 결정하는 것이 매번 이렇게 딱 떨어지는 건 아니지만, 적어도 클래스간의 관계를 맺어주는데 필요한 가장 기본적인 원칙에 대한 감은 잡을 수는 있을 것이다.

Circle을 Shape와 상속관계로, 그리고 Point와는 포함관계로 정의하면 다음과 같다.

```

class Circle extends Shape { // Circle과 Shape는 상속관계
    Point center;           // Circle과 Point는 포함관계
    int r;
    ...
}

```

Circle클래스는 Shape클래스로부터 모든 멤버를 상속받았으므로, Shape클래스에 정의된 color이나 draw()를 사용할 수 있다.

```
class Shape {
    String color = "black";

    void draw() {
        System.out.printf("[color=%s]%n", color);
    }
}
```

그러나 Circle클래스에도 draw()가 정의되어 있다. 그러면 둘 중에 어떤 것이 호출되는 것일까? 이미 결과를 통해 알 수 있듯이 Circle클래스의 draw()가 호출된다.

```
class Circle extends Shape {
    ...
    void draw() { // 원을 그리는 대신에 원의 정보를 출력하도록 했다.
        System.out.printf("[center=(%d, %d), r=%d, color=%s]%n",
                           center.x, center.y, r, color);
    }
}
```

이처럼 조상 클래스에 정의된 메서드와 같은 메서드를 자손 클래스에 정의하는 것을 ‘오버라이딩’이라고 하며, 곧이어 배울 내용이므로 설명은 생략한다.

그리고 한 가지 더 설명할 것은 Circle인스턴스를 생성하는 문장인데, 이 문장에 대해서도 독자들로부터 질문을 많이 받았다.

```
Circle c = new Circle(new Point(150, 150), 50);
```

위 문장이 좀 복잡해 보이지만, 아래의 두 문장을 하나로 합쳐놓은 것일 뿐이다.

```
Point p = new Point(150, 150);
Circle c = new Circle(p, 50);
```

복잡한 문장을 만났을 때는 이처럼 여러 문장으로 분해해보자. 한결 이해하기 쉬워진다.

▼ 예제 7-3/ch7/DeckTest.java

```
class DeckTest {
    public static void main(String args[]) {
        Deck d = new Deck(); // 카드 한 벌(Deck)을 만든다.
        Card c = d.pick(0); // 섞기 전에 제일 위의 카드를 뽑는다.
        System.out.println(c); // System.out.println(c.toString());과 같다.

        d.shuffle(); // 카드를 섞는다.
        c = d.pick(0); // 섞은 후에 제일 위의 카드를 뽑는다.
        System.out.println(c);
    }
}
```

```

class Deck {
    final int CARD_NUM = 52;      // 카드의 개수
    Card cardArr[] = new Card[CARD_NUM]; // Card 객체 배열을 포함

    Deck () { // Deck의 카드를 초기화한다.
        int i=0;

        for(int k=Card.KIND_MAX; k > 0; k--)
            for(int n=0; n < Card.NUM_MAX ; n++)
                cardArr[i++] = new Card(k, n+1);
    }

    Card pick(int index) { // 지정된 위치(index)에 있는 카드 하나를 꺼내서 반환
        return cardArr[index];
    }

    Card pick() { // Deck에서 카드 하나를 선택한다.
        int index = (int)(Math.random() * CARD_NUM);
        return pick(index);
    }

    void shuffle() { // 카드의 순서를 섞는다.
        for(int i=0; i < cardArr.length; i++) {
            int r = (int)(Math.random() * CARD_NUM);

            Card temp = cardArr[i];
            cardArr[i] = cardArr[r];
            cardArr[r] = temp;
        }
    }
} // Deck 클래스의 끝

class Card {
    static final int KIND_MAX = 4; // 카드 무늬의 수
    static final int NUM_MAX = 13; // 무늬별 카드 수

    static final int SPADE = 4;
    static final int DIAMOND = 3;
    static final int HEART = 2;
    static final int CLOVER = 1;
    int kind;
    int number;

    Card() {
        this(SPADE, 1);
    }

    Card(int kind, int number) {
        this.kind = kind;
        this.number = number;
    }

    public String toString() {
        String[] kinds = {"", "CLOVER", "HEART", "DIAMOND", "SPADE"};
        String numbers = "0123456789XJQK"; // 숫자 10은 X로 표현
    }
}

```

```

        return "kind : " + kinds[this.kind]
               + ", number : " + numbers.charAt(this.number);
    } // toString()의 끝
} // Card클래스의 끝

```

▼ 실행결과

```

kind : SPADE, number : 1
kind : HEART, number : 7

```

Deck클래스를 작성하는데 Card클래스를 재사용하여 포함관계로 작성하였다. 카드 한 별(Deck)은 모두 52장의 카드로 이루어져 있으므로 Card클래스를 크기가 52인 배열로 처리하였다. shuffle()은 카드 한 별의 첫 번째 카드부터 순서대로 임의로 위치에 있는 카드와 위치를 서로 바꾸는 방식으로 카드를 섞는다. random()을 사용했기 때문에 매 실행 시마다 결과가 다르게 나타날 것이다.

pick()은 Card객체 배열 cardArr에 저장된 Card객체 중에서 하나를 꺼내서 반환한다. Card객체 배열은 참조변수 배열이고, 이 배열에 실제로 저장된 것은 객체가 아니라 객체의 주소다.

아래의 문장에서 pick(0)을 호출하면, 매개변수 index의 값이 0이 되므로, cardArr[0]에 저장된 Card객체의 주소가 참조변수 c에 저장된다.

```

Card c = d.pick(0);      // pick(int index)를 호출
Card pick(int index) {
    return cardArr[index];
}

```

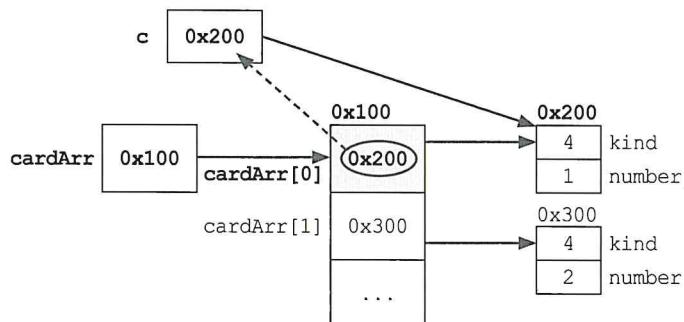
예를 들어 index의 값이 0이고, cardArr[0]의 값이 0x200이라면 pick(0)은 다음과 같은 과정으로 계산된다.

```

return cardArr[index];
→ return cardArr[0];
→ return 0x200;

```

그래서 참조변수 c에 0x200, 즉 cardArr[0]에 저장된 Card인스턴스의 주소가 저장된다.



한 가지 더 설명할 것은 Card클래스에 정의된 `toString()`인데, `toString()`은 인스턴스의 정보를 문자열로 반환할 목적으로 정의된 것이다. 아래의 코드처럼 참조변수 `c`를 출력하면, 참조변수 `c`가 가리키고 있는 인스턴스에 `toString()`을 호출하여 그 결과를 화면에 출력한다.

```
Deck d = new Deck(); // 카드 한 벌(Deck)을 만든다.
Card c = d.pick(0); // 석기 전에 제일 위의 카드를 뽑는다.
System.out.println(c); // System.out.println(c.toString()); 과 같다.

// System.out.println("Card:"+ c.toString());과 같다.
System.out.println("Card:"+ c);
```

결국 ‘`System.out.println(c)`’는 ‘`System.out.println(c.toString())`;’와 같은 결과를 얻는다.

이처럼 참조변수의 출력이나 덧셈연산자를 이용한 참조변수와 문자열의 결합에는 `toString()`이 자동적으로 호출되어 참조변수를 문자열로 대치한 후 처리한다.

`toString()`은 모든 클래스의 조상인 `Object`클래스에 정의된 것으로, 어떤 종류의 객체에 대해서도 `toString()`을 호출하는 것이 가능하다. `Object`클래스와 `toString()`에 대한 보다 자세한 것은 ‘9장 java.lang패키지와 유용한 클래스’에서 설명할 것이다.

1.4 단일 상속(single inheritance)

다른 객체지향언어인 C++에서는 여러 조상 클래스로부터 상속받는 것이 가능한 ‘다중상속(multiple inheritance)’을 허용하지만 자바에서는 오직 단일 상속만을 허용한다. 그래서 둘 이상의 클래스로부터 상속을 받을 수 없다. 예를 들어 TV클래스와 VCR클래스가 있을 때, 이 두 클래스로부터 상속을 받는 TVCR클래스를 작성할 수 없다.

그래서 TVCR클래스는 조상 클래스로 TV클래스와 VCR클래스 중 하나만 선택해야한다.

```
class TVCR extends TV, VCR { // 에러. 조상은 하나만 허용된다.
    //...
}
```

다중상속을 허용하면 여러 클래스로부터 상속받을 수 있기 때문에 복합적인 기능을 가진 클래스를 쉽게 작성할 수 있다는 장점이 있지만, 클래스간의 관계가 매우 복잡해진다는 것과 서로 다른 클래스로부터 상속받은 멤버간의 이름이 같은 경우 구별할 수 있는 방법이 없다는 단점을 가지고 있다.

만일 다중상속을 허용해서 TVCR클래스가 TV클래스와 VCR클래스를 모두 조상으로 하여 두 클래스의 멤버들을 상속받는다고 가정해 보자.

TV클래스에도 power()라는 메서드가 있고, VCR클래스에도 power()라는 메서드가 있을 때 자손인 TVCR클래스는 어느 조상클래스의 power()를 상속받게 되는 것일까?

둘 다 상속받게 된다면, TVCR클래스 내에서 선언부(이름과 매개변수)만 같고 서로 다른 내용의 두 메서드를 어떻게 구별할 것인가?

static메서드라면 메서드 이름 앞에 클래스의 이름을 붙여서 구별할 수 있다지만, 인스턴스 메서드의 경우 선언부가 같은 두 메서드를 구별할 수 있는 방법은 없다.

이것을 해결하는 방법은 조상 클래스의 메서드의 이름이나 매개변수를 바꾸는 방법 밖에 없다. 이렇게 하면 그 조상 클래스의 power()메서드를 사용하는 모든 클래스들도 변경을 해야 하므로 그리 간단한 문제가 아니다.

자바에서는 다중상속의 이러한 문제점을 해결하기 위해 다중상속의 장점을 포기하고 단일상속만을 허용한다.

단일 상속이 하나의 조상 클래스만을 가질 수 있기 때문에 다중상속에 비해 불편한 점도 있지만, 클래스 간의 관계가 보다 명확해지고 코드를 더욱 신뢰할 수 있게 만들어 준다는 점에서 다중상속보다 유리하다.

▼ 예제 7-4/ch7/TVCR.java

```
class Tv {
    boolean power;      // 전원상태(on/off)
    int channel;        // 채널

    void power()         { power = !power; }
    void channelUp()    { ++channel; }
    void channelDown()  { --channel; }
}

class VCR {
    boolean power;      // 전원상태(on/off)
    int counter = 0;
    void power() { power = !power; }
    void play() { /* 내용생략 */ }
    void stop() { /* 내용생략 */ }
    void rew() { /* 내용생략 */ }
    void ff() { /* 내용생략 */ }
}

class TVCR extends Tv {
    VCR vcr = new VCR(); ←———— VCR클래스를 포함시켜서
                           사용한다.

    void play() {
        vcr.play();
    }

    void stop() {
        vcr.stop();
    }
}
```

```

void rew() {
    vcr.rew();
}

void ff() {
    vcr.ff();
}
}

```

자바는 다중상속을 허용하지 않으므로 Tv클래스를 조상으로 하고, VCR클래스는 TVCR 클래스에 포함시켰다. 그리고 TVCR클래스에 VCR클래스의 메서드와 일치하는 선언부를 가진 메서드를 선언하고 내용은 VCR클래스의 것을 호출해서 사용하도록 했다. 외부적으로는 TVCR클래스의 인스턴스를 사용하는 것처럼 보이지만 내부적으로는 VCR클래스의 인스턴스를 생성해서 사용하는 것이다.

이렇게 함으로써 VCR클래스의 메서드의 내용이 변경되더라도 TVCR클래스의 메서드들 역시 변경된 내용이 적용되는 결과를 얻을 수 있을 것이다.

1.5 Object클래스 – 모든 클래스의 조상

Object클래스는 모든 클래스 상속계층도의 최상위에 있는 조상클래스이다. 다른 클래스로부터 상속 받지 않는 모든 클래스들은 자동적으로 Object클래스로부터 상속받게 함으로써 이것을 가능하게 한다.

만일 다음과 같이 다른 클래스로부터 상속을 받지 않는 Tv클래스를 정의하였다고 하자.

```

class Tv {
    ...
}

```

위의 코드를 컴파일 하면 컴파일러는 위의 코드를 다음과 같이 자동적으로 ‘extends Object’를 추가하여 Tv클래스가 Object클래스로부터 상속받도록 한다.

```

class Tv extends Object {
    ...
}

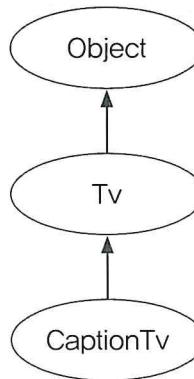
```

이렇게 함으로써 Object클래스가 모든 클래스의 조상이 되도록 한다. 만일 다른 클래스로부터 상속을 받는다고 하더라도 상속계층도를 따라 조상클래스, 조상클래스의 조상클래스를 찾아 올라가다 보면 결국 마지막 최상위 조상은 Object클래스일 것이다.

| 참고 | 이미 어떤 클래스로부터 상속받도록 작성된 클래스에 대해서는 컴파일러가 ‘extends Object’를 추가하지 않는다.

```
class Tv {  
    ...  
}  
  
class CaptionTv extends Tv {  
    ...  
}
```

위와 같이 Tv클래스가 있고, Tv클래스를 상속받는 CaptionTv가 있을 때 상속계층도는 다음과 같다.



| 참고 | 상속계층도를 단순화하기 위해서 Object클래스를 생략하는 경우가 많다.

이처럼 모든 상속계층도의 최상위에는 Object클래스가 위치한다. 그래서 자바의 모든 클래스들은 Object클래스의 멤버들을 상속 받기 때문에 Object클래스에 정의된 멤버들을 사용할 수 있다.

그동안 `toString()`이나 `equals(Object o)`와 같은 메서드를 따로 정의하지 않고도 사용 할 수 있었던 이유는 이 메서드들이 Object클래스에 정의된 것들이기 때문이다.

Object클래스에는 `toString()`, `equals()`와 같은 모든 인스턴스가 가져야 할 기본적인 11개의 메서드가 정의되어 있으며 이에 대해서는 '9장 java.lang패키지와 유용한 클래스'에서 자세히 학습하게 될 것이다.

2. 오버라이딩(overriding)

2.1 오버라이딩이란?

조상 클래스로부터 상속받은 메서드의 내용을 변경하는 것을 오버라이딩이라고 한다. 상속받은 메서드를 그대로 사용하기도 하지만, 자손 클래스 자신에 맞게 변경해야하는 경우가 많다. 이럴 때 조상의 메서드를 오버라이딩한다.

| 참고 | `override`의 사전적 의미는 ‘~위에 덮어쓰다(overwrite)’이다.

2차원 좌표계의 한 점을 표현하기 위한 `Point` 클래스가 있을 때, 이를 조상으로 하는 `Point3D` 클래스, 3차원 좌표계의 한 점을 표현하기 위한 클래스를 다음과 같이 새로 작성하였다고 하자.

```
class Point {
    int x;
    int y;

    String getLocation() {
        return "x :" + x + ", y :" + y;
    }
}

class Point3D extends Point {
    int z;

    String getLocation() {          // 오버라이딩
        return "x :" + x + ", y :" + y + ", z :" + z;
    }
}
```

`Point` 클래스의 `getLocation()`은 한 점의 x, y 좌표를 문자열로 반환하도록 작성되었다.

이 두 클래스는 서로 상속관계에 있으므로 `Point3D` 클래스는 `Point` 클래스로부터 `getLocation()`을 상속받지만, `Point3D` 클래스는 3차원 좌표계의 한 점을 표현하기 위한 것이므로 조상인 `Point` 클래스로부터 상속받은 `getLocation()`은 `Point3D` 클래스에 맞지 않는다. 그래서 이 메서드를 `Point3D` 클래스 자신에 맞게 z축의 좌표값도 포함하여 반환하도록 오버라이딩 하였다.

`Point` 클래스를 사용하던 사람들은 새로 작성된 `Point3D` 클래스가 `Point` 클래스의 자손이므로 `Point3D` 클래스의 인스턴스에 대해서 `getLocation()`을 호출하면 `Point` 클래스의 `getLocation()`이 그랬듯이 점의 좌표를 문자열로 얻을 수 있을 것이라고 기대할 것이다.

그렇기 때문에 새로운 메서드를 제공하는 것보다 오버라이딩을 하는 것이 바른 선택이다.

2.2 오버라이딩의 조건

오버라이딩은 메서드의 내용만을 새로 작성하는 것이므로 메서드의 선언부는 조상의 것과 완전히 일치해야 한다. 그래서 오버라이딩이 성립하기 위해서는 다음과 같은 조건을 만족해야 한다.

자손 클래스에서 오버라이딩하는 메서드는 조상 클래스의 메서드와

- 이름이 같아야 한다.
- 매개변수가 같아야 한다.
- 반환타입이 같아야 한다.

| 참고 | JDK1.5부터 '공변 반환타입(covariant return type)'이 추가되어, 반환타입을 자손 클래스의 타입으로 변경하는 것은 가능하도록 조건이 완화되었다. p.457

한마디로 요약하면 선언부가 서로 일치해야 한다는 것이다. 다만 접근 제어자(access modifier)와 예외(exception)는 제한된 조건 하에서만 다르게 변경할 수 있다.

1. 접근 제어자는 조상 클래스의 메서드보다 좁은 범위로 변경 할 수 없다.

만일 조상 클래스에 정의된 메서드의 접근 제어자가 protected라면, 이를 오버라이딩하는 자손 클래스의 메서드는 접근 제어자가 protected나 public이어야 한다. 대부분의 경우 같은 범위의 접근 제어자를 사용한다. 접근 제어자의 접근범위를 넓은 것에서 좁은 것 순으로 나열하면 public, protected, (default), private이다.

2. 조상 클래스의 메서드보다 많은 수의 예외를 선언할 수 없다.

아래의 코드를 보면 Child클래스의 parentMethod()에 선언된 예외의 개수가 조상인 Parent 클래스의 parentMethod()에 선언된 예외의 개수보다 적으므로 바르게 오버라이딩 되었다.

```
class Parent {  
    void parentMethod() throws IOException, SQLException {  
        ...  
    }  
}  
  
class Child extends Parent {  
    void parentMethod() throws IOException {  
        ...  
    }  
}
```

여기서 주의해야할 점은 단순히 선언된 예외의 개수의 문제가 아니라는 것이다.

```
class Child extends Parent {  
    void parentMethod() throws Exception {  
        ...  
    }  
}
```

만일 위와 같이 오버라이딩을 하였다면, 분명히 조상클래스에 정의된 메서드보다 적은 개수의 예외를 선언한 것처럼 보이지만 Exception은 모든 예외의 최고 조상이므로 가장 많은 개수의 예외를 던질 수 있도록 선언한 것이다.

그래서 예외의 개수는 적거나 같아야 한다는 조건을 만족시키지 못하는 잘못된 오버라이딩인 것이다.

조상 클래스의 메서드를 자손 클래스에서 오버라이딩할 때

- 접근 제어자를 조상 클래스의 메서드보다 좁은 범위로 변경할 수 없다.
 - 예외는 조상 클래스의 메서드보다 많이 선언할 수 없다.
 - 인스턴스메서드를 static메서드로 또는 그 반대로 변경할 수 없다.

Q. 조상 클래스에 정의된 static메서드를 자손 클래스에서 똑같은 이름의 static메서드로 정의할 수 있나요?

A. 가능합니다. 하지만, 이것은 각 클래스에 별개의 static 메서드를 정의한 것일 뿐 오버라이딩이 아니에요. 각 메서드는 클래스 이름으로 구별될 수 있으며, 호출할 때는 '참조변수.메서드이름()' 대신 '클래스이름.메서드이름()'으로 하는 것이 바람직합니다. static 멤버들은 자신들이 정의된 클래스에 묶여 있다고 생각하세요.

2.3 오버로딩 vs. 오버라이딩

오버로딩과 오버라이딩은 서로 혼동하기 쉽지만 사실 그 차이는 명백하다. 오버로딩은 기존에 없는 새로운 메서드를 추가하는 것이고, 오버라이딩은 조상으로부터 상속받은 메서드의 내용을 변경하는 것이다.

오버로딩 (overloading)	기준에 없는 새로운 메서드를 정의하는 것(new)
오버라이딩 (overriding)	상속받은 메서드의 내용을 변경하는 것(change, modify)

아래의 코드를 보고 오버로딩과 오버라이딩을 구별할 수 있어야 한다.

```
class Parent {  
    void parentMethod() {}  
}  
  
class Child extends Parent {  
    void parentMethod() {}          // 오버라이딩  
    void parentMethod(int i) {}    // 오버로딩  
  
    void childMethod() {}  
    void childMethod(int i) {}    // 오버로딩  
    void childMethod() {}
```

2.4 super

super는 자손 클래스에서 조상 클래스로부터 상속받은 멤버를 참조하는데 사용되는 참조 변수이다. 멤버변수와 지역변수의 이름이 같을 때 this를 붙여서 구별했듯이 상속받은 멤버와 자신의 멤버와 이름이 같을 때는 super를 붙여서 구별할 수 있다.

조상 클래스로부터 상속받은 멤버도 자손 클래스 자신의 멤버이므로 super 대신 this를 사용할 수 있다. 그래도 조상 클래스의 멤버와 자손클래스의 멤버가 중복 정의되어 서로 구별해야하는 경우에만 super를 사용하는 것이 좋다.

조상의 멤버와 자신의 멤버를 구별하는데 사용된다는 점을 제외하고는 super와 this는 근본적으로 같다. 모든 인스턴스메서드에는 자신이 속한 인스턴스의 주소가 지역변수로 저장되는데, 이것이 참조변수인 this와 super의 값이 된다.

static메서드(클래스메서드)는 인스턴스와 관련이 없다. 그래서 this와 마찬가지로 super 역시 static메서드에서는 사용할 수 없고 인스턴스메서드에서만 사용할 수 있다.

