자바의 버전별역사

1) 초기 자바 ~ JDK 1.0

- 1991년, OAK 발표: GE사의 요청으로, 썬마이크로 시스템즈에서 C++의 단점을 극복하고자 만든 언어. 메모리 할당/해제의 어려움과 다중상속으로 인한 실수유발을 극복하려고 했다.

- 1996년, JDK 1.0발표 : 언어 이름을 자바라 바꾸고, Java Virtual Machine 1.0을 발표했다. Java Applet도 이때 처음 도입.

2) JDK1.1

- 1997년 : RMI, JDBC, reflection, JIT,Inner Class 개념이 포함되었다.

3) JDK 1.2

- 1998년 Java SE 1.2, ME 1.2, EE 1.2 발표

- 자바를 세가지 버전으로 나눴다. Swing이 SE에 포함, Corba IDL(이종기기간 함수호출 스펙), Collection Framework 포함

4) JDK 1.3

- 2000년도

- HotSpot(Sun에서 만든 JIT구현), JNDI(디렉토리랑 이름으로 원하는 서비스찾는것) 포함

5) JDK 1.4

- 2002년도

- JCP(Java Community Process)에 의해서 오픈소스 정책으로 자바가 관리되기 시작한 버전.

- Java 2 Security 모델의 확립(Sandbox)

- Java Web Start포함 (Java Applet이 브라우저에서 돌아가는 것과 다르게, 외부 sandbox에서 동작)

- Language: assert 도입

- API : Regular Expression, Assert keyword, Security 2 version(현재 security model), Non Blocking IO(NIO)

6) JDK 1.5

- 2004년

- 기능적으로 가장 많은 변화가 생긴버전 (Generics가 가장 대표적)

- LanguageI: Generics , annotation, auto boxing, enum,vararg ,foreach, static imports 도입

- API : java.util.concurrent API, scanner class

7) JDK 1.6

- 2006년도

- 기능에 별 차이 없슴 - 보안, 성능강화 주력.

- JVM/Swing에 있어 많은 Performance 향상(synchronization, compiler, GC,start-up time)

- G1(Garbage First) GC도입.

8) JDK 1.7

- 2011년도

- JVM : Dynamic Language support (invokedynamic - new byte operation)

- Language : Switch에서 String, try-resource, generics에서 타입추론, 숫자에서 underscore사용

- API:Concurrency 강화, NIO 강화, sort강화, crypto강화, GPU강화,

- JavaFX가 기본으로 포함

- 안정적인 ARM지원

9) JDK 1.8

- 2014

- 오라클로 인수된 후 첫번째 버전

- JDK 1.5이후 가장 큰 언어적 변화(Lambda및 함수형프로그래밍,default method)이며 러닝커브가 크다.

- JEP에 의해서 새로운 기능들이 발의되기 시작.

- Language : Lambda expression, Default Method Interface, functional programming for MapReduce style 지원, default method이용한 다중상속지원,메서드 참조

- API : Nashorn (JS엔진), new Date and Time API, stream api,Collection에 대한 함수형화 (Interface에 default가 생김으로서 가능)

- 병철처리에 접합한 구조로 진화

10) JDK 1.9

- 2016 예정

- Modular System (Jigsaw)지원예정

- Money API지원예정

- Java Shell지원예정

- 변수에 대한 타입 추론 지원예정(var,val)

- OpenCL이용한 자동화된 병렬 프로그래밍 지원예정

- value 타입 지원예정

**PART 1**

**CHAPTER1 자바 8을 눈여겨봐야 하는 이유**

**막간> Comparable, Comparator 차이**

기본형 타입이 아닌 Object 내에 특정값을 비교할 경우에는

Comparable를 implements 해서 compareTo를 구현하면 Collections.sort() , Arrays.sort() 등을 사용할 수 있음.

ex)

public class Book implements Comparable<Book>{

private int price;

public Book(int price){

this.price = price;

}

public int getPrice(){

return this.price;

}

@Override

public int compareTo(Book b) {

return this.price - b.price; //자신이 앞에 있는게 ascending order

}

}

Comparator는 compareTo 에서 구현한 기본값 외에 별도의 기준으로 구현하고 싶을 때 다음과 같이 Comparator 구현체 또는 익명 Comparator 구현체를 이용하여 나타낼 수 있다.

public static Comparator<Book> myComparator

= new Comparator<Book>() {

@Override

public int compare(Book b1, Book b2) {

return b1.getPrice() - b2.getPrice();

}

};

public static void main(String[] args) {

List<Book> myBookList = new ArrayList();

myBookList.add(new Book(300));

myBookList.add(new Book(310));

myBookList.add(new Book(100));

myBookList.add(new Book(200));

System.out.println("Original: " + myBookList);

Collections.sort(myBookList, myComparator);

for(Book book : myBookList){

System.out.println(book.getPrice());

}

}

익명 Comparator

Collections.sort(myBookList, new Comparator<Book>() {

@Override

public int compare(Book b1, Book b2) {

return b1.getPrice() - b2.getPrice();

}

});

**결론 :** Comparable과 Comparator은 상호보완 차이

더 간단하게 람다를 사용하면 다음과 같은 형태로도 가능

Collections.sort(myBookList,

(Book b1, Book b2)-> b1.getPrice() - b2.getPrice());

or

list.sort(Comparator.*comparing*(Book::getPrice)); // 오름차순  
list.sort(Comparator.*comparing*(Book::getPrice).reversed()); // 내림차순

참고 : <http://cwondev.tistory.com/15>

지금까지 대부분의 자바 프로그램은 싱글 코어만 사용되었다.

자바 8이전에는 나머지 코어를 사용하려면 스레드를 사용하는 것이 좋다

🡺 하지만 관리의 복잡성이 발생

🡺1.0에서는 스레드와 락으로 관리

🡺5에서는 스레드 풀, 병렬 실행 컬렉션(atomic)

🡺7에서는 포크/조인 프레임워크 제공

🡺8에서는 스트림 (질의 언어에서 표현식을 처리하는 것 처럼)

🡺 synchronized를 사용하지 않아도 된다.

병렬 관점이 아닌 조금 다른 관점에서 스트림 API를 보면 덕분에 두가지 기능이 생김(위험한생각)

* 메서드에 코드를 전달하는 간결 기법(메서드 레퍼런스와 람다)
* 인터페이스의 디폴트 메서드

메서드에 코드를 전달하는 자바8 기법은 함수형 프로그래밍에서 위력을 발휘한다.

**시작하기 앞서**

* 1. : 자바가 멀티코어 병렬성을 더 쉽게 이용할 수 있도록 진화하는 과정과 개념
  2. : 자바8에서 제공하는 코드를 메서드로 전달하는 기법이 어떻게 강력한 도구다 될수 있는지
  3. : 스트림 API가 어때서 강력하고 새로운 프로그래밍 도구인지
  4. : 디폴트 메서드라는 새로운 자바8의 기능을 인터페이스, 라이브러리의 간결성 유지 및

재컴파일을 줄이는데 어떻게 사용하는지

1.5 : JVM을 구성하는 자바 및 기타언어에서 함수형 프로그래밍의 영향이 어떤지

**1.1 왜 아직도 자바는 변화 하는가?**

**1.1.1 프로그래밍 언어 생태계에서 자바의 위치**

처음부터 많은 유용한 라이브러리를 포함하는 잘 설계된 객체지향 언어로 시작.

처음부터 스레드와 락을 이용한 소소한 동시성도 지원

코드를 JVM 바이트코드로 컴파일 하는 특징 때문에 자바는 인터넷 애플리케이션 프로그램의 주요 언어가 되었다.

🡺JVM 때문에 OS에 종속적이지 않다.

다양한 임베디드 컴퓨팅 분야를 성공적으로 장악하고 있다.

더 다양한 프로그래밍 도구를 제공하고, 다양한 프로그래밍 문제를 더 빠르고 정확하며 쉽게 유지보수할 수 있다는 점이 자바 8이 프로그래머에게 제공하는 개이득.

**1.1.2 스트림 처리**

스트림이란 한번에 한 개씩 만들어지는 연속적인 데이터 항목들의 모임

유닉스 C / stdin 🡺 stdout

Java System.in 🡺 System.out

조립라인은 물리적인 순서로 한 개씩 운반하지만 각각의 작업장에서는 동시에 작업 처리



Java.util.stream 패키지에 스트림 API가 추가 (조립라인 처럼 어떤 항목을 연속적으로 제공하는 기능이라고 단순히 생각하자)

🡺데이터베이스 질의처럼 고수준으로 추상화해서 일련의 스트림으로 만들어 처리

🡺스트림 파이프라인을 이용해서 입력 부분을 여러 CPU 코어에 쉽게 할당할 수 있다.

🡺스레드라는 복잡한 작업을 거치지 않아도 병렬성 획득

**1.1.3 동작 파라미터화로 메서드에 코드 전달하기**

자바8 이전에는 메서드를 다른 메서드로 전달할 방법이 없었다.

자바8에서는 가능함 (동작 파라미터화)

🡺스트림 API는 sort의 동작을 파라미터화하기 위해 compareUsingCustomerId를 넘겨주는 것처럼 어떤 동작을 파라미터화 할 수 있는 코드를 넘겨줄 수 있는 기능 제공 (아직 이해 부족)



**1.1.4 병렬성과 공유 가변 데이터**

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

병렬에서 동시성 해결 하기

**공유 리소스 있는 경우**

🡺Synchronized / critical section (성능 저하)

🡺Atomic operation / CAS(Compare and Swap)

**공유 리소스 없는 경우**

🡺Immutable (불변)

🡺순수 함수

function add(a, b) {

return a + b;

}

**1.1.5 자바가 진화해야 하는 이유**

언어는 하드웨어나 프로그래머 기대의 변화에 부응할 수 있도록 변화 해야 한다.

🡺계속 새로운 기능을 추가하면서 인기 언어를 유지하고 있다.

**1.2 자바함수**

자바 8에서는 함수를 새로운 값의 형식으로 추가 (일급 시민, 일급 객체)

ex) scala의 일급 객체

**1) 변수나 데이터에 할당 할 수 있어야 한다**.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | object Main { | |  |  | |  | def test(): Unit ={ | |  | println("test"); | |  | } | |  |  | |  | def main(args: Array[String]): Unit = { | |  | println("Scala") | |  | **val a = test** | |  |  | |  |  | |  | } | |  | } | |

**2) 객체의 인자로 넘길 수 있어야 한다.**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | object Main { | |  |  | |  | def test1(): Unit = { | |  | println("test1") | |  | } | |  |  | |  | def test2(f: => Unit): Unit = { | |  | f | |  | } | |  |  | |  | def main(args: Array[String]): Unit = { | |  | test2(test1) | |  | } | |  | } | |

**3) 객체의 리턴값으로 리턴할 수 있어야 한다.**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | object TEST { | |  |  | |  | def test1(): Any => Unit = { | |  | return { x => println(x) } | |  | } | |  |  | |  | def main(args: Array[String]): Unit = { | |  | test1()("test") | |  | } | |  | } | |

**왜 함수가 필요 할까?**

🡺프로그래밍 언어의 핵심은 값을 바꾸는 것이다.

🡺자바는 이급 시민 (메서드나 클래스를 자유롭게 전달 못함 / 값으로 전달 못함)

🡺자바8에서 이급 시민 -> 일급 시민으로 바꿀 수 있는 기능 추가

**1.2.1 메서드와 람다를 일급 시민으로**

**1) 메서드 레퍼런스 (::)**

🡺메서드를 인자로 넘긴다.

|  |
| --- |
| File[] hiddenFiles = new File(“.”).listFiles(**File::isHidden**); |

**2) 람다: 익명함수(->)**

자바8에서는 메서드를 일급값으로 취급할 뿐 아니라 람다를 포함하여 함수도 값으로 취급

Utils::add1 같이 직접 구현해서 사용할 수 도 있지만

메서드가 없을 때는 (int x) -> x+1 과 같이 람다 문법을 이용하면 더 간결하게 코드를 구현

**🡺메서드화 되어 있으면 람다를 사용할 필요가 없다. (사용 할 때 선택 기준)**

**1.2.2 코드 넘겨주기**

필터 : 특정 항목을 선택해서 반환하는 동작

자바8 이전

public static List<Apple> filterGreenApples(List<Apple> inventory){  
 List<Apple> result = new ArrayList<>();  
 for (Apple apple: inventory){  
 if ("green".equals(apple.getColor())) {  
 result.add(apple);  
 }  
 }  
 return result;  
}  
  
public static List<Apple> filterHeavyApples(List<Apple> inventory){  
 List<Apple> result = new ArrayList<>();  
 for (Apple apple: inventory){  
 if (apple.getWeight() > 150) {  
 result.add(apple);  
 }  
 }  
 return result;  
}

🡺비슷한 기능이지만 목적에 따라서 메서드가 여러 개 생긴다.

자바8에서는 코드를 인수로 넘겨줄 수 있기 때문에 filter 메서드를 중복으로 구현할 필요가 없다.

public static void main(String[] args) {  
 List<Apple> inventory = Arrays.*asList*(new Apple(80, "green"),  
 new Apple(155, "green"),  
 new Apple(120, "red"));

// 자바8 Predicate 방식  
 List<Apple> greeanApples = *filterApples*(inventory, **FilteringApples::*isGreeApple***); // 호출부  
  
 List<Apple> heavyApples = *filterApples*(inventory, **FilteringApples::*isHeavyApple***); // 호출부  
}  
  
public static boolean isGreeApple(Apple apple) {  
 return "green".equals(apple.getColor());  
}  
  
public static boolean isHeavyApple(Apple apple) {  
 return apple.getWeight() > 150;  
}  
  
static List<Apple> filterApples(List<Apple> inventroy, **Predicate<Apple> p**) {  
 List<Apple> result = new ArrayList<>();  
  
 for (Apple apple : inventroy) {  
 if (**p.test(apple)**) {  
 result.add(apple);  
 }  
 }  
 return result;  
}

🡺핵심은 메서드를 전달할 수 있다는 점이다.

**프레디케이트란 무엇인가?**

수학에서는 인수로 값을 받아 true나 false로 반환하는 함수를 프레디케이트라고 한다.

Function<Apple, Boolean> 같이 구현할수도 있지만 Predicate<Apple>이 더 표준적인 방식

**1.2.3 메서드 전달에서 람다로**

Predicate를 사용하여 메서드를 값으로 전달하는 것은 유용하지만 매번 메서드 정의하는 것은 귀찮다. 이런 경우 람다를 사용

// 자바8 람다 방식  
*filterApples*(inventory, (Apple a) -> "green".equals(a.getColor()));  
*filterApples*(inventory, (Apple a) -> a.getWeight() > 150);  
*filterApples*(inventory, (Apple a) -> a.getWeight() < 80 || "brown".equals(a.getColor()));

🡺한번만 사용할 메서드는 따로 정의할 필요가 없다.

🡺하지만 람다가 몇 줄 이상 길어진다면(복잡해진다면) 익명 람다보다는 메서드를 정의하는 것이 더 명확하다.

// 다음과 같은 내장된 라이브러리를 추가하지 못한것은 병렬성 때문이다.  
Collections.filter(inventory, (Apple a) -> a.getWeight() > 150);

🡺병렬성이 없었다면 다음과 같이 라이브러리 형태로 더 간단하게 제공될수도 있었을 것이다.

🡺대신 비슷한 동작을 수행하는 스트림 API(컬렉션과 비슷하지만 함수형 프로그래머에게 더 익숙한) 를 제공한다. 또한 컬렉션과 스트림 간에 변환할 수 있는 메서드 (map, reduce 등)도 제공

**1.3 스트림**

자바8 이전 컬렉션에서 필터링이 필요한 경우

Map<Currency, List<Transaction>> transactionByCurrencies = new HashMap<>();  
for(Transaction transaction : *transactions*) {  
 if(transaction.getPrice() > 1000) {  
 Currency currency = transaction.getCurrency();  
 List<Transaction> transactionForCurrency = transactionByCurrencies.get(currency);  
 if(transactionForCurrency == null) {  
 transactionForCurrency = new ArrayList<>();  
 transactionByCurrencies.put(currency, transactionForCurrency);  
 }  
 transactionForCurrency.add(transaction);  
 }  
}

🡺많은 코드를 구현해야될 뿐만 아니라 흐름 문장이 많아서 한번에 이해하기도 어렵다.

Map<Currency, List<Transaction>> transactionByCurrencies2 =  
 *transactions*.stream()  
 .filter((Transaction t) -> t.getPrice() > 1000)  
 .collect(*groupingBy*(Transaction::getCurrency)); // 통화로 그룹화함 (Map 형태)

🡺스트림 API를 사용하면 간략하게 해결할 수 있음

스트림 API를 이용하면 컬렉션 API와는 다른 방식으로 데이터를 처리할 수 있다.

컬렉션(외부반복) : 반복과정을 for-each 루프를 이용하여 각 요소를 반복하면서 처리

🡺목록이 많다면 시간이 오래 걸린다. 단일 CPU로 거대한 데이터 처리가 힘듬.

스트림(내부반복) : 라이브러리 내부에서 모든 데이터 처리

🡺멀티코어를 이용하여 빨리 처리할 수 있다.

**1.3.1 멀티스레딩은 어렵다.**

동시성 문제가 발생할 수 있다. (공유 데이터 접근)

자바8에서는 synchronized가 아닌 함수형 프로그래밍 형식의 스트림 기반 병렬성을 이용하도록 권고.



멀티스레딩일 경우 발생할 수 있는 문제점. (sum 공유자원)

자바8은 스트림 API

🡺모호함과 반복적인 코드 문제 해결

🡺멀티코어 활용 어려움 해결

기존 컬렉션 데이터는 반복패턴이 너무 많아서 라이브러리에서 이러한 반복패턴을 제공(동기)

🡺주어진 조건에 따라 데이터 필터링

🡺데이터를 추출 (특정 필드만 추출)

🡺데이터 그룹화 (Map 형태로)

🡺쉽게 병렬화 할 수 있다.

스트림 API가 parallel하게 작업할때는 내부적으로 ForkJoinPool 을 사용하는듯

<http://www.popit.kr/java8-stream%EC%9D%98-parallel-%EC%B2%98%EB%A6%AC/>

컬렉션은 어떻게 데이터를 저장하고 접근할지 중점을 둔다

스트림은 데이터에 어떤 계산을 할것인지 묘사하는 것에 중점을 둔다.

Stream API에서 병렬과 순차처리

// 순차 처리  
inventory.stream().filter((Apple a) -> a.getWeight() > 150)  
 .collect(Collectors.*toList*());  
  
// 병렬 처리  
inventory.parallelStream().filter((Apple a) -> a.getWeight() > 150)  
 .collect(Collectors.*toList*());

**1.4 디폴트 메서드**

자바8에서는 더 쉽게 변화할 수 있는 인터페이스를 만들 수 있도록 디폴트 메서드 추가

디폴트 메서드는 프로그램 구현하는데 도움을 주는 기능이 아니라 미래에 프로그램이 쉽게 변화할 수 있는 환경을 제공

🡺부모 interface를 구현한 100개의 클래스가 있는데 만약 interface에 기능이 추가된다면 100개 클래스에 모두 구현을 해야된다. 이런부분의 영향을 줄어주기 위해 디폴트 메서드 사용.

디폴트 메서드를 이용하면 기존의 코드를 건드리지 않고도 원래의 인터페이스 설계를 자유롭게 확장할 수 있다.

public interface OperateCar {  
 default public int startEngine(EncryptedKey key) {  
 // Implementation  
 }  
}  
public interface FlyCar {  
 default public int startEngine(EncryptedKey key) {  
 // Implementation  
 }  
}  
  
public class FlyingCar implements OperateCar, FlyCar {  
 public int startEngine(EncryptedKey key) {  
 FlyCar.super.startEngine(key);  
 OperateCar.super.startEngine(key);  
 }  
}

🡺다중상속을 피하기 위해 단일상속 하게 했는데 디폴트메서드로 인해 다시 다이아몬드 상속 문제가 발생하게 되었다. ( C++과 동일 )

**1.5 함수형 프로그램에서 가져온 다른 유용한 아이디어**

1) 메서드와 람다를 일급값으로 사용 (이급 -> 일급 으로 변경함으로써 람다를 사용할 수 있고

더 간략한 코딩을 할 수 있다.)

2) 가변 공유 상태가 없는 병렬 실행을 이용 (불변)

자바8에서는 NullPointer 예외를 피할 수 있게 **Optional**<T> 클래스 제공

자바의 switch 는 문자열과 기본값만 이용할 수 있는 반면에 함수형 언어는 보통 패턴 매칭을 포함한 다양한 데이터 형식을 사용 할 수 있다.

스칼라 같은 경우 패턴 매치를 통해 다양한(type, class )등의 어떤 종류의 데이터라도 매칭 가능.

object MatchTest2 extends App {  
 def matchTest(x: Any): Any = x match {  
 case 1 => "one"  
 case "two" => 2  
 case y: Int => "scala.Int"

case \_ => "no match"  
 }  
 println(matchTest("two"))  
}

**CHAPTER2 동작 파라미터화 코드 전달하기**

동작 파라미터화를 이용하면 자주 바뀌는 요구사항에 효과적으로 대응할 수 있다.

동작 파라미터화란 아직은 어떻게 실행할 것인지 결정하지 않은 코드 블록

**2.1 변화하는 요구사항에 대응하기**

요구 사항 : 녹색 사과만 필터링

// 자바8 이전의 방식  
public static List<Apple> filterGreenApples(List<Apple> inventory) {  
 List<Apple> result = new ArrayList<>();  
 for (Apple apple : inventory) {  
 if (**"green".equals(apple.getColor())**) {  
 result.add(apple);  
 }  
 }  
 return result;  
}

변경 사항 : 빨간 사과도 필터링

🡺앞으로 어두운 빨간색, 노란색 등 필요할 때마다 메서드를 추가할 수 없기 때문에 비슷한 코드를 구현할 경우 추상화 시켜라

**2.1.2 두번째 시도 : 색을 파라미터화**

public static List<Apple> filterApplesByColor(List<Apple> inventory, String color) {  
 List<Apple> result = new ArrayList<>();  
 for (Apple apple : inventory) {  
 if (**apple.getColor().equals(color)**) {  
 result.add(apple);  
 }  
 }  
 return result;  
}

🡺파라미터로 색을 받을 수 있도록 변경

호출부

// 파라미터화  
*filterApplesByColor*(inventory, "green");  
*filterApplesByColor*(inventory, "red");

변경 사항 : 150그램 이상의 무게가 나가는 사과를 무거운 사과로 구분

public static List<Apple> filterApplesByWeight(List<Apple> inventory, int weight) {  
 List<Apple> result = new ArrayList<>();  
 for (Apple apple : inventory) {  
 if (apple.getWeight() > weight) {  
 result.add(apple);  
 }  
 }  
 return result;  
}

🡺색처럼 무게도 받을 수 있도록 설정

**문제점 도출**

하지만 대부분 코드가 색 필터링 코드와 중복됨

**2.1.3 세번째 시도 : 가능한 모든 속성으로 필터링**

public static List<Apple> filterApples(List<Apple> inventory, String color, int weight, boolean flag) {  
 List<Apple> result = new ArrayList<>();  
 for (Apple apple : inventory) {  
 if (**flag && apple.getColor().equals(color) ||  
 (!flag && apple.getWeight() > weight)**) {  
 result.add(apple);  
 }  
 }  
 return result;  
}

🡺flag를 통해 구분했다.

호출부

// 플래그 추가  
*filterApples*(inventory, "green", 0, true);  
*filterApples*(inventory, "", 150, false);

**문제점 도출**

🡺요구사항이 바뀌었을 때 유연하게 대응할 수도 없다.

🡺true, false의 의미도 잘 모르겠다.

**2.2 동작 파라미터화**

좀더 유연하게 대응할 수 있는 방법이 절실하다.

수학에서는 인수로 값을 받아 true나 false로 반환하는 함수를 **프레디케이트**라고 한다.

전략 패턴 : 각 알고리즘을 캡슐화하는 알고리즘 패밀리를 정의해둔 다음에 런타임에 알고리즘을 선택하는 기법

전략 패턴은 갑작스런 알고리즘의 변화에도 유연하게 대처가능. 알고리즘을 캡슐화하기 때문이다. 따라서 언제든지 알고리즘이 변경된다면 프로세스의 큰 틀을 바꾸지 않고도 유연한 프로그래밍 가능.

템플릿 메소드과 전략 패턴 차이

-템플릿 메소드 : 보통 공통된 기능은 부모 클래스에 설정해 놓고(큰틀), 서브 클래스에서는 알고리즘 단계에서 유연하게 바뀔 수 있는 부분만 의존

-전략 패턴 : 보통 인터페이스 군으로 설정. 다형성을 이용하여 런타임 시점에 알고리즘 선택

전략패턴에서의 캡슐화는 핵심 로직을 캡슐화 하는 것이고, 템플릿 메서드 패턴의 캡슐화는 일련의 알고리즘 단계들을 캡슐화 하는 것이다.

<http://asuraiv.blogspot.kr/2014/07/template-method-pattern-1_9.html>

**2.2.1 네번째 시도 : 추상적 조건으로 필터링**

첫번재 코드에 비해 유연하고 가독성도 좋아졌고 사용하기도 쉬워졌다.

// 추상조건 필터  
public static List<Apple> abstractFilterApples(List<Apple> inventory, **ApplePredicate p**) {  
 List<Apple> result = new ArrayList<>();  
 for (Apple apple : inventory) {  
 if (p.test(apple)) {  
 result.add(apple);  
 }  
 }  
 return result;  
}

호출부

// 추상화 추가  
*abstractFilterApples*(inventory, new AppleGreenColorPredicate());  
*abstractFilterApples*(inventory, new AppleHeavyWeightPredicate());

**문제점 도출**

아직 필요한 필터 기능이 생길때마다 Appledicate 하위 class를 구현해야 되는 문제가 남아 있다. (번거롭다)

중요 : 유연한 API를 위하여 동작 파라미터화는 꼭 필요 하다.

**2.3 복잡한 과정 간소화**

새로운 동작을 전달하려면 ApplePredicate 인터페이스를 구현하는 여러 클래스를 정의해야 한다.

자바는 클래스의 선언과 인스턴스화를 동시에 수행할 수 있도록 익명 클래스라는 기법을 제공한다.

🡺하지만 익명 클래스가 모든 것을 해결하는 것은 아니다.

**2.3.1 익명 클래스**

🡺자바의 지역 클래스와 비슷한 개념이다. (블록 내부에 선언된 클래스)

🡺익명 클래스를 이용하면 클래스 선언과 인스턴스화를 동시에 할 수 있다.

**2.3.2. 다섯 번째 시도 : 익명 클래스 사용**

// 익명 클래스  
*abstractFilterApples*(inventory, new ApplePredicate() {  
 @Override  
 public boolean test(Apple apple) {  
 return "red".equals(apple.getColor());  
 }  
});

🡺익명 클래스를 이용하여 메서드의 동작을 직접 파라미터화했다.

**문제점 도출**

1) 익명 클래스는 여전히 많은 공간을 차지한다.

2) 많은 프로그래머들이 익명 클래스의 사용에 익숙하지 않다.

🡺결국 객체를 만들고 명시적으로 새로운 동작을 정의하는 메서드를 구현해야 한다는 점은 변하지 않았다.

**2.3.3 여섯 번째 시도 : 람다 표현식 사용**

자바8의 람다 표현식을 이용해서 다음처럼 간단하게 재구현할 수 있다.

// 람다  
*abstractFilterApples*(inventory, (Apple apple) -> "red".equals(apple.getColor()));

그래프로



**2.3.4 일곱 번째 시도 : 리스트 형식으로 추상화**

현재 filterApples는 Apple과 관련한 동작만 수행한다. 하지만 Apple 이외의 다양한 물건도 필터링 작동하도록 리스트 형식을 추상화 할 수 있다.

// 리스트 형식의 추상화  
public static <T> List<T> filter(List<T> list, Predicate<T> p) {  
 List<T> result = new ArrayList<>();  
 for (T e : list) {  
 if (p.test(e)) {  
 result.add(e);  
 }  
 }  
 return result;  
}

호출부

// 리스트 형식으로 추상화  
filter(inventory, (Apple apple) -> "red".equals(apple.getColor()));

자바8 아니면 불가능한 일이다.

.

**2.4 실전 예제**

**2.4.1 Comparator로 정렬하기**

// 정렬 처리  
inventory.sort(new Comparator<Apple>() {  
 @Override  
 public int compare(Apple a1, Apple a2) {  
 return a1.getWeight().compareTo(a2.getWeight());  
 }  
});

**2.4.2 Runnable로 코드 블록 실행하기**

// Runnable 사용  
Thread t = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 System.*out*.println("Hello world");  
 }  
});  
  
// 람다 표현식으로 사용  
Thread t2 = new Thread(() -> System.*out*.println("Hello world"));

**2.3.4 GUI 이벤트 처리하기**

Button button = new Button("Send");  
  
button.setOnAction(new EventHandler<ActionEvent>() {  
 @Override  
 public void handle(ActionEvent event) {  
 label.setText("Sent!!");  
 }  
});  
  
// 람다 방식으로  
button.setOnAction((Action event) -> label.setText("Sent!!"));

**CHAPTER 3 람다 표현식**

일반적으로 동작 파라미터화를 이용하면 더 유연하고 재사용할 수 있는 코드를 만들 수 있다.

람다를 통해 하고 유연한 코드를 구현하는 방법을 설명한다.

**3.1 람다란 무엇인가?**

람다 표현식은 메서드로 전달할 수 있는 익명 함수를 단순화한 것.

**-익명**

보통의 메서드와 달리 일므이 없으므로 익명이다.

**-함수**

메서드처럼 특정 클래스에 종속되지 않으므로 함수라 부른다.

**-전달**

람다 표현식을 메서드 인수로 전달하거나 변수로 저장할 수 있다.

**-간결성**

익명 클래스처럼 많은 자질구레한 코드를 구현할 필요가 없다.

Comparator<Apple> byWeight = new Comparator<Apple>() {  
 @Override  
 public int compare(Apple a1, Apple a2) {  
 return a1.getWeight().compareTo(a2.getWeight());  
 }  
};

람다를 이용한 새로운 코드

Comparator<Apple> byWeight =  
 (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());

람다는 세 부분으로 이루어진다.

**1) 파라미터 리스트**

Comparator의 compare 메서드의 파라미터

**2) 화살표**

화살표(->)는 람다의 파라미터 리스트와 바디를 구분한다.

**3) 람다의 바디**

두 사과의 무게를 비교한다. 람다의 반환값에 해당하는 표현식이다.

1) 2) 3)

**(Apple a1, Apple a2)** -> a1.getWeight().compareTo(a2.getWeight());

자바8에서 지원하는 다섯가지 람다 표현식

1) String 형식의 파라미터 하나를 가지며 **int**를 반환한다. (return은 함축)

(String s) -> s.length()

2) Apple 형식의 파라미터 하나를 가지며 **boolean**을 반환 한다.

