객체지향프로그래밍(최지웅)

Junhyeok Lee (wnsx0000@gmail.com)

August 18, 2024

목차

Part I

Java 기초

객체지향프로그래밍을 Java로 학습함.

1. 서론

1.1. 개발환경 구성

1.1.1. Java 개발 환경

이클립스, intelliJ 등의 IDE(통합 개발 환경. Integrated Development Environment)가 존재함. 본 수업에서는 intelliJ를 공식적으로 사용함.

VSC를 사용해도 간편함. 이때 intelliJ처럼 Java 프로젝트를 생성해서 사용할 수도 있고 그냥 Java 파일만 만들어서 사용할 수도 있음.

1.1.2. Java 파일과 폴더

IDE(intelliJ)로 Java 프로젝트를 생성하면 .idea, src 파일이 존재함. src 파일에는 .java 확장자를 사용한 소스 파일을 작성함. 이 소스 코드를 컴파일하면 out 파일이 생성되고, 그 내부에 확장자가 .class인 파일이 생성됨.

c에서는 .c 파일을 컴파일하면 .obj 파일이 생성되고, 이를 링크하면 .exe로 실행 파일이 생성되지만, java에서는 .java를 컴파일하면 .class 파일이 생성됨. 이때 .class 파일은 실행 파일이 아니라 바이트 코드임. 자세한 설명은 뒤에서 함.

더 자세한 파일 구조 또한 뒤에서 설명함.

1.1.3. Java의 버전

최근 Java는 lts(Long Term Version)와 실험적인 신규 버전들을 구분하여 출시하고 있음. lts는 안정성을 우선으로 하는 버전들로, 비교적 긴 시간 동안 표준으로 사용되는 버전임.

이 수업에서는 Java 21 버전을 사용함.

1.1.4. 자바 배포판 종류

Java SE : 자바 표준 배포판. 데스크탑과 서버 등에서 사용.
Java ME : 자바 마이크로 배포판. 휴대전화 등 제한된 리소스의 하드웨어에서 사용.
Java EE : 자바 기업용 배포판. 다중 사용자 등 지원.

1.2. 개발 관련 지식

1.2.1. 프로그래밍 언어

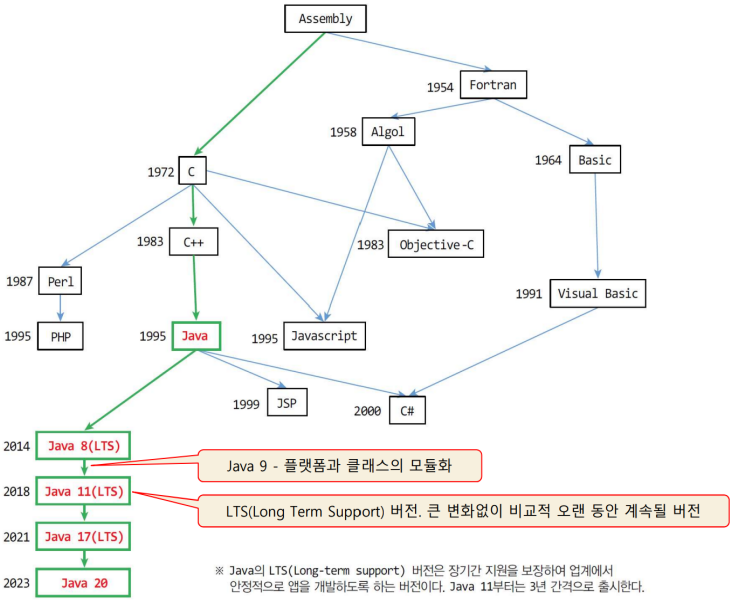
프로그램 언어(PL, Programming Language)는 기계어(machine language), 어셈블리어, 고급언어 순서로 발전되어 왔음. cpu로부터 사람에게 더 가까워진 것.

기계어는 0과 1로 이뤄진 언어로, 기계(cpu)의 언어임.

어셈블리어는 기계어 명령을 니모닉 기호(mnemonic symbol)로 일대일 대응시킨 언어. ADD, SUB, MOVE 등과 같이 표현하기 쉬운 상징적인 단어를 니모닉 기호라고 함.

고급언어는 사람이 이해하기 쉽고, 자료구조/알고리즘 등을 표혀하기 위해 고안된 언어로, 크게 객체지향언어와 절차지향언어로 구분됨. 당연하게도, Java는 고급언어이고, 현재 객체지향언어의 성질을 가지고 있음. 현시점 다양한 기술이 접목되었기에 완전한 객체지향언어는 아님.

PL의 발전 순서는 아래와 같음.



1.2.2. 절차지향언어와 객체지향언어

절차지향언어는 함수에 집중하고, 객체지향언어는 변수(데이터)에 집중함. 절차지향언어에서 객체지향언어 순서로 발전했음.

소스코드가 굉장히 길어지고 생산성이 떨어질 때 객체지향언어는 소스코드를 더 효율성 있게 관리할 수 있음.

1.2.3. 폰 노이만 아키텍처와 어셈블리어

엘런 튜링이 제시한 컴퓨터 구조에 대한 이론(유니버스 머신)을 기반으로 폰 노이만은 최초의 컴퓨터 구조인 폰 노이만 아키텍처를 만들어냈음.

프로그램은 데이터와 데이터의 처리 과정으로 이루어지는데, 튜링은 data1 -> process -> data2의 구조를 생각했음.

이후 최초의 어셈블리는 폰 노이만의 제자가 만들었음.

1.2.4. API

Definition 1 *API(application programming interface)는 일종의 소프트웨어 인터페이스이며, 다른 소프트웨어에 서비스를 제공함.*

api는 레이어/층 역할을 하며 더 편리한 프로그래밍을 할 수 있게 함.

c/c++의 측면에서의 api는 .obj 형태의 함수들로 구성되어 있기 때문에, os가 제공하는 라이브러리로 이해할 수 있음.

모든 프로그램은 하드웨어를 조작함으로써 동작하는데, os가 하드웨어의 조작을 대신함. 프로그램에서 os의 함수를 호출하면 os가 하드웨어를 대신 조작해주는 것. os에는 공개되어 있는 함수들도 있고 공개되지 않은 함수들도 있는데, 사용자에게 공개되어 사용할 수 있게 한 함수들의 집합이 api임. 즉, 프로그램을 위한 인터페이스라고 할 수 있음. 물론 API가 이런 함수들만을 가리키는 용어인 것은 아님.

프레임워크 등도 api로서 기능하는데, 프로그램-프레임워크-os의 계층 구조를 가지며 os와 프로그램 사이의 인터페이스 역할을 하기 때문임.

1.2.5. 메소드

Definition 2 *class 내부에 있는 함수를 메소드라고 함. Java에서는 class 외부에 코드를 작성할 수*

없기 때문에 메소드만이 존재함.

Java에서 메소드는 아래와 같은 형태를 가짐.

```
접근지정자 반환타입 식별(매개변수 목록)
{
    코드
}
```

순수한 객체지향언어에서는 메서드만 존재할 수 있는데, 많은 객체지향언어들이 편의성을 위해 함수도 작성 가능하게 함. 사실상 순수한 객체지향언어는 거의 존재하지 않음.

1.2.6. 키워드와 예약어

Definition 3 *키워드는 문법 내에서 특정한 목적을 위해 사용하는 단어임.*

예약어는 식별자로 사용할 수 없도록 미리 예약해놓은 단어임.

둘 사이에는 미묘한 의미 차이가 있음. 어떤 언어에서는 컴파일러의 업그레이드 이후 사용 가능성이 있어 예약어로 설정되었지만 현재 키워드는 아닌 단어가 있기도 함.

1.3. Java의 특징

1.3.1. Java의 역사

90년대 중반은 c가 대세이던 시절이었는데, c는 플랫폼 종속적이었음. 한 플랫폼에서 실행 가능하더라도 다른 플랫폼에서는 다시 새로 만들어야 했음.

Java는 1991년부터 선마이크로시스템즈의 제임스 고슬링(James Gosling)에 의해 가전제품용 소프트웨어로서 개발이 시작되었고, 1995년 발표되었음. 이후 2009년에 선마이크로시스템즈를 오라클에서 인수하면서 오라클 소유의 기술이 되었음.

Java는 객체지향언어로 시작되었지만, 현재는 다른 언어들의 특징들을 차용하여 완전한 객체지향언어는 아니게 됨.

현재 Java는 os 위에서 돌아가는 app(application, 응용 프로그램), 웹 서버에서 실행되는 app, 안드로이드 app 등의 개발에 사용됨.

1.3.2. c/c++의 플랫폼 종속성

c/c++이 플랫폼 종속적(platform dependence)인 이유는 3가지가 있음.

1. cpu마다 기계어가 다름.
각 cpu가 이해할 수 있는 명령어 집합(어셈블리어, 기계어)으로 컴파일을 해 줘야 하기 때문에 매번 새로 컴파일해야 함.
-> Java 소스코드는 JVM에서 해석하고 실행되기 때문에 cpu 독립적임.

2. os마다 실행파일 형식이 다름.
링크 시에 실행파일의 맨 앞 비트에 파일 정보, 위치 등의 정보를 담은 헤더(ex, ELF header)를 넣는데, 이 헤더가 os마다 다름. 같은 cpu이면 .obj파일을 동일하게 읽을 수 있지만, 해당 os 환경에서 링크를 다시 해줘야 헤더를 다시 삽입하여 실행 파일을 적절한 형식을 만들 수 있음.
-> JVM에서 바이트 코드가 실시간으로 번역되어 실행되므로, 실행 파일을 만들지 않음.

3. os마다 api가 다름.
os별로 포함하고 있는 api가 다르기 때문에 소스 코드 자체를 다시 작성해야 함. 실질적으로 코드를 수정하지 않더라도, 라이브러리에서 가져와 사용하는 표준 입출력 함수 등은 os의 api를 사용하기 때문에 해당 os에 호환되는 라이브러리를 새로 사용해야 함. (물론 호환성 있게 작성된 라이브러리도 존재함.)
-> Java에서는 os의 api 대신, JVM과 동일한 계층에 위치하는 Java API를 사용함.

1.3.3. WORA

선마이크로시스템즈는 Java로 c/c++의 플랫폼 종속성(platform dependence)을 해결한 WORA(Write Once Run Anywhere, platform independence)를 구현했음. WORA는 마이크로시스템즈가 만든 Java의 표어임. 한 번만 작성하면 임의의 cpu, 웹브라우저 등 환경에 종속적이지 않게 실행이 가능하다는 것.

자바 소스 코드를 컴파일하여 바이트 코드를 만들고, 그 바이트 코드를 플랫폼 종속적으로 설치되어 있는 JVM에서 해석하고 실행하는 것.

1.3.4. 바이트 코드

Definition 4 *바이트 코드(byte code)(.class)는 기계어는 아니지만 기계어와 유사한 구조를 가진 파일로, 자바 소스 코드(.java)를 컴파일한 것임. 바이트 코드는 JVM에 의해 해석되고 실행됨.*

1.3.5. 객체지향언어

Java는 플랫폼 독립적인 것 외에도, 객체지향언어라는 큰 특징이 있음. Java는 객체지향언어로서 상속성, 다형성, 캡슐화라는 특성을 가짐.

클래스로 캡슐화를 함. 클래스 내부에 변수(필드)와 함수(메소드)를 작성함.

1.3.6. 기타 특징들

1. Java는 운영체제의 도움 없이 자체적으로 멀티스레드를 지원함.

2. Java에는 메모리 할당 기능만 존재하고, 메모리 반납은 JVM이 자동으로 수행함.
이를 가비지 컬렉션(garbage collection)이라고 하는데, 가비지 컬렉션이 어떤 시점에 수행되는지를 사용자는 알 수 없음. 그렇기에 특정 자원에 제한이 있는 경우 불편할 수 있음.

3. Java는 type 체크가 엄격함.
그래서 각각의 자료형과 형변환을 잘 확인해야 함.

4. 포인터 개념은 존재하나 명시적인 포인터 문법이 없음.
포인터가 존재하지 않기 때문에 실제 메모리 크기가 중요하지 않음.

5. Java 컴파일러는 두 종류가 있음. 바이트 코드로 컴파일하는 컴파일러, 바이트 코드를 기계어로 컴파일하는 JIT(Just In Time) 컴파일러가 그것임.

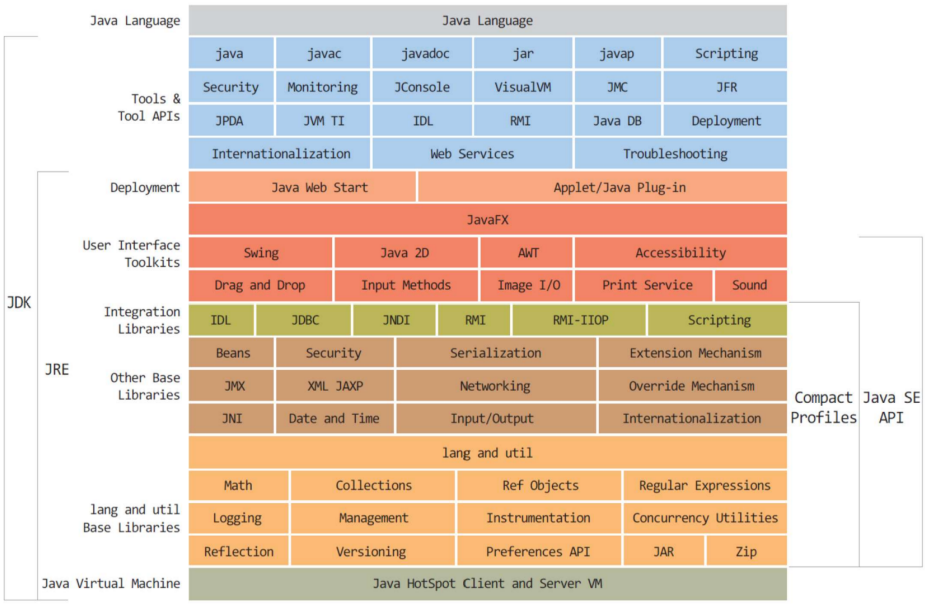
1.4. Java SE

1.4.1. Java SE

Definition 5 *Java SE(Standard Edition)은 자바라는 언어가 어떤 문법적 구성을 가졌는지 정의하는 자바의 표준안을 의미함.*

Java SE는 Java 언어와 JDK로 구성되고, JDK에는 JRE와 Tool이, JRE에는 JVM과 Java API등이 들어 있음.

Java 프로그램이 돌아가는 계층을 생각해 보면 app-JVM/JavaAPI-os-cpu임.



1.4.2. JDK

Definition 6 *개발자가 자바 기반 애플리케이션 개발을 위해 다운로드하는 소프트웨어 패키지를 자바 개발 키트(JDK, Java Development Kit)라고 함.*

JDK는 JRE와 Tool 등을 포함함. 또한 JDK는 (.class로의)컴파일러를 포함하고 있음.

JDK에는 JVM과 Java API, Tool 등이 들어 있기 때문에, JDK만 있으면 Java를 사용한 개발이 가능함. 이클립스, intelliJ, VSC는 사실상 JDK를 사용하기 위한 인터페이스임.

Java는 오라클(Oracle) 소유 기술이기에 오라클 홈페이지에서 JDK를 다운받을 수 있음. 현재는 오픈 소스 JDK가 많이 나와있어서 아무거나 사용하면 됨.

1.4.3. JRE

Definition 7 *JRE(Java Runtime Environment)는 JVM, Java API 등을 포함하는 자바 실행 환경을 말함.*

JRE에는 JVM, Java API 등이 들어 있음.

과거에는 오라클에서 JRE를 배포했지만, 지금은 배포하고 있지 않음.

1.4.4. JVM

Definition 8 *java의 바이트 코드를 실행하는 자바 가상 머신을 JVM(Java Virtual Machine)이라고 함.*

JVM은 플랫폼 종속적으로, 각 플랫폼마다 구현되어 있음.

JVM은 인터프리터 방식으로 .class 파일의 바이트 코드를 가져와 해석하고 실행함. 즉, 필요한 클래스를 그때그때 가져와 내부의 바이트 코드를 해석하고 실행하는 것.

인터프리터 방식을 사용해 실시간으로 해석하고 실행하는 것은 한 번에 전부 컴파일하고 실행하는 것에 비해 느림. 그래서 Java는 소스 코드를 미리 바이트 코드로 컴파일해 둬서 시간을 단축시킴.

JVM은 오라클, IBM 등에서 개발하여 배포함.

1.4.5. Java API

Definition 9 *JVM과 같은 계층에서 Java 프로그램의 실행을 돕는 api. JDK에 포함된 클래스 라이브러리임.*

JVM 위에 .class 파일을 올리고 Java API를 추가로 사용하는 것. C의 라이브러리가 .obj 파일로 구성되어 있는 것처럼, Java API도 .class 파일로 구성되어 있음.

1.4.6. Java Tool

JDK에는 Java 개발을 위한 여러 Tool들이 존재함. 이 Tool들은 JDK(프로젝트 폴더 아님.)의 bin 디렉토리에 들어 있음.

javac : 자바 소스를 바이트 코드로 변환하는 컴파일러
java : 자바 응용프로그램 실행기. JVM을 작동시켜 JIT 컴파일을 수행하도록 하고, 자바프로그램을 실행
javadoc : 자바 소스로부터 HTML 형식의 API 도큐먼트 생성
jar : 자바 클래스들(패키지포함)을 압축한 자바 아카이브 파일(.jar) 생성 관리
jmod : 자바의 모듈 파일(.jmod)을 만들거나 모듈 파일의 내용 출력
jlink : 응용프로그램에 맞춘 맞춤형(custom) JRE 제공
jdb : 자바 응용프로그램의 실행 중 오류를 찾는 데 사용하는 디버거
javap : 클래스 파일의 바이트 코드를 소스와 함께 보여주는 디어셈블러

1.4.7. javadoc

Definition 10 *Java 소스 코드의 특정 위치에 주석을 달아 놓고 명령을 실행하면, 선언문과 주석을 바탕으로 api 문서를 HTML로 생성하는 Tool. 이때 주석은 /** */ 로 작성함. 주석 작성 후 프롬프트에 아래의 명령을 입력해 HTML 파일을 생성함.*

```
javadoc HelloDoc.java
```

주석 작성은 아래와 같이 함.

```
/**
javadoc 사용 예제를 위한 클래스
*/
public class HelloDoc
{
    /**
    두 정수의 합을 구하는 메소드
    @param i 합을 구할 첫번째 정수형 인자
    @param j 합을 구할 두번째 정수형 인자
    @return 두 정수의 합을 리턴
    */
    public static int sum(int i, int j)
    {
        return i + j;
    }
}
```

사용하는 패키지의 함수들에 대한 java doc을 웹에서 찾아보자. 이 문서에 익숙해지라고 함.

