section1-class03-Stream API

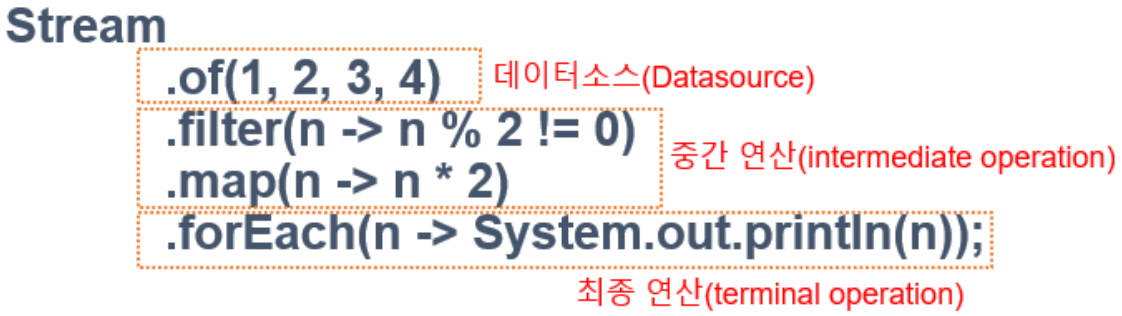
📝 구현을 위한 사전 지식 (3)

✅ Stream API

1️⃣ Stream이란?

- 선언형 프로그래밍 방식으로 컬렉션 데이터를 처리하기 위한 Java의 API.

2️⃣ Stream의 기본 구성



3️⃣ Stream의 중간 연산과 최종 연산

1. **중간 연산(intermediate operation)**
    a. Stream 파이프 라인 형성 가능
    b. filter, map, limit, distinct 등

2. **최종 연산(terminal operation)**

a. 결과 도출

b. 최종 연산이 호출 될 때 Stream이 실행된다.

c. forEach, count, collect 등

4️⃣ Collection과 Stream의 차이점

	Collection	**Stream**
용도	특정 자료구조로 데이터를 저장하는 것이 주 목적이다.	데이터 가공 처리가 주 목적이다.
데이터 수정 여부	데이터 추가/삭제 가능	- 데이터 추가/삭제 불가 - 오로지 데이터 소스를 읽어서 소비하기만 한다.
Iteration 형태	for문 같은 걸로 외부 반복	operation 메서드 내부에서 보이지 않게 반복
탐색 횟수	여러 번 탐색 가능	한번만 탐색 가능
데이터 처리 방식	Eager	Lazy 그리고 Short-circuit

5️⃣ Stream을 언제 사용하는 것이 좋을까?

- **대용량 데이터의 복잡한 가공처리**
- **대용량 데이터가 아니라도 컬렉션 데이터로 복잡한 가공이 필요할 때**
- **멀티쓰레딩이 아닌 진짜 병렬 처리**

6️⃣ Stream API 유형

- **filtering / slicing / mapping / find / match / collect**
- **reducing/ math / grouping / partitioning**
- **parallel**

7️⃣ Stream API 예제 코드 설명

- Stream API 예제 코드 링크
  - https://github.com/ITVillage-Kevin/mini-project-p1-s1-stream-examples/tree/main/src/main/java/com/itvillage/section01/class03/examples

8️⃣ Stream API 연습 문제

- Steam API 연습문제 Solution 코드 링크
  - https://github.com/ITVillage-Kevin/mini-project-p1-s1-stream-examples/tree/main/src/main/java/com/itvillage/section01/class03/practice

