Lambda, Stream, Optional

514770
2018년 가을학기
10/8/2018
박경신

자바7 vs 자바8

- 자바8(2014년)부터 람다식(lambda expression), 스트림(stream), 옵셔널(Optional), 이중 콜론 연산자(::), JavaFX8 지원한다.
- **람다식**은 익명함수 (anonymous function)를 간결하게 작성할 수 있다.
- **스트림**은 간결하게 컬랙션의 데이터를 처리하는 기능을 제공한다. Parallel Stream은 Steam을 여러 쓰레드에서 병렬처리할 수 있도록 분할한 스트림이다.
- **옵셔널**은 값을 Optional<T>로 캡슐화하여 NullPointerException을 막는다.
- **:: 연산자 (Double colon operator)**는 람다식을 대체하여 메서드 참조로 사용된다.

람다 (lambda)

- 함수형 인터페이스 (functional interface)
- 람다식 (lambda expression)

  (인자 목록) -> { 구문 블록 }
  또는
  (인자 목록) -> 식

- 람다식 = 익명 메서드 (anonymous method)
- 대상 타입 추론 (target type inference)
- 매개변수 타입 추론 (parameter type inference)

함수형 인터페이스 (Functional Interface)

- **함수형 인터페이스 (Functional Interface)**
  - **(Object 클래스의 메소드를 제외하고) 단 하나의 추상 메서드만 가진 인터페이스, 그런 이유로 단 하나의 기능적 계약을 표상**
- Java8에서 함수형 인터페이스 예시로는 **Runnable, Callable, Comparator** 등 수 많은 인터페이스 등
- 람다가 해당 함수형 인터페이스의 어떤 메서드를 정의하고 있는지에 대한 모호함이 없기 때문에 문법이 간결해 질 수 있는 이유가 됨
- Java8은 @FunctionalInterface을 제공함
  - @FunctionalInterface는 작성한 인터페이스가 Functional Interface임을 확실히 하기 원할 때 사용하면 됨
  - 문서화시 어떤 인터페이스가 람다와 함께 사용되어지도록 설계되었다라는 것을 알려줌

함수형 인터페이스 (Functional Interface)

- Functional Interface인 경우: 추상 메소드가 하나만 있음

```
public interface Functional {
    public void execute(); // 추상메소드
}
// java.lang.Runnable 도 결과적으로 Functional Interface임
public interface Runnable {
    public void run(); // 추상메소드
}
```

- Functional Interface가 아닌 경우

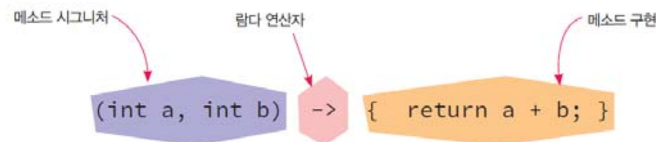
```
public interface NonFunctional {
    public void actionA(); // 추상메소드
    public void actionB(); // 추상메소드
}
```

람다식

- 람다식(lambda expression)은 나중에 실행될 목적으로 다른 곳에 전달될 수 있는 코드 블록이다.
- 람다식을 이용하면 메소드가 필요한 곳에 간단히 메소드를 보낼 수 있다.
- **람다식이 평가(evaluation)되면 그 결과 Functional Interface의 인스턴스를 생성한다.** 람다식의 처리 결과는 표현식 몸통을 실행하는 것이 아니다. 대신 나중에 이 Functional Interface의 적절한 메소드가 실제 호출(invoke)될 때 이 표현식 몸통의 실행이 일어난다.

람다식의 구문

- 람다식은 **(argument-list) -> {body}** 구문 사용하여 작성



- 람다식의 예
  - **() -> System.out.println("Hello World");**
  - **(String s) -> { System.out.println(s); }**
  - **() -> 69**
  - **() -> { return 3.141592; };**
  - **(String s) -> { return “Hello, ” + s; };**
  - **(int n, String str) -> { return str + n; }**
  - **(n, str) -> str + n**
  - **str -> str + 1**

람다식 사용 예

```
@FunctionalInterface
public interface ArithmeticOperator {
    public int operate(int a, int b);
}
public class ArithmeticCalculator {
    public static int calculate(int a, int b, ArithmeticOperator op) {
        return op.operate(a, b);
    }
}
public class ArithmeticCalculatorTest {
    public static void main(String[] args) {
        int result = ArithmeticCalculator.calculate(5, 10, (a, b) -> a + b) ;
        result = ArithmeticCalculator.calculate(5, 10, (a, b) -> a - b) ;
        result = ArithmeticCalculator.calculate(5, 10, (a, b) -> a * b) ;
        result = ArithmeticCalculator.calculate(5, 10, (a, b) -> a / b) ;
    }
}
```