1.5. 문법 기초

1.5.1. 기본 형식

Java 문법의 기본 형식은 아래와 같음. 자세한 내용은 뒤에서 다시 설명함.

```
public class HelloWorld2024\
```

```
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

class는 class 키워드로 선언함. public을 지정하면 다른 클래스에서 접근이 가능함. class 뒤에 붙은 이름은 해당 소스 파일명과 동일해야 함. 이때 파일명은 관습적으로 각 단어의 첫 글자를 대문자로 씀. (ex. HelloWorld2024)

Java 소스 코드는 하나 이상의 클래스로 구성됨.

class 키워드 위치에는 자격에 따라 들어갈 수 있는 요소들이 달라짐. 이때 자격은 class, enum 등 다양하지만, 이는 개발자의 편의를 위한 것임. 실제로 컴파일하면 모두 클래스로 취급되어 같은 구조로 변함. (.class 파일이 됨.)

println은 개행을 자동으로 해 주는 출력 메소드. sout만 쳤을 때 자동완성되기도 함.

1.5.2. 전역 변수/함수

Java에서는 class 블록 밖에 코드를 작성하지 못함. 즉, c에서와 같이 전역 변수 또는 함수를 작성할 수 없고 모두 블록 내부에 작성해야 함.

Java에서는 class 내부이면서 메소드 바깥 부분을 전역으로, 메소드 내부를 지역으로 생각함.

1.5.3. 소스 파일에 class 작성 방식

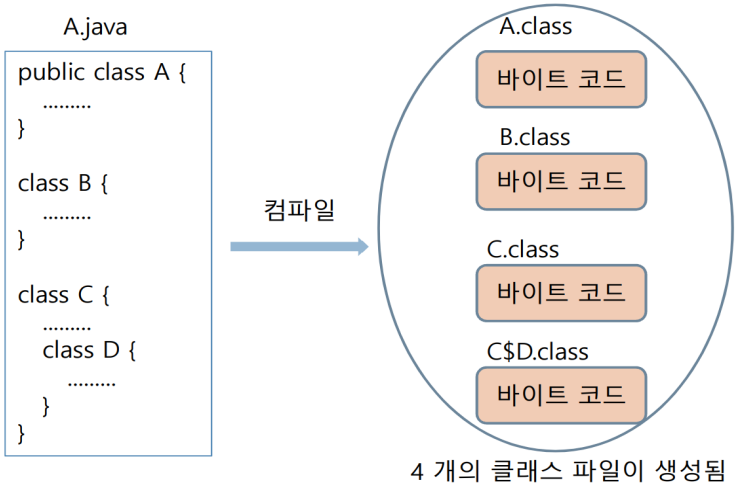
소스 파일에는 적어도 하나의 class가 존재해야 하는데, 이때 하나의 소스 파일에 여러 개의 class를 작성하거나, class 안에 class를 작성할 수 있음.

public class는 하나만 가능. public class의 class 이름과 소스 파일의 파일명은 같아야 함. 물론 public class가 소스 파일 내에 없으면 파일명은 아무거나 작성해도 되지만 아무도 이렇게 하지는 않음.

하나의 파일에 여러 개의 class를 작성하거나 class 내부에 class를 작성해도, .class 파일은 각 class 별로 생성됨. Java에서는 필요한 클래스를 그때그때 가져와 쓰기 때문. 이때 클래스 안에 작성된 클래스는 C$D.class 등으로 생성됨. C클래스 내부에 있는 D클래스라는 의미.

이때 class 내부에 존재하는 class는 nested class라고 함.

기본적으로는 파일당 하나의 class만 작성하는 것이 좋음. 즉, 여러 개의 class를 사용하려면 여러 개의 파일을 만드는 것. 한 파일에 여러 개의 class를 작성하는 것은 subroutine으로 사용할 부분을 숨겨서 따로 작성하기 위한 것임. subroutine은 함수 등에서 작성자의 편의성 등을 위해 기능을 따로 빼서 함수나 클래스 등으로 작성하는 것을 말함. 실제로 이런 의도 없이 한 파일 내에 여러 class를 작성하면 해당 class 사용자가 class에 접근하기가 아주 불편해짐. 괜히 public class명과 파일명이 같은 것이 아님.



1.5.4. 주석

Definition 11 *Java에서는 //로 한 줄 주석, /* */로 다중 줄 주석을 작성할 수 있음.*

1.5.5. expression과 statement

Definition 12 *연산자와 피연산자로 이뤄진 코드를 expression(식)이라고 함.*

;으로 마무리되는 코드를 statement(문)라고 함.

컴파일러는 expression을 값과 형으로 평가함. 즉, expression은 나름의 값과 형을 가짐. 어떤 expression이나 literal, 변수 등에 대해서는 값과 형을 생각할 수 있음.

Java의 모든 단일 문장들은 ;로 끝남.

1.5.6. 스택과 힙

Java에서 지역변수는 스택에 저장됨. 스택에 저장된 변수에는 '값이 할당되지 않은 상태'가 존재함. 변수 선언 이후 값을 대입하지 않으면 c에서처럼 쓰레기 값이 들어있는 것이 아니라, '값이 할당되지 않은 상태'로 취급되는 것.

'값이 할당되지 않은 상태'의 변수를 사용하려는 코드를 작성하면 컴파일 에러가 발생함.

반면에 Java에서 객체는 힙에 저장됨. 힙에서는 그런 상태가 존재하지 않고, 값을 대입하지 않아도 모든 비트에 0이 들어가 있음. 즉, 모든 비트가 0으로 자동 초기화됨.

그래서 힙에 있는 데이터는 int로 읽으면 0, double로 읽으면 0.0, reference로 읽으면 null임.

2. 기본 문법

Java의 특징적인 부분만 정리.

2.1. 식별자

2.1.1. 식별자

Definition 13 *클래스, 변수, 상수, 메소드 등에 붙이는 이름을 식별자(identifier)라고 함.*

필드와 메소드는 식별자가 서로 겹쳐도 됨.

2.1.2. 식별자 작성 규칙

1. '_', '$'을 제외한 특수 문자, 공백 또는 탭은 사용 불가.
2. 유니코드 문자(한글 등) 사용 가능.
3. 키워드 사용 불가.
4. 첫 번째 문자로 숫자 사용 불가.
5. 길이 제한 없음.
6. 대소문자 구분함.
7. true, false, null literal 사용 불가. (예약어)

'_' 와 '$'도 식별자 첫 번째 문자로 사용할 수 있으나 일반적으로 잘 사용하지 않음.

2.1.3. 관습적 규칙

1. class
camel 표기법 사용.
명사로 붙임.
class 내부 메소드 등에 대해 포괄적으로 작성함.
ex. HelloWorld

2. 변수, 메소드
첫 글자가 소문자인 camel 표기법 사용.
변수는 명사로 붙이고, 메소드는 동사로 시작(명령문)하도록 붙임.
boolean 변수와 반환값이 boolean형인 메소드는 be동사로 시작하는 의문문으로 붙임.
ex. helloWorld, printHelloWorld

3. constant
snake 표기법 사용.
ex. HELLO_WORLD

회사 등에서 진행하는 큰 프로젝트에서는 coding convention을 정해서 문서화함.

2.2. 자료형

Java의 자료형은 primitive와 reference로 나뉨.

2.2.1. primitive

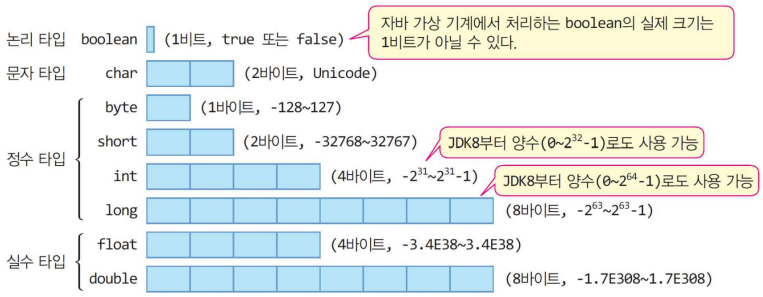
Definition 14 *primitive(기본 자료형, 원시 자료형)는 Java에서 기본으로 제공되는 자료형. 총 8개의 자료형이 있음.*

boolean(1bit), char(2byte), byte(1byte), short(2byte), int(4byte), long(8byte), float(4byte), double(8byte)이 있음.

boolean형에는 true와 false literal만 대입할 수 있음.
JVM이 메모리 공간을 어떻게 할당하는지에 따라 boolean의 메모리 크기가 1비트가 아닐 수도 있음. Java는 명시적인 포인터 문법이 존재하지 않아 주소를 직접 다룰 수 없기 때문에 메모리 크기가 다르게 사용되어도 큰 문제가 없음. 입력과 출력이 제대로 나오가만 하면 됨.

Java에서는 유니코드로 문자를 표기함. 유니코드에는 2바이트가 필요하기 때문에, char가 2바이트임.

Java에는 unsigned 개념이 존재하지 않음. 모든 primitive가 음수를 포함함.



2.2.2. reference

Definition 15 *메모리에 reference가 저장되는 자료형.*

reference 변수는 주소를 대입하여 사용하는 것처럼 생각할 수 있지만, 실제로는 주소가 아닌 reference 값을 대입하여 사용하는 것임. reference는 주소는 아니지만 주소처럼 이해할 수 있는 값이며, JVM에서 실행할 때 이 reference가 주소로 바뀌어 사용됨.

배열, class, interface등에 사용됨.

primitive가 아니면 전부 reference임.

reference 자료형은 reference를 저장하기 때문에 다른 변수에 reference를 대입하면 해당 변수도 reference로 동일한 곳을 가리키게 됨.

2.2.3. 문자열

Java에서 문자열은 String class로 저장함. Java에서 문자열은 자료형은 primitive에 정의되어 있지 않기 때문에 reference를 사용하는 것.

String 클래스에는 +연산이 재정의되어 있음. + 연산의 피연산자에 문자열이 있는 경우 +는 문자열 연결 연산을 수행함. 이때 literal과 연산을 하면 literal은 문자열로 자동 형변환됨. 이때 직관적으로 해당 값을 문자열로 쓴 것처럼 형변환됨.

아래는 String class의 사용 예시임.

```
String toolName="JDK";
System.out.println(toolName + "가 출시됨.");
```

참고로 길이가 0인 문자열도 가능함. ""으로 작성이 가능한 것.

+를 이용한 연결 연산은 오버로딩되어 있지만, ==를 이용한 비교 연산은 오버로딩되어있지 않음. 대신 Objects.equals() 메소드 등을 사용하면 쉽게 비교할 수 있음. Objects.equals(str1, str2) 등으로 사용함.

2.2.4. 인자 전달 방식

java에서는 값에 의한 호출을 따름. 즉, primitive는 값이 전달되고 reference는 reference값이 전달됨.

물론 reference를 전달하면 참조에 의한 호출처럼 사용할 수 있음.

c에서의 방식과 유사함.

2.3. 변수

2.3.1. 변수

Definition 16 *프로그램 실행 중에 값을 임시 저장하기 위한 공간. 선언 시에 지정한 자료형만큼의 메모리가 할당되고, 해당 자료형에 따른 데이터 처리를 함.*

2.3.2. 변수의 선언과 사용

c에서의 문법과 동일함.

2.3.3. var 키워드

Java 10부터 var 키워드로 변수를 선언할 수 있음. 초기화 시에 지정한 데이터의 형으로 해당 변수의 자료형을 지정하는 것.

var는 지역변수에만 사용이 가능함. 즉, 메소드 안에서만 사용할 수 있음.

var를 사용하면서 초기화를 하지 않으면 에러가 발생함. 이때 var로 선언하는 그 문장에서 초기화를 해야 함. 아래는 그 예시임.

```
var name = "jhun"; // name은 String형으로 지정됨

var name; // 에러 발생
```

2.3.4. 변수의 자동 초기화

전역변수 위치(class 안, 메소드 밖)에 변수를 선언하면 자동으로 0으로 초기화됨. 이때 primitive이면 0, 0.0, \u0000, false로 초기화되고, reference이면 null로 초기화됨.

전역변수 위치에 static을 사용하면(class 소유의 변수) 0으로 자동 초기화되고, 작성하지 않은 경우에도 해당 변수는 필드이므로 객체 생성 시에 0으로 자동 초기화됨(힙).

2.4. literal

2.4.1. literal

Definition 17 *코드에서 직접 표현한 값.*

정수, 실수, 문자, 논리, reference 관련 literal이 있음.

상수는 literal과 constant로 나눌 수 있음. c의 워딩을 사용하면 literal은 상수, constant는 상수 변수임. 본 수업의 강의 슬라이드에서는 literal을 리터럴, constant를 상수로 명명함. 이 필기에서는 literal과 constant로 작성하겠음.

2.4.2. 정수 literal

정수 literal에는 10진수, 8진수, 16진수, 2진수 표현법이 있음. 8진수는 앞에 0을, 16진수는 앞에 0x를, 2진수는 앞에 0b를 붙여 작성함.

8, 16, 2진수 표현법에서 주의해야 할 점은, 해당 값은 2의 보수를 표현한 것이 아니라 단순히 2진수로 표현한 것임. 즉, 이들은 전부 unsigned로 생각해야 함. (중간고사 출제 의심됨.)

8진수, 16진수는 2진수의 표현에 있어 편리하기 때문에 사용함.

정수 literal은 int형으로 평가됨. 정수 literal 뒤에 L 또는 l을 붙여 long으로 지정할 수도 있음.

JDK7부터 가독성을 위해 정수/실수 literal에 '_' 사용이 허용되었음. '_'은 코드에서만 보이고 실제 연산이나 출력 시에는 없는 것으로 처리됨. 연속해서 여러 개의 '_'를 작성할 수도 있음. 단, 앞뒤로 숫자가 존재해야 함. 즉, 숫자의 맨앞이나 뒤, 0x등과 붙어 있는 경우에는 에러가 발생함.

2.4.3. 실수 literal

실수 literal은 소수점 표기법과 지수 표기법이 있음. 지수 표기법은 1234E-4 등의 형태를 가짐.

실수 literal은 double형으로 평가됨. 실수 literal 뒤에 f나 F를 붙여 float을, d나 D를 붙여 double을 지정할 수도 있음.

2.4.4. 문자 literal

문자 literal은 작은따옴표(' ')로 표현하거나 또는 유니코드를 바이트로 작성하는 방법으로 표현함.

유니코드를 바이트로 나타내는 방법에서는, \u 뒤에 코드 포인트를 작성함. 코드 포인트는 4자리의 16진수(총 2바이트)를 말하는데, 이는 해당 문자의 유니코드 값임. (ex. \u 0041)

문자 literal은 char형으로 평가됨.

추가로, 특수문자 literal은 백슬래시를 사용하여 작성함.

종류	의미	종류	의미
'\b'	백스페이스(backspace)	'\r'	캐리지 리턴(carriage return)
'\t'	탭(tab)	'\"'	이중 인용부호(double quote)
'\n'	라인피드(line feed)	'\''	단일 인용부호(single quote)
'\f'	폼피드(form feed)	'\\'	백슬래시(backslash)

2.4.5. 논리 literal

논리 literal은 true와 false 2가지만 존재함. (소문자로만 사용 가능)

참고로, c에서와 달리 자바에서는 참거짓을 true와 false로만 표현이 가능함. c에서는 true가 1, false가 0으로 치환되어 0이 아닌 값은 참이고 0은 거짓으로 취급이 되었었음. Java에서 true와 false는 boolean만을 위한 예약어임. Java에서는 자료형을 엄격하게 다루기 때문에 boolean 변수에는 true와 false만 대입이 가능함. 1 등을 대입하려고 하면 컴파일 에러가 발생함.

논리 literal은 boolean으로 평가됨.

2.4.6. reference 관련 literal

1. null literal
null literal은 reference형 변수에 대입하여 저장된 값이 없음을 나타냄. (소문자로만 사용 가능)
primitive 변수에 사용 불가.

2. 문자열 literal
문자열 literal은 큰따옴표("")로 표현함.
문자열 literal은 String 객체로 평가됨. 즉, "test"등과 같은 문자열 literal은 해당 객체의 reference를 값으로 가짐.

2.5. constant

2.5.1. constant

Definition 18 *값의 대입이 한 번만 가능해서 상수처럼 사용이 가능한 변수를 constant라고 함.*

constant는 final 키워드를 사용하여 선언함.

```
final double PI = 3.141592;
```

이때 값의 대입은 초기화로 하지 않아도 됨. 그저 첫 대입이 constant의 값이 됨.

Java에서는 변수 선언 이후 값을 대입하지 않으면 c에서처럼 쓰레기 값이 들어있는 것이 아니라, '할당하지 않은 상태'로 취급함. 이런 측면에서 생각해 보면 constant 선언 이후 첫 대입이 초기화처럼 취급되는 것을 이해할 수 있음.

필드를 final로 선언하고 선언문에서 초기화를 하지 않았다면 컴파일 에러가 발생함. 선언문에서 초기화를 하지 않았다면 다른 메소드에서 초기화를 해야 하는데, 해당 메소드가 호출되지 않는다면 초기화되지 않을 수 있기 때문(인 듯)임.

constant에는 주로 public과 static을 사용하여 public static final double PI = 3.14; 이런 식으로 사용함.

2.6. primitive의 형변환

Java에서는 자료형에 엄격하므로 형변환에 유의해야 함.

2.6.1. 자동형변환

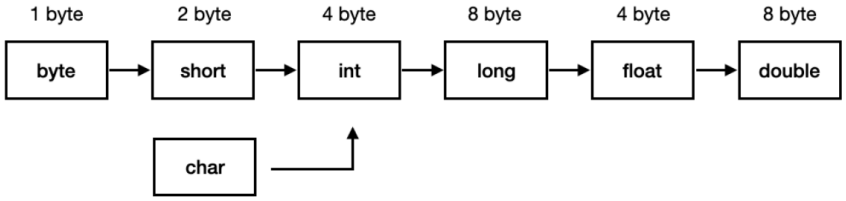
Definition 19 *expression에서 자료형이 일치하지 않을 때 컴파일러에 의해 자동으로 이루어지는 형변환을 자동형변환이라고 함. 이건 primitive끼리의 자동형변환을 말함.*

자동형변환은 아래의 그림과 같은 순위로 이뤄짐.

바이트에 상관없이 실수 자료형이 정수 자료형보다 큼. 표현 가능한 수의 범위를 계산해 보면 실수가 훨씬 크기 때문임.

boolean은 자동이든 강제든 형변환의 개념이 적용되지 않음.

대입 연산 또한 expression이므로, 변수에 값을 대입할 때에도 자동형변환이 일어남. (ex. long a = 32; 32가 int이므로.)



2.6.2. 대입 시의 더 작은 크기로의 자동형변환

기본적으로 메모리 크기가 큰 자료형은 그것보다 메모리 크기가 작게는 자동형변환되지 않음. 이렇게 하려면 강제형변환을 해 줘야 함.

대입 연산이 아닌 expression에서는 더 큰 크기로 자동형변환이 되지만, 대입 연산에 대해서는 더 작은 크기로의 형변환이 이뤄질 수도 있음. 이때 literal과 constant에 대해서는 더 작은 크기로의 형변환이 허용됨.

1. constant/literal에 대한 연산
그 값이 해당 변수의 범위에 포함되는 constant/literal은 대입될 수 있음. (ex. byte v = 10; 10이 int이므로.) 이때 해당 변수의 메모리를 넘어가는 값을 대입하면 에러가 발생함. literal은 런타임 중에 그 값이 바뀌지 않으므로 컴파일러가 허용하는 것(으로 보임).

2. 변수에 대한 연산
하지만 변수에 변수를 대입할 때 대입하는 변수가 호환되지 않는 값을 가질 가능성이 있다면(더 큰 값이 들어올 수도 있는 등), 값에 상관없이 에러가 발생함. 변수에는 어떤 값이 들어와서 대입될지 모르기 때문에 컴파일 시점에 컴파일러는 직접 확인할 수 없으므로 컴파일 에러를 냄.

2.6.3. char형의 자동형변환

1. char가 형변환될 수 있는 것은 int 이상부터임.
char형은 양수 값만을 가지므로 범위가 맞지 않아 short로 자동형변환되지 못함. short는 같은 2바이트이지만 음수까지 표현하기 때문에 char의 값을 전부 표현하지 못하는 것. 그래서 char는 int 이상으로만 자동형변환이 가능함.

2. char형 변수에 대입될 수 있는 것은 byte/short/int형 constant/literal 뿐임.
더 작은 크기로의 자동형변환에서와 동일한 원리임. 변수는 음수를 가질 수 있으므로 불가능함. 다만, 표현 가능 범위 이내여도 long/float/double constant/literal은 대입될 수 없음.

2.6.4. 강제형변환

Definition 20 *개발자가 지정하여 강제로 이루어지는 형변환을 강제형변환이라고 함.*

()(casting operator) 안에 변환할 자료형을 작성하여 지정함.

```
byte b = (byte)n;
```

2.7. 연산자

연산 시에는 expression의 자료형이 어떻게 될 지를 생각하자.

2.7.1. 연산자 목록

c에서의 문법과 거의 동일함.

1. 산술 연산자 : +, -, *, /, %(나머지)

2. 증감 연산자 : ++, –
전위 후위 연산 존재함.

3. 대입 연산자 : = , 복합 대입 연산자
복합 대입 연산자는 *= 꼴로 작성함.

4. 비교 연산자 : <, >, <=, >=, ==, !=

5. 논리 연산자 : !, ||, &&, ∧
∧는 논리 연산자로도 사용할 수 있고, 비트 연산자로도 사용할 수 있음. 둘 다 의미는 XOR임.
true, false literal과 다른 literal을 비교할 수는 없음.

6.조건 연산자 : ?:
삼항 연산자 ?:만 존재함. 형식은 condition ? opr2 : opr3 임.
condition 값이 true이면 opr2를, false이면 op3를 수행하고 그 값을 expression의 값으로 가짐.
condition에는 true 또는 false만 올 수 있는 것 유의.

7. 비트 논리 연산자 : &(AND), |(OR), ∧(XOR), (NOT)

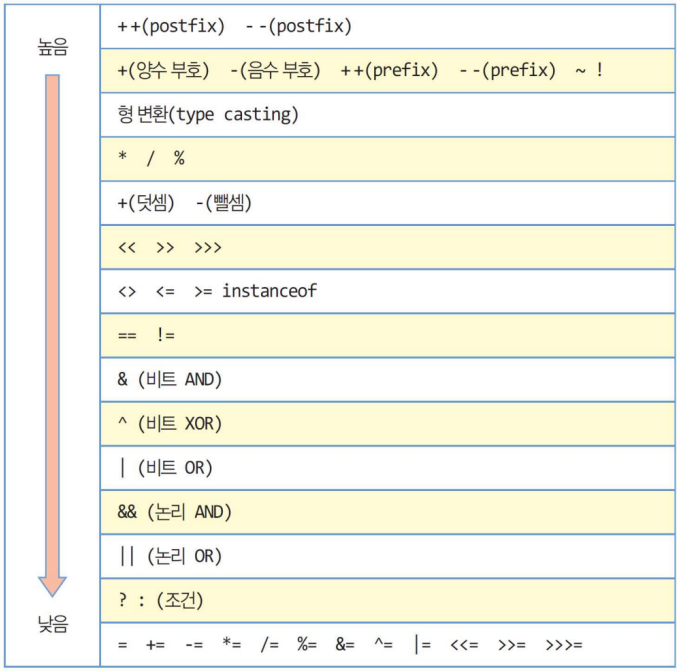
8. 시프트 연산자 : », «, »>
»는 산술적 오른쪽 시프트 연산자임. 생기는 빈자리는 원래의 최상위 비트로 채움. 2의 거듭제곱으로 나눈 것과 동일함.
»>는 논리적 오른쪽 시프트 연산자임. 생기는 빈자리는 0으로 채움.
«는 산술적 왼쪽 시프트 연산자임. 생기는 빈지라는 0으로 채움. 2의 거듭제곱을 곱하는 것과 동일함.
비트 논리 연산자와 시프트 연산을 사용하여 비트 마스킹을 할 수 있음.
시프트 연산은 2를 곱하고 나누는 연산보다 속도가 빠름.

2.7.2. 연산자 우선순위

Java의 연산자 우선순위는 아래의 그림과 같음.

괄호로 묶인 것은 최우선순위임.

같은 순위의 연산자인 경우.
대입 연산자, 증감 연산자, -(음수 부호), !, 캐스트 연산자는 오른쪽에서 왼쪽으로 연산. (대입 연산자와 단항 연산자들)
나머지는 모두 왼쪽에서 오른쪽으로 연산.



2.8. 제어문

c의 문법과 거의 동일함.

2.8.1. 조건문

조건문의 조건 expression의 값은 true 또는 false literal이어야 함.

if, else if, else, switch 사용 가능.

2.8.2. switch

1. 기본적인 사용법.
switch의 expresssion과 내부의 case의 expression의 값이 동일하면 해당 case를 실행함.
switch문의 조건 값은 문자/정수/문자열 형을 가지는 변수/상수 값이어야 함. 단, long은 불가능함.
case문의 값은 constant, 문자/정수/문자열 literal만 가능. 단, long형은 불가능함.
c에서와 동일하게, break으로 끊지 않으면 아래의 case들을 계속 실행함.
case문을 레이블이라고 함.

case문의 값은 정적으로만 지정이 가능함. 이때 case문의 값에는 변수가 들어갈 수 없는 것이지, expression이 들어갈 수 없는 것이 아님. expression은 들어갈 수 있음. (시험..)

2. JDK21에 추가된 사용법.
->로 case별 값을 지정할 수 있음. 이때 switch는 해당 값을 반환함. 이 경우 case별 break는 생략함. case에 값을 여러 개 작성할 수 있음. ->에 중괄호를 지정하고 yield로 반환할 값을 지정할 수 있음.

아래는 그 예시임.

```
int test = 3;
char testVar = switch(test)
{
    case 1, 2, 3, 4 -> ‘a’;
    case 5, 6, 7, 8 -> {
    System.out.println("Printing b..");
    yield ‘b’;
```

```
    }
    default -> 'c';
}
System.out.println(testVar); // a 출력됨
```

2.8.3. 반복문

반복문의 조건 expression의 값은 true 또는 false literal이어야 함.

for 사용 가능.
for는 초기 작업, 반복 조건, 반복 후 작업으로 구분됨.
반복 조건이 비어 있으면 true로 취급됨.
초기 작업과 반복 후 작업은 ','로 구분하여 여러 문장 작성 가능. ;는 생략함.

while, do-while 사용 가능.

continue, break 키워드 사용 가능.

2.8.4. label을 사용한 continue, break

반복문 앞에 레이블을 붙일 수 있음. continue, break 사용 시에 해당 레이블을 지정하여 특정 반복문을 끝내거나 넘길 수 있음. 아래는 그 예시임.