▼ 예제 7-5/ch7/SuperTest.java

```
class SuperTest {
    public static void main(String args[]) {
        Child c = new Child();
        c.method();
    }
}

class Parent {
    int x=10;
}

class Child extends Parent {
    void method() {
        System.out.println("x=" + x);
        System.out.println("this.x=" + this.x);
        System.out.println("super.x=" + super.x);
    }
}
```

▼ 실행결과
x=10 this.x=10 super.x=10

이 경우 x, this.x, super.x 모두 같은 변수를 의미하므로 모두 같은 값이 출력되었다.

▼ 예제 7-6/ch7/SuperTest2.java

```
class SuperTest2 {
    public static void main(String args[]) {
        Child c = new Child();
        c.method();
    }
}

class Parent {
    int x=10;
}
```

```
class Child extends Parent {
    int x=20;

    void method() {
        System.out.println("x=" + x);
        System.out.println("this.x=" + this.x);
        System.out.println("super.x=" + super.x);
    }
}
```

▼ 실행결과
x=20
this.x=20
super.x=10

이전 예제와 달리 같은 이름의 멤버변수가 조상 클래스인 Parent에도 있고 자손 클래스인 Child클래스에도 있을 때는 super.x와 this.x는 서로 다른 값을 참조하게 된다. super.x는 조상 클래스로부터 상속받은 멤버변수 x를 뜻하며, this.x는 자손 클래스에 선언된 멤버변수를 뜻한다.

이처럼 조상 클래스에 선언된 멤버변수와 같은 이름의 멤버변수를 자손 클래스에서 중복해서 정의하는 것이 가능하며 참조변수 super를 이용해서 서로 구별할 수 있다.

변수만이 아니라 메서드 역시 super를 써서 호출할 수 있다. 특히 조상 클래스의 메서드를 자손 클래스에서 오버라이딩한 경우에 super를 사용한다.

```
class Point {
    int x;
    int y;

    String getLocation() {
        return "x :" + x + ", y :" + y;
    }
}

class Point3D extends Point {
    int z;
    String getLocation() { // 오버라이딩
        // return "x :" + x + ", y :" + y + ", z :" + z;
        return super.getLocation() + ", z :" + z; // 조상의 메서드 호출
    }
}
```

getLocation()을 오버라이딩할 때 조상 클래스의 getLocation()을 호출하는 코드를 포함시켰다. 조상클래스의 메서드의 내용에 추가적으로 작업을 덧붙이는 경우라면 이처럼 super를 사용해서 조상클래스의 메서드를 포함시키는 것이 좋다. 후에 조상클래스의 메서드가 변경되더라도 변경된 내용이 자손클래스의 메서드에 자동적으로 반영될 것이기 때문이다.

2.5 super() – 조상 클래스의 생성자

this()와 마찬가지로 super() 역시 생성자이다. this()는 같은 클래스의 다른 생성자를 호출하는 데 사용되지만, super()는 조상 클래스의 생성자를 호출하는데 사용된다.

자손 클래스의 인스턴스를 생성하면, 자손의 멤버와 조상의 멤버가 모두 합쳐진 하나의 인스턴스가 생성된다. 그래서 자손 클래스의 인스턴스가 조상 클래스의 멤버들을 사용할 수 있는 것이다. 이 때 조상 클래스 멤버의 초기화 작업이 수행되어야 하기 때문에 자손 클래스의 생성자에서 조상 클래스의 생성자가 호출되어야 한다.

생성자의 첫 줄에서 조상 클래스의 생성자를 호출해야 하는 이유는 자손 클래스의 멤버가 조상 클래스의 멤버를 사용할 수도 있으므로 조상의 멤버들이 먼저 초기화되어 있어야 하기 때문이다.

이와 같은 조상 클래스 생성자의 호출은 클래스의 상속관계를 거슬러 올라가면서 계속 반복된다. 마지막으로 모든 클래스의 최고 조상인 Object 클래스의 생성자인 Object()까지 가서야 끝이 난다.

그래서 Object 클래스를 제외한 모든 클래스의 생성자는 첫 줄에 반드시 자신의 다른 생성자 또는 조상의 생성자를 호출해야 한다. 그렇지 않으면 컴파일러는 생성자의 첫 줄에 'super();'를 자동적으로 추가할 것이다.

Object 클래스를 제외한 모든 클래스의 생성자 첫 줄에 생성자, this() 또는 super(), 를 호출해야 한다. 그렇지 않으면 컴파일러가 자동적으로 'super();'를 생성자의 첫 줄에 삽입한다.

인스턴스를 생성할 때는 클래스를 선택하는 것만큼 생성자를 선택하는 것도 중요하다.

1. 클래스 – 어떤 클래스의 인스턴스를 생성할 것인가?
2. 생성자 – 선택한 클래스의 어떤 생성자를 이용해서 인스턴스를 생성할 것인가?

▼ 예제 7-7/ch7/PointTest.java

```
class PointTest {  
    public static void main(String args[]) {  
        Point3D p3 = new Point3D(1,2,3);  
    }  
  
    class Point {  
        int x, y;  
  
        Point(int x, int y) {  
            this.x = x;  
            this.y = y;  
        }  
  
        String getLocation() {  
            return "x :" + x + ", y :" + y;  
        }  
    }  
}
```

```

class Point3D extends Point {
    int z;

    Point3D(int x, int y, int z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    String getLocation() {           // 오버라이딩
        return "x :" + x + ", y :" + y + ", z :" + z;
    }
}

```

생성자 첫 줄에서 다른 생성자를 호출하지 않기 때문에 컴파일러가 'super();'를 여기에 삽입한다. super()는 Point3D의 조상인 Point 클래스의 기본 생성자인 Point()를 의미한다.

▼ 컴파일결과

```

C:\jdk1.8\work>javac PointTest.java
PointTest.java:22: cannot resolve symbol
symbol : constructor Point ()
location: class Point
Point3D(int x, int y, int z) {
^
1 error

```

이 예제를 컴파일하면 위와 같은 컴파일에러가 발생할 것이다. Point3D클래스의 생성자에서 조상 클래스의 생성자인 Point()를 찾을 수 없다는 내용이다.

Point3D클래스의 생성자의 첫 줄이 생성자(조상의 것이든 자신의 것이든)를 호출하는 문장이 아니기 때문에 컴파일러는 다음과 같이 자동적으로 'super();'를 Point3D클래스의 생성자의 첫 줄에 넣어 준다.

```

Point3D(int x, int y, int z) {
    super();
    this.x = x;
    this.y = y;
    this.z = z;
}

```

그래서 Point3D클래스의 인스턴스를 생성하면, 생성자 Point3D(int x, int y, int z)가 호출되면서 첫 문장인 'super();'를 수행하게 된다. super()는 Point3D클래스의 조상인 Point클래스의 기본 생성자인 Point()를 뜻하므로 Point()가 호출된다.

그러나 Point클래스에 생성자 Point()가 정의되어 있지 않기 때문에 위와 같은 컴파일에러가 발생한 것이다. 이 에러를 수정하려면, Point클래스에 생성자 Point()를 추가해주던가, 생성자 Point3D(int x, int y, int z)의 첫 줄에서 Point(int x, int y)를 호출하도록 변경하면 된다.

| 주의 | 생성자가 정의되어 있는 클래스에는 컴파일러가 기본 생성자를 자동적으로 추가하지 않는다.

```
Point3D(int x, int y, int z) {
    super(x, y); // 조상클래스의 생성자 Point(int x, int y)를 호출한다.
    this.z = z;
}
```

위와 같이 변경하면 된다. 문제없이 컴파일 될 것이다. 조상 클래스의 멤버변수는 이처럼 조상의 생성자에 의해 초기화되도록 해야 하는 것이다.

▼ 예제 7-8/ch7/PointTest2.java

```
class PointTest2 {
    public static void main(String args[]) {
        Point3D p3 = new Point3D();
        System.out.println("p3.x=" + p3.x);
        System.out.println("p3.y=" + p3.y);
        System.out.println("p3.z=" + p3.z);
    }
}

class Point {
    int x=10;
    int y=20;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class Point3D extends Point {
    int z = 30;

    Point3D() {
        super(); // Point3D(int x, int y, int z)를 호출한다.
    }

    Point3D(int x, int y, int z) {
        super(x, y); // Point(int x, int y)를 호출한다.
        this.z = z;
    }
}
```

생성자 첫 줄에서 다른 생성자를 호출하지 않기 때문에 컴파일러가 'super();'를 여기에 삽입한다.
super()는 Point의 조상인 Object 클래스의 기본 생성자인 Object()를 의미한다.

▼ 실행결과

```
p3.x=100
p3.y=200
p3.z=300
```

| 플래시동영상 | Super.exe는 예제7-8의 실행과정을 설명과 함께 자세히 보여준다.

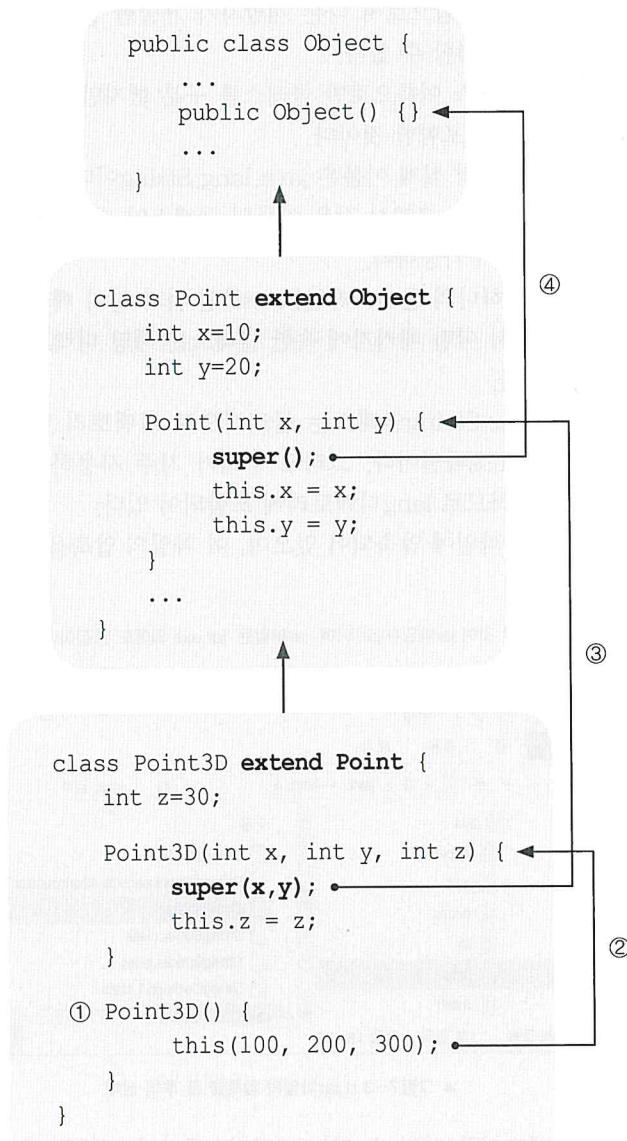
Point3D클래스의 인스턴스를 생성할 때 어떤 순서로 인스턴스의 초기화가 진행되는지 보여주기 위한 예제이다. Point클래스의 생성자 Point(int x, int y)는 어떠한 생성자도 호출하고 있지 않기 때문에 컴파일 후에 다음과 같은 코드로 변경된다.

```
Point(int x, int y) {
    super(); // 조상인 Object클래스의 생성자 Object()를 호출한다.
    this.x = x;
    this.y = y;
}
```

그래서 'Point3D p3 = new Point3D();'와 같이 인스턴스를 생성하면, 아래와 같은 순서로 생성자가 호출된다.

Point3D() → Point3D(int x, int y, int z) → Point(int x, int y) → Object()

어떤 클래스의 인스턴스를 생성하면, 클래스 상속관계의 최고조상인 Object클래스까지 거슬러 올라가면서 모든 조상클래스의 생성자가 순서대로 호출된다는 것을 알 수 있다.



3. package와 import

3.1 패키지(package)

패키지란, 클래스의 묶음이다. 패키지에는 클래스 또는 인터페이스를 포함시킬 수 있으며, 서로 관련된 클래스들끼리 그룹 단위로 묶어 놓음으로써 클래스를 효율적으로 관리할 수 있다. 같은 이름의 클래스 일지라도 서로 다른 패키지에 존재하는 것이 가능하므로, 자신만의 패키지 체계를 유지함으로써 다른 개발자가 개발한 클래스 라이브러리의 클래스와 이름이 충돌하는 것을 피할 수 있다.

지금까지는 단순히 클래스 이름으로만 클래스를 구분 했지만, 사실 클래스의 실제 이름(full name)은 패키지명을 포함한 것이다.

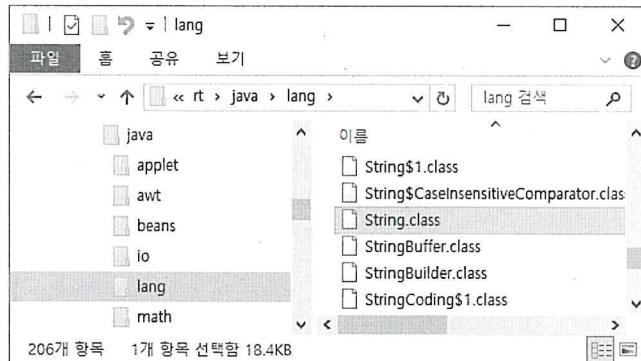
예를 들면 String클래스의 실제 이름은 java.lang.String이다. java.lang패키지에 속한 String클래스라는 의미이다. 그래서 같은 이름의 클래스일 지라도 서로 다른 패키지에 속하면 패키지명으로 구별이 가능하다.

클래스가 물리적으로 하나의 클래스파일(.class)인 것과 같이 패키지는 물리적으로 하나의 디렉토리이다. 그래서 어떤 패키지에 속한 클래스는 해당 디렉토리에 존재하는 클래스 파일(.class)이어야 한다.

예를 들어, java.lang.String클래스는 물리적으로 디렉토리 java의 서브디렉토리인 lang에 속한 String.class파일이다. 그리고 우리가 자주 사용하는 System클래스 역시 java.lang패키지에 속하므로 lang디렉토리에 포함되어 있다.

String클래스는 rt.jar파일에 압축되어 있으며, 이 파일의 압축을 풀면 아래의 그림과 같다.

| 참고 | 클래스 파일들을 압축한 것이 jar파일(*.jar)이며, jar파일은 'jar.exe'외에도 알집이나 winzip으로 압축을 풀 수 있다.



▲ 그림7-3 rt.jar파일의 압축을 풀 후의 상태

디렉토리가 하위 디렉토리를 가질 수 있는 것처럼, 패키지도 다른 패키지를 포함할 수 있으며 점 '.'으로 구분한다. 예를 들면 java.lang패키지에서 lang패키지는 java패키지의 하위패키지이다.

- 하나의 소스파일에는 첫 번째 문장으로 단 한 번의 패키지 선언만을 허용한다.
- 모든 클래스는 반드시 하나의 패키지에 속해야 한다.
- 패키지는 점(.)을 구분자로 하여 계층구조로 구성할 수 있다.
- 패키지는 물리적으로 클래스 파일(.class)을 포함하는 하나의 디렉토리이다.

3.2 패키지의 선언

패키지를 선언하는 것은 아주 간단하다. 클래스나 인터페이스의 소스파일(.java)의 맨 위에 다음과 같이 한 줄만 적어주면 된다.

```
package 패키지명;
```

위와 같은 패키지 선언문은 반드시 소스파일에서 주석과 공백을 제외한 첫 번째 문장이어야 하며, 하나의 소스파일에 단 한번만 선언될 수 있다. 해당 소스파일에 포함된 모든 클래스나 인터페이스는 선언된 패키지에 속하게 된다.

패키지명은 대소문자를 모두 허용하지만, 클래스명과 쉽게 구분하기 위해서 소문자로 하는 것을 원칙으로 하고 있다.

모든 클래스는 반드시 하나의 패키지에 포함되어야 한다고 했다. 그럼에도 불구하고 지금까지 소스파일을 작성할 때 패키지를 선언하지 않고도 아무런 문제가 없었던 이유는 자바에서 기본적으로 제공하는 ‘이름없는 패키지(unnamed package)’ 때문이다.

소스파일에 자신이 속할 패키지를 지정하지 않은 클래스는 자동적으로 ‘이름 없는 패키지’에 속하게 된다. 결국 패키지를 지정하지 않는 모든 클래스들은 같은 패키지에 속하는 셈이다.

간단한 프로그램은 패키지를 지정하지 않아도 별 문제 없지만, 큰 프로젝트나 Java API와 같은 클래스 라이브러리를 작성하는 경우에는 미리 패키지를 구성하여 적용해야 한다.

▼ 예제 7-9/ch7/PackageTest.java

```
package com.codechobo.book;

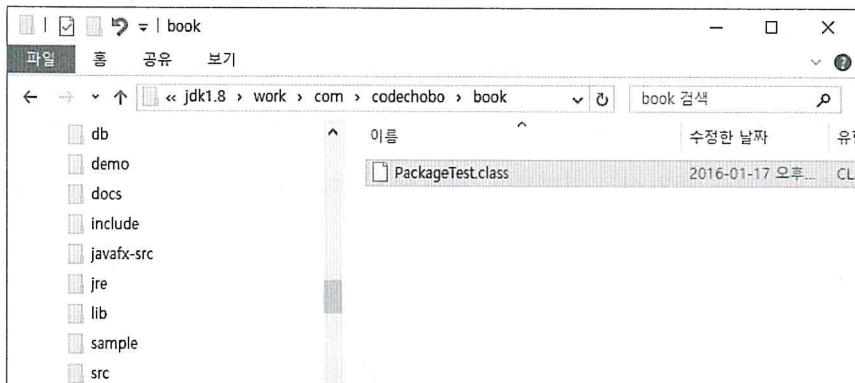
class PackageTest {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

위의 예제를 작성한 뒤 다음과 같이 ‘-d’ 옵션을 추가하여 컴파일을 한다.

```
C:\jdk1.8\work>javac -d . PackageTest.java
```

‘-d’ 옵션은 소스파일에 지정된 경로를 통해 패키지의 위치를 찾아서 클래스파일을 생성한다. 만일 지정된 패키지와 일치하는 디렉토리가 존재하지 않는다면 자동적으로 생성한다.

'-d'옵션 뒤에는 해당 패키지의 루트(root)디렉토리의 경로를 적어준다. 여기서는 현재 디렉토리(.) 즉, 'C:\jdk1.8\work'로 지정했기 때문에 컴파일을 수행하고 나면 다음과 같은 구조로 디렉토리가 생성된다.



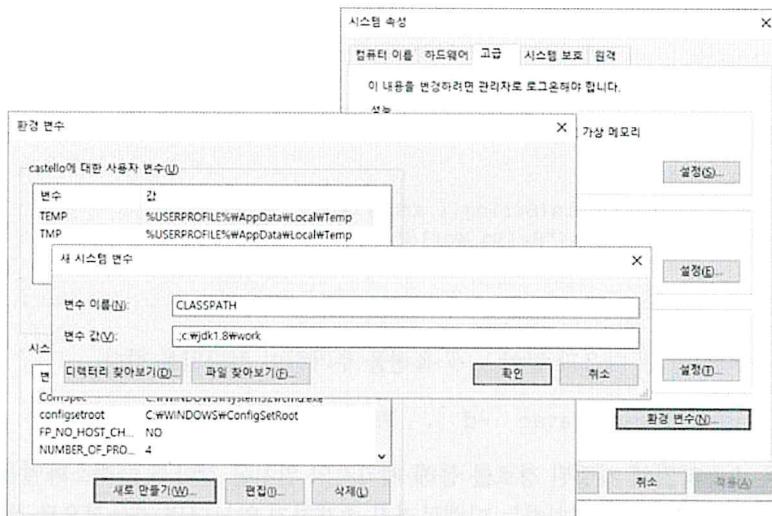
▲ 그림 7-4 예제 7-9를 컴파일한 후 생성된 파일구조

기존에 디렉토리가 존재하지 않았으므로 컴파일러가 패키지의 계층구조에 맞게 새로 디렉토리를 생성하고 컴파일된 클래스파일(PackageTest.class)를 book디렉토리에 놓았다.

이제는 패키지의 루트 디렉토리를 클래스패스(classpath)에 포함시켜야 한다. com.codechobo.book패키지의 루트 디렉토리는 디렉토리 'com'의 상위 디렉토리인 'C:\jdk1.8\work'이다. 이 디렉토리를 클래스패스에 포함시켜야만 실행 시 JVM이 PackageTest클래스를 찾을 수 있다.

| 참고 | 클래스패스(classpath)는 컴파일러(javac.exe)나 JVM 등이 클래스의 위치를 찾는데 사용되는 경로이다.

윈도우즈에서는 '제어판-시스템-고급 시스템 설정-환경변수-새로 만들기'에서 변수 이름에 'CLASSPATH'를 입력하고 변수 값에는 '.;c:\jdk1.8\work'를 입력한다.



▲ 그림 7-5 윈도우즈 10에서의 클래스패스(CLASSPATH) 설정

';'를 구분자로 하여 여러 개의 경로를 클래스패스에 지정할 수 있으며, 맨 앞에 '.';를 추가한 이유는 현재 디렉토리(.)를 클래스패스에 포함시키기 위해서이다.

클래스패스를 지정해 주지 않으면 기본적으로 현재 디렉토리(.)가 클래스패스로 지정되지만, 이처럼 클래스패스를 따로 지정해주는 경우에는 더 이상 현재 디렉토리가 자동적으로 클래스패스로 지정되지 않기 때문에 이처럼 별도로 추가를 해주어야 한다.

jar파일을 클래스패스에 추가하기 위해서는 경로와 파일명을 적어주어야 한다. 예를 들어 'C:\jdk1.8\work\util.jar'파일을 클래스패스에 포함시키려면 다음과 같이 한다.

```
C:\WINDOWS>SET CLASSPATH = .;C:\jdk1.8\work;C:\jdk1.8\work\util.jar;
```

이제 클래스패스가 바르게 설정되었는지 확인하기 위해 다음과 같은 명령어를 입력해보자.