대상 타입 추론 (Target Type Inference)

- 익명클래스를 사용한 방식

```
new Thread( new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello World");
    }
}).start();
```

Target type

```
public Thread (Runnable target)

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

- 람다식를 사용한 방식

```
new Thread( () -> {
    System.out.println("Hello World");
  }
).start();
```

매개변수 타입 추론 (Parameter Type Inference)

- 매개변수 타입 사용

```
button1.addActionListener( (ActionEvent e) -> {
      textfield1.setText("버튼 클릭");
   }
);
```

Target type

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

Parameter type

- 매개변수 타입 추론

```
button1.addActionListener( (e) -> {
      textfield1.setText("버튼 클릭");
   }
);
```

Why lambda?

- 제어흐름 중복과 추상화
- OOP 방식 : OOP is not a one-size-fits-all solution
- 행위 매개변수(parameterized behaviors)를 사용한 제어흐름 추상화
- 가장 비용이 적게 드는 익명 클래스(anonymous class)
- 변수 캡쳐(captured variable)
- 행위 매개변수의 대중화
- 자바8 결국 람다식 지원

제어 흐름 중복

NaturalNumMaxFinder

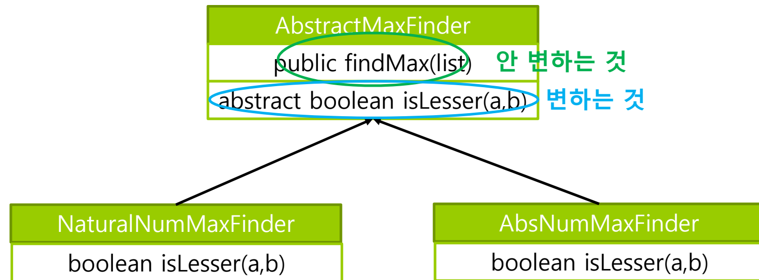
```
int max = numbers[0];
for (int i=1; i<numbers.length; i++) {
    if (max < numbers[i])
        max = numbers[i];
}
return max;
```

AbsNumMaxFinder

```
int max = numbers[0];
for (int i=1; i<numbers.length; i++) {
    if (Math.abs(max) < Math.abs(numbers[i]))
        max = numbers[i];
}
return max;
```

Template Method Design Pattern

- 템플렛 메소드 디자인 패턴 (**상속을 통한 제어흐름 추상화**)
  - 템플렛 메소드에서 알고리즘의 골격을 정의한다.
  - 템플렛 메소드를 이용하면 알고리즘의 구조는 그대로 유지하면서 서브클래스에서 특정 단계를 재정의할 수 있다



Template Method Design Pattern

```java
abstract class AbstractClass
{
    public final void templateMethod()
    {
        operation1();
        operation2();
        operation3();
    }
    abstract void operation1();
    abstract void operation2();
    void operation3()
    {
        System.out.println("General operation,will be common for all subclasses");
    }
}