(Apple a) -> a.getWeight() > 150

3) int 형식의 파라미터 두개를 가지며 리턴값이 없다. 여러 행의 문장을 포함할 수 있다.

(int x, int y) -> {  
 System.*out*.println(“Result:”);  
 System.*out*.println(x+y);  
}

4) 파라미터가 없으며 int를 반환

() -> 42

5) Apple 형식의 파라미터를 두개 가지며 int를 반환한다.

(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())

**매우 중요!!**

람다의 기본 문법

1) **(parameters) -> expression**

ex)

() -> “Iron Man”

(String s) -> “Iron Man”

**OR(또는)**

2) (**parameters) -> { statements; }**

ex)

() -> {}

() -> { return “Iron Man”; }

(String s) -> { return “Iron Man”; }

(Integer i) -> { return “Alan” + I }

**3.2 어디에, 어떻게 람다를 사용할까?**

람다는 함수형 인페이스(Functional Interface) 라는 문맥에서 람다 표현식을 사용할 수 있다.

**3.2.1 함수형 인터페이스**

앞에서 본 Predicate<T>가 함수형 인터페이스이다. 오직 **하나의 추상 메서드**만 지정하기 때문이다.

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

}

함수형 인터페이스는 정확히 하나의 추상 메서드를 지정하는 인터페이스이다.

🡺많은 디폴트 메서드가 있더라도 추상 메서드가 오직 하나면 함수형 인터페이스이다.

함수형 인터페이스로 뭘 할 수 있을까?

🡺람다 표현식으로 함수형 인터페이스의 추상 메서드 구현을 직접 전달할 수 있으므로 전체 표현식을 **함수형 인터페이스의 인스턴스로 취급 (내부에서 추상 메서드를 구현)**

🡺함수형 인터페이스는 람다 뿐만 아니라 **익명 내부 클래스로도 같은 기능 구현 가능**

public static void main(String[] args) {  
  
 Runnable r1 = () -> System.*out*.println("Hello World 1");  
  
 Runnable r2 = new Runnable() { // 내부 클래스  
 @Override  
 public void run() {  
 System.*out*.println("Hello World 2");  
 }  
 };  
  
 *process*(r1);  
 *process*(r2);  
 *process*(() -> System.*out*.println("Hello World 3"));  
  
}  
  
public static void process(Runnable r) {  
 r.run();  
}

**3.2.2 함수 디스크립터**

함수형 인터페이스의 추상 메서드 **시그너처**는 람다 표현식의 **시그너처** 를 가르킨다.

🡺람다 표현식의 시그너처를 서술하는 메서드를 **함수 디스크립터** 라고 부른다.

Ex) Runnable 인터페이스의 유일한 추상 메서드 run은 인수와 반환값이 없으므로 인수와 반환값이 없는 시그너처로 생각할 수 있다.

Remind!) 람다 표현식

🡺변수에 할당할 수 있다. (일급 시민)

🡺함수형 인터페이스를 인수로 받는 메서드로 전달할 수 있다.

🡺함수형 인터페이스의 추상 메서드와 같은 시그너처를 갖는다.

**왜 함수형 인터페이스를 인수로 받는 메서드에만 람다 표현식을 사용할 수 있을까? (나도 궁금)**

🡺자바 프로그래머가 하나의 추상 메서드를 갖는 인터페이스에 익숙하다

🡺언어를 더 복잡하게 만들지 않는 현재의 방법 선택

**@FunctionalInterface란?**

함수형 인터페이스를 가리키는 어노테이션이다. 추상 메서드가 하나만 선언.

**3.3 람다 활용 : 실행 어라운드 패턴**

아래와 같이 설정과 정리 과정에 둘러 싸여 있는 형태를 **실행 어라운드 패턴**(execute around pattern)

|  |
| --- |
| 초기화/준비 코드 |
| 작업 A |
| 정리/마무리 코드 |

try-with-resources (자동 자원 반환)

public static String processFile() throws IOException {  
 try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {  
 return br.readLine();  
 }  
}

**3.3.1 1단계:동작 파라미터화를 기억하라**

위 코드는 파일에서 한번에 한줄만 읽을 수 있다. 두줄을 읽거나 자주 사용되는 단어를 반환 하려면?

🡺processFile을 동작 파라미터화 시킨다.

String result = *processFile*((BufferedReader br) ->  
 br.readLine() + br.readLine());

🡺BufferedReader를 인수로 받아서 String을 반환 하는 람다가 필요하다.

**3.3.2 2단계:함수형 인터페이스를 이용해서 동작 전달**

BufferedReader -> String과 IOException을 던질 수 있는 시그너처와 일치하는 함수형 인터페이스를 만들어야 한다.

@FunctionalInterface  
public interface BufferedReaderProcessor {  
 String process(BufferedReader b) throws IOException; // 이게 함수 디스크립터 인감?  
}

정의한 인터페이스를 processFile 메서드의 인스로 전달할 수 있다. (일급 객체)

public static String processFile(**BufferedReaderProcessor p**) throws IOException {  
  
}

**3.3.3 3단계:동작 실행!**

이제 BufferedReaderProcessor에 정의된 process 메서드의 시그너처(BufferedReader -> String)와 일치하는 람다를 전달할 수 있다.

public static String processFile(BufferedReaderProcessor p) throws IOException {  
 try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {  
 return p.process(br);  
 }  
}

**3.3.4 4단계:람다 전달**

람다를 이용해서 다양한 동작을 메서드로 전달할 수 있다.

String oneLine = *processFile*((BufferedReader br) -> br.readLine());  
String towLines = *processFile*((BufferedReader br) -> br.readLine() + br.readLine());

**3.4 함수형 인터페이스 사용**

함수형 인터페이스는 오직 하나의 추상 메서드를 지정한다. 함수형 인터페이스의 추상 메서드는 람다 표현식의 시그너처를 묘사한다.

함수형 인터페이스의 추상 메서드 시그너처를 함수 **디스크립터**라고 한다.

내가 이해한거는

@FunctionalInterface 내에 추상 메서드

String process(BufferedReader b) 의 형태와 (반환 타입 String, 인자 타입 BufferedReader)

과 람다 호출하는 형태의 타입과 같아야 한다.

(BufferedReader br) -> br.readLine()) (반환 타입 String, 인자 타입 BufferedReader)

**3.4.1 Predicate**

따로 정의할 필요 없이 바로 사용할 수 있다는 점이 특징이다.

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

}

**3.4.2 Consumer**

제네릭 형식 T를 받아서 void를 반환하는 추상메서드를 정의한다. T 형식의 객체를 인수로 받아서 어떤 동작을 수행하고 싶을 때 Consumer 인터페이스를 사용.

@FunctionalInterface  
public interface Consumer<T> {  
void accept(T t);

}

기본형

public static void main(String[] args) {  
 *forEach*(  
 Arrays.*asList*(1,2,3,4,5),  
 (Integer i) -> System.*out*.println(i)  
 );  
}  
  
public static<T> void forEach(List<T> list, Consumer<T> c) {  
 for(T i : list) {  
 c.accept(i);  
 }

실제 사용 예 TDA)

//선언부

public void etAppUpdateResponse(com.tmoncorp.tda.module.version.domain.Version lastAppActiveVersion, Consumer<Version> addProperty) {

**addProperty.accept(versionInfo);**

}

//호출부

appUpdateResponse.setAppUpdateResponse(

lastAppActiveVersion,

(versionInfo) ->

{

        if (paramVersion.getIsActive() == false) {

               versionInfo.setEssential(true);

              }

}

);

**3.4.3 Function**

제네릭 형식 T를 인수로 받아서 제네릭 형식 R 객체를 반환하는 apply 추상메서드 정의.

입력을 출력으로 매핑하는 람다를 정의할 때 Function 인터페이스를 활용할 수 있다.

(예를들어 사과의 무게 정보를 추출하거나 문자열을 길이와 매핑)

@FunctionalInterface  
public interface Function<T, R> {R apply(T t);

}

**기본형 특화**

자바의 모든 형식은 참조형 + 기본형 이다. 하지만 제네릭 파라미터에는 참조형만 사용할 수 있다.

🡺C#은 제약이 없고, 스칼라는 참조형만 있다.

기본형 🡺 참조형 (박싱)

참조형 🡺 기본형 (언박싱)

박싱과 언박싱이 자동으로 이루어지는 것을 오토박싱

🡺이런 과정들은 메모리 더 소모. 래퍼클래스는 힙에 저장되고 기본형을 가져올대도 메모리 탐색과정 필요

자바8에서는 기본형을 입출력으로 사용하는 상황에서 오토박싱 동작을 피할 수 있도록 **특별한 버전의 함수형 인터페이스를 제공**

내 생각 : 제네릭을 잘 사용하면 좀더 활용범위가 넓다.

|  |  |
| --- | --- |
| **함수형 인터페이스** | **함수 디스크립터** |
| Predicate<T> | T -> boolean |
| Consumer<T> | T -> void |
| Function<T, R> | T -> R |
| Supplier<T> | () -> T |
| UnarayOperator<T> | T -> T |
| BinaryOperator<T> | (T, T) -> T |
| BiPredicate<L, R> | (L, R) -> boolean |
| BiConsumer<T, U> | (T, U) -> void |
| BiFunction<T, U, R> | (T, U) -> R |

**Supplier () -> T**

Supplier<Date> dateSupplier= SupplierFunctionExample::getSystemDate;  
Date systemDate = dateSupplier.get();  
System.*out*.println("systemDate->" + systemDate);

public static Date getSystemDate() {  
 return new Date();  
}

🡺 결과 : Wed Dec 16 19:18:15 IST 2015

**UnarayOperator T -> T**

UnaryOperator<Integer> operator = t -> t \* 2;  
  
System.*out*.println(operator.apply(5));  
System.*out*.println(operator.apply(10));  
System.*out*.println(operator.apply(15));

🡺 10, 20, 30

**BinaryOperator (T, T) -> T**

BinaryOperator<Integer> adder = (n1, n2) -> n1 + n2;  
System.*out*.println(adder.apply(3, 4));

🡺 7

**BiPredicate (L, R) -> boolean**

BiPredicate<Integer, Integer> bi = (x, y) -> x > y;  
System.*out*.println(bi.test(2, 3));

🡺 false

**BiConsumer (T, U) -> void**

BiConsumer<String, String> biConsumer = (x, y) -> {  
 System.*out*.println(x);  
 System.*out*.println(y);  
};  
biConsumer.accept("java2s.com", " tutorials");

java2s.com

tutorials

**BiFunction (T, U) -> R**

BiFunction<String, String,String> bi = (x, y) -> {  
 return x + y;  
};  
  
System.*out*.println(bi.apply("java2s.com", " tutorial"));

Java2s.com tutorial

**예외, 람다, 함수형 인터페이스의 관계**

함수형 인터페이스는 확인된 예외를 던지는 동작을 허용하지 않는다.

🡺try/ catch를 사용하거나 예외를 선언하는 함수형 인터페이스를 직접 정의

1)인터페이스에 예외 직접 선언

@FunctionalInterface  
public interface BufferedReaderProcessor {  
 String process(BufferedReader b) throws IOException;  
}

2)try/catch 사용

Function<BufferedReader, String> f =  
 (BufferedReader b) -> {  
 try {  
 return b.readLine();  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 };

**3.5 형식 검사, 형식 추론, 제약**

람다 표현식 자체에는 람다가 어떤 함수형 인터페이스를 구현하는지의 정보가 포함되어 있지 않다. 따라서 람다 표현식을 더 제대로 이해하려면 람다의 실제 형식을 파악해야 한다.

**3.5.1 형식 검사**

람다가 사용되는 콘텍스트(context)를 이용해서 람다의 형식(type)을 추론할 수 있다.

어떤 콘텍스트(람다가 전달될 메서드 파라미터나 람다가 할당되는 변수 등) 에서 기대되는 람다 표현식의 형식을 **대상 형식(**target type) 이라고 부른다.

List<Apple>heavierthan105g = filter(inventory, (Apple a) -> a.getWeight() > 150);

1. filter 메서드의 선언을 확인

2. filter 메서드는 두 번째 파라미터로 Predicate<Apple> 형식을 기대한다.

3. Predicate<Apple>은 test라는 한 개의 추상 메서드를 정의하는 함수형 인터페이스

4. test 메서드는 Apple을 받아 boolean을 반환하는 함수 디스크립터를 묘사한다.

5. filter 메서드로 전달된 인수는 이와 같은 요구사항을 만족해야 한다.

**3.4.2 같은 람다, 다른 함수형 인터페이스**

대상 형식이라는 특징 때문에 같은 람다 표현식이더라도 호환되는 추상 메서드를 가진 다른 함수형 인터페이스로 사용될 수 있다.

@FunctionalInterface  
public interface Callable<V> {V call() throws Exception;  
}

public interface PrivilegedAction<T> {T run();  
}

**호출부**

public static void main(String[] args) {  
 Callable<Integer> c = () -> 42;  
 PrivilegedAction<Integer> p = () -> 42;  
}

🡺둘다 인수를 받지 않고 제네릭 형식 T를 반환하는 함수를 정의한다. 둘다 유효한 코드

**람다**

콘텍스트를 이용해서 람다의 형식을 추론 filter(inventory, **(Apple a) -> a.getWeight() > 150**);

**다이아몬드 연산자 (<>)**

다이아몬드 연산자로 콘텍스트에 따른 제네릭 형식을 추론 List<String listOfStrings = new **ArrayList<>();**

**특별한 void 호환 규칙**

람다의 바디에 일반 표현식이 있으면 void를 반환하는 함수 디스크립터와 호환이 된다.

물론 파라미터 리스트도 호환되어야 한다.

Predicate<String> p = s -> list.add(s); // boolean  
Consumer<String> b = s -> list.add(s); // void

🡺list.add는 boolean을 반환하지만 void랑도 호환이 된다.

**3.5.3 형식 추론**

자바 컴파일러는 람다 표현식이 사용된 콘텍스트(대상 형식)를 이용해서 람다 표현식과 관련된 함수형 인터페이스를 추론한다.

대상 형식 => 함수 디스크립터를 알 수 있음 => 람다의 시그너처도 추론할 수 있음

*filter*(inventory, (Apple apple) -> "red".equals(apple.getColor()));

🡺  
*filter*(inventory, apple -> "red".equals(apple.getColor()));

🡺컴파일러는 표현식의 파라미터 형식에 접근할 수 있으므로 람다 문법에서 이를 생략할 수 있다.

🡺어떤게 좋은지 정해진 규칙은 없음. 개발자가 가독성 좋은걸로 선택

**3.5.4 지역 변수 사용**

지금까지 람다 표 현식은 인수를 자신의 바디 안에서만 사용했다.

하지만 람다 표현식에서 익명 함수가 하는 것처럼 **자유 변수 (외부 정의된 변수)**를 활용할 수 있다.

🡺람다 캡처링이라고 부른다.

int portNumber = 1337;  
Runnable r = () -> System.*out*.println(portNumber);

인스턴스 변수와 정적 변수를 자유롭게 사용할 수 있다. 그러려면 **지역 변수는 명시적으로 final로 선언**되어 있어야 하거나 **실질적으로 final로 선언된 변수와 똑같이 사용**되어야 한다.

**지역 변수의 제약**

인스턴스 변수는 힙에 저장되는 반면 지역 변수는 스택에 위치한다.

스레드에서 실행 시, 지역 변수 접근하려고 할 수 있기 때문에(지역 변수는 이미 해제되었을 수도 있다.) 그래서 지역 변수의 복사본을 제공한다. 따라서 복사본의 값이 변경되지 않아야 하므로 제약이 생긴 것이다. 또한 값이 변경되면 병렬 처리 때 방해요소가 된다.

**클로저**

람다와 클로저의 공통점은 메서드의 인수로 전달될 수 있으며 자신의 외부 영역의 변수에 접근할 수 있다.

차이점은 람다는 정의된 메서드의 지역 변수의 값은 바꿀 수 없다. (지역 변수 값은 final)

**3.6 메서드 레퍼런스**

메서드 레퍼런스는 특정 람다 표현식을 축약한 것이라고 생각하면 된다.

inventory.sort((a1, a2) -> a1.getWeight().compareTo(a2.getWeight()));  
🡺  
inventory.sort(Comparator.*comparing*(Apple::getWeight));

**3.6.1 요약**

명시적으로 메서드명을 참조함으로써 가독성을 높일 수 있다. (::) 형태로 사용.

Ex)

|  |  |
| --- | --- |
| 람다 | 메서드 레퍼런스 단축표현 |
| (Apple a) -> a.getWeight() | Apple::getWeight |
| () -> Thread.currentThread().dumpStak() | Thread.currentThread()::dumpStack |
| (str, i) -> str.substring(i) | String::substring |
| (String s) -> System.out.println(s) | System.out::println |

🡺메서드 레퍼런스는 새로운 기능이 아니라 하나의 메소드를 참조하는 람다를 편리하게 표현할 수 있는 문법

**메서드 레퍼런스를 만드는 방법 (3가지)**

**1) 정적 메서드 래퍼런스**

🡺예를 들면 Integer의 parseInt 메서드는 Integer::parseInt로 표현할 수 있다.

**2) 다양한 형식의 인스턴스 메서드 레퍼런스**

🡺예를 들어 String의 length 메서드는 String::length로 표현할 수 있다.

**3) 기존 객체의 인스턴스 메서드 레퍼런스**

🡺Transaction expensiveTransaction 이 있으면 expensiveTransaction::getValue 로 표현 가능

List<String> str = Arrays.*asList*("a", "b", "A", "B");  
  
str.sort((s1, s2) -> s1.compareToIgnoreCase(s2)); // 람다 표현식  
str.sort(String::compareToIgnoreCase); // 메서드 레퍼런스

🡺컴파일러는 람다 표현식의 형식을 검사하던 방식과 비슷한 과정으로 메서드 레퍼런스가 주어진 함수형 인터페이스와 호환 하는지 확인한다.

**3.6.2 생성자 레퍼런스**

ClassName::new 처럼 클래스명과 new 키워드를 이용해서 기존 생성자의 레퍼런스를 만들 수 있다. (정적 메서드 레퍼런스를 만드는것과 비슷)

**Supplier () -> T**

Supplier<Apple> c1 = () -> new Apple();  
Apple a1 = c1.get();  
🡺 생성자 레퍼런스 방식 (인자가 없는)  
Supplier<Apple> c2 = Apple::new;  
Apple a2 = c2.get();

**Function T -> R**

Function<Integer, Apple> f1 = (weight) -> new Apple(weight);  
Apple b1 = f1.apply(110);  
🡺 생성자 레퍼런스 방식 (인자가 있는)  
Function<Integer, Apple> f2 = Apple::new;  
Apple b2 = f2.apply(110);

Apple(int weight) 를 호출

// 호출부

List<Integer> weights = Arrays.*asList*(7, 3, 4, 10);  
List<Apple> apples = *map*(weights, Apple::new);  
  
  
public static List<Apple> map(List<Integer> list, Function<Integer, Apple> f) {  
 List<Apple> result = new ArrayList<>();  
 for(Integer e : list) {  
 result.add(f.apply(e));  
 }  
 return result;  
}

**BiFunction (T, U) -> R**

// BiFuntion  
BiFunction<String, Integer, Apple> g1 =  
 (color, weight) -> new Apple(color, weight);  
Apple c1 = g1.apply("green", 110);  
  
BiFunction<String, Integer, Apple> g2 = Apple::new;  
Apple c2 = g2.apply("green", 110);

인스턴스화 하지 않고도 생성자에 접근할 수 있게 다양하게 응용할 수 있다.

static Map<String, Function<Integer, Fruit>> *map* = new HashMap<>();  
  
static {  
 *map*.put("apple", Apple::new);  
 *map*.put("orange", Orange::new);  
}  
  
public static Fruit giveMeFruit(String fruit, Integer weight) {  
 return *map*.get(fruit.toLowerCase())  
 .apply(weight);  
}

3개 이상 생성자일 경우 직접 만들어서 사용해야 한다.

**3.7 람다, 메서드 레퍼런스 활용하기**

동작 파라미터화, 익명 클래스, 람다 표현식, 메서드 레퍼런스등 정리

**3.7.1 1단계:코드 전달**

Comparator 객체를 인수로 받아 비교

public static void main(String... args) {  
 List<Apple> inventory = new ArrayList<>();  
 inventory.addAll(Arrays.*asList*(new Apple(80, "green"), new Apple(155, "green"), new Apple(120, "red")));  
  
 inventory.sort(new AppleComparator());  
}  
  
static class AppleComparator implements Comparator<Apple> {  
 public int compare(Apple a1, Apple a2) {  
 return a1.getWeight().compareTo(a2.getWeight());  
 }  
}

**3.7.2 2단계:익명 클래스 사용**

위 코드는 Comparator 코드를 구현해야 되기 때문에 익명 클래스를 이용하는 것이 좋다.

// 익명 클래스  
inventory.sort(new Comparator<Apple>() {  
 @Override  
 public int compare(Apple a1, Apple a2) {  
 return a1.getWeight().compareTo(a2.getWeight());  
 }  
});

**3.7.3 3단계:람다 표현식 사용**

람다 표현식을 이용하여 경량화된 문법을 이용해서 코드 전달. 함수형 인터페이스를 사용하는 곳 어디서나 람다 표현식 사용 가능.

추상 메서드의 시그너처(함수 디스크립터)는 람다 표현식의 시그너처를 정의 한다.

Comparator는 (T, T) -> int다

// 람다  
inventory.sort((a1, a2) -> a1.getWeight().compareTo(a2.getWeight()));

형식 추론

컴파일러에서 콘텍스트를 활용해서 형식 추론

(Apple a1, Apple a2) 🡺 (a1, a2) 로 줄일 수 있다.

Comparing 메서드를 사용하여 더 간소화

// Comparator의 comparing을 이용하여 더 간소화  
inventory.sort(Comparator.*comparing*((a) -> a.getWeight()));

**3.7.4 4단계:메서드 레퍼런스 사용**

// 메서드 레퍼런스  
inventory.sort(Comparator.*comparing*(Apple::getWeight));

코드만 짧아진 것이 아니라 의미도 명확해졌다.

**3.8 람다 표현식을 조합할 수 있는 유용한 메서드**

간단한 여러 개의 람다 표현식을 조합해서 복잡한 람다 표현식을 만들 수 있다.

디폴트 메서드 사용

**3.8.1 Comparator 조합**

역정렬

// 역정렬  
inventory.sort(Comparator.*comparing*(Apple::getWeight).reversed());

**Comparator 연결**

무게가 같은 두 사과가 존재한다면 어떻게 해야 할까? 이럴 땐 비교 결과를 더 다듬을 수 있는 두번째 Comparator를 만들 수 있다.

// 두번째 정렬  
inventory.sort(Comparator.*comparing*(Apple::getWeight).reversed().thenComparing(Apple::getColor));

🡺thenComparing 메서드로 두번째 비교자를 만들 수 있다.

**3.8.2 Predicate 조합**

Predicate는 복잡한 프레디케이트를 만들 수 있도록 negate, and, or 세가지 메서드를 제공한다.

// negate 빨간 사과가 아니다.  
Predicate<Sorting.Apple> notRedApple = redApple.negate();  
  
// and 빨간 사과이면서 무게가 150이상이다.  
Predicate<Sorting.Apple> redAndHeavyApple = redApple.and(a -> a.getWeight() > 150);  
  
// or 빨간 사과이면서 무게가 150이상 또는 그냥 녹색 사과  
Predicate<Sorting.Apple> redAndHeavyAppleOrGree = redApple.and(a -> a.getWeight() > 150)  
 .or(a -> "green".equals(a.getColor()));

**3.8.3 Function 조합**

andThen, compose 두가지 디폴트 메서드를 제공한다.

**andThen** 메서드는 주어진 함수를 먼저 적용한 결과를 다른 함수의 입력으로 전달하는 함수를 반환

Function<Integer, Integer> f = x -> x + 1;  
Function<Integer, Integer> g = x -> x \* 2;  
Function<Integer, Integer> h = f.andThen(g);  
int result = h.apply(1); // (1 + 1) \* 2 = 4

**Compose** 메서드는 인수로 주어진 함수를 먼저 실행한 다음에 그 결과를 외부 함수의 인수로 제공한다.

Function<Integer, Integer> f = x -> x + 1;  
Function<Integer, Integer> g = x -> x \* 2;  
Function<Integer, Integer> h = f.compose(g);  
int result = h.apply(1); // 1 + (1 \* 2) = 3

🡺andThen과 순서 반대

**3.9 비슷한 수학적 개념**

**3.9.1 적분**

**3.9.2 자바8 람다로 연결**

**3.10 요약**

-람다 표현식은 익명 함수의 일종이다. 이름은 없지만, 파라미터 리스트, 바디 반환 형식을 가지며 예외를 던질 수 있다.

-람다 표현식으로 간결한 코드를 구현할 수 있다.

-함수형 인터페이스는 하나의 추상 메서드만을 정의하는 인터페이스다.

-함수형 인터페이스를 기대하는 곳에서만 람다 표현식을 사용할 수 있다.

-람다 표현식을 이용해서 함수형 인터페이스의 추상 메서드를 즉석으로 제공할 수 있으며 람다 표현식 전체가 함수형 인터페이스의 인스턴스로 취급된다.

-java.util.function패키지는 Predicate<T>, Function<T, R>, Supplier<T>, Consumer<T>, BinaryOperator<T> 등을 포함해서 자주 사용하는 다양한 함수형 인터페이스를 제공한다.

-자바8은 Predicate<T>와 Function<T, R> 같은 제네릭 함수형 인터페이스와 관련한 박싱 동작을 피할 수 있도록 IntPredicate, IntToLongFunction 등과 같은 기본형 특화 인터페이스도 제공한다.

-실행 어라운드 패턴(자원할당, 자원 정리 등 코드 중간에 실행해야 하는 메서드에 꼭 필요한 코드)을 람다와 활용하면 유연성과 재사용성을 추가로 얻을 수 있다.

-람다 표현식의 기대 형식을 대상 형식이라고 한다.

-메서드 레퍼런스를 이용하면 기존의 메서드 구현을 재사용하고 직접 전달할 수 있다.

-Comparator, Predicate, Function 같은 함수형 인터페이스는 람다 표현식을 조합할 수 있는 다양한 디폴트 메서드를 제공한다.

**3장의 용어정리**

**대상형식**

람다 표현식으로 구현하고자 하는 함수형 인터페이스.

Function<String> f = str -> "hello"+str; 에서 **Funtion<String>**이 대상형식이 됨

**메서드 시그너처 and 함수 디스크립터**

메서드 시그너처를 람다 표현식 형태로 표현한 것.

**boolean test(T t)** 가 메서드 시그너처라면 함수 디스크립터로 표현할시 T -> boolean 과 같은 형태로 표현할 수 있음

**PART 2. 함수형 데이터 처리**

**CHATER 4. 스트림 소개**

-질의어 / ( select \* from table where color = “red” 🡺 반복자 필요 없다. );

-멀티코어 / (멀티코어 아키텍쳐 병렬 처리<fork join> 🡺 복잡하고 어렵다. )

컬렉션을 많이 사용하지만 완벽한 컬렉션 관련 연산을 지원하려면 한참 멀었다. (점점 발전..)

🡺ex) 교집합, 합집합 차집합

**Java**

// 합집합

public <T> List<T> union(List<T> list1, List<T> list2) {  
 Set<T> set = new HashSet<T>();  
  
 set.addAll(list1);  
 set.addAll(list2);  
  
 return new ArrayList<T>(set);  
}  
// 교집합  
public <T> List<T> intersection(List<T> list1, List<T> list2) {  
 List<T> list = new ArrayList<T>();  
  
 for (T t : list1) {  
 if (list2.contains(t)) {  
 list.add(t);  
 }  
 }  
  
 return list;  
}

**Java (apache CollectionUtils)**

// 차집합

Collection<String> except = CoolectionUtils.usbtract(list, list2);

// 교집합

Collection<String> intersection = CoolectionUtils.union(list, list2);

// 합집합

Collection<String> union = CoolectionUtils.intersecation(list, list2);

**C# Linq**

int[] first = { 2, 4, 6, 8, 10 };  
int[] second = { 3, 6, 9, 12 };  
  
// 차집합

var except = first.Except(second); // 2, 4, 8, 10  
  
// 교집합  
var intersection = first.Intersect(second); // 6  
  
// 합집합  
var union = first.Union(second); // 12, 10, 9, 8, 6, 4, 3, 2

**4.1 스트림이란 무엇인가?**

-질의어 가능

-병렬 처리 쉽게 가능

**자바7 이하**

// 자바7 스타일  
ArrayList<Dish> menu = new ArrayList<>();  
menu.add(new Dish("스테이크", 500));  
menu.add(new Dish("라면", 450));  
menu.add(new Dish("김치찌개", 300));

// 필터

List<Dish> lowCaloricDishes = new ArrayList<>();  
for (Dish d : menu) {  
 if (d.getColories() < 400) {  
 lowCaloricDishes.add(d);  
 }  
}

// 정렬  
Collections.*sort*(lowCaloricDishes, new Comparator<Dish>() {  
 @Override  
 public int compare(Dish d1, Dish d2) {  
 return Integer.*compare*(d1.getColories(), d2.getColories());  
 }  
});  
  
// 특정 형태로 변환.  
List<String> lowCaloricDishesName = new ArrayList<>();  
for (Dish d : lowCaloricDishes) {  
 lowCaloricDishesName.add(d.getName());  
}

**🡺연산을 여러 번 해야되기 때문에 코드가 길어진다.**

**자바8 (Stream API는 고수준 빌딩 블록)**

// 자바8 스타일  
List<String> lowCaloricDishesName2 = menu.parallelStream() // 병렬  
 .filter(d -> d.getColories() < 400) // 필터  
 .sorted(Comparator.*comparing*(Dish::getColories)) // 정렬  
 .map(Dish::getName) // 특정 형태로 변환  
 .collect(Collectors.*toList*()); // 출력 형태

****

🡺선언형으로 코드 구현

🡺빌딩 블록 연산을 연결해서 복잡한 데이터 처리 파이프라인 생성

**🡺주의 할 것이 빌딩 블록의 순서에 따라서 성능 영향을 미친다.**

**고수준 빌딩 블록**

🡺특정 스레딩 모델에 제한되지 않고 자유롭게 어떤 상황에서든 사용할 수 있다. (병렬성)

**구아바, 아파치, 람다j 왜 생겼을까?**

🡺자바가 지원을 안해서..