```
C:\WINDOWS>echo %classpath%
.;C:\jdk1.8\work;
```

현재 디렉토리를 의미하는 '.'과 'C:\jdk1.8\work'가 클래스패스로 잘 지정되었음을 알 수 있다. 자, 이제 PackageTest예제를 실행시켜보자.

▼ 실행결과

```
C:\WINDOWS>java com.codechobo.book.PackageTest
Hello World!
```

실행 시에는 이와 같이 PackageTest클래스의 패키지명을 모두 적어주어야 한다.

JDK에 기본적으로 설정되어 있는 클래스패스를 이용하면 위의 예제에서와 같이 클래스패스를 따로 설정하지 않아도 된다. 새로 추가하고자 하는 클래스를 'JDK설치디렉토리\jre\classes'디렉토리에, jar파일인 경우에는 'JDK설치디렉토리\jre\lib\ext'디렉토리에 넣기만 하면 된다.

| 참고 | jre디렉토리 아래의 classes디렉토리는 JDK설치 시에 자동으로 생성되지 않으므로 사용자가 직접 생성해야 한다.

또는 실행 시에 '-cp'옵션을 이용해서 일시적으로 클래스패스를 지정해 줄 수도 있다.

```
C:\WINDOWS>java -cp c:\jdk1.8\work com.codechobo.book.PackageTest
```

3.3 import문

소스코드를 작성할 때 다른 패키지의 클래스를 사용하려면 패키지명이 포함된 클래스 이름을 사용해야 한다. 하지만, 매번 패키지명을 붙여서 작성하기란 여간 불편한 일이 아니다.

클래스의 코드를 작성하기 전에 import문으로 사용하고자 하는 클래스의 패키지를 미리 명시해주면 소스코드에 사용되는 클래스이름에서 패키지명은 생략할 수 있다.

import문의 역할은 컴파일러에게 소스파일에 사용된 클래스의 패키지에 대한 정보를 제공하는 것이다. 컴파일 시에 컴파일러는 import문을 통해 소스파일에 사용된 클래스들의 패키지를 알아 낸 다음, 모든 클래스이름 앞에 패키지명을 붙여 준다.

이클립스는 단축키 ‘ctrl+shift+o’를 누르면, 자동으로 import문을 추가해주는 편리한 기능을 제공한다.

참고 | import문은 프로그램의 성능에 전혀 영향을 미치지 않는다. import문을 많이 사용하면 컴파일 시간이 아주 조금 더 걸릴 뿐이다.

3.4 import문의 선언

모든 소스파일(.java)에서 import문은 package문 다음에, 그리고 클래스 선언문 이전에 위치해야 한다. import문은 package문과 달리 한 소스파일에 여러 번 선언할 수 있다.

일반적인 소스파일(*.java)의 구성은 다음의 순서로 되어 있다.

- ① package문
- ② import문
- ③ 클래스 선언

import문을 선언하는 방법은 다음과 같다.

```
import 패키지명.클래스명;  
또는  
import 패키지명.*;
```

키워드 import와 패키지명을 생략하고자 하는 클래스의 이름을 패키지명과 함께 써주면 된다. 같은 패키지에서 여러 개의 클래스가 사용될 때, import문을 여러 번 사용하는 대신 ‘패키지명.*’을 이용해서 지정된 패키지에 속하는 모든 클래스를 패키지명 없이 사용할 수 있다.

클래스이름을 지정해주는 대신 ‘*’을 사용하면, 컴파일러는 해당 패키지에서 일치하는 클래스이름을 찾아야 하는 수고를 더 해야 할 것이다. 단지 그 뿐이다. 실행 시 성능상의 차이는 전혀 없다.

```
import java.util.Calendar;
import java.util.Date;
import java.util.ArrayList;
```

이처럼 import문을 여러 번 사용하는 대신 아래와 같이 한 문장으로 처리할 수 있다.

```
import java.util.*;
```

한 패키지에서 여러 클래스를 사용하는 경우 클래스의 이름을 일일이 지정해주는 것보다 ‘패키지명.*’과 같이 하는 것이 편리하다.

하지만, import하는 패키지의 수가 많을 때는 어느 클래스가 어느 패키지에 속하는지 구별하기 어렵다는 단점이 있다.

한 가지 더 알아두어야 할 것은 import문에서 클래스의 이름 대신 ‘*’을 사용하는 것이 하위 패키지의 클래스까지 포함하는 것은 아니라는 것이다.

```
import java.util.*;
import java.text.*;
```

그래서 위의 두 문장 대신 다음과 같이 할 수는 없다.

```
import java.*;
```

▼ 예제 7-10/ch7/ImportTest.java

```
import java.text.SimpleDateFormat;
import java.util.Date;

class ImportTest {
    public static void main(String[] args) {
        Date today = new Date();

        SimpleDateFormat date = new SimpleDateFormat("yyyy/MM/dd");
        SimpleDateFormat time = new SimpleDateFormat("hh:mm:ss a");

        System.out.println("오늘 날짜는 " + date.format(today));
        System.out.println("현재 시간은 " + time.format(today));
    }
}
```

▼ 실행결과

```
오늘 날짜는 2003/01/07
현재 시간은 01:59:53 오후
```

현재 날짜와 시간을 지정된 형식에 맞춰 출력하는 예제이다. `SimpleDateFormat`과 `Date` 클래스는 다른 패키지에 속한 클래스이므로 `import`문으로 어느 패키지에 속하는 클래스인지 명시해 주었다. 그래서 소스에서 클래스이름 앞에 패키지명을 생략할 수 있었다.

만일 `import`문을 지정하지 않았다면 다음과 같이 클래스이름에 패키지명도 적어줘야 했을 것이다.

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

java.text.SimpleDateFormat date
    = new java.text.SimpleDateFormat("yyyy/MM/dd");
java.text.SimpleDateFormat time
    = new java.text.SimpleDateFormat("hh:mm:ss a");
```

`import`문으로 패키지를 지정하지 않으면 위와 같이 모든 클래스이름 앞에 패키지명을 반드시 붙여야 한다. 단, 같은 패키지 내의 클래스들은 `import`문을 지정하지 않고도 패키지명을 생략할 수 있다.

지금까지 `System`과 `String` 같은 `java.lang` 패키지의 클래스들을 패키지명 없이 사용할 수 있었던 이유는 모든 소스파일에는 묵시적으로 다음과 같은 `import`문이 선언되어 있기 때문이다.

```
import java.lang.*;
```

`java.lang` 패키지는 매우 빈번히 사용되는 중요한 클래스들이 속한 패키지이기 때문에 따로 `import`문으로 지정하지 않아도 되도록 한 것이다.

3.5 static import문

`import`문을 사용하면 클래스의 패키지명을 생략할 수 있는 것과 같이 `static import`문을 사용하면 `static` 멤버를 호출할 때 클래스 이름을 생략할 수 있다. 특정 클래스의 `static` 멤버를 자주 사용할 때 편리하다. 그리고 코드도 간결해진다.

```
import static java.lang.Integer.*;      // Integer 클래스의 모든 static 메서드
import static java.lang.Math.random();   // Math.random() 만. 괄호 안붙임.
import static java.lang.System.out;     // System.out을 out만으로 참조 가능
```

만일 위와 같이 `static import`문을 선언하였다면, 아래의 왼쪽 코드를 오른쪽 코드와 같이 간략히 할 수 있다.

```
System.out.println(Math.random());      ←→      out.println(random());
```

▼ 예제 7-11/ch7/StaticImportEx1.java

```
import static java.lang.System.out;
import static java.lang.Math.*;

class StaticImportEx1 {
    public static void main(String[] args) {
        // System.out.println(Math.random());
        out.println(random());

        // System.out.println("Math.PI :" + Math.PI);
        out.println("Math.PI :" + PI);
    }
}
```

▼ 실행결과

```
0.6372776821515502
Math.PI :3.141592653589793
```

4. 제어자(modifier)

4.1 제어자란?

제어자(modifier)는 클래스, 변수 또는 메서드의 선언부에 함께 사용되어 부가적인 의미를 부여한다. 제어자의 종류는 크게 접근 제어자와 그 외의 제어자로 나눌 수 있다.

접근 제어자 public, protected, default, private

그 외 static, final, abstract, native, transient, synchronized, volatile, strictfp

제어자는 클래스나 멤버변수와 메서드에 주로 사용되며, 하나의 대상에 대해서 여러 제어자를 조합하여 사용하는 것이 가능하다.

단, 접근 제어자는 한 번에 네 가지 중 하나만 선택해서 사용할 수 있다. 즉, 하나의 대상에 대해서 public과 private을 함께 사용할 수 없다는 것이다.

| 참고 | 제어자들 간의 순서는 관계없지만 주로 접근 제어자를 제일 왼쪽에 놓는 경향이 있다.

4.2 static – 클래스의, 공통적인

static은 ‘클래스의’ 또는 ‘공통적인’의 의미를 가지고 있다. 인스턴스변수는 하나의 클래스로부터 생성되었더라도 각기 다른 값을 유지하지만, 클래스변수(static멤버변수)는 인스턴스에 관계없이 같은 값을 갖는다. 그 이유는 하나의 변수를 모든 인스턴스가 공유하기 때문이다.

static이 붙은 멤버변수와 메서드, 그리고 초기화 블럭은 인스턴스가 아닌 클래스에 관계된 것이기 때문에 인스턴스를 생성하지 않고도 사용할 수 있다.

인스턴스메서드와 static메서드의 근본적인 차이는 메서드 내에서 인스턴스 멤버를 사용하는가의 여부에 있다.

static이 사용될 수 있는 곳 – 멤버변수, 메서드, 초기화 블럭

제어자	대상	의 미
static	멤버변수	<ul style="list-style-type: none"> - 모든 인스턴스에 공통적으로 사용되는 클래스변수가 된다. - 클래스변수는 인스턴스를 생성하지 않고도 사용 가능하다. - 클래스가 메모리에 로드될 때 생성된다.
	메서드	<ul style="list-style-type: none"> - 인스턴스를 생성하지 않고도 호출이 가능한 static 메서드가 된다. - static메서드 내에서는 인스턴스멤버들을 직접 사용할 수 없다.

인스턴스 멤버를 사용하지 않는 메서드는 static을 붙여서 static메서드로 선언하는 것을 고려해보도록 하자. 가능하다면 static메서드로 하는 것이 인스턴스를 생성하지 않고도 호출이 가능해서 더 편리하고 속도도 더 빠르다.

| 참고 | static초기화 블럭은 클래스가 메모리에 로드될 때 단 한번만 수행되며, 주로 클래스변수(static변수)를 초기화하는데 주로 사용된다.

```
class StaticTest {
    static int width = 200;           // 클래스 변수(static변수)
    static int height = 120;          // 클래스 변수(static변수)

    static {                         // 클래스 초기화 블럭
        // static변수의 복잡한 초기화 수행
    }

    static int max(int a, int b) {    // 클래스 메서드(static메서드)
        return a > b ? a : b;
    }
}
```

4.3 final – 마지막의, 변경될 수 없는

final은 ‘마지막의’ 또는 ‘변경될 수 없는’의 의미를 가지고 있으며 거의 모든 대상에 사용될 수 있다.

변수에 사용되면 값을 변경할 수 없는 상수가 되며, 메서드에 사용되면 오버라이딩을 할 수 없게 되고 클래스에 사용되면 자신을 확장하는 자손클래스를 정의하지 못하게 된다.

final이 사용될 수 있는 곳 – 클래스, 메서드, 멤버변수, 지역변수

제어자	대상	의 미
final	클래스	변경될 수 없는 클래스. 확장될 수 없는 클래스가 된다. 그래서 final로 지정된 클래스는 다른 클래스의 조상이 될 수 없다.
	메서드	변경될 수 없는 메서드. final로 지정된 메서드는 오버라이딩을 통해 재정의 될 수 없다.
	멤버변수 지역변수	변수 앞에 final이 붙으면, 값을 변경할 수 없는 상수가 된다.

| 참고 | 대표적인 final클래스로는 String과 Math가 있다.

```
final class FinalTest {           // 조상이 될 수 없는 클래스
    final int MAX_SIZE = 10;       // 값을 변경할 수 없는 멤버변수(상수)

    final void getMaxSize() {      // 오버라이딩할 수 없는 메서드(변경불가)
        final int LV = MAX_SIZE;   // 값을 변경할 수 없는 지역변수(상수)
        return MAX_SIZE;
    }
}
```

생성자를 이용한 final멤버 변수의 초기화

final이 붙은 변수는 상수이므로 일반적으로 선언과 초기화를 동시에 하지만, 인스턴스 변수의 경우 생성자에서 초기화 되도록 할 수 있다.

클래스 내에 매개변수를 갖는 생성자를 선언하여, 인스턴스를 생성할 때 final이 붙은 멤버변수를 초기화하는데 필요한 값을 생성자의 매개변수로부터 제공받는 것이다.

이 기능을 활용하면 각 인스턴스마다 final이 붙은 멤버변수가 다른 값을 갖도록 하는 것 이 가능하다.

만일 이것이 불가능하다면 클래스에 선언된 final이 붙은 인스턴스변수는 모든 인스턴스에서 같은 값을 가져야만 할 것이다.

예를 들어 카드의 경우, 각 카드마다 다른 종류와 숫자를 갖지만, 일단 카드가 생성되면 카드의 값이 변경되어서는 안 된다. 52장의 카드 중에서 하나만 잘못 바꿔도 같은 카드가 2장이 되는 일이 생기기 때문이다. 그래서 카드의 값을 바꾸기 보다는 카드의 순서를 바꾸는 쪽이 더 안전한 방법이다.

▼ 예제 7-12/ch7/FinalCardTest.java

```
class Card {
    final int NUMBER; // 상수지만 선언과 함께 초기화 하지 않고
    final String KIND; // 생성자에서 단 한번만 초기화할 수 있다.
    static int width = 100;
    static int height = 250;

    Card(String kind, int num) {
        KIND = kind;
        NUMBER = num; // 매개변수로 넘겨받은 값으로
                      // KIND와 NUMBER를 초기화한다.
    }

    Card() {
        this("HEART", 1);
    }

    public String toString() {
        return KIND +" "+ NUMBER;
    }
}

class FinalCardTest {
    public static void main(String args[]) {
        Card c = new Card("HEART", 10);
        // c.NUMBER = 5; // 에러!!! cannot assign a
                      // value to final variable
                      // NUMBER
        System.out.println(c.KIND);
        System.out.println(c.NUMBER);
        System.out.println(c); // System.out.println(c.toString());
    }
}
```

에러!!! cannot assign a
value to final variable
NUMBER

▼ 실행결과
HEART
10
HEART 10

4.4 abstract – 추상의, 미완성의

abstract는 ‘미완성’의 의미를 가지고 있다. 메서드의 선언부만 작성하고 실제 수행내용은 구현하지 않은 추상 메서드를 선언하는데 사용된다.

그리고 클래스에 사용되어 클래스 내에 추상메서드가 존재한다는 것을 쉽게 알 수 있게 한다. 보다 자세한 내용은 ‘추상 클래스’(p.375)에서 다룬다.

abstract가 사용될 수 있는 곳 – 클래스, 메서드

제어자	대상	의미
	클래스	클래스 내에 추상 메서드가 선언되어 있음을 의미한다.
abstract	메서드	선언부만 작성하고 구현부는 작성하지 않은 추상 메서드임을 알린다.

추상 클래스는 아직 완성되지 않은 메서드가 존재하는 ‘미완성 설계도’이므로 인스턴스를 생성할 수 없다.

```
abstract class AbstractTest {           // 추상 클래스(추상 메서드를 포함한 클래스)
    abstract void move();               // 추상 메서드(구현부가 없는 메서드)
}
```

꽤 드물지만 추상 메서드가 없는 클래스, 즉 완성된 클래스도 abstract를 붙여서 추상 클래스로 만드는 경우도 있다. 예를 들어, `java.awt.event.WindowAdapter`는 아래와 같이 아무런 내용이 없는 메서드들만 정의되어 있다. 이런 클래스는 인스턴스를 생성해봐야 할 수 있는 것이 아무것도 없다. 그래서 인스턴스를 생성하지 못하게 클래스 앞에 제어자 ‘abstract’를 붙여 놓은 것이다.

```
public abstract class WindowAdapter
    implements WindowListener,WindowStateListener,WindowFocusListener {
    public void windowOpened(WindowEvent e) {}
    public void windowClosing(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    ...
}
```

이 클래스 자체로는 쓸모가 없지만, 다른 클래스가 이 클래스를 상속받아서 일부의 원하는 메서드만 오버라이딩해도 된다는 장점이 있다. 만일 이 클래스가 없다면 아무런 내용도 없는 메서드를 잔뜩 오버라이딩해야 한다. 아직 추상 클래스와 인터페이스를 배우지 않았으니 ‘이런 경우도 있구나.’라고 가볍게 참고만 하기 바란다.

| 참고 | 가끔 이런 질문을 받기 때문에 참고로 적어놓은 것일 뿐이니 심각하게 고민하지 않기 바란다.

4.5 접근 제어자(access modifier)

접근 제어자는 멤버 또는 클래스에 사용되어, 해당하는 멤버 또는 클래스를 외부에서 접근하지 못하도록 제한하는 역할을 한다.

접근 제어자가 default임을 알리기 위해 실제로 default를 붙이지는 않는다. 클래스나 멤버변수, 메서드, 생성자에 접근 제어자가 지정되어 있지 않다면, 접근 제어자가 default임을 뜻한다.

접근 제어자가 사용될 수 있는 곳 – 클래스, 멤버변수, 메서드, 생성자

private 같은 클래스 내에서만 접근이 가능하다.

default 같은 패키지 내에서만 접근이 가능하다.

protected 같은 패키지 내에서, 그리고 다른 패키지의 자손클래스에서 접근이 가능하다.

public 접근 제한이 전혀 없다.

제어자	같은 클래스	같은 패키지	자손클래스	전체
public				
protected				
(default)				
private				

접근 범위가 넓은 쪽에서 좁은 쪽의 순으로 왼쪽부터 나열하면 다음과 같다.

public > protected > (default) > private

public은 접근 제한이 전혀 없는 것이고, private은 같은 클래스 내에서만 사용하도록 제한하는 가장 높은 제한이다. 그리고 default는 같은 패키지내의 클래스에서만 접근이 가능하도록 하는 것이다.

마지막으로 protected는 패키지에 관계없이 상속관계에 있는 자손클래스에서 접근할 수 있도록 하는 것이 제한목적이지만, 같은 패키지 내에서도 접근이 가능하다. 그래서 protected가 default보다 접근범위가 더 넓다.

| 참고 | 접근 제어자가 default라는 것은 아무런 접근 제어자도 붙이지 않는 것을 의미한다.

대상	사용가능한 접근 제어자
클래스	public, (default)
메서드	public, protected, (default), private
멤버변수	없음

▲ 표 7-1 대상에 따라 사용할 수 있는 접근 제어자

접근 제어자를 이용한 캡슐화

클래스나 멤버, 주로 멤버에 접근 제어자를 사용하는 이유는 클래스의 내부에 선언된 데이터를 보호하기 위해서이다.

데이터가 유효한 값을 유지하도록, 또는 비밀번호와 같은 데이터를 외부에서 함부로 변경하지 못하도록 하기 위해서는 외부로부터의 접근을 제한하는 것이 필요하다.

이것을 데이터 감추기(data hiding)라고 하며, 객체지향개념의 캡슐화(encapsulation)에 해당하는 내용이다.

또 다른 이유는 클래스 내에서만 사용되는, 내부 작업을 위해 임시로 사용되는 멤버 변수나 부분작업을 처리하기 위한 메서드 등의 멤버들을 클래스 내부에 감추기 위해서이다.

외부에서 접근할 필요가 없는 멤버들을 private으로 지정하여 외부에 노출시키지 않음으로써 복잡성을 줄일 수 있다. 이 것 역시 캡슐화에 해당한다.

접근 제어자를 사용하는 이유

- 외부로부터 데이터를 보호하기 위해서
- 외부에는 불필요한, 내부적으로만 사용되는, 부분을 감추기 위해서

만일 메서드 하나를 변경해야 한다고 가정했을 때, 이 메서드의 접근 제어자가 public이라면, 메서드를 변경한 후에 오류가 없는지 테스트해야하는 범위가 넓다. 그러나 접근 제어자가 default라면 패키지 내부만 확인해 보면 되고, private이면 클래스 하나만 살펴보면 된다. 이처럼 접근 제어자 하나가 때로는 상당한 차이를 만들어낼 수 있다. 접근 제어자를 적절히 선택해서 접근 범위를 최소화하도록 노력하자.

이제 구체적인 예제를 통해 자세히 알아보자. 시간을 표시하기 위한 클래스 Time이 다음과 같이 정의되어 있을 때,

```
public class Time {
    public int hour;
    public int minute;
    public int second;
}
```

이 클래스의 인스턴스를 생성한 다음, 멤버 변수에 직접 접근하여 값을 변경할 수 있을 것이다.

```
Time t = new Time();
t.hour = 25;
```

멤버 변수 hour는 0보다는 같거나 크고 24보다는 작은 범위의 값을 가져야 하지만 위의 코드에서처럼 잘못된 값을 지정한다고 해도 이것을 막을 방법은 없다.

이런 경우 멤버 변수를 private이나 protected로 제한하고 멤버 변수의 값을 읽고 변경할 수 있는 public 메서드를 제공함으로써 간접적으로 멤버 변수의 값을 다를 수 있도록 하는 것이 바람직하다.

```

public class Time {
    private int hour;
    private int minute;
    private int second;

    public int getHour() { return hour; }
    public void setHour(int hour) {
        if (hour < 0 || hour > 23) return;
        this.hour = hour;
    }
    public int getMinute() { return minute; }
    public void setMinute(int minute) {
        if (minute < 0 || minute > 59) return;
        this.minute = minute;
    }
    public int getSecond() { return second; }
    public void setSecond(int second) {
        if (second < 0 || second > 59) return;
        this.second = second;
    }
}

```

접근 제어자를 private으로 하여 외부에서 직접 접근하지 못하도록 한다.

get으로 시작하는 메서드는 단순히 멤버변수의 값을 반환하는 일을 하고, set으로 시작하는 메서드는 매개변수에 지정된 값을 검사하여 조건에 맞는 값일 때만 멤버변수의 값을 변경하도록 작성되어 있다.

만일 상속을 통해 확장될 것이 예상되는 클래스라면 멤버에 접근 제한을 주되 자손클래스에서 접근하는 것이 가능하도록 하기 위해 private대신 protected를 사용한다. private이 붙은 멤버는 자손 클래스에서도 접근이 불가능하기 때문이다.

보통 멤버변수의 값을 읽는 메서드의 이름을 ‘get멤버변수이름’으로 하고, 멤버변수의 값을 변경하는 메서드의 이름을 ‘set멤버변수이름’으로 한다. 반드시 그렇게 해야 하는 것은 아니지만 암묵적인 규칙이므로 특별한 이유가 없는 한 따르도록 하자. 그리고 get으로 시작하는 메서드를 ‘겟터(getter)’, set으로 시작하는 메서드를 ‘셋터(setter)’라고 부른다.

