

자바 프로그래밍

(Java Programming)

1장 자바 시작하기

1.1 프로그래밍 언어란?

프로그래밍 언어라고하면 보통 고급언어를 말하는데 대표적인 프로그래밍 언어인 C, C++, Java는 모두 고급언어에 속한다. 이 언어들로 작성된 내용을 소스(source)라고 부르고 이소스는 컴파일러(compiler)라는 소프트웨어에 의해 기계어로 변환된 후 컴퓨터에서 실행할 수 있게 된다.

1.2 자바란?

1.2.1 자바 소개

1995년도에 처음 썬 마이크로시스템즈(Sun Microsystems)에서 자바(Java)언어를 발표한 후 지금까지 자바는 성공한 프로그래밍 언어로서 전 세계적으로 다양한 분야에서 사용되고 있다. 초기의 자바는 메모리 및 CPU를 지나치게 많이 사용하기 때문에 윈도우 프로그래밍 언어로는 부적합하다는 문제점이 있었다. 하지만 1999년도부터 인터넷이 활성화되면서 웹 애플리케이션 구축용 언어로 자바가 급부상했다. 지금은 스마트폰을 비롯해서 각종 장비와 데스크톱에서 실행되는 애플리케이션, 그리고 금융, 공공, 대기업 등의 엔터프라이즈 기업 환경에서 실행되는 서버 애플리케이션을 개발하는 중추적인 언어로 자리매김하고 있다.

1.2.2 자바의 특징

- 이식성이 높은 언어이다. 이식성이란 서로 다른 실행 환경을 가진 시스템 간에 프로그램을 옮겨 실행할 수 있는 것을 말한다.
- 객체 지향 언어이다. 프로그램을 개발하는 기법으로 부품에 해당하는 객체들을 먼저 만들고 이것들을 하나씩 조립 및 연결해서 전체 프로그램을 완성하는 기법을 객체 지향 프로그래밍이라고 한다.
- 함수적 스타일 코딩을 지원한다.
- 메모리를 자동으로 관리한다.
- 다양한 애플리케이션을 개발할 수 있다. 자바는 윈도우, 리눅스, 유닉스, 맥 등 다양한 운영체제에서 실행되는 프로그램을 개발할 수 있다.
- 멀티 스레드(Multi-Thread)를 쉽게 구현할 수 있다. 하나의 프로그램이 동시에 여러 가지 작업을 할 경우와 대용량 작업을 빨리 처리하기 위해 서버 작업으로 분리해서 병렬 처리하는 멀티 스레드 방식이 지원된다.
- 동적 로딩(Dynamic Loading)을 지원한다. 애플리케이션이 실행될 때 모든 객체가 생성되지 않고 객체가 필요한 시점에 클래스를 동적으로당해서 객체를 생성한다.
- 막강한 오픈소스 라이브러리가 풍부하다. 자바는 오픈소스 언어이기 때문에 자바 프로그램에서 사용하는 라이브러리 또한 오픈소스가 넘쳐난다.

1.2.3 자바 가상 기계(JVM)

운영체제는 자바 프로그램을 바로 실행할 수 없는데 그 이유는 자바 프로그램은 완전히 기계어가 아닌, 중간 단계의 바이트 코드이기 때문에 이것을 해석하고 실행할 수 있는 가상의 운영체제가 필요하다. 이것이 자바 가상 기계이다. 자바 가상 기계 JVM(Java Virtual Machine)은 실 운영체제를 대신해서 자바 프로그램을 실행하는 가상의 운영체제 역할을 한다. 바이트 코드는 모든 JVM에서 동일한 실행 결과를 보장하지만 JVM은 운영체제에 종속이다. 자바 프로그램을 운영체제가 이해하는 기계어로 번역해서 실행해야 하므로 JVM은 운영체제에 맞게 설치되어야 한다. JVM은 JDK 또는 JRE를 설치하면 자동으로 설치되는데 JDK와 JRE가 운영체제별로 제공된다.

1.3 자바 개발 환경 구축

1.3.1 자바 개발 도구(JDK) 설치

자바 프로그램을 개발하기 위해서는 먼저 Java SE(Standard Edition)의 구현체인 JDK를 설치해야 한다. Java SE의 구현체는 자바 개발 키트(JDK: Java Development Kit)와 자바 실행환경(JRE: Java Runtime Environment)이라는 두 가지 버전이 있다. JDK는 프로그램 개발에 필요한 자바 가상 기계(JVM), 라이브러리 API, 컴파일러 등의 개발 도구가 포함되어 있고 JRE에는 프로그램 실행에 필요한 자바 가상 기계(JVM), 라이브러리 API만 포함되어 있다. 자바 프로그램을 개발하고자 하는 것이 아니고 개발된 프로그램만 실행한다면 JRE만 설치한다.

- JRE = JVM + 표준 클래스 라이브러리
- JDK = JRE + 개발에 필요한 도구

JDK는 오라클(<http://www.oracle.com>) 사이트에서 무료로 다운로드 받을 수 있다. JDK는 운영체제별로 설치 파일을 별도로 제공하고 있기 때문에 운영체제에 맞게 설치 파일을 다운로드하면 된다. MS윈도우라면 32비트 또는 64비트의 파일을 선택해서 다운로드한다.

JDK를 설치하면 기본 위치는 C:\Program Files\Java인데, 이 디렉터리를 보면 JDK와 JRE도 설치되어 있는 것을 볼 수 있다. JDK 내부에도 JRE가 있기 때문에 자바 프로그램을 개발하고 실행할 수 있다. 그러나 웹 브라우저에서 실행하는 애플릿(Applet)은 JRE를 요구하기 때문에 애플릿을 개발한다면 JRE도 필요하다. JDK 내부의 bin 디렉터리는 컴파일러인 javac.exe와 자바 가상 기계(JVM) 구동 명령어인 java.exe가 포함되어 있다. 이 명령어들을 다른 디렉터리에서도 쉽게 실행할 수 있도록 Path 환경 변수에 bin 위치를 등록할 필요가 있다.

[내PC]-[속성]-[고급시스템설정]-[고급]-[환경변수]-[시스템변수] Path변수 맨 앞에 C:\Program Files\Java\jdk1.8.0\bin:를 입력한다. 환경 변수 Path가 잘 적용되었는지 체크하기 위해 명령 프롬프트를 실행하고 "javac -version"을 실행해 본다.

1.3.2 API 문서

자바 프로그램을 개발하기 위해서는 JDK에서 제공하는 표준 클래스 라이브러리를 반드시 사용해야 한다. 이 클래스는 API(Application Programming Interface)라고도 하는데 JDK에 포함되어 있는 API들은 방대하기 때문에 쉽게 찾을 수 있도록 API 문서를 제공한다.

- <https://docs.oracle.com/javase/8/docs/api/>

1.4 자바 프로그램 개발 순서

자바 프로그램을 개발하려면 아래와 같은 순서로 진행해야 한다.

- .java 소스 파일 작성
- 컴파일러(javac.exe)로 바이트 코드 파일(.class) 생성
- JVM 구동 명령어(java.exe)로 실행

1.4.1 소스 작성에서부터 실행까지

```
c:/temp/Sample.java
```

```
public class Sample {  
    public static void main(String[] args) {  
        System.out.println("안녕하세요!");  
    }  
}
```

[실행결과]

안녕하세요!

- 컴파일러로 Sample.java 소스 파일을 컴파일 한다. (c:\temp>javac Sample.java)

```
C:/Temp>javac Sample.java
```

- dir 명령어를 실행하여 Sample.class가 생성되었는지 확인한다.

```
C:/Temp>dir
```

- Sample.class를 실행하기 위해 JVM 구동 명령어인 java.exe를 이용한다.

```
C:/Temp>java Sample
```

1.4.2 프로그램 소스 분석

자바 실행 프로그램은 반드시 클래스(class) 블록과 main()메서드 블록으로 구성되어야 한다. 메서드 블록은 단독으로 작성될 수 없고 항상 클래스 블록 내부에서 작성되어야 한다. 클래스와 메서드를 간단하게 설명하면 다음과 같다.

- 클래스: 필드 또는 메서드를 포함하는 블록
- 메서드: 어떤 일을 처리하는 실행 명령어들을 모아 놓은 블록

클래스 이름은 개발자가 마음대로 정할 수 있다. 주의할 점은 소스 파일과 대소문자가 일치해야 한다. 그리고 숫자로 시작할 수 없고 공백을 포함해서도 안 된다. 메서드 이름도 개발자가 마음대로 정할 수 있지만 main()메서드 만큼은 다른 이름으로 바꾸면 안 된다. 왜냐하면 java.exe로 JVM을 구동시키면 제일 먼저 main()메서드를 찾아서 실행시키기 때문이다.

1.5 주석과 실행문

1.5.1 주석 사용하기

주석은 프로그램 실행과는 상관없이 코드에 설명을 붙인 것을 말한다. 컴파일 과정에서 주석은 무시되고 실행문만 바이트 코드로 번역된다. 코드에서 사용하는 주석문의 종류에는 아래와 같이 두 가지가 있다.

- // : //부터 라인 끝까지 주석으로 처리한다. (행 주석)
- /* ~ */ : /*와 */상이에 있는 모든 범위를 주석으로 처리한다. (범위 주석)

1.5.2 실행문과 세미콜론(;)

실행문은 변수 선언, 값 저장, 메서드 호출에 해당하는 코드를 말한다. 실행문을 작성 시 주의할 점은 실행문의 마지막에 반드시(;)을 붙여서 실행문이 끝났음을 표시해주어야 한다. 실행문은 여러 줄에 걸쳐 있어도 되고, 한 줄에 여러 개의 실행문이 있어도 된다.

1.6 이클립스 설치

1.6.1 이클립스 소개

이클립스는 자바 프로그램을 개발하기 위한 통합 개발 환경(IDE: Integrated Development Environment)으로 오픈소스 개발 플랫폼이 무료로 제공된다. 기본적으로 자바 프로그램을 개발할 수 있도록 구성되어 있지만 추가적으로 플러그인을 설치하면 안드로이드 앱 개발, 웹 애플리케이션 개발, C, C++, C# 애플리케이션 개발 등 다양한 개발 환경을 구축할 수 있다.

1.6.2 이클립스 다운로드

이클립스는 자바 언어로 개발된 툴이기 때문에 이클립스를 실행하려면 JVM이 필요하다. 이미 JDK를 설치했기 때문에 이클립스 압축 파일만 다운로드 받으면 된다. 이클립스 압축 파일은 아래 사이트에서 무료로 받을 수 있다. 웹 애플리케이션 등의 엔터프라이즈(네트워크) 환경에서 실행되는 자바 애플리케이션을 개발하기 위해서는 Eclipse IDE for Java EE Developers 파일을 다운로드 받아야 한다.

- <http://www.eclipse.org>

1.6.3 워크스페이스

이클립스를 실행하면 제일 먼저 만나는 [Workspace Launcher] 대화상자가 나타난다. [Workspace]는 이클립스에서 생성한 프로젝트가 기본적으로 저장되는 디렉터리를 말한다. 만약 워크스페이스를 변경하고 싶다면 이클립스의 [File]-[Switch Workspace]-[Other]메뉴를 통해 변경한다. 이클립스는 실행할 때 적용되는 메타데이터를 워크스페이스의 하위 디렉터리인 .metadata에 저장하는데 처음 워크스페이스가 생성되면 이 디렉터리가 자동으로 생성된다.

1.6.4 퍼스펙티브와 뷰

퍼스펙티브(Perspective)는 이클립스에서 프로젝트를 개발할 때 유용하게 사용하는 작은 창들인 뷰(view)들을 묶어 놓은 것을 말한다.

- Package Explorer: 프로젝트를 관리하고 자바 소스 파일을 생성 및 삭제하는 작업을 한다.
- Console: 콘솔로 출력하는 내용을 보여주며 Console뷰가 보이지 않는다면 메뉴에서 [Window]-[Show View]-[Console]을 선택한다.

1.6.5 프로젝트 생성

이클립스에서 자바 소스 파일을 작성하려면 메뉴에서 [File]-[New]-[Java Project]로 자바 프로젝트를 생성해야 한다.

- Project Name: 적절한 프로젝트명을 입력한다. 프로젝트는 기본적으로 워크스페이스의 하위 디렉터리에 생성한다.
- JRE: Path 환경 변수 값에 추가된 JDK의 버전이 디폴트로 설정된다.

Sample.java

```
public class Sample {  
    public static void main(String[] args) {  
        System.out.println("안녕하세요!");  
        System.out.println("저는 홍길동입니다....");  
        System.out.println("저는 해원고등학교에 다닙니다.");  
    }  
}
```

[실행결과]

```
안녕하세요!  
저는 홍길동입니다....  
저는 해원고등학교에 다닙니다.
```

1.6.6 소스 파일 생성과 컴파일

소스 파일 생성은 Package Explorer 뷰에서 프로젝트의 src 폴더를 선택하고 [마우스 오른쪽버튼]-[New]-[Class]를 선택한다. 이클립스는 컴파일을 위한 메뉴가 따로 없다. 저장을 하면 내부적으로 javac.exe가 자동 실행되어 컴파일을 수행한다.

컴파일이 성공되면 bin 디렉터리에 바이트 코드 파일이 생성되며 Package Explorer에서는 보이지 않는다. bin 디렉터를 보고 싶다면 이클립스에서 [Window]-[Show View]-[Navigator]를 선택한다.

1.6.7 바이트 코드 실행

이클립스에서 바이트 코드 파일을 실행하려면 Package Explorer 뷰에서 실행을 원하는 소스파일을 선택하고 [마우스 오른쪽 버튼]-[Run As]-[Java Application]을 클릭한다. 그러면 내부적으로 java.exe가 실행되고 JVM은 컴파일 된 바이트 코드를 실행한다.

2장 변수와 타입

2.1 변수

2.1.1 변수란?

변수(Variable)는 값을 저장할 수 있는 메모리의 공간을 의미한다. 변수에는 오직 한 가지 타입의 값만 저장할 수 있다. 변수란 이름을 갖게 된 이유는 프로그램에 의해서 수치로 값이 변동될 수 있기 때문이다. 변수에는 복수의 값을 저장할 수 없고, 하나의 값만 지정할 수 있다.

2.1.2 변수의 선언

변수 선언은 어떤 타입의 데이터를 저장할 것인지 그리고 변수 이름이 무엇인지를 결정한다. 변수의 명명규칙은 다음과 같다.

- 첫 번째 글자는 문자이거나 '\$', '_' 이어야 하고 숫자로 시작할 수 없다. (필수)
- 영어 대소문자가 구분된다. (필수)
- 첫 문자는 영어 소문자로 시작하되, 다른 언어가 붙을 경우 첫 문자를 대문자로 한다. (권례)
- 문자 수(길이)의 제한은 없다.
- 자바 예약어는 사용할 수 없다. (필수)

2.1.3 변수의 사용

변수를 사용한다는 변수에 값을 저장하고 읽는 행위를 말한다. 변수의 초기값은 코드에서 직접 입력하는 경우가 많은데 소스 코드 내에서 직접 입력된 값을 리터럴(literal)이라고 부른다. 리터럴은 값의 종류에 따라 정수 리터럴, 실수 리터럴, 문자 리터럴, 논리 리터럴로 구분된다.

- 정수 리터럴: 소수점이 없는 정수 리터럴은 10진수로 간주한다.
- 실수 리터럴: 소수점이 있는 리터럴은 10진수 실수로 간주한다.
- 문자 리터럴: 작은따옴표로 묶는 텍스트는 하나의 문자 리터럴로 간주한다.
- 문자열 리터럴: 큰따옴표로 묶는 텍스트는 문자열 리터럴로 간주한다.
- 논리 리터럴: true와 false는 논리 리터럴로 간주한다.

2.1.4 변수의 사용 범위

변수는 중괄호 { } 블록 내에서 선언되고 사용된다. 메서드 블록 내에서 선언된 변수를 로컬 변수라고 부르며 로컬 변수는 메서드 실행이 끝나면 메모리에서 자동으로 없어진다.

2.2 데이터 타입

모든 변수에는 타입이 있으며 타입에 따라 저장할 수 있는 값의 종류와 범위가 달라진다. 변수를 선언할 때 주어진 타입은 변수를 사용하는 도중에 변경할 수 없으므로 변수를 선언할 때 어떤 타입을 사용할지 충분히 고려해야 한다.

2.2.1 기본(원시: primitive) 타입

기본(원시) 타입이란 정수, 실수, 문자, 논리 리터럴을 직접 저장하는 타입을 말한다. 정수 타입에는 byte, char, short, int, long이 있고 실수 타입에는 float, double이 있다. 그리고 논리 타입에는 boolean이 있다.

2.2.2 정수타입(int)

int 타입은 4byte($-2^{31} \sim (2^{31}-1)$)로 표현되는 정수값을 저장할 수 있는 데이터 타입으로 자바에서 정수 연산을 위한 기본 타입이다.

[Sample.java] int 타입 변수

```
public class Sample {
    public static void main(String[] args) {
        System.out.println("안녕하세요!");

        int a=10;
        int b=3;
        int c1=a + b;
        int c2=a * b;
        int c3=a - b;
        int c4=a / b;

        System.out.println(a + "+" + b + "=" + c1);
        System.out.println(a + "*" + b + "=" + c2);
        System.out.println(a + "-" + b + "=" + c3);
        System.out.println(a + "/" + b + "=" + c4);
    }
}
```

[실행결과]

```
안녕하세요!
10+3=13
10*3=30
10-3=7
10/3=3
```

2.2.3 실수 타입(float, double)

실수 타입은 소수점이 있는 실수 데이터를 저장할 수 있는 타입으로 메모리 사용 크기에 따라 4byte float 타입과 8 byte double 타입이 있다. float에 비해 double이 약 두 배의 자릿수가 배정되어 있어 높은 정밀도를 요구하는 계산에서는 double을 사용해야 한다. 자바는 실수 리터럴의 기본 타입을 double로 간주한다. 실수 리터럴을 float 타입 변수에 저장하려면 리터럴 뒤에 소문자 'f'나 대문자 'F'를 붙여야 한다.

[Sample.java] float과 double 타입

```
public class Sample {
    public static void main(String[] args) {
        float a=3.14f;

        int b=5;
        float c=b * b * a;
        System.out.println("반지름이 " + b + " 원의 넓이는 " + c + "입니다.");

        int d = 5;
        int e = 7;
        float f = (float)d * e * 1/2;

        System.out.println("밀변:" + d);
        System.out.println("높이:" + e);
        System.out.println("넓이:" + f);
    }
}
```

[실행결과]

반지름이 5 원의 넓이는 78.5입니다.
밀변:5
높이:7
넓이:17.5

2.2.4 문자 타입(char, String)

char 타입 변수는 단 하나의 문자만을 저장하며 작은따옴표로 감싼 문자 리터럴을 대입한다. 만약 문자열을 저장하고 싶다면 String 타입을 사용하며 큰따옴표로 감싼 문자열 리터럴을 대입하면 된다. String은 기본 타입이 아니고 클래스 타입이다.

[Sample.java] char, String 타입

```
public class Sample {
    public static void main(String[] args) {
        char ban='A';
        String name="홍길동";

        int kor=90;
        int eng=95;
        int mat=87;
        int tot=kor + eng + mat;
        float avg=(float)tot/3;

        System.out.println("반:" + ban);
        System.out.println("이름:" + name);
        System.out.println("국어:" + kor);
        System.out.println("영어:" + eng);
        System.out.println("총점:" + tot);
        System.out.println("평균:" + avg);
    }
}
```

[실행결과]

반:A
이름:홍길동
국어:90
영어:95
총점:272
평균:90.666664

2.2.5 논리 타입(boolean)

boolean 타입은 1byte(8bit)로 표현되는 논리값(true/false)을 저장할 수 있는 데이터 타입이다. 두 가지 상태값을 저장할 필요성이 있을 경우에 사용되며, 상태값에 따라 조건문과 제어문의 실행 흐름을 변경하는데 주로 이용된다.

[Sample.java] boolean 타입

```
public class Sample {
    public static void main(String[] args) {
        boolean stop=true;
        System.out.println(stop);

        stop=false;
        System.out.println(stop);
    }
}
```

[실행결과]

true
false

2.2.6 예제 (성적 출력 프로그램)

[Sample.java] 성적 출력 프로그램

```
public class Sample {
    public static void main(String[] args) {
        char b1='A', b2='A', b3='A';
        String n1="홍길동", n2="심청이", n3="이순신";

        int k1=90, k2=85, k3=70;
        int e1=95, e2=70, e3=70;
        int m1=87, m2=60, m3=60;

        int t1=k1+e1+m1, t2=k2+e2+m2, t3=k3+e3+m3;
        float a1=(float)t1/3, a2=(float)t2/3, a3=(float)t3/3;

        System.out.println(b1+" "+n1+" "+k1+" "+e1+" "+m1+" "+t1+" "+a1);
        System.out.println(b2+" "+n2+" "+k2+" "+e2+" "+m2+" "+t2+" "+a2);
        System.out.println(b3+" "+n3+" "+k3+" "+e3+" "+m3+" "+t3+" "+a3);
    }
}
```

[실행결과]

```
A 홍길동 90 95 87 272 90.666664
A 심청이 85 70 60 215 71.666664
A 이순신 70 70 60 200 66.666664
```

2.3 타입 변환

타입 변환이란 데이터 타입을 다른 데이터 타입으로 변환하는 것을 말한다. byte 타입을 int 타입으로 변환하거나 반대로 int 타입을 byte 타입으로 변환하는 행위를 말한다. 타입 변환에는 두 가지 종류가 있다. 하나는 자동(묵시적) 타입 변환이고 다른 하나는 강제(명시적) 타입 변환이다.

2.3.1 자동 타입 변환

자동 타입 변환(Promotion)은 프로그램 실행 도중에 자동적으로 타입 변환이 일어나는 것을 말한다. 자동 타입 변환은 작은 크기를 가지는 타입이 큰 크기를 가지는 타입에 저장될 때 발생한다. 큰 크기 타입과 작은 크기 타입의 구분은 사용하는 메모리 크기이다.

[Sample.java] 자동 타입 변환

```
public class Sample {
    public static void main(String[] args) {
        char charValue = 'A'
        int intValue = charValue
        System.out.println("A의 유니코드=" + intValue);

        intValue = 200;
        double doubleValue = intValue
        System.out.println(doubleValue);
    }
}
```

[실행결과]

```
A의 유니코드=65
200.0
```

2.3.2 강제 타입 변환

큰 크기의 타입은 작은 크기의 타입으로 자동 타입 전환을 할 수 없다. 하지만 큰 그릇을 작은 그릇 사이즈로 쪼개어서 한 조각만 그릇에 넣는다면 가능하다. 이와 같이 강제로 큰 데이터 타입을 작은 데이터 타입으로 쪼개어서 저장하는 것을 강제 타입 변환(캐스팅: Casting)이라고 한다. 강제 타입 변환은 캐스팅 연산자 ()를 사용하는데 괄호 안에 들어가는 타입은 쪼개는 단위이다.

작은 크기 타입 = (작은 크기 타입) 큰 크기 타입

[Sample.java] 자동 타입 변환

```
public class Sample {
    public static void main(String[] args) {
        int intValue = 65;
        char charValue = (char)intValue;
        System.out.println(charValue);

        double doubleValue = 3.14;
        int intValue = (int)doubleValue;
        System.out.println(intValue);
    }
}
```

[실행결과]

```
A
3
```

강제 타입 변환에서 주의할 점은 사용자로부터 입력받은 값을 변환할 때 값의 손실이 발생하면 안 된다는 것이다. 강제 타입 변환을 하기 전에 우선 안전하게 값이 보존될 수 있는지 검사하는 것이 좋다. 아래 예제는 byte 타입으로 변환하기 전에 변환될 값이 byte 타입으로 변환된 후에도 값의 손실이 발생하지 않는지 검사해서 올바른 타입 변환이 되도록 한다.

[Sample.java] 변환으로 인한 데이터 손실이 발생하지 않도록 한다.

```
public class Sample {
    public static void main(String[] args) {
        int i = 128;

        if(i < Byte.MIN_VALUE || (i > Byte.MAX_VALUE)) {
            System.out.println("byte 타입으로 변환할 수 없습니다.");
            System.out.println("값을 다시 확인해 주세요.");
        } else {
            byte b = (byte)i;
            System.out.println(b);
        }
    }
}
```

[실행결과]

byte 타입으로 변환할 수 없습니다.
값을 다시 확인해 주세요.

int 값을 실수값으로 변환할 경우 float 타입으로 자동 변환하면 문제가 발생하므로 안전하게 변환시키는 double 타입을 사용한다.

[Sample.java] 정수 타입을 실수 타입으로 변환할 때 정밀도 손실을 피한다.

```
public class Sample {
    public static void main(String[] args) {
        int num1 = 123456780;
        int num2 = 123456780;

        float floatNum = num2;
        num2 = (int)floatNum;

        int result1 = num1 - num2;
        System.out.println("정수를 float 타입으로 변환한 결과: " + result1);

        int intNum1 = 123456780;
        int intNum2 = 123456780;

        double doubleNum = num2;
        intNum2 = (int)doubleNum;

        int result2 = intNum1 - intNum2;
        System.out.println("정수를 double 타입으로 변환한 결과: " + result2);
    }
}
```

[실행결과]

정수를 float 타입으로 변환한 결과: -4
정수를 double 타입으로 변환한 결과: 0

2.3.3 연산식에서의 자동 타입 변환

연산은 기본적으로 같은 타입의 피연산자 간에만 수행되기 때문에 서로 다른 타입의 피연산자가 있을 경우 두 피연산자 중 크기가 큰 타입으로 자동 변환된 후 연산을 수행한다. 예를 들어 int 타입 피연산자와 double 타입 연산자를 덧셈 연산하면 먼저 int 타입 피연산자가 double 타입으로 자동 변환되고 연산을 수행하며 연산의 결과는 Double이 된다. 만약 int 타입으로 꼭 연산을 해야 한다면 double 타입을 int 타입으로 강제 변환하고 덧셈 연산을 수행한다.

[Sample.java] 자동 타입 변환

```
public class Sample {
    public static void main(String[] args) {
        char charValue1 = 'A';
        int intValue1 = 1;

        int intValue2 = charValue1 + intValue1;
        System.out.println("유니코드=" + intValue2);
        System.out.println("출력문자=" + (char)intValue2);

        int intValue3 = 10;
        double doubleValue = intValue3 / 4.0;
        System.out.println(doubleValue);
    }
}
```

[실행결과]

유니코드=66
출력문자=B
2.5

3장 연산자

3.1 연산자와 연산식

프로그램에서 데이터를 처리하여 결과를 산출하는 것을 연산(operations)이라고 한다. 연산에 사용되는 표시나 기호를 연산자(operator)라고 하고 연산되는 데이터는 피연산자(operand)라고 한다. 연산자와 피연산자를 이용해 연산의 과정을 기술한 것을 연산식(expressions)이라고 한다.

3.2 연산의 방향과 우선순위

- 단항, 이항, 삼항 연산자 순으로 우선순위를 가진다.
- 산술, 비교, 논리, 대입 연산자 순으로 우선순위를 가진다.
- 단항과 대입 연산자를 제외한 모든 연산의 방향은 왼쪽에서 오른쪽이다.
- 복잡한 연산식에는 괄호()를 사용해서 우선순위를 정해준다.

3.3 단항 연산자

단항 연산자는 피연산자가 단 하나뿐인 연산자를 말하며 여기에는 부호 연산자(+, -), 증감 연산자(++, --), 논리 부정 연산자(!)가 있다.

3.3.1 부호 연산자

부호 연산자는 양수 및 음수를 표시하는 +, -를 말한다. boolean 타입과 char 타입을 제외한 나머지 기본 타입에 사용할 수 있다.

[Sample.java] 부호 연산자

```
public class Sample {
    public static void main(String[] args) {
        int x = 100;
        int result1 = +x;
        int result2 = -x;

        System.out.println("result1=" + result1);
        System.out.println("result2=" + result2);
    }
}
```

[실행결과]

```
result1=100
result2=-100
```

3.3.2 증감 연산자(++ , --)

증감 연산자는 변수의 값을 1증가시키거나 1감소시키는 연산자를 말한다. boolean 타입을 제외하고 모든 기본 타입의 피연산자에 사용할 수 있다.

[Sample.java] 증감 연산자

```
public class Sample {
    public static void main(String[] args) {
        int x = 10, y = 10, int z;

        x++; //x=11, y=10, z=0
        ++x; //x=12, y=10, z=0
        System.out.println("x=" + x);

        y--; //x=12, y=9, z=0
        --y; //x=12, y=8, z=0
        System.out.println("-----");
        System.out.println("y=" + y);

        z=x++; //x=13, y=8, z=12
        System.out.println("-----");
        System.out.println("z=" + z);
        System.out.println("x=" + x);

        z=++x; //x=14, y=8, z=14
        System.out.println("-----");
        System.out.println("z=" + z);
        System.out.println("x=" + x);

        z=++x + y++; // x=15, y=9, z=23
        System.out.println("-----");
        System.out.println("z=" + z);
        System.out.println("x=" + x);
        System.out.println("y=" + y);
    }
}
```

[실행결과]

```
x=12
-----
y=8
-----
z=12
x=13
-----
z=14
x=14
-----
z=23
x=15
y=9
```

3.3.3 논리 부정 연산자(!)

논리 부정 연산자는 true를 false로 false를 true로 변경하기 때문에 boolean 타입에만 사용할 수 있다. 논리 부정 연산자는 조건문과 제어문에서 사용되어 조건식의 값을 부정하도록 해서 실행 흐름을 제어할 때 주로 사용된다. 또한 두 가지 상태(true/false)를 번갈아가며 변경하는 토글(toggle) 기능을 구현할 때도 주로 사용한다.

[Sample.java] 논리 부정 연산자

```
public class Sample {  
    public static void main(String[] args) {  
        boolean play = true;  
        System.out.println(play);  
  
        play=!play;  
        System.out.println(play);  
  
        play=!play;  
        System.out.println(play);  
    }  
}
```

[실행결과]

```
true  
false  
true
```

3.4 이항 연산자

이항 연산자는 피연산자가 두 개인 연산자를 말하며 여기에는 산술 연산자, 문자열 연결 연산자, 대입연산자, 비교연산자, 논리연산자 등이 있다.

3.4.1 산술 연산자(+, -, *, /, %)

[Sample.java] 산술 연산자

```
public class Sample {  
    public static void main(String[] args) {  
        int v1 = 5, int v2 = 2;  
  
        int result1 = v1 + v2;  
        System.out.println("result1=" + result1);  
  
        int result2 = v1 - v2;  
        System.out.println("result2=" + result2);  
  
        int result3 = v1 * v2;  
        System.out.println("result3=" + result3);  
  
        int result4 = v1 / v2;  
        System.out.println("result4=" + result4);  
  
        int result5 = v1 % v2;  
        System.out.println("result5=" + result5);  
  
        double result6 = (double)v1 % v2;  
        System.out.println("result5=" + result6);  
    }  
}
```

[실행결과]

```
result1=7  
result2=3  
result3=10  
result4=2  
result5=1  
result5=1.0
```

3.4.2 문자열 연결 연산자(+)

문자열 연결 연산자인 +는 문자열을 서로 결합하는 연산자이다. + 연산자는 산술 연산자, 부호 연산자인 동시에 문자열 연결 연산자이기도 하다. 피연산자 중 한쪽이 문자열이면 + 연산자는 문자열 연결 연산자로 사용되어 다른 피연산자를 문자열로 변환하고 서로 결합한다.

[Sample.java] 문자열 연결 연산자

```
public class Sample {  
    public static void main(String[] args) {  
        String str1 = "JDK" + 6.0;  
        String str2 = str1 + " 특징";  
        System.out.println(str2);  
  
        String str3 = "JDK" + 3 + 3.0;  
        String str4 = 3 + 3.0 + "JDK";  
        System.out.println(str3);  
        System.out.println(str4);  
    }  
}
```

[실행결과]

```
JDK6.0 특징  
JDK33.0  
6.0JDK
```

3.4.3 비교 연산자(<, <=, >=, ==, !=)

비교 연산자는 대소(<, <=, >, >=) 또는 동등(==, !=)을 비교해서 boolean 타입인 true/false를 산출한다. 대소 연산자는 boolean 타입을 제외한 기본 타입에 사용할 수 있고, 동등 연산자는 모든 타입에서 사용될 수 있다. 비교 연산자는 흐름 제어문인 조건문(if), 반복문(for, while)에서 주로 이용되어 실행 흐름을 제어할 때 사용된다.

구분	연산식			설명
동등비교	피연산자1	==	피연산자2	두 피연산자의 값이 같은지를 검사
	피연산자1	!=	피연산자2	두 피연산자의 값이 다른지를 검사
크기비교	피연산자1	>	피연산자2	피연산자1이 큰지를 검사
	피연산자1	>=	피연산자2	피연산자1이 크거나 같은지를 검사
	피연산자1	<	피연산자2	피연산자1이 작은지를 검사
	피연산자1	<=	피연산자2	피연산자1이 작거나 같은지를 검사

[Sample.java] 비교 연산자

```
public class Sample {
    public static void main(String[] args) {
        int num1 = 10;
        int num2 = 10;
        boolean result1 = (num1 == num2);
        boolean result2 = (num1 != num2);
        boolean result3 = (num1 <= num2);
        System.out.println("result1=" + result1);
        System.out.println("result2=" + result2);
        System.out.println("result3=" + result3);

        char char1 = 'A';
        char char2 = 'B';
        boolean result4 = (char1 < char2);
        System.out.println("result4=" + result4);
    }
}
```

[실행결과]

```
result1=true
result2=false
result3=true
result4=true
```

만약 피연산자가 char 타입이면 유니코드 값으로 비교 연산을 수행한다. 예를 들어 'A'의 유니코드는 65이고 'B'의 유니코드는 66이므로 비교 연산자는 65와 66을 비교하게 된다.

('A' < 'B') → (65 < 66)

비교 연산자에서도 연산을 수행하기 전에 타입 변환을 통해 피연산자의 타입을 일치시킨다. 예를 들어 'A'=65는 'A'가 int 타입으로 변환되어 65가 된 다음 65==65로 비교한다. 마찬가지로 3==3.0에서 3은 int 타입이고 3.0은 double 타입이므로 int 타입인 3을 보다 큰 타입인 double타입으로 변환한 다음 3.0==3.0으로 비교한다.

```
'A' == 65 -> true
3 == 3.0 -> true
```

0.1== 0.1f 식은 결과가 false이다. 이유는 가수를 사용하는 모든 부동소수점 타입은 0.1을 정확히 표현할 수가 없어서 0.1f는 0.1의 근사값으로 표현되어 0.1보다 큰 값이 되어버린다. 해결책은 아래 예제와 같이 피연산자를 모두 float 타입으로 강제 타입 변환한 후에 비교 연산을 하든지 정수로 변환해서 비교하면 된다.

[Sample.java] 비교 연산자

```
public class Sample {
    public static void main(String[] args) {
        int v2 = 1;
        double v3 = 1.0;
        System.out.println(v2 == v3);

        double v4 = 0.1;
        float v5 = 0.1f;
        System.out.println(v4 == v5);
        System.out.println((float)v4 == v5);
        System.out.println((int)(v4*10) == (int)(v5 * 10));
    }
}
```

[실행결과]

```
true
false
true
true
```

3.4.4 논리 연산자(&&, ||, !)

논리 연산자는 논리곱(&&), 논리합(||) 그리고 부정(!) 연산을 수행한다. 논리 연산자의 피연산자는 boolean 타입만 사용할 수 있다.

[Sample.java] 논리 연산자

```
public class Sample {
    public static void main(String[] args) {
        int num1 = 55;
        int num2 = 65;

        System.out.println(num1 >=50 && num2 <=60);
        System.out.println(num1 >=50 || num2 <=60);
        System.out.println(!(num1 < 50));
        System.out.println(!(num1 >=50) && !(num2 <= 60));
        System.out.println(!(num1 >=50) || !(num2 <= 60));
    }
}
```

[실행결과]

```
false
true
true
false
true
```

3.4.5 대입 연산자 (=, +=, -=, *=, /=, %=)

대입 연산자는 오른쪽 피연산자의 값을 좌측 피연산자인 변수에 저장한다. 오른쪽 피연산자는 리터럴 및 변수, 그리고 다른 연산식이 올 수 있다. 단순히 오른쪽 피연산자의 값을 변수에 저장하는 단순 대입 연산자가 있고, 정해진 연산을 수행한 후 결과를 변수에 저장하는 복합 대입 연산자도 있다.

[Sample.java] 대입 연산자

```
public class Sample {
    public static void main(String[] args) {
        int result = 0;

        result +=10;
        System.out.println("result=" + result);

        result -=5;
        System.out.println("result=" + result);

        result *=3;
        System.out.println("result=" + result);

        result /=5;
        System.out.println("result=" + result);

        result %=3;
        System.out.println("result=" + result);
    }
}
```

[실행결과]

```
result=10
result=5
result=15
result=3
result=0
```

3.5 삼항 연산자

삼항 연산자는 세 개의 피연산자가 필요로 하는 연산자를 말한다. 삼항 연산자는 ? 앞의 조건식에 따라 콜론(:) 앞뒤의 피연산자가 선택된다고 해서 조건 연산식이라고 부르기도 한다. 삼항 연산자를 사용하는 방법은 다음과 같다.

조건식(피연산자1) ? 값 또는 연산식(피연산자2) : 값 또는 연산식(피연산자3)

조건식을 연산하여 true가 나오면 삼항 연산자의 결과는 피연산자2가 된다. 반면에 조건식을 연산하여 false가 나오면 삼항 연산자의 결과는 피연산자3이 된다. 피연산자2와 3에는 주로 값이 오지만 경우에 따라서는 연산식이 올 수도 있다.

삼항 연산자는 if문으로 변경해서 작성할 수도 있지만, 한 줄에 간단하게 삽입해서 사용할 경우에는 삼항 연산자를 사용하는 것이 더 효율적이다.

[Sample.java] 삼항 연산자

```
public class Sample {
    public static void main(String[] args) {
        int score = 85;

        char grade = (score > 90) ? 'A' : ( (score > 80) ? 'B' : 'C' );
        System.out.println(score + "점은 " + grade + "등급입니다.");
    }
}
```

[실행결과]

85점은 B등급입니다.

4장 조건문과 반복문

4.1 코드 실행 흐름 제어

자바 프로그램을 시작하면 main() 메서드의 시작 중괄호 {에서 시작해서 끝 중괄호 } 까지 위에서부터 아래로 실행하는 흐름을 가지고 있다. 이러한 실행 흐름을 개발자가 원하는 방향으로 바꿀 수 있도록 해주는 것이 흐름 제어문이다. 제어문의 종류는 조건문과 반복문이 있는데, 조건문에는 if문, switch문이 있고 반복문에는 for문, while문, do-while문이 있다.

4.2 조건문(if문, switch문)

4.2.1 if문

if문은 조건식의 결과에 따라 블록 실행 여부가 결정된다. 조건식에는 true 또는 false 값을 산출할 수 있는 연산식이나, boolean 변수가 올 수 있다. 조건식이 true이면 블록을 실행하고 false이면 블록을 실행하지 않는다.

[Sample.java] if문

```
public class Sample {
    public static void main(String[] args) {
        int score=90;
        if(score >= 90) {
            System.out.println("점수가 90보다 큼니다 \n등급은 A입니다.");
        }
        if(score < 90) {
            System.out.println("점수가 90보다 작습니다.\n등급은 B입니다.");
        }
    }
}
```

[실행결과]

점수가 90보다 큼니다.
등급은 A입니다.

4.2.2 if-else문

if문은 else 블록과 함께 사용되어 조건식의 결과에 따라 실행 블록을 선택한다. if문의 조건식이 true이면 if문 블록이 실행되고, 조건식이 false이면 else블록이 실행된다. 조건식의 결과에 따라 이 두 개의 블록 중 어느 한 블록의 내용만 실행하고 전체 if문을 벗어나게 된다.

[Sample.java] if-else문

```
public class Sample {
    public static void main(String[] args) {
        int score = 85;
        if(score >= 90) {
            System.out.println("점수가 90보다 큼니다 \n등급은 A입니다.");
        } else { /score<90일 경우
            System.out.println("점수가 90보다 작습니다.\n등급은 B입니다.");
        }
    }
}
```

[실행결과]

점수가 90보다 작습니다.
등급은 B입니다.

4.2.3 if-else if-else 문

조건문이 여러 개인 if문도 있다. 처음 if문의 조건식이 false일 경우 다른 조건식의 결과에 따라 실행 블록을 선택할 수 있는데 if블록의 끝에 else if문을 붙이면 된다. else if문의 수는 제한이 없으며 여러 개의 조건식 중 true가 되는 블록만 실행하고 전체 if문을 벗어나게 된다.

[Sample.java] if-else if-else 문

```
public class Sample {
    public static void main(String[] args) {
        int score = 75;
        if(score >= 90) {
            System.out.println("점수가 90점 이상입니다.\n등급은 A입니다.");
        } else if(score >= 80) {
            System.out.println("점수가 80~89 입니다.\n등급은 B입니다.");
        } else if(score >= 70) {
            System.out.println("점수가 70~79 입니다.\n등급은 C입니다.");
        } else {
            System.out.println("점수가 70점 미만입니다.\n등급은 F입니다.");
        }
    }
}
```

[실행결과]

점수가 70~79 입니다.
등급은 C 입니다.

4.2.4 중첩 if문

if문의 블록 내부에는 또 다른 if문을 사용할 수 있다. 이것을 중첩 if문이라 부르는데, 중첩단계에는 제한이 없기 때문에 실행 흐름을 잘 판단해서 작성하면 된다. 사실 if문만 중첩이 되는 것은 아니며 if문, switch문, while문, do while문은 서로 중첩시킬 수 있다. 아래 예제는 바깥 if문은 90점과 80점을 기준으로 조건문을 작성하고, 중첩 if문은 좀 더 세부적으로 95점과 85점을 기준으로 조건문을 작성하였다.

[Sample.java] 중첩 if문

```
public class Sample {
    public static void main(String[] args) {
        int score = 88;
        System.out.println("점수:" + score);

        String grade;
        if(score >= 90) {
            if(score >= 95) {
                grade = "A+";
            } else {
                grade = "A0";
            }
        } else {
            if(score >= 85) {
                grade = "B+";
            } else {
                grade = "B0";
            }
        }
        System.out.println("학점:" + grade);
    }
}
```

[실행결과]

점수:88
학점:B+

4.2.5 switch문

switch문은 if문과 마찬가지로 조건 제어문이다. 하지만 switch문은 if문처럼 조건식이 true일 경우에 블록 내부의 실행문을 실행하는 것이 아니라, 변수가 어떤 값을 갖느냐에 따라 실행문이 선택된다. if문은 조건식의 결과가 true, false 두 가지밖에 없기 때문에 경우의 수가 많아질수록 else-if를 반복적으로 추가해야 하므로 코드가 복잡해진다. 그러나 switch문은 변수의 값에 따라서 실행문이 결정되기 때문에 같은 기능의 if문보다 코드가 간결하다.

switch문은 괄호 안의 값과 동일한 값을 갖는 case로 가서 실행문을 실행시킨다. 괄호 안의 값과 동일한 값을 갖는 case가 없으면 default로 가서 실행문을 실행시킨다. default는 생략가능하다.

[Sample.java] switch문

```
public class Sample {
    public static void main(String[] args) {
        int num=5;

        switch(num) {
            case 1:
                System.out.println("1번이 나왔습니다.");
                break;
            case 2:
                System.out.println("2번이 나왔습니다.");
                break;
            case 3:
                System.out.println("3번이 나왔습니다.");
                break;
            case 4:
                System.out.println("4번이 나왔습니다.");
                break;
            case 5:
                System.out.println("5번이 나왔습니다.");
                break;
            default:
                System.out.println("6번이 나왔습니다.");
                break;
        }
    }
}
```

[실행결과]

5번이 나왔습니다.

case 끝에 break가 붙어 있는 이유는 다음 case를 실행하지 말고 switch문을 빠져나오기 위해서이다. break가 없다면 다음 case가 연달아 실행되는데 이때에는 case 값과는 상관없이 실행된다.

[Sample.java] break문이 없는 case

```
public class Sample {
    public static void main(String[] args) {
        int time = 9;

        switch(time) {
            case 8:
                System.out.println("출근을 합니다.");
            case 9:
                System.out.println("회의를 합니다.");
            case 10:
                System.out.println("업무를 봅니다.");
            default:
                System.out.println("외근을 나갑니다.");
        }
    }
}
```

[실행결과]

회의를 합니다.
업무를 봅니다.
외근을 나갑니다.

char 타입 변수도 switch문에 사용될 수 있다. 아래는 영어 대소문자에 관계없이 똑같은 알파벳이라면 동일하게 처리하도록 만든 switch문이다.

[Sample.java] char 타입의 switch문

```
public class Sample {
    public static void main(String[] args) {
        char grade = 'B';

        switch(grade) {
            case 'A':
            case 'a':
                System.out.println("우수 회원입니다.");
                break;
            case 'B':
            case 'b':
                System.out.println("일반 회원입니다.");
                break;
            default:
                System.out.println("손님입니다.");
        }
    }
}
```

[실행결과]

일반 회원입니다.

자바 6까지는 switch문의 괄호에는 정수타입(byte, char, short, int, long) 변수나 정수값을 산출하는 연산식만 올 수 있었다. 자바7부터는 String 타입의 변수도 올 수 있다. 아래는 직급별 월급을 출력한다.

[Sample.java] String 타입의 switch문

```
public class Sample {
    public static void main(String[] args) {
        String position = "과장";

        switch(position) {
            case "부장":
                System.out.println("700만원");
                break;
            case "과장":
                System.out.println("500만원");
                break;
            case "대리":
                System.out.println("350만원");
                break;
            default:
                System.out.println("300만원");
        }
    }
}
```

[실행결과]

500만원

4.3 반복문(for문, while문, do-while문)

반복문은 어떤 작업(코드들)이 반복적으로 실행되도록 할 때 사용되며, 반복문의 종류에는 for문, while문, do-while문이 있다. for문과 while문은 서로 변환이 가능하기 때문에 반복문을 작성할 때 어느 쪽을 선택해도 좋지만, for문은 반복 횟수를 알고 있을 때 주로 사용하고, while문은 조건에 따라 반복할 때 주로 사용한다. while문과 do-while문은 조건을 먼저 검사하느냐 나중에 검사하느냐 차이일 뿐 동작 방식은 동일하다.

4.3.1 for문

반복문은 한 번 작성된 실행문을 여러 번 반복해서 실행해주기 때문에 코드를 절감하고 간결하게 만들어준다. 코드가 간결하면 개발 시간을 줄일 수 있고 오류가 날 확률도 줄어든다. for문은 주어진 횟수만큼 실행문을 반복 실행할 때 적합한 반복 제어문이다. 아래는 for문의 형식을 보여준다.

```
for(①초기화식; ②조건식; ④증감식) {  
    .....  
    ③실행문;  
}
```

for문이 처음 실행될 때 ①초기화식이 제일 먼저 실행된다. 그런 다음 ②조건식을 평가해서 true이면 ③실행문을 실행시키고, false이면 for문 블록을 실행하지 않고 끝나게 된다. 블록 내부의 ③실행문들이 모두 실행되면 ④증감식을 실행시키고 다시 ②조건식을 평가하게 된다. 평가 결과가 true이면 ③→④→②로 다시 진행하고, false이면 for문이 끝나게 된다. 아래는 가장 기본적인 for문의 형태로 1부터 10까지 출력하는 코드이다.

[Sample.java] 1부터 5까지 출력

```
public class Sample {  
    public static void main(String[] args) {  
        for(int i =1; i<=5; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

[실행결과]

1
2
3
4
5

초기화식에 선언된 변수는 for문 블록 내부에서 사용되는 로컬 변수이다. 따라서 for문을 벗어나서 사용할 수 없다. 아래 예제는 1부터 100까지의 합을 구하는 코드이다. 변수 sum을 for문이 시작하기 전에 선언한 이유는 for문을 끝내고 sum을 사용하기 때문이다.

[Sample.java] 1부터 100까지 합을 출력

```
public class Sample {  
    public static void main(String[] args) {  
        int sum = 0;  
  
        for(int i =1; i<=100; i++) {  
            sum += i;  
        }  
        System.out.println("1~100 합:" + sum);  
    }  
}
```

[실행결과]

1~100 합:5050

변수 i가 for문의 초기화식에서 선언되지 않고 for문 전에 선언되었다면 for문 내부뿐만 아니라 for문을 벗어나서도 사용할 수 있다.

[Sample.java] 1부터 100까지 합을 출력

```
public class Sample {  
    public static void main(String[] args) {  
        int sum = 0;  
  
        int i = 0; //for문 밖에서 변수 정의  
        for(i =1; i<=100; i++) {  
            sum += i;  
        }  
  
        System.out.println("i = " + i);  
        System.out.println("sum = " + sum);  
    }  
}
```

[실행결과]

i = 101
sum = 5050

아래 예제는 키보드에서 1, 2를 입력했을 때 속도를 증속, 감속시키고, 3을 입력하면 프로그램을 종료시킨다.

[Sample.java] 키보드로 while문 제어

```
public class Sample {
    public static void main(String[] args) throws Exception{
        boolean run = true;
        int speed = 0;
        int keyCode = 0;

        while(run) { //Enter키가 입력되면 캐리지리턴(13)과 라인피드(10)가 입력되므로 이 값을 제외시킴
            if(keyCode != 13 && keyCode != 10) {
                System.out.println("-----");
                System.out.println("1.증속 | 2.감속 | 3.중지");
                System.out.println("-----");
                System.out.print("선택: ");

                keyCode = System.in.read(); //키보드의 키 코드를 읽음

                if(keyCode == 49) {
                    speed++;
                    System.out.println("현재 속도=" + speed);
                }else if(keyCode == 50) {
                    speed--;
                    System.out.println("현재 속도=" + speed);
                }else if(keyCode == 51) {
                    run = false; //while문을 종료하기 위해 run 변수에 false를 저장
                }
            }

            System.out.println("프로그램 종료");
        }
    }
}
```

[실행결과]

1.증속 | 2.감속 | 3.중지

선택: 1

현재 속도=1

1.증속 | 2.감속 | 3.중지

선택:

1.증속 | 2.감속 | 3.중지

선택: 2

현재 속도=-1

1.증속 | 2.감속 | 3.중지

선택:

1.증속 | 2.감속 | 3.중지

선택: 3

프로그램 종료

위 예제를 실행하여 1을 입력하고 Enter키를 누르면 while문이 세 번 반복되고 세 번의 키 코드를 리턴 하는데, 첫 번째 1의 키코드인 49, 두 번째는 Enter키의 캐리지리턴 13, 세 번째 역시 Enter키의 라인피드 10을 읽는다. 그리고 프로그램은 새로운 키 코드를 받기 위해 대기한다. 2를 입력해도 마찬가지다. 그러나 3을 입력하면 변수 run이 false가 되어 while문이 종료되고 프로그램이 종료된다.

4.3.3 do-while문

do-while문은 조건식에 의해 반복 실행한다는 점에서 while문과 동일하다. while문은 시작 할 때부터 조건식을 검사하여 블록 내부를 실행하지 결정하지만, 경우에 따라서는 블록 내부의 실행문을 우선 실행시키고 실행 결과에 따라서 반복 실행을 계속할지 결정하는 경우도 발생한다. 예를 들어 키보드로 입력받은 내용을 조사하여 계속 루프를 돌 것인지지를 판단하는 프로그램이 있다고 가정하자. 조건식은 키보드로 입력받은 이후에 평가되어야 하므로, 우선적으로 키보드로부터 입력된 내용을 받아야 한다.

[Sample.java] do-while문

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

```
public class Sample {
    public static void main(String[] args) {
        System.out.println("메시지를 입력 하세요");
        System.out.println("프로그램을 종료하려면 q를 입력하세요.");

        Scanner scanner = new Scanner(System.in);
        String inputString;

        do {
            System.out.print(">");
            inputString = scanner.nextLine();
            System.out.println(inputString);
        }while( !inputString.equals("q") );

        System.out.println();
        System.out.println("프로그램 종료");
    }
} //scanner.nextLine()은 '\n'을 포함하는 한 라인을 읽고 '\n'을 버린 나머지만 리턴 한다.
```

[실행결과]

메시지를 입력하세요
프로그램을 종료하려면 q를 입력하세요.

>안녕하세요!

안녕하세요!

>반갑습니다!

반갑습니다!

>q

q

프로그램 종료

4.3.4 break문

break문은 반복적인 for문, while문, do-while문을 실행 중지할 때 사용된다. 또한 이전에 학습한 switch문에서도 break문을 사용하여 switch문을 종료하였다. break문은 대개 if문과 같이 사용되어 if문의 조건식에 따라 for문과 while문을 종료할 때 사용한다. 아래 예제는 while문을 이용해서 주사위 번호 중 하나를 반복적으로 뽑되, 6이 나오면 while문을 종료시킨다.

[Sample.java] break로 while문 종료

```
public class Sample {
    public static void main(String[] args) {
        while(true){
            //Math.random() 함수는 0이상 1미만의 난수 발생
            int num=(int)(Math.random() * 6) + 1; //1~6사이의 난수 발생
            System.out.println(num);

            if(num == 6){
                break;
            }
        }

        System.out.println("프로그램 종료");
    }
}
```

[실행결과]

```
1
4
1
6
프로그램 종료
```

만약 반복문이 중첩되어 있을 경우 break문은 가장 가까운 반복문만 종료하고 바깥쪽 반복문은 종료시키지 않는다. 중첩된 반복문에서 바깥쪽 반복문까지 종료시키려면 바깥쪽 반복문에 이름(라벨)을 붙이고 "break 이름;"을 사용한다. 아래 예제를 보면 바깥쪽 for문은 'A'~'Z'까지 반복하고, 중첩된 for문은 'a'~'z'까지 반복하는데, 중첩된 for문에서 lower 변수가 'g'를 갖게 되면 바깥쪽 for문까지 빠져나오도록 했다.

[Sample.java] 바깥쪽 반복문 종료

```
public class Sample {
    public static void main(String[] args) {
        Outer:
        for(char upper = 'A'; upper <= 'Z'; upper++) {
            for(char lower = 'a'; lower <= 'z'; lower++) {
                System.out.println(upper + "-" + lower);
                if(lower == 'g'){
                    break Outer;
                }
            }
        }
        //바깥쪽 for문을 빠져나온다.
        System.out.println("프로그램 실행 종료");
    }
}
```

[실행결과]

```
A-a
A-b
A-c
A-d
A-e
A-f
A-g
프로그램 실행 종료
```

4.3.5 continue문

continue문은 반복문인 for문, while문, do-while문에서만 사용되는데, 블록 내부에서 continue문이 실행되면 for문의 증감식 또는 while문, do-while문의 조건식으로 이동한다. continue문은 반복문을 종료하지 않고 계속 반복을 수행한다는 점이 break문과 다르다. break문과 마찬가지로 continue문도 대개 if문과 같이 사용되는데, 특정 조건을 만족하는 경우에 continue문을 실행해서 그 이후의 문장을 실행하지 않고 다음 반복으로 넘어 간다. 아래 예제는 1에서 10 사이의 수 중에서 짝수만 출력하는 코드이다.

[Sample.java] continue를 사용한 while문

```
public class Sample {
    public static void main(String[] args) {
        for(int i=1; i<=10; i++) {
            //2로 나눈 나머지가 0이 아닐 경우, 즉 홀수인 경우
            if( i%2 != 0 ) {
                continue;
            }
            //홀수는 실행되지 않는다.
            System.out.println(i);
        }
    }
}
```

[실행결과]

```
2
4
6
8
10
```

5장 참조 타입

5.1 데이터 타입 분류

자바의 데이터 타입에는 크게 기본 타입(primitive type)과 참조 타입(reference type)으로 분류된다. 기본 타입이란 정수, 실수, 문자, 논리 리터럴을 저장하는 타입을 말한다. 참조 타입이란 객체의 번지를 참조하는 타입으로 배열, 열거, 클래스, 인터페이스 타입을 말한다. 기본 타입을 이용해서 선언된 변수는 실제 값을 변수 안에 저장하지만, 참조 타입인 배열, 열거, 클래스로 선언된 변수는 메모리의 번지를 값으로 갖는다.

5.2 메모리 사용 영역

java.exe로 JVM이 시작되면 JVM은 운영체제에서 할당받은 메모리영역을 메서드(Method), 힙(heap), JVM 스택(stack)으로 나눈다.

```
int[] scores = { 10, 20, 30 }
```



5.3 참조 변수의 ==, != 연산

기본 타입 변수의 ==, != 연산은 변수의 값이 같은지, 아닌지를 조사하지만 참조 타입 변수들 간의 ==, != 연산은 동일한 객체를 참조하는지, 다른 객체를 참조하는지 알아볼 때 사용된다. 참조 타입 변수의 값은 힙 영역의 객체 주소이므로 결국 주소 값을 비교하는 것이 된다.

5.4 null과 NullPointerException

참조 타입 변수는 힙 영역의 객체를 참조하지 않는다는 뜻으로 null 값을 가질 수 있다. null 값도 초기값으로 사용할 수 있기 때문에 null로 초기화된 참조 변수는 스택 영역에 생성된다. 실수로 null 값을 가지고 있는 참조 타입 변수를 사용하면 NullPointerException이 발생한다.

5.5 String 타입

변수에 문자열을 저장할 경우에는 문자열 리터럴을 사용하지만, new 연산자를 사용해서 직접 String 객체를 생성시킬 수도 있다. new 연산자는 힙 영역에 새로운 객체를 만들 때 사용하는 연산자로 객체 생성 연산자라고 한다. 동일한 문자열 리터럴로 String 객체를 생성했을 경우 == 연산의 결과는 true가 나오지만 new 연산자로 String 객체를 생성했을 경우 == 연산의 결과는 false가 나온다. 동일한 String 객체인 다른 String 객체인 상관없이 문자열만을 비교할 때에는 String 객체의 equals() 메서드를 사용해야 한다.

[Sample.java] 문자열 비교

```
public class Sample {
    public static void main(String[] args) {
        String strVar1 = "신민철";
        String strVar2 = "신민철";
        if(strVar1 == strVar2) {
            System.out.println("strVar1과 strVar2는 참조가 같음");
        } else {
            System.out.println("strVar1과 strVar2는 참조가 다름");
        }

        if(strVar1.equals(strVar2)) {
            System.out.println("strVar1과 strVar2는 문자열이 같음");
        }

        String strVar3 = new String("신민철");
        String strVar4 = new String("신민철");
        if(strVar3 == strVar4) {
            System.out.println("strVar3과 strVar4는 참조가 같음");
        } else {
            System.out.println("strVar3과 strVar4는 참조가 다름");
        }

        if(strVar1.equals(strVar2)) {
            System.out.println("strVar3과 strVar4는 문자열이 같음");
        }
    }
}
```

[실행결과]

strVar1과 strVar2는 참조가 같음
strVar1과 strVar2는 문자열이 같음
strVar3과 strVar4는 참조가 다름
strVar3과 strVar4는 문자열이 같음

5.6 배열 타입

5.6.1 배열이란?

변수는 한 개의 데이터만을 저장할 수 있으므로 같은 타입의 많은 양의 데이터를 다루는 프로그램에서는 좀 더 효율적인 방법이 필요한데 이것이 배열이다. 배열은 같은 타입의 데이터를 연속된 공간에 나열시키고, 각 데이터에 인덱스(index)를 부여해 놓은 자료구조이다.

5.6.2 배열 선언

대괄호 []는 배열 변수를 선언하는 기호로 사용되는데, 배열 변수는 참조 변수에 속한다. 배열도 객체이므로 힙 영역에 생성되고 배열 변수는 힙 영역의 배열 객체를 참조하게 된다. 참조할 배열 객체가 없다면 배열 변수는 null 값으로 초기화될 수 있다.

5.6.3 값 목록으로 배열 생성

배열 항목에 저장될 값의 목록이 있다면, 다음과 같이 간단하게 배열 객체를 만들 수 있다.

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

아래와 같이 매개 변수로 int[] 배열이 선언된 add() 메서드가 있을 경우, 값 목록으로 배열을 생성함과 동시에 add() 메서드의 매개값으로 사용하고자 할 때는 반드시 new 연산자를 사용해야 한다.

[Sample.java] 값의 리스트로 배열 생성

```
public class Sample {
    public static void main(String[] args) {
        int[] scores;
        scores = new int[] {83, 90, 87};
        int sum1 = 0;

        for(int i=0; i <3; i++) {
            sum1 += scores[i];
        }
        System.out.println("총합 : " + sum1);

        int sum2 = add( new int[] {83, 90, 87} );
        System.out.println("총합 : " + sum2);
        System.out.println();
    }

    public static int add(int[] scores) {
        int sum = 0;
        for(int i=0; i<3; i++) {
            sum += scores[i];
        }
        return sum;
    }
}
```

[실행결과]

총합 : 260
총합 : 260

5.6.4 new 연산자로 배열 생성

값의 목록을 가지고 있지 않지만, 향후 값들을 저장할 배열을 미리 만들고 싶다면 new 연산자로 아래와 같이 배열 객체를 생성시킬 수 있다.

```
타입[] 변수 = new 타입[길이]
```

길이는 배열이 저장할 수 있는 값의 수를 말한다. new 연산자로 배열을 생성할 경우에는 이미 배열 변수가 선언된 후에도 가능하다.

```
타입[] 변수 = null;
변수 = new 타입[길이];
```

배열이 생성되고 나서 새로운 값을 저장하려면 대입 연산자를 사용하면 된다.

```
int[] scores = new int[3]
scores[0] = 83;
scores[1] = 90;
scores[2] = 75;
```

[Sample.java] new 연산자로 배열 생성

```
public class Sample {
    public static void main(String[] args) {
        int[] arr1 = new int[3];
        for(int i=0; i<3; i++) {
            System.out.println("arr1[" + i + "] : " + arr1[i]);
        }

        arr1[0] = 10;
        arr1[1] = 20;
        arr1[2] = 30;
        for(int i=0; i<3; i++) {
            System.out.println("arr1[" + i + "] : " + arr1[i]);
        }

        double[] arr2 = new double[3];
        for(int i=0; i<3; i++) {
            System.out.println("arr2[" + i + "] : " + arr2[i]);
        }

        arr2[0] = 0.1;
        arr2[1] = 0.2;
        arr2[2] = 0.3;
        for(int i=0; i<3; i++) {
            System.out.println("arr2[" + i + "] : " + arr2[i]);
        }

        String[] arr3 = new String[3];
        for(int i=0; i<3; i++) {
            System.out.println("arr3[" + i + "] : " + arr3[i]);
        }

        arr3[0] = "1월";
        arr3[1] = "2월";
        arr3[2] = "3월";
        for(int i=0; i<3; i++) {
            System.out.println("arr3[" + i + "] : " + arr3[i]);
        }
    }
}
```

[실행결과]

```
arr1[0] : 0
arr1[1] : 0
arr1[2] : 0
arr1[0] : 10
arr1[1] : 20
arr1[2] : 30
arr2[0] : 0.0
arr2[1] : 0.0
arr2[2] : 0.0
arr2[0] : 0.1
arr2[1] : 0.2
arr2[2] : 0.3
arr3[0] : null
arr3[1] : null
arr3[2] : null
arr3[0] : 1월
arr3[1] : 2월
arr3[2] : 3월
```

5.6.5 배열 길이

배열의 길이란 배열에 저장할 수 있는 전체 항목 수를 말한다. 코드에서 배열의 길이를 구하려면 length 필드를 읽으면 된다. 참고로 필드는 객체 내부의 데이터를 말한다. 배열의 length 필드를 읽기 위해서는 배열 변수에 도트(.) 연산자를 붙이고 length를 적어주면 된다.

[Sample.java] new 연산자로 배열 생성

```
public class Sample {
    public static void main(String[] args) {
        int[] scores={83, 90, 87};
        int sum=0;
        for(int i=0; i<scores.length; i++) {
            sum += scores[i];
        }
        System.out.println("총합: " + sum);

        double avg=(double)sum / scores.length;
        System.out.println("평균: "+ avg);
    }
}
```

[실행결과]

```
총합: 260
평균: 86.66666666666667
```

5.6.6 커맨드 라인 입력

"java 클래스"로 실행하면 JVM은 길이가 0인 String 배열을 먼저 생성하고 main() 메서드를 호출할 때 매개값으로 전달한다. 만약 아래와 같이 "java 클래스" 뒤에 공백으로 구분된 문자열 목록을 주고 실행하면, String[]배열이 생성되고 main()메서드를 호출할 때 매개값으로 전달된다.

java 클래스 문자열0 문자열1 문자열2 ... 문자열 n-1

아래 예제를 그냥 실행하면 왼쪽 실행결과를 얻는다. 이유는 실행할 때 매개값을 주지 않았기 때문에 길이 0인 String 배열이 매개값으로 전달된다. args.length는 0이므로 3라인의 if 조건식이 true가 되어 if문의 블록이 실행된 것이다. 이클립스에서 프로그램을 실행할 때 매개값을 주고 실행하려면 메뉴에서 [Run-Run Configurations...]를 선택한다.

[Sample.java] main() 메서드의 매개 변수

```
public class Sample {
    public static void main(String[] args) {
        if(args.length != 2) { //입력된 데이터 개수가 두 개가 아닐 경우
            System.out.println("프로그램 사용법");
            System.out.println("java Sample num1 num20");
            System.exit(0); //프로그램 강제 종료
        }

        String strNum1 = args[0]; //첫번째 데이터 얻기
        String strNum2 = args[1]; //두번째 데이터 얻기
        int num1 = Integer.parseInt(args[0]); //문자열을 정수로 변환
        int num2 = Integer.parseInt(args[1]); //문자열을 정수로 변환
        int result = num1 + num2;
        System.out.println(num1 + " + " + num2 + " = " + result);
    }
}
```

[실행결과]

프로그램 사용법
java Sample num1 num20

[실행결과]

10 + 20 = 30

5.6.7 다차원 배열

자바는 2차원 배열을 중첩 배열 방식으로 구현한다. 예를 들어 2(행) x 3(열) 행렬을 만들기 위해 아래와 같은 코드를 사용한다.

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

scores[0]과 scores[1]은 모두 배열을 참조하는 변수 역할을 한다. 따라서 각 배열의 길이는 아래와 같이 얻을 수 있다.

```
scores.length //2
scores[0].length //3
scores[1].length //3
```

[Sample.java] 배열 속의 배열

```
public class Sample {
    public static void main(String[] args) {
        int[][] mathScores = new int[2][3];
        for(int i=0; i<mathScores.length; i++) {
            for(int k=0; k<mathScores[i].length; k++) {
                System.out.println("mathScores[" + i + "][" + k + "]="+ mathScores[i][k]);
            }
        }
        System.out.println();

        int[][] englishScores = new int[2][];
        englishScores[0] = new int[2];
        englishScores[1] = new int[3];
        for(int i=0; i<englishScores.length; i++) {
            for(int k=0; k<englishScores[i].length; k++) {
                System.out.println("englishScores[" + i + "][" + k + "]= " + englishScores[i][k]);
            }
        }
        System.out.println();

        int[][] javaScores = {{95, 80}, {92, 96, 80}};
        for(int i=0; i<javaScores.length; i++) {
            for(int k=0; k<javaScores[i].length; k++) {
                System.out.println("javaScores[" + i + "][" + k + "]= " + javaScores[i][k]);
            }
        }
        System.out.println();
    }
}
```

[실행결과]

mathScores[0][0]=0
mathScores[0][1]=0
mathScores[0][2]=0
mathScores[1][0]=0
mathScores[1][1]=0
mathScores[1][2]=0

englishScores[0][0]=0
englishScores[0][1]=0
englishScores[1][0]=0
englishScores[1][1]=0
englishScores[1][2]=0

javaScores[0][0]=95
javaScores[0][1]=80
javaScores[1][0]=92
javaScores[1][1]=96
javaScores[1][2]=80

5.6.8 객체를 참조하는 배열

기본 타입(int, float, double, char) 배열은 각 항목에 직접 값을 갖고 있지만, 참조 타입(클래스, 인터페이스) 배열은 각 항목에 객체의 번지를 가지고 있다. 예를 들어 String은 클래스 타입이므로 String[] 배열은 각 항목에 문자열이 아니라, String 객체의 주소를 가지고 있다.

[Sample.java] 객체를 참조하는 배열

```
public class Sample {
    public static void main(String[] args) {
        String[] strArray = new String[3];
        strArray[0] = "Java";
        strArray[1] = "Java";
        strArray[2] = new String("Java");
        System.out.println(strArray[0] == strArray[1]);
        System.out.println(strArray[0] == strArray[2]);
        System.out.println(strArray[0].equals(strArray[2]));
    }
}
```

[실행결과]

```
true
false
true
```

5.6.9 배열 복사

배열은 생성하면 크기를 변경할 수 없으므로 더 많은 저장 공간이 필요하면 보다 큰 배열을 만들고 이전 배열로부터 항목 값들을 복사해야 한다.

[Sample.java] for문으로 배열 복사

```
public class Sample {
    public static void main(String[] args) {
        int[] oldIntArray = {1, 2, 3};
        int[] newIntArray = new int[5];
        for(int i=0; i<oldIntArray.length; i++) {
            newIntArray[i] = oldIntArray[i];
        }

        for(int i=0; i<newIntArray.length; i++) {
            System.out.print(newIntArray[i] + ",");
        }
    }
}
```

[실행결과]

```
1,2,3,0,0,
```

[Sample.java] System.arraycopy()로 배열 복사

```
public class Sample {
    public static void main(String[] args) {
        String[] oldStrArray = {"java", "array", "copy"};
        String[] newStrArray = new String[5];
        System.arraycopy(oldStrArray, 0, newStrArray, 0, oldStrArray.length);

        for(int i=0; i<newStrArray.length; i++) {
            System.out.print(newStrArray[i] + ",");
        }
    }
}
```

[실행결과]

```
java,array,copy,null,null,
```

5.6.10 향상된 for문

배열 및 컬렉션 항목의 개수만큼 반복하고, 자동적으로 for문을 빠져나간다.

[Sample.java]

```
public class Sample {
    public static void main(String[] args) {
        int[] scores = {95, 71, 84, 93, 87};
        int sum = 0;
        for(int score : scores) {
            sum = sum + score;
        }
        System.out.println("점수 총합 = " + sum);

        double avg = (double)sum / scores.length;
        System.out.println("점수 평균 = " + avg);
    }
}
```

[실행결과]

```
점수 총합 = 430
점수 평균 = 86.0
```

5.7 열거 타입

데이터 중에는 몇 가지로 한정된 값만을 갖는 경우가 흔히 있다. 예를 들어 요일에 대한 데이터는 월, 화, 수, 목, 금, 토, 일이라는 일곱 개의 값만을 갖는다. 열거 타입은 몇 개의 열거 상수 중에서 하나의 상수를 저장하는 데이터 타입이다.

5.7.1 열거 타입 선언

열거 타입을 생성하려면 프로젝트를 선택한 다음 메뉴에서 [File]-[New]-[Enum]을 선택하고 [Name] 입력란에는 열거 타입 이름인 Week를 입력하고 열거 상수는 열거 타입의 값으로 사용되는데, 관례적으로 모두 대문자를 사용한다.

[Week.java] 열거 타입 선언

```
public enum Week {  
    MONDAY,  
    TUESDAY,  
    WEDNESDAY,  
    THURSDAY,  
    FRIDAY,  
    SATURDAY,  
    SUNDAY  
}
```

5.7.2 열거 타입 변수

열거 타입을 선언했다면 이제 열거 타입을 사용할 수 있다. 열거 타입도 하나의 데이터 타입이므로 변수를 선언하고 사용해야 한다. 예를 들어 열거 타입 Week로 변수를 선언하면 아래와 같다.

```
Week today;  
Week reservationDay;
```

열거 타입 변수를 선언했다면 아래와 같이 열거 상수를 저장할 수 있다. 열거 상수는 단독으로 사용할 수는 없고 반드시 열거타입, 열거상수로 사용된다. 예를 들어 today 열거 변수에 열거 상수인 SUNDAY를 저장하면 아래와 같다.

```
Week today = Week.SUNDAY;
```

[Sample.java] 열거 타입과 열거 상수

```
import java.util.Calendar;
```

```
public class Sample {  
    public static void main(String[] args) {  
        Week today = null; //열거 타입 변수 선언  
        Calendar cal = Calendar.getInstance();  
        int week = cal.get(Calendar.DAY_OF_WEEK); //일(1) ~ 토(7)까지의 숫자를 리턴  
  
        switch(week){  
            case 1:  
                today = Week.SUNDAY; break;  
            case 2:  
                today = Week.MONDAY; break;  
            case 3:  
                today = Week.TUESDAY; break;  
            case 4:  
                today = Week.WEDNESDAY; break;  
            case 5:  
                today = Week.THURSDAY; break;  
            case 6:  
                today = Week.FRIDAY; break;  
            case 7:  
                today = Week.SATURDAY; break;  
        }  
        System.out.println("오늘 요일: " + today);  
  
        if(today == Week.SUNDAY) {  
            System.out.println("일요일에는 축구를 합니다.");  
        }else {  
            System.out.println("열심히 자바를 공부합니다.");  
        }  
    }  
}
```

[실행결과]

오늘 요일: TUESDAY
열심히 자바를 공부합니다.

5.7.3 열거 객체의 메서드

열거 객체는 열거 상수의 문자열을 내부 데이터로 가지고 있다. 또한 모든 열거 타입은 컴파일 시에 Enum 클래스를 상속하게 되어 있기 때문에 java.lang.Enum 클래스에 선언된 모든 메서드들을 사용할 수 있다.

- name() 메서드는 열거 객체가 가지고 있는 문자열을 리턴 한다. 이때 리턴되는 문자열은 열거 타입을 정의할 때 사용한 상수이름과 동일하다.