public class ConcreteClass extends AbstractClass{

    @Override
    void operation1() {
        System.out.println("Implementing abstract method operation1");
    }
    @Override
    void operation2() {
        System.out.println("Implementing abstract method operation2");
    }
}
```

Template method,It is final so that you can not override it

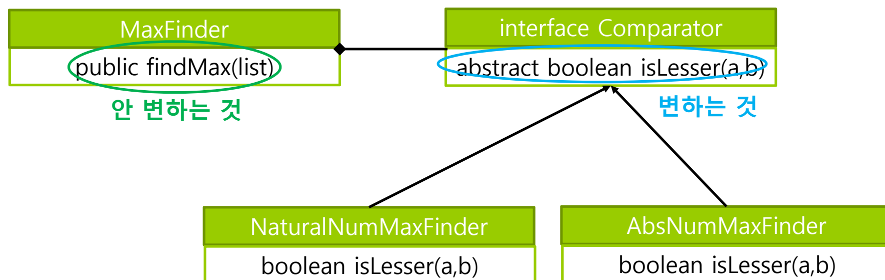
Abstract steps which will be implemented by subclass

This step is common for all so implmenting in base class

Subclass implementing abtract steps of superclass

Strategy Design Pattern

- 전략 디자인 패턴 (**위임을 통한 제어흐름 추상화**)
  - 알고리즘군을 정의하고 각각을 캡슐화하여 교환해서 사용할 수 있도록 한다.
  - 스트래터지를 활용하면 알고리즘을 사용하는 클라이언트와는 독립적으로 알고리즘을 변경할 수 있다.



행위 매개변수 (Parameterized Behaviors)

- 행위 매개변수
  - 메서드 호출시 값을 인자로 전달하듯이 행위를 전달
  - 런타임에 행위를 전달받아서 미리 정해진 제어 흐름에 따라 수행
  - 객체가 아닌 **메서드(함수) 수준의 제어 흐름 추상화**
  - 객체를 사용하는 방법보다 경량
  - **함수형의 고차함수(high order function)**
    - 고차함수는 다른 함수를 인자로 받거나 또는 함수를 결과로 반환한다.
    - Scheme, Haskell, Scala, F# 함수형 언어는 모두 일급함수(functions are first-class citizens)를 사용. Perl, Python, Lua, JavaScript 등 많은 스크립팅 언어도 first-class function를 사용. 일급함수는 고차함수이다.
    - C/C++나 C# 언어는 함수가 first-class가 아니어도, function pointer나 delegate를 통하여 고차함수를 작성 가능하다.
    - 자바에는 함수의 개념이 없다. 자바의 메소드는 일급함수가 아니므로, 다른 메소드로 전달할 수 없다.
    - 자바8에서 함수형 인터페이스 정의와 람다식 표현을 통하여, 함수형 인터페이스 메소드에서 또 다른 함수형 인터페이스를 인자로 받을 수 있도록 하여 고차함수를 정의할 수 있게 되었다.

자바의 행위 매개변수

- java.util.Collections 클래스
  - **public static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp)**
  - **public static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp)**
  - **public static <T> void sort(List<T> list, Comparator<? super T> comp)**
- 자바 객체 사용에 따른 비용
  - 모든 코드는 class 안에 정의되어야 함 (함수만 인자로 사용 불가)
  - 모든 자바 소스 파일에 공개 클래스(public class)나 인터페이스 하나
  - 클래스 이름 작명 필요
  - 캡슐화, 간접 접근

익명 클래스를 사용한 행위 매개변수

- 익명 클래스를 사용한 행위 매개변수

```
List<Person> people = getPersonList();
plist.sort(new Comparator<Person>() {
  @Override
  public int compare(Person p1, Person p2) {
    return p1.getAge() - p2.getAge();
  }
});
```

- 익명 클래스 (anonymous class)
  - 가장 비용이 적게 드는 클래스
  - 클래스 정의와 인스턴스 생성이 동시에 이루어짐
  - 클래스 이름 작명 불필요
  - 클래스 생성자 작성 불필요
  - 일회용 클래스, 즉 클래스 하나에 인스턴스 하나

익명 클래스에서 변수 캡쳐

- 익명 클래스에서 변수 캡쳐 (captured variable)
  - 내부 객체는 외부 객체의 멤버와 외곽 범주(scope)의 변수 값에 접근 가능
  - 내부 객체가 접근할 변수는 **final** 수정 기호로 고정되어야 함
- **익명 클래스 + 변수 캡쳐 = 자바식 클로저(closure)**

```
Button button1 = new Button("press the button");
final TextField textfield1 = new TextField(20);
final Counter counter1 = new Counter();
button1.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    counter1.increse();
    textfield1.setText("button pressed: " + counter.count());
  }
});
```

자유변수 (free variable)란 클로저에서 접근하는 함수 밖의 변수

행위 매개변수의 보급

- 다양한 언어에서 고차함수(high-order function) 지원
- 함수형 프로그래밍 대중화
- C# 3.0 람다식 & LINQ 지원 (2007)
- 자바7까지에서 컬랙션 처리(filter, sort, map 등)를 위한 장황한 코드에 대한 반발로 나타난 각종 라이브러리: Guava, TotallyLazy, Op4j, Lambdaj, 등등
- JVM에서 실행되는 다른 언어의 익명함수 지원: Groovy, Scala, Kotlin, 등등
- **자바에도 람다식 지원 필요 급증으로, 자바8부터 람다 도입 (2014)**

Anonymous Class vs Lambda

- 익명클래스를 사용한 방식

