Modern Java

> 💡 자바 8-11버전별 특징과 자바8 주요 변경점 살펴보기

목차

💥 자바 8-11 버전별 특징

JDK8

- 언어적 변경
  - Lambda Expressions
  - Functional Interfaces
  - 인터페이스 개선
    - default methods
- Collections
  - java.util.stream 패키지 추가
    - stream의 원소들에 함수형 스타일의 작업을 지원.
    - stream API는 Collections API에 통합.
  - HashMap 퍼포먼스 향상
- Date-Time Package
- `Optional`
- `CompletableFuture` - `Future` 인터페이스에서 제공하는 기능을 개선
- IO, NIO
  - `java.lang.String(byte[], *)` 생성자 퍼포먼스 향상.
  - `java.lang.String.getBytes()` 메소드 퍼포먼스 향상.
- java.lang, java.util 패키지

    - Parallel Array Sorting
- HotSpot
    - Removal of PermGen

JDK9

- 라이선스 체계
- 발표 주기 변경(6개월 주기 업그레이드)
    - Update and FAQ on the Java SE Release Cadence
- 모듈화(Project Jigsaw)
- JShell(REPL)
- 통합 JVM 로깅
    - 자바를 실행할 때 `Xlog` 파라미터 옵션을 적용하면 된다.
- HTML5 JavaDoc
    - `javadoc` 명령에 `html5` 옵션을 적용하면 HTML5로 JavaDoc이 빌드된다.
- try-with-resource 개선
- private 메소드도 interface 내에 생성할 수 있게 됨.
- 다이아몬드 연산자 `<>` 개선 - 익명 클래스에서도 `<>`를 쓸 수 있게 됐다.
- 프로세스 API - 프로세스 정보에 접근할 수 있는 새로운 API.
    - 모든 프로세스, 현재 프로세스, 자식 프로세스, 종료 프로세스 등의 정보를 조회하고 관리할 수 있게 됐다.
- `CompletableFuture` 개선 - 타임아웃과 지연 기능 추가
- Reactive stream API

JDK10

- JDK 9 안정화 버전
- 로컬 변수 추론 기능 var
    - JEP 286: Local-Variable Type Inference

- 가비지 컬렉터 개선
  - JDK-8172890 : JEP 307: Parallel Full GC for G1

JDK11

- 발표 주기 변경 후 최초의 LTS 버전
- HTTP 클라이언트 API를 정식으로 추가
- 컬렉션 인터페이스에 `toArray` 메소드 추가
- `var` 키워드 지원 확대 - 람다 표현식에서도 `var` 사용 가능
- String 클래스 기능 추가
  - `isBlank`, `lines`, `strip`, `stripLeading`, `stripTrailing`

참조

- 이종립 블로그

💥 자바8 - Need to know

- 메서드에 코드를 전달하는 기법
  - 람다
- 인터페이스의 디폴드 메서드
- 스트림 API
- Date-Time Package
- Optional

❓ 왜 이와 같은 언어 변경 사항이 생겼을까?

- 다른 언어에 비약한 함수형 프로그래밍 지원
- 빅데이터라는 도전에 직면하면서 멀티코어 컴퓨터나 컴퓨터 클러스터를 이용해서 빅데이터를 효과적으로 처리할 필요성 커짐 but 병렬 프로세싱을 활용해야 하는 데 지금까지의 자바로는 충분히 대응 불가
- 병렬성을 활용하는 코드, 간결한 코드 구현 필요

❓ 왜 메서드에 코드를 전달하는 기법이 필요할까?

> 💡 상품 필터링 하는 메서드를 개선하는 과정을 보며 람다 표현식을 사용함으로써 얻을 수 있는 장점 맛보기

💡 ***상품 필터링*** 예시

1️⃣

```
브랜드가 "라네즈"인 상품리스트

    List<Product> filterProduct(List<Product> products) {
        List<Product> result = new ArrayList<>();

        for (Product product : products) {
            if ("라네즈".equals(product.getBrand()))
                result.add(product);
        }
        return result;
    }

    List<Product> list = filterProduct(products);
```

2️⃣

```
요구사항 변경 - 브랜드 "라네즈" -> "아이오페"


    List<Product> filterProduct(List<Product> products, String brand) {
        List<Product> result = new ArrayList<>();

        for (Product product : products) {
            if (brand.equals(product.getBrand()))
                result.add(product);
        }
        return result;
   }

    List<Product> list = filterProduct(products, "아이오페");
```

3️⃣

```
요구사항 변경 - 브랜드 및 가격

    List<Product> filterProduct(List<Product> products, String brand, int price) {
        List<Product> result = new ArrayList<>();

        for (Product product : products) {
            if (brand.equals(product.getBrand()) && product.getPrice() > 20000)
                result.add(product);
        }
        return result;
    }

    List<Product> list = filterProduct(products, "라네즈", 20000);
```

💃 Strategy Pattern 을 통한 개선

- 동작을 파라미터화하고 다양한 필터 전달
- 컬렉션 탐색 로직과 각 항목에 적용할 동작을 분리할 수 있다.