```
Week today = Week.SUNDAY;
String name = today.name();
```

- ordinal() 메서드는 전체 열거 객체 중 몇 번째 열거 객체인지 알려준다. 열거 객체의 순번은 열거 타입을 정의할 때 주어진 순번을 말한다.

```
Week today = Week.SUNDAY;
int ordinal = today.ordinal();
```

- compareTo() 메서드는 매개값으로 주어진 열거 객체를 기준으로 전후로 몇 번째 위치하는지를 비교한다.

```
Week day1 = Week.MONDAY;
Week day2 = Week.WEDNESDAY;
int result1 = day1.compareTo(day2); //-2
int result2 = day2.compareTo(day1); //2
```

- valueOf() 메서드는 매개값으로 주어지는 문자열과 동일한 문자열을 가지는 열거 객체를 리턴 한다.

```
Week weekDay = Week.valueOf("SATURDAY");
```

- values() 메서드는 열거 타입의 모든 열거 객체들을 배열로 만들어 리턴 한다.

```
Week[] days = Week.values();
for(Week day: days){
    System.out.println(day);
}
```

[Sample.java] 열거 객체의 메서드

```
public class Sample {
    public static void main(String[] args) {
        Week today = Week.SUNDAY;
        String name = today.name();
        System.out.println(name);

        int ordinal = today.ordinal();
        System.out.println(ordinal);

        Week day1 = Week.MONDAY;
        Week day2 = Week.WEDNESDAY;
        int result1 = day1.compareTo(day2);
        int result2 = day2.compareTo(day1);
        System.out.println(result1);
        System.out.println(result2);

        if(args.length == 1) {
            String strDay = args[0];
            Week weekDay = Week.valueOf(strDay);
            if(weekDay == Week.SATURDAY || weekDay == Week.SUNDAY){
                System.out.println("주말 이군요");
            } else {
                System.out.println("평일 이군요");
            }
        }

        Week[] days = Week.values();
        for(Week day: days) {
            System.out.println(day);
        }
    }
}
```

[실행결과]

```
SUNDAY
6
-2
2
주말 이군요
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY
SUNDAY
```

6장 클래스

6.1 객체 지향 프로그래밍

소프트웨어를 개발할 때에는 부품에 해당하는 객체들을 먼저 만들고, 이것들을 하나씩 조립해서 완성된 프로그램을 만드는 기법을 객체 지향 프로그래밍(OOP: Object Oriented Programming)이라고 한다.

6.1.1 객체란?

객체란(Object)란 물리적으로 존재하거나 추상적으로 생각할 수 있는 것 중에서 자신의 속성을 가지고 다른 것과 식별 가능한 것을 말한다. 객체는 속성과 동작으로 구성되어 있으며 자바는 이 속성과 동작들을 각각 필드(field)와 메서드(method)라고 부른다.

6.1.2 객체의 상호작용

객체들은 각각 독립적으로 존재하고, 다른 객체와 서로 상호작용하면서 동작한다. 객체들 사이의 상호작용 수단은 메서드이다. 객체가 다른 객체의 기능을 이용하는 것이 바로 메서드 호출이다. 예를 들어 10과 20을 주고 더하기 기능을 이용한다고 했을 때 10과 20이 더하기 기능의 매개값이다. 리턴값은 메서드가 실행하고 나서 호출한 곳으로 돌려주는(리턴 하는) 값이다. 아래는 자바 코드로 본 메서드 호출이다.

```
int result = Calculator.add(10, 20); //리턴한 값을 int 변수에 저장
```

6.1.3 객체 간의 관계

객체는 개별적으로 사용할 수 있지만 대부분 다른 객체와 관계를 맺고 있다. 이 관계의 종류에는 집합 관계, 사용 관계, 상속 관계가 있다. 집합 관계가 있는 객체는 하나는 부품이고 하나는 완제품에 해당한다. 사용 관계는 객체 간의 상호 작용을 말한다. 객체는 다른 객체의 메서드를 호출하여 원하는 결과를 얻어낸다. 상속 관계는 상위(부모) 객체를 기반으로 하위(자식) 객체를 생성하는 관계를 말한다.

6.1.4 객체 지향 프로그래밍의 특징

객체 지향 프로그램의 특징으로는 캡슐화, 상속, 다형성을 들 수 있다.

- 캡슐화(Encapsulation)

캡슐화된 객체의 필드, 메서드를 하나로 묶고, 실제 구현 내용을 감추는 것을 말한다. 외부 객체는 객체 내부의 구조를 알지 못하며 객체가 노출해서 제공하는 필드와 메서드만 이용할 수 있다. 필드와 메서드를 캡슐화하여 보호하는 이유는 외부의 잘못된 사용으로 인해 객체가 손상되지 않도록 하는데 있다. 자바 언어는 캡슐화된 멤버를 노출시킬 것인지, 숨길 것인지를 결정하기 위해 접근 제한자(Access Modifier)를 사용한다.

- 상속(Inheritance)

일반적으로 상속은 부모가 가지고 있는 재산을 자식에게 물려주는 것을 말한다. 자식은 특별한 노력 없이 부모가 물려준 재산을 갖게 된다. 객체 지향 프로그래밍에서도 부모 역할의 상위 객체와 자식 역할의 하위 객체가 있다. 상위 객체는 자기가 가지고 있는 필드와 메서드를 하위 객체에게 물려주어 하위 객체가 사용할 수 있도록 해준다.

- 다형성(Polymorphism)

다형성은 같은 타입이지만 실행 결과가 다양한 객체를 이용할 수 있는 성질을 말한다. 코드 측면에서 보면 다형성은 하나의 타입에 여러 객체를 대입함으로써 다양한 기능을 이용할 수 있도록 해준다. 예를 들어 자동차를 설계할 때 타이어 인터페이스 타입을 적용했다면 이 인터페이스를 구현한 실제 타이어들은 어떤 것이든 상관없이 정칙(대입)이 가능하다.

6.2 객체와 클래스

자바에서는 설계도가 바로 클래스(class)이다. 클래스는 객체를 생성하기 위한 필드와 메서드가 정의되어 있다. 클래스로부터 만들어진 객체를 해당 클래스의 인스턴스(instance)라고 한다. 자동차 객체는 자동차 클래스의 인스턴스인 셈이다. 그리고 클래스로부터 객체를 만드는 과정을 인스턴스화라고 한다. 객체 지향 프로그래밍 개발은 세 가지 단계가 있다. 첫 번째 단계는 클래스를 설계해야 한다. 두 번째 단계는 설계된 클래스를 가지고 사용할 객체를 생성해야 한다. 그리고 마지막 단계는 생성된 객체를 이용하는 것이다.

6.3 클래스 선언

클래스 이름은 한글이든 영어든 상관없지만, 한글로 클래스 이름을 만드는 경우는 거의 없다. 자바 언어는 영어 대소문자를 다른 문자로 취급하기 때문에 클래스 이름도 영어 대소문자를 구분한다. 관례적으로 클래스 이름이 단일 단어라면 첫 자를 대문자로 하고 나머지는 소문자로 작성한다. 만약 서로 다른 단어가 혼합된 이름을 사용한다면 각 단어의 첫 머리 글자는 대문자로 적정하는 것이 관례이다.

```
Calculator, Car, Member, ChatClient, ChatServer, Web_Browser
```

클래스 이름을 정했다면 "클래스이름.java"로 소스 파일을 생성한다. 소스 파일 이름 역시 대소문자를 구분하므로 반드시 클래스 이름과 대소문자가 같도록 해야 한다. 소스 파일을 생성했다면 소스 파일을 열고 아래와 같이 클래스를 선언한다. `public class` 키워드는 클래스를 선언할 때 사용하면 반드시 소문자로 작성해야 한다. 클래스 이름 뒤에는 반드시 중괄호 `{ }`를 붙여 클래스의 시작과 끝을 알려준다.

```
public class 클래스이름 {
    ...
}
```

6.4 객체 생성과 클래스 변수

클래스를 선언한 다음 컴파일을 했다면 객체를 생성할 설계도가 만들어진 셈이다. 클래스로부터 객체를 생성하는 방법은 아래와 같이 `new` 연산자를 사용하면 된다. `new`는 클래스로부터 객체를 생성하는 연산자이고 `new` 연산자로 생성된 객체는 메모리 힙(heap) 영역에 생성된다. 힙 영역에 객체를 생성시킨 후 객체의 주소를 리턴해 참조 타입인 클래스 변수에 저장해 변수를 통해 객체를 사용한다.

```
클래스 변수;
변수 = new 클래스();
```

아래 예제는 `Student` 클래스를 선언하고 `Sample` 클래스의 `main()` 메서드에서 `Student` 객체를 생성한다.

[Student.java] 클래스 선언

```
public class Student {
}
```

[Sample.java] 클래스로부터 객체 생성

```
public class Sample {
    public static void main(String[] args) {
        Student s1 = new Student();
        System.out.println("s1 변수가 Student 객체를 참조합니다.");

        Student s2 = new Student();
        System.out.println("s2 변수가 또 다른 Student 객체를 참조합니다.");
    }
}
```

[실행결과]

s1 변수가 Student 객체를 참조합니다.
s2 변수가 또 다른 Student 객체를 참조합니다.

예제가 실행되면 `Student` 클래스는 하나지만 `new` 연산자를 사용한 만큼 객체가 메모리에 생성된다. 이러한 객체들은 `Student` 클래스의 인스턴스들이다. 비록 같은 클래스로부터 생성되었지만 각각의 `Student` 객체는 자신만의 고유 데이터를 가지면서 메모리에서 활동한다. `s1`과 `s2`가 참조하는 `Student` 객체는 완전히 독립된 서로 다른 객체이다.

6.5 클래스의 구성 멤버

클래스에는 객체가 가져야 할 구성 멤버가 선언된다. 구성 멤버에는 필드(Field), 생성자(Constructor), 메서드(Method)가 있다. 이 구성 멤버들은 생략되거나 복수 개가 작성될 수 있다.

```
public class ClassName {
    //필드(Field) 객체의 데이터가 저장되는 곳
    int fieldName;

    //생성자(Constructor) 객체 생성 시 초기화 역할 담당
    ClassName() { ... }

    //메서드(Method) 객체의 동작에 해당하는 블록
    void methodName() { ... }
}
```

6.5.1 필드

필드는 객체의 고유 데이터, 부품 객체, 상태 정보를 저장하는 곳이다. 선언 형태는 변수(variable)와 비슷하지만 필드를 변수라고 부르지 않는다. 변수는 생성자와 메서드 내에서만 사용되고 생성자와 메서드가 실행 종료되면 자동 소멸된다. 하지만 필드는 생성자와 메서드 전체에서 사용되며 객체가 소멸되지 않는 한 객체와 함께 존재한다.

6.5.2 생성자

생성자는 new 연산자로 호출되는 특별한 중괄호 { } 블록이다. 생성자의 역할은 객체 생성 시 초기화를 담당한다. 필드를 초기화하거나 메서드를 호출해서 객체를 사용할 준비를 한다. 생성자는 메서드와 비슷하게 생겼지만 클래스 이름으로 되어 있고 리턴 타입이 없다.

6.5.3 메서드

메서드는 객체의 동작에 해당하는 중괄호 { } 블록을 말한다. 중괄호 블록은 이름을 가지고 있는데 이것이 메서드 이름이다. 메서드를 호출하게 되면 중괄호 블록에 있는 모든 코드들이 일괄적으로 실행된다. 메서드는 필드를 읽고 수정하는 역할도 하지만 다른 객체를 생성해서 다양한 기능을 수행하기도 한다. 메서드는 객체 간의 데이터 전달의 수단으로 사용된다. 외부로부터 매개값을 받을 수도 있고, 실행 후 어떤 값을 리턴 할 수도 있다.

6.6 필드

필드(Field)는 객체의 고유 데이터, 객체가 가져야 할 부품, 객체의 현재 상태 데이터를 저장하는 곳이다. 자동차 객체를 예로 들어 보면 제작회사, 모델, 색깔, 최고 속도는 고유 데이터에 해당하고, 현재 속도, 엔진 회전수는 상태 데이터에 해당한다. 그리고 차체, 엔진, 타이어는 부품에 해당한다. 따라서 자동차 클래스를 설계할 때 이 정보들은 필드로 선언되어야 한다.

6.6.1 필드 선언

필드 선언은 클래스 중괄호 { } 블록 어디서든 존재할 수 있다. 생성자 선언과 메서드 선언의 앞과 뒤 어떤 곳에서도 필드 선언이 가능하다. 하지만 생성자와 메서드 중괄호 블록 내부에는 선언될 수 없다. 필드 선언은 변수 선언 형태와 비슷하고 클래스 멤버 변수라고 부르기도 한다.

```
String company = "현대자동차";
String model = "그랜저";
int maxSpeed = 300;
int productionYear;
int currentSpeed;
boolean engineStart;
```

초기값이 지정되지 않은 필드들은 객체 생성 시 자동으로 기본 초기값으로 설정된다. 정수 타입 필드는 0, 실수 타입 필드는 0.0 그리고 boolean 필드는 false로 초기화되는 것을 볼 수 있다. 참조 타입은 객체를 참조하고 있지 않은 상태인 null로 초기화된다.

6.6.2 필드 사용

필드를 사용한다는 것은 필드값을 읽고 변경하는 작업을 말한다. 클래스 내부의 생성자나 메서드에서 사용할 경우 단순히 필드 이름으로 읽고 변경하면 되지만, 클래스 외부에서 사용할 경우 우선적으로 클래스로부터 객체를 생성한 두 필드를 사용해야 한다. 그 이유는 필드는 객체에 소속된 데이터이므로 객체가 존재하지 않으면 필드도 존재하지 않기 때문이다.

[Car.java] Car 클래스 필드 선언

```
public class Car {
    //Car class의 필드 정의
    String company = "현대자동차";
    String model = "그랜저";
    String color = "검정";
    int maxSpeed = 350;
    int speed;
}
```

[Sample.java] 외부 클래스에서 Car 필드값 읽기와 변경

```
public class Sample {
    public static void main(String[] args) {
        //객체 생성
        Car myCar = new Car();

        //필드값 읽기
        System.out.println("제작회사: " + myCar.company);
        System.out.println("모델명: " + myCar.model);
        System.out.println("색깔: " + myCar.color);
        System.out.println("최고속도: " + myCar.maxSpeed);
        System.out.println("현재속도: " + myCar.speed);

        //필드값 변경
        myCar.speed = 60;
        System.out.println("수정된 속도: " + myCar.speed);
    }
}
```

[실행결과]

```
제작회사: 현대자동차
모델명: 그랜저
색깔: 검정
최고속도: 350
현재속도: 0
수정된 속도: 60
```

Car 클래스는 speed 필드 선언 시 초기값을 주지 않았다. 그러나 출력해보면 기본값인 0이 들어있는 것을 볼 수 있다. 아래 예제는 여러 가지 타입의 필드가 어떤 값으로 자동 초기화되는지 확인시켜준다.

[FieldValue.java] 필드 자동 초기화

```
public class FieldValue {
    //FieldValue class의 필드 정의
    int intField;
    boolean booleanField;
    char charField;
    float floatField;
    double doubleField;
    int[] arrField;
    String stringField;
}
```

[Sample.java] 필드값 출력

```
public class Sample {
    public static void main(String[] args) {
        FieldValue fv=new FieldValue();
        System.out.println("intFiled:" + fv.intField);
        System.out.println("booleanField:" + fv.booleanField);
        System.out.println("charField:" + fv.charField);
        System.out.println("floatField:" + fv.floatField);
        System.out.println("doubleField:" + fv.doubleField);
        System.out.println("arrField:" + fv.arrField);
        System.out.println("stringField:" + fv.stringField);
    }
}
```

[실행결과]

```
intFiled:0
booleanField:false
charField:
floatField:0.0
doubleField:0.0
arrField:null
stringField:null
```

6.7 생성자

생성자(Constructor)는 new 연산자와 같이 사용되어 클래스로부터 객체를 생성할 때 호출되어 객체의 초기화를 담당한다. new 연산자에 의해 생성자가 성공적으로 실행되면 힙(heap) 영역에 객체가 생성되고 객체의 주소가 리턴 된다. 리턴된 객체의 주소는 클래스 타입 변수에 저장되어 객체에 접근할 때 이용된다. 만약 생성자가 성공적으로 실행되지 않고 예외(에러)가 발생했다면 객체는 생성되지 않는다.

6.7.1 기본 생성자

모든 클래스는 생성자가 반드시 존재하며 하나 이상을 가질 수 있다. 우리가 클래스 내부에 생성자 선언을 생략했다면 컴파일러는 중괄호 { } 블록 내용이 비어있는 기본 생성자(Default Constructor)를 바이트 코드에 자동 추가한다. 그러나 클래스에 명시적으로 선언한 생성자가 한 개라도 있으면 컴파일러는 기본 생성자를 추가하지 않는다.

6.7.2 생성자 선언

생성자는 메서드와 비슷한 모양을 가지고 있으나 리턴 타입이 없고 클래스 이름과 동일하다. 생성자 블록 내부에는 객체 초기화 코드가 작성되는데, 일반적으로 필드에 초기값을 저장하거나 메서드를 호출하여 객체 사용 전에 필요한 준비를 한다. 클래스에 생성자가 명시적으로 선언되어 있을 경우에는 반드시 선언된 생성자를 호출해서 객체를 생성해야만 한다. 아래 예제를 보면 Car 클래스에 생성자를 선언했기 때문에 기본 생성자 (Car())를 호출해서 객체를 생성할 수 없고 Car(String color, int cc)를 호출해서 객체를 생성해야 한다.

[Car.java] 생성자 선언

```
public class Car {
    //생성자
    Car(String color, int cc) {

    }
}
```

[Sample.java] 생성자를 호출해서 객체 생성

```
public class Sample {
    public static void main(String[] args) {
        //Car mayCar = new Car() 기본 생성자를 호출할 수 없다.
        Car myCar = new Car("검정", 3000);
    }
}
```

6.7.3 필드 초기화

클래스로부터 객체가 생성될 때 필드는 기본 초기값으로 자동 설정된다. 만약 다른 값으로 초기화를 하고 싶다면 두 가지 방법이 있다. 하나는 필드를 선언할 때 초기값을 주는 방법이고, 또 다른 하나는 생성자에서 초기값을 주는 방법이다. 필드를 선언할 때 초기값을 주게 되면 동일한 클래스로부터 생성되는 객체들은 모두 같은 데이터를 갖게 된다. 하지만 객체 생성 시점에 값들을 초기화 하려면 생성자에서 초기화를 해야 한다.

[Korean.java] 생성자에서 필드 초기화

```
public class Korean {
    //필드 정의
    String nation = "대한민국";
    String name;
    String ssn;
    //생성자 정의
    public Korean(String n, String s) {
        name = n;
        ssn = s;
    }
}
```

[Sample.java] 객체 생성 후 필드값 출력

```
public class Sample {
    public static void main(String[] args) {
        Korean k1 = new Korean("박자바", "011225-1234567");
        System.out.println("k1.name: " + k1.name);
        System.out.println("k1.ssn: " + k1.ssn);
        Korean k2 = new Korean("김자바", "930525-0654321");
        System.out.println("k2.name: " + k2.name);
        System.out.println("k2.ssn: " + k2.ssn);
    }
}
```

[실행결과]

```
k1.name: 박자바
k1.ssn: 011225-1234567
k2.name: 김자바
k2.ssn: 930525-0654321
```

매개 변수의 이름이 너무 짧으면 코드의 가독성이 좋지 않기 때문에 가능하면 초기화시킬 필드 이름과 비슷하거나 동일한 이름을 사용한다. 필드와 동일한 매개 변수를 사용하는 경우에는 필드와 매개 변수의 이름이 동일하기 때문에 생성자 내부에서 해당 필드에 접근할 수 없다. 해결 방법은 필드 앞에 "this."를 붙이면 된다. "this.필드"는 this라는 참조 변수로 필드를 사용하는 것과 동일하다.

6.7.4 생성자 오버로딩(Overloading)

Car 객체를 생성시 외부에서 제공되는 데이터가 없다면 기본 생성자로 Car 객체를 생성해야 한다. 외부에서 model 데이터가 제공되거나 model과 color가 제공될 경우에도 Car 객체를 생성할 수 있어야 한다. 생성자가 하나뿐이라면 이러한 요구 조건을 수용할 수 없지만 자바는 다양한 방법으로 객체를 생성할 수 있도록 오버로딩(Overloading)을 제공한다. 생성자 오버로딩은 매개변수를 달리하는 생성자를 여러 개 선언하는 것이다.

[Car.java] 생성자의 오버로딩

```
public class Car {
    String company = "현대자동차";
    String model;
    String color;
    int maxSpeed;

    Car() {
    }

    Car(String model) {
        this.model = model;
    }

    Car(String model, String color) {
        this.model = model;
        this.color = color;
    }

    Car(String model, String color, int maxSpeed) {
        this.model = model;
        this.color = color;
        this.maxSpeed = maxSpeed;
    }
}
```

```

public class Sample {
    public static void main(String[] args) {
        Car car1 = new Car();           //①생성자 선택
        System.out.println("car1.company: " + car1.company);
        System.out.println();

        Car car2 = new Car("자가용");   //②생성자 선택
        System.out.println("car2.company: " + car2.company);
        System.out.println("car2.model: " + car2.model);
        System.out.println();

        Car car3 = new Car("자가용", "빨강"); //③생성자 선택
        System.out.println("car3.company: " + car3.company);
        System.out.println("car3.model: " + car3.model);
        System.out.println("car3.color: " + car3.color);
        System.out.println();

        Car car4 = new Car("택시", "검정", 200); //④생성자 선택
        System.out.println("car4.company: " + car4.company);
        System.out.println("car4.model: " + car4.model);
        System.out.println("car4.color: " + car4.color);
        System.out.println("car4.maxSpeed: " + car4.maxSpeed);
        System.out.println();
    }
}

```

[실행결과]

```

car1.company: 현대자동차
car2.company: 현대자동차
car2.model: 자가용
car3.company: 현대자동차
car3.model: 자가용
car3.color: 빨강
car4.company: 현대자동차
car4.model: 택시
car4.color: 검정
car4.maxSpeed: 200

```

6.7.5 다른 생성자 호출(this())

생성자 오버로딩이 많아질 경우 생성자 간의 중복된 코드가 발생할 수 있다. 매개 변수의 수만 달리고 필드 초기화 내용이 비슷한 생성자에서 이러한 현상을 많이 볼 수 있다. 이 경우에는 필드 초기화 내용은 한 생성자에만 집중적으로 작성하고 나머지 생성자는 초기화 내용을 가지고 있는 생성자를 호출하는 방법으로 개선할 수 있다. 생성자에서 다른 생성자를 호출할 때는 아래와 같이 this()코드를 사용한다.

```

클래스 ( [ 매개변수 선언, ... ] ) {
    this(매개변수, ..., 값, ...);    //클래스의 다른 생성자 호출
}

```

this()는 자신의 다른 생성자를 호출하는 코드로 생성자의 첫줄에만 허용된다. this()의 매개값은 호출되는 생성자의 매개 변수 타입에 맞게 제공해야 한다. this() 다음에는 추가적인 실행문들이 올 수 있다. 아래 예제는 this()를 사용해서 마지막 생성자인 Car(String model, String color, int maxSpeed)를 호출하도록 수정해서 중복 코드를 최소화하였다.

```

public class Car {
    //필드 정의
    String company = "현대자동차";
    String model;
    String color;
    int maxSpeed;

    //생성자 정의
    Car() {
    }
    Car(String model) {
        this(model, "은색", 250);
    }
    Car(String model, String color) {
        this(model, color, 250);
    }

    //공통 실행 코드
    Car(String model, String color, int maxSpeed) {
        this.model = model;
        this.color = color;
        this.maxSpeed = maxSpeed;
    }
}

```

6.8 메서드

6.8.1 메서드 선언

메서드 선언은 선언부(리턴 타입, 메서드이름, 매개변수선언)와 실행 블록으로 구성된다. 메서드 선언부를 메서드 signature라고도 한다.

```
리턴타입 메서드이름 ([매개변수 선언 ... ]) {  
    실행할 코드를 작성하는 곳 (메서드 실행 블록)  
}
```

• 리턴 타입

리턴 타입은 메서드가 실행 후 리턴하는 값의 타입을 말한다. 메서드가 실행 후 결과를 호출한 곳에 넘겨줄 경우에는 리턴값이 있어야 한다. 예를 들어 전자계산기 객체에서 전원을 켜는 `powerOn()` 메서드는 전원만 켜면 되므로 리턴값이 없고, `divide()` 메서드는 결과를 리턴 해야 한다.

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

리턴값이 있느냐 없느냐에 따라 메서드를 호출하는 방법이 조금 다르다. `powerOn()` 메서드는 리턴값이 없기 때문에 변수에 저장할 내용이 없어 단순히 메서드만 호출하면 된다. 그러나 `divide()` 메서드는 10을 20으로 나눈 후 0.5를 리턴하므로 이것을 저장할 변수가 있어야 한다.

```
void powerOn() { ... }  
double result = divide(10, 20)
```

• 매개 변수 선언

매개 변수는 메서드가 실행할 때 필요한 데이터를 외부로부터 받기 위해 사용된다. `powerOn()`과 `powerOff()` 메서드는 그냥 전원만 켜고 끄면 되므로 매개 변수가 필요 없고, `plus()`, `divide()` 메서드는 매개 변수가 두 개 필요하다.

[Calculator.java] 메서드 선언

```
public class Calculator {  
    //메서드 정의  
    void powerOn() {  
        System.out.println("전원을 켭니다.");  
    }  
    int plus(int x, int y) {  
        int result = x + y;  
        return result;  
    }  
    double divide(int x, int y) {  
        double result = (double)x / (double)y;  
        return result;  
    }  
    void powerOff() {  
        System.out.println("전원을 끕니다.");  
    }  
}
```

[Sample.java] 메서드 호출

```
public class Sample {  
    public static void main(String[] args) {  
        Calculator myCalc = new Calculator();  
        myCalc.powerOn();  
  
        int result1 = myCalc.plus(5, 6);  
        System.out.println("result1: " + result1);  
  
        int x = 10;  
        int y = 4;  
        double result2 = myCalc.divide(x, y);  
        System.out.println("result2: " + result2);  
  
        myCalc.powerOff();  
    }  
}
```

[실행결과]

```
전원을 켭니다.  
result1: 11  
result2: 2.5  
전원을 끕니다.
```


- 매개 변수의 수를 모를 경우

메서드의 매개 변수는 개수가 이미 정해져 있는 것이 일반적이지만, 경우에 따라서는 메서드를 선언할 때 매개 변수의 개수를 알 수 없는 경우가 있다. 해결책은 아래와 같이 매개 변수를 배열 타입으로 선언하는 것이다. sum1() 메서드를 호출할 때 배열을 넘겨줌으로써 배열의 항목 값들을 모두 전달할 수 있다. 배열의 항목 수는 호출할 때 결정된다.

```
int values = { 1, 2, 3 };
int result = sum1(values);
int result = sum1(new int[] { 1, 2, 3, 4, 5 });
```

매개 변수를 배열 타입으로 선언하면, 메서드를 호출하기 전에 배열을 생성해야 하는 불편한 점이 있다. 그래서 배열을 생성하지 않고 값의 리스트만 넘겨주는 방법도 있다. 아래와 같이 sum2() 메서드의 매개 변수를 "... "를 사용해서 선언하게 되면 메서드 호출 시 넘겨준 값의 수에 따라 자동으로 배열이 생성되고 매개값으로 사용된다.

```
int sum2(int ... values) { ... }
```

"..."로 선언된 매개 변수의 값은 아래와 같이 메서드 호출시 리스트로 나열해주면 된다.

```
int result = sum2(1, 2, 3);
int result = sum2(1, 2, 3, 4, 5);
```

"..."로 선언된 매개 변수는 배열 타입이므로 아래와 같이 배열을 직접 매개값으로 사용해도 좋다.

```
int values = { 1, 2, 3 };
int result = sum1(values);
int result = sum1(new int[] { 1, 2, 3, 4, 5 });
```

아래는 매개 변수를 배열로 선언한 sum1()과 매개 변수를 "... "로 선언한 sum2()의 작성법이다. 둘 다 항목의 값을 모두 더해서 리턴 한다.

[Computer.java] 매개 변수의 수를 모를 경우

```
public class Computer {
    int sum1(int[] values) {
        int sum = 0;
        for(int i=0; i<values.length; i++) {
            sum += values[i];
        }
        return sum;
    }
    int sum2(int ... values) {
        int sum = 0;
        for(int i=0; i<values.length; i++) {
            sum += values[i];
        }
        return sum;
    }
}
```

[Sample.java] 매개 변수의 수를 모를 경우

```
public class Sample {
    public static void main(String[] args) {
        Computer myCom = new Computer();
        int[] values1 = {1, 2, 3};
        int result1 = myCom.sum1(values1);
        System.out.println("result1: " + result1);

        int result2 = myCom.sum1(new int[] {1, 2, 3, 4, 5});
        System.out.println("result2: " + result2);

        int result3 = myCom.sum2(1, 2, 3);
        System.out.println("result3: " + result3);

        int result4 = myCom.sum2(1, 2, 3, 4, 5);
        System.out.println("result4: " + result4);
    }
}
```

[실행결과]

```
result1: 6
result2: 15
result3: 6
result4: 15
```

6.8.2 리턴(return)문

- 리턴값이 있는 메서드: 메서드 선언에 리턴 타입이 있는 메서드는 반드시 리턴(return)문을 사용해서 리턴값을 지정해야 한다. 만약 return문이 없다면 컴파일 오류가 발행한다. return문이 실행되면 메서드는 즉시 종료된다.
- 리턴값이 없는 메서드(void): void로 선언된 리턴값이 없는 메서드에서도 return문을 사용하면 메서드 실행을 강제 종료 시킨다.

아래는 gas 값이 0보다 크면 계속해서 while문을 실행하고, 0일 경우 return문을 실행해서 run()메서드를 종료한다. while문이 한 번 루핑할 때마다 gas를 1씩 감소하기 때문에 언젠가는 0이 되어 run()메서드를 종료한다. 이 예제에서는 return문 대신 break문을 사용할 수 있다.

[Car.java] return문

```
public class Car {
    //필드정의
    int gas;

    //메서드정의

    void setGas(int gas) {
        this.gas = gas;
    } //리턴값이 없는 메서드로 매개값을 받아서 gas 필드값을 변경

    boolean isLeftGas() {
        if(gas == 0) {
            System.out.println("gas가 없습니다.");
            return false; //false를 리턴
        }
        System.out.println("gas가 있습니다.");
        return true; //true를 리턴
    } //리턴값이 boolean인 메서드로 gas 필드값이 0이면 false를, 0이 아니면 true를 리턴

    void run() {
        while(true) {
            if(gas > 0) {
                System.out.println("달립니다.(gas잔량:" + gas + ")");
                gas -= 1;
            } else {
                System.out.println("멈춥니다.(gas잔량:" + gas + ")");
                return; //메서드 실행 종료
            }
        }
    }
}
```

[Sample.java] return문

```
public class Sample {
    public static void main(String[] args) {
        Car myCar = new Car();

        //Car의 setGas() 메서드 호출
        myCar.setGas(5);

        //Car의 isLeftGas() 메서드 호출
        boolean gasState = myCar.isLeftGas();
        if(gasState) {
            System.out.println("출발합니다.");
            myCar.run(); //Car의 run() 메서드 호출
        }

        if(myCar.isLeftGas()) { //Car의 isLeftGas() 메서드 호출
            System.out.println("gas를 주입할 필요가 없습니다.");
        } else {
            System.out.println("gas를 주입하세요.");
        }
    }
}
```

[실행결과]

```
gas가 있습니다.
출발합니다.
달립니다.(gas잔량:5)
달립니다.(gas잔량:4)
달립니다.(gas잔량:3)
달립니다.(gas잔량:2)
달립니다.(gas잔량:1)
멈춥니다.(gas잔량:0)
gas가 없습니다.
gas를 주입하세요.
```

6.8.3 메서드 호출

메서드는 클래스 내, 외부의 호출에 의해 실행한다. 클래스 내부의 다른 메서드에서 호출할 경우에는 단순한 메서드 이름으로 호출하면 되지만, 클래스 외부에서 호출할 경우에는 우선 클래스로부터 객체를 생성한 뒤, 참조 변수를 이용해서 메서드를 호출해야 한다.

- 객체 내부에서 호출

클래스 내부에서 다른 메서드를 호출할 경우에는 메서드가 매개변수를 가지고 있을 때에는 매개 변수의 타입과 수에 맞게 매개값을 제공한다.

[Calculator.java] 클래스 내부에서 메서드 호출

```
public class Calculator {
    int plus(int x, int y){
        int result = x + y
        return result
    }
    double avg(int x, int y) {
        double sum = plus(x, y);
        double result = sum / 2;
        return result;
    }
    void execute() {
        double result = avg(7, 10);
        println("실행결과:" + result);
    }
    void println(String message) {
        System.out.println(message);
    }
}
```

[Sample.java] Calculator의 execute() 실행

```
public class Sample {
    public static void main(String[] args) {
        Calculator myCalc = new Calculator();
        myCalc.execute(); //Calculator의 execute() 메서드 호출
    }
}
```

[실행결과]

실행결과:8.5

- 객체 외부에서 호출

외부에서 메서드를 호출하려면 객체를 먼저 생성한다. 메서드는 객체에 소속된 멤버이므로 객체가 존재하지 않으면 메서드가 존재하지 않기 때문이다.

[Car.java] 클래스 외부에서 메서드 호출

```
public class Car {
    //필드정의
    int speed;

    //메서드정의
    int getSpeed() {
        return speed;
    }
    void keyTurnOn() {
        System.out.println("키를 돌립니다.");
    }
    void run(){
        for(int i=10; i<=50; i+=10) {
            speed = i;
            System.out.println("달립니다.(시속:" + speed + "km/h)");
        }
    }
}
```

[Sample.java] 클래스 외부에서 메서드 호출

```
public class Sample {
    public static void main(String[] args) {
        Car myCar = new Car();

        myCar.keyTurnOn();
        myCar.run();

        int speed = myCar.getSpeed();
        System.out.println("현재 속도:" + speed + "km/h");
    }
}
```

[실행결과]

키를 돌립니다.
달립니다.(시속:10km/h)
달립니다.(시속:20km/h)
달립니다.(시속:30km/h)
달립니다.(시속:40km/h)
달립니다.(시속:50km/h)
현재 속도:50km/h

6.8.4 메서드 오버로딩

클래스 내에 같은 이름의 메서드를 여러 개 선언하는 것을 메서드 오버로딩(overloading)이라고 한다. 메서드를 오버로딩할 때 주의할 점은 매개 변수의 타입과 개수, 순서가 똑같은 경우 매개 변수 이름만 바꾸는 것은 메서드 오버로딩이라 볼 수 없다. 또한 리턴 타입만 다르고 매개 변수가 동일하다면 이것은 오버로딩이 아니다. 아래 예제를 보면 Calculator 클래스에 areaRectangle() 메서드를 오버로딩해서 매개값이 한 개면 정사각형의 넓이를, 두 개이면 직사각형의 넓이를 계산하여 리턴 하도록 한다.

[Calculator.java] 메서드의 오버로딩

```
public class Calculator {
    double areaRectangle(double width) { //정사각형 넓이
        return width * width;
    }

    double areaRectangle(double width, double height) { //직사각형 넓이
        return width * height;
    }
}
```

[Sample.java] 메서드의 오버로딩

```
public class Sample {
    public static void main(String[] args) {
        Calculator myCalcu = new Calculator();
        double result1 = myCalcu.areaRectangle(10); //정사각형의 넓이 구하기
        double result2 = myCalcu.areaRectangle(10, 20); //직사각형의 넓이 구하기

        System.out.println("정사각형 넓이=" + result1);
        System.out.println("직사각형 넓이=" + result2);
    }
}
```

[실행결과]

정 사각형 넓이=100.0
직 사각형 넓이=200.0

6.9 인스턴스 멤버와 this

인스턴스(instance) 멤버란 객체(인스턴스)를 생성한 후 사용할 수 있는 필드와 메서드를 말하는데, 이들을 각각 인스턴스 필드, 인스턴스 메서드라고 부른다. 인스턴스 필드와 메서드는 객체에 소속된 멤버이기 때문에 객체 없이는 사용할 수 없다. 객체 외부에서 인스턴스 멤버에 접근하기 위해 참조 변수를 사용하는 것과 마찬가지로 객체 내부에서도 인스턴스 멤버에 접근하기 위해 this를 사용할 수 있다.

[Car.java] 인스턴스 멤버와 this

```
public class Car {
    String model;
    int speed;

    //생성자
    Car(String model) {
        this.model = model;
    }

    //메서드
    void setSpeed(int speed) {
        this.speed = speed;
    }
    void run(){
        for(int i=10; i<=50; i+=10) {
            this.setSpeed(i);
            System.out.println(this.model + "가 달립니다.(시속: " + this.speed + "km/h)");
        }
    }
}
```

[실행결과]

포르쉐가 달립니다.(시속:10km/h)
포르쉐가 달립니다.(시속:20km/h)
포르쉐가 달립니다.(시속:30km/h)
포르쉐가 달립니다.(시속:40km/h)
포르쉐가 달립니다.(시속:50km/h)
벤츠가 달립니다.(시속:10km/h)
벤츠가 달립니다.(시속:20km/h)
벤츠가 달립니다.(시속:30km/h)
벤츠가 달립니다.(시속:40km/h)
벤츠가 달립니다.(시속:50km/h)

[Sample.java] 인스턴스 멤버와 this

```
public class Sample {
    public static void main(String[] args) {
        Car myCar = new Car("포르쉐");
        Car yourCar = new Car("벤츠");
        myCar.run();
        yourCar.run();
    }
}
```

6.10 정적 멤버와 static

정적(static)은 '고정된'이란 의미를 가지고 있다. 정적 멤버는 클래스에 고정된 멤버로서 객체를 생성하지 않고 사용할 수 있는 필드와 메서드를 말한다. 정적 멤버는 객체(인스턴스)에 소속된 멤버가 아니라 클래스에 소속된 멤버이기 때문에 클래스 멤버라고도 한다.

6.10.1 정적 멤버 선언

정적 필드와 정적 메서드를 선언하는 방법은 필드와 메서드 선언 시 static 키워드를 추가적으로 붙이면 된다. 예를 들어 Calculator 클래스에서 원의 넓이나 둘레를 구할 때 필요한 파이()는 Calculator 객체마다 가지고 있을 필요가 없는 변하지 않는 공용적인 데이터이므로 정적 필드로 선언하는 것이 좋다. 그러나 객체별로 색깔이 다르다면 색깔은 인스턴스 필드로 선언해야 한다.

```
public class Calculator {
    String color;    //계산기별로 색깔이 다를 수 있다.
    static double pi = 3.14159;    //계산기에서 사용하는 파이( $\pi$ ) 값은 동일하다.
}
```

Calculator 클래스의 덧셈, 뺄셈 기능은 인스턴스 필드를 이용하기보다는 외부에서 주어진 매개값들을 가지고 덧셈과 뺄셈을 수행하므로 정적 메서드로 선언하는 것이 좋다. 그러나 인스턴스 필드인 색깔을 변경하는 메서드는 인스턴스 메서드로 선언해야 한다.

```
public class Calculator {
    String color;    //인스턴스 필드
    void setColor(String color) {this.color = color;}    //인스턴스 메서드
    static int plus(int x, int y) {return x + y;}    //정적 메서드
    static int minus(int x, int y) {return x - y;}    //정적 메서드
}
```

6.10.2 정적 멤버 사용

클래스가 메모리로 로딩 되면 정적 멤버를 바로 사용할 수 있는데, 클래스 이름과 함께 도트(.) 연산자로 접근한다. 예를 들어 위의 Calculator 클래스에서 정적 필드 pi와 정적 메서드 plus(), minus()는 아래와 같이 사용할 수 있다. 정적 필드와 정적 메서드는 객체 참조 변수로도 접근 가능하지만 정적 요소는 클래스 이름으로 접근하는 것이 좋고 객체 참조 변수로 접근했을 경우, 경고 표시가 나타난다.

```
double result1 = 10 * 10 * Calculator.pi;
int result2 = Calculator.plus(10, 5);
int result3 = Calculator.minus(10, 5);
```

[Calculator.java] 정적 멤버 사용

```
public class Calculator {
    static double pi = 3.14159;

    static int plus(int x, int y) {
        return x + y;
    }

    static int minus(int x, int y) {
        return x - y;
    }
}
```

[Sample.java] 정적 멤버 사용

```
public class Sample {
    public static void main(String[] args) {
        double result1 = 10 * 10 * Calculator.pi;

        int result2 = Calculator.plus(10, 5);
        int result3 = Calculator.minus(10, 5);

        System.out.println("result1: " + result1);
        System.out.println("result2: " + result2);
        System.out.println("result3: " + result3);
    }
}
```

[실행결과]

```
result1: 314.159
result2: 15
result3: 5
```

6.10.3 정적 초기화 블록

자바는 정적 필드의 복잡한 초기화 작업을 위해서 정적 블록(static block)을 제공한다. 정적 블록은 클래스가 메모리로 로딩될 때 자동적으로 실행되며, 정적 블록은 클래스 내부에 여러 개가 선언되어도 상관없다. 아래 예제는 company와 model은 선언 시 초기값을 주었고 info는 초기화하지 않았다. info 필드는 정적 블록에서 company와 model 필드값을 서로 연결해서 초기값으로 설정한다.

[Television.java] 정적 초기화 블록

```
public class Television {
    static String company = "Samsung";
    static String model = "LCD";
    static String info;
    static{
        info = company + "-" + model;
    }
}
```

[Sample.java] 정적 초기화 블록

```
public class Sample {
    public static void main(String[] args) {
        System.out.println(Television.info);
    }
}
```

[실행결과]

Samsung-LCD

6.10.4 정적 메서드와 블록 선언 시 주의할 점

정적 메서드와 정적 블록을 선언할 때는 객체가 없어도 실행된다는 특징 때문에, 이들 내부에 인스턴스 필드나 인스턴스 메서드를 사용할 수 없다. 또한 객체 자신의 참조인 this 키워드 사용이 불가능하다. 인스턴스 멤버를 사용하고 싶다면 객체를 먼저 생성하고 참조 변수로 접근해야 한다.

[Sample.java] 정적 초기화 블록

```
public class Car {
    int speed;
    void run(){
        System.out.println(speed + "으로 달립니다.");
    }
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.speed = 60;    //speed = 60 컴파일 에러
        myCar.run();        //run() 컴파일 에러
    }
}
```

[실행결과]

60으로 달립니다.

6.10.5 싱글톤(Singleton)

전체 프로그램에서 단 하나의 객체만 만들도록 보장해야 하는 경우가 있다. 단 하나만 생성된다고 해서 이 객체를 싱글톤(Singleton)이라고 한다. 싱글톤을 만들려면 클래스 외부에서 new 연산자로 생성자를 호출할 수 없도록 막아야 한다. 생성자를 호출한 만큼 객체가 생성되기 때문이다.

[Singleton.java] 싱글톤

```
public class Singleton {
    private static Singleton singleton = new Singleton();
    private Singleton(){ }
    static Singleton getInstance() {
        return singleton;
    }
}
```

[Sample.java] 싱글톤 객체

```
public class Sample {
    public static void main(String[] args) {
        //컴파일 에러 Singleton obj1 = new Singleton();
        Singleton obj1 = Singleton.getInstance();
        //컴파일 에러 Singleton obj2 = new Singleton();
        Singleton obj2 = Singleton.getInstance();
        if(obj1 == obj2){
            System.out.println("같은 Singleton 객체 입니다.");
        }else{
            System.out.println("다른 Singleton 객체 입니다.");
        }
    }
}
```

[실행결과]

같은 Singleton 객체 입니다.

6.11 final 필드와 상수

6.11.1 final 필드

final 필드는 초기값이 저장되면 이것이 최종적인 값이 되어서 프로그램 실행 도중에 수정할 수 없다는 것이다. final 필드의 초기값을 줄 수 있는 방법은 필드 선언 시에 주는 방법과, 생성자에서 주는 방법이 있다. 단순 값이라면 필드 선언 시에 주는 것이 제일 간단하다. 하지만 복잡한 초기화 코드가 필요하거나 객체 생성 시에 외부 데이터로 초기화해야 한다면 생성자에서 초기값을 지정해야한다.

아래 예제의 주민등록번호 필드는 한 번 값이 저장되면 변경할 수 없도록 final 필드로 선언했다. 하지만 주민등록번호는 Person 객체가 생성될 때 부여되므로 Person 클래스 설계 시 초기값을 미리 줄 수 없다. 그래서 생성자 매개값으로 주민등록번호를 받아서 초기값으로 지정해주었다. 반면 nation은 항상 고정된 값을 갖기 때문에 필드 선언 시 초기값으로 'Korea'를 주었다.

[Person.java] final 필드 선언과 초기화

```
public class Person {
    final String nation = "Korea"
    final String ssn
    String name

    public Person(String ssn, String name) {
        this.ssn = ssn
        this.name = name
    }
}
```

[Sample.java] 싱글톤 객체

```
public class Sample {
    public static void main(String[] args) {
        Person p1 = new Person("123456-1234567", "계백");

        System.out.println(p1.nation);
        System.out.println(p1.ssn);
        System.out.println(p1.name);

        //p1.nation = "usa";
        //p1.ssn = "654321-7654321";
        p1.name = "율지문턱";
    }
}
```

[실행결과]

Korea
123456-1234567
계백

6.11.2 상수(static final)

일반적으로 불변의 값을 상수라고 부른다. 불변의 값은 수학에서 사용하는 원주율 파이()나, 지구의 무게 및 둘레 등이 해당된다. 상수는 객체마다 저장할 필요가 없는 공용성을 띠고 있으며, 여러 가지 값으로 초기화될 수 없기 때문에 final 필드를 상수라고 부르진 않는다. 상수는 static이면서 final이어야 한다. static final 필드는 객체마다 저장되지 않고, 클래스에만 포함된다. 그리고 한 번 초기값이 저장되면 변경할 수 없다.

상수 이름은 모두 대문자로 작성하는 것이 관례이다. 만약 서로 다른 단어가 혼합된 이름이라면 언더바(_)로 단어들을 연결해 준다.

[Earth.java] 상수 선언

```
public class Earth {
    static final double EARTH_RADIUS = 6400;
    static final double EARTH_SURFACE_AREA;

    //Math.PI는 자바에서 제공하는 상수
    static {
        EARTH_SURFACE_AREA = 4 * Math.PI * EARTH_RADIUS * EARTH_RADIUS;
    }
}
```

[Sample.java] 상수 사용

```
public class Sample {
    public static void main(String[] args) {
        System.out.println("지구의 반지름: " + Earth.EARTH_RADIUS + " km");
        System.out.println("지구의 표면적: " + Earth.EARTH_SURFACE_AREA + " km^2");
    }
}
```

6.12 패키지

자바에서는 클래스를 체계적으로 관리하기 위해 패키지(package)를 사용한다. 우리가 폴더를 만들어 파일을 저장 관리하듯이 패키지를 만들어 클래스를 저장 관리한다. 클래스 이름이 동일하더라도 패키지가 다르면 다른 클래스로 인식한다. 클래스의 전체 이름은 "패키지명 + . + 클래스명"이다.

6.12.1 패키지 선언

패키지는 컴파일러가 클래스에 포함되어 있는 패키지 선언을 보고, 파일 시스템의 폴더로 자동 생성시킨다. 아래는 패키지를 선언하는 방법이다.

```
package 상위패키지.하위패키지;
```

대규모 프로젝트를 개발할 경우 패키지 이름이 중복될 가능성이 있으므로 회사들 간에 패키지가 서로 중복되지 않도록 흔히 회사의 도메인 이름으로 패키지를 만든다. 도메인 이름으로 패키지를 만들 경우, 도메인 이름 역순으로 패키지 이름을 지어주는데, 그 이유는 포괄적인 이름이 상위 패키지에 되도록 하기 위해서다. 그리고 마지막에는 프로젝트 이름을 붙여주는 것이 관례이다.

```
com.samsung.projectName;  
com.hyundai.projectName;  
com.lg.projectName;
```

6.12.2 이클립스에서 패키지 생성과 클래스 생성

이클립스에서는 패키지만 따로 생성할 수 있고, 클래스를 생성할 때 동시에 생성시킬 수도 있다. 이클립스에서 패키지를 생성하려면 프로젝트의 src 폴더를 선택하고 메뉴에서 [File]-[New]-[Package]를 선택하면 된다. 패키지 생성 후 클래스를 생성하려면 해당 패키지를 선택하고 메뉴에서 [File]-[New]-[Class]를 선택하면 해당 패키지에 소속된 클래스를 생성할 수 있다.

6.12.3 import문

아래 예제는 세 패키지를 모두 사용하는 Car.java 소스이다. Tire 클래스는 import된 두 패키지에 모두 있기 때문에 패키지 이름과 함께 이름이 기술되어야 한다. Engine 클래스는 hyundai 패키지에만 존재하기 때문에 아무런 문제가 없고 SnowTire와 BigWidthTire 클래스도 각각 hankook, kumho 패키지에만 존재하기 때문에 아무런 문제가 없다.

패키지: hankook

[SnowTire.java]

```
package hankook;  
public class SnowTire { }
```

[Tire.java]

```
package hankook;  
public class Tire { }
```

패키지: kumho

[BigWideTire.java]

```
package kumho;  
public class BigWidthTire { }
```

[Tire.java]

```
package kumho;  
public class Tire { }
```

패키지: hyundai

[Engine.java]

```
package hyundai;  
public class Engine { }
```

[Car.java] import문

```
package mycompany;
```

```
import kumho.BigWideTire;  
import hankook.SnowTire;  
import hyundai.Engine
```

```
public class Car {  
    Engine engine = new Engine();  
    SnowTire tire1 = new SnowTire();  
    BigWideTire tire2 = new BigWideTire();  
    hankook.Tire tire3 = new hankook.Tire();  
    kumho.Tire tire4 = new kumho.Tire();  
}
```


이클립스에는 개발자가 import문을 작성하지 않아도 사용된 클래스를 조사해서 필요한 import 문을 자동적으로 추가하는 기능이 있다. 현재 작성 중인 클래스에서 아래와 같이 선택하면 된다.

[Source]-[Organize imports] (단축키: Ctrl + Shift + O)

6.13 접근 제한자

main() 메서드를 가지지 않는 대부분의 클래스는 외부 클래스에서 이용할 목적으로 설계된 라이브러리 클래스이다. 라이브러리 클래스를 설계할 때에는 외부 클래스에서 접근할 수 있는 멤버와 접근할 수 없는 멤버로 구분해서 필드, 생성자, 메서드를 설계하는 것이 바람직하다.

- public: 외부 클래스가 자유롭게 사용할 수 있는 공개 멤버를 만든다.
- protected: 같은 패키지 또는 자식 클래스에서 사용할 수 있는 멤버를 만든다.
- private: 단어의 뜻 그대로 개인적인 것이라 외부에 노출되지 않는 멤버를 만든다.
- default: 같은 패키지에 소속된 클래스에서만 사용할 수 있는 멤버를 만든다.

6.13.1 클래스의 접근 제한

클래스를 선언할 때 고려해야 할 사항은 같은 패키지 내에서만 사용할 것인지, 아니면 다른 패키지에서도 사용할 수 있도록 할 것인지를 결정해야 한다. 클래스에 적용할 수 있는 접근 제한은 public과 default 단 두 가지인데, 아래와 같은 형식으로 작성한다.

```
//default 접근 제한
class 클래스 { ... }

//public 접근 제한
public class 클래스 { ... }
```

- 클래스를 선언할 때 public을 생략했다면 클래스는 default 접근 제한을 가진다. 클래스가 default 접근 제한을 가지게 되면 같은 패키지에서는 아무런 제한 없이 사용할 수 있지만 다른 패키지에서는 사용할 수 없도록 제한된다.
- 클래스가 public 접근 제한을 가지게 되면 같은 패키지뿐만 아니라 다른 패키지에서도 아무런 제한 없이 사용할 수 있다. 인터넷으로 배포되는 라이브러리 클래스들도 모두 public 접근 제한을 가지고 있다.

아래 세 클래스를 이클립스에서 작성해 보면서 이클립스가 어떤 컴파일 에러를 발생시키는지 살펴보자.

[A.java] 클래스의 접근 제한

```
package package1;

class A { } //default 접근 제한
```

B 클래스는 A 클래스와 같은 패키지이므로 A클래스에 접근이 가능하다. 그래서 B 클래스에서 A 클래스를 이용하여 필드 선언 및 생성자와 메서드 내부에서 변수 선언이 가능하다.

[B.java] 클래스의 접근 제한

```
package package1;

public class B {
    A a; //(O) A클래스 접근 가능(필드로 선언할 수 있음)
}
```

C 클래스는 A클래스와 다른 패키지이므로 default 접근이 제한된 A 클래스에는 접근이 되지 않지만, public으로 공개된 B 클래스는 접근이 가능하다. 그래서 C 클래스에서 B 클래스를 이용하여 필드 선언 및 생성자와 메서드 내부에서 변수 선언이 가능하다.

[C.java] 클래스의 접근 제한

```
//패키지가 다름
package package2;

import package1.*;
public class C {
    A a; //(X) A클래스 접근 불가(컴파일 에러)
    B b; //(O)
}
```

6.13.2 생성자의 접근 제한

객체를 생성하기 위해서는 new 연산자로 생성자를 호출해야 한다. 하지만 생성자를 어디에서나 호출할 수 있는 것은 아니다. 생성자가 어떤 접근 제한을 갖느냐에 따라 호출 가능 여부가 결정된다. 생성자는 아래와 같이 public, protected, default, private 접근 제한을 가질 수 있다.

```
public class ClassName {
    //public 접근 제한
    public ClassName( ... ) { ... }

    //protected 접근 제한
    protected ClassName( ... ) { ... }

    //default 접근 제한
    ClassName( ... ) { ... }

    //private 접근 제한
    private ClassName( ... ) { ... }
}
```

- public: 모든 패키지에서 아무런 제한 없이 생성자를 호출할 수 있도록 한다.
- protected: default 접근 제한과 마찬가지로 같은 패키지에 속하는 클래스에서 생성자를 호출할 수 있도록 한다. 차이점은 다른 패키지에 속한 클래스가 해당 클래스의 자식(child) 클래스라면 생성자를 호출할 수 있다.
- default: 같은 패키지에서는 아무런 제한 없이 생성자를 호출할 수 있으나, 다른 패키지에서는 생성자를 호출할 수 없도록 한다.
- private: 동일 패키지이건 다른 패키지이건 상관없이 생성자를 호출하지 못하도록 제한한다.

아래 A, B, C 클래스를 작성해 어떤 컴파일 에러를 발생시키는지 살펴보자. A 클래스에서는 A의 모든 생성자를 호출할 수 있음을 알 수 있다.

[A.java] 생성자의 접근 제한

```
package package1;

public class A {
    //필드 정의
    A a1 = new A(true);    //(O)
    A a2 = new A(1);        //(O)
    A a3 = new A("문자열");  //(O)
    //생성자 정의
    public A(boolean b) {}    //public 접근 제한
    A(int b) {}                //default 접근 제한
    private A(String S) {}    //private 접근 제한
}
```

패키지가 동일한 B 클래스에서는 A 클래스의 private 생성자를 제외하고 다른 생성자를 호출할 수 있다.

[B.java] 클래스의 접근 제한

```
package package1;    //패키지 동일

public class B {
    A a1 = new A(true);    //(O)
    A a2 = new A(1);        //(O)
    A a3 = new A("문자열");  //(X) private 생성자 접근 불가(컴파일 에러)
}
```

아래와 같이 패키지가 다른 C클래스에서는 A클래스의 public 생성자를 제외하고 다른 생성자를 호출할 수 없다.

[C.java] 클래스의 접근 제한

```
//패키지가 다름
package package2;

import package1.A;
public class C {
    A a1 = new A(true);    //(O)
    A a2 = new A(1);        //(X) default 생성자 접근 불가(컴파일 에러)
    A a3 = new A("문자열");  //(X) private 생성자 접근 불가(컴파일 에러)
}
```

6.13.3 필드와 메서드의 접근 제한

필드와 메서드를 선언할 때 고려해야 할 사항은 클래스 내부에서만 사용할 것인지, 패키지 내에서만 사용할 것인지, 아니면 다른 패키지에서도 사용할 수 있도록 할 것인지를 결정해야 한다. 필드와 메서드는 아래와 같이 public, protected, default, private 접근 제한을 가진다.

- public: 모든 패키지에서 아무런 제한 없이 필드와 메서드를 사용할 수 있도록 해준다.
- protected: default 접근 제한과 마찬가지로 같은 패키지에 속하는 클래스에서 필드와 메서드를 사용할 수 있도록 한다.
- default: 필드와 메서드를 선언할 때 public 또는 default 접근 제한을 가지며 다른 패키지에서는 필드와 메서드를 사용할 수 없도록 한다.
- private: 동일 패키지이건 다른 패키지이건 상관없이 필드와 메서드를 사용하지 못하도록 제한한다.

아래 A, B, C 클래스를 작성해 어떤 컴파일 에러를 발생시키는지 살펴보자. A클래스 내부에서는 모든 필드와 메서드를 모두 사용할 수 있다.

[A.java] 필드와 메서드의 접근 제한

```
package package1;

public class A {
    public int field1;    //public 접근 제한
    int field2;          //default 접근 제한
    private int field3;  //private 접근 제한

    //생성자 정의
    public A() {
        field1 = 1;
        field2 = 1;
        field3 = 1;
        method1();
        method2();
        method3();
    }

    public void method1() {}    //public 접근 제한
    void method2() {}          //default 접근 제한
    private void method3() {}  //private 접근 제한
}
```

패키지가 동일한 B클래스에서는 A클래스의 private 필드와 메서드를 제외한 다른 필드와 메서드는 사용할 수 있다.

[B.java] 필드와 메서드의 접근 제한

```
package package1;    //패키지가 동일

public class B {
    public B() {
        A a = new A();
        a.field1 = 1;    //(O)
        a.field2 = 1;    //(O)
        a.field3 = 1;    //(X) private 필드 접근 불가(컴파일 에러)
        a.method1();    //(O)
        a.method2();    //(O)
        a.method3();    //(X) private 메서드 접근 불가(컴파일 에러)
    }
}
```

아래와 같이 패키지가 다른 C클래스에서는 A클래스의 public 필드와 메서드를 제외한 다른 필드와 메서드를 사용할 수 없다.

[A.java] 필드와 메서드의 접근 제한

```
package package2;    //패키지가 다름

import package1.A;
public class C {
    public C() {
        A a = new A();
        a.field1 = 1;    //(O)
        a.field2 = 1;    //(X) default 필드 접근 불가(컴파일 에러)
        a.field3 = 1;    //(X) private 필드 접근 불가(컴파일 에러)
        a.method1();    //(O)
        a.method2();    //(X) default 메서드 접근 불가(컴파일 에러)
        a.method3();    //(X) private 메서드 접근 불가(컴파일 에러)
    }
}
```

6.14 Getter와 Setter 메서드

일반적으로 객체 지향 프로그래밍에서 객체 데이터는 객체 외부에서 직접적으로 접근하는 것을 막는다. 외부에서 접근하기 위해서는 메서드를 통해서 데이터를 변경하는 방법을 선호한다. 데이터는 외부에서 접근할 수 없도록 막고 메서드는 공개해서 외부에서 메서드를 통해 데이터에 접근하도록 유도한다. 그 이유는 메서드는 매개값을 검증해서 유효한 값만 데이터로 저장할 수 있기 때문이다.

- Setter: 외부에서 데이터를 검증해서 유효한 값으로 변경 후 저장하는 메서드이다.
- Getter: 외부로 데이터를 가공해서 전달하는 메서드이다.

이클립스는 클래스에 선언된 필드에 대해 자동적으로 Getter와 Setter 메서드를 생성시키는 기능이 있다. 먼저 필드를 선언한 다음에 메뉴에서 [Source]-[Generate Getter and Setter]를 선택하면 선언된 필드에 대한 Getter와 Setter를 자동 생성시킬 수 있다.

[Car.java] Getter와 Setter 메서드 선언

```
public class Car {
    //필드 정의
    private int speed;
    private boolean stop;

    //생성자 정의

    //메서드
    public int getSpeed() {
        return speed;
    }

    public void setSpeed(int speed) {
        if(speed < 0) {
            this.speed = 0;
            return;
        }else{
            this.speed = speed;
        }
    }

    public boolean isStop() {
        return stop;
    }

    public void setStop(boolean stop) {
        this.stop = stop;
        this.speed = 0;
    }
}
```

비정상적인 속도값으로 시도하지만 speed 필드의 Setter에서 매개값을 검사한 후에 0으로 변경하기 때문에 Getter의 리턴값이 0으로 나온다. stop 필드의 Getter 리턴값이 false일 경우, 자동차를 멈추기 위해 Setter를 호출해서 stop 필드를 true로, speed 필드를 0으로 변경한다.

[Sample.java] Getter와 Setter 메서드 사용

```
public class Sample {
    public static void main(String[] args) {
        Car myCar = new Car();

        //잘못된 속도 변경
        myCar.setSpeed(-50);
        System.out.println("현재 속도: " + myCar.getSpeed());

        //올바른 속도 변경
        myCar.setSpeed(60);

        //멈춤
        if(!myCar.isStop()){
            myCar.setStop(true);
        }
        System.out.println("현재 속도: " + myCar.getSpeed());
    }
}
```

[실행결과]

현재 속도: 0
현재 속도: 0

7장 상속

7.1 상속 개념

객체지향 프로그램에서는 부모 클래스의 멤버를 자식 클래스에게 물려줄 수 있다. 프로그램에서는 부모 클래스를 상위 클래스라고 부르기도 하고, 자식 클래스를 하위 클래스, 또는 파생 클래스라고 부른다. 그러나 상속을 해도 부모 클래스에서 private 접근 제한을 갖는 필드와 메서드는 상속 대상에서 제외된다. 부모 클래스와 자식 클래스가 다른 패키지에 존재한다면 default 접근 제한을 갖는 필드와 메서드는 상속 대상에서 제외된다. 상속을 이용하면 부모 클래스의 수정으로 모든 자식 클래스들의 수정 효과를 가져 오기 때문에 유지 보수 시간을 최소화시켜준다.

7.2 클래스 상속

현실에서 상속은 부모가 자식을 선택해서 물려주지만, 프로그램에서는 자식이 부모를 선택한다. 자식 클래스를 선언할 때 어떤 부모 클래스를 상속받을 것인지를 결정하고 선택된 부모 클래스는 아래와 같이 extends 뒤에 기술한다.

```
class 자식 클래스 extends 부모 클래스 {
    //필드
    //생성자
    //메서드
}
```

다는 언어와는 달리 자바는 다중 상속을 허용하지 않는다. 즉 여러 개의 부모 클래스를 상속할 수 없다. 그러므로 extends 뒤에는 단 하나의 부모 클래스만 와야 한다. 아래 예제는 핸드폰(CellPhone) 클래스를 상속해서 DMB폰(DmbCellPhone) 클래스를 작성한 것이다. 핸드폰이 부모 클래스가 되고, DMB폰이 자식 클래스가 된다.

[CellPhone.java] 부모 클래스

```
public class CellPhone {
    //필드 정의
    String model;
    String color;

    //생성자 정의

    //메서드 정의
    void powerOn() { System.out.println("전원을 켭니다."); }
    void powerOff() { System.out.println("전원을 끕니다."); }
    void bell() { System.out.println("벨이 울립니다."); }
    void sendVoice(String message) { System.out.println("자기:" + message); }
    void receiveVoice(String message) { System.out.println("상대방:" + message); }
    void hangup() { System.out.println("전화를 끊습니다."); }
}
```

[DmbCellPhone.java] 자식 클래스

```
public class DmbCellPhone extends CellPhone {
    //필드 정의
    int channel;

    //생성자 정의
    DmbCellPhone(String model, String color, int channel) {
        this.model = model;
        this.color = color;
        this.channel = channel;
    }

    //메서드 정의
    void turnOnDmb() {
        System.out.println("채널:" + channel + "번 DMB 방송 수신을 시작합니다.");
    }
    void changeCannelDmb(int channel) {
        this.channel = channel;
        System.out.println("채널:" + channel + "번으로 바꿉니다.");
    }
    void turnOffDmb() {
        System.out.println("DMB 방송 수신을 멈춥니다.");
    }
}
```

[Sample.java] 자식 클래스 사용

```
public class Sample {
    public static void main(String[] args) {
        //DmbCellPhone 객체 생성
        DmbCellPhone dmbCellPhone = new DmbCellPhone("자바폰", "검정", 10);

        //CellPhone으로부터 상속받은 필드
        System.out.println("모델:" + dmbCellPhone.model);
        System.out.println("색상:" + dmbCellPhone.color);

        //DmbCellPhone의 필드
        System.out.println("채널:" + dmbCellPhone.channel);

        //CellPhone으로부터 상속받은 메서드 호출
        dmbCellPhone.powerOn();
        dmbCellPhone.bell();
        dmbCellPhone.sendVoice("여보세요");
        dmbCellPhone.receiveVoice("안녕하세요! 저는 홍길동인데요");
        dmbCellPhone.sendVoice("아~ 예 반갑습니다.");
        dmbCellPhone.hangup();

        //DmbCellPhone의 메서드 호출
        dmbCellPhone.turnOnDmb();
        dmbCellPhone.changeCannelDmb(12);
        dmbCellPhone.turnOffDmb();
    }
}
```

[실행결과]

```
모델:자바폰
색상:검정
채널:10
전원을 켭니다.
벨이 울립니다.
자기:여보세요
상대방:안녕하세요! 저는 홍길동인데요
자기:아~ 예 반갑습니다.
전화를 끊습니다.
채널:10번 DMB 방송 수신을 시작합니다.
채널:12번으로 바꿉니다.
DMB 방송 수신을 멈춥니다.
```

7.3 부모 생성자 호출

현실에서 부모 없는 자식이 있을 수 없듯이 자바에서도 자식 객체를 생성하면, 부모 객체가 먼저 생성되고 자식 객체가 그 다음에 생성된다. 부모 객체를 생성하기 위해 부모 생성자는 자식 생성자의 맨 첫 줄에서 호출된다. 부모 클래스에 기본 생성자가 없고 매개 변수가 있는 생성자만 있다면 자식 생성자에서 반드시 부모 생성자 호출을 위해 자식 생성자 첫 줄에 `super(매개값, ...)`를 명시적으로 호출해야 한다.

[People.java] 부모 클래스

```
public class People {
    public String name;
    public String ssn;

    public People(String name, String ssn) {
        this.name = name;
        this.ssn = ssn;
    }
}
```

[Student.java] 자식 클래스

```
public class Student extends People {
    public int studentNo;

    public Student(String name, String ssn, int studentNo) {
        super(name, ssn); //부모 생성자 호출
        this.studentNo = studentNo;
    }
}
```

[Sample.java] 자식 객체 이용

```
public class Sample {
    public static void main(String[] args) {
        Student student = new Student("홍길동", "123456-1234567", 1);

        //부모에서 물려받은 필드 출력
        System.out.println("name:" + student.name);
        System.out.println("ssn:" + student.ssn);
        System.out.println("studentNo:" + student.studentNo);
    }
}
```

[실행결과]

```
name:홍길동
ssn:123456-1234567
studentNo:1
```

7.4 메서드 재정의

부모 클래스의 모든 메서드가 자식 클래스에 맞게 설계되어 있다면 가장 이상적인 상속이지만, 어떤 메서드는 자식 클래스가 사용하기에 적합하지 않을 수도 있다. 이 경우 상속된 일부 메서드는 자식 클래스에서 다시 수정위해 메서드 오버라이딩(Overriding) 기능을 사용한다.

7.4.1 메서드 재정의(@Override)

메서드 오버라이딩은 상속된 메서드의 내용이 자식 클래스에 맞지 않을 경우, 자식 클래스에서 동일한 메서드를 재정의하는 것을 말한다. 메서드가 오버라이딩되었다면 부모 객체의 메서드는 숨겨지기 때문에, 자식 객체에서 메서드를 호출하면 오버라이딩된 자식 메서드가 호출된다.

- 메서드를 오버라이딩할 때는 다음과 같은 규칙에 주의해서 작성해야한다.
- 부모의 메서드와 동일한 리턴타입, 메서드 이름, 매개 변수 리스트를 가져야한다.
- 새로운 예외(Exception)를 throws할 수 없다.

접근 제한을 더 강하게 오버라이딩할 수 없다는 것은 부모 메서드가 public 접근 제한을 가지고 있을 경우 오버라이딩하는 자식 메서드는 default 나 private 접근 제한으로 수정할 수 없다는 뜻이다. 반대는 가능하다. 부모 메서드가 default 접근 제한을 가지면 재정의되는 자식 메서드는 default 또는 public 접근 제한을 가질 수 있다. 아래 예제는 Calculator의 자식 클래스인 Computer에서 원의 넓이를 구하는 Calculator의 areaCircle()메서드를 사용하지 않고 좀 더 정확한 원의 넓이를 구하도록 오버라이딩했다.

[Calculator.java] 부모 클래스

```
public class Calculator {
    double areaCircle(double r) {
        System.out.println("Calculator 객체의 areaCircle() 실행");
        return 3.14159 * r * r;
    }
}
```

Calculator의 areaCircle() 메서드의 파이의 값을 3.14159로 계산하였지만, 좀 더 정밀한 계산을 위해 Computer의 areaCircle() 메서드는 Math.PI 상수를 이용한다. Math는 수학 계산과 관련된 필드와 메서드들을 가지고 있는 클래스로, 자바 표준 API를 제공한다. @Override 어노테이션은 생략해도 좋으나, 이것을 붙여주게 되면 areaCircle() 메서드가 정확히 오버라이딩된 것인지 컴파일러가 체크하기 때문에 개발자의 실수를 줄여준다. 예를 들어 개발자가 areaCircle()처럼 끝에 e를 빼먹게 되면 컴파일 에러가 발생한다.

[Computer.java] 자식 클래스

```
public class Computer extends Calculator {
    @Override
    double areaCircle(double r) {
        System.out.println("Computer 객체의 areaCircle() 실행");
        return Math.PI * r * r;
    }
}
```

[Sample.java] 메서드 오버라이딩 테스트

```
public class Sample {
    public static void main(String[] args) {
        int r = 10;

        Calculator calculator = new Calculator();
        System.out.println("원면적:" + calculator.areaCircle(r));
        System.out.println();

        Computer computer = new Computer();

        //재정의된 메서드 호출
        System.out.println("원면적:" + computer.areaCircle(r));
    }
}
```

[실행결과]

Calculator 객체의 areaCircle() 실행
원면적:314.159

Computer 객체의 areaCircle() 실행
원면적:314.1592653589793

이클립스는 부모 메서드 중 하나를 선택해서 오버라이딩 메서드를 자동 생성해주는 기능이 있다. 이 기능은 부모 메서드의 시그니처를 정확히 모를 경우 매우 유용하게 사용할 수 있다.

1. 자식 클래스에서 오버라이딩 메서드를 작성할 위치로 입력 커서를 옮긴다.
2. 메뉴에서 [Source Override/Implement Methods ...]를 선택한다.
3. 부모 클래스에서 오버라이딩될 메서드를 선택하고 [OK]버튼을 클릭한다.

7.4.2 부모 메서드 호출(super)

자식 클래스에서 부모 클래스의 메서드를 오버라이딩하게 되면, 부모 클래스의 메서드는 숨겨지고 오버라이딩된 자식 메서드만 사용된다. 그러나 자식 클래스 내부에서 오버라이딩된 부모 클래스의 메서드를 호출해야 하는 상황이 발생한다면 명시적으로 super 키워드를 붙여서 부모 메서드를 호출할 수 있다. super는 부모 객체를 참조하고 있기 때문에 부모 메서드에 직접 접근할 수 있다.

[Airplane.java] super 변수

```
public class Airplane {
    public void land() {
        System.out.println("착륙합니다.");
    }
    public void fly() {
        System.out.println("일반비행합니다.");
    }
    public void takeOff() {
        System.out.println("이륙합니다.");
    }
}
```

아래 예제는 Airplane 클래스를 상속해서 SupersonicAirplane 클래스를 만들었다. 부모클래스 Airplane의 fly() 메서드는 일반 비행기이지만 SupersonicAirplane의 fly()는 초음속 비행 모드와 일반 비행 모드 두 가지로 동작하도록 설계했다.

[SupersonicAirplane.java] super 변수

```
public class SupersonicAirplane extends Airplane {
    public static final int NORMAL = 1;
    public static final int SUPERSONIC = 2;
    public int flyMode = NORMAL;

    @Override
    public void fly() {
        if(flyMode == SUPERSONIC) {
            System.out.println("초음속 비행합니다.");
        }else {
            super.fly(); //Airplane 객체의 fly() 메서드 호출
        }
    }
}
```

[Sample.java] super 변수

```
public class Sample {
    public static void main(String[] args) {
        SupersonicAirplane sa = new SupersonicAirplane();
        sa.takeOff();
        sa.fly();

        sa.flyMode = SupersonicAirplane.SUPERSONIC;
        sa.fly();

        sa.flyMode = SupersonicAirplane.NORMAL;
        sa.fly();
        sa.land();
    }
}
```

[실행결과]

```
이륙합니다.
일반비행합니다.
초음속비행합니다.
일반비행합니다.
착륙합니다.
```

7.5 final 클래스와 final 메서드

final 키워드는 클래스, 필드, 메서드 선언 시에 사용할 수 있다. final 키워드는 해당 선언이 최종 상태이고 결코 수정될 수 없음을 뜻한다.

7.5.1 상속할 수 없는 final 클래스

클래스를 선언할 때 final 키워드를 class 앞에 붙이게 되면 이 클래스는 최종적인 클래스이므로 상속할 수 없는 클래스가 된다. 즉 final 클래스는 부모 클래스가 될 수 없어 자식 클래스를 만들 수 없다는 것이다. final 클래스의 대표 예는 자바 표준 API에서 제공하는 String 클래스이다.

7.5.2 오버라이딩할 수 없는 final 메서드

메서드를 선언할 때 final 키워드를 붙이게 되면 이 메서드는 최종적인 메서드이므로 오버라이딩 할 수 없는 메서드가 된다. 즉 부모 클래스를 상속해서 자식 클래스를 선언할 때 부모 클래스에 선언된 final 메서드는 자식 클래스에서 재정의 할 수 없다는 것이다.

아래는 Car 클래스의 stop() 메서드를 final로 선언해서 Car를 상속한 SportsCar 클래스에서 stop() 메서드를 오버라이딩 할 수 없다.

[Car.java] 재정의할 수 없는 final 메서드

```
public class Car {
    //필드정의
    private int speed;
    //메서드정의
    public void speedUp() { speed += 1; }
    //final 메서드정의
    public final void stop() {
        System.out.println("차를 멈춤");
        speed = 0;
    }
}
```

[SportsCar.java] 재정의할 수 없는 final 메서드

```
public class SportsCar extends Car{
    @Override
    public void speedUp() { speed += 10; }

    @Override
    public void stop() {
        System.out.println("스포츠카를 멈춤"); //오버라이딩을 할 수 없음
        speed = 0;
    }
}
```

7.6 protected 접근 제한자

protected는 같은 패키지에서는 default와 같이 접근 제한이 없지만 다른 패키지에서는 자식 클래스만 접근을 허용한다.

[A.java] protected 접근 제한자

```
package package1;

public class A {
    protected String field;
    protected A() {}
    protected void method() {}
}
```

아래 B 클래스는 A 클래스와 동일한 패키지에 있다. B 클래스에서는 A 클래스의 protected 필드, 생성자, 메서드에 얼마든지 접근이 가능하다.

[B.java] protected 접근 제한자 테스트

```
package package1;
public class B {
    public void method() {
        A a = new A();    //(O)
        a.field = "value"; //(O)
        a.method();       //(O)
    }
}
```

아래 C 클래스는 A 클래스와 다른 패키지에 있다. C 클래스에서는 A 클래스의 protected 필드, 생성자, 메서드에 접근할 수 없다.

[C.java] protected 접근 제한자 테스트

```
package package2;
import package1.A;

public class C {
    public void method() {
        A a = new A();    //(X)
        a.field = "value"; //(X)
        a.method();       //(X)
    }
}
```

아래 D 클래스는 A 클래스와 다른 패키지에 있다. C 클래스와는 달리 D는 A의 자식 클래스이다. 그 때문에 A 클래스의 protected 필드, 생성자, 메서드에 접근이 가능하다. 단 new 연산자를 사용해서 생성자를 호출할 수는 없고, 자식 생성자에서 super()로 A 생성자를 호출할 수 있다.

[D.java] protected 접근 제한자 테스트

```
package package2;
import package1.A;

public class D extends A {
    public D(){
        super();           //(O)
        this.field = "value"; //(O)
        this.method();      //(O)
    }
}
```

7.7 타입 변환과 다형성

다형성은 같은 타입이지만 실행 결과가 다양한 객체를 이용할 수 있는 성질을 말한다. 코드 측면에서 보면 다형성은 하나의 타입에 여러 객체를 대입함으로써 다양한 기능을 이용할 수 있도록 해준다. 다형성을 위해 자바는 부모 클래스의 타입 변환을 허용한다. 즉 부모 타입에 모든 자식 객체가 대입될 수 있다. 예를 들어 자동차를 설계할 때 타이어 클래스를 상속한 실제 타이어들은 어떤 것이든 상관없이 장착(대입)이 가능하다.

7.7.1 자동 타입 변환(Promotion)

자동 타입 변환(Promotion)은 프로그램 실행 도중에 자동적으로 타입 변환이 일어나는 것을 말한다. 자동 타입 변환의 개념은 자식은 부모의 특징과 기능을 상속받기 때문에 부모와 동일하게 취급될 수 있다는 것이다. 예를 들어 고양이는 동물의 특징과 기능을 상속받았다. 그래서 "고양이는 동물이다."가 성립된다. Cat 클래스로부터 Cat 객체를 생성하고 이것을 Animal 변수에 대입하면 자동 타입 변환이 일어난다.