```
LABEL:
for(int i = 0; i < N; i++)
{
    for(int j = 0; j < M; j++)
    {
        break LABEL;        // 바깥쪽 for문이 break됨.
    }
}
```

2.9. 기본 입출력

자세한 내용은 추후에 다룸.

2.9.1. 출력

출력 시에는 System.out.println(), System.out.print() 등을 사용함. println()은 출력 후 개행까지 추가로 한 것. 인자로는 문자열이나 변수 등을 작성함.

System.out은 프로그램 실행 시에 생성되는 자바의 표준 출력 스트림임.

2.9.2. 입력

입력 시에는 Scanner 객체를 사용함. 사용 방법은 아래와 같음. System.in은 프로그램 실행 시에 생성되는 자바의 표준 입력 스트림임.

이때 System.in은 입력값을 바이트 데이터로 넘겨줌. 이 값을 문자열, 정수 등으로 사용하려면 프로그램 내에서 변환해야 함. 여기서는 scanner의 메소드들을 사용함.

1. util 패키지에서 Scanner class 가져오기.
ex. import java.util.Scanner;

2. 스캐너 객체 생성.
new 키워드로 Scanner class의 객체를 생성할 수 있음.
ex. Scanner a = new Scanner(System.in);

3. 스캐너 메소드 사용.

2.9.3. Scanner 메소드들

scanner 메소드들은 아래와 같음.

입력에서 whitespace로 구분되는 덩어리들을 토큰이라고 함. scanner 메소드들은 기본적으로 토큰을 가져옴. 이때 메소드들이 whitespace를 읽지만 가져오지는 않음. whitespace를 가져오려면 줄 단위로 입력받아야 함.

입력이 들어오지 않아서 입력 메소드 위치에서 제어가 멈춰있는 것을 block되어있다고 함. enter키로 입력을 집어넣은 순간 block이 해제되고 메소드들이 토큰을 구분해 사용함.

특이한 점은 scanner에서는 char의 입력을 지원하지 않음. 한 문자씩 접근하고 싶으면 다른 함수를 쓰거나, 문자열로 받아서 하나씩 사용함. String의 메소드 중 charAt()을 사용하면 n번째 문자를 얻을 수 있음. ex. charAt(1)이면 2번째 문자를 가져옴.

메소드	설명
String next()	다음 토큰을 문자열로 리턴
byte nextByte()	다음 토큰을 byte 타입으로 리턴
short nextShort()	다음 토큰을 short 타입으로 리턴
int nextInt()	다음 토큰을 int 타입으로 리턴
long nextLong()	다음 토큰을 long 타입으로 리턴
float nextFloat()	다음 토큰을 float 타입으로 리턴
double nextDouble()	다음 토큰을 double 타입으로 리턴
boolean nextBoolean()	다음 토큰을 boolean 타입으로 리턴
String nextLine()	'\n'을 포함하는 한 라인을 읽고 '\n'을 버린 나머지 문자열 리턴
void close()	Scanner의 사용 종료
boolean hasNext()	현재 입력된 토큰이 있으면 true, 아니면 입력 때까지 무한정 대기, 새로운 입력이 들어올 때 true 리턴. crtl-z 키가 입력되면 입력 끝이므로 false 리턴

프로그램 전체에서 표준 입력에서의 입력이 완전히 끝난 후에 close() 메소드를 사용해야 함. 리눅스 등에서는 표준 입출력을 파일 입출력인 것과 동일하게 처리함. 프로그램이 시작되면 자동으로 System.in라는 파일과의 연결이 생성됨. System.in-하드웨어 연결이 생기는 것. 이후 scanner를 System.in을 연결하면 scanner-System.in-하드웨어로 연결이 됨. 이때 close()로 스캐너를 닫으면 scanner-System.in-하드웨어의 모든 연결이 끊겨서 해당 프로그램 내의 작업에서 System.in을 사용할 수 없게 됨. 그래서 close() 메소드는 모든 표준 입력이 끝난 뒤에 사용해야 함.

close()를 작성하지 않아도 프로그램이 돌아가긴 하는데, 작성해 주는 것이 좋음.

hasNext() 메소드는 입력이 아직 들어오지 않았으면 false를 반환하는 것이 아니라 block이 됨. 입력값이 들어오면 true를 리턴함. 이때 ctrl+z를 하면 EOF가 들어가는데, EOF를 넣는 키보드 조합은 환경 별로 다를 수 있음. 아래는 그 사용 예시임.

```
while(s.hasNext())
{
    String a = s.nex();
}
```

2.10. 배열

2.10.1. 배열

Definition 21 *인덱스와 그에 대응하는 데이터들로 이뤄진 자료 구조.*

배열 또한 reference형이므로, 아래와 같이 reference를 저장할 변수를 선언하고 객체를 생성하여 그 reference를 대입함.

```
// 참조(reference) 변수 선언
int[] intArr;                    // 주로 이 형태로 작성함.
int intArr [];
```

```
// 생성
intArr = new int[N];

// 선언&생성
int[] intArr = new int[N];
int[] intArr = {1, 2, 3, 4, 5};     // 선언 시에만 가능
```

배열을 생성하면 내부에 length라는 이름의 변수가 함께 생성됨. length에는 해당 배열의 길이가 저장됨.

배열 또한 객체인데, class가 없어도 생성이 가능한 객체임. c로부터 온 문법을 편리하게 사용하기 위한 것임. 배열은 객체이므로 힙에 저장되고, 자동으로 0으로 초기화됨.

이때 new는 객체 생성 연산자이고, new int[10] 등은 expression임.

배열은 primitive가 아니므로(int[] 등..) reference 타입의 자료형임. 배열을 생성하면 변수에 해당 배열에 대한 reference가 저장되는 것. 당연하게도, 다른 배열에 해당 reference를 대입하여 여러 배열이 동일한 메모리를 사용하게 할 수 있음.

c에서와 같이 인덱스는 0부터 시작함.

c에서처럼 {}로 초기화하며 배열을 생성할 수 있음. {} 안에는 리터럴과 변수 모두 올 수 있음. 물론 이 때에도 객체가 생성되는 것임. 이렇게 초기화하는 문법은 배열 선언 시에만 사용이 가능함. 선언 이후에 이렇게 대입을 시도하면 컴파일 에러 발생함.

2.10.2. for-each

Definition 22 *for-each(Enhanced for)문은 배열 등의 원소를 순차적으로 접근할 때 주로 사용되는 반복문임.*

for-each문에서는 뒤에 작성한 배열의 각 요소가 루프마다 앞에 작성한 변수로 대입됨.

아래와 같이 작성함.

```
int[] num = {1, 2, 3, 4, 5};
for(int k : num) // k에는 루프마다 1, 2, 3, 4, 5가 대입됨.
{
    ...
}
```

enhanced for는 데이터를 수정하지 않고 읽기만 하는 것이 기본임. 그래서 디버깅 시에 개발자는 암묵적으로 enhanced for에서 값이 변하지 않았다고 판단하는데, 그 믿음을 지켜줘야 함. 값을 수정할 것이라면 다른 반복문을 사용하는 것이 좋음.

2.10.3. 다차원 배열

Definition 23 *Java에서 다차원 배열은 다른 배열을 가리키는 참조 변수를 원소로 가지는 일차원 배열임.*

다차원 배열은 아래와 같이 배열은 선언과 생성을 함.

```
// 참조(reference) 변수 선언
int[][] inArr;          // 주로 이 형태로 작성함.
int intArr [][];

// 생성
intArr = new int[N][M];
intArr = new int[N][];  // 맨 앞만 지정해도 됨.
```

```
// 선언&생성
int[][] intArr = new int[N][M];
int[][] intArr = {{1, 2}, {3, 4}, {5, 6}};
```

다차원 배열은 참조 변수로 이뤄진 일차원 배열이므로, 맨 앞 차원만 지정하여 선언할 수 있음.

참고로, Java에서 배열은 주소가 아니라 이름으로 다룬다고 이해하는 것이 좋음. 각각의 요소가 어떤 이름을 가지고 있는지 파악할 수 있어야 함.

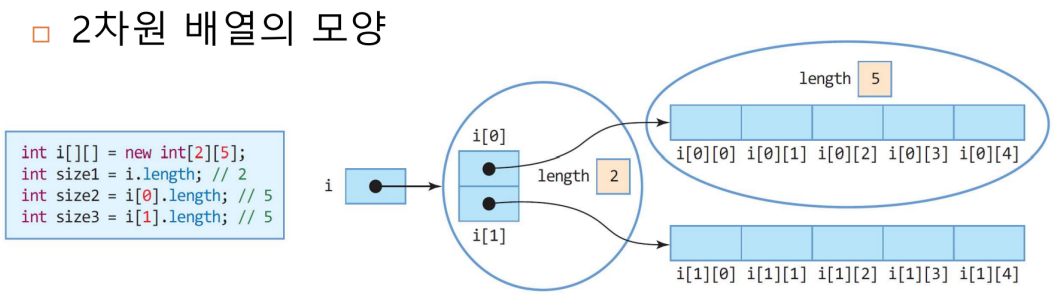
2.10.4. 배열의 메모리 구조

배열은 각 일차원 배열이 어떤 reference값 또는 데이터를 가지고 있는지로 그 메모리 구조를 파악할 수 있음. 다차원 배열은 reference값을 가지는 요소들과 해당 reference가 가리키는 실제 데이터를 저장하는 배열로 이루어짐.

이때 length는 해당 객체의 길이임.

Java 배열의 메모리 구조는 c에서의 n차원 데이터와 약간 다름.

각 요소에 대한 값과 자료형, 이름을 구분할 수 있어야 함.



2.10.5. 비정방형 배열

Definition 24 *Java에서는 다차원 배열을 reference값을 가지는 배열로 생각하기 때문에, 각 reference가 가리키는 배열의 길이를 다르게 지정할 수 있음. 이를 비정방형 배열(ragged array)이라고 함.*

아래는 그 예시임.

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

참고로 아래와 같이 작성하면 ragged든 ragged가 아니든 정상적으로 작동함.

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

2.10.6. String 객체

String 객체는 내부적으로 char[] 배열로 문자열을 저장함. 참고로 이때 c에서처럼 마지막 문제에 0을 넣지는 않음.

String 배열은 각 String 요소들이 또 String 객체를 가리키고, String 객체 내부에는 char[]형 참조 변수가 있어서, 해당 참조 변수가 또 char형 배열을 가리키는 메모리 구조를 가지고 있음.

2.11. main 메소드

2.11.1. main 메소드

하나의 class 내에 main 메소드는 하나만 존재해야 하고, 해당 class를 실행하면 main 메소드가 실행됨.

Java에서는 하나의 프로젝트 내에서 파일 별로 여러 개의 main 메소드가 존재할 수 있음. 그래서 실행 시에 어떤 class, 또는 어떤 main 메소드를 실행할 것인지 잘 명시해 줘야 함.

main 메소드는 항상 public static void로 선언하고, 매개변수의 자료형은 String[]임. 이때 변수 이름은 주로 args로 함.

전통적으로 각 class의 main 부분은 해당 class를 테스트하기 위한 공간임. 지금은 유용한 프레임워크들이 있어 main에 테스트 코드를 작성하지는 않음.

2.11.2. main의 매개변수

main은 기본적으로 공백을 기준으로 나눠서 입력받음. 단, 쌍따옴표로 묶어 작성하면 묶은 만큼 입력받음. (ex. "abc 3.5" 7 로 입력하면 abc 3.5과 7이 들어옴.)

c에서처럼 콘솔을 통해 main함수의 매개변수에 값을 넣을 수 있음. intellij에서는 실행 버튼을 우클릭하면 나오는 modify run configuration로 콘솔을 사용할 수 있음.

2.11.3. 문자열의 변환

입력받은 문자열을 정수나 실수로 변환할 수 있도록 하는 메소드들이 존재함.

Double.parseDouble() : String을 받아서 문자 그대로의 실수(double 형)로 반환함.
Integer.parseInt() : String을 받아서 문자 그대로의 정수(int 형)로 반환함.

2.12. 예외처리

2.12.1. 예외

Definition 25 *런타임 에러 중 예외처리로 처리가 가능한 것을 예외(Exception)라고 함.*

컴파일 시에 발생하는 컴파일 에러가 아닌, 문법에는 맞지만 논리적인 오류 때문에 프로그램 실행 중에 발생하는 에러를 런타임 에러라고 함.

모든 런타임 에러가 예외인 것은 아님. 런타임 에러 중 예외로 정의되어 있어 처리가 가능한 것들이 예외인 것. 그래서 예외가 아닌 런타임 에러(ex. 메모리를 전부 써 버린 경우.)에 대해서는 예외 처리가 불가능함.

예를 들어, 0으로 나누려고 하는 경우, 정의된 것보다 큰 인덱스로 메모리에 접근하려 하는 경우, 존재하지 않는 파일을 읽으려고 하는 경우, 정수 입력에 문자를 입력한 경우 등이 있음.

2.12.2. 예외처리

Definition 26 *예외를 처리하여 프로세스가 종료되지 않도록 하는 것을 예외처리(Exception Handling)라고 함.*

일반적으로 예외가 발생했을 때의 수행되는 과정은 아래와 같음. Java는 os가 아니라 JVM에서 돌아가기 때문에 이 과정을 JVM과 진행함.
추가로, 예외에 대해서는 반환이 아니라 던졌다(throw)고 함.
1. 프로세스에서 에러가 발생했음을 os에 알림.
2. os는 예외가 발생한 지점에 예외 객체를 던짐.
3. 예외 객체를 catch하는 구문이 없으면 해당 함수는 예외 객체를 자신을 호출한 함수에 던짐.
4. 결국 main에서 에외 객체를 os에 던지면, os는 비정상적인 상황으로 판단하여 프로세스를 종료하고 해당 예외 객체의 내용을 프롬프트로 출력함.

예외처리를 하면 예외 객체를 가져오기 때문에 JVM에 예외 객체가 반환되지 않아 프로세스가 종료되지 않음.

사용자가 JVM에 특정 상황을 예외로 지정할 수도 있다고 함. 즉, os에서 예외 객체를 받지 않아도 해당 상황에 대한 예외 처리가 되는 것.

시간적인 측면에서는 예외 처리가 비효율적이지만(물론 임베디드 정도가 아니라면 그렇게 심각한 정도는 아님), 다만 가독성과 생산성의 측면에서는 더 효율적이기 때문에 사용함.

2.12.3. 예외 객체

Definition 27 *Java API에는 에러 종류별로 class가 정의되어 있고, 에러가 발생하면 JVM은 객체를 생성하여 프로세스로 전달함. 이때의 객체를 예외 객체라고 함.*

에러 메세지에서 예외 객체의 class를 확인할 수 있음. 이 class를 사용하여 예외 처리를 함.

자주 발생하는 예외는 아래와 같음.

예외 타입(예외 클래스)	예외 발생 경우	패키지
ArithmeticException	정수를 0으로 나눌 때 발생	java.lang
NullPointerException	null 레퍼런스를 참조할 때 발생	java.lang
ClassCastException	변환할 수 없는 타입으로 객체를 변환할 때 발생	java.lang
OutOfMemoryError	메모리가 부족한 경우 발생	java.lang
ArrayIndexOutOfBoundsException	배열의 범위를 벗어난 접근 시 발생	java.lang
IllegalArgumentException	잘못된 인자 전달 시 발생	java.lang
IOException	입출력 동작 실패 또는 인터럽트 시 발생	java.io
NumberFormatException	문자열이 나타내는 숫자와 일치하지 않는 타입의 숫자로 변환 시 발생	java.lang
InputMismatchException	Scanner 클래스의 nextInt()를 호출하여 정수로 입력받고자 하였지만, 사용자가 'a' 등과 같이 문자를 입력한 경우	java.util

2.12.4. 예외처리 구문

Definition 28 *try, catch, finally로 예외처리 구문을 작성할 수 있음. 아래와 같이 작성함.*

```
try
{
    ;
}
catch( ... )          // 처리할 예외 타입 선언
{
    ;
}
finally               // 예외 발생 여부와 상관없이 항상 실행됨
{
    ;
}
```

try 내부에서 예외가 발생하면 해당 예외 객체의 reference가 적절한 catch의 매개변수로 전달됨.

catch는 매개변수의 자료형에 따라 여러 개 작성할 수 있는데, 이때 예외 객체 class의 종류에 따라

순서를 적절하게 맞춰줘야 함. 이는 상속에 대한 지식이 있어야 이해할 수 있음.

catch, finally는 생략이 가능함.

조건문으로 처리하는 것에 비해 try와 catch로 예외처리하는 것이 가독성과 생산성의 측면에서 이득임.

catch는 예외로 프로세스가 종료되지 않게 하기 위함이지, 일반적으로 대단한 작업을 하지는 않음. 또한, catch로 가져온 예외 객체는 예외의 종류를 구분하기 위한 것이지, 대체로 매개변수의 값을 실제로 사용하지는 않음. 참고로 catch 매개변수의 식별자는 주로 e를 사용함.

catch()는 매개변수의 사용과 제어가 넘어간다는 측면에서 함수 호출과 유사하게 작동함. 이때 catch()가 함수와 유사하지만 굳이 try 바로 뒤에 오는 것은 대체로 해당 블럭의 지역변수를 처리해야 하기 때문임.

finally에는 예외처리와 관련된 코드를 작성하여 가독성을 높임. 주로 파일입출력 후 스트림을 닫는 등의 코드를 작성함.

이때 finally는 말 그대로 무조건 실행됨. 예를 들어, try 안에 return이 있고, 실제로 제어가 return에 도달하여 실행되어도, 그 밑의 finally는 실행됨.

Part II

객체지향

1. 객체지향

1.1. 객체지향의 특성

1.1.1. 객체지향의 특성

객체지향에는 아래와 같이 4대 특성이 있음.

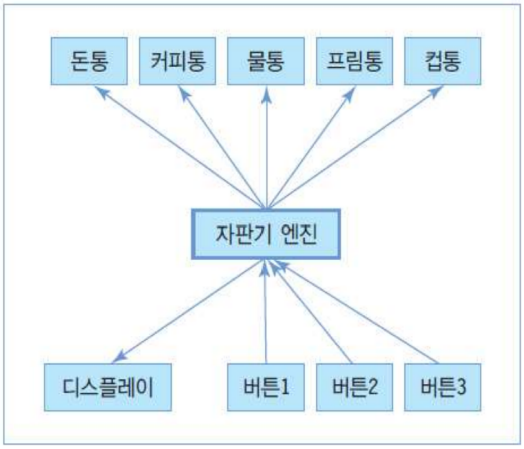
1. abstraction(추상화)
2. encapsulation(캡슐화)
3. inheritance(상속)
4. polymorphism(다형성)

객체지향은 결국 다형성을 쓰기 위한 것임. 추상화->캡슐화->상속->다형성의 순서로 개념이 존재함.

객체지향을 만든 사람들은 현실의 상황을 객체들과 그 사이의 관계로 설명하고, 객체지향언어는 그런 특성을 반영한 언어임. 객체들과 그 관계를 class로 표현함.

1.1.2. 객체지향의 목적

생산성을 높이고, 실세계에 모델링을 쉽게 하는 것이 객체지향의 목적임.



2. 추상화와 캡슐화

2.1. 추상화와 캡슐화

2.1.1. 추상화와 캡슐화

Definition 29 *복잡한 상황에서 필요한 부분에만 집중해서 문제를 단순화하는 것을 추상화(abstraction)라고 함.*

객체를 캡슐로 싸서 내부를 볼 수 없도록 하는 것(information hiding)을 캡슐화(encapsulation)라고 함.

class로 추상화와 캡슐화를 구현할 수 있음.

추상화는 변수와 함수를 적절히 정의하는 것을 의미하는데, 당연히도 객체지향만의 특성은 아님.

내부를 볼 수 없도록 하는 것은, 유출이나 잘못된 조작을 예방하기 위함도 있지만 사용자가 굳이 볼 필요 없는 내용을 가져서 생산성을 높이는 것임.

2.2. class와 객체

2.2.1. class와 객체

Definition 30 *객체 모양을 선언하는 틀을 class, class에 따라 생성된 실체(instance)를 객체(object)라고 함. 객체는 인스턴스라고도 부름.*

class는 필드(멤버 변수)와 메소드(멤버 함수)로 구성되고, 아래와 같이 정의함.

```
public class Animal
{
    public String name;     // 필드
    public int age;

    public int getAge()         // 메소드
    {
        ...
    }
}
```

객체는 항상 아래와 같이 new 키워드를 사용하여 생성함. 이때 new 키워드는 항상 객체의 생성자를 호출하기 때문에 반드시 ()를 붙여줘야 함. 생성자를 작성하지 않았더라도 기본생성자가 삽입되어 정상적으로 동작함.

new 키워드가 사용된 수식(expression)은 해당 객체의 reference를 반환함. 이렇게 생성한 객체의 멤버에는 .으로 접근함.

```
Circle pizza;
pizza = new Circle();
pizza.getArea();
```

class는 자료형이고, Java를 이용한 코딩은 class라는 자료형을 정의하는 것임. 개념적으로 구조체와 class에 다른 점은, class는 함수까지 포함한다는 것임.

실세계의 객체는 고유한 특성(state)과 행동(behavior)을 가짐. class에서는 특성을 필드로, 행동을 메소드로 정의함.

class는 reference형이므로, 객체 생성 시에 필드는 힙에 저장되고 비트는 전부 0으로 초기화되어 있음.

2.2.2. 생성자

Definition 31 *객체 생성 시에 호출되는 메소드를 생성자(constructor)라고 함.*

생성자는 아래와 같이 return type을 작성하지 않고 식별자를 class의 식별자와 같게 함.

```
public class Animal
{
    public String name;     // 필드
    public int age;

    public Animal()         // 생성자
    {
        ...
    }
}
```

메소드 오버로딩(Overloading)으로 하나의 class에 여러 개의 생성자를 작성할 수 있음. 이때 매개변수에 따라 생성자가 호출됨.

정확히는 객체 생성 직후에 호출되는 것.

생성자에는 주로 필드의 초기값을 지정하는 코드를 넣음.

2.2.3. 기본생성자

Definition 32 *생성자를 작성하지 않은 경우 컴파일러에 의해 자동 삽입되는 생성자를 기본생성자(default constructor) 또는 디폴트 생성자라고 함.*

아래와 같은 생성자 코드가 삽입되는 것처럼 처리되는데, 매개변수와 내부 작업이 존재하지 않음.

```
public CIrcle(){}
```

이때 접근자정자는 class의 것과 동일하게 지정됨.

기본생성자는 생성자가 아예 없을 때에만 적용됨. 생성자가 하나라도 있을 경우 처리되지 않음.

생성자는 객체 생성 시에 new 키워드에 의해 반드시 호출되고, 그렇기에 생성자가 class에 하나 이상 선언되어 있어야 함. 그래서 컴파일러가 기본생성자를 삽입하는 것.

실제로 코드가 삽입되는 것은 아니지만, 삽입된 것처럼 처리됨.

2.2.4. this

Definition 33 *this 키워드는 두 가지 사용법이 있음.*

1. this reference
this는 해당 객체 자신에 대한 reference임.

필드와 메소드 변수의 식별자가 겹치거나, 해당 객체의 메소드를 사용해야 하는 경우에 아래와 같은 형태로 사용함.

