

핵심만 콕! Java 실습 가이드

핵심만 콕! Java 실습 가이드

류기열 박혁로 정기숙 조은선 지음

[저자정보]

류기열 아주대학교 소프트웨어학과 교수

박혁로 전남대학교 인공지능학부 교수

정기숙 경북대학교 소프트웨어교육원 초빙교수

조은선 충남대학교 컴퓨터융합학부 교수

핵심만 콕! Java 실습 가이드

초판 인쇄: 2024년 월 일

초판 발행: 2024년 월 일

편집인: 류기열 · 박혁로 · 정기숙 · 조은선

발행처: 아주대학교출판부

주 소: 경기도 수원시 월드컵로 206 아주대학교

연락처:

ISBN:

본 교재는 2024년도 과학기술정보통신부 및 정보통신기획평가원의 "SW중심대학사업(경북대학교, 아주대학교, 전남대학교, 충남대학교)"의 지원을 받아 제작되었습니다.

본 교재의 내용은 전재할 수 없으며, 인용할 때에는 반드시 과학기술정보통신부와 정보통신기획평가원의 "SW중심대학사업(경북대학교, 아주대학교, 전남대학교, 충남대학교)"의 결과물이라는 출처를 밝혀야 합니다.

머리말

객체지향프로그래밍은 현대적인 소프트웨어를 개발함에 있어 가장 많이 활용되는 프로그래밍 방식으로 컴퓨터 공학 전공자나 소프트웨어 개발자라면 반드시 배워야 하는 내용이기 때문에 거의 모든 대학의 컴퓨터 관련 학과에서 필수 교과목으로 지정되어 있다.

이 책은 객체지향프로그래밍을 처음 접하는 학생들이 Java 언어를 중심으로 객체지향프로그래밍 언어의 개념을 학습할 수 있도록 제작되었다. 학생들은 프로그래밍 언어의 개념을 배울 때 이론적 강의보다 실습을 통해 개념을 확인하는 방식이 더 효과적이라고 느낀다. 하지만 저자들은 그동안 객체지향프로그래밍 언어를 강의해오면서 항상 좋은 실습용 교재에 대한 아쉬움을 느꼈고, 이에 4개의 대학(경북대, 아주대, 전남대, 충남대)이 뜻을 모아 그동안 실습교육을 통해 쌓은 경험을 기반으로 대학에서 공동으로 활용할 수 있는 실습 교재를 편찬하게 되었다.

이 책은 Java 언어에서의 데이터 타입이나 연산식, 제어문과 같은 기초적인 지식을 습득한 학생들을 대상으로 주로 객체, 클래스, 상속, 다형성 등과 같은 객체지향 개념의 실습에 초점을 맞춰 제작되었다. 이 책은 총 12개의 장으로 구성되어 있으며, 각 장은 '개념 정리' 및 '개념 확인', '응용 문제'의 세 가지 영역으로 구성된다. '개념 정리'에서는 각 절에서 다루고자 하는 객체지향언어의 핵심 개념을 소개하고 기초 지식을 정리하여 제공한다. '개념 확인'에서는 '개념 정리'에서 소개한 이론적 개념을 이해하고 있는지 확인하기 위한 기초적인 질문이나 간단한 코드 작성하기 등의 문제를 제공한다. 마지막으로 '응용 문제'에서는 '개념 정리'와 '개념 확인'에서 다룬 개념을 실제 문제에 적용할 수 있는 코딩 문제가 제공된다. 문제에 대한 해답과 문제와 해답에 나타나는 소스 코드는 별도의 파일로 제공된다.

본 교재는 여러 가지 용도로 활용될 수 있을 것이다. 우선, 교수가 실습 문제 출제시 직접 사용하거나 문제를 변형하여 사용할 수 있는 실습 문제 참조용으로 활용할 수 있다. 또한 학생들이 스스로 객체지향프로그래밍의 이론적 개념을 공부하면서 핵심 내용을 파악하거나 개념을 정립하는데 도움을 줄 수 있다. 마지막으로, 소프트웨어 개발자들이 소프트웨어를 개발하는 과정에서 복습이 필요한 경우에 참고할 수도 있을 것이다.

마지막으로, 이 책은 과학기술정보통신부의 소프트웨어중심대학 사업의 지원으로 편찬되었음을 밝히며, 아울러 이 책의 편집과 교정을 도와준 여러 학생들에게 고마움을 전한다.

2024년 3월 30일

저자 일동

목차

1장 문자열 및 배열	2
1.1 자바의 문자형 및 문자열형	2
1.2 배열	11
2장 클래스 I	20
2.1 클래스와 객체	20
2.2 새로운 클래스 작성	23
3장 클래스 II	34
3.1 클래스 사이의 관계	34
3.2 패키지	36
3.3 객체지향 시스템 설계	40
4장 상속	48
4.1 슈퍼 클래스와 서브 클래스	48
4.2 접근 제어	52
4.3 상속과 생성자	55
4.4 메소드 오버라이딩과 메소드 오버로딩	58
4.5 Object 클래스	65
5장 다형성, 추상 클래스, 인터페이스	72
5.1 다형성	72
5.2 추상 클래스	79
5.3 인터페이스	82
5.4 다형성 종합	89
6장 예외 처리	94
6.1 예외와 예외 처리	94
6.2 예외 계층	103
6.3 스택 풀기	110
6.4 연결된 예외	112
7장 람다식	116
7.1 람다식의 필요성	116
7.2 람다식 및 함수형 인터페이스	117
7.3 람다식의 용도	122
8장 제네릭 프로그래밍	130

8.1 제네릭 기초	130
8.2 제한된 타입 매개변수	136
8.3 와일드카드	141
9장 컬렉션 프레임워크	148
9.1 인터페이스 & 클래스	148
9.2 반복자와 Map	154
9.3 알고리즘을 인자로 전달	157
10장 Swing: 그래픽 사용자 인터페이스 프레임워크	160
10.1 Swing: Java GUI 프레임워크 소개	160
10.2 이벤트 모델	162
10.3 레이아웃 매니저	168
10.4 Swing 컴포넌트	173
10.5 그래픽 기초	176
11장 파일 입출력 스트림	180
11.1 파일 입출력 스트림	180
11.2 바이트 기반 입출력 스트림	181
11.3 문자 기반 입출력 스트림	185
11.4 기본형 데이터 이진형식 입출력	190
11.5 랜덤 액세스 파일	192
12장 스레드	198
12.1 Thread 클래스	198
12.2 Runnable 인터페이스와 상태	202
12.3 wait & notify	205

1장 문자열 및 배열

1.1 자바의 문자형 및 문자열형

1.2 배열

1장 문자열 및 배열

1.1 자바의 문자형 및 문자열형

A. 개념 정리

자바의 문자형은 글자 하나를 저장하기 위한 자료형이다. 컴퓨터 내부에서 문자를 저장하기 위해서는 각 문자에 번호를 부여한 후, 이 번호를 이진수로 저장한다. 이렇게 각 문자에 부여된 번호를 문자 코드(character code)라고 한다.

컴퓨터 역사 초기에는 영어 문자만을 대상으로 코드를 부여하였기 때문에 1바이트 길이의 ASCII(American Standard Code for Information Interchange:정보 교환용 미국 표준 코드) 코드가 표준으로 사용되었다. 컴퓨터로 처리해야 하는 언어의 수가 확대됨에 따라서 문자의 수도 늘어나게 되었고 새로운 표준 코드가 필요하게 되었다. 1980년대 말부터 여러 국가의 문자들을 수집하여 서로 겹치지 않는 문자 코드를 부여하는 것을 목표로 하는 유니코드(Unicode) 개발이 시작되었다. 1991년 유니코드 컨소시엄은 약 30,000개의 문자에 코드를 부여한 유니코드 1.0을 제정하였으며 2바이트를 사용하여 유니코드 번호를 표현하도록 하였다. 그러나 유니코드 1.0 이후로 새로운 문자들이 계속 추가되었고 이에 따라서 유니코드 1.0에서 생각했던 2바이트 길이의 문자형으로는 이제는 모든 문자 코드를 표현할 수 없는 문제가 발생하였다. 유니코드 컨소시엄은 이 문제를 해결하기 위해 1996년 유니코드 2.0을 발표하였다. 유니코드 2.0에서는 2바이트로 표현할 수 있는 문자 65,536개를 모아서 하나의 문자평면(character plane)으로 구성하고, 총 17개의 문자평면을 제공하도록 하였다. 이 표준에서는 1,114,112(65,536 * 17)개의 문자를 표현할 수 있으며, 문자 코드의 범위는 0x0000로부터 0x10FFFF까지이다. 17개의 문자평면 중 첫 평면은 기본 다국어 평면(BMP: Basic Multilingual Plane)이라고 하며, 기존 유니코드 1.0과 코드 번호가 일치한다. 나머지 16개의 평면은 보충 평면(supplementary plane)이라고 하며, 첫 번째 보충 평면에만 중국에서 자주 사용되지 않는 한자(드문 성씨와 같은), 이모지(emoji) 등의 문자들이 들어가 있고 나머지 평면은 비어있다. 유니코드 2.0 문자 코드를 모두 표현하기 위해서는 적어도 3바이트가 필요하다는 것을 알 수 있다.

자바 언어는 첫 단계부터 유니코드 1.0 개발 결과를 반영하여 기존 C 언어에서 1바이트였던 char 형의 길이를 2바이트로 변경하였다. 2바이트 char 형을 사용하면 최대 65,536개의 서로 다른 문자를 표현할 수 있어서 당시 유니코드 1.0의 모든 문자를 표현하는 데 문제가 없었다. 그러나 유니코드

2.0 문자 코드를 모두 표현할 수는 없다. 따라서 자바는 기본 다국어 평면에 속하는 문자는 2바이트 (코드 단위라고 부름)를, 보충 평면에 속하는 문자는 2개의 코드 단위 즉 4바이트를 이용하여 표시한다. 이 표현 방식을 UTF-16(16-bit Unicode Transformation Format)이라고 부른다. 보충 평면의 문자 코드는 기본 다국어 평면 문자 코드와 구별되도록 코드 단위 중 사용하지 않는 번호 (0xD800~0xDFFF 범위임) 두 개(surrogate pair라고 부름)를 사용하여 표현한다.

프로그램을 작성할 때 어떤 문자를 표현하기 직접적인 방법은 유니코드 번호를 사용하는 것이다. 예를 들면 다음 코드는 영어 대문자 A를 ch라는 변수에 저장하는 것이다.

```
char ch=65;
```

다른 방법으로는 작은따옴표 안에 문자를 넣어서 표시하면 된다. 위와 문장을 아래와 같이 쓸 수도 있다.

```
char ch='A';
```

문자 중 키보드로 입력하기가 어려운 문자는 회피열(escape sequence)이라는 특수한 방법을 사용하여 표시한다. 회피열은 두 종류가 있는데, 모두 역슬래시(\) 문자로 시작한다. 첫 번째 종류에서는 역슬래시 다음에 한 문자가 나와서 특수 문자의 종류를 표시한다. 예를 들어 개행문자는 '\n'으로 같이 표시한다. 다른 종류에서는 역슬래시 다음에 u 문자를 쓰고 16진수로 유니코드 번호를 쓴다. 예를 들어 한글 '가' 문자는 '\uac00'로 표시할 수 있다. 다음 표는 자주 사용되는 문자들에 대한 회피열 표현이다.

회피열	이름	유니코드
\b	백스페이스	\u0008
\t	공백	\u0009
\n	개행문자	\u000a
\\	역슬래시	\u005c
\'	작은따옴표	\u0027
\"	큰따옴표	\u0022

표 1.1 자주 사용되는 문자들에 대한 회피열 표현

자바의 문자열은 유니코드 문자들의 연쇄(sequence)이다. 즉 자바의 문자열은 코드 포인트의 연속이다. 각 코드 포인트는 하나 혹은 두 개의 코드 단위로 구성된다. 자바에서 문자열형은 char, int와 같이 기본형으로 제공되지 않고 객체(object) 형으로 제공된다. 자바의 기본 클래스 패키지인 java.lang에는 String 클래스가 존재하고, 이 클래스를 이용하여 문자열 객체를 생성하고, 필요한 처리를 할 수 있다. 자바의 문자열이 C 언어의 문자 배열과 기본적으로 다른 점은 자바 문자열은 변경 불가능 객체라는 점이다. C 언어의 경우 문자열이 문자 배열에 저장되기 때문에 각 배열의 원소 즉 문자를 변경하는 것이 가능하지만 자바의 문자열은 변경할 수 없다. 문자열 상수는 "hello"에서와 같이 큰따옴표 내부에 문자열을 넣어서 표현한다.

문자열에 적용할 수 있는 유일한 연산은 '+' 연산자를 사용하는 접속(concatenation) 연산이다. '+' 연산자는 두 문자열을 접속한 새로운 문자열을 생성한다. 접속 연산의 피연산자가 하나만 문자열이고 나머지 하나는 다른 형일 경우, 다른 형의 값이 문자열로 자동 변환된다. 즉 "hello" + 123 식에서는 먼저 123이 문자열로 변환되어 "123"이 되고 이후 "hello"와 접속되어 "hello123"이라는 결과를 생성한다.

문자열에 대한 다른 기능들은 대부분 클래스에 정의된 함수인 메소드를 통해서 제공된다. 다음은 문자열에 대해 자주 사용되는 메소드이다.

메소드	설명
char charAt(int index)	index에 의해 지정된 위치의 코드 단위를 반환한다.
int codePointAt(int index)	index에 의해 지정된 위치에서 시작하는 코드 포인트를 반환한다.
int compareTo(String other)	현재 문자열과 other 문자열을 사전 순서로 비교한 결과를 돌려준다. 만약 현재 문자열이 사전 순서로 other보다 뒤에 나오는 경우 양수를, 같으면 0을, 아니면 음수를 반환한다. C 언어의 <string.h>에 정의된 strcmp() 함수와 같은 기능을 한다.
boolean equals(Object other)	현재 문자열과 other 문자열의 내용이 같으면 true를, 아니면 false를 반환한다.
boolean startsWith(String prefix)	현재 문자열이 prefix로 시작하면 true를, 아니면 false를 반환한다. endsWith() 메소드도 있다.
int indexOf(String str)	현재 문자열에서 str 부분 문자열이 처음 발생하는 위치를 반환한다. 만약 str이 현재 문자열에 나오지 않으면 -1을 반환한다. C 언어의 <string.h>에 정의된 strstr() 함수와 비슷한 기능을 한다. 찾는 문자열이 마지막으로 발생한 위치를 찾고 싶을 경우 lastIndexOf() 메소드를 사용한다.
int length()	현재 문자열의 코드 단위 수를 반환한다.
String replace(CharSequence oldString, CharSequence newString)	현재 문자열에서 oldString을 newString으로 교체한 새로운 문자열을 반환한다.
String substring(int beginIndex, int endIndex)	beginIndex로부터 시작하여 endIndex 직전까지의 부분 문자열을 반환한다.
String trim()	현재 문자열에서 앞에 나오거나(leading) 뒤에 나오는(trailing) 모든 공백을 제거한 문자열을 반환한다.
static String join(CharSequence delimiter, CharSequence...elements)	elements 문자열들을 delimiter 구분문자열로 결합한 문자열을 반환한다.
String[] split(String regex)	regex를 구분자로 하여 현재 문자열을 문자열 분해한 결과를 배열에 담아 반환한다.

표 1.2 문자열에 대해 자주 사용되는 메소드

B. 개념 확인

1. 자바 char형의 길이는 몇 바이트인가?
2. 자바 문자 코드와 관련하여 다음 용어를 간단히 설명하시오.

용어	설명
유니코드	
UTF-16	
코드 포인트	
코드 단위	

3. 다음 문장 중 컴파일 오류를 발생시키는 것을 모두 고르시오. 단 ch의 형은 char라고 가정한다.
 - a. `ch=20;`
 - b. `ch=0x10DFEC;`
 - c. `ch=\uac01`
 - d. `ch+=3;`
 - e. `ch=ch+4;`
 - f. `ch*=2;`
 - g. `ch=ch*3;`
 - h. `ch++;`
4. 문자 '0'의 유니코드 포인트는 0x0030이다. 문자 '7'의 유니코드 포인트는 무엇인가?
5. 다음과 같이 나오도록 출력문을 작성하시오. 단 'α' 문자의 유니코드 포인트는 0x03b1이다.

√ 실행 결과

```
I am the "α"
```

6. 다음 중 문자열에 대한 설명으로 올바른 것을 모두 고르시오.
- 자바의 문자열은 모두 길이가 2바이트인 문자로 구성된다.
 - length() 메소드는 문자열의 글자 수를 반환한다.
 - 자바의 문자열은 변경할 수 없는 객체이다.
 - 두 문자열을 접속하면 기존 문자열은 변경되지 않고 접속된 새로운 문자열이 생성된다.
 - 두 문자열의 내용이 같은지 검사할 때는 "==" 연산자를 사용한다.
 - 자바 문자열은 UTF-16 인코딩을 사용한다.
7. 다음 중 왼쪽의 식의 결과를 쓰시오. 단 msg="hello"라고 가정함.

식	결과
msg.substring(1, 3)	
msg.replace("l", "o")	
msg.lastIndexOf("ll")	
msg.compareTo("hel")	

8. 사용자로부터 이름과 출생 연도를 입력받아서 아래와 같이 메시지를 출력하는 다음 main 메소드를 완성하십시오. 단 사용자의 나이는 (2013 - 출생 년도 + 1)로 계산함.

```
import java.util.Scanner
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int birthYear;
    String name;
    String msg;

    System.out.println(msg);
}
```

√ 입력

park 2010

√ 실행 결과

환영합니다 park님. 당신은 14세입니다.

9. 사용자로부터 한 문자열을 입력 받아서 다음과 같이 한 글자씩 뒤로 회전한 문자열들을 출력하는 다음 프로그램을 완성하시오(substr 응용, length 사용)

```
import java.util.Scanner
public static void main(String[] args) {
    }
}
```

√ 입력

hello

√ 실행 결과

```
hello
elloh
llohe
lohel
ohell
```

10. 어떤 문자열을 매개변수로 받아서 역순으로 된 문자열을 반환하는 아래 reverse 메소드를 문자열 접속을 이용하여 완성하시오(loop, substring, length 활용). 예를 들면 다음과 같음.

인수	반환값
"hello"	"olleh"

```
public String reverse(String str) {
}
}
```

11. 위의 문제를 StringBuilder 객체를 이용하여 다시 작성하시오.

```
public String reverse(String str) {
}
}
```

C. 응용 문제

1. 한글에서 사용하는 자모 글자의 종류 다음과 같다.

번호	초성	중성	종성
0	ㄱ	ㅏ	없음(채움)
1	ㄲ	ㅑ	ㄱ
2	ㅋ	ㅓ	ㄲ
3	ㆁ	ㅕ	ㄱ
4	ㄷ	ㅗ	ㄴ
5	ㄸ	ㅛ	ㄴ
6	ㄹ	ㅜ	ㄴ
7	ㄺ	ㅠ	ㄴ
8	ㅂ	ㅡ	ㄷ
9	ㅃ	ㅚ	ㄷ
10	ㅅ	ㅜ	ㄹ
11	ㅆ	ㅠ	ㄹ
12	ㅇ	ㅛ	ㄹ
13	ㅈ	ㅜ	ㄹ
14	ㅊ	ㅠ	ㄹ
15	ㅋ	ㅛ	ㄹ
16	ㅌ	ㅜ	ㄹ
17	ㄴ	ㅡ	ㄹ
18	ㄷ	ㅛ	ㅁ
19	ㄸ	ㅜ	ㅁ
20	ㄹ	ㅡ	ㅁ
21			ㅇ
22			ㅈ
23			ㅊ
24			ㅋ
25			ㅌ
26			ㄴ
27			ㅇ

유니코드 표에서 한글 완성 글자들은 먼저 초성으로 정렬된 다음 같은 초성 내에서는 중성으로 정렬되고 다시 같은 중성 내에서는 종성으로 정렬되어 있다. 따라서 완성형 글자들은 '가', '각' '갉', ..., '나', '낙', '난', ..., '하', '학', '한' ..., '힝' 순서로 나열되어 있다. 그리고 '가' 글자의 코드는 0xac00 (십진수로 44032)이다. 어떤 한글 글자 1자를 인수로 전달받아서 이 한글 글자의 초성, 중성, 종성 번호를 출력하는 다음 메소드를 완성하시오. 예를 들면 '달'이라는 글자가 인수로 들어 오면 초성 번호는 3, 중성 번호는 0, 종성 번호는 8을 출력함. 만약 입력 글자가 한글이 아니라면 모두 -1을 출력하시오.

```
public void hcharDecompose(char ch) {
}
```

2. 인터넷에서 사용되는 URL(Uniform Resource Locator)은 다음과 같은 구조를 가진다.

protocol : // host [":" port] path ["?" query] ["#" fragment]

어떤 URL이 주어지면 이 URL을 위의 구성 요소로 분해하여 출력하는 메소드를 작성하시오

```
public void URLDisassemble(String url) {
}

```

√ 입력

```
http://www.java.com:8080/index?name=kim&age=12#content
```

√ 실행 결과

```
프로토콜: http
호스트: www.java.com
포트: 8080
경로: /index
질의: name=kim&age=12
조각: content
```

3. 어떤 문자열을 인수로 전달 받아서 해당 문자열의 순열(permutation)을 모두 출력하는 메소드를 작성하시오. 예를 들어 인수가 "abc"라면 "abc", "acb", "bac", "bca", "cab", "cba" 등 6개의 문자열을 출력해야 함. 재귀를 이용한 버전과 반복을 이용한 버전 2가지를 작성하시오.

```
public void printPermutations(String s) {
}

```

4. "hello"라는 문자열이 100,000번 반복되는 문자열을 작성하려고 한다. 문자열 접속 연산자 '+'를 이용하여 이 문자열을 작성하는데 걸리는 시간을 측정하라. 또한, StringBuilder 객체를 작성한 후 이 객체에 "hello"라는 문자열을 100,000번 추가하는 작업을 하는데 걸리는 시간을 측정하라. 두 방법 사이에 시간을 비교하고 왜 이런 차이가 나는지 설명하시오.

```
public class StringBuilderTest {
}

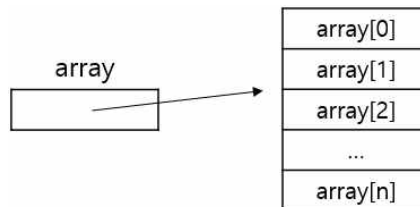
```

1.2 배열

A. 개념 정리

1) 배열

배열은 같은 자료형에 속하는 여러 개의 자료를 저장하는 자료 구조이다. 배열에 저장된 개별 자료를 요소(element)라고 부르고, 각 요소에 접근하기 위해서는 색인(index)을 사용한다. C, C++ 등과 유사하게 자바에서도 색인은 0부터 시작한다. 배열은 객체이기 때문에 배열을 참조하는 변수와 배열 자체는 다른 개념이다. 아래 그림에서 array는 배열 참조 변수(이하 배열 변수)이다. 이 변수는 배열에 대한 참조를 값으로 갖는다.



배열을 생성하고 처리하기 위해서는 배열 변수와 배열 생성이 필요하다. 배열 변수의 선언은 "원소형 [] 변수이름" 형태이다. 아래 선언은 array가 int형 원소를 갖는 배열 변수라는 의미이다.

```
int[] array;
```

다음으로 배열을 생성하여 그 참조를 배열 변수에 저장한다. 배열을 생성하기 위해서는 다른 객체 생성과 마찬가지로 new 연산자를 사용한다. 다만 배열의 크기를 지정하기 위해서 소괄호가 아니라 대괄호를 사용하며, 원시 자료형 배열의 경우 클래스 이름이 아니라 자료형 이름을 사용한다. 다음은 int형 자료를 3개 저장하기 위한 배열을 만들어서 array라는 배열 변수에 저장하는 명령이다. 배열을 한번 생성이 된 후에는 그 크기를 변경할 수 없다.

```
array = new int[3];
```

배열을 생성할 때 초깃값을 줄 수 있다. 초깃값을 지정할 때는 중괄호를 사용한다. 다음은 위에서 생성한 배열에 초깃값으로 1, 2, 3을 주는 명령이다. 초깃값을 지정하지 않으면 디폴트 값으로 지정된

다. 숫자의 경우 디폴트 값은 0이고 논리형의 경우는 false이다.

```
array = new int[3] {1, 2, 3};
```

위의 단계들을 간략화하여 다음과 같이 배열 변수를 선언할 때 바로 초깃값을 함께 줄 수도 있다. 이 경우 배열의 크기를 지정하지 않으면 컴파일러가 초깃값의 숫자를 세서 배열을 크기를 지정해 준다.

```
int[] array = {1, 2, 3}
```

배열은 객체이기 여러 가지 필드가 존재하는데 그 중 length 필드는 배열의 크기를 저장하는 필드이다. 위의 예에서 array.length는 3의 값을 갖는다. 이 length 필드를 이용하면 색인을 통해 배열 전체 원소를 순회하는 코드를 쉽게 작성할 수 있다. 아래는 array 배열의 모든 원소를 순회하여 출력하는 코드이다.

```
for (int idx = 0; idx < array.length; idx++)
    System.out.println(array[idx]);
```

배열을 포함하는 컬렉션에 대한 순회가 매우 자주 발생하기 때문에 자바는 이를 위한 새로운 반복 문장을 제공한다. 통상 for-each 문장으로 불리는 반복문은 "for(변수: 컬렉션) 문장" 형태이다. 위의 반복문을 for-each 문장을 이용하여 다시 작성하면 아래와 같다.

```
for (int element: array)
    System.out.println(element);
```

자바 프로그램 실행은 어떤 클래스에 있는 main 메소드부터 시작된다. 프로그램이 시작될 때 main 메소드에 인수를 줄 수 있는데 이것을 명령줄 인수(command line arguments)라고 부른다. 예를 들면 SampleClass에 main 메소드를 실행할 때 다음과 같이 실행하면

```
$ java SampleClass simple sample class
```

이 클래스의 main 메소드는 "simple", "sample", "class" 등 세 개의 인수가 들어있는 배열을 매개 변수를 통하여 전달받는다. 만약 main 메소드가

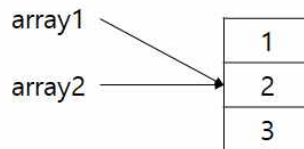
```
public static void main(String[] args) {...}
```

와 같이 선언된 경우, args[0]은 "simple", args[1]은 "sample", args[2]는 "class" 문자열을 참조하게 된다.

2) 복사, 정렬 (Arrays 클래스 API)

자바 배열은 객체이기 때문에 아래와 같이 배열을 다른 변수에 지정하는 경우 원래 변수와 다른 변수는 같은 배열을 공유한다.

```
int[] array1 = {1, 2, 3}
int[] array2 = array1;
```



따라서 array1을 통해서 어떤 요소를 변경할 경우 array2의 요소도 변경된다. 이렇게 공유되는 것을 방지하기 위해서는 배열을 복제한 후 다른 변수에 저장해야 한다. 배열을 복사하기 위해서는 java.util 패키지의 Arrays 클래스에 있는 copyOf() 메소드를 사용한다. 이 메소드는 두 번째 매개 변수인 length를 이용하여 배열의 크기를 줄이거나 늘이면서 복제할 수도 있다. 다음은 array1의 크기를 2배로 늘여서 array2에 복제하는 명령이다. 이때 array2의 처음 3개 요소는 array1의 요소와 값이 같고 나머지 요소들은 숫자의 디폴트 값인 0으로 초기화된다.

```
array2 = Arrays.copyOf(array1, array1.length * 2)
```

Arrays 클래스는 복제 이외에도 배열과 관련된 여러 가지 기능을 제공하는데 다음은 Arrays 클래스가 제공하는 주요 메소드이다. 이 메소드들은 오버로딩되어 있기 때문에 아래에서는 자료형을 따로 표기하지 않았다.

메소드	설명
static int binarySearch(arr, key)	배열에서 이진탐색을 이용하여 키의 위치를 탐색한다.
static int compare(arr1, arr2)	두 배열을 사전 순서로 비교한다.
static type[] copyOf(arr, length)	크기가 length인 배열을 생성하고, arr 내용을 복제하여 반환한다.
static boolean equals(arr1, arr2, begin, end)	두 배열을 begin부터 end 사이가 같은지 비교한다.
static void sort(arr)	배열을 오름차순으로 정렬한다.
static String toString(arr)	배열을 출력을 위해 배열을 문자열로 바꾸어서 반환한다.

3) 다차원 배열

다차원 배열은 개별 요소에 접근하기 위해 하나 이상의 색인을 사용해야 하는 배열이다. 요소에 접근하기 위한 색인 수가 2개이면 2차원 배열, 3개이면 3차원 배열 등으로 부른다. 2차원 이상의 배열을 다차원 배열이라고 한다. 엄격하게 말하면 자바의 모든 배열은 1차원 배열이다. 2차원 배열은 1차원 배열의 배열이다. 마찬가지로 3차원 배열을 2차원 배열의 배열이다. 따라서 모든 배열은 1차원인데 각 요소가 다시 배열일 수도 있다. 다만 다른 프로그래밍 언어 등에서 2차원 배열, 다차원 배열 등의 용어를 사용하기 때문에 이에 따라서 이 용어들을 사용한다.

다음은 2차원, 3차원 배열 변수를 선언하는 예이다.

```
int[][] matrix;
double[][][] cube;
```

1차원 배열의 경우와 마찬가지로 배열 변수를 선언할 때 배열 할당 및 초기값을 지정을 동시에 할 수도 있다. 다음은 matrix 배열 변수에 2x3 형태의 2차원 배열을 지정하는 예이다.

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

한편 아래와 같이 먼저 배열 변수를 먼저 선언한 후 각 배열 요소들을 채워나갈 수도 있다.

```
int [][] matrix;
matrix = new int[2][];
matrix[0] = new int[] {1,2,3};
matrix[1] = new int[] {4,5,6};
```

위의 matrix와 같이 모든 배열 요소의 모양이 같은 경우(이 경우는 모두 크기가 3인 배열이다) 사각 배열(rectangular array)이라고 한다. 그러나 다음과 같이 각 요소의 모양이 다르면 이 배열을 톱니 모양 배열(ragged array)이라고 부른다.

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

이 예에서 ragged 배열의 첫 번째 요소는 크기가 2인 배열이고 두 번째 요소는 크기가 3인 배열이다. 배열 변수를 선언한 후에 각 요소를 추가하려면 다음과 같이 할 수 있다.

```
int[][] ragged;
ragged = new int[3][];
ragged[0] = new int[2] {1,2};
ragged[1] = new int[3] {3,4,5};
ragged[2] = new int[2] {6,7}
```

B. 개념 확인

1. 아래 2줄의 문장을 같은 역할을 하는 1줄 문장으로 바꾸시오.

```
int[] array;
array = new int[3] {1, 2, 3};
```

2. 다음 코드의 틀린 점을 모두 찾아 쓰시오.

```
int[] array;
array = new int[3] {1, 2, 3};
array[3] = array.length() + 2;
```

3. 다음 문장을 for-each 문장을 사용하여 다시 작성하시오.

```
for (int idx = 0; idx < array.length; idx++)
    System.out.println(array[idx]);
```


4. 어떤 double형 배열을 전달받아 그 배열에 저장된 수 중 최솟값과 최댓값을 출력하는 다음 메소드를 완성하시오. 단 배열의 크기는 1 이상이라고 가정한다.

```
public void printMinMax(double[] array) {
}
```

5. array1에 1부터 100 사이의 랜덤 정수 10개를 저장한 후 array2에다 array1를 지정하시오. 이후 array1의 요소들을 모두 2배 한 후 array2의 내용을 출력하시오.

```
public static void main(String[] args) {
}
```

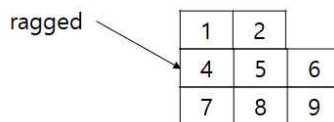
6. array1에 1부터 10사이의 정수 난수 10개를 저장하라. Arrays 클래스의 copyOf 메소드를 이용하여 array1을 같은 크기의 배열 array2로 복제하라. array1의 모든 원소를 2배한 후 array1과 array2를 출력하라.

```
public static void main(String[] args) {
}
```

7. 배열의 중앙값을 구하는 문제를 Arrays 클래스의 정렬을 이용하여 풀어보시오. 단 원본 배열을 변경하지 말고 배열을 복제하여 정렬한 후 중앙값을 구하시오.

```
public void findMedianBySort(int[] array) {
}
```

8. 아래 그림과 같이 ragged 배열 변수가 톱니바퀴 모양 배열을 참조하도록 배열 선언 및 초기화 문장 작성하시오.



9. 문제 1의 ragged 배열의 요소들을 차례로 순회하면서 다음과 같이 출력하는 반복 문장을 작성하시오.

√ 실행 결과

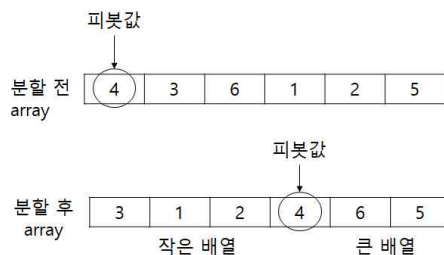
```
{{1, 2}, {4, 5, 6}, {7, 8, 9}}
```

C. 응용 문제

- 어떤 값들을 크기에 따라 정렬했을 때 가장 중앙에 위치하는 값을 중앙값(median)이라고 한다. 예를 들면 값 1, 3, 9, 14, 18이 있으면 중앙값은 9이다. 만약 값이 짝수개 있다면 중앙값은 가운데 있는 두 수의 평균으로 한다. 어떤 정수 배열 array를 전달받아 중앙값을 반환하는 메소드를 다음 지시에 따라 완성하라. 단 값들은 모두 양의 정수이며 서로 다르다고 가정한다.
 - array 배열과 같은 크기의 int 배열 numHigher를 할당받고 0으로 초기화하라.
 - array 배열을 순회하면서 numHigher 배열의 값을 계산하라. numHigher[i]의 값은 array 배열에서 array[i]보다 큰 요소의 숫자다.
 - array 배열의 원소 수가 홀수이면 numHigher에서 array.length/2 값을 갖는 위치 idx를 찾아서 array[idx] 값을 중앙값으로 반환한다.
 - array 배열의 원소 수가 짝수이면 numHigher에서 array.length/2-1 및 array.length/2인 값의 위치 idx1, idx2를 찾아서 array[idx1]과 array[idx2]의 평균값을 반환한다.

```
public void findMedian(int[] array) {
}
```

- 퀵 정렬(Quick sort)에서는 먼저 배열을 피벗값을 기준으로 피벗값보다 작은 값으로 구성된 배열과 피벗값보다 큰 값으로 구성된 배열로 분할(split)하는 처리를 먼저 한다. 예를 들면 배열 array가 다음과 같을 때 피벗값을 array[0]이라고 한다면 이 배열을 분할 알고리즘에 의해 다음과 같이 분할된다. 이러한 분할 알고리즘을 구현하시오



```
public int splitArray(int[] array) {
}
```

3. 두 정수 행렬을 인수로 전달받아서 이들의 행렬 곱을 계산하여 반환하는 multiplyMatrix 메소드를 작성하시오. 단 두 행렬이 곱할 수 없는 모양이면 null을 반환하도록 함. main 함수에서 [[1,2],[3,4],[5,6]] 행렬과 [[1,2,3],[4,5,6]] 행렬의 곱을 출력하시오.

```
public static main(String[] args) {  
  
}  
  
public static int[][] multiplyMatrix(int[][] mat1, int[][] mat2) {  
  
}
```

2장 클래스 I

2.1 클래스와 객체

2.2 새로운 클래스 작성

2장 클래스 I

2.1 클래스와 객체

A. 개념 정리

자바는 객체지향 프로그래밍을 위한 언어이다. 객체지향 프로그래밍에서는 프로그래머가 객체를 만들고 객체들끼리 상호작용하면서 원하는 작업을 수행한다. 자바는 프로그래머가 해결하려고 하는 문제에 존재하는 사물 혹은 개념들을 표현하기 위한 도구로서 객체라는 개념을 제공한다. 이 객체라는 개념은 매우 일반적인 개념이기 때문에 대부분의 문제 영역에서 사용할 수 있는 개념이다.

객체가 문제 영역의 개념 혹은 사물을 표현하는 것이라 하더라도 결국은 컴퓨터 내부에서 구현되어야 한다. 이러한 관점에서 본다면 객체는 하나의 작은 컴퓨터로 비유할 수 있다. 즉 객체는 컴퓨터와 마찬가지로 상태 및 상태를 변경시키는 연산으로 구성된다. 객체란 어떤 자료(객체 필드라고 함)와 그 자료를 처리하는 함수(메소드라고 부름)를 하나의 단위로 결합한 것을 의미한다. 객체는 기본적으로 자료 및 그 구현 방법을 외부에 숨기고 자료 조작에 필요한 기능만 인터페이스로 외부에 공개한다. 이렇게 자료 및 구현 방법을 숨기는 것을 캡슐화(encapsulation)라고 한다. 캡슐화는 자료와 해당 자료를 조작하는 기능을 프로그램 일부에 한정함으로써 프로그램 개발에 있어서 모듈화 및 유지보수에 장점이 있다.

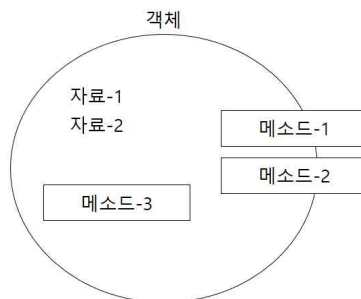


그림 2.1 객체의 예

위의 그림은 객체의 예를 보여주고 있다. 이 객체는 자료-1, 자료-2를 가지고 있지만, 외부에 공개하지 않고 있다. 외부에 공개되는 것은 메소드1, 메소드2이다. 외부에서는 이 객체에 대해 이러한 기능만을 사용할 수 있다.

클래스는 객체를 생성하기 위한 틀(template) 혹은 설계도이다. 객체지향 프로그래밍에서는 클래스를 이용하여 같은 모양을 갖는 객체를 여러 개 만들 수 있다. 이렇게 만들어진 객체들은 같은 객체 필드 및 메소드를 가진다는 점에서 모양이 같다고 할 수 있다. 그러나 생성되는 객체마다 객체 필드는 다른 값을 가질 수 있다. 객체 필드 값들을 그 객체의 상태(state)라고 한다.

자바 언어는 많은 수의 클래스를 제작하여 클래스 라이브러리 형태로 프로그래머에게 제공한다. 이들 클래스 중 java.lang 패키지(이 개념은 뒤에 나옴)에 속하는 클래스는 따로 수입하지 않아도 프로그램에서 사용할 수 있다.

java.lang 패키지가 제공하는 클래스 중 하나는 Math 클래스이다. Math 클래스에는 수학 계산과 관련된 다양한 상수와 함수들이 포함되어 있다. 이 클래스에는 원주율 PI가 들어가 있는데 이 PI 값은 모든 Math 객체에 따로 저장될 필요가 없이 어느 한 곳에 저장되어 있으면 충분하다. 이렇게 각각의 객체에 저장될 필요가 없이 한 곳에만 저장해도 좋은 필드나 메소드를 정적(static) 멤버라고 부르고, 클래스에 이 멤버들을 저장한다. 이러한 멤버들은 "클래스이름.멤버이름" 형태로 접근할 수 있다.

한편 각 객체마다 따로 저장되는 멤버들은 객체 멤버라고 부른다.