**선언형 이란?**

해법을 정의하기보다는 문제를 설명하는 고급언어이다

**그럼 고급언어는?**

사람이 이해하기 쉽게 작성된 프로그래밍 언어

**스트림 API 특징**

**1) 선언형**

🡺이해하기 쉽게 작성되었다. (가독성이 좋아짐)

**2) 조립할 수 있음**

🡺유연성 확장성이 좋아짐

**3)병렬화**

🡺상태가 불변하기 때문에 병렬화가 쉬워지고 그로 인해 성능이 좋아짐

**왜 컬렉션 대신 스트림을 사용할까?**

🡺결국 컬렉션으로는 불가능한 작업을 처리하기 위해

**4.2 스트림 시작하기**

**그래서 스트림이 뭐임?**



**자료의 입출력을 도와 주는 중간매개체**

**데이터 처리 연산** 을 지원하도록 **소스**에서 추출된 **연속된 요소** (잘 안와닿음)

**데이터 처리 연산**

순차적 또는 병렬로 filter, map, reduce, find match, sort (고수준)등을 이용해서 데이터를 연산

(컬렉션의 주제는 데이터고 스트림의 주제는 계산이다)

**소스**

컬렉션, 배열, I/O 등으로부터 데이터를 얻어오는 것이고 (순서 유지)

Files.lines("test.txt")  
Arrays.*stream*(array)

**연속된 요소**

연속된 값 집합의 인터페이스

List<String> stringList = Arrays.*asList*("test", "test2");  
stringList // 소스  
 .stream() // 소스에서 추출된 연속된 요소  
 .filter(x -> x.contains("test")) // 데이터 처리 연산 (파이프 라이닝 구성)  
 .collect(Collectors.*toList*());

🡺스트림 연산

**그 외** / **파이프 라이닝**

대부분의 스트림 연산은 스트림 연산끼리 연결해서 커다란 파이프라인을 구성

(lazy, short circuit)



**그 외2 / 내부 반복**

for loop (외부 반복) 반복 반대.. 자세한건 뒤에서

**결론 스트림 API는 파이프라인을 더 최적화할 수 있는 유연성 제공**

**4.3 스트림과 컬렉션**

컬렉션과 새로운 스트림 모두 **연속된** 요소 형식의 값을 저장하는 자료구조의 인터페이스 제공

🡺**연속된** : 순차적으로 **값**에 접근 (iterator)

**map은 Collection 아님** / 연속된 요소 없기 때문에 entrySet, entryKey 등으로 변환후 작업

new HashMap<>().entrySet().stream()

**컬렉션 :** 추가, 삭제 가능 하며, 메모리에 다 가지고 있는다.

(DVD)

**스트림 :** 추가, 삭제 불가 하며, 요청할 때만 필요한 요소를 계산하는 고정된 자료구조.

(인터넷 스트리밍)

문제) 무한의 수가 저장되어 있는 데이터셋에서 10개를 꺼내서 합을 구하시오

**Collcetion**

List<Integer> list = new ArrayList<>();  
  
for(int i = 0; i < 100000000000000000; i ++) {  
 list.add(i); // 메모리에 적재를 해야 되기 때문에 불가  
}  
  
int sum = 0;  
for(int i = 0; i < 10; i ++) {  
 sum += list.get(i);  
}

**Stream**

int totalSum = IntStream.*range*(0, 100000000000000000)  
 .limit(10)  
 .sum();

**4.3.1 딱 한번만 탐색할 수 있다.**

ex) Stream Deep copy…

탐색된 스트림의 요소는 소비. 다시 탐색하려면 초기 데이터 소스에서 새로운 스트림 생성

List<String> title = Arrays.*asList*("Java8", "In", "Action");  
Stream<String> s = title.stream();  
s.forEach(System.*out*::println);  
s.forEach(System.*out*::println);

**접근시, 해당 에러 나옴**

Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed

**4.3.2 외부 반복과 내부 반복**

컬렉션 : 사용자가 직접 요소를 반복 해야 된다 - **외부 반복** (for-each)

스트림 : 반복을 알아서 처리하고 결과값을 어딘가에 저장해 준다 – **내부 반복**

// for-each (루프 외부 반복)  
List<String> names = new ArrayList<>();  
for(Dish d: menu) {  
 names.add(d.getName());  
}  
  
// iterator (내부적으로 숨겨졌던 반복자를 사용한 외부 반복)  
List<String> names2 = new ArrayList<>();  
Iterator<Dish> iterator = menu.iterator();  
while(iterator.hasNext()) {  
 Dish d = iterator.next();  
 names2.add(d.getName());  
}  
  
// stream (내부반복)  
List<String> names3 = menu.stream().map(Dish::getName).collect(Collectors.*toList*());

🡺내부 반복을 사용하면 투명하게 병렬로 처리하거나 최적화된 다양한 순서로 처리

🡺아무래도 외부 반복은 노출되어 있기 때문에 상태 변화 가능

🡺filter, map 등 순서에 따라 성능이 달라짐

**4.4 스트림 연산**

List<String> name = menu.stream() // 스트림 생성  
 .filter(d -> d.getCalories() > 300) // 중간 연산 (파  
 .map(Dish::getName) // 중간 연산 이  
 .limit(3) // 중간 연산 프 라인 형성)  
 .collect(Collectors.*toList*()); // 최종 연산 (파이프 라인 실행 후 해제)

**4.4.1 중간 연산**

최종 연산 하기 전에 아무 연산도 수행하지 않는다. (lazy)

최종 연산에서 중간연산을 합쳐서 한꺼번에 처리 한다.

// 중간 연산  
List<String> name =  
 menu.stream()  
 .filter(d -> {  
 System.*out*.println("filtering : " + d.getName());  
 return d.getCalories() > 500;  
 })  
 .map(d ->{  
 System.*out*.println("mapping : " + d);  
 return d.getName();  
 })  
 .limit(3)  
 .collect(Collectors.*toList*());

🡺쇼트셔킷 기법 때문에 처음 3개만 선택됨

ex)쇼트셔킷 논리연산자 and, or

🡺filter, map은 서로 다른 연산이지만 한과정으로 병합 ( 루프 퓨전 – 수직적 처리 )

**4.4.2 최종연산**

최종 연산은 보통 List(collect), Integer(count), void(foreach)등 결과가 반환

**4.4.3 스트림 이용하기**

List<String> name = menu.stream() // 1) 스트림 생성  
 .filter(d -> d.getCalories() > 300) // 2) 중간 연산   
 .map(Dish::getName) // 중간 연산   
 .limit(3) // 중간 연산  
 .collect(Collectors.*toList*()); // 3) 최종 연산

빌더 패턴과 비슷

PersonInfo result = personInfoBuilder

.setName("MISTAKE")

.setAge(20)

.setFavoriteAnimal("cat")

.setFavoriteColor("black")

.setFavoriteNumber(7)

// 마지막에 .build() 메소드를 호출해서 최종적인 결과물을 만들어 반환

.build();

**4.5 요약**

-내부 반복지원 (투명성)

-스트림 생성, 중간연산, 최종연산

-중간 연산자는 결과 생성 못함 🡺 최종연산에서 종합

-스트림의 요소는 요청할 때만 계산

**CHATER 5. 스트림 활용**

앞에서 filter를 이용하여 내부 반복을 외부 반복으로 바꾸는 방법을 보았다.

**// 외부 반복**  
List<Dish> vegetarianDishes = new ArrayList<>();  
for (Dish d : menu) {  
 if (d.isVegetarian()) {  
 vegetarianDishes.add(d);  
 }  
}  
  
**// 내부 반복**  
List<Dish> vegetarianDishes2 = menu.stream()  
 .filter(Dish::isVegetarian)  
 .collect(Collectors.*toList*());

**5.1 필터링과 슬라이싱(나눈다)**

**5.1.1 프레디케이트로 필터링**

Stream<T> filter(**Predicate**<? super T> predicate);  
  
@FunctionalInterface  
public interface Predicate<T> {  
 boolean test(T t);  
}

프레디케이트는 불린을 반환 하는 함수이기 때문에 Dish::isVegetarian 을 함수 디스크립션으로 사용 가능

**5.1.2 고유 요소 필터링**

menu.stream()  
 .filter(Dish::isVegetarian)  
 **.distinct()** .forEach(System.*out*::println);

distinct를 이용하여 고유 요소 반환 가능 (hashCode, equals가 같아야 한다)

@Override  
public boolean equals(Object o) {  
 if (this.name.equals(((Dish) o).name))  
 return true;  
 return false;  
}  
  
@Override  
public int hashCode() { return name.hashCode(); }

**5.1.3 스트림 축소**

// filtering  
menu.stream()  
 .filter(d -> d.getCalories() > 300)  
 **.limit(3)** .collect(Collectors.*toList*());

n개의 요소 반환

Set 같이 정렬되지 않은 스트림에 사용되면 limit도 정렬되지 않으 상태로 반환 (숏셔킷 이점)

**5.1.4 요소 건너뛰기**

menu.stream()  
 .filter(d -> d.getCalories() > 300)  
 **.skip(2)** .collect(Collectors.*toList*());

n개의 요소 제외

skip과 limit는 상호 보완적이다.

**5.2 매핑**

객체에서 특정 데이터를 선택하는 작업 (ex SQL 테이블에서 특정 열만 선택)

**5.2.1 스트림의 각 요소에 함수 적용하기**

함수를 적용한 결과가 새로운 요소로 매핑된다 (고치는게 아니라 새로운 버전을 만든다.)

// map  
menu.stream()

.map(Dish::getName)

.collect(Collectors.*toList*());

특정 컬럼의 길이를 알고 싶을때는 map 메서드를 연결해서 구할 수 있다.

// 요리 명의 길이  
menu.stream()

.**map(Dish::getName)**

**.map(String::length)** .collect(Collectors.*toList*());

**5.2.2 스트림 평면화**

최종 목표는 [“Hello”, “World”] 🡺 [“H”, “e”, “l”, “o”, “W”, “r”, “d”]

🡺 map 형태의 반환이 Stream<String>로 되어야 한다.

**참고) Split의 String[]을 반환 한다.**

public String[] split(String regex) {  
 return split(regex, 0);  
}

**첫번째 시도**

strList.stream()  
 .**map(x -> x.split(""))** // stream<String[]> ["H", "e", "l", "l" "o"]["W", "o", "r", "l", "d"] .distinct()  
 .collect(Collectors.*toList*());

map 메소드가 반환한 스트림 형식은 Stream<String[]> 이다.

🡺distinct는 equals, hashCode 비교기 때문에 중복 제거를 할 수 없다.

**두번째 시도**

strList.stream()  
 .map(x -> x.split(""))  
 **.map(Arrays::*stream*)** .distinct()  
 .collect(Collectors.*toList*());

**Arrays::stream** : 문자열을 받아서 스트림을 만드는 메소드

배열 스트림 대신, 문자열 스트림이 필요하다.

map 메소드가 반환한 스트림 형식은 Stream<Stream>이다.

🡺결국 distinct 실패

List<Tmon> list = Arrays.*asList*(new Tmon("심준보", 32), new Tmon("조우현", 32), new Tmon("조우현", 32), new Tmon("송두리", 32));  
  
// map 으로 반환  
List<String> names = list.stream()  
 .map(Tmon::getName) // Stream<String> => 심준보, 조우현, 조우현, 송두리  
 .collect(Collectors.*toList*());  
  
// stream 형태로 반환  
String[] array = names.toArray(new String[names.size()]);  
Stream<String> streamOfName = Arrays.*stream*(array); // Stream<String> => 심준보, 조우현, 조우현, 송두리  
  
// distinct  
List<String> str = streamOfName  
 .distinct().collect(Collectors.*toList*());  
  
for (String s : str) {  
 System.*out*.println(s); // Stream<String> => 심준보, 조우현, 송두리  
}  
  
// split  
List<String> strList = Arrays.*asList*("Hello", "World");  
List<String[]> strArrayList = strList.stream()  
 .map((word -> word.split(""))) // Stream<String[]> => ["H","e","l","l","o"], ["W","o","r","l","d"]  
 .collect(Collectors.*toList*());  
  
// Arrays::stream  
List<Stream<String>> streamList = strList.stream()  
 .map(word -> word.split("")) // Stream<String[]> => ["H","e","l","l","o"], ["W","o","r","l","d"]  
 .map(wordArray -> Arrays.*stream*(wordArray)) // Stream<Stream<String>> => ?  
 .distinct()  
 .collect(Collectors.*toList*()); // List<Stream<String>>  
  
// flatMap  
List<String> streamList2 = strList.stream()  
 .map(word -> word.split("")) // Stream<String[]> => ["H","e","l","l","o"], ["W","o","r","l","d"]  
 .flatMap(wordArray -> Arrays.*stream*(wordArray)) // Stream<Stream<String>> => Stream<String>  
 .distinct()  
 .collect(Collectors.*toList*());

**flatMap 사용**

strList.stream()  
 .map(x -> x.split(""))  
 .flatMap(Arrays::*stream*)  
 .distinct()  
 .collect(Collectors.*toList*());

flatMap은 스트림이 아닌 **스트림의 콘텐츠로 매핑한다**.

즉 flatMap은 하나의 평면화된 스트림을 반환한다.

**5.3 검색과 매칭**

**5.3.1 프레디케이트가 적어도 한 요소와 일치하는지 확인**

anyMatch

if(menu.stream().anyMatch(Dish::isVegetarian)) {  
 System.*out*.println("the menu is (some what) vegetarian friendly!!");  
}

**5.3.2 프레디케이트가 모든 요소와 일치하는지 검사**

allMatch

boolean isHealthy = menu.stream()  
 .allMatch(d -> d.getCalories() < 100);

noneMatch (allMatch 반대)

boolean isHealthy2 = menu.stream()  
 .noneMatch(d -> d.getCalories() >= 1000);

쇼트서킷 : &&, ||

**5.3.3 요소 검색**

**Optional 이란?**

Optional은 값의 존재나 부재 여부를 표현하는 컨테이너 클래스.

🡺컨테이너란 ? 같은 타입의 여러 객체를 저장하는 일종의 집합

🡺null 관련 버그를 피하기 위해

|  |  |
| --- | --- |
| isPresent | Optional이 값이 있으면 true, 없으면 false |
| ifPresent | @FunctionalInterface public interface Consumer<T> { void accept(T t);  }  값이 있으면 주어진 코드 블록 실행 |
| T get() | 존재하면 값을 반환. 없으면 NoSuchElementException발생 |
| T orElse | 값이 있으면 값을 반환, 없으면 기본값을 반환 |

**5.3.4 첫 번째 요소 찾기**

리스트나 정렬된 연속 데이터로 생성된 스트림은 순서가 정해져 있다.

**findFirst vs findAny**

findfirst : 첫번째 요소 (순서 상관 있음)

findAny: 첫번째 요소 (병렬성에서 처럼, 순서를 찾기 어려울 경우 순서상관 없이 첫번째 요소)

**5.4 리듀싱**

**모든 스트림 요소** 를 처리해서 값으로 도출 (폴드 라고도 부른다)

**5.4.1 요소의 합**

List<Integer> numbers = Arrays.*asList*(1, 2, 3, 4, 5);  
**// Java7 이전 방식**

int sum = 0; // 초기값  
for (int x : numbers) {  
 sum += x; // 모든 요소 연산(+)  
}

**// reduce 이용 방식 (람다)**  
int sum2 = numbers.stream()  
 .reduce(0, (a, b) -> a + b); // 초기값, 모든 연산 (+)

int multiply = numbers.stream()  
 .reduce(0, (a, b) -> a \* b);

**// reduce 이용 방식 (메서드 레퍼런스)**

int sum3 = numbers.stream()  
 .reduce(0, Integer::*sum*); // 초기값, 모든 연산 (+)

🡺이점 reduce 방식은 내부 반복으로 병렬이 쉽다.

reduce 파라미터

T reduce(T identity, BinaryOperator<T> accumulator);

BinaryOperator는 BiFunction을 상속받은 FunctionalInterface

@FunctionalInterface  
public interface BiFunction<T, U, R> {  
R apply(T t, U u);

}

@FunctionalInterface  
public interface BinaryOperator<T> extends BiFunction<T,T,T> {  
public static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator) {  
 Objects.*requireNonNull*(comparator);  
 return (a, b) -> comparator.compare(a, b) <= 0 ? a : b;  
 }  
public static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator) {  
 Objects.*requireNonNull*(comparator);  
 return (a, b) -> comparator.compare(a, b) >= 0 ? a : b;  
 }  
}

**자세한 계산 방식은 P. 162 그림 참조**

**응용 문제 number의 값은?**

List<Integer> numbers = Arrays.*asList*(1, 2, 3, 4, 5);  
int number = numbers.stream()  
 .reduce(0, Integer::*min*);

**초깃값 없음**

Optional<Integer> sum = numbers.stream()  
 .reduce((a, b) -> (a + b));

초깃값이 없으면 Optional 사용, 있으면 해당 Type 선언 (당연한건가…)

**5.4.2 최댓값과 최솟값**

**퀴즈 5-3**

// 맵 리듀스 패턴  
int count = menu.stream()  
 .map(d -> 1)

.distinct() // 전체를 1로 매핑했기 때문에 distinct 하면 1됨  
 .reduce(0, (a, b) -> a + b);  
  
  
long count = menu.stream().distinct().count(); // equals, hashcode 를 구현했기 때문에 중복 빼고 계산

**번외>** JAVA8의 Interface

public interface Calculator {  
 public int plus(int pre, int post);  
  
 public int multi(int pre, int post);  
  
 default int execPlus(int pre, int post) {  
 return pre + post;  
 }  
  
 **static int execMulti(int pre, int post) {  
 return pre \* post;  
 }**  
}

사용할 때

**Calculator.execMulti(10, 10);**

**스트림 연산 : 상태 없음과 있음**

**내부 상태 없다.**

map, filter 등은 상태 필요 없다.

🡺단순한 결과 출력 (누적이나 이력이 필요 없다)

**내부 상태 있다. (bound)**

reduce, sum, max 같은 연산은 결과를 누적할 내부 상태가 필요하다.

🡺int나 double과 같은 내부 상태 사용 (크기 한정)

**내부 상태 있다 (unbound)**

sorted, distinct는 과거의 이력을 알고 있어야 한다.

🡺 연산을 수행하는 데 필요한 저장소 크기는 정해져 있지 않다.

따라서 데이터 스트림의 크기가 크거나 무한이라면 문제가 생길 수 있다.

**응용 문제result의 값은?**

참고> int형 범위 **(-2147483648 ~ 2147483647)**

List<Integer> list = Arrays.*asList*(2147483647, 1);  
long result = list.stream().reduce(0, Integer::*sum*);

답 : -2147483648 (오버플로우 발생)

**🡺문제 자주 생길 것 같음 (특히 int 형)**

**숏 셔킷 이점 있는 연산**

|  |  |  |
| --- | --- | --- |
| **연산** | **형식** | **설명** |
| limit | 중간 연산 | 사이즈 이하의 크기를 갖는 새로운 스트림 반환 |
| findFirst | 최종 연산 | 스트림에서 첫번째 요소 반환 |
| findAny | 최종 연산 | 스트림에서 임의의 요소 반환 |
| anyMatch | 최종 연산 | 적어도 한 요소와 일치하는지 확인 |
| allMatch | 최종 연산 | 모든 요소가 일치하는지 확인 |
| noneMatch | 최종 연산 | 모든 요소가 불일치 하는지 확인 |

**상태 있는 언바운드 (스트림의 요소갯수에 따라 무한으로 늘어날 수 있다.)**

|  |  |  |
| --- | --- | --- |
| **연산** | **형식** | **설명** |
| distinct | 중간 연산 | 고유 요소 필터링 (hashCode, equals로 판별) |
| sorted | 중간 연산 | 요소들 정렬 |

**🡺 OOM 발생함**

**상태 있는 바운드 (연산을 수행하는데 필요한 저장소 크기가 정해져 있음)**

|  |  |  |
| --- | --- | --- |
| **연산** | **형식** | **설명** |
| skip | 중간 연산 | n개 요소를 제외한 스트림 반환 |
| limit | 중간 연산 | n개 이하의 크기를 갖는 새로운 스트림 반환 |
| reduce | 최종 연산 | 모든 요소를 조합하여 하나의 결과로 반환 |

**🡺데이터 타입에 따라 크기가 달라진다.**

**🡺Stackoverflow 발생함 (메모리 일정이상 증가 안함)**

**무한 스트림**

|  |  |  |
| --- | --- | --- |
| **연산** | **형식** | **설명** |
| iterate (상태 있는 바운드) | 중간 연산 | 연속된 일련의 값을 만들 때 사용 |
| Generate (상태 없음 ) | 중간 연산 | 연속되지 않은 일련의 값을 만들 때 사용 |

**🡺iterate는 스트림 자체는 언바운드지만, 연산을 수행하는데 필요한 저장소 크기가 정해져 있음**

**5.5 실전 연습**

**내가 푼답**

// 2011년에 일어난 모든 트랜잭션을 찾아 값을 오름차순으로 정리하시오.  
List<Integer> values = transactions.stream()  
 .filter(year -> year.getYear() == 2011)  
 .map(Transcatino::getValue)  
 .sorted()  
 .collect(Collectors.*toList*());  
  
// 거래자가 근무하는 모든 도시를 중복 없이 나열하시오.  
List<String> country = transactions.stream()  
 .map(city -> city.getTrader().getCity())  
 .distinct()  
 .collect(Collectors.*toList*());  
  
// 케임브리지에서 근무하는 모든 거래자를 찾아서 이름순으로 정렬하시오.  
List<String> names = transactions.stream()  
 .filter(city -> city.getTrader().getCity() == "Combridge")  
 .map(name -> name.getTrader().getName())  
 .distinct()  
 .sorted()  
 .collect(Collectors.*toList*());  
  
// 모든 거래자의 이름을 알파벳순으로 정렬해서 반환하시오.  
List<String> alphabets = transactions.stream()  
 .map(name -> name.getTrader().getName().split(""))  
 .flatMap(Arrays::*stream*)  
 .distinct()  
 .sorted()  
 .collect(Collectors.*toList*());  
  
// 밀라노(Milano) 에 거래자가 있는가?  
boolean isMilano = transactions.stream()  
 .map(city -> city.getTrader().getCity())  
 .anyMatch(city -> city.equals("Milan"));  
  
// 케임브리지에 거주하는 거래자의 모든 트랜잭션값을 출력하시오.  
List<Integer> values2 = transactions.stream()  
 .filter(city -> city.getTrader().getCity() == "Combridge")  
 .map(Transcatino::getValue)  
 .collect(Collectors.*toList*());  
  
// 전체 트랜잭션 중 최댓값은 얼마인가?  
int max = transactions.stream()  
 .map(Transcatino::getValue)  
 .reduce(0, Integer::*max*);  
  
// 전체 트랜잭션 중 최소값은 얼마인가?  
Optional min = transactions.stream()  
 .map(Transcatino::getValue)  
 .reduce(Integer::*min*);

**5.6 숫자형 스트림**

// 숫자형 스트림  
int calories = menu.stream()  
 .map(Dish::getCalories)  
 .reduce(0, Integer::*sum*);

스트림 요소 형식은 Stream<Integer>이지만 sum 메서드가 없는 이유는 Stream<Dish> 형식의 요소는 sum을 수행할 수 없기 때문에.

🡺스트림 API 숫자 스트림을 효율적으로 처리하도록 **기본 특화 스트림** 을 제공

**5.6.1 기본형 특화 스트림**

**IntStream, DoubleStream, LongStream** 제공

**숫자 스트림으로 매핑**

int calories = menu.stream()  
 .mapToInt(Dish::getCalories) // IntStream 반환  
 .sum();

IntStream != Strema<Integer>

**객체 스트림으로 복원하기**

IntStream intStream = menu.stream().mapToInt(Dish::getCalories);  
Stream<Integer> stream = intStream.boxed();

**기본값:OptionalInt**

OptionalInt, OptionalDouble, OptionalLong

OptionalInt maxCalories = menu.stream()  
 .mapToInt(Dish::getCalories)  
 .max();  
  
int max = maxCalories.orElse(1);

min,max는 초기값에 따라서 잘못된 결과가 도출될 수 있음. orElse 로 명시 설정 가능

Ex) reduce(0, Integer::min);

**5.6.2 숫자 범위**

IntStream evenNumbers = IntStream.range(1, 10).filter(i -> i % 2 == 0);  
System.out.println(evenNumbers.count()); **// 결과 4**  
IntStream evenNumbers2 = IntStream.rangeClosed(1, 10).filter(i -> i % 2 == 0);  
System.out.println(evenNumbers2.count()); **// 결과 5**

Range와 rangeClosed 차이점은 마지막 숫자 포함 유/무 (range는 미포함, rangeClosed 포함)

**5.6.3 숫자 스트림 활용타고라스 수**

패스

**5.7 스트림 만들기**

다양한 방식으로 스트림 만들기

**5.7.1 값으로 스트림 만들기**

Stream<String> stream = **Stream.of**("Java 8 ", "Lambdas ", "In ", "Action");  
stream.map(String::toUpperCase).forEach(System.out::println);

**5.7.2 배열로 스트림 만들기**

int[] numbers = {2, 3, 5, 7, 11, 13};  
int sum = **Arrays.stream**(numbers).sum();

**5.7.3 파일로 스트림 만들기**

long uniqueWords = 0;  
try (Stream<String> lines = **Files.lines**(Paths.get("data.txt"), Charset.defaultCharset())) {  
 uniqueWords = lines.flatMap(line -> Arrays.stream(line.split(" ")))  
 .distinct()  
 .count();  
  
} catch (IOException e) {  
}

https://homoefficio.github.io/2016/08/06/Java-NIO%EB%8A%94-%EC%83%9D%EA%B0%81%EB%A7%8C%ED%81%BC-non-blocking-%ED%95%98%EC%A7%80-%EC%95%8A%EB%8B%A4/

**5.7.4 함수로 무한 스트림 만들기**

**Stream.iterate, Stream.generate**를 이용하여 **무한 스트림을** 만들 수 있다..

// iterate

**Stream.iterate**(0, n -> n + 2)  
 .limit(10)  
 .forEach(System.out::println);

@FunctionalInterface  
public interface UnaryOperator<T> extends Function<T, T> {

}

@FunctionalInterface  
public interface Function<T, R> {R apply(T t);

}

// generate  
**Stream.generate**(Math::random)  
 .limit(10)  
 .forEach(System.out::println);

@FunctionalInterface  
public interface Supplier<T> {  
 T get();  
}

**내부 상태 있다 (unbound)**

🡺 연산을 수행하는 데 필요한 저장소 크기는 정해져 있지 않다.

따라서 데이터 스트림의 크기가 크거나 무한이라면 문제가 생길 수 있다.

병렬 코드에서 공급자에 상태가 있으면 안전하지 않다.

iterate는 불변이였는데 generate는 가변으로 처리할 수 있기 때문에 병렬처리 하면 올바른 결과를 못 얻을 수 있다.

🡺iterate, generate는 unbound 이기 때문에 크기 제한 안하면 최종 연산 수행 시 아무 결과도 계산되지 않는다. (무한 반복됨 위험!!)

**CHATER 6. 스트림으로 데이터 수집**

-컬렉션(Collection)

-컬렉터(Collector)

-컬렉트(collect)

List<Dish> list = stream().collect(Collectors.toList());

최종연산 컬렉트(collect)는 매개변수인 컬렉터(Collectors.toList)를 사용해서 스트림의

항목을 컬렉션(List<Dish> list) 으로 재구성할 수 있다.

**6.1 컬렉터란 무엇인가?**

List<Dish> vegetarianDishes2 = menu.stream()  
 .filter(Dish::isVegetarian)  
 .collect(Collectors.toList());

Collector인터페이스는 스트림의 요소를 어떤 식으로 도출할지 지정한다.

Map<Currency, List<Traction>> transactionsByCurrencies =  
 transactions.stream().collect(groupingBy(Transaction::getCurrency));

GroupingBy를 이용하여 Map형태로도 만들 수 있다.

**6.1.1 고급 리듀싱 기능을 수행하는 컬렉터**

-훌륭하게 설계된 함수형 API는 높은 수준의 조합성과 재사용성을 꼽을 수 있다.

-collect는 내부적으로 리듀싱 연산이 일어 난다.

🡺최종 결과를 저장하는 자료구조에 값을 누적한다.

🡺Collector 인터페서드를 어떻게 구현하느냐에 따라 어떤 리듀싱 연산을 수행할지 결정된다.

ex) Collecotrs.toList()

**6.1.2 미리 정의된 컬렉터**

Collectors 에서 제공하는 기능 크게 3가지

1. 스트림 요소를 하나의 값으로 리듀스하고 요약 (count, sum, min, max등)
2. 요소 그룹화 (map, list)
3. 요소 분할 (boolean으로 구분)

**6.2 리듀싱과 요약**

long howManyDishes = menu.stream().collect(Collectors.counting()); // Collectors  
  
long howManyDishes2 = menu.stream().count(); // Stream

컬렉터로 스트림의 모든 항목을 하나의 결과로 합칠 수 있다. (내부적으로 reducing 사용)

**6.2.1 스트림값에서 최댓값과 최솟값 검색**

Comparator<Dish> dishCaloriesComparator =  
 Comparator.comparingInt(Dish::getCalories);  
  
// 최댓값  
Optional<Dish> mostCalorieDish =  
 menu.stream()  
 .collect(Collectors.maxBy(dishCaloriesComparator));

Comparator를 구현해서 Collectors.maxBy로 전달

**6.2.2 요약 연산**

스트림에 있는 객체의 숫자 필드의 **합계나 평균** 등도 리듀싱 기능이 자주 사용된다. (요약 연산)

// 합계  
int totalCalories = menu.stream().collect(Collectors.summingInt(Dish::getCalories));  
// 평균  
double avgCalories = menu.stream().collect(Collectors.averagingInt(Dish::getCalories));  
// IntSummaryStatistics (count, sum, min, average, max 다포함)  
IntSummaryStatistics menuStatistics = menu.stream().collect(Collectors.summarizingInt(Dish::getCalories));

**6.2.3 문자열 연결**

// 문자열 연결  
String shortMenu = menu.stream().map(Dish::getName).collect(Collectors.joining(", "));

**연산자 (Operator)**

|  |  |
| --- | --- |
| **연산자** | **종류** |
| **단항 연산자 (피연산자 하나)** | **++, --, +(부호), -(부호)** |
| **이항 연산자 (피연산자 두개)** | **+, -, \*, /, %, <<, >>, >, <=, <=, ==, !=**  **&&, ||** |
| **삼항 연산자 (???)** | **? : (조건식 ? 참 : 거짓)** |

**-피연산자 : 그 연산의 대상이 되는 것**

**단항 연산자 ( UnarayOperator 대상형식 T -> T )**

UnaryOperator<Integer> operator = t -> t \* 2;  
  
System.*out*.println(operator.apply(5));  
System.*out*.println(operator.apply(10));  
System.*out*.println(operator.apply(15));

🡺 10, 20, 30

**이항 연산자 ( BinaryOperator 대상형식 (T, T) -> T )**

BinaryOperator<Integer> adder = (n1, n2) -> n1 + n2;  
System.*out*.println(adder.apply(3, 4));

🡺 7

**항등 함수**

**f(x) = x 일 때, 함수 f가 x의 항등함수이다.**



**6.2.4 범용 리듀싱 요약 연산**

int totalCalories = menu.stream().collect(Collectors  
 .reducing(0, Dish::getCalories, (i, j) -> i + j));

-Static 팩토리 메서드로도 정의할 수 있다.

