더 자바: Java 8

2014년 3월에 처음 출시했고 6년이 넘게 지난 지금도 자바 개발자가 가장 많이 사용하고 있는 자바 버전인 자바 8에 대해 학습합니다.

자바 기초 공부는 마쳤지만 그래도 뭔가 아직 자바에 대해 잘 모르겠고 다른 사람이 작성한 코드를 볼 때 생소한 문법이 보인다면 아마도 자바 8에 추가된 기능을 제대로 이해하지 못했기 때문일 수도 있습니다.

자 보시죠! 여기 Chicken이라는 인터페이스를 구현한 기선닭이 있습니다.

```
public class KeesunChicken implements Chicken {
}
```

보시다시피 인터페이스만 구현했을 뿐, 아무런 메소드도 오버라이딩하지 않았죠. 하지만 이런게 가능합니다.

```
public class App {
    public static void main(String[] args) {
        Chicken keesun = new KeesunChicken();
        Egg egg = keesun.create();
    }
}
```

도대체 Egg를 리턴하는 create() 메소드는 어떻게 쓸 수 있게 된 걸까요?

자, 다음 코드를 보시죠. 여기 닭 한마리가 있습니다. 커서 반반 치킨이 되고 싶은 달걀을 보살피고 있네요.

```
Chicken.takesCare(new Egg() {
    @Override
    public String wannaBe() {
        return "양념반 후라이드반";
    }
});
```

이 코드는 줄여서 이렇게 쓸 수도 있습니다.

```
Chicken.takesCare(() -> "양념반 후라이드반");
```

어떻게 이렇게 Egg라는 타입을 쓰지도 않고 깔끔하게 줄일 수 있는지 궁금하신가요?

이번에는 달걀을 분류해 봅시다.

```
List<EggWithColorAndSize> eggs = new ArrayList<>();
eggs.add(EggWithColorAndSize.of().size(3).color("yellow"));
eggs.add(EggWithColorAndSize.of().size(4).color("white"));
eggs.add(EggWithColorAndSize.of().size(3).color("white"));
eggs.add(EggWithColorAndSize.of().size(5).color("yellow"));
```

```
eggs.add(EggWithColorAndSize.of().size(3).color("brown"));
eggs.add(EggWithColorAndSize.of().size(4).color("yellow"));
```

이 강의를 수강하신다면, 여기 보이는 달걀들 중에 색이 yellow인 달걀만 골라서 사이즈 별로 정렬한 다음 달걀의 wannaBe를 출력하는 다음과 같은 코드를 작성할 수 있습니다.

```
eggs.stream().filter(e -> e.getColor().equals("yellow"))
      .sorted(Comparator.comparingInt(EggWithColorAndSize::getSize))
      .map(EggWithColorAndSize::wannaBe)
      .forEach(System.out::println);
```

별도의 쓰레드로 알을 낳는 작업을 실행하고 알을 낳으면 (콜백으로) 맛있게 먹는 다음과 같은 코드도 이해하고 작성할 수 있습니다.

```
CompletableFuture<Void> future = CompletableFuture.supplyAsync(() -> {
    System.out.println("꼬기오~ 꼬꼬꼬꼬~ " + Thread.currentThread().getName());
    return EggWithColorAndSize.of().size(5).color("white");
}).thenAccept((egg) -> {
    System.out.println("냠냠냠: " + egg.wannaBe());
});

future.get();
```

이밖에도 자바 8이 제공하는 Date와 Time API, 애노테이션의 활용성을 증진 시킨 변화, 메모리 영역의 중요한 변화 등 중요하고 재미있는 내용이 많으니 수강 부탁드립니다.

감사합니다.