B. 개념 확인

1. 캡슐화란 무엇인지 간단히 설명하시오.
2. 객체와 클래스는 어떻게 다른지 설명하시오.
3. Math 객체가 제공하는 함수를 이용하여 4.326.3 및 e3.1을 계산하여 출력하는 문장을 작성하시오. 자바 API 문서를 참고하시오.

```
public class MathTest {
}

```

4. LocalDate API 문서를 참고하여 myBirthday, yourBirthday 객체를 각각 만들고, myBirthday는 2010년 4월 10일, yourBirthday는 2011년 10월 3일이 저장되도록 하시오. 또한 현재 날짜를 나타내는 객체를 따로 만들어서 myAge, yourAge를 계산하시오. 나이 계산은 (현재년도-생일년도+1)로 계산하시오.

```
public class LocalDateTest {
}
```

C. 응용 문제

1. 사용자로부터 연도와 월을 양의 정수로 입력받아서, 그달의 달력을 출력하는 프로그램을 작성하시오. LocalDate 클래스 of(), getDayOfWeek(), getMonthLength() 등 필요한 메소드를 활용하시오. 만약 사용자가 연도와 월을 입력하지 않으면 이번 달 달력을 표시하도록 하시오.

```
public class PrintCalendar {
}
```

2. java.util 패키지에는 Random 클래스가 있어서 정수, 실수 등의 난수를 발생시킬 수 있다. 이 클래스를 사용하여 1부터 100까지 범위의 난수를 하나 발생시키고, 사용자가 해당 숫자를 맞출 때까지 몇 번을 추측하였는지 횟수를 출력하는 프로그램을 작성하시오. 아래와 같이 사용자가 추측한 숫자를 입력하면 정답이 입력된 숫자보다 큰지 혹은 작은지를 출력하시오.

```
public class GuessNumber {
}
```

√ 실행 결과

```
1부터 100 사이 난수를 생성하였습니다
추측: 12
  >> 12보다 큽니다
추측: 78
  >> 78보다 작습니다
...
추측: 63
  >> 정답입니다. 총 8회 추측하였습니다.
```

2.2 새로운 클래스 작성

A. 개념 정리

자바에서 새로운 클래스를 작성하려면 `class`라는 키워드를 먼저 쓰고 한 후 클래스 이름을 쓴다. 이후 중괄호 내부에 클래스의 내용을 적는다. 다음은 사람 객체를 만들기 위한 `Person` 클래스를 정의한 예이다.

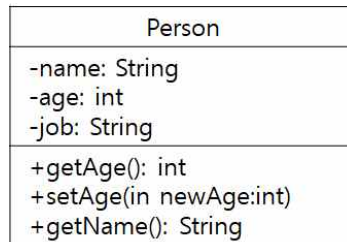
```
public class Person {
    private String name;
    private int age;
    private String job;

    public void setAge(int newAge) {
        age = newAge;
    }
    public int getAge() {
        return age;
    }
    public String getName() {
        return name;
    }
}
```

이 클래스는 `name`, `age`, `job` 등 3개의 객체 필드와 `setAge()`, `getAge()`, `getName()` 등 3개의 메소드로 구성되어 있다. 객체 필드와 메소드 선언 앞에도 접근 제한자가 나올 수 있다. `public` 접근 제한자는 그 단어가 의미하는 바와 같이 모든 외부 클래스에서 이 필드나 메소드를 사용할 수 있음을 의미한다. 반면 `private`로 지정된 필드나 메소드는 이 클래스 내부의 메소드에서만 사용할 수 있음을 의미한다. 접근 제한자를 생략하면 디폴트 접근 제한자가 되고 이것은 같은 패키지(패키지는 뒤에서 공부함) 내에서는 자유롭게 접근할 수 있음을 의미한다. 마지막으로 `protected`라는 접근 제한자가 있는데 이것은 같은 패키지뿐만 아니라 하위 클래스(이 개념을 다음에 나오는 상속을 다루는 장에서 공부한다)에서도 접근 가능하다는 의미이다.

클래스의 구조를 그림으로 표현할 때는 UML(Unified Modeling Language) 클래스 다이어그램을 주로 사용한다. UML 클래스 다이어그램에서는 클래스를 3개로 칸으로 구성된 사각형으로 그린다. 맨 위 칸에는 클래스 이름을, 두 번째 칸에는 객체 필드를, 세 번째 칸에는 메소드를 적는다. 위의

Person 클래스를 UML 다이어그램으로 표시하면 다음과 같다. 아래에서 앞에 나오는 - 기호는 private, + 기호는 public을 의미한다.



클래스 이름 앞에도 접근 제한자가 나올 수 있는데 public과 디폴트이다. 그 의미는 객체 필드와 같다.

작성된 클래스를 이용하여 객체를 생성한다. 객체를 생성하려면 객체 필드를 초기화할 필요가 있다. 자바에서는 객체를 생성하면서 객체 필드를 초기화하는 특별한 메소드를 사용하는데 이 메소드를 생성자(constructor)라고 부른다. 생성자는 메소드는 클래스와 같은 이름을 가지며 반환형은 없다. 한 클래스에서 여러 개의 생성자를 쓸 수 있다. 이렇게 같은 이름의 메소드가 여러 개 정의되는 경우 메소드가 중복(overloaded)되었다고 한다. 다음은 Person 클래스의 생성자를 작성한 예이다. 만약 사용자가 어떤 클래스에 생성자를 하나도 정의하지 않으면 컴파일러가 빈 생성자를 하나 넣어준다. 빈 생성자는 매개변수가 없고 아무 일도 하지 않는 생성자이다. 그러나 사용자가 생성자를 하나라도 정의하면 컴파일러는 빈 생성자를 넣어주지 않는다.

```
public class Person {
    // 중복되는 부분은 생략
    public Person() {
        name = "unknown";
        age = 0;
        job = "unknown";
    }
    public Person(String name, int age, String job) {
        this.name = name;
        this.age = age;
        this.job = job;
    }
    // 나머지 메소드 생략
}
```

생성자 혹은 메소드를 작성할 때 전달되는 인수가 객체의 필드와 이름이 같은 경우가 많다. 예를 들면 위의 Person 생성자에서 name, age, job 인수는 객체의 name, age, job 필드와 이름이 같다. 이 경우 생성자나 메소드에서 name, age 등의 이름을 사용하는 경우 인수의 이름으로 취급된다. 객체 필드를 참조하기 위해서는 this라는 키워드를 이름 앞에 붙여야 한다. this가 생성자 혹은 메소드 내부에서 사용되는 경우는 현재 객체를 의미한다.

한편, 생성자가 여러 개 필요한 경우 이미 작성한 생성자를 활용하기 위하여 this 키워드를 사용하기도 한다. 예를 들어 Person 클래스에서 다음과 같은 생성자가 있다고 가정하자.

```
public Person(String name, int age) {
    this.name = name;
    this.age = age;
    this.job = "unknown";
}
```

이 경우 아래 생성자는 위의 생성자를 이용하여 간단히 작성할 수 있다.

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

이 코드에서 this()는 다른 생성자를 호출한다. 앞에서 사용한 this와 다른 점은 this 다음에 소괄호가 나오는 것으로 생성자 호출임을 알 수 있다.

메소드는 입력을 받을 수 있다. 위의 Person 클래스에서 setAge()는 정수를 하나 입력으로 전달받는데, 이 정수를 메소드 내부에서는 newAge로 부르고 있다. 따라서 외부에서 setAge() 메소드를 호출한다면 다음과 같은 형태일 것이다.

```
Person p1 = new Person();
int age = 10;
p1.setAge(age);
```

이때 메소드 호출에 나오는 변수를 인자(argument) 혹은 실인수(real parameter)라고 부르고, 메소드 정의에 나오는 변수를 매개변수(parameter) 혹은 형식인수(formal parameter)라고 부른다. 자

바에서 호출 인자로부터 매개변수로 값을 전달하는 방법은 오직 한가지이다. 인자의 값이 형식인수로 복사되는 "값에 의한 호출(call-by-value) 방식이다. 다만 전달되는 값이 기본형일 경우 그대로 값이고 객체인 경우는 객체에 대한 참고가 전달된다. 생성자의 경우와 마찬가지로 한 클래스 내에는 같은 이름을 가지는 메소드가 여러 개 존재할 수 있다. 이때 같은 중복된 메소드를 컴파일러가 서로 구별할 수 있어야 한다. 컴파일러는 메소드를 구별하기 위해 각 메소드의 매개변수 개수 및 형을 사용한다. 만약 두 메소드의 반환형이 다르지만, 매개변수 개수 및 형이 같다면 컴파일러는 오류를 발생시킨다. 만약 사용자가 메소드의 매개변수와 다른 형의 인수를 제공한다면 컴파일러는 확장 형 변환으로 인수의 형을 매개변수 형과 맞출 수 있다면 자동으로 인수의 형을 변환한다.

B. 개념 확인

1. 다음 클래스에 대해 답하십시오.

```
public class Student {
    private static int nextStudentId = 1;
    private int studentId;
}
```

- 1) 정적 필드는 무엇인가?
 - 2) 인스턴스 변수는 무엇인가?
2. 다음 클래스에 관한 다음 말 중 잘못된 것은 어느 것인가?
- a. 필드는 초기화하지 않아도 된다.
 - b. 클래스는 생성자 없이 작성해도 된다.
 - c. 생성자의 우선적인 기능은 필드 초기화이다.
 - d. 필드는 생성자에 의하지 않고는 초기화할 수 없다.
3. Student 클래스 객체를 생성하면 자동으로 studentId가 1번부터 차례대로 부여되도록 생성자를 작성해 보시오.

```
public Student() {
}
```

4. 다음 중 클래스의 구성 요소에 속하지 않은 것은 어느 것인가?

- a. 객체 필드
- b. 생성자
- c. 메소드
- d. 지역변수

5. 다음 클래스의 오류를 찾아내시오.

```
class StaticMember {
    static double x;
    static double getX() {
        return x;
    }
    double y;
    double getY() {
        return y;
    }
    double returnX() {
        return getX();
    }
    static double returnY() {
        return getY();
    }
}
```

6. 다음 문장에 대해 맞으면 O, 틀리면 X를 표시하시오,

- 1) static 변수는 객체 변수와 동일한 성질을 가진다.
- 2) static 변수는 그 클래스로부터 생성된 모든 객체에 의해 공유된다.
- 3) 객체 p의 어떤 객체 변수 y도 p.y 같은 방식으로 접근할 수 있다. 같은 방법으로 static 변수 x도 p.x로 접근할 수 있다.
- 4) 어떤 static 필드도 static 메소드에서 사용할 수 있다.
- 5) 어떤 정적 필드도 객체 메소드(Non-static Method)에서 사용할 수 있다.
- 6) 객체 메소드를 정적 메소드에서 호출할 수 있다.
- 7) 클래스의 정적 메소드는 그 클래스로부터 생성된 객체의 객체 메소드에서 호출할 수 있다.
- 8) 필드, 지역변수, 메소드의 형식인자 모두 final로 선언될 수 있다.
- 9) static 필드는 초기화하지 않으면 타입의 기본값으로 자동 초기화 된다.

7. 다음 클래스 정의의 네모 속에 넣어도 좋은 것을 모두 고르시오.

```
public class Ex {
    public static void main(String[] argv) {}
    public void work(int i) {}
    
}
```

- a. public void work(int z) {}
- b. public int work(int i, int j) {return 99;}
- c. protected void work(float f) {}
- d. private void p() {}
- e. int work(int k) {}

8. 메소드 add()가 다음과 같이 중복되어 있다. 컴파일 오류를 발생시키는 것을 모두 고르시오.

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

- a. add(1,2)
- b. add(1.0, 2.0)
- c. add(1.0, 2)
- d. add(1, 2.0)

9. 다음 Java 프로그램의 실행 결과는 무엇인가?

1)

```
public class SwapTest1 {  
    static void swap(int x, int y) {  
        int tmp = x;  
        x = y;  
        y = tmp;  
    }  
    public static void main(String[] args) {  
        int a = 10, b = 20;  
        swap(a, b);  
        System.out.println("a = "+a + "b = "+b);  
    }  
}
```

2)

```
class Point {
    public int x;
    public int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
public class SwapTest2 {
    static void swap(Point p) {
        int tmp = p.x;
        p.x = p.y;
        p.y = tmp;
    }
    public static void main(String[] args) {
        Point p = new Point(10, 20);
        swap(p);
        System.out.println("p.x = "+p.x + "p.y = "+p.y);
    }
}
```

3)

```
class Point {
    public int x;
    public int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
public class SwapTest3 {
    static void swap(Point p1, Point p2) {
        Point tmp = p1;
        p1 = p2;
        p2 = tmp;
    }
    public static void main(String[] args) {
        Point p1 = new Point(10, 20);
        Point p2 = new Point(30, 40);
        swap(p1, p2);
        System.out.println("p1.x = "+p1.x + "p2.x = "+p2.x);
    }
}
```

C. 응용 문제

1. (생성자, static) 다음과 같은 클래스 Triangle을 정의하시오. 단 생성되는 객체는 생성되는 순서대로 일련번호를 가진다.

필드	double형의 높이(height)	
	double형의 밑변(width)	
	int형의 serialNo	
생성자	기본 생성자	
	높이와 밑변의 값을 인자로 받는 생성자	
메소드	double area()	삼각형의 면적을 계산하여 반환한다
	void print()	객체 일련번호, 너비, 높이, 면적을 포함하는 객체의 내용을 인쇄하는 메소드. 예를 들면 "객체번호: 3 너비: 12.0 높이: 5.0 면적: 30.0"을 출력하고 줄 바꿈
	main()	10개의 객체를 적당한 너비와 높이로 생성한다. 7번째로 생성된 삼각형 객체를 인쇄한다.

```
public class Triangle {
}

```

2. 친구 목록을 관리하는 클래스 FriendList를 작성하시오. 친구는 Friend 클래스의 객체로 표시되며 name과 mobile을 필드로 갖는다. 각 필드는 String 형이다. 처음 FriendList 객체 생성하면 5개의 Friend 객체를 저장할 수 있는 배열을 만들어서 저장한다. 생성될 때 이 배열은 비어있다. 사용자에게 다음과 같이 메뉴를 보여주고 사용자가 선택한 번호에 따라서 작업을 수행한다.

1. 친구 추가
2. 친구 삭제
3. 친구 목록 출력
4. 친구 목록 전체 삭제
5. 종료

친구를 추가할 경우 친구의 이름 및 휴대전화 번호를 입력받아 Friend 객체를 만든 후 FriendList 객체에 있는 배열에 저장한다. 만약 같은 이름의 친구가 있는 경우는 배열에 추가하지 않는다. 만약 배열이 가득차서 빈 공간이 없는 경우는 배열의 크기를 2배로 늘린 후 새 배열에 기존 친구 목록 및 새로운 친구를 저장한다. 친구 삭제에서는 친구 이름을 받아서 배열에서 해당 친구 객체를 삭제한다. 삭제한 후 그 뒤에 저장된 친구 객체들을 앞으로 당겨서 배열에 빈 공간이 없도록 한다. 삭제하려는 이름의 친구가 없는 경우는 그냥 복귀한다. 친구 목록 출력은 현재 배열에 저장된 모든 친구들을 한 줄에 한 명씩 출력한다. 친구 목록 전체 삭제는 현재 친구 배열을 삭제하고 처음 상태

즉 크기가 5인 빈 배열 상태로 돌아간다. 각 클래스에 대해 적절한 필드 및 메소드를 작성하시오.

```
public class Triangle {  
  
}
```

```
public class FriendList {  
  
}
```

3장 클래스 II

3.1 클래스 사이의 관계

3.2 패키지

3.3 객체지향 시스템 설계

3장 클래스 II

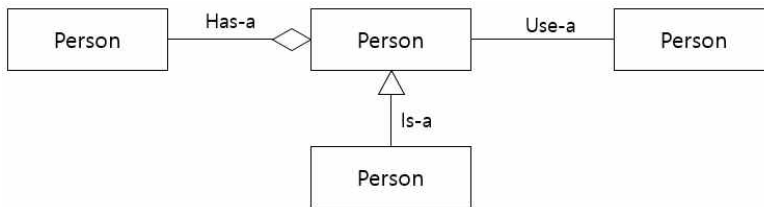
3.1 클래스 사이의 관계

A. 개념 정리

객체지향 프로그래밍에서는 여러 종류의 클래스를 정의하여 사용한다. 클래스들 사이에는 다양한 관계가 존재하는데 중요한 관계를 몇 가지 들어보면 다음과 같다.

- 의존 관계(Use-a relation): 일반적인 관계로 어떤 클래스가 다른 클래스를 사용한다는 의미이다. 예를 들면 사람이 계좌를 사용하는 것을 들 수 있다.
- 소유 관계(Has-a relation): 어떤 클래스가 다른 클래스들을 부품으로 모아서 만들어지는 경우를 말한다. 예를 들면 자동차의 경우 엔진, 몸체, 바퀴 등의 부품으로 구성된다.
- 상속 관계(Is-a relation): 전체 집합과 부분 집합의 관계를 말한다.

클래스 사이의 관계도 UML 클래스 다이어그램을 이용하여 표시한다. 위에서 언급한 세 가지 관계는 다음과 같이 표시된다.



UML 클래스 다이어그램에서 관계를 표시할 때 이 관계에 참여하는 객체의 수를 함께 표시할 수도 있다. 위의 예에서 Face와 Person 클래스 사이의 소유 관계를 예를 들면 한 명의 Person은 하나의 Face만 가질 수 있다. 반대로 하나의 Face는 한 명의 Person에만 소유될 수 있다. 이런 관계는 1:1 관계이다. 반면 Person과 Account 관계는 한 명의 Person이 여러 개의 Account를 사용할 수 있다. 반대로 하나의 Account는 한 명의 Person 소유이다. 따라서 이러한 관계는 1:n 관계이다. 가장 일반적인 관계는 m:n 관계이다. UML 클래스 다이어그램에 1:n 관계에 대해 참여 객체 수를 표시하면 아래 그림과 같다.



B. 개념 확인

1. 대학교에서 과목을 수강하는 상황을 가정해보자. 이 상황에서 과목(Subject) 클래스와 분반(Course) 클래스 그리고 학생(Student) 클래스가 필요하다. 한 과목은 여러 개의 분반을 가질 수 있으며 한 분반에는 여러 명의 학생이 수업들 듣는다. 또한, 한 학생은 여러 개의 분반에 등록한다. 이런 클래스들 사이의 관계를 참여 객체 수를 포함하여 UML 클래스 다이어그램으로 표시하시오.
2. 위에서 표현한 클래스들 사이의 관계를 반영하도록 Subject, Course, Student 클래스를 설계하여 코드를 작성하시오. 단 한 과목의 최대 분반 수는 3, 한 분반의 최대 학생 수는 20명, 한 학생이 수강할 수 있는 최대 분반 수는 5라고 가정하시오. 그 외 다른 객체 필드도 함께 설계하시오. 예를 들면 Subject 객체는 교과목 명(title)을 가져야 하며, 학생은 이름(name)과 학번(id), 분반은 분반 번호를 가져야 한다. 세 개의 클래스를 각각 별도의 파일로 저장하시오.

```

public class Subject {
}
public class Course {
}
public class Student {
}

```

3. Student와 Course 클래스 사이의 관계를 어떻게 표현하는 것이 좋은가? Course에 등록된 학생 객체를 모두 저장하고, Student 객체에 이 학생이 신청한 Course를 모두 저장하는 방법에 대해 장·단점을 토론해보시오. 같은 내용으로 Subject와 Course 사이의 관계도 어떻게 표현하는 것이 적절한 방법인지 의견을 주장해보시오.

3.2 패키지

A. 개념 정리

컴퓨터에 저장된 파일 수가 많아지면 관련 있는 파일들을 모아서 폴더로 구별하면 관리가 편하다. 객체지향 프로그래밍에서도 클래스 수가 많아지면 이렇게 그룹으로 만들어서 관리할 필요가 있다. 자바에서는 이러한 목적으로 패키지(package)를 사용한다. 자바에서 패키지는 관련 있는 클래스들을 하나의 그룹으로 묶은 것이다.

파일 폴더 내부에 다른 폴더가 포함될 수 있듯이 패키지 내부에도 다른 패키지가 존재할 수 있다. 따라서 패키지는 계층적 구조를 가진다. 자바에서 가장 상위 패키지는 java와 javax 패키지이다. java 패키지 내부에는 lang, util 등 다양한 패키지들이 포함되어 있다. 자바에서는 패키지와 패키지 이름 사이에 구분자(delimiter)로 '.' 문자를 사용한다. 따라서 java 패키지 내부에 있는 lang 패키지는 java.lang으로 표시한다.

클래스를 작성할 때 이 클래스를 어떤 패키지에 넣으려면 클래스 파일 가장 상단에 package 명령을 작성하여야 한다. 예를 들면 Student 클래스를 homework.javaclass 패키지에 넣으려면 다음과 같이 작성한다.

```
package homework.javaclass
public class Student { ... }
```

패키지 이름을 정할 때 다른 사람이 사용하는 패키지 이름과 중복되지 않도록 정하는 것이 바람직하다. 이를 위해 관습적으로 URL(Uniform Resource Locator)을 반대 순서로 표기하여 패키지 이름을 정하는 것을 권장한다. 예를 들면 어떤 사람 kim이 근무하는 회사가 java.com이라면 이 사람의 URL은 kim.java.com으로 표시할 수 있다. 이 사람이 Student라는 클래스를 만들 경우 패키지 이름을 com.java.kim.Student로 정하는 방식이다.

자바에서 어떤 클래스의 이름은 패키지이름.클래스이름 형태로 표현된다. 즉 LocalDate 클래스는 java.time 패키지에 속하기 때문에 이 클래스의 완전한 이름은 java.time.LocalDate가 된다. 따라서 어떤 LocalDate 객체를 생성하여 변수에 저장하려면 다음과 같이 코드를 작성해야 한다.

```
java.time.LocalDate now = new java.time.LocalDate()
```

위의 코드를 보면 `java.time` 패키지 이름이 반복되어 사용됨을 알 수 있다. 클래스가 많이 사용될수록 이 반복 횟수가 늘어나서 프로그래머가 불편을 느끼게 된다. 이러한 반복을 없애기 위해 `import` 문장이 도입되었다. `import` 문장은 어떤 클래스 이름을 현재 이름 공간으로 가져옴으로써 클래스 이름을 사용할 때 패키지 이름을 앞에 붙여야 하는 수고를 덜어준다. 위의 예를 `import` 문장을 통해 간단하게 만들면 다음과 같다.

```
import java.time.LocalDate;
LocalDate now = new LocalDate();
```

어떤 패키지에서 하나의 클래스가 아니라 해당 패키지의 모든 클래스를 수입하려면 다음과 같이 작성한다.

```
import java.time.*;
```

자바 컴파일러는 필요한 클래스를 어떻게 찾는가? 예를 들어 `TestStudent.java`에서 `javaExercise` 패키지에 있는 `Student`라는 클래스를 사용한다면 이 `Student` 클래스를 어떤 폴더에서 가져오는가? 자바는 `CLASSPATH`라는 환경변수를 사용하여 사용자가 정의한 클래스를 찾는 경로와 순서를 정한다. 클래스 경로를 지정하는 방법은 컴파일 혹은 실행 시 `-cp` 옵션을 사용하는 것이 한 방법이고 환경변수로 `CLASSPATH`를 지정하는 방법이 있다. `CLASSPATH` 환경변수를 지정하는 방법은 프로젝트별로 다른 경로를 지정하기 어려워서 `-cp` 옵션을 사용하는 것을 선호한다.

자바 언어의 기본 패키지는 `java.lang` 패키지이다. 이 패키지 안에 `System` 클래스와 여러 가지 포장 클래스(wrapper class)가 들어있다. 예를 들면 `boolean` 자료형에 대응하여 `Boolean` 클래스가 있다. 값은 기본 자료 형이 아닌 객체 형으로 표현할 경우 더 많은 기능을 사용할 수 있으며, 다음에 공부할 여러 가지 컬렉션에 저장할 수 있다는 장점이 있다. `Boolean` 클래스에는 문자열을 `Boolean` 객체로 바꾸는 기능, 반대로 `Boolean` 객체를 문자열로 바꾸는 기능 등이 들어있다. `int`에 대응하는 `Integer`, `float`에 대응하는 `Float`, `double`에 대응하는 `Double` 클래스도 존재한다.

B. 개념 확인

1. 사용자가 입력한 정수를 하나 읽어서 이 정수에 해당하는 이진수, 8진수, 16진수를 출력하는 IntTest 클래스를 작성하시오. java.lang.Integer 클래스의 API를 참고하여 적절한 메소드를 사용하시오.

```
public class IntTest {
}

```

2. java.lang.Double 클래스의 API를 참고하여 자바 언어에서 실수로 표현할 수 있는 가장 큰 수, 정규 표현 양수 중 가장 작은 수, 양의 무한대를 출력하시오. 그리고 각각의 64비트 값을 16진수로 출력하시오.

```
public class DoubleTest {
}

```

3. 패키지 작성 및 수입 연습

- 1) 적당한 폴더를 정해서 다음과 같은 Student 클래스를 포함하는 Student.java 파일을 작성하고, Student 클래스가 javaExercise 패키지에 속하도록 하시오.

```
public class Student {
    private int id;
    private String name;
    public Student(id, name) {
        this.id = id;
        this.name = name;
    }
}

```

- 2) 위의 파일을 javac를 이용하여 컴파일하시오. 이후 Student.class 파일이 어디에 생성되었는지 확인하시오.

- 3) 다음과 같은 TestStudent.java 파일을 작성하고, 컴파일하시오. 결과가 어떻게 나오는지 기술하시오.

```
import javaExercise.Student;

public class TestStudent {
    public static void main(String[] args) {
        Student s = new Student(12345, "lee");
        System.out.println(s.toString());
    }
}
```

- 4) 문제 (3)의 과정에서 오류가 발생한다면 오류의 원인을 설명하고 이 오류를 해결하는 방법을 제시하시오.
- 5) 현재 폴더에 tmp 폴더를 새롭게 만드시오. javaExercise 폴더를 tmp 폴더로 이동한 후 TestStudent 클래스가 잘 동작하도록 클래스 탐색 경로를 변경하시오.
4. 두 개의 별도 Student 클래스를 작성하여, 각 클래스가 서로 다른 간단한 메시지를 출력하도록 만드시오. 첫 번째 Student 클래스는 javaExercise 패키지에 속하고 두 번째 Student 클래스는 javaPractice 클래스에 속하도록 하시오.

- 1) TestStudent 클래스를 작성하고 이 클래스에서 javaExercise 및 javaPractice 패키지의 모든 클래스를 수입하시오. 어떤 일이 발생하는지 설명하시오
- 2) TestStudent 클래스에 다음과 같이 main 메소드를 작성하고 실행시켜보시오. 오류가 발생할 때 어떻게 해결할 수 있는지 설명하시오.

```
public class TestStudent {
    public static void main(String[] args) {
        Student s = new Student();
    }
}
```

5. 현재 실행 중인 컴퓨터에서 CLASSPATH 환경 변수가 있는지 확인하고 없으면 없다는 메시지를, 있으면 해당 값을 출력하는 Java 프로그램을 작성하시오. System 클래스에 getEnv 메소드가 있다. 이 메소드는 인수가 없으면 전체 환경 변수 Map을 돌려주고, 인수가 있는 경우는 해당 환경 변수의 값을 문자열로 돌려준다. 이 함수를 이용하여 전체 환경 변수 Map을 가져온 후, 해당 Map에 CLASSPATH라는 환경변수가 있는지 검사하는 방법으로 프로그램을 작성하시오. Map은 키를 값으로 대응시키는 자료 구조로 제너릭 구조로 되어 있기 때문에 Map<String><String>을 사용하시오.

3.3 객체지향 시스템 설계

A. 개념 정리

아래와 같은 시스템 개발 요구명세서가 있다고 가정하자. 이 요구명세서를 기반으로 시스템을 점진적으로 개발해보자. 시스템을 개발하는 과정에서 우리는 IBM사에서 개발된 점진적 소프트웨어 개발 체제인 Rational Unified Process(RUP)를 사용할 것이다.

수강신청 시스템(Course Registration System) 개발 의뢰가 들어왔다. 이 시스템을 사용하여 학생(Student)들은 온라인으로 이번 학기에 수강할 과목(Course)을 신청할 수 있다. 또한, 학사 학위 취득을 위해 어느 정도 수강을 완료했는지도 점검할 수 있다.

학생이 대학에 처음 입학할 때 학생을 CRS 시스템을 사용하여 학위취득을 위한 교과목 이수 계획서(Course Completion Plan)를 작성하고 지도교수(Advisor)를 선정한다. 교과목 이수 계획서는 어떤 교과목을 수강할지를 나타내는 정보가 들어가 있다.

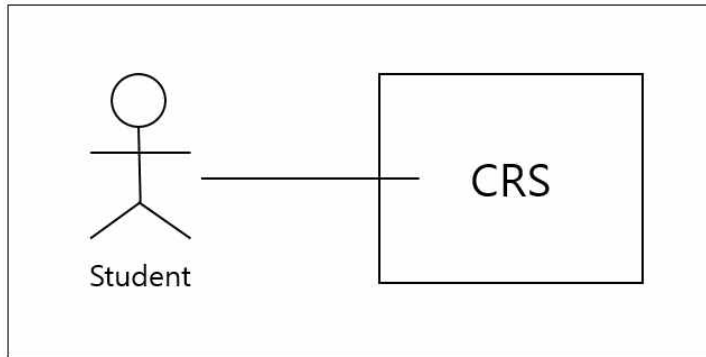
교과목 이수 계획서를 작성한 학생은 매 학기가 시작되기 전 수강신청 기간에 이번 학기에 개설될 강좌(Section) 개설 목록을 확인하고, 본인이 수강하고자 하는 교과목을 신청한다. 만약, 어떤 교과목이 서로 다른 교수(Professor)들에 의해 여러 강좌로 개설된 경우 본인이 선호하는 요일, 시간, 교수 등을 고려하여 강좌를 선택한다. CRS 시스템은 이 학생의 성적증명서(Transcript)를 참조하여 선수과목을 이수했는지 검사한다. 학생 및 해당 학생의 지도교수는 언제라도 해당 학생의 성적증명서를 확인할 수 있다.

만약 (1) 학생이 선수과목을 이수하였고 (2) 이 교과목이 학생의 이수 계획서에 있으며 (3) 강좌에 남는 좌석이 있는 경우 학생을 수강자로 강좌에 등록한다.

(1), (2) 조건은 만족하지만, 강좌에 남는 좌석이 없는 경우 학생을 해당 강좌의 대기 목록에 추가한다. 만약 학생이 대기하고 있던 강좌에 남는 좌석이 생기면(다른 학생이 수강을 취소하거나 강의실 이전으로 좌석 수가 늘어나는 경우) 이 학생은 자동으로 해당 강좌에 수강생이 되고, 이 결과를 학생에게 메일로 통보한다. 강좌를 취소하는 것은 학생의 권한이며, 만약 취소하지 않으면 해당 강좌 수강 비용이 등록금에 합산된다.

B. 개념 확인

1. 개발하는 수강신청 시스템 CRS를 사용하여 어떤 목적을 달성하려는 사람이나 외부 시스템을 액터(Actor)라고 한다. CRS의 액터 중 하나는 분명히 학생이다. 시스템의 액터를 UML 다이어그램으로 표시할 때 시스템은 사각형으로 액터는 사람 모양의 막대 그림으로 표시한다. 아래는 학생 액터만 표시한 다이어그램 예이다. 위의 명세서를 자세히 읽어서 모든 액터를 판별하여 목록을 작성하시오. 이들 액터를 사용하여 UML 다이어그램을 작성하시오.



액터 목록

Student
...

UML 다이어그램

2. 액터가 시스템을 사용하여 달성하는 목적을 기술한 것을 유스케이스(Use Case)라고 한다. 유스케이스의 예를 들면 학생이 시스템에 접속하여 어떤 강좌를 수강 신청하는 것이다. 이러한 유스케이스는 시스템이 외부에 제공해야 할 기능에 의미한다. CRS의 유스케이스를 최대한 많이 나열하시오.

유스케이스
어떤 강좌에 등록한다
어떤 수강신청을 취소한다

3. 1번 문제에서 만든 액터 목록과 2번 문제에서 만든 유스케이스 목록을 이용하여 다음과 같은 유스케이스-액터 참조표를 작성하시오. 표의 행은 유스케이스이고 열은 액터이며 각 셀의 내용은 해당 액터가 이 유스케이스에 대해 정보를 사용하는지 아니면 반대로 정보를 제공하는지를 표시함

	Student		
강좌 수강신청	정보 제공		
...			

4. 문제 1~3은 거쳐서 개발하려는 시스템에 대한 요구 사항을 명확하게 정의하였다. 다음으로는 시스템 개발을 위해 어떤 클래스가 필요한지를 파악하고 각 클래스의 필드와 메소드를 정의해야 한다. 먼저 클래스 목록을 만들기 위해 요구명세서 및 유스케이스 목록 문서들을 분석한다. 일반적으로 클래스는 명사 형태로 표현되기 때문에 이들 문서에 나오는 모든 명사를 분석할 필요가 있다. 주어진 요구명세서 및 유스케이스 목록에서 나오는 모든 명사, 명사구들을 수집하시오.

수강신청시스템
CRS
개발의뢰
...

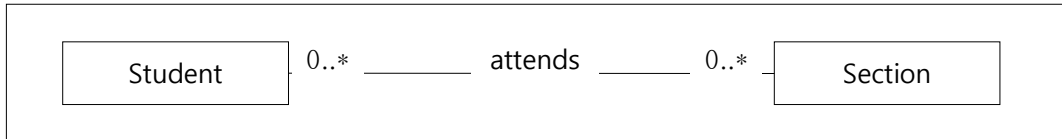
5. 문제 4에서 만든 명사, 명사구 목록을 정렬하고, 같은 의미를 나타내는 다른 용어를 하나로 통일 하시오. 이후 클래스 이름으로 사용할 명사, 명사구를 선정하시오. 어떤 명사, 명사구를 클래스로 표현해야 하는가를 결정할 때 이 명사, 명사구에 대해 데이터 필드가 있는지 또 메소드가 있는지를 생각해서 결정하시오.

CRD
강좌(분반)
교과목
...

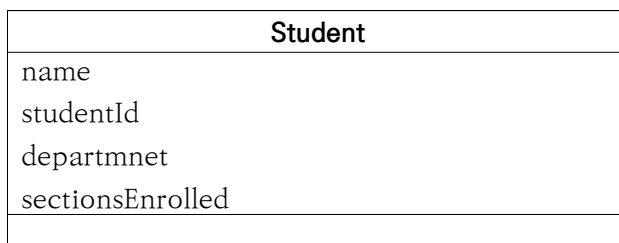
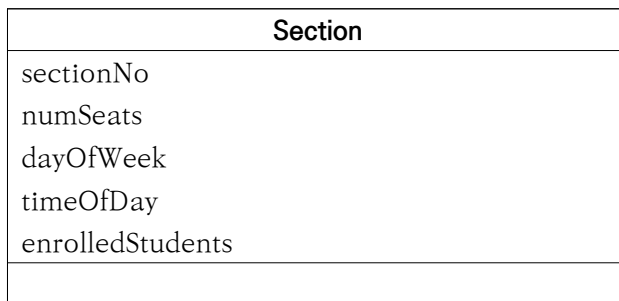
6. 요구명세서에 "학기"라는 명사가 나오는데 이 "학기"를 클래스로 표현하는 것이 좋은지 여부에 대해 본인의 의견을 말하고, 그렇게 주장하는 근거를 설명하시오.
7. 다시 명사구 분석을 하여 문제 5에서 만든 클래스의 자료 필드를 선정하고 각 클래스에 대해 UML 클래스 다이어그램을 그리시오. 예를 들면 강좌(Section) 필드의 경우 좌석 수, 강의 요일, 강의 시간 등이 자료 필드로 나온다는 것을 파악할 수 있고 다음과 같이 클래스 다이어그램을 작성할 수 있다. 각 클래스에 어떤 자료 필드가 필요한지 판단할 때 도메인 지식도 함께 활용해야 한다. 예를 들면, 거의 모든 대학에서 학생들에게 학번을 부여한다든지 이메일, 주소, 휴대전화번호 등을 관리하고 있다는 것을 알 수 있다.

Section
sectionNo
numSeats
dayOfWeek
timeOfDay

8. 클래스 사이의 관계는 크게 연관(Association), 집합(Aggregation), 상속(Inheritance) 관계로 나눌 수 있다. 상속은 다음에 배울 내용이기 때문에 여기서는 고려하지 말고 문제 5에서 만든 클래스들 사이의 관계를 파악하여 클래스 다이어그램을 작성하시오. 예를 들면 학생(Student) 객체는 강좌(Section) 객체를 수강(attend)한다는 관계 있다. 한 학생은 여러 개의 강좌를 수강할 수 있고 한 강좌를 여러 학생이 수강하기 때문에 이들 객체 사이에 다대다 관계가 존재한다. 이를 UML 다이어그램으로 표시하면 다음과 같이 표시된다.



9. 문제 8에서 파악한 클래스 객체들 사이의 관계를 클래스 설계에 표현하여야 한다. 대부분의 관계는 양방향 관계이기 때문에 관계에 참여하는 두 클래스에 각각 상대방 클래스 객체에 대한 참고가 필요하다. 만약 관계가 다대다 관계일 경우 상대방 객체 집합이 각각의 클래스에 존재해야 한다. 즉 문제 8에서 제시한 클래스 다이어그램을 보면 Student 클래스에는 Section의 집합이 있어야 하고, Section 클래스에는 Student 집합이 있어야 한다. 이를 클래스 다이어그램에 표현하면 아래와 같이 표현할 수 있다. 문제 8에서 파악한 클래스 사이의 관계를 모든 클래스 설계에 반영하도록 UML 클래스 다이어그램을 수정하시오.

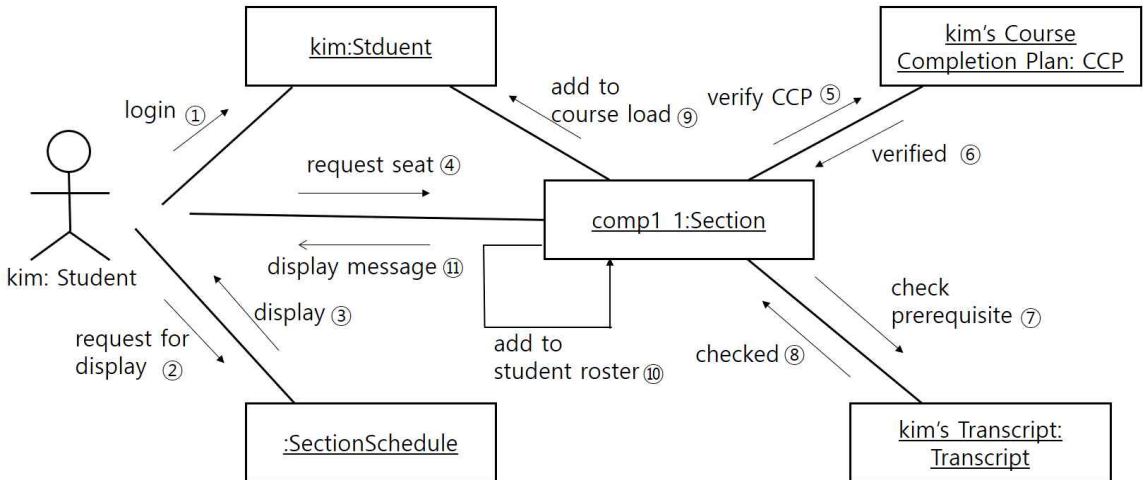


10. 다음으로 각 클래스에 필요한 메소드를 파악하는 문제를 고려해보자. 이를 위해서 시나리오라는 개념을 사용한다. 시나리오는 시스템이 제공하는 어떤 기능을 달성하기 위해서 내부적으로 어떤 객체들이 어떤 순서로 상호 작용하는가를 서술 방식으로 기술한 것을 말한다. 예를 들면 어떤 학생 kim이 어떤 강좌 comp1-1에 성공적으로 수강신청하는 과정을 나타내는 시나리오는 아래와 같다.

