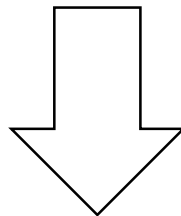


1장. 자바 8, 9, 10, 11 : 무슨 일이 일어나고 있는가?

자바 역사를 통틀어 가장 큰 변화가 자바 8에서 일어났다.

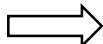
다음은 사과 목록을 무게순으로 정렬하는 고전적 코드이다.

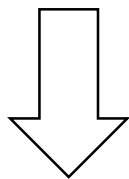
```
1 Collections.sort(inventory, new Comparator<Apple>() {  
2     public int compare(Apple a1, Apple a2) {  
3         return a1.getWeight().compareTo(a2.getWeight());  
4     }  
5 });
```



자바 8을 이용하면 더 간단한 방식으로 코드를 구현할 수 있다.

```
1 inventory.sort(comparing(Apple::getWeight));
```

지금까지의 대부분의 자바 프로그램은 코어 중 하나만을 사용하였다.  하드웨어적인 변화로 스레드를 사용하면 더 효율적이다.



자바 1.0 스레드, 락, 메모리 모델 지원

자바 5 스레드 풀, 병렬 실행 컬렉션 도입

자바 7 포크/조인 프레임워크 도입

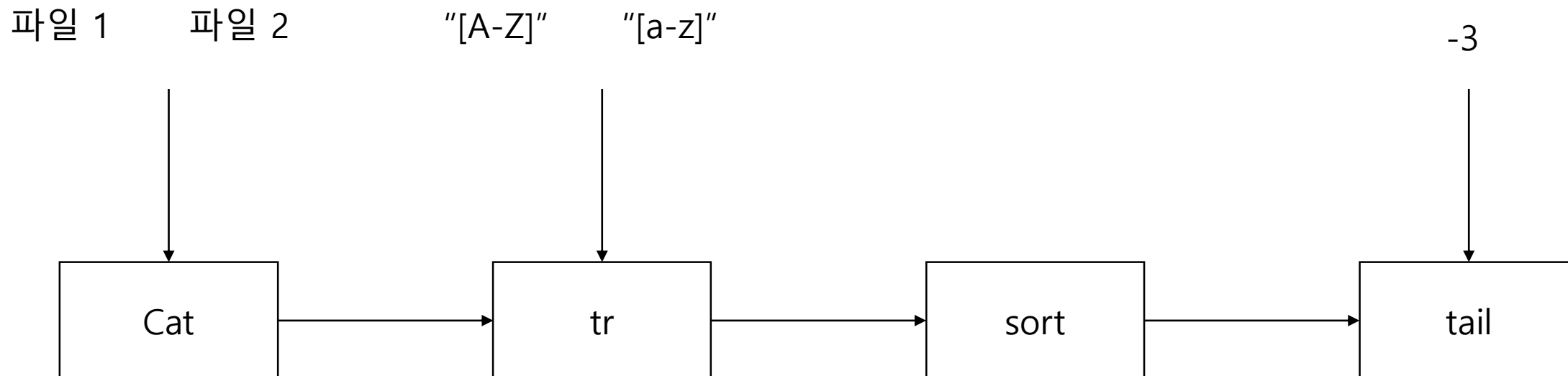
하지만 스레드를 개발자가 활용하기에는 **쉽지 않았다.**

자바 8에서는 병렬 실행을 새롭고 단순한 방식으로 접근할 수 있는 방법을 제공한다.

- 스트림 API
- 메서드 코드를 전달하는 기법
- 인터페이스의 디폴트 메서드

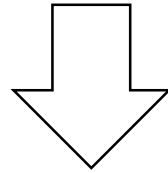
스트림이란 한 번에 한 개씩 만들어지는 연속적인 데이터 항목들의 모임

```
cat file1 file2 | tr "[A-Z]" "[a-z]" | sort | tail -3
```



자바 8에는 `java.util.stream` 패키지에 스트림 API가 추가되었다.

- 기존에는 한 번에 한 항목을 처리했지만 자바 8에서는 일련의 스트림으로 만들어 처리 할 수 있다.
- 스트림 파이프라인을 이용해서 입력 부분을 여러 CPU 코어에 쉽게 할당할 수 있다.