```
this.radius = radius;
```

this. 없이 필드를 사용할 때에도 this.이 생략된 것으로 생각할 수 있음.

메소드 내부에 필드와 동일한 식별자의 지역변수가 존재하면, 해당 메소드 내에서 해당 식별자는 전부 지역변수로 취급됨. 이런 경우 this를 적절히 사용해야 함.

2. this()로 생성자 호출
생성자 내부에서 this()를 사용하여 해당 class 내의 다른 생성자를 호출할 수 있음. 생성자를 호출한 것이므로 해당 생성자의 작업이 끝나면 제어가 다시 호출한 곳으로 돌아옴.

이때 this()는 생성자 코드의 제일 위에 작성해야 함.

인자로 작성한 데이터의 자료형에 따른 생성자가 호출됨.

this를 사용하지 않고 필드를 사용할 때에도 this.가 생략된 것으로 생각할 수 있음. 실제 객체 생성 시에 힙에는 필드만 생성되고, 메소드는 코드 세그먼트에 존재함. 객체에서 메소드 호출 시에 제어는 class 내부의 메소드 코드로 이동하고, 메소드 내부에서 필드를 사용할 때는 메소드를 호출한 객체의 메모리(힙)로 접근함. 이때 메소드 내부에서는 임의의 객체에 대해 처리가 가능해야 하기 때문에 필드는 this.식별자로 작성되어 있고, this 위치에 호출한 객체의 이름이 들어가는 것으로 이해할 수 있음.

this reference를 사용하면 필드 뿐만 아니라 해당 객체의 메소드에도 접근이 가능함.

this()는 주로 인자를 더 적게 받는 경우 등에 대한 처리를 할 때 반복되는 코드를 한 번만 작성하기 위해 사용함.

this()를 생성자 코드의 제일 위에 작성해야 하는 것은 kotlin과의 호환성 등 때문임.

2.2.5. 가비지 컬렉션

Definition 34 *할당된 객체 메모리를 반납하는 것을 객체 소멸이라고 하는데, Java에서는 객체 소멸을 코드에서 직접 조작할 수 없음. 이는 런타임에 JVM이 자체적으로 수행함.*

가리키는 reference가 더 이상 존재하지 않는(더 이상 사용되지 않는) 객체를 가비지(garbage)라고 하고, 가비지를 처리하는 JVM의 연산을 가비지 컬렉션이라고 함. 가비지 컬렉션을 수행하는 JVM의 스레드를 가비지 컬렉터라고 함.

아래와 같이 System또는 Runtime 객체의 gc() 메소드로 가비지 컬렉션을 JVM에 요청할 수 있음.

```
System.gc();
```

객체를 생성하면 해당 객체를 참조하는 변수의 개수가 count되고, count가 0이 되면 해당 객체가 가비지로 취급되어 가비지 컬렉션이 수행됨.

JVM은 하나의 app만을 실행할 수 있고 JVM의 주 목적은 프로그램의 실행이기 때문에, 내부의 알고리즘에 따라 가비지 컬렉션을 함. 가비지가 생기는 즉시 반납하는 것이 아니라, 적절한 타이밍에 반납하기 때문에 힙 공간이 부족한 상황이 나올 수도 있음. JVM에 가비지 컬렉션을 요청하는 메소드도 있지만, 말 그대로 요청이지 강제는 아님. 이 메소드를 사용해도 JVM이 나름대로 판단해서 처리함. 이럴 때는 가비지 컬렉션을 요청하는 대신 JVM의 힙 크기를 직접 지정하는 것으로 해결할 수 있음.

2.2.6. 객체 배열

Definition 35 *비정방형 배열을 만드는 것처럼, 아래와 같이 객체 배열을 만들어 사용할 수 있음.*

```
Circle[] c;
c = new Circle[5];

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

2.3. 패키지

2.3.1. 패키지

Definition 36 *관련 있는 class 파일들을 모아 놓은 디렉토리를 패키지라고 함.*

Java 프로그램은 하나 이상의 패키지로 구성되어 있음.

서로 다른 패키지라면 class명이 겹치는 것이 가능함.

Java API는 JDK에 패키지 형태로 제공됨. 또한 소스 코드 작성자가 패키지를 만들 수도 있음. 이렇게 존재하는 패키지는 필요할 때 import 하여 사용함.

라이브러리는 이런 패키지들(폴더들)과 인터페이스를 .jar 파일로 압축해 놓은 것을 말함. 라이브러리를 다운받아 프로젝트에 포함시키고(classpath에 포함시킴.), 그 내부의 패키지를 import해 사용하는 것.

패키지 이름은 주로 소문자로 작성함.

Java에서 제공하는 패키지는 주로 java 패키지 안에 있음. (ex. java.util.Scanner)

2.3.2. 모듈

Definition 37 *Java 9에서 추가된 모듈(module)은 여러 패키지와 자원(이미지 등)을 모아 놓은 컨테이너임. 패키지, 이미지, XML 파일 등을 묶은 것.*

Java 프로그램은 하나의 프로젝트로 개발이 가능하지만, 여러 개의 모듈로 쪼개 개발하는 모듈 프로그래밍(module programming)도 가능함.

Java 9부터 Java API의 모든 class들은 패키지 기반에서 모듈 기반으로 재구성됨. 모듈 파일들은 .jmod 확장자를 가지고, jmods 디렉토리에서 확인할 수 있음.

현재 모든 함수에 대한 Java API는 그 크기가 너무 커지므로 일부 함수만 포함함. 기본적인 것들만 배포하고 나머지는 사용자가 패키징하도록 함. 자세한 설명은 해 주지 않았음. 더 찾아보자.

JDK의 모듈은 jmods 디렉토리에 있음.

2.3.3. 패키지와 class

Definition 38 *패키지 내부에 작성하는 .java 파일은, 아래와 같이 코드에 package로 해당 패키지를 지정해 줘야 함. 이때 패키지가 다른 패키지 내부에 있을 경우 .으로 구분해서 작성함.*

src파일은 루트 패키지로, src에 대해서는 package 작성하지 않음.

```
package book;
package book.math;        // book 패키지 내부의 math 패키지
```

같은 패키지의 class는 import 없이 접근이 가능하지만(물론 private이면 불가,), 다른 패키지의 class는 import로 가져와야 사용이 가능함. 이때 '같은 패키지'는 동일한 패키지를 말함. 해당 패키지 내부 패키지에 있는 class는 다른 패키지에 있는 것으로 취급됨.

*패키지 내부의 특정 class를 가져올 때는, 아래와 같이 import 패키지명.class명 꼴로 작성함. 이때 *로 해당 패키지의 모든 class를 가져올 수 있음.*

```
import book.math.Algebra;      // book.math 패키지 내부의 Algebra class를 가져옴
import book.math.*;      // book.math 패키지 내부의 모든 class를 가져옴
```

아래와 같이 import로 가져오는 대신 해당 경로를 전부 작성하는 방법도 가능함. 다른 패키지의 class명이 기존의 class명과 겹치는 경우 이런 식으로 사용해야 할 수 있음.

```
java.util.Scanner a = new java.util.Scanner(System.in);
```

예를 들어, import java.util.Scanner;는 java 패키지 내부의 util 패키지 내부의 Scanner class를 가져오는 것임.

2.4. 접근지정자

2.4.1. 접근지정자

Definition 39 *멤버나 class에 접근 권한을 부여하여 다른 class에 이를 공개하거나 숨기도록(캡슐화) 지정하는 것을 접근지정자라고 함.*

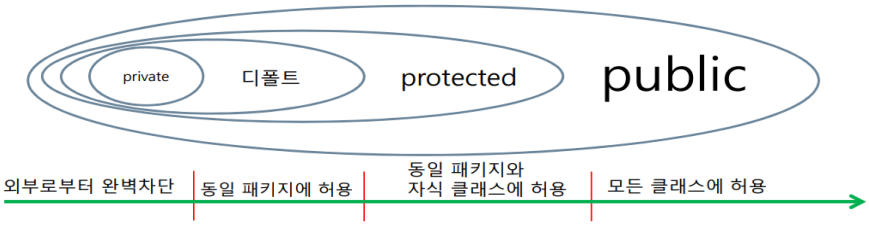
Java의 접근지정자는 private, protected, public, default(생략)가 있음.
private : 해당 class 내부에서만 접근 가능.
default : 동일한 패키지의 class에서만 접근 가능.
protected : 동일한 패키지의 class와 자식 class에서만 접근 가능.
public : 모든 class에서 접근 가능.

접근지정자는 class와 그 멤버(필드와 메소드)에 사용함. 멤버에 대해서는 4가지 모두 사용이 가능하지만, class에 대해서는 public과 default만 사용이 가능함. 물론 class 내부에 작성된 class(nested class)는 멤버이기 때문에 4가지 모두 가능함.

접근지정자는 멤버와 class에만 사용할 수 있으므로, 지역변수에는 사용이 불가능함.

class와 멤버의 접근지정자는 서로 영향 없이 지정이 가능하지만, 당연하게도 실제로 사용 시에는 class가 멤버보다 범위가 좁다면 의미가 없음.

public을 마구 남발하는 것은 마치 c에서 전역변수를 남발하는 것과 같기 때문에, 어떤 기준으로 접근지정자를 지정할 것인지를 명확히 해야 함.



2.4.2. class의 접근지정자 지정

class의 접근지정자로는 public과 default만 사용이 가능하기 때문에, 패키지 단위로 생각해야 함. **외부에서 패키지를 사용할 때 접근하는 class는 public, 외부와의 교류 없이 내부적으로만 작동하는 class는 default로 함.**

2.4.3. 필드의 접근지정자 지정

수행하는 작업에 대한 이해를 바탕으로 어느 범위까지 값을 공개할 것인지에 따라 접근지정자를 지정해야 하지만, **기본적으로 필드는 private으로 지정함.** 더 큰 범위로는 잘 지정하지 않음.

Java 코드에서는 .로 필드를 직접 제공하는 방식은 거의 사용하지 않음. 대신 메소드를 사용하는데, 가독성을 위해 해당 class 내부적으로 사용할 때도 마찬가지임.

외부에 필드의 값을 제공해야 하는 경우 .으로 필드 자체를 노출하면 해당 값을 수정할 수 있게 됨. 이를 방지하기 위해 값을 제공하거나 갱신하는 작업은 메소드로 따로 구현함. 이런 메소드를 관습적으로 getter/setter라고 부르는데, getter는 필드를 읽어주는 메소드이고 setter는 필드 값을 갱신하는 메소드임.

alt+insert로 getter, setter, constructor 등의 자동생성이 가능함. 이때 생성되는 gettser/setter의 형태가 관습적인 표준임.

2.4.4. 메소드의 접근지정자 지정

메소드는 기본적으로 private으로 지정함. 이후 필요한 경우 그 범위를 늘림.

협업 시에는 접근지정자 범위를 최소로 해서 각각 작업하고, 그 범위를 늘려야 할 때는 토의를 하는 식으로 진행하기도 함.

2.5. static

2.5.1. static

Definition 40 *static 키워드를 사용하여 멤버(필드, 메소드)의 소유자를 객체가 아닌 class로 할 수 있음. class 소유의 필드와 메소드를 static 필드와 static 메소드라고 함.*

static은 멤버에만 붙일 수 있음. static 멤버의 소유자는 class(type)이고, non-static 멤버의 소유자는 각각의 객체임.

static 필드는 static 세그먼트에, static 메소드의 코드는 코드 세그먼트에 저장됨. 이때 메모리에 static 필드는 class당 하나만 존재함.

static 멤버는 class 소유이므로 아래와 같이 객체 생성 없이 사용이 가능함.

```
public class TestClass
{
    static int baseNum;             // static 필드
    static int getN() { ... }       // static 메소드
}
...
TestClass.baseNum;
TestClass.getN();
```

static 필드는 0으로 자동 초기화됨. (힙에 저장되는 것으로 보임.)

static은 멤버에만 사용할 수 있으므로, 지역변수에는 사용이 불가능함.

main 메소드도 static으로 정의함. 애초에 static으로 지정하지 않으면 객체를 생성하지 않고 해당 메소드를 호출할 수 없으므로 main 메소드를 사용하는 의미가 없음. 또한, 당연하게도 main에서 사용하는 멤버들은 static으로 정의되어 있어야 함.

우리나라에는 static을 객체들의 공유 공간으로 설명하는 교과서가 많은데, 이런 식의 이해는 잘못되었음. 애초에 공유하는 것이라기보다는 class의 공간이고, 또한 객체로 static에 접근하는 방식으로 사용하는 것은 좋지 않음.

참고로, 실제로는 JVM의 세그먼트 관리 방법이 나름대로 존재하지만, 본 수업에서는 이해하기 쉽게 c에서의 그것과 유사하게 설명함. 실제로는 더 세분화되어 있음.

2.5.2. static 필드

static 필드가 존재하면 class별로 필드에 대한 메모리 공간이 생성됨. 즉, 객체를 생성하지 않아도 class의 자체적인 메모리 공간이 존재하기 때문에, 객체 생성 없이 필드를 사용할 수 있음. 이때 static 필드는 힙이 아니라, static 세그먼트에 저장됨.

class에는 class의 코드와 static 필드의 정의에 의해 선언된 메모리가 포함되는데, 이 수업에서는 각각을 class 코드, class 객체라고 함. class는 이 둘을 모두 포함함.

static 필드는 주로 public static final double PI = 3.14; 이런 식으로 사용함.

System.in, System.out는 reference(class)형을 가지는 static 필드이기 때문에 객체 생성 없이 사용이 가능했음. System.out.println()에서 확인할 수 있듯이 static 필드가 reference형을 가지고 객체의 reference를 저장하고 있으면 해당 객체의 메소드를 사용할 수 있음.

static 필드는 class 자체의 속성을 나타낼 때 사용함. class는 개념어에 대응되는 문법으로, 사람, 차, 가구 등을 표현할 수 있음. 이때 사람의 평균수명 등 사람, 차, 가구에 대한 자체적인 속성이 존재할 수 있음. 이것을 static 필드로 나타내는 것.

2.5.3. static 메소드

static 메소드는 코드이므로 class 객체에 포함되지 않음. 해당 코드는 코드 세그먼트에 저장됨.

static 메소드는 유틸리티성 메소드라고도 하는데, 프로젝트 등에서는 유틸리티성 메소드만을 모아둔 class를 만들기도 함.

Math.random() 메소드도 static 메소드이기 때문에 객체 생성 없이 사용이 가능했음. 참고로, Math.random()는 0 이상 1 미만의 임의의 실수를 반환하는 메소드임. 해당 수에 10의 거듭제곱을 곱하고 형변환하여 임의의 정수를 얻을 수도 있음.

'사람' class의 '걷기' 메소드가 있다고 하자. 걷는다는 것은 사람으로부터 나온 객체가 아닌, 사람이라는 개념의 자체적인 속성임. 이런 경우 소유자를 class로 함.

2.5.4. static 필드의 메모리 할당 시기

class의 아무 멤버나 처음 사용할 때, 해당 명령 그 직전에 static 세그먼트에 class 객체가 생성됨. 즉, static 멤버는 아무때나 사용이 가능함.

class 최초 사용의 경우의 수는 1. new로 객체를 생성하는 경우 2. static 멤버를 사용하는 경우로 2가지임. 1번의 경우 객체 생성 직전에 static 세그먼트에 class 객체를 만들고, 2번의 경우 static 멤버를 사용하기 직전에 static 세그먼트에 class 객체를 만듦.

2.5.5. static 사용의 유의점

객체에서도 .으로 static 멤버에 접근이 가능함. 하지만 이 경우 가독성이 매우 떨어지고 논리적인 오류가 발생하기도 쉬움.

예를 들어, TestClass.pi로 쓰면 딱 봐도 pi가 static이지만, TestClass로 생성한 testVal.pi로 쓰면 static인지 필드인지 확인해봐야 함. static 필드인 경우 값을 수정한 경우 해당 class에 대한 메모리가 수정되기 때문에, 이를 필드처럼 사용하면 문제가 생길 수 있음.

그래서 비교적 최근에 나오는 언어들에서는 애초에 객체로부터의 접근을 막아 놓음.

사용에는 문제가 많지만.. 시험 문제로 내기에는 좋다고 하심.

2.5.6. static 메소드와 this의 사용

static 메소드는 객체가 생성되기 이전에 호출될 수 있으므로 this를 사용할 수 없음. non-static 필드/메소드 또한 this.가 생략된 것이므로 사용이 불가능함. 사용하려고 하면 컴파일 오류가 발생함.

물론 내부적으로 객체를 생성해서 해당 객체의 non-static을 사용할 수 있는 것은 당연함. this가 불가능한 것이지 non-static의 사용 자체가 불가능한 것이 아님. 불가능한 것은 method()와 같이 객체.을 사용하지 않고 호출하는(this.로 호출하는) 것.

static 메소드인 main 메소드를 생각하면 자연스럽게 사용할 수 있음.

2.6. final

2.6.1. final

Definition 41 *final 키워드는 변수(필드, 지역변수), 메소드, class에 사용하여 정적인 속성을 부여함.*

final로 constant를 정의하는 것은 앞에서 이미 다뤘음.

final을 class에 사용하면 해당 class는 상속이 불가능해짐.

final을 메소드에 사용하면 해당 메소드는 오버라이딩이 불가능해짐.

3. 상속

3.1. 상속

3.1.1. 상속

Definition 42 *상위 객체의 속성(필드, 메소드)이 하위 객체에 물려지는 것을 상속(inheritance)이라고 함. 이때 상위 class를 슈퍼 클래스(super class/type), 하위 클래스를 서브 클래스(subclass/subtype)라고 함.*

상속 구조에서 위쪽으로 갈수록 일반화(generalization)된다고 하고, 아래로 갈수록 구체화(specialization)된다고 함.

A가 B를 상속받았으면 화살표로 A->B와 같이 나타냄.

상속은 확장(extension)으로 이해할 수 있음.

상속 구조는 다형성을 사용하기 위한 것임.

UML(unified modeling language)이라는 프로그램 모델링을 위한 언어가 존재하는데, 여기에서도 상속에 대해 상속하는 쪽을 가리키는 화살표로 나타냄.

3.1.2. 상속의 이점

겹치는 코드를 간결하게 작성할 수 있고, class들을 계층적으로 분류할 수 있음. 결과적으로는 상속 구조의 데이터 처리를 통해 다형성을 활용할 수 있음.

3.2. Java에서의 상속

3.2.1. Java에서의 상속

Definition 43 *Java에서는 아래와 같이 extends 키워드를 사용해 상속을 선언함. extends로 작성한 class를 상속받는다는 의미임.*

```
public class Student extends Person
{
    ...
}
```

super class를 상속받은 subclass의 객체를 생성하면, 해당 super class의 모든 필드와 메소드가 subclass의 객체에 추가됨. 이때 객체에는 상속받은 class의 필드와 메소드 전부가 존재하지만, 적절한 접근지정자가 지정되어 있는 멤버에만 접근이 가능함.

객체 생성 시에 모든 멤버가 추가되지만 접근지정자가 적절히 붙어 있어야 함. 해당 멤버를 사용하려면 같은 패키지일 경우 public, protected, default여야 하고, 다른 패키지일 경우 public, protected여야 함. 당연하게도 직접적인 접근이 불가능한 것이지 다른 접근 가능한 메소드 등에서는 접근이 가능함.

상속 구조 등의 구현에 도움을 주는 것 중에 리펙토링(refactoring)이라는 것이 있음. 리펙토링은 프로그램의 기능을 유지한 채 그 구조만 수정하는 것으로, intelliJ같은 IDE에서 제공함. 프로그램을 충분히 설계한 경우에도 실제 구현 시에는 그 구조를 수정해야 할 수 있는데(코드가 겹치는 등..), 이때 리펙토링을 사용함.

3.2.2. Java 상속의 특징

Java의 상속은 아래의 특징들을 가짐.

1. 다중 상속을 지원하지 않음.
하나의 class는 하나의 class만 상속받을 수 있음. 한 번에 두 가지 class를 상속받은 경우 서로 같은 식별자의 메소드가 있는 경우 등의 문제를 피한 것.

2. 상속 횟수에는 제한이 없음.

3. 모든 class의 최상위 class로 java.lang.Object class가 존재함.
모든 class는 자바 컴파일러에 의해 자동으로 Object class를 상속받음. 이를 root class라고도 하고, 상속하지 않아도 자동으로 부모로 지정되는 것을 direct 부모라고도 함.

3.2.3. 멤버의 선택

멤버(필드, 메소드)를 사용하면 객체의 subclass 부분부터 super class 부분으로 이동하며 해당 식별자의 멤버를 찾음. 즉, 동일한 식별자의 멤버가 존재한다면 더 아래쪽에 있는 것이 사용됨.

3.2.4. super class의 생성자 호출

Definition 44 *super 키워드로 super class 부분에 대한 생성자를 호출함. 아래와 같이 작성하고, 객체 생성 시 생성자가 선택되는 것처럼 인자로 작성한 값에 따라 super class의 생성자가 선택됨.*

```
super(10, "12");
```

기본적으로 객체가 생성되면 최하위 class의 생성자가 가장 먼저 호출되고, 각 생성자에서 super()로 상위 class 부분의 생성자를 지정하지 않았다면 첫 문장에 super()가 삽입됨.

단, this()가 존재한다면 super()를 삽입하지 않음. 즉, 상속 구조의 생성자 첫 문장에서는 this() 또는 super()가 수행됨. this()와 super() 모두 첫 문장으로 와야 하므로 둘 다 사용할 수는 없음. 둘 다 작성하면 두 번째 줄에 오는 this()/super()에서 컴파일 에러가 남.

this()와 같이 super()는 생성자의 가장 첫번째 줄에 작성해야 함. 계층 구조로 생각했을 때 this()는 생성자의 좌우 이동으로, super()는 생성자의 상하 이동으로 이해할 수 있음.

즉, A->B->C(A가 최하위)의 상속 구조가 존재한다면, A,B,C의 순서로 호출되고 C,B,A의 순서로 각 생성자의 작업이 수행됨.

4. 다형성

4.1. 다형성

4.1.1. 다형성

Definition 45 *같은 이름의 메소드가 class나 객체에 따라 다르게 작동하도록 구현한 것을 다형성(polymorphism)이라고 함. 'poly'는 '다양한', 'morphism'은 '매핑'이라는 뜻으로, polymorphsim은 1:n 매핑을 말함.*

메소드 오버로딩(Overloading), 메소드 오버라이딩(Overriding)이 다형성을 구현한 문법임.