1. kim이라는 학생이 CRS에 로그인한다.
2. kim 학생이 이번 학기 개설 강좌 목록을 확인한다.
3. kim 학생이 comp1-1이라는 강좌를 수강신청한다.
4. comp1-1 강좌에서 kim이라는 학생의 수업 이수 계획서에 이 과목이 있는지 확인한다.
5. comp1-1 강좌에서 kim이라는 학생의 성적표를 확인하여 이 학생이 선수과목을 수강했는지 검사한다
6. 위의 과정의 모두 성공적으로 끝났다고 가정했을 때 남는 좌석이 있는지 확인한다.
7. 남는 좌석이 있어서 이 학생을 수강학생 목록에 추가한다

만약 위의 과정 중 4, 5에서 실패하면 이 과정은 수강신청에 실패하는 시나리오가 된다. 수강신청을 취소하는 시나리오 등 가능한 많은 수의 시나리오를 작성해 보시오.

11. 문제 10에서 예를 든 시나리오를 객체 간 통신 다이어그램으로 표시하면 아래 그림과 같이 표시할 수 있다. 여러분이 만든 시나리오에 대해 통신 다이어그램을 작성하시오. 통신 다이어그램에서 사람 모양의 막대 그림은 액터를 표시하며 사각형은 객체를 표시한다. 객체를 표시할 때 클래스를 같이 표현해 준다.



이 그림을 보면 강좌 객체가 학생 객체에서 자신을 수강 과목 목록에 추가해 달라고 요청하고 있다는 것을 알 수 있다. 따라서 Student 클래스에서는 addToCourseLoad라는 메소드가 필요하다는 것을 알 수 있다. 이 메소드는 그 결과를 돌려주지 않는 것으로 되어 있어서 반환형을 void로 지정할 수 있다. 비슷하게 강좌 이수 계획서(CCP: Course Completion Plan) 클래스에도 verifyCCP라는 메소드가 필요하고 이 메소드는 논리값을 반환하면 된다는 것을 알 수 있다. 문제 11에서 작성한 여러 가지 시나리오에 대해 통신 다이어그램을 그리고 클래스 다이어그램에 적절한 메소드를 추가하시오.

4장 상속

4.1 슈퍼 클래스와 서브 클래스

4.2 접근 제어

4.3 상속과 생성자

4.4 메소드 오버라이딩과 메소드 오버로딩

4.5 Object 클래스

4장 상속

4.1 수퍼 클래스와 서브 클래스

A. 개념 정리

상속(inheritance)은 기존 클래스를 확장하여 새로운 클래스를 정의하는 것을 말하며 소프트웨어 재사용의 한 형태이다. 이때, 기존 클래스와 새로운 클래스 사이에 상속 계층(inheritance hierarchy)을 형성하게 되며 상위 클래스를 수퍼 클래스(super class), 하위 클래스를 서브 클래스(sub class)라고 부른다. 상위 클래스의 모든 특성을 하위 클래스에서 실체화시키는 관계를 특수화(specialization)라고 하고, 반대로 하위 클래스의 공통적인 특성을 추상화하여 상위 클래스를 정의하는 것을 일반화(generalization)라고 한다. 서브 클래스는 수퍼 클래스의 모든 속성과 행위를 상속하며 자신의 속성과 행위를 추가할 수 있다. 또한 서브 클래스는 그 자신이 또 다른 클래스의 수퍼 클래스가 될 수 있다. Java의 클래스 상속계층에서 최상위 클래스는 Object이며 모든 Java 클래스는 Object를 직접 혹은 간접적으로 상속한다. 클래스를 정의할 때 Object 클래스의 상속을 명시적으로 기술하지 않아도 암묵적으로 상속된다. Java는 오직 단일 상속만 지원하여 수퍼 클래스와 서브 클래스 사이에 is-a 관계를 형성한다. is-a 관계에 있는 서브 클래스 객체는 수퍼 클래스 객체로 취급될 수 있다. 그림 4.1은 UniversityMember의 상속 계층을 나타내는 UML 클래스 다이어그램이다. 상속 계층에서 각 화살표는 is-a 관계를 나타내며, 가장 아래 클래스로부터 최상위 클래스까지 화살표를 따라 연속적으로 is-a 관계를 형성한다. Java에서 클래스 간 상속은 extends 라는 키워드를 사용하여 구현한다.

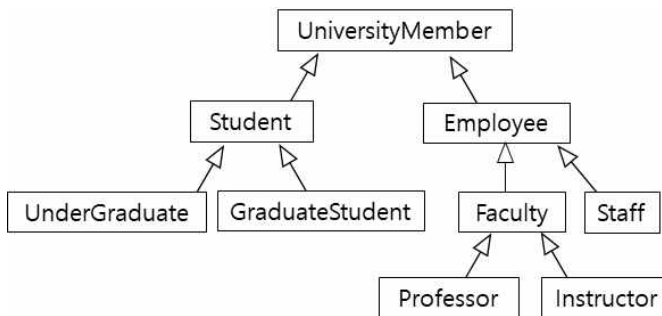


그림 4.1. 상속 계층

B. 개념 확인

1. 그림 4.1의 상속 계층에 대해 기술한 문장이 맞으면 O, 틀리면 X를 표시하시오.

- 1) Employee는 UniversityMember이다. ()
- 2) UniversityMember는 Employee이다. ()
- 3) UniversityMember는 Student와 Employee의 직접 수퍼 클래스(direct super class)이다. ()
- 4) UniversityMember는 Student와 Employee를 제외한 모든 클래스의 간접 수퍼 클래스(indirect super class)이다. ()
- 5) Professor는 Employee이다. ()
- 6) Instructor는 UniversityMember이다. ()
- 7) GraduateStudent는 Employee이다. ()

2. 다음은 Student 클래스를 정의하는 Java 코드이다. 밑줄을 채우시오.

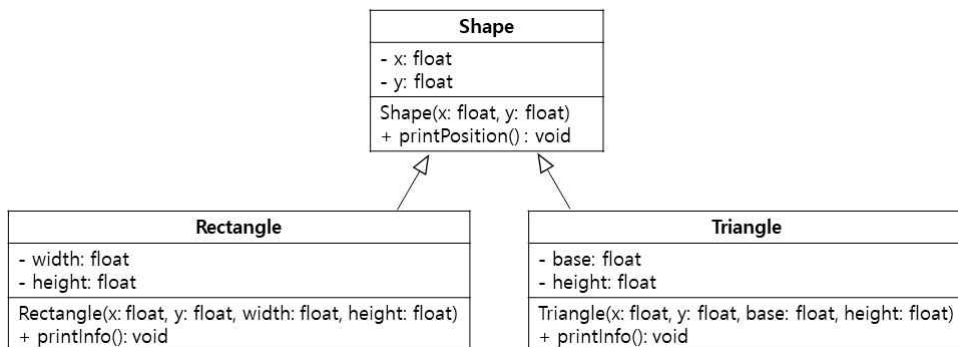
```
class Student e _____ UniversityMember { }
```

3. 다음은 UniversityMember 클래스를 정의하는 Java 코드이다. 밑줄을 채우시오.

```
class UniversityMember extends _____ { }
```

C. 응용 문제

1. 다음 UML 다이어그램으로 주어진 클래스들을 Java로 정의한 후 Rectangle, Triangle 객체를 각각 생성하여 테스트하는 프로그램을 작성하시오.



√ 세부 사항

- 드라이버 클래스인 ShapeTest 클래스를 추가로 정의하여 사용한다.
- Rectangle, Triangle 클래스의 생성자에서 첫 줄에 super(x, y)를 호출한다.
- printPosition 메소드에서 위치를 출력하고, printInfo 메소드에서 필드값 및 면적을 출력한다.

√ 실행 결과

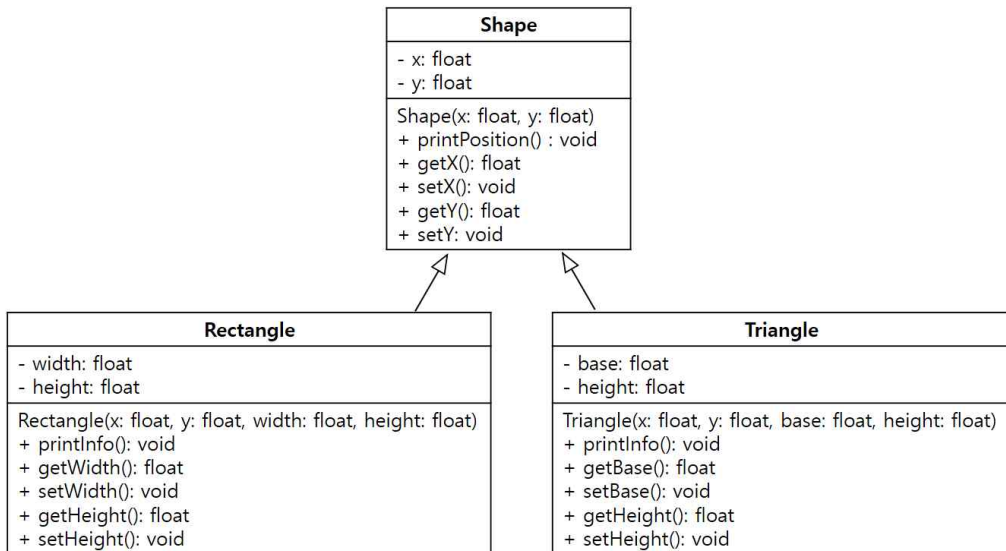
```

Rectangle
position: (0.00, 0.00)
width: 100.00, height: 200.00, area: 20000.00

Triangle
position: (100.00, 100.00)
base: 100.00, height: 200.00, area: 10000.00

```

2. 다음 UML 클래스 다이어그램은 1번 문제의 Shape, Rectangle, Triangle 클래스에 대해 필드의 게터(getter), 세터(setter)를 추가한 것이다. 게터를 사용하여 1번과 동일한 실행 결과를 얻도록 하는 프로그램으로 수정하시오.



√ 세부 사항

- 게터와 세터를 직접 작성해도 되고, 각자가 사용하는 개발환경에서 자동으로 게터와 세터를 추가하는 메뉴를 사용해도 된다. 이클립스의 경우, Package Explorer에서 클래스를 선택하여 마우스 우측 버튼을 눌러 [Source → Generate Getters and Setters...] 메뉴를 사용한다.

-
- Shape 클래스의 printPosition 메소드에서 필드 x, y의 게터를 호출하도록 수정한다.
 - Rectangle 클래스의 printInfo 메소드에서 필드 width, height의 게터를 호출하도록 수정한다.
 - Triangle 클래스의 printInfo 메소드에서 필드 base, height의 게터를 호출하도록 수정한다.

4.2 접근 제어

A. 개념 정리

Java 클래스 멤버인 필드와 메소드에 대해 접근 지정자(access modifier)를 사용하여 접근 제어를 할 수 있다. 접근 제어는 클래스 수준과 멤버 수준으로 구분하여 이루어진다. 클래스 수준에서는 public이나 default 접근성을 가질 수 있으며, 멤버 수준에서는 public, protected, default 혹은 private 접근성을 가질 수 있다. 이러한 접근제어를 위해 접근 지정자로 public, protected, private을 사용하는데 default 접근을 위한 접근 지정자는 따로 존재하지 않는다. 클래스나 멤버에 접근 지정자를 생략하면 그 클래스나 멤버는 default 접근성을 가지게 되는데, default 접근성은 패키지 접근성을 의미한다. 다음 표 4.1은 public 클래스의 멤버에 대한 접근성을, 표 4.2는 default 클래스의 멤버에 대한 접근성을 요약한 것이다.

To From		public 클래스			
		public	protected	default	private
같은 패키지	같은 클래스	○	○	○	○
	서브 클래스	○	○	○	×
	다른 클래스	○	○	○	×
다른 패키지	서브 클래스	○	○	×	×
	다른 클래스	○	×	×	×

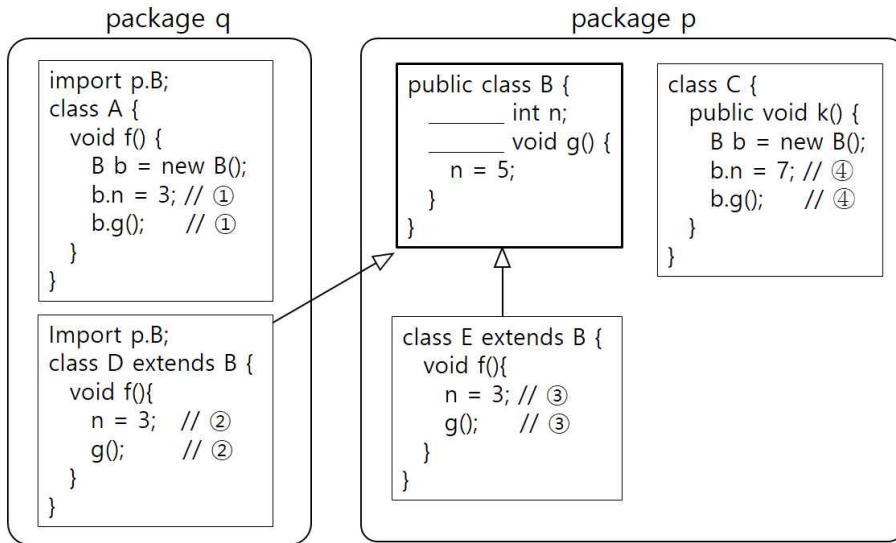
표 4.1 public 클래스의 멤버에 대한 접근성

To From		default 클래스			
		public	protected	default	private
같은 패키지	같은 클래스	○	○	○	○
	서브 클래스	○	○	○	×
	다른 클래스	○	○	○	×
다른 패키지	서브 클래스	×	×	×	×
	다른 클래스	×	×	×	×

표 4.2. default 클래스의 멤버에 대한 접근성

B. 개념 확인

1. 다음은 public 클래스의 멤버에 대한 접근성을 테스트 하는 코드 구조이다. 주어진 여러 상황에서 B클래스의 두 멤버에 대해 적절한 접근 지정자를 지정하시오. 단, 그림에서 패키지 선언문은 생략되었다.

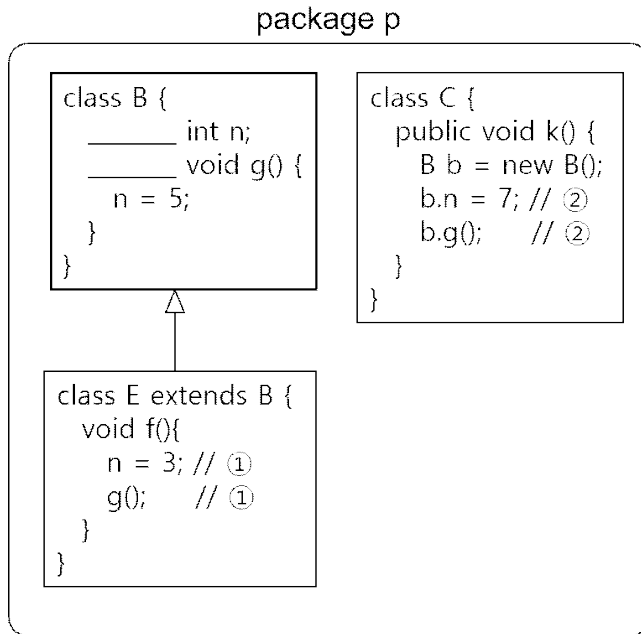


- 1) 에러가 생기지 않도록 하려면?
 - 2) ① 문장에만 에러가 생기도록 하려면?
 - 3) ①, ② 문장에서 에러가 생기도록 하려면?
 - 4) ①, ②, ③, ④ 문장에서 에러가 생기도록 하려면?
2. 1번의 밑줄에 protected가 지정된 경우 클래스 D가 다음과 같이 정의될 수 있는가?

```

class D extends B {
    void f() {
        B b = new B();
        b.n = 3;
        b.g();
    }
}
  
```

3. 다음은 default 클래스의 멤버에 대한 접근성을 테스트 하는 코드 구조이다. 주어진 여러 상황에서 B클래스의 두 멤버에 대해 적절한 접근 지정자를 지정하시오. 단, 그림에서 패키지 선언문은 생략되었다.



- 1) 에러가 생기지 않도록 하는 가능한 모든 접근 지정자는 무엇인가?
 - 2) ①, ② 문장에서 에러가 생기도록 하려면?
4. 3번의 밑줄에 protected가 지정된 경우 q 패키지의 클래스 D가 다음과 같이 정의될 수 있는가?

```

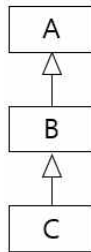
class D extends B {
    void f() {
        n = 3;
        g();
    }
}

```

4.3 상속과 생성자

A. 개념 정리

클래스의 생성자를 정의하지 않고 객체를 생성하는 경우에도 암묵적으로 컴파일러가 제공하는 기본 생성자(default constructor)가 실행된다. 기본 생성자는 매개변수를 가지지 않으며 프로그래머가 정의하는 매개변수가 없는 생성자도 기본 생성자로 인식될 수 있다. 클래스에 명시적으로 생성자를 하나라도 추가하면 기본 생성자는 자동으로 추가되지 않는다. 따라서 매개변수가 있는 생성자가 정의되는 경우 매개 변수가 없는 생성자를 직접 추가하는 것을 권장한다. 서브 클래스 객체를 생성하게 되면 서브 클래스의 생성자로부터 시작해서 슈퍼 클래스 생성자로 향하는 생성자의 연쇄 호출이 일어나게 된다. 서브 클래스의 생성자는 자신의 몸체를 실행하기 전에 바로 위의 슈퍼 클래스의 생성자를 호출한다. 따라서 다음 그림과 같은 상속계층에서 C 클래스의 객체를 생성하게 되면 $C \rightarrow B \rightarrow A$ 순서로 연쇄적인 생성자 호출이 일어난 후 $A \rightarrow B \rightarrow C$ 순서로 각 생성자의 몸체가 실행되게 된다. 이러한 생성자 연쇄 호출에서 항상 Object 클래스의 생성자가 마지막에 호출되며, 최초로 호출된 서브클래스 생성자의 몸체는 마지막에 실행을 마친다. 각각의 생성자는 대체로 자신이 속한 클래스에 추가된 인스턴스 변수만을 초기화한다.



B. 개념 확인

1. 다음 프로그램에서 드라이버 클래스의 main 메소드가 제대로 실행되도록 클래스 A, B를 수정하십시오.

```

public class Test1 {
    public static void main(String[] args) {
        A a2 = new A(1, 2);
        B b1 = new B();
        B b2 = new B(1, 2, 3);
    }
}
  
```



```

    }
}

class A {
    int a;
    int b;
    public A(int a, int b) {
        this.a = a;
        this.b = b;
        System.out.println("public A(int a, int b)");
    }
}

class B extends A {
    int c;
    public B(int a, int b, int c) {
        super(a, b);
        this.c = c;
        System.out.println("public B(int a, int b, int c)");
    }
}

```

2. 다음은 연쇄적인 생성자 호출 개념을 이해하기 위한 코드이다. 클래스 필드의 선언 및 값 할당 코드는 생략하였으며 모든 클래스는 하나의 파일에 선언되어 있다. 각 문제의 물음에 답하되, 예러가 생기는 경우 그 이유를 설명하시오.

```

public class Test2 {
    public static void main(String[] args) {
        B b = new B();
        C c1 = new C();
        C c2 = new C(1, 2);
    }
}

class A {
    // ①
    public A() {
        System.out.printf("%2s", "1");
    }
    // ②
    public A(int a, int b) {
        System.out.printf("%2s", "2");
    }
}

```

```
class B extends A {
    public B() {
        this(1, 3); // ③
        System.out.printf("%2s", "3");
    }
    public B(int a, int b) {
        System.out.printf("%2s", "4");
    }
}

class C extends B {
    public C() {
        super(1, 3);
        System.out.printf("%2s", "5");
    }
    public C(int a, int b) {
        System.out.printf("%2s", "6");
    }
}
```

- 1) 전체 소스의 실행 결과는?
- 2) ① 생성자가 없다면 실행 결과는?
- 3) ② 생성자가 없다면 실행 결과는?
- 4) ③ 문장이 없다면 실행 결과는?

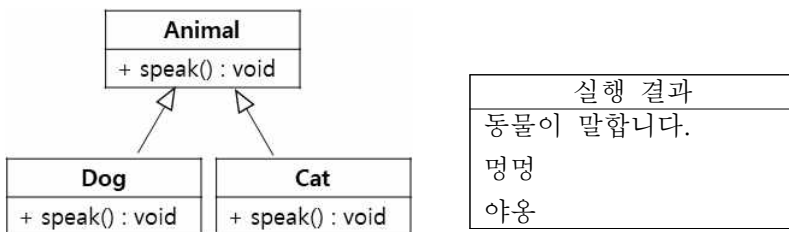
4.4 메소드 오버라이딩과 메소드 오버로딩

A. 개념 정리

메소드 오버라이딩(method overriding)은 슈퍼 클래스에서 정의된 메소드를 서브 클래스에서 재정의하는 것을 말한다. 서브 클래스에서 재정의되는 메소드는 슈퍼 클래스의 메소드와 동일한 원형으로 작성해야 한다. 즉, 메소드의 이름, 매개변수 개수, 매개변수 타입, 반환형이 동일해야 한다. 메소드 오버라이딩 시 슈퍼 클래스의 가시성(visibility)은 서브 클래스에서 줄어들 수 없다. 예를 들어, 슈퍼 클래스의 public 메소드가 서브 클래스에서 default나 private 메소드로 재정의될 수는 없다. 컴파일 시 메소드 오버라이딩을 제대로 구현했는지 체크하기를 원한다면 서브클래스의 재정의 메소드 상단에 @Override 어노테이션(annotation)을 추가한다. 만약 슈퍼 클래스에 그런 이름의 메소드가 없다면 컴파일러가 에러를 발생시킬 것이다. 만약, 슈퍼 클래스의 메소드와 이름이 같고 매개변수 개수나 매개변수 타입이 다르다면 메소드 오버로딩(method overloading)으로 처리되므로 주의해야 한다. 메소드의 반환형은 메소드 오버로딩의 제약사항에 포함되지 않는다.

B. 개념 확인

1. 메소드 오버라이딩의 개념을 이해하기 위한 문제이다. 다음과 같이 주어진 상속 계층 하에 제시된 실행 결과를 보이도록 코드의 빈 곳을 채우시오.



```

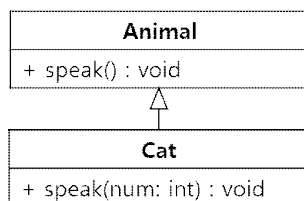
class Animal {
    public void speak() {
        System.out.println("동물이 말합니다.");
    }
}
class Dog extends Animal {
    @Override
    [ ]
}
class Cat extends Animal {
    @Override
    [ ]
}
public class AnimalTest {
    public static void main(String str[]) {
        Animal animal = new Animal();
        animal.speak();

        Dog dog = new Dog();
        dog.speak();

        Cat cat = new Cat();
        cat.speak();
    }
}

```

2. 메소드 오버로딩을 이해하기 위한 문제이다. 다음과 같이 주어진 상속 계층 하에 제시된 실행 결과를 보이도록 코드의 빈 곳을 채우시오. 단, Cat 클래스의 speak 메소드로 전달되는 정수 값에 따라 "야옹"이 반복되어 출력된다.



실행 결과
동물이 말합니다.
야옹야옹

```

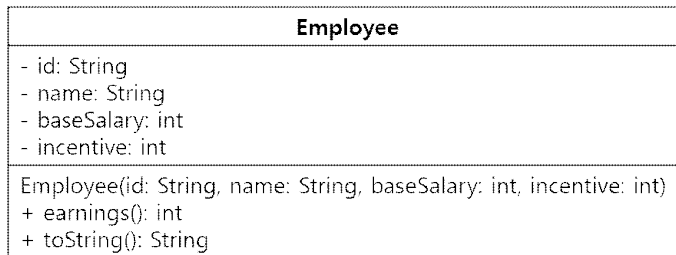
class Animal {
    public void speak() {
        System.out.println("동물이 말합니다.");
    }
}
class Cat extends Animal {
    
}

public class AnimalTest {
    public static void main(String str[]) {
        Cat cat = new Cat();
        cat.speak();
        cat.speak("야옹");
    }
}

```

C. 응용 문제

1. 주어진 UML 클래스 다이어그램과 세부사항을 참고하여 직원 2명의 기본정보 및 급여를 출력하는 프로그램을 작성하시오.



√ 세부 사항

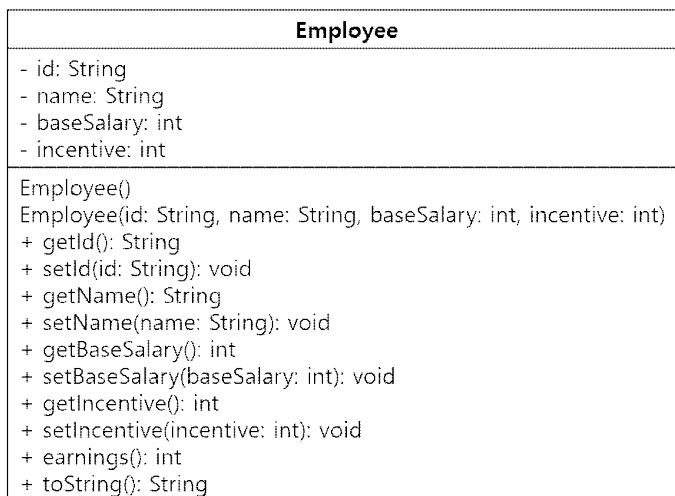
- 생성자: 각 매개변수를 대응되는 필드의 값으로 저장한다.
- earnings() : 급여를 계산하는 메소드로 baseSalary와 incentive의 합을 반환한다.
- toString() : Object 클래스의 toString() 메소드를 오버라이딩하는 메소드로 직원의 id, name, baseSalary, incentive 및 급여를 포함하는 하나의 String을 반환한다. 급여는 earnings()를 호출하여 반환받는다. String 간의 더하기 혹은 String.format() 메소드를 사용하여 하나의 문자열을 생성한다.

- 드라이버 클래스에서 Employee 객체를 생성하여 직원정보를 출력한다. System.out.println() 혹은 System.out.printf()에 객체를 전달하여 toString() 메소드가 자동으로 호출되게 하거나 명시적으로 toString()을 호출할 수 있다.

√ 실행 결과

```
id: 2023001, name: HongGilDong, base salary: 2000000, incentive: 0, earnings: 2000000
id: 2023002, name: SongSoHee, base salary: 2000000, incentive: 500000, earnings: 2500000
```

2. 주어진 UML 클래스 다이어그램과 세부사항을 참고하여 직원 2명의 기본 정보 및 급여를 출력하는 프로그램을 작성하시오.



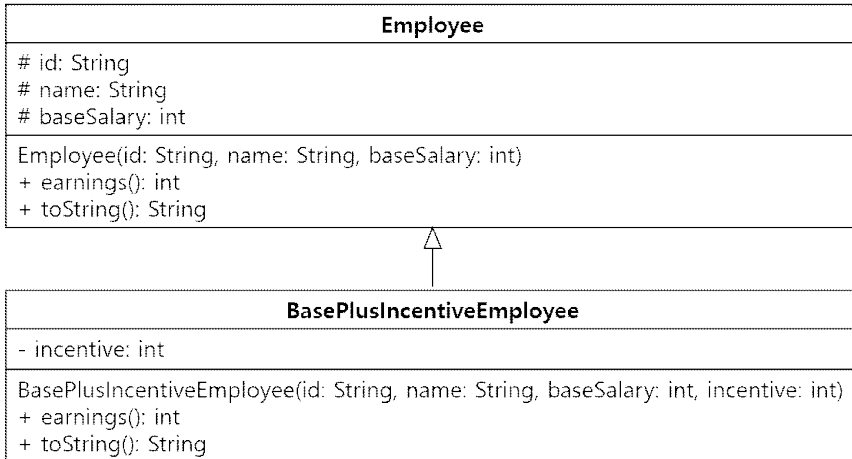
√ 세부 사항

- 1번의 Employee에 대해 매개변수 없는 생성자와 각 필드의 게터와 세터를 추가한다.
- earnings() : 급여를 계산하는 메소드로 baseSalary와 incentive의 게터의 반환값을 더해서 반환한다.
- toString() : Object 클래스의 toString() 메소드를 오버라이딩하는 메소드로 직원의 id, name, baseSalary, incentive 및 급여를 포함하는 하나의 String을 반환한다. 단, 각 필드의 게터와 earnings()를 호출하여 직원 정보를 포함하는 하나의 문자열을 생성한다.
- 첫 번째 Employee 객체는 매개변수가 있는 생성자를 사용하여 생성한다.
- 두 번째 Employee 객체는 매개변수가 없는 생성자를 사용하여 생성한 후 세터로 필드값을 지정한다.

√ 실행 결과

```
id: 2023001, name: HongGilDong, base salary: 2000000, incentive: 0, earnings: 2000000
id: 2023002, name: SongSoHee, base salary: 2000000, incentive: 500000, earnings: 2500000
```

3. 주어진 UML 클래스 다이어그램과 세부사항을 참고하여 직원 2명의 기본정보 및 급여를 출력하는 프로그램을 작성하시오.



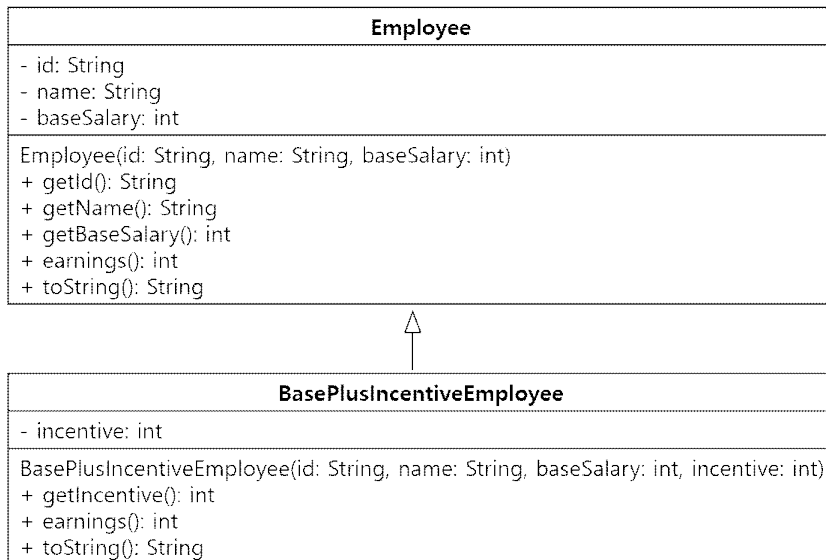
√ 세부 사항

- Employee의 earnings() : 급여를 계산하는 메소드로 baseSalary를 반환한다.
- Employee의 toString() : 직원의 id, name, baseSalary 정보를 포함하는 String을 반환한다.
- BasePlusIncentiveEmployee의 생성자 : 매개변수 중 id, name, baseSalary는 Employee 클래스로 전달하고 incentive는 BasePlusIncentiveEmployee 클래스의 필드값으로 사용한다.
- BasePlusIncentiveEmployee의 earnings() : baseSalary와 incentive의 합을 반환한다.
- BasePlusIncentiveEmployee의 toString() : 직원의 id, name, baseSalary, incentive 정보를 포함하는 하나의 String을 반환한다.
- 드라이버 클래스에서 Employee 객체와 BasePlusIncentiveEmployee 객체를 하나씩 생성하여 직원 정보를 출력한다. 직원의 급여정보는 객체 참조변수를 사용하여 earnings()를 직접 호출하여 출력한다.

√ 실행 결과

```
id: 2023001, name: HongGilDong, base salary: 2000000, earnings: 2000000
id: 2023002, name: SongSoHee, base salary: 2000000, incentive: 500000, earnings: 2500000
```

4. 주어진 UML 클래스 다이어그램과 세부사항을 참고하여 직원 2명의 기본정보 및 급여를 출력하는 프로그램을 작성하시오.



√ 세부 사항

- 3번 Employee 클래스의 각 필드에 대해 접근 지정자를 private으로 수정한다.
- 3번의 Employee와 BasePlusIncentiveEmployee에 대해 각 필드의 게터를 추가한다.
- Employee의 earnings() : 급여를 계산하는 메소드로 baseSalary의 게터의 반환값을 반환한다.
- Employee의 toString() ; 직원의 id, name, baseSalary를 포함하는 하나의 String을 반환한다. 단, 각 필드의 게터를 호출하여 하나의 문자열을 생성한다.
- BasePlusIncentiveEmployee의 earnings() : Employee의 earnings()와 incentive의 게터의 반환값을 더하여 반환한다.
- BasePlusIncentiveEmployee의 toString() : 직원의 id, name, baseSalary, incentive 정보를 포함하는 하나의 String을 반환한다. 단, 필드를 직접 참조하지 않고 Employee의 toString()과 incentive의 게터를 사용하여 문자열을 생성한다.
- 드라이버 클래스에서 Employee 객체와 BasePlusIncentiveEmployee 객체를 하나씩 생성하여 직원 정보를 출력한다. 직원의 급여정보는 객체 참조변수를 사용하여 earnings()를 직접 호출하여 출력한다.

√ 실행 결과

```
id: 2023001, name: HongGilDong, base salary: 2000000, earnings: 2000000
```

```
id: 2023002, name: SongSoHee, base salary: 2000000, incentive: 500000, earnings: 2500000
```

4.5 Object 클래스

A. 개념 정리

Java의 모든 클래스의 최상위 클래스는 `java.lang` 패키지에 있는 `Object` 클래스이다. 모든 클래스는 `Object` 클래스로 간주되어 명시적으로 `Object`를 상속하지 않더라도 암묵적으로 `Object`의 모든 메소드를 상속받게 된다. `Object` 클래스의 메소드로 `equals`, `hashCode`, `toString`, `wait`, `notify`, `notifyAll`, `getClass`, `finalize`, `clone` 등이 있으며 필요한 경우 이들 메소드를 상황에 맞게 적절하게 오버라이딩해서 사용한다. 표 4.3은 `Object`의 메소드 중 일부를 정리한 것이다.

메소드	설명
<code>Class getClass()</code>	Java에서 모든 객체는 실행시간에 그 자신의 타입을 알게 된다. <code>getClass</code> 메소드는 실행 시간 객체의 클래스 정보를 담고 있는 <code>Class</code> 타입 객체를 반환한다. <code>Class</code> 의 <code>getName</code> 메소드를 통해 참조하고 있는 객체의 클래스 이름을 확인할 수 있다.
<code>boolean equals(Object obj)</code>	두 객체의 동등성을 비교하여 <code>true/false</code> 를 반환한다. 특정 클래스 타입의 두 객체를 비교할 때는 두 객체의 콘텐츠를 비교하기 위해 <code>equals</code> 메소드를 오버라이딩해야 한다. 디폴트 구현은 메모리상의 두 객체의 참조를 비교하기 위해 <code>==</code> 연산자를 사용한다.
<code>String toString()</code>	객체를 기술하는 문자열을 반환한다. 디폴트 구현은 객체의 클래스가 속한 패키지 이름, 클래스 이름, 그리고 객체의 <code>hashCode</code> 메소드가 반환하는 값의 16진수 표현을 포함한다.
<code>Objcet clone()</code>	객체 자신의 복사본을 생성하여 반환한다. 디폴트 구현은 얇은 복사(<code>shallow copy</code>)이며 객체 내 인스턴스 변수값이 같은 타입의 다른 객체로 복사되는 방식이다. 참조 타입에 대해서는 참조가 복사된다. 오버라이딩하여 깊은 복사(<code>deep copy</code>)를 하는 경우 참조 타입 인스턴스 변수에 대해 새로운 객체를 생성한다. <code>clone</code> 을 정확하게 구현하는 것은 어렵기 때문에 대신 객체 직렬화(<code>object serialization</code>)를 사용하기도 한다.

표 4.3. Object 클래스의 메소드

B. 개념 확인

1. 다음 두 클래스에 대해 물음에 답하시오.

```
class Employee { }
class HourlyEmployee extends Employee { }
```

1) Employee 객체를 생성한 후 그 클래스 이름을 출력하는 코드의 빈 곳을 채우시오.

```
Employee employee1 = new Employee();
System.out.printf("employee1:%s\n", _____ );
```

2) Employee 객체와 객체를 생성하여 두 객체의 클래스가 같은지 비교하여 그 결과를 출력하는 코드의 빈 곳을 채우시오.

```
Employee employee2 = new Employee();
HourlyEmployee employee3 = new HourlyEmployee();

if ( _____ )
    System.out.println("same class");
else
    System.out.println("not the same class ");
```

3) 두 문자열이 같은지 비교하여 그 결과를 출력하는 코드의 빈 곳을 채우시오.

```
String s1 = "Java Programming";
String s2 = "OOP";
if ( _____ ) // if (s1 == s2)
    System.out.println("same string");
else
    System.out.println("not the same string");
```

4) 다음 코드의 실행 결과가 Employee@5b2133b 형태로 나온다. 실행 결과의 의미는 무엇이며 실제로 호출되는 메소드는 무엇인가?

```
Employee e = new Employee();
System.out.println(e);
```

- 5) 4)번의 실행 결과가 "This is an Employee class without fields."와 같은 문자열이 출력되도록 Object 클래스의 메소드 중 하나를 오버라이딩하시오.

```
class Employee { // class Employee extends Object
```



```
}
```

2. 다음과 같이 Employee 클래스가 정의될 때 물음에 답하시오.

```
class Employee {
    String id;
    String name;
    public Employee(String id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

- 1) 다음은 두 개의 Employee 객체가 같은지 비교하여 그 결과를 출력하는 부분 코드이다. 실행 결과를 적고 그러한 결과가 나온 이유를 설명하시오.

```
Employee employee1 = new Employee("2023001", "HongGilDong");
Employee employee2 = new Employee("2023001", "HongGilDong");
if (employee1.equals(employee2))
    System.out.println("same employee");
else
    System.out.println("not the same employee ");
```

- 2) 1)번이 제대로 실행되도록 Employee 클래스의 equals 메소드를 getClass 메소드 사용 버전으로 오버라이딩하시오. Eclipse 개발 환경에서 자동 생성하는 equals 메소드 코드는 다음과 같다.