1부 소개

1. 자바 8 소개

자바 8

- LTS 버전
- 출시일: 2014년 3월
- 최근 업데이트: 2020년 4월, JDK 8u251
- 현재 자바 개발자 중 약 83%가 사용중.
    - https://www.jetbrains.com/lp/devecosystem-2019/java/

LTS(Long-Term-Support)와 비-LTS 버전의 차이

- 비-LTS는 업데이트 제공 기간이 짧다.
- 비-LTS 배포 주기 6개월
- 비-LTS 지원 기간은 배포 이후 6개월
- LTS 배포 주기 3년 (매 6번째 배포판이 LTS가 된다.)
- LTS 지원 기간은 5년이상으로 JDK를 제공하는 밴더와 이용하는 서비스에 따라 다르다.
- 실제 서비스 운영 환경(production)에서는 LTS 버전을 권장한다.
- Moving Java Forward Faster-Mark Reinhold
- 다음 LTS: 자바 17
- 매년 3월과 9월에 새 버전 배포

“I propose that after Java 9 we adopt a strict, time-based model with a new feature release every six months, update releases every quarter, and a **long-term support** release every three years.”

주요 기능

- 람다 표현식
- 메소드 레퍼런스
- 스트림 API
- Optional<T>
- ...

JDK 다운로드

- 오라클 JDK
    - https://www.oracle.com/java/technologies/javase-downloads.html
- 오픈 JDK
    - 오라클: https://jdk.java.net/14/
    - AdoptOpenJDK: https://adoptopenjdk.net/
    - Amazon Corretto
    - Azul Zulu

    - ...

참고

- https://www.oracle.com/java/technologies/java-se-support-roadmap.html
- https://www.javacodegeeks.com/2019/07/long-term-support-mean-openjdk.html
- https://en.wikipedia.org/wiki/Java_version_history

2부 함수형 인터페이스와 람다 표현식

2. 함수형 인터페이스와 람다 표현식 소개

함수형 인터페이스 (Functional Interface)

- 추상 메소드를 딱 하나만 가지고 있는 인터페이스
- SAM (Single Abstract Method) 인터페이스
- @FuncationInterface 애노테이션을 가지고 있는 인터페이스

람다 표현식 (Lambda Expressions)

- 함수형 인터페이스의 인스턴스를 만드는 방법으로 쓰일 수 있다.
- 코드를 줄일 수 있다.
- 메소드 매개변수, 리턴 타입, 변수로 만들어 사용할 수도 있다.

자바에서 함수형 프로그래밍

- 함수를 First class object로 사용할 수 있다.
- 순수 함수 (Pure function)
    - 사이드 이팩트가 없다. (함수 밖에 있는 값을 변경하지 않는다.)
    - 상태가 없다. (함수 밖에 있는 값을 사용하지 않는다.)
- 고차 함수 (Higher-Order Function)
    - 함수가 함수를 매개변수로 받을 수 있고 함수를 리턴할 수도 있다.
- 불변성

3. 자바에서 제공하는 함수형 인터페이스

Java가 기본으로 제공하는 함수형 인터페이스

- java.lang.funcation 패키지
- 자바에서 미리 정의해둔 자주 사용할만한 함수 인터페이스
- Function<T, R>
- BiFunction<T, U, R>
- Consumer<T>
- Supplier<T>
- Predicate<T>
- UnaryOperator<T>
- BinaryOperator<T>

Function<T, R>

- T 타입을 받아서 R 타입을 리턴하는 함수 인터페이스
    - R apply(T t)
- 함수 조합용 메소드
    - andThen
    - compose

BiFunction<T, U, R>

- 두 개의 값(T, U)를 받아서 R 타입을 리턴하는 함수 인터페이스
    - R apply(T t, U u)

Consumer<T>

- T 타입을 받아서 아무값도 리턴하지 않는 함수 인터페이스
    - void Accept(T t)
- 함수 조합용 메소드
    - andThen

Supplier<T>

- T 타입의 값을 제공하는 함수 인터페이스
    - T get()

Predicate<T>

- T 타입을 받아서 boolean을 리턴하는 함수 인터페이스
    - boolean test(T t)
- 함수 조합용 메소드
    - And
    - Or
    - Negate