```
Cat cat = new Cat();
Animal animal = cat;    //Animal animal = new Cat();도 가능하다.
```

아래 예제의 D객체는 B와 A타입으로 자동 타입 변환이 될 수 있고, E객체는 C와 A타입으로 자동 타입 변환이 될 수 있다. 그러나 D객체는 C타입으로 변환될 수 없고, 마찬가지로 E객체는 B타입으로 변환될 수 없다.

[Sample.java] 자동 타입 변환

```
class A { }

class B extends A { }
class C extends A { }

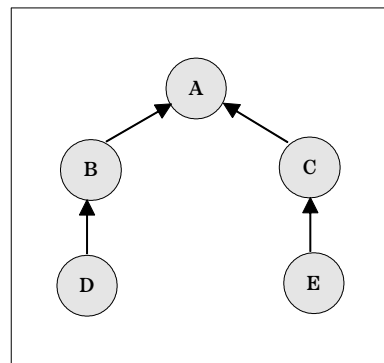
class D extends B { }
class E extends C { }

public class Sample {
    public static void main(String[] args) {
        B b = new B();
        C c = new C();
        D d = new D();
        E e = new E();

        A a1 = b;
        A a2 = c;
        A a3 = d;
        A a4 = e;

        B b1 = d;
        C c1 = e;

        //B b3 = e; 컴파일 에러(상속 관계에 있지 않음)
        //C c2 = d; 컴파일 에러(상속 관계에 있지 않음)
    }
}
```



부모 타입으로 자동 타입 변환된 이후에는 부모 클래스에 선언된 필드와 메서드만 접근이 가능하다. 비록 변수는 자식 객체를 참조하지만 변수로 접근 가능한 멤버는 부모 클래스 멤버로만 한정된다. 그러나 메서드가 자식 클래스에서 오버라이딩 되었다면 자식 클래스의 메서드가 대신 호출된다.

아래 예제의 Child 객체는 method3() 메서드를 가지고 있지만, Parent 타입으로 변환된 이후에는 method3()을 호출할 수 없다. 그러나 method2() 메서드는 부모와 자식 모두에게 있다. 이렇게 오버라이딩 된 메서드는 타입 변환 이후에도 자식 메서드가 호출된다.

[Parent.java] 자동 타입 변환 후의 멤버 접근

```
public class Parent {
    public void method1() {
        System.out.println("Parent-method1()");
    }
    public void method2() {
        System.out.println("Parent-method2()");
    }
}
```

[Child.java] 자동 타입 변환 후의 멤버 접근

```
public class Child extends Parent {
    @Override
    public void method2() {
        System.out.println("Child-method2()"); //재정의
    }

    public void method3() {
        System.out.println("Child-method3()");
    }
}
```

[Sample.java] 자동 타입 변환 후의 멤버 접근

```
public class Sample {
    public static void main(String[] args) {
        Child child = new Child();
        Parent parent = child; //자동 타입 변환

        parent.method1();
        parent.method2(); //재정의된 메서드가 호출됨
        //parent.method3() (호출 불가능)
    }
}
```

[실행결과]

Parent-method1()
Child-method2()

7.7.2 필드의 다형성

다형성이란 동일한 타입을 사용하지만 다양한 결과가 나오는 성질을 말한다. 주로 필드의 값을 다양화함으로써 실행 결과가 다르게 나오도록 구현하는데, 필드의 타입은 변함이 없지만, 실행 도중에 어떤 객체를 필드로 저장하느냐에 따라 실행 결과가 달라질 수 있다. 이것이 필드의 다형성이다. 아래 예제 Tire 클래스의 필드에는 최대 회전수(maxRotation), 누적 회전수(accumulatedRotation), 타이어의 위치가 있다. 최대 회전수는 타이어의 수명으로, 최대 회전수만큼 도달하면 타이어가 펑크 난다고 가정했다. 누적 회전수는 타이어가 1번 회전할 때마다 1씩 증가되는 필드로, 누적 회전수가 최대 회전수가 되면 타이어가 펑크가 난다.

[Tire.java] 타이어 클래스

```
public class Tire {
    public int maxRotation; //최대 회전수(타이어 수명)
    public int accumulatedRotation; //누적 회전수
    public String location; //타이어의 위치

    public Tire(String location, int maxRotation) {
        this.location = location;
        this.maxRotation = maxRotation;
    }

    public boolean roll() {
        ++accumulatedRotation; //누적 회전수 1증가
        if(accumulatedRotation < maxRotation) { //정상 회전(누적<최대)일 경우 실행
            System.out.println(location + " Tire 수명:" + (maxRotation-accumulatedRotation) + "회");
            return true;
        } else { //펑크(누적=최대)일 경우 실행
            System.out.println("*** " + location + "Tire 펑크 ***");
            return false;
        }
    }
}
```

아래 예제 Car 클래스의 필드는 네 개의 타이어가 있다. Tire 객체를 생성할 때 타이어의 위치와 최대 회전수를 생성자의 매개값으로 지정했다. run() 메서드는 네 개의 타이어를 한 번씩 1회전시키는 메서드이다. 각각의 Tire 객체의 roll() 메서드를 호출해서 리턴값이 false가 되면(펑크 나면) stop() 메서드를 호출하고 해당 위치의 타이어 번호를 리턴 한다.

[Car.java] Tire를 부품으로 가지는 클래스

```
public class Car {
    //필드정의
    Tire frontLeftTire = new Tire("앞왼쪽", 6);
    Tire frontRightTire = new Tire("앞오른쪽", 2);
    Tire backLeftTire = new Tire("뒤왼쪽", 3);
    Tire backRightTire = new Tire("뒤오른쪽", 4);
    //메서드정의
    int run() {
        System.out.println("[자동차가 달립니다.]");
        if(frontLeftTire.roll() == false) { stop(); return 1; }
        if(frontRightTire.roll() == false) { stop(); return 2; }
        if(backLeftTire.roll() == false) { stop(); return 3; }
        if(backRightTire.roll() == false) { stop(); return 4; }
        return 0;
    }

    void stop() {
        System.out.println("[자동차가 멈춥니다.]");
    }
}
```

//자동차는 4개의 타이어를 가진다.

//모든 타이어를 1회 회전시키기 위해 각 Tire 객체의 roll()메서드를 호출한다.
//false를 리턴하는 roll()이 있을 경우 stop()을 호출하고 해당 타이어를 번호를 리턴 한다.

//펑크 났을 때 실행

아래 예제 HankookTire와 KumhoTire클래스는 Tire클래스를 상속 받는다. 생성자는 타이어의 위치, 최대 회전수를 매개값으로 받아서 부모인 Tire클래스의 생성자를 호출할 때 넘겨주었다. 정상회전과 펑크 났을 때 출력하는 내용이 Tire클래스의 roll()메서드와 다르다.

[HankookTire.java] Tire의 자식 클래스

```
public class HankookTire extends Tire {
    //생성자
    public HankookTire(String location, int maxRotation) {
        super(location, maxRotation);
    }

    //메서드
    @Override
    public boolean roll() {
        ++accumulatedRotation;
        if(accumulatedRotation < maxRotation) {
            System.out.println(location + " HankookTire 수명:" + (maxRotation-accumulatedRotation) + "회");
            return true;
        } else {
            System.out.println("*** " + location + "HankookTire 펑크 ***");
            return false;
        }
    }
}
```

//출력 내용을 달리하기 위해 재정의 오버라이딩한 roll()메서드

[KumhoTire.java] Tire의 자식 클래스

```
public class KumhoTire extends Tire{
    //생성자
    public KumhoTire(String location, int maxRotation) {
        super(location, maxRotation);
    }

    //메서드
    @Override
    public boolean roll() {
        ++accumulatedRotation;
        if(accumulatedRotation < maxRotation) {
            System.out.println(location + " KumhoTire 수명:" + (maxRotation-accumulatedRotation) + "회");
            return true;
        } else {
            System.out.println("*** " + location + "KumhoTire 펑크 ***");
            return false;
        }
    }
}
```

//출력 내용을 달리하기 위해 재정의 오버라이딩한 roll()메서드

아래 예제는 지금까지 작성한 Tire, Car, HankookTire, KumhoTire 클래스를 이용하는 실행 클래스이다. Car.run() 메서드의 리턴값은 펑크 난 타이어의 번호인데, 정상(0), 앞왼쪽(1), 앞오른쪽(2), 뒤왼쪽(3), 뒤오른쪽(4)인 값이다.

[Sample.java] 실행 클래스

```
public class Sample {
    public static void main(String[] args) {
        Car car = new Car(); //Car 객체 생성

        for(int i=1; i<=5; i++) { //Car 객체의 run() 메서드 5번 반복 실행
            int problemLocation = car.run();

            switch(problemLocation) {
                case 1: //앞왼쪽 타이어가 펑크 났을때 HankookTire로 교체
                    System.out.println("앞왼쪽 HankookTire로 교체");
                    car.frontLeftTire = new HankookTire("앞왼쪽", 15);
                    break;

                case 2: //앞오른쪽 타이어가 펑크 났을때 KumhoTire로 교체
                    System.out.println("앞오른쪽 KumhoTire로 교체");
                    car.frontRightTire = new KumhoTire("앞오른쪽", 13);
                    break;

                case 3: //뒤왼쪽 타이어가 펑크 났을때 HankookTire로 교체
                    System.out.println("뒤왼쪽 HankookTire로 교체");
                    car.backLeftTire = new HankookTire("뒤왼쪽", 14);
                    break;

                case 4: //뒤오른쪽 타이어가 펑크 났을때 KumhoTire로 교체
                    System.out.println("뒤오른쪽 KumhoTire로 교체");
                    car.backRightTire = new KumhoTire("뒤오른쪽", 17);
                    break;
            }

            System.out.println("-----"); //1회전 시 출력되는 내용을 구분
        }
    }
}
```

[실행결과]

```
[자동차가 달립니다.]
앞왼쪽 Tire 수명:5회
앞오른쪽 Tire 수명:1회
뒤왼쪽 Tire 수명:2회
뒤오른쪽 Tire 수명:3회
-----
[자동차가 달립니다.]
앞왼쪽 Tire 수명:4회
*** 앞오른쪽Tire 펑크 ***
[자동차가 멈춥니다.]
앞오른쪽 KumhoTire로 교체
-----
[자동차가 달립니다.]
앞왼쪽 Tire 수명:3회
앞오른쪽 KumhoTire 수명:12회
뒤왼쪽 Tire 수명:1회
뒤오른쪽 Tire 수명:2회
-----
[자동차가 달립니다.]
앞왼쪽 Tire 수명:2회
앞오른쪽 KumhoTire 수명:11회
*** 뒤왼쪽Tire 펑크 ***
[자동차가 멈춥니다.]
뒤왼쪽 HankookTire로 교체
-----
[자동차가 달립니다.]
앞왼쪽 Tire 수명:1회
앞오른쪽 KumhoTire 수명:10회
뒤왼쪽 HankookTire 수명:13회
뒤오른쪽 Tire 수명:1회
-----
```

7.7.3 하나의 배열로 객체 관리

Car 클래스에 4개의 타이어 객체를 4개의 필드로 각각 저장했다. 타이어객체들은 타이어배열로 관리하면 깔끔하게 만들어 줄 수 있다. 아래 예제는 이전 예제에서 작성한 Car 클래스의 타이어 필드를 배열로 수정한 전체 내용을 보여준다.

[Car.java] Tire를 부품으로 가지는 클래스

```
public class Car {
    //필드정의
    Tire[] tires = {
        new Tire("앞왼쪽", 6),
        new Tire("앞오른쪽", 2),
        new Tire("뒤왼쪽", 3),
        new Tire("뒤오른쪽", 4)
    };

    //메서드정의
    int run() {
        System.out.println("[자동차가 달립니다.]");
        for(int i=0; i<tires.length; i++) {
            if(tires[i].roll()==false) {
                stop();
                return(i+1);
            }
        }
        return 0;
    }

    void stop() {
        System.out.println("[자동차가 멈춥니다.]");
    }
}
```

problemLocation은 1~4까지의 값을 가지므로 Car 클래스의 해당 타이어 인덱스는 problemLocation에서 1을 뺀 0~3까지이다. car.tires[problemLocation-1], location은 펑크 난 타이어의 위치 정보로, "앞왼쪽", "앞오른쪽", "뒤왼쪽", "뒤오른쪽"의 값을 얻는다.

[Sample.java] 실행 클래스

```
public class Sample {
    public static void main(String[] args) {
        Car car = new Car();
        for(int i=1; i<=5; i++) {
            int problemLocation = car.run();
            if(problemLocation != 0) {
                System.out.println(car.tires[problemLocation-1].location + "HankookTire로 교체");
                car.tires[problemLocation-1]=new HankookTire(car.tires[problemLocation-1].location, 15);
            }
            System.out.println("-----");
        }
    }
}
```

7.7.4 매개 변수의 다형성

매개 변수 타입이 클래스일 경우, 해당 클래스의 객체뿐만 아니라 자식 객체까지도 매개값으로 사용할 수 있다는 것이다. 매개값을 어떤 자식 객체가 제공되느냐에 따라 메서드의 실행 결과는 다양해질 수 있다(매개 변수의 다형성). 자식 객체가 부모의 메서드를 재정의(오버라이딩)했다면 메서드 내부에서 오버라이딩된 메서드를 호출함으로써 메서드의 실행 결과는 다양해진다.

아래는 Vehicle, Driver, Bus, Taxi 클래스를 이용해서 실행하는 클래스이다. 먼저 Driver 객체와 Bus, Taxi 객체를 생성하고 Driver 객체의 drive() 메서드를 호출할 때 Bus 객체와 Taxi 객체를 제공했다. 결과적으로 매개값이 자동 타입 변환과 메서드 오버라이딩을 통해 매개 변수의 다형성을 구현할 수 있다.

[Vehicle.java] 부모 클래스

```
public class Vehicle {
    public void run() {
        System.out.println("차량이 달립니다.");
    }
}
```

[Driver.java] Vehicle을 이용하는 클래스

```
public class Driver {
    public void drive(Vehicle vehicle) {
        vehicle.run();
    }
}
```

[Bus.java] 자식 클래스

```
public class Bus extends Vehicle {
    @Override
    public void run() {
        System.out.println("버스가 달립니다.");
    }
}
```

[Taxi.java] 자식 클래스

```
public class Taxi extends Vehicle {
    @Override
    public void run() {
        System.out.println("택시가 달립니다.");
    }
}
```

[Sample.java] 실행 클래스

```
public class Sample {
    public static void main(String[] args) {
        Driver driver = new Driver();

        Bus bus = new Bus();
        Taxi taxi = new Taxi();

        driver.drive(bus); //자동 타입 변환: Vehicle vehicle=bus;
        driver.drive(taxi); //자동 타입 변환: Vehicle vehicle=taxi;
    }
}
```

[실행결과]

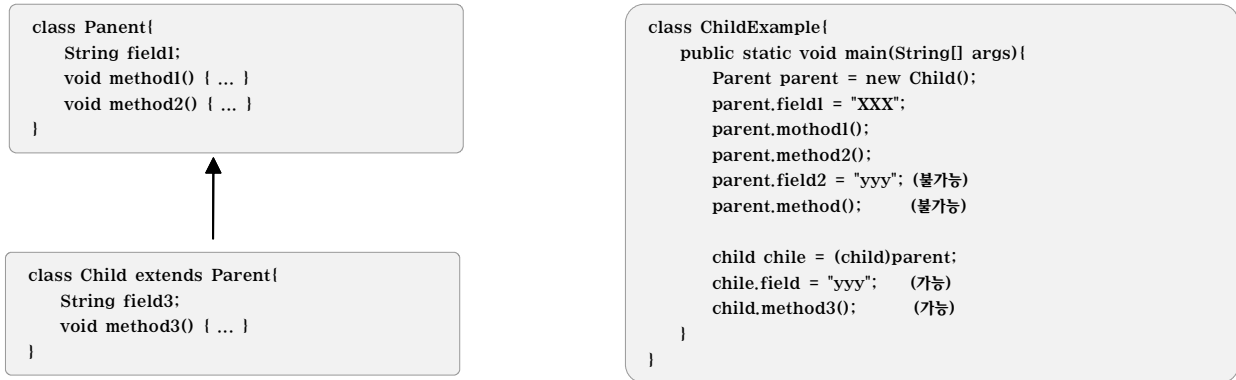
버스가 달립니다.
택시가 달립니다.

7.7.5 강제 타입 변환(Casting)

강제 타입 변환(Casting)은 부모 타입을 자식 타입으로 변환하는 것을 말한다. 그렇다고 해서 모든 부모 타입을 자식 클래스 타입으로 강제 변환할 수 있는 것은 아니다. 자식 타입이 부모 타입으로 자동 변환한 후, 다시 자식 타입으로 변환할 때 강제 타입 변환을 사용할 수 있다.

```
자식 클래스 변수 = (자식클래스) 부모클래스 타입;    //자식 타입이 부모 타입으로 변환된 상태
```

자식 타입이 부모 타입으로 자동 변환하면, 부모 타입에 선언된 필드와 메서드만 사용 가능하다는 제약 사항이 따른다. 만약 자식 타입에 선언된 필드와 메서드를 꼭 사용해야 한다면 강제 타입 변환을 해서 다시 자식 타입으로 변환한 다음 자식 타입의 필드와 메서드를 사용하면 된다.



field2 필드와 method3() 메서드는 Child 타입에만 선언되어 있으므로 Parent 타입으로 자동 변환하면 사용할 수 없다. field2 필드와 method3() 메서드를 사용하고 싶다면 다시 Child 타입으로 강제 타입 변환을 해야 한다.

[Parent.java] 부모 클래스

```
public class Parent {
    public String field1;
    public void method1() {
        System.out.println("Parent-method1()");
    }
    public void method2() {
        System.out.println("Parent-method2()");
    }
}
```

[Child.java] 자식 클래스

```
public class Child extends Parent {
    public String field2;
    public void method3() {
        System.out.println("Child-method3()");
    }
}
```

[Sample.java] 강제 타입 변환(캐스팅)

```
public class Sample {
    public static void main(String[] args) {
        Parent parent = new Child(); //자동 타입 변환
        parent.field1 = "data1";

        parent.method1();
        parent.method2();

        //parent.field2 = "data2"; (불가능)
        //parent.method3(); (불가능)

        Child child = (Child)parent; //강제 타입 변환
        child.field2 = "yyy"; //(가능)
        child.method3(); //(가능)
    }
}
```

[실행결과]

```
Parent-method1()
Parent-method2()
Child-method3()
```

7.7.6 객체 타입 확인(instanceof)

강제 타입 변환은 자식 타입이 부모 타입으로 변환되어 있는 상태에서만 가능하기 때문에 아래와 같이 부모 타입의 변수가 부모 객체를 참조할 경우 자식 타입으로 변환할 수 없다.

```
Parent parent = new Parent();
Child child = (Child) parent;    //강제 타입 변환을 할 수 없다.
```

어떤 객체가 어떤 클래스의 인스턴스인지 확인하려면 instanceof 연산자를 사용할 수 있다. instanceof 연산자의 좌항은 객체가 오고, 우항은 타입이 오는데, 좌항의 객체가 우항의 인스턴스이면 즉 우항의 타입으로 객체가 생성되었다면 true를 산출하고 그렇지 않으면 false를 산출한다.

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

instanceof 연산자는 매개값의 타입을 조사할 때 주로 사용된다. 메서드 내에서 강제 타입 변환이 필요할 경우 반드시 매개값이 어떤 객체인지 instanceof 연산자로 확인하고 안전하게 강제 타입 변환을 해야 한다.

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

만약 타입을 확인하지 않고 강제 타입 변환을 시도한다면 ClassCastException 예외가 발생한다. 아래 예제는 InstanceofExample 클래스에서 method1()과 method2()는 모두 parent 타입의 매개값을 받도록 선언했다.

[Parent.java] 부모 클래스

```
public class Parent {
}
```

[Child.java] 자식 클래스

```
public class Child extends Parent {
}
```

Sample 클래스에서 method1()과 method2()를 호출할 경우, Child 객체를 매개값으로 전달하면 두 메서드 모두 예외가 발생하지 않지만, Parent 객체를 매개값으로 전달하면 method2()에서는 예외가 발생한다.. 예외가 발생하면 즉시 종료되기 때문에 method1()과 같이 강제 타입 변환을 하기 전에 instanceof 연산자로 변환시킬 타입의 객체인지 조사해서 잘못된 매개값으로 인해 프로그램이 종료되는 것을 막아야 한다.

[Sample.java] 강제 타입 변환(캐스팅)

```
public class Sample {
    public static void method1(Parent parent) {
        if(parent instanceof Child) {    //Child 타입으로 변환이 가능한지 확인
            Child child = (Child) parent;
            System.out.println("method1-Child로 변환 성공");
        } else {
            System.out.println("method1-Child로 변환되지 않음");
        }
    }
    public static void method2(Parent parent) {
        Child child = (Child) parent;    //ClassCastException이 발생할 가능성 있음
        System.out.println("method2 - Child로 변환 성공");
    }
    public static void main(String[] args) {
        Parent parentA = new Child();
        method1(parentA);    //Child 객체를 매개값으로 전달
        method2(parentA);

        Parent parentB = new Parent();
        method1(parentB);    //Parent 객체를 매개값으로 전달
        method2(parentB);
    }
}
```

[실행결과]

```
method1-Child로 변환 성공
method2 - Child로 변환 성공
method1-Child로 변환되지 않음
Exception in thread "main" java.lang.ClassCastException
```


7.8 추상 클래스

7.8.1 추상 클래스의 개념

사전적 의미로 추상(abstract)은 실제 간에 공통되는 특징을 추출한 것을 말한다. 객체를 직접 생성할 수 있는 클래스를 실제 클래스라고 한다면 이 클래스들의 공통적인 특징을 추출해서 선언한 클래스를 추상 클래스라고 한다. 객체를 직접 생성해서 사용할 수 없고 새로운 실제 클래스를 만들기 위해 부모 클래스로만 사용된다.

7.8.2 추상 클래스의 용도

추상 클래스로 만드는 이유는 실제 클래스들의 공통된 필드와 메서드의 이름을 통일할 목적과 실제 클래스를 작성할 때 시간을 절약할 수 있다.

7.8.3 추상 클래스 선언

추상 클래스를 선언할 때에는 클래스 선언에 `abstract` 키워드를 붙여야 한다. `abstract`를 붙이게 되면 `new` 연산자를 이용해서 객체를 만들지 못하고 상속을 통해 자식 클래스만 만들 수 있다. 추상 클래스도 일반 클래스와 마찬가지로 필드, 생성자, 메서드 선언을 할 수 있다.

[Phone.java] 추상 클래스

```
public abstract class Phone {
    //필드정의
    public String owner;

    //생성자정의
    public Phone(String owner) {
        this.owner = owner;
    }

    //메서드정의
    public void turnOn() {
        System.out.println("폰 전원을 켭니다.");
    }
    public void turnOff() {
        System.out.println("폰 전원을 끕니다.");
    }
}
```

아래 예제는 Phone 추상 클래스를 상속해서 SmartPhone 자식 클래스를 정의한 것이다. SmartPhone 클래스의 생성자는 `super(owner)`로 Phone의 생성자를 호출하고 있다.

[SmartPhone.java] 실제 클래스

```
public class SmartPhone extends Phone {
    //생성자정의
    public SmartPhone(String owner) {
        super(owner);
    }

    //메서드정의
    public void internetSearch() {
        System.out.println("인터넷 검색을 합니다.");
    }
}
```

아래 Sample 클래스는 Phone의 생성자를 호출해서 객체를 생성할 수 없음을 보여준다. 대신 자식 클래스인 SmartPhone으로 객체를 생성해서 Phone의 메서드인 `turnOn()`, `turnOff()` 메서드를 사용할 수 있음을 보여준다.

[Sample.java] 추상 클래스 사용

```
public class Sample {
    public static void main(String[] args) {
        //Phone phone = new Phone();
        SmartPhone smartPhone = new SmartPhone("홍길동");

        smartPhone.turnOn(); //Phone의 메서드
        smartPhone.internetSearch();
        smartPhone.turnOff(); //Phone의 메서드
    }
}
```

[실행결과]

```
폰 전원을 켭니다.
인터넷 검색을 합니다.
폰 전원을 끕니다.
```

7.8.4 추상 메서드와 오버라이딩

추상 메서드는 추상 클래스에서만 선언할 수 있는데, 메서드의 선언부만 있고 메서드 실행 내용인 중괄호 {}가 없는 메서드를 말한다. 추상 클래스를 설계할 때, 하위 클래스가 반드시 실행 내용을 채우도록 강요하고 싶은 메서드가 있을 경우, 해당 메서드를 추상 메서드로 선언하면 된다. 자식 클래스는 반드시 추상 메서드를 재정의(오버라이딩)해서 실행 내용을 작성한다.

[Animal.java] 추상 메서드 선언

```
public abstract class Animal { //추상클래스
    public String kind;

    public void breathe() {
        System.out.println("숨을 쉰다.");
    }

    public abstract void sound(); //추상 메서드
}
```

[Dog.java] 추상 메서드 오버라이딩

```
public class Dog extends Animal{
    public Dog() {
        this.kind = "포유류";
    }

    @Override
    public void sound() {
        System.out.println("멍멍"); //추상 메서드 재정의
    }
}
```

[Cat.java] 추상 메서드 오버라이딩

```
public class Cat extends Animal {
    public Cat() {
        this.kind = "포유류";
    }

    @Override
    public void sound() {
        System.out.println("야옹"); //추상 메서드 재정의
    }
}
```

아래 예제 Sample 클래스는 첫 번째로 sound() 메서드를 가장 일반적인 방식 Dog와 Cat 변수로 호출했고 두 번째로는 Animal 변수로 타입 변환해서 메서드를 호출했다. 마지막으로 부모 타입의 매개 변수에 자식 객체를 대입해서 메서드의 다형성을 적용했다.

[Sample.java] 실행 클래스

```
public class Sample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();
        dog.sound();
        cat.sound();
        System.out.println("-----");

        //변수의 자동 타입 변환
        Animal animal = null;
        animal = new Dog(); //재정의된 메서드 호출
        animal.sound();

        animal = new Cat(); //재정의된 메서드 호출
        animal.sound();
        System.out.println("-----");

        //메서드의 다형성
        animalSound(new Dog());
        animalSound(new Cat());
    }

    public static void animalSound(Animal animal) {
        animal.sound();
    }
}
```

[실행결과]

```
멍멍
야옹
-----
멍멍
야옹
-----
멍멍
야옹
```

8장 인터페이스

8.1 인터페이스의 역할

자바에서 인터페이스(interface)는 객체의 사용 방법을 정의한 타입이다. 개발 코드가 인터페이스의 메서드를 호출하면 인터페이스는 객체의 메서드를 호출한다. 그렇기 때문에 개발 코드는 객체의 내부 구조를 알 필요가 없고 인터페이스의 메서드만 알고 있으면 된다. 개발 코드가 직접 객체의 메서드를 호출하지 않는 이유는 개발 코드를 수정하지 않고, 사용하는 객체를 변경할 수 있도록 하기 위해서이다. 인터페이스는 하나의 객체가 아니라 여러 객체들과 사용이 가능하므로 어떤 객체를 사용하느냐에 따라서 실행 내용과 리턴값이 다를 수 있다. 따라서 개발 코드 측면에서는 코드 변경 없이 실행 내용과 리턴값을 다양화할 수 있다는 장점을 가지게 된다.

8.2 인터페이스 선언

인터페이스 ".java" 형태의 소스 파일로 작성되고 컴파일러(javac.exe)를 통해 ".class" 형태로 컴파일되기 때문에 물리적인 형태는 클래스와 동일하다. 차이점은 소스를 작성할 때 선언하는 방법이 다르다.

8.2.1 인터페이스 선언

인터페이스 선언은 class 키워드 대신에 interface 키워드를 사용한다.

```
interface 인터페이스명 {
    타입 상수명 = 값; //상수

    타입 메서드명(매개변수, ... ); //추상 메서드

    default 타입 메서드명(매개변수, ...) { ... } //디폴트 메서드

    static 타입 메서드명(매개변수) { ... } //정적 메서드
}
```

- 인터페이스는 객체 사용 설명서이므로 런타임 시 데이터를 저장할 수 있는 필드를 선언할 수 없다. 그러나 상수 필드는 선언이 가능하다.
- 추상 메서드는 객체가 가지고 있는 메서드를 설명한 것으로 호출할 때 어떤 매개값이 필요하고, 리턴 타입이 무엇인지만 알려준다.
- 디폴트 메서드는 인터페이스에 선언되지만 사실은 객체(구현 객체)가 가지고 있는 인스턴스 메서드라고 생각해야 한다.
- 정적 메서드도 역시 자바 8부터 작성할 수 있는데, 디폴트 메서드와는 달리 객체가 없어도 인터페이스만으로 호출이 가능하다.

8.2.2 상수 필드 선언

인터페이스는 데이터를 저장할 수 없기 때문에 데이터를 저장할 인스턴스 또는 정적 필드를 선언할 수 없고 상수 필드만 선언할 수 있다. 상수는 public static final로 선언하는데 생략할 경우 자동적으로 컴파일 과정에서 붙게 된다. 인터페이스 상수는 static{} 블록으로 초기화할 수 없기 때문에 반드시 선언과 동시에 초기값을 지정해야 한다. 아래 예제는 MAX_VALUE와 MIN_VALUE 상수를 선언한 모습을 보여준다.

[RemoteControl.java] 상수 필드 선언

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

8.2.3 추상 메서드 선언

인터페이스를 통해 호출된 메서드는 최종적으로 객체에서 실행되므로 인터페이스의 메서드는 실행 블록이 필요 없는 추상 메서드로 선언한다. 추상 메서드는 리턴타입, 메서드명, 매개 변수만 기술되고 중괄호 {}를 붙이지 않는 메서드를 말한다. 인터페이스에 선언된 추상 메서드는 모두 public abstract의 특성을 갖기 때문에 public abstract를 생략하더라도 자동적으로 컴파일 과정에서 붙게 된다.

[RemoteControl.java] 메서드 선언

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

    //추상 메서드정의
    public void turnOn();
    public void turnOff();
    public void setVolume(int volume);
}
```

//메서드 선언부만 작성(추상 메서드)

8.2.4 디폴트 메서드 선언

디폴트 메서드는 자바 8에서 추가된 인터페이스의 새로운 멤버이다. 형태는 클래스의 인스턴스 메서드와 동일한데, default 키워드가 리턴 타입 앞에 붙는다. 디폴트 메서드는 public 특성을 갖기 때문에 public을 생략하더라도 자동적으로 컴파일 과정에서 붙게 된다.

[RemoteControl.java] 메서드 선언

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

    //추상 메서드정의
    public void turnOn();
    public void turnOff();
    public void setVolume(int volume);

    //디폴트 메서드정의 (실행 내용까지 작성)
    default void setMute(boolean mute) {
        if(mute) {
            System.out.println("무음 처리합니다.");
        } else {
            System.out.println("무음 해제합니다.");
        }
    }
}
```

8.2.5 정적 메서드 선언

정적 메서드는 디폴트 메서드와 마찬가지로 자바 8에서 추가된 인터페이스의 새로운 멤버이다. 형태는 클래스의 정적 메서드와 완전 동일하다. 정적 메서드는 public 특성을 갖기 때문에 public을 생략하더라도 자동적으로 컴파일 과정에서 붙게 된다. 아래 예제는 RemoteControl 인터페이스에서 배터리를 교환하는 기능을 가진 changeBattery() 정적 메서드를 선언하였다.

[RemoteControl.java] 메서드 선언

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

    //메서드 선언부만 작성 (추상메서드)
    public void turnOn();
    public void turnOff();
    public void setVolume(int volume);

    //디폴트 메서드 (실행 내용까지 작성)
    default void setMute(boolean mute) {
        if(mute){
            System.out.println("무음 처리합니다.");
        } else {
            System.out.println("무음 해제합니다.");
        }
    }

    //정적 메서드정의
    static void changeBattery() {
        System.out.println("건전지를 교환합니다.");
    }
}
```

8.3 인터페이스 구현

개발 코드가 인터페이스 메서드를 호출하면 인터페이스는 객체의 메서드를 호출한다. 객체는 인터페이스에서 정의된 추상 메서드와 동일한 메서드 이름, 매개 타입, 리턴 타입을 가진 실제 메서드를 가지고 있어야 한다. 이러한 객체를 인터페이스의 구현(implement) 객체라고 하고, 구현 객체를 생성하는 클래스를 구현 클래스라고 한다.

8.3.1 구현 클래스

구현 클래스는 인터페이스 타입으로 사용할 수 있음을 알려주기 위해 클래스 implements 키워드를 추가하고 인터페이스명을 명시해야 한다.

아래 예제는 Television과 Audio라는 이름을 가지고 있는 RemoteControl 구현 클래스를 작성하는 방법을 보여준다.

[Television.java] 구현 클래스

```
public class Television implements RemoteControl {  
    //필드정의  
    private int volume;  
  
    @Override  
    public void turnOn() {  
        System.out.println("TV를 켭니다.");  
    } //turnOn() 추상 메서드의 실제 메서드  
  
    @Override  
    public void turnOff() {  
        System.out.println("TV를 끕니다.");  
    } //turnOff() 추상 메서드의 실제 메서드  
  
    @Override  
    public void setVolume(int volume) { //인터페이스 상수를 이용해서 volume 필드의 값을 제한  
        if(volume > RemoteControl.MAX_VOLUME) {  
            this.volume = RemoteControl.MAX_VOLUME;  
        } else if(volume < RemoteControl.MIN_VOLUME) {  
            this.volume = RemoteControl.MIN_VOLUME;  
        } else {  
            this.volume = volume;  
        }  
        System.out.println("현재 TV 볼륨: " + volume);  
    } //setVolume() 추상 메서드의 실제 메서드  
}
```

[Audio.java] 구현 클래스

```
public class Audio implements RemoteControl {  
    //필드정의  
    private int volume;  
  
    @Override  
    public void turnOn() {  
        System.out.println("Audio를 켭니다.");  
    } //turnOn() 추상 메서드의 실제 메서드  
  
    @Override  
    public void turnOff() {  
        System.out.println("Audio를 끕니다.");  
    } //turnOff() 추상 메서드의 실제 메서드  
  
    @Override  
    public void setVolume(int volume) { //인터페이스 상수를 이용해서 volume 필드의 값을 제한  
        if(volume > RemoteControl.MAX_VOLUME){  
            this.volume = RemoteControl.MAX_VOLUME;  
        } else if(volume < RemoteControl.MIN_VOLUME){  
            this.volume = RemoteControl.MIN_VOLUME;  
        } else{  
            this.volume = volume;  
        }  
        System.out.println("현재 Audio 볼륨: " + volume);  
    } //setVolume() 추상 메서드의 실제 메서드  
}
```

Television과 Audio를 사용하려면 아래와 같이 RemoteControl 타입 변수 rc를 선언하고 구현 객체를 대입해야 한다.

[Sample.java] 익명 구현 클래스

```
public class Sample {  
    public static void main(String[] args) {  
        RemoteControl rc;  
        rc = new Television();  
        rc = new Audio();  
    }  
}
```

8.3.2 익명 구현 객체

구현 클래스를 만들어 사용하는 것이 일반적이고, 클래스를 재사용할 수 있기 때문에 편리하지만, 일회성의 구현 객체를 만들기 위해 소스 파일을 만들고 클래스를 선언하는 것이 비효율적이다. 자바는 소스 파일을 만들지 않고도 구현 객체를 만들 수 있는 방법을 제공하는데, 그것이 익명 구현 객체이다. 다음은 익명 구현 객체를 생성해서 인터페이스 변수에 대입하는 코드이다.

```
인터페이스 변수 = new 인터페이스() { //인터페이스에 선언된 추상 };
```

new 연산자 뒤에는 클래스 이름이 와야 하는데 이름이 없다. 인터페이스() {}는 인터페이스를 구현해줄 중괄호 {}와 같이 클래스를 선언하라는 뜻이고, new 연산자는 이렇게 선언된 클래스를 객체로 생성한다. 중괄호 {}에는 인터페이스에 선언된 모든 추상 메서드들의 실제 메서드를 작성해야 한다. 그렇지 않으면 컴파일 에러가 발생한다. 추가적으로 필드와 메서드를 선언할 수 있지만, 익명 객체 안에서만 사용할 수 있고 인터페이스 변수로 접근할 수 없다. 아래 예제는 RemoteControl의 익명 구현 객체를 만들어 본 것이다.

[Sample.java] 익명 구현 클래스

```
public class Sample {
    public static void main(String[] args) {
        RemoteControl rc = new RemoteControl() {
            public void turnOn() { /*실행문*/ }
            public void turnOff() { /*실행문*/ }
            public void setVolume(int volume) { /*실행문*/ }
        };
    }
}
```

8.3.3. 다중 인터페이스 구현 클래스

객체는 다수의 인터페이스 타입으로 사용할 수 있다. 다중 인터페이스를 구현할 경우, 구현 클래스는 모든 인터페이스의 추상 메서드에 대해 실제 메서드를 작성해야 한다. 아래 예제는 인터넷을 검색할 수 있는 Searchable 인터페이스이고 search() 추상 메서드는 매개값으로 URL을 받는다.

[Searchable.java] 인터페이스

```
public interface Searchable {
    void search(String url);
}
```

[SmartTelevision.java] 다중 인터페이스 구현 클래스

```
public class SmartTelevision implements RemoteControl, Searchable {
    private int volume;
```

```
    @Override
    public void search(String url) {
        System.out.println(url + "을 검색합니다.");
    }
```

//Searchable의 추상 메서드에 대한 실제 메서드

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

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

```
    @Override
    public void setVolume(int volume) {
        if(volume > RemoteControl.MAX_VOLUME) {
            this.volume = RemoteControl.MAX_VOLUME;
        } else if(volume < RemoteControl.MIN_VOLUME) {
            this.volume = RemoteControl.MIN_VOLUME;
        } else {
            this.volume = volume;
        }
        System.out.println("현재 TV 볼륨: " + volume);
    }
}
```

//RemoteControl의 추상 메서드에 대한 실제 메서드

8.4 인터페이스 사용

인터페이스로 구현 객체를 사용하려면 인터페이스 변수를 선언하고 구현 객체를 대입해야 한다. 인터페이스 변수는 참조 타입이기 때문에 구현 객체가 대입될 경우 구현 객체의 번지를 저장한다. 개발 코드에서 인터페이스는 클래스의 필드, 생성자 또는 메서드의 매개 변수, 생성자 또는 메서드의 로컬 변수로 선언될 수 있다.

8.4.1 추상 메서드 사용

구현 객체가 인터페이스 타입에 대입되면 인터페이스에 선언된 추상 메서드를 호출 할 수 있게 된다. 개발 코드에서 RemoteControl의 변수 rc로 turnOn() 또는 turnOff() 메서드를 호출되면 구현 객체의 turnOn()과 turnOff()메서드가 자동 실행된다.

[Sample.java] 인터페이스 사용

```
public class Sample {
    public static void main(String[] args) {
        RemoteControl rc = null; //인터페이스 변수 선언

        rc = new Television(); //Television 객체를 인터페이스 타입에 대입

        rc.turnOn(); //인터페이스의 turnOn(), turnOff() 호출
        rc.turnOff();

        rc = new Audio(); //Audio 객체를 인터페이스 타입에 대입

        rc.turnOn(); //인터페이스의 turnOn(), turnOff() 호출
        rc.turnOff();
    }
}
```

[실행결과]

TV를 켭니다.
TV를 끕니다.
Audio를 켭니다.
Audio를 끕니다.

8.4.2 디폴트 메서드 사용

디폴트 메서드는 인터페이스에 선언되지만, 인터페이스에서 바로 사용할 수 없다. 디폴트 메서드는 추상 메서드가 아닌 인스턴스 메서드이므로 구현 객체가 있어야 사용할 수 있다.

[Audio.java] 구현 클래스

```
public class Audio implements RemoteControl{
    //필드정의
    private int volume;
    private boolean mute;

    ...

    @Override
    public void setMute(boolean mute) {
        this.mute = mute;

        if(mute) {
            System.out.println("Audio 무음 처리합니다.");
        }else {
            System.out.println("Audio 무음 해제합니다.");
        }
    }
}
```

//디폴트 메서드 재정의

[Sample.java] 디폴트 메서드 사용

```
public class Sample {
    public static void main(String[] args) {
        RemoteControl rc = null;

        rc = new Television();
        rc.turnOn();
        rc.setMute(true);

        rc = new Audio();
        rc.turnOn();
        rc.setMute(true);
    }
}
```

[실행결과]

TV를 켭니다.
무음 처리합니다.
Audio를 켭니다.
Audio 무음 처리합니다.

8.4.3 정적 메서드 사용

인터페이스의 정적 메서드는 인터페이스로 바로 호출이 가능하다. 아래 예제는 RemoteControl의 ChangeBattery() 정적 메서드를 호출한다.

[Sample.java] 정적 메서드 사용

```
public class Sample {  
    public static void main(String[] args) {  
        RemoteControl.changeBattery();  
    }  
}
```

[실행결과]

건전지를 교환합니다.

8.5 타입 변환과 다형성

8.5.1 자동 타입 변환(Promotion)

구현 객체가 인터페이스 타입으로 변환되는 것은 자동 타입 변환(Promotion)에 해당한다. 자동 타입 변환은 프로그램 실행 도중에 자동적으로 타입 변환이 일어나는 것을 말한다.

8.5.2 필드의 다형성

아래 예제의 Tire는 클래스 타입이 아니고 인터페이스이며 HankookTire와 KumhoTire는 자식 클래스가 아니라 구현 클래스이다.

[Tire.java] 인터페이스

```
public interface Tire {  
    public void roll(); //roll() 메서드 호출 방법 설명  
}
```

[KumhoTire.java] 구현 클래스

```
public class KumhoTire implements Tire {  
    @Override  
    public void roll() {  
        System.out.println("금호 타이어가 굴러갑니다."); //Tire 인터페이스 구현  
    }  
}
```

[HankookTire.java] 구현 클래스

```
public class HankookTire implements Tire {  
    @Override  
    public void roll() {  
        System.out.println("한국 타이어가 굴러갑니다."); //Tire 인터페이스 구현  
    }  
}
```

Car 클래스를 설계할 때 아래와 같이 필드 타입으로 타이어 인터페이스를 선언하게 되면 필드값으로 한국 타이어 또는 금호 타이어 객체를 대입할 수 있다. 자동 타입 변환이 일어나기 때문에 아무런 문제가 없다.

[Car.java] 필드 다형성

```
public class Car {  
    Tire frontLeftTire = new HankookTire();  
    Tire frontRightTire = new HankookTire(); //인터페이스 타입 필드 선언과 초기 구현 객체 대입  
    Tire backLeftTire = new HankookTire();  
    Tire backRightTire = new HankookTire();  
  
    void run() {  
        frontLeftTire.roll();  
        frontRightTire.roll();  
        backLeftTire.roll();  
        backRightTire.roll(); //인터페이스에서 설명된 roll()메서드 호출  
    }  
}
```

frontRightTire와 backLeftTire를 교체하기 전에는 HankookTire 객체의 roll() 메서드가 호출되지만, kumhoTire로 교체된 후에는 KumhoTire 객체의 roll() 메서드가 호출된다. Car의 run() 메서드 수정 없이도 다양한 roll() 메서드의 실행 결과를 얻을 수 있게 되는 것이다. 이것이 바로 필드의 다형성이다.

[Sample.java] 필드 다형성 테스트

```
public class Sample {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.run();

        myCar.frontLeftTire = new KumhoTire();

        myCar.frontRightTire = new KumhoTire();
        myCar.run();
    }
}
```

[실행결과]

```
한국 타이어가 굴러갑니다.
한국 타이어가 굴러갑니다.
한국 타이어가 굴러갑니다.
한국 타이어가 굴러갑니다.
금호 타이어가 굴러갑니다.
금호 타이어가 굴러갑니다.
한국 타이어가 굴러갑니다.
한국 타이어가 굴러갑니다.
```

8.5.3 인터페이스 배열로 구현 객체 관리

이전 예제에서는 Car 클래스에서 4개의 타이어 필드를 인터페이스로 각각 선언했지만 아래와 같이 인터페이스 배열로 관리할 수도 있다. tires 배열의 각 항목은 Tire 인터페이스 타입이므로, 구현 객체인 KumhoTire를 대입하면 자동 타입 변환이 발생하기 때문에 아무런 문제가 없다.

[Car.java] 필드 다형성

```
public class Car {
    Tire[] tires = {
        new HankookTire(),
        new HankookTire(),
        new HankookTire(),
        new HankookTire()
    };
    void run() {
        for(Tire tire: tires) {
            tire.roll();
        }
    }
}
```

구현 객체들을 배열로 관리하면 제어문에서 가장 많이 혜택을 본다. 예를 들어 전체 타이어의 roll() 메서드를 호출하는 Car 클래스의 run() 메서드는 아래와 같이 for문으로 작성할 수 있다.

[Sample.java] 필드 다형성 테스트

```
public class Sample {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.run();

        myCar.tires[0] = new KumhoTire();
        myCar.tires[1] = new KumhoTire();

        myCar.run();
    }
}
```

[실행결과]

```
한국 타이어가 굴러갑니다.
한국 타이어가 굴러갑니다.
한국 타이어가 굴러갑니다.
한국 타이어가 굴러갑니다.
금호 타이어가 굴러갑니다.
금호 타이어가 굴러갑니다.
한국 타이어가 굴러갑니다.
한국 타이어가 굴러갑니다.
```

8.5.4 매개 변수의 다형성

자동 타입 변환은 필드의 값을 대입할 때에도 발생하지만, 주로 메서드를 호출할 때 많이 발생한다. 매개값을 다양화하기 위해서 상속에서는 매개 변수를 부모 타입으로 선언하고 호출할 때에는 자식 객체를 대입하였다. 아래 예제는 Sample 클래스를 실행해서 매개 변수 다형성을 이해해 본다.

[Driver.java] 매개 변수의 인터페이스화

```
public class Driver {
    public void drive(Vehicle vehicle) {
        vehicle.run();
    }
}
```

[Vehicle.java] 인터페이스

```
public interface Vehicle {
    public void run();
}
```

[Bus.java] 구현 클래스

```
public class Bus implements Vehicle {
    @Override
    public void run() {
        System.out.println("버스가 달립니다.");
    }
}
```

[Taxi.java] 구현 클래스

```
public class Taxi implements Vehicle {
    @Override
    public void run() {
        System.out.println("택시가 달립니다.");
    }
}
```

Bus와 Taxi는 구현 클래스이며 Driver의 drive() 메서드를 호출할 때 Bus, Taxi 객체를 생성해서 매개값으로 줄 수 있다. drive() 메서드는 Vehicle 타입을 매개 변수로 선언했지만, Vehicle을 구현한 Bus나 Taxi 객체가 매개값으로 사용되면 자동 타입 변환이 발생한다.

[Sample.java] 매개 변수의 다형성 테스트

```
public class Sample {
    public static void main(String[] args) {
        Driver driver = new Driver();

        Bus bus = new Bus();
        Taxi taxi = new Taxi();

        driver.drive(bus); //자동 타입 변환: Vehicle vehicle = bus;
        driver.drive(taxi); //자동 타입 변환: Vehicle vehicle = taxi;
    }
}
```

[실행결과]

버스가 달립니다.
택시가 달립니다.

8.5.5 강제 타입 변환(Casting)

구현 객체가 인터페이스 타입으로 자동 변환하면, 인터페이스에 선언된 메서드만 사용 가능하다는 제약 사항이 따른다. 예를 들어 인터페이스에는 세 개의 메서드가 선언되어 있고, 클래스에는 다섯 개의 메서드가 선언되어 있다면, 인터페이스로 호출 가능한 메서드는 세 개뿐이다. 하지만 경우에 따라서는 구현 클래스에 선언된 필드와 메서드를 사용해야 할 경우도 발생한다. 이때 강제 타입 변환을 해서 다시 구현 클래스 타입으로 변환한 다음, 구현 클래스의 필드와 메서드를 사용할 수 있다.

[Vehicle.java] 인터페이스

```
public interface Vehicle {
    public void run();
}
```

[Bus.java] 구현 클래스

```
public class Bus implements Vehicle {
    @Override
    public void run() {
        System.out.println("버스가 달립니다.");
    }

    public void checkFare() {
        System.out.println("승차요금을 체크합니다.");
    }
}
```

[Sample.java] 강제 타입 변환

```
public class Sample {
    public static void main(String[] args) {
        Vehicle vehicle = new Bus();

        vehicle.run();
        //vehicle.checkFare(); (X) //Vehicle 인터페이스에는 checkFare()가 없음

        Bus bus = (Bus)vehicle; //강제타입변환
        bus.run();
        bus.checkFare(); //Bus 클래스에는 checkFare()가 있음
    }
}
```

[실행결과]

버스가 달립니다.
버스가 달립니다.
승차요금을 체크합니다.

8.5.6 객체 타입 확인(instanceof)

인터페이스 타입으로 자동 변환된 매개값을 메서드 내에서 다시 구현 클래스 타입으로 강제 타입 변환해야 한다면 반드시 매개값이 어떤 객체인지 instanceof 연산자로 확인하고 안전하게 강제 타입 변환을 해야 한다. 아래 예제와 같이 drive() 메서드에서 매개값이 Bus 객체인 경우, Bus의 checkFare() 메서드를 호출해야 한다면 Bus 타입으로 강제 타입 변환을 해야 한다. Vehicle 인터페이스에는 checkFare() 메서드가 없기 때문이다. 매개값으로 어떤 구현 객체가 대입될지 모르기 때문에 instanceof 연산자로 Bus 타입인지 꼭 확인해야 한다.

[Driver.java] 객체 타입 확인

```
public class Driver {
    public void drive(Vehicle vehicle) {
        if(vehicle instanceof Bus) { //vehicle 매개 변수가 참조하는 객체가 Bus인지 조사
            Bus bus = (Bus) vehicle; //Bus 객체일 경우 안전하게 강제 타입 변환시킴
            bus.checkFare(); //Bus 타입으로 강제 타입 변환을 하는 이유
        }
        vehicle.run();
    }
}
```

8.6 인터페이스 상속

인터페이스도 다른 인터페이스를 상속할 수 있다. 인터페이스는 클래스와는 달리 다중 상속을 허용한다. extends 키워드 뒤에 상속할 인터페이스들을 나열할 수 있다. 하위 인터페이스를 구현하는 클래스는 하위 인터페이스의 메서드뿐만 아니라 상위 인터페이스의 모든 추상 메서드에 대한 실제 메서드를 가지고 있어야 한다. 그렇기 때문에 구현 클래스로부터 객체를 생성하고 나서 하위 및 상위 인터페이스 타입으로 변환이 가능하다. 하위 인터페이스로 타입 변환이 되면 상,하위 인터페이스에 선언된 모든 메서드를 사용할 수 있으나, 상위 인터페이스로 타입 변환되면 상위 인터페이스에 선언된 메서드만 사용 가능하고 하위 인터페이스에 선언된 메서드는 사용할 수 있다.

아래 예제 InterfaceC 인터페이스 변수는 methodA(), methodB(), methodC()를 모두 호출할 수 있지만 InterfaceA와 InterfaceB 변수는 각각 methodA()와 methodB()만 호출할 수 있다.

[InterfaceA.java] 부모 인터페이스

```
public interface InterfaceA {
    public void methodA();
}
```

[InterfaceB.java] 부모 인터페이스

```
public interface InterfaceB {
    public void methodB();
}
```

[InterfaceC.java] 하위 인터페이스

```
public interface InterfaceC extends InterfaceA, InterfaceB {
    public void methodC();
}
```

[ImplementationC.java] 하위 인터페이스 구현

```
public class ImplementationC implements InterfaceC {
    //InterfaceA에 실제 메서드도 있어야 한다.
    @Override
    public void methodA() {
        System.out.println("ImplementationC-methodA() 실행");
    }

    //InterfaceB에 실제 메서드도 있어야 한다.
    @Override
    public void methodB() {
        System.out.println("ImplementationC-methodB() 실행");
    }

    @Override
    public void methodC() {
        System.out.println("ImplementationC-methodC() 실행");
    }
}
```

[Sample.java] 호출 가능 메서드

```
public class Sample {
    public static void main(String[] args) {
        ImplementationC impl = new ImplementationC();

        InterfaceA ia = impl;
        ia.methodA();
        System.out.println(); //InterfaceA 변수는 methodA()만 호출 가능

        InterfaceB ib = impl;
        ib.methodB(); //InterfaceB 변수는 methodB()만 호출 가능
        System.out.println();

        InterfaceC ic = impl;
        //InterfaceC 변수는 methodA(), methodB(), methodC()모드 호출 가능
        ic.methodA();
        ic.methodB();
        ic.methodC();
    }
}
```

[실행결과]

```
ImplementationC-methodA() 실행
ImplementationC-methodB() 실행
ImplementationC-methodA() 실행
ImplementationC-methodB() 실행
ImplementationC-methodC() 실행
```

8.7 디폴트 메서드와 인터페이스 확장

8.7.1 디폴트 메서드의 필요성

인터페이스에서 디폴트 메서드를 허용한 이유는 기존 인터페이스를 확장해서 새로운 기능을 추가하기 위해서이다. 기존 인터페이스의 이름과 추상 메서드의 변경 없이 디폴트 메서드만 추가할 수 있기 때문에 이전에 개발한 구현 클래스를 그대로 사용할 수 있으면서 새롭게 개발하는 클래스는 디폴트 메서드를 활용할 수 있다. 디폴트 메서드는 추상 메서드가 아니기 때문에 구현 클래스에서 실제 메서드를 작성할 필요가 없다.

[MyInterface.java] 기존 인터페이스

```
public interface MyInterface {
    public void method1();
}
```

[MyClassA.java] 기존 구현 클래스

```
public class MyClassA implements MyInterface{
    @Override
    public void method1() {
        System.out.println("MyClassA-method1() 실행");
    }
}
```

아래와 같이 MyInterface에 디폴트 메서드인 method2()를 추가해서 수정된 MyInterface를 만들었다.

[MyInterface.java] 수정 인터페이스

```
public interface MyInterface {
    public void method1();

    public default void method2() {
        System.out.println("MyInterface-method2 실행");
    }
} //디폴트 메서드
```

[MyClassB.java] 새로운 구현 클래스

```
public class MyClassB implements MyInterface {
    @Override
    public void method1() {
        System.out.println("MyClassB-method1() 실행");
    }

    @Override
    public void method2() {
        System.out.println("MyClassB-method2() 실행");
    }
} //디폴트 메서드 재정의
```

아래실행 결과를 보면 MyClassA의 method2()는 MyInterface에 정의된 디폴트 메서드가 실행되었고, MyClassB의 method2()는 재정의한 MyClassB의 method2()가 실행되었다.

[Sample.java] 디폴트 메서드 사용

```
public class Sample {
    public static void main(String[] args) {
        MyInterface mil = new MyClassA();
        mil.method1();
        mil.method2();

        MyInterface mi2 = new MyClassB();
        mi2.method1();
        mi2.method2();
    }
}
```

[실행결과]

MyClassA-method1() 실행
MyInterface-method2 실행
MyClassB-method1() 실행
MyClassB-method2() 실행

8.7.2 디폴트 메서드가 있는 인터페이스 상속

인터페이스 간에도 상속이 있다는 것을 이미 학습하였다. 부모 인터페이스에 디폴트 메서드가 정의되었을 경우, 자식 인터페이스에서 디폴트 메서드를 활용하는 방법은 아래 세 가지가 있다.

- 디폴트 메서드를 단순히 상속만 받는다.
- 디폴트 메서드를 재정의(Override)해서 실행 내용을 변경한다.
- 디폴트 메서드를 추상 메서드를 재선언한다.
-

아래 예제는 추상 메서드와 디폴트 메서드가 선언된 ParentInterface 부모 인터페이스이다.

[ParentInterface.java] 부모 인터페이스

```
public interface ParentInterface {
    public void method1();
    public default void method2() { /*실행문*/ }
}
```

아래 ChildInterface1은 ParentInterface를 상속하고 자신의 추상 메서드인 method3()을 선언한다.

[ChildInterface1.java] 자식 인터페이스

```
public interface ChildInterface1 extends ParentInterface {
    public void method3();
}
```

아래는 ChildInterface1 인터페이스를 구현하는 클래스는 method1()과 method3()의 실제메서드를 가지고 있어야 하며 ParentInterface의 method2()를 호출할 수 있다.

[Sample.java]

```
public class Sample {
    public static void main(String[] args) {
        ChildInterface1 cil = new ChildInterface1() {
            @Override
            public void method1() { /*실행문*/ }
            @Override
            public void method3() { /*실행문*/ }
        };
        cil.method1();
        //ParentInterface의 method2() 호출
        cil.method2();
        cil.method3();
    }
}
```

아래 ChildInterface2는 ParentInterface를 상속하고 ParentInterface의 디폴트 메서드인 method2()를 재정의 한다. 그리고 추상 메서드인 method3()을 선언한다.

[ChildInterface2.java] 자식 인터페이스

```
public interface ChildInterface2 extends ParentInterface {
    @Override
    //재정의
    public default void method2() { /*실행문*/ }
    public void method3();
}
```

이 경우도 ChildInterface2 인터페이스를 구현하는 클래스는 method1()과 method3()의 실제메서드가 있어야 하며, ChildInterface2에서 재정의 한 method2()를 호출할 수 있다.

[Sample.java]

```
public class Sample {
    public static void main(String[] args) {
        ChildInterface2 ci2 = new ChildInterface2() {
            @Override
            public void method1() { /*실행문*/ }

            @Override
            public void method3() { /*실행문*/ }
        };

        ci2.method1();
        //ChildInterface2의 method2() 호출
        ci2.method2();
        ci2.method3();
    }
}
```

다음 ChildInterface3은 ParentInterface를 상속하고 ParentInterface의 디폴트 메서드인 method2()를 추상 메서드로 재선언한다. 그리고 자신의 추상 메서드인 method3()을 선언한다.

[ChildInterface3.java] 자식 인터페이스

```
public interface ChildInterface3 extends ParentInterface {
    @Override
    //추상 메서드로 재선언
    public void method2();
    public void method3();
}
```

이 경우 ChildInterface3 인터페이스를 구현하는 클래스는 method1()과 method2(), method3()의 실제메서드를 모두 가지고 있어야 한다.

[Sample.java]

```
public class Sample {
    public static void main(String[] args) {
        ChildInterface3 ci3 = new ChildInterface3(){
            @Override
            public void method1() { /*실행문*/ }

            @Override
            public void method2() { /*실행문*/ }

            @Override
            public void method3() { /*실행문*/ }
        };

        ci3.method1();
        //ChildInterface3의 method2() 호출
        ci3.method2();
        ci3.method3();
    }
}
```

9장 중첩 클래스와 중첩 인터페이스

9.1 중첩 클래스와 중첩 인터페이스란?

중첩 클래스(Nested Class)란 클래스 내부에 선언한 클래스를 말하는데, 중첩 클래스를 사용하면 두 클래스의 멤버들을 서로 쉽게 접근할 수 있다는 장점과 외부에는 불필요한 관계 클래스를 감춤으로써 코드의 복잡성을 줄일 수 있다. 인터페이스도 클래스 내부에 선언할 수 있다. 이런 인터페이스를 중첩 인터페이스라고 한다. 인터페이스를 클래스 내부에 선언하는 이유는 해당 클래스와 긴밀한 관계를 맺는 구현 클래스를 만들기 위해서이다.

9.2 중첩 클래스

중첩 클래스는 클래스 내부에 선언되는 위치에 따라서 두 가지로 분류된다. 클래스의 멤버로서 선언되는 중첩 클래스를 멤버 클래스라고 하고, 메서드 내부에서 선언되는 중첩 클래스를 로컬 클래스라고 한다. 멤버 클래스는 클래스나 객체가 사용 중이라면 언제든지 재사용이 가능하지만, 로컬 클래스는 메서드 실행 시에만 사용되고, 메서드가 실행 종료되면 없어진다.

9.2.1 인스턴스 멤버 클래스

인스턴스 멤버 클래스는 `static` 키워드 없이 선언된 클래스를 말한다. 인스턴스 멤버 클래스는 인스턴스 필드와 메서드만 선언이 가능하고 정적 필드와 메서드는 선언할 수 없다. A클래스 외부에서 인스턴스 멤버 클래스B의 객체를 생성하려면 먼저 A객체를 생성하고 B객체를 생성해야 한다.

<pre>class A { class B { //인스턴스 멤버 클래스 B() {} //생성자 int field1; //인스턴스 필드 static int field2; //정적 필드 (X) void method1() {} //인스턴스 메서드 static void method2() {} //정적 메서드 (X) } }</pre>	<pre>A a = new A(); A.B b = a.new B(); b.field1 = 3; b.method1();</pre>
---	---

9.2.2 정적 멤버 클래스

정적 멤버 클래스는 `static` 키워드로 선언된 클래스를 말한다. 정적 멤버 클래스는 모든 종류의 필드와 메서드를 선언할 수 있다. A클래스 외부에서 정적 멤버 클래스 C의 객체를 생성하기 위해서는 A객체를 생성할 필요가 없고, 아래와 같이 C객체를 생성하면 된다.

<pre>class A{ static class C { //정적 멤버 클래스 C() {} //생성자 int field1; //인스턴스 필드 static int field2; //정적 필드 void method1() {} //인스턴스 메서드 static void method2() {} //정적 메서드 } }</pre>	<pre>A.C c = new A.C(); c.field1 = 3; //인스턴스 필드 사용 c.method1(); //인스턴스 메서드 호출 A.C.field2 = 3; //정적 필드 사용 A.C.method2(); //정적 메서드 호출</pre>
---	---

9.2.3 로컬 클래스

중첩 클래스는 메서드 내에서도 선언할 수 있다. 이것을 로컬(local) 클래스라고 한다. 로컬 클래스는 접근 제한자(public, private) 및 `static` 을 붙일 수 없고 메서드 내부에서만 사용되므로 접근을 제한할 필요가 없기 때문이다. 로컬 클래스 내부에는 인스턴스 필드와 메서드만 선언이 가능하고 정적 필드와 메서드는 선언할 수 없다. 로컬 클래스는 메서드가 실행될 때 메서드 내에서 객체를 생성하고 사용해야 한다. 아래와 같이 비동기 처리를 위해 스레드 객체를 만들 때 사용한다.

<pre>void method(){ class D { //로컬 클래스 D() {} //생성자 int field1; //인스턴스 필드 static int field2; //정적 필드(X) void method1() {} //인스턴스 메서드 static void method2() {} //정적 메서드(X) } D d = new D(); d.field1 = 3; d.method1(); }</pre>	<pre>void method() { class DownloadThread extends Thread { ... } DownloadThread thread = new DownloadThread(); thread.start(); }</pre>
--	--

[A.java] 중첩 클래스

```
//바깥 클래스
public class A {
    A() {
        System.out.println("A 객체가 생성됨");
    }

    //인스턴스 멤버 클래스
    class B {
        B() {
            System.out.println("B 객체가 생성됨");
        }
        int field1;
        //static int field2;
        void method1() {}
        //static void method2() {}
    }

    //정적 멤버 클래스
    static class C {
        C() {
            System.out.println("C 객체가 생성됨");
        }
        int field1;
        static int field2;
        void method1() {}
        static void method2() {}
    }

    void method() {
        //로컬 클래스
        class D {
            D() {
                System.out.println("D 객체가 생성됨");
            }
            int field1;
            //static int field2;
            void method1() {}
            //static void method2() {}
        }

        D d = new D();
        d.field1 = 3;
        d.method1();
    }
}
```

[Sample.java] 중첩 클래스 객체 생성

```
public class Sample {
    public static void main(String[] args) {
        A a = new A();

        //인스턴스 멤버 클래스 객체 생성
        A.B b = a.new B();
        b.field1 = 3;
        b.method1();

        //정적 멤버 클래스 객체 생성
        A.C c = new A.C();
        c.field1 = 3;
        c.method1();
        A.C.field2 = 3;
        A.C.method2();

        //로컬 클래스 객체 생성을 위한 메서드 호출
        a.method();
    }
}
```

[실행결과]

```
A 객체가 생성됨
B 객체가 생성됨
C 객체가 생성됨
D 객체가 생성됨
```


9.3 중첩 클래스의 접근 제한

9.3.1 바깥 필드와 메서드에서 사용 제한

[A.java] 바깥 필드와 메서드에서 사용 제한

```
public class A {
    class B { } //인스턴스 멤버 클래스
    static class C { } //정적 멤버 클래스
    //인스턴스 필드
    B field1 = new B();
    C field2 = new C();
    //정적 필드 초기화
    //static B field3 = new B(); (X)
    static C field4 = new C();
    //정적 메서드
    static void method2(){
        //B var1 = new B();
        C var2 = new C();
    }
}
```

9.3.2 멤버 클래스에서 사용 제한

[A.java] 멤버 클래스에서 사용 제한

```
public class A {
    int field1;
    void method1() { }
    static int field2;
    static void method2() { }
    class B {
        void method() {
            field1 = 10;
            method1();
            field2 = 10;
            method2();
        }
    }
    static class C {
        void method() {
            //field1 = 10; method1(); 인스턴스 필드와 메서드는 접근할 수 없다.
            field2 = 10;
            method2();
        }
    }
}
```

9.3.3 로컬 클래스에서 사용 제한

[Outter.java] 로컬 클래스에서 사용 제한

```
public class Outter {
    public void method1(final int arg) { //자바7 이전
        final int localVariable = 1;
        //arg = 100; localVariable = 100; (X)
        class Innter {
            public void method() {
                int result = arg + localVariable;
            }
        }
    }
    public void method2(int arg) { //자바8 이후
        int localVariable = 1;
        //arg = 100; localVariable = 100; (X)
        class Innter {
            public void method() {
                int result = arg + localVariable;
            }
        }
    }
}
```

9.3.4 중첩 클래스에서 바깥 클래스 참조 얻기

클래스 내부에서 this는 객체 자신의 참조이다. 중첩 클래스 내부에서 this.필드, this.메서드()로 호출하면 중첩 클래스의 필드와 메서드가 사용된다. 중첩 클래스 내부에서 바깥 클래스의 객체 참조를 얻으려면 바깥 클래스의 이름을 this 앞에 붙여주면 된다.

[Outter.java] 중첩 클래스에서 바깥 클래스 참조 얻기

```
public class Outter {
    String field = "Outter-field";
    void method() {
        System.out.println("Outter-method");
    }
    class Nested {
        String field = "Nested-field";
        void method() {
            System.out.println("Nested-method");
        }
        void print() {
            System.out.println(this.field); //중첩 객체 참조
            this.method();
            System.out.println(Outter.this.field); //바깥 객체 참조
            Outter.this.method();
        }
    }
}
```

[Sample.java] 실행 클래스

```
public class Sample {
    public static void main(String[] args) {
        Outter outter = new Outter();
        Outter.Nested nested = outter.new Nested();
        nested.print();
    }
}
```

[실행결과]

```
Nested-field
Nested-method
Outter-field
Outter-method
```

9.4 중첩 인터페이스

중첩 인터페이스는 클래스의 멤버로 선언된 인터페이스를 말한다. 인터페이스를 클래스 내부에 선언하는 이유는 해당 클래스와 긴밀한 관계를 맺는 구현 클래스를 만들기 위해서이다. UI 프로그래밍에서 이벤트를 처리할 목적으로 많이 활용한다. 아래 코드는 Button을 클릭했을 때 이벤트를 처리하는 객체를 받는 예제이다. Button 내부에 선언된 중첩 인터페이스를 구현한 객체만 받아야 한다면 아래와 같이 Button 클래스를 선언하면 된다.