```
@Override
public boolean equals(Object obj) {
    if ( _____ ) // 객체 참조를 비교함
        return true;
    if (obj == null)
        return false;
    if ( _____ ) //클래스가 다른지 비교
```

```

    return false;
    Employee other = (Employee) obj;
    return id.equals(other.id) && name.equals(other.name); // 필드 비교 ver1
    // return Objects.equals(id,other.id) && Objects.equals(name, other.name);
    // 필드 비교 ver2-jdk 1.7 이상, java.util.Objects 클래스의 static 메소드인 equals 사용
}

```

3) 1)번이 제대로 실행되도록 Employee 클래스의 equals 메소드를 instanceof 연산자 사용 버전으로 오버라이딩하시오. Eclipse 개발 환경에서 자동 생성하는 equals 메소드 코드는 다음과 같다.

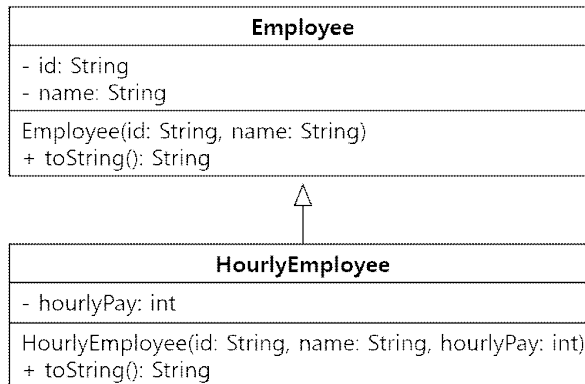
```

@Override
public boolean equals(Object obj) {
    if ( _____ ) // 객체 참조를 비교함
        return true;
    if ( _____ ) // obj가 Employee의 인스턴스인가 아닌 지 검사
        return false;
    Employee other = (Employee) obj; // 타입 캐스팅은 is-a 관계의 클래스 사이에서만 가능
    return Objects.equals(id, other.id) && Objects.equals(name, other.name);
}

```

C. 응용 문제

1. 다음 UML 클래스 다이어그램과 드라이버 클래스를 참고하여 toString() 메소드를 오버라이딩하여 테스트하는 프로그램을 작성하시오.



```

public class EmployeeTest7 {
    public static void main(String[] args) {
        Employee employee1 = new Employee("2023001", "HongGilDong");
        HourlyEmployee employee2 = new HourlyEmployee("2023002", "JungSoRa", 10000);

        // Employee의 toString

```

```

System.out.printf("%s\n", employee1);
System.out.printf("%s\n", employee1.toString());
System.out.println(employee1);
System.out.print(employee1 + "\n");

// HourlyEmployee의 toString
System.out.println(employee2);
}
}

```

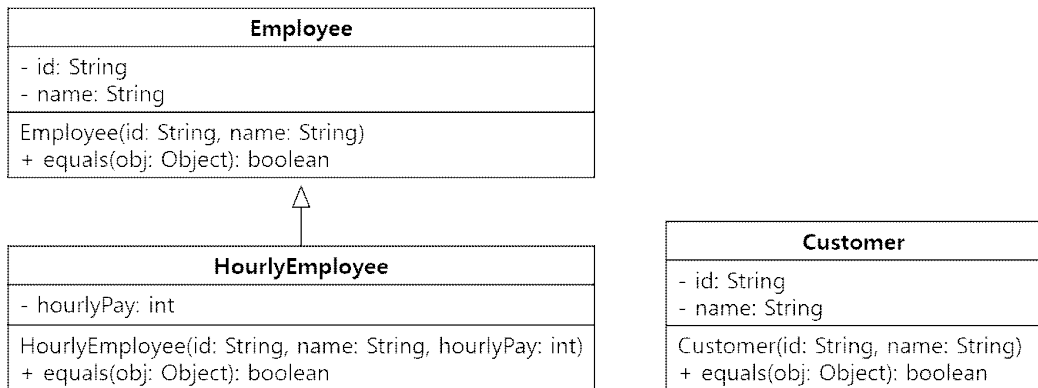
√ 실행 결과

```

Employee [id=2023001, name=HongGilDong]
Employee [id=2023001, name=HongGilDong]
Employee [id=2023001, name=HongGilDong]
Employee [id=2023001, name=HongGilDong]
HourlyEmployee [hourlyPay=10000, Employee [id=2023002, name=JungSoRa]]

```

2. 다음 UML 클래스 다이어그램과 드라이버 클래스를 참고하여 equals() 메소드를 오버라이딩하여 테스트하는 프로그램을 작성하시오.



```

public class EmployeeTest8 {
    public static void main(String[] args) {
        Employee employee1 = new Employee("2023001", "HongGilDong");
        HourlyEmployee employee2 = new HourlyEmployee("2023002", "KimKookJin", 10000);
        HourlyEmployee employee3 = new HourlyEmployee("2023002", "KimKookJin", 10000);
        Customer customer = new Customer("2023001", "HongGilDong");

        if (employee1.equals(employee2)) { ... }
        if (employee2.equals(employee1)) { ... }
        if (employee2.equals(employee3)) { ... }
        if (employee1.equals(customer)) { ... }
        if (customer.equals(employee1)) { ... }
    }
}

```

```
}  
}
```

√ 실행 결과

```
not the same employee  
not the same employee  
same employee  
different  
different
```

5장 다형성, 추상 클래스, 인터페이스

5.1 다형성

5.2 추상 클래스

5.3 인터페이스

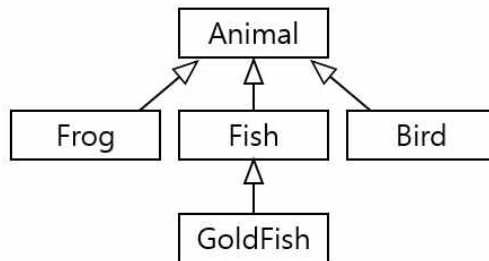
5.4 다형성 종합

5장 다형성, 추상 클래스, 인터페이스

5.1 다형성

A. 개념 정리

다형성(polymorphism)은 다양한 객체에 같은 메시지를 보냈을 때 다른 결과를 보이는 것을 말한다. 다형성은 구체적인 프로그램보다는 일반적인 프로그램을 작성할 수 있게 한다. 다형성을 사용하면 동일한 슈퍼 클래스를 공유하는 여러 서브 클래스 객체를 모두 슈퍼 클래스의 객체인 것처럼 처리하도록 단순화시켜 프로그램을 작성할 수 있다. 예를 들어, 다음 클래스 상속 계층에서 Frog, Fish, Bird, GoldFish 타입 객체는 모두 Animal 타입 객체로 처리될 수 있다.

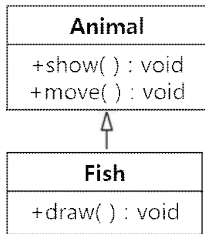


다형성을 구현한 Java 프로그램을 작성하기 위해서는 업캐스팅(up-casting)과 다운캐스팅(down-casting)을 이해할 필요가 있다. 업캐스팅은 아래에서 위로, 즉 서브 클래스에서 슈퍼 클래스 타입으로 캐스팅하는 것을 말하며 암묵적 형변환이 일어난다. 서브 클래스 객체에 대한 슈퍼 클래스 타입 참조 시 업캐스팅이 일어나며 기본적으로 슈퍼 클래스 멤버만 사용할 수 있다. 그러나 오버라이딩된 메소드의 경우, 실제 호출되는 메소드는 실행 시간에 다형적으로 결정되어 하위 클래스의 재정의된 메소드가 호출된다. 다운캐스팅은 위에서 아래로, 즉 슈퍼 클래스 타입에서 서브 클래스 타입으로 캐스팅하는 것을 말하며 명시적 타입 캐스팅이 필요하다. 업캐스팅이 일어난 경우 다운캐스팅을 통해 슈퍼 클래스에 없는 서브 클래스 메소드를 호출할 수 있다. 이때, 주의 사항은 반드시 업캐스팅된 것에 대해서만 다운캐스팅이 가능하다는 것이다. 참조하고 있는 객체가 슈퍼 클래스의 것인지 판단하기 위해, 즉 업캐스팅 여부를 알기 위해 instanceof 연산자를 사용하여 검사한 후 다운캐스팅할 수 있다.

B. 개념 확인

1. 다음은 다형성에 대해 단계적으로 이해하기 위한 문제들이다. 각 문제의 main 메소드 실행 시 업캐스팅이나 다운캐스팅이 일어났는지, 어떤 메소드가 호출되는지 혹은 예외가 발생되는지를 설명하시오. 단, 각 메소드는 슈퍼 클래스의 메소드를 호출하지 않는다는 가정 하에 답하시오.

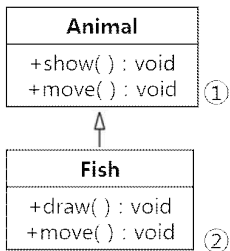
1) 다음 코드의 실행 결과는?



```

public class AnimalTest {
    public static void main(String[] args) {
        Fish fish = new Fish();
        fish.move();
    }
}
  
```

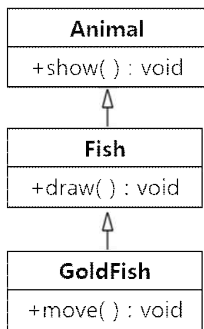
2) 다음 코드의 실행 결과는?



```

public class AnimalTest {
    public static void main(String[] args) {
        Fish fish = new Fish();
        fish.move();
    }
}
  
```

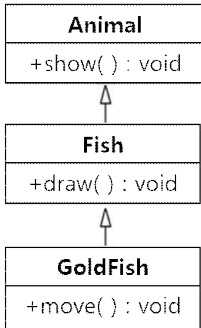
3) 다음 코드의 실행 결과는?



```

public class AnimalTest {
    public static void main(String[] args){
        Fish fish = new Fish();
        fish.move();
    }
}
  
```

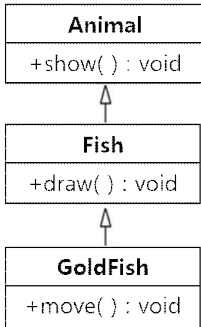
4) 다음 코드의 실행 결과는?



```

public class AnimalTest {
    public static void main(String[] args) {
        Fish fish = new Fish();
        ((GoldFish) fish).move();
    }
}
  
```

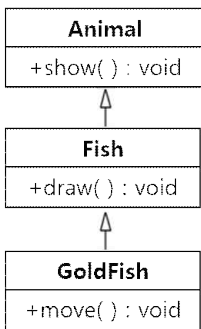
5) 다음 코드의 실행 결과는?



```

public class AnimalTest {
    public static void main(String[] args) {
        Fish fish = new Fish();
        ((GoldFish) fish).move();
    }
}
  
```

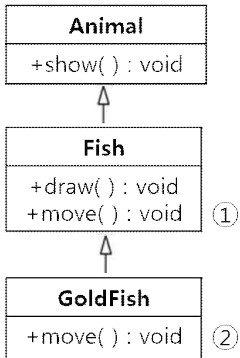
6) 다음 코드의 실행 결과는?



```

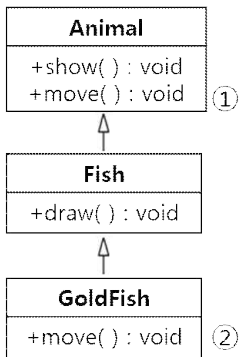
public class AnimalTest {
    public static void main(String[] args) {
        Fish fish = new GoldFish();
        if (fish instanceof GoldFish)
            ((GoldFish) fish).move();
    }
}
  
```

7) 다음 코드의 실행 결과는?



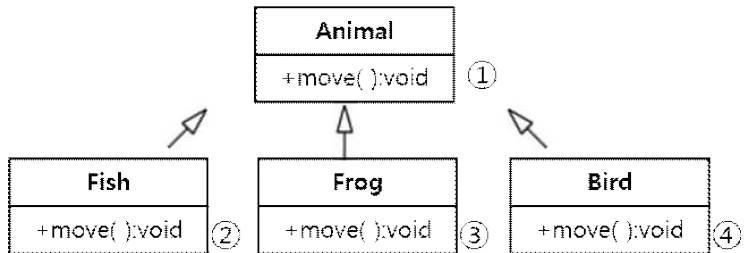
```
public class AnimalTest {
    public static void main(String[] args) {
        Fish fish = new GoldFish();
        fish.move();
    }
}
```

8) 다음 코드의 실행 결과는?



```
public class AnimalTest {
    public static void main(String[] args) {
        Fish fish = new GoldFish();
        fish.move();
    }
}
```

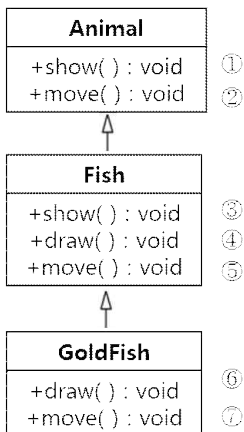
9) 다음 코드의 실행 결과는?



```

public class AnimalTest {
    public static void main(String[] args) {
        Animal[] animal = new Animal[3];
        animal[0] = new Fish();
        animal[1] = new Frog();
        animal[2] = new Bird();
        for (int i = 0; i < animal.length; i++) {
            animal[i].move();
        }
    }
}
  
```

10) 다음 코드의 실행 결과는?



```

public class AnimalTest {
    public static void main(String[] args) {
        GoldFish goldFish = new GoldFish();
        goldFish.show();
        Animal animal = new Fish();
        animal.draw();
        Animal animal2 = new Fish();
        animal2.show();
        animal2.move();
        Animal animal3 = new GoldFish();
        ((Fish) animal3).draw();
    }
}
  
```

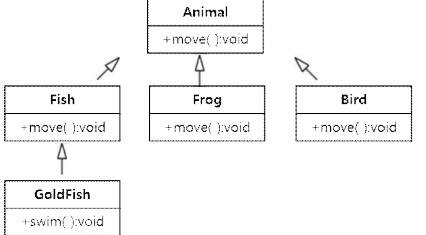
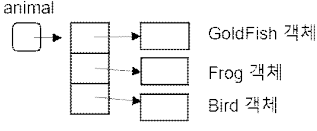
C. 응용 문제

1. 개념 확인 1번 문제를 직접 구현해 보면서 다형성에 대한 이해를 프로그램으로 확인하시오. 10개의 프로젝트로 구성하여 각 단계별 진행 상황을 잘 알아볼 수 있게 하여야 한다.

√ 세부 사항

- Animal의 move 메소드는 System.out.println("Animal : move()"); 형태의 출력문으로 구현한다.
- 다른 클래스의 메소드도 모두 System.out.println("클래스명 : 메소드명()"); 형태로 구현한다.
- main 메소드에서 다형적으로 호출되는 메소드 혹은 에러 발생 이유를 주석으로 처리한다.

2. 다음과 같은 조건하에서 드라이버 클래스 코드의 빈 곳을 채운 후 프로그램을 작성하시오.

① UML 클래스 다이어그램	② 객체 배열 구조	③ 호출순서
 <pre> classDiagram class Animal { +move() :void } class Fish { +move() :void } class Frog { +move() :void } class Bird { +move() :void } class GoldFish { +swim() :void } Animal < -- Fish Animal < -- Frog Animal < -- Bird Fish < -- GoldFish </pre>		<pre> Fish's move() GoldFish's swim() Frog's move() Bird's move() </pre>

④ 드라이버 클래스
<pre> public class AnimalTest { public static void main(String[] args) { // 객체 배열 생성 _____; _____; _____; _____; // enhanced for문 사용 // for문 헤더에서 선언된 파라미터를 사용하여 메소드를 호출하며 // 그 외 변수의 선언 및 사용 불가. 배열명 animal를 사용한 메소드 호출 불가 for (_____) { _____; if (_____) _____; } } } </pre>

√ 세부 사항

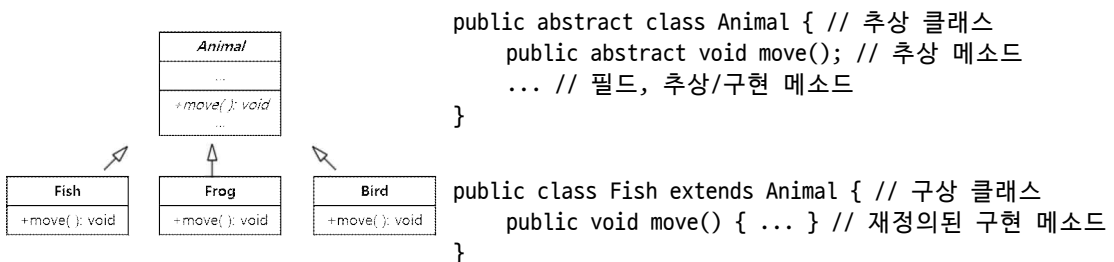
- Animal의 move 메소드는 System.out.println("Animal : move()"); 형태의 출력문으로 구현한다.
- 다른 클래스의 메소드도 모두 System.out.println("클래스명 : 메소드명()"); 형태로 구현한다.

5.2 추상 클래스

A. 개념 정리

클래스를 크게 구상 클래스(concrete class)와 추상 클래스(abstract class)로 구분할 수 있다. 구상 클래스는 선언하는 모든 메소드의 구현을 제공하며 객체를 인스턴스화하는데 사용할 수 있다. 메소드 정의 시 헤더 부분만 있고 몸체, 즉 구현 부분을 제공하지 않는 메소드를 추상 메소드(abstract method)라고 한다. 추상 클래스는 abstract 키워드를 사용하여 선언하는 클래스인데 추상 메소드를 포함하는 경우와 포함하지 않은 경우가 있다. 두 경우 모두 구현 메소드(non-abstract method 혹은 concrete method) 및 필드를 포함할 수 있으나 객체 인스턴스화에는 사용할 수 없다. 추상 클래스에서 생성자와 정적 메소드는 추상으로 선언할 수 없다. 추상 클래스는 서로 관련된 여러 클래스에서 공통된 개념을 추출해서 추상화하여 다른 클래스가 상속할 수 있는 공통된 설계를 제공한다. 추상 클래스를 상속하여 추상 클래스의 모든 추상 메소드를 구현한 서브 클래스는 구상 클래스가 되며, 일부 추상 메소드만 구현한 서브 클래스는 여전히 추상 클래스로 남게 된다.

다음은 추상 메소드 move를 멤버로 가진 추상 클래스 Animal과 이를 상속하여 구현한 Fish, Frog, Bird 클래스의 UML 클래스 다이어그램과 관련된 클래스 정의문이다. UML 다이어그램에서 추상 클래스와 추상 메소드는 이탤릭체로 나타낸다.



추상 클래스는 변수 선언 시 타입으로 사용할 수 있으며, 정적 메소드를 호출할 때도 사용할 수 있다.

B. 개념 확인

1. 다음 두 클래스 정의문에서 `CommissionEmployee` 클래스는 오류를 발생시킨다. `Employee` 클래스를 그대로 유지한 채 오류를 수정하기 위한 두 가지 방법을 설명하시오.

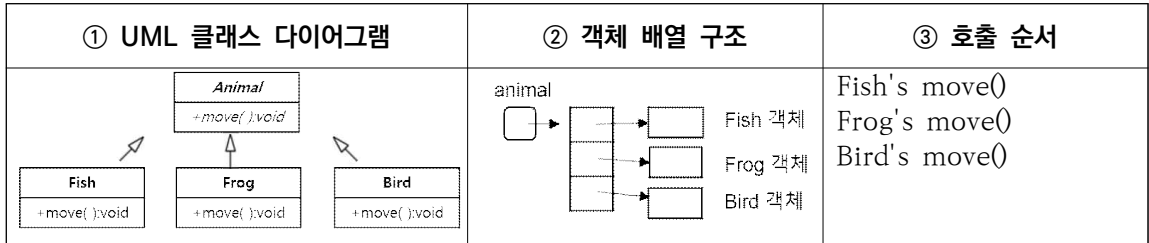
```
abstract class Employee {
    private String name;
    public String getName() {
        return name;
    }
    public abstract double earnings();
}

class CommissionEmployee extends Employee {
    private double grossSale;
    public double getGrossSale() {
        return grossSales;
    }
}
```

2. 다음은 추상 클래스와 관련된 기술이다. 틀린 것을 모두 고르시오.
- 추상 클래스는 하나 이상의 추상 메소드를 가지는 클래스이다.
 - 추상 메소드는 헤더만 있는 메소드로 구현, 즉 몸체 부분을 제공하지 않는다.
 - 추상 클래스로 객체를 생성할 수 없다.
 - 추상 클래스는 필드와 구현 메소드를 가질 수 없다.
 - 추상 클래스를 상속받은 서브 클래스에서 슈퍼 클래스의 추상메소드를 구현하는 것을 메소드 오버라이딩으로 볼 수 있다.
 - 추상 클래스를 상속받은 서브 클래스로 객체를 생성하려면 반드시 추상 클래스의 추상 메소드를 서브클래스에서 구현해야 한다.
 - 추상 클래스 타입은 변수 선언 시 타입으로 사용할 수 없고 정적 메소드 호출 시에는 사용 가능하다.
 - 추상 클래스는 주로 서로 관련 없는 여러 클래스에 대해 공통된 설계를 제공한다.

C. 응용 문제

1. 다음과 같은 조건하에서 각 클래스를 정의하고 드라이버 클래스 코드의 빈 곳을 채운 후 프로그램을 작성하시오.



④ 클래스 정의			
Animal	Fish	Frog	Bird

⑤ 드라이버 클래스
<pre> public class AnimalTest { public static void main(String[] args) { _____; _____; _____; _____; for (_____) { _____; } } } </pre>

5.3 인터페이스

A. 개념 정리

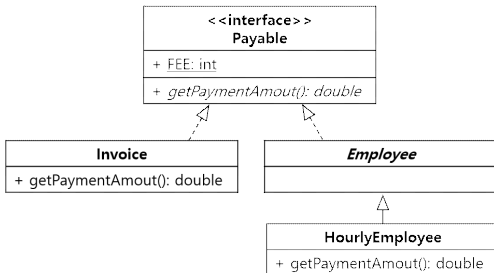
인터페이스는 서로 관련 없는 클래스가 공통 메소드와 상수를 공유해야 할 때 주로 사용되는데, 동일한 메소드 호출에 응답하여 관련 없는 클래스의 객체를 다형적으로 처리할 수 있게 해 준다. 인터페이스는 키워드 `interface`를 사용하여 정의되는데, Java 7 버전까지는 상수와 추상 메소드만을 멤버로 가질 수 있다. 인터페이스의 모든 필드는 암묵적으로 `public static final`, 즉 상수 필드이며 모든 메소드는 암묵적으로 `public abstract`이다. 인터페이스 멤버 선언 시 이들 수정자들을 생략할 수 있으므로 다음 두 가지 인터페이스 정의 방식은 동등한 표현이다.

```
public interface Payable {                // 인터페이스
    public static final int FEE = 100 ;    // 상수
    public abstract double getPaymentAmout(); // 추상 메소드
    ... // 상수 혹은 추상 메소드
}
```

```
public interface Payable {                // 인터페이스
    int FEE = 100 ;                       // 상수
    double getPaymentAmout();             // 추상 메소드
    ... // 상수 혹은 추상 메소드
}
```

인터페이스를 구현하여 추상 클래스나 구상 클래스를 정의할 수 있다. 키워드 `implements`를 사용하여 인터페이스의 모든 추상 메소드를 구현하면 구상 클래스가 되고, 일부 추상 메소드만 구현하면 추상 클래스가 된다. `public` 인터페이스는 인터페이스와 동일한 이름의 `.java` 파일에 선언되어야 한다. 인터페이스는 객체를 생성할 수 없으나 인터페이스 타입의 레퍼런스 변수는 선언 가능하다.

다음은 인터페이스 Payable과 이를 구현한 Invoice, Employee, HourlyEmployee 클래스의 UML 클래스 다이어그램과 관련 정의문이다.



```

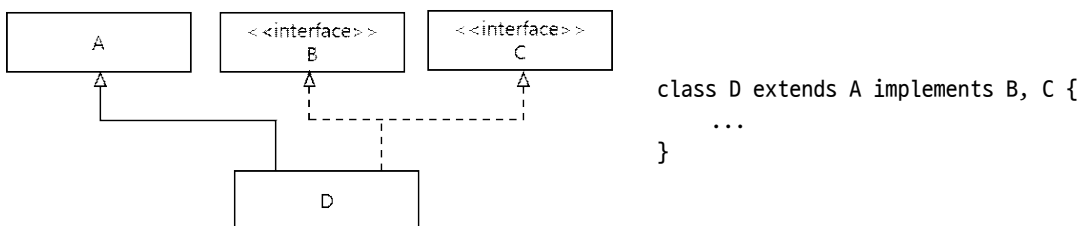
class Invoice implements Payable {
    public double getPaymentAmount() {
        ...
    }
}

abstract class Employee implements Payable {}

class HourlyEmployee extends Employee {
    public double getPaymentAmount() {
        ...
    }
}
  
```

인터페이스 `Payable`의 추상 메소드를 구현한 `Invoice`는 구상 클래스가 되고, 구현하지 않은 `Employee`는 추상 클래스가 된다. 추상 클래스 `Employee`의 서브 클래스인 `HourlyEmployee`가 구상 클래스가 되었다는 것은 is-a 관계에 있는 상위 계층의 인터페이스와 추상 클래스의 모든 추상 메소드가 여기서 구현되었음을 의미한다.

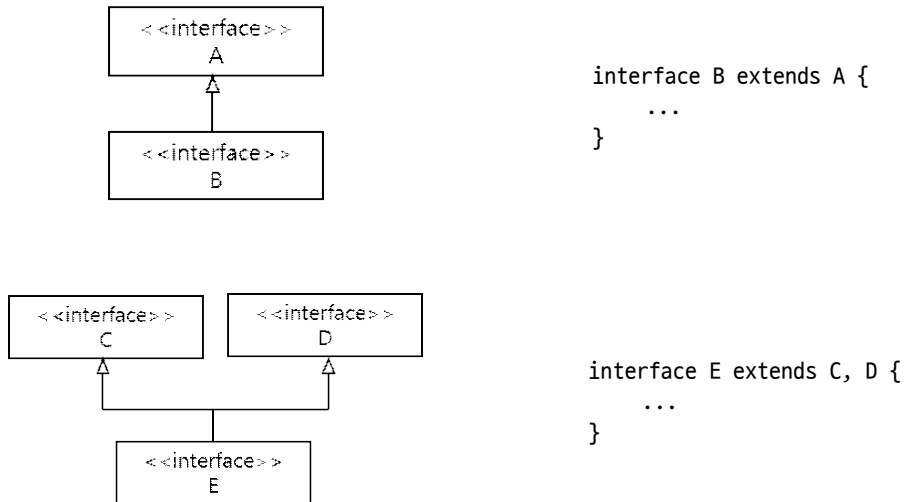
Java는 하나의 슈퍼 클래스를 상속하면서 필요한 만큼 여러 개의 인터페이스를 동시에 구현할 수 있다. 여러 개의 인터페이스를 구현한 클래스는 각 인터페이스와 is-a 관계를 가지게 된다.



```

class D extends A implements B, C {
    ...
}
  
```

또한, Java는 인터페이스 간의 다중 상속을 허용한다. 이 경우 서브 인터페이스를 정의할 때 슈퍼 인터페이스의 추상 메소드를 구현하는 것이 아니므로 키워드를 `extends`를 사용해야 한다.



Java 8 버전이 발표되면서 인터페이스에 추가된 개념이 디폴트 메소드(default method), 정적 메소드(static method) 그리고 함수형 인터페이스(functional interface)이다. 디폴트 메소드는 인터페이스 정의 시 `default` 키워드를 사용하여 기본적인 구현을 제공하는 메소드이다. 디폴트 메소드가 정의되어 있으면 인터페이스를 구현하는 클래스가 해당 메소드를 구현하지 않아도 메소드를 호출할 수 있다. 정적 메소드는 `static` 키워드를 사용하여 정의되는 메소드로 디폴트 메소드와 마찬가지로 구현 부분을 제공한다. 디폴트 메소드는 인터페이스를 구현한 클래스의 객체를 생성한 후 호출할 수 있으며 정적 메소드는 인터페이스 타입을 사용하여 바로 호출해야 한다는 것이 차이점이다. 다음은 인터페이스의 추상 메소드, 디폴트 메소드, 정적 메소드의 차이점을 확인할 수 있는 코드이다.

```

interface A {
    void method1(); // 추상 메소드, 암묵적으로 public abstract, 재정의의 구현 필수
    default void method2() { // 디폴트 메소드, 재정의 가능
        System.out.println("method2()");
    }
    static void method3() { // 정적 메소드, 재정의 불가
        System.out.println("method3()");
    }
}

class B implements A {
  
```

```

@Override
public void method1() { // public 이어야 함
    System.out.println("method1()");
}
}
public class InterfaceTest {
    public static void main(String[] args) {
        B b = new B();
        b.method1(); // 재정의 메소드 호출
        b.method2(); // 디폴트 메소드 호출
        A.method3(); // 정적 메소드 호출
    }
}

```

디폴트 메소드가 추가된 이유는 인터페이스가 기존 코드를 수정하지 않고 쉽게 확장될 수 있도록 하기 위함이다. 인터페이스의 추상 메소드는 클래스에 의해서 반드시 구현되어야 한다. 인터페이스에 새로운 추상 메소드가 추가되면 이 인터페이스를 사용하는 모든 클래스들이 수정되어야 한다. 디폴트 메소드는 이러한 문제를 쉽게 해결하는 방법이다. 인터페이스의 정적 메소드는 유틸리티 메소드나 팩토리 메소드를 제공하기 위해 주로 사용한다. 팩토리 메소드는 공장처럼 객체를 생성하는 정적 메소드를 말한다.

함수형 인터페이스(functional interface)는 오직 한 개의 추상 메소드를 가지는 인터페이스를 말하며, 추가로 디폴트 메소드나 정적 메소드를 멤버로 가질 수도 있다. 함수형 인터페이스는 단독으로 큰 의미가 없으며 주로 람다식(lambda expression)을 지원하기 위한 개념으로 사용된다. 함수형 인터페이스 선언 시 @FunctionalInterface라는 어노테이션을 추가하여 컴파일 시 함수형 인터페이스 조건에 맞는지 검사할 수도 있다. 함수형 인터페이스의 형식은 다음과 같다.

```

@FunctionalInterface
interface A {
    void method();
}

```

```

@FunctionalInterface
interface B {
    void method1();
    default void method2() {
        ...
    }
    static void method3() {
        ...
    }
}

```

Java 버전에 따른 인터페이스의 멤버 구성요소를 정리하면 다음과 같다.

인터페이스 멤버	Java 버전		
상수, 추상 메소드	Java 7	Java 8	Java 9
default 메소드, static 메소드			Java 10
private 메소드			

인터페이스의 추상 메소드는 public abstract로, default 메소드는 public으로 고정되어 있으며 이들 수정자는 생략될 수 있다. static 메소드의 경우 접근 지정자가 생략되면 public이며, private으로 지정될 수 있다. private 메소드는 인터페이스 내에서만 호출 가능하다. 추상 메소드 외 default, static, private 메소드는 모두 인터페이스 정의 시 구현 부분을 제공해야 한다.

B. 개념 확인

1. 다음 각 문장의 밑줄 친 곳을 채우시오.

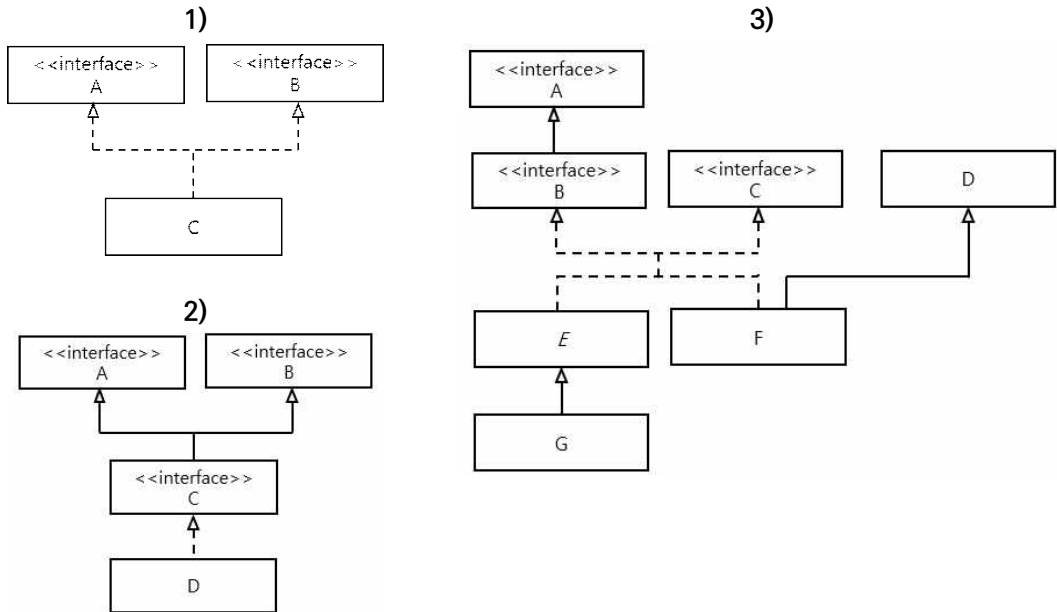
- 1) 인터페이스의 추상 메소드 중 일부를 구현한 클래스는 _____이다.
- 2) 인터페이스의 모든 추상 메소드를 구현한 클래스는 _____이다.
- 3) Java 8 버전의 인터페이스 멤버 종류는 _____이다.
- 4) 함수 인터페이스 정의 시 사용하는 어노테이션은 _____이다.
- 5) 인터페이스의 상수 필드의 수정자는 암묵적으로 _____이다.
- 6) 인터페이스의 추상 메소드의 수정자는 암묵적으로 _____이다.

2. 다음 각 진술문의 참 거짓을 말하시오.

- 1) 인터페이스 멤버로 디폴트 메소드나 정적 메소드가 아닌 구현 메소드가 포함될 수 있다.
- 2) 함수 인터페이스는 한 개의 추상 메소드를 멤버로 가지는 인터페이스이다.
- 3) 함수 인터페이스는 한 개의 추상 메소드 외 디폴트 메소드나 정적 메소드를 추가로 가질 수 없다.
- 4) Java는 인터페이스 간의 상속을 허용한다.
- 5) 클래스는 한 번에 두 개 이상의 클래스를 상속할 수 있다.
- 6) 클래스는 한 번에 두 개 이상의 인터페이스를 구현할 수 있다.

C. 응용 문제

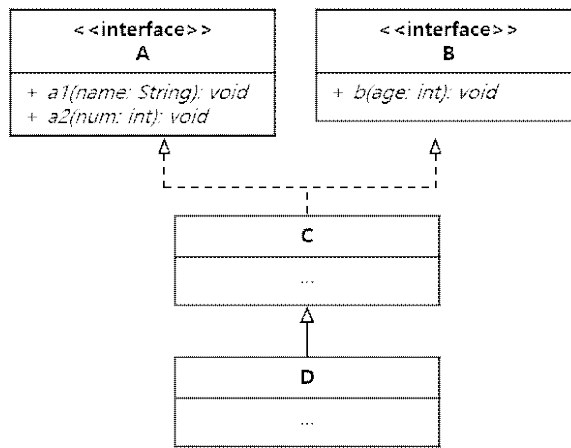
- 주어진 UML 다이어그램 구조대로 클래스와 인터페이스를 정의한 후 드라이버 클래스를 추가하여 테스트하는 프로그램을 작성하시오.



√ 세부 사항

- 최소 하나의 인터페이스는 상수 필드를 가진다.
- 최소 하나의 인터페이스는 디폴트 메소드나 정적 메소드를 가진다.
- 서로 다른 슈퍼 인터페이스가 같은 혹은 다른 이름의 추상 메소드를 가질 수 있다.
- 이탤릭체로 표시된 클래스 E는 추상클래스이다.
- 메소드의 구현 부분은 간단한 출력문으로 작성한다.
- 각 클래스에 자체적으로 메소드를 추가할 수 있다.
- UML 다이어그램의 인터페이스와 클래스 외 추가로 드라이버 클래스를 정의해야 한다.
- UML 다이어그램에서 실선 화살표는 상속을 점선 화살표는 구현을 의미한다.

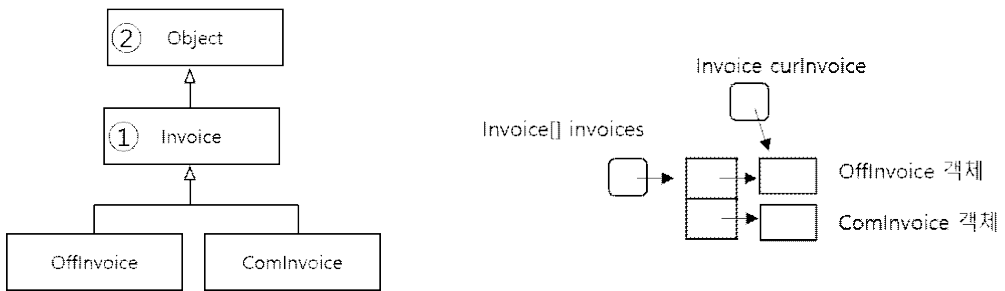
2. 다음 UML 다이어그램 구조를 만족시키는 클래스 C와 D를 정의하여 테스트하는 프로그램을 작성 하시오. 이때, 인터페이스 A, B의 메소드는 모두 추상 메소드이다. C는 상위 인터페이스의 모든 추상 메소드를 재정의하여 구현하되 그 몸체에 다른 실행 문장 없이 정의하는 어댑터 클래스이다. D는 어댑터 클래스 C의 메소드 중 필요한 메소드만 다시 목적에 맞게 재정의하여 사용하는 클래스이다. D의 메소드는 a1, a2, b 중 하나만 재정의 하여 간단하게 문자열을 출력하도록 한다.



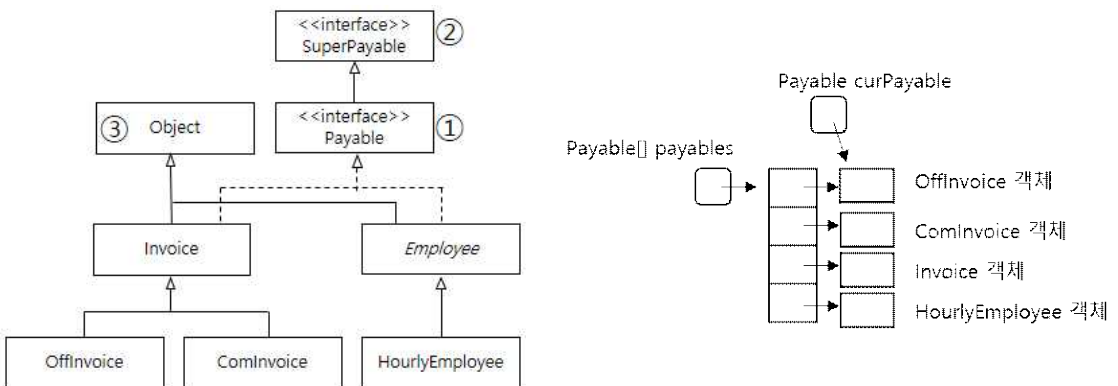
5.4 다형성 종합

A. 개념 정리

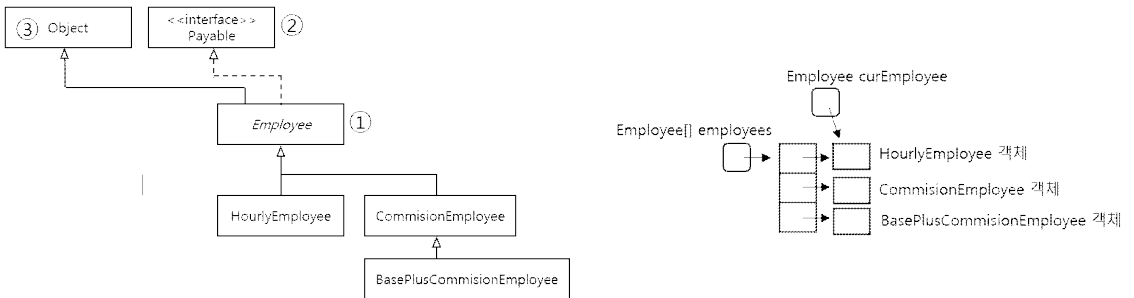
메소드를 다형적으로 호출하기 위해 구상 수퍼 클래스 참조, 인터페이스 참조 그리고 추상 수퍼 클래스 참조를 사용할 수 있다. 구상 수퍼 클래스 참조를 사용하여 ①구상 수퍼 클래스에 선언된 메소드, ②구상 수퍼 클래스의 수퍼클래스(예를 들어, Object)에 선언된 메소드를 다형적으로 호출할 수 있다. 이것은 다음 그림의 ①, ②에서 선언된 메소드가 서브 클래스인 OffInvoice와 ComInvoice에서 재정의되고, 실행 시 재정의된 메소드가 호출되는 경우이다.



인터페이스 참조를 사용하여 ①인터페이스에 선언된 메소드, ②인터페이스의 수퍼 인터페이스에 선언된 메소드, ③Object 클래스에 선언된 메소드를 다형적으로 호출할 수 있다. 이것은 다음 그림의 ①, ②, ③에서 선언된 메소드가 서브 클래스인 OffInvoice, ComInvoice, Invoice, HourlyEmployee에서 재정의되고, 실행 시 재정의된 메소드가 호출되는 경우이다.



추상 슈퍼 클래스 참조를 사용하여 ①추상 슈퍼 클래스에 선언된 메소드, ②추상 슈퍼 클래스의 슈퍼 인터페이스에 선언된 메소드, ③Object 클래스에 선언된 메소드를 다형적으로 호출할 수 있다. 이것은 다음 그림의 ①, ②, ③에서 선언된 메소드가 서브 클래스인 HourlyEmployee, CommissionEmployee, BasePlusCommisionEmployee에서 재정의되고, 실행 시 재정의된 메소드가 호출되는 경우이다.



B. 개념 확인

1. A. 개념 정리에서 기술한 내용을 확인하는 다형성 테스트 프로그램을 작성하고 그 결과에 대한 보고서 작성하시오. 개념 정리의 UML 다이어그램과 같이 인터페이스와 클래스를 정의하고 드라이버 클래스의 main 메소드에서 다음 세 가지 경우에 대해 테스트하면 된다.

√ 세부 사항

- Object 클래스는 암묵적으로 상속되도록 한다.
- 인터페이스나 클래스에 테스트 목적에 맞게 메소드를 적절하게 추가한다.
- 각 메소드 구현 시 어떤 클래스에서 선언 혹은 재정의된 메소드인지 알 수 있게 정보를 출력한다.
ex) `System.out.println("Invoice : printInfo()"); // Invoice의 멤버인 printInfo() 메소드`
- 메소드가 표현된 UML 다이어그램을 그려서 각 메소드의 재정의 관계를 한 눈에 볼 수 있도록 한다.
- UML 다이어그램의 클래스나 인터페이스 외 드라이버 클래스를 추가로 작성한다.
 - 1) 구상 수퍼 클래스 참조를 사용하여 ①구상 수퍼 클래스에 선언된 메소드, ②구상 수퍼 클래스의 수퍼클래스(예를 들어, Object)에 선언된 메소드를 다형적으로 호출하는 경우
 - 2) 인터페이스 참조를 사용하여 ①인터페이스에 선언된 메소드, ②인터페이스의 수퍼 인터페이스에 선언된 메소드, ③Object 클래스에 선언된 메소드를 다형적으로 호출하는 경우
 - 3) 추상 수퍼 클래스 참조를 사용하여 ①추상 수퍼 클래스에 선언된 메소드, ②추상 수퍼 클래스의 수퍼 인터페이스에 선언된 메소드, ③Object 클래스에 선언된 메소드를 다형적으로 호출하는 경우

6장 예외 처리

6.1 예외와 예외 처리

6.2 예외 계층

6.3 스택 풀기

6.4 연결된 예외

6장 예외 처리

6.1 예외와 예외 처리

A. 개념 정리

Java에서 예외(exception)란 프로그램 실행 중 발생하는 문제를 기술하기 위해 생성하는 객체를 말한다. 실행 중 어떤 문제가 생겼을 때 정해진 예외를 생성하여 예외 처리(exception handling) 블록으로 전달할 수 있다. 예외 처리 블록은 예외 처리기(exception handler)라고 불리며 전달된 예외를 기반으로 문제를 해결한 후 실행이 계속되도록 지원한다. 발생한 예외에 대해 예외 처리를 하지 않는 경우, Java 가상 기체가 그 예외에 담긴 에러 메시지와 메소드 호출 스택 정보를 출력하고 프로그램을 종료한다.

Program 6.1은 키보드 입력 형식 불일치로 인해 발생하는 InputMismatchException 예외에 대해 예외 처리를 하지 않은 경우이다. Scanner의 nextInt() 메소드로 정수를 입력 받는 곳에서 실수가 입력되면 InputMismatchException 예외가 발생된다. 이 예외에 대한 처리 블록이 없으므로 최종적으로는 에러 메시지와 호출 스택 정보가 출력되고 프로그램은 종료된다.