▼ 예제 7-13/ch7/TimeTest.java

```

public class TimeTest {
    public static void main(String[] args) {
        Time t = new Time(12, 35, 30);
        System.out.println(t);
        // t.hour = 13;————
        t.setHour(t.getHour()+1); // 현재시간보다 1시간 후로 변경한다.
        System.out.println(t); // System.out.println(t.toString());과 같다.
    }
}

```

에러! 변수 hour의 접근 제어자가 private이므로 접근할 수 없다.

```

class Time {
    private int hour, minute, second;

    Time(int hour, int minute, int second) {
        setHour(hour);
        setMinute(minute);
        setSecond(second);
    }

    public int getHour() { return hour; }
    public void setHour(int hour) {
        if (hour < 0 || hour > 23) return;
        this.hour = hour;
    }
    public int getMinute() { return minute; }
    public void setMinute(int minute) {
        if (minute < 0 || minute > 59) return;
        this.minute = minute;
    }
    public int getSecond() { return second; }
    public void setSecond(int second) {
        if (second < 0 || second > 59) return;
        this.second = second;
    }
    public String toString() {
        return hour + ":" + minute + ":" + second;
    }
}

```

▼ 실행결과
12:35:30
13:35:30

Time클래스의 모든 멤버변수의 접근 제어자를 private으로 하고, 이들을 다루기 위한 public메서드를 추가했다. 그래서 't.hour = 13;'과 같이 멤버변수로의 직접적인 접근은 허가되지 않는다. 메서드를 통한 접근만이 허용될 뿐이다.

| 참고 | 하나의 소스파일(*.java)에는 public클래스가 단 하나만 존재할 수 있으며, 소스파일의 이름은 반드시 public클래스의 이름과 같아야 한다.

생성자의 접근 제어자

생성자에 접근 제어자를 사용함으로써 인스턴스의 생성을 제한할 수 있다. 보통 생성자의 접근 제어자는 클래스의 접근 제어자와 같지만, 다르게 지정할 수도 있다.

생성자의 접근 제어자를 private으로 지정하면, 외부에서 생성자에 접근할 수 없으므로 인스턴스를 생성할 수 없게 된다. 그래도 클래스 내부에서는 인스턴스를 생성할 수 있다.

```

class Singleton {
    private Singleton() {
        ...
    }
    ...
}

```

대신 인스턴스를 생성해서 반환해주는 public메서드를 제공함으로써 외부에서 이 클래스의 인스턴스를 사용하도록 할 수 있다. 이 메서드는 public인 동시에 static이어야 한다.

```
class Singleton {
    ...
    private static Singleton s = new Singleton();
    ...
    private Singleton() {
        ...
    }
    ...
    // 인스턴스를 생성하지 않고도 호출할 수 있어야 하므로 static이어야 한다.
    public static Singleton getInstance() {
        return s;
    }
    ...
}
```

getInstance()에서 사용될 수 있도록
인스턴스가 미리 생성되어야 하므로
static이어야 한다.

이처럼 생성자를 통해 직접 인스턴스를 생성하지 못하게 하고 public메서드를 통해 인스턴스에 접근하게 함으로써 사용할 수 있는 인스턴스의 개수를 제한할 수 있다.

또 한 가지, 생성자가 private인 클래스는 다른 클래스의 조상이 될 수 없다. 왜냐하면, 자손클래스의 인스턴스를 생성할 때 조상클래스의 생성자를 호출해야만 하는데, 생성자의 접근 제어자가 private이므로 자손클래스에서 호출하는 것이 불가능하기 때문이다.

그래서 클래스 앞에 final을 더 추가하여 상속할 수 없는 클래스라는 것을 알리는 것이 좋다.

Math클래스는 몇 개의 상수와 static메서드만으로 구성되어 있기 때문에 인스턴스를 생성할 필요가 없다. 그래서 외부로부터의 불필요한 접근을 막기 위해 다음과 같이 생성자의 접근 제어자를 private으로 지정하였다.

```
public final class Math {
    private Math() {}
    ...
}
```

▼ 예제 7-14/ch7/SingletonTest.java

```
final class Singleton {
    private static Singleton s = new Singleton();
    private Singleton() {
        //...
    }
    public static Singleton getInstance() {
        if(s==null)
            s = new Singleton();
        return s;
    }
}
```

```

class SingletonTest {
    public static void main(String args[]) {
        // Singleton s = new Singleton(); •—————> 에러! Singleton() has
        Singleton s = Singleton.getInstance(); private access in
    } Singleton
}

```

4.6 제어자(modifier)의 조합

지금까지 접근 제어자와 static, final, abstract에 대해서 학습했다. 이 외에도 더 많은 제어자들이 있으나 관련 내용이 현재 학습범위를 넘어선다고 판단되어 생략하였다. 이들은 앞으로 자바를 더 깊게 공부하게 되면서 자연스럽게 학습하게 될 것이다.

제어자가 사용될 수 있는 대상을 중심으로 제어자를 정리해보았다. 제어자의 기본적인 의미와 그 대상에 따른 의미 변화를 다시 한 번 되새겨 보도록 하자.

대상	사용가능한 제어자
클래스	public, (default), final, abstract
메서드	모든 접근 제어자, final, abstract, static
멤버변수	모든 접근 제어자, final, static
지역변수	final

▲ 표7-2 대상에 따라 사용할 수 있는 제어자

마지막으로 제어자를 조합해서 사용할 때 주의해야 할 사항에 대해 정리해 보았다.

1. 메서드에 static과 abstract를 함께 사용할 수 없다.

static메서드는 몸통이 있는 메서드에만 사용할 수 있기 때문이다.

2. 클래스에 abstract와 final을 동시에 사용할 수 없다.

클래스에 사용되는 final은 클래스를 확장할 수 없다는 의미이고 abstract는 상속을 통해서 완성되어야 한다는 의미이므로 서로 모순되기 때문이다.

3. abstract메서드의 접근 제어자가 private일 수 없다.

abstract메서드는 자손클래스에서 구현해주어야 하는데 접근 제어자가 private이면, 자손클래스에서 접근할 수 없기 때문이다.

4. 메서드에 private과 final을 같이 사용할 필요는 없다.

접근 제어자가 private인 메서드는 오버라이딩될 수 없기 때문이다. 이 둘 중 하나만 사용해도 의미가 충분하다.

5. 다형성(polymorphism)

5.1 다형성이란?

상속과 함께 객체지향개념의 중요한 특징 중의 하나인 다형성에 대해서 배워 보도록 하자. 다형성은 상속과 깊은 관계가 있으므로 학습하기에 앞서 상속에 대해 충분히 알고 있어야 한다.

객체지향개념에서 다형성이란 '여러 가지 형태를 가질 수 있는 능력'을 의미하며, 자바에서는 한 타입의 참조변수로 여러 타입의 객체를 참조할 수 있도록 함으로써 다형성을 프로그램적으로 구현하였다.

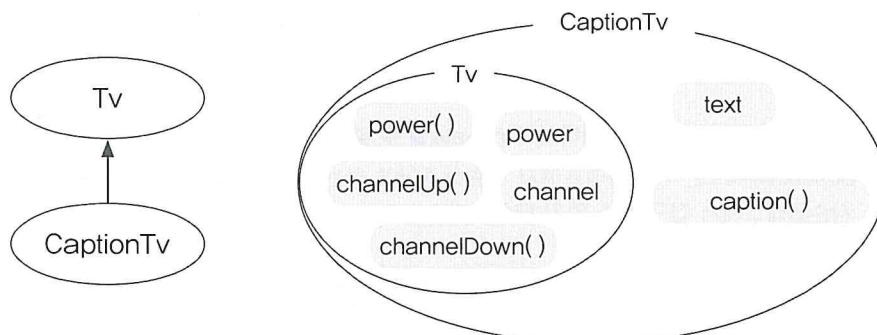
이를 좀 더 구체적으로 말하자면, 조상클래스 타입의 참조변수로 자손클래스의 인스턴스를 참조할 수 있도록 하였다는 것이다. 예제를 통해서 보다 자세히 알아보도록 하자.

```
class Tv {
    boolean power;          // 전원상태 (on/off)
    int channel;            // 채널

    void power() { power = !power; }
    void channelUp() { ++channel; }
    void channelDown() { --channel; }
}

class CaptionTv extends Tv {
    String text;           // 캡션을 보여 주기 위한 문자열
    void caption() { /* 내용생략 */ }
}
```

Tv클래스와 CaptionTv클래스가 이와 같이 정의되어 있을 때, 두 클래스간의 관계를 그림으로 나타내면 아래와 같다.



클래스 **Tv**와 **CaptionTv**는 서로 상속관계에 있으며, 이 두 클래스의 인스턴스를 생성하고 사용하기 위해서는 다음과 같이 할 수 있다.

```
Tv t = new Tv();
CaptionTv c = new CaptionTv();
```

지금까지 우리는 생성된 인스턴스를 다루기 위해서, 인스턴스의 타입과 일치하는 타입의 참조변수만을 사용했다. 즉, `Tv`인스턴스를 다루기 위해서는 `Tv`타입의 참조변수를 사용하고, `CaptionTv`인스턴스를 다루기 위해서는 `CaptionTv`타입의 참조변수를 사용했다.

이처럼 인스턴스의 타입과 참조변수의 타입이 일치하는 것이 보통이지만, `Tv`와 `CaptionTv`클래스가 서로 상속관계에 있을 경우, 다음과 같이 조상 클래스 타입의 참조변수로 자손 클래스의 인스턴스를 참조하도록 하는 것도 가능하다.

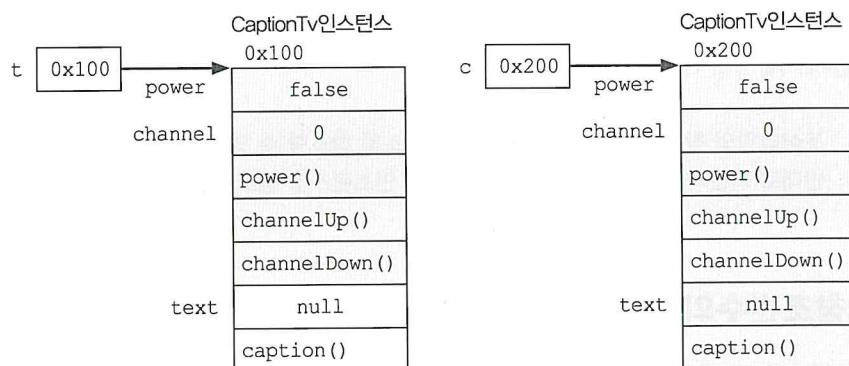
```
Tv t = new CaptionTv(); // 조상 타입의 참조변수로 자손 인스턴스를 참조
```

그러면 이제 인스턴스를 같은 타입의 참조변수로 참조하는 것과 조상타입의 참조변수로 참조하는 것은 어떤 차이가 있는지에 대해서 알아보도록 하자.

```
CaptionTv c = new CaptionTv();
Tv t = new CaptionTv();
```

위의 코드에서 `CaptionTv`인스턴스 2개를 생성하고, 참조변수 `c`와 `t`가 생성된 인스턴스를 하나씩 참조하도록 하였다. 이 경우 실제 인스턴스가 `CaptionTv`타입이라 할지라도, 참조변수 `t`로는 `CaptionTv`인스턴스의 모든 멤버를 사용할 수 없다.

`Tv`타입의 참조변수로는 `CaptionTv`인스턴스 중에서 `Tv`클래스의 멤버들(상속받은 멤버 포함)만 사용할 수 있다. 따라서, 생성된 `CaptionTv`인스턴스의 멤버 중에서 `Tv`클래스에 정의 되지 않은 멤버, `text`와 `caption()`은 참조변수 `t`로 사용이 불가능하다. 즉, `t.text` 또는 `t.caption()`과 같이 할 수 없다는 것이다. 둘 다 같은 타입의 인스턴스지만 참조변수의 타입에 따라 사용할 수 있는 멤버의 개수가 달라진다.



참고! 실제로는 모든 클래스의 최고조상인 `Object`클래스로부터 상속받은 부분도 포함되어야 하지만 간단히 하기 위해 생략했다.

반대로 아래와 같이 자손타입의 참조변수로 조상타입의 인스턴스를 참조하는 것은 가능할까?

```
CaptionTv c = new Tv();
```

그렇지 않다. 위의 코드를 컴파일 하면 에러가 발생한다. 그 이유는 실제 인스턴스인 Tv의 멤버 개수보다 참조변수 c가 사용할 수 있는 멤버 개수가 더 많기 때문이다. 그래서 이를 허용하지 않는다.

CaptionTv 클래스에는 text와 caption()이 정의되어 있으므로 참조변수 c로는 c.text, c.caption()과 같은 방식으로 c가 참조하고 있는 인스턴스에서 text와 caption()을 사용하여 할 수 있다.

하지만, c가 참조하고 있는 인스턴스는 Tv타입이고, Tv타입의 인스턴스에는 text와 caption()이 존재하지 않기 때문에 이들을 사용하려 하면 문제가 발생한다.

그래서, 자손타입의 참조변수로 조상타입의 인스턴스를 참조하는 것은 존재하지 않는 멤버를 사용하고자 할 가능성이 있으므로 허용하지 않는 것이다. 참조변수가 사용할 수 있는 멤버의 개수는 인스턴스의 멤버 개수보다 같거나 적어야 한다.

| 참고 | 클래스는 상속을 통해서 확장될 수는 있어도 축소될 수는 없어서, 조상 인스턴스의 멤버 개수보다 항상 적거나 같다.

참조변수의 타입이 참조변수가 참조하고 있는 인스턴스에서 사용할 수 있는 멤버의 개수를 결정한다는 사실을 이해하는 것은 매우 중요하다.

그렇다면, 인스턴스의 타입과 일치하는 참조변수를 사용하면 인스턴스의 멤버들을 모두 사용할 수 있을 텐데 왜 조상타입의 참조변수를 사용해서 인스턴스의 일부 멤버만을 사용하도록 할까?

이에 대한 답은 앞으로 배우게 될 것이며, 지금은 조상타입의 참조변수로도 자손클래스의 인스턴스를 참조할 수 있다는 것과 그 차이에 대해서만 이해하면 된다.

| 참고 | 모든 참조변수는 null 또는 4 byte의 주소값이 저장되며, 참조변수의 타입은 참조할 수 있는 객체의 종류와 사용할 수 있는 멤버의 수를 결정한다.

조상타입의 참조변수로 자손타입의 인스턴스를 참조할 수 있다.

반대로 자손타입의 참조변수로 조상타입의 인스턴스를 참조할 수는 없다.

5.2 참조변수의 형변환

기본형 변수와 같이 참조변수도 형변환이 가능하다. 단, 서로 상속관계에 있는 클래스사이에서만 가능하기 때문에 자손타입의 참조변수를 조상타입의 참조변수로, 조상타입의 참조변수를 자손타입의 참조변수로의 형변환만 가능하다.

| 참고 | 바로 윗 조상이나 자손이 아닌, 조상의 조상으로도 형변환이 가능하다. 따라서 모든 참조변수는 모든 클래스의 조상인 Object 클래스 타입으로 형변환이 가능하다.

기본형 변수의 형변환에서 작은 자료형에서 큰 자료형의 형변환은 생략이 가능하듯이, 참조형 변수의 형변환에서는 자손타입의 참조변수를 조상타입으로 형변환하는 경우에는 형변환을 생략할 수 있다.

자손타입 → 조상타입(Up-casting) : 형변환 생략 가능
자손타입 ← 조상타입(Down-casting) : 형변환 생략 불가

조상타입의 참조변수를 자손타입의 참조변수로 변환하는 것을 다운캐스팅(down-casting)이라고 하며, 자손타입의 참조변수를 조상타입의 참조변수로 변환하는 것을 업캐스팅(up-casting)이라고 한다.

참조변수간의 형변환 역시 캐스트연산자를 사용하며, 괄호()안에 변환하고자 하는 타입의 이름(클래스명)을 적어주면 된다.

```

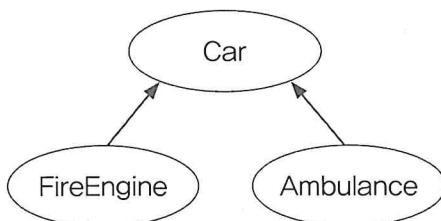
class Car {
    String color;
    int door;
    void drive() { // 운전하는 기능
        System.out.println("drive, Brrrr~");
    }
    void stop() { // 멈추는 기능
        System.out.println("stop!!!");
    }
}

class FireEngine extends Car { // 소방차
    void water() { // 물 뿌리는 기능
        System.out.println("water!!!");
    }
}

class Ambulance extends Car { // 앰뷸런스
    void siren() { // 사이렌을 울리는 기능
        System.out.println("siren~~~");
    }
}

```

이와 같이 세 클래스, Car, FireEngine, Ambulance가 정의되어 있을 때, 이 세 클래스간의 관계를 그림으로 표현하면 아래와 같다.



| 참고 | 이처럼 클래스 간의 상속관계를 그림으로 나타내 보면, 형변환의 가능여부를 쉽게 확인할 수 있다.

Car클래스는 FireEngine클래스와 Ambulance클래스의 조상이다. 그렇다고 해서 Fire Engine클래스와 Ambulance클래스가 형제관계는 아니다. 자바에서는 조상과 자식관계만 존재하기 때문에 FireEngine클래스와 Ambulance클래스는 서로 아무런 관계가 없다.

따라서 Car타입의 참조변수와 FireEngine타입의 참조변수 그리고 Car타입의 참조변수와 Ambulance타입의 참조변수 간에는 서로 형변환이 가능하지만, FireEngine타입의 참조변수와 Ambulance타입의 참조변수 간에는 서로 형변환이 가능하지 않다.

```
FireEngine f;
Ambulance a;
a = (Ambulance) f;           // 에러. 상속관계가 아닌 클래스간의 형변환 불가
f = (FireEngine) a;          // 에러. 상속관계가 아닌 클래스간의 형변환 불가
```

먼저 Car타입 참조변수와 FireEngine타입 참조변수 간의 형변환을 예로 들어보자.

```
Car car = null;
FireEngine fe = new FireEngine();
FireEngine fe2 = null;

car = fe;                  // car = (Car) fe;에서 형변환 생략됨. 업캐스팅
fe2 = (FireEngine) car;    // 형변환을 생략불가. 다운 캐스팅
```

참조변수 car와 fe의 타입이 서로 다르기 때문에, 대입연산(=)이 수행되기 전에 형변환을 수행하여 두 변수간의 타입을 맞춰주어야 한다.

그러나, 자손타입의 참조변수를 조상타입의 참조변수에 할당할 경우 형변환을 생략할 수 있어서 ‘car = fe;’와 같이 하였다. 원칙적으로는 ‘car = (Car)fe;’와 같이 해야 한다.

반대로 조상타입의 참조변수를 자손타입의 참조변수에 저장할 경우 형변환을 생략할 수 없으므로, ‘fe2 = (FireEngine)car;’와 같이 명시적으로 형변환을 해주어야 한다.

참고로 형변환을 생략할 수 있는 경우와 생략할 수 없는 경우에 대한 이유를 설명하자면 다음과 같다.

Car타입의 참조변수 c가 있다고 가정하자. 참조변수 c가 참조하고 있는 인스턴스는 아마도 Car인스턴스이거나 자손인 FireEngine인스턴스일 것이다.

Car타입의 참조변수 c를 Car타입의 조상인 Object타입의 참조변수로 형변환 하는 것은 참조변수가 다를 수 있는 멤버의 개수가 실제 인스턴스가 갖고 있는 멤버의 개수보다 적을 것이 분명하므로 문제가 되지 않는다. 그래서 형변환을 생략할 수 있도록 한 것이다.

하지만, Car타입의 참조변수 c를 자손인 FireEngine타입으로 변환하는 것은 참조변수가 다를 수 있는 멤버의 개수를 늘이는 것이므로, 실제 인스턴스의 멤버 개수보다 참조변수가 사용할 수 있는 멤버의 개수가 더 많아지므로 문제가 발생할 가능성성이 있다.

그래서 자손타입으로의 형변환은 생략할 수 없으며, 형변환을 수행하기 전에 instanceof 연산자를 사용해서 참조변수가 참조하고 있는 실제 인스턴스의 타입을 확인하는 것이 안전하다.

형변환은 참조변수의 타입을 변환하는 것이지 인스턴스를 변환하는 것은 아니기 때문에 참조변수의 형변환은 인스턴스에 아무런 영향을 미치지 않는다.

단지 참조변수의 형변환을 통해서, 참조하고 있는 인스턴스에서 사용할 수 있는 멤버의 범위(개수)를 조절하는 것뿐이다.

전에 예로 든 'Tv t = new CaptionTv();'도 'Tv t = (Tv)new CaptionTv();'의 생략된 형태이다. 이해가 잘 안 간다면, 'Tv t = (Tv)new CaptionTv();'는 아래의 두 줄을 간략히 한 것이라고 생각하면 이해하기 쉽다.

```
CaptionTv c = new CaptionTv();
Tv t = (Tv)c;
```

▼ 예제 7-15/ch7/CastingTest1.java

```
class CastingTest1 {
    public static void main(String args[]) {
        Car car = null;
        FireEngine fe = new FireEngine();
        FireEngine fe2 = null;

        fe.water();
        car = fe;      // car = (Car) fe;에서 형변환이 생략된 형태다.
        // car.water(); ←
        fe2 = (FireEngine)car; // 자손타입 ← 조상타입
        fe2.water();
    }
}

class Car {
    String color;
    int door;

    void drive() {      // 운전하는 기능
        System.out.println("drive, Brrrr~");
    }

    void stop() {       // 멈추는 기능
        System.out.println("stop!!!");
    }
}
class FireEngine extends Car {      // 소방차
    void water() {      // 물을 뿌리는 기능
        System.out.println("water!!!");
    }
}
```

컴파일 에러!!! Car타입의
참조변수로는 water()를
호출할 수 없다.

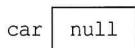
▼ 실행결과

water!!!
water!!!

이 예제의 주요 실행과정을 그림과 함께 자세히 살펴보자.

