11. 자바8 기초

2021.04.27

박경현

- 역사의 흐름은 무엇인가?
  - 1996년 자바 개발 키트(JDK 1.0) 이후 자바 7(2011)에 이르기 까지 새로운 기능과 더불어 계속 발전했다.
  - 자바 8에서 큰 변화가 발생
    - 간결한 코드, 멀티코어 프로세서의 쉬운 활동
    - 사과 무게별 정렬 예시
      - 자바 8 이전

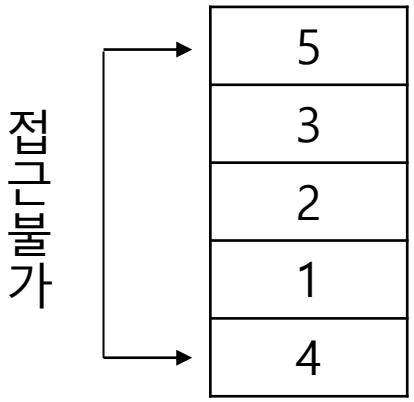
```
Collections.sort(inventory, new Comparator<Apple>() {
    @Override
    public int compare(Apple o1, Apple o2) {
        return o1.waight-o2.waight;
    }
});
```

      - 자바 8 이후(자연어에 더 가깝게 간단한 방식으로 코드를 구현 가능

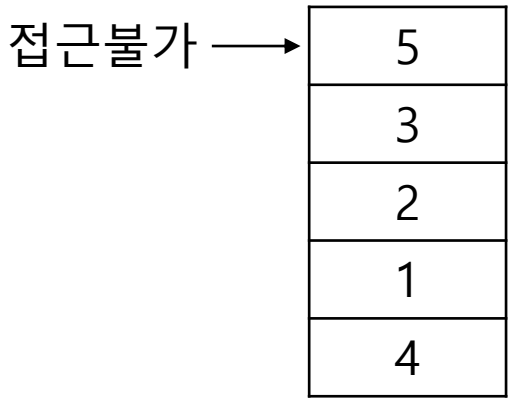
```
inventory.sort(Comparator.comparing(Apple::getWeight));
```

```
static class Apple {
    int weight;

    public Apple(int weight) {
        this.weight = weight;
    }

    @Override
    public String toString() {
        return "Apple [waight=" + weight + "]";
    }

    public int getWeight() {
        return weight;
    }

    public void setWeight(int weight) {
        this.weight = weight;
    }


}
```

- 역사의 흐름은 무엇인가?
  - 자바의 멀티 코어 사용의 변천사
    - 자바 1.0
      - 스레드와 락과 메모리 모델 지원
      - 저수준 기능을 완전히 사용하기 어려운 단점을 가짐
    - 자바 5
      - 스레드 풀, 병렬 실행 컬렉션 같은 도구를 도입
        - Thread-safe : 멀티 스레드 프로그래밍에서 여러 스레드가 동시에 접근해도 실행에 문제가 없음을 뜻함
        - Collection(컬렉션) 컬렉션은 대부분 싱글 스레드 환경에서 사용할 수 있도록 설계되어있다.
        - 동기화된 컬렉션(Thread-safe한 collection)은 멀티 스레드 환경에도 사용 가능하지만 Lock이 발생하기 때문에 느림
        - 병렬 실행 컬렉션은 부분(Segment) 잠금을 사용하기 때문에 병렬적으로 작업 수행이 가능
          - EX) Map에 10개의 요소가 저장되어 있으면 1개의 요소를 처리할 때 그 부분만 Lock걸고 나머지 9개는 다른 스레드가 처리 가능

