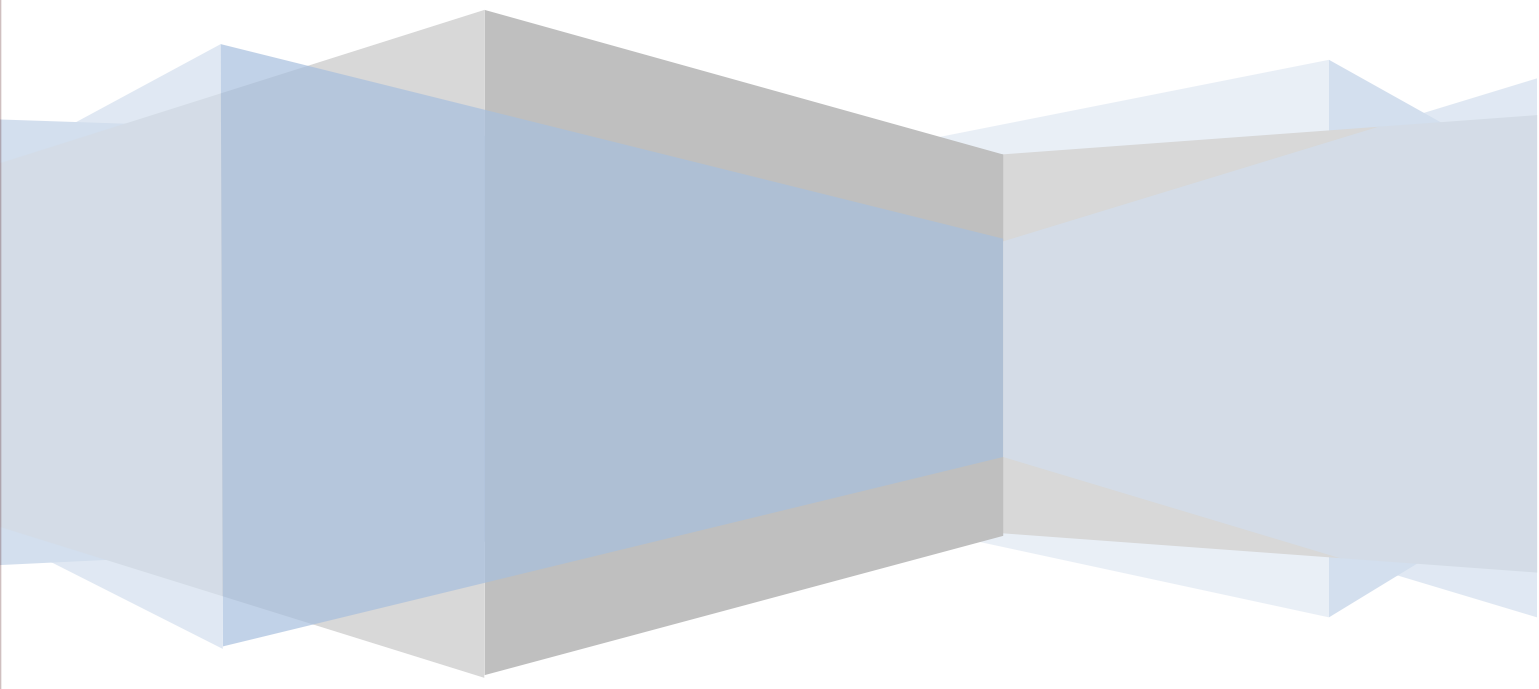
MEGAZONE CLOUD

Java 8 추가 기능



INDEX

함수형 프로그래밍

```
Program(x) = f ° g ° ... (x)
```

● 함수형 프로그래밍 언어를 사용해야 하는 이유

1. 높은 표현력을 통해 불필요한 코드를 줄일 수 있다.

   (무엇을 하는지를 명확히 알 수 있다.)

2. 함수형 프로그래밍 언어군은 프로그래밍 언어론의 최신 연구 결과를 반영하고 있다.

● 함수형 프로그래밍의 요소

· 고계함수, 일급합수

· 커링과 부분적용

· 재귀함수

· 멱등성 (순수함수)

  - 사이드 이팩트가 없다 (함수 밖에 있는 값을 변경하지 않는다.)

  - 상태가 없다. (함수 밖에 있는 값을 사용하지 않는다..)

커링 (Currying)

· 함수를 반환하는 함수를 말한다.

· 함수형 프로그래밍 언어에서 중복 코드의 재활용을 구현하는 방법이다.