4️⃣

```
요구사항 변경 - 브랜드 및 가격

    interface ProductFilter {
        boolean filter(Product product);
    }

    class BrandAndPriceProductFilter implements ProductFilter {
        @Override
        public boolean filter(Product product) {
            return "라네즈".equals(product.getBrand()) && product.getPrice() > 20000;
        }
    }

    List<Product> filterProduct(List<Product> products, ProductFilter productFilter) {
        List<Product> result = new ArrayList<>();

        for (Product product : products) {
            if (productFilter.filter(product))
                result.add(product);
        }
        return result;
    }

    List<Product> list = filterProduct(products, new BrandAndPriceProductFilter());
```

5️⃣

```
복잡함을 간소화 하기 위해서 익명 클래스 사용
```

```
    List<Product> list = filterProduct(products, new ProductFilter(){
        @Override
        public boolean filter(Product product) {
            return "라네즈".equals(product.getBrand()) && product.getPrice() > 20000;
        }
    });
```

6️⃣

```
인터페이스 정의 및 클래스 생성 등 복잡함을 간소화하지만 호출하는 입장에서 가독성 저하 및 코드량 비대
- 람다 표현식 사용

    List<Product> list = filterProduct(products,
    (Product product) -> "라네즈".equals(product.getBrand() && product.getPrice() > 2000
0);
```

7️⃣

```
- 추상화를 통한 메서드 재사용성 증가

public interface Predicate<T> {
    boolean test(T t);
}

public static <T> List<T> filter(List<T> list, Predicate<T> p) {
    List<T> result = new ArrayList<>();
    for (T t: list) {
        if(p.test(t)) {
            result.add(t);
        }
    }
}

List<Product> list = filter(products, (Product product) -> "라네즈".equals(product.getBr
and());

List<Banner> list = filter(banner, (Banner banner) -> "아리따움RUN".equals(banner.getNam
e()));
```

💥 람다 표현식

람다 표현식을 어떻게 만드는지, 어떻게 사용하는지, 어떻게 코드를 간결하게 만들 수 있들 수 있는지 설명 또한 자바8 API 에 추가된 중요한 인터페이스와 형식 추론 등의 기능을 확인 및 람다 표현식과 함께 쓰이고 새로운 기능인 메서즈 참조를 설명

목차

📑 람다?

- 람다 표현식은 메서드로 전달할 수 있는 익명 함수를 단순화한 것
- 람다 표현식에는 이름은 없지만, 파라미터 리스트, 바디, 반환 형식, 발행할 수 있는 예외 리스트는 가질 수 있음

람다의 특징

- 익명 - 보통의 메서드와 달리 이름이 없음
- 함수 - 람다는 메서드처럼 클래스에 종속되지 않으므로 함수라고 부름
- 전달 - 람다 표현식을 메서드 인수로 전달하거나 변수로 저장할 수 있다.
  - 일급 객체
- 간결성

    - 익명 클래스처럼 필요없는 코드를 구현할 필요가 없음
- 지연 평가

> Lambda expression evaluation does not cause the execution of the expression's body; instead, this may occur at a later time when an appropriate method of the functional interface is invoked.

```
Ex)  Predicate<Product> filter = product -> product.getPrice() > 20000
```

📢 일급 객체 (First-class Citizen)파라미터로 전달 가능하다. 리턴 값으로 사용할 수 있다.변수나 데이터 구조에 담을 수 있다.

💡 일급 객체인 **자바 스크립트 함수**

```
- 파라미터로 전달 가능하다.
function A() {
}

function B(handler) {
    hanlder();
}

B(A);

- 리턴값으로 사용할 수 있다.
function A() {
    return function B() {

    }
}

- 변수나 데이터 구조에 담을 수 있다.
var a = function A() {

}
```

//TODO 람다 표현식 구성요소 사진 필요

- 파라미터 리스트
- 화살표
- 람다 바디

📑 람다 표현식 예제

```
() -> {}                // No parameters; result is void
() -> 42                // No parameters, expression body
() -> null              // No parameters, expression body
() -> { return 42; }    // No parameters, block body with return
() -> { System.gc(); }  // No parameters, void block body

() -> {                 // Complex block body with returns
  if (true) return 12;
  else {
    int result = 15;
    for (int i = 1; i < 10; i++)
      result *= i;
    return result;
  }
}

(int x) -> x+1              // Single declared-type parameter
(int x) -> { return x+1; }  // Single declared-type parameter
(x) -> x+1                  // Single inferred-type parameter
x -> x+1                    // Parentheses optional for
                            // single inferred-type parameter

(String s) -> s.length()      // Single declared-type parameter
(Thread t) -> { t.start(); }  // Single declared-type parameter
s -> s.length()               // Single inferred-type parameter
t -> { t.start(); }           // Single inferred-type parameter

(int x, int y) -> x+y  // Multiple declared-type parameters
(x, y) -> x+y          // Multiple inferred-type parameters
(x, int y) -> x+y    // Illegal: can't mix inferred and declared types
(x, final y) -> x+y  // Illegal: no modifiers with inferred types
```

📑 함수형 인터페이스

📢 추상 메서드가 하나인 인터페이스, 디폴드 메서드가 여럿 존재해도 추상 메서드가 하나이면 해당 인터페이스는 함수형 인터페이스Evaluation of a lambda expression produces an instance of a functional interface (§9.8).