[Button.java] 중첩 인터페이스

```
public class Button {
    OnClickListener listener; //인터페이스 타입 필드

    void setOnClickListener(OnClickListener listener) {
        this.listener = listener; //매개 변수의 다형성
    }

    void touch() {
        listener.onClick(); //구현 객체의 onClick()메서드 호출
    }

    interface OnClickListener {
        void onClick(); //중첩인터페이스
    }
}
```

Button 클래스 내용을 보면 중첩 인터페이스(OnClickListener) 타입으로 필드(listener)를 선언하고 setOnClickListener()로 구현 객체를 받아 필드에 대입한다. touch() 메서드가 발생했을 때 인터페이스를 통해 구현 객체의 메서드를 호출(listener.onClick())한다. 아래는 Button의 중첩 인터페이스인 OnClickListener를 구현한 두 개의 클래스를 보여준다.

[CallListener.java] 구현 클래스

```
public class CallListener implements Button.OnClickListener {
    @Override
    public void onClick() {
        System.out.println("전화를 겁니다.");
    }
}
```

[MessageListener.java] 구현 클래스

```
public class MessageListener implements Button.OnClickListener {
    @Override
    public void onClick() {
        System.out.println("메세지를 보냅니다.");
    }
}
```

아래는 버튼을 클릭했을 때 두 가지 방법으로 이벤트를 처리하는 방법이다. 어떤 구현 객체를 생성해서 Button객체의 setOnClickListener() 메서드로 세팅하느냐에 따라서 Button의 touch() 메서드의 실행 결과가 달라진다.

[Sample.java] 버튼 이벤트 처리

```
public class Sample {
    public static void main(String[] args) {
        Button btn = new Button();

        btn.setOnClickListener(new CallListener());
        btn.touch();

        btn.setOnClickListener(new MessageListener());
        btn.touch();
    }
}
```

[실행결과]
전화를 겁니다.
메세지를 보냅니다.

9.5 익명 객체

익명객체는 이름이 없는 객체를 말한다. 익명객체는 단독으로 생성할 수 없고 클래스를 상속하거나 인터페이스를 구현해야만 생성할 수 있다.

9.5.1 익명 자식 객체 생성

[Person.java] 부모클래스

```
public class Person {
    void wake() {
        System.out.println("7시에 일어납니다.");
    }
}
```

[Anonymous.java] 익명 자식 객체 생성

```
public class Anonymous {
    Person field = new Person() {
        void work() {
            System.out.println("출근합니다.");
        }
        @Override
        void wake() {
            System.out.println("6시에 일어납니다.");
            work();
        }
    };
    //필드 선언과 초기값 대입

    void method1(){
        Person localVar = new Person() {
            void walk() {
                System.out.println("산책합니다.");
            }
            @Override
            void wake() {
                System.out.println("7시에 일어납니다.");
                walk();
            }
        };
        //로컬 변수값으로 대입

        localVar.wake(); //로컬 변수 사용
    }

    void method2(Person person) {
        person.wake();
    }
}
```

[Sample.java] 익명 자식 객체 생성

```
public class Sample {
    public static void main(String[] args) {
        Anonymous anony = new Anonymous();

        //익명 객체 필드 사용
        anony.field.wake();

        //익명 객체 로컬 변수 사용
        anony.method1();

        //익명 객체 매개값 사용
        anony.method2(
            new Person() {
                void study() {
                    System.out.println("공부합니다.");
                }
                @Override
                void wake() {
                    System.out.println("8시에 일어납니다.");
                    study();
                }
            }
        );
    }
}
```

[실행결과]

6시에 일어납니다.
출근합니다.
7시에 일어납니다.
산책합니다.
8시에 일어납니다.
공부합니다.

9.5.2 익명 구현 객체 생성

[RemoteControl.java] 인터페이스

```
public interface RemoteControl {
    public void turnOn();
    public void turnOff();
}
```

[Anonymous.java] 익명 구현 클래스와 객체 생성

```
public class Anonymous {
    //필드 선언과 초기값 대입
    RemoteControl field = new RemoteControl() {
        @Override
        public void turnOn() {
            System.out.println("TV를 켭니다.");
        }

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

    //로컬 변수선언과 초기값 대입
    void method1() {
        RemoteControl localVar = new RemoteControl() {
            @Override
            public void turnOn() {
                System.out.println("Audio를 켭니다.");
            }

            @Override
            public void turnOff() {
                System.out.println("Audio를 끕니다.");
            }
        };

        //로컬 변수 사용
        localVar.turnOn();
    }

    void method2(RemoteControl rc) {
        rc.turnOn();
    }
}
```

[Sample.java] 익명 구현 클래스와 객체 생성

```
public class Sample {
    public static void main(String[] args) {
        Anonymous anony = new Anonymous();
        //익명 객체 필드 사용
        anony.field.turnOn();

        //익명 객체 로컬 변수 사용
        anony.method1();

        //익명 객체 매개값 사용
        anony.method2(new RemoteControl() {
            @Override
            public void turnOn() {
                System.out.println("SmartTV를 켭니다.");
            }
            @Override
            public void turnOff() {
                System.out.println("SmartTV를 끕니다.");
            }
        });
    }
}
```

[실행결과]

TV를 켭니다.
Audio를 켭니다.
SmartTV를 켭니다.

9.5.3 익명 객체의 로컬 변수 사용

메서드 내에서 생성된 익명 객체는 메서드 실행이 끝나도 힙 메모리에 존재해서 계속 사용할 수 있다. 그러나 매개 변수나 로컬 변수는 메서드 실행이 끝나면 스택 메모리에서 사라지기 때문에 익명 객체에서 사용할 수 없게 되므로 문제가 발생한다. 해결 방법은 익명 객체 내부에서 메서드의 매개 변수나 로컬 변수를 사용할 경우, 이 변수들은 final 특성을 가져야 한다. 아래 예제는 매개 변수와 로컬 변수가 익명 객체에서 사용할 때 final 특성을 갖고 있음을 보여준다. 자바 7에서는 반드시 final 키워드를 붙여야 되지만, 자바 8부터는 붙이지 않아도 final 특성을 가지고 있다.

[Calculable.java] 인터페이스

```
public interface Calculable {
    public int sum();
}
```

[Anonymous.java] 익명 객체의 로컬 변수 사용

```
public class Anonymous {
    private int field;

    public void method(final int arg1, int arg2) {
        final int var1 = 0;
        int var2 = 0;
        field = 10;

        //arg1 = 20; (X)
        //arg2 = 20; (X)
        //var1 = 30; (X)
        //var2 = 30; (X)

        Calculable calc = new Calculable() {
            @Override
            public int sum() {
                int result = field + arg1 + arg2 + var1 + var2;
                return result;
            }
        };
        System.out.println(calc.sum());
    }
}
```

[Sample.java]

```
public class Sample {
    public static void main(String[] args) {
        Anonymous anony = new Anonymous();
        anony.method(0, 0);
    }
}
```

[실행결과]

10

10장 예외처리

10.1 예외와 예외 클래스

컴퓨터 하드웨어의 오동작 또는 고장으로 인해 응용프로그램 실행 오류가 발생하는 것을 자바에서는 에러(error)라고 한다. 자바에서는 에러 이외에 예외(Exception)라고 부르는 오류가 있다. 예외란 사용자의 잘못된 조작 또는 개발자의 코딩으로 인해 발생하는 프로그램 오류를 말한다. 예외가 발생되면 프로그램은 곧바로 종료된다는 점에서는 에러가 동일하다. 그러나 예외는 예외처리(Exception Handling)를 통해 프로그램을 종료하지 않고 정상 실행 상태가 유지되도록 할 수 있다. 자바에서는 예외를 클래스로 관리한다. JVM은 프로그램을 실행하는 도중에 예외가 발생하면 해당 예외 클래스로 객체를 생성한 후 예외 처리 코드에서 예외 객체를 이용할 수 있도록 해준다.

10.2 실행 예외

실행 예외는 자바 컴파일러가 체크를 하지 않기 때문에 오로지 개발자의 경험에 의해서 예외 처리 코드를 삽입해야 한다. 만약 개발자가 실행 예외에 대해 예외 처리 코드를 넣지 않았을 경우, 해당 예외가 발생하면 프로그램은 곧바로 종료된다.

10.2.1 NullPointerException

객체 상태가 없는 상태, 즉 null 값을 갖는 참조 변수로 객체 접근 연산자인 도트(.)를 사용했을 때 아래 에러가 발생한다. 객체가 없는 상태에서 객체를 사용하려 했으니 예외가 발생하는 것이다.

[Sample.java]

```
public class Sample {
    public static void main(String[] args) {
        String data = null;
        System.out.println(data.toString());
    }
}
```

10.2.2 ArrayIndexOutOfBoundsException

배열에서 인덱스 범위를 초과하여 사용할 경우 아래 오류가 발생한다. 예를 들어 길이가 3인 int[] arr = new int[3] 배열을 선언했다면 배열 항목을 지정하기 위해 arr[0]~arr[2]를 사용할 수 있다. 하지만 arr[3]을 사용하면 인덱스 범위를 초과했기 때문에 아래 오류가 발생한다.

[Sample.java]

```
public class Sample {
    public static void main(String[] args) {
        String data1 = args[0];
        String data2 = args[1];

        System.out.println("args[0]:" + data1);
        System.out.println("args[1]:" + data1);
    }
}
```

이클립스의 메뉴에서 [Run]-[Run Configuration...]를 선택한 후, [Arguments] 탭의 [Program arguments] 입력란에 두 개의 매개 값을 입력하고 실행하면 예외가 발생하지 않는다. 아래 예제와 같이 배열값을 읽기 전에 배열의 길이를 먼저 조사하면 실행 매개값이 없거나 부족할 경우 조건문을 이용해서 사용자에게 실행 방법을 알려준다.

[Sample.java]

```
public class Sample {
    public static void main(String[] args) {
        if(args.length == 2) {
            String data1 = args[0];
            String data2 = args[1];
            System.out.println("args[0]:" + data1);
            System.out.println("args[1]:" + data2);
        } else {
            System.out.println("[실행 방법]");
            System.out.println("java ArrayIndexOutOfBoundsExceptionExample");
            System.out.println("값1 값2");
        }
    }
}
```

10.2.3 NumberFormatException

프로그램을 개발하다 보면 문자열로 되어 있는 데이터를 숫자로 변경하는 경우가 자주 발생한다. 문자열을 숫자로 변환하는 방법은 여러 가지가 있지만 가장 많이 사용되는 코드는 아래와 같다.

변환 타입	메서드명(매개 변수)	설명
int	Integer.parseInt(String s)	주어진 문자열을 정수로 변환해서 리턴
double	Double.parseDouble(String s)	주어진 문자열을 실수로 변환해서 리턴

[Sample.java]

```
public class Sample {
    public static void main(String[] args) {
        String data1 = "100";
        String data2 = "a100";
        int value1 = Integer.parseInt(data1);

        //NumberFormatException 발생
        int value2 = Integer.parseInt(data2);
        int result = value1 + value2;
        System.out.println(data1 + "+" + data2 + "=" + result);
    }
}
```

[실행결과]

```
Problems Javadoc Declaration Console
<terminated> Sample (4) [Java Application] C:\Program Files (x86)\Java\jre1.8.0_121\bin\javaw.exe (2017. 6. 9. 오후 1:00:00)
Exception in thread "main" java.lang.NumberFormatException: For input string: "a100"
    at java.lang.NumberFormatException.forInputString(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at 실습02.Sample.main(Sample.java:9)
```

10.2.4 ClassCastException

타입 변환(Casting)은 상위 클래스와 하위 클래스 간에 발생하고 구현 클래스와 인터페이스 간에도 발생한다. 이러한 관계가 아니라면 클래스는 다른 클래스로 타입 변환할 수 없다. 강제로 타입 변환할 경우 아래 오류가 발생한다.

[Sample.java]

```
public class Sample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        changeDog(dog);
        Cat cat = new Cat();
        changeDog(cat);
    }

    public static void changeDog(Animal animal) {
        //if(animal instanceof Dog) {
        //ClassCastException 발생 가능
        Dog dog = (Dog) animal;
        //}
    }
}

class Animal {}
class Dog extends Animal {}
class Cat extends Animal {}
```

[실행결과]

```
Problems Javadoc Declaration Console
<terminated> Sample (4) [Java Application] C:\Program Files (x86)\Java\jre1.8.0_121\bin\javaw.exe (2017. 6. 9. 오후 1:00:00)
Exception in thread "main" java.lang.ClassCastException: 실습02.Cat cannot be cast to 실습02.Dog
    at 실습02.Sample.changeDog(Sample.java:14)
    at 실습02.Sample.main(Sample.java:9)
```

10.3 예외 처리 코드

프로그램에서 예외가 발생했을 경우 프로그램의 종료를 막고, 정상 실행을 유지할 수 있도록 처리하는 코드를 예외 처리 코드라고 한다. 예외 처리 코드는 try-catch-finally 블록을 이용한다. try 블록에는 예외 발생 가능 코드가 위치한다. try 블록의 코드가 예외 발생 없이 정상 실행되면 finally 블록의 코드를 실행한다. try 블록의 코드에서 예외가 발생하면 실행을 멈추고 catch 블록으로 이동하여 예외 처리 코드를 실행한다.

[Sample.java] 일반 예외 처리

```
public class Sample {
    public static void main(String[] args) {
        try {
            Class clazz = Class.forName("java.lang.String2");
        } catch (ClassNotFoundException e) {
            System.out.println("클래스가 존재하지 않습니다.");
        }
    }
}
```

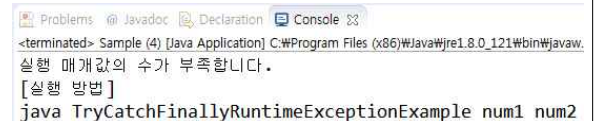
[실행결과]

클래스가 존재하지 않습니다.

[Sample.java] 실행 예외 처리

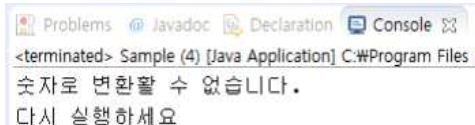
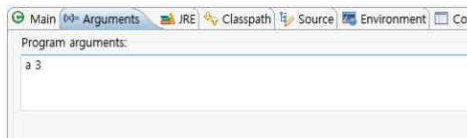
```
public class Sample {
    public static void main(String[] args) {
        String data1 = null;
        String data2 = null;
        try {
            data1 = args[0];
            data2 = args[1];
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("실행 매개값의 수가 부족합니다.");
            System.out.println("[실행 방법]");
            System.out.println("java TryCatchFinallyRuntimeExceptionExample num1 num2");
            return;
        }
        try {
            int value1 = Integer.parseInt(data1);
            int value2 = Integer.parseInt(data2);
            int result = value1 + value2;
            System.out.println(data1 + "+" + data2 + "=" + result);
        } catch (NumberFormatException e) {
            System.out.println("숫자로 변환할 수 없습니다.");
        } finally {
            System.out.println("다시 실행하세요");
        }
    }
}
```

[실행결과]



이클립스 메뉴에서 [Run]-[Run Configurations ...]를 선택한 후 첫 번째 실행 매개값을 숫자가 아닌 문자를 주고 실행한다. 문자 매개값은 숫자로 변환할 수 없기 때문에 예외가 발생하므로 예외 처리를 해준다.

[실행결과]



10.4 예외 종류에 따른 처리 코드

10.4.1 다중 catch

try 블록 내부는 다양한 예외가 발생할 수 있으므로 다중 catch 블록이 가능하다. catch 블록이 여러 개라 할지라도 단 하나의 catch 블록만 실행된다. 그 이유는 try 블록에서 동시에 예외가 발생하지 않고, 하나의 예외가 발생하면 실행을 멈추고 해당 catch 블록으로 이동하기 때문이다.

[Sample.java] 다중 Catch

```
public class Sample {
    public static void main(String[] args) {
        String data1 = null;
        String data2 = null;
        try {
            data1 = args[0];
            data2 = args[1];
            int value1 = Integer.parseInt(data1);
            int value2 = Integer.parseInt(data2);
            int result = value1 + value2;
            System.out.println(data1 + "+" + data2 + "=" + result);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("실행 매개값의 수가 부족합니다.");
            System.out.println("[실행 방법]");
            System.out.println("java TryCatchFinallyRuntimeExceptionExample num1 num2");
        } catch (NumberFormatException e) {
            System.out.println("숫자로 변환할 수 없습니다.");
        } finally {
            System.out.println("다시 실행하세요");
        }
    }
}
```


10.4.2 catch 순서

다중 catch 블록을 작성할 때 주의할 점은 상위 예외 클래스가 하위 예외 클래스보다 아래쪽에 위치해야 한다. 만약 상위 예외 클래스의 catch 블록이 위에 있다면, 하위 예외 클래스의 catch 블록은 실행되지 않는다. 아래 예제는 try 블록에서 ArrayIndexOutOfBoundsException이 발생하면 첫 번째 catch 블록을 실행하고, 그 밖의 다른 예외가 발생하면 두 번째 catch 블록을 실행한다.

[Sample.java] 다중 Catch

```
public class Sample {
    public static void main(String[] args) {
        String data1 = null;
        String data2 = null;
        try{
            data1 = args[0];
            data2 = args[1];
            int value1 = Integer.parseInt(data1);
            int value2 = Integer.parseInt(data2);
            int result = value1 + value2;
            System.out.println(data1 + "+" + data2 + "=" + result);
        }catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("실행 매개값의 수가 부족합니다.");
        }catch(Exception e) {
            System.out.println("숫자로 변환할 수 없습니다.");
        }finally {
            System.out.println("다시 실행하세요");
        }
    }
}
```

[실행결과]

실행 매개값의 수가 부족합니다.
다시 실행하세요

10.4.3 멀티 catch

자바 7부터 하나의 catch 블록에서 여러 개의 예외를 처리할 수 있도록 멀티(multi) catch 기능을 추가했다. catch 괄호()안에 동일하게 처리하고 싶은 예외를 |로 연결하면 된다.

[Sample.java] Multi Catch

```
public class Sample {
    public static void main(String[] args) {
        String data1 = null, data2 = null;
        try{
            data1 = args[0], data2 = args[1];
            int value1 = Integer.parseInt(data1);
            int value2 = Integer.parseInt(data2);
            int result = value1 + value2;
            System.out.println(data1 + "+" + data2 + "=" + result);
        }catch(ArrayIndexOutOfBoundsException | NumberFormatException e) {
            System.out.println("실행 매개값의 수가 부족하거나 숫자로 변환할 수 없습니다.");
        }catch(Exception e) {
            System.out.println("숫자로 변환 할 수 없습니다.");
        }finally {
            System.out.println("다시 실행 하세요");
        }
    }
}
```

[실행결과]

실행 매개값의 수가 부족하거나 숫자로 변환할 수 없습니다.
다시 실행하세요

10.5 자동 리소스 닫기

자바 7에서 새로 추가된 try-with-resource를 사용하면 예외 발생 여부와 상관없이 사용했던 리소스 객체의 close() 메서드를 호출해서 안정하게 리소스를 닫아준다. 여기서 리소스란 데이터를 읽고 쓰는 객체라고 생각해 두자. 예를 들어 파일의 데이터를 읽는 FileInputStream 객체와 파일에 쓰는 FileOutputStream를 리소스 객체라고 보면 된다. 자바6 이전까지는 파일을 닫으려면 finally 블록에서 다시 try-catch를 사용해서 close() 메서드로 예외 처리해야 하므로 다소 복잡했다. 자바 7에서 추가된 try-with-resources를 사용하면 다음과 같이 간단해진다. 어디를 봐도 close()를 명시적으로 호출한 곳이 없다.

```
try( FileInputStream fis = new FileInputStream("file.txt") ) {
    ...
}catch(IOException e) {
    ...
}
```

try 블록이 정상적으로 실행을 완료했거나 도중에 예외가 발생하게 되면 자동으로 FileOutputStream의 close() 메서드가 호출된다. try{ }에서 예외가 발생하면 우선 close()로 리소스를 닫고 catch 블록을 실행한다. 복수 개의 리소스를 사용해야 한다면 다음과 같이 작성할 수 있다.

```
try( FileInputStream fis = new FileInputStream("file1.txt")
    FileOutputStream fos = new FileOutputStream("file2.txt")
) {
    ...
} catch(IOException e) {
    ...
}
```

try-with-resources를 사용하기 위해서는 조건이 있는데, 리소스 객체는 java.lang.AutoCloseable 인터페이스를 구현하고 있어야 한다. AutoCloseable에는 close() 메서드가 정의되어 있는데 try-with-resources는 바로 이 close() 메서드를 자동 호출한다. API 문서에서 AutoCloseable 인터페이스를 찾아 "All Known Implementing Classes:"를 보면 try-with-resources와 함께 사용할 수 있는 리소스가 어떤 것이 있는지 알 수 있다.

다음 예제는 직접 AutoCloseable을 구현해서 FileInputStream 클래스를 작성했다. Smain() 메서드에서 try-with-resources를 사용하면 예외가 발생하는 즉시 자동으로 FileInputStream의 close()가 호출되는 것을 볼 수 있다.

[FileInputStream.java] AutoCloseable 구현 클래스

```
public class FileInputStream implements AutoCloseable {
    private String file;

    public FileInputStream(String file) {
        this.file = file;
    }

    public void read() {
        System.out.println(file + "을 읽습니다.");
    }

    @Override
    public void close() throws Exception {
        System.out.println(file + "을 닫습니다.");
    }
}
```

[Sample.java] AutoCloseable 구현 클래스

```
public class Sample {
    public static void main(String[] args) {
        try(FileInputStream fis = new FileInputStream("file.txt")) {
            fis.read();
            throw new Exception(); //강제적으로 예외 발생시킴
        } catch(Exception e) {
            System.out.println("예외 처리 코드가 실행되었습니다.");
        }
    }
}
```

[실행결과]

file.txt를 읽습니다.
file.txt를 닫습니다.
예외 처리 코드가 실행되었습니다.

10.6 예외 떠넘기기

메서드 내부에서 try-catch 블록으로 예외를 처리하는 것이 기본이지만, 경우에 따라서는 메서드를 호출한 곳으로 예외를 떠넘길 수도 있다.

[Sample.java] 예외 처리 떠넘기기

```
public class Sample {
    public static void main(String[] args) {
        try{
            findClass();
        } catch(ClassNotFoundException e) {
            System.out.println("클래스가 존재하지 않습니다.");
        }
    }

    public static void findClass() throws ClassNotFoundException {
        Class clazz = Class.forName("java.lang.String2");
    }
}
```

[실행결과]

클래스가 존재하지 않습니다.

10.7 사용자 정의 예외와 예외 발생

프로그램을 개발하다 보면 자바 표준 API에서 제공하는 예외 클래스만으로는 다양한 종류의 예외를 표현할 수가 없다. 그러므로 개발자가 직접 정의해서 만들어야 하는데 이런 예외를 사용자 정의 예외라고도 한다.

10.7.1 사용자 정의 예외 클래스 선언

사용자 정의 예외 클래스는 컴파일러가 체크하는 일반 예외로 선언할 수도 있고, 컴파일러가 체크하지 않는 실행 예외로 선언할 수도 있다. 일반 예외로 선언할 경우 Exception을 상속하면 되고, 실행 예외로 선언할 경우에는 RuntimeException을 상속하면 된다.

[BalanceInsufficientException.java] 사용자 정의 예외 클래스

```
public class BalanceInsufficientException extends Exception {
    public BalanceInsufficientException() {}
    public BalanceInsufficientException(String message) {
        super(message);
    }
}
```

10.7.2 예외 발생시키기

예외 객체를 생성할 때는 기본 생성자 또는 예외 메시지를 갖는 생성자 중 어떤 것을 사용해도 된다. 만약 catch 블록에서 예외 메시지가 필요하다면 예외 메시지를 갖는 생성자를 이용해야 한다. 예외 발생 코드를 가지고 있는 메서드는 내부에서 try-catch 블록으로 예외를 처리할 수 있지만, 대부분은 자신을 호출한 곳에서 예외를 처리하도록 throws 키워드로 예외를 떠넘긴다.

[Account.java] 사용자 정의 예외 발생시키기

```
public class Account {
    private long balance;
    public Account() {}

    public long getBalance() {
        return balance;
    }
    public void deposit(int money) {
        balance += money;
    }

    public void withdraw(int money) throws BalanceInsufficientException { //사용자 정의 예외 떠넘기기
        if(balance < money) {
            throw new BalanceInsufficientException("잔고부족:" + (money - balance) + " 모자람"); //사용자 정의 예외 발생
        }
        balance -= money;
    }
}
```

10.8 예외 정보 얻기

try 블록에서 예외가 발생되면 예외 객체는 catch 블록의 매개 변수에서 참조하게 되므로 매개 변수를 이용하면 예외 객체의 정보를 알 수 있다. 모든 예외 객체는 Exception 클래스를 상속하기 때문에 Exception의 모든 예외 객체에서 호출할 수 있다. 그중에서도 가장 많이 사용되는 메서드는 getMessage()와 printStackTrace()이다.

[Sample.java] 사용자 정의 예외 발생시키기

```
public class Sample {
    public static void main(String[] args) {
        Account account = new Account();

        //예금하기
        account.deposit(10000);
        System.out.println("예금액: " + account.getBalance());

        //출금하기
        try {
            account.withdraw(30000);
        } catch (BalanceInsufficientException e) { //예외 메시지 얻기
            String message = e.getMessage();
            System.out.println(message);
            System.out.println(); //예외 추적 후 출력
            e.printStackTrace();
        }
    }
}
```

[실행결과]

예금액: 10000
잔고부족: 20000 모자람

실습03.BalanceInsufficientException: 잔고부족: 20000 모자람
at 실습03.Account.withdraw(Account.java:20)
at 실습03.Sample.main(Sample.java:13)

11장 기본 API 클래스

11.1 자바 API 도큐먼트

API(Application Programming Interface)는 라이브러리(library)라고 부르기도 하는데, 프로그램 개발에 자주 사용되는 클래스 및 인터페이스의 모음이다. 이 API들은 <JDK설치경로>\jre\lib\rt.jar라는 압축 파일에 저장되어 있다. API 도큐먼트는 쉽게 API를 찾아 이용할 수 있도록 문서화한 것을 말한다. 웹 브라우저를 열고 오라클에서 제공하는 다음 URL을 방문하면 볼 수 있다.

<http://docs.oracle.com/javase/8/docs/api/>

11.2 java.lang과 java.util 패키지

11.2.1 java.lang 패키지

java.lang 패키지는 자바 프로그램의 기본적인 클래스(Object, System, Class, String, Wrapper, Math ...)를 담고 있는 패키지이다. 그러므로 java.lang 패키지에 있는 클래스와 인터페이스는 import 없이 사용할 수 있다.

11.2.2 java.util 패키지

java.util 패키지는 자바 프로그램 개발에 조미료 같은 역할을 하는 클래스(Arrays, Calendar, Date, Object, Random ...)를 담고 있다.

11.3 Object 클래스

11.3.1 객체 비교(equals())

Object 클래스의 equals() 메서드는 두 객체를 비교해서 논리적으로 동등하면 true를 리턴하고, 그렇지 않으면 false를 리턴 한다. 논리적으로 동등하다는 것은 같은 객체인건 다른 객체인건 상관없이 객체가 저장하고 있는 데이터가 동일함을 뜻한다.

[Member.java] 객체 동등 비교 (equals() 메서드)

```
public class Member {
    public String id;

    public Member(String id) {
        this.id = id;
    }

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof Member) { //매개값이 Member 타입인지 확인
            Member member = (Member) obj;
            if(id.equals(member.id)) { //Member 타입으로 강제 타입 변환하고 id 필드값이 동일한지 검사한 후, 동일하다면 true를 리턴
                return true;
            }
        }
        return false; //매개값이 Member 타입이 아니거나 id 필드값이 다른 경우 false를 리턴
    }
}
```

[Sample.java] 객체 동등 비교 (equals() 메서드)

```
public class Sample {
    public static void main(String[] args) {
        Member obj1 = new Member("blue");
        Member obj2 = new Member("blue");
        Member obj3 = new Member("red");
        if(obj1.equals(obj2)) { //매개값이 Member 타입이고 id 필드값도 동일하므로 true
            System.out.println("obj1과 obj2는 동등합니다.");
        } else {
            System.out.println("obj1과 obj2는 동등하지 않습니다.");
        }
        if(obj1.equals(obj3)) { //매개값이 Member 타입이지만 id 필드값이 다르므로 false
            System.out.println("obj1과 obj3은 동등합니다.");
        } else {
            System.out.println("obj1과 obj3은 동등하지 않습니다.");
        }
    }
}
```

[실행결과]

obj1과 obj2는 동등합니다.
obj1과 obj3은 동등하지 않습니다.

11.3.2 객체 문자 정보(toString())

Object 클래스의 toString() 메서드는 객체의 문자 정보를 리턴 한다. 객체의 문자 정보란 객체를 문자열로 표현한 값을 말한다. Object 하위 클래스는 toString() 메서드를 재정의(오버라이딩)하여 간결하고 유익한 정보를 리턴하도록 되어 있다. 예를 들어 java.util 패키지의 Date 클래스는 toString() 메서드를 재정의하여 현재 시스템의 날짜와 시간 정보를 리턴 한다.

[Sample.java] 객체의 문자 정보(toString() 메서드)

```
import java.util.Date;
```

```
public class Sample {  
    public static void main(String[] args) {  
        Object obj1 = new Object();  
        Date obj2 = new Date();  
        System.out.println(obj1.toString());  
        System.out.println(obj2.toString());  
    }  
}
```

[실행결과]

```
java.lang.Object@1db9742  
Tue Jun 13 13:09:44 KST 2017
```

[SmartPhone.java] 객체의 문자 정보(toString() 메서드)

```
public class SmartPhone {  
    private String company;  
    private String os;  
  
    public SmartPhone(String company, String os) {  
        this.company = company;  
        this.os = os;  
    }  
  
    @Override  
    public String toString() {  
        return company + "," + os;  
    }  
}
```

//toString() 재정의

[Sample.java] 객체의 문자정보(toString() 메서드)

```
public class Sample {  
    public static void main(String[] args) {  
        SmartPhone myPhone = new SmartPhone("구글", "안드로이드");  
        String strObj = myPhone.toString();  
        System.out.println(strObj);  
        System.out.println(myPhone); //myPhone.toString()을 자동 호출해서 리턴값을 얻은 후 출력  
    }  
}
```

[실행결과]

```
구글,안드로이드  
구글,안드로이드
```

11.4 String 클래스

어떤 프로그램이건 문자열은 데이터로서 아주 많이 사용된다. 그렇기 때문에 문자열을 생성하는 방법과 추출, 비교, 찾기, 변환 등을 제공하는 메서드를 잘 익혀두어야 한다.

11.4.1 String 생성자

자바의 문자열은 java.lang 패키지의 String 클래스의 인스턴스로 관리된다. 파일의 내용을 읽거나 네트워크를 통해 받은 데이터는 byte[] 배열이므로 이것을 문자열로 변환하기 위해 사용된다. 아래 예제는 바이트 배열을 문자열로 변환하는 예제이다.

[Sample.java] 바이트 배열을 문자열로 변환

```
public class Sample {  
    public static void main(String[] args) {  
        byte[] bytes={72, 101, 108, 108, 111, 32, 74, 97, 118, 97};  
        String str1 = new String(bytes);  
        System.out.println(str1);  
  
        String str2 = new String(bytes, 6, 4); //74번 위치 부터 4개  
        System.out.println(str2);  
    }  
}
```

[실행결과]

```
Hello Java  
Java
```

11.4.2 String 메서드

- 문자 추출: `charAt()` 메서드는 매개값으로 주어진 인덱스의 문자를 리턴 한다. 인덱스란 0에서부터 "문자열길이-1"까지의 번호를 말한다.

[Sample.java] 주민등록번호에서 남자와 여자를 구분하는 방법

```
public class Sample {
    public static void main(String[] args) {
        String ssn = "010624-1230123";

        char sex = ssn.charAt(7);
        switch(sex) {
            case '1':
            case '3':
                System.out.println("남자 입니다.");
                break;
            case '2':
            case '4':
                System.out.println("여자 입니다.");
                break;
        }
    }
}
```

[실행결과]
남자 입니다.

- 문자열 비교: 원래 `equals()`는 `Object`의 번지 비교 메서드이지만, `String` 클래스가 오버라이딩해서 문자열을 비교하도록 변경했다. 아래 예제는 `strVar1`과 `strVar2`의 `==` 연산은 `false`를 산출하고 `equals()` 메서드는 `true`가 산출된다.

[Sample.java] 문자열 비교

```
public class Sample {
    public static void main(String[] args) {
        String strVar1 = new String("신민철");
        String strVar2 = "신민철";

        if(strVar1 == strVar2) {
            System.out.println("같은 String 객체를 참조");
        } else {
            System.out.println("다른 String 객체를 참조");
        }
        if(strVar1.equals(strVar2)) {
            System.out.println("같은 문자열을 가짐");
        } else {
            System.out.println("다른 문자열을 가짐");
        }
    }
}
```

[실행결과]
다른 String 객체를 참조
같은 문자열을 가짐

- 바이트 배열로 변환: 종종 문자열을 바이트 배열로 변환하는 경우가 있다. 대표적인 예로 네트워크로 문자열을 전송하거나, 문자열을 암호화할 때 문자열을 바이트 배열로 변환한다. 어떤 문자셋으로 인코딩하느냐에 따라 바이트 배열의 크기가 달라지는데, EUC-KR은 `getBytes()`와 마찬가지로 알파벳 1바이트, 한글은 2바이트로 변환하고, UTF-8은 알파벳은 1바이트, 한글은 3바이트로 변환한다.

[Sample.java] 바이트 배열로 변환

```
public class Sample {
    public static void main(String[] args) {
        String str = "안녕하세요";
        byte[] bytes1 = str.getBytes();
        System.out.println("bytes1.length: " + bytes1.length);

        String str1 = new String(bytes1);
        System.out.println("bytes1->String: " + str1);

        try {
            byte[] bytes2 = str.getBytes("EUC-KR");
            System.out.println("bytes2.length: " + bytes2.length);
            String str2 = new String(bytes2);
            System.out.println("bytes2->String: " + str2);
            byte[] bytes3 = str.getBytes("UTF-8");
            System.out.println("bytes3.length: " + bytes3.length);
            String str3 = new String(bytes3, "UTF-8");
            System.out.println("bytes3->String: " + str3);
        } catch (Exception e) {
            System.out.println(e.toString());
        }
    }
}
```

[실행결과]
bytes1.length: 10
bytes1->String: 안녕하세요
bytes2.length: 10
bytes2->String: 안녕하세요
bytes3.length: 15
bytes3->String: 안녕하세요

- 문자열 찾기: `indexOf()` 메서드는 매개값으로 주어진 문자열이 시작되는 인덱스를 리턴 한다. 만약 주어진 문자열이 포함되어 있지 않으면 `-1`을 리턴 한다. "자바 프로그래밍"에서 "프로그래밍" 문자열의 인덱스 위치가 3이 리턴 된다. `indexOf()` 메서드는 `if`문의 조건식에서 특정 문자열이 포함되어 있는지 여부에 따라 실행 코드를 달리할 때 자주 사용된다.

[Sample.java] 문자열 포함 여부 조사

```
public class Sample {
    public static void main(String[] args) {
        String subject = "자바 프로그래밍";
        int location = subject.indexOf("프로그래밍");
        System.out.println(location);

        if(subject.indexOf("자바") != -1) {
            System.out.println("자바와 관련된 책이군요");
        } else {
            System.out.println("자바와 관련없는 책이군요");
        }
    }
}
```

[실행결과]

```
3
자바와 관련된 책이군요
```

- 문자열 길이: `length()` 메서드는 문자열의 길이(문자의 수)를 리턴 한다. "자바 프로그래밍"에서 문자열의 길이가 8이므로 `length()` 메서드 값으로 8이 리턴된다.

[Sample.java] 문자열 추출하기

```
public class Sample {
    public static void main(String[] args) {
        String ssn = "7306241230123";
        int length = ssn.length();

        if(length == 13) {
            System.out.println("주민번호 자리수가 맞습니다.");
        } else {
            System.out.println("주민번호 자리수가 틀립니다.");
        }
    }
}
```

[실행결과]

```
주민번호 자리수가 맞습니다.
```

- 문자열 대체: `replace()` 메서드는 첫 번째 매개값인 문자열을 찾아 두 번째 매개값인 문자열로 대체한 새로운 문자열을 생성하고 리턴 한다. `String` 객체의 문자열은 변경이 불가능한 특성을 갖기 때문에 `replace()` 메서드가 리턴하는 문자열은 원래 문자열의 수정본이 아니라 완전히 새로운 문자열이다.

[Sample.java] 문자열 대체하기

```
public class Sample {
    public static void main(String[] args) {
        String oldStr = "자바는 객체지향언어 입니다. 자바는 풍부한 API를 지원합니다.";
        System.out.println(oldStr);
        String newStr = oldStr.replace("자바", "JAVA");
        System.out.println(newStr);
    }
}
```

[실행결과]

```
자바는 객체지향언어 입니다. 자바는 풍부한 API를 지원합니다.
JAVA는 객체지향언어 입니다. JAVA는 풍부한 API를 지원합니다.
```

- 문자열 잘라내기: `substring()` 메서드는 주어진 인덱스에서 문자열을 추출한다. `ssn.substring(0,6)`은 인덱스 0(포함) ~ 6(제외) 사이의 문자열을 추출하는 것이고 `substring(7)`은 인덱스 7이후의 문자열을 추출한다.

[Sample.java] 문자열 추출하기

```
public class Sample {
    public static void main(String[] args) {
        String ssn = "880815-1234567";
        String firstNum = ssn.substring(0, 6);
        System.out.println(firstNum);

        String secondNum = ssn.substring(7);
        System.out.println(secondNum);
    }
}
```

[실행결과]

```
880815
1234567
```

- 알파벳 소,대문자 변경: toLowerCase()메서드는 문자열을 소문자로 바꾼 새로운 문자열을 생성한 후 리턴 한다. toUpperCase() 메서드는 문자열을 모두 대문자로 바꾼 새로운 문자열을 생성한 후 리턴 한다. equal()메서드를 사용하려면 toLowerCase()와 toUpperCase()로 대문자를 맞추어야 하지만, equalsIgnoreCase() 메서드를 사용하면 이 작업이 생략된다.

[Sample.java] 전부 소문자 또는 대문자로 변경

```
public class Sample {
    public static void main(String[] args) {
        String str1 = "Java Programming";
        String str2 = "JAVA Programming";
        System.out.println(str1.equals(str2));

        String lowerStr1 = str1.toLowerCase();
        String lowerStr2 = str2.toLowerCase();
        System.out.println(lowerStr1.equals(lowerStr2));
        System.out.println(str1.equalsIgnoreCase(str2));
    }
}
```

[실행결과]

```
false
true
true
```

- 문자열 앞뒤 공백 잘라내기: trim() 메서드는 문자열의 앞뒤 공백을 제거한 새로운 문자열을 생성하고 리턴 한다. trim() 메서드는 앞뒤의 공백만 제거할 뿐中间的 공백은 제거하지 않는다.

[Sample.java] 앞뒤 공백 제거

```
public class Sample {
    public static void main(String[] args) {
        String tel1 = " 02";
        String tel2 = "123 ";
        String tel3 = " 1234 ";
        String tel = tel1.trim() + tel2.trim() + tel3.trim();
        System.out.println(tel);
    }
}
```

[실행결과]

021231234

- 문자열 변환: valueOf() 메서드는 기본 타입의 값을 문자열로 변환하는 기능을 가지고 있다.

[Sample.java] 기본 타입 값을 문자열로 변환

```
public class Sample {
    public static void main(String[] args) {
        String str1 = String.valueOf(10);
        String str2 = String.valueOf(10.5);
        String str3 = String.valueOf(true);

        System.out.println(str1);
        System.out.println(str2);
        System.out.println(str3);
    }
}
```

[실행결과]

```
10
10.5
true
```

11.5 StringTokenizer 클래스

문자열이 특정 구분자(delimiter)로 연결되어 있을 경우, 구분자를 기준으로 부문 문자열을 분리하기 위해서 String의 split() 메서드를 이용하거나, java.util 패키지의 StringTokenizer 클래스를 이용할 수 있다.

11.5.1 split() 메서드

String 클래스의 split() 메서드는 정규 표현식을 구분자로 해서 문자열을 분리한 후, 배열에 저장하고 리턴 한다.

```
String[] result = "문자열".split("정규표현식");
```

11.5.2 StringTokenizer 클래스

문자열이 한 종류의 구분자로 연결되어 있을 경우, StringTokenizer 클래스를 사용하면 손쉽게 문자열(토큰:token)을 분리해 낼 수 있다. StringTokenizer 객체를 생성할 때 첫 번째 매개값으로 전체 문자열을 주고, 두 번째 매개값으로 구분자를 주면 된다.

```
StringTokenizer str = new StringTokenizer("문자열", "구분자");
```


nextToken() 메서드로 토큰을 하나 꺼내오면 StringTokenizer 객체에는 해당 토큰이 없어진다. 만약 StringTokenizer 객체에서 더 이상 가져올 토큰이 없다면 nextToken() 메서드는 java.util.NoSuchElementException 예외를 발생시킨다. 그래서 nextToken() 메서드를 사용하기 전에 hasMoreTokens() 메서드로 꺼내올 토큰이 있는지 조사한 후 nextToken() 메서드를 호출하는 것이 좋은 방법이다. 아래 예제는 두 가지 방법으로 토큰을 추출하는 방법을 보여준다.

[Sample.java] StringTokenizer로 토큰 분리하기

```
public class Sample {
    public static void main(String[] args) {
        String text = "홍길동/이수홍/박연수";

        //방법1: 전체 토큰 수를 얻어 for문으로 루핑
        StringTokenizer st = new StringTokenizer(text, "/");
        int countTokens = st.countTokens();
        for(int i=0; i<countTokens; i++) {
            String token = st.nextToken();
            System.out.println(token);
        }
        System.out.println();

        //방법2: 남아 있는 토큰을 확인하고 while문으로 루핑
        st = new StringTokenizer(text, "/");
        while(st.hasMoreElements()) {
            String token = st.nextToken();
            System.out.println(token);
        }
    }
}
```

[실행결과]

홍길동
이수홍
박연수

홍길동
이수홍
박연수

11.6 StringBuffer, StringBuilder 클래스

문자열을 변경하는 작업이 많을 경우에는 String 클래스를 사용하는 것보다는 java.lang 패키지의 StringBuffer 또는 StringBuilder 클래스를 사용하는 것이 좋다. 이 두 클래스는 내부 버퍼(buffer: 데이터를 임시로 저장하는 메모리)에 문자열을 저장해 두고, 그 안에서 추가, 수정, 삭제 작업을 할 수 있도록 설계되어 있다. String처럼 새로운 객체를 만들지 않고도 문자열을 조작할 수 있는 것이다.

[Sample.java] StringBuilder에서 문자열 조작

```
public class Sample {
    public static void main(String[] args) {
        //StringBuffer 객체 생성
        StringBuilder sb = new StringBuilder();

        //문자열을 끝에 추가
        sb.append("Java ");
        sb.append("Program Study");
        System.out.println(sb.toString());

        //4번째 문자 뒤에 2를 삽입
        sb.insert(4, "2");
        System.out.println(sb.toString());

        //4번째 문자 뒤에 문자를 6으로 변경
        sb.setCharAt(4, '6');
        System.out.println(sb.toString());

        //6번째 문자부터 13번째 문자까지를 "Book" 문자열로 대체
        sb.replace(6, 13, "Book");
        System.out.println(sb.toString());

        //5번째 문자를 삭제
        sb.delete(4, 5);
        System.out.println(sb.toString());

        //총 문자 수 얻기
        int length = sb.length();
        System.out.println("총문자수:" + length);

        //버퍼에 있는 것을 String 타입으로 리턴
        String result = sb.toString();
        System.out.println(result);
    }
}
```

[실행결과]

Java Program Study
Java2 Program Study
Java6 Program Study
Java6 Book Study
Java Book Study
총문자수:15
Java Book Study

11.7 정규 표현식과 Pattern 클래스

문자열이 정해져 있는 형식으로 구성되어 있는지 검증해야 하는 경우가 있다. 예를 들어, 이메일, 전화번호를 사용자가 제대로 입력했는지 검증해야 할 때 정규표현식과 비교한다. 이번 절에서는 정규 표현식을 작성하는 방법과 문자열 검증하는 방법에 대해 살펴보기로 하자.

11.7.1 정규 표현식 작성 방법

정규 표현식 작성 방법은 API 문서에서 `java.util.regex.Pattern` 클래스를 찾아 Summary of regular-expression constructs를 참조한다. 간단히 말해서 정규 표현식은 문자 또는 숫자 기호와 반복 기호가 결합된 문자열이다.

11.7.2 Pattern 클래스

아래 예제는 정규 표현식을 이용해 전화번호와 이메일을 검증하는 코드를 보여준다. 이메일 검증에서 일치하지 않는 이유는 navercom이라고 되어 있기 때문이다. 반드시 점(.)이 포함되어야 하므로 naver.com이라고 해야 맞다.

[Sample.java] 문자열 검증하기

```
public class Sample {
    public static void main(String[] args) {
        String regExp = "(02|010)-\\d{3,4}-\\d{4}";
        String data = "010-123-4567";
        boolean result = Pattern.matches(regExp, data);
        if(result) {
            System.out.println("정규식과 일치합니다.");
        } else {
            System.out.println("정규식과 일치하지 않습니다.");
        }

        regExp = "\\w+@\\w+\\.\\w+(\\.\\w+)?";
        data = "angel@navercom";
        result = Pattern.matches(regExp, data);
        if(result) {
            System.out.println("정규식과 일치합니다.");
        } else {
            System.out.println("정규식과 일치하지 않습니다.");
        }
    }
}
```

[실행결과]

정규식과 일치합니다.
정규식과 일치하지 않습니다.

11.8 Arrays 클래스

Array 클래스는 배열 조작 기능을 가지고 있다. 배열 조작이란 배열의 복사, 항목 정렬, 항목 검색과 같은 기능을 말한다. 단순한 배열 복사는 `System.arraycopy()` 메서드를 사용할 수 있으나, Arrays는 추가적으로 항목 정렬, 항목 검색, 항목 비교와 같은 기능을 제공해 준다.

11.8.1 배열 복사

배열 복사를 위해 사용할 수 있는 메서드는 `copyOf(원본배열, 복사할길이)`, `copyOfRange(원본배열, 시작인덱스, 끝인덱스)`이다. `copyOf()` 메서드는 원본 배열의 0번 인덱스에서 복사할 길이만큼 복사한 타겟 배열을 리턴하는데, 복사할 길이는 원본 배열의 길이보다 커도 되며, 타겟 배열의 길이가 된다. 아래 예제는 Arrays와 `System.arraycopy()` 메서드를 이용해서 배열을 복사하고 있다.

[Sample.java] 배열 복사

```
public class Sample {
    public static void main(String[] args) {
        char[] arr1 = {'J', 'A', 'V', 'A'};

        //방법1
        char[] arr2 = Arrays.copyOf(arr1, arr1.length);
        System.out.println(Arrays.toString(arr2));

        //방법2
        char[] arr3 = Arrays.copyOfRange(arr1, 1, 3);
        System.out.println(Arrays.toString(arr3));

        //방법3
        char[] arr4 = new char[arr1.length];
        System.arraycopy(arr1, 0, arr4, 0, arr1.length);
        for(int i=0; i<arr4.length; i++) {
            System.out.println("arr4[" + i + "]=" + arr4[i]);
        }
    }
}
```

[실행결과]

[J, A, V, A]
[A, V]
arr4[0]=J
arr4[1]=A
arr4[2]=V
arr4[3]=A

11.8.2 배열 항목 비교

Arrays의 equals()와 deepEquals()는 배열 항목을 비교한다. Arrays.equals()는 1차 항목의 값만 비교하고, deepEquals()는 중첩된 배열의 항목까지 비교한다. 아래 예제는 배열 복사 후 항목을 비교한다.

[Sample.java] 배열 비교

```
public class Sample {
    public static void main(String[] args) {
        int[][] original = {{1, 2}, {3, 4}};

        //얕은 복사 후 비교
        System.out.println("[얕은 복사후 비교]");
        int[][] cloned1 = Arrays.copyOf(original, original.length);
        System.out.println("배열 번지 비교: " + original.equals(cloned1));
        System.out.println("1차 배열 항목값 비교: " + Arrays.equals(original, cloned1));
        System.out.println("중첩 배열 항목값 비교: " + Arrays.deepEquals(original, cloned1));

        //깊은 복사 후 비교
        System.out.println("\n[깊은 복제후 비교]");
        int[][] cloned2 = Arrays.copyOf(original, original.length);
        cloned2[0] = Arrays.copyOf(original[0], original[0].length);
        cloned2[1] = Arrays.copyOf(original[1], original[1].length);
        System.out.println("배열 번지 비교: " + original.equals(cloned2));
        System.out.println("1차 배열 항목값 비교: " + Arrays.equals(original, cloned2));
        System.out.println("중첩 배열 항목값 비교: " + Arrays.deepEquals(original, cloned2));
    }
}
```

[실행결과]

```
[얕은 복사후 비교]
배열 번지 비교: false
1차 배열 항목값 비교: true
중첩 배열 항목값 비교: true
```

```
[깊은 복제후 비교]
배열 번지 비교: false
1차 배열 항목값 비교: false
중첩 배열 항목값 비교: true
```

11.8.3 배열 항목 정렬

기본 타입 또는 String 배열은 Arrays.sort() 메서드의 매개값으로 지정해주면 자동으로 오름차순 정렬이 된다. 사용자 정의 클래스 타입일 경우에는 클래스가 Comparable 인터페이스를 구현하고 있어야 정렬이 된다.

[Member.java] Comparable 구현 클래스

```
public class Member implements Comparable<Member>{
    String name;
    Member(String name) {
        this.name = name;
    }
    @Override
    public int compareTo(Member o) {
        return name.compareTo(o.name);
    }
}
```

[Sample.java] 배열 비교

```
public class Sample {
    public static void main(String[] args) {
        int[] scores = {99, 97, 98};
        Arrays.sort(scores);
        for(int i=0; i<scores.length; i++) {
            System.out.println("scores[" + i + "]=" + scores[i]);
        }
        System.out.println();

        String[] names = {"홍길동", "박동수", "김민수"};
        Arrays.parallelSort(names);
        for(int i=0; i<names.length; i++) {
            System.out.println("names[" + i + "]=" + names[i]);
        }
        System.out.println();
        Member m1 = new Member("홍길동");
        Member m2 = new Member("박동수");
        Member m3 = new Member("김민수");
        Member[] members = {m1, m2, m3};
        Arrays.parallelSort(members);
        for(int i=0; i<members.length; i++) {
            System.out.println("members[" + i + "].name=" + members[i].name);
        }
    }
}
```

[실행결과]

```
scores[0]=97
scores[1]=98
scores[2]=99
```

```
names[0]=김민수
names[1]=박동수
names[2]=홍길동
```

```
members[0].name=김민수
members[1].name=박동수
members[2].name=홍길동
```

11.8.4 배열 항목 검색

배열 항목에서 특정 값이 위치한 인덱스를 얻는 것을 배열 검색이라고 한다. 배열 항목을 검색하려면 먼저 Arrays.sort(0 메서드로 항목들을 오름차순으로 정렬한 후, Arrays.binarySearch() 메서드로 항목을 찾아야 한다. 아래 예제는 배열 항목을 검색하는 방법을 보여준다.

[Sample.java] 배열 검색

```
public class Sample {
    public static void main(String[] args) {
        //기본 타입값 검색
        int[] scores = {99, 97, 98};
        Arrays.sort(scores);
        int index = Arrays.binarySearch(scores, 99);
        System.out.println("찾은 인덱스: " + index);

        //문자열 검색
        String[] names = {"홍길동", "박동수", "김민수"};
        Arrays.sort(names);
        index = Arrays.binarySearch(names, "홍길동");
        System.out.println("찾은 인덱스: " + index);

        //객체검색
        Member m1 = new Member("홍길동");
        Member m2 = new Member("박동수");
        Member m3 = new Member("김민수");
        Member[] members = {m1, m2, m3};
        Arrays.sort(members);
        index = Arrays.binarySearch(members, m1);
        System.out.println("찾은 인덱스: " + index);
    }
}
```

[실행결과]

```
찾은 인덱스: 2
찾은 인덱스: 2
찾은 인덱스: 2
```

11.9 Wrapper(포장) 클래스

자바는 기본 타입(byte, char, short, int, long, float, double, boolean)의 값을 갖는 객체를 생성할 수 있다. 이 객체를 포장(Wrapper) 객체라고 한다. 그 이유는 기본 타입의 값을 내부에 두고 포장하기 때문이다. 포장 객체의 특징은 포장하고 있는 기본 타입 값은 외부에서 변경할 수 없다. 만약 내부의 값을 변경하고 싶다면 새로운 포장 객체를 만들어야 한다.

포장 클래스는 java.lang 패키지에 포함되어 있는데, 기본 타입에 대응하는 클래스들이 있다. chr 타입과 int 타입이 각각 Character와 Integer로 변경되고, 기본 타입의 첫 문자를 대문자로 바꾼 이름을 가지고 있다.

11.9.1 박싱(Boxing)과 언박싱(Unboxing)

기본타입의 값을 포장객체로 만드는 과정을 박싱(Boxing)이라고 하고, 반대로 포장객체에서 기본타입의 값을 얻어내는 과정을 언박싱(Unboxing)이라고 한다. 박싱하는 방법은 간단하게 포장 클래스의 생성자 매개값으로 기본 타입의 값 또는 문자열을 넘겨주면 된다.

박싱된 포장 객체에서 다시 기본 타입의 값을 얻어 내기 위해서는(언박싱하기 위해서는) 각 포장 클래스마다 가지고 있는 "기본타입면+Value()" 메서드를 호출하면 된다. 아래 예제는 박싱, 언박싱 한 후 데이터값을 출력하는 코드이다.

[Sample.java] 기본 타입의 값을 박싱하고 언박싱하기

```
public class Sample {
    public static void main(String[] args) {
        //Boxing
        Integer obj1 = new Integer(100);
        Integer obj2 = new Integer("200");
        Integer obj3 = Integer.valueOf("300");

        //UnBoxing
        int value1 = obj1.intValue();
        int value2 = obj2.intValue();
        int value3 = obj3.intValue();

        System.out.println(value1);
        System.out.println(value2);
        System.out.println(value3);
    }
}
```

[실행결과]

```
100
200
300
```

11.9.2 자동 박싱과 언박싱

기본 타입 값을 직접 박싱, 언박싱하지 않아도 자동적으로 박싱과 언박싱이 일어나는 경우가 있다. 자동 박싱은 포장 클래스 타입에 기본값이 대입될 경우에 발생한다. 예를 들어 int 타입의 값을 Integer 클래스 변수에 대입하면 자동 박싱이 일어날 힙 영역에 Integer 객체가 생성된다.

자동 언박싱은 기본 타입에 포장 객체가 대입될 경우에 발생한다. 예를 들어 Integer 객체를 int 타입 변수에 대입하거나, Integer 객체와 int 타입값을 연산하면 Integer 객체로부터 int 타입의 값이 자동 언박싱되어 연산된다.

[Sample.java] 자동 박싱과 언박싱

```
public class Sample {
    public static void main(String[] args) {
        //자동 Boxing
        Integer obj = 100;
        System.out.println("value: " + obj.intValue());

        //대입 시 자동 Unboxing
        int value = obj;
        System.out.println("value: " + value);

        //연산 시 자동 Unboxing
        int result = obj + 100;
        System.out.println("result: " + result);
    }
}
```

[실행결과]
value: 100
value: 100
result: 200

11.9.3 문자열을 기본 타입 값으로 변환

포장 클래스의 용도는 기본 타입의 값을 박싱해서 포장 객체로 만드는 것이지만, 문자열을 기본 타입 값으로 변환할 때에도 많이 사용된다. 포장 클래스에는 "parse+기본타입" 명으로 되어 있는 정적(static) 메서드가 있다. 이 메서드는 문자열을 메개값으로 받아 기본 타입 값으로 변환한다.

[Sample.java] 문자열을 기본 타입 값으로 변환

```
public class Sample {
    public static void main(String[] args) {
        int value1 = Integer.parseInt("10");
        double value2 = Double.parseDouble("3.14");
        boolean value3 = Boolean.parseBoolean("true");

        System.out.println("value1: " + value1);
        System.out.println("value2: " + value2);
        System.out.println("value3: " + value3);
    }
}
```

[실행결과]
value1: 10
value2: 3.14
value3: true

11.9.4 포장 값 비교

포장 객체는 내부의 값을 비교하기 위해 ==와 !=연산자를 사용할 수 없다. 이 연산자는 내부의 값을 비교하는 것이 아니라 포장 객체의 참조를 비교하기 때문이다. 포장 객체에 정확히 어떤 값이 저장될지 모르는 상황이라면 직접 내부 값을 언박싱해서 비교하거나, equals() 메서드로 내부 값을 비교하는 것이 좋다. 포장 클래스의 equals() 메서드는 내부의 값을 비교하도록 오버라이딩되어 있다.

[Sample.java] 포장 객체 비교

```
public class Sample {
    public static void main(String[] args) {
        System.out.println("[-128 ~ 127 초과값일 경우]");
        Integer obj1 = 300;
        Integer obj2 = 300;
        System.out.println("==결과: " + (obj1 == obj2));
        System.out.println("언박싱후 ==결과: " + (obj1.intValue() == obj2.intValue()));
        System.out.println("equals() 결과: " + obj1.equals(obj2));
        System.out.println();

        System.out.println("[-128 ~ 127 범위값일 경우]");
        Integer obj3 = 10;
        Integer obj4 = 10;
        System.out.println("==결과: " + (obj3 == obj4));
        System.out.println("언박싱후 ==결과: " + (obj3.intValue() == obj4.intValue()));
        System.out.println("equals() 결과: " + obj3.equals(obj4));
    }
}
```

[실행결과]
[-128 ~ 127 초과값일 경우]
==결과: false
언박싱후 ==결과: true
equals() 결과: true

[-128 ~ 127 범위값일 경우]
==결과: true
언박싱후 ==결과: true
equals() 결과: true

11.10 Math, Random 클래스

11.10.1 Math 클래스

java.lang.Math 클래스는 수학 계산에 사용할 수 있는 메서드를 제공하고 있다. Math 클래스가 제공하는 메서드는 모두 정적(static)이므로 Math 클래스로 바로 사용이 가능하다. 아래 예제는 Math 클래스가 제공하는 메서드를 설명하는 코드이다.

[Sample.java] Math의 수학 메서드

```
public class Sample {
    public static void main(String[] args) {
        int v1 = Math.abs(-5);
        double v2 = Math.abs(-3.14); //절대값
        System.out.println("v1=" + v1);
        System.out.println("v2=" + v2);

        double v3 = Math.ceil(5.3);
        double v4 = Math.ceil(-5.3); //올림값
        System.out.println("v3=" + v3);
        System.out.println("v4=" + v4);

        double v5 = Math.floor(5.3);
        double v6 = Math.floor(-5.3); //버림값
        System.out.println("v5=" + v5);
        System.out.println("v6=" + v6);

        int v7 = Math.max(5, 9);
        double v8 = Math.max(5.3, 2.5); //최대값
        System.out.println("v7=" + v7);
        System.out.println("v8=" + v8);

        int v9 = Math.min(5, 9);
        double v10 = Math.min(5.3, 2.5); //최소값
        System.out.println("v9=" + v9);
        System.out.println("v10=" + v10);

        double v11 = Math.random(); //랜덤값
        System.out.println("v11=" + v11);

        double v12 = Math rint(5.3);
        double v13 = Math rint(5.7); //가까운 정수의 실수값
        System.out.println("v12=" + v12);
        System.out.println("v13=" + v13);

        double v14 = Math.round(5.3);
        double v15 = Math.round(5.7); //반올림값
        System.out.println("v14=" + v14);
        System.out.println("v15=" + v15);

        double value = 12.3456;
        double temp1 = value * 100;
        long temp2 = Math.round(temp1);
        double v16 = temp2 / 100.0;
        System.out.println("v16=" + v16);
    }
}
```

[실행결과]

```
v1=5
v2=3.14
v3=6.0
v4=-5.0
v5=5.0
v6=-6.0
v7=9
v8=5.3
v9=5
v10=2.5
v11=0.3790319596221654
v12=5.0
v13=6.0
v14=5.0
v15=6.0
v16=12.35
```

아래 예제는 한 번 던져서 나오는 주사위의 번호와 로또 번호를 Math.random() 메서드를 이용해서 얻는 방법을 보여준다.

[Sample.java] 임의의 주사위의 눈 얻기

```
public class Sample {
    public static void main(String[] args) {
        //주사위 번호 뽑기
        int num = (int)(Math.random() * 6) + 1;
        System.out.println("주사위 눈:" + num);

        //로또 번호 뽑기
        int lotto = (int)(Math.random() * 45) + 1;
        System.out.println("로또 번호: " + lotto);
    }
}
```

[실행결과]

```
주사위 눈: 6
로또 번호: 44
```


11.10.2 Random 클래스

java.util.Random 클래스는 난수를 얻어내기 위해 다양한 메서드를 제공한다. Math.random() 메서드는 0.0에서 1사이의 double 난수를 얻는 데만 사용한다면, Random 클래스는 boolean, int, long, float, double 난수를 얻을 수 있다. 또 다른 차이점은 Random 클래스는 종자값(seed)을 설정할 수 있다. 종자값은 난수를 만드는 알고리즘에 사용되는 값으로 종자값이 같으면 같은 난수를 얻는다.

[Sample.java] 로또 번호 얻기

```
public class Sample {
    public static void main(String[] args) {
        //선택 번호
        int[] selectNumber = new int[6]; //선택 번호 6개가 저장될 배열 생성
        Random random = new Random(3); //선택 번호를 얻기 위한 Random 객체 생성
        System.out.print("선택 번호: ");
        for(int i=0; i<6; i++) {
            selectNumber[i] = random.nextInt(45) + 1; //선택 번호를 얻어 배열에 저장
            System.out.print(selectNumber[i] + " ");
        }
        System.out.println();

        //당첨 번호
        int[] winningNumber = new int[6]; //당첨 번호 6개가 저장될 배열 생성
        random = new Random(5); //당첨 번호를 얻기 위한 Random 객체 생성
        System.out.print("당첨 번호: ");
        for(int i=0; i<6; i++) {
            winningNumber[i] = random.nextInt(45) + 1; //당첨 번호를 얻어 배열에 저장
            System.out.print(winningNumber[i] + " ");
        }
        System.out.println();

        //당첨 여부
        Arrays.sort(selectNumber);
        Arrays.sort(winningNumber);
        boolean result = Arrays.equals(selectNumber, winningNumber); //배열 항목 비교
        System.out.print("당첨 여부: ");
        if(result) {
            System.out.println("1등에 당첨되었습니다.");
        } else {
            System.out.println("당첨되지 않았습니다.");
        }
    }
}
```

[실행결과]

선택 번호: 15 21 16 17 34 28
당첨 번호: 18 38 45 15 22 36
당첨 여부: 당첨되지 않았습니다.

11.11 Date, Calendar 클래스

날짜는 프로그램에서 자주 사용되는 데이터이다. 자바는 시스템의 날짜 및 시각을 읽을 수 있도록 Date와 Calendar 클래스를 제공하고 있다. 이 두 클래스는 모두 java.util 패키지에 포함되어 있다.

11.11.1 Date 클래스

Date는 날짜를 표현하는 클래스이다. Date 클래스는 객체 간에 날짜 정보를 주고받을 때 주로 사용된다. Date 클래스에는 여러 개의 생성자가 선언되어 있지만 대부분 Deprecated(비권장)되어 현재는 Date() 생성자만 주로 사용한다. Date() 생성자는 컴퓨터의 현재 날짜를 읽어 Date 객체로 만든다. 현재 날짜를 문자열로 얻고 싶다면 toString() 메서드를 사용하면 된다. toString() 메서드는 영문으로 된 날짜를 리턴하는데 만약 특정 문자열 포맷으로 얻고 싶다면 java.text.SimpleDateFormat 클래스를 이용하면 된다.

[Sample.java] 현재 날짜를 출력하기

```
public class Sample {
    public static void main(String[] args) {
        Date now = new Date();
        String strNow1 = now.toString();
        System.out.println(strNow1);

        SimpleDateFormat sdf =
            new SimpleDateFormat("yyyy년 MM월 dd일 hh시 mms분 ss초");
        String strNow2 = sdf.format(now);
        System.out.println(strNow2);
    }
}
```

[실행결과]

Sat Mar 07 12:00:37 KST 2020
2020년 03월 07일 12시 00분 37초

11.11.2 Calendar 클래스

Calendar 클래스는 달력을 표현한 클래스이다. Calendar 클래스는 추상(abstract) 클래스이므로 new 연산자를 사용해서 인스턴스를 생성할 수 없다. Calendar 클래스의 정적 메서드인 getInstance() 메서드를 이용하면 Calendar 하위 객체를 얻을 수 있다.

[Sample.java] 운영체제의 시간대를 기준으로 Calendar 얻기

```
public class Sample {
    public static void main(String[] args) {
        Calendar now = Calendar.getInstance();

        int year = now.get(Calendar.YEAR);
        int month = now.get(Calendar.MONTH) + 1;
        int day = now.get(Calendar.DAY_OF_MONTH);

        int week = now.get(Calendar.DAY_OF_WEEK);
        String strWeek = null;
        switch(week) {
            case Calendar.MONDAY:
                strWeek = "월";
                break;
            case Calendar.TUESDAY:
                strWeek = "화";
                break;
            case Calendar.WEDNESDAY:
                strWeek = "수";
                break;
            case Calendar.THURSDAY:
                strWeek = "목";
                break;
            case Calendar.FRIDAY:
                strWeek = "토";
                break;
            case Calendar.SATURDAY:
                strWeek = "일";
                break;
            default:
                strWeek = "일";
        }

        int amPm = now.get(Calendar.AM_PM);
        String strAmPm = null;
        if(amPm == Calendar.AM) {
            strAmPm = "오전";
        } else {
            strAmPm = "오후";
        }

        int hour = now.get(Calendar.HOUR);
        int minute = now.get(Calendar.MINUTE);
        int second = now.get(Calendar.SECOND);

        System.out.println(year + "년 " + month + "월 " + day + "일 ");
        System.out.print(strWeek + "요일 ");
        System.out.println(strAmPm);
        System.out.println(hour + "시 " + minute + "분 " + second + "초 ");
    }
}
```

[실행결과]
2020년 3월 7일
일요일 오후
0시 34분 55초

다른 시간대에 해당하는 날짜와 시간을 출력하기 위해서는 Calendar 클래스의 오버로딩된 getInstance() 메서드를 이용해 다른 시간대의 Calendar를 얻을 수 있다. 알고 싶은 시간대의 TimeZone 객체를 얻어 getInstance() 메서드의 매개값으로 넘겨주면 된다.

[Sample.java] 미국 로스앤젤레스 시간 출력

```
public class Sample {
    public static void main(String[] args) {
        TimeZone timeZone = TimeZone.getTimeZone("America/Los_Angeles");
        Calendar now = Calendar.getInstance(timeZone);

        int hour = now.get(Calendar.HOUR);
        int minute = now.get(Calendar.MINUTE);
        int second = now.get(Calendar.SECOND);

        System.out.println(hour + "시 " + minute + "분 " + second + "초 ");
    }
}
```

[실행결과]
7시 49분 21초

11.12 Format 클래스

원하는 형태로 출력하기 위해서 자바에서는 형식 클래스를 제공한다. 형식 클래스는 java.text 패키지에 포함되어 있는데, 숫자 형식을 위해 DecimalFormat, 날짜 형식을 위해 SimpleDateFormat, 매개 변수화된 문자열 형식을 위해 MessageFormat 등을 제공한다.

11.12.1 숫자 형식 클래스(DecimalFormat)

DecimalFormat은 숫자 데이터를 원하는 형식으로 표현하기 위해 패턴을 사용하며 아래 예제는 다양한 패턴을 적용해서 얻은 문자열을 출력한다.

[Sample.java] 숫자를 원하는 형식으로 출력

```
public class Sample {
    public static void main(String[] args) {
        double num = 1234567.89;

        DecimalFormat df = new DecimalFormat("0");
        System.out.println("[0]: " + df.format(num));

        df = new DecimalFormat("0.0");
        System.out.println("[0.0]: " + df.format(num));

        df = new DecimalFormat("0000000000.00000");
        System.out.println("[0000000000.00000]: " + df.format(num));

        df = new DecimalFormat("#");
        System.out.println("[#]: " + df.format(num));

        df = new DecimalFormat("#####.##");
        System.out.println("[#####.##]: " + df.format(num));

        df = new DecimalFormat("#.0");
        System.out.println("[#.0]: " + df.format(num));

        df = new DecimalFormat("+#.0");
        System.out.println("[+#.0]: " + df.format(num));

        df = new DecimalFormat("-#.0");
        System.out.println("[-#.0]: " + df.format(num));

        df = new DecimalFormat("#,###.0");
        System.out.println("[#,###.0]: " + df.format(num));

        df = new DecimalFormat("0.0E0");
        System.out.println("[0.0E0]: " + df.format(num));

        df = new DecimalFormat("+#,###; -#,###");
        System.out.println("[+#,###; -#,###]: " + df.format(num));

        df = new DecimalFormat("#,##%");
        System.out.println("[#,##%]: " + df.format(num));

        df = new DecimalFormat("\u00A4 #,###"); //통화기호
        System.out.println("[\u00A4 #,###]: " + df.format(num));
    }
}
```

[실행결과]

```
[0]: 1234568
[0.0]: 1234567.9
[0000000000.00000]: 0001234567.89000
[#]:1234568
[#####.##]:1234567.89
[#.0]: 1234567.9
[+#.0]: +1234567.9
[-#.0]: -1234567.9
[#,###.0]: 1,234,567.9
[0.0E0]: 1.2E6
[+#,###; -#,###]: +1,234,568
[#,##%]: 123456789%
[\u00A4 #,###]: ₩ 1,234,568
```

11.12.2 날짜 형식 클래스(SimpleDateFormat)

Date 클래스의 toString() 메서드의 리턴값을 특정 문자열 포맷으로 얻고 싶다면 java.text.SimpleDateFormat 클래스를 이용하면 된다.

[Sample.java] 날짜를 원하는 형식으로 출력

```
public class Sample {
    public static void main(String[] args) {
        Date now = new Date();

        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        System.out.println("[yyyy-MM-dd]: " + sdf.format(now));

        sdf = new SimpleDateFormat("yyyy년 MM월 dd일");
        System.out.println("[yyyy년 MM월 dd일]: " + sdf.format(now));

        sdf = new SimpleDateFormat("yyyy.MM.dd a HH:mm:ss");
        System.out.println("[yyyy.MM.dd a HH:mm:ss]: " + sdf.format(now));

        sdf = new SimpleDateFormat("오늘은 E요일");
        System.out.println("[오늘은 E요일]: " + sdf.format(now));

        sdf = new SimpleDateFormat("올해의 D번째 날");
        System.out.println("[올해의 D번째 날]: " + sdf.format(now));

        sdf = new SimpleDateFormat("이달의 d번째 날");
        System.out.println("[이달의 d번째 날]: " + sdf.format(now));
    }
}
```

[실행결과]

```
[yyyy-MM-dd]: 2020-03-07
[yyyy년 MM월 dd일]: 2020년 03월 07일
[yyyy.MM.dd a HH:mm:ss]: 2020.03.07 오후 14:05:03
[오늘은 E요일]: 오늘은 토요일
[올해의 D번째 날]: 올해의 67번째 날
[이달의 d번째 날]: 이달의 7번째 날
```

11.12.3 문자열 형식 클래스(MessageFormat)

데이터를 파일에 저장하거나, 네트워크를 전송할 때, 그리고 데이터베이스 SQL문을 작성할 때 등 많은 부분에서 일정한 형식의 문자열을 사용한다. MessageFormat클래스를 사용하면 문자열에 데이터가 들어갈 자리를 표시해 두고, 프로그램이 실행하면서 동적으로 데이터를 삽입해 문자열을 완성시킬 수 있다. 아래 예제는 매개 변수화된 문자열을 이용해서 회원 정보 및 SQL문을 콘솔에 출력한다.