같은 식별자와 다른 매개변수 목록을 가진 메소드를 정의하는 것을 메소드 오버로딩(Overloading)이라고 함. 특히 생성자 등에서 사용됨.

다형성을 쓰기 위해 객체지향을 사용하는 것임. 다형성을 최대한 많이 사용하려고 하면 생산성이 올라가고, 기존 코드의 변경을 최소화할 수 있음.

4.2. 메소드 오버로딩

4.2.1. 메소드 오버로딩

Definition 46 *식별자가 같고 매개변수 목록이 다른 메소드들을 여러 개 작성하는 것을 메소드 오버로딩(Overloading)이라고 함.*

Java에서는 식별자와 매개변수 목록을 통틀어 메소드의 signature라고 하는데, signature가 동일하다면 두 메소드는 동일한 메소드로 취급됨. 식별자가 같아도 매개변수 목록이 다르면 signature가 다른 것이므로 다른 메소드로 취급됨.

인자로 작성한 값에 따라 메소드가 선택되어 호출됨.

반환 자료형은 signature가 아니므로 메소드 오버로딩과 관련이 없음.

주로 다양한 인자에 대한 처리를 위해 메소드 오버로딩을 함.

메소드 오버로딩은 하나의 class에 대해서도 사용할 수 있고, 상속 구조에서도 사용이 가능함. signature가 다르면 단순히 다른 메소드로 취급된다고 생각하면 됨.

4.3. reference의 형변환

4.3.1. 업캐스팅

Definition 47 *상속 관계에 있는 class들에 대해서, 하위 class의 객체가 상위 class로 자동형변환되는 것을 업캐스팅(upcasting)이라고 함. 업캐스팅은 reference에서의 자동형변환임.*

즉, 아래와 같이 ***객체지향에서는 상위 class 참조 변수로 하위 class 객체를 (캐스팅 없이) 가리킬 수 있음****. 다형성을 잘 구현하기 위해서는 가능한 한 상위 class 참조 변수로 객체를 가리키는 것이 좋음.*

```
Person a = new Student();
```

상위 class 참조 변수로 객체를 가리키면 그 class로서 객체를 다룬다는 것이므로, 해당 class의 멤버에만 접근이 가능함. 컴파일러는 해당 참조 변수의 자료형을 기준으로 문법의 오류를 검사하기 때문에, subclass 멤버에 접근하려고 하면 컴파일 에러가 발생함.

이때 형변환은 이루어지지만 객체(값)가 변하는 것은 아님.

4.3.2. 다운캐스팅

Definition 48 *상속 관계에 있는 class들에 대해서, 캐스트 연산자를 사용하여 상위 class의 객체를 하위 class로 강제형변환하는 것을 다운캐스팅(downcasting)이라고 함.*

즉, 캐스팅 연산을 사용하여 아래와 같이 하위 class 참조 변수로 상위 class 객체를 가리킬 수 있음. 이때 아예 상하관계에 있지 않은 타입으로 강제형변환하려고 하면 컴파일 에러가 발생함.

```
Person b = new Person();
Student a = (Student)b;
```

강제형변환 후, 객체에 존재하지 않는 멤버에 접근하려고 하면 런타임 에러가 발생함.

자료형에 대해 호환성에 문제가 있을 가능성이 존재하거나 실제로 문제가 있다면 컴파일러는 그것을 에러로서 알리는데, 강제형변환(캐스팅 연산)은 컴파일러의 경고를 무시한다는 의미임. 캐스팅 연산을 하지 않은 경우 컴파일러 입장에서는 해당 참조 변수가 가리키는 객체(값)가 어떤 것일지 모르기 때문에 컴파일 에러를 내는 것임. 강제형변환으로 이 경고를 무시할 수 있지만, 추후에 잘못된 접근을 하면 런타임 에러가 날 수 있음.

예를 들어, 아래와 같이 함수가 작성된 경우 p에는 Person 객체가 올 수도, Student 객체가 올 수도 있기에 컴파일러가 에러를 내는 것.

```
void doTest(Person p) // Person/Student 객체 모두 올 수 있음.
{
    Student s = p; // 컴파일 에러
}
```

4.3.3. instanceof 연산자

Definition 49 *instanceof 연산자로 해당 객체를 어떤 참조 변수로 가리킬 수 있는지를 알 수 있음. a instanceof b는 a는 b의 예시라는 의미로, a에 b class에 해당하는 부분이 있음을 판단함.*

instanceof는 reference에만 사용 가능한 연산자로, 아래와 같이 작성함. 그 결과는 boolean형으로 반환됨.

```
s instanceof Person // true/false
10 instanceof int // 컴파일 에러. reference에만 사용 가능
"test" instanceof String // true
```

이때 캐스팅이 불가능한 타입(상하관계가 아닌 경우)에 대해 instanceof를 연산하려고 하면 컴파일 에러가 발생함. 컴파일러의 입장에서는 객체로 작성한 값의 타입을 기준으로 상속 구조에서 위아래로 움직이며 판단하는 것인데, 아예 해당 상속 구조에 존재하지 않으면 컴파일 에러를 내는 것.

주로 상위 class 참조 변수로 하위 class 객체를 가리키고 있을 때, 가리키고 있는 실제 객체의 타입에 대해 판정하려고 사용함. 하지만 instanceof의 사용은 다형성의 사용을 비효율적으로 대체하므로 지양해야 함.

최근 버전의 JDK에서는 아래와 같이 대입까지 한꺼번에 수행하는 instanceof 문법이 추가되었음. 이때 평가되는 값과 형은 기존의 instanceof와 동일함.

```
// 기존 코드
if(p instanceof Student)
{
    s = (Student)p;
}
// 새로 추가된 형태
if(p instanceof Student s)
{
    ...
}
```

4.3.4. 업캐스팅과 protected

객체가 protected 멤버를 상속받았더라도, 해당 객체에 하위 class로서 접근해야 protected 멤버에 접근할 수 있음.

protected 멤버를 상속받는 class에서 직접적으로 멤버를 사용하는 것은 당연히 가능함.

헷갈릴 수 있는 부분은 해당 class 내부에서 객체를 생성하여 접근하는 경우임. 접근지정자에 대해서는 해당 멤버의 사용 코드 위치로 사용 가능 여부를 판단할 수 있어야 함. 다른 패키지에서 접근하는 경우, protected 멤버를 상속받는 class 내부에서는 해당 class와 해당 class를 상속받는 class의 객체로 protected 멤버에 접근이 가능함.

4.4. 메소드 오버라이딩

4.4.1. 메소드 오버라이딩

Definition 50 *super class의 메소드를 subclass에서 재정의하는 것을 메소드 오버라이딩(overriding)*

이라고 함.

subclass에서 super class 메소드와 같은 signature, 반환 자료형을 가진 메소드를 작성하면 동적 바인딩에 의해 메소드 오버라이딩이 됨. 즉, subclass 객체에서는 새롭게 쓰인 메소드가 반영이 됨.

이때 메소드의 signature를 같게 하고 반환 자료형을 다르게 하면 컴파일 에러가 남. signature를 다르게 하면 메소드 오버로딩이 되어 아예 다른 메소드로 취급됨.

메소드 오버라이딩을 통해 각 subclass별로 해당 signature의 메소드가 적절한 작업을 수행할 수 있게 함. 즉, 하나의 인터페이스가 내부적으로 여러 가지 형태를 가지는 다형성을 구현할 수 있음.

메소드 오버라이딩 시에 메소드의 접근지정자는 다르게 지정할 수 있음. 단, 부모 메소드의 접근지정자보다 넓은 범위로는 지정할 수 있지만 더 좁은 범위로는 지정할 수 없음. 더 좁은 범위로 지정하면 컴파일 에러가 발생함.

메소드 오버라이딩이 정의된 상속 구조를 만들고, 상위 타입 참조 변수로 하위 타입 객체를 가리키도록 함수를 작성하여 다형성을 구현함.

예를 들어, Shape이라는 super class에 Line, Rect, Circle이라는 subclass가 있다고 하자. 각각에 대해 draw() 메소드를 오버라이딩하고, 사용 시에는 아래와 같이 작성하여 코드를 비약적으로 간단히 할 수 있음. 이런 식으로 사용하기 때문에 객체는 가능한 한 상위 class 참조 변수로 가리키는 것이 좋다고 한 것임.

```
public Class UsingExample
{
    static void paint(Shape s)
    {
        s.draw();
    }
}
```

메소드 오버라이딩은 메소드에 대한 것으로, 필드에 대해서는 당연히 성립하지 않음.

4.4.2. 동적 바인딩

Definition 51 *메소드 호출문과 실제 호출되는 메소드를 묶어 주는 것을 바인딩(binding)이라고 하고, 런타임에 수행되는 바인딩을 동적 바인딩(dynamic binding)이라고 함. 반대로 컴파일 시에 수행되는 바인딩은 정적 바인딩(static binding)이라고 함.*

실행 중에 메소드의 호출이 발생하면 JVM은 해당 객체의 최하위 class 부분부터 검사하여 해당 메소드를 찾아 호출함. 이런 식의 탐색을 동적 바인딩이라고 함.

필드나 static 메소드 등은 컴파일 시에 확인할 수 있는 해당 타입부터 시작하여 상위 타입으로 거슬러 올라가며 처음 발견한 필드/메소드를 체택함. 객체 기준 최하위부터 보는 것이 아니라 컴파일 시점에 가리키고 있는 참조 타입을 기준으로 보는 것.

computer science에서 정적(static)이라는 것은 컴파일 타임에 컴파일러에 의해 결정되는 작업을 말하고, 동적(dynamic)이라는 것은 런타임에 JVM/프로그램에 의해 결정되는 작업을 말함.

컴파일러는 변수에 실제로 대입될 값(객체)은 확인할 수 없으므로 자료형만을 가지고 판단함. 반면에 JVM은 런타임에 변수에 대입된 값을 확인할 수 있어, reference의 경우 해당 객체가 어떤 class로 생성된 객체인지를 판단할 수 있음. 그렇기 때문에 JVM은 런타임에 동적으로 바인딩을 수행함. 즉, 컴파일 타임에는 메소드의 존재만 확인하고, 실행 시에는 실제 호출 함수를 결정(동적 바인딩)함.

동적 바인딩은 가리키는 참조 변수의 타입이 아니라 객체에 대한 것임. 객체를 검사하여 가장 아래의 메소드를 실행함.

아래와 같이 작성한 코드에서도 동적 바인딩이 일어남. 해당 객체에서 바로 선택되었을 뿐 런타임 시에 JVM에 의해 호출될 함수가 정해짐.

```
Line line = new Line();
line.draw();
```

4.4.3. super class 멤버로의 접근

Definition 52 *super 키워드로 super class의 맴버(필드, 메소드)에 접근할 수 있음. 이때 super가 사용된 class의 바로 상위 class부터 위로 올라가며 해당 멤버를 찾아 사용함.*

메소드 오버라이딩이 되어 있는 경우에도 상위 class의 멤버를 사용할 수 있음.

아래와 같이 사용함.

```
System.out.print(super.name);
super.draw();
```

컴파일러는 super를 사용한 접근을 정적 바인딩으로 처리함.

예를 들어, draw()를 오버라이딩한다고 할 때 공통되는 코드는 상위 draw()에 적어 놓고, 오버라이딩할 때 super.draw()를 작성하면 해당 코드를 공통적으로 사용할 수 있음.

4.4.4. 애너테이션

Definition 53 *애너테이션(Annotation)은 코드에 대한 메타데이터(데이터에 대한 데이터)의 한 형태로, 정보를 제공하거나 컴파일러에게 특정 작업을 수행하도록 하는 주석임.*

comment(//)와 애너테이션은 모두 주석으로 번역되는데, comment는 사용자에게 정보를 제공하고 애너테이션은 컴파일러에게 정보를 제공함.

예를 들어, 메소드 오버라이딩에서는 아래와 같이 메소드 위에 애너테이션을 작성할 수 있음. 애너테이션은 안 적어도 되지만, 작성하면 컴파일러에게 의도를 제공한 것으로, 컴파일러는 이에 따라 적절하게 오버라이드될 수 있도록 검사하거나 코드를 직접 생성해주는 등의 작업을 수행함.

```
@Override
```

4.4.5. 프레임워크에서의 메소드 오버라이딩

프레임워크는 라이브러리에 몇 가지 요소가 추가된 것을 말함. Spring같은 프레임워크는 Java 기반인데, 상위 class와 실제 사용 메소드 등을 제공함. 각 class별로 상위 class를 상속받아서 실제 사용 메소드에 사용된 메소드를 오버라이딩하여 사용하도록 되어 있음.

상위 class가 존재하고, 하위 class들은 해당 상위 class의 오버라이딩만 갖는 것이 이상적인 프레임워크 상속 구조임.

4.4.6. 메소드 오버라이딩이 불가능한 경우

final, static, private은 메소드 오버라이딩이 불가능함.

1. 컴파일 에러
final이 붙은 메소드는 오버라이딩이 불가능함. 오버라이딩을 시도하면 컴파일 에러가 발생함.

2. @Override 에러
static과 private에 대해서는 오버라이딩처럼 작성하는 것은 가능한데, 오버라이딩이 되지는 않음. 해당 메소드 위에 @Override를 작성하면 오버라이딩이 되지 않는 경우이므로 @Override 부분에서 컴파일 에러를 냄.

static과 private 메소드에 대해서는 정적 바인딩이 수행됨.

static에 대해서는 컴파일러가 해당 class에 대해 메소드를 바인딩함. private에 대해서는 애초에 해당 class 내부에서만 접근이 가능하므로, 객체를 생성해서 접근하려고 하면 컴파일 에러가 남.

4.5. 추상 메소드/class

4.5.1. 추상 메소드

Definition 54 *subclass에서 메소드의 구현부를 구현하도록 하여 선언한 메소드를 추상 메소드(abstract method)라고 함. 추상 메소드는 구현부가 존재하지 않음에도 해당 class에 등록됨.*

추상 메소드는 아래와 같이 abstract를 붙이고 구현부를 작성하지 않음.

```
public abstract String getName();
```

추상 메소드가 존재하는 class는 추상 class여야 함. 추상 class가 아닌 경우 컴파일 에러가 남. 추상 메소드는 언젠가는 subclass에서 오버라이딩하여 구현해야 함. 이때 오버라이딩을 하는 class는 abstract가 아니어도 됨.

즉, 추상 메소드를 상속받으면서 구현(오버라이딩)하지 않는 class는 추상 class이고, 구현(오버라이딩)하는 class부터는 추상 class가 아니어도 됨.

super를 사용한 접근은 정적 바인딩이므로 추상 메소드를 super로 호출하려고 하면 컴파일 에러가 남. 애초에 추상 메소드는 구현이 되어 있지 않으므로 호출이 불가능함.

추상 메소드를 오버라이딩하는 것을 구현한다고 표현함. 구현이 된 메소드는 concrete method라고 함.

물론 계속 추상 class에게 상속된다면 해당 함수를 호출할 수는 없지만 컴파일 에러가 나지는 않음. 모든 하위 class가 추상 class라면 해당 상속 구조의 class로 객체를 생성할 수 없기 때문에 호출하는 코드 자체가 불가능함. 객체를 생성하지 않고 호출하려면 static이어야 하는데, static 메소드는 애초에 오버라이딩이 불가능하므로 컴파일 에러가 남.

추상 메소드 중 일부는 어떤 subclass에서 구현하고, 나머지는 그 subclass의 subclass에서 구현하는 형태로도 사용함.

4.5.2. 추상 class

Definition 55 *추상 class(abstract class)는 다른 class들이 상속받을 수 있지만 독립적으로 객체를 생성할 수는 없는 class임.*

추상 class는 아래와 같이 abstract를 붙임.

```
public abstract class Shape()
{
    ...
}
```

추상 class는 해당 class만을 가지는 객체를 생성할 수 없음. 생성자를 작성하거나 subclass로 객체를 생성하는 것은 가능함. 즉, new로 해당 class 객체를 생성하는 것이 불가능함.

추상 class는 추상 메소드를 가질 수도 있고, 가지지 않을 수도 있음. 추상 메소드가 존재하지 않아도 개념적으로 적절하다면(명시적 객체 생성 제한) 추상 class로 하는 것이 자연스러움. 또한 메소드를 상속하여 오버라이딩하더라도 공통 작업은 super class에 정의해 두고 super로 접근하도록 구현할 수도 있음.

상속 구조는 위쪽으로 갈수록 추상적이고 아래쪽으로 갈수록 구체적임. 추상적인 개념에 대해 메소드를 구현하거나 객체를 생성하는 것이 의미적으로 어색해질 수 있는데, 이런 필요에 의해 추상 메소드/class가 존재하는 것. 만약 추상 메소드/class를 사용하는 대신 해당 class를 아예 빼 버리면 다형성을 활용할 수 없게 됨.

4.6. interface

4.6.1. interface

Definition 56 *interface는 가질 수 있는 멤버가 제한된 class로, 소프트웨어에서의 인터페이스로 기능함. interface에는 주로 class가 구현해야 할 메소드들을 정의함.*

interface 또한 class이므로 .java 파일로 작성하고, 컴파일하면 .class 파일이 됨.

pc의 조립으로 이해해 보자면, 메모리, cpu, 그래픽카드 등의 컴포넌트들을 연결하는 메인보드를 인터페이스로 생각할 수 있음. 인터페이스를 우선 정의함으로써, 각각의 컴포넌트를 독립적으로 만들 수 있고, 필요시 컴포넌트를 간단하게 교체할 수 있음.

이때 당연하게도 각 컴포넌트는 독립적이어야 함. 컴포넌트1에서 class A가 정의되었다면 컴포넌트2에서는 class A가 명시적으로 등장하지 않아야 독립적이라고 할 수 있음. 상위타입 참조변수로 객체를 가리켜 이를 구현할 수 있음.

interface로 추상 메소드를 정의해 두고 각 컴포넌트에서 구현하면, 해당 메소드를 사용하는 컴포넌트에서는 interface만 보고 해당 메소드를 사용할 수 있음.

4.6.2. interface 정의

Definition 57 *1. interface 정의*
interface는 아래와 같이 정의함. interface를 작성하면 abstract class를 작성한 것과 동일하게 동작함. interface의 접근지정자로는 class와 같이 public과 default가 올 수 있음.

```
public interface A{
    ...
}
```

2. 필드와 메소드 정의
일반 class와 interface는 가질 수 있는 멤버에 있어 차이가 있음. 우선 멤버의 접근지정자로는 public만 가능함.

필드로는 상수만 올 수 있음. 아래와 같이 자료형과 식별자만 작성하면 public static final이 추가됨.

```
int a; // public static final int a;
```

메소드로는 추상 메소드만 올 수 있음. 아래와 같이 자료형과 식별자만 작성하면 public abstract가 추가됨. default처럼 작성해도 public인 것 유의해야 함.

```
void f(); // public abstract void f();
```

3. 생성자
interface에는 생성자가 존재하지 않음. 기본생성자 또한 존재하지 않음.

4.6.3. interface 사용

Definition 58 *1. class의 구현*
interface의 추상 메소드는 class에서 구현함. 아래와 같이 implements를 작성하여 추상 메소드를 구현하는 class를 interface와 연결할 수 있음. 이때의 implements는 구현한다는 것을 의미하고, interface도 class이므로 interface가 해당 class의 상위 class가 됨. 다이어그램으로 나타낼 때 implements는 점선 화살표로 나타냄.

interface에서 멤버를 작성할 때는 public 등을 생략할 수 있었지만, 구현할 때는 class에 작성하는 것이므로 생략할 수 없음.

```
public class A implements I{
    ...
}
```

이때 아래와 같이 implemnets로 지정하는 interface는 여러 개일 수 있음. 기존의 상속에서 여러 class를 상속받을 수 없었던 것은 겹치는 메소드에 대해 처리가 존재하지 않기 때문이었는데, interface에는 애초에 메소드가 구현되어 있지 않으므로 겹치는 경우에 해당 interface 모두에 class의 구현이 반영됨.

또한 다른 class를 상속받는 class에 대해서도 interface를 implements로 지정할 수 있음.

```
public class A extends B implements I1, I2, I3{
    ...
}
```

2. interface끼리의 상속
interface끼리도 상속이 가능함. interface끼리는 구현하는 것이 아니므로 extends로 상속을 나타냄. 이때 implements로 여러 개의 interface를 지정할 수 있는 것처럼, 여러 개의 interface를 상속할 수 있음. 또한 interface와 그것을 상속받는 interface는 서로 추상 메소드가 겹쳐도 됨.

```
interface I extends I1, I2{
    ...
}
```

인터페이스가 상속되어 있으면 그 인터페이스를 implements하는 class는 부모 인터페이스도 구현해야 함.

3. data type으로 사용
interface도 class이므로 data type으로 사용이 가능함. 변수의 자료형이나 캐스트 연산자에 사용이 가능한데, 추상 class이므로 new로 단일 객체로서 생성할 수는 없음.

일반적인 class와 동일하게 가리키는 객체에서 해당 interface에 존재하는 멤버만 사용이 가능함.

4.6.4. companion class

Definition 59 *interface에서는 하위 객체의 식별자가 등장해서는 안 되므로, 객체의 생성 등을 위해서 사용하는 것이 comapanion class임. interface를 지원하는 class인 것.*

주로 companion class는 지원할 인터페이스 식별자에 s만 붙여 정의함.

companion class에는 대체로 유틸리티성(static) 메소드를 작성함.

예를 들어, interface를 사용하는 main에서 I game = new A();와 같이 작성하도록 구현해서는 안 됨. companion class에 getInstance() return new A(); 를 정의해 두고 interface와 함께 getInstance()를 사용하도록 하는 것.

4.7. Java8부터의 interface

Java7까지는 interface의 멤버로 constant와 추상 메소드만 가능했지만, Java8부터 default 메소드와 static 메소드가 추가되었음.

4.7.1. default 메소드

Definition 60 *interface에서 구현부를 작성할 수 있도록 한 메소드가 default 메소드임. 즉, default 메소드는 추상 메소드가 아니어서 오버라이드하지 않고도 호출할 수 있음.*

아래와 같이 default 키워드를 메소드 앞에 작성하여 default 메소드를 정의할 수 있음.

```
interface I{
    default void doA
    {
        ...
    }
}
```

이미 다른 class들에 의해 implements된 interface에 새로운 추상 메소드를 추가하면 해당 interface를 구현하고 있는 모든 class에 해당 메소드에 대한 코드를 추가해야 함. 특히 해당 interface를 라이브러리로 배포했는데 이를 수정하면 그 여파가 큼. 이런 경우를 위해 interface에서 구현하여 이와 연결된

class에서 컴파일 에러가 나지 않도록 한 것이 default 메소드임.

4.7.2. default 메소드와 상속

Definition 61 *1. default 메소드의 상속*
default 메소드는 마치 일반 class의 메소드가 상속되는 것처럼 동작함.