UnaryOperator<T>

- Function<T, R>의 특수한 형태로, 입력값 하나를 받아서 동일한 타입을 리턴하는 함수 인터페이스

BinaryOperator<T>

- BiFunction<T, U, R>의 특수한 형태로, 동일한 타입의 입력값 두개를 받아 리턴하는 함수 인터페이스

4. 람다 표현식

람다

- `(인자 리스트) -> {바디}`

인자 리스트

- 인자가 없을 때: ()
- 인자가 한개일 때: (one) 또는 one
- 인자가 여러개 일 때: (one, two)
- 인자의 타입은 생략 가능, 컴파일러가 추론(infer)하지만 명시할 수도 있다. (Integer one, Integer two)

바디

- 화상표 오른쪽에 함수 본문을 정의한다.
- 여러 줄인 경우에 { }를 사용해서 묶는다.
- 한 줄인 경우에 생략 가능, return도 생략 가능.

변수 캡처 (Variable Capture)

- 로컬 변수 캡처
  - final이거나 effective final 인 경우에만 참조할 수 있다.
  - 그렇지 않을 경우 concurrency 문제가 생길 수 있어서 컴파일가 방지한다.
- effective final
  - 이것도 역시 자바 8부터 지원하는 기능으로 “사실상" final인 변수.
  - final 키워드 사용하지 않은 변수를 익명 클래스 구현체 또는 람다에서 참조할 수 있다.
- 익명 클래스 구현체와 달리 ‘쉐도윙’하지 않는다.
  - 익명 클래스는 새로 스콥을 만들지만, 람다는 람다를 감싸고 있는 스콥과 같다.

참고

- https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html#shadowing
- https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

5. 메소드 레퍼런스

람다가 하는 일이 기존 메소드 또는 생성자를 호출하는 거라면, 메소드 레퍼런스를 사용해서 매우 간결하게 표현할 수 있다.

메소드 참조하는 방법

스태틱 메소드 참조	타입::스태틱 메소드
특정 객체의 인스턴스 메소드 참조	객체 레퍼런스::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.

참고

- https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html

3부 인터페이스의 변화

6. 인터페이스 기본 메소드와 스태틱 메소드

기본 메소드 (Default Methods)

- 인터페이스에 메소드 선언이 아니라 구현체를 제공하는 방법
- 해당 인터페이스를 구현한 클래스를 깨트리지 않고 새 기능을 추가할 수 있다.
- 기본 메소드는 구현체가 모르게 추가된 기능으로 그만큼 리스크가 있다.
    - 컴파일 에러는 아니지만 구현체에 따라 런타임 에러가 발생할 수 있다.
    - 반드시 문서화 할 것. (@implSpec 자바독 태그 사용)
- Object가 제공하는 기능 (equals, hasCode)는 기본 메소드로 제공할 수 없다.
    - 구현체가 재정의해야 한다.
- 본인이 수정할 수 있는 인터페이스에만 기본 메소드를 제공할 수 있다.
- 인터페이스를 상속받는 인터페이스에서 다시 추상 메소드로 변경할 수 있다.
- 인터페이스 구현체가 재정의 할 수도 있다.

스태틱 메소드

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

참고

- https://docs.oracle.com/javase/tutorial/java/IandI/nogrow.html
- https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

7. 자바 8 API의 기본 메소드와 스태틱 메소드

자바 8에서 추가한 기본 메소드로 인한 API 변화

Iterable의 기본 메소드

- forEach()
- spliterator()

Collection의 기본 메소드

- stream() / parallelStream()
- removeIf(Predicate)
- spliterator()

Comparator의 기본 메소드 및 스태틱 메소드

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

참고

- https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html
- https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html
- https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html
- https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

4부 Stream

8. Stream 소개

Stream

- sequence of elements supporting sequential and parallel aggregate operations
- 데이터를 담고 있는 저장소 (컬렉션)이 아니다.
- Funtional in nature, 스트림이 처리하는 데이터 소스를 변경하지 않는다.
- 스트림으로 처리하는 데이터는 오직 한번만 처리한다.
- 무제한일 수도 있다. (Short Circuit 메소드를 사용해서 제한할 수 있다.)
- 중개 오퍼레이션은 근본적으로 lazy 하다.
- 손쉽게 병렬 처리할 수 있다.