```
import java.util.Scanner;
public class NoExceptionHandlerTest1 {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter an integer: ");
        int number = input.nextInt(); // 정수 입력. InputMismatchException 발생 가능

        System.out.printf("You entered %d\n", number);
    }
}
```

Program 6.1 InputMismatchException에 대한 예외 처리를 하지 않은 경우

√ 실행 결과

```
Enter an integer: 100
You entered 100
```

√ 실행 결과

```

Enter an integer: 3.1
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:939)
    at java.base/java.util.Scanner.next(Scanner.java:1594)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
    at NoExceptionHandlingTest.main(NoExceptionHandlingTest.java:8)

```

Program 6.2는 정수 나누기에서 0으로 나누는 경우 발생하는 `ArithmeticException` 예외에 대해 예외 처리를 하지 않은 경우이다. "by zero"가 추가된 에러 메시지가 호출 스택 정보와 함께 출력되고 프로그램은 종료된다.

```

import java.util.Scanner;
public class NoExceptionHandlingTest2 {
    public static int quotient(int numerator, int denominator) {
        return numerator / denominator; // 정수 나누기의 분모가 0이면
        // ArithmeticException 예외 발생
    }
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter an integer numerator: "); // 분자
        int numerator = input.nextInt();
        System.out.print("Enter an integer denominator: "); // 분모
        int denominator = input.nextInt();
        int result = quotient(numerator, denominator);
        System.out.printf("quotient: %d / %d = %d\n", numerator, denominator, result);
    }
}

```

Program 6.2 `ArithmeticException`에 대한 예외 처리를 하지 않은 경우

√ 실행 결과

```

Enter an integer numerator: 123
Enter an integer denominator: 10
quotient: 123 / 10 = 12

```


√ 실행 결과

```

Enter an integer numerator: 123
Enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at NoExceptionHandlingTest.quotient(NoExceptionHandlingTest.java:19)
    at NoExceptionHandlingTest.main(NoExceptionHandlingTest.java:28)

```

예외 처리를 위한 기본 구문은 다음과 같다.

```

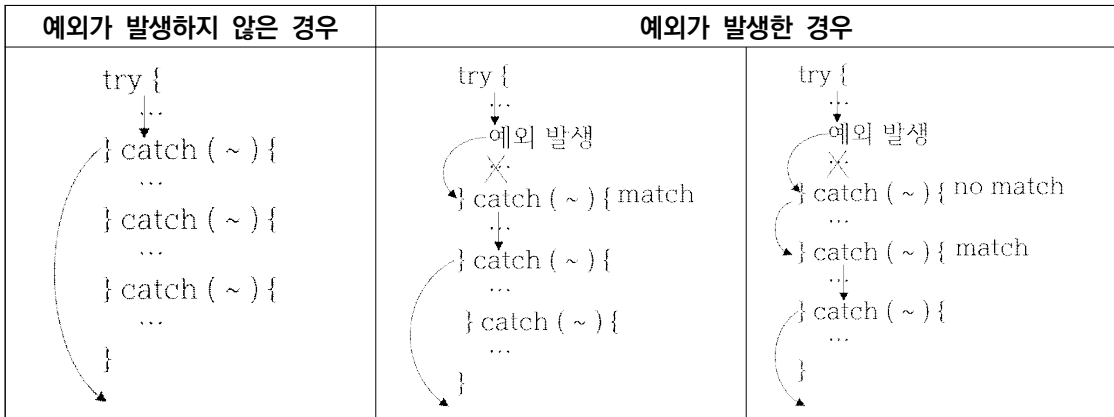
try {
    // 예외가 발생할 수 있는 코드
} catch ( 예외클래스 참조변수 ) { // 예외 매개변수
    // 예외 처리 코드
} finally {
    // 예외 발생 여부와 상관없이 마지막에 항상 실행되는 코드
}

```

try 블록에는 예외가 발생할 수 있는 코드가 오며 예외 처리기인 catch 블록은 예외를 전달받아 처리하는 코드가 온다. finally 블록은 try 블록의 예외 발생 유무에 무관하게 항상 실행되는 블록이다. 적어도 하나의 catch 블록이나 finally 블록이 try 블록 바로 뒤에 와야 한다. 예외 매개변수(exception parameter)는 예외 처리기가 잡아서 처리할 수 있는 예외 클래스 타입의 참조 변수이다. catch 블록은 필요한 만큼 여러 개 추가할 수 있으며 finally 블록은 불필요한 경우 생략할 수 있다.

예외는 try 블록에서 명시적으로 생성되거나 try 블록에서 시작된 연쇄적인 메소드 호출 과정에서 생성될 수도 있다. 혹은 Java 바이트코드를 실행함에 따라 Java 가상 기계에서 예외가 생성될 수도 있다.

예외 처리의 종료 모델은 다음과 같다.



try 블록에서 예외가 발생하지 않은 경우 모든 catch 블록을 건너 띄며, 예외가 발생한 경우 예외 객체의 타입과 매치되는 첫 번째 catch 블록을 실행한 후 마지막 catch 블록 다음으로 제어를 이동한다.

다음과 같이 예외 처리 구문을 반복문에 포함하는 코드 패턴을 사용할 수도 있다. 이 경우, try 블록에서 예외가 발생하면 catch 블록에서 문제를 해결한 후 다시 try 블록을 제대로 실행한 후 반복문을 벗어나도록 한다.

```

boolean flag = true;
do {
  try {
    예외가 발생할 수 있는 코드
    flag = false;
  } catch (예외클래스 참조변수) {
    ...
  }
}while(flag);
          
```

Program 6.3은 Program 6.1에 대해 InputMismatchException 예외 처리를 추가한 프로그램이다.

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class ExceptionHandlingTest1 {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        boolean flag = true;
        int number = 0;

        do {
            try {
                System.out.print("Enter an integer: ");
                number = input.nextInt();
                flag = false;
            } catch (InputMismatchException e) {
                System.out.printf("%nException: %s%n", e); // e.toString() 호출
                input.nextLine();
                System.out.printf("Not an integer. Try again.%n%n");
            }
        } while (flag);

        System.out.printf("You entered %d%n", number);
    }
}
```

Program 6.3 InputMismatchException 예외 처리

√ 실행 결과

```
Enter an integer: 3.1
```

```
Exception: java.util.InputMismatchException
Not an integer. Try again.
```

```
Enter an integer: 10
You entered 10
```

Program 6.4는 Program 6.2에 대해 InputMismatchException과 ArithmeticException 예외 처리를 추가한 프로그램이다.

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class ExceptionHandlingTest2 {
    public static int quotient(int numerator, int denominator) throws ArithmeticException {
        return numerator / denominator; // ArithmeticException 발생 가능
    }

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        boolean flag = true;

        do {
            try {
                System.out.print("Enter an integer numerator: "); // 분자
                int numerator = input.nextInt();
                System.out.print("Enter an integer denominator: "); // 분모
                int denominator = input.nextInt();
                int result = quotient(numerator, denominator);
                System.out.printf("quotient: %d / %d = %d\n", numerator, denominator,
result);
                flag = false;
            } catch (ArithmeticException e) {
                System.out.printf("%nException: %s\n", e); // e.toString() 호출
                System.out.printf("Zero is an invalid denominator. Try again.%n\n");
            } catch (InputMismatchException e) {
                System.out.printf("%nException: %s\n", e); // e.toString() 호출
                input.nextLine();
                System.out.printf("Not an integer. Try again.%n\n");
            }
        } while (flag);
    }
}
```

Program 6.4 InputMismatchException과 ArithmeticException 예외 처리

√ 실행 결과

```

Enter an integer numerator: 12
Enter an integer denominator: a

Exception: java.util.InputMismatchException
Not an integer. Try again.

Enter an integer numerator: 12
Enter an integer denominator: 0

Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Try again.

Enter an integer numerator: 12
Enter an integer denominator: 10
quotient: 12 / 10 = 1

```

메소드 헤더의 throws 절은 메소드 몸체가 던질 수 있는 예외를 기술하는 것이다. 여러 가지 예외를 던질 수 있는 경우, 콤마(,)를 사용하여 예외 종류를 나열하면 된다. 메소드는 throws 절에 기술된 예외 클래스와 그 서브 클래스의 예외를 모두 던질 수 있다.

```

public int method1() throws ExceptionA, ExceptionB {
    ...
    throw new ExceptionA(); // ExceptionA 예외를 던짐. 즉, 예외를 발생시킴
    ...
    method2(); // 메소드 호출을 통해 예외를 던질 수 있음
}

```

메소드에서 예외를 던지게 되면, 그 메소드는 남은 코드를 실행하지 않고 종료하며 값을 반환하지 않는다.

B. 개념 확인

1. 다음 코드의 밑줄 친 곳을 적절하게 채우시오.

```
try {
    if(!(point < 0)) {
        Exception e = new Exception("negative point error"); // 예외 객체 생성
        _____ e; // 예외를 던짐
    }
} _____ ( _____ ) {
    e.printStackTrace();
}
```

2. 다음 코드의 실행 결과는?

1)

```
public class ExceptionHandlingTest {
    public static void main(String[] args) {
        try {
            System.out.println("try...");
            throw new Exception();
        } catch (Exception e) {
            System.out.println("catch...");
        } finally {
            System.out.println("finally...");
        }
    }
}
```

2)

```
public class ExceptionHandlingTest {
    public static void main(String[] args) {
        try {
            System.out.println("try...");
        } catch (Exception e) {
            System.out.println("catch...");
        } finally {
            System.out.println("finally...");
        }
    }
}
```

3. 다음 코드의 밑줄 친 곳을 적절하게 채우시오

1)

```
public class ExceptionHandlingTest {
    public static void main(String[] args) {
        try {
            System.out.println("try...");
            int[] a = { 0, 1, 2, 3, 4 };
            a[5] = 5;
        } catch( _____ ) {
            System.out.println("catch...");
        }
    }
}
```

√ 실행 결과

```
try...
catch...
```

2)

```
public class ExceptionHandlingTest {
    public static void main(String[] args) {
        try {
            System.out.print("input a real number >> ");
            Scanner in = new Scanner(System.in);
            double dNum = in.nextDouble();
            System.out.printf("You entered %f\n", dNum);
        } catch (InputMismatchException e) {
            System.out.println("catch...");
        }
    }
}
```

√ 실행 결과

```
input a real number >> real
catch...
```

6.2 예외 계층

A. 개념 정리

Java에서 예외 처리에 사용될 수 있는 예외는 Throwable 클래스의 상속 계층에 속한 클래스의 예외이어야만 한다. Throwable 클래스는 Exception과 Error 2개의 서브 클래스를 가진다. Exception과 그 서브 클래스는 Java 프로그램 안에서 발생할 수 있는 예외적인 상황을 나타내며 이런 예외는 어플리케이션에서 잡아서 처리할 수 있다. Error와 그 서브 클래스는 JVM에서 매우 드물게 발생하는 비정상적인 상황을 나타내며 어플리케이션에 의해 잡혀서는 안 된다. 어플리케이션은 일반적으로 Error를 복구할 수 없다.

Throwable 상속 계층의 클래스는 점검 예외(checked exception)와 무점검 예외(unchecked exception)로 나뉜다. 컴파일러는 점검 예외에 대해서 잡거나 선언하기(catch or declare requirement)를 강제한다. 점검 예외에 대해서는 반드시 catch 블록으로 잡거나 throws절로 선언해야 하며 이를 지키지 않으면 컴파일 에러를 발생시킨다. 메소드 헤더의 throws절은 메소드 바디에서 잡히지 않은 예외를 명시하는 것이다. 무점검 예외는 이 규칙을 강제하지는 않으나 잡거나 선언하기를 구현할 수도 있다.

다음 그림은 Throwable 상속 계층의 점검 예외와 무점검 예외를 나타낸다. Error와 그 서브 클래스, RuntimeException과 그 서브 클래스는 무점검 예외이며 그 외 나머지 Throwable, Exception, IOException의 서브 클래스는 점검 예외이다.

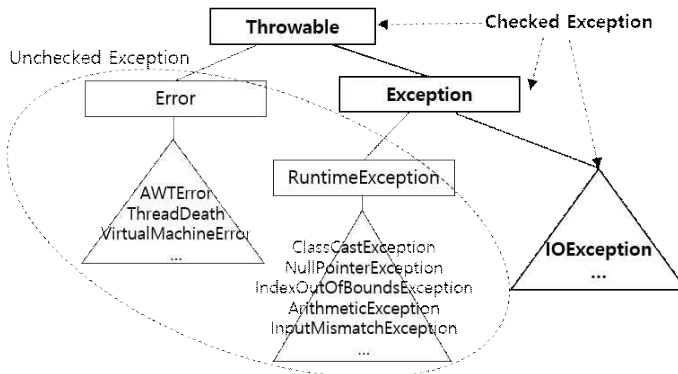
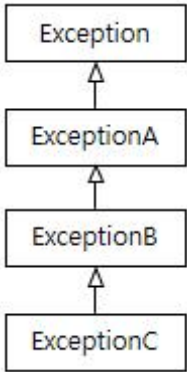


그림 6.1 점검 예외와 무점검 예외

수퍼 클래스 타입의 catch 매개변수는 해당 타입의 모든 서브 클래스 타입 예외를 잡을 수 있다. 이를 통해 catch는 관련 예외를 다형적으로 처리할 수 있다.



```

try {
    throw new Exception(); // ExceptionA(), ExceptionB(), ExceptionC()
} catch(Exception e) { // 수퍼 클래스 타입의 참조 변수
    // 다형적으로 메소드 호출 가능
}
  
```

is-a 관계에 있는 여러 예외에 대해 개별적인 처리가 필요한 경우 각 서브 클래스 별로 예외 처리기를 작성할 수도 있다. 이 경우, 예외 처리기의 catch 매개변수의 타입은 하위 클래스에서 상위 클래스 순서로 선언해야 한다. 예외가 여러 개의 catch 블록의 매개변수 타입과 매치되는 경우, 첫 번째 매치하는 catch블록에서 잡는다. 따라서 상위 클래스에서 하위 클래스 순서로 catch 매개변수를 선언한다면 첫 번째 catch 블록에서 모두 잡히게 되므로 개별적인 처리가 불가능하게 되며 컴파일 에러가 난다.

예외를 개별적으로 처리하는 코드	잘못된 코드
<pre> try { throw new Exception(); // ExceptionA(), ExceptionB(), ExceptionC() } catch(ExceptionC e) { ... } catch(ExceptionB e) { ... } catch(ExceptionA e) { ... } catch(Exception e) { ... } </pre>	<pre> try { throw new Exception(); //ExceptionA(), ExceptionB(), ExceptionC() } catch(Exception e) { ... } catch(ExceptionA e) { ... } catch(ExceptionB e) { ... } catch(ExceptionC e) { ... } </pre>

finally 블록은 try 블록의 예외 발생 유무에 상관없이 항상 실행된다. return, break 혹은 continue문을 사용하여 try 블록을 벗어나는 경우에도 finally 블록은 실행된다. 오직 System.exit() 호출로 try 블록을 조기 탈출하는 경우에만 finally 블록이 실행되지 않는다.

예외를 던지는 메소드

```

public static void throwException() throws
Exception {
    try {
        throw new Exception();
    } catch (Exception e) {
        throw e; // 예외 다시 던지기
        // 이 곳에 코드를 둘 수 없음
    } finally { // 실행됨
        System.err.println("finally...");
    }
    // 이 곳에 코드를 둘 수 없음
}

```

예외를 던지지 않는 메소드

```

public static void doseNotThrowException() {
    try {
        // 예외를 던지지 않음
    } catch (Exception e) {
        // 실행되지 않는 블록
    } finally { // 실행됨
        System.err.println("finally");
    }
    System.out.println("end of method");
}

```

throw 문장은 예외를 던지는데 사용된다. throw의 피연산자는 Throwable 클래스에서 파생된 클래스의 객체이어야 한다. catch 블록에서 잡은 예외를 다시 던지는 경우에는 외부 try 블록과 관련된 catch 블록에서 처리하거나 메소드 헤더에 throws 절로 해당 예외를 명시할 수 있다.

B. 개념 확인

1. 다음 프로그램의 실행 결과는?

1)

```
public class ExceptionTest {
    public static void main(String[] args) {
        try {
            System.out.println("try...");
            return;
        } finally {
            System.out.println("finally...");
        }
    }
}
```

2)

```
public class ExceptionTest {
    public static void main(String[] args) {
        try {
            System.out.println("try...");
            System.exit(0);
        } finally {
            System.out.println("finally...");
        }
    }
}
```

2. 다음 프로그램의 실행 결과는?

```
import java.io.IOException;
public class IOExceptionTest {
    static void method() throws IOException {
        throw new IOException();
    }
    public static void main(String[] args) {
        method();
    }
}
```

3. 다음 프로그램에 대해 주어진 실행 결과 나오도록 밑줄 친 곳을 채우시오.

```
import java.io.IOException;
import java.util.Scanner;
public class ExceptionTest {
    public static void method(int option) _____ {
        if (option == 1)
            throw new IOException();
        else if (option == 2)
            throw new Exception();
        System.out.println("A");
    }
    public static void main(String[] args) {
        try {
            Scanner in = new Scanner(System.in);
            int option = in.nextInt();
            method(option);
        } catch ( _____ ) {
            _____;
        } catch ( _____ ) {
            _____;
        }
    }
}
```

√ 실행 결과

input option [1/2/3] >> 1
B

√ 실행 결과

input option [1/2/3] >> 2
C

√ 실행 결과

input option [1/2/3] >> 3
A

4. 다음 프로그램의 실행 결과는?

```
public class PollTest {
    public static void main(String[] args) {
        int[] data = { 1, 3, 1, 2 };
        int[] freq = new int[3];
        for (int i = 0; i < data.length; i++) {
            try {
                System.out.print("a");
                ++freq[data[i]];
                System.out.print("b");
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.print("c");
            }
        }
        for (int i = 1; i < freq.length; i++) {
            System.out.printf("%d", i, freq[i]);
        }
    }
}
```

5. 다음 메소드에 대해 답하라.

```
public static void throwException() throws Exception {
    try {
        System.out.print("A"); // ①
        throw new Exception();
        System.out.print("B"); // ②
    } catch (Exception e) {
        System.out.print("C"); // ③
        throw e;
    } finally {
        System.out.print("D"); // ④
    }
    System.out.print("E"); // ⑤
}
```

- 1) ①~⑤ 중에서 "Unreachable Code"라는 컴파일 에러가 나는 문장을 모두 고르시오.
- 2) Unreachable Code 문장을 모두 삭제하고 메소드 헤더의 throws Exception 절을 제거한 후 코드를 추가하여 ACDF가 출력되도록 수정하라.

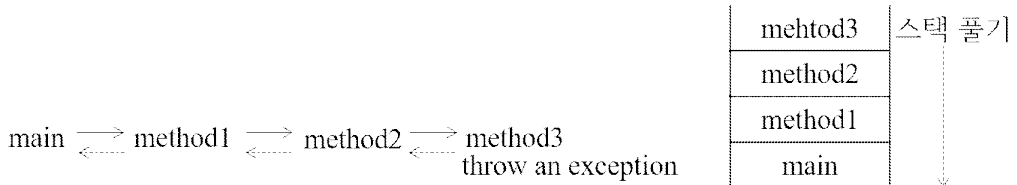
6. 다음 코드의 ① ~ ⑧ 중에서 틀린 것을 찾아 올바르게 수정하십시오.

```
public class ExceptionTest {
    // ①
    public static void method1() {
        throw new IndexOutOfBoundsException();
    }
    // ②
    public static void method2() throws IndexOutOfBoundsException {
        throw new IndexOutOfBoundsException();
    }
    // ③
    public static void method3() throws Exception {
        throw new IndexOutOfBoundsException();
    }
    // ④
    public static void method4() {
        try {
            throw new IndexOutOfBoundsException();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        method1(); // ⑤
        method2(); // ⑥
        method3(); // ⑦
        method4(); // ⑧
    }
}
```

6.3 스택 풀기

A. 개념 정리

예외가 발생했지만 특정 범위에서 잡히지 않은 경우 메소드 호출 스택을 풀어가면서(stack unwinding) 다음 바깥 try 블록과 관련된 catch 블록에서 예외를 잡으려고 시도한다. 만약 main → method1 → method2 → method3 순으로 메소드가 호출된 상태에서 예외가 던져진 경우, 이 예외를 처리하기 위해 method3 → method2 → method1 → main 순으로 호출 스택을 풀어가면서 예외를 처리할 수 있는 외부 catch 블록을 찾는다. 만약 특정 메소드에서 그러한 catch 블록이 없다면 점점 예외인 경우 메소드 헤더에 throws 절을 명시해야 한다.



```
public class StackUnwindingTest {
    public static void method1() throws Exception {
        method2();
    }
    public static void method2() throws Exception {
        method3();
    }
    public static void method3() throws Exception {
        throw new Exception("Exception thrown in method3");
    }
    public static void main(String[] args) {
        try {
            method1();
        } catch (Exception e) {
            System.err.printf("%s\n", e.getMessage());
            e.printStackTrace();
            StackTraceElement[] elements = e.getStackTrace();
            for (StackTraceElement element: elements) {
                System.out.printf("%s\t", element.getClassName());
                System.out.printf("%s\t", element.getFileName());
                System.out.printf("%s\t", element.getLineNumber());
            }
        }
    }
}
```

```

        System.out.printf("%s\n", element.getMethodName());
    }
}
}
}

```

Program 6.5 스택 풀기

√ 실행 결과

```

Exception thrown in method3
java.lang.Exception: Exception thrown in method3
    at StackUnwindingTest.method3(StackUnwindingTest.java:11)
    at StackUnwindingTest.method2(StackUnwindingTest.java:7)
    at StackUnwindingTest.method1(StackUnwindingTest.java:3)
    at StackUnwindingTest.main(StackUnwindingTest.java:16)
StackUnwindingTest  StackUnwindingTest.java  11  method3
StackUnwindingTest  StackUnwindingTest.java  7   method2
StackUnwindingTest  StackUnwindingTest.java  3   method1
StackUnwindingTest  StackUnwindingTest.java  16  main

```

Throwable 클래스의 메소드 `printStackTrace`, `getStackTrace` 그리고 `getMessage`가 주요하게 사용된다. `printStackTrace`는 스택 추적 정보를 표준 에러 스트림으로 출력한다.

`getStackTrace`는 스택 추적 정보를 반환하며 `getMessage`는 예외에 저장된 예외 문자열을 반환한다.

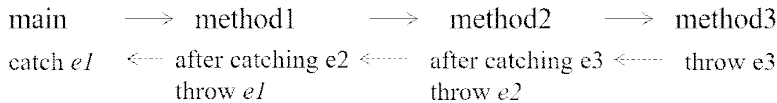
B. 개념 확인

1. Throwable 클래스의 메소드 중, 스택 추적 정보를 표준 에러 스트림으로 출력하는 메소드는 _____ 스택 추적 정보를 반환하는 메소드는 _____ 예외 문자열을 반환하는 메소드는 _____이다.

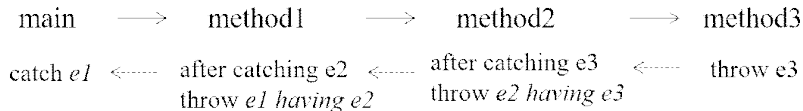
6.4 연결된 예외

A. 개념 정리

catch 블록이 새 예외를 생성하여 던지면 원래 예외의 정보와 스택 추적 정보가 손실된다. 예를 들어, 다음과 같이 메소드 호출과 스택 풀기가 진행되면서 연쇄적으로 새로운 예외를 생성하여 던질 경우, 최종적으로 main 메소드에서 잡은 예외 e1에는 e2, e3의 정보가 포함되지 않게 된다.



이와 달리 연결된 예외(chained exception)를 사용하면 예외 객체가 원래 예외의 완전한 스택 추적 정보를 유지할 수 있게 된다. 다음과 같이 catch 블록에서 잡은 예외를 포함하는 새로운 예외, 즉 연결된 예외를 생성하여 연쇄적으로 던지게 되면 마지막 main 메소드에서 잡은 e1에는 e2, e3의 모든 정보가 포함되게 된다.



```

public class ChainedExceptionTest {
    public static void method1() throws Exception {
        try {
            method2();
        } catch (Exception e) {
            throw new Exception("Exception thrown in method1", e); // chained exception
        }
    }

    public static void method2() throws Exception {
        try {
            method3();
        } catch (Exception e) {
            throw new Exception("Exception thrown in method2", e); // chained exception
        }
    }
}
  
```

```

    }
}

public static void method3() throws Exception {
    throw new Exception("Exception thrown in method3");
}

public static void main(String[] args) {
    try {
        method1();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

Program 6.6 연결된 예외

√ 실행 결과

```

java.lang.Exception: Exception thrown in method1
    at ChainedExceptionTest.method1(ChainedExceptionTest.java:6)
    at ChainedExceptionTest.main(ChainedExceptionTest.java:24)
Caused by: java.lang.Exception: Exception thrown in method2
    at ChainedExceptionTest.method2(ChainedExceptionTest.java:14)
    at ChainedExceptionTest.method1(ChainedExceptionTest.java:4)
    ... 1 more
Caused by: java.lang.Exception: Exception thrown in method3
    at ChainedExceptionTest.method3(ChainedExceptionTest.java:19)
    at ChainedExceptionTest.method2(ChainedExceptionTest.java:12)
    ... 2 more

```

사용자가 정의하는 예외 클래스는 예외 처리 메커니즘과 함께 사용할 수 있도록 기존 예외 클래스를 상속해야 한다. 일반적으로 예외 클래스에는 다음과 같이 네 가지 생성자를 포함하여 정의한다. 이들 중 Throwable 객체를 매개변수로 전달받는 마지막 두 생성자는 연결된 예외를 생성하는데 사용할 수 있다.

```

class MyException extends Exception {
    // ① 매개 변수가 없는 생성자이며 디폴트 에러 메시지를 수퍼 클래스의 생성자로 전달
    public MyException() {
        super("
            default error message");
    }
}

```

```

// ② 사용자 정의 예외 메시지를 매개 변수로 받아 수퍼 클래스의 생성자로 전달
public MyException(String message) {
    super(message);
}
// ③ Throwable 객체를 전달 받아 수퍼 클래스 생성자에 전달. 연결된 예외 생성
public MyException(Throwable cause) {
    super(cause);
}
// ④ 사용자 정의 예외 메시지와 Throwable 객체를 전달 받아 수퍼 클래스 생성자로 전달. 연
결된 예외 생성
public MyException(String message, Throwable cause) {
    super(message, cause);
}
}

```

Program 6.7 사용자 정의 예외 클래스

B. 개념 확인

1. 다음 각 문장에 대해 참 거짓을 말하십시오.

- 1) new Exception("Exception thrown in method1", e)은 연결된 예외를 생성하는 문장이다.
- 2) catch 블록에서 연결된 예외를 생성하여 던지면 원래 예외와 스택 추적 정보를 보존할 수 있다.
- 3) 연결된 예외를 생성하는 예외 생성자는 Throwable 타입 매개변수를 가져야 한다.
- 4) Java의 예외 처리 메커니즘을 사용하기 위해서는 반드시 Java에서 제공하는 예외만을 사용해야 한다.

C. 응용 문제

1. Java의 예외 처리 메커니즘을 따르는 MyIOException 클래스를 정의하고 테스트하는 프로그램을 작성하십시오. MyIOException 클래스는 IOException 클래스를 상속하여 네 개의 생성자를 정의하여야 한다.

√ 세부 사항

- MyIOException 클래스의 네 가지 생성자를 모두 사용해 보도록 테스트 코드를 작성한다.
- Eclipse 개발 환경의 경우, class MyIOException extends IOException { } 코드의 클래스 몸체 괄호 내부에서 마우스 우측 버튼을 눌러 [Source → Generate Constructors from Superclass...] 메뉴를 활용할 수 있다.

7장 람다식

7.1 람다식의 필요성

7.2 람다식 및 함수형 인터페이스

7.3 람다식의 용도

7장 람다식

7.1 람다식의 필요성

A. 개념 정리

Java 언어에서의 객체는 변수에 저장되거나 메소드에 매개변수로 전달되는 일급 객체(first class entity)이다. 하지만 하나의 코드 블록(또는 함수)은 일급 객체가 아니기 때문에 변수에 저장하거나 메소드의 매개변수로 전달할 수 없다. 나중에 실행시키기 위한 코드 블록을 변수에 저장하기 위해서는 이 코드 블록을 메소드화 하여 클래스를 정의한 후, 이 클래스의 객체를 생성하여 필요시 객체의 메소드를 호출해서 그 코드를 실행하는 것이 일반적이다. 이러한 방식은 함수형 언어가 일반적으로 제공하는 람다식(lambda expression)과 같은 간단한 코드 블록을 전달하는 방식에 비해 코드가 복잡하고 매우 번거롭다. 작은 함수와 같은 코드 블록을 변수에 저장하거나 매개변수로 전달할 수 있다면 코드를 매우 간결하게 작성할 수 있어 코드의 가독성을 높일 수 있다.

7.2 람다식 및 함수형 인터페이스

A. 개념 정리

람다식은 간단히 이름이 없는 함수(메소드)라고 정의할 수 있다. 람다식이 일반적인 메소드와 다른 점은 이름이 없다는 것 외에도 람다식은 변수에 저장될 수 있다는 점이다. 람다식의 일반적인 형식은 다음과 같다.

```
(parameters) -> expression
```

혹은

```
(parameters) -> { statement_list }
```

예를 들어, 두 개의 정수를 더하는 람다식은 다음과 같이 작성할 수 있다.

```
(int a, int b) -> a + b
```

위 코드의 의미는 두 개의 정숫값을 매개변수로 받아 합을 계산하여 반환하는 함수와 같다. 일반적인 함수와 다른 점은 함수의 이름을 가지고 있지 않다는 것이다.

다음 예는 하나의 매개변수 값 x 를 받아 제곱값을 반환하는 람다식이다.

```
(double x) -> x * x
```

Java 언어에서 람다식이 일급 객체로 사용되기 위해 타입 정보가 필요하다. Java 언어에서 람다식의 타입은 함수형 인터페이스(functional interface)로 정의된다. 함수형 인터페이스는 하나의 추상 메소드(abstract method)만을 가지는 인터페이스를 말한다. 함수형 인터페이스 F 가 람다식 e 의 타입이라는 말은 e 가 F 에 정의된 추상 메소드를 구현하고 있다는 것을 의미한다. 따라서 람다식의 매개변수의 수와 타입 및 반환값의 타입은 람다식이 구현하는 인터페이스가 정의하는 메소드와 일치해야 한다.

다음은 함수형 인터페이스의 예들이다.

```
public interface Exp {
    int eval(int a, int b);
}

public interface Func {
    double apply(double x);
}

public interface Predicate < T > {
    boolean test(T t);
}
```

위 함수형 인터페이스 Exp는 앞에서 예를 든 람다식 "(int a, int b) -> a + b"의 타입으로 사용될 수 있다. 또한 함수형 인터페이스 Func은 "(double x) -> x * x"의 타입으로 사용될 수 있다. 따라서 다음과 같은 변수의 선언과 지정문이 가능해진다.

```
Exp e = (int a, int b) -> a + b;
```

```
Func f = (double x) -> x * x;
```

마지막 함수형 인터페이스의 예인 Predicate<T>는 Java 라이브러리로 정의되어 있는 제너릭 인터페이스(generic interface)의 예이다. 제너릭 프로그래밍(generic programming)의 개념은 8장에서 자세하게 다룰 예정이다.

이제 변수에 저장된 람다식이 어떻게 실행되는지 살펴보자. 람다식이 실행은 람다식의 타입에 정의된 추상 메소드의 호출로 이루어진다. 다음은 위에서 정의한 람다식을 실행한 예이다.

```
e.eval(10, 20) // 계산 결과: 30
f.apply(2.0)   // 계산 결과: 4.0
```

람다식은 메소드의 매개변수로 전달될 수도 있다. 아래 코드는 Exp 타입의 람다식을 매개변수로 전달받아 실행한 후 그 결과를 반환하는 메소드이다.

```
int foo(Exp e, int a, int b) {
    return e.eval(a, b);
}
```

다음 두 문장은 메소드 foo에 람다식을 매개변수로 전달하는 예를 보여준다. 각 예에서 람다식 "(int a, int b) -> a + b"와 "(int x, int y) -> x - y"는 모두 foo 메소드의 매개변수 e의 타입인 Exp와 일치하므로 매개변수 e에 전달이 가능하다.

```
System.out.println("result = " + foo((int a, int b) -> a + b, 10, 20)); // result = 30
System.out.println("result = " + foo((int x, int y) -> x - y, 30, 10)); // result = 20
```

앞에서는 람다식의 몸체가 하나의 수식인 경우를 보았다. 몸체가 하나의 수식이 아닌 하나 이상의 문장으로 이루어져 있을 경우에는 문장들을 { }로 묶는다.

```
(String first, String second) -> {
    if (first.length() < second.length()) return -1;
    else if (first.length() > second.length()) return 1;
    else return 0;
}
```

위 람다식을 아래와 같이 간단하게 구현할 수도 있다. 반환 값을 -1, 0, 1 대신에 음수, 0, 양수로 나타낸다.

```
(String first, String second) -> first.length() - second.length()
```

매개변수가 하나만 있는 경우에는 ()를 생략할 수도 있다.

```
x -> x * x
```

매개변수의 타입을 유추할 수 있는 경우라면 타입을 생략할 수 있다. 앞에서 언급한 Exp 인터페이스를 구현한 "(int a, int b) -> a + b"에서 매개변수의 int형은 Exp의 eval 메소드의 매개변수 타입으로

로 유추할 수 있기 때문에 다음과 같이 표현할 수 있다.

(a, b) -> a + b

아래 문장의 람다식은 Comparator<String>의 메소드 compare(String, String)을 구현하고 있기 때문에 first와 second가 String 타입이라는 사실을 컴파일러는 쉽게 유추할 수 있다.

```
Comparator<String> c = (first, second) -> first.length() - second.length();
```

B. 개념 확인

1. 다음 각 문항을 위한 람다식을 정의하시오.

- 1) 하나의 int 형 값을 받아 세제곱을 계산하여 반환
- 2) 하나의 double 형 값을 받아 제곱근을 계산하여 반환, 단 매개변수 값은 0보다 크다고 가정함
- 3) 두 개의 스트링을 받아 결합(concatenate)하여 반환
- 4) 하나의 스트링을 받아 그 스트링이 null이면 빈 스트링("")을 반환하고 null이 아니면 그 스트링을 그대로 반환

2. 다음 함수형 인터페이스와 형식이 일치하는 람다식을 모두 고르시오.

```
interface Consumer {
    void accept(int x);
}
```

- a. x -> x * x
- b. x -> System.out.println(x * x)
- c. (int x) -> { return x * x; }
- d. (x, y) -> x * y

3. 다음 함수형 인터페이스와 형식이 일치하는 람다식을 모두 고르시오.

```
interface Function {
    double apply(double x);
}
```

- a. x -> x * x

-
- b. `(x) -> Math.sin(x)`
 - c. `(int x) -> System.out.println(x * x)`
 - d. `(int x) -> {
 if(x>0) return x+1;
 else if(x<0) return x-1;
 else return x;
}`

4. Comparable<String>과 형식이 일치하지 않는 램다식을 모두 고르시오.

- a. `s -> { if(s.equals("hello")) return 1; else return 0; }`
- b. `(String s) -> { if(s.equals("exit")) System.exit(-1); }`
- c. `(x, y) -> { if(x.equals(y)) return 1; else return 0; }`

5. 본문에서 정의한 함수형 인터페이스 Exp 형의 램다식을 세 개만 정의하시오.