-특화 컬렉터를(maxBy, Sum 등) 사용한 이유는 프로그래밍적 편의성

**reducing(0, Dish::getCalories, (i, j) -> i + j));**

1. 초기값
2. 변환 함수
3. BinaryOperator (T, T) -> T

**collect와 reduce 차이**

둘다 기능은 비슷하지만 (스트림 요소들을 다른 방식으로 결합)

-collect는 도출하려는 결과를 누적하는 컨테이너를 바꾸도록 설계된 메서드.

-reduce는 두 값을 하나로 도출하는 불변형 연산.

🡺 reduce를 잘못 구현하면 병렬수행할 수가 없다.

🡺 병렬성을 확보하려면 collect 메서드로 리듀싱 연산을 구현하는게 좋다

**컬렉션 프레임워크 유연성 : 같은 연산도 다양한 방식으로 수행할 수 있다.**

int totalCalories = menu.stream().map(Dish::getCalories).reduce(Integer::sum).get();



🡺orElse, orElseGet 등을 사용하는 것이 좋다.

**자신의 상황에 맞는 최적의 해법 선택**

// CASE1

int totalCalories = menu.stream().collect(Collectors  
 .reducing(0, Dish::getCalories, (i, j) -> i + j));

// CASE2

int totalCalories = menu.stream().collect(Collectors

.reducing(0, Dish::getCalories, Integer::sum));

// CASE3

int totalCalories = menu.stream().mapToInt(Dish::getCalories).sum();

CASE 1 : reducing 메서드 사용 (람다)

CASE 2 : reducing 메서드 사용 (메서드 레퍼런스)

CASE 3 : 가독성이 가장 좋고, mapToInt를 이용하여 자동 언박싱 과정을 피함으로 성능까지 좋음

**6.3 그룹화**

Map<Dish.Type, List<Dish>> dishesByType =  
 menu.stream().collect(Collectors.groupingBy(Dish::getType));

🡺 {FISH=[prawns, salmon], MEAT=[pork, beef, chicken], OTHER=[french fries, rice, season fruit, pizza]}

🡺기준으로 스트림이 그룹화되므로 이를 **분류 함수** 라고 한다.

Map<CaloricLevel, List<Dish>> dishesByCaloriceLevel = menu.stream().collect(  
 Collectors.groupingBy(dish -> {  
 if (dish.getCalories() <= 400) {  
 return CaloricLevel.DIET;  
 } else if (dish.getCalories() <= 700) {  
 return CaloricLevel.NORMAL;  
 } else {  
 return CaloricLevel.FAT;  
 }  
 }));

람다 표현식으로도 로직을 구현할 수 있다.

**6.3.1 다수준 그룹화**

Map<Dish.Type, Map<CaloricLevel, List<Dish>>> dishesByTypeCaloricLevel =  
 menu.stream().collect(  
 Collectors.groupingBy(Dish::getType,  
 Collectors.groupingBy(dish -> {  
 if (dish.getCalories() <= 400) {  
 return CaloricLevel.DIET;  
 } else if (dish.getCalories() <= 700) {  
 return CaloricLevel.NORMAL;  
 } else {  
 return CaloricLevel.FAT;  
 }  
 })));

groupingBy 메서드 내에 두번째 기준을 정의하는 내부 groupingBy를 전달해서 두 수준으로 스트림의 항목을 그룹화 할 수 있다. (n 수준 그룹화)

|  |  |  |  |
| --- | --- | --- | --- |
| 칼로리 종류 | FISH | MEAT | OTHER |
| DIET | prawns | chicken | Pizza  Fries |
| NORMAL | salmon | beef | fruit  rice |
| FAT |  | pork |  |

**6.3.2 서브그룹으로 데이터 수집**

Map<Dish.Type, Long> typesCount = menu.stream()  
 .collect(Collectors.groupingBy(Dish::getType, Collectors.counting()));

🡺{FISH=2, MEAT=3, OTHER=5}

요리 종류 중 가장 칼로리 높은 음식

Map<Dish.Type, Optional<Dish>> mostCaloricByType =  
 menu.stream()  
 .collect(Collectors.groupingBy(Dish::getType,

Collectors.maxBy(Comparator.comparingInt(Dish::getCalories))));

🡺{FISH=Optional[prawns], MEAT=Optional[pork], OTHER=Optional[pizza]}

groupingBy(f) 는 groupingBy(f, toList()) 의 축약

public static <T, K> Collector<T, ?, Map<K, List<T>>>  
groupingBy(Function<? super T, ? extends K> classifier) {  
 return *groupingBy*(classifier, *toList*());  
}

**컬렉터 결과를 다른 형식에 적용하기**

Map<Dish.Type, Dish> mostCaloricByType2 =  
 menu.stream()  
 .collect(groupingBy(Dish::getType, // 분류 함수   
 collectingAndThen(  
 maxBy(comparingInt(Dish::getCalories)), // 감싸인 컬렉터

Optional::get))); // 변환 함수

-collectingAndThen은 적용할 컬렉터와 변환 함수를 인수로 받아 다른 컬렉터를 반환한다.

-반환되는 컬렉터는 기존 컬렉터의 래퍼 역할.

-collect의 마지막 과정에서 변환 함수로 자신이 반환하는 값을 매핑.

-리듀싱 컬렉터는 절대 Optional.empty()를 반환하지 않는다.

**groupingBy와 함께 사용하는 다른 컬렉터 예제**

Map<Dish.Type, Integer> totalCaloriesByType =  
 menu.stream().collect(groupingBy(Dish::getType, summingInt(Dish::getCalories) ));

같은 그룹으로 분류된 모든 요소에 리듀싱 작업을 수행할 때는 팩토리 메서드 groupingBy 두 번째 인수로 전달한 컬렉터를 사용한다.

Map<Dish.Type, Set<CaloricLevel>> caloricLevelsByType =  
 menu.stream().collect(  
 groupingBy(Dish::getType,  
 mapping(dish -> {  
 if (dish.getCalories() <= 400) {  
 return CaloricLevel.DIET;  
 } else if (dish.getCalories() <= 700) {  
 return CaloricLevel.NORMAL;  
 } else {  
 return CaloricLevel.FAT;  
 }  
  
 }, toSet()))  
 );

Mapping 메서드도 groupingBy와 자주 사용된다.

toSet을 이용하면 리스트가 아닌 집합으로 스트림의 요소가 누적된다.

*mapping*(dish -> {  
 if (dish.getCalories() <= 400) {  
 return CaloricLevel.*DIET*;  
 } else if (dish.getCalories() <= 700) {  
 return CaloricLevel.*NORMAL*;  
 } else {  
 return CaloricLevel.*FAT*;  
 }  
  
}, *toCollection*(HashSet::new))));

toCollection을 이용하면 원하는 방식를 제거 가능

**6.4 분할**

분할은 분할 함수라 불리는 프레디케이트를 분류 함수로 사용하는 특수한 그룹화기능이다.

List<Dish> vegetarianDishes =  
 menu.stream().collect(partitioningBy(Dish::isVegetarian)).get(true);  
  
List<Dish> vegetarianDishes2 =  
 menu.stream().filter(Dish::isVegetarian).collect(Collectors.toList());

collect에서 partitioningBy(분할 함수)를 사용 하거나 filter(프레디케이트)로 필터하거나 같은 결과를 얻을 수 있다.

**6.4.1 분할의 장점**

분할 함수가 반환하는 참, 거짓 두 가지 요소의 스트림 리스트를 모두 유지한다는 것이 분할의 장점이다.

Map<Boolean, Map<Dish.Type, List<Dish>>> vegetarianDishesByType =  
 menu.stream().collect(  
 partitioningBy(Dish::isVegetarian,  
 groupingBy(Dish::getType)));

분할은 groupingBy 대신 partitioningBy를 사용한다.

**6.4.2 숫자를 소수와 비소수로 분할하기**

**static factory method**

일반 생성자

public Blog();  
public Blog( String powerBloggerID );

스태틱 팩토리 메서드 (생성자와 달리 자기 나름의 이름을 가질 수 있다.)

public static Blog **newBlogInstance**();  
public static Blog **newPowerBlogInstance**( String powerBloggerID );

**6.5 Collector 인터페이스**

public interface Collector<T, A, R> {  
Supplier<A> supplier();BiConsumer<A, T> accumulator();BinaryOperator<A> combiner();  
Function<A, R> finisher();  
Set<Characteristics> characteristics();

}

-T는 수집될 스트림 항목의 제네릭 형식

-A는 누적자, 즉 중간 결과를 누적하는 객체의 형식

-R은 수집 연산 결과 객체의 형식

**6.5.1 Collecotr 인터페이스의 메서드 살펴 보기**

네 개의 메서드는 collect 메서드에서 실행하는 함수를 반환

다섯 번째 메서드 characteristics는 collect 메서드가 어떤 최적화를 이용해서 리듀싱 연산을 수행할 것인가 힌트 특성 집합

**Supplier<A> supplier 메서드 : 새로운 결과 컨테이너 만들기**

public Supplier<List,T>> supplier() {  
 return () -> new ArrayList<T>()  
}  
  
public Supplier<List<T>> supplier() {  
 return ArrayList::new;  
}

supplier는 수집 과정에서 빈 누적자 인스턴스를 만드는 파라미터가 없는 함수.

ex)만약 ToList가 비어 있을 경우 비어있는 스트림의 수집 과정의 결과가 될 수 있다.

**BiConsumer<A,T> accumulator 메서드 : 결과 컨테이너에 요소 추가 하기**

public BiConsumer<List<T>, T> accumulator() {  
 return (list, item) -> list.add(item);  
}  
  
public BiConsumer<List<T>, T> accumulator() {  
 return List::add;  
}

스트림에서 n번째 요소를 탐색할 때 두 인수, 즉 누적자(n-1개 수지상ㅌ)와 n 번째 요소를 함수에 적용

**Function<A, R> finisher 메서드 : 최종 변환값을 결과 컨테이너로 적용하기**

public Function<List<T>, List<T>> finisher() {  
 return Function.*identity*();  
}

스트림 탐색을 끝내고 누적자 객체를 최종 결과로 변환하면서 누적 과정을 끝낼 때 호출할 함수를 반환

**BinaryOperator<A> combiner 메서드 : 두 결과 컨테이너 병합 (병렬과 연관이 깊다)**

public BinaryOperator<List<T>> combiner() {  
 return (list1, list2) -> {  
 list1.add(list2);  
 return list1;  
 }  
}

스트림의 서로 다른 서브파트를 병렬로 처리할 때 누적자가 이 결과를 어떻게 처리할지 정의한다.

**Set<Characteristics> characteristics 메서드**

컬렉터의 연산을 정의하는 Characteristics 형식의 불변 집합을 반환한다.

Characteristics는 스트림을 병렬로 리듀스할 것인지 병렬로 리듀스 한다면 어떤 최적화를 선택해야 할지 힌트를 제공한다.

Characteristics는 다음 세 항목을 포함하는 열거형이다.

**1)UNORDERED**

리듀싱 결과는 스트림 요소의 방문 순서나 누적 순서에 영향을 받지 않는다.

**2)CONCURRENT**

다중 스레드에서 accumulator 함수를 동시에 호출할 수 있으며 이 컬렉터는 스트림의 병렬 리듀싱을 수행할 수 있다.

UNORDERED를 함께 설정하지 않았다면 정렬되지 않은(set) 상황에서만 병렬 리듀싱 수행가능

**3)IDENTITY\_FINISH**

finisher 메서드가 반환하는 함수는 단순히 identity를 적용할 뿐이므로 이를 생략할 수 있다.

따라서 리듀싱 과정의 최종 결과로 누적자 객체를 바로 사용할 수 있다.

**6.5.2 응용하기**

public class ToListCollector<T> implements Collector<T, List<T>, List<T>> {  
  
 @Override  
 public Supplier<List<T>> supplier() {  
 return ArrayList::new;  
 }  
  
 @Override  
 public BiConsumer<List<T>, T> accumulator() {  
 return List::add;  
 }  
  
 @Override  
 public Function<List<T>, List<T>> finisher() {  
 return Function.*identity*();  
 }  
  
 @Override  
 public BinaryOperator<List<T>> combiner() {  
 return (list1, list2) -> {  
 list1.addAll(list2);  
 return list1;  
 };  
 }  
  
  
 @Override  
 public Set<Characteristics> characteristics() {  
 return Collections.*unmodifiableSet*(EnumSet.*of*(  
 *IDENTITY\_FINISH*, *CONCURRENT* ));  
 }  
}

List<Dish> dishes = menu.stream().collect(new ToListCollector<Dish>());

toList는 스태틱 팩토리 메서드이지만 ToListCollector는 new로 인스턴스화 한다는 점이 다르다.

**6.6 커스텀 컬렉터를 구현해서 성능 개선하기**

생략

**6.7 요약**

-collect는 스트림의 요소를 요약 결과로 누적하는 다양한 방법(컬렉터)을 인수로 갖는 최종 연산

-스트림 요소를 하나의 값으로 리듀스하고 요약하는 컬렉터뿐 아니라 최솟값, 최댓값, 평균값을 계산하는 컬렉터 등이 미리 정의되어 있다.

-미리 정의된 컬렉터인 groupingBy로 스트림의 요소를 **그룹화** 할 수 있다.

-partitioningBy로 스트림의 요소를 **분할**할 수 있다.

-컬렉터는 다수준의 그룹화, 분할, 리듀싱 연산에 적합하게 설계되어 있다.

-Collector 인터페이스에 정의된 메서드를 구현해서 커스텀 컬렉터 개발할 수 있음.

**CHATER7. 병렬 데이터 처리와 성능**

자바7은 더 쉽게 병렬화를 수행하면서 에러를 최소화할 수 있도록 **포크/조인 프레임워크** 라는 기능을 제공한다.

**7.1 병렬 스트림**

parallelStream을 호출하면 **병렬 스트림** 이 생성된다.

병렬 스트림이란 각각의 스레드에서 처리할 수 있도록 스트림 요소를 여러 청크로 분할한 스트림이다.

**전통적 자바방식**

public static long iterativeSum(long n) {  
 long result = 0;  
 for (long i = 1L; i <= n; i++) {  
 result += i;  
 }  
 return result;  
}

**스트림 연산 방식**

public static long sequentialSum(long n) {  
 return Stream.*iterate*(1L, i -> i + 1)  
 .limit(n)  
 .reduce(0L, Long::*sum*);  
}

n (limit)가 커진다면 병렬 처리하는 것이 좋다.

동기화는? 몇 개의 스레드를? 결과는 어떻게 더할까?

🡺병렬 스트림을 이용하면 걱정 끝

**7.1.1 순차 스트림을 병렬 스트림으로 변환하기**

public static long parallerSum(long n) {  
 return Stream.*iterate*(1L, i -> i + 1)  
 .limit(n)  
 .parallel() // 병렬 스트림으로 변환  
 .reduce(0L, Long::*sum*);  
}



<병렬 리듀싱 연산>

여러 청크로 분할되어 병렬로 수행 후, 마지막으로 리듀싱 연산으로 생성된 부분결과를 다시 리듀싱 연산으로 합쳐서 전체 스트림의 리듀싱 결과를 도출한다.

.filter()  
.sequential()  
.map()  
**.parallel()**  
.reduce();

Parallel과 sequential 두 메서드 중 최종적으로 호출된 메서드가 전체 파이프라인에 영향을 미친다.

**병렬 스트림에서 사용하는 스레드 풀 설정**

기본적으로 ForkJoinPoll은 프로세서 수,

즉 Runtime.getRuntime().availableProcessors()가 반환하는 값에 상응하는 스레드를 갖는다.

// CPU 코어의 수만큼 최대 스레드를 사용하는 스레드풀을 생성

Runtime**.**getRuntime**().**availableProcessors**()**

🡺기본값 권장

**7.1.2 스트림 성능 측정**

// 반복 Iterative sum done in: 7 msecs (생각보다 빠름)  
public static long iterativeSum(long n) {  
 long result = 0;  
 for (long i = 1L; i <= n; i++) {  
 result += i;  
 }  
 return result;  
}  
  
// 순차 Sequential sum done in: 118 msecs  
public static long sequentialSum(long n) {  
 return Stream.*iterate*(1L, i -> i + 1)  
 .limit(n)  
 .reduce(0L, Long::*sum*);  
}  
  
// 병렬 Parallel sum done in: 399 msecs (실망...)  
public static long parallerSum(long n) {  
 return Stream.*iterate*(1L, i -> i + 1)  
 .limit(n)  
 .parallel() // 병렬 스트림으로 변환  
 .reduce(0L, Long::*sum*);  
}

for 루프는 저수준으로 동작하며 기본값을 박싱하거나 언박싱할 필요가 없으므로 수행 속도가 빠르다.

**왜 병렬이 느릴까?**

-iterate가 박싱된 객체를 생성하므로 이를 다시 언박싱하는 과정이 필요했다.

-iterate는 병렬로 실행될 수 있도록 독립적인 청크로 분할하기가 어렵다.

Iterate는 본질적으로 순차적이기 때문에 청크로 분할하기가 어렵다.

🡺리듀싱 연산이 수행되지 않는다. (전체 숫자를 모른다.)

이처럼 병렬 프로그래밍은 까다롭다.

🡺parallel 메서드 호출 시, 내부적으로 어떤 일이 일어나는지 꼭 이해해야 한다. (테스트도 해보고)

**더 특화된 메서드 사용**

LongStream.rangeClosed

-LongStream.rangeClosed 는 기본형 long을 사용하기 때문에 박싱 언박싱 오버헤드 없음.

-LongStream.rangeClosed는 쉽게 청크로 분할할 수 있는 숫자 범위를 생산한다.

ex) 1-5, 6-10, 11-15, 16-20 범위로 숫자 분할

// 순차 Sequential Ranged sum done in 8 msecs  
public static long rangedSum(long n) {  
 return LongStream.*rangeClosed*(1, n)  
 .reduce(0L, Long::*sum*);  
}  
  
// 병렬 Parallel Ranged sum done in 3 msecs (드디어 for보다 더 빠름)  
public static long parallelRangedSum(long n) {  
 return LongStream.*rangeClosed*(1, n)  
 .parallel()  
 .reduce(0L, Long::*sum*);  
}

올바른 자료구조를 선택해야 병렬실행도 최적의 성능을 발휘할 수 있다는 사실을 확인할 수 있다.

하지만 병렬화는 완전 공짜는 아니다.

-각 서브스트림을 서로 다른 스레드의 리듀싱 연산으로 할당하고

-이들 결과를 하나의 값으로 합쳐야 한다

🡺멀티코어 간의 데이터 이동은 생각보다 비싸다.

🡺상황에 따라서 병렬화를 이용할 수 있거나 없을 수도 있다.

🡺스트림을 병렬화해서 코드 실행 속도를 빠르게 하고 싶으면 항상 병렬화를 올바르게 사용

**7.1.3 병렬 스트림의 올바른 사용법**

// 순차 Sequential SideEffect sum 8 msecs / Result: 50000005000000 (10번다 동일)  
public static long sideEffectSum(long n) {  
 Accumulator accumulator = new Accumulator();  
 LongStream.*rangeClosed*(1, n).forEach(accumulator::add);  
 return accumulator.total;  
}  
  
// 병렬 Parallel SideEffect sum 3 msecs / Result: 24895627783262 (10번다 다름)  
public static long sideEffectParallelSum(long n) {  
 Accumulator accumulator = new Accumulator();  
 LongStream.*rangeClosed*(1, n).parallel().forEach(accumulator::add);  
 return accumulator.total;  
}

class Accumulator {  
 public long total = 0;  
  
 public void add(long value) {  
 total += value;  
 }  
}

total += value를 실행하면 이런 문제가 생긴다.

total += value는 아토믹 연산이 아니다.

공유된 가변 상태를 피해야 한다.

**7.1.4 병렬 스트림 효과적으로 사용하기**

고정된 양을 기준으로 병렬 스트림을 사용하는 것은 적절하지 않음

🡺상황이 달라지면 (기기 등) 기준이 제 역할을 못함

-확신이 서지 않는다면 직접 측정하라. (적절한 베치마크로 직접 성능 측정)

-박싱을 주의 하라.

🡺자동 박싱과 언박싱은 성능을 크게 저하시키는 요소

🡺박싱을 피하도록 기본 특화 스트림(IntStream, LongStream, DoubleStream)을 제공

-순차 스트림보다 병렬 스트림에서 성능이 떨어지는 연산이 있다.

🡺요소의 순서에 의존하는(limit, findFirst)연산을 병렬 스트림에서 수행하려면 비싼 비용을 치러야 한다. (순서를 유지 해야 되기 때문)

🡺findAny는 요소의 순서와 상관없이 연산하므로 findFirst 보다 성능이 좋다.

🡺정렬된 스트림에 unordered를 호출하면 비정렬된 스트림을 얻을 수 있다.

-스트림에서 수행하는 전체 파이프라인 연산 비용을 고려하라.

🡺N(요소수)\*Q(비용) = N\*Q(처리비용). Q가 높아진다는 것은 병렬 스트림으로 개선할 수 있다는 가능성을 의미

-소량의 데이터에서는 병렬 스트림이 도움 되지 않는다.

🡺병렬화 과정에서 생기는 부가 비용을 상쇄할 수 있을 만큼 이득이 없다.

-스트림을 구성하는 자료구조가 적절한지 확인하라

🡺예를들어 ArrayList를 LinkedList 보다 효율적으로 분할할 수 있다.

🡺range 팩토리 메서드로 만든 기본형 스트림도 쉽게 분해할 수 있다.

🡺결국 IntStream, LongStream, DoubleStream (range를 쓰려면) 사용하는게 좋다.

-스트림의 특성과 파이프라인의 중간 연산이 스트림의 ㄹ 어떻게 바꾸는지에 따라 분해 과정의 성능이 달라질 수 있다.

🡺필터 연산이 있으면 스트림의 길이를 예측할 수 없어서 효과적으로 병렬처리 못함

-최종 연산의 병합 과정의 비용을 살펴 보라

🡺병합 과정의 비용이 비싸다면 병렬 스트림으로 얻은 성능의 이익이 합치는 과정에서 상쇄될 수 있다.

|  |  |
| --- | --- |
| 소스 | 분해성 |
| ArrayList | 훌륭함 |
| LinkedList | 나쁨 |
| IntStream.range | 훌륭함 |
| Stream.iterate | 나쁨 (무한스트림) |
| HashSet | 좋음 |
| TreeSet | 좋음 |

**7.2. 포크/조인 프레임워크**

포크/조인 프레임워크는 병렬화할 수 있는 작업을 재귀적으로 작은 작업으로 분활한 다음에 서브태스크 각각의 결과를 합쳐서 전체 결과를 만든다 (분할 / 정복)

7.2.1 RecursiveTask 활용

스레드 풀을 이용하려면 RecursiveTask<R>의 서브클래스를 만들어야 한다.

🡺R은 결과 형식 또는 결과가 없을 때는 RecursiveAction 형식

RecursiveTask를 정의하려면 추상 메서드 compute를 구현

protected abstract R compute();

compute구현은 다음과 같은 형식을 유지

if(태스크가 충분히 작거나 더 이상 분할할 수 없으면) {  
 순차적으로 태스크 계산  
}else {  
 태스크를 두 서브태스크로 분할  
 태스크가 다시 서브태스크로 분할되도록 이 메서드를 재귀적으로 호출함  
 모든 서브태스크의 연산이 완료될 때까지 기다림  
 각 서브태스크의 결과를 합침  
}

package chater7;  
  
*/\*\*  
 \* Created by simjunbo on 2018-02-26.  
 \*/*public class ForkJoinSumCalculator extends java.util.concurrent.RecursiveTask<Long> {  
  
 private final long[] numbers;  
 private final int start;  
 private final int end;  
 public static final long *THRESHOLD* = 10\_000;  
  
 public ForkJoinSumCalculator(long[] numbers) {  
 this(numbers, 0, numbers.length);  
 }  
  
 private ForkJoinSumCalculator(long[] numbers, int start, int end) {  
 this.numbers = numbers;  
 this.start = start;  
 this.end = end;  
 }  
  
 @Override  
 protected Long compute() {  
 int length = end - start;  
 if (length <= *THRESHOLD*) {  
 return computeSequentially();  
 }  
 ForkJoinSumCalculator leftTask =  
 new ForkJoinSumCalculator(numbers, start, start + length / 2);  
  
 leftTask.fork(); //비동기로  
  
 ForkJoinSumCalculator rightTask =  
 new ForkJoinSumCalculator(numbers, start + length / 2, end);  
  
 Long rightResult = rightTask.compute(); // 오른쪽 동기로  
 Long leftResult = leftTask.join(); // 왼쪽 취합  
  
 return leftResult + rightResult;  
 }  
  
 private long computeSequentially() {  
 long sum = 0;  
 for (int i = start; i < end; i++) {  
 sum += numbers[i];  
 }  
 return sum;  
 }  
}

다음 코드 처럼 ForkJoinSumCalculator의 생성자로 원하는 수의 배열로 넘겨줄 수 있다.

public static long forkJoinSum(long n) {  
 long[] numbers = LongStream.*rangeClosed*(1, n).toArray();  
 ForkJoinTask<Long> task = new ForkJoinSumCalculator(numbers);  
 return new ForkJoinPool().invoke(task);  
}

**7.2.2 포크/조인 프레임워크 제대로 사용 하는 방법**

-join 메서드를 태스크에 호출하면 태스크가 생산하는 결과가 준비될 때까지 호출자를 블록시킨다.

🡺따라서 두 서브태스크가 모두 시작된 다음에 join을 호출해야 한다. 그러면 다른 태스크가 끝나길 기다리는 일이 발생하며 원래 순차 보다 느리고 복잡해 진다.

-RecursiveTask 내에 ForkJoinPool의 invoke 메서드를 사용하지 말아야 한다.

대신 compute나 fork 메서드를 직접 호출할 수 있다. 순차 코드에서 병렬 계산을 시작할 때만

Invoke를 사용한다.

-서브태스크에 fork 메서드를 호출해서 ForkJoinPool의 일정을 조절할 수 있다.

🡺왼쪽, 오른쪽 모두 fork를 호출하는것보다 한쪽 작업에는 compute를 것이 효율적이다.

🡺그러면 두 태스크에서 한 태스크에는 같은 스레드를 재사용할 수 있으므로 불필요한 태스크를 할당하는 오버헤드를 피할 수 있다.

-포크/조인 프레임워크를 이용하여 병렬 계산은 디버깅하기 어렵다.

-병렬 스트림에서 살펴본 것처럼 멀티코어에 포크/조인 프레임워크를 사용하는 것이

순차처리보다 무조건 빠를거라는 생각은 버려야 한다.

**7.2.3 작업 훔치기**

실제로는 코어 개수와 관계없이 적절한 크기로 분할된 많은 태스크를 포킹하는 것이 바람직하다.

훨씬 복잡한 시나리오가 사용되는 현실에는 각각의 서브태스크의 작업 완료 시간이 달라 질 수 있다.

🡺포크/조인 프레임워크에서는 **작업 훔치기** 라는 기법으로 이 문제를 해결한다.



**P. 241 예제**

****

**둘다 fork를 사용한다면?**



**작업 훔치기(work stealing)**

****

**Spliterator**

List, Queue

|  |  |
| --- | --- |
| 종류 | Spliterator 특성 |
| ArrayList, Vector | Spliterator.ORDERED | Spliterator.SIZED | Spliterator.SUBSIZED |
| LinkedList | Spliterator.ORDERED | Spliterator.SIZED |
| PriorityQueue | Spliterator.SIZED | Spliterator.SUBSIZED | Spliterator.NONNULL |

Set, Map

|  |  |
| --- | --- |
| 종류 | Spliterator 특성 |
| HashSet, HashMap | Spliterator.SIZED | Spliterator.DISTINCT |
| SortedSet | Spliterator.DISTINCT | Spliterator.SORTED | Spliterator.ORDERED |
| TreeSet, TreeMap | Spliterator.DISTINCT | Spliterator.SORTED | Spliterator.ORDERED  | Spliterator.SIZED |
| LinkedHashSet,  LinkedHashMap | Spliterator.DISTINCT | Spliterator.ORDERED | Spliterator.SIZED |

**7.3 Spliterator**

Spliterator는 분할할 수 있는 반복자라는 의미이다.

Iterator(반복자) 랑 비슷하지만 Spliterator는 병렬작업에 특화되어 있다.

Spliterator가 어떻게 동작하는지 이해하면 병렬 스트림 동작과 관련한 통찰력을 얻을 수 있다.

public interface Spliterator<T> {  
 boolean tryAdvance(Consumer<? super T> action);  
 Spliterator<T> trySplit();  
 long estimateSize();  
 int characteristics();  
}

-tryAdvance 메서드는 Spliterator의 요소를 하나씩 순차적으로 수비하면서 탐색해야 할 요소가 남아 있으면 반환한다. (Iterator와 같다)

-trySplit 메서드는 Spliterator의 일부 요소(자신이 반환 요소)를 분할해서 두 번째 Spliterator를 생성하는 메서드다.

-estimateSize 메서드로 탐색해야 할 요소 수 정보를 제공할 수 있다.

-characteristics 메서드는 Spliterator 자체의 특성 집합을 포함하는 int를 반환한다.

**7.3.1 분할 과정**

final String SENTENCE =  
 " Nel mezzo del cammin di nostra vita " +  
 "mi ritrovai in una selva oscura" +  
 " ch la dritta via era smarrita ";  
System.*out*.println("Found " + countWordsIteratively(SENTENCE) + " words");

public static int countWordsIteratively(String s) {  
 int counter = 0;  
 boolean lastSpace = true;  
 for (char c : s.toCharArray()) {  
 if (Character.*isWhitespace*(c)) {  
 lastSpace = true;  
 } else {  
 if (lastSpace) {  
 counter++;  
 }  
 lastSpace = false;  
 }  
 }  
 return counter;  
}

🡺Found 19 words

**함수형 단어 개수 계산 메서드 재구현하기**

**7.4 요약**

-내부 반복을 이용하면 명시적으로 다른 스레드를 사용하지 않고도 스트림을 병렬로 처리 가능

-병렬처리가 항상 빠른건 아니다. 성능을 직접 측정해봐야 한다.

-데이터가 아주 많거나 각 요소를 처리하는데 오랜 시간이 걸릴 때 성능을 높일 수 있다.

-가능하면 기본형 특화 스트림을 사용하는 등 올바른 자료구조 선택하느냐에 따라 성능적으로

영향을 미칠 수 있다.