스트림 파이프라인

- 0 또는 다수의 중개 오퍼레이션 (intermediate operation)과 한개의 종료 오퍼레이션 (terminal operation)으로 구성한다.
- 스트림의 데이터 소스는 오직 터미널 오퍼네이션을 실행할 때에만 처리한다.

중개 오퍼레이션

- **Stream을 리턴한다.**
- Stateless / Stateful 오퍼레이션으로 더 상세하게 구분할 수도 있다. (대부분은 Stateless지만 distinct나 sorted 처럼 이전 이전 소스 데이터를 참조해야 하는 오퍼레이션은 Stateful 오퍼레이션이다.)
- filter, map, limit, skip, sorted, ...

종료 오퍼레이션

- **Stream을 리턴하지 않는다.**
- collect, allMatch, count, forEach, min, max, ...

참고

- https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html
- https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

9. Stream API

걸러내기
- Filter(Predicate)
- 예) 이름이 3글자 이상인 데이터만 새로운 스트림으로

변경하기
- Map(Function) 또는 FlatMap(Function)
- 예) 각각의 Post 인스턴스에서 String title만 새로운 스트림으로
- 예) List<Stream<String>>을 String의 스트림으로

생성하기
- generate(Supplier) 또는 Iterate(T seed, UnaryOperator)
- 예) 10부터 1씩 증가하는 무제한 숫자 스트림
- 예) 랜덤 int 무제한 스트림

제한하기
- limit(long) 또는 skip(long)
- 예) 최대 5개의 요소가 담긴 스트림을 리턴한다.
- 예) 앞에서 3개를 뺀 나머지 스트림을 리턴한다.

스트림에 있는 데이터가 특정 조건을 만족하는지 확인
- anyMatch(), allMatch(), nonMatch()
- 예) k로 시작하는 문자열이 있는지 확인한다. (true 또는 false를 리턴한다.)
- 예) 스트림에 있는 모든 값이 10보다 작은지 확인한다.

개수 세기
- count()
- 예) 10보다 큰 수의 개수를 센다.

스트림을 데이터 하나로 뭉치기
- reduce(identity, BiFunction), collect(), sum(), max()
- 예) 모든 숫자 합 구하기
- 예) 모든 데이터를 하나의 List 또는 Set에 옮겨 담기

5부 Optional

10. Optional 소개

자바 프로그래밍에서 NullPointerException을 종종 보게 되는 이유

- null을 리턴하니까! && null 체크를 깜빡했으니까!

메소드에서 작업 중 특별한 상황에서 값을 제대로 리턴할 수 없는 경우 선택할 수 있는 방법

- 예외를 던진다. (비싸다, 스택트레이스를 찍어두니까.)
- null을 리턴한다. (비용 문제가 없지만 그 코드를 사용하는 클리어인트 코드가 주의해야 한다.)
- (자바 8부터) Optional을 리턴한다. (클라이언트에 코드에게 명시적으로 빈 값일 수도 있다는 걸 알려주고, 빈 값인 경우에 대한 처리를 강제한다.)

Optional

- 오직 값 한 개가 들어있을 수도 없을 수도 있는 컨네이너.

주의할 것

- 리턴값으로만 쓰기를 권장한다. (메소드 매개변수 타입, 맵의 키 타입, 인스턴스 필드 타입으로 쓰지 말자.)
- Optional을 리턴하는 메소드에서 null을 리턴하지 말자.
- 프리미티브 타입용 Optional을 따로 있다. OptionalInt, OptionalLong,...
- Collection, Map, Stream Array, Optional은 Opiontal로 감싸지 말 것.

참고

- https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html
- https://www.oracle.com/technical-resources/articles/java/java8-optional.html
- 이팩티브 자바 3판, 아이템 55 적절한 경우 Optional을 리턴하라.

11. Optional API

Optional 만들기
- Optional.of()
- Optional.ofNullable()
- Optional.empty()

Optional에 값이 있는지 없는지 확인하기
- isPresent()
- isEmpty() (Java 11부터 제공)

Optional에 있는 값 가져오기
- get()
- 만약에 비어있는 Optional에서 무언가를 꺼낸다면??