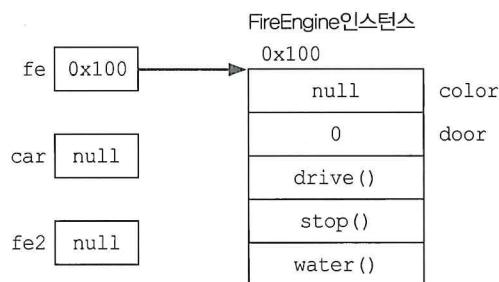
```
1. Car car = null;
```

Car타입의 참조변수 car를 선언하고 null로 초기화한다.



```
2. FireEngine fe = new FireEngine();
```

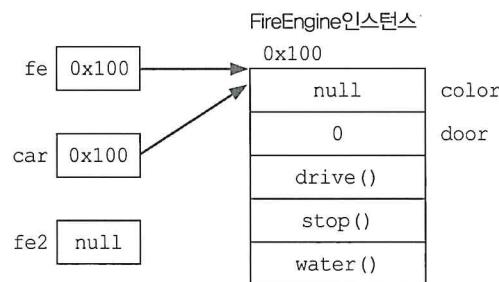
FireEngine 인스턴스를 생성하고 FireEngine 타입의 참조변수가 참조하도록 한다.



3. **car = fe;** // 조상 타입 ← 자손 타입

참조변수 `fe`가 참조하고 있는 인스턴스를 참조변수 `car`가 참조하도록 한다. `fe`의 값(`fe`가 참조하고 있는 인스턴스의 주소)이 `car`에 저장된다. 이 때 두 참조변수의 타입이 다르므로 참조변수 `fe`가 형변환되어야 하지만 생략되었다.

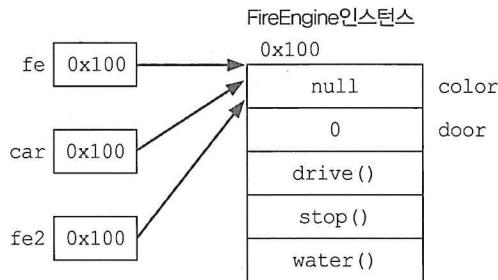
이제는 참조변수 car를 통해서도 FireEngine인스턴스를 사용할 수 있지만, fe와는 달리 car는 Car타입이므로 Car클래스의 멤버가 아닌 water()는 사용할 수 없다.



4. fe2 = (FireEngine) car; // 자손 타입 ← 조상 타입

참조변수 car가 참조하고 있는 인스턴스를 참조변수 fe2가 참조하도록 한다. 이 때 두 참조변수의 타입이 다르므로 참조변수 car를 형변환하였다. car에는 FireEngine인스턴스의 주소가 저장되어 있으므로 fe2에도 FireEngine인스턴스의 주소가 저장된다.

이제는 참조변수 fe2를 통해서도 FireEngine인스턴스를 사용할 수 있지만, car외는 달리, fe2는 FireEngine타입이므로 FireEngine인스턴스의 모든 멤버들을 사용할 수 있다.



▼ 예제 7-16/ch7/CastingTest2.java

```

class CastingTest2 {
    public static void main(String args[]) {
        Car car = new Car();
        Car car2 = null;
        FireEngine fe = null;

        car.drive();
        fe = (FireEngine)car;           // 8번째 줄. 컴파일은 OK. 실행 시 에러가 발생
        fe.drive();
        car2 = fe;
        car2.drive();
    }
}

```

▼ 실행결과
drive, Brrrr~
java.lang.ClassCastException: Car
at CastingTest2.main(CastingTest2.java:8)

이 예제는 컴파일은 성공하지만, 실행 시 에러(ClassCastException)가 발생한다. 에러가 발생한 곳은 문장은 ‘CastingTest2.java’의 8번째 라인인 ‘fe = (FireEngine)car;’이며, 발생이유는 형변환에 오류가 있기 때문이다. 캐스트 연산자를 이용해서 조상타입의 참조 변수를 자손타입의 참조변수로 형변환한 것이기 때문에 문제가 없어 보이지만, 문제는 참조변수 car가 참조하고 있는 인스턴스가 Car타입의 인스턴스라는데 있다. 전에 배운 것처럼 조상타입의 인스턴스를 자손타입의 참조변수로 참조하는 것은 허용되지 않는다.

위의 예제에서 ‘Car car = new Car();’를 ‘Car car = new FireEngine();’와 같이 변경하면, 컴파일할 때 뿐만 아니라 실행할 때도 에러가 발생하지 않을 것이다.

컴파일 시에는 참조변수간의 타입만 체크하기 때문에 실행 시 생성될 인스턴스의 타입에 대해서는 전혀 알지 못한다. 그래서 컴파일 시에는 문제가 없었지만, 실행 시에는 에러가 발생하여 실행이 비정상적으로 종료된 것이다.

서로 상속관계에 있는 타입간의 형변환은 양방향으로 자유롭게 수행될 수 있으나, 참조변수가 가리키는 인스턴스의 자손타입으로 형변환은 허용되지 않는다.
 그래서 참조변수가 가리키는 인스턴스의 타입이 무엇인지 확인하는 것이 중요하다.

5.3 instanceof연산자

참조변수가 참조하고 있는 인스턴스의 실제 타입을 알아보기 위해 instanceof연산자를 사용한다. 주로 조건문에 사용되며, instanceof의 왼쪽에는 참조변수를 오른쪽에는 타입(클래스명)이 피연산자로 위치한다. 그리고 연산의 결과로 boolean값인 true와 false 중의 하나를 반환한다.

instanceof를 이용한 연산결과로 true를 얻었다는 것은 참조변수가 검사한 타입으로 형변환이 가능하다는 것을 뜻한다.

| 참고 | 값이 null인 참조변수에 대해 instanceof연산을 수행하면 false를 결과로 얻는다.

```
void doWork(Car c) {
    if (c instanceof FireEngine) {
        FireEngine fe = (FireEngine)c;
        fe.water();
        ...
    } else if (c instanceof Ambulance) {
        Ambulance a = (Ambulance)c;
        a.siren();
        ...
    }
    ...
}
```

위의 코드는 Car타입의 참조변수 c를 매개변수로 하는 메서드이다. 이 메서드가 호출될 때, 매개변수로 Car클래스 또는 그 자손 클래스의 인스턴스를 넘겨받겠지만 메서드 내에서는 정확히 어떤 인스턴스인지 알 길이 없다. 그래서 instanceof연산자를 이용해서 참조변수 c가 가리키고 있는 인스턴스의 타입을 체크하고, 적절히 형변환한 다음에 작업을 해야 한다.

조상타입의 참조변수로 자손타입의 인스턴스를 참조할 수 있기 때문에, 참조변수의 타입과 인스턴스의 타입이 항상 일치하지는 않는다는 것을 배웠다. 조상타입의 참조변수로는 실제 인스턴스의 멤버들을 모두 사용할 수 없기 때문에, 실제 인스턴스와 같은 타입의 참조변수로 형변환을 해야만 인스턴스의 모든 멤버들을 사용할 수 있다.

▼ 예제 7-17/ch7/InstanceOfTest.java

```
class InstanceofTest {
    public static void main(String args[]) {
        FireEngine fe = new FireEngine();

        if(fe instanceof FireEngine) {
            System.out.println("This is a FireEngine instance.");
        }

        if(fe instanceof Car) {
```

```

        System.out.println("This is a Car instance.");
    }

    if(fe instanceof Object) {
        System.out.println("This is an Object instance.");
    }

    System.out.println(fe.getClass().getName()); // 클래스의 이름을 출력
}
} // class
class Car {}
class FireEngine extends Car {}

```

▼ 실행결과

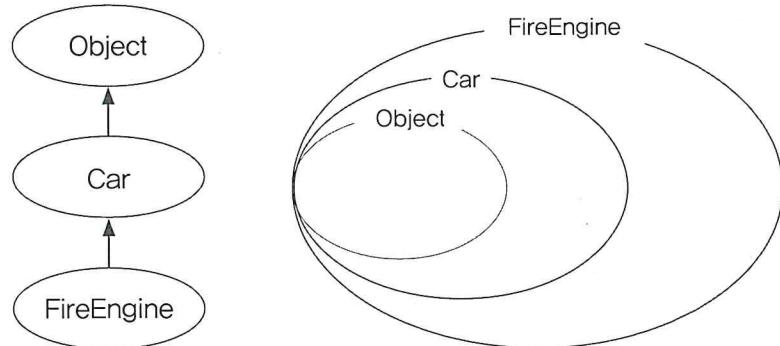
```

This is a FireEngine instance.
This is a Car instance.
This is an Object instance.
FireEngine

```

생성된 인스턴스는 FireEngine타입인데도, Object타입과 Car타입의 instanceof연산에서도 true를 결과로 얻었다. 그 이유는 FireEngine클래스는 Object클래스와 Car클래스의 자손 클래스이므로 조상의 멤버들을 상속받았기 때문에, FireEngine인스턴스는 Object인스턴스와 Car인스턴스를 포함하고 있는 셈이기 때문이다.

요약하면, 실제 인스턴스와 같은 타입의 instanceof연산 이외에 조상타입의 instanceof연산에도 true를 결과로 얻으면, instanceof연산의 결과가 true라는 것은 검사한 타입으로의 형변환을 해도 아무런 문제가 없다는 뜻이다.



'참조변수.getClass().getName()'은 참조변수가 가리키고 있는 인스턴스의 클래스 이름을 문자열(String)로 반환한다. getClass()에 대한 자세한 내용은 9장에서 배우게 된다.

어떤 타입에 대한 instanceof연산의 결과가 true라는 것은 검사한 타입으로 형변환이 가능하다는 것을 뜻한다.

5.4 참조변수와 인스턴스의 연결

조상 타입의 참조변수와 자손 타입의 참조변수의 차이점이 사용할 수 있는 멤버의 개수에 있다고 배웠다. 여기서 한 가지 더 알아두어야 할 내용이 있다.

조상 클래스에 선언된 멤버변수와 같은 이름의 인스턴스변수를 자손 클래스에 중복으로 정의했을 때, 조상타입의 참조변수로 자손 인스턴스를 참조하는 경우와 자손타입의 참조 변수로 자손 인스턴스를 참조하는 경우는 서로 다른 결과를 얻는다.

메서드의 경우 조상 클래스의 메서드를 자손의 클래스에서 오버라이딩한 경우에도 참조 변수의 타입에 관계없이 항상 실제 인스턴스의 메서드(오버라이딩된 메서드)가 호출되지만, 멤버변수의 경우 참조변수의 타입에 따라 달라진다.

| 참고 | static메서드는 static변수처럼 참조변수의 타입에 영향을 받는다. 참조변수의 타입에 영향을 받지 않는 것은 인스턴스메서드 뿐이다. 그래서 static메서드는 반드시 참조변수가 아닌 '클래스이름.메서드()'로 호출해야 한다.

결론부터 말하자면, 멤버변수가 조상 클래스와 자손 클래스에 중복으로 정의된 경우, 조상타입의 참조변수를 사용했을 때는 조상 클래스에 선언된 멤버변수가 사용되고, 자손타입의 참조변수를 사용했을 때는 자손 클래스에 선언된 멤버변수가 사용된다.

하지만 중복 정의되지 않은 경우, 조상타입의 참조변수를 사용했을 때와 자손타입의 참조변수를 사용했을 때의 차이는 없다. 중복된 경우는 참조변수의 타입에 따라 달라지지만, 중복되지 않은 경우 하나뿐이므로 선택의 여지가 없기 때문이다.

▼ 예제 7-18/ch7/BindingTest.java

```
class BindingTest{
    public static void main(String[] args) {
        Parent p = new Child();
        Child c = new Child();

        System.out.println("p.x = " + p.x);
        p.method();

        System.out.println("c.x = " + c.x);
        c.method();
    }
}

class Parent {
    int x = 100;

    void method() {
        System.out.println("Parent Method");
    }
}

class Child extends Parent {
    int x = 200;

    void method() {
        System.out.println("Child Method");
    }
}
```

▼ 실행결과
p.x = 100 Child Method c.x = 200 Child Method

타입은 다르지만, 참조변수 p와 c 모두 Child인스턴스를 참조하고 있다. 그리고 Parent클래스와 Child클래스는 서로 같은 멤버들을 정의하고 있다.

이 때 조상타입의 참조변수 p로 Child인스턴스의 멤버들을 사용하는 것과 자손타입의 참조변수 c로 Child인스턴스의 멤버들을 사용하는 것의 차이를 알 수 있다.

메서드인 method()의 경우 참조변수의 타입에 관계없이 항상 실제 인스턴스의 타입인 Child클래스에 정의된 메서드가 호출되지만, 인스턴스변수인 x는 참조변수의 타입에 따라서 달라진다.

▼ 예제 7-19/ch7/BindingTest2.java

```
class BindingTest2 {
    public static void main(String[] args) {
        Parent p = new Child();
        Child c = new Child();

        System.out.println("p.x = " + p.x);
        p.method();

        System.out.println("c.x = " + c.x);
        c.method();
    }
}

class Parent {
    int x = 100;

    void method() {
        System.out.println("Parent Method");
    }
}

class Child extends Parent { }
```

▼ 실행결과
p.x = 100 Parent Method c.x = 100 Parent Method

이전의 예제와는 달리 Child클래스에는 아무런 멤버도 정의되어 있지 않고 단순히 조상으로부터 멤버들을 상속받는다. 그렇기 때문에 참조변수의 타입에 관계없이 조상의 멤버들을 사용하게 된다.

이처럼 자손 클래스에서 조상 클래스의 멤버를 중복으로 정의하지 않았을 때는 참조변수의 타입에 따른 변화는 없다. 어느 클래스의 멤버가 호출되어야 할지, 즉 조상의 멤버가 호출되어야 할지, 자손의 멤버가 호출되어야 할지에 대해 선택의 여지가 없기 때문이다.

참조변수의 타입에 따라 결과가 달라지는 경우는 조상 클래스의 멤버변수와 같은 이름의 멤버변수를 자손 클래스에 중복해서 정의한 경우뿐이다.

▼ 예제 7-20/ch7/BindingTest3.java

```
class BindingTest3{
    public static void main(String[] args) {
        Parent p = new Child();
        Child c = new Child();

        System.out.println("p.x = " + p.x);
        p.method();
        System.out.println();
        System.out.println("c.x = " + c.x);
        c.method();
    }
}

class Parent {
    int x = 100;

    void method() {
        System.out.println("Parent Method");
    }
}

class Child extends Parent {
    int x = 200;

    void method() {
        System.out.println("x=" + x); // this.x와 같다.
        System.out.println("super.x=" + super.x);
        System.out.println("this.x=" + this.x);
    }
}
```

▼ 실행결과
p.x = 100 x=200 super.x=100 this.x=200
c.x = 200 x=200 super.x=100 this.x=200

자손 클래스 Child에 선언된 인스턴스변수 x와 조상 클래스 Parent로부터 상속받은 인스턴스변수 x를 구분하는데 참조변수 super와 this가 사용된다. 자손인 Child클래스에서의 super.x는 조상 클래스인 Parent에 선언된 인스턴스변수 x를 뜻하며, this.x 또는 x는 Child클래스의 인스턴스변수 x를 뜻한다. 그래서 위 결과에서 x와 this.x의 값이 같다.

전에 배운 것과 같이 멤버변수들은 주로 private으로 접근을 제한하고, 외부에서는 매서드를 통해서만 멤버변수에 접근할 수 있도록 하지, 이번 예제에서처럼 다른 외부 클래스에서 참조변수를 통해 직접적으로 인스턴스변수에 접근할 수 있게 하지 않는다.

예제에서 알 수 있듯이 인스턴스변수에 직접 접근하면, 참조변수의 타입에 따라 사용되는 인스턴스변수가 달라질 수 있으므로 주의해야 한다.

5.5 매개변수의 다형성

참조변수의 다형적인 특징은 메서드의 매개변수에도 적용된다. 아래와 같이 Product, Tv, Computer, Audio, Buyer클래스가 정의되어 있다고 가정하자.

```
class Product {
    int price;                                // 제품의 가격
    int bonusPoint;                            // 제품구매 시 제공하는 보너스점수
}
class Tv          extends Product {}
class Computer   extends Product {}
class Audio       extends Product {}

class Buyer {
    int money = 1000;                         // 고객, 물건을 사는 사람
    int bonusPoint = 0;                        // 소유금액
                                                // 보너스점수
}
```

Product클래스는 Tv, Audio, Computer클래스의 조상이며, Buyer클래스는 제품(Product)을 구입하는 사람을 클래스로 표현한 것이다.

Buyer클래스에 물건을 구입하는 기능의 메서드를 추가해보자. 구입할 대상이 필요하므로 매개변수로 구입할 제품을 넘겨받아야 한다. Tv를 살 수 있도록 매개변수를 Tv타입으로 하였다.

```
void buy(Tv t) {
    // Buyer가 가진 돈(money)에서 제품의 가격(t.price)만큼 뺀다.
    money = money - t.price;

    // Buyer의 보너스점수(bonusPoint)에 제품의 보너스점수(t.bonusPoint)를 더한다.
    bonusPoint = bonusPoint + t.bonusPoint;
}
```

buy(Tv t)는 제품을 구입하면 제품을 구입한 사람이 가진 돈에서 제품의 가격을 빼고, 보너스점수는 추가하는 작업을 하도록 작성되었다. 그런데 buy(Tv t)로는 Tv밖에 살 수 없기 때문에 아래와 같이 다른 제품들도 구입할 수 있는 메서드가 추가로 필요하다.

```
void buy(Computer c) {
    money = money - c.price;
    bonusPoint = bonusPoint + c.bonusPoint;
}

void buy(Audio a) {
    money = money - a.price;
    bonusPoint = bonusPoint + a.bonusPoint;
}
```

이렇게 되면, 제품의 종류가 늘어날 때마다 Buyer클래스에는 새로운 buy메서드를 추가해 주어야 할 것이다.

그러나 메서드의 매개변수에 다형성을 적용하면 아래와 같이 하나의 메서드로 간단히 처리할 수 있다.

```
void buy(Product p) {
    money = money - p.price;
    bonusPoint = bonusPoint + p.bonusPoint;
}
```

매개변수가 Product타입의 참조변수라는 것은, 메서드의 매개변수로 Product클래스의 자손타입의 참조변수면 어느 것이나 매개변수로 받아들일 수 있다는 뜻이다.

그리고 Product클래스에 price와 bonusPoint가 선언되어 있기 때문에 참조변수 p로 인스턴스의 price와 bonusPoint를 사용할 수 있기에 이와 같이 할 수 있다.

앞으로 다른 제품 클래스를 추가할 때 Product클래스를 상속받기만 하면, buy(Product p)메서드의 매개변수로 받아들여질 수 있다.

```
Buyer b = new Buyer();
Tv t = new Tv();
Computer c = new Computer();
b.buy(t);
b.buy(c);
```

| 참고 | Tv t = new Tv(); b.buy(t);를 한 문장으로 줄이면 b.buy(new Tv());가 된다.

Tv클래스와 Computer클래스는 Product클래스의 자손이므로 위의 코드와 같이 buy(Product p)메서드에 매개변수로 Tv인스턴스와 Computer인스턴스를 제공하는 것이 가능하다.

▼ 예제 7-21/ch7/PolyArgumentTest.java

```
class Product {
    int price;           // 제품의 가격
    int bonusPoint;     // 제품구매 시 제공하는 보너스점수

    Product(int price) {
        this.price = price;
        bonusPoint = (int)(price/10.0); // 보너스점수는 제품가격의 10%
    }
}

class Tv extends Product {
    Tv() {
        // 조상클래스의 생성자 Product(int price)를 호출한다.
        super(100);      // Tv의 가격을 100만원으로 한다.
    }
}
```

```

// Object클래스의 toString()을 오버라이딩한다.
public String toString() { return "Tv"; }
}

class Computer extends Product {
    Computer() { super(200); }

    public String toString() { return "Computer"; }
}

class Buyer {           // 고객, 물건을 사는 사람
    int money = 1000;   // 소유금액
    int bonusPoint = 0; // 보너스점수

    void buy(Product p) {
        if(money < p.price) {
            System.out.println("잔액이 부족하여 물건을 살 수 없습니다.");
            return;
        }

        money -= p.price;           // 가진 돈에서 구입한 제품의 가격을 뺀다.
        bonusPoint += p.bonusPoint; // 제품의 보너스 점수를 추가한다.
        System.out.println(p + "을/를 구입하셨습니다.");
    }
}

class PolyArgumentTest {
    public static void main(String args[]) {
        Buyer b = new Buyer();

        b.buy(new Tv());
        b.buy(new Computer());

        System.out.println("현재 남은 돈은 " + b.money + "만원입니다.");
        System.out.println("현재 보너스점수는 " + b.bonusPoint + "점입니다.");
    }
}

```

▼ 실행결과

Tv을/를 구입하셨습니다.
 Computer을/를 구입하셨습니다.
 현재 남은 돈은 700만원입니다.
 현재 보너스점수는 30점입니다.

고객(Buyer)이 buy(Product p)메서드를 이용해서 Tv와 Computer를 구입하고, 고객의 잔고와 보너스점수를 출력하는 예제이다.

매개변수의 다형성의 또 다른 예로 PrintStream클래스에 정의되어 있는 print(Object o) 메서드를 살펴보자. print(Object o)는 매개변수로 Object타입의 변수가 선언되어 있는데 Object클래스는 모든 클래스의 조상이므로 이 메서드의 매개변수로 어떤 타입의 인스턴스도 가능하므로, 이 하나의 메서드로 모든 타입의 인스턴스를 처리할 수 있는 것이다.

이 메서드는 매개변수에 toString()을 호출하여 문자열을 얻어서 출력한다. 실제 코드는 아래와 같다.

```

public void print(Object obj) {
    write(String.valueOf(obj)); // valueof()가 반환한 문자열을 출력한다.
}

public static String valueOf(Object obj) {
    return (obj == null) ? "null" : obj.toString(); // 문자열을 반환한다.
}

```

5.6 여러 종류의 객체를 배열로 다루기

조상타입의 참조변수로 자손타입의 객체를 참조하는 것이 가능하므로, Product클래스가 Tv, Computer, Audio클래스의 조상일 때, 다음과 같이 할 수 있는 것을 이미 배웠다.

```
Product p1 = new Tv();
Product p2 = new Computer();
Product p3 = new Audio();
```

위의 코드를 Product타입의 참조변수 배열로 처리하면 아래와 같다.

```
Product p[] = new Product[3];
p[0] = new Tv();
p[1] = new Computer();
p[2] = new Audio();
```

이처럼 조상타입의 참조변수 배열을 사용하면, 공통의 조상을 가진 서로 다른 종류의 객체를 배열로 묶어서 다룰 수 있다. 또는 묶어서 다루고 싶은 객체들의 상속관계를 따져서 가장 가까운 공통조상 클래스 타입의 참조변수 배열을 생성해서 객체들을 저장하면 된다.