-포크/조인 프레임워크에서는 병렬화할 수 있는 태스크를 작은 태스크로 분할한 다음에 분할된

태스크를 각각의 스레드로 실행하며 서브태스크 각각의 결과를 합쳐서 최종 결과를 생성한다.

-Spliterator는 탐색하려는 데이터를 포함하는 스트림을 어떻게 병렬화할 것인지 정의한다.

보충

Java 병렬에서 하드웨어 적 사용 (CAS)

<http://mygumi.tistory.com/112>

**PART 3 효과적인 자바8프로그래밍**

**CHATER8. 리팩토링, 테스팅, 디버깅**

**8.1 가독성과 유연성을 개선하는 리팩토링**

람다 표현식은 익명 클래스보다 코드를 좀 더 간결하게 만든다.

동작 파라미터화의 형식(일급 시민)을 지원하므로 람다 표현식을 이용한 코드는 더 큰 유연성을 갖출 수 있다.

**8.1.1 코드 가독성 개선**

코드 가독성 : 어떤 코드를 다른 사람도 쉽게 이해할 수 있음

-코드의 장황함을 줄여서 쉽게 이해할 수 있는 코드를 구현할 수 있다.

-메서드 레퍼런스와 스트림 API를 이용해서 코드의 의도를 쉽게 표현할 수 있다.

**8.1.2 익명 클래스를 람다 표현식으로 리팩토링하기**

// 익명 클래스 이용  
Runnable r1 = new Runnable() {  
 public void run() {  
 System.*out*.println("Hello");  
 }  
};  
  
// 람다 이용  
Runnable r2 = () -> System.*out*.println("Hello");

모든 익명 클래스를 람다 표현식으로 변환할 수 있는 것으 아니다.

1) 익명 클래스에서 사용한 this와 super는 람다 표현식에서 다른 의미를 갖는다.

익명 클래스 this는 익명 클래스 자신, 람다에서 this는 람다를 감싸는 클래스

2) 익명 클래스는 감싸고 있는 클래스의 변수를 가릴 수도 있다(섀도우 변수)

3) 익명 클래스를 람다 표현식으로 바꾸면 컨텍스트 오버로딩에 따른 모호함이 초래될 수 있다.

익명 클래스는 인스턴스화할 때 명시적으로 형식이 정해지는 반면 람다의 형식은 컨텍스트에 따라 달라지기 떄문이다.

**8.1.3 람다 표현식을 메서드 레퍼런스로 리팩토링하기**

람다 표현식 대신 메서드 레퍼런스를 이용하면 가독성을 높일 수 있다.

Map<CaloricLevel, List<Dish>> dishesByCaloricLevel =  
 menu.stream()  
 .collect(  
 Collectors.*groupingBy*(dish -> {  
 if (dish.getCalories() <= 400)  
 return CaloricLevel.*DIET*;  
 else if (dish.getCalories() <= 700)  
 return CaloricLevel.*NORMAL*;  
 else  
 return CaloricLevel.*FAT*;  
 })  
 );  
  
// 메서드 레퍼런스 방식으로 변경  
Map<CaloricLevel, List<Dish>> dishesByCaloricLevel2 =  
 menu.stream().collect(Collectors.*groupingBy*(Dish::getCaloricLevel));

람다 표현식과 저수준 리듀싱 연산을 조합하는 것보다 Collectors API를 사용하면 코드의 의도가 더 명확해진다.

// 저수준 리듀싱을 Collectors API로 바꾸기  
int totalCalories = menu.stream().map(Dish::getCalories)  
 .reduce(0, (c1, c2) -> c1 + c2);  
  
int totalCalories2 = menu.stream()  
 .collect(Collectors.*summingInt*(Dish::getCalories));

**8.1.4 명령형 데이터 처리를 스트림으로 리팩토링하기**

외부 반복자를 이용한 기존의 모든 컬렉션 처리 코드를 스트림 API로 바꿔야 한다.

🡺스트림 API는 데이터 처리 파이프라인의 의도를 더 명확하게 보여준다.

🡺스트림은 쇼트서킷과 게으름이라는 강력한 최적화뿐 아니라 멀티코어 아키텍처를 활용할 수 있게 해준다.

// 명령형 데이터 처리를 스트림으로 리팩토링 하기  
List<String> dishNames = new ArrayList<>();  
for (Dish dish : menu) {  
 if (dish.getCalories() > 300) {  
 dishNames.add(dish.getName());  
 }  
}  
  
// 스트림 API로 변환  
menu.parallelStream()  
 .filter(d -> d.getCalories() > 300)  
 .map(Dish::getName)  
 .collect(Collectors.*toList*());

**8.1.5 코드 유연성 개선**

람다 표현식을 이용하면 동작 파라미터화를 쉽게 구현할 수 있다.

**함수형 인터페이스 적용**

먼저 람다 표현식을 이용하려면 함수형 인터페이스가 필요하다.

**조건부 연기 실행**

if(logger.isLoggable(Log.FINER)) {  
 logger.finer("Problem: " + generateDiagnostic());  
}  
  
==>  
  
logger.log(Level.FINER, "Problem: " + generateDiagnostic());

-코드 가독성이 좋아질 뿐아니라 캡슐화도 강화된다.

**실행 어라운드**

매번 같은 준비, 종료 과정을 반복적으로 수행 하는 코드가 있다면 이를 람다로 변환할 수 있다.

준비, 종료 과정을 처리하는 로직을 재사용함으로써 코드 중복을 줄일 수 있다.

// 실행 어라운드  
String oneLine = *processFile*((BufferedReader b) -> b.readLine()); // 람다 전달  
String towLines = *processFile*((BufferedReader b) -> b.readLine() + b.readLine()); // 다른 람다 전달

public static String processFile(BufferedReaderProcessor p) throws IOException {  
 try (BufferedReader br = new BufferedReader(new FileReader("java8incation/chap8/data.txt"))) {  
 return p.process(br);  
 }  
}

**8.2 람다로 객체지향 디자인 패턴 리팩토링하기**

다양한 패턴을 유형별로 정리한 것이 디자인 패턴이다.

**8.2.1 전략**

전략 패턴은 한 유형의 알고리즘을 보유한 상태에서 런타임에 적절한 알고리즘을 선택하는 기법이다.

세 부분으로 구성된다

-알고리즘을 나타내는 인터페이스(Strategy 인터페이스)

-다양한 알고리즘을 나타내는 한 개 이상의 인터페이스 구현

(ConcreteStrategyA, ConcreateStrategyB)

-전략 객체를 사용하는 한 개 이상의 클라이언트

// Client

public class StrategyPattern {  
 private final ValidationStrategy strategy;  
  
 public StrategyPattern(ValidationStrategy v) {  
 this.strategy = v;  
 }  
  
 public boolean validate(String s) {  
 return strategy.execute(s);  
 }  
  
 public static void main(String[] args) {  
 StrategyPattern numericStrategyPattern = new StrategyPattern(new IsNumeric());  
 boolean b1 = numericStrategyPattern.validate("aaaa");  
  
 StrategyPattern lowerCaseStrategyPattern = new StrategyPattern(new IsAllLowerCase());  
 boolean b2 = lowerCaseStrategyPattern.validate("bbbb");  
  
 // 람다로 변환  
 StrategyPattern numericStrategyPattern2 = new StrategyPattern((String s) -> s.matches("a-z+"));  
 boolean b3 = numericStrategyPattern2.validate("aaaa");  
  
 StrategyPattern lowerCaseStrategyPattern2 = new StrategyPattern((String s) -> s.matches("\\d+"));  
 boolean b4 = lowerCaseStrategyPattern2.validate("bbbb");  
 }  
}

// Strategy  
class IsAllLowerCase implements ValidationStrategy {  
  
 @Override  
 public boolean execute(String s) {  
 return s.matches("[a-z]+");  
 }  
}

// ConcreteStrategy  
class IsNumeric implements ValidationStrategy {  
 @Override  
 public boolean execute(String s) {  
 return s.matches("\\d+");  
 }  
}  
  
interface ValidationStrategy {  
 boolean execute(String s);  
}

**람다 표현식 사용**

람다 표현식을 이용하면 전략 디자인 패턴에서 발생하는 자잘한 코드를 제거할 수 있다.

람다 표현식은 코드 조각을 캡슐화한다. 즉, 람다 표현식으로 전략 디자인 패턴을 대신할 수 있다.

**8.2.2 템플릿 메서드**

템플릿 메서드는 ‘이 알고리즘을 사용하고 싶은데 그대로는 안 되고 조금 고쳐야 하는’ 상황에 적합하다.

/\*  
템플릿 메서드패턴  
 \*/  
public class TemplateMethodPattern {  
 public static void main(String[] args) {  
 Database.*addCustomerInfo*(1337, new Customer(1337, "심준보"));  
 new TemplateMethodPattern().processCustomer(1337, (Customer c) -> System.*out*.println("Hello " + c.getName()));  
 }  
  
 public void processCustomer(int id, Consumer<Customer> makeCustomerHappy) {  
 Customer c = Database.*getCustomerWithId*(id);  
 makeCustomerHappy.accept(c);  
 }  
}  
  
// 템플릿 일반  
abstract class OnlineBanking {  
 public void processCustomer(int id) {  
 Customer c = Database.*getCustomerWithId*(id);  
 makeCustomerHappy(c);  
 }  
  
 abstract void makeCustomerHappy(Customer c);  
}  
  
class Database {  
 public static HashMap<Integer, Customer> *map* = new HashMap<Integer, Customer>();  
  
 public static void addCustomerInfo(int id, Customer customer) {  
 if (!*map*.containsKey(id)) {  
 *map*.put(id, customer);  
 }  
 }  
  
 public static Customer getCustomerWithId(int id) {  
 if (*map*.containsKey(id)) {  
 return *map*.get(id);  
 } else {  
 return null;  
 }  
 }  
}  
  
class Customer {  
 private int id;  
 private String name;  
  
 Customer(int id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public String getName() {  
 return name;  
 }  
}

**람다 표현식 사용**

람다 표현식을 이용하면 템플릿 메서드 디자인 패턴에서 발생하는 자잘한 코드도 제거할 수 있다.

**8.2.3 옵저버**

어떤 이벤트가 발생했을 때 한 객체(subject)가 다른 객체 리스트(observer)에 자동으로 알림을 보내야 하는 상황에 옵저버 디자인 패턴을 사용한다.

public class ObserverPattern implements Subject {  
 // composition  
 private final List<Observer> observers = new ArrayList<>();  
  
 @Override  
 public void registerObserver(Observer o) {  
 this.observers.add(o);  
 }  
  
 @Override  
 public void notifiyObservers(String tweet) {  
 observers.forEach(o -> o.notifiy(tweet));  
 }  
  
 public static void main(String[] args) {  
 ObserverPattern f = new ObserverPattern();  
 f.registerObserver(new NYTimes());  
 f.registerObserver(new Guarding());  
 f.registerObserver(new LeMOnde());  
 f.notifiyObservers("The queen said her favourite book is Java 9 in Action!");  
  
 // 람다화  
 f.registerObserver((String tweet) -> {  
 if (tweet != null && tweet.contains("money")) {  
 System.*out*.println("Breaking news in NY! " + tweet);  
 }  
 });  
  
 f.registerObserver((String tweet) -> {  
 if (tweet != null && tweet.contains("queen")) {  
 System.*out*.println("Yet another news in London... " + tweet);  
 }  
 });  
 }  
}

// ConcreateObserver  
class NYTimes implements Observer {  
 @Override  
 public void notifiy(String tweet) {  
 if (tweet != null && tweet.contains("money")) {  
 System.*out*.println("Breaking news in NY! " + tweet);  
 }  
 }  
}  
// ConcreateObserver  
class Guarding implements Observer {  
 @Override  
 public void notifiy(String tweet) {  
 if (tweet != null && tweet.contains("queen")) {  
 System.*out*.println("Yet another news in London... " + tweet);  
 }  
 }  
}  
// ConcreateObserver  
class LeMOnde implements Observer {  
 @Override  
 public void notifiy(String tweet) {  
 if (tweet != null && tweet.contains("wine")) {  
 System.*out*.println(("Today cheese, wine and news! " + tweet));  
 }  
 }  
}  
  
// Observer  
interface Observer {  
 void notifiy(String tweet);  
}  
  
// Subject  
interface Subject {  
 void registerObserver(Observer o);  
  
 void notifiyObservers(String tweet);  
}

**람다 표현식 사용하기**

그렇다면 항ㅇ상 람다 표현식을 사용해야 할까? 물론 아니다.

옵저버가 상태를 가지며, 여러 메서드를 정의하는 등 복잡하다면 람다 표현식보다 기존의 클래스 구현 방식을 고수하는 것이 바람직할 수도 있다.

**8.2.4 의무 체인**

한 객체가 어떤 작업을 처리한 다음에 다른 객체로 결과를 전달하고, 다른 객체도 해야 할 작업을 처리한 다음에 또 다른 객체로 전달하는 식이다.

일반적으로 다음으로 처리할 객체 정보를 유지하는 필드를 포함하는 작업 처리 추상 클래스로 의무 체인 패턴을 구성한다.

// Client

public class ChainOfResponsibility {  
 public static void main(String[] args) {  
 ProcessingObject<String> p1 = new HeaderTextProcessing();  
 ProcessingObject<String> p2 = new SpellCheckerProcessing();  
  
 p1.setSuccessor(p2);  
  
 String result = p1.handle("Aren't labdas really sexy?!!");  
 System.*out*.println(result);  
  
 // 람다  
 UnaryOperator<String> headerProcessing =  
 (String text) -> "From Raoul, Mario And Alan:" + text;  
  
 UnaryOperator<String> spellCheckerProcessing =  
 (String text) -> text.replaceAll("labda", "lambda");  
  
 Function<String, String> pipeline =  
 headerProcessing.andThen(spellCheckerProcessing);  
  
 String result2 = pipeline.apply(" Aren't labdas really sexy?!!");  
 System.*out*.println(result2);  
 }  
}  
  
// ProcessingObject  
abstract class ProcessingObject<T> {  
 protected ProcessingObject<T> successor;  
  
 public void setSuccessor(ProcessingObject<T> successor) {  
 this.successor = successor;  
 }  
  
 public T handle(T input) {  
 T r = handleWork(input);  
 if (successor != null) {  
 return successor.handle(r);  
 }  
 return r;  
 }  
 abstract protected T handleWork(T input); // 템플릿 메서드  
}  
  
// ConcreateProcessingObject  
class HeaderTextProcessing extends ProcessingObject<String> {  
 @Override  
 protected String handleWork(String text) {  
 return "From Raoul, Mario and Alan: " + text;  
 }  
}  
// ConcreateProcessingObject  
class SpellCheckerProcessing extends ProcessingObject<String> {  
 @Override  
 protected String handleWork(String text) {  
 return text.replaceAll("labda", "lambda");  
 }  
}

**람다 표현식 사용**

이 패턴은 함수 체인과 비슷하다. UnarayOperator형식의 인스턴스로 표현할 수 있다.

andThen 메서드로 이들 함수를 조합해서 체인을 만들 수 있다.

**8.2.5 팩토리**

인스턴스화 로직을 클라이언트에 노출하지 않고 객체를 만들 때 팩토리 디자인 패턴을 사용한다.

public class FactoryPattern {  
 final static Map<String, Supplier<Product>> *map* = new HashMap<>();  
 Map<String, TriFunction<Integer, Integer, String, Product>> map2 = new HashMap<>();  
  
 static {  
 *map*.put("loan", Loan::new);  
 *map*.put("loan", Stock::new);  
 *map*.put("loan", Bond::new);  
 }  
  
 public static Product createProduct(String name) {  
 Supplier<Product> p = *map*.get(*map*);  
 if (p != null)  
 return p.get();  
 throw new IllegalArgumentException("No such product " + name);  
 }  
  
 public static void main(String[] args) {  
 Product p = ProductFactory.*createProduct*("loan");  
 p.print();  
  
 // 람다  
 Supplier<Product> loanSupplier = Loan::new;  
 Loan loan = (Loan) loanSupplier.get();  
 }  
}  
  
class ProductFactory {  
 public static Product createProduct(String name) {  
 switch (name) {  
 case "loan":  
 return new Loan();  
 case "stock":  
 return new Stock();  
 case "bond":  
 return new Bond();  
 default:  
 throw new RuntimeException("No such product " + name);  
 }  
 }  
}  
  
abstract class Product {  
 abstract void print();  
}  
  
class Loan extends Product {  
 @Override  
 void print() {  
 System.*out*.println("Loan");  
 }  
}  
  
class Stock extends Product {  
 @Override  
 void print() {  
 System.*out*.println("Stock");  
 }  
}  
  
class Bond extends Product {  
 @Override  
 void print() {  
 System.*out*.println("Bond");  
 }  
}

**8.3 람다 테스팅**

개발자의 최종 업무 목표는 제대로 작동하는 코드를 구현하느 ㄴ것이지 깔끔한 코드를 구현하는 것이 아니다.

class Point {  
 private final int x;  
 private final int y;  
  
 public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
  
 public int getX() {  
 return x;  
 }  
  
 public int getY() {  
 return y;  
 }  
}

일반적으로 좋은 소프트웨어 공학자라면 프로그램이 의도대로 동작하는지 호가인할 수 있는 단위 테스팅을 진행한다.

@Test  
public void testMoveRightBy() throws Exception {  
 Point p1 = new Point(5, 5);  
 Point p2 = p1.moveRightBy(10);  
  
 *assertEquals*(15, p2.getX());  
 *assertEquals*(5, p2.getY());  
}

**8.3.1 보이는 람다 표현식의 동작 테스팅**

람다는 익명(결국 익명 함수) 이므로 텥스트 코드 이름을 호출할 수 없다. 따라서 필요하면 람다를 필드에 저장해서 재사용할 수 있으며 람다의 로직을 테스트할 수 있다.

@Test  
public void testComparingTwoPoints() throws Exception {  
 Point p1 = new Point(10, 15);  
 Point p2 = new Point(10, 20);  
 int result = Point.*compareByXAndThenY*.compare(p1, p2);  
 *assertEquals*(-1, result);  
}

public final static Comparator<Point> *compareByXAndThenY* =  
 Comparator.*comparing*(Point::getX).thenComparing(Point::getY);

람다 표현식은 함수형 인터페이스의 인스턴스를 생성한다는 사실을 기억하자. 따라서 생성된 인스턴스의 동작으로 람다 표현식을 테스트할 수 있다. (대상형식)

**8.3.2 람다를 사용하는 메서드의 동작에 집중하라**

람다의 목표는 정해진 동작을 다른 메서드에서 사용할 수 있도록 하나의 조각으로 캡슐화하는 것이다. 그러려면 세부 구현을 포함하는 람다 표현식을 공개하지 말아야 한다.

@Test  
public void testMoveAllPointsRightBy() throws Exception {  
 List<Point> points = Arrays.*asList*(new Point(5, 5), new Point(10, 5));  
 List<Point> expectedPoints = Arrays.*asList*(new Point(15, 5), new Point(20, 5));  
 List<Point> newPoints = Point.*moveAllPointsRightBy*(points, 10);  
 *assertEquals*(expectedPoints, newPoints);  
}

public static List<Point> moveAllPointsRightBy(List<Point> points, int x) {  
 return points.stream()  
 .map(p -> new Point(p.getX() + x, p.getY()))  
 .collect(Collectors.*toList*());  
}

**8.3.4 고차원 함수 테스팅**

함수를 인수로 받거나 다른 함수를 반환하는 메서드(이를 고차원 함수라고 한다)는 좀 더 사용하기 어렵다.메서드가 람다를 인수로 받는다면 다른 람다로 메서드의 동작을 테스트할 수 있다.

@Test  
public void testFilter() throws Exception {  
 List<Integer> numbers = Arrays.*asList*(1, 2, 3, 4);  
 List<Integer> even = *filter*(numbers, i -> i % 2 == 0); // 2, 4  
 List<Integer> smallerThanThree = *filter*(numbers, i -> i < 3); // 1, 2  
 *assertEquals*(Arrays.*asList*(2, 4), even);  
 *assertEquals*(Arrays.*asList*(1, 2), smallerThanThree);  
}  
  
// 리스트 형식의 추상화  
public static <T> List<T> filter(List<T> list, Predicate<T> p) {  
 List<T> result = new ArrayList<>();  
 for (T e : list) {  
 if (p.test(e)) {  
 result.add(e);  
 }  
 }  
 return result;  
}

**8.4 디버깅**

-스택 트레이스

-로깅 (peek)

**8.4.1 스택 트레이스 확인**

람다 표현식은 이름이 없기 때문에 조금 복잡한 스택 트레이스가 생성된다.

**8.4.2 정보로깅**

public class Peek {  
 public static void main(String[] args) {  
 List<Integer> numbers = Arrays.*asList*(2, 3, 4, 5);  
 numbers.stream()  
 .map(x -> x + 17)  
 .filter(x -> x % 2 == 0)  
 .limit(3)  
 .forEach(System.*out*::println);  
  
 List<Integer> result = numbers.stream()  
 .peek(x -> System.*out*.println("from stream: " + x))  
 .map(x -> x + 17)  
 .peek(x -> System.*out*.println("after map: " + x))  
 .filter(x -> x % 2 == 0)  
 .peek(x -> System.*out*.println("after filter: " + x))  
 .limit(3)  
 .peek(x -> System.*out*.println("after limit: " + x))  
 .collect(Collectors.*toList*());  
 }  
}

peek은 스트림의 각 요소를 소비한 것처럼 동작을 실행한다.

Peek은 자신이 확인한 요소를 파이프라인의 다음 연산으로 그대로 전달한다.

**CHATER. 9 디폴트 메서드**

자바8에서는 기본 구현을 포함하는 인터페이스를 정의하는 두가지 방법을 제공

1)인터페이스 내부에 정적 메서드 사용

2)디폴트 메서드라는 기능을 사용

public interface A {  
 default void hello() {  
 System.*out*.println("Hello from A");  
 }  
}

**정적 메서드와 인터페이스**

자바8에서는 인터페이스에 직접 정적 메서드를 선언할 수 있으므로 유틸리티 클래스를 없애고 직접 인터페이스 내부에 정적 메서드를 구현할 수 있다.

호환성 문제로 유틸클래스는 남아 있음

**9.1 변화하는 API**

바이너리 호환성, 소스 호환성, 동작 호환성

-바이너리 호환성 : 뭔가를 바꾼 이후에도 에러 없이 기존 바이너리가 실행될 수 있는 상황

-소스 호환성 : 코드를 고쳐도 기존 프로그램을 성공적으로 재컴파일할 수 있음을 의미한다.

-동작 호환성 : 코드를 바꾼 다음에도 값은 입력값이 주어지면 프로그램이 같은 동작을 실행한다는 의미다.

**9.2 디폴트 메서드란 무엇인가?**

자바8에서 호환성을 유지하면서 API를 바꿀 수 있도록 새로운 기능인 디폴트 메서드를 제공한다.

인터페이스에 디폴트 메서드를 추가하면 소스 호환성이 유지된다.

**추상 클래스와 자바8의 인터페이스**

다른점

1)클래스는 하나의 추상 클래스만 상속받을 수 있지만, 인터페이스는 여러 개 구현할 수 있다.

2)추상 클래스는 인스턴스 변수로 공통 상태를 가질 수 있다. 하지만 인터페이스는 인스턴스 변수를 가질 수 없다. (new로 인스턴스 생성이 안되니까…)

**9.3 디폴트 메서드 활용 패턴**

**9.3.1 선택형 메서드**

인터페이스를 구현하는 클래스에서 메서드의 내용이 비어있는 상황을 본 적이 있을 것이다.

ex)기본 구현이 제공되므로 Iterator 인터페이스를 구현하는 클래스는 빈 remove 메서드를 구현할 필요가 없어졌고, 불필요한 코드를 줄일 수 있다.

default void remove() {  
 throw new UnsupportedOperationException();  
}

**9.3.2 동작 다중 상속**

클래스는 한 개의 다른 클래스만 상속할 수 있지만, 인터페이스는 여러 개 구현할 수 있다.

**옳지 못한 상속**

상속으로 코드 재사용 문제를 모두 해결할 수 있는 것은 아니다. 예를 들어 한 개의 메서드를 재사용하려고 100개의 메서드와 필드가 정의되어 있는 클래스를 상속받는 것은 좋은 생각이 아니다.

🡺델리게이션 이용

**9.4 해석 규칙**

**9.4.1 알아야할 세 가지 해결 규칙**

1) 클래스가 항상 이긴다. 클래스나 슈퍼클래스에서 정의한 메서드가 디폴트 메서드보다 우선권을 가진다.

2) 1번 규칙 이외의 상황에서는 서브인터페이스가 이긴다. 상속관계를 갖는 인터페이스에서 같은 시그너처를 갖는 메서드를 정의할 때는 서브인터페이스가 이긴다.

3) 여전히 디폴트 메서드의 우선순위가 결정되지 않았다면 여러 인터페이스를 상속받는 클래스가 명시적으로 디폴트 메서드를 오버라이드 하고 호출해야 한다.

**CHATER 10. Null 대신 Optional**

**10.1 값이 없는 상황을 어떻게 처리할까?**

class Person {  
 private Car car;  
  
 public Car getCar() {  
 return car;  
 }  
}  
  
class Car {  
 private Insurance insurance;  
  
 public Insurance getInsurance() {  
 return insurance;  
 }  
}  
  
class Insurance {  
 private String name;  
  
 public String getName() {  
 return name;  
 }  
}

자동차와 자동차 보험을 갖고 있는 사람 객체를 중첩 구조로 구현했다고 하자.

// 문제점1) 체크 안함.  
public static String getCarInsuranceName(Person person) {  
 return person.getCar().getInsurance().getName();  
}

getCar, getInsurance, getName 값이 없으면 NullPointerException이 발생할 수 있다.

**10.1.1 보수적인 자세로 NullPointerException 줄이기**

대부분의 프로그래머는 필요한 곳에 다양한 null 확인 코드를 추가해서 null 예외 문제를 해결하려 할 것이다.

// 문제점2) 깊은 의심  
// if 블록을 추가하면 코드 들여쓰기 수준이 증가한다.  
// 코드의 구조가 엉망이 되고 가독성도 떨어진다.  
public static String getCarInsuranceName2(Person person) {  
 if (person != null) {  
 Car car = person.getCar();  
 if (car != null) {  
 Insurance insurance = car.getInsurance();  
 if (insurance != null) {  
 return insurance.getName();  
 }  
 }  
 }  
 return "Unknown";  
}

변수를 접근할 때마다 중첩된 if가 추가되면서 코드 들여쓰기 수준이 증가한다. 이와 같은 반복 패턴 코드를 ‘**깊은 의심’** 이라고 부른다.

// 문제점3) 너무 많은 출구  
// 출구 때문에 유지보수가 어려워진다.  
// 문자열을 반복하면서 오타 등의 실수가 생길 수 있다.  
public static String getCarInsuranceName3(Person person) {  
 if (person == null) {  
 return "Unknown";  
 }  
  
 Car car = person.getCar();  
 if (car == null) {  
 return "Unknown";  
 }  
  
 Insurance insurance = car.getInsurance();  
 if (insurance == null) {  
 return "Unknown";  
 }  
  
 return insurance.getName();  
}

중첩 if 블록을 없앴다. 즉 null 변수가 있으면 즉시 ‘Unknown’을 반환한다.

하지만 이 예제도 그렇게 좋은 코드는 아니다. 메서드에 네 개의 출구가 생겼기 때문이다.

같은 문자열을 반복하면서 오타 등의 실수가 생길 수 있다.

**10.1.2 null 때문에 발생하는 문제**

자바에서 null 레퍼런스를 사용하면서 발생할 수 있는 이론적, 실용적 문제를 확인하자.

**에러의 근원이다**

NullPointerException은 자바에서 가장 흔희 발생하는 에러다.

**코드를 어지럽힌다.**

때로는 중첩된 null 확인 코드를 추가해야 하므로 null 때문에 코드 가독성이 떨어진다.

**아무 의미가 없다.**

null은 아무 의미도 표현하지 않는다. 특히 정적 형식 언어에서 값이 없음을 표현하는 방법으로는 적절하지 않다.

**자바 철학에 위배된다.**

자바는 개발자로부터 모든 포인터를 숨겼다. 하지만 예외가 있는데 그것은 바로 null 포인터다.

**형식 시스템에 구멍을 만든다.**

Null은 무형식이며 정보를 포함하고 있지 않으므로 모든 레퍼런스 형식에 null을 할당할 수 있다. 이런 식으로 null이 할당되기 시작하면서 시스템의 다른 부분으로 null이 퍼졌을 때 애초에 null이 어떤 의미로 사용되었는지 알 수 없다.

**10.1.3 다른 언어는 null 대신 무얼 사용하나?**

그루비 : 안전 내비게이션 연산자를 도입해서 null 문제를 해결

Def carInsuranceName = person?.car?.insurance?.name

하스켈, 스칼라 : 다른 관점에서 null 문제를 접근한다.

하스켈은 선택형값(optional value)을 저장할 수 있는 Maybe라는 형식을 제공한다.

Maybe는 주어진 형식의 값을 갖거나 아니면 아무 값도 갖지 않을 수 있다.

스칼라도 T 형식의 값을 갖거나 아무 값도 갖지 않을 수 있는 Option[T]라는 구조를 제공한다.

**10.2 Optional 클래스 소개**

자바8은 하스켈과 스칼라의 영향을 받아서 java.util.Optional<T>라는 새로운 클래스를 제공한다.

값이 있으면 Optional 클래스는 값을 감싼다.

값이 없으면 Optional.empty 메서드로 Optional을 반환한다.

Null 레퍼런스와 Optional.empty()는 서로 무엇이 다른지 궁금할 것이다.

Null을 참조하려 하면 NullPointerException이 발생하지만 Optional.empty()는 Optional 객체이므로 이를 다양한 방식으로 활용할 수 있다.

class Person {  
 private Optional<Car> car;  
  
 public Optional<Car> getCar() {  
 return car;  
 }  
}  
  
class Car {  
 private Optional<Insurance> insurance;  
  
 public Optional<Insurance> getInsurance() {  
 return insurance;  
 }  
}  
  
class Insurance {  
 private String name;  
  
 public String getName() {  
 return name;  
 }  
}

Optional 클래스를 사용하면서 모델의 의미가 더 명확해졌음을 확인할 수 있다.

사람은 Optional<Car>를 참조하며 자동차는 Optional<Insurance>를 참조하는데, 이는 사람이 자동차를 소유했을 수도 아닐 수도 있으며, 자동차 보험에 가입되어 있을 수도 아닐 수도 있다.

또한 보험회사 이름은 Optional<String>이 아니라String으로 선언되어 있는데, 이는 보험회사는 반드시 이름을 가져야 함을 보여준다.

Null 레퍼런스를 Optional로 대치하는 것은 바람직하지 않다. Optional의 역할은 더 이해하기 쉬운 API를 설계하도록 돕는 것이다.