Optional에 값이 있는 경우에 그 값을 가지고 ~~를 하라.
- ifPresent(Consumer)
- 예) Spring으로 시작하는 수업이 있으면 id를 출력하라.

Optional에 값이 있으면 가져오고 없는 경우에 ~~를 리턴하라.
- orElse(T)
- 예) JPA로 시작하는 수업이 없다면 비어있는 수업을 리턴하라.

Optional에 값이 있으면 가져오고 없는 경우에 ~~를 하라.
- orElseGet(Supplier)
- 예) JPA로 시작하는 수업이 없다면 새로 만들어서 리턴하라.

Optional에 값이 있으면 가졍고 없는 경우 에러를 던져라.
- orElseThrow()

Optional에 들어있는 값 걸러내기
- Optional filter(Predicate)

Optional에 들어있는 값 변환하기
- Optional map(Function)
- Optional flatMap(Function): Optional 안에 들어있는 인스턴스가 Optional인 경우에 사용하면 편리하다.

6부 Date와 Time API

12. Date와 Time API 소개

자바 8에 새로운 날짜와 시간 API가 생긴 이유

- 그전까지 사용하던 java.util.Date 클래스는 mutable 하기 때문에 thead safe하지 않다.
- 클래스 이름이 명확하지 않다. Date인데 시간까지 다룬다.
- 버그 발생할 여지가 많다. (타입 안정성이 없고, 월이 0부터 시작한다거나..)
- 날짜 시간 처리가 복잡한 애플리케이션에서는 보통 Joda Time을 쓰곤했다.

자바 8에서 제공하는 Date-Time API

- JSR-310 스팩의 구현체를 제공한다.
- 디자인 철학
    - Clear
    - Fluent
    - Immutable
    - Extensible

주요 API

- 기계용 시간 (machine time)과 인류용 시간(human time)으로 나눌 수 있다.
- 기계용 시간은 EPOCK (1970년 1월 1일 0시 0분 0초)부터 현재까지의 타임스탬프를 표현한다.
- 인류용 시간은 우리가 흔히 사용하는 연,월,일,시,분,초 등을 표현한다.
- 타임스탬프는 Instant를 사용한다.
- 특정 날짜(LocalDate), 시간(LocalTime), 일시(LocalDateTime)를 사용할 수 있다.
- 기간을 표현할 때는 Duration (시간 기반)과 Period (날짜 기반)를 사용할 수 있다.
- DateTimeFormatter를 사용해서 일시를 특정한 문자열로 포매팅할 수 있다.

참고

- https://codeblog.jonskeet.uk/2017/04/23/all-about-java-util-date/
- https://docs.oracle.com/javase/tutorial/datetime/overview/index.html
- https://docs.oracle.com/javase/tutorial/datetime/iso/overview.html

13. Date와 Time API

지금 이 순간을 기계 시간으로 표현하는 방법

- Instant.now(): 현재 UTC (GMT)를 리턴한다.
- Universal Time Coordinated == Greenwich Mean Time

```
Instant now = Instant.now();
System.out.println(now);
System.out.println(now.atZone(ZoneId.of("UTC")));

ZonedDateTime zonedDateTime = now.atZone(ZoneId.systemDefault());
System.out.println(zonedDateTime);
```

인류용 일시를 표현하는 방법

- LocalDateTime.now(): 현재 시스템 Zone에 해당하는(로컬) 일시를 리턴한다.
- LocalDateTime.of(int, Month, int, int, int, int): 로컬의 특정 일시를 리턴한다.
- ZonedDateTime.of(int, Month, int, int, int, int, ZoneId): 특정 Zone의 특정 일시를 리턴한다.