● 예시

```
function multiply(a, b) {
      return a * b;
}

-- X 배수값을 계산하는 함수 : 커링
function multiplyX(x) {
      return funciton(a) {
            return multiply(a, x)
      }
}

multiplyX(2)(3);
```

함수형 인터페이스

● 함수형 인터페이스 (Functional Interface)

· default method 또는 static method 는 여러개 존재해도 상관없이 추상 메소드를 오직 하나만 가지고 있는 인터페이스를 말한다.

· Java8 부터 인터페이스는 기본 구현체를 포함한 default 메소드를 포함할 수 있다.

· @FunctionalInterface 애노테이션을 가지고 있는 인터페이스이다. (인터페이스 검증과 유지보수를 위해 사용 권장)

● 함수형 인터페이스 생성

```
@FunctionalInterface
interface CustomerInterface<T> {
      // 추상 메소드
      T myCall();

       //default method
       default void printDefault() {
            System.out.println(”Hello Default!!”);
       }

       // static method
       static void printStatic() {
           System.out.println(”Hello Static!!”);
       }
}
```

자바에서 제공하는 함수형 인터페이스

· 제네릭 함수형 인터페이스, 기본형 특화 함수형 인터페이스를 제공한다.

· java.util.function 패키지에서 제공한다.

● Function<T, R>

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t) ;
}
```

- T 타입 인자를 받아서 R 타입을 리턴한다.
- 람다식으로는 T -> R 로 표현한다.
- 함수 조합용 메소드
  - andThen
  - compose

● Consumer<T>

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t) ;
}
```

- T 타입 인자 하나를 받고 아무값도 리턴하지 않는다.
- 람다식으로는 T -> void 로 표현한다
- 함수 조합용 메소드
  - andThen

● Supplier<T>

```
@FunctionalInterface
public interface Supplier<T> {
    T get( ) ;
}
```

- 아무런 인자를 받지 않고 T 타입의 객체를 리턴한다.
- 람다식으로는 () -> T 로 표현한다

● Predicate<T>

```
@FunctionalInterface
public interface Predicate<T> {

      boolean test(T t) ;
}
```

- T 타입의 인자 하나를 받아서 boolean 타입을 리턴한다.
- 람다식으로는 T -> boolean 로 표현한다
- 함수 조합용 메소드
  - ° And
  - ° Or
  - ° Negate

● Comparator<T>

```
@FunctionalInterface
public interface Comparator<T> {

      int compare(T o1, T o2) ;
}
```

- T 타입의 인자 두개를 받아서 int 타입을 리턴한다.

- 람다식으로는 (T, T) -> int 로 표현한다

● Runnable

```
@FunctionalInterface
public interface Runnable {

      public abstract void run();
}
```

- 아무런 인자를 받지 않고 리턴도 하지 않는다.

- 람다식으로는 ( ) -> void 로 표현한다

두 개의 인자를 받는 Bi 인터페이스

- 두 개의 타입을 인자로 받을 수 있는 인터페이스이다.

함수형 인터페이스	람다식	추상 메소드
BiPredicate	(T, U) -> boolean	boolean test(T t, U u)
BiConsumer	(T, U) -> void	void accept(T t, U u)
BiFunction	(T, U) -> R	R apply(T t, U u)

● UnaryOperator<T>

- Function<T, R>의 특수한 형태로 입력값 하나를 받아서 동일한 타입을 리턴하는 함수형 인터페이스이다.

● BinaryOperator<T>

- BiFunction<T, U, R>의 특수한 형태로 동일한 타입의 입력값 두개를 받아서 동일한 타입을 리턴하는 함수형 인터페이스이다.

람다 표현식 (Lambda Expression)

- 함수를 하나의 식으로 표현한 것이다.

```
( 매개변수, ... ) -> { 실행문 }
```

● 매개변수 리스트

매개변수가 없을 때	( )
매개변수가 한 개 일때	(a) 또는 a
매개변수가 여러 개 일때	(a, b)
매개변수 타입은 생략 가능하다. 명시할 수도 있다.	

● 바디 { }

- 여러 줄인 경우에 { } 를 사용한다.

- 한 줄인 경우에는 { } 생략 가능, return 도 생략 가능하다.

● 변수 캡처

로컬 변수 캡처

- final 이거나 effective final 인 경우에만 참조할 수 있다.
- 그렇지 않을 경우 동시성 문제가 생길수 있어서 컴파일러가 방지한다.

effective final

- 자바 8 부터 지원하는 기능으로 사실상 final 변수이다.
- final 키워드를 사용하지 않는 변수를 익명 클래스 구현체 또는 람다에서 참조할 수 있다.