6. 다음 각 문항을 위한 함수형 인터페이스를 정의하시오. 인터페이스와 메소드의 명칭은 자유롭게 정한다.

- 1) 두 개의 double 형 값을 받아 double 형 값을 반환
- 2) 하나의 스트링을 받아 int 형 값을 반환
- 3) 하나의 Employee 객체를 받아 boolean 값을 반환

7.3 람다식의 용도

A. 개념 정리

람다식은 대표적인 용도는 다음과 같다.

- 고차원 함수(high-order function)의 구현
- GUI 이벤트 리스너(event listener)의 구현
- 무명 내부 클래스(anonymous inner class)의 대체
- 컬렉션(collection) 데이터의 처리
- 스트림 API(stream API)에서 스트림 데이터의 처리

위에서 언급한 용도 중 몇 가지에 대해서 예를 들어 설명하기로 한다.

고차원 함수는 함수를 매개변수로 가지는 함수를 말한다. 예를 들어, 주어진 배열의 원소에 특정 함수를 적용하여 모두 합한 값을 구하는 함수를 작성한다고 하자. 이 함수는 다음과 같은 수식으로 정의될 수 있다. 여기서 함수 f 는 \sum 의 매개변수로 전달되는 함수이다.

$$\sum f(x), x = a_0, a_1, a_2, \dots, a_n$$

이 함수를 Java 메소드(sigma)로 구현해보자.

```
static double sigma(Func f, double[] a) {
    double sum = 0;
    for (int i = 0; i < a.length(), i++) {
        sum += f.apply(a[i]);
    }
    return sum;
}
```

sigma의 첫 번째 매개변수는 Func 타입으로(앞 절에서 정의한 함수형 인터페이스) 람다식을 전달받을 수 있다. 여러 가지 람다식을 매개변수로 전달하여 sigma를 호출해보자.

```
double[] a = {1.0, 2.0, 3.0, 4.0};
System.out.println("result = " + sigma(x ->x * x, a));
System.out.println("result = " + sigma(x ->1.0 / x, a));
System.out.println("result = " + sigma(x -> Math.sin(x), a));
```

Java GUI 프레임워크인 Swing에서의 대표적인 이벤트는 Action 이벤트로 버튼이나 체크박스과 같은 컴포넌트에서 발생하는 이벤트이다. Action 이벤트를 처리하기 위해서는 이벤트를 발생시키는 컴포넌트에 ActionListener라는 이벤트 리스너를 등록해주어야 한다. 이러한 이벤트 리스너를 구현할 때 클래스를 사용하지 않고 간단하게 람다식으로 구현할 수 있다. 다음은 ActionListener 인터페이스의 모습이다.

```
public interface ActionListener {
    void actionPerformed(ActionEvent e);
}
```

아래는 ActionListener를 람다식으로 구현해 버튼에 등록하는 예로써 EXIT 버튼을 누르면 시스템을 종료시키는 작업을 한다. (괄호와 세미콜론에 주의!)

```
JButton button = new JButton("EXIT");
button.addActionListener(e -> { System.exit(-1); });
```

Java 언어는 다양한 집합적인 데이터를 처리하기 위해 컬렉션 프레임워크(collection framework)를 제공한다. 이러한 집합적인 데이터를 처리하는 메소드의 매개변수로 람다식이 유용하게 사용된다. 아래는 String의 ArrayList에 removeIf와 forEach 메소드에 각각 람다식을 전달하고 있다.

```
String[] stra = {"Apple", "Banana", "Kiwi", "Peach"};
List<String> strb = Arrays.asList(stra);
strb.removeIf(s -> s.strlen() == 5);
strb.forEach(s -> { System.out.println(s); });
```

세 번째 라인은 strb에서 각 스트링의 길이가 5인 원소를 제거하는 일을 하고 마지막 라인은 각 원소를 출력하는 일을 한다. 마지막 라인은 메소드 참조(method reference)라는 개념으로 다음과 같이 좀 더 간결하게 작성할 수도 있다. 메소드 참조에 대해서는 여기서 생략하기로 한다.

```
strb.forEach(System.out::println);
```

B. 개념 확인

1. 다음 프로그램은 주어진 String 배열의 원소들을 그 길이를 비교하여 오름차순으로 정렬하는 프로그램이다. (가) 부분의 밑줄 친 곳을 동일한 의미의 람다식으로 변경하시오.

```
import java.util.*;

class StringLengthComparator implements Comparator<String> {
    public int compare(String a, String b) {
        return a.length() - b.length();
    }
}

public class StringSortTest {
    public static void main(String[] args) {
        String[] names = {"Lee", "Chang", "Hong"};
        Arrays.sort(names, new StringLengthComparator()); // (가)
        for (String s: names) {
            System.out.printf("%s ", s);
        }
        System.out.println();
    }
}
```

2. 다음은 javax.swing.Timer를 이용하여 일정한 시간(interval)이 지나면 시간 정보를 출력하는 프로그램이다. 밑줄 친 곳에 ActionListener 변수에 TimerActionListener 객체를 생성하여 지정하지 않고 람다식을 지정하여 사용하도록 수정하시오.

```
import java.util.*;
import java.awt.event.*;
import javax.swing.Timer;

class TimerActionListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("The time is " + new Date());
    }
}

public class TimerTest {
    public void main(String[] args) {
        ActionListener listener = new TimerActionListener();
        Timer timer = new Timer(1000, listener);
        timer.start();
    }
}
```

다음 Employee 클래스는 문제 3~5번에 공통적으로 사용된다.

```
class Employee {
    private String name;
    private int salary;

    public Employee(String n, int s) {
        name = n;
        salary = s;
    }

    public String getName() {
        return name;
    }
    public int getSalary() {
        return salary;
    }
    public String toString() {
        return name + ":" + salary;
    }
}
```

3. 아래 프로그램은 Employee 객체를 급여(salary) 순(오름차순)으로 정렬하는 프로그램이다. 컴파일해 본 후 어떤 문제가 있는지 설명하시오.

```
import java.util.*;

public class EmployeeSortTest {
    public static void main(String[] args) {
        List<Employee> elist = new ArrayList<>();
        elist.add(new Employee("Lee", 10000));
        elist.add(new Employee("Kim", 20000));
        elist.add(new Employee("Park", 8000));
        elist.add(new Employee("Han", 15000));

        Collections.sort(elist);
        System.out.println(elist);
    }
}
```

4. 3번의 문제점을 Comparator 인터페이스를 구현하는 람다식을 이용하여 해결하고자 한다. 어디를 어떻게 수정하면 되는가? 단, Employee 클래스는 수정할 수 없다고 가정한다.

5. 다음은 Predicate<T t> 인터페이스를 이용하여 Employee 배열에서 특정 조건을 만족하는 객체의 개수를 파악하는 코드이다. 가) ~ 나)의 조건을 만족하도록 빈 곳에 적합한 람다식을 각각 채우시오.

```
public class HowManyEmployeeTest {
    Employee[] emps = new Employee[3];

    emps[0] = new Employee("Kim", 10000);
    emps[1] = new Employee("Lee", 20000);
    emps[2] = new Employee("Kim", 30000);

    int n = Employee.howMany(emps, _____); // (가)
    int m = Employee.howMany(emps, _____); // (나)
    System.out.println("Number of High - salaried Employees = "+n);
    System.out.println("Number of Employees named Kim = "+m);

    public static int howMany(Employee[] emps, Predicate < Employee > t) {
        int n = 0;
        for (Employee e: emps) {
            if (t.test(e)) {
                n++;
            }
        }
        return n;
    }
}
```

- 1) salary가 20000 이상인 Employee 객체의 수
- 2) Kim이라는 이름을 가진 Employee 객체의 수

C. 응용 문제

1. 람다식을 이용하여 다음과 같은 형식을 가진 수식(prefix expression)을 하나 읽어 들여 계산하는 프로그램을 작성하고자 한다.

```
+ 10 20
- 30 5
* 3 5
```

아래 코드는 인터페이스 Expression과 메소드 f를 이용하여 세 가지 연산(+, -, *)을 계산하는 코드의 일부이다. 물음에 답하시오.

```
interface Expression {
    int eval(int a, int b);
}
public class ExpressionTest {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        char op = in.next().charAt(0);
        int x = in.nextInt();
        int y = in.nextInt();
        Expression exp = null;

        switch (op) {
            // (A)
        }
        // (B)
        System.out.println("result = " + result);
    }
    public static int f(Expression e, int a, int b) {
        return e.eval(a, b);
    }
}
```

- 1) (A)부분에 op에 따라 exp 변수에 적절한 람다식을 지정하는 문장을 작성하시오.
 - 2) (B)부분에 exp를 f에 전달하여 계산한 계산 결과를 result에 저장하는 문장을 작성하시오.
 - 3) 위 프로그램을 수정하여 연속적인 수식을 읽어 계산 결과를 출력하는 프로그램을 완성하시오.
2. 다음은 구분구적법을 이용하여 주어진 함수의 적분을 계산하는 프로그램이다. main 메소드에서의 integral 메소드 호출을 보고 이에 적합한 integral 메소드를 구현하시오.

```
interface F {
    double apply(double x);
}
```



```
public class IntegralTest {
    // f는 적분할 함수, a, b는 적분 구간, n은 등분 수
    public static double integral(F f, double a, double b, int n) {
        // 채울 것
    }
    public static void main(String[] args) {
        F f;
        double r;
        f = x -> x * x;
        r = integral(f, 0, 10, 100);
        System.out.printf("f(x)=x*x, 0, 10, 100: %.5f%n", r);
        f = x -> Math.sin(x);
        r = integral(f, 0, Math.PI, 200);
        System.out.printf("f(x)=sin(x), 0, 10, 200: %.5f%n", r);
    }
}
```

8장 제네릭 프로그래밍

8.1 제네릭 기초

8.2 제한된 타입 매개변수

8.3 와일드카드

8장 제네릭 프로그래밍

8.1 제네릭 기초

A. 개념 정리

1) 원시(raw) 타입

타입 인자를 적용할 수 있지만 적용하지 않은 타입을 raw 타입 클래스라고 한다. 예를 들어 인터페이스 List는 List<String>처럼 타입 인자를 사용할 수 있는 타입이지만 List x 와 같이 raw 타입으로 사용할 수도 있다. 이러한 raw 타입은 런타임 예외를 발생시키는 문제가 있다.

```
List animalList = new ArrayList();
animalList.add(animal1);
animalList.add("String"); // Bug!!
animalList.add(animal5);
. . .
for ( Object animal: animalList ) {
    feed((Animal)animal); // Exception
}
```

2) 제네릭(generic) 타입

위의 예에서 "String"이 add된 세 번째 줄에서 오류를 발생되어 확인할 수 있다면 개발 부담이 적어진다. 또한 런타임 예외 대신 컴파일 오류를 발생시키면 더욱 바람직하다. generic 타입은 타입 인자를 통해 이와 같이 지원한다. 아래 예는 위의 예에서 발생하는 예외를, generic 타입을 통해 컴파일 오류로 처리하는 것을 보여준다. for문에서 타입 캐스팅이 필요없다는 장점도 가진다.

```
List<Animal> animalList = new ArrayList<Animal>();
animalList.add(animal1);
animalList.add("String"); // Compile error!!!
animalList.add(animal5);
. . .
for ( Animal animal: animalList ) {
    feed( animal );
}
```

3) 배열과 비교

유사한 개념인 배열도 원소의 타입에 대해 generic 타입의 인자와 같이 컴파일타임 오류를 내어줄 수 있을 것처럼 보이지만 사실이 아니다.

```
Animal[] animals = new Animal[3];
animals[0] = new Animal();
animals[1] = "String" ; // no Compile error!!!
animals[2] = new Animal();
...
Animal a = animals[1];
a.feeding();           // Runtime Exception!!!
```

generic 타입을 가지는 클래스의 정의: 클래스는 타입 인자와 <, > 기호를 통해 generic 타입 인자를 필요로 함을 나타낸다. 다음은 generic 클래스의 정의와 활용의 예이다.

```
// defining a class
public class Cache<T> {
    private T value;
    public Cache(T v) { value = v; }
    public T get() {return value; }
}

// using the class
Cache<String> stringCache =
    new Cache<String>("abc");
String s = stringCache.get();
Cache<Integer> integerCache =
    new Cache<Integer>(new Integer(1));
Integer i = integerCache.get();
```

4) generic 타입의 static 메소드 정의

static 메소드는 클래스와 독립적인 타입인자를 가져야한다. 아래는 static 메소드 f가 generic 인자를 받기위해 정의되는 방법을 보여준다. 정의하는 클래스와 동일한 타입 인자를 가지는 경우 컴파일 오류를 발생시킨다.

```
public class M <T> {
    static void f(T a) {... // compile error
    // cannot make a static reference to the non static type T

public class M {
```

```

static<E> void f(E a) {...
// O.K. the static type E
usage
M.<String>f("aaa");

```

B. 개념 확인

1. 다음은 raw 타입 클래스 Box이다. 해당 클래스를 Generic으로 수정하여 임의의 타입의 content를 담을 수 있도록 코드를 작성하시오.

```

public class Box {
    private Object content;

    public Box(Object content) {
        this.content = content;
    }

    public Object getContent() {
        return content;
    }

    public void setContent(Object content) {
        this.content = content;
    }
}

```

2. Generic으로 수정된 Box 클래스에서 아래와 같은 코드를 실행하면 오류가 발생한다. 어떤 오류인지 쓰고, 그 이유를 서술하시오.

```

Box<Integer> intBox = new Box<>("Hello");
Box<String> stringBox = new Box<>(5);

```

3. 다음은 raw 타입 클래스 Pair이다. 해당 클래스를 Generic으로 수정하여 first는 타입 T, second는 타입 V로 설정하여 담을 수 있도록 코드를 작성하시오.

```

public class Pair {
    private Object first;
    private Object second;

    public Pair(Object first, Object second) {
        this.first = first;
        this.second = second;
    }
}

```

```
}

public Object getFirst() {
    return first;
}

public Object getSecond() {
    return second;
}

public void setFirst(Object first) {
    this.first = first;
}

public void setSecond(Object second) {
    this.second = second;
}
}
```

4. Generic으로 수정된 Pair 클래스에서 아래와 같은 코드를 실행하면 오류가 발생한다. 어떤 오류인지 쓰고, 그 이유를 서술하시오.

```
Pair<String, Integer> pair1 = new Pair<>("Hello", 5.0);
Pair<Integer, Double> pair2 = new Pair<>(3, "World");
```

5. 다음은 Object 배열 array를 매개변수로 받아 원소를 출력하는 함수이다. generic을 사용하여 임의의 타입의 배열을 출력할 수 있도록 코드를 작성하시오.

```
public class ArrayUtil {
    public static void printArray(Object[] array) {
        for (Object element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

6. 다음은 raw 타입 클래스 Calculator이다. 해당 클래스를 Generic으로 수정하고 add 함수를 사용하여 임의의 타입 숫자를 더할 수 있게 코드를 작성하시오.

```
public class Calculator {
    public static double add(Object num1, Object num2) {
        return num1.doubleValue() + num2.doubleValue();
    }
}
```

7. 다음은 raw 타입 인터페이스 List이다. 해당 List 인터페이스를 Generic으로 수정하여 임의의 타입의 원소를 담을 수 있도록 코드를 작성하시오.

```
public interface List {
    void add(Object element);
    Object get(int index);
}
```

8. 아래 코드를 보고 어떤 오류가 일어날 지 생각해보고, 코드를 실행해보시오.

```
public class Main {
    public static void main(String[] args) {
        HashSet<Integer> set = new HashSet<>();
        // 인자없는 HashSet으로 지정하는 경우에는
        // HashSet s = set;
        // s.add("Hello");
        // warning으로 처리 가능

        set.add(10);
        set.add("Hello");
    }
}
```

9. 다음은 위 8번의 코드에 대해 오류가 발생하지 않도록 수정하고 간단한 확인 코드를 추가한 것이다. 이와 같은 수정을 하면 어떠한 단점이 있을지 생각해보시오.

```
public class Main {
    public static void main(String[] args) {
        HashSet<Object> set = new HashSet<>();

        set.add((Object) 10);
        set.add((Object) "Hello");

        Iterator itr = set.iterator();

        // 잘 동작하는지 확인하기 위한 코드
        while(itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

C. 응용 문제

1. 아래 Raw 타입 클래스인 MySet 정의를 Set의 원소의 타입을 타입 인자로 받는 generic 타입으로 바꾸어 보시오.

```
class MySet {
    static int MAX = 1000;
    void insert(Object o) {
        ...
    }
    void remove(Object o) {
        ...
    }
}
```


8.2 제한된 타입 매개변수

A. 개념 정리

1) Bounded 타입 인자

generic 타입에서는 타입 인자의 범위를 한정(bound)할 수 있다. 예를 들어 `class A<E extends Animal>` 과 같이 generic 클래스 A의 타입 인자를 키워드 `extends`를 통해 클래스 `Animal`의 하위 타입 또는 `Animal` 자신으로 한정하는 경우가 대표적이다. 따라서 E가 나타낼 수 있는 타입은 `Animal`, `Dog`, `Cat` 또는 이들의 하위 타입으로 한정되게 된다.

2) Bounded 타입의 장점

generic 타입의 인자의 범위를 특정 타입의 하위 타입으로 한정하는 경우 클래스 캐스팅이 줄어들고 동시에, 해당 타입의 변수에 대해 더 풍부한 메소드 호출을 할 수 있다. 예를 들어 `Animal`에 `feed`라는 메소드가 정의되어 있다고 할 때, 클래스 `A<E extends Animal>`에 대해 A 내에서 E 타입으로 정의된 변수에 대해 `feed`를 호출할 수 있다. 이 때 x의 실제 타입은 `Animal`뿐 아니라 `Dog`이나 `Cat`을 전달하는 것도 가능하다.

```
class A<E extends Animal> {
    ...
    void doSomething(E x) {
        x.feed();
    }
}
...
a.doSomething(new Dog());
a.doSomething(new Cat());
a.doSomething(new Animal());
```

3) 재귀적 Bounded 타입

generic 타입과 generic 타입의 인자는 자기자신을 써서 재귀적으로 정의될 수 있다. 예를 들어 타입 인자 T를 가지는 generic 클래스 A는 "`class A<T extends A<T>>{...}`"와 같이 정의될 수 있는데, 이것은 generic 클래스 A가 취하는 타입 인자 T가 A<T>의 하위 타입이라는 의미로 다소 복잡하다. 그러나 이것은 타입 T의 특성이 같은 타입 T의 개체들간에 특정 작업을 할 수 있는 타입이라는

의미를 가지게 된다.

아래 예에서와 같이 인터페이스 Comparable<T>는 타입 T와 비교 가능한 (compareTo(T o)를 호출할 수 있는) 객체들을 지칭하는 인터페이스이다. 최대값을 구하는 max함수는 List<T>를 인자로 받는데, 크기 비교를 위해 내부적으로 사용하는 메소드 compareTo를 이 인자 t에 대해 호출할 수 있으려면 T는 compareTo 메소드를 가지고 있어야 동작한다. 즉 t의 타입인 T는 인터페이스 Comparable<T>의 하위 클래스여야 동작하므로 T extends Comparable<T>를 명시해야한다.

```
public interface Comparable < T > {
    int compareTo(T o);
}

// Returns the maximum value in a list
public static <T extends Comparable <T>> T max(List <T> list) {
    Iterator<T> i = list.iterator();
    T result = i.next();
    while (i.hasNext()) {
        T t = i.next();
        if (t.compareTo(result) > 0) // t should have compareTo()
            result = t;
    }
    return result;
}
```

B. 개념 확인

1. 아래의 Box 클래스에서 T의 타입을 Animal 또는 Animal의 하위 타입으로 제한하여 코드를 작성하시오.

```
public class Box<T> {
    private T content;

    public Box(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }

    public void setContent(T content) {
        this.content = content;
    }
}
```

2. 수정된 a 코드를 활용하여 아래 코드를 실행하면 어떠한 문제가 생기는 지 서술하시오.

```
Box<String> stringBox = new Box<>("Hello");
```

3. 다음 코드에서 'K'와 'V' 타입을 String 또는 String의 하위 타입으로 제한하세요.

```
public class KeyValue<K, V> {
    private K key;
    private V value;

    public KeyValue(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

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

```

public void setKey(K key) {
    this.key = key;
}

public void setValue(V value) {
    this.value = value;
}
}

```

4. 다음 코드에서 E의 타입을 Comparable로 제한한 클래스로 제한하세요.

```

public class MaxFinder<E> {
    // generic 부분 생략하여 문제 만들기.
    public static <E> E findMax(List<E> list) {
        E max = list.get(0);
        for (E item : list) {
            if (item.compareTo(max) > 0) {
                max = item;
            }
        }
        return max;
    }
}

```

5. 다음은 Calculator 클래스이다. 이 클래스는 제네릭 method add를 가지며, T는 Number 클래스 혹은 하위 클래스이다. 아래 코드에서 어떤 문제가 생기는 지 서술하시오.

```

public class Calculator {
    public static <T extends Number> double add(T num1, T num2) {
        return num1.doubleValue() + num2.doubleValue();
    }
}

double result = Calculator.add(5, "10");

```

C. 응용 문제

1. Union(합집합)이 구현된 MySet 함수를 Generic T를 활용하여 제작하시오.

```

import java.util.HashSet;
import java.util.Set;

public class MySet<T> {
    private Set<T> set = new HashSet<>();

    public void add(T element) {

```

```
        set.add(element);
    }

    public boolean contains(T element) {
        return set.contains(element);
    }

    public Set<T> getSet() {
        return set;
    }

    // 합집합을 반환하는 메소드
    public MySet<T> union(MySet<? extends T> otherSet) {
        // 빈 칸을 채우시오.
    }

    public static void main(String[] args) {
        MySet<Integer> set1 = new MySet<>();
        set1.add(1);
        set1.add(2);
        set1.add(3);

        MySet<Integer> set2 = new MySet<>();
        set2.add(3);
        set2.add(4);
        set2.add(5);

        MySet<String> set3 = new MySet<>();
        set3.add("apple");
        set3.add("banana");

        // Integer 타입의 합집합
        MySet<Integer> unionSet = set1.union(set2);
        System.out.println("Integer Union Set: " + unionSet.getSet());

        // String 타입의 합집합
        MySet<String> unionSet2 = set3.union(set3);
        System.out.println("String Union Set: " + unionSet2.getSet());
    }
}
```

- 차집합, 교집합을 구현한다고 할 때, 당면하는 문제는 어떤 것이 있는지 생각해보고, 이유를 적으시오.

8.3 와일드카드

A. 개념 정리

1) 타입 Compatibility

타입 Super가 타입 Sub의 상위타입이고, 타입 Sub의 개체는 타입 Super가 나타날 수 있는 곳 어디에든 사용될 수 있다. 예를 들어 Super x = new Sub(); 와 같은 식이 가능한데, Sub y = new Super(); 는 컴파일 오류를 발생시킨다. 이에 따라 클래스 Dog과 Cat이 클래스 Animal의 하위 타입이면, 아래와 같이 클래스 Animal 개체를 인자로 받는 메소드 add의 인자로 Dog이나 Cat 객체가 전달 될 수 있다. 이렇게 하위클래스 객체를 상위클래스 타입 변수에 대입 가능한 경우를, 해당 하위클래스 타입이 상위 클래스 타입에 compatible 하다고 한다.

```
List<Animal> animalList1 = new ArrayList<Animal>();
animalList.add(new Cat());
animalList.add(new Dog());
```

2) 주의사항

상위타입과 하위타입을 각각 인자로 하는 generic 타입은 compatible 하지 않다. 예를 들어, Cat은 Animal의 하위클래스이지만 List<Cat>은 List<Animal>에 compatible 하지 않다. 예를 들어, 아래와 같이 List<Dog> 타입의 개체를 List<Animal>에 지정하면 컴파일타임 오류가 발생한다. Dog의 List와 Animal의 List는 서로 compatible 하지 않기 때문이다. 만일 오류를 발생시키지 않는다면, dogList 에 Cat 객체를 삽입되어 런타임 오류가 발생하는 것을 막을 수 없기 때문이다.

```
List<Dog> dogList = new ArrayList<Dog>();
dogList.add(new Dog());
```

```
List<Cat> catList = new ArrayList<Cat>();
CatList.add(new Cat());
```

```
List<Animal> animalList = dogList; // Compile error
// to prevent inserting a Cat to dogList;
// by animalList.add(new Cat());
```

3) 와일드카드 타입(Wildcard type)

이러한 경우 타입은 compatible 하지 않지만 지정을 허용하는 것이 자연스럽다. 특히 animalList에 Cat 객체 등 다른 것을 삽입하지 않는 것이 보장된다면 문제가 될 것이 없다. '?' 등과 같이 와일드카드 타입으로 선언된 변수는 내용이 변화되지 않는다는 것이 보장되는 객체를 나타내는 타입이다. 즉, animalList 변수에는, 이후 Cat 등을 삽입할 수 있는 add 메소드의 호출이 불가능하다. List<Dog>은 List<Animal>에 compatible 하지 않지만, List<? extends Animal>에는 compatible 하다.

```
List<? extends Animal> animalList = dogList; // OK
```

참고로 "<?)" 는 <? extends Object>를 나타낸다.

인터페이스 List의 메소드 addAll을 사용하면 해당 리스트에 다른 리스트를 통째로 add할 수 있다. 실제로 Java API의 클래스 Collection 타입의 경우, addAll을 아래와 같이 정의하고 있다. 즉 Collection<E>의 인자로 Collection<E> 타입 개체를 받지 않고, Collection<? extends E> 개체를 받음으로써 E의 하위 타입 Collection을 인자로 받을 수 있다. 대신 인자 c는 addAll 메소드 내에서 변경될 수 없다.

	addAll(Collection<? extends E> c)
boolean	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.

B. 개념 확인

1. (타입 compatability) 다음 밑줄에 알맞은 (컴파일 오류가 없는) generic type을 기술하시오. 단 Super는 Sub의 상위 클래스이다.

```
(1)_____ x = new (2)_____ ();
x.add(new Sub());
x.add(new Super());
(3)_____ y ;
if (getUserInput()) { // 그냥 사용자 입력으로 생각할 것
    y = new List<Super> ();
} else y = new List<Sub> ();
for (Super i : y) {
    System.out.println(i);
}
```

2. 어떤 동물들의 목록을 다루는 클래스를 만드려고 한다. 이 동물들의 종류는 최소 5종류 이상이다. 아래 질문에 답하시오.

- 1) 이에 대해 와일드 카드가 필요한가? 본인의 생각과 그에 대한 근거를 작성하시오.
- 2) 아래 Animal 클래스를 Animal을 상속받는 와일드 카드 타입을 사용하여 작성해보시오.

```
import java.util.*;

class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

class AnimalList {
    List<Animal> animals;

    public AnimalList(List<Animal> animals) {
        this.animals = animals;
    }

    public void printAllAnimals(List<?> animals) {
        for (Object animal : animals) {
```



```

        System.out.println(((Animal) animal).getName());
    }
}

```

3. 두 개의 리스트를 비교하여 공통된 요소를 찾는 findCommonElement 메소드를 작성하려고 한다. 아래 질문에 답하시오.

1) 두 리스트의 요소 타입이 동일한 경우, 와일드 카드가 필요한가? 본인의 생각과 그에 대한 근거를 작성하시오.

2) 두 리스트의 요소 타입이 다른 경우, 와일드 카드가 필요한가? 필요하다면 그에 대한 본인의 생각과 근거를 작성하고, 이를 토대로 findCommonElement 함수를 작성하시오.

```

import java.util.*;

class ListComparer {
    // 2.1) 두 리스트의 요소 타입이 동일한 경우
    public static List<Object> findCommonElements(List<Object> list1, List<Object> list2)
    {
        List<Object> commonElements = new ArrayList<>();

        for (Object element : list1) {
            if (list2.contains(element)) {
                commonElements.add(element);
            }
        }

        return commonElements;
    }
}

```

C. 응용 문제

1. 8.2의 C. 응용 문제 문제에서 구현한 MySet 함수를 Generic T과 super를 활용하여 차집합 메소드를 추가 구현하시오.

```
import java.util.HashSet;
import java.util.Set;

public class MySet<T> {
    private Set<T> set = new HashSet<>();

    public void add(T element) {
        set.add(element);
    }

    public boolean contains(T element) {
        return set.contains(element);
    }

    public Set<T> getSet() {
        return set;
    }

    // 합집합을 반환하는 메소드
    public MySet<T> union(MySet<? extends T> otherSet) {
        MySet<T> newSet = new MySet<>();
        newSet.getSet().addAll(this.getSet());
        newSet.getSet().addAll(otherSet.getSet());
        return newSet;
    }

    // 차집합을 반환하는 메소드
    public MySet<T> difference(MySet<? super T> otherSet) {
        // 빈 칸을 채우시오.
    }

    public static void main(String[] args) {
        MySet<Integer> set1 = new MySet<>();
        set1.add(1);
        set1.add(2);
        set1.add(3);

        MySet<Integer> set2 = new MySet<>();
        set2.add(3);
        set2.add(4);
        set2.add(5);
    }
}
```

```
MySet<String> set3 = new MySet<>();
set3.add("apple");
set3.add("banana");

// Integer 타입의 합집합
MySet<Integer> unionSet = set1.union(set2);
System.out.println("Integer Union Set: " + unionSet.getSet());

// Integer 타입의 차집합
MySet<Integer> differenceSet = set1.difference(set2);
System.out.println("Integer Difference Set: " + differenceSet.getSet());
}
}
```

9장 컬렉션 프레임워크

9.1 인터페이스 & 클래스

9.2 반복자와 Map

9.3 알고리즘을 인자로 전달

9장 컬렉션 프레임워크

9.1 인터페이스 & 클래스

A. 개념 정리

Java의 컬렉션 프레임워크(Collection framework)는, 일반적인 자료구조의 구현체를 제공하기 위한 것으로 인터페이스(Interface, 아래 그림 9.11)와 이를 구현하는 클래스로 이루어져 있다.

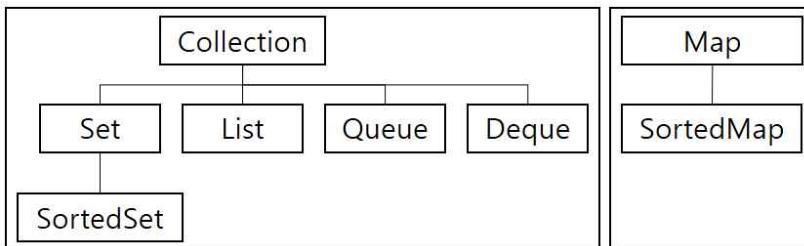


그림 9.1 Java의 Collection framework에서 인터페이스

대표적으로 인터페이스 Collection은 인터페이스 Set, List, Map 등의 상위 인터페이스이다. Collection은 원소를 더하고 제거하는 등의 기본적인 연산을 메소드로 제공한다. 하위 인터페이스들은 각 개념의 특징을 추가로 메소드로 가지게 된다. 예를 들어 List는 i 번째 요소를 가져오는 메소드가 있고 Set에는 대응되는 메소드가 존재하지 않는다.

각 인터페이스들은 자신을 구현하는 클래스들을 가지고 있다. 예를 들어 인터페이스 List는 클래스 ArrayList와 LinkedList에 의해 구현된다. 중간에 삽입, 삭제가 자주 일어나면 LinkedList가 선호되고 특정 원소를 인덱스로 접근하는 작업이 더 자주 일어난다면 ArrayList를 사용하는 것이 낫다. 비슷하게, 인터페이스 Set은 클래스 HashSet과 TreeSet에 의해 구현되며, 순서와 정렬이 자연스러운 집합에 대해서는 TreeSet을, 좋은 해시함수가 존재할 경우 HashSet이 선호된다.

주어진 인터페이스에 대해 원하는 구현 클래스가 없다면 새로 만들 수도 있는데, 이 때에는 처음부터 만들지 않고 추상 Skeleton 클래스들을 이용하면 좀더 수월하다. 예를 들어 인터페이스 List를 구현하는 MyList를 만들고자 할 때, List를 implements 하는 것과 함께 추상 Skeleton 클래스인

1) The Java™ Tutorials,

<https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html>

AbstractList를 extends 하면 AbstractList 에 이미 구현된 메소드를 사용함으로써 기계적인 코딩의 양을 줄일 수 있다. 또한 추가 코드량이 적어지기 때문에 추상 Skeleton 클래스인 anonymous 클래스에서 활용되기 쉬워진다,

B. 개념 확인

1. AbstractList를 상속받는 MyList 클래스를 작성해보시오.

```
import java.util.AbstractList;

public class MyList<E> extends AbstractList<E> {
    private Object[] elements;
    private int size;

    public MyList(int initialCapacity) {
        if (initialCapacity < 0) {
            throw new IllegalArgumentException("Illegal Capacity: " + initialCapacity);
        }
        this.elements = new Object[initialCapacity];
    }

    @Override
    public E get(int index) {
        if (index < 0 || index >= size()) {
            throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size());
        }
        return (E) elements[index];
    }

    @Override
    public int size() {
        // 빈 칸을 채우시오.
    }

    @Override
    public boolean add(E element) {
        // 빈 칸을 채우시오.
    }
}
```

2. AbstractSet를 상속받는 MySet 클래스를 작성해보시오.

```
import java.util.AbstractSet;
import java.util.Iterator;

public class MySet<E> extends AbstractSet<E> {
    private Object[] elements;
    private int size;
    public MySet() {
        elements = new Object[10]; // 초기 크기를 10으로 설정
        size = 0;
    }
    @Override
    public Iterator<E> iterator() {
        return new Iterator<E>() {
            private int currentIndex = 0;
            @Override
            public boolean hasNext() {
                return currentIndex < size;
            }
            @Override
            public E next() {
                if (!hasNext()) {
                    throw new IndexOutOfBoundsException("No more elements to iterate.");
                }
                return (E) elements[currentIndex++];
            }
        };
    }
    @Override
    public int size() {
        // 빈 칸을 채우시오.
    }
    @Override
    public boolean add(E element) {
        // 빈 칸을 채우시오.
    }
    \ @Override
    public boolean contains(Object element) {
        // 빈 칸을 채우시오.
    }
    @Override
    public boolean remove(Object element) {
        // 빈 칸을 채우시오.
    }
}
```

3. AbstractMap을 상속받는 MyMap 클래스를 작성해보시오.

```

import java.util.AbstractMap;
import java.util.Set;

public class MyMap<K, V> extends AbstractMap<K, V> {
    private Object[] keys;
    private Object[] values;
    private int size;

    public MyMap() {
        keys = new Object[10]; // 초기 크기를 10으로 설정
        values = new Object[10];
        size = 0;
    }

    @Override
    public Set<Entry<K, V>> entrySet() {
        return new AbstractSet<Entry<K, V>>() {
            @Override
            public Iterator<Entry<K, V>> iterator() {
                return new Iterator<Entry<K, V>>() {
                    private int currentIndex = 0;
                    @Override
                    public boolean hasNext() {
                        return currentIndex < size;
                    }
                    @Override
                    public Entry<K, V> next() {
                        if (!hasNext()) {
                            throw new IndexOutOfBoundsException("No more elements to
iterate.");
                        }
                        K key = (K) keys[currentIndex];
                        V value = (V) values[currentIndex];
                        currentIndex++;
                        return new SimpleEntry<>(key, value);
                    }
                };
            }
        };
    }

    @Override
    public int size() {
        return size;
    }
};
}

```



```
@Override
public V put(K key, V value) {
    // 빈 칸을 채우시오.
}
}
```

4. 구현한 클래스를 활용하여 아래 코드를 실행해보고 결과와 같은지 비교하시오.

```
import java.util.*;

public class CollectionExample {
    public static void main(String[] args) {
        List<Integer> list = new MyList<>();
        Set<String> set = new MySet<>();
        Map<String, Integer> map = new MyMap<>();
        System.out.println("MyList Test");
        // List test
        for(int i = 5, j = 2; i >= 0 && j <= 7; i--, j++) {
            list.add((i * 10) + j);
        }
        for(int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }
        System.out.println();
        System.out.println("Myset Test");
        // Set test
        set.add("Hi");
        System.out.println("set size is " + set.size());
        set.add("My");
        System.out.println("set size is " + set.size());
        set.add("Name");
        System.out.println("set size is " + set.size());
        set.add("Is");
        System.out.println("set size is " + set.size());
        set.add("Java");
        System.out.println("set size is " + set.size());
        System.out.println("Is \"Hi\" in Set? : " + set.contains("Hi"));
        System.out.println("Is \"C++\" in Set? : " + set.contains("C++"));
        System.out.println();
        System.out.println("MyMap Test");
        // Map test
        map.put("Apple", 1);
        System.out.println("map size is " + map.size());
        map.put("Banana", 5);
        System.out.println("map size is " + map.size());
    }
}
```

```
map.put("Orange", 3);
System.out.println("map size is " + map.size());
map.put("Pineapple", 3);
System.out.println("map size is " + map.size());
map.put("Melon", 1);
System.out.println("map size is " + map.size());
System.out.println("Is \"Pineapple\" in Map? : " + map.containsKey("Pineapple"));
System.out.println("Is \"Lion\" in Map? : " + map.containsKey("Lion"));
System.out.println("What is the value of \"Apple\"? : " + map.get("Apple"));
}
}
```

√ 실행 결과

```
MyList Test
52
43
34
25
16
7

Myset Test
set size is 1
set size is 2
set size is 3
set size is 4
set size is 5
Is "Hi" in Set? : true
Is "C++" in Set? : false

MyMap Test
map size is 1
map size is 2
map size is 3
map size is 4
map size is 5
Is "Pineapple" in Map? : true
Is "Lion" in Map? : false
What is the value of "Apple"? : 1
```

C. 응용 문제

1. Interface Map을 이용하여 전화번호부를 만들어보시오.

9.2 반복자와 Map

A. 개념 정리

Collection의 각 원소를 순회하는 대표적인 방법 중에는 반복자(iterator)를 취해서 각 원소를 접근하는 방법이 있다. 아래는 iterator의 사용 코드이다. Collection의 메소드 iterator는 iterator를 반환하며, 메소드 hasNext는 순회할 원소가 남아있는지 여부를 주어 반복문의 종료 검사에 사용된다. 원소를 가지고 오는 핵심 메소드는 next이다. 변경이 없을 때에는 for문으로 좀더 간결하게 표현할 수도 있다.

```
Collection < String > x = ...;

// for(String t : x) {
//     System.out.println(t + " "); 더 간결한 for문
// }

Iterator<String> iter1 = x.iterator();
while (iter1.hasNext()) {
    System.out.println(iter1.next() + "");
}
```

while 문 내에서 메소드 remove 또는 removeIf 등을 사용하여 특정 조건을 만족하는 경우 원소를 바로 삭제할 수 있다. 그러나 iterator의 메소드 remove, removeIf 등은 특수한 메소드로서 주의해서 사용해야한다. 예를 들어 두 개의 iterator가 동시에 실행되는 경우에 remove 호출은 안전하지 않으며, 마지막에 해당 iterator를 통해 접근한 원소를 제거하게 되므로 remove를 연속으로 두 번 이상 호출은 불가능하다,

인터페이스 Map은 다수의 <키, 값>의 쌍을 보관하는 자료구조를 표현하고 있으며, Set과 유사하게 TreeMap과 HashMap 등의 클래스에서 구현되어 있다. 메소드 get, put 등으로 키를 통해 값을 가져오거나 키에 새로운 값을 지정한다, entrySet은 모든 원소를 반환하는 메소드인데, iterator나 for문으로 순회하여 원소를 가져올 때에는 각 쌍에 대해 아래와 같이 getKey와 getValue를 통해 키와 값에 접근할 수 있다.