```
Comparator<Person> byName = new Comparator<Person>() {
   @Override
   public int compare(Person o1, Person o2) {
      return o1.getName().compareTo(o2.getName());
   }
};
```

```
public interface  Comparator<T> {
  int compare(T o1, T o2);
}
```

- 람다식를 사용한 방식

```
Comparator<Person> byName = (Person o1, Person o2) ->
                                          o1.getName().compareTo(o2.getName());
```

Anonymous Class vs Lambda

- 익명클래스를 사용한 방식

```
List<Person> plist = getPersonList();
// sort by age
plist.sort(new Comparator<Person>() {
   @Override
   public int compare(Person o1, Person o2) {
      return o1.getAge() - o2.getAge();
   }
});
```

```
public interface  Comparator<T> {
  int compare(T o1, T o2);
}
```

- 람다식을 사용한 방식

```
// sort by age
plist.sort( (Person o1, Person o2) -> o1.getAge() - o2.getAge() );
```

Stream API

- **Collection**
  - **명시적 외부 반복**
  - 제어 흐름 중복 발생
  - 효율적이고 직접적인 요소 처리
  - 지저분한 코드
  - 유한 데이터 구조 API
- **Stream**
  - **내부 반복**
  - 반복 구조 캡슐화
  - 제어 흐름 추상화
  - 파이프-필터 기반 API
  - 최적화와 알고리즘 분리
  - 함축적인 표현
  - 무한 연속 데이터 흐름 API
  - 데이터 외 I/O, 값 생성 등 적용

```
for (int n : numbers) {
 …
}
```

```
numbers.forEach(n -> … )
```

Stream 유형

- java.util.stream
  - **Stream<T>** — **객체를 요소로 하는 가장 일반적인 스트림**
  - DoubleStream — 요소가 double인 스트림
  - IntStream — 요소가 int인 스트림
  - LongStream — 요소가 long인 스트림
- java.lang
  - **Optional<T>** — **값이 있을 수도 없을 수도 있을 때 사용**

Stream 생성

- Stream 유형 of(*), range(...)
- **Collection stream(), parallelStream()**
- BufferedReader lines()
- Random doubles(*), ints(*), longs(*)
- BitSet IntStream()
- Arrays stream(*)
- xxxStream.Builder build()

Stream Method

- **Stream<R> map(Function<? super T, ? extends R> mapper)**
  - T 타입의 요소를 1:1로 R 타입의 요소로 변환 후 스트림 생성
- **Stream<R> flatMap(Function<T, Stream<? extends R>> mapper)**
  - T 타입의 요소를 1:n으로 R 타입의 요소로 변환 후 스트림 생성
- **Stream<T> filter(Predicate<? super T> predicate)**
  - T 타입의 요소를 확인해서 기준에 통과한 요소만으로 새 스트림 생성
- **Stream<T> skip(long n)**
  - 처음 n개 버린 후 나머지 요소로 새 스트림 생성
- **Stream<T> limit(long n)**
  - 처음 n개의 요소로 새 스트림 생성
- **Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)**
  - 두 스트림을 연결하여 새 스트림 생성

Stream Method

- **Stream<T> distinct()**
  - 중복된 요소를 제거한 스트림 생성
- **Stream<T> sorted()**
  - Comparable 기준으로 정렬된 스트림 생성
- **Stream<T> sorted(Comparator<? Super T> comparator)**
  - Comparator 기준으로 정렬된 스트림 생성

Stream Method

- **Optional<T> findFirst()**
  - 첫번째 요소를 찾아서 반환
- **Optional<T> findAny()**
  - 임의의 요소를 찾아서 반환
- **Optional<T> max(Comparator<? Super T> comparator)**
  - 조건에 따른 가장 큰 요소 반환
- **Optional<T> min(Comparator<? Super T> comparator)**
  - 조건에 따른 가장 작은 요소 반환
- **boolean anyMatch(Predicate<? super T> predicate)**
  - 주어진 조건에 일치하는 요소가 있으면 true
- **boolean allMatch(Predicate<? super T> predicate)**
  - 주어진 조건에 모두 일치하면 true
- **boolean noneMatch(Predicate<? super T> predicate)**
  - 주어진 조건에 모두 불일치하면 true

Stream Method

- **void forEach(Consumer<? Super T> action)**
  - 각 요소마다 지정된 액션 수행
- **R collect(Collector<? Super T, R> collector)**
  - T 타입의 요소를 모두 모아 하나의 자료구조나 값으로 변환
- **Optional<T> reduce(BinaryOperator<T> accumulator)**
  - T 타입의 요소 둘 씩 accumulator로 계산해 반복적으로 수행 후 최종적으로 하나의 값 (reduced value)이 존재하면 반환
- **T reduce(T identity, BinaryOperator<T> accumulator)**
  - T 타입의 요소 둘 씩 accumulator로 계산해 반복적으로 수행 후 최종적으로 하나의 값 (reduced value) 반환
- **Iterator<T> iterator()**
  - Iterator 객체 반환

중개 연산자 관련 함수형 인터페이스

- java.util.function.* 패키지