익명 클래스 구현체와 달리 '쉐도윙' 하지 않는다.

- 익명 클래스는 새로 scope(유효범위)을 만들지만 람다는 람다를 감싸고 있는 scope 과 같다.

메소드 레퍼런스

- 람다가 기존 메소드 또는 생성자 메소드를 호출하는 거라면, 메소드 레퍼런스를 사용해서 매우 간결하게 표현 할 수 있다.

스태틱 메소드 참조	타입::스태틱 메소드
특정 객체의 인스턴스 메소드 참조	객체레퍼런스::인스턴스 메소드
임의 객체의 인스턴스 메소드 참조	타입::인스턴스 메소드
생성자 참조	타입::new

- 메소드 또는 생성자의 매개변수로 람다의 입력값을 받는다.
- 리턴값 또는 생성한 객체는 람다의 리턴값이다.

인터페이스 기본 메소드와 스태틱 메소드

● 기본 메소드 (Default Method)

- 인터페이스에 메소드 선언이 아니라 구현체를 제공하는 방법이다.
- 헤당 인터페이스를 구현한 클래스를 깨뜨리지 않고 새 기능을 추가할 수 있다.
- 기본 메소드는 구현체가 모르게 추가된 기능이므로 리스크가 있다.
  - 컴파일 에러는 아니지만 구현체에 따라 런타임 에러가 발생할 수 있다.
  - 반드시 문서화 할 것. (@implSpec 문서화 태그 사용)

※ `@implSpec`

@implSpec 주석은 해당 메서드와 하위 클래스 사이의 관계를 설명하여, 하위 클래스들이 그 메서드를 상속하거나 super 키워드를 이용해 호출할 때 그 메서드가 어떻게 동작하는지를 명확히 인지하고 사용하게 해야 한다.

- Object 가 제공하는 기능 (equals, hashCode)는 기본 메소드로 제공할 수 없다.

  · 구현체가 재정의해야 한다.

- 인터페이스를 상속받는 인터페이스에서 다시 추상메소드로 변경할 수 있다.

- 인터페이스 구현체가 재정의할 수도 있다.

● 스태틱 (static) 메소드

- 해당 타입 관련 헬퍼 또는 유틸리티 메소드를 제공할 때 인터페이스에 스태틱 메소드를 제공할 수 있다.

Java 8 API의 기본 메소드와 스태틱 메소드

● Iterable<T> 기본 메소드

- default void forEach(Consumer<? super T> action)
- default Spliterator<T> spliterator()

● Collection<E> 기본 메소드

- default Stream<E> stream()
- default boolean removeIf(Predicate<? super E> filter)
- default Spliterator<T> spliterator()

● Comparator<T> 기본 메소드와 스태틱 메소드

- reversed()
- thenComparing()
- static reverseOrder() / naturalOrder()
- static nullsFirst() / nullsLast()
- static comparing()

스트림 (Stream)

- Java 8 부터 지원

- 컬렉션, 배열등에 저장되어있는 요소들을 하나씩 참조하며 내부적으로 반복 작업을 처리한다.

- 스트림은 데이터 소스로 부터 데이터를 읽기만 할 뿐 데이터를 변경하지 않는다.

- 스트림으로 처리하는 데이터는 오직 한번만 처리한다. (일회용)

- 무제한일 수도 있다. (Short Circuit) 메소드를 사용해서 제한 할 수 있다.

- 중개 오퍼레이션은 근본적으로 Lazy(지연)하다.

지연(lazy) 되었던 모든 중개 연산들이 최종 연산(terminal operation) 시에 모두 수행되므로 성능을 최적화 할 수 있다.

- 손쉽게 병렬처리 할 수 있다.

● 스트림 파이프라인

- 0 또는 다수의 중개 오퍼레이션 (intermediate operation) 과 한 개의 종료 오퍼레이션 (terminal operation)으로 구성된다.

- 스트림의 데이터 소스는 오직 터미널 오퍼레이션을 실행할 때만 처리한다.

● Stream<T> 생성 (java.util.stream 패키지)

- 스트림은 다음과 같은 다양한 데이터 소스에서 생성할 수 있다.

- 컬렉션
- 배열
- 가변 매개변수
- 지정된 범위의 연속된 정수
- 특정 타입의 난수
- 람다 표현식
- 파일
- 빈 스트림

```
ArrayList<Integer> list = new ArrayList<Integer>():
Stream<Integer> stream = list.stream();
stream.forEach(System.out::println);
```

```
String[] arr = new String[] {”first”, “second”, “third”};
Stream<String> stream = Arrays.stream(arr);
stream.forEach(e -> System.out.println(e + “ “));
```

```
Stream<Double> stream = Stream.of(4.2, 2.5, 3.1, 1.9);
stream.forEach(System.out::println);
```

https://ahndding.tistory.com/23 사이트 참고할 것!!