[Sample.java] 매개 변수화된 문자열 형식

```
public class Sample {
    public static void main(String[] args) {
        String id = "java";
        String name = "신용권";
        String tel = "010-123-5678";

        String text = "회원 ID: {0} \n회원 이름: {1} \n회원 전화: {2}";
        String result1 = MessageFormat.format(text, id, name, tel);
        System.out.println(result1);

        System.out.println();

        String sql = "insert into number values({0}, {1}, {2})";
        Object[] arguments = {"java", "신용권", "010-123-5678"};
        String result2 = MessageFormat.format(sql, arguments);
        System.out.println(result2);
    }
}
```

[실행결과]

회원 ID: java
회원 이름: 신용권
회원 전화: 010-123-5678

insert into number values(java,신용권,010-123-5678)

11.13 java.time 패키지

날짜와 시간을 조작하거나 비교하는 기능이 불충분하므로 자바 8부터 날짜와 시간을 나타내는 API를 새롭게 추가했다. 이 API는 java.util 패키지에 없고 별도로 java.time 패키지와 하위 패키지로 제공된다.

11.13.1 날짜와 시간 객체 생성

java.time 패키지에는 날짜와 시간을 표현하는 LocalDate, LocalTime, LocalDateTime, ZonedDateTime, Instant 클래스가 있다.

[Sample.java] 날짜와 시간 객체 생성

```
public class Sample {
    public static void main(String[] args) throws Exception {
        //날짜 얻기
        LocalDate currDate = LocalDate.now();
        System.out.println("현재 날짜: " + currDate);

        LocalDate targetDate = LocalDate.of(2024, 5, 10);
        System.out.println("목표 날짜: " + targetDate + "\n");

        //시간 얻기
        LocalTime currTime = LocalTime.now();
        System.out.println("현재 시간: " + currTime);

        LocalTime targetTime = LocalTime.of(6, 30, 0, 0);
        System.out.println("목표 시간: " + targetTime + "\n");

        //날짜와 시간 얻기
        LocalDateTime currDateTime = LocalDateTime.now();
        System.out.println("현재 날짜와 시간: " + currDateTime);

        LocalDateTime targetDateTime = LocalDateTime.of(2024, 5, 10, 6, 30, 0, 0);
        System.out.println("목표 날짜와 시간: " + targetDateTime + "\n");

        //협정 세계시와 시간존(TimeZone)
        ZonedDateTime utcDateTime = ZonedDateTime.now(ZoneId.of("UTC"));
        System.out.println("협정 세계시: " + utcDateTime);
        ZonedDateTime newyorkDateTime = ZonedDateTime.now(ZoneId.of("America/New_York"));
        System.out.println("뉴욕 시간존: " + newyorkDateTime + "\n");

        //특정 시점의 타임스탬프 얻기
        Instant instant1 = Instant.now();
        Thread.sleep(10);
        Instant instant2 = Instant.now();
        if(instant1.isBefore(instant2)) {
            System.out.println("instant1이 빠릅니다.");
        } else if(instant1.isAfter(instant2)) {
            System.out.println("instant1이 늦습니다.");
        } else {
            System.out.println("동일한 시간입니다.");
        }
        System.out.println("차이(nanos): " + instant1.until(instant2, ChronoUnit.NANOS));
    }
}
```

[실행결과]

현재 날짜: 2020-03-07
목표 날짜: 2024-05-10

현재 시간: 15:04:11.807
목표 시간: 06:30

현재 날짜와 시간: 2020-03-07T15:04:11.807
목표 날짜와 시간: 2024-05-10T06:30

협정 세계시: 2020-03-07T06:04:11.807Z[UTC]
뉴욕 시간존: 2020-03-07T01:04:11.807-05:00[America/New_York]

instant1이 빠릅니다.
차이(nanos): 16000000

11.3.2 날짜와 시간에 대한 정보 얻기

LocalDate와 LocalTime은 날짜와 시간 정보를 이용할 수 있도록 다양한 메서드들을 제공한다. LocalDateTime과 ZonedDateTime은 날짜와 시간 정보를 모두 갖고 있기 때문에 대부분의 메서드들을 가지고 있다. isLeapYear()는 LocalDate에만 있기 때문에 toLocalDate() 메서드로 LocalDate로 변환한 후 사용할 수 있다. ZonedDateTime은 시간존에 대한 정보를 제공하는 다음 메서드들을 추가적으로 가지고 있다.

클래스	리턴 타입	메서드(매개 변수)	설명
ZonedDateTime	Zoned	getZone()	존아이디를 리턴 (예:Asia/Seoul)
	ZoneOffset	getOffset()	존오프셋(시차)을 리턴

[Sample.java] 날짜와 시간 정보

```
public class Sample {
    public static void main(String[] args) throws Exception {
        LocalDateTime now = LocalDateTime.now();
        System.out.println(now);

        String strDateTime = now.getYear() + "년 ";
        strDateTime += now.getMonthValue() + "월 ";
        strDateTime += now.getDayOfMonth() + "일 ";
        strDateTime += now.getDayOfWeek() + " ";
        strDateTime += now.getHour() + "시 ";
        strDateTime += now.getMinute() + "분 ";
        strDateTime += now.getSecond() + "초 ";
        strDateTime += now.getNano() + "나노초";
        System.out.println(strDateTime + "\n");

        LocalDate nowDate = now.toLocalDate();
        if(nowDate.isLeapYear()) {
            System.out.println("올해는 윤년: 2월은 29일까지 있습니다.\n");
        } else {
            System.out.println("올해는 평년: 2월은 28일까지 있습니다.\n");
        }

        //협정 세계시와 존오프셋
        ZonedDateTime utcDateTime = ZonedDateTime.now(ZoneId.of("UTC"));
        System.out.println("협정 세계시: " + utcDateTime);
        ZonedDateTime seoulDateTime = ZonedDateTime.now(ZoneId.of("Asia/Seoul"));
        System.out.println("서울 타임존: " + seoulDateTime);
        ZoneId seoulZoneId = seoulDateTime.getZone();
        System.out.println("서울 존아이디: " + seoulZoneId);
        ZoneOffset seoulZoneOffset = seoulDateTime.getOffset();
        System.out.println("서울 존오프셋: " + seoulZoneOffset + "\n");
    }
}
```

[실행결과]

2020-03-07T15:38:56.398

2020년 3월 7일 SATURDAY 15시 38분 56초 398000000나노초

올해는 윤년: 2월은 29일까지 있습니다.

협정 세계시: 2020-03-07T06:38:56.398Z[UTC]

서울 타임존: 2020-03-07T15:38:56.398+09:00[Asia/Seoul]

서울 존아이디: Asia/Seoul

서울 존오프셋: +09:00

11.13.3 날짜와 시간을 조작하기

날짜와 시간 클래스들은 날짜와 시간을 조작하는 메서드와 상대 날짜를 리턴하는 메서드들을 가지고 있다. 아래 예제는 현재 날짜와 시간을 얻어 1년을 더하고, 2달을 빼고, 3일을 더하고, 4시간을 더하고, 5분을 빼고, 6초를 더한 날짜와 시간을 얻는다.

[Sample.java] 날짜와 시간 연산

```
public class Sample {
    public static void main(String[] args) throws Exception {
        LocalDateTime now = LocalDateTime.now();
        System.out.println("현재시: " + now);

        LocalDateTime targetDateTime = now
            .plusYears(1)
            .minusMonths(2)
            .plusDays(3)
            .plusHours(4)
            .minusMinutes(5)
            .plusSeconds(6);
        System.out.println("연산후: " + targetDateTime);
    }
}
```

[실행결과]

현재시: 2020-03-07T15:48:57.921

연산후: 2021-01-10T19:44:03.921

with() 메서드를 제외한 나머지는 메서드 이름만 보면 어떤 정보를 수정하는지 알 수 있다. with() 메서드는 상대 변경으로 현재 날짜를 기준으로 해의 첫 번째 일 또는 마지막일, 달의 첫 번째 일 또는 마지막 일, 달의 첫 번째 요일, 지난 요일 및 돌아오는 요일 등 상대적인 날짜를 리턴한다. 매개값은 TemporalAdjuster 타입으로 다음 표에 있는 TemporalAdjusters의 정적 메서드를 호출하면 얻을 수 있다.

리턴타입	메서드(매개 변수)	설명
TemporalAdjuster	firstDayOfYear()	이번 해의 첫 번째 일
	lastDayOfYear()	이번 해의 마지막 일
	firstDayOfNextYear()	다음 해의 첫 번째 일
	firstDayOfMonth()	이번 달의 첫 번째 일
	lastDayOfMonth()	이번 달의 마지막일
	firstDayOfNextMonth()	다음 달의 첫 번째 일
	firstInMonth(DayOfWeek dayOfWeek)	이번 달의 첫 번째 요일
	lastInMonth(DayOfWeek dayOfWeek)	이번 달의 마지막 요일
	next(DayOfWeek dayOfWeek)	돌아오는 요일
	nextOrSame(DayOfWeek dayOfWeek)	돌아오는 요일(오늘 포함)
	previous(DayOfWeek dayOfWeek)	지난 요일
	PreviousOrSame(DayOfWeek dayOfWeek)	지난 요일(오늘 포함)

[Sample.java] 날짜와 시간 변경

```
public class Sample {
    public static void main(String[] args) throws Exception {
        LocalDateTime now = LocalDateTime.now();
        System.out.println("현재: " + now);

        LocalDateTime targetDateTime = null;

        //직접변경
        targetDateTime = now
            .withYear(2024)
            .withMonth(10)
            .withDayOfMonth(5)
            .withHour(13)
            .withMinute(30)
            .withSecond(20);
        System.out.println("직접 변경: " + targetDateTime);

        //년도 상대 변경
        targetDateTime = now.with(TemporalAdjusters.firstDayOfYear());
        System.out.println("이번 해의 첫 일: " + targetDateTime);
        targetDateTime = now.with(TemporalAdjusters.lastDayOfYear());
        System.out.println("이번 해의 마지막 일: " + targetDateTime);
        targetDateTime = now.with(TemporalAdjusters.firstDayOfNextYear());
        System.out.println("다음 해의 첫 일: " + targetDateTime);

        //월의 상대 변경
        targetDateTime = now.with(TemporalAdjusters.firstDayOfMonth());
        System.out.println("이번 달의 첫 일: " + targetDateTime);
        targetDateTime = now.with(TemporalAdjusters.lastDayOfMonth());
        System.out.println("이번 달의 마지막 일: " + targetDateTime);
        targetDateTime = now.with(TemporalAdjusters.firstDayOfNextMonth());
        System.out.println("다음 달의 첫 일: " + targetDateTime);

        //요일 상대 변경
        targetDateTime = now.with(TemporalAdjusters.firstInMonth(DayOfWeek.MONDAY));
        System.out.println("이번 달의 첫 월요일: " + targetDateTime);
        targetDateTime = now.with(TemporalAdjusters.next(DayOfWeek.MONDAY));
        System.out.println("돌아오는 월요일: " + targetDateTime);
        targetDateTime = now.with(TemporalAdjusters.previous(DayOfWeek.MONDAY));
        System.out.println("지난 월요일: " + targetDateTime);
    }
}
```

[실행결과]

```
현재: 2020-03-07T16:45:34.624
직접 변경: 2024-10-05T13:30:20.624
이번 해의 첫 일: 2020-01-01T16:45:34.624
이번 해의 마지막 일: 2020-12-31T16:45:34.624
다음 해의 첫 일: 2021-01-01T16:45:34.624
이번 달의 첫 일: 2020-03-01T16:45:34.624
이번 달의 마지막 일: 2020-03-31T16:45:34.624
다음 달의 첫 일: 2020-04-01T16:45:34.624
이번 달의 첫 월요일: 2020-03-02T16:45:34.624
돌아오는 월요일: 2020-03-09T16:45:34.624
지난 월요일: 2020-03-02T16:45:34.624
```

11.13.4 날짜와 시간을 비교하기

Period와 Duration은 날짜와 시간의 양을 나타내는 클래스이다. Period는 년, 달, 일의 양을 나타내는 클래스이고, Duration은 시, 분, 초, 나노초의 양을 나타내는 클래스이다. 이 클래스들은 D-day, D-time을 구할 때 사용될 수 있다. 다음은 Period와 Duration에서 제공하는 메소드들이다.

클래스	리턴 타입	메서드(매개 변수)	설명
Period	int	getYears()	년의 차이
	int	getMonths()	달의 차이
	int	getDays()	일의 차이
Duration	int	getSeconds()	초의 차이
	int	getNano()	나노초의 차이

between() 메서드는 period와 Duration 클래스, 그리고 ChronoUnit 열거 타입에도 있다. Period와 Duration의 between()은 년, 달, 일, 초의 단순 차이를 리턴하고, ChronoUnit 열거 타입의 between()은 전체 시간을 기준으로 차이를 리턴한다. 예를 들어 2023년 1월과 2024년 3월의 달의 차이를 구할 때 Period의 between()은 2가 되고 ChronoUnit.MONTHS.between()은 14가 된다.

[Sample.java] 날짜와 시간 비교

```
public class Sample {
    public static void main(String[] args) throws Exception {
        LocalDateTime startDateTime = LocalDateTime.of(2023, 1, 1, 9, 0, 0);
        System.out.println("시작일: " + startDateTime);

        LocalDateTime endDateTime = LocalDateTime.of(2024, 3, 31, 18, 0, 0);
        System.out.println("종료일: " + endDateTime + "\n");

        //-----
        if(startDateTime.isBefore(endDateTime)) {
            System.out.println("진행 중입니다." + "\n");
        } else if(startDateTime.isEqual(endDateTime)) {
            System.out.println("종료합니다." + "\n");
        } else if(startDateTime.isAfter(endDateTime)) {
            System.out.println("종료했습니다." + "\n");
        }

        //-----
        System.out.println("[종료까지 남은 시간]");
        long remainYear = startDateTime.until(endDateTime, ChronoUnit.YEARS);
        long remainMonth = startDateTime.until(endDateTime, ChronoUnit.MONTHS);
        long remainDay = startDateTime.until(endDateTime, ChronoUnit.DAYS);
        long remainHour = startDateTime.until(endDateTime, ChronoUnit.HOURS);
        long remainMinute = startDateTime.until(endDateTime, ChronoUnit.MINUTES);
        long remainSecond = startDateTime.until(endDateTime, ChronoUnit.SECONDS);

        remainYear = ChronoUnit.YEARS.between(startDateTime, endDateTime);
        remainMonth = ChronoUnit.MONTHS.between(startDateTime, endDateTime);
        remainDay = ChronoUnit.DAYS.between(startDateTime, endDateTime);
        remainHour = ChronoUnit.HOURS.between(startDateTime, endDateTime);
        remainSecond = ChronoUnit.SECONDS.between(startDateTime, endDateTime);

        System.out.println("남은 해: " + remainYear);
        System.out.println("남은 달: " + remainMonth);
        System.out.println("남은 일: " + remainDay);
        System.out.println("남은 시간: " + remainHour);
        System.out.println("남은 분: " + remainMinute);
        System.out.println("남은 초: " + remainSecond + "\n");

        //-----
        System.out.println("[종료까지 남은 기간]");
        Period period = Period.between(startDateTime.toLocalDate(), endDateTime.toLocalDate());
        System.out.print("남은 기간: " + period.getYears() + "년 ");
        System.out.print(period.getMonths() + "달 ");
        System.out.print(period.getDays() + "일\n");

        //-----
        Duration duration = Duration.between(startDateTime.toLocalTime(), endDateTime.toLocalTime());
        System.out.println("남은 초: " + duration.getSeconds());
    }
}
```

[실행결과]

시작일: 2023-01-01T01:09

종료일: 2024-03-31T18:00

진행 중입니다.

[종료까지 남은 시간]

남은 해: 1

남은 달: 14

남은 일: 455

남은 시간: 10936

남은 분: 656211

남은 초: 39372660

11.13.5 파싱과 포매팅

날짜와 시간 클래스는 문자열 파싱(parsing)해서 날짜와 시간을 생성하는 메서드와 이와 반대로 날짜와 시간을 포매팅(Formatting)된 문자열로 변환하는 메서드를 제공하고 있다.

- 파싱(Parsing) 메서드

LocalDate의 parse(CharSequence) 메서드는 기본적으로 ISO_LOCAL_DATE 포맷터를 사용해서 문자열을 파싱한다. ISO_LOCAL_DATE는 DateTimeFormatter의 상수로 정의되어 있는데 "2014-05-21" 형식의 포맷터이다.

```
LocalDate localDate = LocalDate.parse("2024-05-21");
```

다른 포맷터를 이용해 문자열을 파싱하고 싶다면 parse(CharSequence, DateTimeFormatter) 메서드를 사용할 수 있다. DateTimeFormatter는 ofPattern() 메서드로 정의할 수 있는데, 다음 코드는 "2024.05.21." 형식의 DateTimeFormatter를 정의하고 문자열을 파싱했다.

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy.MM.dd");
LocalDate localDate = LocalDate.parse("2024.05.21.", formatter);
```

parse(CharSequence)와 동일하게 "2024-05-21"이라는 문자열을 파싱해서 LocalDate 객체를 얻고 싶다면 다음과 같이 코드를 작성하면 된다.

```
LocalDate localDate = LocalDate.parse("2024-05-21", DateTimeFormatter.ISO_LOCAL_DATE);
```

만약 포맷터의 형식과 다른 문자열을 파싱하게 되면 DateTimeParseException이 발생한다.

[Sample.java] 문자열 파싱

```
public class Sample {
    public static void main(String[] args) throws Exception {
        DateTimeFormatter formatter;
        LocalDate localDate;

        localDate = LocalDate.parse("2024-05-21");
        System.out.println(localDate);

        formatter = DateTimeFormatter.ISO_LOCAL_DATE;
        localDate = LocalDate.parse("2024-05-21", formatter);
        System.out.println(localDate);

        formatter = DateTimeFormatter.ofPattern("yyyy/MM/dd");
        localDate = LocalDate.parse("2024/05/21", formatter);
        System.out.println(localDate);

        formatter = DateTimeFormatter.ofPattern("yyyy.MM.dd");
        localDate = LocalDate.parse("2024.05.21", formatter);
        System.out.println(localDate);
    }
}
```

[실행결과]
2024-05-21
2024-05-21
2024-05-21
2024-05-21

- 포매팅(Formatting) 메서드

format()는 날짜와 시간을 포맷팅된 문자열로 변환시키는 메서드이다. format()의 매개값은 DateTimeFormatter인데 해당 형식대로 문자열을 리턴한다. 다음은 LocalDateTime으로부터 "2024년 5월 21일 오후6시 30분"과 같은 문자열을 얻는 코드이다.

```
LocalDateTime now = LocalDateTime.now();
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy년 M월 d일 a 시 m분");
String nowString = now.format(formatter);
```

[Sample.java] 포맷팅된 문자열

```
public class Sample {
    public static void main(String[] args) throws Exception {
        LocalDateTime now = LocalDateTime.now();
        DateTimeFormatter formatter =
            DateTimeFormatter.ofPattern("yyyy년 M월 d일 a 시 m분");
        String nowString = now.format(formatter);
        System.out.println(nowString);
    }
}
```

[실행결과]
2020년 3월 7일 오후 7시 10분

12장 멀티 스레드

12.1 멀티 스레드 개념

12.1.1 프로세스와 스레드

운영체제에서는 실행 중인 하나의 애플리케이션을 **프로세스(process)**라고 부른다. 사용자가 애플리케이션을 실행하면 운영체제로부터 실행에 필요한 메모리를 할당받아 애플리케이션의 코드를 실행하는데 이것이 프로세스이다. 하나의 애플리케이션은 다중 프로세스를 만들기도 하는데, 예를 들어 Chrome 브라우저를 두 개 실행했다면 두 개의 Chrome 프로세스가 생성된 것이다.

멀티태스킹(multi tasking)은 두 가지 이상의 작업을 동시에 처리하는 것을 말하는데, 운영체제는 멀티태스킹을 할 수 있도록 CPU 및 메모리 자원을 프로세스마다 적절히 할당해 주고, 병렬로 실행시킨다. 예를 들어 워드로 문서 작업을 하면서 동시에 윈도우 미디어 플레이어로 음악을 들을 수 있다. 멀티태스킹이 꼭 멀티 프로세스를 뜻하지는 않는다. 한 프로세스 내에서 멀티태스킹을 할 수 있도록 만들어진 애플리케이션도 있다. 대표적인 것이 미디어 플레이어(Media player)와 메신저(Messenger)이다. 미디어 플레이어는 동영상 재생과 음악 재생이라는 두 작업을 동시에 처리하고, 메신저는 채팅 기능을 제공하면서 동시에 파일 전송 기능을 수행하기도 한다.

스레드(thread)는 사전적 의미로 한 가닥의 실이라는 뜻인데, 한 가지 작업을 실행하기 위해 순차적으로 실행할 코드를 실처럼 이어 놓았다고 해서 유래된 이름이다. 하나의 스레드는 하나의 코드 실행 흐름이기 때문에 한 프로세스 내에 스레드가 두 개라면 두 개의 코드 실행 흐름이 생긴다는 의미이다. 멀티 프로세스가 애플리케이션 단위의 멀티태스킹이라면 **멀티스레드**는 애플리케이션 내부에서의 멀티태스킹이라고 볼 수 있다.

멀티 프로세스들은 운영체제에서 할당받은 자신의 메모리를 가지고 실행하기 때문에 서로 독립적이다. 따라서 하나의 프로세스에서 오류가 발생해도 다른 프로세스에게 영향을 미치지 않는다. 하지만 멀티스레드는 하나의 프로세스 내부에 생성되기 때문에 하나의 스레드가 예외를 발생시키면 프로세스 자체가 종료될 수 있어 다른 스레드에게 영향을 미치게 된다.

멀티스레드는 다양한 곳에서 사용된다. 대용량 데이터의 처리 시간을 줄이기 위해 데이터를 분할해 병렬로 처리하는 곳에서 사용되기도 하고, UI를 가지고 있는 애플리케이션에서 네트워크 통신을 하기 위해 사용되기도 한다. 또한 다수 클라이언트의 요청을 처리하는 서버를 개발할 때에도 사용된다. 멀티 스레드는 애플리케이션을 개발하는데 꼭 필요한 기능이기 때문에 반드시 이해하고 활용할 수 있도록 해야 한다.

12.1.2 메인 스레드

모든 자바 애플리케이션은 메인스레드가 main()메서드를 실행하면서 시작된다. 메인 스레드는 멀티스레드를 생성해서 멀티태스킹을 수행할 수 있다. 싱글스레드 애플리케이션에서는 메인스레드가 종료하면 프로세스도 종료되지만 멀티스레드 애플리케이션에서는 실행 중인 스레드가 하나라도 있다면, 프로세스는 종료되지 않는다. 메인스레드가 작업스레드보다 먼저 종료되더라도 작업스레드가 계속 실행 중이라면 프로세스는 종료되지 않는다.

12.2 작업 스레드 생성과 실행

어떤 자바 애플리케이션이건 메인스레드는 반드시 존재하기 때문에 메인 작업 이외에 추가적인 병렬 작업의 수만큼 스레드를 생성하면 된다. 자바에서는 작업 스레드도 객체로 생성되기 때문에 클래스가 필요하다. java.lang.Thread 클래스를 직접 객체화해서 생성해도 되지만, Thread를 상속해서 하위 클래스를 만들어 생성할 수도 있다.

12.2.1 Thread 클래스로부터 직접 생성

java.lang.Thread 클래스로부터 작업 스레드 객체를 직접 생성하려면 아래와 같이 Runnable을 매개값으로 갖는 생성자를 호출해야 한다.

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

Runnable은 작업 스레드가 실행할 수 있는 코드를 가지고 있는 객체라고 해서 붙여진 이름이다. Runnable은 인터페이스 타입이기 때문에 구현 객체를 만들어 대입해야 한다. Runnable에는 run() 메서드 하나가 정의되어 있는데, 구현 클래스는 run()을 재정의해서 작업 스레드가 실행할 코드를 작성해야 한다. 다음은 Runnable 구현 클래스를 작성하는 방법을 보여준다.

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

코드를 절약하기 위해 Thread 생성자를 호출할 때 Runnable 익명 객체를 매개값으로 사용할 수 있다. 오히려 이 방법이 더 많이 사용된다.

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

아래 예제는 비프음을 발생시키면서 동시에 프린팅을 하는 예제이다. 비프음을 발생시키면서 동시에 프린팅을 하려면 두 작업 중 하나를 메인 스레드가 아닌 다른 스레드에서 실행시켜야 한다. 프린팅은 메인 스레드가 담당하고 비프음을 들려주는 것은 작업 스레드가 담당하도록 작성하였다.

[BeepTask.java] 비프음을 들려주는 작업 정의

```
import java.awt.Toolkit;

public class BeepTask implements Runnable {
    @Override
    public void run() {
        Toolkit toolkit = Toolkit.getDefaultToolkit();
        for(int i=0; i<5; i++) {
            toolkit.beep();
            try{Thread.sleep(500);} catch(Exception e){}
        }
    }
}
```

//스레드 실행 내용

[Sample.java] 메인 스레드와 작업 스레드가 동시에 실행

```
public class Sample {
    public static void main(String[] args) { //메인 스레드
        Runnable beepTask = new BeepTask();
        Thread thread = new Thread(beepTask);
        thread.start(); //BeepTask 시작
        for(int i=0; i<5; i++) {
            System.out.println("땡");
            try {
                thread.sleep(500);
            }catch(Exception e) {}
        }
    }
}
```

[실행결과]

땡
땡
땡
땡
땡

다음은 위 프로그램을 Runnable 익명 객체를 이용해 작업 스레드를 만들 수 있는 또 다른 방법으로 작성한 예제이다.

[Sample.java] Runnable 익명 객체 이용

```
public class Sample {
    public static void main(String[] args) { //메인 스레드
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                Toolkit toolkit = Toolkit.getDefaultToolkit();
                for(int i=0; i<5; i++) {
                    toolkit.beep();
                    try { Thread.sleep(500); } catch(Exception e) {}
                }
            }
        });
        thread.start(); //BeepTask 시작
        for(int i=0; i<5; i++) {
            System.out.println("땡");
            try {
                thread.sleep(500);
            }catch(Exception e) {}
        }
    }
}
```

//스레드 실행 내용

12.2.2 Thread 하위 클래스로부터 생성

작업 스레드가 실행할 작업을 Runnable로 만들지 않고, Thread의 하위 클래스로 작업 스레드를 정의하면서 작업 내용을 포함시킬 수도 있다.

```
public class WorkerThread extends Thread {
    @Override
    public void run() {
        //스레드가 실행할 코드
    }
}
```

// run()메서드 정의

12.2.3 스레드의 이름

스레드는 자신의 이름을 가지고 있다. 메인 스레드는 "main"이라는 이름을 가지고 있고, 직접 생성한 스레드는 자동적으로 "Thread-n"이라는 이름으로 설정된다. n은 스레드의 번호를 말한다. Thread-n 대신 다른 이름으로 설정하고 싶다면 Thread 클래스의 setName() 메서드로 변경하면 된다. 반대로 스레드 이름을 알고 싶을 경우에는 getName() 메서드를 호출하면 된다. 아래 예제는 메인 스레드의 참조를 얻어 스레드 이름을 콘솔에 출력하고, 생성한 스레드의 이름을 setName() 메서드로 설정한 후, getName() 메서드로 읽어오도록 했다.

[ThreadA.java] ThreadA 클래스

```
public class ThreadA extends Thread {
    public ThreadA() {
        setName("ThreadA"); //스레드 이름 설정
    }

    public void run() {
        for(int i=0; i<2; i++) {
            System.out.println( getName() + "가 출력한 내용"); //ThreadA 실행 내용
        }
    }
}
```

[ThreadB.java] ThreadB 클래스

```
public class ThreadB extends Thread {
    public void run() {
        for(int i=0; i<2; i++) {
            System.out.println(getName() + "가 출력한 내용"); //ThreadB 실행 내용
        }
    }
}
```

[Sample.java] 메인 스레드 이름 출력 및 UserThread 생성 및 시작

```
public class Sample {
    public static void main(String[] args) {
        Thread mainThread = Thread.currentThread(); //이 코드를 실행하는 스레드 객체 얻기
        System.out.println("프로그램 시작 스레드 이름:" + mainThread.getName());

        ThreadA threadA = new ThreadA(); //Thread 생성
        System.out.println("작업 스레드 이름:" + threadA.getName());
        threadA.start(); //ThreadA 시작

        ThreadB threadB = new ThreadB(); //ThreadB 생성
        //스레드 이름 얻기
        System.out.println("작업 스레드 이름:" + threadB.getName());
        threadB.start(); //ThreadB 시작
    }
}
```

[실행결과]

```
프로그램 시작 스레드 이름:main
작업 스레드 이름:ThreadA
ThreadA가 출력한 내용
ThreadA가 출력한 내용
작업 스레드 이름:Thread-1
Thread-1가 출력한 내용
Thread-1가 출력한 내용
```

12.3 스레드 우선순위

멀티스레드의 동시성은 멀티 작업을 위해 하나의 코어에서 멀티 스레드가 번갈아가며 실행하는 성질을 말하고, 병렬성은 멀티 작업을 위해 멀티 코어에서 개별 스레드를 동시에 실행하는 성질을 말한다. 싱글 코어 CUP를 이용한 멀티 스레드 작업은 병렬적으로 실행되는 것처럼 보이지만, 사실은 번갈아가며 실행하는 동시성 작업이다. 번갈아 실행하는 것이 워낙 빠르다보니 병렬성으로 보일 뿐이다.

스레드의 개수가 코어의 수보다 많을 경우, 스레드를 어떤 순서에 의해 동시성으로 실행할 것인가를 결정해야 하는데, 이것을 스레드 스케줄링이라고 한다. 스레드 스케줄링에 의해 스레드들은 아주 짧은 시간에 번갈아 가면서 그들의 run() 메서드를 조금씩 실행한다.

자바의 스레드 스케줄링은 우선순위방식(Priority)과 순환 할당 방식(Round-Robin)을 사용한다. 우선순위 방식은 우선순위가 높은 스레드가 실행 상태를 더 많이 가지도록 스케줄링 하는 것을 말한다. 순환 할당 방식은 시간 할당량을 정해서 하나의 스레드를 정해진 시간만큼 실행하고 다시 다른 스레드를 실행하는 방식을 말한다. 우선순위 방식은 스레드 객체에 우선순위를 부여할 수 있기 때문에 개발자가 코드로 제어할 수 있다. 하지만 순환 할당 방식은 자바 가상 기계에 의해서 정해지기 때문에 코드로 제어할 수 없다.

우선순위 방식에서 우선순위는 1에서부터 10까지 부여되는데, 1이 가장 우선순위가 낮고, 10이 가장 높다. 우선순위를 부여하지 않으면 모든 스레드들은 기본적으로 5의 우선순위를 받는다. 만약 우선순위를 변경하고 싶다면 Thread 클래스가 제공하는 setPriority() 메서드를 이용하면 된다.

```
thread.setPriority(우선순위);
```

아래 예제는 스레드 10개를 생성하고 20억 번의 루핑을 누가 더 빨리 끝내는가를 테스트한 예제이다. Thread1~Thread9는 우선순위를 가장 낮게 주었고, Thread10은 우선순위를 가장 높게 주었다. 결과는 Thread10의 계산 작업이 가장 빨리 끝난다.

[CalcThread.java] 작업 스레드

```
public class CalcThread extends Thread {
    public CalcThread(String name) {
        setName(name); //스레드 이름 변경
    }
    public void run() {
        for(int i=0; i<2000000000; i++) { } //스레드가 실행할 내용
        System.out.println(getName());
    }
}
```

[Sample.java] 우선순위를 설정해서 스레드 실행

```
public class Sample {
    public static void main(String[] args) {
        for(int i=1; i<=10; i++) {
            Thread thread = new CalcThread( "thread" + i ); //스레드 이름
            if(i != 10) {
                thread.setPriority( Thread.MIN_PRIORITY ); //가장 낮은 우선순위 설정
            } else {
                thread.setPriority( Thread.MAX_PRIORITY ); //가장 높은 우선순위 설정
            }
            thread.start();
        }
    }
}
```

[실행결과]

```
thread10
thread1
thread3
thread7
thread6
thread4
thread5
thread8
thread9
thread2
```

12.4 동기화 메서드와 동기화 블록

12.4.1 공유 객체를 사용할 때 주의할 점

싱글 스레드 프로그램에서는 한 개의 스레드가 객체를 독차지해서 사용하면 되지만, 멀티 스레드 프로그램에서는 스레드들이 객체를 공유해서 작업해야 하는 경우가 있다. 이 경우, 스레드 A를 사용하던 객체가 스레드 B에 의해 상태가 변경될 수 있기 때문에 스레드 A가 의도했던 것과는 다른 결과를 산출할 수도 있다. 이는 마치 여러 사람이 계산기를 함께 나눠 쓰는 상황과 같아서 사람 A가 계산기로 작업을 하다가 계산 결과를 메모리에 저장한 뒤 잠시 자리를 비웠을 때 사람 B가 계산기를 만져서 앞 사람이 메모리에 저장한 값을 다른 값으로 변경하는 것과 같다.

12.4.2 동기화 메서드 및 동기화 블록

스레드가 사용 중인 객체를 다른 스레드가 변경할 수 없도록 하려면 스레드 작업이 끝날 때까지 객체에 잠금을 걸어서 다른 스레드가 사용할 수 없도록 해야 한다. 자바는 단 하나의 스레드만 실행할 수 있는 영역을 지정하기 위해 동기화(synchronized) 메서드와 동기화 블록을 제공한다. 스레드가 동기화 메서드 또는 블록에 들어가면 즉시 객체에 잠금을 걸어 다른 스레드가 지정된 영역을 실행하지 못하도록 한다. 동기화 메서드를 만드는 방법은 메서드 선언에 synchronized 키워드를 붙이면 된다. synchronized 키워드는 인스턴스와 정적 메서드 어디든 붙일 수 있다.

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

메서드 전체 내용이 아니라, 일부 내용만 임계 영역으로 만들고 싶다면 아래와 같이 동기화(synchronized) 블록을 만들면 된다. 동기화 블록의 외부 코드들은 여러 스레드가 동시에 실행할 수 있지만, 동기화 블록의 내부코드는 임계 영역이므로 한 번만 실행할 수 있고 다른 스레드는 실행할 수 없다. 만약 동기화 메서드와 동기화 블록이 여러 개 있을 경우, 스레드가 이들 중 하나를 실행할 때 다른 스레드는 해당 메서드는 물론이고 다른 동기화 메서드 및 블록도 실행할 수 없다. 하지만 일반 메서드는 실행이 가능하다.

```
public void method() {
    //여러 스레드가 실행 가능 영역
    ...
    synchronized( 공유객체 ) {
        임계 영역 //단 하나의 스레드만 실행 //동기화 블록
    }
    //여러 스레드가 실행 가능 영역
    ...
}
```

12.5 스레드 상태

스레드 객체를 생성하고, start() 메서드를 호출하면 곧바로 스레드가 실행되는 것처럼 보이지만 사실을 실행 대기 상태가 된다. 실행 대기 상태란 스케줄링이 되지 않아서 실행을 기다리고 있는 상태를 말한다. 실행 대기 상태에 있는 스레드 중에서 스레드 스케줄링으로 선택된 스레드가 바로 CPU를 점유하고 run() 메서드를 실행한다. 이때를 실행(Running) 상태라고 한다. 실행 상태의 스레드는 run() 메서드를 모두 실행하기 전에 스레드 스케줄링에 의해 다시 실행 대기 상태로 돌아갈 수 있다. 그리고 실행 대기 상태에 있는 다른 스레드가 선택되어 실행 상태가 된다. 이렇게 스레드는 실행 대기 상태와 실행 상태를 번갈아가면서 자신의 run() 메서드를 조금씩 실행한다. 실행상태에서 run() 메서드가 종료되면, 더 이상 실행할 코드가 없기 때문에 스레드의 실행은 멈추게 된다. 이 상태를 종료 상태라고 한다.

아래 예제는 스레드의 상태를 출력하는 StatePrintThread 클래스이다. 생성자 매개값으로 받은 타겟 스레드의 상태를 0.5초 주기로 출력한다.

[StatePrintThread.java] 타겟 스레드의 상태를 출력하는 스레드

```
public class StatePrintThread extends Thread {
    private Thread targetThread;

    public StatePrintThread( Thread targetThread ) { //상태를 조사할 스레드
        this.targetThread = targetThread;
    }

    public void run() {
        while(true) {
            Thread.State state = targetThread.getState(); //스레드 상태 얻기
            System.out.println("타겟 스레드 상태:" + state);

            if(state == Thread.State.NEW) { //객체 생성 상태일 경우 실행 대기 상태로 만듦
                targetThread.start();
            }

            if(state == Thread.State.TERMINATED) { //종료 상태일 경우 while문을 종료함
                break;
            }

            try{
                Thread.sleep(500); //0.5초간 일시 정지
            }catch(Exception e) {}
        }
    }
}
```

다음은 타겟 스레드 클래스이다. 10억 번 루핑을 돈 후 RUNNABLE 상태를 유지하고 sleep() 메서드를 호출해서 1.5초간 TIMED_WAITING 상태를 유지한다. 그리고 다시 10억 번 루핑을 돌게 해서 RUNNABLE 상태를 유지한다.

[TargetThread.java] 타겟 스레드

```
public class TargetThread extends Thread {
    public void run() {
        for(long i=0; i<1000000000; i++) {}
        try{
            Thread.sleep(1500); //1.5초간 일시 정지
        }catch(Exception e) {}
        for(long i=0; i<1000000000; i++) {}
    }
}
```

TargetThread가 객체로 생성되면 NEW 상태를 가지고 run() 메서드가 종료되면 TERMINATED 상태가 되므로 결국 다음 상태로 변한다.

NEW → RUNNABLE → TIMED_WAITING → RUNNABLE → TERMINATED

아래 예제는 StatePrintThread를 생성해서 매개값으로 전달 받은 TargetThread의 상태를 출력하도록 작성된 실행 클래스이다.

[Sample.java] 실행 클래스

```
public class Sample {
    public static void main(String[] args) {
        StatePrintThread statePrintThread = new StatePrintThread(new TargetThread());
        statePrintThread.start();
    }
}
```

12.6 스레드 상태 제어

12.6.1 주어진 시간동안 일시 정지(sleep())

실행 중인 스레드를 일정 시간 멈추게 하고 싶다면 Thread 클래스의 정적 메서드인 sleep()을 사용하면 된다. Thread.sleep() 메서드를 호출한 스레드는 주어진 시간 동안 일시 정지 상태가 되고, 다시 실행 대기 상태로 돌아간다. 매개값에는 얼마 동안 일시 정지 상태로 있을 것인지, 밀리세컨트(1/1000) 단위로 시간을 주면 된다. 1000이라는 값을 주면 스레드는 1초가 경과할 동안 일시 정지 상태로 있게 된다.

12.6.2 다른 스레드에게 실행 양보(yield())

스레드는 yield() 메서드를 제공하고 yield() 메서드를 호출한 스레드는 실행 대기 상태로 돌아가고 동일한 우선순위 또는 높은 우선순위를 갖는 다른 스레드가 실행 기회를 가질 수 있도록 해준다.

12.6.3 다른 스레드의 종료를 기다림(join())

스레드는 다른 스레드와 독립적으로 실행되므로 다른 스레드가 종료될 때까지 기다렸다가 실행해야 하는 경우가 발생할 수도 있다. 이런 경우를 위해서 Thread는 join() 메서드를 제공하고 있다.

12.6.4 스레드 간 협업(wait(), notify(), notifyAll())

두 개의 스레드를 교대로 번갈아가며 실행해야 할 경우, 한 스레드가 작업을 완료하면 notify() 메서드를 호출해서 일시 정지 상태에 있는 다른 스레드를 실행 대기 상태로 만들고, 자신은 두 번 작업을 하지 않도록 wait()메서드를 호출하여 일시 정지 상태로 만든다. notify()는 wait()에 의해 정지된 스레드 중 한 개를 실행 대기 상태로 만들고, notifyAll() 메서드는 wait()에 의해 정지된 모든 스레드들을 실행 대기 상태로 만든다.

12.6.5 스레드의 안전한 종료(stop 플래그, interrupt())

스레드는 자신의 run()메서드가 모두 실행되면 자동적으로 종료된다. 경우에 따라서는 실행 중인 스레드를 즉시 종료할 필요가 있다. Thread는 스레드를 강제로 종료하기 위해 stop()메서드를 제공한다. 그러나 갑자기 종료하게 되면 스레드가 사용 중이던 자원들이 불안정한 상태로 남겨지게 된다. 때문에 스레드를 즉시 종료시키기 위해 stop 플래그를 이용하는 방법과 interrupt() 메서드를 이용하는 방법이 있다.

12.7 데몬 스레드

데몬(daemon)스레드는 주 스레드의 작업을 돕는 보조적인 역할을 수행하는 스레드이다. 주 스레드가 종료되면 데몬 스레드의 존재 의미가 없어지기 때문에 강제로 자동 종료된다. 스레드를 데몬으로 만들기 위해서는 주 스레드가 데몬이 될 스레드의 setDaemon(true)를 호출해주면 된다. 아래 예제는 1초 주기로 save()메서드를 자동 호출하도록 AutoSaveThread를 작성하고, 메인 스레드가 3초 후 종료되면 AutoSaveThread도 같이 종료되도록 AutoSaveThread를 데몬 스레드로 만들었다.

[AutoSaveThread.java] 1초 주기로 save() 메서드를 호출하는 데몬 스레드

```
public class AutoSaveThread extends Thread {
    public void save() {
        System.out.println("작업 내용을 저장함.");
    }

    @Override
    public void run() {
        while(true) {
            try{
                Thread.sleep(1000);
            }catch(InterruptedException e) {
                break;
            }
            save();
        }
    }
}
```

[DeamonExample.java] 메인 스레드가 실행하는 코드

```
public class Sample {
    public static void main(String[] args) {
        AutoSaveThread autoSaveThread = new AutoSaveThread();
        autoSaveThread.setDaemon(true);
        autoSaveThread.start();

        try {
            Thread.sleep(3000);
        }catch(InterruptedException e) { }
        System.out.println("메인 스레드 종료");
    }
}
```

[실행결과]

작업 내용을 저장함.
작업 내용을 저장함.
작업 내용을 저장함.
메인 스레드 종료

12.8 스레드 그룹

스레드 그룹은 관련된 스레드를 묶어서 관리할 목적으로 이용된다. JVM이 실행되면 system 스레드 그룹을 만들고, JVM 운영에 필요한 스레드들을 생성해서 system 스레드 그룹에 포함시킨다. 그리고 system의 하위 스레드 그룹으로 main을 만들고 메인 스레드를 main 스레드 그룹에 포함시킨다. 스레드는 반드시 하나의 스레드 그룹에 포함되며, 명시적으로 스레드 그룹에 포함시키지 않으면 자신을 생성한 스레드와 같은 스레드 그룹에 속하게 된다. 우리가 생성하는 작업 스레드는 대부분 main 스레드가 생성하므로 기본적으로 main 스레드 그룹에 속하게 된다.

12.8.1 스레드 그룹 이름 얻기

현재 스레드가 속한 스레드 그룹의 이름을 얻고 싶다면 getName() 메서드를 이용해 얻을 수 있고, 프로세스 내에서 실행하는 모든 스레드에 대한 정보를 얻고 싶다면 Thread의 정적 메서드인 getAllStackTraces()를 이용하면 된다.

```
Map<Thread, StackTraceElement[]> map = Thread.getAllStackTraces();
```

12.8.2 스레드 그룹 생성

명시적으로 스레드 그룹을 만들고 싶다면 아래 생성자 중 하나를 이용해서 ThreadGroup 객체를 만든다. ThreadGroup 이름만 주거나, 부모 ThreadGroup과 이름을 매개값으로 줄 수 있다.

```
ThreadGroup tg = new ThreadGroup(String name);  
ThreadGroup tg = new ThreadGroup(ThreadGroup parent, String name);
```

12.8.3 스레드 그룹의 일괄 interrupt()

interrupt() 메서드를 이용하면 그룹 내에 포함된 모든 스레드들을 일괄 interrupt할 수 있다. 아래 예제는 스레드 그룹을 생성하고, 정보를 출력해 본다. 그리고 3초 후 스레드 그룹의 interrupt() 메서드를 호출해서 스레드 그룹에 포함된 모든 스레드들을 종료시킨다.

[WorkThread.java] InterruptedException이 발생할 때 스레드가 종료되도록 함

```
public class WorkThread extends Thread {  
    public WorkThread(ThreadGroup threadGroup, String threadName) {  
        super(threadGroup, threadName); //스레드 그룹과 스레드 이름을 설정  
    }  
    @Override  
    public void run() {  
        while(true) {  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
                System.out.println(getName() + " interrupted");  
                break; //while문을 빠져나와 스레드를 종료시킴  
            }  
        }  
        System.out.println(getName() + " 종료됨");  
    }  
}
```

[실행결과]

```
[ main 스레드 그룹의 list() 메서드 출력 내용 ]  
java.lang.ThreadGroup[name=main,maxpri=10]  
  Thread[main,5,main]  
    java.lang.ThreadGroup[name=myGroup,maxpri=10]  
      Thread[workThreadA,5,myGroup]  
      Thread[workThreadB,5,myGroup]  
  
[ myGroup 스레드 그룹의 interrupt() 메서드 호출 ]  
workThreadA interrupted  
workThreadA 종료됨  
workThreadB interrupted  
workThreadB 종료됨
```

[Sample.java] 스레드 그룹을 이용한 일괄 종료 예제

```
public class Sample {  
    public static void main(String[] args) {  
        ThreadGroup myGroup = new ThreadGroup("myGroup");  
        WorkThread workThreadA = new WorkThread(myGroup, "workThreadA");  
        WorkThread workThreadB = new WorkThread(myGroup, "workThreadB"); //myGroup에 두 스레드를 포함시킴  
  
        workThreadA.start();  
        workThreadB.start();  
  
        System.out.println("[ main 스레드 그룹의 list() 메서드 출력 내용 ]");  
        ThreadGroup mainGroup = Thread.currentThread().getThreadGroup();  
        mainGroup.list();  
        System.out.println();  
  
        try {Thread.sleep(3000);} catch (InterruptedException e) {}  
  
        System.out.println("[myGroup 스레드 그룹의 interrupt() 메서드 호출 ]");  
        myGroup.interrupt();  
    }  
}
```

12.9 스레드 풀

병렬 작업 처리가 많아지면 스레드 개수가 증가되고 그에 따른 스레드 생성과 스케줄링으로 인해 CPU가 바빠져 메모리 사용량이 늘어난다. 따라서 애플리케이션의 성능이 저하된다. 갑작스런 병렬 작업의 폭증으로 인한 스레드의 폭증을 막으려면 스레드풀(ThreadPool)을 사용해야 한다. 스레드 풀은 작업 처리에 사용되는 스레드를 제한된 개수만큼 정해 놓고 작업 큐(Queue)에 들어오는 작업들을 하나씩 스레드가 맡아 처리한다.

12.9.1 스레드 풀 생성 및 종료

- 스레드풀 생성

ExecutorService 구현 객체는 Executors 클래스의 newCachedThreadPool(), newFixedThreadPool(int n Threads) 메서드를 이용해서 간편하게 생성할 수 있다.

- 스레드풀 종료

스레드풀의 스레드는 기본적으로 데몬 스레드가 아니기 때문에 main 스레드가 종료되더라도 작업을 처리하기 위해 계속 실행 상태로 남아있다. 애플리케이션을 종료하려면 스레드풀을 종료시켜 스레드들이 종료 상태가 되도록 처리해주어야 한다.

12.9.2 작업 생성과 처리 요청

- 작업 생성

하나의 작업은 Runnable 또는 Callable 구현 클래스로 이용해 작성할 수 있다. 두 클래스의 차이점은 작업 처리 완료 후 리턴값이 있느냐 없느냐이다. Runnable의 run() 메서드는 리턴값이 없는 반면에, Callable의 call() 메서드는 리턴값이 있다. 스레드풀의 스레드는 작업 큐에서 Runnable또는 Callable 객체를 가져와 run()과 call() 메서드를 실행한다.

- 작업 처리 요청

작업 처리 요청이란 ExecutorService의 작업 큐에 Runnable 또는 Callable 객체를 넣는 행위를 말한다. ExecutorServices는 작업 처리 요청을 위해 execute()와 submit() 메서드를 제공한다. 하나는 execute()는 작업 처리 결과를 받지 못하고 submit()은 처리 결과를 받을 수 있도록 Future를 리턴 한다. 또 다른 차이점은 execute()는 작업 처리 도중 예외가 발생하면 스레드가 종료되고 해당 스레드는 스레드풀에서 제거된다. 따라서 스레드풀은 다른 작업 처리를 위해 새로운 스레드를 생성한다. 반면에 submit()은 작업 처리 도중 예외가 발생하더라도 스레드는 종료되지 않고 다음 작업을 위해 재사용된다. 그렇기 때문에 가급적이면 스레드의 생성 오버헤드를 줄이기 위해서 submit()을 사용하는 것이 좋다.

아래 예제는 Runnable 작업을 정의할 때 Integer.parseInt("삼")을 넣어 NumberFormatException이 발생하도록 유도했다. 10개의 작업을 execute()와 submit() 메서드로 각각 처리 요청 했을 경우 스레드풀의 상태를 살펴보자. 먼저 execute() 메서드로 작업 요청을 했다.

[Sample.java] execute() 메서드로 작업 처리 요청한 경우

```
public class Sample {
    public static void main(String[] args) throws Exception {
        ExecutorService executorService = Executors.newFixedThreadPool(2); //최대 스레드 개수가 2인 스레드풀 생성

        for(int i=0; i<10; i++) {
            Runnable runnable = new Runnable() {
                @Override
                public void run() {
                    //스레드 총 개수 및 작업 스레드 이름 출력
                    ThreadPoolExecutor threadPoolExecutor = (ThreadPoolExecutor)executorService;
                    int poolSize = threadPoolExecutor.getPoolSize();
                    String threadName = Thread.currentThread().getName();
                    System.out.println("[총 스레드 개수: " + poolSize + "] 작업 스레드 이름: " + threadName);

                    //예외 발생 시점
                    int value = Integer.parseInt("삼");
                }
            };

            executorService.execute(runnable);
            //executorService.submit(runnable)

            Thread.sleep(10);
        }
        executorService.shutdown(); //스레드풀 종료
    }
}
```

위 결과를 실행하면 스레드풀의 스레드 최대 개수 2는 변함이 없지만, 실행 스레드의 이름을 보면 모두 다른 스레드가 작업을 처리하고 있다. 이것은 작업 처리 도중 예외가 발생했기 때문에 스레드는 제거되고 새 스레드가 계속 생성되기 때문이다. submit() 메서드로 작업을 하고 실행 결과를 보면 확실히 execute()와의 차이점을 발견할 수 있다. 예외가 발생하더라도 스레드가 종료되지 않고 계속 재사용되어 다른 작업을 처리하고 있는 것을 볼 수 있다.

12.9.3 블로킹 방식의 작업 완료 통보

- 리턴값이 없는 작업 완료 통보

리턴값이 없는 작업일 경우는 Runnable 객체로 생성한다.. 작업이 정상적으로 완료되었다면 get() 메서드는 null을 리턴하고 스레드가 작업 처리 도중 interrupt 되면 InterruptedException을 발생시키고, 작업 처리 도중 예외가 발생하면 ExecutionException을 발생시킨다. 아래 예제는 리턴값이 없고 단순히 1부터 10까지의 합을 출력하는 작업을 Runnable 객체로 생성하고, 스레드풀의 스레드가 처리하도록 요청한 것이다.

[Sample.java] 리턴값이 없는 작업 완료 통보

```
public class Sample {
    public static void main(String[] args) throws Exception {
        ExecutorService executorService = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

        System.out.println("[작업 처리 요청]");
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                int sum = 0;
                for(int i=1; i<=10; i++) { sum += i; }
                System.out.println("[처리 결과] " + sum);
            }
        };
        Future future = executorService.submit(runnable);

        try {
            future.get();
            System.out.println("[작업 처리 완료]");
        } catch (Exception e) {
            System.out.println("[실행 예외 발생함] " + e.toString());
        }

        executorService.shutdown();
    }
}
```

[실행결과]

```
[작업 처리 요청]
[처리 결과] 55
[작업 처리 완료]
```

- 리턴값이 있는 작업 완료 통보

스레드풀의 스레드가 작업을 완료한 후에 애플리케이션이 처리 결과를 얻어야 된다면 작업 객체를 Callable로 생성하면 된다. 아래 예제는 1부터 10까지의 합을 리턴하는 작업을 Callable 객체로 생성하고, 스레드풀의 스레드가 처리하도록 요청한 것이다.

[Sample.java] 리턴값이 있는 작업 완료 통보

```
public class Sample {
    public static void main(String[] args) throws Exception {
        ExecutorService executorService = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

        System.out.println("[작업 처리 요청]");
        Callable<Integer> task = new Callable<Integer>() {
            @Override
            public Integer call() throws Exception {
                int sum = 0;
                for(int i=1; i<=10; i++) {
                    sum += i;
                }
                return sum;
            }
        };
        Future<Integer> future = executorService.submit(task);

        try {
            int sum = future.get();
            System.out.println("[처리 결과] " + sum);
            System.out.println("[작업 처리 완료]");
        } catch (Exception e) {
            System.out.println("[실행 예외 발생함] " + e.toString());
        }

        executorService.shutdown();
    }
}
```

[실행결과]

```
[작업 처리 요청]
[처리 결과] 55
[작업 처리 완료]
```


- 작업 처리 결과를 외부 객체에 저장

상황에 따라서 스레드가 작업한 결과를 외부 객체에 저장해야 할 경우도 있다. 예를 들어 스레드가 작업을 완료하고 외부 Result 객체에 작업 결과를 저장하면, 애플리케이션 Result 객체를 사용해서 어떤 작업을 진행할 수 있을 것이다. 대개 Result 객체는 공유 객체가 되어, 두 개 이상의 스레드 작업을 취합할 목적으로 이용된다.

이런 작업을 하기 위해서 ExecutorService의 submit(Runnable task, V result) 메서드를 사용할 수 있는데, V가 바로 Result 타입이 된다. 메서드를 호출하면 즉시 Future<V>가 리턴되는데, Future의 get() 메서드를 호출하면 스레드가 작업을 완료할 때까지 블로킹되었다가 작업을 완료하면 V 타입 객체를 리턴한다. 리턴된 객체는 submit()의 두 번째 매개값으로 준 객체와 동일한데, 차이점은 스레드 처리 결과가 내부에 저장되어 있다는 것이다.

```
Result result = ...;
Runnable task = new Task(result);
Future<Result> future = executorService.submit(task, result);
result = future.get();
```

작업 객체는 Runnable 구현 클래스로 생성하는데, 주의할 점은 스레드에서 결과를 저장하기 위해 외부 Result 객체를 사용해야 하므로 생성자를 통해 Result 객체를 주입받도록 해야 한다.

아래 예제는 1부터 10까지의 합을 계산하는 두 개의 작업을 스레드풀에 처리 요청하고, 각각의 스레드가 작업을 완료한 후 산출된 값을 외부 Result 객체에 누적하도록 했다.

[Sample.java] 직접 처리 결과를 외부 객체에 저장

```
public class Sample {
    public static void main(String[] args) throws Exception {
        ExecutorService executorService = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

        System.out.println("[작업 처리 요청]");
        class Task implements Runnable{
            Result result;
            public Task(Result result) {
                this.result = result;
            }

            @Override
            public void run() {
                int sum = 0;
                for(int i=0; i<=10; i++) {
                    sum += i;
                }
                result.addValue(sum); //Result 객체에 작업 결과 저장
            }
        }

        Result result = new Result();
        Runnable task1 = new Task(result);
        Runnable task2 = new Task(result);
        Future<Result> future1 = executorService.submit(task1, result);
        Future<Result> future2 = executorService.submit(task2, result);

        try {
            result = future1.get();
            result = future2.get();
            System.out.println("[처리 결과] " + result.accmValue);
            System.out.println("[작업 처리 완료]");
        } catch (Exception e) {
            System.out.println("[실행 예외 발생함] " + e.toString());
        }
        executorService.shutdown();
    }
}

class Result{
    int accmValue;

    synchronized void addValue(int value) {
        accmValue += value;
    }
}
```

//작업 정의

//두가지 작업 결과를 취합

//처리 결과를 저장하는 Result 클래스

• 작업 완료 순으로 통보

작업 요청 순서대로 작업 처리가 완료되는 것은 아니다. 작업의 양과 스레드 스케줄링에 따라서 먼저 요청한 작업이 나중에 완료되는 경우도 발생한다. 여러 개의 작업들이 순차적으로 처리될 필요성이 없고, 처리 결과도 순차적으로 이용할 필요가 없다면 작업 처리가 완료된 것부터 결과를 얻어 이용하면 된다.

스레드풀에서 작업 처리가 완료된 것만 통보받는 방법이 있는데, CompletionService를 이용하는 것이다. CompletionService는 처리 완료된 작업을 가져오는 poll()과 take() 메서드를 제공한다.

리턴 타입	메서드(매개 변수)	설명
Future<V>	poll()	완료된 작업의 Future를 가져옴. 완료된 작업이 없다면 null을 리턴함
Future<V>	poll(long timeout, TimeUnit unit)	완료된 작업의 Future를 가져옴. 완료된 작업이 없다면 timeout까지 블로킹됨.
Future<V>	take()	완료된 작업의 Future를 가져옴. 완료된 작업이 없다면 있을 때까지 블로킹됨.
Future<V>	submit(Callable<V> task)	스레드풀에 Callable 작업 처리 요청
Future<V>	submit(Runnable task, V result)	스레드풀에 Runnable 작업 처리 요청

아래 예제는 3개의 Callable 작업을 처리 요청하고 처리가 완료되는 순으로 작업의 결과를 콘솔에 출력하도록 했다.

[Sample.java] 작업 완료 순으로 통보 받기

```

public class Sample {
    public static void main(String[] args) throws Exception {
        ExecutorService executorService = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

        //CompletionSeervice 생성
        CompletionService<Integer> completionService = new ExecutorCompletionService<Integer>(executorService);

        System.out.println("[작업 처리 요청]");

        for(int i=0; i<3; i++) {
            completionService.submit(new Callable<Integer>() {
                @Override
                public Integer call() throws Exception {
                    int sum = 0;
                    for(int i=1; i<=10; i++) {
                        sum += i;
                    }
                    return sum;
                }
            });
        }

        System.out.println("[처리 완료된 작업 확인]");

        executorService.submit(new Runnable() { //스레드풀의 스레드에서 실행하도록 함
            @Override
            public void run() {
                while(true) {
                    try {
                        Future<Integer> future = completionService.take(); //완료된 작업 가져오기
                        int value = future.get();
                        System.out.println("[처리 결과] " + value);
                    } catch (Exception e) {
                        break;
                    }
                }
            }
        });

        try {
            Thread.sleep(3000);
        } catch (Exception e) {}
        executorService.shutdown();
    }
}

```

//스레드풀에게 작업 처리 요청

[실행결과]

[작업 처리 요청]

[처리 완료된 작업 확인]

[처리 결과] 55

[처리 결과] 55

[처리 결과] 55

//3초 후 스레드풀 종료

12.9.4 콜백 방식의 작업 완료 통보

콜백이란 애플리케이션이 스레드에게 작업 처리를 요청한 후, 스레드가 작업을 완료하면 특정 메시지를 자동 실행하는 기법을 말한다. 이때 자동 실행되는 메시지를 콜백 메시지라고 한다. 블록킹 방식은 작업 처리를 요청한 후 작업이 완료될 때까지 블로킹되지만, 콜백 방식은 작업 처리를 요청한 후 결과를 기다릴 필요 없이 다른 기능을 수행할 수 있다. 작업 처리가 완료되면 자동적으로 콜백 메시지가 실행되어 결과를 알 수 있기 때문이다.

아쉽게도 ExecutorService는 콜백을 위한 별도의 기능을 제공하지 않는다. 하지만 Runnable 구현 클래스를 작성할 때 콜백 기능을 구현할 수 있다. 먼저 콜백 메시지를 가진 클래스가 있어야 하는데, 직접 정의해도 좋고 java.nio.channels.CompletionHandler를 이용해도 좋다. 이 인터페이스는 NIO 패키지에 포함되어 있는데 비동기 통신에서 콜백 객체를 만들 때 사용된다.

CompletionHandler는 completed()와 failed() 메시지가 있는데, completed()는 작업을 정상 처리 완료했을 때 호출되는 콜백 메시지이고, failed()는 작업 처리 도중 예외가 발생했을 때 호출되는 콜백 메시지이다. CompletionHandler의 V 타입 파라미터는 결과값의 타입이고, A는 첨부값의 타입이다. 첨부값은 콜백 메시지에 결과값 이외에 추가적으로 전달하는 객체라고 생각하면 된다. 만약 첨부값이 필요 없다면 A는 Void로 지정해주면 된다.

작업 처리가 정상적으로 완료되면 completed() 콜백 메시지를 호출해서 결과값을 전달하고, 예외가 발생하면 failed() 콜백 메시지를 호출해서 예외 객체를 전달한다. 아래 예제는 두 개의 문자열을 정수화해서 더하는 작업을 처리하고 결과를 콜백 방식으로 통보한다. 첫 번째 작업은 "3", "3"을 주었고 두 번째 작업은 "3", "삼"을 주었다. 첫 번째 작업은 정상적으로 처리되기 때문에 completed()가 자동 호출되고, 두 번째는 NumberFormatException이 발생되어 failed() 메시지가 호출된다.

[Sample.java] 콜백 방식의 작업 완료 통보받기

```
public class Sample {
    private ExecutorService executorService;

    public Sample(){
        executorService = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
    }

    private CompletionHandler<Integer, Void> callBack = new CompletionHandler<Integer, Void>(){
        @Override
        public void completed(Integer result, Void attachment) {
            System.out.println("completed() 실행: " + result);
        }
        //콜백 메시지를 가진 CompletionHandler 객체 생성

        @Override
        public void failed(Throwable exc, Void attachment) {
            System.out.println("failed() 실행: " + exc.toString());
        }
    };

    public void doWork(final String x, final String y) {
        Runnable task = new Runnable() {
            @Override
            public void run() {
                try {
                    int intX = Integer.parseInt(x);
                    int intY = Integer.parseInt(y);
                    int result = intX + intY;
                    callBack.completed(result, null); //정상 처리했을 경우 호출
                } catch (Exception e) {
                    callBack.failed(e, null); //예외가 발생했을 경우 호출
                }
            }
        };
        executorService.submit(task); //스레드풀에게 작업 처리 요청
    }

    public void finish() {
        executorService.shutdown(); //스레드풀 종료
    }

    public static void main(String[] args) throws Exception {
        Sample sample = new Sample();
        sample.doWork("3", "3");
        sample.doWork("3", "삼");
        sample.finish();
    }
}
```

[실행결과]

```
completed() 실행: 6
failed() 실행: java.lang.NumberFormatException: For input string: "삼"
```

13장 제네릭

13.1 왜 제네릭을 사용해야 하는가?

Java 5부터 제네릭(Generic) 타입이 새로 추가되었는데, 제네릭 타입을 이용함으로써 잘못된 타입이 사용될 수 있는 문제를 컴파일 과정에서 제거할 수 있게 되었다. 제네릭은 클래스와 인터페이스, 그리고 메서드를 정의할 때 타입(type)을 파라미터(parameter)로 사용할 수 있도록 한다. 타입 파라미터는 코드 작성 시 구체적인 타입으로 대체되어 다양한 코드를 생성하도록 해준다. 제네릭을 사용하는 이점은 아래와 같다.

- 컴파일 시 강한 타입 체크를 할 수 있다.

자바 컴파일러는 코드에서 잘못 사용된 타입 때문에 발생하는 문제점을 제거하기 위해 제네릭 코드에 대한 강한 타입 체크를 한다. 실행 시 타입 에러가 나는 것보다는 컴파일 시에 미리 타입을 강하게 체크해서 에러를 사전에 방지하는 것이 좋다.

```
List list = new ArrayList();
list.add("hello");
list.add(1234); //실행 시에 에러가 발생한다.
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(1234); //컴파일 시에 에러가 발생한다.
```

- 타입 변환(casting)을 제거한다.

비제네릭 코드는 불필요한 타입 변환을 하기 때문에 프로그램 성능에 악영향을 미친다. 하지만 제네릭 코드로 수정하면 특정 타입으로 국한하기 때문에 타입 변환을 할 필요가 없어 프로그램 성능이 향상된다.

```
List list = new ArrayList();
list.add("hellow");
String str = (String)list.get(0) //타입 변환을 해야 한다.
List<String> list = new ArrayList<String>();
list.add("hello");
String str = list.get(0); //타입 변환을 하지 않는다.
```

13.2 제네릭 타입 (class<T>, interface<T>)

제네릭 타입은 타입을 파라미터로 가지는 클래스와 인터페이스를 말한다. 제네릭 타입은 클래스 또는 인터페이스 이름 뒤에 "<>" 부호가 붙고, 사이에 타입 파라미터가 위치한다. 타입 파라미터는 변수명과 동일한 규칙에 따라 작성할 수 있지만, 일반적으로 대문자 알파벳 한글자로 표현한다. 아래 예제는 제네릭 클래스를 설계할 때 구체적인 타입을 명시하지 않고, 타입 파라미터로 대체했다가 실제 클래스가 사용될 때 구체적인 타입을 지정함으로써 타입 변환을 최소화한다. 타입 파라미터 T는 Sample 클래스에서 객체 생성시 String 타입과 Integer 타입으로 자동 변경된다.

[Box.java] 제네릭 타입

```
public class Box<T> {
    private T t;
    public T get() {
        return t;
    }
    public void set(T t) {
        this.t = t;
    }
}
```

[Sample.java] 제네릭 타입 이용

```
public class Sample {
    public static void main(String[] args) {
        Box<String> box1 = new Box<String>();
        box1.set("hello");
        String str = box1.get();
        System.out.println("str=" + str);

        Box<Integer> box2= new Box<Integer>();
        box2.set(6);
        int value = box2.get();
        System.out.println("value=" + value);
    }
}
```

[실행결과]
str=hello
value=6

13.3 멀티 타입 파라미터 (class<K, V, ..>, interface<K, V, ...>)

제네릭 타입은 두 개 이상의 멀티 타입 파라미터를 사용할 수 있는데, 이 경우 각 타입 파라미터를 콤마로 구분한다. 아래 예제는 Product<T, M> 제네릭 타입을 정의하고 Sample 클래스에서 Product<TV, String> 객체와 Product<Car, String> 객체를 생성한다. 그리고 Getter와 Setter를 호출하는 방법을 보여준다.

[TV.java] TV 클래스

```
public class TV { }
```

[Car.java] Car 클래스

```
public class Car { }
```

[Product.java] Product 제네릭 클래스

```
public class Product<T, M> {  
    private T kind;  
    private M model;  
    public T getKind() { return this.kind; }  
    public M getModel() { return this.model; }  
    public void setKind(T kind) { this.kind = kind; }  
    public void setModel(M model) { this.model=model; }  
}
```

[Sample.java] 제네릭 객체 생성

```
public class Sample {  
    public static void main(String[] args) {  
        Product<TV, String> product1 = new Product<TV, String>();  
        product1.setKind(new TV());  
        product1.setModel("스마트TV");  
        TV tv = product1.getKind();  
        String tvModel = product1.getModel();  
        System.out.println("TV종류:" + tvModel);  
  
        Product<Car, String> product2 = new Product<Car, String>();  
        product2.setKind(new Car());  
        product2.setModel("디젤");  
        Car car = product2.getKind();  
        String carModel = product2.getModel();  
        System.out.println("자동차종류:" + carModel);  
    }  
}
```

[실행결과]

TV 종류:스마트TV
자동차 종류:디젤

13.4 제네릭 메서드 (<T, R> R method(T t))

제네릭 메서드는 매개 타입과 리턴 타입으로 타입 파라미터를 갖는 메서드를 말한다. 제네릭 메서드를 선언하는 방법은 리턴 타입 앞에 <> 기호를 추가하고 타입 파라미터를 기술한 다음, 리턴타입과 매개 타입으로 타입 파라미터를 사용하면 된다. 아래 예제는 Util 클래스에 정적 제네릭 메서드로 boxing()을 정의하고 Sample 클래스에서 호출했다.

[Box.java] 제네릭 타입

```
public class Box<T> {  
    private T t;  
    public T get() { return t; }  
    public void set(T t) { this.t = t; }  
}
```

[Util.java] 제네릭 메서드

```
public class Util {  
    public static <T> Box<T> boxing(T t){  
        Box<T> box = new Box<T>();  
        box.set(t);  
        return box;  
    }  
}
```

[Sample.java] 제네릭 메서드 호출

```
public class Sample {
    public static void main(String[] args) {
        Box<Integer> box1 = Util.<Integer>boxing(100); //명시적으로 구체적 타입을 지정
        int intValue = box1.get();
        System.out.println("intValue=" + intValue);

        Box<String> box2 = Util.boxing("홍길동"); //매개값을 보고 구체적 타입을 지정
        String strValue = box2.get();
        System.out.println("strValue=" + strValue);
    }
}
```

[실행결과]

```
intValue=100
strValue=홍길동
```

아래 예제는 Util 클래스에 정적(static) 제네릭 메서드로 compare()를 정의하고 Sample 클래스에서 호출했다. 타입 파라미터는 K와 V로 선언되었는데, 제네릭 타입 Pair가 K와 V를 가지고 있기 때문이다. compare() 메서드는 두 개의 Pair를 매개값으로 받아 K와 V값이 동일한지 검사하고 boolean 값을 리턴 하는 코드이다.

[Util.java] 제네릭 메서드

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        boolean keyCompare = p1.getKey().equals(p2.getKey());
        boolean valueCompare = p1.getValue().equals(p2.getValue());
        return keyCompare && valueCompare;
    }
}
```

[Pair.java] 제네릭 타입

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

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

    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey() { return key; }
    public V getValue() { return value; }
}
```

[Sample.java] 제네릭 메서드 호출

```
public class Sample {
    public static void main(String[] args) {
        Pair<Integer, String> p1=new Pair<Integer, String>(1, "사과");
        Pair<Integer, String> p2=new Pair<Integer, String>(1, "사과");
        //구체적인 타입을 명시적으로 지정
        boolean result1=Util.<Integer, String>compare(p1, p2);
        if(result1){
            System.out.println("논리적으로 동등한 객체입니다.");
        }else{
            System.out.println("논리적으로 동등하지 않는 객체입니다.");
        }

        Pair<String, String> p3=new Pair<String, String>("user1", "홍길동");
        Pair<String, String> p4=new Pair<String, String>("user2", "홍길동");
        //구체적인 타입을 추정
        boolean result2 = Util.compare(p3, p4);
        if(result2) {
            System.out.println("논리적으로 동등한 객체입니다.");
        } else {
            System.out.println("논리적으로 동등하지 않는 객체입니다.");
        }
    }
}
```

[실행결과]

```
논리적으로 동등한 객체입니다.
논리적으로 동등하지 않는 객체입니다.
```

13.5 제한된 타입 파라미터(<T extends 최상위타입>)

숫자를 연산하는 제네릭 메서드는 매개값으로 Number 타입 또는 하위 클래스 타입(Byte, Short, Integer, Long, Double)의 인스턴스만 가져야하므로 이것이 제한된 타입 파라미터가 필요한 이유이다. 제한된 타입 파라미터를 선언하려면 타입 파라미터 뒤에 extends 키워드를 붙이고 상위 타입을 명시하면 된다. 상위 타입은 클래스뿐만 아니라 인터페이스도 가능하다. 인터페이스라고 해서 implements를 사용하지 않는다.

아래 예제는 숫자 타입만 구체적인 타입으로 갖는 제네릭 메서드 compare()이다. doubleValue() 메서드는 Number 클래스에 정의되어 있는 메서드로 숫자를 double 타입으로 변환한다. Double.compare() 메서드는 첫 번째 매개값이 작으면 -1을, 같으면 0을, 크면 1을 리턴 한다.

[Util.java] 제네릭 메서드

```
public class Util {
    public static <T extends Number> int compare(T t1, T t2) {
        double v1 = t1.doubleValue();
        double v2 = t2.doubleValue();
        return Double.compare(v1, v2);
    }
}
```

[Sample.java] 제네릭 메서드 호출

```
public class Sample {
    public static void main(String[] args) {
        //String str=Util.compare("a", "b") String은 Number 타입이 아님 (X)
        int result1 = Util.compare(10, 20); //int → Integer(자동 Boxing)
        System.out.println("result1=" + result1);

        int result2 = Util.compare(4.5, 3); //double Double(자동 Boxing)
        System.out.println("result2=" + result2);
    }
}
```

13.6 와일드카드 타입(<?>, <? extends ...>, <? super ...>)

코드에서 ?를 일반적으로(wildcard)라고 부른다. 제네릭 타입을 매개값이나 리턴 타입으로 사용할 때 구체적인 타입 대신에 와일드카드를 다음과 같이 세 가지 형태로 사용할 수 있다.

- 제네릭타입<?> (제한없음): 타입 파라미터를 대체하는 구체적인 타입으로 모든 클래스나 인터페이스 타입이 올 수 있다.
- 제네릭타입<? extends 상위타입> (상위 클래스 제한): 타입 파라미터를 대체하는 구체적인 타입으로 상위 타입이나 하위 타입만 올 수 있다.
- 제네릭타입<? super 하위타입> (하위 클래스 제한): 타입 파라미터를 대체하는 구체적인 타입으로 하위 타입이나 상위 타입이 올 수 있다.

Course는 과정 클래스로 과정 이름과 수강생을 지정할 수 있는 배열을 가지고 있다. 타입 파라미터 T가 적용된 곳은 수강생 타입 부분이다.

[Course.java] 제네릭 타입

```
public class Course<T> {
    private String name;
    private T[] students;

    public Course(String name, int capacity) {
        this.name = name;
        //타입 파라미터로 배열을 생성하려면 (T[]) (new Object[n])으로 생성해야 한다.
        students = (T[])(new Object[capacity]);
    }

    public String getName() { return name; }
    public T[] getStudents() { return students; }

    public void add(T t) {
        for(int i=0; i<students.length; i++) {
            if(students[i] == null) {
                students[i] = t;
                break;
            }
        }
    }
}
```

//배열에 비어있는 부분을 찾아서 수강생을 추가하는 메서드

Person의 하위 클래스로 Worker 클래스와 Student 클래스를, Student의 하위 클래스로 HighStudent 클래스를 정의하였다.

[Person.java] Person 클래스

```
public class Person {
    private String name;
    public Person(String name) { this.name = name; }
    public String getName() { return name; }
    public String toString() { return name; }
}
```

[Worker.java] Worker 클래스

```
public class Worker extends Person {
    public Worker(String name) {
        super(name);
    }
}
```

[Student.java] Student 클래스

```
public class Student extends Person {
    public Student(String name) {
        super(name);
    }
}
```

[HighStudent.java] Student 클래스

```
public class HighStudent extends Student {
    public HighStudent(String name) {
        super(name);
    }
}
```

아래 예제는 registerCourseXXX() 메서드의 매개값으로 와일드카드 타입을 사용하였다. registerCourse()는 모든 수강생이 들을 수 있는 과정을, registerCoursesStudent()는 학생만 들을 수 있는 과정을, registerCoursesWorker는 직장인만 들을 수 있는 과정을 등록한다.

[Sample.java] 와일드카드 타입 매개 변수

```
public class Sample {
    public static void registerCourse( Course<?> course ) { //모든 과정
        System.out.println(course.getName() + "수강생:" + Arrays.toString(course.getStudents()));
    }
    public static void registerCourseStudent( Course<? extends Student> course ) { //학생 과정
        System.out.println(course.getName() + "수강생:" + Arrays.toString(course.getStudents()));
    }
    public static void registerCourseWorker( Course<? super Worker> course ) { //직장인과 일반인 과정
        System.out.println(course.getName() + "수강생:" + Arrays.toString(course.getStudents()));
    }
    public static void main(String[] args) {
        Course<Person> personCourse = new Course<Person>("일반과정", 5);
        personCourse.add(new Person("일반인"));
        personCourse.add(new Worker("직장인"));
        personCourse.add(new Student("학생"));
        personCourse.add(new HighStudent("고등학생"));

        Course<Worker> workerCourse = new Course<Worker>("직장인과정", 5);
        workerCourse.add(new Worker("직장인"));

        Course<Student> studentCourse = new Course<Student>("학생과정", 5);
        studentCourse.add(new Student("학생"));
        studentCourse.add(new HighStudent("고등학생"));

        Course<HighStudent> highStudentCourse = new Course<HighStudent>("고등학생", 5);
        highStudentCourse.add(new HighStudent("고등학생"));

        registerCourse(personCourse);
        registerCourse(workerCourse);
        registerCourse(studentCourse);
        registerCourse(highStudentCourse);
        System.out.println();

        //registerCourseStudent(personCourse); 오류
        //registerCourseStudent(workerCourse); 오류

        registerCourseStudent(studentCourse);
        registerCourseStudent(highStudentCourse);
        System.out.println();

        registerCourseWorker(personCourse);
        registerCourseWorker(workerCourse);

        //registerCourseWorker(studentCourse); 오류
        //registerCourseWorker(highStudentCourse); 오류
    }
}
```

13.7 제네릭 타입의 상속과 구현

제네릭 타입도 다른 타입과 마찬가지로 부모 클래스가 될 수 있고 자식 제네릭 타입은 추가적으로 타입 파라미터를 가질 수 있다.

[Product.java] 부모 제네릭 클래스

```
public class Product<T, M> {
    private T kind;
    private M model;

    public T getKind() { return kind; }
    public M getModel() { return model; }
    public void setKind(T kind) { this.kind = kind; }
    public void setModel(M model) { this.model = model; }
}
```

[ChildProduct.java] 자식 제네릭 클래스

```
public class ChildProduct<T, M, C> extends Product<T, M> {
    private C company;

    public C getCompany() { return company; }
    public void setCompany(C company) { this.company = company; }
}
```

[Storage.java] 제네릭 인터페이스

```
public interface Storage<T> { //제네릭 인터페이스
    public void add(T item, int index);
    public T get(int index);
}
```

[StorageImpl.java] 제네릭 구현 클래스

```
public class StorageImpl<T> implements Storage<T> {
    private T[] array;

    public StorageImpl(int capacity) {
        this.array = (T[]) (new Object[capacity]); //타입 파라미터로 배열을 생성하려면 (T[])(new Object[n])으로 생성해야 한다.
    }

    @Override
    public void add(T item, int index) {
        array[index] = item;
    }

    @Override
    public T get(int index) {
        return null;
    }
}
```

[TV.java] TV 클래스

```
public class TV { }
```

[Sample.java] 제네릭 타입 사용 클래스

```
public class Sample {
    public static void main(String[] args) {
        ChildProduct<TV, String, String> product = new ChildProduct<>();
        product.setKind(new TV());
        product.setModel("SmartTV");
        product.setCompany("Samsung");

        Storage<TV> storage = new StorageImpl<TV>(100);
        storage.add(new TV(), 0);
        TV tv = storage.get(0);
    }
}
```


14장 람다식

14.1 람다식이란?

자바는 함수적 프로그래밍을 위해 자바 8부터 람다식(Lambda Expressions)을 지원하면서 기존의 코드 패턴이 많이 달라졌다. 람다식은 익명 함수(anonymous function)를 생성하기 위한 식으로 객체 지향 언어보다는 함수지향 언어에 가깝다. 자바에서 람다식을 수용한 이유는 자바 코드가 매우 간결해지고, 컬렉션의 요소를 필터링하거나 매핑해서 원하는 결과를 쉽게 집계할 수 있기 때문이다. 람다식의 형태는 매개 변수를 가진 코드 블록이지만, 런타임 시에는 익명 구현 객체를 생성한다.

예를 들어 Runnable 인터페이스의 익명 구현 객체를 생성하는 전형적인 코드는 아래와 같다.