**10.3 Optional 적용 패턴**

**10.3.1 Optional 객체 만들기**

**빈 Optional**

// 빈 Optional  
Optional<Car> optCar = Optional.*empty*();

**null이 아닌 값으로 Optional 만들기**

// null이 아닌 값으로 Optional  
Optional<Car> optCar = Optional.*of*(car);

**null값으로 Optional 만들기**

// null값으로 Optional 만들기  
Optional<Car> optCar = Optional.ofNullable(car);

**10.3.2 맵으로 Optional 값을 추출하고 변환하기**

Optional<Insurance> optInsurance = Optional.*ofNullable*(insurance);  
Optional<String> name = optInsurance.map(Insurance::getName);

여기서 Optional 객체를 최대 요소의 개수가 한 개 이하인 데이터 컬렉션으로 생갈할 수 있다.

Optional이 값을 포함하면 map의 인수로 제공된 함수가 값을 바꾼다.

**10.3.3 flatMap으로 Optional 객체 연결**

Optional<Person> optPerson = Optional.*of*(person);  
Optional<String> name = optPerson.map(Person::getCar) // 컴파일 오류  
 .map(Car::getInsurance)  
 .map(Insurance::getName);

안타깝게도 위 코드는 컴파일되지 않는다. 왜 그럴까? 변수 optPeople의 형식은 Optional<People> 이므로 map 메서드를 호출할 수 있다.

하지만 getCar는 Optional<Car> 형식의 객체를 반환한다.

즉 map 연산의 결과는 Optional<Optional<Car>> 형식의 객체다.

🡺flatMap으로 해결할 수 있다.

flatMap을 사용하면 이차원 Optional이 하나의 삼각형을 포함하는 하나의 Optional로 바뀐다.

**Optional로 자동차의 보험회사 이름 찾기**

public void getCarInsuranceName(Optional<Person> person) {  
 return person.flatMap(Person::getCar)  
 .flatMap(Car::getInsurance)  
 .map(Insurance::getName)  
 .orElse("Unknown");  
}

Optional을 이용해서 값이 없는 상황을 처리하는 것이 어떤 장점을 제공하는지 확인할 수 있다.

즉, null을 확인하느라 조건 분기문을 추가해서 코드를 복잡하게 만들지 않으면서도 쉽게 이해할 수 있는 코드를 완성했다.

**Optional을 이용한 Person/Car/Insurance 참조 체인**

Optional<Person>으로 시작해서 Person의 Car, Car의 Insurance, Insurance의 이름 문자열을 참조(map, flatMap을 이용) 하는 방법을 살펴봤다.

평준화 과정이란 이론적으로 두 Optional을 합치는 기능을 수행하면서 둘 중 하나라도 null이면 빈 Optional을 생성하는 연산이다.

호출 체인 중 어떤 메서드가 빈 Optional을 반환한다면 전체 결과로 빈 Optional을 반환하고 아니면 관련 보험회사의 이름을 포함하는 Optional을 반환한다.

**도메인 모델에 Optional을 사용했을 때 데이터를 직렬화할 수 없는 이유**

자바 언어 아키텍트인 브라이언 고츠는 Optional의 용도가 선택형 반환값을 지원하는 것이라고 명확하게 못박았다.

Optional 클래스는 필드 형식으로 사용할 것을 가정하지 않았으므로 Serializable 인터페이스를 구현하지 않는다.

🡺직렬화 모델을 사용하는 도구나 프레임워크에서 문제가 생길 수 있다.

**10.3.4 디폴트 액션과 Optional 언랩**

Optional이 비어있을 때 디폴트값을 제공할 수 있는 orElse 메서드로 값을 읽자.

-get()은 값을 읽는 가장 간단한 메서드면서 동시에 가장 안전하지 않은 메서드다.

메서드 get은 래핑된 값이 있으면 해당 값을 반환하고 값이 없으면 NoSuchElementException을 발생시킨다.

-orElse(T other)를 사용하면 Optional이 값을 포함하지 않을 때 디폴트값을 제공할 수 있다.

-orElseGet(Sipplier<? Extrends T> other)는 orElse 메서드에 대응하는 게으른 버전의 메서드이다.

Optional에 값이 없을 때만 Supplier가 실행되기 때문이다.

-orElseThrow(Supplier<? Extends X> exceptionSupplier)는 Optional이 비어있을 때 예외를 발생시킨다는 점에서 get 메서드와 비슷하다. 하지만 예외의 종류를 선택할 수 있다.

-ifPresent(Consumer<? Super T> consumer) 를 이용하면 값이 존재할 때 인수로 넘겨준 동작을 실행할 수 있다.

**10.3.5 두 Optional 합치기**

**10.3.6 필터로 특정값 거르기**

Insurance insurance = ...;  
if(insurance != null && "CambridgeInsurance".equals(insurance.getName())) {  
 System.*out*.println("ok");  
}

🡺Optional 객체에 filter 메서드를 이용해서 다음과 같이 코드를 재구현할 수 있다.

Optional<Insurance> optInsurance = ...;  
optInsurance.filter(insurance -> "CambridgeInsurance".equals(insurance.getName()))  
 .ifPresent(x -> System.*out*.println("ok"));

Optional 객체가 값을 가지며 프레디케이트와 일치하며 filter 메서드는 그 값을 반환하고 그렇지 않으면 빈 Optional 객체를 반환한다.

**10.4.1 잠재적으로 null이 될 수 있는 대상을Optional로 감싸기**

Object value = map.get("key");

🡺

Optional<Object> value = Optional.*ofNullable*(map.get("key"));

**10.4.2 예외와 Optional**

public static Optional<Integer> stringToInt(String s) {  
 try {  
 return Optional.*of*(Integer.*parseInt*(s));  
 } catch (NumberFormatException e) {  
 return Optional.*empty*();  
 }  
}

**기본형 Optional과 이를 사용하지 말아야 하는 이유**

스트림처럼 Optional도 기본형으로 특화된 OptionalInt, OptionalLong, OptionalDouble등의 클래스를 제공한다.

🡺하지만 Optional의 최대 요소 수는 한 개이므로 Optional에서는 기본형 특화 클래스로 성능을 개선할 수 없다.

🡺기본형 Optional은 map, flatMap, filter 등을 지원하지 않는다.

**CHATER 11. CompletableFuture : 조합할 수 있는 비동기 프로그래밍**

최근에 소프트웨어 구현 방법에 큰 변화를 불러온 두 가지 추세가 있다.

1. 하나는 애플리케이션을 실행하는 하드웨어와 관련된 변화
2. 하나는 애플리케이션끼리 어떻게 상호작용하는가와 관련된 변화다.

외부 네트워크 서비스의 반응속도가 느릴 수도 있으므로 빈 화면이나 검은 화면을 보여주는 것보다는 부분결과라도 먼저 보여주는 것이 좋다.

매시업 : 웹으로 제공하고 있는 정보(크롤링, API)와 서비스를 융합하여 새로운 소프트웨어나 서비스, 데이터베이스 등을 만드는 것을 말한다.

예를 들어 페이스북의 데이터가 도착할 때까지 기다리는 동안 트위터에서 도착한 데이터 처리를 미루고 싶진 않다.

🡺

병렬성 : 하나의 동작을 여러 서브동작으로 분할하고 각각의 서브동작을 다른 코어, 다른 CPU, 심지어 다른 기기로 할당할 수 있다.

동시성 : 즉 하나의 CPU 사용을 가장 극대화할 수 있도록 느슨하게 연관된 여러 작업을 수행해야 하는 상황이라면 원격 서비스 결과를 기다리거나, 데이터베이스 결과를 기다리면서 스레드를 블록하기 원치 않을 것이다.

**11.1 Future**

자바5부터 미래의 어느 시점에 결과를 얻는 모델에 활용할 수 있도록 Future 인터페이스를 제공하고 있다. 비동기 계산을 모델링하는데 Future를 이용할 수 있으며, Future는 계산이 끝났을 때 결과에 접근할 수 있는 레퍼런스를 제공한다.

// 비동기 API 형태  
ExecutorService executor = Executors.*newCachedThreadPool*();  
Future<Double> future = executor.submit(new Callable<Double>() {  
 public Double call() {  
 return *doSomeLongComputation*();  
 }  
});  
*doSomethingElse*();  
try {  
 Double result = future.get(1, TimeUnit.*SECONDS*);  
} catch (ExecutionException ee) {  
  
} catch (InterruptedException ie) {  
  
} catch (TimeoutException te) {  
  
}

다음에서 보는것과 같은 유형의 프로그래밍에서는 ExecutorService에서 제공하는 스레드가 시간이 오래 걸리는 작업을 처리하는 동안 우리 스레드로 다른 작업을 동시에 실행할 수 있다.

결과가 필요한 시점이 되었을 때 Future의 get 메서드로 결과를 가져올 수 있다.

🡺get 호출했을 때 이미 계산이 완료되어 결과가 준비되었다면 즉시 결과를 반환하지만

결과가 준비되지 않았다면 작업이 완료될 때까지 우리 스레드를 블록시킨다.

🡺작업이 영원히 끝나지 않으면 문제가 생길 수 있기 때문에 get 메서드를 오버로드해서 우리 스레드가 대기할 최대 타임아웃 시간을 설정하는 것이 좋다.

**11.1.1 Future 제한**

오래 걸리는 A라는 계산이 끝나면 그 결과를 다른 오래 걸리는 계산 B로 전달하시오. 그리고 B의 결과가 나오면 다른 질의의 결과와 B의 결과를 조합하시오

🡺Future로 이와 같은 동작을 구현하는 것은 쉽지 않다.

**선언형 기능이 필요 하다.**

-두 개의 비동기 계산 결과를 하나로 합친다. 두 가지 계산 결과는 서로 독립적일 수 있으며 또는 두 번째 결과가 첫 번째 결과에 의존하는 상황일 수 있다.

-Future 집합이 실행하는 모든 태스크의 완료를 기다린다.

-Future 집합에서 가장 빨리 완료되는 태스크를 기다렸다가 결과를 얻는다. (예를 들어 여러 태스크가 다양한 방식으로 같은 결과를 구하는 상황)

-프로그램적으로Future를 완료시킨다. (즉, 비동기 동작에 수동으로 결과 제공)

-Future 완료 동작에 반응한다 (즉, 결과를 기다리면서 블록되지 않고 결과가 준비되었다는 알림을 받은 다음에 Future의 결과로 원하는 추가 동작을 수행할 수 있음.)

🡺이럴때 CompletableFuture를 사용한다.

Future와 ComputableFuture의 관계를 Collection과 Stream의 관계에 비유할 수 있다.

**11.1.2 CompletableFuture로 비동기 애플리케이션 만들기**

다음과 같은 기술을 배울 수 있다.

-첫째, 고객에게 비동기 API를 제공하는 방법을 배운다.

-둘째, 동기 API를 사용해야 할 때 코드를 비블록으로 만드는 방법을 배운다. 두 개의 비동기 동작을 **파이프라인** 으로 만드는 방법과 두 개의 동작 결과를 하나의 비동기 계산으로 **합치는** 방법을 살펴 본다.

-셋째, 비동기 동작의 완료에 대응하는 방법을 배운다. 즉 모든 상점에서 가격 정보를 얻을 때까지 기다리는 것이 아니라 각 상점에서 가격 정보를 얻을 때마다 즉시 최저가격을 찾는 애플리케이션을 갱신하는 방버을 설명한다.

**동기 API와 비동기 API**

전통적인 동기 API에서는 메서듣를 호출한 다음에 메서드가 계산을 완료할 때까지 기다렸다가 메서드가 반환되면 호출자는 반환된 값으로 계속 다른 동작을 수행한다.

호출자와 피호출자가 각각 다른 스레드에서 실행되는 상황이었더라도 호출자는 피호출자의 동작 완료를 기다렸을 것이다.

🡺이처럼 동기 API를 사용하는 상황을 **블록 호출** 이라고 한다.

비동기 API 에서는 메서드가 즉시 반환되며 끝내지 못한 나머지 작업을 호출자 스레드와 동기적으로 실행될 수 있도록 **다른 스레드에** 할당한다.

🡺이와 같은 비동기 API를 사용하는 상황을 비블록 호출 이라고 한다.

다른 스레드에 할당된 나머지 계산 결과는 콜백 메서드를 호출해서 전달하거나 호출자가 ‘계산 결과가 끝날 때까지 기다림’ 메서드를 추가로 호출하면서 전달된다.

🡺주로 I/O 시스템 프로그래밍에서 이와 같은 방식으로 동작을 수행한다.

**11.2 비동기 API 구현**

// 동기 메서드  
public double getPrice(String product) {  
 return calculatePrice(product);  
}  
  
private double calculatePrice(String product) {  
 delay(); // 외부 API 호출 하는 부분을 대신 delay로 인위적 1초 지연  
 return Math.*random*() \* product.charAt(0) + product.charAt(1);  
}  
  
public void delay() {  
 try {  
 Thread.*sleep*(1000L);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
}

사용자가 이 API를 호출하면 비동기 동작이 완료될 때까지 1초 동안 블록된다.

**11.2.1 동기 메서드를 비동기 메서드로 변환**

Future는 결과값의 핸들일 뿐이며 계산이 완료되면 get 메서드로 결과를 얻을 수 있다.

getPriceAsync 메서드는 즉시 반환되므로 호출자 스레드는 다른 작업을 수행할 수 있다.

// 동기 메서드를 비동기 메서드로 변환  
public Future<Double> getPriceAsync(String product) {  
 CompletableFuture<Double> futurePrice = new CompletableFuture<>();  
 new Thread(() -> {  
 double price = calculatePrice(product);  
 futurePrice.complete(price);  
 }).start();  
  
 return futurePrice;  
}  
  
private double calculatePrice(String product) {  
 delay(); // 외부 API 호출 하는 부분을 대신 delay로 인위적 1초 지연  
 return Math.*random*() \* product.charAt(0) + product.charAt(1);  
}

위 코드에서 비동기 계산과 완료 결과를 포함하는 CompletableFuture 인스턴스를 만들었다.

그리고 실제 가격을 계산할 다른 스레드를 만든 다음에 오래 걸리는 계산 결과를 기다리지 않고 결과를 포함할 Future 인스턴스를 바로 반환했다.

요청한 제품의 가격 정보가 도착하면 complete 메서드를 이용해서 CompletableFuture를 종료할 수 있다.

// 비동기 API 사용  
long start = System.*nanoTime*();  
Future<Double> futurePrice = shop.getPriceAsync("my favorite product");  
long invocationTime = ((System.*nanoTime*() - start) / 1\_000\_000);  
System.*out*.println("Invocation returned after " + invocationTime + " msecs");  
*doSomethingElse*();  
try {  
 double price = futurePrice.get();  
 System.*out*.printf("Price is %.2f\n", price);  
} catch (Exception e) {  
}

클라이언트는 특정 제품의 가격 정보를 상점에 요청한다. 상점은 비동기 API를 제공하므로 즉시 Future를 반환한다.

클라이언트는 반환된 Future를 이용해서 나중에 결과를 얻을 수 있다.

그 사이 클라이언트는 다른 상점에 가격 정보를 요청하는 등 첫 번째 상점의 결과를 기다리면서 대기하지 않고 다른 작업을 처리할 수 있다.

나중에 클라이언트가 특별히 할일이 없으면 Future의 get 메서드를 호출한다. 이때 Future가 결과값을 가지고 있다면 Future에 포함된 값을 읽거나 아니면 값이 계산될 때까지 블록한다.

**11.2.2 에러 처리 방법**

가격을 계산하는 동안 에러가 발생하면 어떻게 될까? 예외가 발생하면 해당 스레드에만 영향을 미친다. 즉, 에러가 발생해도 가격 계산은 계속 진행되며 일의 순서가 꼬인다.

Get 메서드가 반환될 때까지 영원히 기다리게 될 수도 있다.

클라이언트는 타임아웃값을 받는 get 메서드의 오버로드 버전을 만들어 이 문제를 해결할 수 있다.

🡺하짖만 왜 에러가 발생했는지 알 수 있는 방법이 없다. 따라서 completeExceptionally 메서드를 이용해서 CompletableFuture 내부에서 발생한 예외를 클라이언트로 전달한다.

public Future<Double> getPriceAsync(String product) {  
 CompletableFuture<Double> futurePrice = new CompletableFuture<>();  
 new Thread(() -> {  
 try {  
 double price = calculatePrice(product);  
 futurePrice.complete(price);  
 } catch (Exception ex) {  
 futurePrice.completeExceptionally(ex);  
 }  
 }).start();  
 return futurePrice;  
}

**팩토리 메서드 supplyAsync로 CompletableFuture 만들기**

다음과 같이 좀 더 간단하게 CompletableFuture를 만드는 방법도 있다.

public Future<Double> getPriceAsync(String product) {  
 return CompletableFuture.*supplyAsync*(() -> calculatePrice(product));  
}

supplyAsync 메서드는 Supplier를 인수로 받아서 CompletableFuture를 반환한다.

CompletableFuture는 Supplier를 실행해서 비동기적으로 결과를 생성한다.

두 번째 인수를 받는 오버로드 버전의 supplyAsync 메서드를 이용해서 다른 Executor를 지정할 수 있다.

지금부터는 모든 API는 동기 방식의 블록 메서드라고 가정할 것이다.

블록 메서드를 사용할 수 밖에 없는 상황에서 비동기적으로 여러 상점에 질의하는 방법, 즉 한 요청의 응답을 기다리며 블록하는 상황을 피해 최저가격 검색 애플리케이션의 성능을 높일 수 있는 방법을 살펴보다.

**11.3 비블록 코드 만들기**

public static List<Shop> shops = Arrays.*asList*(  
 new Shop("BestPrice"),  
 new Shop("LetsSaveBig"),  
 new Shop("MyFavoriteShop"),  
 new Shop("BuyItAll");

// 순차 Stream 이용  
public static List<String> findPrices(String product) {  
 return *shops*.stream() // 순차 스트림으로 상점에서 가격 정보를 병렬로 요청한다.  
 .map(shop -> String.*format*("%s price is %.2f", shop.getName(), shop.getPrice(product)))  
 .collect(Collectors.*toList*());  
}

long start = System.*nanoTime*();  
System.*out*.println(*findPrices*("myPhone275"));  
long duration = (System.*nanoTime*() - start) / 1\_000\_000;  
System.*out*.println("Done in " + duration + " msecs");

**Done in 4032 msecs**

네 개의 상점에서 가격을 검색하는 동안 각각 1초(delay)의 대기시간이 있으므로 전체 가격 검색 결과는 4초보다 조금 더 걸린다.

**11.3.1 병렬 스트림으로 요청 병렬화하기**

// 병렬 Stream 이용

public static List<String> findPrices(String product) {  
 return *shops*.parallelStream() // 병렬 스트림으로 상점에서 가격 정보를 병렬로 요청한다.  
 .map(shop -> String.*format*("%s price is %.2f", shop.getName(), shop.getPrice(product)))  
 .collect(Collectors.*toList*());  
}

**Done in 1180 msecs**

병렬 스트림을 이용해서 순차 계산을 병렬 처리해서 성능을 개선할 수 있다.

4개의 코어에서 각각의 shope을 병렬로 처리해서 성능 향상이 나타났다.

**11.3.2 CompletableFuture로 비동기 호출 구현하기**

// CompletableFuture 이용

public static List<String> findPrices(String product) {  
 List<CompletableFuture<String>> priceFutures =  
 *shops*.stream()  
 .map(shop -> CompletableFuture.*supplyAsync*(

() -> shop.getName() + " price is " + shop.getPrice(product)))  
 .collect(Collectors.*toList*());

return priceFutures.stream()  
 .map(CompletableFuture::join)  
 .collect(Collectors.*toList*());  
}

**Done in 2005 msecs**

첫번째 map 연산은 List<CompletableFuture<String>>을 얻을 수 있다.

두번째 map 연산을 List<CompletableFuture<String>>에 적용할 수 있다. 즉 리스트의 모든 CompletableFuture에 join을 호출해서 모든 동작이 끝나기를 기다린다.

CompletableFuture 클래스의 join 메서드는 Future 인터페이스의 get 메서드와 같은 의미를 갖는다.

다만 join은 아무 예외도 발생시키지 않는다는 점이 다르다. 따라서 두 번째 map의 람다 표현식을 try/catch로 감쌀 필요가 없다.

두 map 연산을 하나의 스트림 처리 파이프라인으로 처리하지 않고 두 개의 스트림 파이프라인으로 처리했다.

🡺스트림 연산은 게으른 특성이 있으므로 하나의 파이프라인으로 연산을 처리했다면 모든 가격 정보 요청 동작이 동기적, 순차적으로 이루어지는 결과가 된다.

CompletableFuture로 각 상점의 정보를 요청할 때 기존 요청 작업이 완료되어야 join이 결과를 반환하면서 다음 상점으로 정보를 요청할 수 있기 때문이다.

**11.3.3 더 확장성이 좋은 해결 방법**

병렬 스트림 버전의 코드는 정확히 네 개의 상점에 하나의 스레드를 할당해서 네 개의 작업을 병렬로 수행하면서 검색 시간을 최소화할 수 있었다. 만약 4개 이상이 된다면?

**11.3.4 커스텀 Executor 사용하기**

**스레드 풀 크기 조절**

스레드 풀의 최적값을 찾는 방법을 제안한다.

N(thread) =N(CPU) \* U(CPU) \* (1 + W/C)

-N(CPU)는 Runtime.getRuntime().availableProcessors()가 반환하는 코어 수

-U(CPU)는 0과 1사이의 값을 갖는 CPU 활용 비율

-W/C 대기시간과 계산시간의 비율

// CompletableFuture 이용  
public static List<String> findPrices(String product) {  
 List<CompletableFuture<String>> priceFutures =  
 *shops*.stream()  
 .map(shop -> CompletableFuture.*supplyAsync*(() -> shop.getName() + " price is " + shop.getPrice(product), executor))  
 .collect(Collectors.*toList*());  
 return priceFutures.stream()  
 .map(CompletableFuture::join)  
 .collect(Collectors.*toList*());  
}  
  
private static final Executor executor = Executors.*newFixedThreadPool*(Math.*min*(*shops*.size(), 100), new ThreadFactory() {  
 @Override  
 public Thread newThread(Runnable r) {  
 Thread t = new Thread(r);  
 t.setDaemon(true);  
 return t;  
 }  
});

**Done in 1022 msecs**

우리가 만든는 풀은 데몬 스레드를 포함한다.

**데몬 스레드란?**

1. 일반 스레드를 돕는 보조적인 역할
2. 일반 스레드 종료 시, 데몬 스레드도 강제종료
3. Ex) GC
4. SetDaemon(true)로 호출

**스트림 병렬화와 CompletableFuture 병렬화**

1)병렬 스트림으로 변환해서 컬렉션을 처리 하는 방법

2)컬렉션을 반복하면서 CompletableFuture 내부의 연산으로 만드는 것

🡺CompletableFuture를 이용하면 전체적인 계산이 블록되지 않도록 **스레드풀의 크기를 조절할** 수 있다.

-I/O가 포함되지 않은 계산 중심의 동작을 실행할 때는 스트림 인터페이스가 가장 구현하기 간단하며 효율적일 수 있다. (모든 스레드가 계산 작업을 수행하는 상황에서는 프로세서 코어수 이상의 스레드를 가질 필요가 없다).

-반면 작업이 I/O를 기다리는 작업을 병렬로 실행할 때는 CompletableFuture가 더 많은 유연성을 제공하며 대기/계산(W/C)의 비율에 적합한 스레드 수를 설정할 수 있다.

🡺스트림은 게으른 특성 때문에 I/O를 실제로 언제 처리할지 예측하기 어려운 문제도 있다.

**11.4 비동기 작업 파이프라인 만들기**

Enum 으로 할인 코드 정의하기

Public class Discount {  
 public enum Code {  
 *NONE*(0), *SILVER*(5), *GOLD*(10), *PLATINUM*(15), *DIAMOND*(20);  
  
 private final int percentage;  
  
 Code(int percentage) {  
 this.percentage = percentage;  
 }  
 }

…  
}

**11.4.1 할인 서비스 구현**

상점에서 제공한 문자열 파싱은 다음처럼 Quote 클래스로 캡슐화할 수 있다.

class Quote {  
 private final String shopName;  
 private final double price;  
 private final Discount.Code discountCode;  
  
 public Quote(String shopName, double price, Discount.Code code) {  
 this.shopName = shopName;  
 this.price = price;  
 this.discountCode = code;  
 }  
  
 public static Quote parse(String s) {  
 String[] split = s.split(":");  
 String shopName = split[0];  
 double price = Double.*parseDouble*(split[1]);  
 Discount.Code discountCode = Discount.Code.*valueOf*(split[2]);  
 return new Quote(shopName, price, discountCode);  
 }  
  
 public String getShopName() {  
 return shopName;  
 }  
  
 public double getPrice() {  
 return price;  
 }  
  
 public Discount.Code getDiscountCode() {  
 return discountCode;  
 }  
}

class Discount {

…  
 public static String applyDisCount(Quote quote) {  
 return quote.getShopName() + " price is " +  
 Discount.*apply*(quote.getPrice(), quote.getDiscountCode());  
 }  
  
 private static double apply(double price, Code code) {  
 *delay*();  
 return (price \* (100 - code.percentage) / 100);  
  
 }  
  
 public static void delay() {  
 try {  
 Thread.*sleep*(1000L);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
}

다음 코드에서 보여주는 것처럼 Discount 서비스에서는 Quote 객체를 인수로 받아 할인된 가격 문자열을 반환하는 applyDiscount 메서드도 제공한다.

**11.4.2 할인 서비스 사용 (동기적 처리)**

Discount는 원격 서비스므로 1초의 지연이 추가되어 있다.

// 할인 서비스 사용  
public static List<String> findPrices(String product) {  
 return *shops*.stream()  
 .map(shop -> shop.getPrice(product)) // 각 상점에서 할인전 가격 얻기  
 .map(Quote::*parse*) // 상점에서 반환한 문자열을 Quote 객체로 변환  
 .map(Discount::*applyDisCount*) // Discount 서비스를 이용해서 각 Quote에 할인 적용  
 .collect(Collectors.*toList*());  
}

순차적과 동기 방식으로 findPrices 메서드를 구현한다.

세 개의 map 연산을 상점 스트림에 파이프라인으로 연결해서 원하는 결과를 얻었다.

-첫 번째 연산에서는 각 상점을 요청한 제품의 가격과 할인 코드로 변환한다

ex) myPhone275:1000:SILVER

-두 번째 연산에서는 이들 문자열을 파싱해서 Quote 객체를 만든다

-세 번째 연산에서는 원격 Discount 서비스에 접근해서(Dealy 1초) 최종 할인가격을 계산하고 가격에 대응하는 상점 이름을 포함하는 문자열을 반환한다.

🡺결국 가격 정보 요청 5초(순차기 때문에) + 할인 서비스 5초(순차기 때문에) 10초 이상 걸린다.

**11.4.3 동기 작업과 비동기 작업 조합하기**

private static final Executor executor = Executors.*newFixedThreadPool*(Math.*min*(*shops*.size(), 100), new ThreadFactory() {  
 @Override  
 public Thread newThread(Runnable r) {  
 Thread t = new Thread(r);  
 t.setDaemon(true);  
 return t;  
 }  
});  
  
// 동기 작업과 비동기 작업 조합하기  
public static List<String> findPrices (String product) {  
 List<CompletableFuture<String>> pricefutures = *shops*.stream()  
 .map(shop -> CompletableFuture.*supplyAsync*(() -> shop.getPrice(product), executor))

// 각 상점에서 할인전 가격을 비동기적으로 얻는다  
 .map(future -> future.thenApply(Quote::*parse*))  
 .map(future -> future.thenCompose(quote ->  
 CompletableFuture.*supplyAsync*(() -> Discount.*applyDisCount*(quote), executor)))  
 .collect(Collectors.*toList*());

// 결과 Future를 다른 비동기 작업과 조합해서 할인 코드를 적용한다.  
  
 return pricefutures.stream()  
 .map(CompletableFuture::join) // 스트림의 모든 Future가 종료되길 기다렸다가 각각의  
 .collect(Collectors.*toList*()); // 결과를 추출한다.  
}

이번에는 CompletableFuture 클래스의 기능을 이용해서 이들 동작을 비동기로 만들어야 한다.

**가격 정보 얻기**

팩토리 메서드 supplyAsync에 람다 표현식을 전달해서 비동기적으로 상점에서 정보를 조회했다.

첫 번째 변환의 결과는 Stream<CompletableFuture<String>> 이다.

Stream<CompletableFuture<String>> completableFutureStream = *shops*.stream()  
 .map(shop -> CompletableFuture.*supplyAsync*(() -> shop.getPrice(product), *executor*));

**Quote 파싱하기**

두 번째 변환 과정에서는 첫 번째 결과 문자열으 Quote로 변환한다. 파싱 동작에서는 Dealy가 없으므로 지연 없이 동작을 수행할 수 있다.

따라서 첫번째 과정에서 생성된 CompletableFuture에 thenApply 메서드를 호출한 다음 문자열을 Quote 인스턴스로 변환하는Function을 전달한다.

.map(future -> future.thenApply(Quote::*parse*))

thenApply 메서드는 CompletableFuture가 끝날 때까지 블록하지 않는다는 점을 주의해야 한다.

즉, Completablefuture가 동작을 완전히 완료한 다음에 thenApply 메서드로 전달된 람다 표현식을 적용할 수 있다.

**CompletableFuture<String> -> CompletableFuture<Quote>로 변환할 것이다.**

(map 을 사용했기 때문에 해당 형태로 형변환)

**CompletableFuture를 조합해서 할인된 가격 계산하기**

세 번째 map 연산에서는 상점에서 받은 할인전 가격에 원격 Discount 서비스에서 제공하는 할인율을 적용해야 한다. 이번에는 원격 실행이 포함되므로 이전 두 변환과 다르며 동기적으로 작업을 수행해야 한다.(1초의 delay)

람다 표현식으로 이 동작을 팩토리 메서드 supplyAsync에 전달할 수 있다.

그러면 다른 Compleablefuture가 반환된다.

결국 두 가지 CompletableFuture로 이루어진 연쇄적으로 수행되는 두개의 비동기 동작을 만들 수 있다.

1. **상점에서 가격 정보를 얻어 와서 Quote로 변환하기**
2. **변환된 Quote를 Discount 서비스로 전달해서 할인된 최종가격 획득하기**

자바9의 CompletableFuture API는 이와 같이 두 비동기 연산을 파이프라인으로 만들 수 있도록 thenCompose 메서드를 제공한다.

thenCompose는 첫 번째 연산의 결과를 두 번째 연산으로 전달한다.