이러한 특징을 이용해서 예제7-21의 Buyer클래스에 구입한 제품을 저장하기 위한 Product배열을 추가해보도록 하자.

```
class Buyer {
    int money = 1000;
    int bonusPoint = 0;
    Product[] item = new Product[10]; // 구입한 제품을 저장하기 위한 배열
    int i = 0; // Product배열 item에 사용될 index

    void buy(Product p) {
        if(money < p.price) {
            System.out.println("잔액이 부족하여 물건을 살수 없습니다.");
            return;
        }

        money -= p.price; // 가진 돈에서 제품가격을 뺀다.
        bonusPoint += p.bonusPoint; // 제품의 보너스포인트를 더한다.
        item[i++] = p; // 제품을 Product[] item에 저장한다.
        System.out.println(p + "을/를 구입하셨습니다.");
    }
}
```

구입한 제품을 담기 위해 Buyer클래스에 Product배열인 item을 추가해주었다. 그리고 buy메서드에 ‘item[i++] = p;’문장을 추가함으로써 물건을 구입하면, 배열 item에 저장되도록 했다.

이렇게 함으로써, 모든 제품클래스의 조상인 Product클래스 타입의 배열을 사용함으로써 구입한 제품을 하나의 배열로 간단하게 다룰 수 있게 된다.

▼ 예제 7-22/ch7/PolyArgumentTest2.java

```

class Product {
    int price;           // 제품의 가격
    int bonusPoint;     // 제품구매 시 제공하는 보너스점수

    Product(int price) {
        this.price = price;
        bonusPoint =(int) (price/10.0);
    }

    Product() {} // 기본 생성자
}

class Tv extends Product {
    Tv() { super(100); } •———— 조상클래스의 생성자
    public String toString() { return "Tv"; }
}

class Computer extends Product {
    Computer() { super(200); }
    public String toString() { return "Computer"; }
}

class Audio extends Product {
    Audio() { super(50); }
    public String toString() { return "Audio"; }
}

class Buyer {           // 고객, 물건을 사는 사람
    int money = 1000;   // 소유금액
    int bonusPoint = 0; // 보너스점수
    Product[] item = new Product[10]; // 구입한 제품을 저장하기 위한 배열
    int i =0;           // Product배열에 사용될 카운터

    void buy(Product p) {
        if(money < p.price) {
            System.out.println("잔액이 부족하여 물건을 살 수 없습니다.");
            return;
        }

        money -= p.price;           // 가진 돈에서 구입한 제품의 가격을 뺀다.
        bonusPoint += p.bonusPoint; // 제품의 보너스 점수를 추가한다.
        item[i++] = p;              // 제품을 Product[] item에 저장한다.
        System.out.println(p + "을/를 구입하셨습니다.");
    }

    void summary() {             // 구매한 물품에 대한 정보를 요약해서 보여 준다.
        int sum = 0;              // 구입한 물품의 가격합계
        String itemList ="";      // 구입한 물품목록

        // 반복문을 이용해서 구입한 물품의 총 가격과 목록을 만든다.
        for(int i=0; i<item.length;i++) {
            if(item[i]==null) break;
            sum += item[i].price;
            itemList += item[i] + ", ";
        }
        System.out.println("구입하신 물품의 총금액은 " + sum + "만원입니다.");
        System.out.println("구입하신 제품은 " + itemList + "입니다.");
    }
}

```

```

class PolyArgumentTest2 {
    public static void main(String args[]) {
        Buyer b = new Buyer();
        b.buy(new Tv());
        b.buy(new Computer());
        b.buy(new Audio());
        b.summary();
    }
}

```

▼ 실행결과

Tv을/를 구입하셨습니다.
 Computer을/를 구입하셨습니다.
 Audio을/를 구입하셨습니다.
 구입하신 물품의 총금액은 350만원입니다.
 구입하신 제품은 Tv, Computer, Audio, 입니다.

| 참고 | 구입한 제품목록의 마지막에 출력되는 콤마(.)가 눈에 거슬린다면, itemList += item[i] + ", ";를 itemList += (i==0) ? "" + item[i] : "," + item[i];과 같이 변경하자.

위 예제에서 Product배열로 구입한 제품들을 저장할 수 있도록 했지만, 배열의 크기를 10으로 했기 때문에 11개 이상의 제품을 구입할 수 없는 것이 문제다. 그렇다고 해서 배열의 크기를 무조건 크게 설정할 수만도 없는 일이다.

이런 경우, Vector클래스를 사용하면 된다. Vector클래스는 내부적으로 Object타입의 배열을 가지고 있어서, 이 배열에 객체를 추가하거나 제거할 수 있게 작성되어 있다. 그리고 배열의 크기를 알아서 관리해주기 때문에 저장할 인스턴스의 개수에 신경 쓰지 않아도 된다.

```

public class Vector extends AbstractList
    implements List, Cloneable, java.io.Serializable {
    protected Object elementData[];
    ...
}

```

Vector클래스는 이름 때문에 클래스의 기능을 오해할 수 있는데, 단지 동적으로 크기가 관리되는 객체배열일 뿐이다.

메서드 / 생성자	설명
Vector()	10개의 객체를 저장할 수 있는 Vector인스턴스를 생성한다. 10개 이상의 인스턴스가 저장되면, 자동적으로 크기가 증가된다.
boolean add(Object o)	Vector에 객체를 추가한다. 추가에 성공하면 결과값으로 true, 실패하면 false를 반환한다.
boolean remove(Object o)	Vector에 저장되어 있는 객체를 제거한다. 제거에 성공하면 true, 실패하면 false를 반환한다.
boolean isEmpty()	Vector가 비어있는지 검사한다. 비어있으면 true, 비어있지 않으면 false를 반환한다.
Object get(int index)	지정된 위치(index)의 객체를 반환한다. 반환타입이 Object타입이 므로 적절한 타입으로의 형변환이 필요하다.
int size()	Vector에 저장된 객체의 개수를 반환한다.

▲ 표7-3 Vector클래스의 주요 메서드

▼ 예제 7-23/ch7/PolyArgumentTest3.java

```

import java.util.*; // Vector클래스를 사용하기 위해서 추가해 주었다.

class Product {
    int price; // 제품의 가격
    int bonusPoint; // 제품구매 시 제공하는 보너스점수

    Product(int price) {
        this.price = price;
        bonusPoint = (int)(price/10.0);
    }

    Product() {
        price = 0;
        bonusPoint = 0;
    }
}

class Tv extends Product {
    Tv() { super(100); }
    public String toString() { return "Tv"; }
}

class Computer extends Product {
    Computer() { super(200); }
    public String toString() { return "Computer"; }
}

class Audio extends Product {
    Audio() { super(50); }
    public String toString() { return "Audio"; }
}

class Buyer { // 고객, 물건을 사는 사람
    int money = 1000; // 소유금액
    int bonusPoint = 0; // 보너스점수
    Vector item = new Vector(); // 구입한 제품을 저장하는데 사용될 Vector객체

    void buy(Product p) {
        if(money < p.price) {
            System.out.println("잔액이 부족하여 물건을 살수 없습니다. ");
            return;
        }
        money -= p.price; // 가진 돈에서 구입한 제품의 가격을 뺀다.
        bonusPoint += p.bonusPoint; // 제품의 보너스 점수를 추가한다.
        item.add(p); // 구입한 제품을 Vector에 저장한다.
        System.out.println(p + "을/를 구입하셨습니다.");
    }

    void refund(Product p) { // 구입한 제품을 환불한다.
        if(item.remove(p)) { // 제품을 Vector에서 제거한다.
            money += p.price;
            bonusPoint -= p.bonusPoint;
            System.out.println(p + "을/를 반품하셨습니다.");
        } else { // 제거에 실패한 경우
    }
}

```

```

        System.out.println("구입하신 제품 중 해당 제품이 없습니다.");
    }

}

void summary() { // 구매한 물품에 대한 정보를 요약해서 보여준다.
    int sum = 0; // 구입한 물품의 가격합계
    String itemList = ""; // 구입한 물품목록

    if(item.isEmpty()) { // Vector가 비어있는지 확인한다.
        System.out.println("구입하신 제품이 없습니다.");
        return;
    }

    // 반복문을 이용해서 구입한 물품의 총 가격과 목록을 만든다.
    for(int i=0; i<item.size();i++) {
        Product p = (Product)item.get(i); ← Vector의 i번째에 있는
        sum += p.price; 객체를 얻어온다.
        itemList += (i==0) ? "" + p : ", " + p;
    }
    System.out.println("구입하신 물품의 총금액은 " + sum + "만원입니다.");
    System.out.println("구입하신 제품은 " + itemList + "입니다.");
}
}

class PolyArgumentTest3 {
    public static void main(String args[]) {
        Buyer b = new Buyer();
        Tv tv = new Tv();
        Computer com = new Computer();
        Audio audio = new Audio();

        b.buy(tv);
        b.buy(com);
        b.buy(audio);
        b.summary();
        System.out.println();
        b.refund(com);
        b.summary();
    }
}

```

▼ 실행결과

```

Tv을/를 구입하셨습니다.
Computer을/를 구입하셨습니다.
Audio을/를 구입하셨습니다.
구입하신 물품의 총금액은 350만원입니다.
구입하신 제품은 Tv, Computer, Audio입니다.

Computer을/를 반품하셨습니다.
구입하신 물품의 총금액은 150만원입니다.
구입하신 제품은 Tv, Audio입니다.

```

구입한 물건을 다시 반환할 수 있도록 refund(Product p)를 추가하였다. 이 메서드가 호출되면, 구입물품이 저장되어 있는 item에서 해당제품을 제거한다.

| 참고 | 문자열과 참조변수의 덧셈(결합연산)은 참조변수에 toString()을 호출해서 문자열을 얻어 결합한다. 그래서 위 예제에 나오는 "" + p는 "" + p.toString()이 되고, 만일 p.toString()의 결과가 "Audio"라면 "+" + "Audio"가 되어 결국 "Audio"가 된다.

6. 추상클래스(Abstract Class)

6.1 추상클래스란?

클래스를 설계도에 비유한다면, 추상클래스는 미완성 설계도에 비유할 수 있다. 미완성 설계도란, 단어의 뜻 그대로 완성되지 못한 채로 남겨진 설계도를 말한다.

클래스가 미완성이라는 것은 멤버의 개수에 관계된 것이 아니라, 단지 미완성 메서드(추상메서드)를 포함하고 있다는 의미이다.

미완성 설계도로 완성된 제품을 만들 수 없듯이 추상클래스로 인스턴스는 생성할 수 없다. 추상클래스는 상속을 통해서 자손클래스에 의해서만 완성될 수 있다.

추상클래스 자체로는 클래스로서의 역할을 다 못하지만, 새로운 클래스를 작성하는데 있어서 바탕이 되는 조상클래스로서 중요한 의미를 갖는다. 새로운 클래스를 작성할 때 아무 것도 없는 상태에서 시작하는 것보다는 완전하지는 못하더라도 어느 정도 틀을 갖춘 상태에서 시작하는 것이 나을 것이다.

실생활에서 예를 들자면, 같은 크기의 TV라도 기능의 차이에 따라 여러 종류의 모델이 있지만, 사실 이들의 설계도는 아마 90%정도는 동일할 것이다. 서로 다른 세 개의 설계도를 따로 그리는 것보다는 이들의 공통부분만을 그린 미완성 설계도를 만들어 놓고, 이 미완성 설계도를 이용해서 각각의 설계도를 완성하는 것이 훨씬 효율적일 것이다.

추상클래스는 키워드 'abstract'를 붙이기만 하면 된다. 이렇게 함으로써 이 클래스를 사용할 때, 클래스 선언부의 abstract를 보고 이 클래스에는 추상메서드가 있으니 상속을 통해서 구현해주어야 한다는 것을 쉽게 알 수 있을 것이다.

```
abstract class 클래스이름 {
    ...
}
```

추상클래스는 추상메서드를 포함하고 있다는 것을 제외하고는 일반클래스와 전혀 다르지 않다. 추상클래스에도 생성자가 있으며, 멤버변수와 메서드도 가질 수 있다.

| 참고 | 추상메서드를 포함하고 있지 않은 클래스에도 키워드 'abstract'를 붙여서 추상클래스로 지정할 수도 있다. 추상메서드가 없는 완성된 클래스라 할지라도 추상클래스로 지정되면 클래스의 인스턴스를 생성할 수 없다.

6.2 추상메서드(Abstract Method)

메서드는 선언부와 구현부(몸통)로 구성되어 있다고 했다. 선언부만 작성하고 구현부는 작성하지 않은 채로 남겨 둔 것이 추상메서드이다. 즉, 설계만 해 놓고 실제 수행될 내용은 작성하지 않았기 때문에 미완성 메서드인 것이다.

메서드를 이와 같이 미완성 상태로 남겨 놓는 이유는 메서드의 내용이 상속받는 클래스에 따라 달라질 수 있기 때문에 조상 클래스에서는 선언부만을 작성하고, 주석을 덧붙여 어떤 기능을 수행할 목적으로 작성되었는지 알려 주고, 실제 내용은 상속받는 클래스에서 구현하도록 비워 두는 것이다. 그래서 추상클래스를 상속받는 자손 클래스는 조상의 추상 메서드를 상황에 맞게 적절히 구현해주어야 한다.

추상메서드 역시 키워드 'abstract'를 앞에 붙여 주고, 추상메서드는 구현부가 없으므로 괄호{} 대신 문장의 끝을 알리는 ';'을 적어준다.

```
/* 주석을 통해 어떤 기능을 수행할 목적으로 작성하였는지 설명한다.*/
abstract 리턴타입 메서드이름();
```

추상클래스로부터 상속받는 자손클래스는 오버라이딩을 통해 조상인 추상클래스의 추상 메서드를 모두 구현해주어야 한다. 만일 조상으로부터 상속받은 추상메서드 중 하나라도 구현하지 않는다면, 자손클래스 역시 추상클래스로 지정해 주어야 한다.

```
abstract class Player { // 추상클래스
    abstract void play(int pos); // 추상메서드
    abstract void stop(); // 추상메서드
}

class AudioPlayer extends Player {
    void play(int pos) { /* 내용 생략 */ } // 추상메서드를 구현
    void stop() { /* 내용 생략 */ } // 추상메서드를 구현
}

abstract class AbstractPlayer extends Player {
    void play(int pos) { /* 내용 생략 */ } // 추상메서드를 구현
}
```

실제 작업내용인 구현부가 없는 메서드가 무슨 의미가 있을까 싶기도 하겠지만, 메서드를 작성할 때 실제 작업내용인 구현부보다 더 중요한 부분이 선언부이다.

메서드의 이름과 메서드의 작업에 필요한 매개변수, 그리고 작업의 결과로 어떤 타입의 값을 반환할 것인가를 결정하는 것은 쉽지 않은 일이다. 선언부만 작성해도 메서드의 절반 이상이 완성된 것이라 해도 과언이 아니다.

메서드를 사용하는 쪽에서는 메서드가 실제로 어떻게 구현되어있는지 몰라도 메서드의 이름과 매개변수, 리턴타입, 즉 선언부만 알고 있으면 되므로 내용이 없을 지라도 추상메서드를 사용하는 코드를 작성하는 것이 가능하며, 실제로는 자손클래스에 구현된 완성된 메서드가 호출되도록 할 수 있다.

6.3 추상클래스의 작성

여러 클래스에 공통적으로 사용될 수 있는 클래스를 바로 작성하기도 하고, 기존의 클래스의 공통적인 부분을 뽑아서 추상클래스로 만들어 상속하도록 하는 경우도 있다. 참고로 추상의 사전적 정의는 다음과 같다.

추상[抽象] 낱낱의 구체적 표상(表象)이나 개념에서 공통된 성질을 뽑아 이를 일반적인 개념으로 파악하는 정신 작용

상속이 자손 클래스를 만드는데 조상 클래스를 사용하는 것이라면, 이와 반대로 추상화는 기존의 클래스의 공통부분을 뽑아내서 조상 클래스를 만드는 것이라고 할 수 있다.

추상화를 구체화와 반대되는 의미로 이해하면 보다 쉽게 이해할 수 있을 것이다. 상속계층도를 따라 내려갈수록 클래스는 점점 기능이 추가되어 구체화의 정도가 심해지며, 상속계층도를 따라 올라갈수록 클래스는 추상화의 정도가 심해진다고 할 수 있다. 즉, 상속계층도를 따라 내려 갈수록 세분화되며, 올라갈수록 공통요소만 남게 된다.

추상화 클래스간의 공통점을 찾아내서 공통의 조상을 만드는 작업
구체화 상속을 통해 클래스를 구현, 확장하는 작업

아래에 Player라는 추상클래스를 작성해 보았다. 이 클래스는 VCR이나 Audio와 같은 재생 가능한 기기(Player)를 클래스로 작성할 때, 이들의 조상으로 사용될 수 있을 것이다.

```

abstract class Player {
    boolean pause;           // 일시정지 상태를 저장하기 위한 변수
    int currentPos;          // 현재 Play되고 있는 위치를 저장하기 위한 변수

    Player() {               // 추상클래스도 생성자가 있어야 한다.
        pause = false;
        currentPos = 0;
    }
    /** 지정된 위치(pos)에서 재생을 시작하는 기능이 수행하도록 작성되어야 한다. */
    abstract void play(int pos);      // 추상메서드
    /** 재생을 즉시 멈추는 기능을 수행하도록 작성되어야 한다. */
    abstract void stop();            // 추상메서드

    void play() {
        play(currentPos);          // 추상메서드를 사용할 수 있다.
    }

    void pause() {
        if(pause) { // pause가 true일 때 (정지상태)에서 pause가 호출되면,
            pause = false;         // pause의 상태를 false로 바꾸고,
            play(currentPos);       // 현재의 위치에서 play를 한다.
        } else { // pause가 false일 때 (play상태)에서 pause가 호출되면,
            pause = true;          // pause의 상태를 true로 바꾸고
            stop();                 // play를 멈춘다.
        }
    }
}

```

이제 Player클래스를 조상으로 하는 CDPlayer 클래스를 만들어 보자.

```
class CDPlayer extends Player {
    void play(int currentPos) {
        /* 조상의 추상메서드를 구현. 내용 생략 */
    }

    void stop() {
        /* 조상의 추상메서드를 구현. 내용 생략 */
    }

    // CDPlayer클래스에 추가로 정의된 멤버
    int currentTrack; // 현재 재생 중인 트랙

    void nextTrack() {
        currentTrack++;
        ...
    }

    void preTrack() {
        if(currentTrack > 1) {
            currentTrack--;
        }
        ...
    }
}
```

조상 클래스의 추상메서드를 CDPlayer클래스의 기능에 맞게 완성해주고, CDPlayer만의 새로운 기능들을 추가하였다.

사실 Player클래스의 play(int pos)와 stop()을 추상메서드로 하는 대신, 아무 내용도 없는 메서드로 작성할 수도 있다. 아무런 내용도 없이 단지 팔호{}만 있어도, 추상메서드가 아닌 일반 메서드로 간주되기 때문이다.

```
class Player {
    ...
    void play(int pos) {}
    void stop() {}

    ...
}
```

어차피 자손 클래스에서 오버라이딩하여 자신의 클래스에 맞게 구현할 테니 추상메서드로 선언하는 것과 내용없는 빈 몸통만 만들어 놓는 것이나 별 차이가 없어 보인다.

그래도 굳이 abstract를 붙여서 추상메서드로 선언하는 이유는 자손 클래스에서 추상메서드를 반드시 구현하도록 강요하기 위해서이다.

만일 추상메서드로 정의되어 있지 않고 위와 같이 빈 몸통만 가지도록 정의되어 있다면, 상속받는 자손 클래스에서는 이 메서드들이 온전히 구현된 것으로 인식하고 오버라이딩을 통해 자신의 클래스에 맞도록 구현하지 않을 수도 있기 때문이다.

하지만 abstract를 사용해서 추상메서드로 정의해놓으면, 자손 클래스를 작성할 때 이들

이 추상메서드이므로 내용을 구현해주어야 한다는 사실을 인식하고 자신의 클래스에 알맞게 구현할 것이다.

이번엔 기존의 클래스로부터 공통된 부분을 뽑아내어 추상클래스를 만들어 보도록 하자.

```
class Marine {      // 보병
    int x, y;      // 현재 위치
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stop()        { /* 현재 위치에 정지 */ }
    void stimPack()    { /* 스팀팩을 사용한다. */ }
}

class Tank {        // 탱크
    int x, y;      // 현재 위치
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stop()        { /* 현재 위치에 정지 */ }
    void changeMode() { /* 공격모드를 변환한다. */ }
}

class Dropship {   // 수송선
    int x, y;      // 현재 위치
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stop()        { /* 현재 위치에 정지 */ }
    void load()        { /* 선택된 대상을 태운다. */ }
    void unload()       { /* 선택된 대상을 내린다. */ }
}
```

유명한 컴퓨터 게임에 나오는 유닛들을 클래스로 간단히 정의해보았다. 이 유닛들은 각자 나름대로의 기능을 가지고 있지만 공통부분을 뽑아내어 하나의 클래스로 만들고, 이 클래스로부터 상속받도록 변경해보자.

```
abstract class Unit {
    int x, y;
    abstract void move(int x, int y);
    void stop() { /* 현재 위치에 정지 */ }
}

class Marine extends Unit { // 보병
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stimPack() { /* 스팀팩을 사용한다. */ }
}

class Tank extends Unit { // 탱크
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void changeMode() { /* 공격모드를 변환한다. */ }
}

class Dropship extends Unit { // 수송선
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void load() { /* 선택된 대상을 태운다. */ }
    void unload() { /* 선택된 대상을 내린다. */ }
}
```

각 클래스의 공통부분을 뽑아내서 Unit클래스를 정의하고 이로부터 상속받도록 하였다. 이 Unit클래스는 다른 유닛을 위한 클래스를 작성하는데 재활용될 수 있을 것이다. 이들 클래스에 대해서 stop메서드는 선언부와 구현부 모두 공통적이지만, Marine, Tank는 지상유닛이고 Dropship은 공중유닛이기 때문에 이동하는 방법이 서로 달라서 move메서드의 실제 구현 내용이 다를 것이다.

그래도 move메서드의 선언부는 같기 때문에 추상메서드로 정의할 수 있다. 최대한의 공통부분을 뽑아내기 위한 것이기도 하지만, 모든 유닛은 이동할 수 있어야 하므로 Unit클래스에는 move메서드가 반드시 필요한 것이기 때문이다.

move메서드가 추상메서드로 선언된 것에는, 앞으로 Unit클래스를 상속받아서 작성되는 클래스는 move메서드를 자신의 클래스에 알맞게 반드시 구현해야 한다는 의미가 담겨 있는 것이기도 하다.

```
Unit[] group = new Unit[4];
group[0] = new Marine();
group[1] = new Tank();
group[2] = new Marine();
group[3] = new Dropship();

for(int i = 0;i < group.length;i++)
    group[i].move(100, 200);
```

Unit배열의 모든 유닛을 좌표(100, 200)의 위치로 이동한다.