● 중개 오퍼레이션 (intermediate operation)

- Stream을 리턴한다. 따라서 연속적으로 연결해서 사용할 수 있다.

- filter, map, limit, skip. sorted ...

1. 스트림 필터링 : filter()

```
Stream<T> filter(Predicate<? super T) predicate)
```

- 해당 스트림에서 주어진 조건(predicate)에 맞는 요소만으로 구성된 새로운 스트림을 반환한다.

2. 스트림 변환 : map(), flatMap()

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

- 해당 스트림의 요소들을 주어진 함수에 인수로 전달하여 그 반환값으로 이루어진 새로운 스트림을 반환한다.

```
<R> Stream<R> flatMap(Function<? super T, ? extends R> mapper)
```

- 해당 스트림의 요소가 배열일 경우, 배열의 각 요소를 주어진 함수에 인수로 전달하여 그 번환값으로 이루어진 새로운 스트림을 반환한다.

```
String[] arr = { ”I study hard”, “You study JAVA”, “I am hungry” };
Arrays.stream(arr)
    .flatMap(s -> Stream.of(s.split(” +”)))
    .forEach(System.out::println);
```

3. 스트림 제한 : limit( ), skip( )

```
//최대 n 개의 요소가 담긴 스트림을 반환한다.
Stream<T> limit(long maxSize)

//앞에서 n 개의 요소를 뺀 나머지 스트림을 반환한다.
Stream<T> skip(long n)
```

4. 스트림 정렬: sorted()

```
Stream<T> sorted( )
```

● 종료 오퍼레이션 (terminal operation)

- Stream 을 리턴하지 않는다.

- collect, allMatch, count, forEach, min, max, ...

1. 요소의 출력 : forEach

```
void forEach(Consumer<? super T> action)
```

2. 요소의 소모 : reduce

- 처음 두 요소를 가지고 연산을 수행한 뒤, 그 결과와 다음 요소를 가지고 또 다시 연산을 수행한다. 이런식으로 해당 스트림의 모든 요소에 대해 연산을 수행하고, 그 결과를 반환한다.

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

```
// 0 + 1 + 2 + 3 + 4 + 5
int[] arr = {1, 2, 3, 4, 5} ;
int result = Arrays.stream(arr)
                    .reduce(0, (a, b) -> a + b);
```

3. 요소의 검색 : findFirst, findAny

```
// filter 조건에 일치하는 요소 1 개를 Optional<T>로 리턴한다.
// 조건에 일치하는 요소가 없는 경우 empty 가 리턴된다.

Optional<T> findFirst()

Optional<T> findAny()
```

- 스트림을 병렬로 처리할 때 차이가 있다.
- findFirst()는 여러 요소가 조건에 부합해도 Stream 의 순서를 고려하여 가장 앞에 있는 요소를 리턴한다.
- findAny()는 Multi thread 에서 Stream 을 처리할 때 가장 먼저 찾은 요소를 리턴한다. (병렬 스트림에서 사용함)

4. 요소의 검사 : anyMatch(), allMatch(), noneMatch()

```
// 스트림의 일부 요소가 특정 조건을 만족할 경우에 true 를 반환한다.
 boolean anyMatch(Predicate<? super T> predicate)

 // 스트림의 모든 요소가 특정 조건을 만족할 경우에 true 를 반환한다.
 boolean allMatch(Predicate<? super T> predicate)

 // 스트림의 모든 요소가 특정 조건을 만족하지 않을 경우에 true 를 반환한다.
 boolean noneMatch(Predicate<? super T> predicate)
```

5. 요소의 통계 : count(), min(), max()

6. 요소의 연산 : sum(), average()

7. 요소의 수집 : collect()

- 인수로 전달된 Collector 객채에 구현된 방법대로 스트림의 요소를 수집함.

```
<R, A> R collect(Collector<? super T, A, R> collector)
```

Optional<T> (java.util 패키지)

- Java 8 에서 Optional<T> 클래스를 사용해 NullPointerException 을 방지할 수 있도록 도와준다.

```
public final class Optional<T> {
    //If non-null, the value; if null, indicates no value is present
     private final T value;
     ...
}
```

- Optional 은 리턴 타입으로만 사용하기를 권장한다.