```
for (Map.Entry<String, Integer> entry : counts.entrySet()) {
    String k = entry.getKey();
    String v = entry.getValue();
}
```

B. 개념 확인

1. 아래는 HashMap에 임의로 설정된 key-value 쌍을 추가하는 코드이다. Iterator를 사용하여 다음 HashMap에 어떤 값이 저장되어 있는지 확인하는 코드를 작성해보시오.

```
import java.util.HashMap;

public class Main {
    public static void main(String args[]) {
        HashMap<String, Integer> map = new HashMap<>();
        String[] arr = {"Java", "Hello", "C++", "XYZ", "ABC"};

        for(int i = 0; i < 3; i++) {
            int key = (int) (Math.random() * 5);
            int value = (int) (Math.random() * 10);
            map.put(arr[key], value);
        }

        // 빈칸 채우기
    }
}
```

2. 아래는 리스트의 요소들을 Iterator를 이용하여 출력하는 함수인 useIterator와 이를 활용하는 코드를 나타낸다. ???로 되어 있는 부분을 적절히 채우시오.

```
import java.util.Iterator;
import java.util.List;

public class IteratorExample {
    // Iterator를 이용해 List의 요소들을 출력하는 함수
    public static void useIterator(???) {
        ???
    }

    public static void main(String[] args) {
        List<String> myList = List.of("Apple", "Banana", "Cherry");
    }
}
```

```
// List의 Iterator를 가져옴
Iterator<String> iterator = myList.iterator();

// printListUsingIterator 함수에 Iterator를 전달하여 실행
useIterator(iterator);
}
}
```

9.3 알고리즘을 인자로 전달

A. 개념 정리

알고리즘은 특정 자료구조에 국한되지 않는 경우가 많다. Collection framework에서 다양한 알고리즘을 지원하는 메소드들은 각 인터페이스나 클래스에 각각 정의되어 있지 않고, 클래스 Collections에 모여있다. 클래스 Collections는 객체를 생성할 수 없는 클래스이며 대신 최대/최소 값 구하기, 정렬 등을 유용한 static 메소드 max, min, sort 등을 통해서 지원한다. 그런데, 이러한 메소드들은 범용적인 Collection에 대해 정의된 것이기 때문에, 추가로 정렬 기준 및 크기 비교 방법 등의 정보를 필요로한다. 이러한 정보는 클래스의 Comparator의 객체로 포장되어 메소드의 인자로 전달된다. Comparator는 두 객체를 비교하여 음수, 0, 양수를 리턴하는 메소드인 int compare(T o1, T o2)가 구현되어 있는 기능 위주의 객체이다. 아래의 첫 번째는 comparator를 사용하지 않은 sort이고 두번째는 comparator를 사용한 sort 호출 예이다.

```
List<String> ls = new ArrayList();
'''
Collections.sort(ls); // without comparator
Collections.sort(ls, new Comparator<String>(){
    int comparable(String s1, String s2) {return s1.length - s2.length;}
}); // with a Comparator
Collections.sort(ls, Collections.reverseOrder()); // with a built-in Comparator
```

B. 개념 확인

1. 다음 (a)와 (b)는 `java.util.HashMap<K,V>`의 메소드 `keySet()`에 대한 두 가지 구현의 일부이다. `keySet()`은 `map`에 들어있는 `key`들에 대한 `Set`을 리턴 한다. (a)와 (b)의 차이점을 기술하십시오.

(a)

```
public Set<K> keySet() {
    Set res = new HashSet();
    Iterator<Entry<K, V>> i = entrySet().iterator();
    while (i.hasNext()) {
        res.add(i.next());
    }
    return res;
}
```

(b)

```
transient volatile Set<K> keySet = null;
public Set<K> keySet() {
    keySet = new AbstractSet<K> () {
        public Iterator <K> iterator() {
            return new Iterator<K> () {
                private Iterator<Entry<K, V>> i = entrySet().iterator();
                public boolean hasNext() {
                    return i.hasNext();
                }
            }...
        };
    }
    public int size() {
        return AbstractMap.this.size();
    }
    public boolean isEmpty() {
        return AbstractMap.this.isEmpty();
    }
    }...
};
return keySet;
}
```

10장 Swing: 그래픽 사용자 인터페이스 프레임워크

10.1 Swing: Java GUI 프레임워크 소개

10.2 이벤트 모델

10.3 레이아웃 매니저

10.4 Swing 컴포넌트

10.5 그래픽 기초

10장 Swing: 그래픽 사용자 인터페이스 프레임워크

10.1 Swing: Java GUI 프레임워크 소개

A. 개념 정리

그래픽 사용자 인터페이스(GUI)는 텍스트 기반 사용자 인터페이스와는 달리 키보드 외에 마우스 등과 같은 편리한 수단으로 응용프로그램과의 대화를 가능케 한다. 응용 프로그램의 그래픽 사용자 인터페이스는 버튼, 체크박스, 메뉴 등과 같은 GUI 컴포넌트라고 불리는 요소들로 구성된다. 이러한 컴포넌트를 컨트롤(control) 또는 위젯(widget)이라고도 부른다. 사용자는 이런 GUI 컴포넌트를 통해 프로그램과 정보를 주고받게 된다.

응용프로그램에서 이러한 GUI를 효과적으로 사용하기 위해서는 언어 자체 또는 별도의 형태로 GUI 프레임워크를 제공해야 한다. GUI 프레임워크는 사용자가 단순히 사용하는 라이브러리 클래스를 넘어, GUI 컴포넌트의 디스플레이, 사용자의 입력과 입력 정보(이벤트)의 전달, 입력 정보에 따른 적절한 객체의 실행 등 총괄적 업무를 처리해 프로그래머의 수고를 최대한 줄여 준다.

Java 언어는 여러 가지 GUI 프레임워크를 제공하는데 초기에는 AWT(Abstract Window Toolkit)이라고 불리는 프레임워크를 주로 사용하였다. AWT에서는 GUI 컴포넌트를 구현함에 있어 운영체제의 네이티브 컴포넌트(peer component라고 부름)를 이용한다. 이렇게 네이티브 컴포넌트를 이용하여 구현한 Java 컴포넌트를 중량 컴포넌트(heavy-weight component)라고 부른다. 중량 컴포넌트는 사용자 인터페이스가 플랫폼에 따라 다른 룩앤필(look-and-feel)을 제공하여 일관성이 결여된다. 이러한 문제점을 개선하기 위해 Java 1.2부터 Swing이라고 불리는 새로운 GUI 프레임워크를 제공하여 왔다. Swing에서 GUI 컴포넌트는 일부를 제외하고 모두 Java 언어로 구현되었다. 이러한 GUI 컴포넌트를 경량 컴포넌트(light-weight component)라고 부른다. 경량 컴포넌트는 플랫폼과 독립적으로 구현되어 있어 실행환경과는 무관하게 일관성이 있는 사용자 인터페이스를 제공할 수 있다. 필요하다면 다른 룩앤필을 설정하여 사용할 수 있도록 플러그형 룩앤필(pluggable look-and-feel) 기능을 제공한다.

앞에서 언급했듯이 GUI 프레임워크는 단순히 GUI 컴포넌트 클래스들만으로 이루어진 것이 아니다. GUI 컴포넌트에 주어진 입력(이벤트라고 부름)을 처리하기 위한 이벤트 모델과 화면상에 GUI 컴포넌트의 배치 등의 개념을 지원해야 하고, 또한 선이나 사각형과 같은 도형을 직접 그릴 수 있는 그래

픽 기능도 제공해야 한다. 다음 절들에서 이러한 구성 요소에 대해 설명한다.

10.2 이벤트 모델

A. 개념 정리

1) 이벤트 모델의 구성요소

Java GUI 프레임워크는 사용자의 입력을 이벤트 형식으로 받아들여 처리한다. 이러한 프로그램의 실행구조를 이벤트 구동 모델(event-driven model) 방식이라고 한다. AWT 뿐만 아니라 Swing 등 대부분의 GUI 프레임워크는 이러한 이벤트 구동 방식을 사용하고 있다. 이벤트 구동 방식의 주요 요소는 다음과 같다.

- 이벤트 소스(Event Source) - 이벤트를 발생시키는 GUI 컴포넌트를 말한다. 컴포넌트를 통해 사용자의 입력이나 컴포넌트의 상태 변경에 따라 이벤트가 발생한다.
- 이벤트 객체(Event Object) - 이벤트 정보를 포함하는 객체를 말한다. 이벤트 유형마다 클래스로 만들어져 있으며 이벤트가 발생할 때마다 해당 유형의 이벤트 객체가 생성되어 내부에 저장되어 있다가 처리할 차례가 되면 이벤트 리스너에게 전달된다.
- 이벤트 리스너(Event Listener) - 이벤트를 처리하는 코드를 말한다. 이벤트 종류마다 이벤트 리스너 인터페이스가 정의되어 있으며, 프로그래머가 이벤트를 처리할 이벤트 리스너를 구현해야 한다.

2) 이벤트 처리 방식

이러한 세 가지 요소를 이용하여 이벤트를 처리(event handling)하는 방식을 설명하기로 한다. GUI가 없는 프로그램(non-GUI program)에서는 사용자가 작성한 함수는 일반적으로 사용자 코드에 의해 호출된다. GUI 프로그램에서는 비-GUI 프로그램과는 달리 이벤트를 처리하는 사용자 코드(이벤트 리스너)를 사용자 코드가 직접 호출하지 않는다. 대신 GUI 프레임워크 내부 스레드에서 호출한다. 이러한 방식을 콜백(callback) 방식이라고 한다. 콜백 방식의 이벤트 처리를 위해 프로그래머는 먼저 이벤트 소스(GUI 컴포넌트)의 이벤트 유형에 알맞은 이벤트 리스너를 구현해야 한다.

아래는 이벤트 처리 모델을 순서화 한 것이다.

1. 이벤트 리스너를 생성하여 이벤트 소스에 등록한다.
2. 이벤트 소스에서 이벤트를 발생시킨다.
3. 등록된 이벤트 리스너를 실행시킨다.

버튼(button)의 예를 들어 이벤트 처리 모델을 살펴보자. 버튼을 클릭한 경우 ActionEvent라는 유형의 이벤트가 발생한다. 이러한 유형의 이벤트를 처리하는 이벤트 리스너는 ActionListener라는 인터페이스이다. 아래 ButtonEventHandler 클래스는 ActionListener 인터페이스를 구현하여 버튼의 이벤트를 처리하는 이벤트 리스너의 모습이다.

```
class ButtonEventHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        // 이벤트를 처리할 실제 코드
    }
}
```

actionPerformed 메소드는 버튼에서 ActionEvent가 발생할 경우 호출되는 메소드이다. 이제 남은 일은 이벤트 리스너 객체를 생성하여 버튼에 등록하기만 하면 된다.

```
JButton b = new JButton("OK");
b.addActionListener(new ButtonEventHandler());
```

이벤트의 발생은 GUI 컴포넌트에서 발생하지만 이벤트 핸들링은 이벤트의 발생과 동시에 실행되지 않는다. 사용자 입력이나 프로그램 내부에서의 컴포넌트 상태 변경에 따라 이벤트가 동시다발적으로 발생할 수 있지만 이들은 이벤트 큐라는 곳에 저장된 후 순차적으로 실행된다. 이렇게 이벤트를 순차적으로 실행시키는 별도의 스레드를 이벤트 디스패치 스레드(Event Dispatch Thread, EDT)라고 부른다. 즉, 위 코드에서 actionPerformed 메소드는 사용자 스레드가 호출하지 않고 EDT가 호출한다.

3) 이벤트 및 이벤트 리스너 종류

GUI 프로그램을 작성하기 위해서 프로그래머는 어떤 GUI 컴포넌트에서 어떤 이벤트가 발생할 수 있는지 알아야 하며 이벤트 유형별 이벤트 리스너 종류를 알아야 한다. 앞에서 언급한 ActionEvent는 JButton 외에도 JTextField, JCheckBox, JMenuItem 등에서도 발생한다. 따라서 이러한 컴포넌트의 이벤트 리스너는 동일한 ActionListener 인터페이스를 이용하여 정의할 수 있다. ActionEvent 외에 대표적인 이벤트로 ItemEvent가 있다. 이 유형의 이벤트는 컴포넌트에서 아이템(item)을 선택하여 상태가 변경되는 경우에 주로 발생한다. ItemEvent를 발생시킬 수 있는 컴포넌트는 JCheckBox, JComboBox, JRadioButton, JList 등이 있다. ItemEvent를 담당하는 이벤트

리스너는 ItemListener이다. 다음은 여러 체크박스의 체크를 활성화할 때 어떤 체크박스가 활성화되었는지를 확인하는 ItemListener 구현의 예를 보여준다.

```
class CheckBoxEventHandler implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
        if (e.getStateChange() == ItemEvent.SELECTED) {
            System.out.println(e.getItem() + " is selected.");
        }
    }
}
```

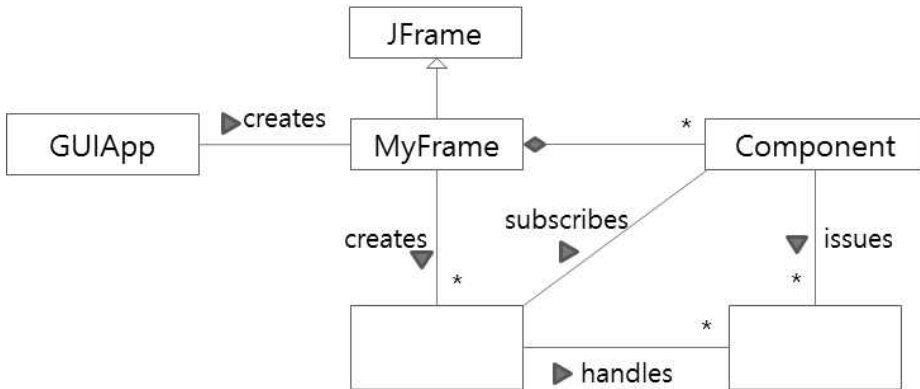
이벤트 리스너를 구현할 때 이벤트 소스나 이벤트 객체로부터 필요한 정보를 가져올 수 있다. 가령, 위 CheckBoxEventHandler에서 ItemEvent e로부터 체크박스의 상태정보(SELECTED 혹은 DESELECTED)를 확인할 수 있다.

한 종류의 컴포넌트에서 여러 가지 이벤트 유형이 발생할 수도 있다. 예를 들어, JCheckBox에서 체크를 활성화하거나 해제할 때 ActionEvent도 발생하지만 ItemEvent도 발생한다. ActionEvent는 사용자의 클릭에 의해서만 발생하지만 ItemEvent는 사용자에 의한 상태 변경 외에도 프로그램 내에서 체크박스의 상태를 변경하는 경우에도 발생한다. 따라서 대부분의 경우 두 가지 중 어떤 이벤트를 이용하여 처리해도 상관없지만 때에 따라 적절한 이벤트를 선택해야 할 때도 있다.

이외에도 AdjustmentEvent, KeyEvent, MouseEvent 등 다양한 이벤트 종류가 있지만 여기서는 설명을 생략하기로 한다.

B. 개념 확인

1. 아래 그림은 Java Swing을 이용한 응용프로그램의 전형적인 구조이다. 이벤트 모델에 기반하여 빈 곳을 알맞게 채우시오.



2. 다음 코드는 OK 또는 Cancel이라는 버튼을 누르면 어떤 액션을 취하는 프로그램이다(프레임을 생성하고 속성을 설정하는 main 코드는 생략). 다음 물음에 답하시오.

```

class ButtonFrame extends JFrame {
    private JButton ok;
    private JButton cancel;

    public ButtonFrame() {
        setLayout(new FlowLayout());
        ok = new JButton("OK");
        cancel = new JButton("Cancel");

        ActionListener handler = new ButtonHandler();

        (A)

        add(ok);
        add(cancel);
    }
    private class ButtonHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(SimpleFrame.this, String.format(
                "You pressed: " + e.getActionCommand()));
        }
    }
}
  
```

- 1) 버튼을 눌렀을 때 ButtonHandler가 작동하도록 밑 줄 친 (A)를 적절하게 채우시오.
 - 2) 위 코드에서 ButtonHandler 클래스를 없애고 동일한 역할을 하도록 빈 곳 (A)를 람다식을 사용하여 채우시오.
 - 3) 위 프로그램을 완성하여 수행해 본 후 어떤 일을 하는 프로그램인지 설명하시오.
 - 4) 1번 문제의 UML 클래스 다이어그램을 수정하여 이 문제를 위한 클래스 다이어그램을 그리시오.
3. 이벤트 구동 방식의 실행구조와 기존의 프로시저 호출에 의한 실행구조의 차이점을 다음 예제를 이용하여 설명하시오.
- 프로시저 호출 방식: 사용자에게 화면에 메뉴를 보여준 후 특정 메뉴 값을 키보드로 입력하도록 하여 그 값에 따라 특정 작업을 수행하도록 한다.
 - 이벤트 구동 방식: 사용자에게 선택 가능한 버튼을 화면에 제공하여 사용자가 버튼을 클릭함으로써 해당 작업을 수행하도록 한다.
4. 다음 프로그램은 체크박스를 클릭할 때 발생하는 ActionEvent를 확인해보는 코드이다. 물음에 답하시오.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CheckBoxFrame extends JFrame {

    private JCheckBox check;

    public CheckBoxFrame() {
        setLayout(new FlowLayout());

        check = new JCheckBox("Test");

        ActionListener actionHandler = new CheckBoxActionHandler();
        check.addActionListener(actionHandler);

        add(check);
    }

    public static void main(String[] args) {
        CheckBoxFrame f = new CheckBoxFrame();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

```
f.setSize(300, 200);
f.setVisible(true);

}

private class CheckBoxActionHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if (check.isSelected()) {
            System.out.println("CheckBox is selected!");
        }
        else {
            System.out.println("CheckBox is deselected!");
        }
    }
}
}
```

- 1) ActionEvent 대신에 ItemEvent를 이용하여 동일한 작업을 하도록 코드를 수정하시오.
- 2) ActionListener와 ItemListener를 모두 사용하여 실행해보시오. 어떤 현상이 생기는지 설명하시오.
- 3) CheckBoxFrame 클래스의 생성자(constructor)의 끝부분 다음과 같은 코드를 넣어 실행해보시오. 어떤 현상이 생기는지 설명하시오.

```
check.setSelected(true);
```


10.3 레이아웃 매니저

A. 개념 정리

화면의 구성과 설계는 GUI를 가진 응용프로그램을 작성할 때 중요한 부분 중 하나이다. 필요한 컴포넌트를 선택한 후 이들을 화면상에 어떤 크기로 어디에 배치해야 할지를 정해야 한다. GUI 컴포넌트를 화면상에 배치하는 방법으로는 크게 세 가지가 있다. 하나는 프로그래머가 코드상에서 일일이 크기와 위치를 지정하는 방법이고, 다른 하나는 별도의 디자인 도구(GUI Designer for IntelliJ IDEA, NetBeans GUI Builder 등)를 이용하여 끌어놓기(drag-n-drop) 방식으로 컴포넌트를 배치하는 방법이다. 마지막 하나는 Swing 프레임워크에서 제공하는 레이아웃 매니저(Layout Manager)를 이용하는 것이다. 하나의 레이아웃 매니저는 자신의 고유한 배치방식을 가지고 있는데, 이 방법은 앞의 두 방법에 비해 정교하지는 않지만 화면을 빨리 설계하여 프로그램의 프로토타입을 만드는데 적합한 방법이다. 본 절에서는 레이아웃 매니저에 대하여 알아보기로 한다.

다음은 Swing이 제공하는 대표적인 레이아웃 매니저들이다.

- BorderLayout
- Flow Layout
- Grid Layout
- CardLayout
- SpringLayout
- GridBag Layout
- 등등

예제를 통해 몇 가지 레이아웃에 대하여 살펴보기로 한다.

BorderLayout은 JFrame의 기본 레이아웃이다. 다음 코드는 BorderLayout을 이용한 예제이다.

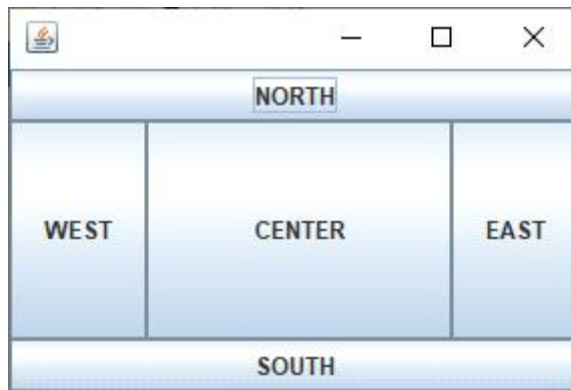
```
import java.awt.*;
import javax.swing.*;
public class LayoutTestFrame extends JFrame {
    public LayoutTestFrame() {
        JButton east = new JButton("EAST");
```

```

        JButton west = new JButton("WEST");
        JButton south = new JButton("SOUTH");
        JButton north = new JButton("NORTH");
        JButton center = new JButton("CENTER");
        add(east, BorderLayout.EAST);
        add(west, BorderLayout.WEST);
        add(south, BorderLayout.SOUTH);
        add(north, BorderLayout.NORTH);
        add(center, BorderLayout.CENTER);
    }
    public static void main(String[] args) {
        LayoutTestFrame f = new LayoutTestFrame();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(300, 200);
        f.setVisible(true);
    }
}

```

위 프로그램을 실행시켜보면 다음과 같이 5개의 버튼이 동, 서, 남, 북, 중앙에 자동으로 배치되는 것을 볼 수 있다.



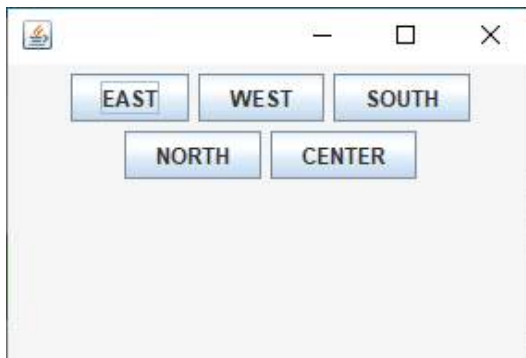
위 예제에서 프레임을 BorderLayout을 이용하도록 변경해보자. 나머지는 비슷하므로 여기서는 생성자 부분만 보여준다. 맨 첫 줄에서 프레임의 레이아웃 매니저를 BorderLayout에서 FlowLayout으로 변경하였다.

```

public LayoutTestFrame() {
    setLayout(new FlowLayout()); // layout 변경
    JButton east = new JButton("EAST");
    JButton west = new JButton("WEST");
    JButton south = new JButton("SOUTH");
    JButton north = new JButton("NORTH");
    JButton center = new JButton("CENTER");
    add(east);
    add(west);
    add(south);
    add(north);
    add(center);
}

```

위 코드를 실행시켜보자. 각 버튼들이 추가하는 순서대로 맨 위쪽 줄의 왼쪽에서부터 오른쪽으로 순으로 배치된다. 화면의 오른쪽 끝을 만나면 화면 아래로 진행한다. 화면의 크기를 조정하면 버튼들의 배치도 화면 크기에 따라 조정된다. FlowLayout은 JPanel의 기본 레이아웃이다.



GridLayout은 격자구조 형식으로 컴포넌트를 배치하는데 적합하며, CardLayout은 카드 스택에서 하나의 카드를 빼내 볼 수 있듯이 여러 개의 컴포넌트를 겹쳐 놓은 채로 하나씩 꺼내어 보여주는 방식이다. SpringLayout은 컴포넌트와 컴포넌트, 컴포넌트와 컨테이너 경계 사이에 Spring을 사용하여 컴포넌트의 상대적인 크기와 위치를 유연하게 정할 수 있는 레이아웃 매니저이다. 마지막으로 GridBagLayout은 GridLayout에 기반하고 있지만 모든 셀의 크기가 동일하게 설정하는 GridLayout과는 다르게 확장(span)과 같은 개념을 이용하여 컴포넌트의 크기를 다양한 제한조건으로 설정하여 보다 복잡한 레이아웃을 설정할 수 있다는 점에서 크게 다르다. 여기서는 자세한 설명을 생략한다.

B. 개념 확인

1. 본문의 FlowLayout을 이용하여 5개의 버튼을 배치하는 코드에서 5개의 버튼을 프레임에 직접 넣지 않고 JPanel을 하나 만들어 이 패널에 추가한 후 이 패널을 프레임에 추가하여 동일한 효과를 내도록 프로그램을 수정하시오.
2. 다음 코드는 White와 Black이라는 버튼을 누르면 panel1의 배경색을 버튼 명령어의 색으로 변경하는 프로그램이다. 다음 물음에 답하시오. 단, Frame을 생성하고 속성을 설정하는 main() 코드는 생략함.

```
class BorderLayoutFrame extends JFrame {
    JButton white, black;
    JPanel panel1;

    public BorderLayoutFrame() {
        white = new JButton("White");
        black = new JButton("Black");
        Panel panel2 = new Panel();
        panel2.add(white);
        panel2.add(black);
        panel1 = new JPanel();

        ActionListener listener = new ButtonHandler();

        (A)

        panel1.setBackground(Color.WHITE);
        add(panel2, BorderLayout.SOUTH);
        add(panel1, BorderLayout.CENTER);
        setSize(300, 200);
    }
    private class ButtonHandler implements ActionListener {

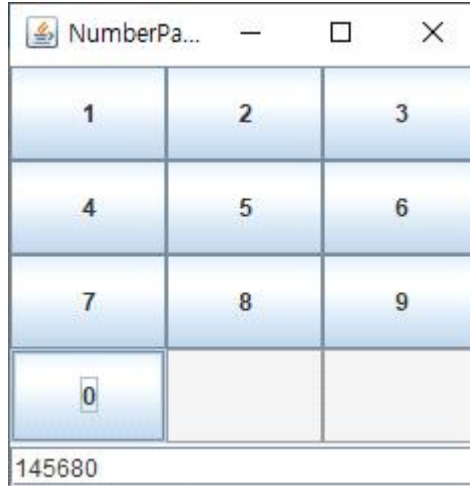
        (B)

    }
}
```

- 1) BorderLayoutFrame의 화면구성을 그리시오.
- 2) (B) 부분에 알맞은 코드를 채우시오.
- 3) 위 코드의 (A) 부분을 채우시오.

C. 응용 문제

- 아래 화면과 같이 번호 버튼을 누르면 하단의 텍스트 필드에 순서대로 번호 값이 입력되는 프로그램을 작성하시오. 상단의 번호 패널은 GridLayout을 사용하고 전체 프레임은 BorderLayout을 이용한다. 번호 패널의 빈 곳은 클릭이 되지 않아야 한다.



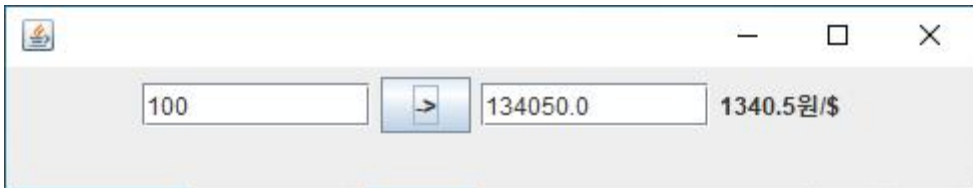
10.4 Swing 컴포넌트

A. 개념 정리

Swing 프레임워크는 GUI를 구성하기 위한 수많은 컴포넌트를 제공한다. 기본적인 컴포넌트로 JButton, JLabel, JTextField, JTextArea, JCheckBox, JRadioButton, JComboBox, JList이 있으며, 좀 더 복잡한 기능을 가진 컴포넌트로 JSlider, JTable, JMenu, JDialogBox 등이 있다. 다른 컴포넌트를 포함하기 위한 컨테이너(Container)로는 JFrame, JPanel 등이 있다. GUI 구성이 컴포넌트를 활용하기 위해서는 각 컴포넌트가 가지는 속성(property)과 이들 속성을 처리하는 메소드 등을 알아야 한다. 또한 컴포넌트는 이벤트 소스로서의 역할을 하기 때문에 각 컴포넌트가 어떤 이벤트를 발생시킬 수 있는지 확인이 필요하다. 여기서는 자세한 설명은 생략한다.

B. 개념 확인

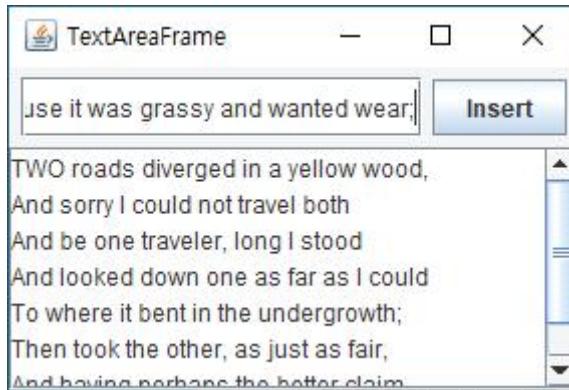
1. 아래 화면에서 첫 번째와 세 번째 컴포넌트는 텍스트 필드(text field)이고 두 번째 컴포넌트는 버튼(->)이며 마지막은 레이블(label)로 환율을 표시한다. 왼쪽 텍스트 필드에 달러(\$) 금액을 입력한 후 버튼을 누르면 오른쪽 텍스트 필드에 환율을 곱한 원화 금액을 표시하는 프로그램을 작성하시오. 단, 텍스트 필드의 초기값은 모두 0으로 설정하고 크기는 10으로 한다. 환율 값은 각자 적당하게 설정한다.



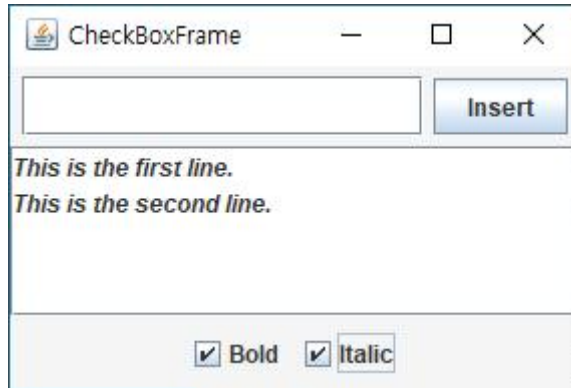
2. 아래 그림에서처럼 화면 아랫부분의 텍스트 필드에 입력된 폰트(font) 이름으로 화면 중앙의 텍스트의 폰트를 설정하는 프로그램을 작성하시오. (Times, SansSerif, 고딕, 굴림과 같은 폰트 이름을 사용해 볼 것)



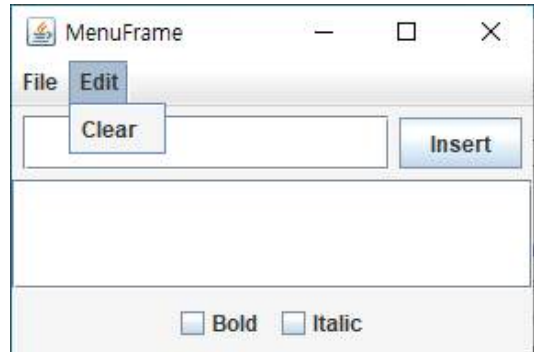
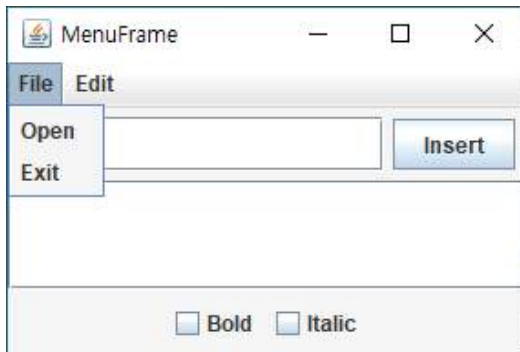
3. 다음 그림과 같이 상단에 텍스트 필드와 Insert 버튼이 있고 하단에 텍스트 영역(text area)가 있다. 텍스트 필드에 텍스트를 입력한 후 Insert 버튼을 누르면 입력한 내용이 텍스트 영역에 새로운 라인으로 추가된다. 내용의 길이가 텍스트 영역의 범위를 벗어나면 텍스트 영역의 오른쪽이나 하단에 스크롤 바(scroll bar)가 생긴다. 단, 텍스트 영역은 직접 편집할 수 없다. 힌트) 텍스트 필드와 버튼을 하나의 패널로 묶어 프레임의 NORTH에 배치한다.



4. 3번 문제의 화면 하단에 아래 check box 두 개를 추가하여 각 체크박스(check box)를 클릭할 때마다 텍스트 영역의 내용의 스타일을 체크박스에 명시된 스타일로 설정한다.



5. 4번의 문제에 다음과 같이 메뉴를 추가한다. File 메뉴의 Open 명령은 선택된 텍스트 파일의 내용을 텍스트 영역에 보여주는 명령이다. 텍스트 영역에 내용이 있다면 삭제한 뒤 보여준다. Open 명령을 선택하면 파일선택 다이얼로그 박스(file chooser dialog box)를 이용하여 파일 이름을 입력하거나 파일을 선택한다. Exit 명령을 선택하여 프로그램을 종료한다. Edit 메뉴의 Clear 명령을 선택하면 텍스트 영역의 내용을 모두 지운다. Clear 명령시 내용을 지우기 전에 다이얼로그 박스를 이용하여 지울 것인지 확인을 한 후 지운다.



10.5 그래픽 기초

A. 개념 정리

앞 절에서는 GUI를 만들 수 있도록 미리 만들어진 GUI 컴포넌트의 사용법에 대하여 살펴보았다. 이 절에서는 선이나 사각형과 같은 도형을 다루는 그래픽 기초에 대하여 소개한다. 화면에 그림을 그릴 때 일반적으로 컴포넌트에 그린 후 컴포넌트를 프레임에 배치한다. 아래 템플릿 코드에서처럼 JComponent 클래스를 상속한 클래스를 정의하고 그 클래스의 paintComponent() 메소드에서 필요한 그림을 그리면 된다. paintComponent() 메소드는 JComponent에 정의된 것으로 오버라이드(override)하여 정의하면 된다.

```
class DrawingComponent extends JComponent {
    @Override
    public void paintComponent(Graphics g) {
        g.drawRect(200, 150, 20, 20);
    }
}
class DrawFrame extends JFrame {
    add(new DrawingComponent());
}
```

paintComponent() 메소드의 매개변수인 Graphics 객체는 Swing 렌더링 시스템 내부에서 생성하여 제공하는 객체로 그림을 그릴 때 필요한 정보 (가령 텍스트의 폰트의 종류, 크기, 색깔이나 선의 굵기, 색상 등)를 가진 캔버스라고 간주하면 된다. 뭔가를 그릴 때 이 객체를 통해 그린다.

paintComponent() 메소드는 사용자 코드에서 호출하는 메소드가 아니라 그림이 그려져야 하는 상황이 발생하면 자동으로 호출되는 메소드이다. 그림이 그려져야 하는 상황에는 다음과 같은 경우가 있다.

- 컴포넌트가 처음 화면에 나타날 때
- 컴포넌트의 사이즈가 변경될 때
- 컴포넌트를 가리고 있던 다른 컴포넌트(가령, 다른 프레임)가 이동할 때
- 프로그램에서 명시적으로 요청할 때

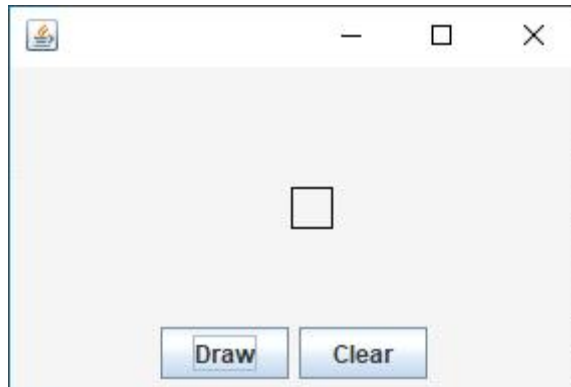
프로그램에서 컴포넌트를 다시 그리기를 직접 요청해야 할 경우에는 `repaint()` 메소드를 호출하면 된다.

Graphics 객체는 정수 좌표만을 지원한다. 실수 좌표를 사용하는 `Rectangle2D`와 같이 그림을 보다 정교하게 그리기를 원할 때에는 `Graphics2D`를 사용하면 된다. 아래와 같이 `Graphics` 객체를 형변환 시킨 후 `Graphics2D`의 보다 다양한 메소드를 사용할 수 있다.

```
Graphics2D g2 = (Graphics2D) g;  
g2.draw(new Rectangle2D.Double(200, 150, 10, 10));
```

B. 개념 확인

1. 아래 화면과 같이 화면의 하단 부분의 Draw 버튼을 누르면 화면의 중앙 부분에 사각형이 하나 그려지고 Clear 버튼을 누르면 그 사각형이 사라지는 프로그램을 작성하시오.



11장 파일 입출력 스트림

11.1 파일 입출력 스트림

11.2 바이트 기반 입출력 스트림

11.3 문자 기반 입출력 스트림

11.4 기본형 데이터 이진형식 입출력

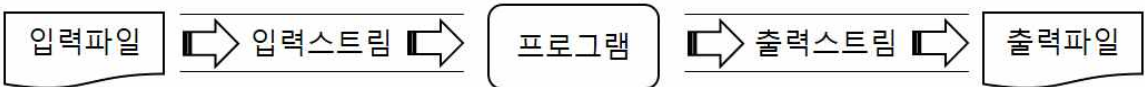
11.5 랜덤 액세스 파일

11장 파일 입출력 스트림

11.1 파일 입출력 스트림

A. 개념 정리

Java 언어에서 파일은 연속적인 일련의 바이트로 이루어진다. 파일은 입력 공간(source) 또는 출력 공간(destination)으로 활용된다. Java에서 파일로부터의 입력이나 파일로의 출력을 하나의 스트림(stream)이라는 개념으로 취급한다. 파일뿐만 아니라 바이트 시퀀스를 가지는 스트링이나 메모리 블록(배열)도 이러한 스트림으로 다룰 수 있다. 바이트를 연속적으로 읽어 들이는 스트림을 입력 스트림(Input Stream)이라고 하고 출력하는 스트림을 출력 스트림(Output Stream)이라고 부른다.



입출력 스트림은 데이터의 형식에 따라 다음 두 가지로 나뉜다.

- 바이트 기반 입출력 스트림 (byte-based IO streams)
- 문자 기반 입출력 스트림 (character-based IO streams)

바이트 기반 입출력 스트림은 입출력의 기본 단위가 바이트로 주로 바이너리 데이터의 입출력에 사용한다. 문자 기반의 입출력 스트림은 이러한 바이트 기반 입출력 스트림 상에서 문자 단위로 입출력을 할 수 있도록 지원하는 스트림이다. 주로 텍스트를 입출력할 때 사용한다. 문자는 기본적으로는 일련의 바이트들로 이루어지기 때문에 문자를 제대로 인식하려면 문자의 인코딩 체계(encoding scheme)를 정확하게 사용해야 한다.

11.2 바이트 기반 입출력 스트림

A. 개념 정리

Java는 바이트 기반 입출력 스트림을 지원하기 위해서 여러 가지 라이브러리 클래스들을 제공한다. 아래는 대표적인 입력 스트림과 출력 스트림 클래스들을 보여준다.