```
Runnable runnable = new Runnable() { //익명 구현 객체
    public void run() { ... }
};
```

위 코드에서 익명 구현 객체를 람다식으로 표현하면 아래와 같다.

```
Runnable runnable = () -> { ... }; //람다식
```

람다식은 마치 함수 정의 형태를 띠고 있지만 런타임 시에 인터페이스의 익명 구현 객체로 생성된다. 어떤 인터페이스를 구현할 것인가는 대입되는 인터페이스가 무엇이나에 달려있다. 위 코드는 Runnable 변수에 대입되므로 람다식 Runnable의 익명 구현 객체를 생성하게 된다.

14.2 람다식 기본 문법

함수적 스타일의 람다식을 작성하는 방법은 아래와 같다.

```
(타입 매개변수, ...) -> { 실행문: ... }
```

(타입 매개변수, ...)는 오른쪽 중괄호 { } 블록을 실행하기 위해 필요한 값을 제공하는 역할을 한다. 매개 변수의 이름은 개발자가 자유롭게 줄 수 있다. -> 기호는 매개 변수를 이용해서 중괄호 { }를 실행한다는 뜻으로 해석하면 된다. int a의 값을 콘솔에 출력하기 위한 람다식은 아래와 같다.

```
(int a) -> { System.out.println(a); }
```

14.3 타겟 타입과 함수적 인터페이스

람다식의 형태는 매개 변수를 가진 코드 블록이기 때문에 마치 자바의 메서드를 선언하는 것처럼 보여진다. 자바는 메서드를 단독으로 선언할 수 없고 항상 클래스의 구성 멤버로 선언하기 때문에 람다식은 단순히 메서드를 선언하는 것이 아니라 이 메서드를 가지고 있는 객체를 생성해 낸다.

```
인터페이스 변수 = 람다식;
```

람다식은 인터페이스 변수에 대입된다. 이 말은 람다식은 인터페이스의 익명 구현 객체를 생성하며 인터페이스는 직접 객체화할 수 없기 때문에 구현 클래스가 필요한데, 람다식은 익명 구현 클래스를 생성하고 객체화한다. 람다식은 대입될 인터페이스의 종류에 따라 작성 방법이 달라지기 때문에 람다식이 대입될 인터페이스를 람다식의 **타겟 타입(target type)**이라고 한다.

14.3.1 함수적 인터페이스 (@FunctionalInterface)

모든 인터페이스를 람다식의 타겟 타입으로 사용할 수는 없다. 람다식이 하나의 메서드를 정의하기 때문에 두 개 이상의 추상 메서드가 선언된 인터페이스는 람다식을 이용해서 구현 객체를 생성할 수 없다. 하나의 추상 메서드가 선언된 인터페이스만이 람다식의 타겟 타입이 될 수 있는데, 이러한 인터페이스를 **함수적 인터페이스(functional interface)**라고 한다. 함수적 인터페이스를 작성할 때 두 개 이상의 추상 메서드가 선언되지 않도록 컴파일러가 체크해주는 기능이 있는데, 인터페이스 선언시 @FunctionalInterface 어노테이션을 붙이면 된다.

```
@FunctionalInterface
public interface MyFunctionalInterface{
    public void method();
    public void otherMethod(); //컴파일 오류
}
```

@FunctionalInterface 어노테이션은 선택사항이다. 이 어노테이션이 없더라도 하나의 추상 메서드만 있다면 모두 함수적 인터페이스이다. 그러나 실수로 두 개 이상의 추상 메서드를 선언하는 것을 방지하고 싶다면 붙여주는 것이 좋다.

14.3.2 매개 변수와 리턴값이 없는 람다식

아래와 같이 매개 변수와 리턴값이 없는 추상 메서드를 가진 함수적 인터페이스가 있다고 가정해 보자.

[MyFunctionalInterface.java] 함수적 인터페이스

```
@FunctionalInterface
public interface MyFunctionalInterface {
    public void method();
}
```

이 인터페이스를 타겟 타입으로 갖는 람다식은 다음과 같다. 람다식에서 매개 변수가 없는 이유는 method()가 매개 변수를 가지지 않기 때문이다.

MyFunctionalInterface fi = () -> { ... }

람다식이 대입된 인터페이스의 참조 변수는 fi.method로 method()를 호출할 수 있다. method()호출은 람다식의 중괄호 { }를 실행시킨다.

[Sample.java] 람다식

```
public class Sample {
    public static void main(String[] args) {
        MyFunctionalInterface fi;

        fi = () -> {
            String str = "method call1";
            System.out.println(str);
        };

        fi.method();
        fi = () -> { System.out.println("method call2"); };
        fi.method();
        fi = () -> { System.out.println("method call3"); }; //실행문이 하나라면 중괄호 { }는 생략가능
        fi.method();
    }
}
```

[실행결과]

```
method call1
method call2
method call3
```

14.3.3 매개 변수가 있는 람다식

아래와 같이 매개 변수가 있고 리턴값이 없는 추상 메서드를 가진 함수적 인터페이스가 있다고 보자.

[MyFunctionalInterface.java] 함수적 인터페이스

```
@FunctionalInterface
public interface MyFunctionalInterface {
    public void method(int x);
}
```

이 인터페이스를 타겟 타입으로 갖는 람다식은 다음과 같다. 람다식에서 매개 변수가 없는 이유는 method()가 매개 변수를 가지지 않기 때문이다.

MyFunctionalInterface fi = (x) -> { ... } 또는 x -> { ... }

람다식이 대입된 인터페이스 참조 변수는 fi.method(5)와 같이 method()를 호출할 수 있다. 매개값으로 5를 주면 람다식 x 변수에 5가 대입되고 x는 중괄호 { }에서 사용된다.

[Sample.java] 람다식

```
public class Sample {
    public static void main(String[] args) {
        MyFunctionalInterface fi;

        fi = (x) -> {
            int result = x * 5;
            System.out.println(result);
        };

        fi.method(2);
        fi = (x) -> { System.out.println(x * 5); };
        fi.method(2);
        fi = x -> System.out.println(x * 5); //매개 변수가 하나일 경우에는 괄호()를 생략할 수 있다.
        fi.method(2);
    }
}
```

[실행결과]

```
10
10
10
```

14.3.4 리턴값이 있는 람다식

아래와 같이 매개 변수가 있고 리턴값이 있는 추상 메서드를 가진 함수적 인터페이스가 있다고 보자.

[MyFunctionalInterface.java] 함수적 인터페이스

```
@FunctionalInterface
public interface MyFunctionalInterface {
    public int method(int x, int y);
}
```

이 인터페이스를 타겟 타입으로 갖는 람다식은 다음과 같은 형태로 작성해야 한다. 람다식에서 매개 변수가 두 개인 이유는 method()가 매개 변수를 두 개 가지기 때문이다. 그리고 method()가 리턴 타입이 있기 때문에 중괄호 { }에는 return문이 있어야 한다.

MyFunctionalInterface fi = (x, y) -> { ...; return 값; }

람다식이 대입된 인터페이스 참조 변수는 다음과 같이 method()를 호출할 수 있다. 매개값으로 2와 5를 주면 람다식의 x변수에 2, y변수에 5가 대입되고 x와 y는 중괄호 { }에서 사용된다.

int result = fi.method(2, 5);

[Sample.java] 람다식

```
public class Sample {
    public static void main(String[] args) {
        MyFunctionalInterface fi;
        fi = (x, y) -> {
            int result = x + y;
            return result;
        };
        System.out.println(fi.method(2, 5));

        fi = (x, y) -> {return x + y;};
        System.out.println(fi.method(2, 5));

        //return문만 있을 경우 중괄호 { }와 return문 생략 가능
        fi = (x, y) -> {x + y;};
        System.out.println(fi.method(2, 5));

        //return문만 있을 경우 중괄호 { }와 return문 생략 가능
        fi = (x, y) -> {sum(x, y);};
        System.out.println(fi.method(2, 5));
    }
    public static int sum(int x, int y){
        return (x + y);
    }
}
```

[실행결과]

7
7
7
7

14.4 클래스 멤버와 로컬 변수 사용

람다식의 실행 블록에는 클래스의 멤버(필드와 메서드) 및 로컬 변수를 사용할 수 있다. 클래스의 멤버는 제약 사항 없이 사용 가능하지만, 로컬 변수는 제약 사항이 따른다. 자세한 내용을 알아보기로 하자.

14.4.1 클래스의 멤버 사용

람다식 실행 블록에는 클래스의 멤버인 필드와 메서드를 제약 사항 없이 사용할 수 있다. 하지만 this 키워드를 사용할 때에는 주의가 필요하다. 일반적으로 익명 객체 내부에서 this는 익명 객체의 참조이지만, 람다식에서 this는 내부적으로 생성되는 익명 객체의 참조가 아니라 람다식을 실행한 객체의 참조이다.

다음 예제는 람다식에서 바깥 객체와 중첩 객체의 참조를 얻어 필드값을 출력하는 방법을 보여주고 있다. 중첩 객체 Inner에서 람다식을 실행했기 때문에 람다식 내부에서의 this는 중첩 객체 Inner이다.

[MyFunctionalInterface.java] 함수적 인터페이스

```
@FunctionalInterface
public interface MyFunctionalInterface {
    public int method();
}
```

[UsingThis.java] this 사용

```
public class UsingThis {
    public int outterField = 10;
    class Inner{
        int innerField = 20;
        void method(){
            MyFunctionalInterface fi = () -> {
                System.out.println("outterField:" + outterField);
                //바깥 객체의 참조를 얻기 위해서는 클래스명.this를 사용
                System.out.println("outterField:" + UsingThis.this.outterField + "\n");
                //람다식 내부에서 this는 inner 객체를 참조
                System.out.println("innerField:" + innerField);
                System.out.println("innerField:" + this.innerField + "\n");
            };
            fi.method();
        }
    }
}
```

[Sample.java] 실행 클래스

```
public class Sample {
    public static void main(String[] args) {
        UsingThis usingThis = new UsingThis();
        UsingThis.Inner inner = usingThis.new Inner();
        inner.method();
    }
}
```

[실행결과] outterField:10 outterField:10 innerField:20 innerField:20

14.4.2 로컬 변수 사용

람다식은 메서드 내부에서 주로 작성되기 때문에 로컬 익명 구현 객체를 생성시킨다고 봐야 한다. 람다식에서 바깥 클래스의 필드나 메서드는 제한 없이 사용할 수 있으나, 메서드의 매개 변수 또는 로컬 변수를 사용하면 이 두 변수는 final 특성을 가져야 한다. 따라서 매개 변수 또는 로컬 변수를 람다식에서 읽는 것은 허용되지만, 람다식 내부 또는 외부에서 변경할 수 없다.

[MyFunctionalInterface.java] 함수적 인터페이스

```
@FunctionalInterface
public interface MyFunctionalInterface {
    public int method();
}
```

[UsingLocalVariable.java] Final 특성을 가지는 로컬 변수

```
public class UsingLocalVariable {
    void method(int arg){ //arg는 final 특성을 가짐
        int localVar = 40; //localVar는 final 특성을 가짐
```

```
//arg = 31; final 특성 때문에 수정 불가
//localVar = 41; final 특성 때문에 수정 불가
```

```
//람다식
```

```
MyFunctionalInterface fi = () -> {
```

```
//로컬 변수 읽기
```

```
System.out.println("arg:" + arg);
System.out.println("localVar:" + localVar + "\n");
```

```
};
fi.method();
}
```

[Sample.java] 실행 클래스

```
public class Sample {
    public static void main(String[] args) {
        UsingLocalVariable ulv = new UsingLocalVariable();
        ulv.method(20);
    }
}
```

[실행결과] arg:20 localVar:40
--

14.5 표준 API의 함수적 인터페이스

자바에서 제공되는 표준 API에서 한 개의 추상 메서드를 가지는 인터페이스들은 모두 람다식을 이용해서 익명 구현 객체로 표현이 가능하다. 예를 들어 스레드의 작업을 정의하는 Runnable 인터페이스는 매개 변수와 리턴값이 없는 run() 메서드만 존재하기 때문에 아래와 같이 람다식을 이용해서 Runnable 인스턴스를 생성시킬 수 있다.

[Sample.java] 함수적 인터페이스와 람다식

```
public class Sample {
    public static void main(String[] args) {
        Runnable runnable = () -> {
            for(int i=0; i<10; i++){
                System.out.println(i);
            }
        };

        Thread thread = new Thread(runnable);
        thread.start();
    }
}
```

//람다식(스레드가 실행하는 코드)

[실행결과]

0
1
2
3
4
5
6
7
8
9

Thread 생성자를 호출할 때 아래와 같이 람다식을 매개값으로 대입해도 된다.

```
Thread thread = new Thread() -> {
    for(int i=0; i<10; i++) {
        System.out.println(i);
    }
});
```

14.5.1 Consumer 함수적 인터페이스

Consumer의 특징은 리턴값이 없는 accept() 메서드를 가지고 있다. accept() 메서드는 리턴값이 없이 매개값을 소비하는 역할만 한다.

[Sample.java] Consumer 함수적 인터페이스

```
public class Sample {
    public static void main(String[] args) {
        Consumer<String> consumer = t -> System.out.println(t + "8");
        consumer.accept("java");

        BiConsumer<String, String> bigConsumer = (t, u) -> System.out.println(t + u);
        bigConsumer.accept("java", "8");

        DoubleConsumer doubleConsumer = d -> System.out.println("java" + d);
        doubleConsumer.accept(8.0);

        ObjIntConsumer<String> objIntConsumer = (t, i) -> System.out.println(t + i);
        objIntConsumer.accept("Java", 8);
    }
}
```

[실행결과]

java8
java8
java8.0
Java8

14.5.2 Supplier 함수적 인터페이스

Supplier 함수적 인터페이스의 특징은 매개 변수가 없고 리턴값이 있는 getXXX() 메서드를 가지고 있다. 이 메서드들은 실행 후 호출한 곳으로 데이터 리턴(공급)하는 역할을 한다.

[Sample.java] Supplier 함수적 인터페이스

```
public class Sample {
    public static void main(String[] args) {
        IntSupplier intSupplier = () -> {
            int num = (int) (Math.random() * 6) + 1;
            return num;
        };

        int num = intSupplier.getAsInt();
        System.out.println("눈의 수: " + num);
    }
}
```

//람다식

[실행결과]

눈의 수:4

14.5.3 Function 함수적 인터페이스

다음 예제는 list에 저장된 학생 객체를 꺼내서 이름과 점수를 출력한다. printString()는 Function<Student, String> 매개 변수를 가지고 있고, printInt()는 ToIntFunction<Student> 매개 변수를 가지고 있으므로 이 메서드들을 호출할 때 람다식을 사용할 수 있다.

[Sample.java] Function 함수적 인터페이스

```
public class Sample {
    private static List<Student> list = Arrays.asList(new Student("홍길동", 90, 96), new Student("신용권", 95, 93));

    public static void printString(Function<Student, String> function) {
        for(Student student : list) { //list에 저장된 항목 수만큼 루핑
            System.out.println(function.apply(student) + " ");
        }
        System.out.println();
    }

    public static void printInt(ToIntFunction<Student> function) {
        for(Student student : list) { //list에 저장된 항목 수만큼 루핑
            System.out.println(function.applyAsInt(student) + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        System.out.println("[학생 이름]");
        printString(t -> t.getName());
        System.out.println("[영어 점수]");
        printInt(t -> t.getEnglishScore());
        System.out.println("[수학 점수]");
        printInt(t -> t.getMathScore());
    }
}
```

[실행결과]
[학생 이름]
홍길동
신용권
[영어 점수]
90
95
[수학 점수]
96
93

[Student.java] Student 클래스

```
public class Student {
    private String name;
    private int englishScore;
    private int mathScore;

    public Student(String name, int englishScore, int mathScore) {
        this.name = name;
        this.englishScore = englishScore;
        this.mathScore = mathScore;
    }

    public String getName() { return name; }
    public int getEnglishScore() { return englishScore; }
    public int getMathScore() { return mathScore; }
}
```

아래 예제는 List 저장된 학생 객체를 꺼내서 영어 점수와 수학 점수의 평균값을 산출한다. avg() 메서드는 ToIntFunction<Student> 매개 변수를 가지고 있다. 따라서 avg() 메서드를 호출할 때 매개값으로 람다식을 사용할 수 있다. 람다식 getEnglishScore()와 getMathScore()를 호출해서 영어 점수와 수학 점수로 Student 객체를 매핑(변환)시킨다.

[Sample.java] Function 함수적 인터페이스

```
public class Sample {
    private static List<Student> list = Arrays.asList(new Student("홍길동", 90, 96), new Student("신용권", 95, 93));

    public static double avg(ToIntFunction<Student> function) {
        int sum = 0;
        for(Student student:list) {
            sum += function.applyAsInt(student);
        }
        double avg = (double)sum/list.size();
        return avg;
    }

    public static void main(String[] args) {
        double englishAvg = avg(s -> s.getEnglishScore());
        System.out.println("영어 평균 점수:" + englishAvg);
        double mathAvg = avg(s -> s.getMathScore());
        System.out.println("수학 평균 점수:" + mathAvg);
    }
}
```

[실행결과]
영어 평균 점수:92.5
수학 평균 점수:94.5

14.5.4 Operator 함수적 인터페이스

Operator 함수적 인터페이스는 Function과 동일하게 매개변수와 리턴값이 있는 applyXXX() 메서드를 가지고 있다. 하지만 이 메서드들은 매개값을 리턴값으로 매핑(타입 변환)하는 역할보다는 매개값을 이용해서 연산을 수행한 후 동일한 타입으로 리턴값을 제공하는 역할을 한다.

[Sample.java] Operator 함수적 인터페이스

```
public class Sample {
    private static int[] scores = {92, 95, 87};

    public static int maxOrMin(IntBinaryOperator operator) {
        int result = scores[0];

        for(int score:scores){
            result = operator.applyAsInt(result, score); //람다식 | | 실행
        }
        return result;
    }

    public static void main(String[] args) {
        int max = maxOrMin(
            (a, b) -> {
                if(a>=b) return a;
                else return b;
            }
        );
        System.out.println("최대값:" + max);

        int min = maxOrMin(
            (a, b) -> {
                if(a<=b) return a;
                else return b;
            }
        );
        System.out.println("최소값:" + min);
    }
}
```

[실행결과]
최대값:95
최소값:87

14.5.5 Predicate 함수적 인터페이스

Predicate 함수적 인터페이스는 매개 변수와 boolean 리턴값이 있는 testXXX() 메서드를 가지고 있다. 이 메서드들은 매개값을 조사해서 true 또는 false를 리턴 하는 역할을 한다. 다음 예제는 List에 저장되어 있는 남자 또는 여자 학생들의 평균 점수를 출력한다. avg() 메서드는 Predicate<Student> 매개 변수를 가지고 있다. 따라서 avg() 메서드를 호출할 때 매개값으로 람다식을 사용할 수 있다.

[Sample.java] Predicate 함수적 인터페이스

```
public class Sample {
    private static List<Student> list = Arrays.asList(
        new Student("홍길동", "남자", 90),
        new Student("김순희", "여자", 90),
        new Student("김자바", "남자", 95),
        new Student("박한나", "여자", 92)
    );

    public static double avg(Predicate<Student> predicate) {
        int count = 0, sum = 0;

        for(Student student:list){
            if(predicate.test(student)){
                count++;
                sum += student.getScore();
            }
        }
        return (double)sum/count;
    }

    public static void main(String[] args) {
        double maleAvg = avg(t -> t.getSex().equals("남자"));
        System.out.println("남자 평균 점수:" + maleAvg);

        double femaleAvg = avg(t -> t.getSex().equals("여자"));
        System.out.println("여자 평균 점수:" + femaleAvg);
    }
}
```

[실행결과]
남자 평균 점수:92.5
여자 평균 점수:91.0

[Students.java] Student 클래스

```
public class Student {
    private String name;
    private String sex;
    private int score;

    public Student(String name, String sex, int score) {
        this.name = name;
        this.sex = sex;
        this.score = score;
    }
    public String getName() { return name; }
    public String getSex() { return sex; }
    public int getScore() { return score; }
}
```

14.5.6 andThen()과 compose() 디폴트 메서드

Consumer, Function, Operator의 함수적 인터페이스는 andThen()과 compose()를 가지고 있다. andThen()과 compose() 디폴트 메서드는 두 개의 함수적 인터페이스를 순차적으로 연결하고, 첫 번째 처리 결과를 두 번째 매개값으로 제공해서 최종 결과값을 얻을 때 사용한다.

아래 예제는 Consumer<Member> 함수적 인터페이스 두 개를 순차적으로 연결해서 실행한다. 첫 번째 Consumer<Member>는 이름을 출력하고, 두 번째 Consumer<Member>는 아이디를 출력한다.

[Sample.java] Consumer의 순차적 연결

```
public class Sample {
    public static void main(String[] args) {
        Consumer<Member> consumerA = (m) -> {
            System.out.println("consumerA:" + m.getName());
        };

        Consumer<Member> consumerB = (m) -> {
            System.out.println("consumerB:" + m.getId());
        };

        Consumer<Member> consumerAB = consumerA.andThen(consumerB);
        consumerAB.accept(new Member("홍길동", "hong", null));
    }
}
```

[실행결과]

```
consumerA:홍길동
consumerB:hong
```

[Member.java] 회원 클래스

```
public class Member {
    private String name;
    private String id;
    private Address address;

    public Member(String name, String id, Address address) {
        this.name = name;
        this.id = id;
        this.address = address;
    }
    public String getName() { return name; }
    public String getId() { return id; }
    public Address getAddress() { return address; }
}
```

[Address.java] 주소 클래스

```
public class Address {
    private String country;
    private String city;
    public Address(String country, String city) {
        this.country = country;
        this.city = city;
    }
    public String getCountry() { return country; }
    public String getCity() { return city; }
}
```


Function과 Operator 종류의 함수적 인터페이스는 먼저 실행한 함수적 인터페이스의 결과를 다음 함수적 인터페이스의 매개값으로 넘겨주고, 최종 처리 결과를 리턴 한다. 아래 예제는 Member 객체의 필드인 Address에서 city 정보를 얻어내기 위해 두 Function 함수적 인터페이스를 addThen()과 compose()를 이용해서 순차적으로 연결했다.

[Sample.java] Function의 순차적 연결

```
public class Sample {
    public static void main(String[] args) {
        Function<Member, Address> functionA;
        Function<Address, String> functionB;
        Function<Member, String> functionAB;

        String city;
        functionA = (m) -> m.getAddress();
        functionB = (a) -> a.getCity();
        functionAB = functionA.andThen(functionB);
        city = functionAB.apply(
            new Member("홍길동", "hong", new Address("한국", "서울"))
        );
        System.out.println("거주 도시:" + city);

        functionAB = functionB.compose(functionA);
        city = functionAB.apply(
            new Member("홍길동", "hong", new Address("한국", "서울"))
        );
        System.out.println("거주 도시:" + city);
    }
}
```

[실행결과]

거주 도시:서울
거주 도시:서울

14.5.7 and(), or(), negate() 디폴트 메서드와 isEqual() 정적 메서드

Predicate 종류의 함수적 인터페이스 and(), or(), negate() 디폴트 메서드를 가지고 있다. 이 메서드들은 각각 논리 연산자인 &&, ||, !과 대응된다고 볼 수 있다. 아래 예제는 2의 배수와 3의 배수를 조사하는 두 Predicate를 논리 연산한 새로운 Predicate를 생성한다.

[Sample.java] Predicate 간의 논리 연산

```
public class Sample {
    public static void main(String[] args) {
        IntPredicate predicateA = a -> a%2 == 0; //2의 배수 검사
        IntPredicate predicateB = (a) -> a%3 == 0; //3의 배수 검사
        IntPredicate predicateAB;
        boolean result;

        predicateAB = predicateA.and(predicateB); //and()
        result = predicateAB.test(9);
        System.out.println("9는 2와 3의 배수입니까?" + result);
        predicateAB = predicateA.or(predicateB); //or()
        result = predicateAB.test(9);
        System.out.println("9는 2 또는 3의 배수입니까?" + result);
        predicateAB = predicateA.negate(); //negate()
        result = predicateAB.test(9);
        System.out.println("9는 홀수입니까?" + result);
    }
}
```

[실행결과]

9는 2와 3의 배수입니까?false
9는 2 또는 3의 배수입니까?true
9는 홀수입니까?true

[Sample.java] IsEqual() 정적 메서드

```
public class Sample {
    public static void main(String[] args) {
        Predicate<String> predicate;
        predicate = Predicate.isEqual(null);
        System.out.println("null,null:" + predicate.test(null));
        predicate = Predicate.isEqual("Java8");
        System.out.println("null,Java8:" + predicate.test(null));
        predicate = Predicate.isEqual(null);
        System.out.println("Java8,null:" + predicate.test("Java8"));
        predicate = Predicate.isEqual("Java9");
        System.out.println("Java8,Java8:" + predicate.test("Java8"));
        predicate = Predicate.isEqual(null);
        System.out.println("Java7,Java8:" + predicate.test("Java7"));
    }
}
```

[실행결과]

null,null:true
null,Java8:false
Java8,null:false
Java8,Java8:true
Java7,Java8:false

14.5.8 minBy(), maxBy() 정적 메서드

BinaryOperator<T>는 minBy()와 maxBy() 정적 메서드를 제공한다. 이 두 메서드는 매개값으로 제공되는 Comparator를 이용해서 최대 T와 최소 T를 얻는 BinaryOperator<T>를 리턴 한다.

[Sample.java] minBy(), maxBy() 정적 메서드

```
public class Sample {
    public static void main(String[] args) {
        BinaryOperator<Fruit> binaryOperator;
        Fruit fruit;
        binaryOperator = BinaryOperator.minBy((f1, f2)->Integer.compare(f1.price, f2.price));
        fruit = binaryOperator.apply(new Fruit("딸기", 6000), new Fruit("수박", 10000));
        System.out.println(fruit.name);
        binaryOperator = BinaryOperator.maxBy((f1, f2)->Integer.compare(f1.price, f2.price));
        fruit = binaryOperator.apply(new Fruit("딸기", 6000), new Fruit("수박", 10000));
        System.out.println(fruit.name);
    }
}
```

[실행결과]

딸기
수박

[Fruit.java]

```
public class Fruit {
    String name;
    int price;

    public Fruit(String name, int price) {
        this.name = name;
        this.price = price;
    }
}
```

14.6 메서드 참조

14.6.1 정적 메서드와 인스턴스 메서드 참조

정적(static) 메서드를 참조할 경우에는 클래스 이름 뒤에 :: 기호를 붙이고 정적 메서드 이름을 기술하면 된다.

[Calculator.java] 정적 및 인스턴스 메서드

```
public class Calculator {
    public static int staticMethod(int x, int y){
        return x + y;
    }

    public int instanceMethod(int x, int y){
        return x + y;
    }
}
```

//정적 메서드

//인스턴스 메서드

[Sample.java] 정적 및 인스턴스 메서드 참조

```
import java.util.function.IntBinaryOperator;
public class Sample {
    public static void main(String[] args) {
        IntBinaryOperator operator;

        //정적 메서드 참조
        operator = (x, y) -> Calculator.staticMethod(x, y);
        System.out.println("결과1:" + operator.applyAsInt(1, 2));
        operator = Calculator :: staticMethod;
        System.out.println("결과2:" + operator.applyAsInt(3, 4));

        //인스턴스 메서드 참조
        Calculator obj = new Calculator();
        operator = (x, y) -> obj.instanceMethod(x, y);
        System.out.println("결과3:" + operator.applyAsInt(5, 6));
        operator = obj :: instanceMethod;
        System.out.println("결과4:" + operator.applyAsInt(7, 8));
    }
}
```

[실행결과]

결과1:3
결과2:7
결과3:11
결과4:15

14.6.2 매개 변수의 메서드 참조

아래 예제는 두 문자열이 대소문자와 상관없이 동일한 알파벳으로 구성되어 있는지 비교한다. 비교를 위해 사용된 메서드는 String의 인스턴스 메서드인 compareToIgnoreCase()이다. a.compareToIgnoreCase(b)로 호출될 때 a가 b보다 먼저 오면 음수를, 동일하면 0을, 나중에 오면 양수를 리턴한다. 사용된 함수적 인터페이스는 두 String 매개값을 받고 int 값을 리턴하는 ToIntBiFunction<String,String>이다.

[Sample.java] 매개 변수의 메서드 참조

```
import java.util.function.ToIntBiFunction;

public class Sample {
    public static void main(String[] args) {
        ToIntBiFunction<String, String> function;

        function = (a, b) -> a.compareToIgnoreCase(b);
        print(function.applyAsInt("Java8", "JAVA8"));
        function = String::compareToIgnoreCase;
        print(function.applyAsInt("Java8", "JAVA8"));
    }

    public static void print(int order){
        if(order < 0) { System.out.println("동일한 문자열입니다."); }
        else if(order == 0) { System.out.println("동일한 문자열입니다."); }
        else { System.out.println("자전순으로 나중에 올립니다."); }
    }
}
```

[실행결과]

동일한 문자열입니다.
동일한 문자열입니다.

14.6.3 생성자 참조

아래 예제는 생성자 참조를 이용해서 두 가지 방법으로 Member 객체를 생성한다. 하나는 Function<String, Member> 함수적 인터페이스의 Member apply(String) 메서드를 이용해서 Member 객체를 생성하였고, 다른 하나는 BiFunction<String, String, Member> 함수적 인터페이스의 Member apply(String, String) 메서드를 이용해서 Member 객체를 생성하였다.

[Sample.java] 생성자 참조

```
import java.util.function.BiFunction;
import java.util.function.Function;

public class Sample {
    public static void main(String[] args) {
        Function<String, Member> function1 = Member::new;
        Member member1 = function1.apply("angle"); //매개값 1개

        BiFunction<String, String, Member> function2 = Member::new;
        Member member2 = function2.apply("신천사", "angel"); //매개값 2개
    }
}
```

[실행결과]

Member(String id) 실행
Member(String name, String id)

[Member.java] 생성자 오버로딩

```
public class Member {
    private String name;
    private String id;

    public Member(){
        System.out.println("Member() 실행");
    }

    public Member(String id){
        System.out.println("Member(String id) 실행");
        this.id = id;
    }

    public Member(String name, String id){
        System.out.println("Member(String name, String id)");
        this.name = name;
        this.id = id;
    }

    public String getId() { return id; }
}
```

15장 컬렉션 프레임워크

15.1 컬렉션 프레임워크 소개

자바는 배열의 이러한 문제점을 해결하고, 널리 알려져 있는 자료구조(Data Structure)를 바탕으로 객체들을 효율적으로 추가, 삭제, 검색할 수 있도록 java.util 패키지에 컬렉션과 관련된 인터페이스와 클래스들을 포함시켜 놓았다. 이들을 총칭해서 컬렉션 프레임워크라고 부른다. 자바 컬렉션 프레임워크는 몇 가지 인터페이스(List, Set, Map)를 통해서 다양한 컬렉션 클래스를 이용할 수 있도록 하고 있다.

15.2 List 컬렉션

List 컬렉션은 객체를 일렬로 늘어놓은 구조를 가지고 있다. 객체를 인덱스로 관리하기 때문에 객체를 저장하면 자동 인덱스가 부여되고 인덱스로 객체를 검색, 삭제할 수 있는 기능을 제공한다. List 컬렉션은 객체 자체를 저장하는 것이 아니라 객체의 번지를 참조한다. 동일한 객체를 중복 저장할 수 있는데, 이 경우 동일한 번지가 참조된다. null도 저장 가능한데, 이 경우 해당 인덱스는 객체를 참조하지 않는다. List 컬렉션에는 ArrayList, Vector, LinkedList의 구현 클래스들이 있다.

15.2.1 ArrayList

아래 예제는 ArrayList에 객체의 인덱스를 이용하여 String 객체를 추가, 검색, 삭제하는 방법을 보여준다.

[Sample.java] String 객체를 저장하는 ArrayList

```
public class Sample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("Java"); //String 객체를 저장
        list.add("JDBC");
        list.add("Servlet/JSP");
        list.add(2, "Database");
        list.add("iBATIS");
        int size = list.size(); //저장된 총 객체 수 얻기
        System.out.println("총 객체수:" + size);
        System.out.println();

        String skill = list.get(2); //2번 인덱스의 객체 얻기
        System.out.println("2:" + skill);
        System.out.println();
        for(int i=0; i<list.size(); i++) { //저장된 총 객체 수만큼 루핑
            String str = list.get(i);
            System.out.println(i + ":" + str);
        }
        System.out.println();

        list.remove(2); //2번 인덱스 객체(Database) 삭제됨
        list.remove(2); //2번 인덱스 객체(Servlet/JSP) 삭제됨
        list.remove("iBATIS");
        for(int i=0; i<list.size(); i++){ //저장된 총 객체 수만큼 루핑
            String str = list.get(i);
            System.out.println(i + ":" + str);
        }
    }
}
```

[실행결과]

총 객체 수 : 5

2:Database

0:Java

1:JDBC

2:Database

3:Servlet/JSP

4:iBATIS

0:Java

1:JDBC

아래는 고정된 String 객체를 요소로 갖는 ArrayList 객체를 생성하기 위해 Arrays.asList(T ... a) 메서드를 사용한 예제이다.

[Sample.java] Arrays.asList() 메서드

```
public class Sample {
    public static void main(String[] args) {
        List<String> list1 = Arrays.asList("홍길동", "신용권", "감자바");
        for(String name:list1) {
            System.out.println(name);
        }

        List<Integer> list2 = Arrays.asList(1, 2, 3);
        for(int value:list2) {
            System.out.println(value);
        }
    }
}
```

[실행결과]

홍길동

신용권

감자바

1

2

3

15.2.2 Vector

Vector는 ArrayList와 동일한 내부 구조를 가지고 있다. Vector를 생성하기 위해서는 저장할 객체 타입을 타입 파라미터로 표기하고 기본 생성자를 호출하면 된다. ArrayList와 다른 점은 Vector는 동기화된 메서드로 구성되어 있기 때문에 멀티 스레드가 동시에 이 메서드들을 실행할 수 없고, 하나의 스레드가 실행을 완료해야만 다른 스레드를 실행할 수 있다. 그래서 멀티 스레드 환경에서 안정하게 객체를 추가, 삭제할 수 있다.

[Sample.java] Board 객체를 저장하는 Vector

```
public class Sample {
    public static void main(String[] args) {
        List<Board> list = new Vector<Board>();

        list.add(new Board("제목1", "내용1", "글쓴이1"));
        list.add(new Board("제목2", "내용2", "글쓴이2"));
        list.add(new Board("제목3", "내용3", "글쓴이3"));
        list.add(new Board("제목4", "내용4", "글쓴이4"));
        list.add(new Board("제목5", "내용5", "글쓴이5"));

        list.remove(2); //2번 인덱스 객체(제목3) 삭제(뒤의 인덱스는 1씩 앞으로 당겨짐)
        list.remove(3); //3번 인덱스 객체(제목5) 삭제

        for(int i = 0; i<list.size(); i++) {
            Board board = list.get(i);
            System.out.println(board.subject + "\t" + board.content + "\t" + board.writer);
        }
    }
}
```

//Board 객체를 저장

[실행결과]

제목1	내용1	글쓴이1
제목2	내용2	글쓴이2
제목4	내용4	글쓴이4

[Board.java] 게시물 정보 객체

```
public class Board {
    String subject;
    String content;
    String writer;
    public Board(String subject, String content, String writer) {
        this.subject = subject;
        this.content = content;
        this.writer = writer;
    }
}
```

15.2.3 LinkedList

LinkedList는 List 구현 클래스이므로 ArrayList와 사용 방법은 똑같지만 내부 구조는 완전 다르다. ArrayList는 내부 배열에 객체를 저장해서 인덱스로 관리하지만, LinkedList는 인접 참조를 링크해서 체인처럼 관리한다. 아래 예제는 ArrayList와 LinkedList에 10000개의 객체를 삽입하는데 걸릴 시간을 측정한 것이다. 0번 인덱스에 String 객체를 10000번 추가하기 위해 List 인터페이스의 add(int index, E element)메서드를 사용하였다. 실행 결과를 보면 LinkedList가 훨씬 빠른 성능을 낸다.

[Sample.java] ArrayList와 LinkedList의 실행 성능 비교

```
public class Sample {
    public static void main(String[] args) {
        List<String> list1 = new ArrayList<String>();
        List<String> list2 = new LinkedList<String>();
        long startTime;
        long endTime;
        startTime = System.nanoTime();
        for(int i=0; i<10000; i++) {
            list1.add(0, String.valueOf(i));
        }
        endTime = System.nanoTime();
        System.out.println("ArrayList 걸린시간:" + (endTime - startTime) + "ns");

        startTime = System.nanoTime();
        for(int i=0; i<10000; i++) {
            list2.add(0, String.valueOf(i));
        }
        endTime = System.nanoTime();
        System.out.println("LinkedList 걸린시간:" + (endTime - startTime) + "ns");
    }
}
```

[실행결과]

ArrayList 걸린시간:	10719106ns
LinkedList 걸린시간:	2087025ns

15.3 Set 컬렉션

List 컬렉션은 저장 순서를 유지하지만, Set 컬렉션은 저장 순서가 유지되지 않는다. 또한 객체를 중복해서 저장할 수 없고, 하나의 null만 저장할 수 있다. Set 컬렉션은 수학의 집합에 비유될 수 있다. 집합은 순서와 상관없고 중복이 허용되지 않기 때문이다. Set 컬렉션은 또한 구슬 주머니와도 같다. 동일한 구슬을 두 개 넣을 수 없고, 들어갈(저장할) 때의 순서와 나올(찾을) 때의 순서가 다를 수도 있기 때문이다. Set 컬렉션에는 HashSet, LinkedHashSet, TreeSet 등의 구현 클래스들이 있다.

15.3.1 HashSet

HashSet은 Set 인터페이스의 구현 클래스이다. HashSet은 객체들을 순서 없이 저장하고 동일한 객체는 중복 저장하지 않는다. HashSet이 관단하는 동일한 객체란 꼭 같은 인스턴스를 뜻하지는 않는다. HashSet은 객체를 저장하기 전에 먼저 객체의 hashCode() 메서드를 호출해서 해시 코드를 얻어낸다. 그리고 이미 저장되어 있는 객체들의 해시코드와 비교한다.

[Sample.java] String 객체를 중복 없이 저장하는 HashSet

```
public class Sample {
    public static void main(String[] args) {
        Set<String> set = new HashSet<String>();
        set.add("Java"); // "Java"는 한 번만 저장됨
        set.add("JDBC");
        set.add("Servlet/JSP");
        set.add("Java");
        set.add("iBATIS");
        int size = set.size(); // 저장된 객체 수 얻기
        System.out.println("총 객체수:" + size);

        Iterator<String> iterator = set.iterator(); // 반복자 얻기
        while(iterator.hasNext()) { // 객체 수만큼 루핑
            String element = iterator.next(); // 한 개의 객체를 가져온다.
            System.out.println("\t" + element);
        }
        set.remove("JDBC"); // 한 개의 객체 삭제
        set.remove("iBATIS");
        System.out.println("총 객체수:" + set.size()); // 저장된 객체수 얻기
        iterator = set.iterator(); // 반복자 얻기
        while(iterator.hasNext()) { // 객체 수 만큼 루핑
            String element = iterator.next();
            System.out.println("\t" + element);
        }
        set.clear(); // 모든 객체를 제거하고 비움
        if(set.isEmpty()) { System.out.println("비어 있음"); }
    }
}
```

[실행결과]

```
총 객체수:4
    Java
    JDBC
    Servlet/JSP
    iBATIS
총 객체수:2
    Java
    Servlet/JSP
비어 있음
```

아래 예제는 사용자 정의 클래스인 Member를 만들고 hashCode()와 equals() 메서드를 오버라이딩하였다. 인스턴스가 달라도 이름과 나이가 동일하다면 동일한 객체로 간주하고 중복 저장되지 않도록 하기 위해서이다.

[Member.java] hashCode()와 equals() 메서드 재정의

```
public class Member {
    public String name;
    public int age;
    public Member(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public boolean equals(Object obj) { // name과 age값이 같으면 true를 리턴
        if(obj instanceof Member) {
            Member member = (Member)obj;
            return member.name.equals(name) && (member.age == age);
        } else {
            return false;
        }
    }
    @Override
    public int hashCode() { // name과 age값이 같으면 동일한 hashCode가 리턴
        return name.hashCode() + age;
    }
}
```

[Sample.java] Member 객체를 중복없이 저장하는 HashSet

```
import java.util.HashSet;
import java.util.Set;
public class Sample {
    public static void main(String[] args) {
        Set<Member> set = new HashSet<Member>();

        //인스턴스는 다르지만 내부 데이터가 동일하므로 객체 1개만 저장
        set.add(new Member("홍길동", 30));
        set.add(new Member("홍길동", 30));

        //저장된 객체 수 얻기
        System.out.println("총 객체수:" + set.size());
    }
}
```

[실행결과]
총 객체수:1

15.4 Map 컬렉션

Map 컬렉션은 키(key)와 값(value)으로 구성된 Entry 객체를 저장하는 구조를 가지고 있다. 키는 중복 저장될 수 없지만 값은 중복 저장될 수 있다. 만약에 기존에 저장된 키와 동일한 키로 값을 저장하면 기존의 값은 없어지고 새로운 값으로 대체된다. Map 컬렉션에는 HashMap, Hashtable, LinkedHashMap, Properties, TreeMap 등의 구현 클래스들이 있다.

15.4.1 HashMap

HashMap의 키로 사용할 객체는 hashCode()와 equals() 메서드를 재정의해서 동등 객체가 될 조건을 정해야 한다. 동등 객체의 동일한 키가 될 조건은 hashCode()의 리턴값이 같아야 하고, equals() 메서드가 true를 리턴 해야 한다.

[Sample.java] 이름을 키로 점수를 값으로 저장하기

```
public class Sample {
    public static void main(String[] args) {
        //Map 컬렉션 생성
        Map<String, Integer> map = new HashMap<String, Integer>();
        //객체 저장
        map.put("신용권", 85);
        map.put("홍길동", 90);
        map.put("동장군", 80);
        map.put("홍길동", 95); // "홍길동" 키가 같기 때문에 제일 마지막에 저장한 값으로 대체
        System.out.println("총 Entry 수:" + map.size()); //저장된 총 Entry 수 얻기
        //객체 찾기
        System.out.println("\t홍길동:" + map.get("홍길동")); //이름(키)으로 점수(값)를 검색
        System.out.println();
        //객체를 하나씩 처리
        Set<String> keySet = map.keySet(); //Key Set 얻기
        Iterator<String> keyIterator = keySet.iterator();
        while(keyIterator.hasNext()){ //반복해서 키를 얻고 값을 Map에서 얻어냄
            String key = keyIterator.next();
            Integer value = map.get(key);
            System.out.println("\t" + key + ":" + value);
        }
        System.out.println();
        //객체 삭제
        map.remove("홍길동"); //키로 Map.Entry를 제거
        System.out.println("총 Entry 수:" + map.size());
        //객체를 하나씩 처리
        Set<Map.Entry<String, Integer>> entrySet = map.entrySet(); //Map.Entry Set 얻기
        Iterator<Map.Entry<String, Integer>> entryIterator = entrySet.iterator();
        while(entryIterator.hasNext()){ //반복해서 Map.Entry를 얻고 키와 값을 얻어냄
            Map.Entry<String, Integer> entry = entryIterator.next();
            String key = entry.getKey();
            Integer value = entry.getValue();
            System.out.println("\t" + key + ":" + value);
        }
        System.out.println();
        //객체 전체 삭제
        map.clear(); //모든 Map.Entry 객체
        System.out.println("총 Entry 수:" + map.size());
    }
}
```

[실행결과]

```
총 Entry 수:3
      홍길동:95

      홍길동:95
      신용권:85
      동장군:80

총 Entry 수:2
      신용권:85
      동장군:80

총 Entry 수:0
```


아래 예제는 사용자 정의 객체인 Student를 키로하고 점수를 저장하는 HashMap 사용방법을 보여준다. 학번과 이름이 동일한 Student를 동등한 키로 간주하기 위해 Student 클래스에는 hashCode()와 equals() 메서드가 재정의되어 있다.

[Student.java] 키로 사용할 객체 - hashCode()와 equals() 재정의

```
public class Student {
    public int sno;
    public String name;

    public Student(int sno, String name) {
        this.sno = sno;
        this.name = name;
    }
    public boolean equals(Object obj) { //학번과 이름이 동일한 경우 true를 리턴
        if(obj instanceof Student) {
            Student student = (Student)obj;
            return (sno == student.sno) && (name.equals(student.name));
        } else {
            return false;
        }
    }
    public int hashCode() { //학번과 이름이 같다면 동일한 값을 리턴
        return sno + name.hashCode();
    }
}
```

[Sample.java] 학번과 이름이 동일한 경우 같은 키로 인식

```
public class Sample {
    public static void main(String[] args) {
        Map<Student, Integer> map = new HashMap<Student, Integer>();
        map.put(new Student(1, "홍길동"), 95); //학번과 이름이 동일한 Student를 키로 저장
        map.put(new Student(1, "홍길동"), 95);
        System.out.println("총 Entry 수:" + map.size()); //저장된 총 Map.Entry 수 알기
    }
}
```

[실행결과]

총 Entry 수:1

15.4.2 Hashtable

Hashtable은 HashMap과 동일한 구조를 가지고 있다. 차이점은 동기화된 메서드로 구성되어 있기 때문에 멀티 스레드가 동시에 이 메서드들을 실행할 수 없고, 하나의 스레드가 실행을 완료해야만 다른 스레드를 실행할 수 있기 때문에 멀티 스레드에서 안전하게 객체를 추가, 삭제할 수 있다.

[Sample.java] 아이디와 비밀번호 검사하기

```
public class Sample {
    public static void main(String[] args) {
        Map<String, String> map = new Hashtable<String, String>();

        map.put("spring", "12");
        map.put("summer", "123");
        map.put("fall", "1234");
        map.put("winter", "12345");

        Scanner scanner = new Scanner(System.in); //키보드로부터 입력된 내용을 받기 위해 생성
        while(true) {
            System.out.println("아이디와 비밀번호를 입력해주세요.");
            System.out.print("아이디:");
            String id = scanner.nextLine(); //키보드로 입력한 아이디를 읽는다.
            System.out.print("비밀번호:");
            String password = scanner.nextLine(); //키보드를 입력한 비밀번호를 읽는다.
            System.out.println();

            if(map.containsKey(id)) { //아이디인 키가 존재하지는 확인한다.
                if( map.get(id).equals(password) ) { //비밀번호를 비교한다.
                    System.out.println("로그인 되었습니다.");
                    break;
                } else {
                    System.out.println("비밀번호가 일치하지 않습니다.");
                }
            } else {
                System.out.println("입력하신 아이디가 존재하지 않습니다.");
            }
        }
    }
}
```

[실행결과]

아이디와 비밀번호를 입력해주세요
아이디 : summer
비밀번호 : 123
로그인 되었습니다.

15.4.3 Properties

Properties는 Hashtable의 하위 클래스이므로 Hashtable의 모든 특징을 그대로 가지고 있다. 차이점은 Hashtable은 키와 값을 다양한 타입으로 지정하는데 비해 Properties는 키와 String 타입으로 제한한 컬렉션이다. Properties는 애플리케이션의 옵션 정보, 데이터베이스 연결 정보 그리고 국제화(다국어) 정보가 저장된 프로퍼티(~.properties) 파일을 읽을 때 주로 사용한다.

프로퍼티 파일은 키와 값이 = 기호로 연결되어 있는 텍스트 파일로 ISO 8859-1 문자셋으로 저장된다. 이 문자셋으로 직접 표현할 수 없는 한글은 유니코드(Unicode)로 변환되어 저장된다. 예를 들어 county와 language 키고 각각 "대한민국", "한글"을 입력하면 자동으로 유니코드로 변환되어 저장된다. 이클립스에서 유니코드로 변환된 내용을 다시 한글로 보려면 마우스를 유니코드 위에 올려놓으면 된다.

아래는 데이터베이스 연결 정보가 있는 프로퍼티 파일의 내용이다. driver, url, username, password는 키가 되고 그 뒤의 문자열은 값이다.

[database.properties] 키=값으로 구성된 프로퍼티

```
driver=oracle.jdbc.OracleDriver
url=jdbc:oracle:thin:@localhost:1521:orcl
username=scott
password=tiger
```

[Sample.java] 프로퍼티 파일로부터 읽기

```
public class Sample {
    public static void main(String[] args) throws Exception{
        Properties properties = new Properties();

        String path = Sample.class.getResource("database.properties").getPath();
        path = URLDecoder.decode(path, "utf-8");
        properties.load(new FileReader(path));

        String driver = properties.getProperty("driver");
        String url = properties.getProperty("url");
        String username = properties.getProperty("username");
        String password = properties.getProperty("password");

        System.out.println("driver:" + driver);
        System.out.println("url:" + url);
        System.out.println("username:" + username);
        System.out.println("password:" + password);
    }
}
```

[실행결과]

```
driver:oracle.jdbc.OracleDriver
url:jdbc:oracle:thin:@localhost:1521:orcl
username:scott
password:tiger
```

15.5 검색 기능을 강화시킨 컬렉션

컬렉션 프레임워크는 검색 기능을 강화시킨 TreeSet과 TreeMap을 제공하고 있다. TreeSet은 Set 컬렉션이고, TreeMap은 Map 컬렉션이다. 이 컬렉션들은 이진 트리(binary tree)를 이용해서 계층적 구조(Tree 구조)를 가지면서 객체를 저장한다.

15.5.1 이진 트리 구조

이진 트리(binary tree)는 여러 개의 노드(node)가 트리 형태로 연결된 구조로, 루트 노드(root node)라고 불리는 하나의 노드에서부터 시작해서 각 노드에 최대 2개의 노드를 연결할 수 있는 구조를 가지고 있다. 위아래로 연결된 두 노드를 부모-자식관계에 있다고 하며 위의 노드를 부모 노드, 아래의 노드를 자식 노드라고 한다. 하나의 부모 노드는 최대 두 개의 자식 노드와 연결될 수 있다.

15.5.2 TreeSet

TreeSet은 이진 트리(binary tree)를 기반으로 한 Set 컬렉션이다. 하나의 노드는 노드값인 value와 왼쪽과 오른쪽 자식 노드를 참조하기 위한 두 개의 변수로 구성된다. TreeSet에 객체를 지정하면 자동으로 정렬되는데 부모값과 비교해서 낮은 것은 왼쪽 자식 노드에, 높은 것은 오른쪽 자식 노드에 저장한다. TreeSet을 생성하기 위해서는 저장할 객체 타입을 파라미터로 표기하고 기본 생성자를 호출하면 된다.

```
TreeSet<E> treeSet = new TreeSet<E>();
```

String 객체를 저장하는 TreeSet은 다음과 같이 생성할 수 있다.

```
TreeSet<String> treeSet = new TreeSet<String>();
```

Set 인터페이스 타입 변수에 대입해도 되지만, 객체를 찾거나 범위 검색과 관련된 메시지를 사용하기 위해 TreeSet 클래스 타입으로 대입했다.

아래 예제는 점수를 무작위로 저장하고 특정 점수를 찾는 방법을 보여준다.

[Sample.java] 특정 객체 찾기

```
public class Sample {
    public static void main(String[] args) {
        TreeSet<Integer> scores = new TreeSet<Integer>();
        scores.add(new Integer(87));
        scores.add(new Integer(98));
        scores.add(new Integer(75));
        scores.add(new Integer(95));
        scores.add(new Integer(80));

        Integer score = null;

        score = scores.first();
        System.out.println("가장 낮은 점수:" + score);

        score = scores.last();
        System.out.println("가장 높은 점수:" + score + "\n");

        score = scores.lower(new Integer(95));
        System.out.println("95점 아래 점수:" + score);

        score = scores.higher(new Integer(95));
        System.out.println("95점 위의 점수:" + score + "\n");

        score = scores.floor(new Integer(95));
        System.out.println("95점이거나 바로 아래 점수:" + score);

        score = scores.ceiling(new Integer(85));
        System.out.println("85점이거나 바로 위의 점수:" + score + "\n");

        while(!scores.isEmpty()) {
            score = scores.pollFirst();
            System.out.println(score + "(남은 객체 수:" + scores.size() + ")");
        }
    }
}
```

[실행결과]

```
가장 낮은 점수:75
가장 높은 점수:98

95점 아래 점수:87
95점 위의 점수:98

95점 이거나 바로 아래 점수:95
85점 이거나 바로 위의 점수:87

75(남은 객체 수:4)
80(남은 객체 수:3)
87(남은 객체 수:2)
95(남은 객체 수:1)
98(남은 객체 수:0)
```

descendingSet() 메서드는 내림차순으로 정렬된 NavigableSet 객체를 리턴하는데 NavigableSet은 TreeSet과 마찬가지로 first(), last(), lower(), higher(), floor(), ceiling() 메서드를 제공하고, 정렬 순서를 바꾸는 descendingSet() 메서드들을 제공한다. 오름 차순으로 정렬하고 싶다면 다음과 같이 descendingSet() 메서드를 두 번 호출하면 된다.

```
NavigableSet<E> descendingSet = treeSet.descendingSet();
NavigableSet<E> ascendingSet = descendingSet.descendingSet();
```

[Sample.java] 객체 정렬하기

```
import java.util.NavigableSet;
import java.util.TreeSet;

public class Sample {
    public static void main(String[] args) {
        TreeSet<Integer> scores = new TreeSet<Integer>();

        scores.add(new Integer(87));
        scores.add(new Integer(98));
        scores.add(new Integer(75));
        scores.add(new Integer(95));
        scores.add(new Integer(80));

        NavigableSet<Integer> descendingSet = scores.descendingSet();
        for(Integer score:descendingSet) {
            System.out.println(score);
        }
        System.out.println();

        NavigableSet<Integer> ascendingSet = descendingSet.descendingSet();
        for(Integer score:ascendingSet) {
            System.out.println(score);
        }
    }
}
```

[실행결과]

```
98
95
87
80
75

75
80
87
95
98
```

아래는 영어 단어를 무작위로 TreeSet에 저장한 후 알파벳 c~f 사이의 단어를 검색해보는 예제이다.

[Sample.java] 영어 단어를 정렬하고, 범위 검색해보기

```
import java.util.NavigableSet;
import java.util.TreeSet;
public class Sample {
    public static void main(String[] args) {
        TreeSet<String> treeSet = new TreeSet<String>();

        treeSet.add("apple");
        treeSet.add("forever");
        treeSet.add("description");
        treeSet.add("ever");
        treeSet.add("zoo");
        treeSet.add("base");
        treeSet.add("guess");
        treeSet.add("cherry");

        System.out.println("[c~f 사이의 단어 검색]");
        NavigableSet<String> rangeSet = treeSet.subSet("c", true, "f", true);

        for(String word:rangeSet) {
            System.out.println(word);
        }
    }
}
```

[실행결과]

[c~f 사이의 단어 검색]
cherry
description
ever

15.5.3 TreeMap

TreeMap은 이진 트리를 기반으로 한 Map 컬렉션이다. TreeSet과의 차이점은 키와 값이 저장된 Map.Entry를 저장한다는 점이다. TreeMap에 객체를 저장하면 자동으로 정렬되는데, 기본적으로 부모 키값과 비교해서 키 값이 낮은 것은 왼쪽 자식 노드에, 키 값이 높은 것은 오른쪽 자식 노드에 Map.Entry 객체를 저장한다. 아래는 TreeMap이 가지고 있는 검색 관련 메서드들이다. 아래 예제는 점수를 키로, 이름을 값으로 해서 무작위로 저장하고 특정 Map.Entry를 찾는 방법을 보여준다.

[Sample.java] 특정 Map.Entry 찾기

```
public class Sample {
    public static void main(String[] args) {
        TreeMap<Integer, String> scores = new TreeMap<Integer, String>();
        scores.put(new Integer(87), "홍길동");
        scores.put(new Integer(98), "이동수");
        scores.put(new Integer(75), "박길순");
        scores.put(new Integer(95), "신용권");
        scores.put(new Integer(80), "김자바");

        Map.Entry<Integer, String> entry = null;

        entry = scores.firstEntry();
        System.out.println("가장 낮은 점수:" + entry.getKey() + "-" + entry.getValue());

        entry = scores.lastEntry();
        System.out.println("가장 높은 점수:" + entry.getKey() + "-" + entry.getValue()+"\n");

        entry = scores.lowerEntry(new Integer(95));
        System.out.println("95점 아래 점수:" + entry.getKey() + "-" + entry.getValue());

        entry = scores.higherEntry(new Integer(95));
        System.out.println("95점 위의 점수:" + entry.getKey() + "-" + entry.getValue() + "\n");

        entry = scores.floorEntry(new Integer(95));
        System.out.println("95점 이거나 바로 아래 점수:" + entry.getKey() + "-" + entry.getValue());

        entry = scores.ceilingEntry(new Integer(85));
        System.out.println("85점 이거나 바로 아래 점수:" + entry.getKey() + "-" + entry.getValue() + "\n");

        while(!scores.isEmpty()) {
            entry = scores.pollFirstEntry();
            System.out.println(entry.getKey() + "-" + entry.getValue() + "(남은 객체 수: " + scores.size() + ")");
        }
    }
}
```

[실행결과]

가장 낮은 점수:75-박길순
가장 높은 점수:98-이동수

95점 아래 점수:87-홍길동
95점 위의 점수:98-이동수

95점 이거나 바로 아래 점수:95-신용권
85점 이거나 바로 아래 점수:87-홍길동

75-박길순 (남은 객체 수:4)
80-김자바 (남은 객체 수:3)
87-홍길동 (남은 객체 수:2)
95-신용권 (남은 객체 수:1)
98-이동수 (남은 객체 수:0)

[Sample.java] 객체 정렬하기

```
public class Sample {
    public static void main(String[] args) {
        TreeMap<Integer, String> scores = new TreeMap<Integer, String>();
        scores.put(new Integer(87), "홍길동");
        scores.put(new Integer(98), "이동수");
        scores.put(new Integer(75), "박길순");
        scores.put(new Integer(95), "신용권");
        scores.put(new Integer(80), "김자바");
        NavigableMap<Integer, String> descendingMap = scores.descendingMap();
        Set<Map.Entry<Integer, String>> descendingEntrySet = descendingMap.entrySet();
        for (Map.Entry<Integer, String> entry : descendingEntrySet) {
            System.out.print(entry.getKey() + "-" + entry.getValue() + " ");
        }
        System.out.println();

        NavigableMap<Integer, String> ascendingMap = descendingMap.descendingMap();
        Set<Map.Entry<Integer, String>> ascendingEntrySet = ascendingMap.entrySet();
        for (Map.Entry<Integer, String> entry : ascendingEntrySet) {
            System.out.print(entry.getKey() + "-" + entry.getValue() + " ");
        }
    }
}
```

[실행결과]

98-이동수 95-신용권 87-홍길동 80-김자바 75-박길순
75-박길순 80-김자바 87-홍길동 95-신용권 98-이동수

아래는 영어 단어와 페이지 정보를 무작위로 TreeMap에 저장한 후 알파벳 c~f 사이의 단어를 검색해보는 예제이다.

[Sample.java] 키로 정렬하고 범위 검색하기

```
public class Sample {
    public static void main(String[] args) {
        TreeMap<String, Integer> treeMap = new TreeMap<String, Integer>();
        treeMap.put("apple", new Integer(10));
        treeMap.put("forever", new Integer(60));
        treeMap.put("description", new Integer(40));
        treeMap.put("ever", new Integer(50));
        treeMap.put("zoo", new Integer(10));
        treeMap.put("base", new Integer(20));
        treeMap.put("guess", new Integer(70));
        treeMap.put("cherry", new Integer(30));
        System.out.println("[c~f 사이의 단어 검색]");
        NavigableMap<String, Integer> rangeMap = treeMap.subMap("c", true, "f", true);
        for (Map.Entry<String, Integer> entry : rangeMap.entrySet()) {
            System.out.println(entry.getKey() + "-" + entry.getValue() + "페이지");
        }
    }
}
```

[실행결과]

[c~f 사이의 단어 검색]
cherry-30페이지
description-40페이지
ever-50페이지

15.5.4 Comparable과 Comparator

TreeSet과 TreeMap은 정렬을 실행하기 위해 java.lang.Comparable을 구현한 객체를 요구하는데, Integer, Double, String,은 모두 Comparable 인터페이스를 구현하고 있다. 사용자 정의 클래스도 Comparable을 구현한다면 자동 정렬이 가능하다. Comparable에는 CompareTo() 메서드가 정의되어 있기 때문에 사용자 정의 클래스에서는 이 메서드를 오버라이딩하여 아래와 같이 리턴값을 만들어 내야 한다. 아래는 나이를 기준으로 Person 객체를 오름차순으로 정렬하기 위해 Comparable 인터페이스를 구현한 것이다.

[Person.java] Comparable 구현 클래스

```
public class Person implements Comparable<Person> {
    public String name;
    int age;

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

    public int compareTo(Person o) {
        if (age < o.age) return -1;
        else if (age == o.age) return 0;
        else return 1;
    }
}
```

[Sample.java] 사용자 정의 객체를 나이순으로 정렬하기

```
public class Sample {
    public static void main(String[] args) {
        TreeSet<Person> treeSet = new TreeSet<Person>();
        treeSet.add(new Person("홍길동", 45)); //저장될 때 나이순으로 정렬됨
        treeSet.add(new Person("김자바", 25));
        treeSet.add(new Person("박지원", 31));

        Iterator<Person> iterator = treeSet.iterator(); //왼쪽 마지막 노드에서 오른쪽 마지막 노드까지 반복해서 가져오기(오름차순)
        while(iterator.hasNext()) {
            Person person = iterator.next();
            System.out.println(person.name + ":" + person.age);
        }
    }
}
```

[실행결과]

```
김자바:25
박지원:31
홍길동:45
```

compare() 메서드는 비교하는 두 객체가 동등하면 0, 비교하는 값보다 앞에 오게 하려면 음수, 뒤에 오게 하려면 양수를 리턴하도록 구현하면 된다. 아래는 가격을 기준으로 Fruit 객체를 내림차순으로 정렬시키는 정렬자이다.

[DescendingComparator.java] Fruit의 내림차순 정렬자

```
public class DescendingComparator implements Comparator<Fruit>{
    @Override
    public int compare(Fruit o1, Fruit o2) {
        if(o1.price < o2.price) return 1; //가격이 적을 경우 뒤에 오게 함
        else if(o1.price == o2.price) return 0;
        else return -1; //가격이 클 경우 앞에 오게 함
    }
}
```

[Fruit.java] Comparable을 구현하지 않은 클래스

```
public class Fruit {
    public String name;
    public price;

    public Fruit(String name, int price){
        this.name = name;
        this.price = price;
    }
}
```

아래 예제는 내림차순 정렬자를 이용해서 TreeSet에 Fruit을 저장한다. 정렬자를 주지 않고 TreeSet에 저장하면 ClassCastException 예외가 발생하지만, 정렬자를 TreeSet의 생성자에 주면 예외가 발생하지 않고 자동으로 내림차순 정렬되는 것을 볼 수 있다.

[Sample.java] 내림차순 정렬자를 사용하는 TreeSet

```
public class Sample {
    public static void main(String[] args) {
        /*
        TreeSet<Fruit> treeSet = new TreeSet<Fruit>();

        treeSet.add(new Fruit("포도", 3000));
        treeSet.add(new Fruit("수박", 10000)); //Fruit이 Comparable을 구현하지 않았기 때문에 예외 발생
        treeSet.add(new Fruit("딸기", 6000));
        */

        TreeSet<Fruit> treeSet = new TreeSet<Fruit>( new DescendingComparator()); //내림차순 정렬자 제공
        treeSet.add(new Fruit("포도", 3000));
        treeSet.add(new Fruit("수박", 10000)); //저장될 때 가격을 기준으로 내림차순 정렬됨
        treeSet.add(new Fruit("딸기", 6000));

        Iterator<Fruit> iterator = treeSet.iterator();
        while(iterator.hasNext()){
            Fruit fruit = iterator.next();
            System.out.println(fruit.name + ":" + fruit.price);
        }
    }
}
```

[실행결과]

```
수박:10000
딸기:6000
포도:3000
```

15.6 LIFO와 FIFO 컬렉션

후입선출(LIFO: Last In First Out)은 나중에 넣은 객체가 먼저 빠져나가는 자료구조를 말한다. 반대로 선입선출(FIFO: First In First Out)은 먼저 넣은 객체가 먼저 빠져나가는 구조를 말한다. 컬렉션 프레임워크에는 LIFO 자료구조를 제공하는 스택(Stack) 클래스와 FIFO 자료구조를 제공하는 큐(Queue) 인터페이스를 제공하고 있다.

15.6.1 Stack

Stack 클래스는 LIFO 자료구조를 구현한 클래스이다. 아래는 Stack 클래스의 주요 메서드들이다.

리턴타입	메서드	설명
E	push(E item)	주어진 객체를 스택에 넣는다.
E	peek()	스택의 맨 위 객체를 가져온다. 객체를 스택에서 제거하지 않는다.
E	pop()	스택의 맨 위 객체를 가져온다. 객체를 스택에서 제거한다.

아래는 택시에서 많이 볼 수 있는 동전케이스를 Stack 클래스로 구현한 예제이다. 동전케이스는 위에만 오픈되어 있는 스택 구조를 가지고 있다. 먼저 넣은 동전은 제일 밑에 깔리고 나중에 넣은 동전이 위에 쌓이기 때문에 Stack에서 동전을 빼면 마지막에 넣은 동전이 먼저 나온다.

[Coin.java] 동전 클래스

```
public class Coin {
    private int value;

    public Coin(int value){
        this.value = value;
    }
    public int getValue(){
        return value;
    }
}
```

[Sample.java] Stack을 이용한 동전 케이스

```
public class Sample {
    public static void main(String[] args) {
        Stack<Coin> coinBox = new Stack<Coin>();

        coinBox.push(new Coin(100));
        coinBox.push(new Coin(50));
        coinBox.push(new Coin(500));
        coinBox.push(new Coin(10));

        while( !coinBox.isEmpty() ) { //동전케이스가 비었는지 확인
            Coin coin = coinBox.pop(); //동전케이스에서 제일 위의 동전 꺼내기
            System.out.println("꺼내온 동전:" + coin.getValue() + "원");
        }
    }
}
```

[실행결과]