세 개의 map 연산 결과 스트림의 요소를 리스트로 수집하면 List<CompletableFuture<String>> 형식의 자료를 얻을 수 있다. 마지막으로 Completablefuture가 완료되기를 기다렸다가 join으로 값을 추출 할 수 있다.

return pricefutures.stream()  
 .map(CompletableFuture::join)  
 .collect(Collectors.*toList*());

Async로 끝나지 않는 메서드는 이전 작업을 수행한 스레드와 같은 스레드에서 작업을 실행함을 의미하며, Async로 끝나는 메서드는 다음 작업이 다른 스레드에서 실행되도록 스레드 풀로 작업을 제출한다.

**11.4.4 독립 CompletableFuture와 비독립 CompletableFuture 합치기**

위에서는 첫 번째 Completablefuture에 thenCompose 메서드를 실행한 다음에 실행 결과를 첫 번째 실행 결과를 입력으로 받는 두 번째 CompletableFuture로 전달했다.

실전에서는 독립적으로 실행된 두개의 CompletableFuture 결과를 합쳐야 하는 상호아이 종종 발생한다. 물론 첫 번째 CompletableFuture의 동작 완료와 관계 없이 두 번쨰 CompletableFuture를 실행할 수 있어야 한다.

🡺이런 상황에서는 thenCombine 메서드를 사용한다.

List<CompletableFuture<Double>> priceFutures = new ArrayList<>();  
for (Shop4 shop : *shops*) {  
 CompletableFuture<Double> futurePriceInUSD =  
 CompletableFuture.*supplyAsync*(() -> shop.getPriceToDobule(product))  
 .thenCombine(  
 CompletableFuture.*supplyAsync*(  
 () -> ExchangeService.*getRate*(ExchangeService.Money.*EUR*, ExchangeService.Money.*USD*)),  
 (price, rate) -> price \* rate  
 );  
 priceFutures.add(futurePriceInUSD);  
}

thenCombine 메서드는 BiFunction을 두 번째 인수로 받는다. BiFunction은 두 개의 CompletableFuture 결과를 어떻게 합칠지 정의한다.

thenCombine 메서드에도 Async 버전이 존재한다. thenCombineAsync 메서드에서는 BiFunction이 정의하는 조합 동작이 스레드 풀로 제출되면서 별도의 태스크에서 비동기적으로 수행된다.

**11.4.5 Future의 리플렉션과 CompletableFuture의 리플렉션**

CompletableFuture는 람다 표현식을 사용한다. 덕분에 다양한 동기 태스크, 비동기 태스크를 활용해서 복잡한 연산 수행 방법을 효과적으로 쉽게 정의할 수 있는 선언형 API를 만들 수 있다.

ExecutorService executor = Executors.*newCachedThreadPool*();  
 List<Future<Double>> priceFutures = new ArrayList<>();  
 for (Shop4 shop : *shops*) {  
 final Future<Double> futureRate = executor.submit(new Callable<Double>() {  
 public Double call() { // 첫번째 퓨처  
 return ExchangeService.*getRate*(ExchangeService.Money.*EUR*, ExchangeService.Money.*USD*);  
 }  
 });  
 Future<Double> futurePriceInUSD = executor.submit(new Callable<Double>() {  
 public Double call() { // 두번째 퓨처  
 try {  
 double priceInEUR = shop.getPriceToDobule(product);  
 return priceInEUR \* futureRate.get(); // 가격과 환율을 곱하는 세번째 퓨처  
 } catch (InterruptedException | ExecutionException e) {  
 throw new RuntimeException(e.getMessage(), e);  
 }  
 }  
 });

Executor에 EUR과 USD 간의 환율 외부 서비스를 이용하는 Callable을 submit의 인수로 전달해서 첫 번째 Future를 만들었다. 그리고 상점에서 해당 제품 가격을 EUR로 변환하는 두 번째 Future를 만들었다.

마지막으로 가격과 환율을 곱하는 세 번째 Future를 만들었다. (앞에서 thenCombine 대신 thenOmbineAsync를 사용하는것과 동일)

🡺선언형이 얼마나 유용한지 보여주기 위한 예제임 (결과는 동일하다)

**11.5 CompletableFuture의 종료에 대응하는 방법**

실전에서 사용하는 다양한 우너격 서비스는 얼마나 지연될지 예측하기 어렵다.

서버 부하에서 네트워크 문제에 이르기까지 다양한 지연 요소가 있기 때문이다.

**11.5.1 최저가격 검색 애플리케이션 리팩토링**

먼저 모든 가격 정보를 포함할 때까지 리스트 생성을 기다리지 않도록 프로그램을 고쳐야 한다.

public static Stream<CompletableFuture<String>> findPricesStream(String product) {  
 return *shops*.stream()  
 .map(shop -> CompletableFuture.*supplyAsync*(() -> shop.getPrice(product), *executor*))  
 .map(future -> future.thenApply(Quote::*parse*))  
 .map(future -> future.thenCompose(quote ->  
 CompletableFuture.*supplyAsync*(  
 () -> Discount.*applyDisCount*(quote), *executor*)));  
}

이제 findPricesStream 메서드 내부에서 세 가지 map연산을 적용하고 반환하는 스트림에 네번째 map 연산을 적용하자.

새로 추가한 연산은 단순하게 각 CompletableFuture에 동작을 등록한다.

CompletableFuture에 등록된 동작은 CompletableFuture의 계산이 끝나면 값을 소비한다.

자바8의 CompletableFuture API는 **thenAccep**라는 메서드로 이기능을 제공한다.

*findPricesStream*("myPhone").map(f -> f.thenAccept(System.*out*::println));

thenCompose, thenCombine 메서드와 마찬가지로 thenAccept에도 thenAcceptAsync 라는 Async 버전이 존재한다.

thenAcceptAsync 메서드는 CompletableFuture가 완료된 스레드가 아니라 새로운 스레드를 이용해서 Consumer를 실행한다.

thenAccept 메서드는 Completablefuture가 생성한 결과를 어떠헥 소비할지 미리 지정했으므로 CompletableFuture<Void>를 반환한다.

따라서 네 번째 map 연산은 <CompletableFuture<Void>>로 반환한다.

🡺출력하기 위해서는 다음 코드에서 보여주는 것 처럼 모든 CompletableFuture<Void>를 배열로 추가하고 실행 결과를 기다려야 한다.

//CompletableFuture 종료에 반응하기  
CompletableFuture[] futures = (CompletableFuture[])*findPricesStream*("myPhone")  
 .map(f -> f.thenAccept(System.*out*::println))  
 .toArray(size -> new CompletableFuture[size]);  
  
CompletableFuture.*allOf*(futures).join();

팩토리 메서드 allOf는 CompletableFuture 배열을 입력으로 받아 Completablefuture<Void>를 반환한다. 전달된 모든 Completablefuture가 완료되어야 Completablefuture<Void>가 완료된다.

따라서 allOf 메서드가 반환하는 Completablefuture에 join을 호출하면 원래 스트림의 모든 Completablefuture의 실행 완료를 기다릴 수 있다.

🡺이를 이용해서 최저 가격 검색 애플리케이션은 ‘모든 상점이 결과를 반환했거나 타임아웃되었음’ 같은 메시지를 사용자에게 보여줌으로써 사용자는 추가 정보를 기다리지 않아도 된다.

반면 Completablefuture중 하나의 작업이 끝나길 기다리는 상황인 경우는

**anyOf**를 사용한다.

**CHAPTER 12. 새로운 날짜 시간 API**

자바 1.0에서는 java.util.Date 클래스 하나로 날짜와 시간 관련 기능을 제공했다.

날짜를 의미하는 Date라는 클래스의 이름과 달리 Date클래스는 특정 시점을 날짜가 아닌 밀리초 단위로 표현한다.

게다가 1900년을 기준으로 하는 오프셋, 0에서 시작하는 달 인덱스 등 모호한 설계로 유용성이 떨어졌다.

Date date = new Date(114, 2, 18);

🡺Tue Mar 18 00:00:00 CET 2014

결과가 직관적이지 않다. 출력 결과에서 알 수 있듯이 Date는 JVM 기본 시간대인 CET, 즉

중앙 유럽 시간대를 사용했다.

자바 1.0의 Date 클래스에 문제가 있다는 점에는 의문의 여지가 없었지만 과거 버전과 호환성을 깨뜨리지 않으면서 이를 해결할 수 있는 방법이 없었다.

🡺결과적으로 자바 1.1에서는 Date 클래스의 여러 메서드를 사장 시키고 java.util.Calendar 라는 클래스를 대안으로 제공했다.

🡺안타깝게고 Calendar 클래스 역시 쉽게 에러를 일으키는 설계 문제를 갖고 있었다.

🡺Date와 Calendar 두 가지 클래스가 등장하면서 개발자들에게 혼란이 가중 되었다.

DateFormat에도 문제가 있었다. (스레드에 안전하지 않다.)

Date와 Calendar 모두 가변 클래스다.

이런 부실한 날짜와 시간 라이브러리 때문에 많은 개발자는 Joda-Time 같은 써드파티 날짜와 시간 라이브러리를 사용했다.

🡺결국 자바8에서는 Joda-Time의 많은 기능을 java.time 패키지로 추가했다.

**12.1 LocalDate, LocalTime, Instant, Duration, Period**

java.time 패키지는 LocalDate, LocalTime, LocalDateTime, Instant, Duration, Period 등 새로운 클래스를 제공한다.

**12.1.1 LocalDate와 LocalTime 사용**

LocalDate 인스턴스는 시간을 제외한 날짜를 표현하는 불변 객체다.

정적 팩토리 of로 LocalDate 인스턴스를 만들 수 있다.

LocalDate date = LocalDate.*of*(2014, 3, 18); // 2014-03-18  
int year = date.getYear(); // 2014  
Month month = date.getMonth(); // MARCH  
int day = date.getDayOfMonth(); // 18  
DayOfWeek dow = date.getDayOfWeek(); // TUESDAY  
int len = date.lengthOfMonth(); // 31 (3월의 일수)  
boolean leap = date.isLeapYear(); // false (윤년이 아님)

팩토리 메서드 now는 시스템 시계의 정보를 이용해서 현재 날짜 정보를 얻는다.

// 현재 날짜 정보  
LocalDate today = LocalDate.*now*();

Get 메서드에 TemporalField를 전달해서 정보를 얻는 방법도 있따.

TemporalField는 시간 관련 객체에서 어떤 필드의 값에 접근할지 정의하는 인터페이스다.

// TemporalField를 이용해서 LocalDate 값 읽기  
int year = date.get(ChronoField.*YEAR*);  
int month = date.get(ChronoField.*MONTH\_OF\_YEAR*);  
int day = date.get(ChronoField.*DAY\_OF\_MONTH*);

시간은 LocalTime 클래스로 표현할 수 있다. 오버로드 버전의 두 가지 정적 메서드 of로 LocalTime 인스턴스를 만들 수 있다.

// LocalTime 만들고 값 읽기  
LocalTime time = LocalTime.*of*(13, 45, 20);  
int hour = time.getHour(); // 13  
int minute = time.getMinute(); // 45  
int second = time.getSecond(); // 20

날짜와 시간 문자열로 LocalDate와 LocalTime의 인스턴스를 만드는 방법도 있따.

다음처럼 parse 정적 메서드를 사용할 수 있다.

// 날짜와 시간 문자열로 인스턴스 만들기  
LocalDate date = LocalDate.*parse*("2014-03-18");  
LocalTime time = LocalTime.*parse*("13:45:20");

Parse 메서드에 DateTimeFormatter를 전달할 수도 있다. DateTimeFormatter의 인스턴스는 날짜, 시간 객체의 형식을 지정한다.

DateTimeFormatter는 이전에 설명했던 java.util.DateFormat 클래스를 대체하는 클래스다.

**12.1.2 날짜와 시간 조합**

LocalDateTime은 LocalDate와 LocalTime을 쌍으로 갖는 복합 클래스다.

날짜와 시간을 조합할 수도 있다.

// LocalDateTime (LocalDate + LocalTime)  
LocalDateTime dt1 = LocalDateTime.*of*(2014, Month.*MARCH*, 18, 13, 45, 20);  
LocalDateTime dt2 = LocalDateTime.*of*(date, time);  
LocalDateTime dt3 = date.atTime(13, 45, 20);  
LocalDateTime dt4 = date.atTime(time);  
LocalDateTime dt5 = time.atDate(date);

toLocalDate나 toLocalTime 메서드로 LocalDate나 LocalTime 인스턴스를 추출할 수 있다.

LocalDate date1 = dt1.toLocalDate();  
LocalDate time1 = dt1.toLocalTime();

**12.1.3 Instant: 기계의 날짜와 시간**

사람은 보통 주, 날짜, 시간, 분으로 날짜와 시간을 계산한다. 하지만 기계에서는 이와 같은 단위로 시간을 표현하기가 어렵다.

기계의 관점에서는 연속된 시간에서 특정 지점을 하나의 큰 수로 표현하는 것이 가장 자연스러운 시간 표현 방법이다.

새로운 java.time.Instant 클래스에서는 이와 같은 기계적인 관점에서 시간을 표현한다.

Instant 클래스는 유닉스 **에포크 시간**(1970년 1월 1일 0시 0분 0초 UTC) 기준으로 특정 지점까지의 시간을 초로 표현한다.

Instant 클래스는 나노초(10억분의 1초)의 정밀도를 제공한다. 또한 오버로드된 ofEpochSecond 메서드 버전에서는 두 번째 인수를 이용해서 나노초 단위로 시간을 보정할 수 있다.

// 기계의 날짜 Instant  
Instant.*ofEpochSecond*(3);  
Instant.*ofEpochSecond*(3, 0);  
Instant.*ofEpochSecond*(2, 1\_000\_000\_000);  
Instant.*ofEpochSecond*(4, -1\_000\_000\_000);

**12.1.4 Duration과 Period 정의**

지금 까지 살펴본 모든 클래스는 Temporal 인터페이스를 구현하고 있다.

Temporal 인터페이스는 특정 시간을 모델링하는 객체의 값을 어떻게 읽고 조작할지 정의한다.

Dureation 클래스는 정적 팩토리 메서드 between으로 두 시간 객체 사이의 지속시간을 만들 수 있다.

LocalTime, LocalDateTime, Instant로 Duration을 만들 수 있다.

Duration d1 = Duration.*between*(time, time2);  
Duration d2 = Duration.*between*(dateTime1, dateTime2);  
Duration d3 = Duration.*between*(instant1, instant2);

LocalDateTime은 사람이 사용하도록, Instant는 기계가 사용하도록 만들어진 클래스로 두 인스턴스는 서로 혼합할 수 없다.

또한 Duratino 클래스는 초와 나노초로 시간 단위를 표현하므로 between 메서드에 LocalDate를 전달할 수 없다.

년, 월, 일로 시간을 표현할 때는 Period 클래스를 사용한다. 즉 Peroid 클래스의 팩토리 메서드 between을 이용하면 두 LocalDate의 차이를 확인할 수 있다.

// Period  
Period tenDays = Period.*between*(LocalDate.*of*(2014, 3, 8), LocalDate.*of*(2014, 3, 18));

Duration과 Period 클래스는 자신의 인스턴스를 만들 수 있도록 다양한 팩토리 메서드를 제공한다.

// Duration과 Period 만들기  
Duration threeMinutes = Duration.*ofMinutes*(3);  
Duration threeMinutes2 = Duration.*of*(3, ChronoUnit.*MINUTES*);  
  
Period tenDays = Period.*ofDays*(10);  
Period threeWeeks = Period.*ofWeeks*(3);  
Period twoYearsSixMonthsOneDay = Period.*of*(2, 6, 1);

Duration = 시간단위 (시간)

Period = 년,월,일 (기간)

**지금까지 살펴본 모든 클래스는 불변이다.**

**12.2 날짜 조정, 파싱, 포매팅**

withAttribute 메서드로 기존의 LocalDate를 바꾼 버전을 직접 간단하게 만들 수 있다.

// 절대적인 방식으로 LocalDate의 속성 바꾸기  
LocalDate abDate1 = LocalDate.*of*(2014, 3, 18); // 2014-03-18  
LocalDate abDate2 = abDate1.withYear(2011); // 2011-03-18  
LocalDate abDate3 = abDate2.withDayOfMonth(25); // 2011-03-25  
LocalDate abDate4 = abDate3.with(ChronoField.*MONTH\_OF\_YEAR*, 9); // 2011-09-25

abDate3.with(ChronoField.MONTH\_OF\_YEAR, 9) 와 같이 첫번째 인수로 TemporalField를 갖는 메서드를 사용하면 좀 더 범용적으로 메서드를 활용할 수 있다.

마지막 with 메서드는 get메서드와 쌍을 이룬다.

🡺get과 with 메서드로 Temporal 객체의 필드값을 읽거나 고칠 수 있다.

어떤 Temporal 객체가 지정된 필드를 지원하지 않으면 UnsupportedTemporalTypeExceptino이 발생한다.

Ex) Instan에 ChronField.MONTH\_OF\_YEAR를 사용하거나

LocalDate에 ChronoField.NANO\_OF\_SECOND를 사용

선언형으로 LocalDate를 사용하는 방법도 있다.

// 상대적인 방식으로 LocalDate 속성 바꾸기  
LocalDate reDate1 = LocalDate.*of*(2014, 3, 18); // 2014-03-18  
LocalDate reDate2 = reDate1.plusWeeks(1); // 2014-03-25  
LocalDate reDate3 = reDate2.minusYears(3); // 2011-03-25  
LocalDate reDate4 = reDate3.plus(6, ChronoUnit.*MONTHS*); // 2011-09-25

With, get 메서드와 비슷한 plus, minus 메서드를 사용했다. plus, minus 메서드도 Temporal 인터페이스에 정의되어 있다.

**12.2.1 TemporalAdjusters 사용하기**

지금까지 살펴본 날짜 조정 기능은 비교적 간단한 편에 속한다.

때로는 다음 주 일요일, 돌아오는 평일, 어떤 달의 마지막 날 등 좀 더 복잡한 날짜 조정 기능이 필요할 것이다.

TemporalAdjuster를 전달하는 방법으로 문제를 해결할 수 있다.

LocalDate date1 = LocalDate.*of*(2014, 3, 18);  
LocalDate date2 = date1.with(*nextOrSame*(DayOfWeek.*SUNDAY*));  
LocalDate date3 = date2.with(*lastDayOfMonth*());

필요한 기능이 정의되어 있지 않을 때는 비교적 쉽게 커스텀 TemporalAdjuster 구현을 만들 수 있다. 실제로 TemporalAdjuster 인터페이스는 다음처럼 하나의 메서드만 정의한다.

@FunctionalInterface  
public interface TemporalAdjuster {Temporal adjustInto(Temporal temporal);  
  
}

// TemporalAdjuster 인터페이스를 구현하는 NextWorkingDay 클래스를 구현하시오.  
// 이 클래스는 날짜를 하루씩 다음날로 바꾸는데 이때 토요일과 일요일은 건너뛴다.  
// 만약 날짜가 평일이 아니라면, 즉 토요일이나 월요일이라면 월요일로 이동한다.  
  
// 람다 표현식  
LocalDate d2 = LocalDate.*now*().with(temporal -> {  
 DayOfWeek dow = DayOfWeek.*of*(temporal.get(ChronoField.*DAY\_OF\_WEEK*));  
 int dayToAdd = 1;  
 if (dow == DayOfWeek.*FRIDAY*) { // 금요일이면 3일 추가  
 dayToAdd = 3;  
 } else if (dow == DayOfWeek.*SATURDAY*) { // 토요일이면 2일 추가  
 dayToAdd = 2;  
 }  
 return temporal.plus(dayToAdd, ChronoUnit.*DAYS*);  
});  
  
// TemporalAdjuster 람다 표현식  
TemporalAdjuster nextWorkingDay = TemporalAdjusters.*ofDateAdjuster*(  
 temporal -> {  
 DayOfWeek dow = DayOfWeek.*of*(temporal.get(ChronoField.*DAY\_OF\_WEEK*));  
 int dayToAdd = 1;  
 if (dow == DayOfWeek.*FRIDAY*) { // 금요일이면 3일 추가  
 dayToAdd = 3;  
 } else if (dow == DayOfWeek.*SATURDAY*) { // 토요일이면 2일 추가  
 dayToAdd = 2;  
 }  
 return temporal.plus(dayToAdd, ChronoUnit.*DAYS*);  
 });  
LocalDate d3 = LocalDate.*now*().with(nextWorkingDay);

**12.2.2 날짜와 시간 객체 출력과 파싱**

날짜와 시간 관련 작업에서 포매팅과 파싱은 서로 떨어질 수 없는 관계다.

심지어 포매팅과 파싱 전용 패키지인 java.time.format이 새로 추가되었다.

이 패키지에서 가장 중요한 클래스는 DateTimeFormatter다.

DateTimeFormatter 클래스는 BASIC\_ISO\_DATE와 ISO\_LOCAL\_DATE 등의 상수를 미리 정의하고 있다.

두개의 서로 다른 포매터로 문자열을 만드는 예제다.

// DateTimeFormatter ( 날짜 -> 문자열 )  
LocalDate localDate = LocalDate.*of*(2014, 3, 18);  
String s1 = localDate.format(DateTimeFormatter.*BASIC\_ISO\_DATE*); // 20140318  
String s2 = localDate.format(DateTimeFormatter.*ISO\_LOCAL\_DATE*); // 2014-03-18

반대로 날짜나 시간을 표현하는 문자열을 파싱해서 날짜 객체를 다시 만들 수 있다.

// DateTimeFormatter ( 문자열 -> 날짜 )  
LocalDate date1 = LocalDate.*parse*("20140318", DateTimeFormatter.*BASIC\_ISO\_DATE*);  
LocalDate date2 = LocalDate.*parse*("2014-03-18", DateTimeFormatter.*ISO\_LOCAL\_DATE*);

기존의 java.util.Dateformat 클래스와 달리 모든 DateTimeFormatter는 스레드에서 안전하게 사용할 수 있는 클래스다.

DateTimeFormatter 클래스는 특정 패턴으로 포매터를 만들 수 있는 정적 팩토리 메서드도 제공한다.

// DateTimeFormatter ( 문자열 -> 날짜 )  
DateTimeFormatter formatter = DateTimeFormatter.*ofPattern*("dd/MM/yyyy");  
LocalDate localDate1 = LocalDate.*of*(2014, 3, 18);  
String formattedDate = localDate1.format(formatter);  
LocalDate localDate2 = LocalDate.*parse*(formattedDate, formatter);

LocalDate의 format 메서드는 요청 형식의 패턴에 해당하는 문자열을 생성한다. 그리고 정적 메서드 parse는 같은 포매터를 적용해서 생성된 문자열을 파싱함으로써 다시 날짜를 생성한다.

// 지역화된 DateTimeFormatter 만들기  
DateTimeFormatter italianFormatter = DateTimeFormatter.*ofPattern*("d. MMMM yyyy", Locale.*ITALIAN*);  
LocalDate localDateFormatter1 = LocalDate.*of*(2014, 3, 18);  
String formattedDate2 = localDateFormatter1.format(italianFormatter);  
LocalDate localDateFormatter2 = LocalDate.*parse*(formattedDate2, italianFormatter);

DateTimeFormatterBuilder 클래스로 복합적인 포매터를 정의해서 좀 더 세부적으로 포매터를 제어할 수 있다.

// DateTimeFormatter 만들기  
DateTimeFormatter italianFormatter2 = new DateTimeFormatterBuilder()  
 .appendText(ChronoField.*DAY\_OF\_MONTH*)  
 .appendLiteral(". ")  
 .appendText(ChronoField.*MONTH\_OF\_YEAR*)  
 .appendLiteral(" ")  
 .appendText(ChronoField.*YEAR*)  
 .parseCaseInsensitive()  
 .toFormatter(Locale.*ITALIAN*);

**12.3 다양한 시간대와 캘린더 활용 방법**

새로운 날짜와 시간 API의 큰 편리함 중 하나는 시간대를 간단하게 처리할 수 있다는 점이다.

기존의 java.util.TimeZone 🡺 java.time.ZoneId 클래스가 새로 등장했다.

다른 클래스와 마찬가지로 ZoneId는 불변 클래스다.

다음 처럼 지역 ID로 특정 ZoneID를 구분한다.

ZoneId romeZone = ZoneId.*of*("Europe/Rome");

ZoneId 객체를 얻은 다음에는 LocalDate, LocalDateTime, Instant를 이용해서 ZonedDateTime 인스턴스로 변환할 수 있다.

ZoneId romeZone = ZoneId.*of*("Europe/Rome");

// 특정 시점에 시간대 적용  
LocalDate zonedDate1 = LocalDate.*of*(2014, Month.*MARCH*, 18);  
ZonedDateTime zdt1 = zonedDate1.atStartOfDay(romeZone);  
  
LocalDateTime zonedDate2 = LocalDateTime.*of*(2014, Month.*MARCH*, 18, 13, 45);  
ZonedDateTime zdt2 = zonedDate2.atZone(romeZone);  
  
Instant instant = Instant.*now*();  
ZonedDateTime zdt3 = instant.atZone(romeZone);

LocalDateTime = LocalDate + LocalTime

ZonedDateTime = LocalDateTime + ZonedID

**12.3.1 UTC/GMT 기준의 고정 오프셋**

때로는 UTC(협정 세계시) / GMT (그리니치 표준시)를 기준으로 시간대를 표현하기도 한다.

Ex) 뉴욕은 런던보다 5시간 느리다. (서머타임을 제대로 처리 못해서 권장 하는 방법 아니다)

// 뉴욕은 런던보다 5시간 느리다  
ZoneOffset newYorkOffset = ZoneOffset.*of*("-05:00");

ISO-8601 캘린더 시스템에서 정의하는 UTC/GMT와 오프셋으로 날짜와 시간을 표현하는 OffsetDateTime을 만드는 방법도 있다.

ZoneOffset newYorkOffset = ZoneOffset.*of*("-05:00");  
  
LocalDateTime offsetDateTime = LocalDateTime.*of*(2014, Month.*MARCH*, 18, 13, 45);  
OffsetDateTime dateTimeInNewYokr = OffsetDateTime.*of*(offsetDateTime, newYorkOffset);

**12.3.2 대안 캘린더 시스템 사용하기**

ISO-8601 캘린더는 실질적으로 전 세계 통용된다.

하지만 Java8에서는 추가로 4개의 캘린더를 제공한다.

ex) ThaiBuddhistDate, MinguoDate, JapaneseDate, HijrahDate

**날짜와 시간API 설계자는 ChronoLocalDate보다 LocalDate를 사용하라고 권고한다.**

**CHAPTER 13 함수형 관점으로 생각하기**

**13.1 시스템 구현과 유지보수**

자바 8의 스트림을 이용하면 잠금 문제를 신경 쓰지 않을 수 있다.

🡺 단, 자바 8의 스트림을 이용하려면 상태 없는 동작이어야 한다는 조건을 만족해야 한다.

또한 쉽게 유지보수할 수 있으려면 프로그램이 어떤 모습이어야 할까?

🡺프로그램이 시스템의 구조를 이해하기 쉽게 클래스 계층으로 반영한다면 좋을 것이다.

🡺시스템의 각 부분의 상호 의존성을 가리키는 결합성(coupling)과 시스템의 다양한 부분이 서로 어떤 관계를 갖는지 가리키는 응집성(cohesion)이라는 소프트웨어 엔지니어링 도구로 프로그램 구조를 평가할 수 있다.

유지보수 중 코드 크래시 디버깅 문제를 가장 많이 겪게 된다.

🡺함수형 프로그래밍이 제공하는 부작용 없음과 불변성이라는 개념이 이 문제를 해결하는데 도움

**13.1.1 공유된 가변 데이터**

변수가 예상하지 못한 값을 갖는 이유는 결국 우리가 유지보수하는 시스템의 여러 메서드에서 공유된 가변 데이터 구조를 읽고 갱신하기 때문이다.

어떤 자료구조도 바꾸지 않는 시스템이 있다고 가정하자. 예상하지 못하게 자료구조의 값이 바뀔 일이 없으니 얼마나 유지보수하기 쉽겠는가!

자신을 포함하는 클래스의 상태 그리고 다른 객체의 상태를 바꾸지 않으며 return문을 통해서만 자신의 결과를 반환하는 메서드를 순서(pure)메서드 또는 부작용 없는(side-effect free) 메서드라고 부른다.

구체적으로 부작용은 무엇일까? 함수 내에 포함되지 못한 기능을 부작용이라고 한다.

-자료구조를 고치거나 필드에 값을 할당

-예외 발생

-파일에 쓰기 등의 I/O 동작 수행

불변 객체를 이용해서 부작용을 없애는 방법도 있다.(불변 객체는 인스턴스화한 다음에는 객체의 상태를 바꿀 수 없는 객체이므로 함수 동작에 영향을 받지 않는다.)

**13.1.2 선언형 프로그래밍**

프로그램으로 시스템을 구현하는 방식은 크게 두 가지로 구분할 수 있다. (명령형, 선언형)

어떻게(how)에 집중하는 프로그래밍 형식은 고전의 객체지향 프로그래밍에서 이용하는 방식이다.

때로는 이를 명령형 프로그래밍이라고 부르기도 하는데 다음 코드에서 부여주는 것 처럼(할당, 조건문, 분기문, 루프 등) 명령어가 컴퓨터의 저수준 언어와 비슷하게 생겼기 때문이다.

public static void Command() {  
 Transaction mostExpensive = transactions.get(0);  
 if (Transaction == null)  
 throw new IllegalArgumentException("Empty list of transactions");  
  
 for (Transaction t : transactions.subList(1, transactions.size())) {  
 if (t.getValue() > mostExpensive.getValue()) {  
 mostExpensive = t;  
 }  
 }  
}

무엇을(what)에 집중하는 방식도 있다. 다음과 같은 질의를 만들 수 있었다.

public static void Function() {  
 Optional<Transaction> mostExpensive = transactions.stream()  
 .max(comparing(Transaction::getValue));  
}

이와 같은 구현 방식을 내부 반복이라고 한다.

질의문 자체로 문제를 어떻게 푸는지 명확하게 보여준다는 것이 내부 반복 프로그래밍의 큰 장점이다.

🡺선언형 프로그래밍이라고 부르기도 한다.

**13.1.3 왜 함수형 프로그래밍인가?**

함수형 프로그래밍은 선언형 프로그래밍을 따르는 대표적인 방식이며, 이전에 설명한 것처럼 부작용이 없는 계산을 지향한다. 선언형 프로그래밍과 부작용을 멀리한다는 두 가지 개념은 좀더 쉽게 시스템을 구현하고 유지보수하는데 도움을 준다.

**13.2 함수형 프로그래밍이란 무엇인가?**

함수를 이용하는 프로그래밍이다.

🡺그럼 함수란 무엇인가?

함수형 프로그래밍에서 함수란 수학적인 함수와 같다. 즉, 함수는 0개 이상의 인수를 가지며, 한 개 이상의 결과를 반환하지만 부작용이 없어야 한다.

