1. Java Reflection

-리플렉션은 구체적인 클래스 타입을 알지 못해도 그 클래스의 메소드와 타입 그리고 변수들을 등에 동적으로 접근 할수 있도록 해주는 api

- 동적으로 클래스를 사용해야 할 때 유용한 api (스프링 프레임워크에서도 많이 사용)
- 클래스타입, 메소드, 생성자 등을 조회해서 사용 가능
  ClassName, Class Modifiers, Package info, Superclass, Implemented interfaces, Constructiors, Methods fields, Annotations
- Ex) Class c = Data.class; // 클래스에서 클래스 타입 조회
  Class c= class.forName("이름") //클래스 이름에서 클래스타입 조회
  Method[] m = c. getMethods(); // 메소드 가져오기
  Field[] f = c.getFields(); // 변수 가져오기
  Constructor[] cs = c.getConstructors(); // 생성자 가져오기
  Class[] inter = c.getInterfaces(); // 인터페이스
  Class superClass = c.getSuperClass(); //부모 클래스

리플렉션 활용

-새로운 객체도 만들 수 있고, 필드 값도 수정 가능

Ex) 새로운 객체 만들기

```
 import java.lang.reflect.*;
public class constructor2 {
        public constructor2() {
        }
        public constructor2(int a, int b) {
                System.out.println("a = " + a + " b = " + b);
        }
        public static void main(String args[]) {
                try {
                        Class cls = Class.forName("constructor2");
                        Class partypes[] = new Class[2];
                        partypes[0] = Integer.TYPE;
                        partypes[1] = Integer.TYPE;
                        Constructor ct = cls.getConstructor(partypes);
                        Object arglist[] = new Object[2];
                        arglist[0] = new Integer(37);
                        arglist[1] = new Integer(47);
                        Object retobj = ct.newInstance(arglist);
                } catch (Throwable e) {
                        System.err.println(e);
```

```
                }
        }
}
```

필드값 바꾸기

```
 import java.lang.reflect.*;
public class field2 {
        public double d;
        public static void main(String args[]) {
                try {
                        Class cls = Class.forName("field2");
                        Field fld = cls.getField("d");
                        field2 f2obj = new field2();
                        System.out.println("d = " + f2obj.d);
                        fld.setDouble(f2obj, 12.34);
                        System.out.println("d = " + f2obj.d);
                } catch (Throwable e) {
                        System.err.println(e);
                }
        }
}
```

2. Java Stream

- 자바에서 많은 양의 데이터를 저장하기 위해 주로 배열이나 컬렉션을 사용
- 하지만 이 데이터에 접근하기 위해서는 반복문이나 반복자를 사용하여 매번 새로운 코드를 작성해야함
- 코드의 재사용도 거의 불가능하고 데이터마다 다른 방법으로 접근해야하는 불편함이 있음
- 이러한 문제점을 해결하기 위해 스트림 api를 도입
- 스트림 api는 데이터를 추상화하여 다뤄, 다양한 방식으로 저장된 데이터를 읽고 쓰기 위한 공통된 방법 제공

Stream의 특징

- 내부 반복을 통해 작업을 수행
- 재사용이 가능한 컬레션과 달리 단 한번만 사용 가능
- 원본 데이터 변경하지 않음
- 손쉬운 병렬 처리를 지원

Stream의 동작 흐름

- 스트림 생성 -> 중개 연산 (스트림 변환) -> 최종 연산 (스트림 사용)