---

✅ 예제 1

- 여러 브랜드의 자동차를 판매하는 자동차 판매 총판 기업의 데이터베이스에 여러 브랜드의 자동차 정보가 등록되어 있다고 가정해 봅시다.
- 그리고 데이터베이스에 등록된 자동차 정보를 조회해 아래와 같은 결과 값을 리턴 받았다고 가정합시다.
- 조회된 데이터는 2022년에 가장 많이 팔린 자동차 10위 안에 드는 데이터입니다.
- 매출 10위권 안에 드는 자동차 브랜드(중복이 제거된)를 확인하는 것이 이번 예제 코드의 핵심입니다.

```
import static com.codestates.stream.practice.common.Car.CarBrand;
import static com.codestates.stream.practice.common.Car.CarType;
import java.util.List;

public class SampleDatasource {
    public static List<Car> cars = List.of(
            new Car(CarBrand.Volkswagen, CarType.SEDAN, "티구안",
            new Car(CarBrand.BMW, CarType.SUV,"미니",  33_000_000,
            new Car(CarBrand.Benz, CarType.SUV,"G바겐", 50_000_000
            new Car(CarBrand.Benz, CarType.SEDAN, "E-Class",  28_
            new Car(CarBrand.Ford, CarType.SUV, "익스플로러",  18_0
            new Car(CarBrand.Jeep, CarType.SUV, "랭글러",  23_000_
```

```
            new Car(CarBrand.Volkswagen, CarType.SUV, "투아렉",  4
            new Car(CarBrand.Volkswagen, CarType.SUV, "골프",  43_
            new Car(CarBrand.Jeep, CarType.SEDAN, "체로키",  35_00
            new Car(CarBrand.Jeep, CarType.PICKUPTRUCK, "글래디에이
    );
}
```

- 이 `List<Car> cars` 에는 여러 건의 자동차 정보가 존재하는데, 이 `List<Car> cars`를 이용해 중복이 제거된 `CarBrand(자동차 브랜드)`만 추출해서 콘솔에 출력해 보세요.
- 출력 결과는 아래와 같습니다.

==== 출력 결과 ====

```
Volkswagen, 티구안
BMW, 미니
Benz, 지바겐
Ford, 익스플로러
Jeep, 랭글러
```

: 사실 자동차 이름은 필요없지만 CarBrand(자동차 브랜드)의 중복이 잘 제거되는지 확인하기 위해 자동차 이름도 같이 출력해 봅니다.

==== Car.java ====

```
import lombok.Getter;
import lombok.RequiredArgsConstructor;

@Getter   // lombok 라이브러리 필요
@RequiredArgsConstructor  // lombok 라이브러리 필요
public class Car {
    private final CarBrand carBrand;
    private final CarType carType;
    private final String name;
    private final int price;
    private final boolean isNew;

    public enum CarBrand {
```

```
        BMW,
        Benz,
        Volkswagen,
        Ford,
        Jeep
    }

    public enum CarType {
        SEDAN,
        SUV,
        PICKUPTRUCK

    }
}
```

✅ 예제 2

- Java Stream을 이용해 1부터 400억(`40_000_000_000`)까지의 누적 합계를 구하세요.
- 예)
  - `1 + 2 + 3 + … + 40_000_000_000` = 6790004850489280512
- 단, 누적 합계를 구하기까지의 수행 시간(millisecond)도 아래와 같이 출력해 보세요.

==== 출력 결과 ====

```
6790004850489280512
# 작업 시간: 3964
```

✅ 예제 3

- 아래와 같이 세 개의 베이커리 지점에서 각 지점별로 1월부터 12월 까지 월별 매출이 발생했다고 가정하겠습니다.
  (연습은 로컬에서 실행되는 샘플 데이터이지만 실제로는 물리적으로 떨어진 세 개의 서버에서 각 지점의 데이터를 조회할 수 있음을 머리 속에서 상상해 보면 좋을 것 같습니다.)
- 본사에서는 **이 세 개 베이커리 지점의 매출액을 모두 합친 전체 매출액을 확인하고 싶어합니다.**
- Java Stream API를 이용해 세 개 베이커리 지점들의 월별 매출액을 모두 더한 전체 매출액을 계산해 보세요.