기간을 표현하는 방법

- Period / Duration . beteen()

```
Period between = Period.between(today, birthDay);
System.out.println(between.get(ChronoUnit.DAYS));
```

파싱 또는 포매팅

- 미리 정의해둔 포맷 참고 https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#predefined
- LocalDateTime.parse(String, DateTimeFormatter);
- Dateteme

```
DateTimeFormatter formatter =
        DateTimeFormatter.ofPattern("MM/d/yyyy");
LocalDate date = LocalDate.parse("07/15/1982", formatter);
System.out.println(date);
System.out.println(today.format(formatter));
```

레거시 API 지원

- GregorianCalendar와 Date 타입의 인스턴스를 Instant나 ZonedDateTime으로 변환 가능.
- java.util.TimeZone에서 java.time.ZoneId로 상호 변환 가능.

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

```
TimeZone legacyZoneAPI = TimeZone.getTimeZone(newZoneAPI);

Instant newInstant = new Date().toInstant();
Date legacyInstant = Date.from(newInstant);
```

7부 CompletableFuture

14. 자바 Concurrent 프로그래밍 소개

Concurrent 소프트웨어

- 동시에 여러 작업을 할 수 있는 소프트웨어
- 예) 웹 브라우저로 유튜브를 보면서 키보드로 문서에 타이핑을 할 수 있다.
- 예) 녹화를 하면서 인텔리J로 코딩을 하고 워드에 적어둔 문서를 보거나 수정할 수 있다.

자바에서 지원하는 컨커런트 프로그래밍

- 멀티프로세싱 (ProcessBuilder)
- 멀티쓰레드

자바 멀티쓰레드 프로그래밍

- Thread / Runnable

Thread 상속

```
    public static void main(String[] args) {
        HelloThread helloThread = new HelloThread();
        helloThread.start();
        System.out.println("hello : " + Thread.currentThread().getName());
    }

    static class HelloThread extends Thread {
        @Override
        public void run() {
            System.out.println("world : " + Thread.currentThread().getName());
        }
    }
```

Runnable 구현 또는 람다

```
Thread thread = new Thread(() -> System.out.println("world : " + Thread.currentThread().getName()));
thread.start();
System.out.println("hello : " + Thread.currentThread().getName());
```

쓰레드 주요 기능

- 현재 쓰레드 멈춰두기 (sleep): 다른 쓰레드가 처리할 수 있도록 기회를 주지만 그렇다고 락을 놔주진 않는다. (잘못하면 데드락 걸릴 수 있겠죠.)
- 다른 쓰레드 깨우기 (interupt): 다른 쓰레드를 깨워서 interruptedExeption을 발생 시킨다. 그 에러가 발생했을 때 할 일은 코딩하기 나름. 종료 시킬 수도 있고 계속 하던 일 할 수도 있고.
- 다른 쓰레드 기다리기 (join): 다른 쓰레드가 끝날 때까지 기다린다.

참고:

- https://docs.oracle.com/javase/tutorial/essential/concurrency/
- https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#interrupt--

15. Executors

고수준 (High-Level) Concurrency 프로그래밍

- 쓰레드를 만들고 관리하는 작업을 애플리케이션에서 분리.
- 그런 기능을 Executors에게 위임.

Executors가 하는 일

- 쓰레드 만들기: 애플리케이션이 사용할 쓰레드 풀을 만들어 관리한다.
- 쓰레드 관리: 쓰레드 생명 주기를 관리한다.
- 작업 처리 및 실행: 쓰레드로 실행할 작업을 제공할 수 있는 API를 제공한다.

주요 인터페이스

- Executor: execute(Runnable)
- ExecutorService: Executor 상속 받은 인터페이스로, Callable도 실행할 수 있으며, Executor를 종료 시키거나, 여러 Callable을 동시에 실행하는 등의 기능을 제공한다.
- ScheduledExecutorService: ExecutorService를 상속 받은 인터페이스로 특정 시간 이후에 또는 주기적으로 작업을 실행할 수 있다.