```
public interface Consumer<T> {
    void accept(T t); // T 객체를 받아 소비 (반환이 없다)
}
public interface Function<T, R> {
    R apply(T t); // T 객체를 받아 R 객체로 매핑(타입 변환)
}
public interface Predicate<T> {
    boolean test(T t); // T 객체를 받아 조사해서 boolean 반환
}
public interface Supplier<T> {
    T get(); // T 객체를 반환
}
public interface UnaryOperator<T> { // Function<T,R> 하위 인터페이스
    T apply(T t); // T 객체를 받아 연산 후 반환
}
```

Collection vs Stream

- 람다없이 컬렉션 사용 방식

```
public static void printPersonWithinAge(List<Person> people, int low, int high) {
    for (Person p : people) {
        if (low <= p.getAge() && p.getAge() < high) p.print();
    }
}
// 사용하기
printPersonWithinAge(people, 10, 20);
```

- Stream과 람다식을 사용한 방식

```
// 메소드나 클래스 생성없이 바로 사용
people.stream()
        .filter( p -> p.getAge() >= 10 && p.getAge() <= 20 )
        .forEach( p -> p.print() );
```

Collection vs Stream

```
//람다없이 컬렉션 사용 방식
List<Person> youngMale = new ArrayList<>();
for (Person p : people) {
    if (p.getAge() <= 30 && p.getGender() == Gender.MALE) youngMale.add(p);
}
youngMale.sort(new Comparator() {
    public int compare(Person p1, Person p2) {
        return p2.getAge() – p1.getAge(); // descending order
    }
});
List<String> youngMaleNames= new ArrayList<>();
for (Person p : youngMale) {
    youngMaleNames.add(p.getName());
}
```

```
//Stream과 lambda expression을 사용한 방식
List<String> youngMaleNames = people.stream()
        .filter( p -> p.getAge() < 30 && p.getGender() == Gender.MALE )
        .sorted( Comparator.comparing(Person::getAge).reversed() )
        .map( Person::getName )
        .collect( Collectors.toList() );
```

Collection vs Stream

```
//람다없이 컬렉션 사용 방식
List<Integer> evenNumbers = new ArrayList<>();
for (int i=1; i<100; i++) {
    if (i%2 == 0) evenNumbers.add(i);
}
List<Integer> evenNumbersSquare = new ArrayList<>();
for (Integer i : evenNumbers) {
    evenNumbersSquare.add(i * i);
}
List<Integer> evenNumbersSquare2 = new ArrayList<>();
for (int i = 10; i < 20; i++) {
    evenNumbersSquare2.add(evenNumbersSquare.get(i));
}
int sum = 0;
for (Integer i : evenNumbersSquare2) {
    sum += i;
}
```

```
//Stream과 lambda expression을 사용한 방식
IntStream.range(1, 100)
        .filter( n -> n % 2 == 0 )
        .map( n -> n * n)
        .skip(10)
        .limit(10)
        .reduce( 0, Integer::sum );
```

Stream