- 메소드의 반환 값이 절대 null 이 아니라면 Optional 을 사용하지 말자.

Optional 은 값을 Wrapping 하고 다시 풀고, null 일 경우에는 대체하는 함수를 호출하는 등의 오버헤드가 있으므로 잘못 사용하면 성능을 저하시킬 수 있다.

Optional<T> API

● Optional 생성

1. Optional.empty() - 값이 비어있는 경우

```
public static <T> Optional<T> empty()
```

2. Optional.of() - 값이 null 이 아닌 경우

```
public static <T> Optional<T> of(T value)
```

3. Optional.ofNullable() - 값이 null 이거나 아닌 경우

```
public static <T> Optional<T> ofNullable(T value)
```

● Optional 에 값이 존재여부 확인

1. isPresent()

```
public boolean isPresent()
```

2. isEmpty() : Java 11 부터 지원

```
public boolean isEmpty()
```

● Optional 에 있는 값 가져오기

- 값이 존재하면 값을 반환하고, 그렇지 않은 경우 NoSuchElementException 예외를 던진다.

```
public T get()
```

● Optional 의 orElse 와 orElseGet 의 차이

- orElse : 매개변수로 값을 받는다.
- orElseGet : 매개변수로 함수형 인터페이스를 받는다.

```
public T orElse (T other) {
  return value != null ? value : other ;
}
```

```
 public T orElseGet (Supplier<? extends T> other) {
  return value != null ? value : other.get() ;
 }
```

● 예시 (orElse 인 경우)

```
public void findByUserEmail (String userEmail) {
   // orElse 에 의해 userEmail 이 이미 존재해도 유저 생셩 함수가  호출되어 에러가 발생
   // 단, userEmail 이 유니크한 값을 갖는 시스템인 경우
   return userRepository.findByUserEmail(userEmail)
              .orElse(createUserWithEmail(userEmail);
}
private String createUserWithEmail (String userEmail) {
    User newUser = new User();
    newUser.setUserEmail(userEmail);
    return userRepository.save(newUser);
}
```

● 예시 (orElseGet 인 경우)

```
public void findByUserEmail (String userEmail) {
   // orElseGet 에 의해 파라미터로 함수가 전달되므로 null 이 아니면 유저 생성 함수가
   // 호출되지 않는다.
   return userRepository.findByUserEmail(userEmail)
              .orElseGet(createUserWithEmail(userEmail);
}
private String createUserWithEmail (String userEmail) {
    User newUser = new User();
    newUser.setUserEmail(userEmail);
    return userRepository.save(newUser);
}
```

- orElseThrow

- 값이 존재하면 값을 반환하고, 그렇지 않은 경우 NoSuchElementException 예외를 던진다.

```
public T orElseThrow()
```

- filter

```
public Optional<T> filter(Predicate<? super T> predicate)
```

- map

```
public <U> Optional<U> map(Function<? super T, ? extends U> mapper
```

- flatMap

```
public <U> Optional<U> flatMap
          (Function<? super T, ? extends U> mapper
```

Date 와 Time API

- 자바 8 에서 기존 Date 와 Calendar 의 문제점을 해소하기 위해서 새로운 날짜와 시간 API 가 추가되었다.

● 자바 8 에 새로운 Date 와 Time API 가 추가된 이유

- 기존에 사용했던 Date 와 Calendar 클래스는 mutable(변경 가능한) 하기 때문에 스레드에 안전(thread safe)하지 않다.
- 클래스 이름이 명확하지 않다. (Date 인데 시간까지 다룬다.)
- 버그 발생 여지가 많다. (일관성 없는 요일 상수, 월이 0 부터 시작한다거나 ...)

java.time	날짜와 시간을 다루는데 필요햔 핵심 클래스를 제공
java.time.chrono	표준(ISO)이 아닌 달력 시스템을 위한 클래스 제공
java.time.format	날짜와 시간을 파싱, 형식화하기 위한 클래스 제공
java.time.temporal	날짜와 시간의 필드와 단위(Unit)을 위한 클래스 제공
java.time.zone	시간대(time-zone)와 관련된 클래스 제공

※ 타임존이란?

- 지구의 자전에 따라 경도별로 생기는 낮과 밤의 차이를 인위적으로 조정하기 위해서 고안된 시간의 구분선이다.

※ UTC (Coordinated Universal Time/Universal Time Coordinated)란?