Stream 생성

- 컬렉션, 배열 , 가변 매개변수, 파일 , 난수 다양한 데이터 소스에서 생성 가능
- Ex) 컬렉션

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(4);
list.add(2);
list.add(3);
list.add(1);
// 컬렉션에서 스트림 생성
Stream<Integer> stream = list.stream();
// forEach() 메소드를 이용한 스트림 요소의 순차 접근
stream.forEach(System.out::println);
```

Stream의 중개연산

- 스트림을 전달받아 스트림을 반환하므로, 중개 연산은 연속으로 연결해서 사용 가능
- 스트림의 중개 연산은 필터맵 기반의 api를 사용함으로 지연 연산을 통해 성능 최적화 가능
- 대표적인 중개 연산과 그에 따른 메소드
  1. 스트림 필터링: filter(), distinct()
  2. 스트림 변환: map(), flatMap()
  3. 스트림 제한: limit(),skip()
  4. 스트림 정렬: sorted()
  5. 스트림 연산 결과 확인: peek()
- Ex)

```
Stream<String> stream1 = Stream.of("JAVA", "HTML", "JAVASCRIPT", "CSS");
Stream<String> stream2 = Stream.of("JAVA", "HTML", "JAVASCRIPT", "CSS");
stream1.sorted().forEach(s -> System.out.print(s + " "));
System.out.println();
stream2.sorted(Comparator.reverseOrder()).forEach(s -> System.out.print(s + " “))
```

Stream의 최종연산

- 중개 연산을 통해 변환된 스트림은 최종 연산을 통해 각 요소를 소모하여 결과를 표시
- 최종 연산시에 모든 요소를 소모한 해당 스트림은 더는 사용할 수 없게 됨
- 대표적인 최종 연산과 그에 따른 메소드
  1. 요소의 출력: forEach()
  2. 요소의 소모: reduce()
  3. 요소의 검색: findFirst(), findAny()
  4. 요소의 검사 : anyMatch(), allMatch(), noneMatch()
  5. 요소의 통계 : count(), min(), max()

6. 요소의 연산 : sum(), average()

7. 요소의 수집 : collect()

- Ex)

```
Stream<String> stream = Stream.of("넷", "둘", "셋", "하나");
stream.forEach(System.out::println);
```

3. Java Optional

- Optional <T> 클래스는 integer 나 double 클래스처럼 'T' 타입의 객체를 포장해주는 래퍼 클래스
- T 타입에는 string, int, long, double 다양한 데이터 타입과, 클래스 가능
- Optional 객체를 사용하면 예상치 못한 NullPointerException 예외를 제공되는 메소드 회피할 수 있음

Optional 객체의 생성

- of() 메소드나 ofNullable() 메소드를 사용하여 Optional 객체를 생성
- of() 메소드는 null이 아닌 명시된 값을 가지는 Optional 객체 반환
  null이 저장되면 NullPointerException 예외 발생
- Null 가능성이 있다면, ofNullable() 메소드 사용하여 optional 객체를 생성하는 것이 좋음
- Ex) Optional<String> opt =Optional.ofNullable("자바 Optional 객체");
  System.out.println(opt.get());

Optional 객체에 접근

-get() 메소드 사용하면 Optional 객체에 저장된 값에 접근할 수 있음.

만약 Optional 객체에 저장된 값이 null이면, NoSuchElementException 예외 발생

- get() 메소드 호출전에 isPresent() 메소드를 사용하여 null인지 아닌지 확인 후 호출 하는 것이 좋음
- Ex) Optional<String> opt = Optional.ofNullable("test");

```
if(opt.isPresent()){
  System.out.println(opt.get());
}
```

- 메소드 종류

1. orElse(): 저장된 값이 존재하면 그 값 반환, 존재하지 않으면 인수로 전달된 값 반환함.
2. orElseGet(): 저장된 값이 존재하면 그 값 반환, 존재하지 않으면 전달된 람다 표현식 결과값 반환
3. orElseThrow(): 저장된 값 존재하면 그 값 반환, 값 존재하지 않으면 인수로 전달된 예외 발생

Ex) Optional<String> opt = Optional.empty();
System.out.println(opt.orElse("empty"));

기본타입의 Optional 클래스

1. OptionalInt => int getAsInt();
2. OptionalLong => long getAsLong();
3. OptionalDouble => double getAsDouble(); d