@FunctionalInterface

함수형 인터페이스임을 가리키는 어노테이션, 해당 어노테이션을 달고 있지만 실제로 함수형 인터페이스가 아니면 컴파일 에러를 발생

전체 표현식을 함수형 인터페이스의 인스턴스로 취급, **함수형 인터페이스를 인수로 받는 메서드에만 람다 표현식을 사용**가능

함수형 인터페이스의 추상 메서드 시그니처 = function descriptor

이미 자바 API 는 Runnable, Callable 등의 다양한 함수 인터페이스를 포함

자바 8 라이브러리 설계자들이 java.util.function 새로운 함수형 인터페이스 추가

- Predicate
- Supplier
- Consumer
- Function

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

    /**
     * Evaluates this predicate on the given argument.
     *
     * @param t the input argument
     * @return {@code true} if the input argument matches the predicate,
     * otherwise {@code false}
     */
    boolean test(T t);
}

@FunctionalInterface
public interface Supplier<T> {

    /**
     * Gets a result.
     *
     * @return a result
     */
```

```
    T get();
}

@FunctionalInterface
public interface Consumer<T> {

    /**
     * Performs this operation on the given argument.
     *
     * @param t the input argument
     */
    void accept(T t);
}

@FunctionalInterface
public interface Function<T, R> {

    /**
     * Applies this function to the given argument.
     *
     * @param t the function argument
     * @return the function result
     */
    R apply(T t);
}
```

📑 어디에, 어떻게 람다를 사용하는가?

- 함수형 인터페이스에만 람다 표현식을 사용가능 → 대상 형식에 맞춤

```
1. 변수에 저장
Function<Product> filter = product -> product.getPrice() > 20000

2. 메서드의 인자
List<Product> filter(Function<Product> filter) {
  // do something
}

filter( product -> product.getPrice() > 20000);

3. 리턴
Predicate<Product> getProductFilter() {
  return product -> product.getPrice() > 20000;
}
```

📑 메서드 참조

특정 메서드만을 호출하는 람다의 축약형

- 가독성 향상

```
// 기존 코드
productList.sort((Product a, Product b) -> a.getCode.compareTo(b.getCode()));

// 메서드 참조와 java.util.Comparator.comparing 를 활용한 개선 코드
productList.sort(comparing(Product::getCode))
```

메서드 참조 세가지 표현 방식

```
1. 정적 메서드 참조

(args) -> ClassName.staticMethod(args);
ClassName::staticMethod

(product) -> CmFunction.isNotEmpty(product.getName());
CmFunction::isNotEmpty

2. 다양한 형식의 인스턴스 메서드 참조

(arg0, rest) -> arg0.instanceMethod(rest);
ClassName::instanceMethod

(productA, productB) -> productA.equals(productB);
Product::equals

3. 기존 객체의 인스턴스 메서드 참조

(args) -> expr.instanceMethod(args)
expr::instanceMethod

(name) -> new Product(name);
Product::new
```

📑 지연평가

📢 람다가 주는 성능삼 이점도 있을까?
Lambda expression evaluation does not cause the execution of the expression's body; instead, this may occur at a later time when an appropriate method of the functional interface is invoked.

지연 평가 - 예시

```
  private void lazy(Supplier<String> supplier, boolean is) {
        if (is)
            System.out.print(supplier.get());
    }

    private void eager(String text, boolean is) {
        if (is)
            System.out.print(text);
    }

    private String getName() {
        try {
            Thread.sleep(2000);
        } catch (Exception e) {
        }
        return "Ishift";
    }

    @Test
    void 즉시평가_지연평가_비교() {

        StopWatch stopWatch = new StopWatch();

        stopWatch.start();
        lazy(() -> getName(), false);
        stopWatch.stop();
        log.info("lazy evaluation time : {}", stopWatch.getLastTaskTimeMillis());

        stopWatch.start();
        eager(getName(), false);
        stopWatch.stop();
        log.info("eager evaluation time : {}", stopWatch.getLastTaskTimeMillis());

    }
```

결과

```
lazy evaluation time : 0
eager evaluation time : 2008
```

스트림 API

> 비지니스 로직의 작성 → Application vs Database
> Performance - Scaling out a database is much harder than scaling out your application
> Maintainability - reuse and abstraction of an implementation is hardly supported
> Reusability
> Integrity
> Security