```
java.io.InputStream (abstract class)
    java.io.FileInputStream
    java.io.ObjectInputStream
    java.io.FilterInputSream
    java.io.DataInputStream

java.io.OutputStream (abstract class)
    java.io.FileOutputStream
    java.io.ObjectOutputStream
    java.io.FilterOutputStream
    java.io.DataOutputStream
    java.io.PrintStream
```

가장 기본적인 바이트 단위의 파일 입력 스트림인 `FileInputStream` 클래스를 이용하여 "in.dat" 파일에서 전체 데이터를 읽어 처리하는 코드를 살펴보자.

```
FileInputStream fis = new FileInputStream("in.dat");
int byteRead = fis.read();
while ((byteRead != -1) {
    // process byteRead
    byteRead = fis.read();
}
```

파일을 입력하기 위해서는 우선 해당 파일을 위한 입력 스트림을 생성해야 한다(위 코드의 첫 번째 문장). 다음으로 입력 스트림으로부터 데이터를 읽는 메소드를 적용한다(위 코드의 `fis.read()` 부분). 출력 스트림의 사용도 이와 유사하다. 여기서 각 입출력 스트림이나 메소드에 대해 자세하게 설명하지 않는다. 각자 매뉴얼을 참고하도록 한다.

여기서 유의할 점은 하나의 바이트를 읽어 저장하는 변수를 `byte` 형이 아니라 `int` 형으로 선언한다는 것이다. 하나의 바이트 값은 0~255의 값을 가지는데 `byte` 타입은 -128~127까지의 값을 가진다.

또한 Java에서는 -1을 EOF(End of File)를 나타내는 값으로 사용하는데 byte 형으로 입력하면 255 값과 -1을 구분할 수 없게 된다. 이러한 이유로 하나의 바이트 값을 int 형 변수에 저장하는 것이다. 파일 입출력을 할 때 한 가지 주의해야 할 점은 반드시 IOException을 처리해야 한다는 점이다. 위 코드에서 사용한 FileInputStream 생성자는 IOException의 일종인 FileNotFoundException을 발생시킬 수 있다. 또한 read 메소드는 IOException을 발생시킬 수 있다. 이러한 IOException은 점검 예외(chekced exception)의 일종이므로 반드시 처리하거나 아니면 throws 명세를 통해 호출 메소드에게 전달해야 한다. 아래는 이 두 가지의 처리방식의 예를 보여준다. 우선 예외를 직접 처리하는 경우를 보자.

```
try {
    FileInputStream fis = new FileInputStream("in.dat");
    int byteRead = fis.read();
    while ((byteRead != -1) {
        // process byteRead
        byteRead = fis.read();
    }
}
catch (FileNotFoundException e) {
    System.out.println("File does not exist!");
    System.exit(-1);
} catch (IOException e) {
    System.out.println("Unexpected exceptions occurs in read()!");
    System.exit(-1);
}
```

다음은 예외를 직접 처리하지 않고 throws를 명시하는 예이다. 위 코드가 포함된 메소드가 아래 foo()라고 가정한다.

```
public void foo() throws IOException
{ ... }
```

이후부터는 코드를 간단하게 작성하기 위해 예외 처리 부분은 생략하기로 한다. 예외 처리에 대해서는 6장을 참고하길 바란다.

아래 코드는 FileOutputStream을 이용하여 바이트 배열의 데이터를 파일에 출력하는 예제이다.

```
FileOutputStream fos = new FileOutputStream("bin.dat");

byte[] byteArray = {35, 78, 100, 5, 120, 111, 127, 105};
for(byte b: byteArray) {
    fos.write(b);
}
```

B. 개념 확인

1. 다음은 주어진 파일 "in.dat"의 바이트 수를 계산하는 프로그램이다. 빈 곳을 적절하게 채우시오.

```
InputStream in = _____ ;
int n = 0;
int b = in.read();
while(b != -1) {
    n++;
    b = in.read();
}
System.out.println("no. of bytes : "+n);
in.close();
```

2. 1번 문제는 IOException이 발생할 수 있다. 물음에 답하시오. 필요하다면 클래스와 메소드를 정의하여 사용할 것.

- 1) 어떤 예외가 발생할 수 있는지 모두 밝히시오.
- 2) try-block을 이용하여 예외를 처리하시오.
- 3) throws 선언을 이용하여 예외를 처리하시오.

3. 1번 문제에서 하나의 byte를 읽은 메소드인 read의 반환값의 타입이 int이다. byte가 아니고 int인 이유는 무엇인가?

4. 1번 문제의 코드를 완성한 후 주어진 "in.dat"을 이용하여 실행해 보시오(BinaryInputTest.java). 이 파일의 속성정보를 이용하여 바이트 수를 실행 결과와 비교해보시오.

5. "in.dat" 파일을 삭제한 후 실행해보시오. 어떤 결과가 나오는가?

6. 본문에서 주어진 byte 형 배열을 "bin.dat" 파일에 출력하는 코드를 완성하여 실행해보시오. 4번

문제에서 구현한 프로그램에 "bin.dat"를 넣어 실행해 본 후 실행 결과의 바이트 수가 바이트 배열과 일치하는지 확인하시오.

C. 응용 문제

1. 개념 확인의 6번 문제에서 얻은 "bin.dat" 파일로부터 바이트 데이터를 읽어 들여 배열에 저장한 후 출력하는 프로그램을 작성하시오.

11.3 문자 기반 입출력 스트림

A. 개념 정리

다음은 문자 기반 입출력 스트림 클래스 계층을 보여준다.

```
java.io.Reader (abstract class)
    java.io.BufferedReader
    java.io.StringReader
    java.io.InputStreamReader
    java.io.FileReader

java.io.Writer (abstract class)
    java.io.BufferedWriter
    java.io.StringWriter
    java.io.PrintWriter
    java.io.OutputStreamWriter
    java.io.FileWriter
```

문자 단위로 읽어 들이는 대표적인 입력 스트림이 `FileReader`이다. 그런데 `FileReader` 클래스는 `InputStreamReader`의 서브클래스로 `InputStreamReader`의 `read` 메소드를 사용한다. `InputStreamReader` 클래스는 바이트 입력 스트림에서 문자 기반 입력 스트림으로의 변환을 담당하는 역할을 담당한다. 따라서 입력할 파일의 인코딩 체계를 지정하여 정확하게 변환을 할 수 있도록 해주어야 한다. 반면 `FileReader`는 기본 인코딩 체계(Java 18부터는 UTF-8, 그 이전 버전에서는 플랫폼에서 지정한 기본 인코딩 체계)를 사용한다. 다음은 `InputStreamReader`를 이용하여 utf-16 형식의 파일에서 문자를 읽어 들여 화면에 출력하는 코드이다.

```
InputStreamReader isr = new InputStreamReader(new FileInputStream("in.txt"), "utf-16");
int c;
while((c=isr.read())!=-1) {
    System.out.print((char) c);
}
```

위 예제와 같이 `InputStreamReader`를 생성할 때 입력의 기반이 되는 바이트 입력 스트림인 `FileInputStream` 객체를 이용한다. 이러한 형식의 클래스를 래퍼 클래스(wrapper class)라고 한다. 파일 입출력 스트림에서 이러한 래퍼 클래스 형식이 많이 사용되는 것을 볼 수 있다.

BufferedReader는 입력 스트림을 버퍼링하여 하나의 문자뿐만 아니라 라인 단위나 배열과 같은 단위 등으로 효율적인 입출력이 가능하게 하는 클래스다. 아래는 BufferedReader를 이용하여 in.txt 파일에서 문자들을 라인 단위로 읽어 들이는 예이다. BufferedReader 클래스는 다른 Reader 객체를 이용하여 객체를 생성하는 래퍼 클래스이다.

```
FileReader fileReader = new FileReader("in.txt");
BufferedReader bufferedReader = new BufferedReader(fileReader)
{
    String line;
    while ((line = bufferedReader.readLine()) != null) {
        System.out.println(line);
    }
}
System.out.println(line);
```

이러한 Reader 클래스들을 이용하여 문자와 수치정보 등이 포함된 파일로부터 원하는 데이터를 읽어 들이는 일은 매우 까다롭다. java.util.Scanner를 이용하면 다양한 형식의 텍스트 파일로부터 데이터를 보다 쉽게 읽을 수 있다. 또한 Scanner는 입력 파일로부터 구분자(delimiter)나 정규표현식(regular expression)을 이용하여 필요한 데이터만을 추출하여 읽어 들이는 것이 용이하다. 아래 코드는 문자열과 정수, 실수가 공백으로 구분된 파일에서 각각 데이터를 분리하여 차례대로 읽어 들이는 코드이다.

```
File f = new File("in.txt");
Scanner in = new Scanner(f, "utf-16");
String name = in.next();
int id = in.nextInt();
double salary = in.nextDouble();
```

아래 코드는 구분자로 분리된 문자열을 차례로 읽어 들여 출력하는 코드이다.

```
File f = new File("in.txt");
Scanner in = new Scanner(f, "utf-8");
in.useDelimiter("\\s*,\\s*");
while(in.hasNext()) {
    String s = in.next();
    System.out.println(s);
}
```

useDelimiter() 메소드는 문자열의 구분자를 명시하는 메소드로 구분자는 정규표현식을 이용하여 표현한다. 위 예는 하나의 콤마와 양쪽에 공백이 여러 개 포함되는 문자열을 구분자로 설정한 것이다. 여기서 자세한 정규표현식의 형식은 생략한다.

FileWriter 클래스 군을 이용하여 문자를 파일에 출력하는 방식은 Reader 클래스 군을 사용하는 것과 유사하다. OutputStreamWriter는 특정 인코딩을 가진 문자를 바이트 시퀀스로 변환하여 출력해주는 역할을 한다(Reader클래스 군의 InputStreamReader에 대응되는 클래스임). FileWriter와 OutputStreamWriter의 출력방식은 Reader 부분의 예제와 유사하므로 여기서는 생략하기로 한다.

Reader 클래스들과 마찬가지로 다양한 데이터를 원하는 형식으로 출력하는 것은 매우 번거롭다. 이를 도와주는 클래스가 바로 PrintWriter 클래스이다. PrintWriter 클래스는 문자열뿐만 아니라 기본 타입의 데이터들을 형식에 맞춰 출력하는 메소드를 제공한다. PrintWriter는 System.out 객체의 클래스인 PrintStream 클래스와 기능이 거의 동일하다.(PrintStream은 문자기반 바이트 기반 출력 스트림 군에 포함되어 있음). 대표적인 메소드에는 다음과 같은 것들이 있다.

- print() - 하나의 기본형 데이터 뿐만 아니라 문자열, 객체를 출력하는 메소드
- println() - 하나의 문자열을 출력하는 메소드
- printf() - 여러 데이터를 형식에 맞춰 문자열로 만들어 출력하는 메소드(C언어의 printf함수와 유사)

B. 개념 확인

1. 본문에서 주어진 예제에서 처럼 InputStreamReader를 이용하여 "in.txt" 파일을 문자 기반으로 읽어 들여 출력하는 프로그램을 완성하시오. 입력 데이터의 길이가 길명 한 라인에 30 문자씩 잘라 출력할 것.

2. 메소드 copy는 Scanner와 PrintWriter를 이용하여 텍스트 파일 (inFile)과 출력 파일 (outFile) 복사하는 메소드이다. 빈 곳 (A~C)을 적절하게 채우시오. 단, inFile은 UTF-16으로 만들어져 있어야 한다. 힌트) 프로그램으로 UTF-16형식의 파일을 만들 수도 있고 메모장에서 파일을 만들어 UTF-16형식으로 저장할 수 있다.

```
public static void copy(String inFile, String outFile) throws IOException {
    Scanner in = _____ (A) _____ ;
    PrintWriter out = _____ (B) _____ ;

    while(in.hasNextLine()) {
        _____ (C) _____
    }
    in.close();
    out.close();
}
```

3. copy() 메소드를 테스트할 수 있는 main() 메소드를 작성하시오. 아래와 같은 형식의 command로부터 inFile과 outFile을 읽어들인다. 아울러 copy() 메소드에서 처리하지 않은 IOException을 적절하게 처리하시오. 주의사항) in16.txt를 UTF-16 형식으로 만들어야 함.

```
java Copy in16.txt out8.txt
```

C. 응용 문제

아래 Account 클래스를 이용하여 물음에 답하시오.

```
class Account {
    private String name;
    private int number;
    private double balance;
    public Account(String n, int no, double b) {
        name = n;
        number = no;
        balance = b;
    }
    public String getName() {
        return name;
    }
    public int getNumber() {
        return number;
    }
    public double getBalance() {
```

```

        return balance;
    }
    public void setBalance(double b) {
        balance = b;
    }
}

```

1. 아래 main 메소드는 Account 객체 배열의 각 객체의 정보를 "accounts.txt" 파일에 저장하는 프로그램이다. (A) 부분을 적절하게 채우시오. 단, PrintWriter 객체를 이용할 것.

```

static void main(String[] args) {
    Account[] accts = new Account[3];
    accts[0] = new Account("Kim", 1111, 10000.0);
    accts[1] = new Account("Lee", 2222, 20000.0);
    accts[2] = new Account("Park", 3333, 30000.0);

    (A)

}

```

"accounts.txt" 파일은 아래와 같은 구조의 텍스트 파일로서, 첫 번째 라인은 Account 객체 정보의 수를 나타내며 이후 하나의 라인은 하나의 Account 객체의 데이터 정보를 가지고 있다. 파일은 UTF-8 형식으로 저장된다고 가정한다.

```

3
Kim 1111 10000.0
Lee 2222 20000.0
Park 3333 30000.0

```

2. 1번 문제에서 생성한 "accounts.txt" 파일로부터 Scanner 객체를 이용하여 데이터를 읽어들이고 Account 객체 배열을 만든 후, 각 Account 객체의 데이터를 화면에 출력하는 프로그램을 작성하시오. 출력 형식은 자유롭게 설계해도 좋다.

11.4 기본형 데이터 이진형식 입출력

A. 개념 정리

`DataInputStream` 및 `DataOutputStream`은 기본형 데이터를 이진 형식으로 입출력하는 입출력 스트림을 위한 클래스들이다. `int`형과 같은 기본형 데이터를 텍스트 형식으로 저장하면 그 길이가 숫자의 값에 따라 다르다(예, 123은 세 개의 문자로 이루어지지만 12345는 5개의 문자로 이루어지기 때문에 utf-16으로 저장하면 123는 6바이트, 12345는 10바이트가 됨). 하지만 이진 형식으로 저장하면 숫자의 값과는 무관하게 각 데이터 형(`int`형은 4바이트, `double`형은 8바이트)의 크기에 따라 결정된다. 따라서 데이터를 저장한 후 읽거나 네트워크를 통해 전송하는 경우 효율성도 높아지고 특정 데이터의 위치를 정확하게 파악할 수 있는 장점도 있다.

`DataInputStream`과 `DataOutputStream`은 이러한 기본형의 입출력을 위한 메소드를 정의하는 `DataInput`과 `DataOutput` 인터페이스를 각각 구현하고 있다. `DataInput`과 `DataOutput` 인터페이스의 주요 메소드는 각각 다음과 같다.

- `readByte()` / `writeByte()`
- `readChar()` / `writeChar()`
- `readInt()` / `writeInt()`
- `readLong()` / `writeLong()`
- `readDouble()` / `writeDouble()`
- `readFloat()` / `writeFloat()`
- `readLine()` / (대응되는 메소드 없음)
- etc.

`DataInput`과 `DataOutput` 인터페이스는 다음 절에서 설명한 랜덤 액세스 파일에서 더 유용하게 사용되기 때문에 다음 절에서 좀 더 살펴보기로 한다.

B. 응용 문제

1. 매개변수로 전달받은 double 형 배열의 원소를 Java의 기본 데이터 타입 (8바이트) 표현 방식으로 "numbers.dat" 파일에 연속적으로 저장하는 메소드(메소드 명: writeData)를 작성하시오. 파일의 맨 앞부분에 저장하는 원소의 개수를 int 형으로 저장한다. writeData 메소드를 테스트할 main 메소드를 적절하게 작성하여 실행해보시오.
2. 1번 문제에서 생성한 "numbers.dat" 파일에서 실수값을 읽어 들여 배열에 저장한 후 그 배열을 반환하는 메소드(메소드 명: readData)를 작성하고 1번 문제에서 작성한 main 메소드를 적절하게 수정하여 readData를 테스트해보시오.

11.5 랜덤 액세스 파일

A. 개념 정리

RandomAccessFile은 랜덤 액세스 파일의 입출력을 지원하는 클래스로 임의의 위치에 데이터를 읽고 쓸 수 있도록 한다. 파일 포인터(file pointer)라고 불리는 일종의 인덱스가 존재하여 현재 읽고 쓰는 위치(offset)를 가리키고 있다. RandomAccessFile의 객체를 생성할 때 입출력 모드를 제공할 수 있다.

- "r" : 입력만 지원
- "rw" : 입출력을 동시에 지원

파일 포인터를 조작하는 메소드로 다음과 같은 것들이 있다.

- getFilePointer() : 파일 포인터 값(offset)을 반환한다.
- seek() : 파일 포인터의 값을 설정한다. 즉, 위치를 이동시킨다.

RandomAccessFile은 DataInput과 DataOutput 인터페이스를 동시에 구현하고 있어 두 인터페이스의 메소드를 모두 이용할 수 있다.

RandomAccessFile을 이용하여 int형 배열이 저장한 랜덤 액세스 파일(integer.dat)의 예를 살펴보자.

우선, int형 배열을 순서대로 저장하는 코드이다.

```
int[] a = {10, 20, 30, 40, 50};
RandomAccessFile raf = new RandomAccess("integer.dat", "rw");

for(int v : a) {
    raf.writeInt(v);
}
```

정숫값 5개가 저장된 "integer.dat"에서 세 번째 정숫값(30)을 읽어 내보자. 아래 코드는 앞부분 코드에서 연결되는 코드이다.

```
raf.seek(raf.getFilePointer()-12); // or raf.seek(8)
int v = raf.readInt();
System.out.println(v);
```

앞에서 읽어낸 값에 5를 더한 값을 다시 그 자리에 저장해보자.

```
raf.seek(raf.getFilePointer()-4);
raf.writeInt(v+5);
```

B. 개념 확인

1. 다음 프로그램은 3개의 Employee 객체를 emp.dat 파일에 저장하는 프로그램이다. 각 객체의 instance variable들은 binary 형식으로 저장된다. 물음에 답하시오.

```
class Employee {
    private int id;
    private double salary;
    private char skill;
    public Employee(int n, double s, char k) {
        id = n;
        salary = s;
        skill = k;
    }
    public int getId() {
        return id;
    }
    public double getSalary() {
        return salary;
    }
    public char getSkill() {
        return skill;
    }
    public String toString() {
        return id + "" + salary + "" + skill;
    }
}

public class RandomAccessTest {
    public static void main(String[] args) throws IOException {
        emps[0] = new Employee(1111, 80000.0, 'A');
        emps[1] = new Employee(2222, 100000.0, 'B');
        emps[2] = new Employee(3333, 70000.0, 'C');

        DataOutputStream out = new DataOutputStream(new FileOutputStream("emp.dat"));
```

```

    {
        for (Employee e: emps) {
            out.writeInt(e.getId());
            out.writeDouble(e.getSalary());
            out.writeChar(e.getSkill());
        }
    }
    // (A)
}
}

```

- 1) emp.dat 파일에서 두 번째 Employee 객체의 salary에 5000을 더해 저장하는 코드를 main 메소드의 (A)부분에 추가하시오.
- 2) main 메소드 (A) 아래 부분에 emp.dat의 각 Employee 객체의 데이터를 순서대로 읽어 출력하는 코드를 추가하시오. 단, 하나의 라인에 하나의 Employee 객체의 데이터를 출력한다.

C. 응용 문제

아래 Account 클래스를 이용하여 물음에 답하시오.

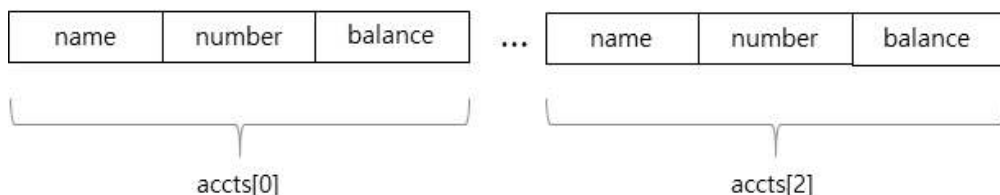
```

class Account {
    private String name;
    private int number;
    private double balance;
    public Account(String n, int no, double b) {
        name = n;
        number = no;
        balance = b;
    }
    public String getName() {
        return name;
    }
    public int getNumber() {
        return number;
    }
    public double getBalance() {
        return balance;
    }
    public void setBalance(double b) {
        balance = b;
    }
}

```

1. 아래 main 메소드는 Account 배열의 객체들을 아래 그림과 같이 연속적으로 "accounts.dat"

파일에 저장한다. 각 Account 객체의 속성 값들은 내부 바이너리 표현방식으로 저장된다. 저장된 객체의 수는 파일의 첫 부분에 int 형으로 저장된다. 단, name은 8자리 char로 저장된다. 8자리 초과하는 자리는 절단되며, 8자리 미만인 경우 name의 끝부분은 모두 '\0'으로 채운다.



다음 main 메소드를 보고 물음에 답하십시오. 단, IOException을 처리하는 코드를 추가한다.

```
public static void main(String[] args) {
    Account[] accts = new Account[3];
    accts[0] = new Account("Kim", 1111, 10000.0);
    accts[1] = new Account("Lee", 2222, 20000.0);
    accts[2] = new Account("Park", 3333, 30000.0);

    RandomAccessFile raf =
        new RandomAccessFile(new File("accounts.dat"), "rw");
    storeAccounts(raf, accts);
    updateAccount(2222, 500.0);
    readAccounts(raf, accts);
    for (int i = 0; i < accts.length; i++) {
        System.out.println(accts[i].getName() + ":" + accts[i].getNumber() + ":" +
            accts[i].getBalance());
    }
}
```

- 1) Account array를 매개변수로 받아 "accounts.dat"에 저장하는 storeAccounts 메소드를 작성하십시오.

```
tatic void storeAccounts(RandomAccessFile raf, Account[] accts) {
}
```

- 2) "accounts.dat" 파일에서 계좌번호 number를 가진 Account 객체의 balance를 amt만큼 증액시키는 작업을 하는 updateAccount 메소드를 작성하시오.

```
static void updateAccount(RandomAccessFile raf, int number, double amt) {
}

```

- 3) "accounts.dat" 파일에서 모든 계좌정보를 읽어 Account 배열에 저장한 후 반환하는 readAccounts 메소드를 작성하시오.

```
static Account[] storeAccounts(RandomAccessFile raf) {
}

```

2. 문제 1번에서 만든 accounts.dat를 이용하여 다음과 같은 문제를 해결하고 한다. 하루 동안 각 account에 발생한 트랜잭션(deposit, withdraw 등)은 하나의 텍스트 파일(trans.txt)에 저장된다. 하나의 트랜잭션은 account number와 입금한 금액 또는 인출한 금액으로 구성된다.

```
4
1111 500 // 1111 account에 500 입금
2222 -1000 // 2222 account에 1000 인출
1111 1000
3333 -3000

```

trans.txt의 각 라인을 읽어 이를 accounts.dat에 반영하도록 하는 프로그램을 작성하시오. 4번에서 작성한 updateAccount 메소드를 이용해도 좋다.

3. RandomAccessFile을 이용하여 int 형 배열을 (integer.dat)에 저장한 후 역순으로 읽어 출력하는 메소드 printReverse() 를 작성해보자. 빈 곳을 채우시오.

```
public static void printReverse(int[] a) throws IOException {
// 빈 곳
}

```

12장 스레드

12.1 Thread 클래스

12.2 Runnable 인터페이스와 상태

12.3 wait & notify

12장 스레드

12.1 Thread 클래스

A. 개념 정리

멀티 태스킹이란 여러 프로그램(작업)을 한번에 실행할 수 있는 기능을 의미한다. 예를 들어 웹에서 이메일을 편집하면서 동시에 파일을 다운로드하는 것이다. 이 중에서 멀티프로세스는 변수나 메모리를 온전히 각기 독립적으로 가지는 다수의 프로세스들이 동시에 실행되는 것을 의미한다. 멀티 스레딩은 멀티 프로세싱과 달리 작업들간의 일부의 데이터를 공유할 수 있는 형태를 의미한다. 스레드 간의 통신 비용은 프로세스 간 통신보다 적게 들고, 생성의 부담도 스레드가 더 낮다는 면에서 장점이 있다. 반면 안전성, 교착상태 등의 위험성이 존재한다.

Java의 클래스 Thread는 스레드를 생성하고 운영할 수 있도록 해준다. 방법은 Thread의 하위클래스를 만들고 Thread의 메소드인 run과 start를 이용하는 것이다. 메소드 run은 하위클래스에서 오버라이드가 필요하며, 멀티 스레드로 실행하기 원하는 내용을 담는다. 메소드 start는 실제로 새로운 스레드를 생성하는 메소드이다. 즉, Thread의 하위 클래스에 대해 객체를 만든 후 메소드 run을 적절히 채우고 메소드 start를 호출하면 멀티스레드의 실행이 가능하다. 그 밖에도 메소드 sleep 은 현재 스레드에 대해 인자값 만큼의 millisecond 동안 실행 중지한다.

다음은 클래스 PingPong 이 생성자의 인자로 받은 word와 delay 값을 활용하여, 특정 문자열을 출력하고 잠시 중지하는 것을 반복하는 것을 멀티 스레드로 실행하는 예이다.

```
class PingPong extends Thread {
    String word; // what word to print
    int delay; // how long to pause

    PingPong(String whatToSay, int delayTime) {
        word = whatToSay;
        delay = delayTime;
    }

    public void run() {
        try {
            for (;;) {
                System.out.print(word + " ");
                sleep(delay); // wait until next time
            }
        } catch (InterruptedException e) {}
    }
}
```

```

    }
    } catch (InterruptedException e) {
        return; // end this thread
    }
}

public static void main(String[] args) {
    new PingPong("ping", 33).start(); // 33 msec
    new PingPong("PONG", 100).start(); // 100 msec
}
}

```

B. 개념 확인

1. 멀티 스레딩과 싱글 스레딩의 특징을 서술하시오.
2. 멀티 스레딩과 멀티 프로세싱의 각 장점을 작성하고, 둘의 차이점을 서술하시오.
3. 5초마다 "Hello Thread"를 출력하는 스레드 클래스와 이를 실행하는 메인 함수를 작성하시오.
4. 스레드에서 run()과 start() 메소드를 활용하여 다음 코드를 실행해보고, 다른 출력을 보이는 이유를 설명하시오.

```

public class MyThread extends Thread {
    public void run() {
        System.out.println(this.currentThread().getName() + " 실행");
    }
    public static void main(String[] args) {
        MyThread thread0 = new MyThread();
        MyThread thread1 = new MyThread();
        MyThread thread2 = new MyThread();
        thread0.run();
        thread1.run();
        thread2.run();
        thread0.start();
        thread1.start();
        thread2.start();
    }
}

```


5. (Thread 간섭) 아래는 반복문을 포함한 Thread 2개가 포함된 코드이다. 실행 결과를 예상하고 근거를 제시하시오.

```
class PrintNumbers extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.print(i + " ");
        }
    }
}
public class ThreadExample {
    public static void main(String[] args) {
        PrintNumbers thread1 = new PrintNumbers();
        PrintNumbers thread2 = new PrintNumbers();
        thread1.start();
        thread2.start();
    }
}
```

6. 5번의 코드를 실행하고, 결과에 대해 해석하시오.

7. (Thread 간섭) 아래 코드는 하나 이상의 Thread가 하나의 변수에 관여하는 코드이다. n = 1일 때의 실행 결과를 예상하고 근거를 제시하시오.(Ref. Power JAVA, p597)

```
class Counter {
    private int value = 0;
    public void increment() {
        value++;
    }
    public void decrement() {
        value--;
    }
    public void printCounter() {
        System.out.println(value);
    }
}
class MyThread extends Thread {
    Counter sharedCounter;
    public MyThread(Counter c) {
        this.sharedCounter = c;
    }
    public void run() {
        int i = 0;
        while (i < 20000) {
```

```
        sharedCounter.increment();
        sharedCounter.decrement();
        if (i % 40 == 0)
            sharedCounter.printCounter();
        try {
            sleep((int)(Math.random() * 2));
        } catch (InterruptedException e) {}
    }
}
}
public class CounterTest {
    public static void main(String[] args) {
        Counter c = new Counter();
        int n = 1;
        // n 값을 바꾸어가며 실행해보시오.
        for (int i = 0; i < n; i++) {
            new MyThread(c).start();
        }
    }
}
```

8. 7번의 코드에 $n = 1$ 을 적용하여 실행하고, 결과에 대해 해석하시오.
9. 7번의 코드에 $n > 1$ 을 적용할 때 실행 결과를 예상하고 근거를 제시하시오.
10. 7번의 코드에 $n > 1$ 을 적용하여 실행하고, 결과에 대해 해석하시오.

12.2 Runnable 인터페이스와 상태

A. 개념 정리

GUI 나 사용자 입력에서 정보를 얻어서 처리하는 작업이 시간을 많이 소모한다면, 얻은 정보를 다른 스레드에 전달한 후 스레드를 start 시키는 방법을 사용한다.

그러나 Java는 다중 상속이 허용되지 않으므로 스레드를 상속하는 순간 해당 하위 클래스는 다른 상위클래스를 가질 수 없게 된다. 인터페이스 Runnable은 메소드 run을 오버라이드 하는 것 등 사용 방법은 클래스 Thread와 유사하지만 인터페이스를 구현하는 방식으로 이용하게 되므로 단일 상속의 한계에서 자유롭다.

스레드는 생성과 소멸 사이에 몇가지 상태에 놓이며 몇가지 이벤트나 메소드 호출에 의해 상태전이가 이루어진다. 아래의 그림 12.1을 보면 Thread의 하위클래스 객체를 키워드 new로 생성하는 경우, 스레드 1개가 만들어지며 아직 실행하지 않는 상태로 존재한다(New Thread 상태). 이후 메소드 start가 호출되면 실행가능한 상태가 되는데 운영체제 등 상황이 허락되면 바로 실행된다(Runnable 상태).

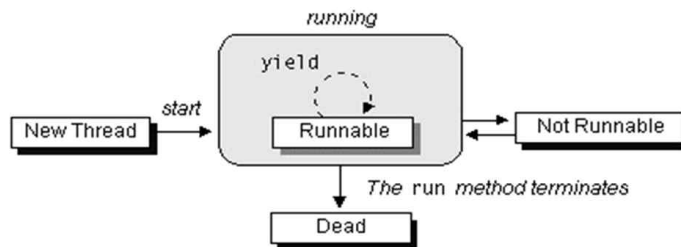


그림 12.1 스레드의 상태 전환

스레드의 실행은 메소드 run을 수행함으로써 진행되는데, 만일 run을 끝까지 실행하여 정상 또는 비정상 종료에 다다랐을 때는 자동적으로 스레드도 종료 된다(Dead 상태)

sleep 이 호출되거나 I/O 연산을 위해 잠시 CPU 실행을 중지하면 스레드의 실행이 중간에 멈추는 상황이 발생한다. 이 경우는 runnable 하지 않은 상태로서 block 되었다고 표현한다(Block 상태). 이 상태에서 빠져 나가는 방법은 각 원인에 따라 다르며, sleep에서 명시한 시간이 경과되었거나,

2) Core Java, Cay S. Horstmann and Gary Cornell, Oracle Press

I/O 연산이 완료되었을 때이다.

B. 개념 확인

1. Thread를 구현하는 방법은 Thread class 상속과 Runnable interface 상속으로 두 가지이다. 이 두 가지 방법의 특징과 차이점을 서술하시오.
2. 5초마다 "Hello Thread"를 출력하는 Thread 클래스를, Runnable Interface 상속 방법으로 작성하시오.
3. (Thread 상태) 다음 코드의 출력을 예상하시오.

```
public class ThreadStatus {
    public static class MyThread extends Thread{
        public void run() {
            int i = 0;
            while(true) {
                try {
                    Thread.sleep(1000);
                    System.out.println("Count : " + i++);
                }
                catch(Exception e) {
                    System.out.println("Interrupted");
                }
            }
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    MyThread thread = new MyThread();
    System.out.println("Thread State : " + thread.getState());
    thread.start();
    System.out.println("Thread State : " + thread.getState());
    // Sleep 5 seconds
    Thread.sleep(5000);
    thread.stop();
    System.out.println("Thread State : " + thread.getState());
}
}
```

4. 3번 코드에서 while의 조건을 ($i < 2$)로 바꾸었을 때, 출력 값이 어떻게 바뀔 지 예상하시오.

5. 코드 실행 결과에 근거하여 아래 질문에 O, X로 답하십시오
- a. Not Runnable 상태에서 Runnable 상태로 이동할 수 있다.
 - b. 스레드를 생성하면 해당 스레드는 Runnable 상태이다.
 - c. 프로그램이 정상적으로 종료되면 Terminate 상태로 바뀐다.
 - d. 프로그램이 비정상적으로 종료되면 Not Runnable 상태로 바뀐다.
 - e. 스레드가 Block 상태일 때 I/O 연산이 시작되면 Runnable 상태로 바뀐다.

12.3 wait & notify

A. 개념 정리

앞서 살펴본 Block 상태를 인위적으로 진입하고 빠져나가는 방법으로 메소드 wait와 이에 대응되는 메소드인 notify, notifyall이 있다. 해당 스레드 객체에 대해 메소드 wait가 호출되면 Block 상태로 진입하고, 메소드 notify 또는 notifyall가 호출되면 Block 상태에서 빠져나간다. 이와 같은 특수 메소드들은 스레드 간에 통신에 쓰인다.

특히, 공유된 메모리를 스레드들이 번갈아 독점 사용할 필요가 있을 때 이러한 메소드들이 유용하다. 특정 객체나 데이터를 변경하는 경우에는 독점 사용이 불가피하다. 독점 사용되는 자원이 있다고 하면, 먼저 독점 사용을 시작한 스레드가 해당 부분을 실행 하는 동안 다른 스레드는 메소드 wait를 통해 명시적으로 기다려야한다(Block 상태 진입). 먼저 독점 사용을 시작한 스레드가 사용을 종료하면 기다리고 있는 스레드들에게 명시적으로 메소드 notify/notifyall을 사용하여 독점사용이 종료됨을 알려서 Block 상태에서 해제될 수 있게 한다. wait는 반복문 내부에 위치해서 특정 조건이 만족되어 있는지 확인하며 반복하는 로직으로 많이 사용된다.

메모리 값이 독점 사용되고 있는지 여부를 확인하는 것은 lock을 활용한다. 특정 변수 등을 독점 사용을 하고자 하는 경우 스레드는 해당 변수에 대한 lock을 획득하고, 독점사용을 마치면 lock을 반환한다. Lock을 획득하고자 할 때 다른 스레드가 이미 lock을 가지고 있다면 lock이 반환되기를 기다린다. lock을 가지고 있는 스레드가 해당 lock을 반환한 후에는 기다리는 스레드들에게 반환 사실을 알린다. 이 과정은 메소드 wait 와 notify를 사용하여 구현할 수 있다.

키워드로 synchronized는 lock의 획득과 반환을 지원한다. synchronized 메소드는 메소드 시작할 때 대상 객체의 lock을 획득하여 종료될 때 반환한다. synchronized 블록은 괄호로 명시된 데이터에 대해서 블록 시작시 lock을 획득하여 종료시 반환한다. 아래는 synchronized 블록의 예를 의미한다. 공용 queue에 대한 변경 (removeFirst)을 위해서 독점사용을 해야하며, while 문의 조건을 확인하며 빠져나가지 않으면 계속 wait를 하는 경우이다.

```

public Object pop() throws InterruptedException {
    synchronized(queue) {
        while (queue.isEmpty()) {
            queue.wait();
        }
        return queue.removeFirst();
    }
}

```

synchronized 메소드나 블록 내에서 다시 다른 lock을 얻기위해 기다리는 경우, 스스로 블록 상태에서 빠져나갈 수 없으므로 무한히 기다릴 수도 있다. 아래 예는 다른 스레드에게 while 문을 빠져나갈 수 있는 조건을 달성할 기회를 주지 않으면서 무한히 기다리는 메소드를 나타낸다.

```

public synchronized void transfer (int from, int to, int amount) {
    while (accounts[from] < amount) {
        // wait
    }
    accounts[from] -= amount;
    accounts[to] += amount;
    ntransacts++;
}

```

이것은 infinite loop는 wait, notify를 적절히 사용하면 예방할 수 있다. 즉, 특정 객체의 상태가 바뀌는 것을 기다리는 경우는 가급적 그 객체 내부에서 기다리고, 독점 사용을 마치면 notifyAll을 잊지 않는다. 아래는 공용 queue에 객체를 추가하는 메소드의 구현을 나타낸다. synchronized 블록의 종료 직전 notify를 호출하고 있다.

```

public void push(Object o) {
    synchronized(queue) {
        queue.add(o);
        queue.notify();
    }
}

```

그 밖에도 메소드 yield, join으로 다른 스레드에게 실행 기회를 주거나 다른 스레드가 종료되는 것을 기다리는 방법도 있다.

B. 개념 확인

1. synchronized 키워드를 사용하여 메소드나 블록을 동기화하면 해당 객체의 lock을 획득한다. [O / X]
2. wait 메소드는 호출하면 스레드는 Dead 상태로 진입한다. [O / X]
3. notify 메소드는 Block 상태에 있는 모든 스레드를 깨우는 역할을 한다. [O / X]
4. synchronized 메소드나 블록 내에서 다시 다른 lock을 얻기 위해 기다리는 경우, 무한히 기다릴 수 있다. [O / X]
5. wait 메소드는 반드시 notify 또는 notifyAll 메소드에 의해 깨워져야 한다. [O / X]
6. wait 메소드는 어떤 상태로 스레드를 만들어냅니까?
 - a. Running
 - b. Block
 - c. Terminated
 - d. New
7. 다음 중 공유된 메모리를 스레드들이 번갈아 독점 사용할 때 사용되는 메소드는?
 - a. lock
 - b. wait
 - c. notify
 - d. synchronized
8. push 메소드에서 queue.notify()를 호출하는 이유는 무엇인가?
 - a. 스레드를 중지시키기 위해
 - b. Block 상태에 있는 스레드를 깨우기 위해
 - c. 예외를 발생시키기 위해
 - d. 코드를 간결하게 하기 위해

9. synchronized 블록을 사용할 때 lock을 얻으려고 하는데, 다른 스레드가 이미 lock을 가지고 있다면 어떻게 됩니까?
- 기다린다.
 - 예외가 발생한다.
 - 다음 라인으로 넘어간다.
 - 무시된다.

C. 응용 문제

1. (wait/notify) 다음은 여러 스레드에서 공유하는 데이터인 int 필드 'content'의 값을 접근하고 쓰기 위해 get/put 메소드이다.

```
boolean available = false;
public synchronized int get() {
    if (available == true) {
        available = false;
        return contents;
    }
}
public synchronized void put(int value) {
    if (available == false) {
        available = true;
        contents = value;
    }
}
```

- content 데이터에 누구도 읽어내지 않은 새 값이 채워지기 기다렸다가 읽어오도록 get을 고치시오.
- 한번 적었던 값을 누군가 get으로 읽어낸 후에 기다렸다가 overwrite 할 수 있도록 put 을 다시 작성하시오.