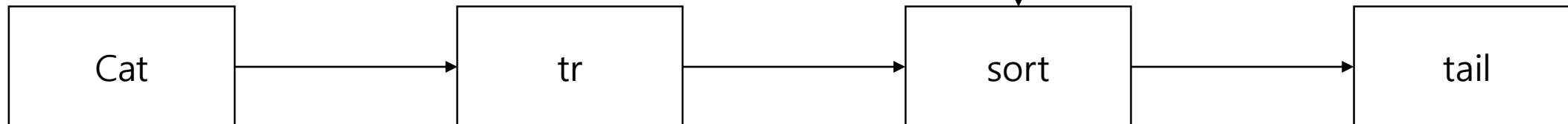


스레드라는 복잡한 작업을 사용하지 않으면서도 공짜로 병렬성을 얻을 수 있다.

자바 8에 추가된 두 번째 프로그램 개념은 코드 일부를 API로 전달하는 기능이다.

예를 들어 2013K001, 2014US0002, ... 등의 형식을 갖는 송장 ID가 있다고 가정하자.
sort 명령을 이용하려면 sort가 고객 ID나 국가 코드로 송장 ID를 정렬하도록 sort에 따로 코드를
제공해야 한다.

```
public int compareUsingCustomerId(String inv1, String inv2) {  
    ....  
}
```

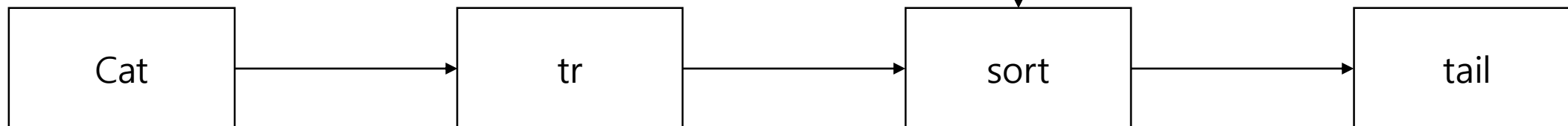


이러한 기능을 이론적으로 동작 파라미터화라고 부른다.

자바 8에 추가된 두 번째 프로그램 개념은 코드 일부를 API로 전달하는 기능이다.

예를 들어 2013K001, 2014US0002, ... 등의 형식을 갖는 송장 ID가 있다고 가정하자.
sort 명령을 이용하려면 sort가 고객 ID나 국가 코드로 송장 ID를 정렬하도록 sort에 따로 코드를
제공해야 한다.

```
public int compareUsingCustomerId(String inv1, String inv2) {  
    ....  
}
```



이러한 기능을 이론적으로 동작 파라미터화라고 부른다.

세 번째 프로그래밍 개념은 ‘병렬성은 공짜로 얻을 수 있다.’라는 말에서 시작된다.

스트림 메서드로 전달하는 코드는 다른 코드와 동시에 실행하더라도 안전하게 실행될 수 있어야 한다.

안전하게 실행할 수 있는 코드를 만들려면 공유된 가변 데이터에 접근하지 않아야 한다.

이러한 함수를 **순수 함수**, **부작용 없는 함수**, **상태 없는 함수**라 부른다.

메서드 참조라는 새로운 자바 8의 기능을 소개한다.

디렉터리에서 모든 숨겨진 파일을 필터링한다고 가정하자. 우선 주어진 파일이 숨겨져 있는지 여부를 알려주는 메서드를 구현해야한다.

```
1 File[] hiddenFiles = new File(".").listFiles(new FileFilter() {  
2     public boolean accept(File file) {  
3         return file.isHidden();  
4     }  
5 });
```

맘에 들지 않음

메서드 참조라는 새로운 자바 8의 기능을 소개한다.

디렉터리에서 모든 숨겨진 파일을 필터링한다고 가정하자. 우선 주어진 파일이 숨겨져 있는지 여부를 알려주는 메서드를 구현해야한다.

```
1 File[] hiddenFiles = new File(".").listFiles(File::isHidden);
```

기존에 객체 참조를 이용해서 객체를 이리저리 주고받았던 것처럼 자바 8에서는 `File::isHidden`을 이용해서 메서드 참조를 만들어 전달할 수 있게 되었다.

자바 8은 메서드를 값으로 취급할 뿐 아니라 람다를 포함하여 함수도 값으로 취급할 수 있다.

ex) `(int x) -> x + 1`

즉 `x라는 인수로 호출하면 $x + 1$ 을 반환` 하는 동작을 수행하도록 코드를 구현할 수 있다.

Apple 클래스와 getColor 메서드가 있고, Apples 리스트를 포함하는 변수 inventor가 있다고 가정하자.