해당 interface를 implements하는 class들의 객체에는 default 메소드가 존재하게 됨. 또한 default 메소드는 해당 interface를 implements하는 class들에서 오버라이딩이 가능함. 해당 interface를 상속받는 interface에서도 오버라이딩이 가능함.

2. I.super를 이용한 접근
해당 interface를 implements하는 class에서 super로 상위 default 메소드에 접근하려면 아래와 같이 interface 식별자와 super를 함께 작성함. 어떤 class가 다른 class를 상속하고 다른 interface를 구현하고 있는 경우, super만 작성하면 상속하고 있는 쪽에서 바인딩을 하도록 되어 있음.

```
interface I{
    default void doI() { ... }
}
class A implements I{
    @Override
    public void doI()
    {
        I.super.doI();
    }
}
```

이때 interface.super로 지정할 수 있는 interface는 해당 class에 implements로 지정된 것만 가능함. I1 <- I2 <... A인 경우 A에서는 I2.super는 가능하지만 I1.super로 작성하면 컴파일 에러가 남.

여러 개의 interface를 implements하고 있는 경우 각 interface의 식별자로 super를 지정하여 해당 interface의 메소드로 바인딩할 수 있음.

4.7.3. default 메소드가 충돌하는 경우

Definition 62 *우선 정리하면, default 메소드와 메소드가 충돌하는 모든 경우들에서는 interface의 default 메소드가 해당 class에 반영되지 않음. 메소드가 겹치는데 어느 한 쪽에 구현이 존재하는 경우 대체로 오버라이딩을 통해 이를 해결함.*

1. extends와 implements의 메소드가 겹치는 경우
extends와 implements의 메소드가 겹치는 경우 extends의 메소드의 구현이 채택됨. 정확히는, extends가 먼저 처리되어 해당 메소드가 class에 추가되고, class에 메소드가 존재하므로 interface의 default 메소드가 오버라이딩되는 것임.

기존의 interface에는 추상 메소드만이 존재하여 extends와 동일한 signature의 메소드가 존재해도 구현이 없기 때문에 문제가 없었음. 하지만 default 메소드가 등장하면서 extends와 implements의 구현이 겹치는 경우가 발생하게 된 것.

물론 해당 메소드에 대해 오버라이딩을 명시적으로 할 수도 있음.

2. 두 interface의 메소드가 겹치는데 둘 중 하나가 default인 경우
하나의 class에서 implements하는 두 interface의 메소드가 겹칠 때 둘 중 하나가 default 메소드라고 하면, extends의 경우와는 달리 해당 구현이 반영되고 오버라이딩되지 않음. 이 경우는 컴파일 에러가 남.

interface끼리의 충돌은 해당 메소드의 signature로 오버라이드하여 해결함. 그러면 두 interface의 메소드가 모두 오버라이드되면서 컴파일 에러가 해결됨.

추상 class와 메소드가 겹치는 경우도 동일하게 처리함.

3. 두 interface의 메소드가 겹치는데 둘 다 default인 경우
하나의 class에서 implements하는 두 interface의 메소드가 겹칠 때 둘 다 default 메소드라고 하면, 이 경우에도 컴파일 에러가 남.

이 경우에도 오버라이드하여 해결하면 됨.

4.7.4. static 메소드

Definition 63 *static 메소드는 아래와 같이 static 키워드를 사용하고 구현부가 존재해야 함. 추상 메소드에 static을 붙일 수는 없음. default와 static을 모두 작성할 수는 없음. 작성하면 컴파일 에러 남.*

```
interface I{
    static void doI() { ... }
}
```

일반적인 static과 같이 static 메소드의 소유자는 class(interface)이고, 정적 바인딩이 됨. 단, 일반적인 static은 class명으로 호출하는 것과 객체로서 호출하는 것 모두를 허용했지만, static 메소드는 class 명으로 호출하는 것만 허용함. 객체로서 호출하려고 하면 컴파일 에러가 남. static 메소드가 추가되는 시점의 개발자들은 class 소유의 static을 객체로 접근하는 것을 막도록 반영한 것임.

4.8. Java9부터의 interface

4.8.1. private 메소드

Definition 64 *private 메소드는 아래와 같이 private 키워드를 사용하고 구현부가 존재해야 함. 추상 메소드에 private을 붙일 수는 없음. default와 private을 모두 작성할 수는 없음. 작성하면 컴파일 에러 남.*

```
interface I{
    private void doI() { ... }
}
```

일반적인 private과 같이 private 메소드는 해당 class(interface) 내에서만 사용이 가능함. 주로 default 메소드 작성 시에 코드가 길어지는 경우 subroutine으로 쪼갤 때 사용함.

private 메소드와 default 메소드는 서로 호출할 수 있음. 또한 독특한 점은 default, private 메소드에서는 해당 interface의 추상 메소드를 호출할 수 있음. 이 메소드가 실제로 호출되어 실행되는 시점에는 해당 interface를 구현하는 class의 객체가 존재한다는 것이므로 가능한 것.

정리하자면, default, private 메소드에서는 interface의 모든 메소드를 호출할 수 있음.

4.8.2. private static 메소드

Definition 65 *private static 메소드는 아래와 같이 static 메소드에 private 키워드를 사용한 것임. 마찬가지로 추상 메소드에 private static을 붙일 수는 없고, default와 함께 작성할 수 없음.*

```
interface I{
    private static void doI() { ... }
}
```

static 메소드에서는 일반적인 static과 같이 non-static 메소드를 호출할 수 없기 때문에, static 메소드에 대한 subroutine을 작성할 수 있도록 하는 문법임. 그러므로 당연하게도 private static과 static은 서로 호출할 수 있고, static이 붙지 않은 것들은 호출할 수 없음.

Part III

패키지와 class

1. java.lang

본 장에서는 유용한 패키지들을 배움.

1.1. java 패키지

1.1.1. java 주요 패키지들

java에서 제공하는 주요 패키지로는 아래와 같은 것들이 있음.

java.lang : String, Math, System 등 기본적인 class/interface가 들어 있는 패키지로, 자동으로 import됨.
java.util : 다양한 유틸리티(static) class/interface가 들어 있는 패키지.
java.io : 입출력을 위한 class/interface가 들어 있는 패키지.
java.awt : GUI 프로그래밍을 위한 class/interface가 들어 있는 패키지.
java.swing : GUI 프로그래밍을 위한 swing 패키지.

특정 패키지/class에 대한 자세한 내용이 궁금하다면 웹에 있는 Java API 문서를 확인하자. 구글에 Java API 라고만 쳐도 찾을 수 있음.

1.2. Object class

1.2.1. Object class

Definition 66 *Object class는 모든 class에 강제 상속되는 super class로, 객체의 속성을 나타내는 메소드들을 보유하고 있음.*

아래는 가장 핵심적인 메소드들임.

boolean equals(Object obj) : obj가 가리키는 객체와 같은지 비교.
int hashCode() : 객체의 해시 코드 값 반환.
String toString() : 객체에 대한 문자열 반환.
Class getClass() : 객체의 런타임 class의 정보를 반환.

어떤 class를 만들 때는 기본적으로 equals(), hashCode(), toString()을 오버라이드 해야 다른 개발자가 사용할 때 제 기능을 할 수 있음.

1.2.2. equals()

equals()로 두 객체의 값이 동일한지를 반환하도록 오버라이딩해야 함.

두 대상의 비교에는 아래와 같이 세 가지 종류가 있음.
1. identity의 비교 : 실제로 같은 대상인지를 비교함. java에서는 ==로 비교함.
2. equality의 비교 : 값이 같은지를 비교함. java에서는 equals로 비교함.
3. equivalency의 비교 : 두 대상이 동등한지를 비교함.

오버라이딩을 하지 않으면 equals()는 identity를 비교하는 연산을 수행함. 암묵적인 기대는. equals()는 equality를 비교해야 하기 때문에, 오버라이드해 줘야 함.

이때 내부적으로 instanceof를 사용하여 비교 가능한 class인지 검사해야 함. 아예 비교 불가능한 객체가 들어올 수 있음.

1.2.3. toString()

toString()은 class의 식별자와 필드 값 등을 문자열로 구성하여 반환하도록 오버라이딩해야 함. 오버라이딩하지 않은 toString()은 'Point@14db9742'의 형태로 문자열을 반환하는데, 오버라이딩하여 보기 쉽게 작성해야 함. 물론 이때 제공하고 싶은 필드만 출력하는 식으로 구현할 수 있음.

객체에 대해서 class 식별자와 필드 값을 확인하는 것은 디버깅 시에 유용함. toString()을 디버깅 시에 사용할 수 있다는 암묵적인 기대에 부흥해야 함.

참고로, 어떤 객체를 가리키고 있는 참조 변수를 System.out.print()로 출력하려고 하면 해당 객체의 toString() 값으로 자동 변환됨.

아래와 같이 출력되도록 오버라이딩할 수 있음.

```
System.out.println(p.toString()); // Point(2,3) 출력됨.
```

1.2.4. getClass()

getClass()는 Class class의 객체를 반환하는데, Class 객체는 식별자, 멤버 등 해당 class에 대한 런타임 정보를 가지고 있음. Class의 메소드들을 사용하여 정보를 확인할 수 있음.

JVM은 Class 타입 객체를 사용하여 런타임에 객체에 대한 정보를 획득함. 개발자 입장에서는 Class 객체를 사용한 정보 확인은 크게 쓸모있지 않음. 다형성에 활용하기에 무리가 있으므로.

1.3. boxing/unboxing

1.3.1. Wrapper class

Definition 67 *primitive형 데이터를 reference형(객체)으로 다루기 위한 class들을 Wrapper class라고 함. Wraper class로는 아래와 같은 것들이 있음.*

Byte(byte용), Short(short용), Integer(int용), Long(long용), Character(char용), Float(float용), Double(double용), Boolean(boolean용).

1. Wrapper 객체의 생성
Wrapper class의 객체는 new를 사용하여 생성할 수 없음. 대신 아래와 같이 valueOf() 유틸리티 메소드를 사용함. 매개변수로는 Wrapper에 해당하는 primitive형 데이터 또는 문자열을 작성할 수 있음. 추가로, Float의 객체는 double형 값을 넣어 생성할 수 있음.

```
Integer a = Integer.valueOf(10);
Integer b = Integer.valueOf("10");
Float c = Float.valueOf((double)3.14);
```

2. Integer의 주요 메소드들 wrapper class들은 서로 다른 class이지만 동일한 식별자의 메소드가 정의되어 있음. 같은 규칙으로 식별자명을 작성하면 대체로 맞음.

대표적으로 Integer의 주요 메소드들은 아래와 같음.

** *Value() : *에 int/float/long/short를 작성하여 해당 primitive형으로 값 반환.*
static int bitCount(int i) : 해당 정수의 이진 표현에서 1의 개수 반환.
static int parseInt(String s) : 해당 문자열을 10진 정수로 변환하여 반환.
*static String to*String(int i) : *에 Binary/Octal/Hex를 작성하여 정수를 해당 진법으로 변환하여 문자열로 반환.*
static String toString(int i) : 정수를 문자열로 변환하여 반환.

Wrapper class는 형변환 유틸리티 메소드가 주 메소드임. Wrapper 객체의 생성과 값의 추출은 자동 boxing/unboxing으로 처리하기 때문.

특히 자료구조 등에 대한 class들에서는 인자로 reference형(객체)를 받도록 구현되어 있는데, 이런 경우 primitive형을 reference형으로 바꾸어 줘야 함.

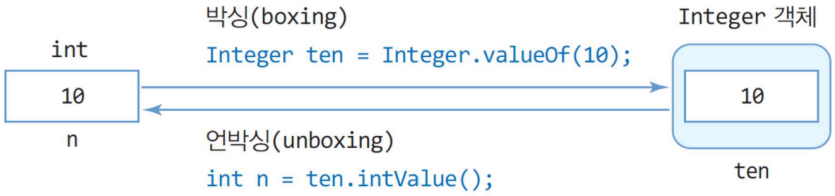
Wrapper class에는 Wrapper 객체와 그에 해당되는 primitive 데이터 사이의 변환 외에도, String 객체로의 변환이 정의되어 있음. String 객체로 변환한 값이 파일 입출력, 네트워크 등에서 자주 사용되기 때문임. (String <-> Integer <-> int 로 생각하자.)

1.3.2. boxing/unboxing

Definition 68 *primitive형 데이터를 Wrapper 객체로 변환하는 것을 boxing, Wrapper 객체의 값을 primitive형 데이터로 변환하는 것을 unboxing이라고 함.*

valueOf(), intValue() 메소드로 boxing/unboxing을 할 수 있지만, 대체로 아래와 같이 자동 boxing/unboxing으로 boxing/unboxing을 수행함.

```
Integer a = 10; // 자동 boxing
int b = a; // 자동 unboxing
```



1.4. Math

1.4.1. Math

Definition 69 *Math는 산술 연산을 제공하는 class임.*

java.lang에 있음.

메소드들이 대체로 인자는 double형으로 받고, 연산 결과는 double로 반환함. abs(절댓값), cos, sin, tan, ceil(올림), floor(내림), max, min, random([0.0, 1.0)의 값 반환) sqrt(제곱), round(실수를 반올림하여 정수로 반환. long으로 반환함.) 등의 연산이 존재함.

난수 발생 시에는 주로 random()에 10의 거듭제곱을 곱하고 특정 정수로 modular 연산을 하는 방식을 사용함. 난수 생성을 위한 java.util.Random class도 존재함.

1.5. Calendar

1.5.1. Calendar

Definition 70 *Calendar는 시간과 날짜에 대한 정보를 저장하고 관리하는 class임.*

java.util에 있음.

Calendar는 abstract class로, new로 객체를 생성할 수 없음. 아래와 같이 객체 생성을 위한 별도의 메소드가 존재함.

```
Calendar c = Calendar.getInstance();
```

Calendar는 아래와 같은 필드들을 가지고 있음.

필드	의미	필드	의미
YEAR	년도	DAY_OF_MONTH	한 달의 날짜
MONTH	달(0~11)	DAY_OF_WEEK	한 주의 요일
HOUR	시간(0~11)	AM_PM	오전인지 오후인지 구분
HOUR_OF_DAY	24시간을 기준으로 한 시간	MINUTE	분
SECOND	초	MILLISECOND	밀리초

2. 문자열

2.1. CharSequence

2.1.1. CharSequence

Definition 71 *CharSequence는 char형 값들의 나열인 읽기 전용 데이터에 대한 interface임. charAt(), length() 등이 존재함.*

String, StringBuffer, StringBuilder가 CharSequence를 구현(implements)하는 가장 대표적인 class임. 즉. 이 세 class가 java에서 문자열을 다루는 가장 핵심적인 class들임.

CharSequence도 java.lang에 있음.

2.2. String

2.2.1. String 객체의 생성

Definition 72 *String 또한 class이므로 아래와 같은 생성자들을 가지고, new 키워드를 사용해 객체를 생성할 수 있음.*

String() : 빈 String 객체 생성.
String(char[] value) : char형 배열로 String 객체 생성.
String(String value) : String 객체로 동일한 값의 String 객체 생성.
String(StringBuffer buffer) : StringBuffer 객체로 String 객체 생성.

```
String s = new String(data);
```

특히 네트워크 등에서 입력을 문자 하나씩 계속 받아야 하는 경우, 배열로 입력받아 String 객체로 변환하는 것이 유용할 수 있음.

2.2.2. String literal과 new로 생성한 String

literal로 생성한 String 객체는 JVM이 관리하는데, 해당 값의 String literal에 대해서는 모두 같은 메모리 공간을 사용하게 됨. 이때의 힙에 존재하는 메모리 공간을 literal pool이라고 함.

new로 String 객체를 생성하면 literal pool이 아닌 힙의 다른 부분에 객체가 생성됨. 즉, String literal과 메모리 공간을 공유하지 않음.

예를 들어, "Hello"를 여러 번 사용하면 각각 동일한 참조가 사용되지만, new로 "Hello" 값을 가지는 객체를 생성하면 다른 메모리 공간이 할당되어 다른 참조가 사용됨.

2.2.3. String 객체 특징

1. String 객체의 값은 수정이 불가능함.(String objects are immutable)
String 객체의 값 뒤에 다른 문자열을 추가한다거나 하는 메소드를 수행하면, 원래의 String 객체는 버려지고 문자열이 추가된 String 객체가 반환되는 식으로 동작함.

최근 언어들은 immutable 데이터를 늘리려고 함. 멀티코어 등이 등장하면서 하나의 데이터를 여러 프로세서에서 처리할 수 있게 되었는데, 서로 다른 값으로 변경하여 각 캐시에 저장해 뒀다면 그 결과를 메모리에 store했을 때 값이 완전히 이상해질 수 있음. 이에 대한 해결법으로 한 코어에서 처리 중일 때 다른 코어에서 건드릴 수 없도록 한다면 멀티코어로서의 효능이 없음. 하지만 데이터가 immutable하다면 각 프로세서에서 무지성하게 가져가서 사용할 수 있음. java는 기본값이 int이고 final 키워드로 immutable 지정을 하지만, 최근 언어들은 기본값이 final이고 int로 쓰려면 별도의 처리가 필요하기도 함.

2. ==은 연산자 오버로딩이 되어있지 않음.
비교 시에는 equals(), compareTo() 등의 메소드를 사용해야 함.

2.2.4. String 메소드들

Definition 73 *String class에는 대표적으로 아래와 같은 메소드들이 있음.*

char charAt(int index) : 해당 index의 문자 값 반환.
int codePointAt(int index) : 해당 index 문자의 유니코드 값 반환.
int compareTo(String s) : 두 String을 사전순으로 비교하여 같으면 0, 인자로 작성한 String이 더 먼저이면 양수, 더 나중이면 음수를 반환함. (순서 유의)
String concat(String s) : 해당 String에 s를 붙인 String 객체 반환.
boolean contains(CharSequence s) : s에 해당하는 부분을 가지고 있는지 반환.
int length() : 문자열 길이 반환.
String replace(CharSequence target, CharSequence s) : target을 s로 대체한 String 객체 반환.
String[] split(String regex) : regex(정규식)에 일치하는 부분을 기준으로 String을 자르고 배열로 저장하여 배열 객체 반환.
String toLowerCase() : 소문자로 변환한 String 객체 반환.
String toUpperCase() : 대문자로 변환한 String 객체 반환.
String trim() : 앞뒤의 공백 문자(tab, space, enter)를 전부 제거한(중간의 공백에는 적용x) String 객체 반환.
String intern() : 해당 String 객체의 값과 일치하는(equals()) String literal을 literal pool에서 찾아서 반환하고, 해당 literal이 없다면 해당 값으로 literal을 생성하여 반환함.

물론 concat() 대신 +로 이어붙일 수도 있음.

메소드를 적용해도 기존의 String 객체의 값은 바뀌지 않는 것 유의. 사용 예시를 보면 대체로 기존의 참조변수에 메소드의 반환값을 대입하는 형태임.

compareTo() 메소드는 Comparable이라는 Interface를 implements한 것임. 객체의 비교가 가능한 class에 대해서는 Comparable을 implements하고 compareTo()를 구현해두는 것이 좋음. 특히 정렬이나 자료구조에 대해서는 각 객체의 비교가 필수적이기 때문.

intern()으로 literal pool에 값을 올려 사용하면 메모리를 아낄 수 있음.

API를 보면 어떤 메소드에는 regex가 매개변수로 들어가 있는데, 이는 정규식(정규표현식, regular expression)을 의미함. 특정 규칙에 따라 작성하여 문자열에 대한 특정 패턴을 지정할 수 있음. (복잡하므로 시험에는 내지 않음. 필요한 경우에 가져다 쓰자.)

2.3. StringBuffer

2.3.1. StringBuffer

Definition 74 *StringBuffer class는 가변길이(mutable) String임. 즉, 값의 변경이 가능한 문자열임.*

java.lang에 있음.

StringBuffer 객체는 아래와 같은 생성자들을 가지고, new 키워드를 사용해 생성할 수 있음.

StringBuffer() : 초기 buffer의 크기가 16인 객체 생성.
StringBuffer(charSequence s) : s를 포함하는 객체 생성.
StringBuffer(int capacity) : capacity만큼의 초기 buffer 크기를 갖는 객체 생성.

StringBuffer(String s) : s로 초기화된 객체 생성.

```
StringBuffer s = new StringBuffer("Hello");
```

2.3.2. StringBuffer 메소드들

Definition 75 *StringBuffer class에는 대표적으로 아래와 같은 메소드들이 있음.*

StringBuffer append(String s) : s를 해당 객체 뒤에 붙임.
StringBuffer append(StringBuffer s) : s를 해당 객체 뒤에 붙임.
int capacity() : 객체의 현재 메모리 크기 반환. (문자열 길이 아님)
StringBuffer delete(int start, int end) : start end-1까지 삭제.
StringBuffer insert(int offset, String s) : s를 offset 위치에 삽입.
StringBuffer replace(int start, int end, String s) : start end-1까지를 s로 대체.
StringBuffer reverse() : 문자열 뒤집기.
void setLength(int newLength) : 문자열의 길이 지정. 현재 문자열 길이보다 작으면 문자열이 잘리고, 크면 null 문자가 들어감.

StringBuffer 객체는 그 값을 수정할 수 있으므로 String에서처럼 메소드의 반환값을 다시 배정하는 식으로 사용하지 않고, 단순히 메소드를 호출하기만 하면 반영이 됨.

start, end 등으로 범위를 지정하는 메소드에 대해서는 대체로 start $\leq \cdots <$ end임.

null 문자는 유니코드기 0인 것을 의미함. c에서의 null 문자가 아니라, 단순히 빈 문자를 뜻하는 것.

StringBuffer로 문자열을 편리하게 다루고, StringBuffer class에서 오버라이드한 toString()을 사용해 String으로 전환할 수 있음.

2.4. StringBuilder

2.4.1. StringBuilder

Definition 76 *StringBuilder는 StringBuffer처럼 mutable이지만, 동기화(synchronization)가 보장되지 않는 class임.*

StringBuilder는 StringBuilder와 동일한 API를 가짐.

java.lang에 있음.

2.4.2. thread

Definition 77 *thread는 실행 흐름을 말함.*

프로그램을 실행하면 기본적으로 하나의 thread(main)가 존재하는데, multiple thread를 사용하여 여러 실행 흐름을 가지고 병렬적으로 작업수행할 수 있음.

2.4.3. 동기화

Definition 78 *class가 동기화(synchronization)가 가능하다는 것은 multiple thread에서 사용이 가능하다는 것을 말함. 반대로 동기화되어 있지 않다면 single thread에서만 사용해야 함을 말함.*

java에서 객체는 힙에 저장되므로 java에서 각 스레드는 모두 힙에 동시에 접근할 수 있음. 두 개의 스레드가 각각의 메소드를 동시에 호출하여 수행한다면, 작업이 꼬이면서 이상한 결과가 도출될 수 있음. 동기화되어 있다는 것은 특정 작업들이 서로 겹치지 않도록 했다는 것임. 두 메소드가 모두 값을 수정한다면.. 하나의 메소드가 수행될 때 다른 메소드가 수행되지 않도록 처리하는 등으로 부여한 순서에 따라 작업을 수행하도록 함.