```
Arrays.asList(5,3,4,7,2).stream(); // 스트림 생성
Arrays.asList(5,3,4,7,2).stream().forEach(System.out::println);
Arrays.asList(5,3,4,7,2).stream().map(i -> i*i).forEach(System.out::println);
Arrays.asList(5,3,4,7,2).stream().filter(i -> i>3).forEach(System.out::println);
int sum = Arrays.asList(5,3,4,7,2).stream().reduce(0, (i, j) -> i+j); //
((((5+3)+4)+7)+2)
System.out.println("sum=" + sum);
List<Integer> intList = Arrays.asList(5,3,4,7,2).stream().filter(i ->
i>3).collect(Collectors.toList());
intList.forEach(System.out::println);
int result = Arrays.asList(5,3,4,7,2).stream().filter(i ->
i>3).findFirst().orElse(null); // 3보다 큰 첫번째 요소 반환
System.out.println("result=" + result);
Arrays.asList(5,3,4,7,2).stream().filter(i ->
i>3).findAny().ifPresent(System.out::println); // 3보다 큰 요소 반환
boolean result2 = Arrays.asList(5,3,4,7,2).stream().allMatch(i -> i>0); //
모두다 0보다 큰지
System.out.println("result2=" + result2);
```

Stream

- 지연 연산
  - 스트림을 반환하는 필터와 맵 연산은 기본적으로 **지연(lazy) 연산**
  - **지연 연산을 통한 성능 최적화 (무상태 중개 연산)**
  - 종단 연산 시 모든 연산 수행 (반복 작업 최소화)
- 병렬 처리
  - 동일한 코드로 순차 연산과 병렬 연산 모두 지원, 선택 가능
  - 순차 연산에서 parallel() 메소드로 병렬연산으로 전환 가능
  - Collection.parallelStream()으로 병렬처리 스트림 생성 가능
  - **경고: 스트림 처리 중에는 스트림 요소를 변경하지 마시오!**
  - 쓰레드가 안전하지 않은 컬렉션에서도 병렬처리 수행

Optional

- **NPE (NullPointerException)**
  - Null 참조로 인해 자바 개발자들이 가장 골치 아프게 겪는 문제는 악명높은 널 포인터 예외 (NPE)이다.
  - Null 처리가 취약한 코드에서는 런타임에 NPE 발생 확률이 높다.
- NPE 발생 시나리오

```
// 주문을 한 회원이 살고 있는 도시를 반환한다
public String getCityOfMemberFromOrder(Order order) {
    return order.getMember().getAddress().getCity();
}
```

  - order가 null이 넘어옴
  - order.getmember()의 결과가 null
  - order.getMember().getAddress()의 결과가 null
  - order.getMember().getAddress().getCity()가 null 반환
- NPE 방어 패턴
  - 1. 중첩 null 체크 2. 사방에서 return 하기
  - Null 체크 로직때문에 코드 가독성 및 유지 보수성이 떨어진다.

Optional

- 전통적인 NPE 방어 패턴
  - 중첩 null 체크

```
public String getCityOfMemberFromOrder(Order order) {
    if (order != null) {
        Member member = order.getMember();
        if (member != null) {
            Address address = member.getAddress();
            if (address != null) {
                String city = address.getCity();
                if (city != null) return city;
            }
        }
    }
    return "Seoul"; // default
}
```

Optional

- 전통적인 NPE 방어 패턴
  - 사방에서 return 하기

```
public String getCityOfMemberFromOrder(Order order) {
    if (order == null)
        return "Seoul";
    Member member = order.getMember();
    if (member == null)
        return "Seoul";
    Address address = member.getAddress();
    if (address == null)
        return "Seoul";
    String city = address.getCity();
    if (city == null)
        return "Seoul";
    return city; // default
}
```

Optional

- **Java.util.Optional<T>**
  - **존재할 지 안 할지 모르는 객체**
  - NPE를 유발할 수 있는 null을 직접 다루지 않아도 된다.
  - Null 체크를 직접 하지 않아도 된다.
  - 명시적으로 해당 변수가 null일 수도 있다는 가능성을 표현할 수 있다.

Optional

- Optional 객체 생성
  - **static <T> Optional<T> Optional.empty()**
    - null을 담은 Optional 객체 생성
  - **static <T> Optional<T> of(T value)**
    - null이 아닌 객체를 담은 Optional 객체 생성
  - **static <T> Optional<T> ofNullable(T value)**
    - **null인지 아닌지 확신할 수 없는 객체를 담은 Optional 객체 생성**

```
Optional<Member> maybeMember = Optional.ofNullable(aMember);
Optional<Member> maybeMember = Optional.ofNullable(null);
```

Optional

- Optional이 담고 있는 객체 접근
  - **T get()**
    - Optional 값을 반환하거나 null인 Optional 객체에 대해 NoSuchElementException
  - **T orElse(T other)**
    - Optional 값을 반환하거나 null인 Optional 객체에 대해 넘어온 인자를 반환
  - **T orElseGet(Supplier<? extends T> other)**
    - Optional 값을 반환하거나 null인 Optional 객체에 대해 넘어온 함수형 인자를 통해 생성된 객체 반환
  - **<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)**
    - Optional 값을 반환하거나 null인 Optional 객체에 대해 넘어온 예외처리 발생
  - **boolean isPresent()**
    - 객체가 존재하면(즉, null 아니면) true 반환

Optional

- **<U> Optional<U> map(Function<? Super T, ? extends U> mapper)**