녹색 사과를 선택해서 리스트를 반환하는 프로그램을 구현하려 한다.
이처럼 특정 항목을 선택해서 반환하는 동작을 **필터**라고 한다.

```
1 public static List<Apple> filterGreenApples(List<Apple> inventory) {  
2     List<Apple> result = new ArrayList<>();  
3  
4     for(Apple apple: inventory) {  
5         if (GREEN.equals(apple.getColor())) {  
6             result.add(apple);  
7         }  
8     } return result;  
9 }
```

```
1 public static List<Apple> filterGreenApples(List<Apple> inventory) {  
2     List<Apple> result = new ArrayList<>();  
3  
4     for(Apple apple: inventory) {  
5         if (apple.getWeight() > 150) {  
6             result.add(apple);  
7         }  
8     } return result;  
}
```

자바 8에서는 코드를 인수로 넘겨줄 수 있으므로 filter 메서드를 중복으로 구현할 필요가 없다.

```
1 public static boolean isGreenApple(Apple apple) {
2     return GREEN.equals(apple.getColor());
3 }
4
5 public static boolean isHeavyApple(Apple apple) {
6     return apple.getWeight() > 150;
7 }
8
9 public interface Predicate<T> {
10     boolean test(T t);
11 }
```

```
1 filterApples(inventory, Apple::isGreenApple);
2 filterApples(inventory, Apple::isHeavyApple);
```

Predicate란?

수학에서는 인수로 값을 받아 true나 false를 반환하는 함수를 Predicate라고 한다.

```
1 static List<Apple> filterApples(List<Apple> inventory,
2                                 Predicate<Apple> p) {
3     List<Apple> result = new ArrayList<>();
4     for (Apple apple: inventory) {
5         if (p.test(apple)) {
6             result.add(apple);
7         }
8     }
9     return result;
10 }
11
```

자바 8에서는 다음처럼 (익명 함수 또는 람다)라는 새로운 개념을 이용하여 코드를 구현할 수 있다.



```
1 filterApples(inventory, (Apple a) → GREEN.equals(a.getColor()) );
2 filterApples(inventory, (Apple a) → a.getWeight() > 150 );
3 filterApples(inventory, (Apple a) → a.getWeight() < 80 ||
4                                RED.equals(a.getColor()) );
```

하지만 람다가 몇 줄 이상으로 길어진다면 익명 람다 보다는 코드가 수행하는 일을 잘 설명하는 이름을 가진 메서드를 정의하고 참조를 활용하는 것이 바람직하다.

거의 모든 자바 애플리케이션은 컬렉션을 만들고 활용한다. 하지만 컬렉션으로 모든 문제가 해결되는 것은 아니다.

```
1 Map<Currency, List<Transaction>> transactionsByCurrencies =  
2   new HashMap<>();  
3 for (Transaction transaction : transactions) {  
4   if (transaction.getPrice() > 1000) {  
5     Currency currency = transaction.getCurrency();  
6     List<Transaction> transactionForCurrency =  
7       transactionsByCurrencies.get(currency);  
8     if (transactionForCurrency == null) {  
9       transactionForCurrency = new ArrayList<>();  
10      transactionsByCurrencies.put(currency,  
11                                   transactionForCurrency);  
12    }  
13    transactionForCurrency.add(transaction);  
14  }  
15 }
```

트랜잭션의 리스트를 반복

고가의 트랜잭션을 필터링

트랜잭션의 통화 추출

현재 통화의 그룹화된 맵에 항목이 없으면 새로 만든다.

현재 탐색된 트랜잭션을 같은 통화의 트랜잭션 리스트에 추가한다.

스트림 API를 이용하면 다음처럼 문제를 해결할 수 있다.

```
1 import static java.util.stream.Collectors.groupingBy;
2 Map<Currency, List<Transaction>> transactionsByCurrencies =
3     transactions.stream()
4         .filter((Transaction t) → t.getPrice() > 1000)
5         .collect(groupingBy(Transaction::getCurrency));
```