==== 샘플 데이터 ====

```
public class SampleDatasource {
    // A Bakery 지점의 월별 매출
    public static final List<Integer> salesOfBakeryA = List.of(
            5_000_000, 5_000_000, 4_500_000, 5_000_000, 3_000_000
            3_500_000, 6_000_000, 4_500_000, 4_500_000
    );

    // B Bakery 지점의 월별 매출
    public static final List<Integer> salesOfBakeryB = List.of(
            3_000_000, 3_500_000, 3_300_000, 2_600_000, 3_000_000
            5_300_000, 4_400_000, 3_500_000, 3_000_000
    );

    // C Bakery 지점의 월별 매출
    public static final List<Integer> salesOfBakeryC = List.of(
            6_000_000, 5_500_000, 4_800_000, 6_700_000, 7_000_000
            5_300_000, 6_200_000, 7_000_000, 8_000_000
    );
}
```

==== 출력 결과 ====

```
# 전체 매출액: 173000000
```

✅ 예제 4

- 커피 전문점의 사장님은 고객이 주문한 커피 정보 전체 목록을 보고 싶어합니다.
- OrderCoffee 클래스는 DB에서 조회한 고객이 주문한 커피 정보를 담는 클래스입니다.
- OrderCoffee 클래스의 코드는 다음과 같습니다.

==== OrderCoffee 클래스 ====

```
import lombok.AllArgsConstructor;
import lombok.Getter;


@AllArgsConstructor
@Getter
public class OrderCoffee {
    private long orderId;   // 주문 ID
    private long memberId;  // 회원 ID
    private String createdAt;  // 주문 일시
    private long coffeeId;     // 커피 ID
    private int quantity;      // 주문 수량
    private String korName;    // 커피명(한글)
    private String engName;    // 커피명(영문)
    private int price;         // 커피 한 잔 가격
    private String coffeeCode; // 커피 코드
}
```

- 주문한 커피 정보를 List<OrderCoffee> 형태로 DB에서 조회했다고 가정했을 때 주문한 커피 정보에 해당하는 샘플 데이터는 다음과 같습니다.

==== 샘플 데이터 ====

```
public class SampleDatasource {
    public static final List<OrderCoffee> orderCoffees = List.of(
            new OrderCoffee(1L, 1L, "2023-01-25T16:14:28.0876713(
                    1, "아메리카노", "Americano", 2500, "AMR"),
            new OrderCoffee(1L, 1L, "2023-01-25T16:14:28.0876713(
```

```
                    2, "바닐라 라떼", "Vanilla Latte", 4500, "VNL")
            new OrderCoffee(2L, 2L, "2023-01-26T16:14:28.0876713(
                    1, "아메리카노", "Americano", 2500, "AMR"),
            new OrderCoffee(3L, 3L, "2023-01-27T16:14:28.0876715(
                    1, "아메리카노", "Americano", 2500, "AMR"),
            new OrderCoffee(3L, 3L, "2023-01-27T16:14:28.0876715(
                    2, "카라멜 라떼", "Caramel Latte", 5000, "CRL")
}
```

주문한 커피 정보에 해당하는 샘플 데이터인 `orderCoffees`는 일반적으로 웹 애플리케이션 환경에서 JSON 포맷으로 변환되어 클라이언트에게 응답으로 전달될 수 있습니다.

- `SampleDatasource.orderCoffees`를 JSON으로 변환해서 출력하는 예제 코드는 다음과 같습니다.

```
public class StreamExample0501 {
    public static void main(String[] args) {
        Gson gson = new GsonBuilder()
                .setPrettyPrinting()
                .create();
        System.out.println(gson.toJson(SampleDatasource.orderCoff
    }
}
```

SampleDatasource.orderCoffees를 JSON으로 변환하기 위한 라이브러리로는 Gson을 사용했습니다. 어떤 라이브러리를 사용하는지는 중요하지 않습니다. 자신이 사용하기 편한 라이브러리를 사용하면 됩니다.