```
// 주문을 한 회원이 살고 있는 도시를 반환한다
public String getCityOfMemberFromOrder(Order order) {
    return Optional.ofNullable(order)          // Optional<Order>
                   .map(Order::getMember)      // Optional<Member>
                   .map(Member::getAddress)    // Optional<Address>
                   .map(Address::getCity)      // Optional<String>
                   .orElse("Seoul");           // default
}
```

- **Optional<T> filter(Predicate<? Super T> predicate)**
  - 조건으로 필터하기

```
// 주문을 min 보다 나중에 한 회원이 살고 있는 도시를 반환한다
public Optional<Member> getMemberWithin(Order order, int min) {
    return Optional.ofNullable(order)          // Optional<Order>
                   .filter(o -> o.getDate().getTime() > min)
                   .map(Order::getMember);
}
```

Optional

- **void ifPresent(Consumer<? Super T> consumer)**
  - 객체가 존재하면(즉, null 아니면) consumer 동작
  - 특정 결과를 반환하는 대신에 Optional 객체가 감싸고 있는 값이 존재하는 경우에만 실행될 로직을 함수형 인자로 넘길 수 있다.

```
Optional<String> maybeCity = getAsOptional(cities, 3); // null이 아닌 city
maybeCity.ifPresent(city -> {
                System.out.println("length: " + city.length());
});
```

Double Colon Operator (::)

- **:: 연산자 (Double colon operator)**는 람다식을 대체하여 **메서드 참조(method reference)**로 사용된다.
  - 즉, 람다식이 사용될 수 있는 Functional Interface Implementation에서만 사용가능

```
// using lambda
Comparator c = (Person p1, Person p2)
                -> p1.getName().compareTo(p2.getName());
```

```
// using lambda with type inference
Comparator c = (p1, p2) -> p1.getName().compareTo(p2.getName());
```

```
// using :: operator (method reference)
Comparator c = Comparator.comparing(Person::getName);
```

```
// method reference
Function<Person, String> getName = Person::getName; // Person -> String
String name = getName.apply(person1); // person1 객체에 적용
```

Double Colon Operator (::)

- :: **연산자 (즉, method reference)**는 람다식과 동일한 처리를 하는 식이지만 메소드 본문을 제공하는 대신 기존 메소드를 이름으로 참조한다.

```
Function<Double, Double> sq = (Double x) -> x * x; // lambda
double result = sq.apply(3); // 3^2 = 9
```

```
Function<Double, Double> sq = MyClass::square; // MyClass에 정의된 square 정적 메소드
double result= sq.apply(3); // 3^2 = 9
```

```
BiFunction<Double, Double, Double> add = (x, y) -> x + y; // lambda
double result = add.apply(3.2, 3.4); // 3.2 + 3.4 = 6.6
```

```
BiFunction<Double, Double, Double> add = MyClass::sum; // MyClass에 정의된 sum 정적 메소드
double result= add.apply(3.2, 3.4); // 3.2 + 3.4 = 6.6
```

User-Defined Functional Interface

- 빈번하게 사용되는 함수형 인터페이스 (Functional Interface)는 java.util.function 표준 API 패키지로 제공
- User-Defined Functional Interface 생성

```
@FunctionalInterface
interface MultiFunction<T, U, V, W, R> {
    public R apply(T t, U u, V v, W w);
}
```

```
MultiFunction<String, Integer, Double, List<Double>, String> concat =
    (a, b, c, d) -> a + b + c + d; // lambda
String answer= concat.apply("abc", 1, 3.0, Arrays.asList(3.4, 4.4, 6.7)); //
abc13.0[3.4, 5.5, 6.7]
```

Reference

- http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html
- http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
- http://docs.oracle.com/javase/8/docs/api/java/util/Optional.html
- https://www.slideshare.net/gyumee/java-8-lambda-35352385
- http://www.daleseo.com/java8-optional-before/
- http://www.daleseo.com/java8-optional-after/
- http://www.daleseo.com/java8-optional-effective/
- http://palpit.tistory.com/673