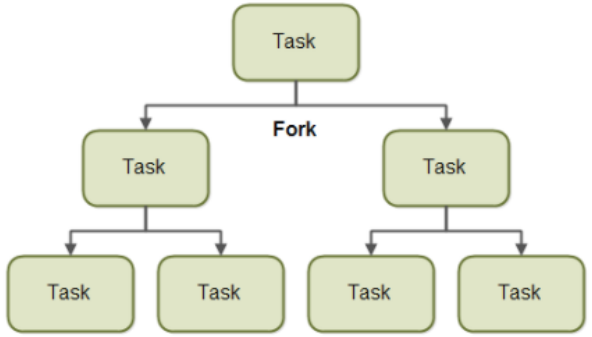


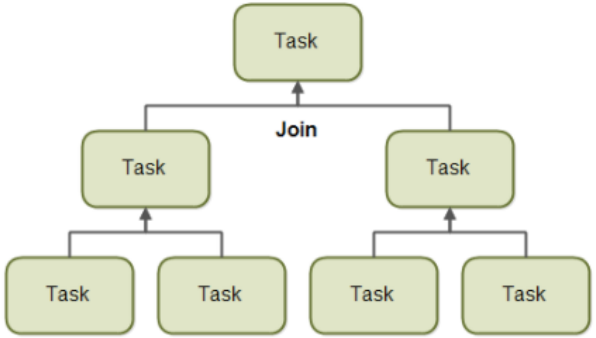
Thread-safe한 collection



병렬 실행 컬렉션

- 역사의 흐름은 무엇인가?
  - 자바의 멀티 코어 사용의 변천사
    - 자바7
      - 병렬 실행에 도움을 줄 수 있는 포크/조인 프레임워크 제공
        - 포크/조인 프레임워크는 큰 업무를 작은 업무 단위로 쪼개고, 그것을 다른 CPU에서 병렬 실행 후 결과 취합하는 방식
          - 큰 업무를 작은 단위의 업무로 쪼갠다
          - 부모 스레드로부터 처리로직을 복사하여 새로운 스레드에서 쪼개진 업무(fork)를 수행
          - 2을 반복하다가, 특정 스레드에서 더 이상 Fork가 일어나지 않고 업무가 완료되면 결과를 부모에게 Join
          - 3을 반복하다가 최초에 생성된 스레드로 값을 리턴하면 종료





- 역사의 흐름은 무엇인가?
  - 자바 8
    - 병렬 연산을 지원하는 Stream이라는 새로운 API를 제공
      - 코드를 전달하는 간결 기법(메서드 참조와 람다)
      - 인터페이스의 디폴트 메서드

- 왜 아직도 자바는 변화하는가
  - 1966년에 이미 프로그래밍 언어가 700개가 생성
  - 그러나 완벽한 언어라는 것이 존재하지는 않았다.
    - C,C++은 낮은 안전성 때문에 보안 구멍이 있으나 작은 런타임 풋프린트 덕분에 OS와 임베디드 시스템에서 인기
    - 런타임 풋프린트에 여유가 있는 애플리케이션에서는 자바와 C#같이 안전한 언어가 유행

- 프로그래밍 언어 생태계에서 자바의 위치
  - 처음부터 많은 유용한 라이브러리를 포함한 잘 설계된 객체지향 언어로 시작
    - 스레드와 락을 이용한 소소한 동시성 지원
      - 자바의 하드웨어 중립적인 메모리 모델 때문에 예기치 못한 상황 일으킬 수 있다.
    - 코드를 JVM 바이트 코드로 컴파일 특징
      - 모든 브라우저에서 가상 머신 코드를 지원하기 때문에 인터넷 애플릿 프로그램의 주요 언어가 됨
      - JVM과 바이트 코드를 JAVA언어보다 중요시하는 일분 애플리케이션에서는 JVM에서 실행되는 스칼라, 그루비등이 자바를 대체