ExecutorService로 작업 실행하기

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
executorService.submit(() -> {
    System.out.println("Hello :" + Thread.currentThread().getName());
});
```

ExecutorService로 멈추기

```
executorService.shutdown(); // 처리중인 작업 기다렸다가 종료
executorService.shutdownNow(); // 당장 종료
```

Fork/Join 프레임워크

- ExecutorService의 구현체로 손쉽게 멀티 프로세서를 활용할 수 있게끔 도와준다.

참고

- https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html

16. Callable과 Future

Callable

- Runnable과 유사하지만 작업의 결과를 받을 수 있다.

Future

- 비동기적인 작업의 현재 상태를 조회하거나 결과를 가져올 수 있다.
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html

결과를 가져오기 get()

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Future<String> helloFuture = executorService.submit(() -> {
    Thread.sleep(2000L);
    return "Callable";
});
System.out.println("Hello");
String result = helloFuture.get();
System.out.println(result);
executorService.shutdown();
```

- **블록킹 콜이다.**
- 타임아웃(최대한으로 기다릴 시간)을 설정할 수 있다.

작업 상태 확인하기 isDone()

- 완료 했으면 true 아니면 false를 리턴한다.

작업 취소하기 cancel()

- 취소 했으면 true 못했으면 false를 리턴한다.
- parameter로 true를 전달하면 현재 진행중인 쓰레드를 interrupt하고 그러지 않으면 현재 진행중인 작업이 끝날때까지 기다린다.

여러 작업 동시에 실행하기 invokeAll()

- 동시에 실행한 작업 중에 제일 오래 걸리는 작업 만큼 시간이 걸린다.

여러 작업 중에 하나라도 먼저 응답이 오면 끝내기 invokeAny()

- 동시에 실행한 작업 중에 제일 짧게 걸리는 작업 만큼 시간이 걸린다.
- **블록킹 콜이다.**

17. CompletableFuture 1

자바에서 비동기(Asynchronous) 프로그래밍을 가능케하는 인터페이스.

- Future를 사용해서도 어느정도 가능했지만 하기 힘들 일들이 많았다.

Future로는 하기 어렵던 작업들

- Future를 외부에서 완료 시킬 수 없다. 취소하거나, get()에 타임아웃을 설정할 수는 있다.
- 블로킹 코드(get())를 사용하지 않고서는 작업이 끝났을 때 콜백을 실행할 수 없다.
- 여러 Future를 조합할 수 없다, 예) Event 정보 가져온 다음 Event에 참석하는 회원 목록 가져오기
- 예외 처리용 API를 제공하지 않는다.

CompletableFuture

- Implements Future
- Implements **CompletionStage**

비동기로 작업 실행하기

- 리턴값이 없는 경우: runAsync()
- 리턴값이 있는 경우: supplyAsync()
- 원하는 Executor(쓰레드풀)를 사용해서 실행할 수도 있다. (기본은 ForkJoinPool.commonPool())

콜백 제공하기

- thenApply(Function): 리턴값을 받아서 다른 값으로 바꾸는 콜백
- thenAccept(Consumer): 리턴값을 또 다른 작업을 처리하는 콜백 (리턴없이)
- thenRun(Runnable): 리턴값 받지 다른 작업을 처리하는 콜백
- 콜백 자체를 또 다른 쓰레드에서 실행할 수 있다.

18. CompletableFuture 2

조합하기

- thenCompose(): 두 작업이 서로 이어서 실행하도록 조합
- thenCombine(): 두 작업을 독립적으로 실행하고 둘 다 종료 했을 때 콜백 실행
- allOf(): 여러 작업을 모두 실행하고 모든 작업 결과에 콜백 실행
- anyOf(): 여러 작업 중에 가장 빨리 끝난 하나의 결과에 콜백 실행

예외처리

- exeptionally(Function)
- handle(BiFunction):

참고

- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html

8부 그밖에...

19. 애노테이션의 변화

애노테이션 관련 두가지 큰 변화

- 자바 8부터 애노테이션을 타입 선언부에도 사용할 수 있게 됨.
- 자바 8부터 애노테이션을 중복해서 사용할 수 있게 됨.

타입 선언 부

- 제네릭 타입
- 변수 타입
- 매개변수 타입
- 예외 타입
- ...

타입에 사용할 수 있으려면

- TYPE_PARAMETER: 타입 변수에만 사용할 수 있다.
- TYPE_USE: 타입 변수를 포함해서 모든 타입 선언부에 사용할 수 있다.