위의 코드는 공통조상인 Unit클래스 타입의 참조변수 배열을 통해서 서로 다른 종류의 인스턴스를 하나의 뮤음으로 다룰 수 있다는 것을 보여 주기 위한 것이다.

다형성에서 배웠듯이 조상 클래스타입의 참조변수로 자손 클래스의 인스턴스를 참조하는 것이 가능하기 때문에 이처럼 조상 클래스타입의 배열에 자손 클래스의 인스턴스를 담을 수 있는 것이다.

만일 이들 클래스간의 공통조상이 없었다면 이처럼 하나의 배열로 다룰 수 없을 것이다. Unit클래스에 move메서드가 비록 추상메서드로 정의되어 있다 하더라도 이처럼 Unit클래스 타입의 참조변수로 move메서드를 호출하는 것이 가능하다. 메서드는 참조변수의 타입에 관계없이 실제 인스턴스에 구현된 것이 호출되기 때문이다.

group[i].move(100, 200)과 같이 호출하는 것이 Unit클래스의 추상메서드인 move를 호출하는 것 같이 보이지만 실제로는 이 추상메서드가 구현된 Marine, Tank, Dropship 인스턴스의 메서드가 호출되는 것이다.

모든 클래스의 조상인 Object클래스 타입의 배열로도 서로 다른 종류의 인스턴스를 하나의 뮤음으로 다룰 수 있지만, Object클래스에는 move메서드가 정의되어 있지 않기 때문에 move메서드를 호출하는 부분에서 에러가 발생한다.

```
Object[] group = new Object[4];
group[0] = new Marine();
group[1] = new Tank();
group[2] = new Marine();
group[3] = new Dropship();

for(int i = 0;i < group.length;i++)
    group[i].move(100, 200);
```

에러!!! Object클래스에 move메서드가 정의되어 있지 않다.

7. 인터페이스(interface)

7.1 인터페이스란?

인터페이스는 일종의 추상클래스이다. 인터페이스는 추상클래스처럼 추상메서드를 갖지만 추상클래스보다 추상화 정도가 높아서 추상클래스와 달리 봄통을 갖춘 일반 메서드 또는 멤버변수를 구성원으로 가질 수 없다. 오직 추상메서드와 상수만을 멤버로 가질 수 있으며, 그 외의 다른 어떠한 요소도 허용하지 않는다.

추상클래스를 부분적으로만 완성된 '미완성 설계도'라고 한다면, 인터페이스는 구현된 것은 아무 것도 없고 밑그림만 그려져 있는 '기본 설계도'라 할 수 있다.

인터페이스도 추상클래스처럼 완성되지 않은 불완전한 것이기 때문에 그 자체만으로 사용되기 보다는 다른 클래스를 작성하는데 도움 줄 목적으로 작성된다.

7.2 인터페이스의 작성

인터페이스를 작성하는 것은 클래스를 작성하는 것과 같다. 다만 키워드로 class 대신 interface를 사용한다는 것만 다르다. 그리고 interface에도 클래스와 같이 접근제어자로 public 또는 default를 사용할 수 있다.

```
interface 인터페이스이름 {
    public static final 타입 상수이름 = 값;
    public abstract 메서드이름(매개변수목록);
}
```

일반적인 클래스의 멤버들과 달리 인터페이스의 멤버들은 다음과 같은 제약사항이 있다.

- 모든 멤버변수는 public static final 이어야 하며, 이를 생략할 수 있다.
- 모든 메서드는 public abstract 이어야 하며, 이를 생략할 수 있다.
단, static메서드와 디폴트 메서드는 예외(JDK1.8부터)

인터페이스에 정의된 모든 멤버에 예외없이 적용되는 사항이기 때문에 제어자를 생략할 수 있는 것이며, 편의상 생략하는 경우가 많다. 생략된 제어자는 컴파일 시에 컴파일러가 자동적으로 추가해준다.

```
interface PlayingCard {
    public static final int SPADE = 4;
    final int DIAMOND = 3; // public static final int DIAMOND = 3;
    static int HEART = 2; // public static final int HEART = 2;
    int CLOVER = 1; // public static final int CLOVER = 1;

    public abstract String getCardNumber();
    String getCardKind(); // public abstract String getCardKind();
}
```

원래는 인터페이스의 모든 메서드는 추상메서드이어야 하는데, JDK1.8부터 인터페이스에 static메서드와 디폴트 메서드(default method)의 추가를 허용하는 방향으로 변경되었다. 실무에서는 아직 JDK1.8을 사용하지 않는 곳이 많기 때문에, JDK1.8이전의 규칙과 이후의 규칙을 모두 알고 있어야 한다.

7.3 인터페이스의 상속

인터페이스는 인터페이스로부터만 상속받을 수 있으며, 클래스와는 달리 다중상속, 즉 여러 개의 인터페이스로부터 상속을 받는 것이 가능하다.

| 참고 | 인터페이스는 클래스와 달리 Object클래스와 같은 최고 조상이 없다.

```
interface Movable {
    /** 지정된 위치(x, y)로 이동하는 기능의 메서드 */
    void move(int x, int y);
}

interface Attackable {
    /** 지정된 대상(u)을 공격하는 기능의 메서드 */
    void attack(Unit u);
}

interface Fightable extends Movable, Attackable { }
```

클래스의 상속과 마찬가지로 자손 인터페이스(Fightable)는 조상 인터페이스(Movable, Attackable)에 정의된 멤버를 모두 상속받는다.

그래서 Fightable자체에는 정의된 멤버가 하나도 없지만 조상 인터페이스로부터 상속받은 두 개의 추상메서드, move(int x, int y)와 attack(Unit u)을 멤버로 갖게 된다.

7.4 인터페이스의 구현

인터페이스도 추상클래스처럼 그 자체로는 인스턴스를 생성할 수 없으며, 추상클래스가 상속을 통해 추상메서드를 완성하는 것처럼, 인터페이스도 자신에 정의된 추상메서드의 몸통을 만들어주는 클래스를 작성해야 하는데, 그 방법은 추상클래스가 자신을 상속받는 클래스를 정의하는 것과 다르지 않다. 다만 클래스는 확장한다는 의미의 키워드 ‘extends’를 사용하지만 인터페이스는 구현한다는 의미의 키워드 ‘implements’를 사용할 뿐이다.

```
class 클래스이름 implements 인터페이스이름 {
    // 인터페이스에 정의된 추상메서드를 구현해야 한다.
}

class Fighter implements Fightable {
    public void move(int x, int y) { /* 내용 생략 */ }
    public void attack(Unit u) { /* 내용 생략 */ }
}
```

| 참고 | 이 때 ‘Fighter클래스는 Fightable인터페이스를 구현한다.’라고 한다.

만일 구현하는 인터페이스의 메서드 중 일부만 구현한다면, abstract를 붙여서 추상클래스로 선언해야 한다.

```
abstract class Fighter implements Fightable {
    public void move(int x, int y) { /* 내용 생략 */ }
}
```

그리고 다음과 같이 상속과 구현을 동시에 할 수도 있다.

```
class Fighter extends Unit implements Fightable {
    public void move(int x, int y) { /* 내용 생략 */ }
    public void attack(Unit u) { /* 내용 생략 */ }
}
```

I 참고 | 인터페이스의 이름에는 주로 Fightable과 같이 '~을 할 수 있는'의 의미인 'able'로 끝나는 것들이 많은데, 그 이유는 어떠한 기능 또는 행위를 하는데 필요한 메서드를 제공한다는 의미를 강조하기 위해서이다. 또한 그 인터페이스를 구현한 클래스는 '~를 할 수 있는' 능력을 갖추었다는 의미이기도 하다. 이름이 'able'로 끝나는 것은 인터페이스라고 추측할 수 있지만, 모든 인터페이스의 이름이 반드시 'able'로 끝나야 하는 것은 아니다.

▼ 예제 7-24/ch7/FighterTest.java

```
class FighterTest {
    public static void main(String[] args) {
        Fighter f = new Fighter();

        if (f instanceof Unit)
            System.out.println("f는 Unit클래스의 자손입니다.");

        if (f instanceof Fightable)
            System.out.println("f는 Fightable인터페이스를 구현했습니다.");

        if (f instanceof Movable)
            System.out.println("f는 Movable인터페이스를 구현했습니다.");

        if (f instanceof Attackable)
            System.out.println("f는 Attackable인터페이스를 구현했습니다.");

        if (f instanceof Object)
            System.out.println("f는 Object클래스의 자손입니다.");
    } // main
}

class Fighter extends Unit implements Fightable {
    public void move(int x, int y) { /* 내용 생략 */ }
    public void attack(Unit u) { /* 내용 생략 */ }
}

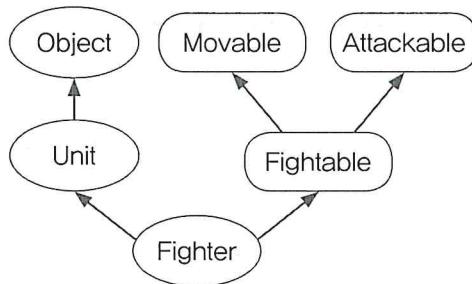
class Unit {
    int currentHP; // 유닛의 체력
    int x;          // 유닛의 위치(x좌표)
    int y;          // 유닛의 위치(y좌표)
}

interface Fightable extends Movable, Attackable { }
interface Movable { void move(int x, int y); }
interface Attackable { void attack(Unit u); }
```

▼ 실행결과

```
f는 Unit클래스의 자손입니다.
f는 Fightable인터페이스를 구현했습니다.
f는 Movable인터페이스를 구현했습니다.
f는 Attackable인터페이스를 구현했습니다.
f는 Object클래스의 자손입니다.
```

예제에 사용된 클래스와 인터페이스간의 관계를 그려보면 다음과 같다.



실제로 Fighter클래스는 Unit클래스로부터 상속받고 Fightable인터페이스만을 구현했지만, Unit클래스는 Object클래스의 자손이고, Fightable인터페이스는 Attackable과 Movable인터페이스의 자손이므로 Fighter클래스는 이 모든 클래스와 인터페이스의 자손이 되는 셈이다.

인터페이스는 상속 대신 구현이라는 용어를 사용하지만, 인터페이스로부터 상속받은 추상메서드를 구현하는 것이기 때문에 인터페이스도 조금은 다른 의미의 조상이라고 할 수 있다. 여기서 주의 깊게 봐두어야 할 것은 Movable인터페이스에 정의된 ‘void move(int x, int y)’를 Fighter클래스에서 구현할 때 접근 제어자를 public으로 했다는 것이다.

```

interface Movable {
    void move(int x, int y);
}

class Fighter extends Unit implements Fightable {
    public void move(int x, int y) { /* 실제구현내용 생략 */ }
    public void attack(Unit u) { /* 실제구현내용 생략 */ }
}
  
```

오버라이딩 할 때는 조상의 메서드보다 넓은 범위의 접근 제어자를 지정해야 한다는 것을 기억할 것이다. Movable인터페이스에 ‘void move(int x, int y)’와 같이 정의되어 있지만 사실 ‘public abstract’가 생략된 것이기 때문에 실제로 ‘public abstract void move(int x, int y)’이다. 그래서, 이를 구현하는 Fighter클래스에서는 ‘void move(int x, int y)’의 접근 제어자를 반드시 public으로 해야 하는 것이다.

7.5 인터페이스를 이용한 다중상속

두 조상으로부터 상속받는 멤버 중에서 멤버변수의 이름이 같거나 메서드의 선언부가 일치하고 구현 내용이 다르다면 이 두 조상으로부터 상속받는 자손클래스는 어느 조상의 것을 상속받게 되는 것인지 알 수 없다. 어느 한 쪽으로부터의 상속을 포기하던가, 이름이 충돌하지 않도록 조상클래스를 변경하는 수밖에 없다.

그래서 다중상속은 장점도 있지만 단점이 더 크다고 판단하였기 때문에 자바에서는 다중상속을 허용하지 않는다. 그러나 또 다른 객체지향언어인 C++에서는 다중상속을 허용하기 때문에 자바는 다중상속을 허용하지 않는다는 것이 단점으로 부각되는 것에 대한 대응으로 ‘자바도 인터페이스를 이용하면 다중상속이 가능하다.’라고 하는 것일 뿐 자바에서 인터페이스로 다중상속을 구현하는 경우는 거의 없다.

이러한 이유로 인터페이스가 다중상속을 위한 것으로 오해를 사곤 하는데, 앞으로 이 단원을 학습해 나가면서 인터페이스의 참다운 의미를 알게 될 것이다.

인터페이스를 이용한 다중상속에 대한 내용은 가볍게 맛만 보고 넘어가는 정도면 충분 할 것 같다.

인터페이스는 static상수만 정의할 수 있으므로 조상클래스의 멤버변수와 충돌하는 경우는 거의 없고 충돌된다 하더라도 클래스 이름을 붙여서 구분이 가능하다. 그리고 추상메서드는 구현내용이 전혀 없으므로 조상클래스의 메서드와 선언부가 일치하는 경우에는 당연히 조상 클래스 쪽의 메서드를 상속받으면 되므로 문제되지 않는다.

그러나, 이렇게 하면 상속받는 멤버의 충돌은 피할 수 있지만, 다중상속의 장점을 잊게 된다. 만일 두 개의 클래스로부터 상속을 받아야 할 상황이라면, 두 조상클래스 중에서 비중이 높은 쪽을 선택하고 다른 한쪽은 클래스 내부에 멤버로 포함시키는 방식으로 처리하거나 어느 한쪽의 필요한 부분을 뽑아서 인터페이스로 만든 다음 구현하도록 한다.

예를 들어, 다음과 같이 Tv클래스와 VCR클래스가 있을 때, TVCR클래스를 작성하기 위해 두 클래스로부터 상속을 받을 수만 있으면 좋겠지만 다중상속을 허용하지 않으므로, 한 쪽만 선택하여 상속받고 나머지 한 쪽은 클래스 내에 포함시켜서 내부적으로 인스턴스를 생성해서 사용하도록 한다.

```
public class Tv {
    protected boolean power;
    protected int channel;
    protected int volume;

    public void power() { power = ! power; }
    public void channelUp() { channel++; }
    public void channelDown() { channel--; }
    public void volumeUp() { volume++; }
    public void volumeDown() { volume--; }
}
```

```
public class VCR {  
    protected int counter; // VCR의 카운터  
  
    public void play() {  
        } // Tape을 재생한다.  
  
    public void stop() {  
        } // 재생을 멈춘다.  
  
    public void reset() {  
        counter = 0;  
    }  
  
    public int getCounter() {  
        return counter;  
    }  
  
    public void setCounter(int c) {  
        counter = c;  
    }  
}
```

VCR클래스에 정의된 메서드와 일치하는 추상메서드를 갖는 인터페이스를 작성한다.

```
public interface IVCR {  
    public void play();  
    public void stop();  
    public void reset();  
    public int getCounter();  
    public void setCounter(int c);  
}
```

이제 IVCR 인터페이스를 구현하고 Tv클래스로부터 상속받는 TVCR클래스를 작성한다.
이때 VCR클래스 타입의 참조변수를 멤버변수로 선언하여 IVCR인터페이스의 추상메서드를 구현하는데 사용한다.

```
public class TVCR extends Tv implements IVCR {
    VCR vcr = new VCR();

    public void play() {
        vcr.play();
    }

    public void stop() {
        vcr.stop();
    }

    public void reset() {
        vcr.reset();
    }

    public int getCounter() {
        return vcr.getCounter();
    }

    public void setCounter(int c) {
        vcr.setCounter(c);
    }
}
```

코드를 작성하는 대
메서드를 호출한다.

IVCR인터페이스를 구현하기 위해서는 새로 메서드를 작성해야하는 부담이 있지만 이처럼 VCR클래스의 인스턴스를 사용하면 손쉽게 다중상속을 구현할 수 있다.

또한 VCR클래스의 내용이 변경되어도 변경된 내용이 TVCR클래스에도 자동적으로 반영되는 효과도 얻을 수 있다.

사실 인터페이스를 새로 작성하지 않고도 VCR클래스를 TVCR클래스에 포함시키는 것으로도 충분하지만, 인터페이스를 이용하면 다형적 특성을 이용할 수 있다는 장점이 있다.

7.6 인터페이스를 이용한 다형성

다형성에 대해 학습할 때 자손클래스의 인스턴스를 조상타입의 참조변수로 참조하는 것이 가능하다는 것을 배웠다.

인터페이스 역시 이를 구현한 클래스의 조상이라 할 수 있으므로 해당 인터페이스 타입의 참조변수로 이를 구현한 클래스의 인스턴스를 참조할 수 있으며, 인터페이스 타입으로의 형변환도 가능하다.

인터페이스 Fightable을 클래스 Fighter가 구현했을 때, 다음과 같이 Fighter인스턴스를 Fightable타입의 참조변수로 참조하는 것이 가능하다.

```
Fightable f = (Fightable) new Fighter();
또는
Fightable f = new Fighter();
```

| 참고 | Fightable타입의 참조변수로는 인터페이스 Fightable에 정의된 멤버들만 호출이 가능하다.

따라서 인터페이스는 다음과 같이 메서드의 매개변수의 타입으로 사용될 수 있다.

```
void attack(Fightable f) {
    //...
}
```

인터페이스 타입의 매개변수가 갖는 의미는 메서드 호출 시 해당 인터페이스를 구현한 클래스의 인스턴스를 매개변수로 제공해야한다는 것이다.

그래서 attack메서드를 호출할 때는 매개변수로 Fightable인터페이스를 구현한 클래스의 인스턴스를 넘겨주어야 한다.

```
class Fighter extends Unit implements Fightable {
    public void move(int x, int y) { /* 내용 생략 */ }
    public void attack(Fightable f) { /* 내용 생략 */ }
}
```

위와 같이 Fightable인터페이스를 구현한 Fighter클래스가 있을 때, attack메서드의 매개변수로 Fighter인스턴스를 넘겨 줄 수 있다. 즉, attack(new Fighter())와 같이 할 수 있다는 것이다.

그리고 다음과 같이 메서드의 리턴타입으로 인터페이스의 타입을 지정하는 것 역시 가능하다.

```
Fightable method() {
```

```
    ...
    Fighter f = new Fighter();
    return f;
}
```

이 두 문장을 한 문장으로 바꾸면 다음과 같다. return new Fighter();

리턴타입이 인터페이스라는 것은 메서드가 해당 인터페이스를 구현한 클래스의 인스턴스를 반환한다는 것을 의미한다. 이 문장은 외울 때까지 반복해야 한다.

위의 코드에서는 method()의 리턴타입이 Fightable인터페이스이기 때문에 메서드의 return문에서 Fightable인터페이스를 구현한 Fighter클래스의 인스턴스를 반환한다.

▼ 예제 7-25/ch7/ParserTest.java

```
interface Parseable {
    // 구문 분석작업을 수행한다.
    public abstract void parse(String fileName);
}

class ParserManager {
    // 리턴타입이 Parseable인터페이스이다.
    public static Parseable getParser(String type) {
        if(type.equals("XML")) {
            return new XMLParser();
        } else {
            Parseable p = new HTMLParser();
            return p;
            // return new HTMLParser();
        }
    }
}

class XMLParser implements Parseable {
    public void parse(String fileName) {
        /* 구문 분석작업을 수행하는 코드를 적는다. */
        System.out.println(fileName + "- XML parsing completed.");
    }
}

class HTMLParser implements Parseable {
    public void parse(String fileName) {
        /* 구문 분석작업을 수행하는 코드를 적는다. */
        System.out.println(fileName + "-HTML parsing completed.");
    }
}
```

```

class ParserTest {
    public static void main(String args[]) {
        Parseable parser = ParserManager.getParser("XML");
        parser.parse("document.xml");
        parser = ParserManager.getParser("HTML");
        parser.parse("document2.html");
    }
}

```

▼ 실행결과

document.xml - XML parsing completed.
document2.html - HTML parsing completed.

Parseable 인터페이스는 구문분석(parsing)을 수행하는 기능을 구현할 목적으로 추상메서드 'parse(String fileName)'을 정의했다. 그리고 XMLParser 클래스와 HTMLParser 클래스는 Parseable 인터페이스를 구현하였다.

ParserManager 클래스의 getParser 메서드는 매개변수로 넘겨받는 type의 값에 따라 XMLParser 인스턴스 또는 HTMLParser 인스턴스를 반환한다.

```

Parseable parser = ParseManager.getParser("XML");

```

```

public static Parseable getParser(String type) {
    if(type.equals("XML")) {
        return new XMLParser();
    } else {
        Parseable p = new HTMLParser();
        return p;
    }
}

```

getParser 메서드의 수행 결과로 참조변수 parser는 XMLParser 인스턴스의 주소값을 갖게 된다. 마치 'Parseable parser = new XMLParser();'이 수행된 것과 같다.

```
parser.parse("document.xml"); // parser는 XMLParser 인스턴스를 가리킨다.
```

참조변수 parser를 통해 parse()를 호출하면, parser가 참조하고 있는 XMLParser 인스턴스의 parse 메서드가 호출된다.

만일 나중에 새로운 종류의 XML 구문분석기 NewXMLParser 클래스가 나와도 ParserTest 클래스는 변경할 필요 없이 ParserManager 클래스의 getParser 메서드에서 'return new XMLParser();' 대신 'return new NewXMLParser();'로 변경하기만 하면 된다.

이러한 장점은 특히 분산환경 프로그래밍에서 그 위력을 발휘한다. 사용자 컴퓨터에 설치된 프로그램을 변경하지 않고 서버측의 변경만으로도 사용자가 새로 개정된 프로그램을 사용하는 것이 가능하다.

7.7 인터페이스의 장점

인터페이스를 사용하는 이유와 그 장점을 정리해 보면 다음과 같다.

- 개발시간을 단축시킬 수 있다.
- 표준화가 가능하다.
- 서로 관계없는 클래스들에게 관계를 맺어 줄 수 있다.
- 독립적인 프로그래밍이 가능하다.

1. 개발시간을 단축시킬 수 있다.

일단 인터페이스가 작성되면, 이를 사용해서 프로그램을 작성하는 것이 가능하다. 메서드를 호출하는 쪽에서는 메서드의 내용에 관계없이 선언부만 알면 되기 때문이다.