```
꺼내온 동전 :10원
꺼내온 동전 :500원
꺼내온 동전 :50원
꺼내온 동전 :100원
```

15.6.2 Queue

Queue 인터페이스는 FIFO 자료구조에서 사용되는 메서드를 정의하고 있다. 아래는 Queue 인터페이스에 정의되어 있는 메서드를 보여준다.

리턴타입	메서드	설명
boolean	offer(E e)	주어진 객체를 넣는다.
E	peek()	객체 하나를 가져온다. 객체를 큐에서 제거하지 않는다.
E	poll()	객체 하나를 가져온다. 객체를 큐에서 제거한다.

Queue 인터페이스를 구현한 대표적인 클래스는 LinkedList이다. LinkedList는 List 인터페이스를 구현했기 때문에 List 컬렉션이기도 하다. 아래 코드는 LinkedList 객체를 Queue 인터페이스 타입으로 변환한 것이다.

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

아래는 Queue를 이용해서 간단한 메시지 큐를 구현한 예제이다. 먼저 넣은 메시지가 먼저 나오기 때문에 넣은 순서대로 메시지가 처리된다.

[Message.java] Message 클래스

```
public class Message {
    public String command;
    public String to;

    public Message(String command, String to) {
        this.command = command;
        this.to = to;
    }
}
```

[Sample.java] Queue를 이용한 메시지 큐

```
public class Sample {
    public static void main(String[] args) {
        Queue<Message> messageQueue = new LinkedList<Message>();

        messageQueue.offer(new Message("sendMail", "홍길동"));
        messageQueue.offer(new Message("sendSMS", "신용권"));
        messageQueue.offer(new Message("sendKakaotalk", "홍두깨"));

        while( !messageQueue.isEmpty() ) { //메시지 큐가 비었는지 확인
            Message message = messageQueue.poll();
            switch(message.command){
                case "sendMail":
                    System.out.println(message.to+"님에게 메일을 보냅니다.");
                    break;
                case "sendSMS":
                    System.out.println(message.to+"님에게 SMS를 보냅니다.");
                    break;
                case "sendKakaotalk":
                    System.out.println(message.to+"님에게 카카오톡을 보냅니다.");
                    break;
            }
        }
    }
}
```

//메시지 저장

[실행결과]

홍길동님에게 메일을 보냅니다.
신용권님에게 SMS를 보냅니다.
홍두깨님에게 카카오톡을 보냅니다.

15.7 동기화된 컬렉션

ArrayList, HashSet, HashMap을 싱글 스레드 환경에서 사용하다가 멀티 스레드 환경으로 전달할 필요도 있다. 이런 경우를 대비해서 컬렉션 프레임워크는 비동기화된 메서드를 동기화된 메서드로 래핑하는 Collections의 synchronizedXXX() 메서드를 제공한다. 매개값으로 비동기화된 컬렉션을 대입하면 동기화된 컬렉션을 리턴 한다.

리턴타입	메서드	설명
List<T>	synchronizedList(List<T> list)	List를 동기화된 List로 리턴
Map<K, V>	synchronizedMap(Map<K, V> m)	Map을 동기화된 Map으로 리턴
Set<T>	synchronizedSet(Set<T> s)	Set을 동기화된 Set으로 리턴

15.8 병렬 처리를 위한 컬렉션

자바는 컬렉션의 요소를 병렬 처리할 수 있도록 java.util.concurrent 패키지의 ConcurrentHash.Map과 ConcurrentLinkedQueue 컬렉션을 제공한다. ConcurrentHashMap은 Map 구현 클래스이고, ConcurrentLinkedQueue는 Queue 구현 클래스이다.

ConcurrentHashMap은 컬렉션에 10개의 요소가 저장되어 있을 경우, 1개를 처리할 동안 전체 10개의 요소를 다른 스레드가 처리하지 못하도록 하는 것이 전체 잠금이며 처리하는 요소가 포함된 부분만 잠금하고 나머지 부분은 다른 스레드가 변경할 수 있도록 하는 것이 부분 잠금이다.

ConcurrentLinkedQueue는 락-프리 알고리즘을 구현한 컬렉션이다. 락-프리 알고리즘은 여러 개의 스레드가 동시에 접근할 경우, 잠금을 사용하지 않고도 최소한 하나의 스레드가 안전하게 요소를 저장하거나 얻도록 해준다.

16장. 스트림과 병렬 처리

16.1 스트림 소개

스트림(Stream)은 자바 8부터 추가된 컬렉션(배열 포함)의 저장 요소를 하나씩 참조해서 랩다식으로 처리할 수 있도록 해주는 반복자이다.

16.1.1 반복자 스트림

자바 7 이전까지는 List<String> 컬렉션에서 요소를 순차적으로 처리하기 위해 Iterator 반복자를 사용했다. Iterator를 사용한 코드와 Stream을 사용한 코드를 비교해보면 Stream을 사용하는 것이 훨씬 단순해 보인다. 아래 예제는 List<String> 컬렉션의 String 요소를 Iterator와 Stream을 이용해서 순차적으로 콘솔에 출력한다.

[Sample.java] Iterator와 Stream 순차 처리 코드

```
public class Sample {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("홍길동", "신용권", "김자바");

        //Iterator 이용
        Iterator<String> iterator = list.iterator();
        while(iterator.hasNext()){
            String name = iterator.next();
            System.out.println(name);
        }

        System.out.println();

        //Stream 이용
        Stream<String> stream = list.stream();
        stream.forEach(name -> System.out.println(name));
    }
}
```

[실행결과]

홍길동
신용권
김자바

홍길동
신용권
김자바

16.1.2 스트림의 특징

Stream은 Iterator와 비슷한 역할을 하는 반복자이지만, 랩다식으로 요소 처리 코드를 제공하는 점과 내부 반복자를 사용하므로 병렬 처리가 쉽다는 점 그리고 중간 처리와 최종 처리 작업을 수행하는 점에서 많은 차이를 가지고 있다. 아래 예제는 컬렉션에 저장된 Student를 하나씩 가져와 학생 이름과 성적을 콘솔에 출력하도록 forEach() 메서드 매개값으로 랩다식을 주었다.

[Sample.java] 요소 처리를 위한 랩다식

```
public class Sample {
    public static void main(String[] args) {
        List<Student> list = Arrays.asList(
            new Student("홍길동", 90),
            new Student("신용권", 92)
        );

        Stream<Student> stream = list.stream(); //스트림 얻기
        stream.forEach(s -> {
            String name = s.getName();
            int score = s.getScore();
            System.out.println(name + "-" + score);
        });
    }
}
```

[실행결과]

홍길동-90
신용권-92

//List 컬렉션에서 Student를 가져와 랩다식의 매개값으로 제공

[Student.java] 학생 클래스

```
public class Student {
    private String name;
    private int score;

    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }

    public String getName() {return name;}
    public int getScore() {return score;}
}
```


아래 예제는 순차 처리 스트림과 병렬 처리 스트림을 이용할 경우, 사용된 스레드의 이름이 무엇인지 콘솔에 출력한다. 실행 결과를 보면 병렬 처리 스트림은 main 스레드를 포함해서 ForkJoinPool(스레드풀)의 작업 스레드들이 병렬적으로 요소를 처리하는 것을 볼 수 있다.

[Sample.java] 병렬 처리

```
public class Sample {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("홍길동", "신용권", "김자바", "람다식", "박병렬");

        Stream<String> stream = list.stream(); //순차 처리
        stream.forEach(Sample :: print);
        System.out.println();

        Stream<String> parallelStream = list.parallelStream(); //병렬 처리
        parallelStream.forEach(Sample :: print);
    }
    public static void print(String str){
        System.out.println(str + ":" + Thread.currentThread().getName());
    }
}
```

[실행결과]

```
홍길동:main
신용권:main
김자바:main
람다식:main
박병렬:main

김자바:main
박병렬:ForkJoinPool.commonPool-worker-2
신용권:ForkJoinPool.commonPool-worker-1
홍길동:ForkJoinPool.commonPool-worker-1
람다식:ForkJoinPool.commonPool-worker-2
```

아래 예제는 List에 저장되어 있는 Student 객체를 중간 처리에서 score 필드값으로 매핑하고, 최종 처리에서 score의 평균값을 산출한다.

[Sample.java] 중간 처리와 최종 처리

```
public class Sample {
    public static void main(String[] args) {
        List<Student> studentList = Arrays.asList(new Student("홍길동", 10), new Student("신용권", 20), new Student("유미선", 30));

        double avg = studentList.stream()
            .mapToInt(Student :: getScore) //중간 처리(학생 객체를 점수로 매핑)
            .average() //최종 처리(평균 점수)
            .getAsDouble();

        System.out.println("평균 점수:" + avg);
    }
}
```

[실행결과]

평균 점수:20.0

16.2 스트림 종류

16.2.1 컬렉션으로부터 스트림 얻기

아래 예제는 List<Student> 컬렉션에서 Stream<Student>를 얻어내고 요소를 콘솔에 출력한다.

[Sample.java] 컬렉션으로부터 스트림 얻기

```
public class Sample {
    public static void main(String[] args) {
        List<Student> studentList = Arrays.asList(
            new Student("홍길동", 10),
            new Student("신용권", 20),
            new Student("유미선", 30)
        );
        Stream<Student> stream = studentList.stream();
        stream.forEach(s -> System.out.println(s.getName()));
    }
}
```

[실행결과]

```
홍길동
신용권
유미선
```

[Student.java] 학생 클래스

```
public class Student {
    private String name;
    private int score;

    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
    public String getName() {
        return name;
    }
    public int getScore() {
        return score;
    }
}
```

16.2.2 배열로부터 스트림 얻기

아래 예제는 String[]과 int[] 배열로부터 스트림을 얻어내고 콘솔에 출력한다.

[Sample.java] 배열로부터 스트림 얻기

```
public class Sample {
    public static void main(String[] args) {
        String[] strArray = {"홍길동", "신용권", "김미나"};
        Stream<String> strStream = Arrays.stream(strArray);
        strStream.forEach(a -> System.out.print(a + ","));
        System.out.println();
        int[] intArray = {1, 2, 3, 4, 5};
        IntStream intStream = Arrays.stream(intArray);
        intStream.forEach(a -> System.out.print(a + ","));
        System.out.println();
    }
}
```

[실행결과]

홍길동,신용권,김미나,
1,2,3,4,5,

16.2.3 숫자 범위로부터 스트림 얻기

아래는 1부터 100까지의 합을 구하기 위해 IntStream의 rangeClosed() 메서드를 이용하였다. rangeClosed()는 첫 번째 매개값에서부터 두 번째 매개값까지 순차적으로 제공하는 IntStream을 리턴 한다. IntStream의 또 다른 range() 메서드도 동일한 IntStream을 리턴 하는데, 두 번째 매개값은 포함하지 않는다.

[Sample.java] 정수 범위를 소스로 하는 스트림

```
public class Sample {
    public static int sum;
    public static void main(String[] args) {
        IntStream stream = IntStream.rangeClosed(1, 100);
        stream.forEach(a -> sum += a);
        System.out.println("총합:" + sum);
    }
}
```

[실행결과]

총합:5050

16.2.4 파일로부터 스트림 얻기

아래 예제는 Files의 정적 메서드인 lines()와 BufferedReader의 lines() 메서드를 이용하여 파일을 읽어 콘솔에 출력한다.

[Sample.java] 파일 내용을 소스로 하는 스트림

```
public class Sample {
    public static void main(String[] args) throws IOException {
        Path path = Paths.get("src/linedata.txt"); //파일의 경로 정보를 가지고 있는 Path 객체 생성
        Stream<String> stream;
        //Files.lines() 메서드 이용
        stream = Files.lines(path, Charset.defaultCharset()); //운영 체제의 기본 문자셋
        stream.forEach(System.out :: println);
        System.out.println(); //메서드 참조(s -> System.out.println(s))와 동일)
        //BufferedReader lines() 메서드 이용
        File file = path.toFile();
        FileReader fileReader = new FileReader(file);
        BufferedReader br = new BufferedReader(fileReader);
        stream = br.lines();
        stream.forEach(System.out :: println);
    }
}
```

[실행결과]

Java8에서 추가된 새로운 기능
1.람다식
2.메서드 참조

Java8에서 추가된 새로운 기능
1.람다식
2.메서드 참조

16.2.5 디렉터리로부터 스트림 얻기

아래 예제는 Files의 정적 메서드인 list()를 이용해서 디렉터리의 내용(서브 디렉터리 또는 파일목록)을 스트림을 통해 읽고 콘솔에 출력한다.

[Sample.java] 디렉터리 내용을 소스로 하는 스트림

```
public class Sample {
    public static void main(String[] args) throws IOException {
        Path path = Paths.get("D:/자바");
        Stream<Path> stream = Files.list(path);
        stream.forEach(p -> System.out.println(p.getFileName()));
    }
}
```

[실행결과]

0418
0421
0424
0425
0428

16.3 스트림 파이프라인

16.3.1 중간 처리와 최종처리

스트림은 데이터의 필터링, 매핑, 정렬, 그룹핑 등의 중간 처리와 함께, 평균, 카운팅, 최대값, 최소값 등의 최종 처리를 파이프라인(pipelines)으로 해결한다. 파이프라인은 여러 개의 스트림이 연결되어 있는 구조를 말한다. 파이프라인에서 최종 처리를 제외하고는 모두 중간 처리 스트림이다.

[Sample.java] 스트림 파이프라인

```
public class Sample {
    public static void main(String[] args) {
        List<Member> list = Arrays.asList(
            new Member("홍길동", Member.MAIL, 30),
            new Member("김나리", Member.FEMAIL, 30),
            new Member("신용권", Member.MAIL, 45),
            new Member("박수미", Member.FEMAIL, 27)
        );

        double ageAvg = list.stream()
            .filter(m -> m.getSex() == Member.MAIL)
            .mapToInt(Member :: getAge)
            .average()
            .getAsDouble();

        System.out.println("남자 평균 나이: " + ageAvg);
    }
}
```

[실행결과]

남자 평균 나이: 37.5

[Member.java] 회원 클래스

```
public class Member {
    public static int MAIL = 0;
    public static int FEMAIL = 1;
    private String name;
    private int sex;
    private int age;

    public Member(String name, int sex, int age) {
        this.name = name;
        this.sex = sex;
        this.age = age;
    }

    public int getSex() {return sex;}
    public int getAge() {return age;}
}
```

16.4 필터링 (distinct(), filter())

필터링은 중간 처리 기능으로 요소를 걸러내는 역할을 한다. 필터링 메서드인 distinct()와 filter() 메서드는 모든 스트림이 가지고 있는 공통 메서드이다. 아래 예제는 이름 List에서 중복된 이름을 제거하고 출력한다. 그리고 성이 "신"인 이름만 필터링해서 출력한다.

[Sample.java] 필터링

```
public class Sample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("홍길동", "신용권", "김자바", "신용권", "신민철");

        names.stream()
            .distinct() //중복 제거
            .forEach(n -> System.out.println(n));
        System.out.println();

        names.stream()
            .filter(n -> n.startsWith("신")) //필터링
            .forEach(n -> System.out.println(n));
        System.out.println();

        names.stream()
            .distinct()
            .filter(n -> n.startsWith("신"))
            .forEach(n -> System.out.println(n));
    }
}
```

[실행결과]

홍길동
신용권
김자바
신민철

신용권
신용권
신민철

신용권
신민철

16.5 매핑 (flatMapXXX(), mapXXX(), asXXXStream(), boxed())

매핑은 중간 처리 기능으로 스트림의 요소를 다른 요소로 대체하는 작업을 말한다. 스트림에서 제공하는 매핑 메서드는 flatXXX()와 mapXXX(), 그리고 asDoubleStream(), asLongStream(), boxed()가 있다.

16.5.1 flatMapXXX() 메서드

아래 예제는 입력된 데이터(요소)들이 List<String>에 저장되어 있다고 가정하고, 요소별로 단어를 뽑아 단어 스트림으로 재생성 한다. 만약 입력된 데이터들이 숫자라면 숫자를 뽑아 숫자 스트림으로 재생성 한다.

[Sample.java] 복수 개의 요소로 대체

```
import java.util.Arrays, java.util.List;
public class Sample {
    public static void main(String[] args) {
        List<String> inputList1 = Arrays.asList("java8 lambda", "stream mapping");

        inputList1.stream()
            .flatMap(data -> Arrays.stream(data.split(" ")))
            .forEach(word -> System.out.println(word));
        System.out.println();

        List<String> inputList2 = Arrays.asList("10, 20, 30", "40, 50, 60");
        inputList2.stream().flatMapToInt(data -> {
            String[] strArr = data.split(",");
            int[] intArr = new int[strArr.length];
            for(int i=0; i<strArr.length; i++){
                intArr[i] = Integer.parseInt(strArr[i].trim());
            }
            return Arrays.stream(intArr);
        })
        .forEach(number -> System.out.println(number));
    }
}
```

[실행결과]

java8
lambda
stream
mapping

10
20
30
40
50
60

16.5.2 mapXXX() 메서드

mapXXX() 메서드는 요소를 대체하는 요소로 구성된 새로운 스트림을 리턴 한다. 아래 예제는 학생 List에서 학생의 점수를 요소로 하는 새로운 스트림을 생성하고, 점수를 순차적으로 콘솔에 출력한다.

[Sample.java] 다른 요소로 대체

```
public class Sample {
    public static void main(String[] args) {
        List<Student> studentList = Arrays.asList(
            new Student("홍길동", 10),
            new Student("신용권", 20),
            new Student("유미선", 30)
        );

        studentList.stream()
            .mapToInt(Student::getScore)
            .forEach(score -> System.out.println(score));
    }
}
```

[실행결과]

10
20
30

[Student.java] 학생 클래스

```
public class Student {
    private String name;
    private int score;

    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }

    public String getName() {return name;}
    public int getScore() {return score;}
}
```

16.5.3 asDoubleStream(), asLongStream(), boxed() 메서드

asDoubleStream() 메서드는 IntStream의 int 요소 또는 LongStream의 long 요소를 double 요소로 타입 변환해서 DoubleStream을 생성한다. 마찬가지로 asLongStream() 메서드는 IntStream의 int 요소를 long 요소로 타입 변환해서 LongStream을 생성한다.

[Sample.java] 다른 요소로 대체

```
public class Sample {
    public static void main(String[] args) {
        int[] intArray = {1, 2, 3, 4, 5};

        IntStream intStream = Arrays.stream(intArray);
        intStream
            .asDoubleStream() //DoubleStream 생성
            .forEach(d -> System.out.println(d));

        System.out.println();

        intStream = Arrays.stream(intArray);
        intStream
            .boxed() //Stream<Integer> 생성
            .forEach(obj -> System.out.println(obj.intValue()));
    }
}
```

[실행결과]

1.0
2.0
3.0
4.0
5.0

1
2
3
4
5

16.6 정렬 (sorted())

스트림은 요소가 최종 처리되기 전에 중간 단계에서 요소를 정렬해서 최종 처리 순서를 변경할 수 있다. 아래는 점수를 기준으로 Student 요소를 오름차순으로 정렬하기 위해 Comparable을 구현했다.

[Student.java] 정렬 가능한 클래스

```
public class Student implements Comparable<Student> {
    private String name;
    private int score;
    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
    public String getName() {return name;}
    public int getScore() {return score;}

    public int compareTo(Student o) {
        return Integer.compare(score, o.score); //score<o.score:음수리턴 score==o.score:0리턴 score>o.score:양수리턴
    }
}
```

아래 예제를 보면 숫자 요소일 경우에는 오름차순으로 정렬한 후 출력했다. Student 요소일 경우에는 Student의 기본 비교(Comparable) 방법을 이용해서 점수를 기준으로 오름차순으로 정렬한 후 출력했다. 그리고 Comparator로 점수를 기준으로 내림차순으로 정렬한 후 출력했다.

[Sample.java] 정렬

```
public class Sample {
    public static void main(String[] args) {
        //숫자 요소일 경우
        IntStream intStream = Arrays.stream(new int[] {5, 3, 2, 1, 4});
        intStream.sorted().forEach(n -> System.out.print(n + ","));
        System.out.println();

        //객체 요소일 경우
        List<Student> studentList = Arrays.asList(
            new Student("홍기동", 30),
            new Student("신용권", 10),
            new Student("유미선", 20)
        );

        //점수를 기준으로 오름차순으로 Student 정렬
        studentList.stream().sorted().forEach(s -> System.out.print(s.getScore() + ","));
        System.out.println();

        //점수를 기준으로 내림차순으로 Student 정렬
        studentList.stream().sorted(Comparator.reverseOrder()).forEach(s -> System.out.print(s.getScore() + ","));
    }
}
```

[실행결과]

1,2,3,4,5,
10,20,30,
30,20,10,

16.7 루핑 (peek(), forEach())

루핑(looping)은 요소 전체를 반복하는 것을 말한다. 루핑하는 메서드에는 peek(), forEach()가 있다. 이 두 메서드는 루핑한다는 기능에서 동일하지만, 동작 방식은 다르다. peek()은 중간 처리 메서드이고, forEach()는 최종 처리 메서드이다.

[Sample.java] 루핑

```
public class Sample {
    public static void main(String[] args) {
        int[] intArr = {1, 2, 3, 4, 5};
        System.out.println("[peek()를 마지막에 호출한 경우]");
        Arrays.stream(intArr).filter(a -> a%2 == 0).peek(n -> System.out.println(n)); //동작하지 않음
        System.out.println("[최종 처리 메서드를 마지막에 호출한 경우]");
        int total = Arrays.stream(intArr).filter(a -> a%2 == 0).peek(n -> System.out.println(n)).sum(); //동작함
        System.out.println("총합:" + total);
        System.out.println("[forEach()를 마지막에 호출한 경우]");
        Arrays.stream(intArr).filter(a -> a%2 == 0).forEach(n -> System.out.println(n)); //최종 메서드로 동작함
    }
}
```

16.8 매칭 (allMatch(), anyMatch(), noneMatch())

스트림 클래스는 최종단계에서 요소들이 특정조건에 만족하는지 세 가지 매칭 메서드를 제공하고 있다. allMatch() 메서드는 모든 요소들이 매개값으로 주어진 Predicate의 조건을 만족하는지 조사하고, anyMatch() 메서드는 최소한 한 개의 요소가 매개값으로 주어진 Predicate의 조건을 만족하는지 조사한다. 그리고 noneMatch()는 모든 요소들이 매개값으로 주어진 Predicate의 조건을 만족하지 않는지 조사한다.

[Sample.java] 매칭

```
public class Sample {
    public static void main(String[] args) {
        int[] intArr = {2, 4, 6};

        boolean result = Arrays.stream(intArr).allMatch(a -> a%2 == 0);
        System.out.println("모두 2의 배수인가?" + result);

        result = Arrays.stream(intArr).anyMatch(a -> a%3 == 0);
        System.out.println("하나라도 3의 배수가 있는가?" + result);

        result = Arrays.stream(intArr).noneMatch(a -> a%3 == 0);
        System.out.println("3의 배수가 없는가?" + result);
    }
}
```

[실행결과]

모두 2의 배수인가?true
하나라도 3의 배수가 있는가?true
3의 배수가 없는가?false

16.9 기본 집계 (sum(), count(), average(), max(), min())

16.9.1 스트림이 제공하는 기본 집계

집계 메서드에서 리턴 하는 OptionalXXX는 자바 8에서 추가한 Optional, OptionalDouble, OptionalInt, OptionalLong 클래스 타입을 말한다. 이들은 값을 저장하는 값 기반 클래스(value-based class)들이다. 이 객체에서 값을 얻기 위해서 get(), getAsDouble(), getAsInt(), getAsLong()을 호출하면 된다.

[Sample.java] 집계

```
public class Sample {
    public static void main(String[] args) {
        long count = Arrays.stream(new int[] {1, 2, 3, 4, 5}).filter(n -> n%2 == 0).count();
        System.out.println("2의 배수 개수:" + count);
        long sum = Arrays.stream(new int[] {1, 2, 3, 4, 5}).filter(n -> n%2 == 0).sum();
        System.out.println("2의 배수의 합:" + sum);
        double avg = Arrays.stream(new int[] {1, 2, 3, 4, 5}).filter(n -> n%2 == 0).average().getAsDouble();
        System.out.println("2의 배수의 평균:" + avg);
        int max = Arrays.stream(new int[] {1, 2, 3, 4, 5}).filter(n -> n%2 == 0).max().getAsInt();
        System.out.println("최대값:" + max);
        int min = Arrays.stream(new int[] {1, 2, 3, 4, 5}).filter(n -> n%2 == 0).min().getAsInt();
        System.out.println("최소값:" + min);
        int first = Arrays.stream(new int[] {1, 2, 3, 4, 5}).filter(n -> n%3 == 0).findFirst().getAsInt();
        System.out.println("첫번째 3의 배수:" + first);
    }
}
```

[실행결과]

2의 배수 개수:2
2의 배수의 합:6
2의 배수의 평균:3.0
최대값:4
최소값:2
첫번째 3의 배수:3

16.9.2 Optional 클래스

Optional 클래스는 단순히 집계 값만 저장하는 것이 아니라, 집계 값이 존재하지 않을 경우 디폴트 값을 설정할 수도 있고, 집계 값을 처리하는 Consumer도 등록할 수 있다.

[Sample.java] 집계

```
public class Sample {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        // 예외 발생(java.util.NoSuchElementException)
        //double avg = list.stream().mapToInt(Integer :: intValue).average().getAsDouble();

        //방법1
        OptionalDouble optional = list.stream().mapToInt(Integer :: intValue).average();
        if( optional.isPresent() ) {
            System.out.println("방법1_평균:" + optional.getAsDouble());
        }else{
            System.out.println("방법1_평균: 0.0");
        }
        //방법2
        double avg = list.stream()
            .mapToInt(Integer :: intValue)
            .average()
            .orElse(0.0);
        System.out.println("방법2_평균:" + avg);
        //방법3
        list.stream()
            .mapToInt(Integer :: intValue)
            .average()
            .ifPresent(a -> System.out.println("방법3_평균:" + a));
    }
}
```

[실행결과]

방법1_평균 : 0.0
방법2_평균 : 0.0

16.10 커스텀 집계 (reduce())

스트림은 기본 집계 메서드인 sum(), average(), count(), max(), min()와 집계 결과물을 만들 수 있도록 reduce() 메서드도 제공한다.

[Sample.java] reduce() 메서드 이용

```
public class Sample {
    public static void main(String[] args) {
        List<Student> studentList = Arrays.asList(
            new Student("홍길동", 92),
            new Student("신용권", 95),
            new Student("김자바", 88)
        );

        int sum1 = studentList.stream().mapToInt(Student :: getScore).sum();
        int sum2 = studentList.stream().map(Student :: getScore).reduce((a, b) -> a+b).get();
        int sum3 = studentList.stream().map(Student :: getScore).reduce(0, (a, b) -> a+b);

        System.out.println("sum1:" + sum1);
        System.out.println("sum2:" + sum2);
        System.out.println("sum3:" + sum3);
    }
}
```

[실행결과]

sum1:275
sum2:275
sum3:275

[Student.java] 학생 클래스

```
public class Student {
    private String name;
    private int score;

    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
    public String getName() { return name; }
    public int getScore() { return score; }
}
```

16.11 수집 (collect())

스트림은 요소들을 필터링 또는 매핑한 후 요소들을 수집하는 최종 처리 메서드인 collect()를 제공하고 있다. 이 메서드를 이용하면 필요한 요소만 컬렉션으로 담을 수 있고, 요소들을 그룹핑한 후 집계(리덕션)할 수 있다.

16.11.1 필터링한 요소 수집

Stream의 collect(Collector<T,A,R> collector) 메서드는 필터링 또는 매핑된 요소들을 새로운 컬렉션에 수집하고, 이 컬렉션을 리턴 한다. 매개값인 Collector(수집기)는 어떤 요소를 어떤 컬렉션에 수집할 것인지를 결정한다. 그리고 R은 요소가 저장될 컬렉션이다. 풀어서 해석하면 T 요소를 A 누적이 R에 저장한다는 의미이다.

[Sample.java] 필터링에서 새로운 컬렉션 생성

```
public class Sample {
    public static void main(String[] args) {
        List<Student> totalList = Arrays.asList(
            new Student("홍길동", 10, Student.Sex.MALE),
            new Student("김수애", 6, Student.Sex.FEMALE),
            new Student("신용권", 10, Student.Sex.MALE),
            new Student("박수미", 6, Student.Sex.FEMALE)
        );

        //남학생들만 묶어 List 생성
        List<Student> maleList = totalList.stream()
            .filter(s -> s.getSex() == Student.Sex.MALE)
            .collect(Collectors.toList());

        maleList.stream()
            .forEach(s -> System.out.println(s.getName()));

        System.out.println();

        //여학생들만 묶어 HashSet 생성
        Set<Student> femaleSet = totalList.stream()
            .filter(s -> s.getSex() == Student.Sex.FEMALE)
            .collect(Collectors.toCollection(HashSet :: new));

        femaleSet.stream()
            .forEach(s -> System.out.println(s.getName()));
    }
}
```

[실행결과]

홍길동
신용권

김수애
박수미

[Students.java] 학생 클래스

```
public class Student {
    public enum Sex {MALE, FEMALE}
    public enum City {Seoul, Pusan}

    private String name;
    private int score;
    private Sex sex;
    private City city;

    public Student(String name, int score, Sex sex) {
        this.name = name;
        this.score = score;
        this.sex = sex;
    }

    public Student(String name, int score, Sex sex, City city) {
        this.name = name;
        this.score = score;
        this.sex = sex;
        this.city = city;
    }

    public String getName() { return name; }
    public int getScore() { return score; }
    public Sex getSex() { return sex; }
    public City getCity() { return city; }
}
```


16.11.2 사용자 정의 컨테이너에 수집하기

아래 예제는 순차 처리를 이용해서 사용자 정의 객체에 요소를 수집하는 것을 보여준다.

[MainStudent.java] 남학생이 저장되는 컨테이너

```
public class MaleStudent {
    private List<Student> list; //요소를 저장할 컬렉션
    public MaleStudent() {
        list = new ArrayList<Student>();
        System.out.println("[ " + Thread.currentThread().getName() + " ] MaleStudent()");
    }
    public void accumulate(Student student) { //요소를 수집하는 메서드
        list.add(student);
        System.out.println("[ " + Thread.currentThread().getName() + " ] accumulate()");
    }
    public void combine(MaleStudent other) { //두 MaleStudent를 결합하는 메서드(병렬 처리 시에만 호출)
        list.addAll(other.getList());
        System.out.println("[ " + Thread.currentThread().getName() + " ] combine()");
    }
    public List<Student> getList() { //요소가 저장된 컬렉션을 리턴
        return list;
    }
}
```

[Sample.java] 남학생을 MaleStudent에 누적

```
public class Sample {
    public static void main(String[] args) {
        List<Student> totalList = Arrays.asList(
            new Student("홍길동", 10, Student.Sex.MALE),
            new Student("김수애", 6, Student.Sex.FEMALE),
            new Student("신용권", 10, Student.Sex.MALE),
            new Student("박수미", 6, Student.Sex.FEMALE) );

        MaleStudent maleStudent = totalList.stream().filter(s -> s.getSex() == Student.Sex.MALE)
            .collect(MaleStudent :: new, MaleStudent :: accumulate, MaleStudent :: combine);

        maleStudent.getList().stream().forEach(s -> System.out.println(s.getName()));
    }
}
```

[실행결과]

```
[main] MaleStudent()
[main] accumulate()
[main] accumulate()
홍길동
신용권
```

16.11.3 요소를 그룹핑해서 수집

collect()는 단순히 요소를 수집하는 기능 이외에 컬렉션의 요소들을 그룹핑 해서 Map 객체를 생성하는 기능도 제공한다. collect()를 호출할 때 Collectors의 groupingBy() 또는 groupingByConcurrent()가 리턴 하는 Collector를 매개값으로 대입하면 된다.

[Sample.java] 성별로 그룹핑하기

```
public class Sample {
    public static void main(String[] args) {
        List<Student> totalList = Arrays.asList(
            new Student("홍길동", 10, Student.Sex.MALE, Student.City.Seoul),
            new Student("김수애", 6, Student.Sex.FEMALE, Student.City.Pusan),
            new Student("신용권", 10, Student.Sex.MALE, Student.City.Pusan),
            new Student("박수미", 6, Student.Sex.FEMALE, Student.City.Seoul)
        );

        Map<Student.Sex, List<Student>> mapBySex = totalList.stream().collect(Collectors.groupingBy(Student :: getSex));
        System.out.print("[남학생]");
        mapBySex.get(Student.Sex.MALE).stream().forEach(s -> System.out.print(s.getName() + " "));
        System.out.print("\n[여학생]");
        mapBySex.get(Student.Sex.FEMALE).stream().forEach(s -> System.out.print(s.getName() + " "));
        System.out.println();

        Map<Student.City, List<String>> mapByCity = totalList.stream().collect(
            Collectors.groupingBy(Student::getCity, Collectors.mapping(Student::getName, Collectors.toList())));

        System.out.print("\n[서울]");
        mapByCity.get(Student.City.Seoul).stream().forEach(s->System.out.print(s + " "));
        System.out.print("\n[부산]");
        mapByCity.get(Student.City.Pusan).stream().forEach(s->System.out.print(s + " "));
    }
}
```

[실행결과]

```
[남학생]홍길동 신용권
[여학생]김수애 박수미

[서울]홍길동 박수미
[부산]김수애 신용권
```

16.11.4 그룹핑 후 매핑 및 집계

아래는 학생들을 성별로 그룹핑 한 다음 같은 그룹에 속하는 학생들의 평균 점수를 구하고, 성별을 키로, 평균 점수를 값을 갖는 Map을 생성한다.

[Sample.java] 그룹핑 후 리덕션

```
public class Sample {
    public static void main(String[] args) {
        List<Student> totalList = Arrays.asList(
            new Student("홍길동", 10, Student.Sex.MALE),
            new Student("김수애", 6, Student.Sex.FEMALE),
            new Student("신용권", 10, Student.Sex.MALE),
            new Student("박수미", 6, Student.Sex.FEMALE)
        );

        //성별로 평균 점수를 저장하는 Map 얻기
        Map<Student.Sex, Double> mapBySex = totalList.stream()
            .collect(Collectors.groupingBy(Student :: getSex, Collectors.averagingDouble(Student :: getScore)));
        System.out.println("남학생 평균 점수:" + mapBySex.get(Student.Sex.MALE));
        System.out.println("여학생 평균 점수:" + mapBySex.get(Student.Sex.FEMALE));

        //성별을 키포로 구분한 이름을 저장하는 Map 얻기
        Map<Student.Sex, String> mapByName = totalList.stream()
            .collect(Collectors.groupingBy(Student :: getSex, Collectors.mapping(Student :: getName, Collectors.joining(","))));
        System.out.println("남학생 전체 이름:" + mapByName.get(Student.Sex.MALE));
        System.out.println("여학생 전체 이름:" + mapByName.get(Student.Sex.FEMALE));
    }
}
```

[실행결과]

```
남학생 평균 점수:10.0
여학생 평균 점수:6.0
남학생 전체 이름:홍길동,신용권
여학생 전체 이름:김수애,박수미
```

16.12 병렬 처리

16.12.1 동시성(Concurrency)과 병렬성(Parallelism)

동시성과 병렬성은 멀티 스레드의 동작 방식이라는 점에서는 동일하지만 서로 다른 목적을 가지고 있다. 동시성은 멀티 작업을 위해 멀티 스레드가 번갈아가며 실행하는 성질을 말하고, 병렬성은 멀티 작업을 위해 멀티 코어를 이용해서 동시에 실행하는 성질을 말한다. 싱글 코어 CPU를 이용한 멀티 작업은 병렬적으로 실행되는 것처럼 보이지만, 번갈아가며 실행하는 동시성 작업이다.

16.12.2 포크조인(ForkJoin) 프레임워크

병렬 스트림은 요소들을 병렬 처리하기 위해 포크조인(ForkJoin) 프레임워크(Framework)를 사용한다. 병렬 스트림을 이용하면 런타임 시에 포크조인 프레임워크가 동작하는데, 포크 단계에서는 전체 데이터를 서브 데이터로 분리한다. 그다음에 서브 데이터를 멀티 코어에서 병렬로 처리한다. 조인 단계에서는 서브 결과를 결합해서 최종 결과를 만들어 낸다.

16.12.3 병렬 스트림 생성

병렬 처리를 위해 코드에서 포크조인 프레임워크를 직접 생성할 수는 있지만, 병렬 스트림을 이용할 경우에는 백그라운드에서 포크조인 프레임워크가 사용되기 때문에 개발자는 매우 쉽게 병렬 처리를 할 수 있다.

[Sample.java] 남학생을 MaleStudent에 누적

```
public class Sample {
    public static void main(String[] args) {
        List<Student> totalList = Arrays.asList(
            new Student("홍길동", 10, Student.Sex.MALE),
            new Student("김수애", 6, Student.Sex.FEMALE),
            new Student("신용권", 10, Student.Sex.MALE),
            new Student("박수미", 6, Student.Sex.FEMALE)
        );

        MaleStudent maleStudent = totalList.parallelStream()
            .filter(s -> s.getSex() == Student.Sex.MALE)
            .collect(MaleStudent :: new, MaleStudent :: accumulate, MaleStudent
                :: combine);

        maleStudent.getList().stream()
            .forEach(s -> System.out.println(s.getName()));
    }
}
```

[실행결과]

```
[ForkJoinPool.commonPool-worker-3] MaleStudent()
[ForkJoinPool.commonPool-worker-1] MaleStudent()
[ForkJoinPool.commonPool-worker-2] MaleStudent()
[main] MaleStudent()
[ForkJoinPool.commonPool-worker-3] accumulate()
[main] accumulate()
[main] combine()
[ForkJoinPool.commonPool-worker-3] combine()
[ForkJoinPool.commonPool-worker-3] combine()
홍길동
신용권
```

16.12.4 병렬 처리 성능

아래 예제는 work() 메서드의 실행시간을 조정함으로써 순차 처리와 병렬 처리 중 어떤 것이 전체 요소를 빨리 처리하는지 테스트한다.

[Sample.java] 순차 처리와 병렬 처리 성능 비교

```
public class Sample {
    public static void work(int value) {
        try{Thread.sleep( 100 );} catch(InterruptedException e){ } //100: 값이 작을수록 처리가 빠름
    }
    public static long testSequencial(List<Integer> list) { //순차 처리
        long start = System.nanoTime();
        list.stream().forEach((a) -> work(a));
        long end = System.nanoTime();
        long runTime = end - start;
        return runTime;
    }
    public static long testParallel(List<Integer> list) { //병렬 처리
        long start = System.nanoTime();
        list.stream().parallel().forEach((a) -> work(a));
        long end = System.nanoTime();
        long runTime = end - start;
        return runTime;
    }
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7, 8, 9); //소스 컬렉션
        long timeSequencial = testSequencial(list); //순차 스트림 처리 시간 구하기
        long timeParallel = testParallel(list); //병렬 스트림 처리 시간 구하기
        if(timeSequencial < timeParallel) {
            System.out.println("성능 테스트 결과:순차 처리가 더 빠름");
        }else {
            System.out.println("성능 테스트 결과:병렬 처리가 더 빠름");
        }
    }
}
```

[실행결과]

성능 테스트 결과:병렬 처리가 더 빠름

아래 예제는 스트림 소스가 ArrayList인 경우와 LinkedList일 경우 대용량 데이터의 병렬 처리 성능을 테스트한 것이다.

[Sample.java] 스트림 소스와 병렬 처리 성능

```
public class Sample {
    public static void work(int value){ }
    //병렬 처리
    public static long testParallel(List<Integer> list) {
        long start = System.nanoTime();
        list.stream().parallel().forEach((a) -> work(a));
        long end = System.nanoTime();
        long runTime = end - start;
        return runTime;
    }
    public static void main(String[] args) {
        //소스 컬렉션
        List<Integer> arrayList = new ArrayList<Integer>();
        List<Integer> linkedList = new LinkedList<Integer>();
        for(int i=0; i<1000000; i++) {
            arrayList.add(i);
            linkedList.add(i);
        }
        //워밍업
        long arrayListParallel = testParallel(arrayList);
        long linkedListParallel = testParallel(linkedList);
        //순차 스트림 처리 시간 구하기
        arrayListParallel = testParallel(arrayList);
        linkedListParallel = testParallel(linkedList);
        if(arrayListParallel < linkedListParallel) {
            System.out.println("성능 테스트 결과:ArrayList 처리가 더 빠름");
        }else {
            System.out.println("성능 테스트 결과:LinkedList 처리가 더 빠름");
        }
    }
}
```

[실행결과]

성능 테스트 결과:ArrayList 처리가 더 빠름

17장. JavaFX

17.1 JavaFX 개요

JavaFX는 자바로 윈도우용 GUI 응용프로그램을 만들기 위한 라이브러리로 기존의 AWT와 Swing을 대체 하는 라이브러리이다. JavaFX 애플리케이션에서 UI생성, 이벤트 처리, 멀티미디어 재생, 웹 뷰 등과 같은 기능은 JavaFX API로 개발하고 그 이외의 기능은 자바 표준 API를 활용해서 개발할 수 있다. JavaFX는 화면 레이아웃과 스타일, 애플리케이션 로직을 분리할 수 있기 때문에 디자이너와 개발자들이 협력해서 JavaFX 애플리케이션을 개발할 수 있는 구조를 가지고 있다.

17.2 JavaFX 애플리케이션 개발시작

JavaFX API를 사용하기위해 Eclipse를 실행하고, [Help]-[Eclipse Marketplace]에서 e(fx)clipse 로 검색해서 플러그인설치 한다.

17.2.1 메인클래스

[Sample.java] 윈도우 생성

```
import javafx.application.Application;
import javafx.stage.Stage;
public class Sample extends Application{
    public static void main(String[] args) {
        launch(args); //Sample 객체 생성 및 메인 윈도우 생성
    }

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.show(); //윈도우 보여주기
    }
}
```

17.2.2 무대(Stage)와 장면(Scene)

JavaFX는 윈도우를 무대(Stage)로 표현한다. 무대는 한 번에 하나의 장면을 가질 수 있는데 JavaFX는 장면을 javafx.scene.Scene으로 표현한다. 메인 윈도우는 start() 메서드의 primaryStage 매개값으로 전달되지만 장면(Scene)은 직접 생성해야한다. 아래 예제는 Parent의 하위 클래스인 javafx.scene.layout.VBox를 이용해서 Scene을 생성하고 메인 윈도우(primaryStage)의 장면으로 설정했다. VBox에는 Label과 Button 컨트롤을 배치했다.

[Sample.java] Stage와 Scene

```
public class Sample extends Application{
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) throws Exception {
        VBox root = new VBox();
        root.setPrefWidth(350);
        root.setPrefHeight(150);
        root.setAlignment(Pos.CENTER);
        root.setSpacing(20);

        Label label = new Label();
        label.setText("Hello JavaFX");
        label.setFont(new Font(50));

        Button button = new Button();
        button.setText("확인");
        button.setOnAction(event->Platform.exit());

        root.getChildren().add(label);
        root.getChildren().add(button);

        Scene scene = new Scene(root); //VBox를 루트 컨테이너로 해서 Scene 생성

        primaryStage.setTitle("Sample Application");
        primaryStage.setScene(scene); //윈도우에 장면 설정
        primaryStage.show();
    }
}
```

[실행결과]



17.3 JavaFX 레이아웃

장면(Scene)에는 다양한 컨트롤이 포함되는데 이들을 배치하는 것이 레이아웃이다. 레이아웃을 작성하는 방법은 두 가지가 있다. 하나는 자바 코드로 작성하는 프로그램적 레이아웃이고 다른 하나는 FXML로 작성하는 선언적 레이아웃이다.

17.3.1 프로그램적 레이아웃

프로그램적 레이아웃이란 자바 코드로 UI 컨트롤을 배치하는 것을 말한다. 자바 코드에 익숙한 개발자들이 선호하는 방식으로 컨트롤 배치, 스타일 지정, 이벤트 처리 등을 모두 자바 코드로 작성한다. 레이아웃이 복잡해지면 이 방법은 코드까지 복잡해져 난해한 프로그램이 될 확률이 높다.

[Sample.java] 프로그램적 레이아웃

```
public class Sample extends Application{
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) throws Exception {
        HBox hbox = new HBox();
        hbox.setPadding(new Insets(10));
        hbox.setSpacing(10);

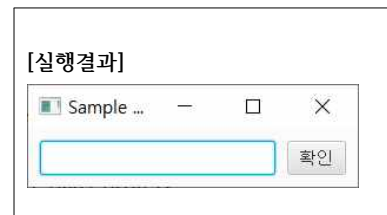
        TextField textField = new TextField();
        textField.setPrefWidth(200);

        Button button = new Button();
        button.setText("확인");

        ObservableList list = hbox.getChildren(); //HBox의 ObservableList 얻기
        list.add(textField);
        list.add(button);

        Scene scene = new Scene(hbox);

        primaryStage.setTitle("Sample Application");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```



17.3.2 FXML 레이아웃

FXML은 XML 기반의 마크업 언어로 JavaFX 애플리케이션의 UI 레이아웃을 자바 코드에서 분리해서 태그로 선언하는 방법을 제공한다. FXML 태그로 레이아웃을 정의하기 때문에 태그에 익숙한 디자이너와 협업이 가능하다. 또한 개발 완료 후 간단한 레이아웃 변경이나 스타일 변경이 필요한 경우에는 자바 소스를 수정할 필요 없이 FXML 태그만 수정하면 된다. 그리고 레이아웃이 비슷한 장면들 간에 재사용이 가능하다.

[ex04]-[src]-[ex04]-[root.fxml] FXML 레이아웃

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.HBox?>
<?import javafx.geometry.Insets?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.TextField?>

<HBox xmlns:fx="http://javafx.com/fxml">
    <padding>
        <Insets top="10" right="10" bottom="10" left="10"/>
    </padding>
    <spacing>10</spacing>

    <children>
        <TextField>
            <prefWidth>200</prefWidth>
        </TextField>

        <Button>
            <text>확인</text>
        </Button>
    </children>
</HBox>
```

[Sample.java] FXML 로딩

```
public class Sample extends Application{
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) throws Exception {
        Parent root = FXMLLoader.load(getClass().getResource("root.fxml"));
        Scene scene = new Scene(root);

        primaryStage.setTitle("Sample Application");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

17.4 JavaFX 컨테이너

레이아웃을 작성할 때 컨트롤들을 쉽게 배치할 수 있도록 도와주는 클래스가 컨테이너이다. javafx.scene.layout 패키지에는 다양한 컨테이너 클래스들이 존재한다. 접미사가 Pane으로 끝나는 클래스는 모두 컨테이너이고 그 외에는 Hbox, VBox가 있다.

17.4.1 AnchorPane 컨테이너

AnchorPane 컨테이너는 좌표를 이용하여 AnchorPane의 좌상단(0, 0)을 기준으로 컨트롤을 배치한다. 컨트롤 좌표는 좌상단 값을 말하는데 (0, 0)에서 떨어진 거리이다.

[ex04]-[src]-[ex04]-[anchorPan.fxml] AnchorPane 컨테이너

```
<AnchorPane xmlns:fx="http://javafx.com/fxml" prefHeight="150" prefWidth="400">
    <children>
        <Label layoutX="40" layoutY="30" text="아이디"/>
        <TextField layoutX="120" layoutY="30"/>

        <Label layoutX="40" layoutY="70" text="비밀번호"/>
        <PasswordField layoutX="120" layoutY="70"/>

        <Button layoutX="100" layoutY="110" text="로그인"/>
        <Button layoutX="200" layoutY="110" text="취소"/>
    </children>
</AnchorPane>
```



17.4.2 HBox와 VBox 컨테이너

아래 예제는 VBox로 ImageView 컨트롤과 HBox 컨테이너를 수직으로 배치하고 HBox 안에는 두 개의 버튼을 수평으로 배치했다.

[ex04]-[src]-[ex04]-[box.fxml] HBox와 VBox 컨테이너

```
<VBox xmlns:fx="http://javafx.com/fxml">
    <padding>
        <Insets bottom="10" left="10" right="10" top="10"/>
    </padding>
    <children>
        <ImageView fitWidth="200" preserveRatio="true"> //그림의 비율에 맞게 높이를 설정
            <image>
                <Image url = "@javafx.jpg"/> //현재 경로 기준으로 상대경로로 파일 지정
            </image>
        </ImageView>
        <HBox alignment="CENTER" spacing="20">
            <children>
                <Button text="이전"/>
                <Button text="다음"/>
            </children>
            <VBox.margin>
                <Insets top="10"/>
            </VBox.margin>
        </HBox>
    </children>
</VBox>
```

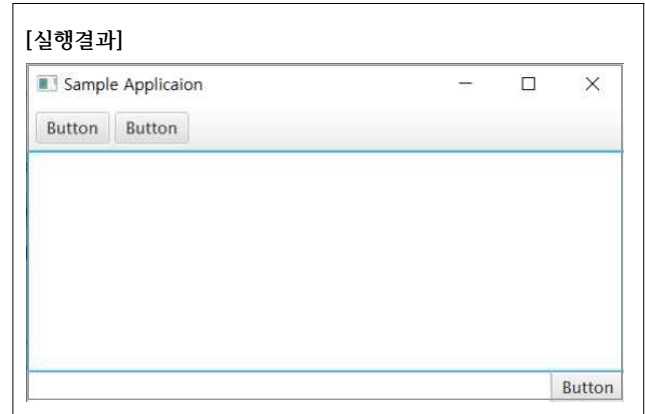


17.4.3 BorderPane 컨테이너

BorderPane은 top, bottom, left, right, center 셀에 컨트롤을 배치하는 컨테이너이다. 컨트롤만 배치하는 것이 아니라 다른 컨테이너도 배치할 수 있기 때문에 다양한 레이아웃을 만들어 낼 수 있다. 주의할 점은 각 셀에는 하나의 컨트롤 또는 컨테이너만 배치할 수 있다.

[borderPane.fxml] BorderPane 컨테이너

```
<BorderPane xmlns:fx="http://javafx.com/fxml/1">
  <top>
    <ToolBar>
      <items>
        <Button text="Button"/>
        <Button text="Button"/>
      </items>
    </ToolBar>
  </top>
  <center>
    <TextArea/> //left와 right까지 확장
  </center>
  <bottom>
    <BorderPane>
      <center>
        <TextField/> //top, bottom, left까지 확장
      </center>
      <right>
        <Button text="Button"/>
      </right>
    </BorderPane>
  </bottom>
</BorderPane>
```



17.4.4 FlowPane 컨테이너

FlowPane은 행으로 컨트롤을 배치하되 공간이 부족하면 새로운 행에 배치하는 컨테이너이다.

[flowPane.fxml] FlowPane 컨테이너

```
<FlowPane xmlns:fx="http://javafx.com/fxml" prefWidth="300" prefHeight="70" hgap="10" vgap="10">
  <padding>
    <Insets bottom="10" left="10" right="10" top="10"/>
  </padding>
  <children>
    <Button text="Button"/>
    <Button text="Button"/>
    <Button text="Button"/>
    <Button text="Button"/>
    <Button text="Button"/>
    <Button text="Button"/>
    <Button text="Button"/>
  </children>
</FlowPane>
```



17.4.5 TitlePane 컨테이너

TitlePane은 그리드로 컨트롤을 배치하되 고정된 셀(타일) 크기를 갖는 컨테이너이다.

[titlePane.fxml] TitlePane 컨테이너

```
<TitlePane xmlns:fx="http://javafx.com/fxml" prefTileHeight="100" prefTileWidth="100">
  <children>
    <Label text="딸기"/>
    <Label text="오렌지"/>
    <Label text="포도"/>
    <Label text="파인애플"/>
  </children>
</TitlePane>
```



17.4.6 GridPane 컨테이너

GridPane은 그리드로 컨트롤을 배치하되 셀의 크기가 고정적이지 않고 유동적인 컨테이너이다. 셀 병합이 가능해 다양한 입력폼 화면을 만들 때 유용하게 사용할 수 있다. 각 컨트롤은 자신이 배치될 행 인덱스와 컬럼 인덱스를 속성으로 가지며 몇 개의 셀을 병합할 것인지도 지정할 수 있다.

[GridPane.fxml] GridPane 컨테이너

```
<GridPane xmlns:fx="http://javafx.com/fxml" prefWidth="300" hgap="10" vgap="10">
  <padding>
    <Insets topRightBottomLeft="10"/>
  </padding>
  <children>
    <Label
      text="아이디"
      GridPane.rowIndex="0"
      GridPane.columnIndex="0"/>
    <TextField
      GridPane.rowIndex="0"
      GridPane.columnIndex="1"
      GridPane.hgrow="ALWAYS"/>
    <Label
      text="패스워드"
      GridPane.rowIndex="1"
      GridPane.columnIndex="0"/>
    <TextField
      GridPane.rowIndex="1"
      GridPane.columnIndex="1"
      GridPane.hgrow="ALWAYS"/>

    <HBox
      GridPane.rowIndex="2"
      GridPane.columnIndex="0"
      GridPane.columnSpan="2"
      GridPane.hgrow="ALWAYS"
      alignment="CENTER" spacing="20">
      <children>
        <Button text="로그인"/>
        <Button text="취소"/>
      </children>
    </HBox>
  </children>
</GridPane>
```



17.4.7 StackPane 컨테이너

StackPane은 컨트롤을 겹쳐 배치하는 컨테이너이다. 흔히 카드 레이아웃이라고 하는데 카드가 겹쳐 있는 것처럼 컨트롤도 겹쳐질 수 있다. 만약 위에 있는 컨트롤이 투명이라면 밑에 있는 컨트롤이 겹쳐 보이게 된다.

[StackPane.fxml] StackPane 컨테이너

```
<StackPane xmlns:fx="http://javafx.com/fxml">
  <children>
    <ImageView fitWidth="500" fitHeight="300">
      <image>
        <Image url="@javafx.jpg"/>
      </image>
    </ImageView>

    <Label text="StackPane">
      <font>
        <Font size="50"/>
      </font>
    </Label>
  </children>
</StackPane>
```



17.5 JavaFX 이벤트 처리

UI 애플리케이션은 사용자와 상호작용을 하면서 코드를 실행한다. 사용자가 UI 컨트롤을 사용하면 이벤트(event)가 발생하고 프로그램은 이벤트를 처리하기 위해 코드를 실행한다.

17.5.1 이벤트 핸들러(EventHandler)

아래는 프로그램적 레이아웃을 작성하고 버튼의 `ActionEvent`를 처리하는 것이다. 첫 번째 버튼은 직접 `EventHandler` 객체를 생성한 후 등록했고 두 번째 버튼은 람다식을 이용해서 `EventHandler`를 등록했다.

[Sample.java] `EventHandler`를 이용한 이벤트 처리

```
public class Sample extends Application{
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) throws Exception {
        HBox root = new HBox();
        root.setPrefSize(200, 50);
        root.setAlignment(Pos.CENTER); //수평 중앙 정렬
        root.setSpacing(20);

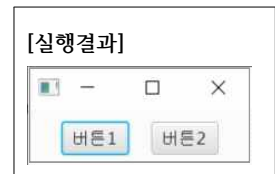
        Button btn1 = new Button("버튼1");

        btn1.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("버튼1 클릭");
            }
        });

        Button btn2 = new Button("버튼2");
        btn2.setOnAction(event->System.out.println("버튼2 클릭"));

        root.getChildren().addAll(btn1, btn2);
        Scene scene = new Scene(root);

        primaryStage.setTitle("Sample Application");
        primaryStage.setScene(scene);
        primaryStage.setOnCloseRequest(event->System.out.println("종료 클릭"));
        primaryStage.show();
    }
}
```



17.5.2 FXML 컨트롤러(Controller)

프로그램적 레이아웃은 레이아웃 코드와 이벤트 처리 코드를 모두 자바 코드로 작성해야 하므로 코드가 복잡하고 유지보수도 힘들어지며 디자이너와 협력해서 개발하는 것도 쉽지 않다. FXML 레이아웃은 FXML 파일당 별도의 컨트롤러(Controller)를 지정해서 이벤트를 처리할 수 있기 때문에 FXML 레이아웃과 이벤트 처리 코드를 완전히 분리할 수 있다.

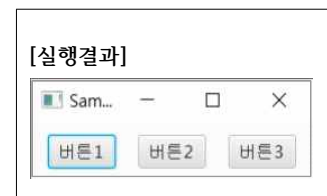
[Sample.java]

```
public class Sample extends Application{
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) throws Exception {
        Parent root = FXMLLoader.load(getClass().getResource("root.fxml"));

        Scene scene = new Scene(root);

        primaryStage.setTitle("Sample Applicaion");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```



FXML 파일의 루트 태그에서 fx:controller 속성으로 컨트롤러를 지정하면 UI컨트롤에서 발생하는 이벤트를 컨트롤러가 처리한다.

[root.fxml]

```
<HBox xmlns:fx="http://javafx.com/fxml" prefHeight="50" prefWidth="250" alignment="CENTER" spacing="20"
fx:controller="ex06.RootController">
  <children>
    <Button fx:id="btn1" text="버튼1"/>
    <Button fx:id="btn2" text="버튼2"/>
    <Button fx:id="btn3" text="버튼3"/>
  </children>
</HBox>
```

fx:id 속성을 가진 컨트롤들은 컨트롤러의 @FXML 어노테이션이 적용된 필드에 자동 주입된다. fx:id 속성값과 필드명은 동일해야 한다.

[RootController.java] EventHandler 생성 및 등록

```
public class RootController implements Initializable{
    @FXML private Button btn1;
    @FXML private Button btn2;
    @FXML private Button btn3;
    @Override
    public void initialize(URL location, ResourceBundle resources) {
        btn1.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("버튼1 클릭");
            }
        });
        btn2.setOnAction(event->handleBtn2Action(event));
        btn3.setOnAction(event->handleBtn3Action(event));
    }
    public void handleBtn2Action(ActionEvent event) {
        System.out.println("버튼2 클릭");
    }
    private void handleBtn3Action(ActionEvent event) {
        System.out.println("버튼3 클릭");
    }
}
```

람다식 이용

컨트롤에서 EventHandler를 생성하지 않고도 바로 이벤트 처리 메서드와 연결할 수 있는 방법이 있다. Button 컨트롤을 작성할 때 다음과 같이 onAction 속성값으로 "#메서드명"을 주면 내부적으로 EventHandler 객체가 생성되기 때문에 컨트롤러에서는 해당 메서드만 작성하면 된다.

```
<Button fx:id="btn" text="버튼" onAction="#handleBtnAction" />
```

17.6 JavaFX 속성 감시와 바인딩

JavaFX는 컨트롤의 속성을 감시하는 리스너를 설정할 수 있다. 예를 들어 Slider의 value 속성값을 감시하는 리스너를 설정해서 value 속성값이 변경되면 리스너가 다른 컨트롤러의 폰트나 이미지의 크기를 변경할 수 있다.

17.6.1 속성 감시

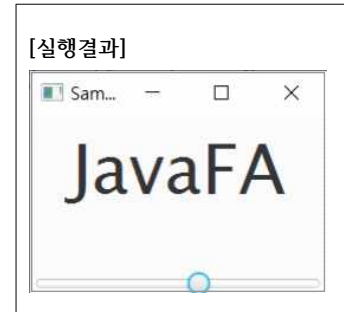
[slider.fxml] Label과 Slider 컨트롤 배치

```
<BorderPane xmlns:fx="http://javafx.com/fxml" prefHeight="250" prefWidth="350" fx:controller="ex06.SliderController">
  <center>
    <Label fx:id="label" text="JavaFA">
      <font>
        <Font size="0" />
      </font>
    </Label>
  </center>
  <bottom>
    <Slider fx:id="slider"/>
  </bottom>
</BorderPane>
```

//Label의 기본 폰트 크기는 0

```
public class SliderController implements Initializable{
    @FXML private Slider slider;
    @FXML private Label label;

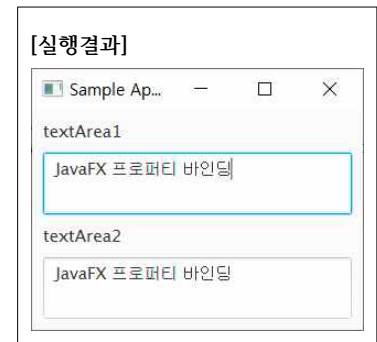
    @Override
    public void initialize(URL location, ResourceBundle resources) {
        slider.valueProperty().addListener(new ChangeListener<Number>() {
            @Override
            public void changed(ObservableValue<? extends Number> observable,
                               Number oldValue, Number newValue) {
                label.setFont(new Font( newValue.doubleValue() ));
            }
        });
    }
}
```



17.6.1 속성 감지

JavaFX 속성은 다른 속성과 바인딩될 수 있다. 바인딩된 속성들은 하나가 변경되면 자동적으로 다른 하나도 변경된다. 바인딩(bind()) 메서드는 단방향인데 textArea1에서 입력된 내용만 textArea2로 자동 입력되고 반대로 textArea2에서 입력된 내용은 textArea1로 자동 입력되지 않는다. 만약 역방향으로 바인딩하고 싶다면 bindBidirectional() 메서드를 이용하거나 Bindings.bindBidirectional() 메서드를 이용한다.

```
<VBox xmlns:fx="http://javafx.com/fxml"
    prefHeight="200" prefWidth="300" spacing="10"
    fx:controller="ex06.TextAreaController">
    <padding>
        <Insets bottom="10" left="10" right="10" top="10" />
    </padding>
    <children>
        <Label text="textArea1"/>
        <TextArea fx:id="textArea1"/>
        <Label text="textArea2"/>
        <TextArea fx:id="textArea2"/>
    </children>
</VBox>
```



```
public class TextAreaController implements Initializable{
    @FXML private TextArea textArea1;
    @FXML private TextArea textArea2;

    @Override
    public void initialize(URL arg0, ResourceBundle arg1) {
        Bindings.bindBidirectional(textArea1.textProperty(), textArea2.textProperty());
    }
}
```

17.7 JavaFX 컨트롤

JavaFX는 버튼 컨트롤, 입력 컨트롤, 뷰 컨트롤, 미디어 컨트롤, 차트 컨트롤 등 다양한 UI 컨트롤을 제공하고 있다.

17.7.1 버튼 컨트롤

버튼 컨트롤은 클릭할 수 있는 컨트롤로 ButtonBase를 상속하는 하위 컨트롤이다. Button, CheckBox, RadioButton, ToggleButton, Hyperlink 등이 있다. RadioButton 또는 ToggleButton 그룹 내에서 선택 변경을 감시하고 싶다면 ToggleGroup의 selectToggle 속성을 다음과 같이 작성하면 된다.

```
groupName.selectedToggleProperty().addListener(new ChangeListener<Toggle>() {
    @Override
    public void changed(ObservableValue<? extends Toggle> observable, Toggle oldValue, Toggle newValue) { ... }
});
```

[button.fxml] 버튼 컨트롤 배치와 이벤트 처리

```
<BorderPane xmlns:fx="http://javafx.com/fxml" prefHeight="150" prefWidth="420" fx:controller="ex06.ButtonController">
    <padding>
        <Insets bottom="10" left="10" right="10" top="10"/>
    </padding>

    <center>
        <HBox alignment="CENTER" prefHeight="100" prefWidth="200" spacing="10">
            <children>
                <VBox prefHeight="200" prefWidth="100" spacing="20" alignment="CENTER_LEFT">
                    <children>
                        <CheckBox fx:id="chk1" text="등산" onAction="#handleChkAction"/>
                        <CheckBox fx:id="chk2" text="독서" onAction="#handleChkAction"/>
                    </children>
                </VBox>
                <Label fx:id="label1" text=""/>

                <Separator orientation="VERTICAL" prefHeight="200"/>

                <VBox prefHeight="100" prefWidth="150" spacing="20" alignment="CENTER_LEFT">
                    <children>
                        <RadioButton fx:id="rad1" text="Male" userData="남" toggleGroup="$group">
                            <toggleGroup><ToggleGroup fx:id="group"/></toggleGroup>
                        </RadioButton>
                        <RadioButton fx:id="rad2" text="Female" userData="여" toggleGroup="$group" selected="true"/>
                    </children>
                </VBox>
                <Label fx:id="label2" text="여"/>
            </children>
        </HBox>
    </center>

    <bottom>
        <Button fx:id="btnExit" BorderPane.alignment="CENTER" text="닫기" onAction="#handleBtnExitAction"/>
    </bottom>
</BorderPane>
```

[ButtonController.java] 버튼 컨트롤 이벤트 처리

```
public class ButtonController implements Initializable{
    @FXML private CheckBox chk1;
    @FXML private CheckBox chk2;
    @FXML private Label label1;
    @FXML private Label label2;
    @FXML private ToggleGroup group;

    @Override
    public void initialize(URL location, ResourceBundle resources) {
        group.selectedToggleProperty().addListener(new ChangeListener<Toggle>() {
            @Override
            public void changed(ObservableValue<? extends Toggle> observable, Toggle oldValue, Toggle newValue) {
                label2.setText(newValue.getUserData().toString());
            }
        });
    }

    public void handleChkAction(ActionEvent e) {
        if(chk1.isSelected() && chk2.isSelected()) {
            label1.setText("등산,독서");
        } else if(chk1.isSelected()) {
            label1.setText("등산");
        } else if(chk2.isSelected()) {
            label1.setText("독서");
        } else {
            label1.setText("");
        }
    }

    public void handleBtnExitAction(ActionEvent e) {
        Platform.exit();
    }
}
```



17.7.2 입력 컨트롤

[textField.fxml] 입력 컨트롤 배치

```
<AnchorPane xmlns:fx="http://javafx.com/fxml" prefHeight="380" prefWidth="550" fx:controller="ex06.TextFieldController">
    <children>
        <Label layoutX="20" layoutY="40" text="제목"/>
        <TextField fx:id="txtTitle" layoutX="100" layoutY="35"/>

        <Label layoutX="20" layoutY="80" text="비밀번호"/>
        <PasswordField fx:id="txtPassword" layoutX="100" layoutY="75" prefHeight="25" prefWidth="120"/>

        <Label layoutX="20" layoutY="120" text="공개"/>
        <ComboBox fx:id="comboPublic" layoutX="100" layoutY="115" prefHeight="25" prefWidth="130" promptText="선택하세요">
            <items>
                <FXCollections fx:factory="observableArrayList" >
                    <String fx:value="공개" />
                    <String fx:value="비공개" />
                </FXCollections>
            </items>
        </ComboBox>
        <Label layoutX="250" layoutY="120" text="게시종료"/>
        <DatePicker fx:id="dateExit" layoutX="320" layoutY="115" promptText="날짜를 선택하세요"/>

        <Label layoutX="20" layoutY="160" text="내용"/>
        <TextArea fx:id="txtContent" layoutX="20" layoutY="150" prefHeight="150" prefWidth="500"/>

        <Separator layoutX="20" layoutY="320" prefHeight="0" prefWidth="500"/>

        <Button fx:id="btnReg" layoutX="200" layoutY="340" text="등록" onAction="#handleBtnRegAction"/>
        <Button fx:id="btnCancel" layoutX="280" layoutY="340" text="취소" onAction="#handleBtnCancelAction"/>
    </children>
</AnchorPane>
```

[TextFieldController.java] 입력값 얻기

```
public class TextFieldController implements Initializable {
    @FXML private TextField txtTitle;
    @FXML private PasswordField txtPassword;
    @FXML private ComboBox<String> comboPublic;
    @FXML private DatePicker dateExit;
    @FXML private TextArea txtContent;

    @Override
    public void initialize(URL location, ResourceBundle resources) {
    }

    public void handleBtnRegAction(ActionEvent e) {
        String title = txtTitle.getText();
        System.out.println("title:" + title);

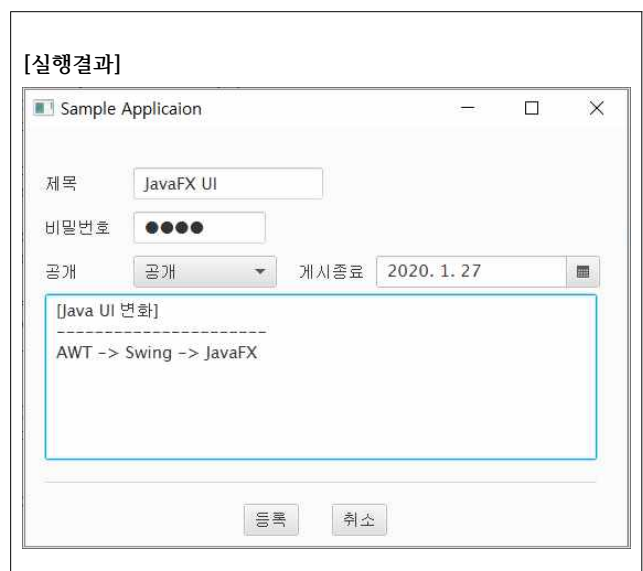
        String password = txtPassword.getText();
        System.out.println("password:" + password);

        String strPublic = comboPublic.getValue();
        System.out.println("public:" + strPublic);

        LocalDate localDate = dateExit.getValue();
        if(localDate != null) {
            System.out.println("dateExit:" + localDate.toString());
        }

        String content = txtContent.getText();
        System.out.println("content:" + content);
    }

    public void handleBtnCancelAction(ActionEvent e) {
        Platform.exit();
    }
}
```



17.7.3 뷰 컨트롤

뷰 컨트롤은 텍스트 또는 이미지 등을 보여주는데 목록 형태로 보여주는 ListView, 테이블 형태로 보여주는 TableView, 이미지를 보여주는 ImageView가 있다.

아래 예제는 ListView의 항목을 선택하면 같은 인덱스를 가지는 TableView의 행이 자동 선택되도록 한다. 그리고 TableView에서 행이 선택 되면 이미지 컬럼 값을 읽고 ImageView에 이미지를 보여준다. 하단의 [확인] 버튼을 클릭하면 ListView와 TableView에 선택된 정보를 콘솔에 출력한다.



[view.fxml] 뷰 컨트롤 배치

```
<AnchorPane xmlns:fx="http://javafx.com/fxml" prefHeight="200" prefWidth="600" fx:controller="ex06.ViewController">
  <children>
    <Label layoutX="10" layoutY="10" text="ListView"/>
    <ListView fx:id="listView" layoutX="10" layoutY="30" prefHeight="100" prefWidth="100"/>

    <Label layoutX="120" layoutY="10" text="TableView"/>
    <TableView fx:id="tableView" layoutX="120" layoutY="30" prefHeight="100" prefWidth="300">
      <columns>
        <TableColumn prefWidth="100" text="스마트폰"/>
        <TableColumn prefWidth="100" text="이미지"/>
      </columns>
    </TableView>

    <Label layoutX="450" layoutY="10" text="ImageView"/>
    <ImageView fx:id="imageView" fitHeight="100" fitWidth="60" layoutX="450" layoutY="30" preserveRatio="true"/>

    <Button layoutX="230" layoutY="150" text="확인" onAction="#handleBtnOkAction"/>
    <Button layoutX="300" layoutY="150" text="취소" onAction="#handleBtnCancelAction"/>
  </children>
</AnchorPane>
```

[Phone.java] 모델 클래스

```
public class Phone {
  private String smartPhone;
  private String image;

  public Phone(String smartPhone, String image) {
    this.smartPhone = smartPhone;
    this.image = image;
  }

  public String getSmartPhone() {
    return smartPhone;
  }

  public void setSmartPhone(String smartPhone) {
    this.smartPhone = smartPhone;
  }

  public String getImage() {
    return image;
  }

  public void setImage(String image) {
    this.image = image;
  }
}
```

```

public class ViewController implements Initializable{
    @FXML private ListView<String> listView;
    @FXML private TableView<Phone> tableView;
    @FXML private ImageView imageView;

    @Override
    public void initialize(URL arg0, ResourceBundle arg1) {
        listView.setItems(FXCollections.observableArrayList("갤럭시S1", "갤럭시S2", "갤럭시S3", "갤럭시S4", "갤럭시S5"));

        listView.getSelectionModel().selectedIndexProperty().addListener(new ChangeListener<Number>() {
            @Override
            public void changed(ObservableValue<? extends Number> observable, Number oldValue, Number newValue) {
                tableView.getSelectionModel().select(newValue.intValue());
                tableView.scrollTo(newValue.intValue());
            }
        });

        ObservableList<Phone> phoneList=FXCollections.observableArrayList(new Phone("갤럭시1", "phone01.png"),
            new Phone("갤럭시2", "phone02.png"), new Phone("갤럭시3", "phone03.png"),
            new Phone("갤럭시4", "phone04.png"), new Phone("갤럭시5", "phone05.png"));

        TableColumn tcSmartPhone = tableView.getColumns().get(0);
        tcSmartPhone.setCellValueFactory(new PropertyValueFactory("smartPhone"));
        tcSmartPhone.setStyle("-fx-alignment: CENTER;");

        TableColumn tcImage = tableView.getColumns().get(1);
        tcImage.setCellValueFactory(new PropertyValueFactory("image"));
        tcImage.setStyle("-fx-alignment: CENTER;");

        tableView.setItems(phoneList);

        tableView.getSelectionModel().selectedItemProperty().addListener(new ChangeListener<Phone>() {
            @Override
            public void changed(ObservableValue<? extends Phone> observable, Phone oldValue, Phone newValue) {
                if(newValue != null) {
                    Image image = new Image(getClass().getResource(newValue.getImage()).toString());
                    imageView.setImage(image);
                }
            }
        });

        public void handleBtnOkAction(ActionEvent e) {
            String item = listView.getSelectionModel().getSelectedItem();
            System.out.println("ListView 스마트폰: " + item);

            Phone phone = tableView.getSelectionModel().getSelectedItem();
            System.out.println("TableView 스마트폰: " + phone.getSmartPhone());
            System.out.println("TableView 이미지: " + phone.getImage());
        }

        public void handleBtnCancelAction(ActionEvent e) {
            Platform.exit();
        }
    }
}

```

17.8 JavaFX 컨트롤

UI 애플리케이션 개발을 할 경우 JavaFX도 메뉴바와 툴바를 생성할 수 있도록 MenuBar 컨트롤과 Toolbar 컨트롤을 제공한다.

17.8.1 MenuBar 컨트롤

```

<MenuBar>
  <menus>
    <Menu text="파일"> ... </Menu>
    <Menu text="편집"> ... </Menu>
  </menus>
</MenuBar>

```

17.8.2 Toolbar 컨트롤

```
<ToolBar>
  <items>
    <Button onAction="#handleNew">
      <graphic>
        <ImageView><image><Image url="@icons/new.png"/></image></ImageView>
      </graphic>
    </Button>
  </items>
</ToolBar>
```

17.9 JavaFX 다이얼로그

다이얼로그(Dialog)는 주 윈도우에서 알림 또는 사용자의 입력을 위해서 실행되는 서브 윈도우라고 볼 수 있다. 다이얼로그는 자체적으로 실행될 수 없고, 주 윈도우에 의해서 실행되는데 다이얼로그를 띄우는 주 윈도우를 다이얼로그의 소유자 윈도우라고 한다.

17.9.1 FileChooser, DirectoryChooser

FileChooser는 로컬 PC의 파일을 선택할 수 있는 다이얼로그이다. 열기 또는 저장 옵션으로 실행할 수 있고, 파일 확장자명을 필터링해서 원하는 파일만 볼 수 있다. 버튼이나 메뉴 아이템의 ActionEvent를 처리할 때 자바 코드로 FileChooser를 생성하고 showOpenDialog() 또는 showSaveDialog()를 호출해야 한다.