- 국제 표준시 (세계 협정시)로 전세계가 공유하고 있는 시간이다.
- 1970 년 1 월 1 일 자정(00:00:00)을 0 밀리초로 설정하여 기준을 삼아 그 후로 시간의 흐름을 밀리초로 계산한다.
- 영국을 기준(UTC+0:00) 으로 각 지역의 시차를 규정한 것이다.
- 서울은 UTC+09:00 이다. (한국은 영국보다 9 시간이 빠르다.)
- timestamp : 서버의 타임존에 맞춰서 UTC 시간을 표시한다.

예] timestamp 형식에 2020-08-25 11:00:00 라고 입력시 현재 타임존이 Asia/Seoul 이라면 2020-08-25 02:00:00 으로 나온다. 출력된 시간은 UTC 이기 때문에 2 시라고 나왔지만 한국시간으로는 11 시를 뜻한다.

Instant vs LocalDateTime

- LocalDateTime

- LocalDateTime 클래스는 타임존을 갖지 않는다.

- Local 은 타임존, 그리고 오프셋이 없음을 의미한다.

※ LocalDateTime 은 언제 사용해야 할까?

· 특정 날짜와 시간을 여러 위치에서 적용하려는 경우

· 예약을 하는 경우

· 타임존이 정해지지 않은 경우

- Instant

```
// UTC 의 현재 순간을 표시한다.
Instant instant = Instant.now() ;
```

- 대부분이 비즈니스 로직, 데이터 스토리지 및 데이터 교환은 UTC 여야 하므로 Instatnt 를 사용한다.

- 로그를 남기거나 현재 시간을 불러올 때 사용한다.

Date-time types in Java	Legacy class	Modern class
Moment in UTC	java.util.Date	java.time.Instant
Moment with offset-from-UTC (hours-minutes-seconds)	(lacking)	java.time.OffsetDateTime
Moment with time zone ('Continent/Region')	java.util.GregorianCalendar	ZonedDateTime
Date & Time-of-day (no offset, no zone) Not a moment	(lacking)	java.time.LocalDateTime
Date only (no offset, no zone)	java.sql.Date	java.time.LocalDate
Time-of-day only (no offset, no zone)	java.sql.Time	java.time.LocalTime

DateTimeFormatter (java.time.format package)

- 날짜 포맷

```
LocalDate date = LocalDate.now();
DateTimeFormatter ormatter =
        DateTimeFormatter.ofPattern(”yyyy MM dd”);
String text = date.format(formatter);
System.out.println(text);
```

● 오전/오후 구하기

```
LocalDateTime dateTime = LocalDateTime.now();
DateTimeFormatter formatter =
        DateTimeFormatter.ofPattern(”a HH시 mmq분”));
String text = datedateTime.format(formatter);
System.out.println(text);
```

● 레거시 API 지원

- GregorianCalendar 와 Date 타입의 인스턴스를 Instant 와 ZonedDateTime 으로 상호 변환 가능하다.

- java.util.TimeZone 에서 java.time.ZoneId 로 상호 변환 가능하다.

```
ZoneId newZoneAPI = TimeZone.getTimeZone(”PST”).toZoneId();

TimeZone legacyZoneAPI = TimeZone.getTimeZone(newZoneAPI);


Instant newInstant = new Date().toInstant();

Date legacyInstant = Date.from(newInstant);
```

```
LocalDate now = LocalDate.now();
System.out.println(now);


LocalDate today = LocalDate.of(2022, 4, 1);
System.out.println(today);


System.out.printf("Year : %d%n", now.getYear());
System.out.printf("Month : %s%n", now.getMonth().getValue());
System.out.printf("DayOfMonth : %d%n", now.getDayOfMonth());
System.out.printf("DayOfWeek : %d%n", now.getDayOfWeek().getValue());
System.out.printf("DayOfWeek : %s%n", now.getDayOfWeek());
System.out.printf("isLeapYear : %s%n", now.isLeapYear());
```

```
LocalTime currentTime = LocalTime.now();

System.out.println(currentTime);


System.out.printf("Hour : %d%n", currentTime.getHour());

System.out.printf("Minute : %d%n", currentTime.getMinute());

System.out.printf("Second : %d%n", currentTime.getSecond());
```

● 현재 시스템에 설정된 타임 존 구하기

```
System.out.println(ZoneId.systemDefault());
```

```
Asia/Seoul
```

```
-------------------------------------------------------
   타임존 : Asia/Seoul
-------------------------------------------------------
ZoneId zoneId = ZoneId.of("Asia/Seoul");
ZonedDateTime seoulTime = ZonedDateTime.now(zoneId);
System.out.println(seoulTime);
```

```
2022-04-01T16:01:10.617160+09:00[Asia/Seoul]
```

```
-------------------------------------------------------
   타임존 : UTC
-------------------------------------------------------
 ZoneId utc = ZoneId.of("UTC");
 ZonedDateTime utcTime = ZonedDateTime.now(utc);
 System.out.println(utcTime);
```

```
 2022-04-01T07:01:10.617160Z[UTC]
```

Nested Interace (중첩 인터페이스)

- 클래스의 멤버로 선언된 인터페이스를 말한다.