중복 사용할 수 있는 애노테이션을 만들기

- 중복 사용할 애노테이션 만들기
- 중복 애노테이션 컨테이너 만들기
    - 컨테이너 애노테이션은 중복 애노테이션과 @Retention 및 @Target이 같거나 더 넓어야 한다.

Chicken.java (중복 사용할 애노테이션)

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE_USE)
@Repeatable(ChickenContainer.class)
public @interface Chicken {
    String value();
}
```

ChickenContainer.java (중복 애노테이션의 컨테이너 애노테이션)

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE_USE)
public @interface ChickenContainer {

    Chicken[] value();
}
```

컨테이너 애노테이션으로 중복 애노테이션 참조하기

```
@Chicken("양념")
@Chicken("마늘간장")
public class App {

    public static void main(String[] args) {
        ChickenContainer chickenContainer = App.class.getAnnotation(ChickenContainer.class);
        Arrays.stream(chickenContainer.value()).forEach(c -> {
            System.out.println(c.value());
        });
    }

}
```

20. 배열 Parallel 정렬

Arrays.parallelSort()

- Fork/Join 프레임워크를 사용해서 배열을 병렬로 정렬하는 기능을 제공한다.

병렬 정렬 알고리듬

- 배열을 둘로 계속 쪼갠다.
- 합치면서 정렬한다.

sort()와 parallelSort() 비교

```
int size = 1500;
int[] numbers = new int[size];
Random random = new Random();
IntStream.range(0, size).forEach(i -> numbers[i] = random.nextInt());

long start = System.nanoTime();
Arrays.sort(numbers);
System.out.println("serial sorting took " + (System.nanoTime() - start));

IntStream.range(0, size).forEach(i -> numbers[i] = random.nextInt());
start = System.nanoTime();
Arrays.parallelSort(numbers);
System.out.println("parallel sorting took " + (System.nanoTime() - start));
```

- serial sorting took 548957
- parallel sorting took 364074
- 알고리듬 효율성은 같다. 시간 O(n logN) 공간 O(n)

21. Metaspace

JVM의 여러 메모리 영역 중에 PermGen 메모리 영역이 없어지고 Metaspace 영역이 생겼다.

PermGen

- permanent generation, 클래스 메타데이터를 담는 곳.
- **Heap 영역에 속함.**
- **기본값으로 제한된 크기를 가지고 있음.**
- -XX:PermSize=N, PermGen 초기 사이즈 설정
- -XX:MaxPermSize=N, PermGen 최대 사이즈 설정

Metaspace

- 클래스 메타데이터를 담는 곳.
- **Heap 영역이 아니라, Native 메모리 영역이다.**
- **기본값으로 제한된 크기를 가지고 있지 않다. (필요한 만큼 계속 늘어난다.)**
- 자바 8부터는 PermGen 관련 java 옵션은 무시한다.
- -XX:MetaspaceSize=N, Metaspace 초기 사이즈 설정.
- -XX:MaxMetaspaceSize=N, Metaspace 최대 사이즈 설정.

참고

- http://mail.openjdk.java.net/pipermail/hotspot-dev/2012-September/006679.html
- https://m.post.naver.com/viewer/postView.nhn?volumeNo=23726161&memberNo=36733075
- https://m.post.naver.com/viewer/postView.nhn?volumeNo=24042502&memberNo=36733075
- https://dzone.com/articles/java-8-permgen-metaspace

9부 마무리

22. 요약

자바 8의 주요 기능의 의미

- 함수형 프로그래밍
    - 함수형 인터페이스
    - 람다 표현식
    - 메소드 레퍼런스
- 비동기 프로그래밍
    - CompletableFuture
- 편의성 개선
    - Optional
    - Date & Time
    - 인터페이스
    - ...

앞으로

- 당분간은 자바8이 가장 많이 쓰이는 버전으로 남을 가능성이 높다.
- 하지만 시간이 지날수록 계속해서 자바 11로 넘어가는 비율이 높아질 것이다.
- 따라서 자바 9부터 자바 11까지의 기능도 학습해 둘 것!