```
FileChooser fileChooser = new FileChooser();
fileChooser.getExtensionFilters().addAll(
    new ExtensionFilter("Text Files", "*.txt"),
    new ExtensionFilter("Image Files", "*.png", "*.jpg", "*.gif"),
    new ExtensionFilter("Audio Files", "*.wav", "*.mp3", "*.aac"),
    new ExtensionFilter("All Files", "*.*"));
File selectedFile = fileChooser.showOpenDialog(primaryStage);
String selectedFilePath = selectedFile.getPath();
```

디렉터리(폴더)를 선택하고 싶다면 FileChooser 대신 DirectoryChooser를 사용하면 된다. 방법은 FileChooser와 비슷하다.

```
DirectoryChooser directoryChooser = new DirectoryChooser();
File selectedDir = directoryChooser.showDialog(primaryStage);
String selectedDirPath = selectedDir.getPath();
```

17.9.2 Popup

Popup은 투명한 컨테이너를 제공하는 모달리스 다이얼로그이다. 따라서 소유자 윈도우는 계속 사용될 수 있다. Popup은 컨트롤러의 톨팁, 메시지 통지, 드롭다운 박스, 마우스 오른쪽 버튼을 클릭을 클릭했을 때 나타나는 메뉴 등을 만들 때 사용될 수 있다.

Popup 내용은 자바 코드로 작성하거나 FXML 파일로 작성할 수 있다. 다음을 FXML 파일을 로딩해서 Popup의 내용으로 추가하는 코드이다.

```
Popup popup = new Popup();
popup.getContent().add(FXMLLoader.load(getClass().getResource("popup.fxml")));
```

Popup을 실행하려면 다음과 같이 show() 메서드를 호출하면 된다.

```
popup.show(primaryStage)    //PC 화면 정중앙에서 실행
popup.show(primaryStage, anchorX, anchorY);    //지정된 좌표에서 실행
```

setAutoHide(true)로 설정했을 경우 다른 윈도우로 포커스를 이동하면 Popup은 자동으로 닫힌다.

```
popup.setAutoHide(true);
```


17.9.3 커스텀 다이얼로그

다양한 내용의 다이얼로그를 만들고 싶다면 Stage로 직접 생성하면 된다. Stage로 다이얼로그를 만들려면 다음과 같은 설정이 필요하다.

```
Stage dialog = new Stage(StageStyle.UTILITY);
dialog.initModality(Modality.WINDOW_MODAL);
dialog.initOwner(primaryStage);
```

아래 예제는 다섯 개의 버튼을 배치하고 지금까지 설명한 다이얼로그를 실행하도록 했다. 첫 번째 버튼은 파일 오픈 다이얼로그를, 두 번째 버튼은 파일 저장 다이얼로그를, 세 번째 버튼은 디렉터리 선택 다이얼로그를, 네 번째 버튼은 팝업을, 다섯 번째 버튼은 커스텀 다이얼로그를 실행한다.

[popup.fxml] 팝업 내용을 정의

```
<HBox xmlns:fx="http://javafx.com/fxml" alignment="CENTER_LEFT"
      style="-fx-background-color:black; -fx-background-radius:20;">
  <children>
    <ImageView id="imgMessage" fitHeight="30" fitWidth="30" preserveRatio="true"/>
    <Label id="lblMessage" style="-fx-text-fill:white;">
      <HBox.margin>
        <Insets left="5" right="5"/>
      </HBox.margin>
    </Label>
  </children>
</HBox>
```

[custom_dialog.fxml] 확인 다이얼로그 내용을 정의

```
<AnchorPane xmlns:fx="http://javafx.com/fxml" prefHeight="150" prefWidth="400">
  <children>
    <ImageView fitHeight="50" fitWidth="50" layoutX="15" layoutY="15" preserveRatio="true">
      <image><Image url="@dialog-info.png"/></image>
    </ImageView>
    <Button id="btnOk" layoutX="300" layoutY="100" text="확인"/>
    <Label id="txtTitle" layoutX="100" layoutY="30" prefHeight="15" prefWidth="300"/>
  </children>
</AnchorPane>
```

[Sample.java] 실행 클래스

```
public class Sample extends Application{
  public static void main(String[] args) {
    launch(args);
  }

  @Override
  public void start(Stage primaryStage) throws Exception {
    FXMLLoader loader = new FXMLLoader(getClass().getResource("view.fxml"));
    Parent root = loader.load();
    ViewController controller = loader.getController();
    controller.setPrimaryStage(primaryStage);

    Scene scene = new Scene(root);

    primaryStage.setTitle("Sample Applicaion");
    primaryStage.setScene(scene);
    primaryStage.show();
  }
}
```

[view.fxml] 선언적 레이아웃

```
<HBox xmlns:fx="http://javafx.com/fxml" alignment="TOP_LEFT" spacing="10" fx:controller="ex07.ViewController">
  <children>
    <Button text="Open FileChooser" onAction="#handleOpenFileChooser"/>
    <Button text="Save FileChooser" onAction="#handleSaveFileChooser"/>
    <Button text="DirectoryChooser" onAction="#handleDirectoryChooser"/>
    <Button text="Popup" onAction="#handlePopup"/>
    <Button text="Custom" onAction="#handleCustom"/>
  </children>
</padding>
  <Insets bottom="10" left="10" right="10" top="10"/>
</padding>
</HBox>
```



```

public class ViewController implements Initializable{
    @Override
    public void initialize(URL arg0, ResourceBundle arg1) {
    }

    private Stage primaryStage;
    public void setPrimaryStage(Stage primaryStage) {
        this.primaryStage = primaryStage;
    }

    public void handleOpenFileChooser(ActionEvent e) { //파일 열기 다이얼로그 실행
        FileChooser fileChooser = new FileChooser();
        fileChooser.getExtensionFilters().addAll(
            new ExtensionFilter("Text Files", "*.txt"),
            new ExtensionFilter("Image Files", "*.png", "*.jpg", "*.gif"),
            new ExtensionFilter("Audio Files", "*.wav", "*.mp3", "*.aac"),
            new ExtensionFilter("All Files", "*.*"));
        File selectedFile = fileChooser.showOpenDialog(primaryStage);
        if(selectedFile != null) {
            System.out.println(selectedFile.getPath());
        }
    }

    public void handleSaveFileChooser(ActionEvent e) { //파일 저장 다이얼로그 실행
        FileChooser fileChooser = new FileChooser();
        fileChooser.getExtensionFilters().add(new ExtensionFilter("All Files", "*.*"));
        File selectedFile = fileChooser.showSaveDialog(primaryStage);
        if(selectedFile != null) {
            System.out.println(selectedFile.getPath());
        }
    }

    public void handleDirectoryChooser(ActionEvent e) { //디렉토리 선택 다이얼로그 실행
        DirectoryChooser directoryChooser = new DirectoryChooser();
        File selectedDir = directoryChooser.showDialog(primaryStage);
        if(selectedDir != null) {
            System.out.println(selectedDir.getPath());
        }
    }

    public void handlePopup(ActionEvent e) throws Exception{ //popup 다이얼로그 실행
        Popup popup = new Popup();

        HBox hbox = (HBox)FXMLLoader.load(getClass().getResource("popup.fxml"));
        ImageView imageView = (ImageView)hbox.lookup("#imgMessage");
        imageView.setImage(new Image(getClass().getResource("dialog-info.png").toString()));
        imageView.setOnMouseClicked(event->popup.hide());
        Label lblMessage = (Label)hbox.lookup("#lblMessage");
        lblMessage.setText("메시지가 왔습니다.");

        popup.getContent().add(hbox);
        popup.setAutoHide(true);
        popup.show(primaryStage);
    }

    public void handleCustom(ActionEvent e) throws Exception{ //커스텀 다이얼로그 실행
        Stage dialog = new Stage(StageStyle.UTILITY);
        dialog.initModality(Modality.WINDOW_MODAL);
        dialog.initOwner(primaryStage);
        dialog.setTitle("확인");

        AnchorPane anchorPane = (AnchorPane)FXMLLoader.load(getClass().getResource("custom_dialog.fxml"));
        Label txtTitle = (Label)anchorPane.lookup("#txtTitle");
        txtTitle.setText("확인하셨습니다습니까?");
        Button btnOk=(Button)anchorPane.lookup("#btnOk");
        btnOk.setOnAction(event->dialog.close());

        Scene scene = new Scene(anchorPane);
        dialog.setScene(scene);
        dialog.setResizable(false);
        dialog.show();
    }
}

```

17.10 JavaFX 스레드 동시성

JavaFX UI는 스레드에 안전하지 않기 때문에 UI를 생성하고 변경하는 작업은 JavaFX Application Thread가 담당하고 다른 작업 스레드들은 UI를 생성하거나 변경할 수 없다. main 스레드가 Application의 launch() 메서드를 호출되면서 생성된 JavaFX Application Thread는 start() 메서드를 실행시키면서 모든 UI를 생성한다. 컨트롤에서 이벤트가 발생할 경우 컨트롤러의 이벤트 처리 메서드를 실행하는 것도 JavaFX Application Thread이다.

17.10.1 Task 클래스

Task는 작업 스레드에서 실행되는 하나의 작업을 표현한 추상 클래스이다. 하나의 작업을 정의할 때에는 Task를 상속해서 클래스를 생성한 후, call() 메서드를 재정의하면 된다. call() 메서드는 리턴값이 있는데 리턴값은 작업 결과를 말한다. 아래 예제는 작업 스레드가 0.1초 주기로 0부터 100까지 카운팅한 값을 ProgressBar의 progress 속성 및 Label의 text 속성과 바인딩해서 UI를 변경한다.

[progress.fxml] 컨트롤 배치

```
<AnchorPane xmlns:fx="http://javafx.com/fxml" prefHeight="150" prefWidth="400" fx:controller="ex07.ProgressController">
    <children>
        <ProgressBar fx:id="progressBar" layoutX="20" layoutY="20" prefWidth="350" progress="0"/>
        <Label layoutX="20" layoutY="60" text="진행정도:"/>
        <Label fx:id="lblWorkDone" layoutX="90" layoutY="60"/>
        <Button fx:id="btnStart" layoutX="120" layoutY="100" text="시작"/>
        <Button fx:id="btnStop" layoutX="200" layoutY="100" text="멈춤"/>
    </children>
</AnchorPane>
```

[ProgressController.java] 컨트롤러

```
public class ProgressController implements Initializable{
    @FXML private ProgressBar progressBar;
    @FXML private Label lblWorkDone;
    @FXML private Button btnStart;
    @FXML private Button btnStop;
    private Task<Void> task;

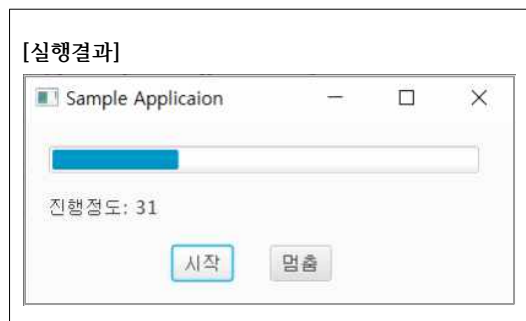
    @Override
    public void initialize(URL location, ResourceBundle resources) {
        btnStart.setOnAction(event->handleBtnStart(event));
        btnStop.setOnAction(event->handleBtnStop(event));
    }

    public void handleBtnStart(ActionEvent e) {
        task = new Task<Void>() {
            @Override
            protected Void call() throws Exception {
                for(int i=0; i<=100; i++) {
                    if(isCancelled()) {
                        updateMessage("취소됨");
                        break;
                    }
                    updateProgress(i, 100);
                    updateMessage(String.valueOf(i)); //Task의 progress와 message 속성을 업데이트
                    try{
                        Thread.sleep(100);
                    }catch(Exception e) {
                        if(isCancelled()) {
                            updateMessage("취소됨");
                            break;
                        }
                    }
                }
            }
        };
        return null;
    };
    progressBar.progressProperty().bind(task.progressProperty()); //속성 바인딩
    lblWorkDone.textProperty().bind(task.messageProperty());

    Thread thread = new Thread(task);
    thread.setDaemon(true);
    thread.start();

    public void handleBtnStop(ActionEvent e) {
        task.cancel(); //작업 취소
    }
}
```

[실행결과]



17.10.2 Service 클래스

Service 클래스는 작업 스레드상에서 Task를 간편하게 시작, 취소, 재시작할 수 있는 기능을 제공한다. 아래 예제는 0부터 100까지 합을 구하는 Service를 작성하고 작업 스레드에서 실행시킨다. 작업 완료 결과는 succeeded() 메서드에서 얻어 Label 컨트롤에 나타난다.

[service.fxml] 컨트롤 배치

```
<AnchorPane xmlns:fx="http://javafx.com/fxml" prefHeight="150" prefWidth="400" fx:controller="ex07.ServiceController">
    <children>
        <ProgressBar fx:id="progressBar" layoutX="20" layoutY="20" prefWidth="350" progress="0"/>
        <Label layoutX="20" layoutY="60" text="진행정도:"/>
        <Label fx:id="lblWorkDone" layoutX="90" layoutY="60"/>
        <Label layoutX="200" layoutY="60" text="작업결과:"/>
        <Label fx:id="lblResult" layoutX="270" layoutY="60"/>
        <Button fx:id="btnStart" layoutX="120" layoutY="100" text="시작"/>
        <Button fx:id="btnStop" layoutX="200" layoutY="100" text="멈춤"/>
    </children>
</AnchorPane>
```

[ServiceController.java] 컨트롤러

```
public class ServiceController implements Initializable{
    @FXML private ProgressBar progressBar;
    @FXML private Label lblWorkDone;
    @FXML private Label lblResult;
    @FXML private Button btnStart;
    @FXML private Button btnStop;
    private TimeService timeService;

    @Override
    public void initialize(URL location, ResourceBundle resources) {
        btnStart.setOnAction(event->handleBtnStart(event));
        btnStop.setOnAction(event->handleBtnStop(event));
        timeService = new TimeService();
        timeService.start();
    }
    public void handleBtnStart(ActionEvent e) {
        timeService.restart();
        lblResult.setText("");
    }
    public void handleBtnStop(ActionEvent e) {
        timeService.cancel();
    }

    class TimeService extends Service<Integer> {
        @Override
        protected Task<Integer> createTask() {
            Task<Integer> task = new Task<Integer>() {
                @Override
                protected Integer call() throws Exception {
                    int result = 0;
                    for(int i=0; i<=100; i++) {
                        if(isCancelled()) {
                            updateMessage("취소됨");
                            break;
                        }
                        result += i;
                        updateProgress(i, 100);
                        updateMessage(String.valueOf(i)); //Task의 progress와 message 속성을 업데이트
                        try{ Thread.sleep(100); }catch(Exception e) {
                            if(isCancelled()) {
                                updateMessage("취소됨");
                                break;
                            }
                        }
                    }
                    return result;
                }
            };
            progressBar.progressProperty().bind(task.progressProperty());
            lblWorkDone.textProperty().bind(task.messageProperty());
            return task;
        }
    }
}
```

```

@Override
protected void cancelled() { //작업이 취소되었을때 호출
    lblResult.setText(String.valueOf(getValue()));
}
@Override
protected void failed() { //작업 처리 도중 예외가 발생할 경우 호출
    lblResult.setText("실패");
}
@Override
protected void succeeded() { //성공적으로 작업이 완료되었을때 호출
    lblResult.setText(String.valueOf(getValue()));
}
}
}

```

17.11 화면 이동

화면을 이동하는 가장 쉬운 방법은 Stage에 새로운 Scene을 세팅하는 것이다. 메인 화면에서 로그인 화면으로 이동하기 위해 버튼을 클릭한 경우 컨트롤러의 이벤트 처리 메서드는 로그인 Scene을 생성하고, primaryStage의 setScene() 메서드로 메인 Scene을 로그인 Scene으로 변경할 수 있다.

[root.fxml] 메인 화면

```

<StackPane xmlns:fx="http://javafx.com/fxml" prefHeight="500" prefWidth="350" fx:controller="ex08.RootController">
  <children>
    <BorderPane>
      <top>
        <BorderPane style="-fx-background-color:#eaeaea;">
          <center>
            <Label alignment="CENTER" prefWidth="215" text="메인 화면"/>
          </center>
          <right>
            <Button fx:id="btnLogin" text="로그인"/>
          </right>
          <padding>
            <Insets bottom="10" left="10" right="10" top="10"/>
          </padding>
        </BorderPane>
      </top>
      <center>
        <VBox></VBox>
      </center>
    </BorderPane>
  </children>
</StackPane>

```

루트 컨테이너가 StackPne으로 되어 있고, 메인 화면은 BorderPane으로 추가되어있다. [로그인] 버튼을 클릭하면 로그인 화면으로 이동한다.

[RootController.java] 메인 화면 컨트롤러

```

public class RootController implements Initializable{
    @FXML private Button btnLogin;

    @Override
    public void initialize(URL location, ResourceBundle resources) {
        btnLogin.setOnAction(e->handleBtnLogin(e));
    }

    public void handleBtnLogin(ActionEvent event) {
        try {
            Parent login=FXMLLoader.load(getClass().getResource("login.fxml"));
            StackPane root = (StackPane)btnLogin.getScene().getRoot();
            root.getChildren().add(login);
        }catch(Exception e) {
            System.out.println(e.toString());
        }
    }
}

```

[login.fxml] 로그인 화면

```
<BorderPane xmlns:fx="http://javafx.com/fxml" fx:id="login" prefHeight="500" prefWidth="350"
            fx:controller="ex08.LoginController">
    <top>
        <BorderPane style="-fx-background-color:#eaeaea;">
            <left>
                <Button fx:id="btnMain" text="메인"/>
            </left>
            <center>
                <Label alignment="CENTER" prefWidth="215" text="로그인 화면"/>
            </center>
            <padding>
                <Insets bottom="10" left="10" right="10" top="10"/>
            </padding>
        </BorderPane>
    </top>

    <center>
        <VBox></VBox>
    </center>
</BorderPane>
```

로그인 화면의 좌측 상단에는 [메인] 버튼이 있는데, 이 버튼을 클릭하면 로그인 화면을 제거하고 메인 화면을 보여준다.

[LoginController.java] 로그인 화면 컨트롤러

```
public class LoginController implements Initializable{
    @FXML private BorderPane login;
    @FXML private Button btnMain;

    @Override
    public void initialize(URL location, ResourceBundle resources) {
        btnMain.setOnAction(e->handleBtnMain(e));
    }

    public void handleBtnMain(ActionEvent event) {
        try {
            StackPane root = (StackPane)btnMain.getScene().getRoot();
            root.getChildren().remove(login);
        }catch(Exception e) {
            System.out.println(e.toString());
        }
    }
}
```

[Sample.java] 메인 클래스

```
public class Sample extends Application{
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("Sample Application");
        Parent root=FXMLLoader.load(getClass().getResource("root.fxml"));
        Scene scene = new Scene(root);
        primaryStage.setScene(scene);
        primaryStage.setWidth(350); //윈도우의 고정 폭 설정
        primaryStage.setHeight(500); //윈도우의 고정 높이 설정
        primaryStage.setResizable(false); //윈도우 크기를 조정할 수 없도록 함
        primaryStage.show();
    }
}
```

18장. IO 기반 입출력 및 네트워킹

18.1 IO 패키지 소개

데이터는 사용자로부터 키보드를 통해 입력될 수도 있고, 파일 또는 네트워크로부터 입력될 수도 있다. 데이터는 반대로 모니터로 출력될 수도 있고, 파일로 출력되어 저장될 수도 있으며 네트워크로 출력되어 전송될 수도 있다.

자바에서 데이터는 스트림(Stream)을 통해 입출력되므로 스트림의 특징을 잘 이해해야 한다. 스트림은 단일 방향으로 연속적으로 흘러가는 것을 말하는데, 물이 높은 곳에서 낮은 곳으로 흐르듯이 데이터는 출발지에서 나와 도착지로 들어간다는 개념이다.

18.2 입력 스트림과 출력 스트림

프로그램이 출발지나 또는 도착지나에 따라서 스트림의 종류가 결정되는데, 프로그램이 데이터를 입력받을 때에는 입력 스트림(InputStream)이라 부르고, 프로그램이 데이터를 보낼 때에는 출력 스트림(OutputStream)이라고 부른다.

18.2.1 InputStream

InputStream은 바이트 기반, 입력 스트림의 최상위 클래스로 추상 클래스이다. 모든 바이트 기반 입력 스트림은 이 클래스를 상속받아서 만들어진다. FileInputStream, BufferedInputStream, DataInputStream 클래스는 모두 InputStream 클래스를 상속하고 있다.

- read() 메서드

read() 메서드는 입력 스트림으로부터 1바이트를 읽고 4바이트 int 타입으로 리턴한다. 따라서 리턴된 4바이트 중 끝의 1바이트에만 데이터가 들어 있다. 더 이상 입력 스트림으로부터 바이트를 읽을 수 없다면 read() 메서드는 -1을 리턴하는데 이것을 이용하면 읽을 수 있는 마지막 바이트까지 루프를 돌며 한 바이트씩 읽을 수 있다.

```
InputStream is = new FileInputStream("C:/test.jpg");
int readByte;
while((readByte = is.read()) != -1) { ... }
```

- read(byte[] b) 메서드

입력 스트림으로부터 100개의 바이트가 들어온다면 read() 메서드는 100번을 루핑해서 읽어 들여야 한다. 그러나 read(byte[] b) 메서드는 한번 읽을 때 매개값으로 주어진 바이트 배열 길이만큼 읽기 때문에 루핑 횟수가 현저히 줄어든다.

```
InputStream is = new FileInputStream("C:/test.jpg");
int readByteNo;
byte[] readBytes = new byte[100];
while((readByteNo = is.read(readBytes)) != -1) { ... }
```

- close() 메서드

InputStream을 더 이상 사용하지 않을 경우에는 close() 메서드를 호출해서 InputStream에서 사용했던 시스템 자원을 풀어준다.

```
is.close();
```

18.2.2 OutputStream

OutputStream은 바이트 기반 출력 스트림의 최상위 클래스로 추상 클래스이다. 모든 바이트 기반 출력 스트림 클래스는 이 클래스를 상속받아서 만들어진다. FileOutputStream, PrintStream, BufferedOutputStream, DataOutputStream 클래스는 모두 OutputStream 클래스를 상속하고 있다.

- write(int b) 메서드

write(int b) 메서드는 매개 변수로 주어진 int 값에서 끝에 있는 1바이트만 출력 스트림으로 보낸다. 매개 변수가 int 타입이므로 4바이트 모두를 보내는 것으로 오해할 수 있다.

```
OutputStream os = new FileOutputStream("C:/test.txt");
byte[] data = "ABC".getBytes();
for(int i=0; i < data.length; i++){
    os.write(data[i]);    //"A", "B", "C"를 하나씩 출력
}
```

- write(byte[] b) 메서드

write(byte[] b)는 매개값으로 주어진 바이트 배열의 모든 바이트를 출력 스트림으로 보낸다.

```
OutputStream os = new FileOutputStream("C:/test.txt");
byte[] data = "ABC".getBytes();
os.write(data)    //"ABC" 모두 출력
```

- flush()와 close() 메서드

출력 스트림은 출력 전 버퍼에 쌓여 출력된다. flush()메서드는 있는 데이터를 모두 출력시키고 버퍼를 비우는 역할을 한다. OutputStream을 더 이상 사용하지 않을 경우에는 close() 메서드를 호출해서 OutputStream에서 사용했던 시스템 자원을 풀어준다.

```
OutputStream os = new FileOutputStream("C:/test.txt");
byte[] data = "ABC".getBytes();
os.write(data);
os.flush();
os.close();
```

18.2.3 Reader

Reader는 문자 기반 입력 스트림의 최상위 추상 클래스이다. 모든 문자 기반 입력 스트림은 이 클래스를 상속받아서 만들어진다. FileReader, BufferedReader, InputStreamReader 클래스는 모두 Reader 클래스를 상속하고 있다.

- read() 메서드

read() 메서드는 입력 스트림으로부터 한 개의 문자(2바이트)를 읽고 4바이트 int 타입으로 리턴한다. 더 이상 입력 스트림으로부터 문자를 읽을 수 없다면 read() 메서드는 -1을 리턴하는데 이것을 이용하면 읽을 수 있는 마지막 문자까지 루프를 돌며 한 문자씩 읽을 수 있다.

```
Reader reader = new FileReader("C:/test.txt");
int readData;
while((readData=reader.read()) != -1){
    char charData = (char)readData;
}
```

- read(char[] cbuf) 메서드

read(char[] cbuf) 메서드는 입력 스트림으로부터 매개값으로 주어진 문자 배열의 길이만큼 문자를 읽고 배열에 저장한다. 그리고 읽은 문자수를 리턴한다. 실제로 읽은 문자수가 배열의 길이보다 작을 경우 읽은 수만큼만 리턴한다.

```
Reader reader = new FileReader("C:/test.txt");
int readCharNo;
char[] cbuf = new char[2]
while(readCharNo = reader.read(cbuf) != -1){ ... }
```

- close() 메서드

Reader를 더 이상 사용하지 않을 경우에는 close() 메서드를 호출해서 Reader에서 사용했던 시스템 자원을 풀어준다.

```
reader.close();
```

18.2.4 Writer

Writer는 문자 기반 출력 스트림의 최상위 클래스로 추상 클래스이다. 모든 문자 기반 출력 스트림 클래스는 이 클래스를 상속받아서 만들어진다. FileWriter, BufferedWriter, PrintWriter, OutputStreamWriter 클래스는 모두 Writer 클래스를 상속하고 있다.

- write(int c) 메서드

write(int c) 메서드는 매개 변수로 주어진 int 값에서 끝에 있는 2바이트(한 개의 문자)만 출력 스트림으로 보낸다. 매개 변수가 int 타입이므로 4바이트 모두를 보내는 것으로 오해할 수 있다.

```
Writer writer = new FileWriter("C:/test.txt");
char[] data = "홍길동".toCharArray();
for(int i=0; i < data.length; i++){
    writer.write(data[i]);    //"홍", "길", "동"을 하나씩 출력
}
```


• write(char[] cbuf) 메서드

write(char[] cbuf) 메서드는 매개값으로 주어진 char[] 배열의 모든 문자를 출력 스트림으로 보낸다.

```
Writer writer = new FileWriter("C:/test.txt");
char[] data = "홍길동".toCharArray();
writer.write(data);    //"홍길동" 모두 출력
```

18.3 콘솔 입출력

콘솔(Console)은 시스템을 사용하기 위해 키보드로 입력을 받고 화면으로 출력하는 소프트웨어를 말한다. 자바는 콘솔로부터 데이터를 입력받을 때 System.in을 사용하고, 콘솔에 데이터를 출력할 때 System.out을 사용한다. 그리고 에러를 출력할 때는 System.err를 사용한다.

18.3.1 System.in 필드

자바는 프로그램이 콘솔로부터 데이터를 입력받을 수 있도록 System 클래스의 in 정적 필드를 제공하고 있다. 아래는 현금 자동 입출금기인 ATM과 비슷하게 사용자에게 메뉴를 제공하고 사용자가 어떤 번호를 입력했는지 알아내는 예제이다.

[Sample.java] 콘솔에서 입력한 번호 알아내기

```
public class Sample {
    public static void main(String[] args) throws Exception{
        System.out.println("== 메뉴 ==");
        System.out.println("1. 예금 조회");
        System.out.println("2. 예금 출금");
        System.out.println("3. 예금 입금");
        System.out.println("4. 종료 하기");
        System.out.print("메뉴를 선택하세요:");

        InputStream is = System.in;    //키보드 입력 스트림 얻기
        char inputChar = (char)is.read();    //아스키 코드를 읽고 문자로 리턴
        switch(inputChar){
            case '1':
                System.out.println("예금 조회를 선택하셨습니다."); break;
            case '2':
                System.out.println("예금 출금을 선택하셨습니다."); break;
            case '3':
                System.out.println("예금 입금을 선택하셨습니다."); break;
            case '4':
                System.out.println("종료 하기를 선택하셨습니다."); break;
        }
    }
}
```

[실행결과]

== 메뉴 ==
1. 예금 조회
2. 예금 출금
3. 예금 입금
4. 종료 하기
메뉴를 선택하세요: 2
예금 출금을 선택하셨습니다.

다음은 이름과 하고 싶은 말을 키보드로 입력받아 다시 출력하는 예제이다.

[Sample.java] 콘솔에서 입력한 한글 알아내기

```
public class Sample {
    public static void main(String[] args) throws Exception {
        InputStream is= System.in;
        byte[] datas = new byte[100];

        System.out.print("이름: ");
        int nameBytes = is.read(datas);

        //끝에 2바이트는 Enter키에 해당하는 캐리지 리턴(13)과 라인 피드(10)이므로 문자열에서 제외시킴
        //0은 시작인덱스 nameBytes-2는 읽은바이트수-2
        String name = new String(datas, 0, nameBytes-2);

        System.out.print("하고 싶은말: ");
        int commentBytes = is.read(datas);
        String comment = new String(datas, 0, commentBytes-2);

        System.out.println("입력한 이름: " + name);
        System.out.println("입력한 하고 싶은말: " + comment);
    }
}
```

[실행결과]

이름: 홍길동
하고 싶은말: 동에 번쩍 서에 번쩍 안합니다.
입력한 이름: 홍길동
입력한 하고 싶은말: 동에 번쩍 서에 번쩍 안합니다.

18.3.2 System.out 필드

콘솔에서 입력된 데이터를 System.in으로 읽었다면, 반대로 콘솔로 데이터를 출력하기 위해서는 System 클래스의 out 정적 필드를 사용한다. 아래는 write(int b) 메서드를 사용해서 연속된 숫자, 영어를 출력하고 write(byte[] b) 메서드를 사용해서 한글을 출력하는 예제이다.

[Sample.java] 연속된 숫자, 영어, 한글 출력

```
public class Sample {
    public static void main(String[] args) throws Exception {
        OutputStream os = System.out;

        for(byte b=48; b<58; b++) {    //아스키 코드 48에서 57까지의 문자를 출력
            os.write(b);
        }
        os.write(13);    //캐리지 리턴(13)을 출력하면 다음 행으로 넘어간다.

        for(byte b=97; b<123; b++) {
            os.write(b);    //아스키 코드 97에서 122까지의 문자를 출력
        }
        os.write(13);

        String hangul = "가나다라마바사아자차카타파하";
        byte[] hangulBytes = hangul.getBytes();
        os.write(hangulBytes);

        os.flush();
    }
}
```

[실행결과]

```
0123456789
abcdefghijklmnopqrstuvwxyz
가나다라마바사아자차카타파하
```

18.3.3 Console 클래스

자바 6부터는 콘솔에서 입력받은 문자열을 쉽게 읽을 수 있도록 java.io.Console 클래스를 제공하고 있다. 주의할 점은 이클립스에서 실행하면 null을 리턴하기 때문에 반드시 명령 프롬프트에서 실행해야 한다. 아래는 콘솔로부터 아이디와 패스워드를 입력받아 출력하는 예제이다.

[Sample.java] 아이디와 패스워드를 콘솔로부터 읽음

```
public class Sample {
    public static void main(String[] args) throws Exception {
        Console console = System.console();

        System.out.print("아이디: ");
        String id = console.readLine();

        System.out.print("패스워드: ");
        char[] charPass = console.readPassword();
        String strPassword = new String(charPass);    //char[] 배열을 문자열로 생성

        System.out.println("-----");
        System.out.println(id);
        System.out.println(strPassword);
    }
}
```

Sample 클래스를 실행하려면 명령 프롬프트(cmd)를 열고 소스코드가 존재하는 디렉토리/bin 으로 이동한 다음 아래와 같이 java명령을 사용한다.

```
C:/>cd c:/data/java/17장/ex11/bin
C:/data/java/17장/ex11/bin>java ex11.Sample
```

아이디를 키보드로 입력할 때에는 입력 문자가 에코(echo) 출력이 되지만, 패스워드는 입력 문자가 에코 출력이 되지 않기 때문에 보안상 유리하다.



18.3.4 Scanner 클래스

Console 클래스는 콘솔로부터 문자열을 읽을 수 있지만 기본 타입(정수, 실수) 값을 바로 읽을 수는 없다. java.io 패키지의 클래스는 아니지만 java.util 패키지의 Scanner 클래스를 이용하면 콘솔로부터 기본 타입의 값을 바로 읽을 수 있다.

[Sample.java] 문자열, 정수, 실수를 직접 읽는 예제

```
public class Sample {
    public static void main(String[] args) throws Exception {
        Scanner scanner = new Scanner(System.in);

        System.out.print("문자열 입력> ");
        String inputString = scanner.nextLine();
        System.out.println(inputString);
        System.out.println();

        System.out.print("정수 입력> ");
        int inputInt = scanner.nextInt();
        System.out.println(inputInt);
        System.out.println();

        System.out.print("실수 입력> ");
        double inputDouble = scanner.nextDouble();
        System.out.println(inputDouble);
        System.out.println();
    }
}
```

[실행결과]

문자열 입력> 이제 봄이군요. 따뜻합니다.
이제 봄이군요. 따뜻합니다.

```
정수 입력> 100
100
```

실수 입력> 3.14159
3.14159

18.4 파일 입출력

18.4.1 File 클래스

IO 패키지(java.io)에서 제공하는 File 클래스는 파일 크기, 파일 속성, 파일 이름 등의 정보를 얻어내는 기능과 파일 생성 및 삭제 기능을 제공하고 있다. 그리고 디렉터리를 생성하고 디렉터리에 존재하는 파일 리스트를 얻어내는 기능도 있다. 그러나 파일의 데이터를 읽고 쓰는 기능은 지원하지 않는다. 파일의 입출력은 스트림을 사용해야 한다. 아래는 C:/temp 디렉터리에 DIR 디렉터리와 file1.txt, file2.txt, file3.txt 파일을 생성하고 temp 디렉터리에 있는 파일 목록을 출력하는 예제이다.

[Sample.java] File 클래스를 이용한 파일 및 디렉토리 정보 출력

```

public class Sample {
    public static void main(String[] args) throws Exception{
        File dir = new File("C:/temp/DIR");
        File file1 = new File("C:/temp/file1.txt");
        File file2 = new File("C:/temp/file2.txt");

        //파일 경로를 URI 객체로 생성해서 매개값으로 제공해도 됨
        File file3 = new File(new URI("file:///C:/temp/file3.txt"));

        if(dir.exists() == false) { dir.mkdirs(); }
        if(file1.exists() == false) { file1.createNewFile(); }
        if(file2.exists() == false) { file2.createNewFile(); } //파일 1
        if(file3.exists() == false) { file3.createNewFile(); } //파일 2

        File temp = new File("C:/temp");
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        File[] contents = temp.listFiles();

        System.out.println("날짜\t\t\t형태\t\t크기\t이름");
        System.out.println("-----");

        for(File file:contents) {
            System.out.print(sdf.format(new Date(file.lastModified())));
            if(file.isDirectory() ) {
                System.out.print("\t<DIR>\t\t" + file.getName());
            }else {
                System.out.print("\t\t" + file.length() + "\t" + file.getName());
            }
            System.out.println();
        }
    }
}

```

[실행결과]

날짜	형태	크기	이름
2020-03-01 오후 18:30	<DIR>		Dir
2020-03-01 오후 18:30		0	file1.txt
2020-03-01 오후 18:30		0	file2.txt
2020-03-01 오후 18:30		0	file3.txt

18.4.2 FileInputStream

FileInputStream 클래스는 파일로부터 바이트 단위로 읽어 들일 때 사용하는 바이트 기반 입력 스트림이다. 바이트 단위로 읽기 때문에 그림, 오디오, 비디오, 텍스트 파일 등 모든 종류의 파일을 읽을 수 있다. 아래는 file1.txt 소스 파일을 읽고 콘솔에 보여주는 예제이다.

[Sample.java] 텍스트 파일을 읽고 출력

```
public class Sample {
    public static void main(String[] args){
        try {
            FileInputStream fis = new FileInputStream("C:/temp/file1.txt");

            int data;
            while((data = fis.read()) != -1) {
                System.out.write(data);
            }
            fis.close();
        }catch(Exception e) {
            System.out.println(e.toString());
        }
    }
}
```

[실행결과]

죽는 날까지 하늘을 우러러
한점 부끄럼이 없기를
앞새에 이는 바람에도
나는 괴로워 했다.

18.4.3 FileOutputStream

FileOutputStream은 바이트 단위로 데이터를 파일에 저장할 때 사용하는 바이트 기반 출력스트림이다. 바이트 단위로 저장하기 때문에 그림, 오디오, 텍스트 등 모든 종류의 데이터를 파일로 저장할 수 있다. 아래는 원본 파일을 타겟 파일로 복사하는 예제이다. 복사 프로그램의 원리는 원본 파일에서 읽은 바이트를 바로 타겟 파일로 저장하는 것이기 때문에 FileInputStream에서 읽은 바이트를 바로 FileOutputStream으로 저장하면 된다.

[Sample.java] 파일 복사

```
public class Sample {
    public static void main(String[] args) throws Exception{
        String originalFileName = "C:/temp/phone.png";
        String targetFileName = "C:/temp/phone1.png";

        FileInputStream fis = new FileInputStream(originalFileName);
        FileOutputStream fos = new FileOutputStream(targetFileName);

        int data;
        while((data = fis.read()) != -1) {
            fos.write(data);
        }
        fos.flush();
        fos.close();
        fis.close();
        System.out.println("복사가 잘 되었습니다.");
    }
}
```

18.4.4 FileReader

FileReader 클래스는 텍스트 파일을 프로그램으로 읽어 들일 때 사용하는 문자 기반 스트림이다. 문자 단위로 읽기 때문에 텍스트가 아닌 그림, 오디오, 비디오 등의 파일은 읽을 수 없다. 다음은 file1.txt 텍스트 파일을 읽고 콘솔에 출력하는 예제이다.

[Sample.java] 텍스트 파일 읽기

```
public class Sample {
    public static void main(String[] args) throws Exception{
        FileReader fr = new FileReader("C:/temp/file1.txt");

        int readCharNo;
        char[] cbuf = new char[100];
        while((readCharNo = fr.read(cbuf)) != -1) {
            String data = new String(cbuf);
            System.out.print(data);
        }
        fr.close();
    }
}
```

18.4.5 FileWriter

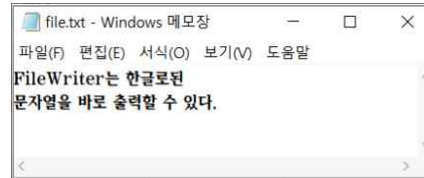
FileWriter는 텍스트 데이터를 파일에 저장할 때 사용하는 문자 기반 스트림이다. 문자 단위로 저장하기 때문에 텍스트가 아닌 그림, 오디오, 비디오 등의 데이터를 파일로 저장할 수 없다.

[Sample.java] 문자열을 파일에 저장

```
public class Sample {
    public static void main(String[] args) throws Exception{
        File file = new File("C:/temp/file.txt");

        FileWriter fw = new FileWriter(file, true);
        fw.write("FileWriter는 한글로된 " + "\r\n");
        fw.write("문자열을 바로 출력할 수 있다." + "\r\n");

        fw.flush();
        fw.close();
        System.out.println("파일에 저장되었습니다.");
    }
}
```



18.5 보조 스트림

보조 스트림이란 다른 스트림과 연결되어 여러 가지 편리한 기능을 제공해주는 스트림을 말한다. 보조 스트림은 문자 변환, 입출력 성능향상, 기본 데이터 타입 입출력, 객체 입출력 등의 기능을 제공한다.

18.5.1 문자 변환 보조 스트림

소스 스트림이 바이트 기반 스트림(InputStream, OutputStream, FileInputStream, FileOutputStream)이면서 입출력 데이터가 문자라면 Reader와 Writer로 변환해서 사용하는 것을 고려해야 한다.

• InputStreamReader

InputStreamReader는 바이트 입력 스트림에 연결되어 문자 입력 스트림인 Reader로 변환시키는 보조 스트림이다. 아래는 콘솔에서 입력한 한글을 Reader를 이용해서 읽고, 다시 콘솔로 출력하는 예제이다.

[Sample.java] 콘솔에서 한글 입력받기

```
public class Sample {
    public static void main(String[] args) throws Exception{
        InputStream is = System.in;
        Reader reader = new InputStreamReader(is);

        int readCharNo;
        char[] cbuf = new char[100];
        while((readCharNo=reader.read(cbuf)) != -1) {
            String data = new String(cbuf);
            System.out.println(data);
        }

        reader.close();
    }
}
```

[실행결과]

바이트 입력스트림을 문자입력스트림으로 변환
바이트 입력스트림을 문자입력스트림으로 변환

• OutputStreamWriter

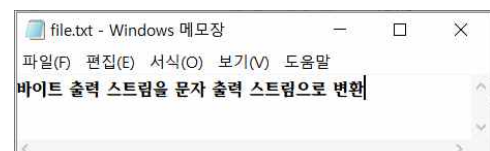
OutputStreamWriter는 바이트 출력 스트림에 연결되어 문자 출력 스트림인 Writer로 변환시키는 보조 스트림이다. 아래는 FileOutputStream을 Writer로 변환해서 문자열을 파일에 저장한다.

[Sample.java] 파일로 출력하기

```
public class Sample {
    public static void main(String[] args) throws Exception{
        FileOutputStream fos = new FileOutputStream("C:/temp/file.txt");
        Writer writer = new OutputStreamWriter(fos);

        String data = "바이트 출력 스트림을 문자 출력 스트림으로 변환";
        writer.write(data);

        writer.flush();
        writer.close();
        System.out.println("파일 저장이 끝났습니다.");
    }
}
```



18.5.2 성능 향상 보조 스트림

프로그램 실행 성능은 CPU와 메모리가 아무리 뛰어나도 하드 디스크의 입출력이 늦어지면 하드 디스크의 처리 속도에 맞춰진다. 해결책은 프로그램이 입출력 소스와 직접 작업하지 않고 중간에 메모리 버퍼와 작업함으로써 실행 성능을 향상시킬 수 있다. 보조 스트림 중에서는 메모리 버퍼를 제공하여 프로그램의 실행 성능을 향상시키는 것들이 있다. 바이트 기반 스트림에는 `BufferedInputStream`, `BufferedOutputStream`이 있고, 문자 기반 스트림에는 `BufferedReader`, `BufferedWriter`가 있다.

• `BufferedInputStream`과 `BufferedReader`

`BufferedInputStream`은 바이트 입력 스트림에 연결되어 버퍼를 제공해주는 보조 스트림이고, `BufferedReader`는 문자 입력 스트림에 연결되어 버퍼를 제공해주는 보조 스트림이다. 프로그램은 외부의 소스로부터 직접 읽는 대신 버퍼로부터 읽음으로써 읽기 성능이 향상된다. 아래 코드는 Enter키를 입력하기 전까지 콘솔에서 입력한 모든 문자열을 한꺼번에 얻는다.

[Sample.java] 콘솔로부터 라인 단위로 읽기

```
public class Sample {
    public static void main(String[] args) throws Exception{
        InputStream is = System.in;
        Reader reader = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(reader);
        System.out.print("입력: ");
        String lineString = br.readLine();
        System.out.print("출력: " + lineString);
    }
}
```

[실행결과]

입력: 한줄 전체를 읽습니다.
출력: 한줄 전체를 읽습니다.

• `BufferedOutputStream`과 `BufferedWriter`

`BufferedOutputStream`은 바이트 출력 스트림에 연결되어 버퍼를 제공해주는 보조 스트림이고, `BufferedWriter`는 문자 출력 스트림에 연결되어 버퍼를 제공해주는 보조 스트림이다. 프로그램 입장에서 보면 직접 데이터를 보내는 것이 아니라, 메모리 버퍼로 데이터를 고속 전송하기 때문에 실행 성능이 향상되는 효과를 얻게 된다.

[Sample.java] 버퍼를 사용했을 때의 성능 테스트

```
public class Sample {
    public static void main(String[] args) throws Exception{
        FileInputStream fis = null;
        FileOutputStream fos = null;
        BufferedInputStream bis = null;
        BufferedOutputStream bos = null;

        int data = -1;
        long start = 0;
        long end = 0;

        fis = new FileInputStream("C:/Temp/phone.png");
        bis = new BufferedInputStream(fis);
        fos = new FileOutputStream("C:/Temp/phone1.png");
        start = System.currentTimeMillis();
        while((data = bis.read()) != -1) {
            fos.write(data);
        }
        fos.flush();
        end = System.currentTimeMillis();
        fos.close();fis.close();fis.close();
        System.out.println("버퍼를 사용하지 않았을 때: " + (end-start) + "ms");

        fis = new FileInputStream("C:/Temp/phone.png");
        bis = new BufferedInputStream(fis);
        fos = new FileOutputStream("C:/Temp/phone1.png");
        bos = new BufferedOutputStream(fos); //BufferedOutputStream 사용
        start = System.currentTimeMillis();
        while((data = bis.read()) != -1) {
            bos.write(data);
        }
        fos.flush();
        end = System.currentTimeMillis();
        fos.close();fis.close();fis.close();
        System.out.println("버퍼를 사용했을 때: " + (end-start) + "ms");
    }
}
```

[실행결과]

버퍼를 사용하지 않았을 때: 4453ms
버퍼를 사용했을 때: 18ms

18.5.3 객체 입출력 보조 스트림

자바는 메모리에 생성된 객체를 파일 또는 네트워크로 출력할 수가 있다. 객체는 문자가 아니기 때문에 바이트 기반 스트림으로 출력해야 한다. 객체를 출력하기 위해서는 객체의 데이터(필드값)를 일렬로 늘어선 연속적인 바이트로 변경해야 하는데, 이것을 객체 직렬화(serialization)라고 한다. 반대로 파일에 저장되어 있거나 네트워크에서 전송된 객체를 읽을 수도 있는데, 입력 스트림으로부터 읽어 들인 연속적인 바이트를 객체로 복원하는 것을 역직렬화(deserialization)라고 한다.

- **ObjectInputStream, ObjectOutputStream**

자바는 객체를 입출력할 수 있는 두 개의 보조 스트림인 **ObjectInputStream**과 **ObjectOutputStream**을 제공한다. **ObjectOutputStream**은 바이트 출력 스트림과 연결되어 객체를 직렬화하는 역할을 하고, **ObjectInputStream**은 바이트 입력 스트림과 연결되어 객체로 역직렬화하는 역할을 한다.

ObjectInputStream과 **ObjectOutputStream**은 다른 보조 스트림과 마찬가지로 연결할 바이트 입출력 스트림을 생성자의 매개값으로 받는다.

```
ObjectInputStream ois = new ObjectInputStream(바이트입력스트림)
ObjectOutputStream oos= new ObjectOutputStream(바이트출력스트림)
```

ObjectOutputStream으로 객체를 직렬화하기 위해서는 **writeObject()** 메서드를 사용한다.

```
oos.writeObject(객체);
```

반대로 **ObjectInputStream**의 **readObject()** 메서드는 입력 스트림에서 읽은 바이트를 역직렬화해서 객체로 생성한다. **readObject()** 메서드의 리턴 타입은 **Object** 타입이기 때문에 객체 원래의 타입으로 변환해야 한다.

```
객체타입 변수 = (객체타입)ois.readObject();
```

18.5.4 파일 입력, 목록출력, 삭제 예제 프로그램

[Sample.java]

```
import java.util.*;

public class Sample {
    public static void main(String[] args) throws Exception{
        Scanner scanner=new Scanner(System.in);
        boolean run=true;
        int menu;
        StudentDAO dao=new StudentDAO();

        while(run){
            System.out.println("-----");
            System.out.println("1.학생등록|2.학생목록|3.학생수정|4.학생삭제|5.종료");
            System.out.println("-----");
            System.out.print("선택>");

            menu=scanner.nextInt();scanner.nextLine();
            switch(menu){
                case 1:
                    dao.insert();
                    break;
                case 2:
                    dao.list();
                    break;
                case 3:
                    case 4:
                    dao.delete();
                    break;
                case 5:
                    default:
                    System.out.println("1~5번 메뉴를 선택하세요!");
            }
            scanner.close();
        }
    }
}
```

[실행결과]

1. 학생등록 | 2. 학생목록 | 3. 학생수정 | 4. 학생삭제 | 5. 종료

선택>1
학생번호>20
학생이름>park kim
학생주소>seoul
학생등록완료!

1. 학생등록 | 2. 학생목록 | 3. 학생수정 | 4. 학생삭제 | 5. 종료

선택>2
40 kim me sook incho
10 cho hyang duk busan
20 park kim seoul

1. 학생등록 | 2. 학생목록 | 3. 학생수정 | 4. 학생삭제 | 5. 종료

선택>4
학생번호>10

1. 학생등록 | 2. 학생목록 | 3. 학생수정 | 4. 학생삭제 | 5. 종료

선택>2
40 kim me sook incho
20 park kim seoul

```

import java.util.*;
import java.io.*;

public class StudentDAO{
    Scanner scanner=new Scanner(System.in);
    File file = new File("c:/temp/students.txt");

    public void insert(){
        try{
            FileWriter writer = null;
            writer = new FileWriter(file, true); //파일의 내용에 이어서 쓰려면 true를, 기존 내용을 없애고 새로 쓰려면 false를 지정한다.
            System.out.print("학생번호>");String no=scanner.nextLine();
            System.out.print("학생이름>");String name=scanner.nextLine();
            System.out.print("학생주소>");String address=scanner.nextLine();
            writer.write(no + "," + name + "," + address + "\n");
            writer.flush();
            writer.close();
            System.out.println("학생등록완료!");
        }catch(Exception e){
            System.out.println("에러:" + e.toString());
        }
    }

    public void list(){
        try{
            BufferedReader reader=new BufferedReader(new FileReader(file));
            String line=null;
            StringTokenizer st=null;

            while((line=reader.readLine()) != null){
                st = new StringTokenizer(line, ",");
                String no=st.nextToken();
                String name=st.nextToken();
                String address=st.nextToken();
                System.out.println(no + "\t" + name + "\t" + address);
            }
            reader.close();
        }catch(Exception e){
            System.out.println("에러:" + e.toString());
        }
    }

    public void delete(){
        try{
            System.out.print("학생번호>");String selNo=scanner.nextLine();
            BufferedReader reader=new BufferedReader(new FileReader(file));
            String line=null;
            StringTokenizer st=null;
            String strFile="";

            while((line=reader.readLine()) != null){
                st = new StringTokenizer(line, ",");
                String no=st.nextToken();
                String name=st.nextToken();
                String address=st.nextToken();
                if(!selNo.equals(no)) {
                    strFile=strFile + no + "," + name + "," + address + "\n";
                }
            }
            reader.close();

            FileWriter writer = new FileWriter(file, false);
            writer.write(strFile);
            writer.flush();
            writer.close();
        }catch(Exception e){
            System.out.println("에러:" + e.toString());
        }
    }
}

```


18.5.5 Database 접속을 위한 JDBC

JDBC란 자바에서 제공해 주는 DB관련 처리를 하는 데 필요한 API들이다. 애플리케이션에서 DB에 연동, SQL 문장 전송 등의 작업을 할 수 있게 제공되는 여러 가지의 인터페이스들이 JDBC에 존재한다. JDBC는 자바에서 가장 성공적인 API중에 하나이다.

- system 비밀번호 변경 시

1.Oracle Database - Run SQL Command Line 실행

2.sql>connect

3.sql>Enter user-name:sys as sysdba

4.sql>Enter password:그냥 엔터키를 누른다.

5.sql>alter user system identified by 1234;

- 자바에서 jdbc 사용 시

1)프로젝트 디렉터리에서 마우스 우측버튼을 클릭하고 [Properties]메뉴를 클릭

2)[java Build Path]메뉴-[Libraries]탭-[Add External JARs]버튼을 클릭

3)오라클 [jdbc]-[lib]경로에서 ojdbc6.jar 파일을 선택한다.

[JDBCUtil.java]

```
public class JDBCUtil {
    public static Connection getConnection(){
        String driver = "oracle.jdbc.driver.OracleDriver";
        String url = "jdbc:oracle:thin:@localhost:1521:xe";
        String user = "system";
        String password = "1234";
        Connection con = null;
        try{
            Class.forName(driver);
            con=DriverManager.getConnection(url, user, password);
            System.out.println("접속성공");
        }catch(Exception e){
            System.out.println("예러:" + e.toString());
        }
        return con;
    }
}
```

[Sample.java]

```
public class Sample {
    public static void main(String[] args) throws Exception{
        Connection con=JDBCUtil.getConnection();
        String sql="select * from tbl_user";
        PreparedStatement ps=con.prepareStatement(sql);
        ResultSet rs=ps.executeQuery();
        while(rs.next()){
            System.out.print(rs.getString("no") + "\t");
            System.out.print(rs.getString("name") + "\t");
            System.out.print(rs.getString("address") + "\t");
            System.out.println("");
        }
    }
}
```

18.6 네트워크 기초

네트워크(network)는 여러 대의 컴퓨터를 통신 회선으로 연결한 것을 말한다. 만약 집에 방마다 컴퓨터가 있고, 이 컴퓨터들을 유선, 무선 등의 통신 회선으로 연결했다면 홈네트워크가 형성된 것이다. 지역 네트워크는 회사, 건물, 특정 영역에 존재하는 컴퓨터를 통신 회선으로 연결한 것을 말하고, 인터넷은 지역 네트워크를 통신 회선으로 연결한 것을 말한다.

18.6.1 서버와 클라이언트

컴퓨터가 인터넷에 연결되어 있다면 실제로 데이터를 주고받는 행위는 프로그램들이 한다. 서비스를 제공하는 프로그램을 일반적으로 서버(server)라고 부르고, 서비스를 받는 프로그램을 클라이언트(client)라고 부른다. 클라이언트는 서비스를 받기 위해 연결을 요청하고, 서버는 연결을 수락하여 서비스를 제공해준다. 클라이언트/서버 모델은 한 개의 서버와 다수의 클라이언트로 구성되는 것이 보통이나 두 개의 프로그램이 서버인 동시에 클라이언트 역할을 하는 P2P(peer to peer) 모델도 있다. P2P 모델에서는 먼저 접속을 시도한 컴퓨터가 클라이언트가 된다.

18.6.2 IP 주소와 포트(Port)

컴퓨터에도 고유한 주소가 있다. 이것이 바로 IP(Internet Protocol) 주소이다. IP 주소는 네트워크 어댑터(랜카드)마다 할당되는데, 한 개의 컴퓨터에 두 개의 네트워크 어댑터가 장착되어 있다면 두 개의 IP 주소를 할당할 수 있다. 네트워크 어댑터에 어떤 IP 주소가 부여되어 있는지 확인하려면 명령 프롬프트(cmd.exe)에서 다음과 같이 실행하면 된다.

```
C:\>ipconfig /all
```

IP주소는 xxx.xxx.xxx.xxx와 같은 형식으로 표현한다. 여기서 xxx는 부호 없는 0~255 사이의 정수이다. 연결할 상대방 컴퓨터의 IP 주소를 모른다면 프로그램들은 통신을 할 수 없다. 우리가 전화번호를 모르면 114로 문의하듯이, 프로그램은 운(Domain Name System)을 이용해서 연결할 컴퓨터의 IP 주소를 찾는다. 아래와 같이 DNS에 도메인 이름으로 IP를 등록해 놓는다.

[DNS]	도메인 이름(www.naver.com)	:	등록된 IP주소(222.122.195.5)
-------	-----------------------	---	-------------------------

한 대의 컴퓨터에는 다양한 서버 프로그램들(웹(Web) 서버, 데이터베이스 관리 시스템(DBMS), FTP 서버)이 하나의 IP를 갖는 컴퓨터에서 동시에 실행될 수 있다. IP는 컴퓨터의 네트워크 어댑터까지만 갈 수 있는 정보이기 때문에 컴퓨터 내에서 실행하는 서버를 선택하기 위해서는 추가적인 정보가 필요하다. 이 정보가 포트(port) 번호이다. 서버는 시작할 때 고정적인 포트 번호를 가지고 실행하는데, 이것을 포트 바인딩(binding)이라고 한다. 클라이언트도 서버에서 보낸 정보를 받기 위해 포트 번호가 필요한데, 서버와 같이 고정적인 포트 번호가 아니라 운영체제가 자동으로 부여하는 동적 포트 번호를 사용한다. 이 동적 포트 번호는 클라이언트가 서버로 연결 요청을 할 때 전송되어 서버가 클라이언트로 데이터를 보낼 때 사용한다.

18.6.3 InetAddress로 IP 주소 얻기

자바는 IP 주소를 java.net.InetAddress 객체로 표현한다. InetAddress는 로컬 컴퓨터의 IP 주소뿐만 아니라 도메인 이름을 DNS에서 검색한 후 IP 주소를 가져오는 기능을 제공한다. 아래 예제는 로컬 컴퓨터의 IP와 네이버(www.naver.com)의 IP 정보를 출력한다.

[Sample.java] IP 주소 얻기

```
public class Sample {
    public static void main(String[] args) {
        try {
            InetAddress local = InetAddress.getLocalHost();
            System.out.println("내컴퓨터 IP주소: " + local.getHostAddress());

            InetAddress[] iaArr = InetAddress.getAllByName("www.naver.com");
            for(InetAddress remote:iaArr) {
                System.out.println("www.naver.com IP주소: " + remote.getHostAddress());
            }
        } catch (Exception e) {
            System.out.println(e.toString());
        }
    }
}
```

[실행결과]

```
내컴퓨터 IP주소: 192.168.1.119
www.naver.com IP주소: 125.209.222.142
www.naver.com IP주소: 125.209.222.141
```

18.7 TCP 네트워크

TCP(Transmission Control Protocol)는 연결 지향적 프로토콜이다. 연결 지향 프로토콜이란 클라이언트와 서버가 연결된 상태에서 데이터를 주고 받는 프로토콜을 말한다. 클라이언트가 연결 요청을 하고, 서버가 연결을 수락하면 통신 선로가 고정되고, 모든 데이터는 고정된 통신 선로를 통해서 순차적으로 전달된다. TCP는 안정적으로 데이터를 정확히 전달하는 장점이 있지만, 데이터를 보내기 전에 연결이 형성되어야하므로 시간이 많이 걸린다.

18.7.1 ServerSocket과 Socket의 용도

TCP는 클라이언트가 연결 요청을 해오면 연결을 수락하고 연결된 클라이언트와 통신하는 두 가지 역할이 있다. 클라이언트의 연결 요청을 기다리면서 연결 수락을 담당하는 것이 java.net.ServerSocket 클래스이고, 연결된 클라이언트와 통신을 담당하는 것이 java.net.Socket 클래스이다.

서버는 클라이언트가 접속할 포트를 가지고 있어야 하는데, 이 포트를 바인딩(binding) 포트라고 한다. 서버는 고정된 포트 번호에 바인딩해서 실행하므로 ServerSocket을 생성할 때 포트 번호 하나를 지정해야 한다.

18.7.2 ServerSocket 생성과 연결 수락

서버를 개발하려면 우선 ServerSocket 객체를 얻어야 한다. ServerSocket을 얻는 가장 간단한 방법은 생성자에 바인딩 포트를 대입하고 객체를 생성하는 것이다. 아래는 5001번 포트에 바인딩하는 ServerSocket을 생성한다.

```
ServerSocket serverSocket = new ServerSocket();
serverSocket.bind(new InetSocketAddress("localhost", 5001));
```

포트 바인딩이 끝났다면 ServerSocket은 클라이언트 연결 수락을 위해 accept() 메서드를 실행해야 한다. 아래 예제는 반복적으로 accept() 메서드를 호출해서 다중 클라이언트 연결을 수락하는 가장 기본적인 코드를 보여준다.

[ServerSample.java]

```
public class ServerSample {
    public static void main(String[] args) {
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket();
            serverSocket.bind(new InetSocketAddress("localhost", 5001));

            while(true) {
                System.out.println("[연결 기다림]");
                Socket socket = serverSocket.accept();
                InetSocketAddress isa = (InetSocketAddress)socket.getRemoteSocketAddress();
                System.out.println("[연결 수락함]" + isa.getHostName());
            }
        } catch (Exception e) {
            System.out.println(e.toString());
        }

        if(!serverSocket.isClosed()) {
            try {
                serverSocket.close();
                System.out.println("[ServerSocket 닫기]");
            } catch (Exception e) {}
        }
    }
}
```

18.7.3 Socket 생성과 연결 요청

클라이언트가 서버에 연결 요청을 하려면 java.net.Socket을 이용해야 한다. Socket 객체를 생성함과 동시에 연결 요청을 하려면 생성자의 매개값으로 서버의 IP 주소와 바인딩 포트 번호를 제공하면 된다. 아래 예제는 localhost 5001 포트로 연결을 요청하는 코드이다. connect() 메서드가 정상적으로 리턴하면 연결이 성공한 것이다.

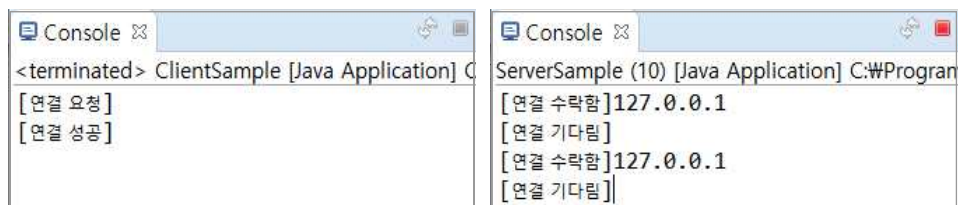
[ClientSample.java] 연결 요청

```
public class ClientSample {
    public static void main(String[] args) {
        Socket socket = null;

        try {
            socket = new Socket();
            System.out.println("[연결 요청]");
            socket.connect(new InetSocketAddress("localhost", 5050));
            System.out.println("[연결 성공]");
        } catch (Exception e) {
            System.out.println(e.toString());
        }

        if(!socket.isClosed()) {
            try {
                socket.close();
            } catch (Exception e) {}
        }
    }
}
```

아래는 이전 예제에서 작성한 ServerSample과 ClientSample을 실행한 결과이다. 먼저 ServerSample부터 실행하고 ClientSample을 실행한다.



18.7.4 Socket 데이터 통신

클라이언트가 연결 요청(connect())하고 서버가 연결 수락(accept())했다면, 양쪽의 Socket 객체로부터 각각 입력 스트림(InputStream)과 출력 스트림(OutputStream)을 얻을 수 있다. 아래 예제는 ①연결 성공 후, 클라이언트 먼저 "Hello Server"를 서버로 보낸다. ②서버가 이 데이터를 받고 ③ "Hello Client"를 클라이언트로 보내면 ④클라이언트가 이 데이터를 받는다.

[ClientSample.java] 데이터 보내고 받기

```
public class ClientSample {
    public static void main(String[] args) {
        Socket socket = null;
        try {
            socket = new Socket();
            System.out.println("[연결 요청]");
            socket.connect(new InetSocketAddress("localhost", 5001));
            System.out.println("[연결 성공]");

            byte[] bytes = null;
            String message = null;

            OutputStream os = socket.getOutputStream();           ①
            message = "Hello Server";
            bytes = message.getBytes("UTF-8");
            os.write(bytes);
            os.flush();
            System.out.println("[데이터 보내기 성공]");

            InputStream is = socket.getInputStream();               ④
            bytes = new byte[100];
            int readByteCount = is.read(bytes);
            message = new String(bytes, 0, readByteCount, "UTF-8");
            System.out.println("[데이터 받기 성공]: " + message);

            os.close(); is.close();
        } catch (Exception e) {System.out.println(e.toString());}
        if(!socket.isClosed()) {
            try {socket.close(); } catch (Exception e) {}
        }
    }
}
```

[ServerSample.java] 데이터 받고 보내기

```
public class ServerSample {
    public static void main(String[] args) {
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket();
            serverSocket.bind(new InetSocketAddress("localhost", 5052));
            while(true) {
                System.out.println("[연결 기다림]");
                Socket socket = serverSocket.accept();
                InetSocketAddress isa = (InetSocketAddress)socket.getRemoteSocketAddress();
                System.out.println("[연결 수락함] " + isa.getHostName());

                byte[] bytes = null;
                String message = null;

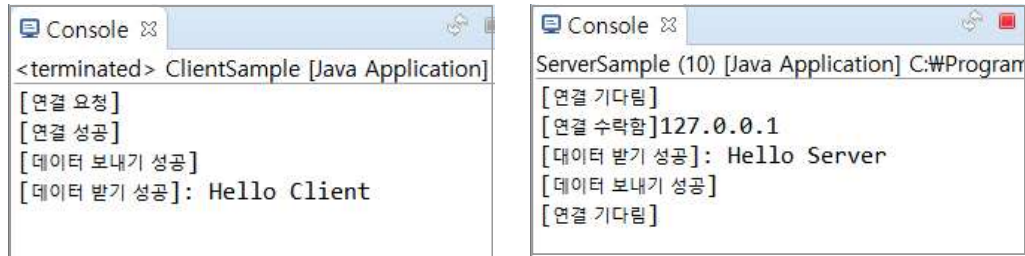
                InputStream is = socket.getInputStream();           ②
                bytes = new byte[100];
                int readByteCount = is.read(bytes);
                message = new String(bytes, 0, readByteCount, "UTF-8");
                System.out.println("[데이터 받기 성공]: " + message);

                OutputStream os = socket.getOutputStream();       ④
                message = "Hello Client";
                bytes = message.getBytes("UTF-8");
                os.write(bytes);
                os.flush();
                System.out.println("[데이터 보내기 성공]");

                is.close(); os.close(); socket.close();
            }
        } catch (Exception e) {System.out.println(e.toString());}

        if(!serverSocket.isClosed()) {
            try { serverSocket.close(); System.out.println("[ServerSocket 닫기]"); } catch (Exception e) {}
        }
    }
}
```

아래는 ServerSample과 ClientSample을 실행한 결과이다. 먼저 ServerSample부터 실행하고 ClientSample을 실행한다.



18.7.5 스레드 병렬 처리

연결 수락을 위해 ServerSocket의 accept()를 실행하거나, 서버 연결 요청을 위해 Socket 생성자는 Connect를 실행할 경우에는 해당 작업이 완료 되기 전까지 블로킹(blocking)된다. 데이터 통신을 할 때에도 마찬가지로 InputStream의 read() 메서드는 상대방이 데이터를 보내기 전까지 블로킹 이 되고 OutputStream의 write() 메서드는 데이터를 완전하게 보내기 전까지 블로킹된다. 결론적으로 ServerSocket과 Socket은 동기 방식이다.

만약 서버를 실행시키는 main 스레드가 직접 입출력 작업을 담당하게 되면 입출력이 완료될 때까지 다른 작업을 할 수 없는 상태가 된다. 서버 애플리케이션은 지속적으로 클라이언트의 연결 수락 기능을 수행해야 하는데, 입출력에서 블로킹되면 이 작업을 할 수 없게 된다. 또한 클라이언트와 입출력하는 동안에는 클라이언트와 입출력을 할 수 없게 된다. 그렇기 때문에 accept(), connect(), read(), write()는 별도의 작업 스레드를 생성해서 병렬적으로 처리하는 것이 좋다.

스레드로 병렬 처리를 할 경우, 수천 개의 클라이언트가 동시에 연결되면 서버에서 수천 개의 스레드가 생성되기 때문에 서버 성능이 급격히 저하되고, 다운 되는 현상이 발생할 수 있다. 클라이언트의 폭증으로 인해 서버의 과도한 스레드 생성을 방지하려면 스레드풀을 사용하는 것이 바람직하다.

18.8 UDP 네트워킹

UDP(User Datagram Protocol)는 비연결 지향적 프로토콜이다. 비연결 지향적이란 데이터를 주고받을 때 연결 절차를 거치지 않고, 발신자가 일방적으로 데이터를 발신하는 방식이다. 연결 과정이 없기 때문에 TCP 보다는 빠른 전송을 할 수 있지만 데이터 전달의 신뢰성은 떨어진다.

UDP는 편지에 비유할 수 있다. 발신자는 봉투(패킷)에 수신자의 주소(원격지 IP와 포트)와 발신자의 주소(로컬 IP와 포트)를 쓴다. 그리고 봉투 안에 편지(전송할 데이터)를 넣고 편지를 보낸다. 발신자는 수신자가 편지를 받았는지, 못 받았는지를 알지 못한다. 게다가 최근에 보낸 편지가 예전에 보낸 편지보다 더 빨리 도착할 수도 있고, 전혀 도착하지 않을 수도 있다.

일반적으로 데이터 전달의 신뢰성보다는 속도가 중요한 프로그램에서는 UDP를 사용하고, 데이터 전달의 신뢰성이 중요한 프로그램에서는 TCP를 사용한다. 자바에서는 UDP 프로그램을 위해 java.net.DatagramSocket과 java.net.DatagramPacket 클래스를 제공하고 있다. DatagramSocket은 발신점과 수신점에 해당하는 클래스이고, DatagramPacket은 주고받는 패킷 클래스이다.

18.8.1 발신자 구현

아래는 발신자 프로그램의 전체 코드이다. for문을 두 번 반복하는데 각각 "메시지1", "메시지2" 문자열을 전송하도록 했다.

[SendSample.java] 발신자

```
public class SendSample {
    public static void main(String[] args) throws Exception {
        DatagramSocket datagramSocket = new DatagramSocket();

        System.out.println("[발신시작]");

        for(int i=1; i<3; i++) {
            String data = "메시지" + i;
            byte[] byteArr = data.getBytes("UTF-8");

            DatagramPacket packet = new DatagramPacket(byteArr, byteArr.length, new InetSocketAddress("localhost", 5001));

            datagramSocket.send(packet);
            System.out.println("[보낸 바이트 수]: " + byteArr.length + " bytes");
        }

        System.out.println("[발신 종료]");
        datagramSocket.close();
    }
}
```

18.8.2 수신자 구현

아래는 수신자 프로그램의 전체 코드이다. 실행 후 10초가 지나면 수신자를 종료하도록 했다.

[ReceiveSample.java] 수신자

```
public class ReceiveSample {
    public static void main(String[] args) throws Exception {
        DatagramSocket datagramSocket = new DatagramSocket(5001);

        Thread thread = new Thread() {
            @Override
            public void run() {
                System.out.println("[수신시작]");
                try {
                    while(true) {
                        DatagramPacket packet = new DatagramPacket(new byte[100], 100);
                        datagramSocket.receive(packet);

                        String data = new String(packet.getData(), 0, packet.getLength(), "UTF-8");
                        System.out.println("[받은 내용: " + packet.getSocketAddress() + "]" + data);
                    }
                } catch (Exception e) {
                    System.out.println("[수신 종료]");
                }
            }
        };
        thread.start();

        Thread.sleep(10000);
        datagramSocket.close();
    }
}
```

아래는 수신자와 발신자를 실행한 모습을 보여준다. 실행 순서는 상관없지만, 수신자를 먼저 실행하고 발신자를 실행해야만 발신자가 보낸 데이터를 수신자가 모두 받을 수 있다. 발신자를 먼저 실행하면 수신자가 실행하기 전에 보낸 데이터는 받을 수 없다.