StringBuilder와 StringBuffer의 관계와 같이 사용법은 동일하지만 동기화 유무에만 차이가 있는 경우가 종종 등장함.

동기화되어 있다는 것은 동기화 코드가 추가로 들어가 있다는 것임.

2.5. StringTokenizer

2.5.1. StringTokenizer

Definition 79 *StringTokenizer는 문자열을 여러 개의 토큰으로 분리하는 class임.*

java.util에 있음.

아래와 같이 구분 문자(delimiter)를 기준으로 문자열(토큰)을 분리하여

```
StringTokenizer a = new StringTokenizer(inputString, "&="); // &와 =로 구분
```

StringTokenizer는 아래와 같은 생성자들을 가짐.

```
StringTokenizer(inputString) // 기본적으로 공백을 기준으로 구분함
StringTokenizer(inputString, "&=")
StringTokenizer(inputString, "&=", true) // 구분 문자도 토큰에 포함시킬지를 지정
```

2.5.2. StringTokenizer 메소드들

Definition 80 *StringTokenizer에는 아래와 같은 메소드들이 있음.*

int countTokens() : 토큰의 개수 반환.
boolean hasMoreTokens() : 현재 위치 기준 다음 토큰이 존재하는지를 반환.
String nextToken() : 현재 위치 기준 다음 토큰 반환. 이 메소드로 순회하며 토큰을 사용함.

3. Collection

3.1. Generic

3.1.1. Generic

Definition 81 *Generic(일반적인)은 하나의 class나 메소드를 여러가지 타입으로 다룰 수 있도록 하는 문법임. generic은 class와 interface에 대해서 사용이 가능함. generic이 적용된 class/interface의 타입을 generic type이라고 하고, 적용되지 않은 것은 raw type이라고 함.*

1. <>
generic은 아래와 같이 <>를 붙여 나타내는데, 이를 다이아몬드라고 함. <>안에 작성하는 것을 타입 매개변수(type parameter)라고 함.

```
public class Box<T>{    // 이와 같이 정의함
    private T t;
    ...
}
```

type parameter는 개수 제한 없이 작성이 가능함. type parameter로는 뭘 작성해도 되지만 관습적으로 아래의 대문자들을 작성함.

E : element
K : key
N : number
T : type
V : value

2. 치환
generic type으로 정의하였다면, 결과적으로 사용 시에는 실제 타입을 명시해 줘야 하는데 이를 치환한다고 함. 기본적으로 <>안의 문자는 사용 시에 치환해야 함. 아래와 같이 단순히 자료형을 작성하여 치환함. new로 객체 생성 시에 앞 generic에 타입을 작성했다면 뒤에는 생략이 가능함. 또한 var를 사용하여 앞쪽 자료형을 생략하는 문법도 존재함.

```
Vector<Integer> v = new Vector<Integer>();
Vector<Integer> v = new Vector<>();
var v = new Vector<Integer>();
```

extends, implements로 들어온 문자에 대해서는 당장 치환하지 않아도 됨(물론 치환해도 됨)(물론 언젠가는 치환해야 함). 이때 치환하지 않는다면 자식 class가 동일한 type parameter를 가져야 함. 즉, 아래와 같이 type parameter를 맞춰 줘야 함. 이때 부모 class의 type parameter가 존재하기만 하면 됨. 자식 type parameter 순서가 바뀌어도 되고, 그 개수가 늘어나도 됨.

```
class OrderedPair<K, V> implements Pair<K, V>{
    ...
}
```

치환 시에 작성하는 타입은 reference여야 함. primitive는 작성할 수 없음. primitive 값을 저장하려면 해당하는 wrapper class(Integer, Double 등)를 사용해야 함.

3. generic 사용의 이유
generic이 없다면 Object로 데이터를 저장하고 반환하도록 구현해야 하는데, 이 경우 데이터를 꺼내서 Object가 아닌 타입 참조 변수에 저장하려면 매번 캐스트 연산자를 작성해야 함. generic을 사용하면 타입을 명확히 할 수 있고 캐스트 연산자를 매번 사용할 필요가 없음.

4. import
참고로, import 시에는 <>를 작성하지 않음.

generic은 딥하게 가면 복잡해지기 때문에 여기서는 generic을 만든다기보단 만들어진 것을 보고 이해할 수 있을 정도로 정리함.

Collection은 generic을 사용함.

3.1.2. generic 메소드

Definition 82 *generic 메소드는 선언부에 작성한 generic에 대해서 반환 타입과 인자의 타입이 정해지는 메소드임.*

아래와 같이 작성하여 type paratmeter에 해당되는 타입으로 반환 타입과 인자 타입을 지정할 수 있음. 물론 class의 generic과 동일하게 여러 개의 type parameter도 작성 가능함.

```
public static <T> T getName(T name)
{
    ...
}
```

static 메소드에도 generic 사용이 가능함.

1. class의 type parameter와의 구분
class의 type parameter와 generic 메소드의 type parameter는 구분됨. generic class 안에서 generic 메소드를 사용하는 경우, 전역/지역변수 선언하는 것처럼 type parameter가 같더라도 generic 메소드의 type parameter는 개별적인 것으로 취급됨. 이에 따라 type parameter가 같은 변수끼리 대입했는데 실제로는 다른 것이므로 컴파일 에러가 날 수 있음.

메소드 정의에 <>가 존재해야 generic 메소드이므로 class의 type parameter를 사용하는 메소드와 혼동해서는 안 됨.

2. generic 메소드의 사용
사용 시에는 컴파일러가 자료형을 확인할 수 있으므로 <>를 생략하고 일반적인 메소드와 동일하게 작성함.

generic 메소드를 사용하면 인자로 generic type 객체를 받을 수 있음.

예를 들어, 아래와 같은 경우를 조심해야 함.

```
class Box<T> {
private T t;
public <T> T method1(T t) { return this.t = t; } // 제네릭 메소드(O), T와 T는 다름, 컴파일 에러
public T method2(T t) { return this.t = t; } // 제네릭 메소드(X)
}
```

아래와 같이 작성하는 경우, Pair의 K와 V는 실제로 사용되는 것이므로 치환되어야 함. 여기서는 generic 메소드의 K, V로 치환된 것임.

```
public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2)
{
    ...
}
```

3.1.3. bounded type parameter

Definition 83 *bounded type parameter는 type parameter 자리에 작성하여 치환할 수 있는 타입을 제한하는 문법임. 작성 가능한 타입을 제한한다면 개발자 입장에서 해당 class의 구현은 더 자유로워짐.*

1. 작성 방법
아래와 같이 작성하면 extends 뒤에 작성한 타입과 그 타입의 하위 타입들만 <>안에 작성하여 U에 대해 치환할 수 있음.

```
<U extends Number>
<U extends Number, V extends Number>
```

extends를 작성하지 않은 기존의 type parameter는 <U extends Objects>인 것으로 생각할 수 있음.

2. 여러 개의 class/interface 작성하기
하나의 type parameter를 extends할 때 아래와 같이 여러 개의 class, interface를 작성할 수 있음. 이때 class와 interface가 섞여서 작성되는 경우 class를 맨 앞에 작성해야 함.

```
<U extends A & B & C> // A는 class, B/C는 interface
```

3.1.4. type parameter만 다른 경우의 대입

동일한 generic class에 대해, <>에 다른 타입을 작성하여 생성한 두 객체는 다른 타입으로 취급됨. 즉, <> 안에 작성한 타입이 서로 상속 관계에 있더라도 대입이 불가능함. 이렇게 대입이 가능하다면 Box<Number>가 Box<Integer>의 상위 타입이거나 동일한 타입이라는 것인데 이는 의미적으로 말이 안 됨.

```
Box<Number> n = new Box<Integer>; // 컴파일 에러
```

3.1.5. 와일드카드 작성

Definition 84 *<>에 와일드카드를 작성하는 것은 generic class의 실제 사용 시에 참조 변수, 메소드 리턴 타입, 매개변수 등에 와일드카드를 작성하여, 다양한 치환 타입을 모두 가리킬 수 있도록 한 문법임.*

객체 생성 시에 new에는 와일드카드를 작성할 수 없음. 와일드카드는 범용성 있게 가리킬 수 있도록 한 문법임.

upper, lower, unbounded 와일드카드가 존재함. upper는 extends 뒤 class와 그 상위 타입만 가리킬 수 있고, lower는 super 뒤 class와 그 하위 타입만 가리킬 수 있음. unbounded는 모두 가리킬 수 있음. 아래와 같이 작성함.

```
<? extends Number>  // upper
<? super Integer>   // lower
<?>                 // unbounded
```

generic은 Collection에서 계속 등장함. List<?>에서 단순히 <?>로 객체를 가리키면 해당 참조 변수가 가리키는 객체의 실제 타입을 특정하기 어렵기 때문에 안전하게 구현하려면 그 값을 사용하는 경우 Object로 가리켜야 함. 문제는 이 경우 객체가 가진 대부분의 기능을 사용할 수 없음.

자료구조에 대한 읽기 전용으로 구현할 경우 List<? extends Number>로 작성하고(내부적으로 Number 참조 변수로 원소를 받으면 되므로), 쓰기 전용으로 구현할 경우 List<? Integer>로 작성함(내부적으로 Integer 원소를 추가하면 되므로).

와일드카드는 특히 매개변수에서 자주 사용함. 하나의 참조 변수로 다양한 객체를 가리킬 수 있어야 다형성을 구현할 수 있음.

bounded type parameter는 객체 생성 시에 치환할 타입의 범위를 지정하는 것이고, 와일드카드는 그렇게 생성된 객체들을 가리킬 참조 변수 등에 대해 가리킬 수 있는 범위를 지정하는 것임.

3.2. collection

3.2.1. collection

Definition 85 *collection은 java에서 자료구조에 해당하는 class들임. 요소들을 저장하는 컨테이너로 생각할 수 있음.*

1. collection interface와 class
collection에 대한 interface와 이를 구현하는 class들이 있는데, 아래와 같이 interface를 참조 변수로 하고 구현 방식에 맞는 class로 객체를 생성하는 것이 좋음.

```
List<Integer> v = new Vector<Integer>();
```

collection의 생성자로는 대체로 다른 collection을 인자로 넣어 해당 collection으로 변환하는 기능이 존재함.

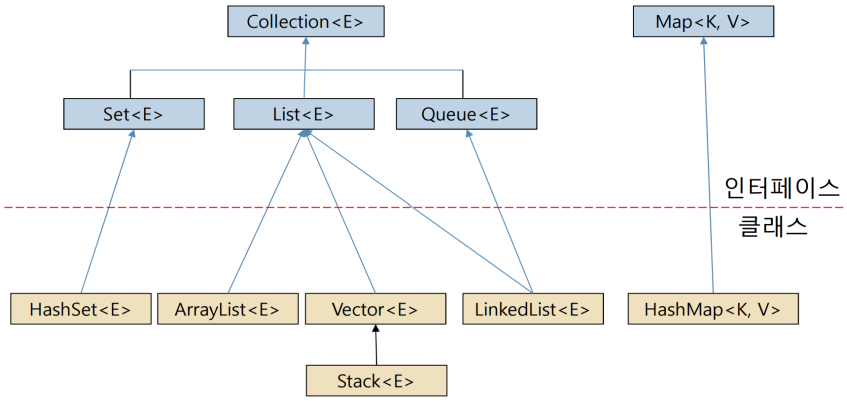
collection interface와 이를 구현하는 class로는 아래와 같은 것들이 있음.

Collection<E> : Set<E>, List<E>, Queue<E>의 부모 interface.
Set<E> : HashSet<E>
List<E> : ArrayList<E>, Vector<E>, LinkedList<E>
Queue<E> : LinkedList<E>
Map<K,V> : HashMap<K,V>

list는 순서가 존재하고 중복 요소를 허용하는 자료구조이고, set은 순서가 존재하지 않고 중복 요소를 허용하지 않는 자료구조임. map은 순서가 존재하지 않고 key-value 요소를 중복 없이(key 중복x) 저장하는 자료구조임.

2. collection 요소
collection의 요소로는 reference 타입만 가능함. 애초에 generic에 primitive를 사용할 수 없으므로 당연함. auto-boxing에 의해 wrapper로도 편리하게 primitive 값을 사용할 수 있음.

이때 <>에 작성한 타입의 참조 변수로 가리킬 수 있는 타입의 데이터들을 collection에 넣을 수 있음. 예를 들어, ArrayList<Number>로 생성했다면 Integer 타입 reference를 add할 수 있음.



stack이 아래에 있는 것은 예전에 만들어졌음에도 호환성을 유지하기 위함임. 현재 java 프로그램 작성 시에는 사용하지 않음. 사용하려고 하면 deque를 사용하라고 함.

Collection과 Map이 분리되어 있는 것은 요소의 개수가 다르기 때문임.

3.2.2. Collections

Definition 86 *Collections class는 Collection interface를 위한 companion class임.*

sort(), reverse(), max(), min(), binarySearch() 등 자료구조에 대한 일반적인 연산을 유틸리티성 메소드로 가지고 있음. 즉, Collections.sort(myList); 등으로 정렬할 수 있음.

3.3. iteration

3.3.1. iteration

Definition 87 *자료구조 내에서의 순회를 iteration이라고 하고, iteration을 수행하는 별도의 객체를 iterator라고 함.*

Collection<E>는 Iterable이라는 interface를 implements함. Iterable에는 iterator 객체를 생성하는 iterator()가 있음. 참고로 Map<K,V>는 Iterable을 implements하지 않음.

iterator()는 해당 collection에 대한 순회를 수행하는 Iterator<E> 객체를 반환함. Iterator<E>가 가지고 있는 메소드는 아래와 같음.

boolean hasNext() : 다음으로 방문할 요소가 남아 있는지 검사.
E next() : 다음 요소를 반환하고 해당 위치로 이동.
void remove() : 마지막으로 반환된 요소 제거.

아래와 같이 작성하여 순회할 수 있음.

```
List<Integer> l = new Vector<>();
...
Iterator<Integer> it = l.iterator();
while(it.hasNext())
{
    int n = it.next();
}

for(Iterator<Integer> it2 = l.iterator(); it2.hasNext;)
{
    int n = it2.next();
}
```

3.3.2. map의 iteration

map은 Iterable을 implements하고 있지 않기 때문에 list에서처럼 iteration할 수 없음. 이 경우 keySet() 메소드를 사용하여 key를 set으로 담아서 반환받고, 이 set에 대해서 iteration하면 얻은 각 key값으로 value를 구할 수 있음. 이런 방식으로 iteration함.

3.3.3. enhanced for

iterator 대신 enhanced for를 사용하여 더 쉽게 순회할 수 있음.

enhanced for는 Iterable interface를 implemnets하는 class라면 enhanced for를 사용할 수 있음. 즉, iterator를 사용할 수 있는 class에 대해서 사용 가능.

```
List<String> list = Vector<>();
for(String str : list)
{
    ...
}
```

3.3.4. ListIterator<E>

Definition 88 *ListIterator<E>는 list만을 위한 iterator용 class임. 기존의 Iterator<E>을 상속받아 list용 기능을 추가한 것.*

java.util에 있음.

List<E>에 있는 listIterator()라는 메소드를 사용하면 해당 list에 대한 ListIterator<E> 객체를 반환받을 수 있고, 이른 기존의 iterator처럼 사용하면 됨.

기존의 iterator는 순방향으로만 순회가 가능했는데, ListIterator<E>에 있는 아래의 메소드들을 사용하면 역방향으로도 list의 순회가 가능함.

add(E e) : list에서 해당 커서 위치에 e를 삽입.
hasPrevious() : 역방향으로의 hasNext().
previous() : 역방향으로의 next().

이때 주의할 점은, add()는 순방향이든 역방향이든 항상 커서 위치 바로 왼쪽에 요소를 삽입함. 커서의 위치는 다음과 같이 생각할 수 있음. 요소 A B C D가 있다면 커서는 ∧ A B C D에 위치하고, next()를 수행하면 A ∧ B C D로 이동함. 이 상태에서 add(Z)를 하면 A Z ∧ B C D와 같이 삽입되는 것. 그래서 순방향으로 순회할 때 add()를 하면 해당 원소를 가져오지 않지만, 역방향으로 순회할 때 add()를 하면 넣자마자 바로 다음 previous()로는 해당 원소를 가져오게 됨.

3.4. List<E>

List<E>는 Vector<E>, ArrayList<E>, LinkedList<E>로 구현함.

3.4.1. Vector<E>, ArrayList<E>, LinkedList<E>

Definition 89 *Vector<E>와 ArraryList<E>는 array 기반 자료구조로, List<E>를 구현함.*

LinkedList<E>는 노드 기반(linked list) 자료구조로, List<E>와 Queue<E>를 구현함.

이 세 class는 거의 동일한 API를 가지는데, 이때 Vector<E>는 동기화되어 있고 ArraryList<E>는 동기화되어 있지 않음.

java.util에 있음.

사용 가능한 메소드들은 아래와 같음. element는 전부 E로, index는 전부 int로 받음.

boolean add(E element) : 맨 뒤에 element 추가.
void add(int index, E element) : index에 element 추가.

E get(int index) : index의 요소 반환.
E remove(int index) : index의 요소 삭제 후 반환.
int capacity() : 현재 용량 반환.
boolean addAll(Collection<? extends E> c) : collection c의 모든 요소를 맨 뒤에 추가.
void clear() : 모든 요소 삭제.
boolean contains(Object o) : 해당 객체가 존재하는지 검사.
E elementAt(int intdex) : index의 요소 반환.
int indexOf(Object o) : 첫 번째로 등장하는 해당 객체의 인덱스 반환. 없으면 -1 반환.
boolean isEmpty() : 비어있는지 검사.
boolean remove(Object o) : 첫 번째로 등장하는 해당 객체 삭제.
void removeAllElements() : 요소 전부 삭제하고 전체 크기를 0으로 함.
int size() : 요소의 개수 반환.
Object[] toArray() : 모든 요소를 담은 배열 반환. <T> T[] toArray(T[] a) : 모든 요소를 담은 배열을 반환. 이때 인자로 지정한 배열을 해당 자료형의 배열로 반환하기 위한 것인데, 런타임에 자료형을 지정하기 위한 특수한 사용법임.

참고로 이때 add()는 기존 원소가 존재하는 위치와 마지막 빈 공간만 index로 지정할 수 있음. 실제로 메모리 공간이 존재해도(capacity) 모든 공간을 인덱스로 지정할 수 있는 것은 아님. 이 경우 잘못된 인덱스 지정이 어떻게 처리될지는 해당 class의 내부에서 어떻게 구현하고 있는지에 달려 있는데, 본 class들의 경우 예외을 던짐.

elementAt()과 get()은 같은 것인데, elementAt()은 Vector<E>를 위해 만들어졌고 get()은 나중에 추가된 List<E>를 위해 만들어졌음.

3.4.2. Arrays.asList()

Definition 90 *java.util에 있는 Arrays class의 asList() static 메소드를 사용하여 list를 생성할 수 있음. 해당 메소드는 아래의 형태를 가짐.*

```
static <T> List<T> asList(T... a);
```

즉, 가변인자를 사용하므로 인자로 값들을 나열해서 작성하면 해당 값들로 list를 생성해 반환함. 또한 해당 메소드가 generic 메소드이므로 인자로 작성한 값들의 타입으로 list가 생성됨.

이때 작성하는 값들은 타입이 동일해야 함. 참고로 asList()는 서로 다른 타입의 데이터를 입력받은 경우 공통된 상위 타입으로 묶는 것으로 보임. 정수, 실수는 함께 작성해도 되던데, Number로 auto-boxing 되서 그런 것 같음.

asList()로 반환받은 list는 size가 고정된 list임. size에 변동을 주고 싶으면 다른 list로 변환해야 함.

요즘에는 사용법이 같은 List.of()를 대신 사용한다고 함.

3.4.3. 가변인자

Definition 91 *메소드의 매개변수를 가변인자로 지정하여 임의의 개수로 인자를 전달받을 수 있음. 아래와 같이 작성함.*

```
print(String... args) // (String[] args)
{
    ...
}
```

이때 내부적으로는 String... args가 String[] args로 처리되고 인자의 값들이 저장됨. 정의 내부에서는 해당 배열을 가지고 작업을 구성함. 아예 인자 작성 시에 배열을 넣어도 정상 동작함.

3.5. Set<E>

Set<E>은 HashSet<E>, TreeSet<E>으로 구현함.

3.5.1. HashSet<E>

Definition 92 *HashSet<E>은 hash table을 사용하여 Set<E>을 구현하는 class임.*

java.util에 있음.

사용 가능한 주요 메소드들은 아래와 같음.

add(E e) : 요소를 삽입.
remove(Object o) : 해당 요소를 찾아 삭제.

1. hash algorithm
HashSet<E>은 hash algorithm을 사용하는데, 이에 사용되는 것이 hash function임. 임의의 함수는 hash function인데, hash function에는 좋은 hash function과 나쁜 hash function이 있음. 좋은 해시 함수일수록 다른 입력값에 대해 가능한 한 다른 출력값이 도출됨. hash algorithm에서는 이렇게 도출된 출력값을 활용해 데이터를 저장함.

Object의 hashCode()는 해당 객체의 reference값을 반환하는데, 이 값이 겹치지는 않으므로 괜찮은 해시임.

2. HashSet()의 원리
HashSet<E>는 내부적으로 hash table을 가지고 데이터를 저장함. set.add(e)를 하면 e.hashCode()값을 modular 연산(hash function 중 하나)을 하여 그 연산 결과값에 따른 칸에 e를 넣음. 이때 연산 결과에 대한 각 칸을 bucket이라고 함. bucket에는 동일한 modular 연산값을 가지는 데이터들이 linked list의 형태로 저장됨. 특정 bucket에 새로운 값이 저장될 때는 해당 linked list의 각 원소들과 equals()연산을 수행하여 삽입 또는 갱신함.

참고로 이에 따라 호출되는 순서는 1. hashCode() 2. equals()임.

3. equals(), hashCode()
HahsSet()는 E로 지정한 reference 타입의 equals()와 hashCode()가 오버라이드 되어 있지 않다면 정상 동작하지 않음.

bucket 내부에서 equals()를 사용해 객체들끼리 비교하므로, equality의 비교를 수행할 수 있도록 오버라이드 해야 함.

