JAVA 심화

복습: 상속과 다형성

- 다형성

: 부모 클래스의 참조 변수로 자식 클래스 객체를 참조할 수 있다.

→ 실제 부모 영역만 접근 가능

예) Person p = new Student (”김소현”, 29, “국어”);

- instanceof: person 변수가 현재 참조하고 있는 실제 객체가 Student 타입으로 참조 가능한지?

가. 추상 클래스

1. 정의

- 상속 전용 클래스
- 클래스에 구현부가 없는 메소드가 있으므로 객체 생성 X
- 상위 클래스 타입으로 자식 참조 가능
- 조상 클래스에서 상속받은 absrract 메서드 재정의하지 않는 경우,

  자식 클래스는 abstract 클래스로 선언

나. 인터페이스

: 추상메소드의 모임

- 완벽히 추상화된 객체: 모든 메소드가 abstract
- 반쯤 완성된 객체 → 직접 구현 필요하기 때문

(1) interface 키워드 이용하여 선언

예) public interface MyInterface { };

(2) 선언되는 변수는 모두 상수로 적용 (final 없이도)

(3) 선언되는 메소드는 모두 추상 메소드로 적용 (abstract 없이도)

(4) 객체 생성 불가능 (추상 클래스라서) → 타입으로서는 동작 가능

(5) 클래스가 인터페이스 상속: implements 키워드 이용 (extends X)

(6) 인터페이스를 상속받는 하위클래스는 추상 메소드를 반드시 오버라이딩(재정의) 해야 한다

(7) 인터페이스 다형성 적용

- 인터페이스의 필요성
  - 구현의 강제로 표준화 처리 (abstract 메서드 사용)
  - 인터페이스를 통한 간접적 클래스 사용으로 손쉬운 모듈 교체 지원
  - 서로 상속 관계 없는 클래스들에게 인터페이스를 통한 관계 부여로 다형성 확장
  - 모듈 간 독립적 프로그래밍 가능 → 개발 기간 단축

보충

method [접근 제어자] [abstract, static] (자료형) 이름, 함수명, 메소드명([매개변수]) {

}　　반환형, 리턴타입　　인자, 파라미터

void → 반환할 게 없을 때

- private : 자기 자신
- (default) : 자기 자신 / 같은 패키지
- protected: 자기 자신 / 같은 패키지 / 상속 관계일 경우에는 다른 패키지 O
- public: 아무나

- 메서드명: 작명할 때 관련 있는 이름으로 짓기

→ 카멜 케이스 : 띄어쓰기마다 대문자 (cf) 파스칼 케이스: 첫 글자부터 대문자)

- 생성자
- 인스턴스

다. 예외 처리

1. 에러와 예외

- 어떤 원인 때문에 오동작하거나 비정상적 종료
- 심각도에 따른 분류
  - Error
    - 메모리 부족, stack overflow처럼 발생하면 복구할 수 없음
  - Exception

2. 예외처리란?

- 예외 발생 시 프로그램 비정상 종료를 막고 정상적인 실행 상태 유지
- 예외 감지 및 예외 발생 시 동작할 코드 작성 필요

3. 예외 클래스 계층

- Checked exception → SQLEception, IOException, FileNotFOundException
  - 예외 대한 대처 코드 없으면 컴파일 진행 X
- Unchecked exception (RuntimeException의 하위 클래스) → RuntimeException, ArithmethicException

  (실행 전에 빨간 줄로 안 알려 줌)
  - 예외에 대한 대처 코드 없더라도 컴파일 진행

4. 예외 처리 키워드

   1) 직접 처리
   - try
   - catch
   - finally

   2) 간접 처리 (대신 처리해 달라고 던짐)
   - throws

   3) 사용자 정의 예외 처리

- throw

<try~catch 구문>

try {

// 예외가 발생할 수 있는 코드

} catch (Exception e) {

// 예외가 발생했을 때 처리할 코드

}

- try 블록 예외

5. Exception 객체의 정보 활용

6. 다중 exception handling

- try 블록에서 여러 종류 예외 발생, 하나의 try 블록에 여러 가지 catch 구문 작성 가능
    - 예외 종류별로 catch 블록 구성

<try ~ catch ~ finally> 구문

- finally 이용한 자원 정리
- 해당 코드는 이렇게 작성하면 어떨까?

8. throws 통한 처리 위임

- method에서 처리해야 할 하나 이상 예외를 호출한 곳으로 전달 (처리 위임)
- 예외가 없어지는 것이 아니라 단순 전달
- 예외 전달받은 메서드는 다시 예외 처리 책임 발생

→ checked 예외만 가능

예) void exceptionMethod() throws Exception 1, Exception 2...

- 처리하려는 예외 조상 타입으로 throws 처리 가능

- checked exception은 반드시 try ~ catch 또는 throws 필요

9. 사용자 정의 예외

- API 정의된 exception 외에

상황 전달의 장점

10. 로그 분석과 예외 추적

- Throwable의 printStrackTrace는 메서드 호출 스택 정보 조회 가능
  - 최초 호출 메서드에서부터 예외 발생 메서드까지~

11. 메서드 재정의와 throws

- 메서드 재정의 시 조상클래스 메서드가 던지는 예외보다 부모 예외를 던질 수 없다.

<싱글턴 패턴>

1. 내 타입 변수를 static으로 갖는다 (객체 생성)
2. 생성자를 private를 갖는다
3. 1에 대한 getter 생성한다

라. Generic

: 다양한 타입의 객체를 다루는 메서드, 컬렉션 클래스에서 컴파일 시 타입 체크

- 객체 만들 때, 사용할 타입을 특정 키워드로 미리 명시해서 형 변환 하지 않아도 되게 함
- 객체 타입에 대한 안정성 향상 및 형 변환 번거로움 감소

cf) A instanceof B → A가 B로 만들어졌는지?