- 인터페이스를 클래스 내부에 선언하는 이유는 해당 클래스와 긴밀한 관계를 맺는 구현 클래스를 만들기 위해서이다.

```
public class ClassName {
      [static] interface InterfaceName {
            //abstract method
      }
}
```

- 클래스 멤버로 선언된 중첩 인터페이스는 static 키워드를 붙일수도 있고, 생략도 가능하다.

- 주로 정적 멤버 인터페이스를 많이 사용하는데 UI 프로그래밍에서 이벤트를 처리할 목적으로 많이 사용된다.

● 중첩 인터페이스

```
public class Button {
    OnClickListener listener;

    void setOnClickListener(OnClickListener listener) {
       this.listener = listener;
    }
    void touch() {
      listener.onClick();
    }
    // 중첩 인터페이스
    static interface OnClickListener {
      void onClick();
   }
}
```

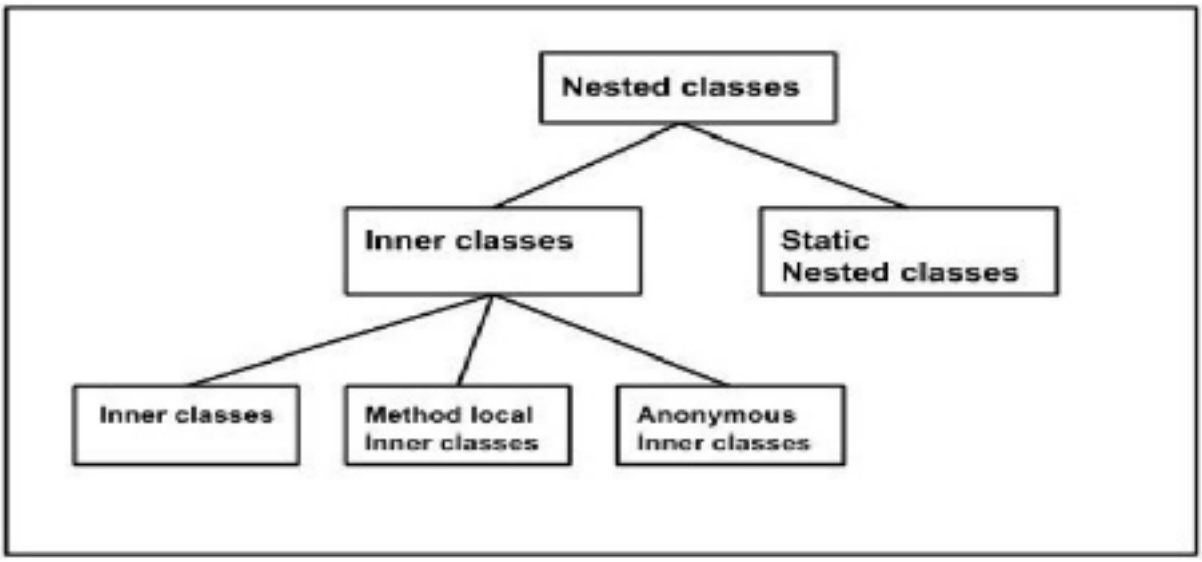
● 구현 클래스

```
public class CallListener implements Button.OnClickLisener {
   @Override
   public void onClick() {
      System.out.println("Call CallListener");
   }
}

public class MessageListener implements Button.OnClickLisener {
   @Override
   public void onClick() {
      System.out.println("Call MessageListener");
   }
}
```

● 테스트 클래스

```
public class Test {

    public static void main(String[] args) {

        Button btn = new Button();


        btn.setOnClickListener(new CallListener());

        btn.touch();


        btn.setOnClickListener(new MessageListener ());

        btn.touch();

    }

}
```

Nested Class (중첩 클래스)



- 클래스 내부에 선언한 클래스이다.

- 클래스가 다른 클래스에만 유용 할 경우 (서로간 결합만이 존재하는 클래스의 경우)

해당 클래스에 클래스를 포함시키면 가독성이 좋고 관리하기 편하다.