고가의 트랜잭션을 필터링
통화로 그룹화함

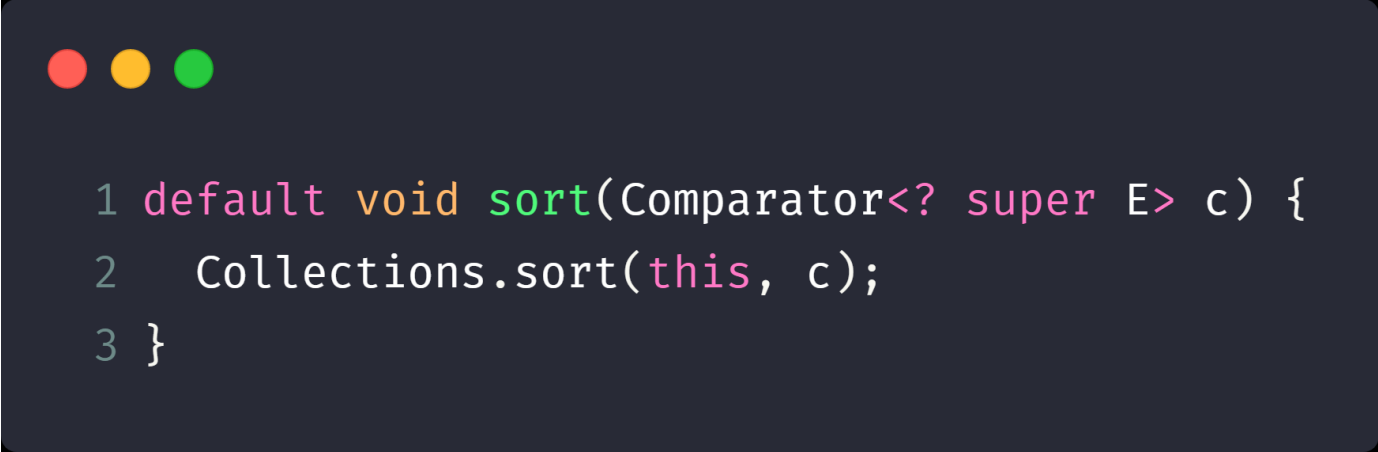
지금까지 자바에서는 특별한 구조가 아닌 평범한 자바 패키지 집합을 포함하는 JAR 파일을 제공하는 것이 전부였다. 게다가 이러한 패키지의 인터페이스를 바꿔야 하는 상황에서는 인터페이스를 구현하는 모든 클래스의 구현을 바꿔야 했다.

- 자바 9의 모듈 시스템 모듈을 정의하는 문법을 제공하므로 이를 이용해 패키지 모음을 포함하는 모듈을 정의할 수 있다.
-> JAR 같은 컴포넌트에 구조를 적용할 수 있고 문서화와 모듈 확인 작업이 용이해졌다.
- 자바 8에서는 인터페이스를 쉽게 바꿀 수 있도록 디폴트 메서드를 지원한다.

여기서는 예제를 이용해서 간단히 디폴트 메서드를 설명한다.

```
1 List<Apple> heavyApples1 =  
2     inventory.stream().filter((Apple a) → a.getWeight() > 150)  
3         .collect(toList());  
4 List<Apple> heavyApples2 =  
5     inventory.parallelStream().filter((Apple a) → a.getWeight() > 150)  
6         .collect(toList());
```

자바 8은 구현 클래스에서 구현하지 않아도 되는 메서드를 인터페이스에 추가할 수 있는 기능을 제공한다.



```
1 default void sort(Comparator<? super E> c) {  
2     Collections.sort(this, c);  
3 }
```

따라서 자바 8 이전에는 List를 구현하는 모든 클래스가 sort를 구현해야 했지만 자바 8부터는 구현하지 않아도 된다.

그런데 하나의 클래스에서 여러 인터페이스를 구현할 수 있지 않은가?

여러 인터페이스에 다중 디폴트 메서드가 존재할 수 있다는 것은 결국 다중 상속이 허용된다는 의미일까?

그렇다고 말할 수 있지만 여기서는 **다이아몬드 상속** 문제를 피할 수 있는 방법을 설명한다.

컴퓨터의 거장인 토니 호아레는 2009년 QCon London의 프레젠테이션에서 다음과 같은 말을 했다.

1965년에 널 참조를 발명했던 일을 회상하며 `그 결정은 정말 뼈아픈 실수였다'고 반성하고 있다...
단지 구현이 편리하단 이유로 널 참조를 만들어야겠다는 유혹을 뿌리치지 못했다.

자바 8에서는 NullPointerException **예외**를 피할 수 있도록 도와주는 Optional<T> 클래스를 제공한다.
Optional<T>는 값이 없는 상황을 어떻게 처리할지 명시적으로 구현하는 메서드를 포함하고 있다.

그 외 패턴 매칭 활용 등 흥미로운 기법을 배운다.

패턴 매칭을 이용하면 “Brakes 클래스는 Car 클래스를 구성하는 클래스 중 하나입니다. Brakes를 어떻게 처리해야 할지 설정하지 않았습니다.” 와 같은 에러를 검출할 수 있다.