1. 표현

- 클래스 또는 인터페이스 선언 시 < >에 타입 파라미터 표시

예) public class Class Name<T>{ }

- T: reference Type ★ (많이 씀)
- E: Element
- K: Key
- V: Value

2. 객체 생성

- 변수 쪽과 생성 쪽의 타입은 반드시 같아야 함

Class_Name<String> generic = new Class_Name<String>();

Class_Name<String> generic 2 = new Class_Name<>();

i → integer (wrapper class)

예) class GenericBox <T> {

private T some; T ← 정수, 문자열, 객체

public T getSome () {

  return some;

)

public void setSome(T some) {

some = this.some;

3. Type parameter 제한

- 필요에 따라 구체적 타입 제한 필요
- 계산기 프로그램 구현 시 Number 이하의 타입

4. Generic Type 객체 할당받을 때 와일드 카드처럼 이용

- 구체적 타입 대신 이용

Generic type<?> 타입에 재한 없음

Generic type<? extends T> T or T 상속받은 타입만 사용 가능

Generic type<? super T> T or T 조상 타입만 사용 가능

5. Generic Method

- 파라미터와 리턴 타입으로 type parameter를 갖는 메서드
- 메서드 리턴 타입 앞에 타입 파라미터 변수 선언

마. Collection Framework

: 객체들을 한곳에 모아 놓고 편리하게 사용할 수 있는 환경 제공 틀

- 정적 자료 구조
  - 고정된 크기의 자료 구조

  예) 배열
  - 선언 시 크기를 명시하면 바꿀 수 없음
- 동적 자료 구조
  - 요소 개수에 따라 자료 구조 크기가 동적으로 증가하거나 감소

  예) 리스트, 스택, 큐

1. 자료 구조 : 어떤 구조에서 얼마나 빨리 데이터 찾는가

- 순서 유지? 중복 허용? 어떤 단점과 장점?

2. java.util 패키지

cf) comparable: 나와 다른 것 비교 / comparator: 두 개 비교

- 핵심 인터페이스

(1) List : 순서 있는 데이터의 집합, **순서가 있어서** 데이터 **중복, 정렬 허락**, 기준으로 비교.

예) 일렬로 줄 서기→ ArrayList, LinkedList

(2) Set: 순서 없는 데이터의 집합. 같은 데이터 구별 불가. → **중복 X**

예) 알파벳이 한 종류씩 있는 주머니 → HashSet, TreeSet

(3) Map: key와 value 쌍으로 데이터 관리. 순서 X, key는 중복 X, value는 중복 O.

예) 속성 - 값, 지역 번호 - 지역 → HashMap, TreeMap

3. List : 순서 있고, 중복 허용(배열 유사)

예) ArrayList, LInkedList

- 내부적으로 배열 이용해 데이터 관리, 배열과 유사하게 사용 가능
- 배열과 다르게 크기가 유동적으로 변함 (동적 자료 구조)

예) add, get, remove, set, subList

4. 배열과 ArrayList

- 배열의 장점
  - 가장 기본적 형태 자료 구조, 간단하고 사용 쉬움
  - 접근 속도 빠름
- 배열의 단점
  - 크기 변경할 수 없어 추가 데이터 위해 새로운 배열 만들고 복사해야 함
  - 비 순차적 데이터의 추가, 삭제에 많은 시간 걸림
- 종류
  - add(E e): 데이터 입력

5. LinkedList

- 각 요소를 Node로 정의, Node는 다음 요소 참조 값과 데이터로 구성
- 각 요소가 다음 요소의 링크 정보를 가지며 연속적으로 구성될 필요 없음

: 삽입 / 삭제 시 연결 순서 중요

6. Set

- 순서 없고, 중복 허용 X
- 장점: 빠른 속도, 효율적인 중복 데이터 제거 수단

- 단점: 단순 집합 개념으로 정렬하려면 별도의 처리가 필요
- 구현 클래스: HashSet, TreeSet

바. I/O와 Stream

1. I/O : 데이터 입력(input). 출력(output)

- 데이터는 한쪽에서 주고 한쪽에서 받는 구조
- 입출력 끝단: 노드(Node).
- 두 노드 연결하고 데이터 전송: 스트림(Stream)
- 스트림은 단방향 통신 가능, 하나의 스트림으로 입출력 같이 처리 X (양방향 X)

	byte	Char
입력	InputStream	Reader
출력	OutputStream	Writer

2. InputStream의 주요 메서드

- read()
  - int read() : byte 하나를 읽어 int로 반환. 읽을 값 없으면 -1 리턴.
  - int read(byte b[]): 데이터를 읽어 b를 채우고 읽은 바이트 개수 리턴.

    0이 리턴되면 더 이상 읽을 값 없음.
  - int read(byte b[], int offseet, int lne): 최대 len만큼 데이터를 읽어 b의 offset부터 b에 저장하고, 읽은 바이트 개수 리턴. len+offset은 b의 크기 이하.
- close(): 스트림을 종료해 자원 반납.

4. OutputStream의 주요 메서드

- write()
  - write(int b): b의 내용 byte로 출력

    - write(byte b[]): b를 byte로 변환하여 출력 → 무조건 1024로 읽어 파일 용량 차지, 밑의 거로
    - wirte(byte b[], int off, int len): b의 off부터 off+len-1만큼을 byte로 출력
- close()

5. FileInputStream, FileOutputStream

(1) 파일 복사하기

(2) 문자열 버퍼로 읽기

(3) 객체 저장 / 불러오기

6. 보조 스트림: FIlter Stream, Processing Stream

- 종류

byte 스트림을 char 스트림으로 변환