Object의 hashCode()를 사용하면 reference값을 기준으로 비교한다는 것인데, 같은 값을 가지면서 다른 reference를 가지는 객체에 대해 modular 연산값(hash function 출력값)이 서로 달라질 수 있음. 같은 값을 가지는데 다른 bucket에 들어간다면 잘못 처리되는 것이므로, hashCode() 값을 필드값을 기준으로 생성하도록 오버라이드 해야 함.

좋은 해시코드를 만드는 것은 어렵기 때문에, java.util.Objects에 있는 Obejcts.hash(x,y)로 좋은 해시코드 값을 얻을 수 있음(Object 아니라 Objects인 것 유의). Objects.hash()는 인자로 작성된 값들을 primitive로 변환하여 값을 생성하고, reference가 입력된 경우 해당 객체의 hashCode()값을 사용하여 값을 생성함. 즉, 필드로 reference 타입이 있는 경우 해당 reference의 class에서도 Objects.hash() 등을 사용해서 hashCode()를 오버라이드해 줘야 함.

참고로 list로 HashSet<E> 객체를 생성하면(생성자에 넣어서) 기존의 순서가 뒤섞여 생성됨. 물론 set은 순서가 없으므로 상관 없음.

null 또한 하나의 요소로 취급함. null이 여러 개 들어오면 하나만 저장함.

3.5.2. TreeSet<E>

Definition 93 *TreeSet<E>은 tree를 사용하여 Set<E>을 구현하는 class임.*

TreeSet<E>은 원소를 오름차순으로 정렬(compareTo()를 사용)하여 저장함.

java.util에 있음.

1. NavigableSet<E>
add(E e), clear(), size() 등의 기본적은 메소드들(Collection<E>에 있는 것들)을 가지고 있고, 추가로 NavigableSet<E>이라는 interface를 구현함. 구현하는 메소드들은 아래와 같음.

E first() : 첫 번째 원소 반환.
E last() : 마지막 원소 반환.
E lower(E e) : 지정한 인자보다 작은 것 중에서 가장 큰 원소 반환.
E higher(E e) : 지정한 인자보다 큰 것 중에서 가장 작은 원소 반환.
E floor(E e) : 지정한 인자보다 작거나 같은 것 중에서 가장 큰 원소 반환.
E ceiling(E e) : 지정한 인자보다 크거나 같은 것 중에서 가장 작은 원소 반환.
E pollFisrt() : 첫 번째 원소 반환하고 삭제.
E pollLast() : 마지막 원소 반환하고 삭제.
Iterator<E> descendingIterator() : 역방향으로 동작하는 iterator 생성.
NavigableSet<E> descendingSet() : 기존의 set으로 역방향 set을 구성하여 반환.
NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive) : from 이상(false이면 초과) to 이하(false이면 미만)의 원소들로 set을 구성하여 반환.

2. Comparable<T>
TreeSet()은 요소 객체의 compareTo()를 사용하여 비교를 수행함. 즉, 요소로 넣는 객체 class에서 Comparable<T> interface를 implements하고 compareTo()를 구현해야 함.

```
int comapareTo(T o)
```

compareTo()는 인자로 지정한 객체보다 해당 객체가 작다면 음수, 같다면 0, 크다면 양수를 반환하도록 구현해야 함.

이때 descendingIterator(), descendingSet()은 읽는 순서만 바꾼 것이지 실제 데이터가 수정되는 것이 아님. 특히 descendingSet()로 set을 반환받았다고 해도 역방향으로 읽을 뿐 동일한 데이터를 가리키고 있음. 실제로 데이터는 Set 객체에 '존재'하는 것이 아니라, 어떤 참조 변수가 가리키도록 되어 있음.

라이브러리 등에서 class를 가져왔는데 내부적으로 비교 작업이 구현되어 있지 않다면, 생성자의 인자로 Comparator<T> 객체를 넣어주는 식으로 동작하게 되어있을 수 있음. 즉, class를 하나 만들어 Comparator를 implements하고 compare()만 구현하여 이 객체를 전달하면 됨. 이때 compare()는 내부적으로 compareTo() 등으로 구성할 수 있음.

3.6. Queue<E>

3.6.1. Queue<E>의 구현

Definition 94 *1. Queue<E>의 구현 Queue<E>는 LinkedList<E>, ArrayBlockingQueue<E>, LinkedBlockingQueue<E> 등으로 구현함.*

LinkedList<E>는 capacity가 자동으로 늘어나고, ArrayBlockingQueue<E>, LinkedBlockingQueue<E>는 생성 시에 capacity가 고정됨. 아래와 같이 생성자에서 지정함.

```
Queue<E> q = ArrayBlockingQueue<>(100);
```

2. Queue<E>의 메소드들
QUeue<E>의 핵심 연산은 insert, remove, examine(peek 연산)이고 아래와 같은 메소드들을 가지고 있음. 예외 상황 처리 방법에 따라 각 연산 별로 두 가지 메소드가 존재함.

예외 상황 발생 시 예외(unchecked)를 던지는 메소드들.
boolean add(E e)
E remove()
E element()

예외 상황 발생 시 특정 값(null 등)을 반환하는 메소드들.
boolean offer(E e)
E poll()
E peek()

3.7. Deque<E>

3.7.1. Deque<E>의 구현

Definition 95 *Deque<E>는 Queue<E> interface를 extends하는 interface임. 즉, queue에서 tail에 대한 연산이 추가된 버전임. 이에 따라 Deque<E>로 stack을 만들 수 있음.*

1. Deque<E>의 구현
Deque<E>는 LinkedList<E>, LinkedBlockingDeque<E>, ArrayDeque<E> 등으로 구현함.

2. Deque<E>의 메소드들
Queue<E>와 거의 동일한 메소드들을 가지는데, 차이점은 아래와 같이 head에 대한 연산은 뒤에 First가 붙고 tail에 대한 연산은 뒤에 Last가 붙는다는 것임.

```
addLast(e)
offerFirst(e)
```

3.8. Map<K,V>

Map<K,V>은 HashMap<K,V>으로 구현함.

3.8.1. HashMap<K,V>

Definition 96 *HashMap<K,V>은 key-value 쌍으로 이뤄진 요소를 다루는 자료구조로, Map<K,V>를 구현함.*

java.util에 있음.

각 요소는 key값에 의해 저장되는 위치가 결정되고, 해당 요소의 검색에 key가 사용됨. Map은 삽입/삭제/탐색에 주 연산임. 당연하게도 key/value 모두 reference임.

사용 가능한 메소드들은 아래와 같음.

V get(Object key) : key에 해당하는 value 반환. 없으면 null 반환.
V put(K key, V value) : key와 value를 map에 삽입.
V remove(Object key) : key에 해당하는 요소 삭제.
void clear() : 모든 요소 삭제
boolean containsKey(Object key) : key가 존재하는지 검사.
boolean containsValue(Object value) : value가 존재하는지 검사.
boolean isEmpty() : 비어 있는지 검사.
Set<K> keySet() : 모든 key를 담은 set을 반환.
int size() : 요소의 개수 반환.

Part IV

기타 문법

1. 기타 문법

1.1. initialization

1.1.1. 정적 초기화 블록

Definition 97 *정적 초기화 블록(static initialization block)은 static 필드의 초기화를 위한 블록임. 특히 초기화에 여러 문장이 필요한 경우 등에 사용함.*

정적 초기화 블록은 static 필드와 같이 해당 class가 최초 사용되었을 때 단 한 번만 수행됨. 즉, non-static 필드를 초기화하는 것은 불가능함(메모리가 존재하지 않으므로 당연함).

아래와 같이 static만 붙인 블록으로 나타냄. 이를 필드 위치에 작성하면 됨.

```
static {
    ... // 초기화
}
```

정적 초기화 블록에 대한 작업 실행 순서는 아래와 같음.

1. 모든 정적 변수들에 대해 메모리 공간을 할당함.
2. 위에서 아래로 내려오면서 값을 배정함. 이때 정적 초기화 블록과 단순 대입 초기화를 모두 적용함.

정적 초기화 블록에서 주의할 점은, 선언된 것보다 위에서 값을 할당하는 코드는 작성될 순 있는데 그 값을 읽는 것은 안됨. 물론 선언 아래에서 읽는 것은 가능함. 그래서 보통은 선언을 위에 몰아서 작성하고 초기화 블록을 가장 아래에 작성함.

1.1.2. 인스턴스 초기화 블록

Definition 98 *인스턴스 초기화 블록(instance initialization block)은 필드의 초기화를 위한 블록임. static 필드와 non-static 필드 모두에 사용 가능함.*

인스턴스 초기화 블록은 해당 class의 객체가 생성될 때마다 수행됨.

아래와 같이 단순 블록으로 나타냄. 이를 필드 위치에 작성하면 됨.

```
{
    ... // 초기화
}
```

인스턴스 초기화 블록에 대한 작업 실행 순서는 아래와 같음.

1. 모든 변수들에 대해 메모리 공간을 할당함.
2. 위에서 아래로 내려오면서 값을 배정함. 이때 인스턴스 초기화 블록과 단순 대입 초기화를 모두 적용함.
3. 생성자를 호출함.

인스턴스 초기화 블록에서도, 선언된 것보다 위에서 값을 할당하는 코드는 작성될 순 있는데 그 값을 읽는 것은 안됨. 물론 선언 아래에서 읽는 것은 가능함.

생성자로 하는 초기화는 주로 생성 시에 외부에서 값을 받아서 그 값으로 초기화할 때 주로 사용하고 default값을 지정하기 위한 초기화는 초기화 블록을 사용함.

1.2. exeption handling

1.2.1. Throwable

Definition 99 *Throwable은 런타임에 발생하는 에러와 예외의 상위 타입임.*

Throwable의 상속 구조에서 Error(에러. 항상 프로그램 종료)와 일부 예외들은 컴파일러에 의해 체크되지 않고, 나머지 예외들은 컴파일러에 의해 체크됨.

1. unchecked exception
unchecked exception은 지금까지 설명한 exception임. 컴파일러는 해당 예외가 발생할 가능성을 검토하지 않고, 런타임에 예외가 발생하면 예외 객체가 해당 위치에 던져짐. 이를 try-catch로 handling할 수 있음. handling하지 않은 경우 프로그램이 종료됨.

unchecked는 모두 RuntimeExeption을 조상으로 가짐.

unchecked의 경우에도 throws를 작성할 수 있는데, 이는 단순히 정보 제공을 위한 것임.

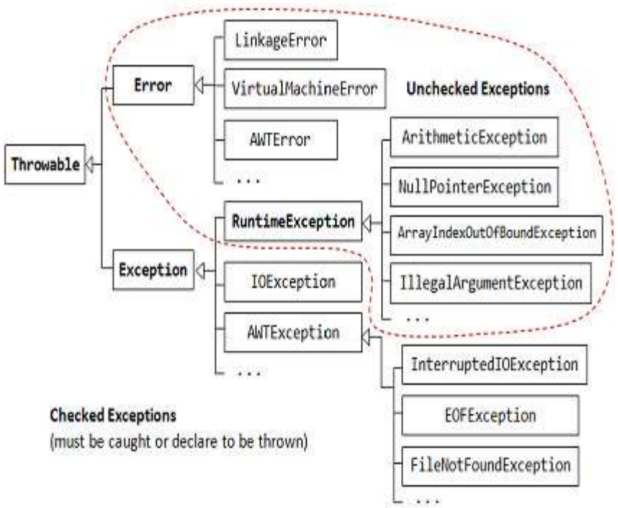
2. checked exception
checked exception은 컴파일러에 의해 체크되는 exception임. 해당 exception이 발생할 것으로 컴파일러가 판단하면 해당 부분에서 컴파일 에러를 냄. 이 경우에 대한 해결 방법은 두 가지가 있음.

1) try-catch로 handling 처리하기.
2) 해당 메소드의 뒤에 throws 추가하기.

1번은 당연하고, 2번은 해당 객체의 처리를 해당 메소드를 호출한 지점에 떠넘기는 것임. 즉, 해당 메소드 호출 지점에서 handling해야 함. 해당 메소드를 호출한 메소드에서도 throws를 작성하면 또 그 메소드를 호출한 지점에서 처리해야 함. main까지 throws를 작성하게 되면 handling을 하지 않은 것이고, 컴파일은 통과하지만 해당 예외 발생 시에 프로그램이 종료됨.

throws는 아래와 같이 작성함. exception 타입은 해당 exception의 상위 타입도 작성이 가능함. 이때 throws로 작성하는 exception의 타입을 ,로 구분하여 여러 개 작성할 수 있는데, Exception 하나로 작성해도 되지만 정보 제공을 위해 이렇게 작성할 수 있음.

```
public class testClass() throws Exception {
    ...
}
```



1.2.2. 에러 메세지 출력

원하는 형식을 사용하거나 디버깅에 이용하기 위해 에러 메세지를 출력할 수 있음. 에러 메세지는 아래와 같이 출력이 가능함. 또한 printStackTrace로 에러 메세지의 throw trace를 출력할 수 있음.

```
String str = e.getMessage();
System.err.println(str);
e.printStackTrace();
```

1.2.3. catch와 다형성

Definition 100 *발생한 예외를 catch로 handling할 때, 예외 객체는 해당 타입의 상위 타입으로 들어갈 수 있음.*

catch가 여러 개이면 위에서 아래로 검사하며 해당 예외 객체가 들어갈 수 있는 타입의 catch가 선택됨. 이때 상위 타입이 위에, 하위 타입이 아래면 항상 위에만 들어가게 되는데, 처리되지 않게 되는 catch

가 존재하면 컴파일 에러가 남.

즉, catch가 여러 개인 경우 하위 타입을 위에, 상위 타입을 아래에 작성해야 함.

1.2.4. 사용자 정의 exception

Definition 101 *Exception을 extends하여 사용자 정의 exception class를 정의할 수 있음.*

아래와 같이 작성하여 예외 메세지를 지정할 수 있음.

```
public class userException extends Exception{
    public userException()
    {
        super("잘못된 입력입니다.");
    }
}
```

이 객체를 throw하도록 코드를 작성할 수 있음.

1.2.5. try-with-resources

Definition 102 *try-with-resources 구문은 try에서 resource(Scanner 등)를 생성하고 해당 try-catch가 끝나면 해당 resource를 해제(close())하는 구문임.*

기존에는 finally로 이를 처리했는데, finally의 쓰임이 많지 않아 별도의 구문을 만든 것.

아래와 같이 작성함. 이때 try()안에는 ;로 구분하여 여러 개의 resource를 작성할 수 있음.

```
try(Scanner sc = new Scanner(System.in))
{
    ...
}
catch(Exception e)
{
    ...
}
// sc.close()가 수행되었음.
```

1.3. object cloning

1.3.1. object cloning

Definition 103 *object cloning는 동일한 값을 가지는 객체를 복사하여 반환하는 것을 말함. Object class의 clone() 메소드로 object cloning을 수행할 수 있음.*

1. clone()
아래와 같이 clone()을 오버라이드하여 사용함. clone()은 CloneNotSupportedException을 던지는데, 이는 checked exception임.

```
@Override
public Object clone() throws CloneNotSupportedException
{
    return super.clone();
}
```

이때 기본적으로 clone()에 의해 반환되는 객체는 Object 타입이기 때문에 캐스트 연산으로 형변환 해야

컴파일 에러가 나지 않는 것이 맞지만, 오버라이드 시에 반환 타입을 다른게 지정할 수 있는 공변 반환 타입이라는 문법이 등장해 해당 타입으로도 반환이 가능함. 당연히 이 경우 캐스트 연산을 할 필요가 없음.

2. Cloneable
Cloneable interface는 멤버를 가지지 않는 interface인데, 해당 class가 object cloning을 지원함을 나타냄. clone()을 사용하려면 Cloneable을 implements해야 하는데, Cloneable을 implements하지 않는다면 clone()이 CloneNotSupportedException을 발생시키도록 되어 있음. 즉, clone()이 정상 동작하지 않음.

3. shallow copy, deep copy
cloning하려는 객체의 class가 reference 필드를 가지는 경우 단순히 return super.clone();만을 작성한다면 reference가 복제되어 동일한 객체를 가리키게 되는데, 당연히 이러면 정상적으로 cloning된 것이 아님. 이런 경우를 shallow copy라고 함.

reference 필드가 존재한다면 해당 필드(객체)에 대해 cloning을 해야 함. 이렇게 구현한 경우를 deep copy라고 함. 아래와 같이 구현할 수 있음.

```
@Override
public Object clone() throws CloneNotSupportedException
{
    Circle c = (Circle)super.clone(); // 객체 cloning
    c.center = (Point)center.clone(); // 해당 객체의 reference 타입 필드 cloning
    return c;
}
```

참고로 clone()은 jni로 구현되어 있는 native 메소드임.

1.3.2. 공변 반환 타입

Definition 104 *공변 반환 타입(covariant return type)은 오버라이드 시에 반환 타입을 더 구체적인 타입으로 지정할 수 있도록 한 문법임. 즉, 오버라이드 시에 기존 반환 타입의 하위 타입으로 반환 타입을 지정할 수 있음.*

예를 들어, B가 A를 상속하는 경우 A print()를 아래와 같이 오버라이드할 수 있음.

```
@Override
B print()   // A로도 지정 가능
{
    ...
}
```

기존에는 오버라이딩 시에 자료형을 맞춰줘야 했어서 clone() 등에서 객체를 반환받을 때 캐스트 연산을 해야 했지만, 이제는 단순히 해당 타입으로 반환하도록 할 수 있음.

1.4. nested class

1.4.1. nested class

Definition 105 *nested class는 class 안에 정의된 class를 말함. nested class는 outer class의 부품으로 기능할 때 사용함. 바깥에 정의할 수도 있지만, 내부에 작성하면 해당 outer class의 멤버에 접근이 용이함.*

nested class도 멤버이므로 4가지 접근지정자 모두 가능함.

nested class를 가지고 있는 바깥쪽 class를 outer class라고 함. nested class로는 static/non-static이 모두 가능한데, non-static nested class는 특별히 inner class라고 함.

즉, 아래와 같음.

```
public class OuterClass{
    static class StaticNestedClass{
        ...
    }
    class InnerClass{
        ...
    }
}
```

nested class의 멤버로는 일반적인 class의 멤버 모두가 가능함.

1. 객체 생성 방법
객체를 생성하여 참조를 얻었다면 그때부터는 일반적인 class와 동일하게 사용이 가능함.

외부의 class에서 static/non-static의 객체 생성은 아래와 같이 함. OuterClass 내부에서 생성하는 경우에는 OuterClass.등의 작성 대신 일반적인 class처럼 생성이 가능함.

```
// non-static
OuterClass oc = new OuterClass();
OuterClass.InnerClass ic = oc.new InnerClass();

// static
OuterClass.StaticNestedClass snc = new OuterClass.StaticNestedClass();
```

2. this
OuterClass에서 nested class 객체 생성 시에는 static에서의 생성을 유의해야 함. static 멤버에는 this가 들어갈 수 없음. new InnerClass();에는 this.이 생략된 것이므로 static에 들어갈 수 없음.

또한 당연하게도 OuterClass의 non-static 멤버를 StaticNestedClass에서 사용할 수 없음.

3. nested class에서 outer class의 멤버 사용하기
기본적으로 nested class에서는 outer class의 멤버를 사용할 수 있지만, 식별자가 겹치는 경우에는 아래와 같이 작성하여 사용할 수 있음. Innerclass의 경우 단순히 this만 작성하면 해당 class에 대한 것이 되므로 outer class의 식별자를 작성해야 함.

식별자가 겹치는데 단순히 사용하면 내부의 것이 사용됨. this.가 생략된 것이므로 당연함.

```
// InnerClass에서 사용하는 경우
OuterClass.this.instanceField = 10;
OuterClass.this.instanceMethod();

// StaticNestedClass에서 사용하는 경우
OuterClass.staticField = 10;
OuterClass.staticMethod();
```

참고로, outer class에서는 inner class의 private 멤버에도 접근이 가능함.

1.4.2. local class

Definition 106 *메소드 안에 정의된 class를 지역 클래스(local class)라고 함.*

아래와 같이 단순히 메소드 안에 정의할 수 있음.

```
void method()
{
    class LocalClass{
        ...
    }
```

```
}
```

local class는 지역변수와 같이 접근지정자나 static 등을 붙일 수 없음. 또한 지역변수와 같은 사용 범위를 가짐.

local class의 멤버는 일반적인 class의 멤버로 가능한 것 모두가 가능함.

1. 객체 생성 방법
여기서도 참조만 얻으면 일반적인 class와 동일하게 사용이 가능함. 단순히 new로 참조를 얻을 수 있는데, 해당 메소드 내에서만 얻을 수 있음.

물론 얻은 참조를 외부로 전달하여 외부에서 사용할 수 있음.

2. 메소드가 속한 class의 멤버 사용
nested class가 outer class의 멤버를 사용하는 것처럼, local class 내에서도 아래와 같이 메소드가 속한 class의 멤버를 사용할 수 있음.

```
AClass.this.instanceField = 10;
AClass.this.instanceMethod();
```

3. 메소드의 지역변수 사용
local class에서는 해당 메소드의 지역변수에도 접근이 가능한데, class 정의 위쪽에 존재하는 지역변수만 접근이 가능함.

단, 이때 지역변수 값을 읽을 수는 있지만 값을 배정할 수는 없음. 지역변수는 스택, 객체는 힙에 있는ㄷ, 스택은 반환 시에 메모리 해제, 힙은 가비지로 처리됨. 힙의 클래스가 지역변수 값을 배정한다는 것은 사라진 스택의 메모리공간을 객체가 수정하려고 한다는 것임. local class에서 지역변수를 사용하는 것은 해당 클래스 위의 값이 배정된 지역변수들을 final처럼 취급하여 가져와 놓은 것임. 애초에 초기 java에는 final만 접근이 가능했음.

정리하면, class 정의 위에 존재하고 전체 코드에서 값이 단 한 번만 배정되는 것만이 사용될 수 있음. 이런 지역변수를 effectively constant라고 함.

1.4.3. anonymous class

Definition 107 *익명 class(anonymous class)는 class 식별자가 없는 local class임.*

local class의 경우 객체 생성을 메소드 내부에서 하지만 이 참조를 외부로 전달하여 사용한다면 메소드 내부에서는 굳이 식별자를 작성할 필요가 없는데, 이런 경우를 위해 anonymous class가 존재함.

아래와 같이 작성함. 즉, 해당 부모 타입을 extends하는 class의 정의를 작성하고, 해당 객체의 참조를 반환하는 문법임. 부모타입()의 괄호 안에 값을 작성하면 해당 값으로 부모타입의 생성자를 호출함.

```
new 부모타입() {
    자식타입 정의
}
```

부모타입으로 interface 등을 작성하기도 함.

즉, 아래와 같이 작성이 가능함.

```
interface Interface{
    void method();
}

public class ExClass{
    void print(Interface i) { i.method(); }
```

```
    public static void main(String[] args)
    {
        print(new Interface(){
            @Override
            public void method()
            {
                // override
            }
        });
    }
}
```