- 중첩 클래스는 클래스 내부에 숨겨져 있으므로 캡슐화에 도움이 된다.

A 와 B 라는 클래스가 있다고 가정하자. 여기서 B 는 선언 될 A 의 멤버에 액세스해야 한다. 클래스 A 내에 클래스 B 를 숨기면 A 의 구성원을 비공개(private)로 선언하고 B 가 액세스 할 수 있다. 또한 B 자체는 외부 세계에서 숨길 수 있다.

```
public class Outer {
	private int x = 10;
	// 중첩 클래스
	class Inner {
		int y = 20;
		Inner() { }
		void method1() {
			// Outer 클래스의 private 선언된 멤버를 참조할 수 있다.
			System.out.println(x);
			System.out.println(y);
		}
	}
	public void method2() {
		// 외부 참조
		Inner inner = new Inner();
		inner.method1();
	}
}
```

Nested Class 종류

```
public class ClassName {

    /* 인스턴스 멤버 클래스
       A 객체를 생성해야만 사용할 수 있음.  */
    class A { }

    /* static 멤버 클래스
       A 클래스를 바로 접근 가능 */
    static class B { }

    void method() {
        /* 로컬 클래스
       메소드가 실행될 때만 사용할 수 있음. */
       class C { }
    }

}
```

● 인스턴스 멤버 클래스

- static 키워드 없이 중첩 선언된 클래스이다.

- 인스턴스 필드와 메소드만 선언이 가능하고 정적 필드와 메소드는 선언할 수 없다.

```
class B {
      int filed1;
       class A {
           A() { }
           int filed1;
           static int filed2;
           void method1() {
              System.out.println(this.field1);
              System.out.println(B.this.field1);
           }
           static void method2() { }
      }
}
```

- 인스턴스 멤버 클래스의 인스턴스는 바깥 클래스의 인스턴스와 암묵적으로 연결된다. 따라서, 인스턴스 멤버 클래스의 인스턴스 메서드에서 정규화된 this 를 사용해 바깥 인스턴스의 메서드를 호출하거나 바깥 인스턴스의 참조를 가져올 수 있다. 정규화된 this 란 클래스명.this 형태로 바깥 클래스의 이름을 명시하는 용법을 말한다.

- 인스턴스 멤버 클래스의 인스턴스와 바깥 인스턴스 사이의 관계는 멤버 클래스가 인스턴스화될 때 확립되며, 더 이상 변경할 수 없다. 이 관계는 바깥 클래스의 인스턴스 메서드에서 인스턴스 멤버 클래스의 생성자를 호출할 때 자동으로 만들어지는 게 보통이지만, 드물게는 직접 바깥 인스턴스의 클래스.new A(args) 를 호출해 수동으로 만들기도 한다. 이 관계 정보는 인스턴스 멤버 클래스의 인스턴스 안에 만들어져 메모리 공간을 차지하며, 생성 시간도 더 걸린다.

- 외부 참조가 유지된다는 것은 메모리에 대한 참조가 유지되고 있다는 뜻이다. GC 가 바깥 클래스의 인스턴스를 수거하지 못하는 메모리 누수가 생길 수 있다.

● 정적(Static) 멤버 클래스

- 내부 클래스가 자신이 바깥 클래스 인스턴스를 참조하지 않는다면 내부 클래스는 static nested class 로 만드는 것이 낫다.

```
class ClassName {
      static class A {
          A() { }
          int filed1;
          static int filed2;
          void method1() { }
          static void method2() { }
      }
}
```

● 로컬 클래스

- 로컬 클래스는 내부에서만 사용되므로 접근을 제한할 필요가 없기 때문에 접근 제한자를 붙일 수 없고 static 을 붙일 수 없다.

```
void memthod() {
      [final] int field = 10;  // jdk8 부터 final 생략 가능
      class A {
            A() {}
            int field1 = field;
            static int field2;
            void method1() { }
            static void method2() { }
      }
       A obj = new A();
       obj.field1 = 10;
       obj.method1();
}
```

- 로컬 클래스는 메소드가 실행될 때 메소드 내에서 객체를 생성하고 사용해야 한다.

● 익명 클래스 (Anonymous class)

- 클래스 선언과 객체 생성을 동시에 하는 클래스 이름이 없는 일회용 클래스이다.

- 이름이 없기 때문에 생성자도 가질 수 없으며, 오로지 단 하나의 클래스를 상속받거나 단 하나의 인터페이스만을 구현할 수 있다.

```
new [상위클래스이름 / 구현인터페이스이름] ( ) {
        // 메소드 구현
} ;
```