  - 빅데이터 처리를 위해 멀티코어 컴퓨터나 컴퓨팅 클러스터를 이용해야 함
    - 자바 8은 다양한 프로그래밍 도구와 다양한 프로그래밍 문제를 더 빠르고 정확하며 쉽게 유지보수 할 수 있음
      - 스트림 처리
        - 한 번에 한 개씩 만들어지는 연속적인 데이터 항목들의 모임(파이프라인)
      - 동작 파라미터화로 메서드에 코드 전달하기
        - 코드 일부를 API로 전달하는 기능
      - 병렬성과 공유 가변 데이터
        - 병렬성을 공짜로 얻을 수 있다. -> 자바 스트림을 이용하여 처리

- 메서드 참조
  - 예제
    - 디렉터리에서 모든 숨겨진 파일을 필터링한다고 가정
    - File 클래스는 이미 isHidden메서드를 이용해 숨겨진 파일 여부 확인 가능
    - isHidden메서드는 File 클래스를 인수로 받아 boolean을 반환하는 함수이다.

```
File[] hiddenFiles = new File(".").listFiles(new FileFilter() {
    @Override
    public boolean accept(File file) {
        return file.isHidden(); //숨겨진 파일 필터링
    }
});
```

    - 자바 8에서는 다음처럼 코드를 구현 할 수 있다.

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

      - 메서드 참조(::)를 이용하여 listFiles에 직접 전달할 수 있다
      - 기명 메서드 뿐만 아니라 람다를 포함하여 함수도 값으로 취급 가능

- 코드 넘겨주기
  - 음영으로 된 부분만 제외하고는 똑같은 코드이다.

```
for(Apple a : inventory) {
    if(a.getWeight() > 5) {
        System.out.println(a);
    }
}
```

```
for(Apple a : inventory) {
    if(a.getColor().equals("Green")) {
        System.out.println(a);
    }
}
```

  - 이를 Java 8로 바꾸는 것은 다음과 같이 메서드를 호출해야한다

```
public static boolean isGreenApple(Apple a) {
    return a.getColor().equals("Green");
}

public static boolean isHeavyApple(Apple a) {
    return a.getWeight() > 5;
}

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

static void filterApples(ArrayList<Apple> inventory, Predicate<Apple> p) {
    for(Apple a : inventory) {
        if(p.test(a)) {
            System.out.println(a);
        }
    }
}
```

```
System.out.println("==녹색 사과===");
filterApples(inventory, Apple::isGreenApple);
System.out.println("==무게 5 이상===");
filterApples(inventory, Apple::isHeavyApple);
```

- 메서드 전달에서 람다로
  - 한 두번만 사용할 메서드를 매번 정의하는 것이 귀찮은 일
  - 이를 해결하기 위해 람다로 전달 가능

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

static void filterApples(ArrayList<Apple> inventory, Predicate<Apple> p) {
    for(Apple a: inventory) {
        if(p.test(a))
            System.out.println(a);
    }
}


filterApples(inventory, (Apple a) -> a.getWeight() > 15);
```

- 순차처리 방식

```
List<Apple> heavyApples = inventory.stream().filter((Apple a) -> a.getWeight() > 15).collect(Collectors.toList());
```

- 병렬처리 방식

```
List<Apple> heavyApples = inventory.parallelStream().filter((Apple a) -> a.getWeight() > 15).collect(Collectors.toList());
```

- 메서드 전달에서 람다로
  - 한 두번만 사용할 메서드를 매번 정의하는 것이 귀찮은 일
  - 이를 해결하기 위해 람다로 전달 가능

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

static void filterApples(ArrayList<Apple> inventory, Predicate<Apple> p) {
    for(Apple a: inventory) {
        if(p.test(a))
            System.out.println(a);
    }
}


filterApples(inventory, (Apple a) -> a.getWeight() > 15);
```

- 순차처리 방식

```
List<Apple> heavyApples = inventory.stream().filter((Apple a) -> a.getWeight() > 15).collect(Collectors.toList());
```

- 병렬처리 방식

```
List<Apple> heavyApples = inventory.parallelStream().filter((Apple a) -> a.getWeight() > 15).collect(Collectors.toList());
```