그리고 동시에 다른 한 쪽에서는 인터페이스를 구현하는 클래스를 작성하게 하면, 인터페이스를 구현하는 클래스가 작성될 때까지 기다리지 않고도 양쪽에서 동시에 개발을 진행할 수 있다.

2. 표준화가 가능하다.

프로젝트에 사용되는 기본 틀을 인터페이스로 작성한 다음, 개발자들에게 인터페이스를 구현하여 프로그램을 작성하도록 함으로써 보다 일관되고 정형화된 프로그램의 개발이 가능하다.

3. 서로 관계없는 클래스들에게 관계를 맺어 줄 수 있다.

서로 상속관계에 있지도 않고, 같은 조상클래스를 가지고 있지 않은 서로 아무런 관계도 없는 클래스들에게 하나의 인터페이스를 공통적으로 구현하도록 함으로써 관계를 맺어 줄 수 있다.

4. 독립적인 프로그래밍이 가능하다.

인터페이스를 이용하면 클래스의 선언과 구현을 분리시킬 수 있기 때문에 실제구현에 독립적인 프로그램을 작성하는 것이 가능하다. 클래스와 클래스간의 직접적인 관계를 인터페이스를 이용해서 간접적인 관계로 변경하면, 한 클래스의 변경이 관련된 다른 클래스에 영향을 미치지 않는 독립적인 프로그래밍이 가능하다.

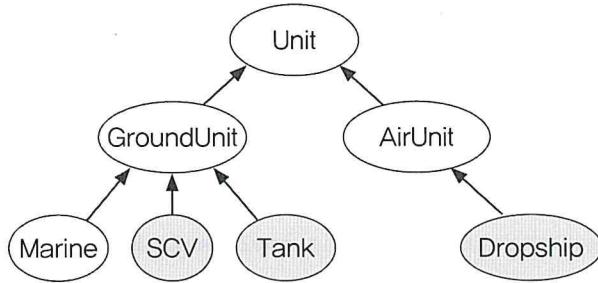
예를 들어 한 데이터베이스 회사가 제공하는 특정 데이터베이스를 사용하는데 필요한 클래스를 사용해서 프로그램을 작성했다면 이 프로그램은 다른 종류의 데이터베이스를 사용하기 위해서는 전체 프로그램 중에서 데이터베이스 관련된 부분은 모두 변경해야 할 것이다.

그러나 데이터베이스 관련 인터페이스를 정의하고 이를 이용해서 프로그램을 작성하면, 데이터베이스의 종류가 변경되더라도 프로그램을 변경하지 않도록 할 수 있다.

단, 데이터베이스 회사에서 제공하는 클래스도 인터페이스를 구현하도록 요구해야 한다. 데이터베이스를 이용한 응용프로그램을 작성하는 쪽에서는 인터페이스를 이용해서 프로그램을 작성하고, 데이터베이스 회사에서는 인터페이스를 구현한 클래스를 작성해서 제공해야 한다.

실제로 자바에서는 다수의 데이터베이스와 관련된 다수의 인터페이스를 제공하고 있으며, 프로그래머는 이 인터페이스를 이용해서 프로그래밍하면 특정 데이터베이스에 종속되지 않는 프로그램을 작성할 수 있다.

게임에 나오는 유닛을 클래스로 표현하고 이들의 관계를 상속계층도로 표현해 보았다.



게임에 나오는 모든 유닛들의 최고 조상은 Unit클래스이고 유닛의 종류는 지상유닛(GroundUnit)과 공중유닛(AirUnit)으로 나누어진다.

그리고 지상유닛에는 Marine, SCV(건설인부), Tank가 있고, 공중유닛으로는 Dropship(수송선)이 있다. SCV에게 Tank와 Dropship과 같은 기계화 유닛을 수리할 수 있는 기능을 제공하기 위해 repair메서드를 정의한다면 다음과 같을 것이다.

```

void repair(Tank t) {
    // Tank를 수리한다.
}

void repair(Dropship d) {
    // Dropship을 수리한다.
}
  
```

이런 식으로 수리가 가능한 유닛의 개수만큼 다른 버전의 오버로딩된 메서드를 정의해야 할 것이다.

이것을 피하기 위해 매개변수의 타입을 이들의 공통 조상으로 하면 좋겠지만 Dropship은 공통조상이 다르기 때문에 공통조상의 타입으로 메서드를 정의한다고 해도 최소한 2개의 메서드가 필요할 것이다.

```

void repair(GroundUnit gu) {
    // 매개변수로 넘겨진 지상유닛 (GroundUnit)을 수리한다.
}

void repair(AirUnit au) {
    // 매개변수로 넘겨진 공중유닛 (AirUnit)을 수리한다.
}
  
```

그리고 GroundUnit의 자손 중에는 Marine과 같이 기계화 유닛이 아닌 클래스도 포함될 수 있기 때문에 repair메서드의 매개변수 타입으로 GroundUnit은 부적합하다.

현재의 상속관계에서는 이들의 공통점은 없다. 이 때 인터페이스를 이용하면 기존의 상속체계를 유지하면서 이들 기계화 유닛에 공통점을 부여할 수 있다.

다음과 같이 Repairable이라는 인터페이스를 정의하고 수리가 가능한 기계화 유닛에게 이 인터페이스를 구현하도록 하면 된다.

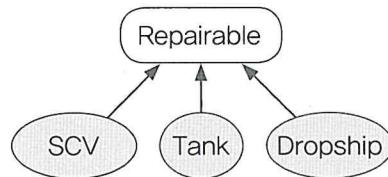
```
interface Repairable {}

class SCV extends GroundUnit implements Repairable {
    //...
}

class Tank extends GroundUnit implements Repairable {
    //...
}

class Dropship extends AirUnit implements Repairable {
    //...
}
```

이제 이 3개의 클래스에는 같은 인터페이스를 구현했다는 공통점이 생겼다. 인터페이스 Repairable에 정의된 것은 아무것도 없고, 단지 인스턴스의 타입체크에만 사용될 뿐이다. Repairable인터페이스를 중심으로 상속계층도를 그려보면 다음과 같다.



그리고 repair메서드의 매개변수의 타입을 Repairable로 선언하면, 이 메서드의 매개변수로 Repairable인터페이스를 구현한 클래스의 인스턴스만 받아들여질 것이다.

```
void repair(Repairable r) {
    // 매개변수로 넘겨받은 유닛을 수리한다.
}
```

앞으로 새로운 클래스가 추가될 때, SCV의 repair메서드에 의해서 수리가 가능하도록 하려면 Repairable인터페이스를 구현하도록 하면 될 것이다.

▼ 예제 7-26/ch7/RepairableTest.java

```
class RepairableTest{
    public static void main(String[] args) {
        Tank tank = new Tank();
        Dropship dropship = new Dropship();

        Marine marine = new Marine();
        SCV scv = new SCV();
```

```

        scv.repair(tank);           // SCV가 Tank를 수리하도록 한다.
        scv.repair(dropship);
        scv.repair(marine); ←—————
    }

interface Repairable {}

class Unit {
    int hitPoint;
    final int MAX_HP;
    Unit(int hp) {
        MAX_HP = hp;
    }
    //...
}

class GroundUnit extends Unit {
    GroundUnit(int hp) {
        super(hp);
    }
}

class AirUnit extends Unit {
    AirUnit(int hp) {
        super(hp);
    }
}

class Tank extends GroundUnit implements Repairable {
    Tank() {
        super(150);           // Tank의 HP는 1500이다.
        hitPoint = MAX_HP;
    }

    public String toString() {
        return "Tank";
    }
    //...
}

class Dropship extends AirUnit implements Repairable {
    Dropship() {
        super(125);          // Dropship의 HP는 1250이다.
        hitPoint = MAX_HP;
    }

    public String toString() {
        return "Dropship";
    }
    //...
}

class Marine extends GroundUnit {
    Marine() {
        super(40);
        hitPoint = MAX_HP;
    }
    //...
}

```

에러! repair(Repairable)
in SCV cannot be
applied to (Marine)

```

class SCV extends GroundUnit implements Repairable{
    SCV() {
        super(60);
        hitPoint = MAX_HP;
    }

    void repair(Repairable r) {
        if (r instanceof Unit) {
            Unit u = (Unit)r;
            while(u.hitPoint!=u.MAX_HP) {
                /* Unit의 HP를 증가시킨다. */
                u.hitPoint++;
            }
            System.out.println( u.toString() + "의 수리가 끝났습니다." );
        }
    }
    //...
}

```

▼ 실행결과

Tank의 수리가 끝났습니다.
Dropship의 수리가 끝났습니다.

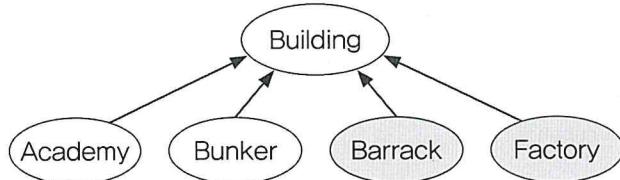
repair메서드의 매개변수 r은 Repairable타입이기 때문에 인터페이스 Repairable에 정의된 멤버만 사용할 수 있다. 그러나 Repairable에는 정의된 멤버가 없으므로 이 타입의 참조변수로는 할 수 있는 일은 아무 것도 없다.

그래서 instanceof연산자로 타입을 체크한 뒤 캐스팅하여 Unit클래스에 정의된 hitPoint와 MAX_HP를 사용할 수 있도록 하였다.

그 다음엔 유닛의 현재 체력(hitPoint)이 유닛이 가질 수 있는 최고 체력(MAX_HP)이 될 때까지 체력을 증가시키는 작업을 수행한다.

Marine은 Repairable인터페이스를 구현하지 않았으므로 SCV클래스의 repair메서드의 매개변수로 Marine을 사용하면 컴파일 시에 에러가 발생한다.

이와 유사한 예를 한 가지 더 들어보자. 게임에 나오는 건물들을 클래스로 표현하고 이들의 관계를 상속계층도로 표현하였다.



건물을 표현하는 클래스 Academy, Bunker, Barrack, Factory가 있고 이들의 조상인 Building클래스가 있다고 하자. 이 때 Barrack클래스와 Factory클래스에 다음과 같은 내용의, 건물을 이동시킬 수 있는, 새로운 메서드를 추가하고자 한다면 어떻게 해야 할까?

```

void liftOff()           { /* 내용생략 */ }
void move(int x, int y) { /* 내용생략 */ }
void stop()              { /* 내용생략 */ }
void land()              { /* 내용생략 */ }

```

Barrack클래스와 Factory클래스 모두 위의 코드를 적어주면 되긴 하지만, 코드가 중복 된다는 단점이 있다. 그렇다고 해서 조상클래스인 Building클래스에 코드를 추가해주면, Building클래스의 다른 자손인 Academy와 Bunker클래스도 추가된 코드를 상속받으므로 안 된다.

이런 경우에도 인터페이스를 이용해서 해결할 수가 있다. 우선 새로 추가하고자하는 메서드를 정의하는 인터페이스를 정의하고 이를 구현하는 클래스를 작성한다.

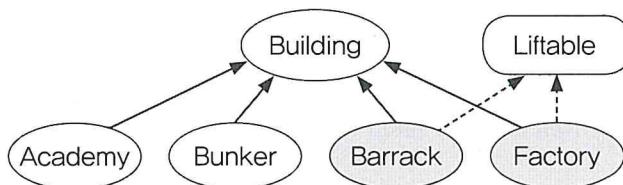
```

interface Liftable {
    /** 건물을 들어 올린다. */
    void liftOff();                                // public abstract가 생략되었음.
    /** 건물을 이동한다. */
    void move(int x, int y);
    /** 건물을 정지시킨다. */
    void stop();
    /** 건물을 착륙시킨다. */
    void land();
}

class LiftableImpl implements Liftable {
    public void liftOff()           { /* 내용 생략 */ }
    public void move(int x, int y) { /* 내용 생략 */ }
    public void stop()              { /* 내용 생략 */ }
    public void land()              { /* 내용 생략 */ }
}

```

마지막으로 새로 작성된 인터페이스와 이를 구현한 클래스를 Barrack과 Factory클래스에 적용하면 된다.



Barrack클래스가 Liftable인터페이스를 구현하도록 하고, 인터페이스를 구현한 LiftableImpl클래스를 Barrack클래스에 포함시켜서 내부적으로 호출해서 사용하도록 한다.

이렇게 함으로써 같은 내용의 코드를 Barrack클래스와 Factory클래스에서 각각 작성하지 않고 LiftableImpl클래스 한 곳에서 관리할 수 있다. 그리고 작성된 Liftable인터페이스와 이를 구현한 LiftableImpl클래스는 후에 다시 재사용될 수 있을 것이다.

```
class Barrack extends Building implements Liftable {
    LiftableImpl l = new LiftableImpl();
    void liftOff() { l.liftOff(); }
    void move(int x, int y) { l.move(x, y); }
    void stop() { l.stop(); }
    void land() { l.land(); }
    void trainMarine() { /* 내용 생략 */ }
    ...
}

class Factory extends Building implements Liftable {
    LiftableImpl l = new LiftableImpl();
    void liftOff() { l.liftOff(); }
    void move(int x, int y) { l.move(x, y); }
    void stop() { l.stop(); }
    void land() { l.land(); }
    void makeTank() { /* 내용 생략 */ }
    ...
}
```

7.8 인터페이스의 이해

지금까지 인터페이스의 특징과 구현하는 방법, 장점 등 인터페이스에 대한 일반적인 사항들에 대해서 모두 살펴보았다. 하지만 ‘인터페이스란 도대체 무엇인가?’라는 의문은 여전히 남아있을 것이다. 이번 절에서는 인터페이스의 규칙이나 활용이 아닌, 본질적인 측면에 대해 살펴보자.

먼저 인터페이스를 이해하기 위해서는 다음의 두 가지 사항을 반드시 염두에 두고 있어야 한다.

- 클래스를 사용하는 쪽(User)과 클래스를 제공하는 쪽(Provider)이 있다.
- 메서드를 사용(호출)하는 쪽(User)에서는 사용하려는 메서드(Provider)의 선언부만 알면 된다.(내용은 몰라도 된다.)

▼ 예제 7-27/ch7/InterfaceTest.java

```
class A {
    public void methodA(B b) {
        b.methodB();
    }
}
```

```

class B {
    public void methodB() {
        System.out.println("methodB()");
    }
}

class InterfaceTest {
    public static void main(String args[]) {
        A a = new A();
        a.methodA(new B());
    }
}

```

▼ 실행결과
methodB()

예제7-27과 같이 클래스 A와 클래스 B가 있다고 하자. 클래스 A(User)는 클래스 B(Provider)의 인스턴스를 생성하고 메서드를 호출한다. 이 두 클래스는 서로 직접적인 관계에 있다. 이것을 간단히 ‘A–B’라고 표현하자.



이 경우 클래스 A를 작성하려면 클래스 B가 이미 작성되어 있어야 한다. 그리고 클래스 B의 methodB()의 선언부가 변경되면, 이를 사용하는 클래스 A도 변경되어야 한다.

이와 같이 직접적인 관계의 두 클래스는 한 쪽(Provider)이 변경되면 다른 한 쪽(User)도 변경되어야 한다는 단점이 있다.

그러나 클래스 A가 클래스 B를 직접 호출하지 않고 인터페이스를 매개체로 해서 클래스 A가 인터페이스를 통해서 클래스 B의 메서드에 접근하도록 하면, 클래스 B에 변경사항이 생기거나 클래스 B와 같은 기능의 다른 클래스로 대체 되어도 클래스 A는 전혀 영향을 받지 않도록 하는 것이 가능하다.

두 클래스간의 관계를 간접적으로 변경하기 위해서는 먼저 인터페이스를 이용해서 클래스 B(Provider)의 선언과 구현을 분리해야 한다.

먼저 다음과 같이 클래스 B에 정의된 메서드를 추상메서드로 정의하는 인터페이스 I를 정의한다.

```

interface I {
    public abstract void methodB();
}

```

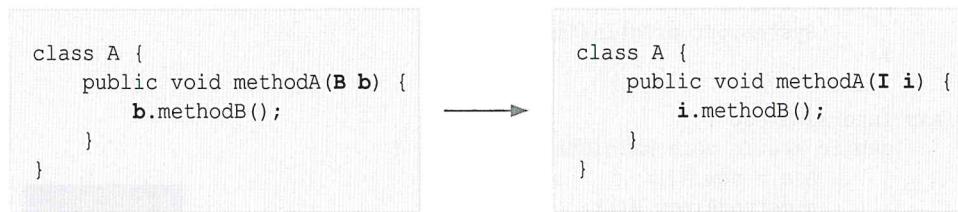
그 다음에는 클래스 B가 인터페이스 I를 구현하도록 한다.

```

class B implements I {
    public void methodB() {
        System.out.println("methodB in B class");
    }
}

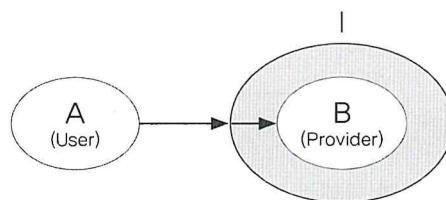
```

이제 클래스 A는 클래스 B 대신 인터페이스 I를 사용해서 작성할 수 있다.



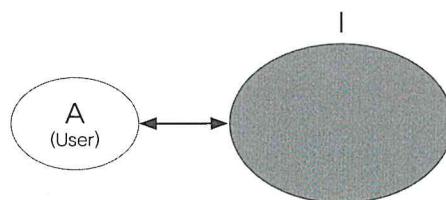
| 참고 | methodA가 호출될 때 인터페이스 I를 구현한 클래스의 인스턴스(클래스 B의 인스턴스)를 제공받아야 한다.

클래스 A를 작성하는데 있어서 클래스 B가 사용되지 않았다는 점에 주목하자. 이제 클래스 A와 클래스 B는 'A-B'의 직접적인 관계에서 'A-I-B'의 간접적인 관계로 바뀐 것이다.



결국 클래스 A는 여전히 클래스 B의 메서드를 호출하지만, 클래스 A는 인터페이스 I하고만 직접적인 관계에 있기 때문에 클래스 B의 변경에 영향을 받지 않는다.

클래스 A는 인터페이스를 통해 실제로 사용하는 클래스의 이름을 몰라도 되고 심지어는 실제로 구현된 클래스가 존재하지 않아도 문제되지 않는다. 클래스 A는 오직 직접적인 관계에 있는 인터페이스 I의 영향만 받는다.



인터페이스 I는 실제구현 내용(클래스 B)을 감싸고 있는 껍데기이며, 클래스 A는 껍데기 안에 어떤 알맹이(클래스)가 들어 있는지 몰라도 된다.

▼ 예제 7-28/ch7/InterfaceTest2.java

```

class A {
    void autoPlay(I i) {
        i.play();
    }
}

```

```

interface I {
    public abstract void play();
}

class B implements I {
    public void play() {
        System.out.println("play in B class");
    }
}

class C implements I {
    public void play() {
        System.out.println("play in C class");
    }
}

class InterfaceTest2 {
    public static void main(String[] args) {
        A a = new A();
        a.autoPlay(new B()); // void autoPlay(I i) 호출
        a.autoPlay(new C()); // void autoPlay(I i) 호출
    }
}

```

▼ 실행결과
play in B class
play in C class

| 참고 | 클래스 A를 작성하는데 클래스 B가 관련되지 않았다는 사실에 주목하자.

클래스 A가 인터페이스 I를 사용해서 작성되긴 하였지만, 이처럼 매개변수를 통해서 인터페이스 I를 구현한 클래스의 인스턴스를 동적으로 제공받아야 한다.

클래스 Thread의 생성자인 Thread(Runnable target)로 이런 방식으로 되어 있다.

| 참고 | Runnable은 인터페이스이다.

이처럼 매개변수를 통해 동적으로 제공받을 수 도 있지만 다음과 같이 제3의 클래스를 통해서 제공받을 수도 있다. JDBC의 DriverManager 클래스가 이런 방식으로 되어 있다.

▼ 예제 7-29/ch7/InterfaceTest3.java

```

class InterfaceTest3 {
    public static void main(String[] args) {
        A a = new A();
        a.methodA();
    }
}

class A {
    void methodA() {
        I i = InstanceManager.getInstance(); // 제3의 클래스의 메서드를 통해서
        i.methodB();                      // 인터페이스 I를 구현한 클래스의
        System.out.println(i.toString());   // 인스턴스를 얻어온다.
        // i로 Object 클래스의 메서드 호출 가능
    }
}

```

제3의 클래스의 메서드를 통해서
인터페이스 I를 구현한 클래스의
인스턴스를 얻어온다.

```

interface I {
    public abstract void methodB();
}

class B implements I {
    public void methodB() {
        System.out.println("methodB in B class");
    }

    public String toString() { return "class B"; }
}

class InstanceManager {
    public static I getInstance() {
        return new B();
    }
}

```

▼ 실행결과

```
methodB in B class
class B
```

인스턴스를 직접 생성하지 않고, `getInstance()`라는 메서드를 통해 제공받는다. 이렇게 하면, 나중에 다른 클래스의 인스턴스로 변경되어도 A클래스의 변경 없이 `getInstance()`만 변경하면 된다는 장점이 생긴다.

```

class InstanceManager {
    public static I getInstance() {
        return new B(); // 다른 인스턴스로 바꾸려면 여기만 변경하면 됨.
    }
}

```

그리고 인터페이스 I 타입의 참조변수 i로도 Object클래스에 정의된 메서드들을 호출할 수 있다는 것도 알아두자. i에 `toString()`이 정의되어 있지 않지만, 모든 객체는 Object클래스에 정의된 메서드를 가지고 있을 것이기 때문에 허용하는 것이다.

```

class A {
    void methodA() {
        I i = InstanceManager.getInstance();
        i.methodB();
        System.out.println(i.toString()); // i로 Object의 메서드 호출 가능
    }
}

```

7.9 디폴트 메서드와 static메서드

원래는 인터페이스에 추상 메서드만 선언할 수 있는데, JDK1.8부터 디폴트 메서드와 static메서드도 추가할 수 있게 되었다. static메서드는 인스턴스와 관계가 없는 독립적인 메서드이기 때문에 예전부터 인터페이스에 추가하지 못할 이유가 없었다.

그러나 자바를 보다 쉽게 배울 수 있도록 규칙을 단순히 할 필요가 있어서 인터페이스의 모든 메서드는 추상 메서드이어야 한다는 규칙에 예외를 두지 않았다. 덕분에 인터페이스 와 관련된 static메서드는 별도의 클래스에 따로 두어야 했다.