코드를 실행하면 아래와 같은 출력 결과를 확인할 수 있습니다.

==== orderCoffees를 JSON 포맷으로 변환한 출력 결과 ====

```
[
  {
    "orderId": 1,
    "userId": 1,
    "createdAt": "2023-01-25T11:54:03.987224100",
    "coffeeId": 1,
```

```
    "quantity": 1,
    "korName": "아메리카노",
    "engName": "Americano",
    "price": 2500
  },
  {
    "orderId": 1,
    "userId": 1,
    "createdAt": "2023-01-25T11:54:03.987224100",
    "coffeeId": 2,
    "quantity": 2,
    "korName": "바닐라 라떼",
    "engName": "Vanilla Latte",
    "price": 4500
  },
  {
    "orderId": 2,
    "userId": 2,
    "createdAt": "2023-01-25T11:54:03.987224100",
    "coffeeId": 1,
    "quantity": 1,
    "korName": "아메리카노",
    "engName": "Americano",
    "price": 2500
  },
  {
    "orderId": 3,
    "userId": 3,
    "createdAt": "2023-01-25T11:54:03.987224100",
    "coffeeId": 1,
    "quantity": 1,
    "korName": "아메리카노",
    "engName": "Americano",
    "price": 2500
  },
  {
    "orderId": 3,
    "userId": 3,
    "createdAt": "2023-01-25T11:54:03.987224100",
    "coffeeId": 3,
    "quantity": 2,
```

```
    "korName": "카라멜라떼",
    "engName": "Caramel Latte",
    "price": 5000
  }
]
```

출력 결과를 보면, 주문 ID(orderId)가 동일해도(동일한 주문에 포함되는 커피 정보인데도 불구하고) 주문한 커피 정보가 json object(프로퍼티)별로 각각 출력되는 것을 볼 수 있습니다.

- 주문한 커피 정보를 `주문 ID(orderId), 회원 ID(memberId), 주문한 날짜(createdAt)` 별로 묶어서 가장 최근에 주문한 정보부터 보여지도록 클라이언트에게 전달하면 가독성도 좋고, 웹앱 같은 클라이언트 측에서 화면에 표시하기도 더 수월할 것 같군요.
- Java의 Stream을 사용해 주문한 커피 정보가 아래와 같은 형태의 JSON 포맷으로 출력될 수 있도록 변환해 보세요.
- JSON 포맷으로 변환하기 위해 필요한 클래스 생성에는 제한이 없습니다.

아래와 같은 출력 결과를 표시할 수 있어야 합니다.

==== 조금 더 가독성 있는 출력 결과 ====

```
[
  {
    "orderId": 3,
    "userId": 3,
    "createdAt": "2023-01-27T16:14:28.087671500",
    "orderCoffees": [
      {
        "coffeeId": 1,
        "korName": "아메리카노",
        "engName": "Americano",
        "price": 2500,
        "quantity": 1
      },
      {
        "coffeeId": 3,
        "korName": "카라멜 라떼",
        "engName": "Caramel Latte",
        "price": 5000,
```

```
        "quantity": 2
      }
    ]
  },
  {
    "orderId": 2,
    "userId": 2,
    "createdAt": "2023-01-26T16:14:28.087671300",
    "orderCoffees": [
      {
        "coffeeId": 1,
        "korName": "아메리카노",
        "engName": "Americano",
        "price": 2500,
        "quantity": 1
      }
    ]
  },
  {
    "orderId": 1,
    "userId": 1,
    "createdAt": "2023-01-25T16:14:28.087671300",
    "orderCoffees": [
      {
        "coffeeId": 1,
        "korName": "아메리카노",
        "engName": "Americano",
        "price": 2500,
        "quantity": 1
      },
      {
        "coffeeId": 2,
        "korName": "바닐라 라떼",
        "engName": "Vanilla Latte",
        "price": 4500,
        "quantity": 2
      }
    ]
  }
]
```