자바와 같은 언어에서는 바로 수학적인 함수냐 아니냐가 메서드와 함수를 구분하는 핵심이다.

결론적으로 함수 그리고 if-then-else 등의 수학적 표현만 사용하는 방식을 순수 함수형 프로그래밍이라고 하며, 시스템의 다른 부분에 영향을 미치지 않는다면 내부적으로는 함수형이 아닌 기능도 사용 하는 방식을 함수형 프로그래밍이라고 한다.

**13.2.1 함수형 자바**

실질적으로 자바로는 완벽한 순수 함수형 프로그래밍을 구현하기 어렵다.

Ex)I/O 모델 자체에는 부작용 메서드가 포함된다. (Scanner.nextLine을 호출하면 파일의 행을 소비한다. 즉, Scanner.nextLine을 두 번 호출하면 다른 결과가 반환될 가능성이 있다.)

하지만 시스템의 컴포넌트가 순수한 함수형인 것처럼 동작하도록 코드를 구현할 수 있다.

자바에서는 순수 함수형이 아니라 함수형 프로그램을 구현할 것이다.

🡺실제 부작용이 있지만 아무도 이를 보지 못하게 함으로써 함수형을 달성할 수 있다.

함수나 메서드는 지역 변수만을 변경해야 함수형이라 할 수 있다. 그리고 함수나 메서드에서 참조하는 객체가 있다면 그 객체는 불변 객체여야 한다. (즉 객체의 모든 필드가 final 이어야 하고 모든 참조 필드는 불변 객체를 직접 참조해야 한다.)

이 외에도 함수형이라면 함수나 메서드가 어떤 예외도 일으키지 않아야 한다.

예외가 발생하면 이전에 설명한 것처럼 블랙박스 모델에서 return으로 결과를 반환할 수 없게 될 수 있기 때문이다.

🡺수학적 함수는 주어진 인수값에 대응하는 하나의 결과를 반환한다.

예외를 발생 시키지 않으려면 어떻게 할까?

🡺바로 Optional<T>를 사용하면 이 문제를 해결할 수 있다.

🡺double sqrt(double) 대신 Optional<Double> sqrt를 이용하면 예외 없이도 결과값으로 성공했는지 수행하지 못했는지 확인할 수 있다.

마지막으로 함수형에서는 비함수형 동작을 감출 수 있는 상황에서만 부작용을 포함하는 라이브러리 함수를 사용해야 한다. (즉, 먼저 자료구조를 복사한다든가 발생할 수 있는 예제를 적절하게 내부적으로 처리함으로써 자료구조의 변경을 호출자가 알 수 없도록 감춰야 한다.)

이와 같은 설명을 주석으로 표현하거나 **마커 어노테이션** 으로 메서드를 정의할 수 있다.

public @interface InProgress{  
   
}

@Deprecated  
public Date(int year, int month, int date) {  
 this(year, month, date, 0, 0, 0);  
}

**13.2.2 참조 투명성**

부작용을 감춰야 한다라는 제약은 참조 투명성 개념으로 귀결된다.

즉 같은 인수로 함수를 호출했을 때 항상 같은 결과를 반환한다면 참조적으로 투명한 함수라고 표현한다.

Ex) “raoul”.replace(‘r’, R’) 이라는 코드는 항상 같은 결과가 나오므로 String.replace는 참조적으로 투명하다. (새로운 String반환)

다시 말해, 함수는 어떤 입력이 주어졌을 때 언제, 어디서 호출하든 같은 결과를 생성해야 한다.

참조 투명은 프로그램 이해에 큰 도움을 준다. 또한 참조 투명성은 비싸거나 오랜 시간이 걸리는 연산을 기억화 또는 캐싱을 통해 다시 계산하지 않고 저장하는 최적화 기능도 제공한다.

**13.2.3 객체지향 프로그래밍과 함수형 프로그래밍**

사실 자바9은 함수형 프로그래밍을 익스트림 객체지향 프로그래밍의 일종으로 간주한다.

대부분의 자바 프로그래머는 무의식적으로 함수형 프로그래밍의 일부 기능과 익스트림 객체지향 프로그래밍의 일부 기능을 사용하게 될 것이다.

프로그래밍 형식을 스펙트럼으로 표현하자면

스펙트럼의 한 쪽 끝에는 모든 것을 객체로 간주하고 프로그램이 객체의 필드를 갱신하고, 메서드를 호출하고, 관련 객체를 갱신하는 방식으로 동작하는 익스트림 객체지향 방식이 위치한다. (명령형)

스펙트럼 반대쪽 끝에는 참조적 투명성을 중시하는, 즉 변화를 허용하지 않는 함수형 프로그래밍 형식이 위치한다. (함수형)

실제 자바 프로그래머는 이 두가지 프로그래밍 형식을 혼합한다.

**13.2.4 함수형 실전 연습**

List<Integer>가 주어졌을 때 이것의 모든 서브집합의 멤버로 구성된 List<List<Integer>>를 만드는 프로그램을 만든다고 가정.

ex) {1,4,9}의 서브집합 {1,4,9},{1,4},{1,9},{4,9},{1},{4},{9}, {}다. (총 8개)

public static List<List<Integer>> subsets(List<Integer> list) {  
 if (list.isEmpty()) { // 입력 리스트가 비었다면 빈 리스트 자신이 서브집합  
 List<List<Integer>> ans = new ArrayList<>();  
 ans.add(Collections.*emptyList*());  
 return ans;  
 }  
  
 Integer first = list.get(0);  
 List<Integer> rest = list.subList(1, list.size());  
  
 List<List<Integer>> subans = *subsets*(rest); // 빈 리스트가 아니면 먼저 하나의 요소를 꺼내고 나머지 요소의 모든 서브집합을 찾아서 subans로 전달한다. subans는 절반의 정답을 포함한다.  
 List<List<Integer>> subans2 = *insertAll*(first, subans); // 정답의 나머지 절반을 포함하는 subans2는 subans의 모든 리스트에 처음 꺼낸 요소를 앞에 추가해서 만든다.  
 return *concat*(subans, subans2); // subans, subans2를 연결하면 정답이 완성된다.  
}  
  
public static List<List<Integer>> insertAll(Integer first, List<List<Integer>> lists) {  
 List<List<Integer>> result = new ArrayList<>();  
 for (List<Integer> list : lists) {  
 List<Integer> copyList = new ArrayList<>();  
 copyList.add(first);  
 copyList.addAll(list);  
 result.add(copyList);  
 }  
 return result;  
}  
  
public static List<List<Integer>> concat(List<List<Integer>> a, List<List<Integer>> b) {  
 List<List<Integer>> r = new ArrayList<>(a);  
 r.addAll(b);  
 return r;  
}

입력으로 {1,4,9}를 제공하면 결과로 {{}, {9}, {4}, {4,9}, {1}, {1,9}, {1,4}, {1,4,9}}를 얻는다.

Subsets 자체도 기존의 구조를 갱신하지 않으므로 함수형이다.

Subans의 모든 요소를 포함하는 새로운 List를 만들었다는 점에 주목하라.

Concat도 인수정보를 변경하지 않게 해서 순수 함수다.

**13.3 재귀와 반복**

순수함수형 프로그래밍 언어에서는 while, for 같은 반복문을 포함하지 않는다. 왜그럴까?

🡺이러한 반복문 때문에 변화가 자연스럽게 코드에 스며들 수 있기 때문이다.

Iterator<Apple> it = apples.iterator();  
while(it.hasNext()) {  
 Apple apple = it.next();  
}

위 코드에서 호출자는 변화를 확인할 수 없으므로 아무 문제가 없다.

하지만 다음 코드처럼 for-each 루프를 사용하는 검색 알고리즘이기 떄문에 문제가 될 수 있다.

public void searchForGold(List<String> l, Stats stats) {  
 for(String s : l) {  
 if("gold".equals(s)) {  
 stats.incrementFor("gold"); // 인자값 정보를 변경 시킨다.  
 }  
 }  
}

루프 내부에서 프로그램의 다른 부분과 공유되는 stats 객체의 상태를 변화 시킨다.

재귀를 이용하면 루프 단계마다 갱신되는 반복 변수를 제거할 수 있다.

// 반복 방식 팩토리얼  
public static int factorialIterative(int n) {  
 int r = 1;  
 for (int i = 0; i <= n; i++) {  
 r \*= i;  
 }  
 return r;  
}  
  
// 재귀 방식의 팩토리얼  
public static long factorialRecursive(long n) {  
 return n == 1 ? 1 : n \* *factorialRecursive*(n - 1);  
}

첫번째 예제는 일반적인 루프를 사용한 코드로 매 반복마다 변수 r과 i가 갱신된다.

두번째 예제는 재귀 방식의 코드로 좀 더 수학적인 형식으로 문제를 해결한다.

자바8 스트림으로 좀 더 단순하게 팩토리얼을 정의할 수 있다.

// 스트림 팩토리얼  
public static long factorialStreams(long n) {  
 return LongStream.*rangeClosed*(1, n)  
 .reduce(1, (long a, long b) -> a \* b);  
}

일반적으로 반복 코드 보다 재귀 코드가 더 비싸다. 함수가 호출될 때마다 호출 스택에 각 호출시 생성되는 정보를 저장할 새로운 스택 프레임이 만들어 진다.

(즉 메모리 사용량이 증가한다.)

🡺입력값이 크면 StackOverflowError가 발생한다.

🡺함수형 언어는 꼬리 호출 최적화 라는 해결책을 제공한다.

// 꼬리재귀 형태  
public static long factorialTailRecursive(long n) {  
 return *factorialHelper*(1, n);  
}  
  
public static long factorialHelper(long acc, long n) {  
 return n == 1 ? acc : *factorialHelper*(acc \* n, n - 1);  
}

factorialHelper에서 재귀 호출이 가장 마지막에 이루어지므로 꼬리 재귀다.

반면 이전의 factorialRecursive에서 마지막으로 수행한 연산은 n과 재귀 호출의 결과값의 곱셈이다.

중간 결과를 각각의 스택 프레임으로 저장해야 하는 일반 재귀와 달리 꼬리 재귀에서는 컴파일러가 하나의 스택 프레임을 재활용할 가능성이 생긴다.

안타깝게도 자바는 이와 같은 최적화를 제공하지 않는다. 그럼에도 여전히 고전적인 재귀보다는 여러 컴파일러 최적화 여지를 남겨둘 수 있는 꼬리 재귀를 적용하는 것이 좋다.

**14 함수형 프로그래밍 기법**

좀 더 고급적인 함수형 프로그래밍 기법을 소개한다.

고차원 함수, 커링, 영구 자료구조, 게으른 리스트, 패턴 매칭, 참조 투명성을 이용한 캐싱,

콤비네이터 등을 살펴 본다,

**14.1 함수는 모든 곳에 존재한다**

-함수형 프로그래밍이란 함수나 메서드가 수학의 수학처럼 동작함을, 즉 부작용 없이 동작함을 의미했다.

-일반값 처럼 취급할 수 있는 함수를 일급 함수라고 한다.

-자바8의 가장 큰 특징은 이 일급 함수를 지원한다는 점이다.

**14.1.1 고차원 함수**

Comparator<Apple> c = comparing(Apple::getWeight);

-Comparator.comparing 처럼 다음 중 하나 이상의 동작을 수행하는 함수를 고차원 함수라 부른다.

🡺하나 이상의 함수를 인수로 받음

🡺함수를 결과로 반환

**부작용과 고차원 함수**

스트림 연산으로 전달하는 함수는 부작용이 없어야 하며, 부작용을 포함하는 함수를 사용하면 문제가 발생한다는 사실을 설명했다.

고차원 함수나 메서드를 구현할 때 어떤 인수가 전달될지 알 수 없으므로 인수가 부작용을 포함할 가능성을 염두에 두어야 한다.

따라서 인수로 전닫ㄹ된 함수가 어떤 부작용을 포함하게 될지 정확하게 문서화하는 것이 좋다.

**14.1.2 커링**

기존 로직을 활용해서 변환기를 특정 상황에 적용할 수 있는 방법이 있다.

다음은 커링이라는 개념을 활용해서 한 개의 인수를 갖는 변환 함수를 생산하는 팩토리를 정의하는 코드다.

// 일 반

static double convert(double x, double f, double b) {  
 return x \* f + b;  
}

// 커링 적용

static DoubleUnaryOperator curriedConverter(double f, double b) {  
 return (double x) -> x \* f + b;  
}

위 메서드에 변환 요소(f)와 기준치(b)만 넘겨주면 우리가 원하는 작업을 수행할 함수가 반환된다.

DoubleUnaryOperator convertCtoF = *curriedConverter*(9.0 / 5, 32);  
DoubleUnaryOperator convertUSDtoGBP = *curriedConverter*(0.6, 0);  
DoubleUnaryOperator convertKmtoMi = *curriedConverter*(0.6214, 0);

다음처럼 변환기를 사용할 수 있다.

double gbp = convertUSDtoGBP.applyAsDouble(1000);

🡺결과적으로 기존의 변환 로직을 재활용하는 유연한 코드를 얻었다!

**커링의 이론적 정의**

커링은 x와 y라는 두 인수를 받는 함수 f를 한 개의 인수를 받는 g라는 함수로 대체하는 기법이다.

**14.2 영속 자료구조**

-함수형 프로그램에서는 함수형 자료구조, 불변 자료구조 등의 용어도 사용하지만 보통은 영속 자료구조라고 부른다.

🡺여기서 영속(persistent)라는 단어는 데이터베이스에서 프로그램 종료 후에도 남아있음을 의미하는 영속과는 다른 의미

-함수형 메서드에서는 전역 자료구조나 인수로 전달된 구조를 갱신할 수 없다.

🡺자료 구조를 바꾼다면 참조 투명성에 위배되고 인수를 결과로 단순하게 매핑할 수 있는 능력이 상실되기 때문이다.

**14.2.1 파괴적인 갱신과 함수형**

-TrainJourney는 간단한 단방향 연결 리스트로 구현되며 여행 구간의 가격 등 상세 정보를 포함하는 int 필드를 포함한다.

-직통열차나 여정의 마지막 구간에서는 onward가 null이 된다.

public class TrainJourney {  
 public int price;  
 public TrainJourney onward;  
  
 public TrainJourney(int p, TrainJourney t) {  
 price = p;  
 onward = t;  
 }

}

-이때 X에서 Y까지 그리고 Y에서 Z까지 여행을 나타내는 별도의 TrainJourney 객체가 있다.

두 객체를 연결해서 하나의 여행을 만들 수 있을 것이다. (X ->Y, Y->X)

기존의 단순한 명령형 이라면 다음처럼 기차여행을 연결할 것이다.

// 명령형  
public TrainJourney link(TrainJourney a, TrainJourney b) {  
 if(a == null)  
 return b;  
  
 TrainJourney t = a;  
 while(t.onward != null) {  
 t = t.onward;  
 }  
 t.onward = b;  
 return a;  
}

a의 리스트 끝부분을 가리키는 null을 리스트 b로 대체한다. 여기서 문제가 발생한다.

-firstJourney라는 변수는 X에서 Y로의 경로를 포함하고 있다.

-secondJourney라는 변수는 Y에서 Z로의 경로를 포함한다.

link(firstJourney, secondJourney);

를 호출하면 firstJourney가 secondJourney를 포함하면서 파괴적인 갱신(즉, firstJourney를 변경)이 일어난다.

🡺결과 적으로 X에서 Y로의 여정이 아닌 X에서 Z로의 여정을 의미하게 된다.

🡺이렇게 되면 firstJourney에 의존하는 코드가 동작하지 않게 된다.

함수형에서는 이 같은 부작용을 수반하는 메서드를 제한하는 방식으로 문제를 해결한다.

🡺계산 결과를 표현할 자료구조가 필요하면 기존의 자료구조를 갱신하지 않도록 새로운 자료구조를 만들어야 한다.

다음처럼 깔끔한 함수형 해결 방법을 사용하는 것이 좋다.

// 함수형  
TrainJourney append(TrainJourney a, TrainJourney b) {  
 return a == null ? b : new TrainJourney(a.price, append(a.onward, b));  
}

🡺이 코드는 명확하게 함수형이며 기존 자료구조를 변경하지 않는다.

**14.2.2 트리를 사용한 다른 예제**

HashMap 같은 인터페이스를 구현할 때는 이진 탐색 트리가 사용된다. Tree는 문자열 키와 int값을 포함한다.

예를 들어 이름 키와 나이 정보값을 포함핳ㄹ 수 있다.

class Tree {  
 protected String key;  
 protected int val;  
 protected Tree left, right;  
  
 public Tree(String k, int v, Tree l, Tree r) {  
 key = k;  
 val = v;  
 left = l;  
 right = r;  
 }  
}

class TreeProcessor {  
 public static int lookup(String k, int defaultval, Tree t) {  
 if (t == null)  
 return defaultval;  
  
 if (k.equals(t.key))  
 return t.val;  
  
 return *lookup*(k, defaultval, k.compareTo(t.key) < 0 ? t.left : t.right);  
 }

}

🡺이제 이진 탐색 트리를 이용해서 ㅁ누자열값으로 int를 얻을 수 있다.

public static void update(String k, int newval, Tree t) {  
 if (t == null) {  
 // 새로운 노드를 추가해야 함  
 } else if (k.equals(t.key)) {  
 t.val = newval;  
 } else {  
 *update*(k, newval, k.compareTo(t.key) < 0 ? t.left : t.right);  
 }  
}

🡺새로운 노드를 추가할 때는 주의해야 한다. 가장 쉬운 방법은 update 메서드가 탐색한 트리를 그대로 반환하는 것이다. (하지만 이 방법은 깔끔하지 못하다. 사용자는 update가 즉석에서 트리를 갱신할 수 있으며, 전달한 트리가 그대로 반환된다는 사실, 원래 트리가 비어있으면 새로운 노드가 반환될 수 있다는 사실을 모두 기억해야 하기 때문이다.)

public static Tree update2(String k, int newval, Tree t) {  
 if (t == null) {  
 t = new Tree(k, newval, null, null);  
 } else if (k.equals(t.key)) {  
 t.val = newval;  
 } else if (k.compareTo(t.key) < 0) {  
 t.left = *update2*(k, newval, t.left);  
 } else {  
 t.right = *update2*(k, newval, t.right);  
 }  
 return t;  
}

🡺위 두가지 update 버전 모두 기존 트리를 변경한다. 즉, 트리에 저장된 맵의 모든 사용자가 변경에 영향을 받는다.

**14.2.3 함수형 접근법 사용**

새로운 키/값 쌍을 저장할 새로운 노드를 만들어야 한다.

또한 트리의 루트에서 새로 생성한 노드의 경로에 있는 노드들도 새로 만들어야 한다.

public static Tree fupdate(String k, int newval, Tree t) {  
 return (t == null) ?  
 new Tree(k, newval, null, null) :  
 k.equals(t.key) ?  
 new Tree(k, newval, t.left, t.right) :  
 k.compareTo(t.key) < 0 ?  
 new Tree(t.key, t.val, *fupdate*(k, newval, t.left), t.right) :  
 new Tree(t.key, t.val, t.left, *fupdate*(k, newval, t.right));  
}

🡺위 코드에서는 if-then-else 대신 하나의 조건문을 사용했는데 이렇게 해서 위 코드가 부작용이 없는 하나의 표현식임을 강조했따.

**update와 fupdate의 차이는 뭘까?**

update 메서드는 모든 사용자가 같은 자료구조를 공유하며 프로그램에서 누군가 자료구조를 갱신했을 때 영향을 받는다. 따라서 값을 추가 할때마다 값을 복사했다.

반면 fupdate는 순수한 함수형이다. fupdate에서는 새로운 tree를 만든다. 하지만 인수를 이용해서 가능한 한 많은 정보를 공유한다.

이와 같은 함수형 자료고즈롤 영속(저장된 값이 다른 누군가에 의해 영향을 받지 않는 상태)이라고 하며 따라서 프로그래머는 fupdate가 인수로 전달된 자료구조를 변화시키지 않을 것이라는 사실을 확신할 수 있다.

Tree 클래스의 key, val, left, right 필드를 final로 선언함으로써 ‘기존 구조를 변화시키지 않는다’는 규칙을 강제할 수 있다. 하지만 final은 필드에만 적용되며 객체에는 적용되지 않으므로 각 객체의 필드에 final을 적절하게 사용해야 한다.

**일부 사용자만 볼 수 있게 트리를 갱신하면서도 다른 일부 사용자는 이를 알아차릴 수 없게 하고 싶다**

1) 하나는 고전적인 자바 해법이다. (어떤 값을 갱신할 때 ㅁ너저 복사해야 하는지 주의 깊게 확인)

2) 다른 하나는 함수형 해법이다.(즉, 갱신을 수행할 때마다 논리적으로 새로운 자료구조를 만든 다음에 사용자에게 적절한 버전의 자료구조를 전달할 수 있다.

**14.3 스트림과 게으른 평가**

스트림은 데이터 컬렉션을 처리하는 편리한 도구임을 살펴 보았다.

🡺스트림은 단 한번만 소비할 수 있다는 제약이 있어서 스트림은 재귀적으로 정의할 수 없다. (불변)

**14.3.1 자기 정의 스트림**

6장에서 다음 코드처럼 소수 스트림을 계산할 수 있었다.

public static Stream<Integer> primes(int n) {  
 return Stream.*iterate*(2, i -> i + 1)  
 .filter(MymathUtils::isPrime)  
 .limit(n);  
}  
  
public static boolean isPrime(int candidate) {  
 int candidateRoot = (int) Math.*sqrt*((double) candidate);  
 return IntStream.*rangeClosed*(2, candidateRoot)  
 .noneMatch(i -> candidate % i == 0);  
}

그다지 멋지게 해결하는 코드는 아니다. 우선 후보 수로 정확히 나누어떨어지는지 매번 모든 수를 반복 확인했다.

다음은 소수로 나눌 수 있는 수를 제외하는 과정을 설명한다.

1)소수를 선택할 숫자 스트림이 필요하다.

2)스트림에서 첫 번째 수(스트림의 머리)를 가져온다. 이 숫자는 소수다.

3)이제 스트림의 꼬리에서 가져온 수로 나누어떨어지는 모든 수를 걸러 제외시킨다.

4)이렇게 남은 숫자만 포함하는 새로운 스트림에서 소수를 찾는다. 이제 1번부터 다시 이 과정을 반복하게 된다. 따라서 이 알고리즘은 재귀다.

🡺 이 알고리즘은 여러가지 면에서 부족한 알고리즘이다.

스트림 APi로 이 알고리즘을 구현해 보자

**단계 1: 스트림 숫자 얻기**

IntStream.iterate 메서드를 이용하면 2에서 시작하는 무한 숫자 스트림을 생성할 수 있다.

public static IntStream numbers() {  
 return IntStream.*iterate*(2, n -> n + 1);  
}

**단계 2: 머리 획득**

IntStream은 첫 번째 요소를 반환하는 findFirst라는 메서드를 제공한다.

public static int head(IntStream numbers) {  
 return numbers.findFirst().getAsInt();  
}

**단계 3: 꼬리 필터링**

스트림의 꼬리를 얻는 메서드를 정의한다.

public static IntStream tail(IntStream numbers) {  
 return numbers.skip(1);  
}

다음처럼 획득한 머리로 숫자를 필터링할 수 있다.

IntStream numbers = *numbers*();  
int head = *head*(numbers);  
IntStream filtered = *tail*(numbers).filter(n -> n % head != 0);

**단계 4: 재귀적으로 소수 스트림 생성**

반복적으로 머리를 얻어서 스트림을 필터링하려 할 수 있다.

public static IntStream primes(IntStream numbers) {  
 int head = *head*(numbers);  
 return IntStream.*concat*(  
 IntStream.*of*(head),  
 *primes*(*tail*(numbers).filter(n -> n % head != 0))  
 );  
}

**나쁜소식**

안타깝게도 단계 4의 코드를 실행하면 다음과 같은 에러가 발생한다.

Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed

🡺스트림을 머리 꼬리를 분리하는 두 개의 최종 연산findFirst와 skip을 사용했다. (스트림이 완전 소비된다.)

**게으른 평가**

위 나쁜 소식보다 더 심각함 누제가 있다. 정적 메서드 IntStream.concat은 두 개의 스트림 인스턴스를 인수로 받는다. 두 번째 인수가 primes를 직접 재귀적으로 호출하면서 무한 재귀에 빠진다.

‘재귀적 정의 허용하지 않음’ 같은 자바8의 스트림 규칙은 우리에게 아무 해도 끼치 않으며 오히려 이 규칙 덕분에 데이터베이스 같은 질의를 표현하고 병렬화할 수 있는 능력을 얻을 수 있다.

결론적으로 concat의 두 번째 인수에서 primes를 게으르게 평가하는 방식으로 문제를 해결할 수 있다.

**14.3.2 게으른 리스트 만들기**

자바8의 스트림은 요청할 때만 값을 생성하는 블랙박스와 같다. 스트림에 일련의 연산을 적용하면 연산이 수행되지 않고 일단 저장된다. 스트림에 최종 연산을 적용해서 실제 계산을 해야 하는 상황에서만 실제 연산이 이루어진다.

**기본적인 연결 리스트**

다음 코드처럼 MyLinkedList라는 단순한 ㅇ녀결 리스트 형태의 클래스를 정의할 수 있다는 것을 기억하라.

interface MyList<T> {  
 T head();  
  
 MyList<T> tail();  
  
 default boolean isEmpty() {  
 return true;  
 }  
}  
  
public class MyLinkedList<T> implements MyList<T> {  
 private final T head;  
 private final MyList<T> tail;  
  
 public MyLinkedList(T head, MyList<T> tail) {  
 this.head = head;  
 this.tail = tail;  
 }  
  
 @Override  
 public T head() {  
 return head;  
 }  
  
 @Override  
 public MyList<T> tail() {  
 return tail;  
 }  
  
 public boolean isEmpty() {  
 return false;  
 }  
}  
  
class Empty<T> implements MyList<T> {  
  
 @Override  
 public T head() {  
 throw new UnsupportedOperationException();  
 }  
  
 @Override  
 public MyList<T> tail() {  
 throw new UnsupportedOperationException();  
 }  
}

**기본적인 게으른 리스트**

Supplier<T>를 이용해서 게으른 리스트를 만들면 꼬리가 모두 메모리에 존재하지 않게 할 수 있다.

다음은 게으른 리스트를 만드는 코드다.

public class LazyList<T> implements MyList<T> {  
 final T head;  
 final Supplier<MyList<T>> tail;  
  
 public LazyList(T head, Supplier<MyList<T>> tail) {  
 this.head = head;  
 this.tail = tail;  
 }  
  
 @Override  
 public T head() {  
 return head;  
 }  
  
 @Override  
 public MyList<T> tail() {  
 return tail.get();  
 }  
  
 public boolean isEmpty() {  
 return false;  
 }  
}

Supplier의 get 메서드를 호출하면 LazyList의 노드가 만들어진다.

LazyList의 생성자에 tail 인수로 Supplier를 전달하는 방식으로 n으로 시작하는 무한히 게으른 리스트를 만들 수 있다.

public static LazyList<Integer> from(int n) {  
 return new LazyList<Integer>(n, () -> *from*(n + 1));  
}

실제로 숫자는 요청햇을 때 만들어 진다.

만약 요청했을 때 코드가 실행되는 것이 아니라 2부터 시작해서 모든 수를 미리 계산하려 한다면 프로그램은 영원히 종료되지 않을 것이다.

LazyList<IntStream> numbers = *from*(2);  
int two = numbers.head();  
int three = numbers.tail().head();  
int four = numbers.tail().tail().head();

**소수 생성으로 돌아와서**

다음 코드처럼 기존에 스트림 API를 사용했던 코드에 새로운 LazyList를 적용하려 할 것이다.

public static MyList<Integer> primes(MyList<Integer> numbers) {  
 return new LazyList<>(  
 numbers.head(),  
 () -> *primes*(  
 numbers.tail()  
 .filter(n -> n % numbers.head() != 0) // 필터 없음  
 )  
 )  
}

**게으른 필터 구현**

안타깝게도 LazyList는 filter 메서드를 정의하지 않으므로 위 코드는 컴파일 에러가 발생한다.

다음과 같이 정의하자

public MyList<T> filter(Predicate<T> p) {  
 return isEmpty() ? this : p.test(head()) ? new LazyList<>(head(), () -> tail().filter(p)) : tail().filter(p);  
}

tail과 head 호출을 연결해서 처음 세 개의 소수를 계산할 수 있다.

LazyList<Integer> numbers = *from*(2);  
int two = primes(numbers).head();  
int three = primes(numbers).tail().head();  
int four = primes(numbers).tail().tail().head();

**드디어 완성!**

지금까지 게으른 리스트와 함수를 만들었고 모든 소수를 포함하는 자료구조를 정의했다. 그런데 지금가지 만든 자료구조가 어떤 유용성을 제공하는 걸까?

🡺함수를 자료구조 내부로 추가할 수 있다는 사실을 알았다.

🡺이런 함수는 자료구조를 만드는 시점이 아니라 요청 시점에 실행된다는 사실도 확인했다.

게으른 자료구조는 강력한 프로그래밍 도구라는 사실을 기억하자.

**14.4 패턴 매칭**

일반적으로 함수형 프로그래밍을 구분하는 또 하나의 중요한 특징으로 패턴매칭을 들 수 있다.

**정규표현식**

문자열을 처리하는 방법중의 하나로 특정한 조건의 문자를 검색하거나 치환하는 과정을 매우 간편하게 할 수 있도록 하는 수단

**14.4.1 방문자 디자인 패턴**

자바에서는 방문자 디자인 패턴으로 자료형을 언랩할 수 있다. 특히 특정 데이터 형식을 방문 하는 알고리즘을 캡슐화하는 클래스를 따로 만들 수 있다.

지정된 데이터 형식의 인스턴스를 입력으로 받는다. 그리고 인스턴스의 모든 멤버에 접근한다.

class BinOp extends Expr {  
 public Expr accept(SimplifyExprVisitor v) {  
 return v.visit(this);  
 }  
}  
  
class SimplifyExprVisitor {  
 public Expr visit(BinOp e) {  
 if("+".equals(e.opname) && e.right instanceof Number && ...) {  
 return e.left;  
 }  
 return e;  
 }  
}

**14.4.2 패턴 매칭의 힘**

패턴 매칭이라는 좀 더 단순한 해결 방법도 있다. 자바는 패턴 매칭을 지원하지 않으므로 스칼라 프로그래밍 언어로 패턴 매칭이 뭔지 보여주려 한다.

def simplifyExpression(expr: Expr): Expr = expr match {  
 case BinOp("+", e, Number(0)) => e // 0 더하기  
 case BinOp("\*", e, Number(1)) => e // 1 곱하기  
 case BinOp("/", e, Number(1)) => e // 1 나누기  
 case \_ => expr // expr을 단수화할 수 없다.  
}

트리와 비슷한 자료구조를 다룰 때 이와 같은 패턴 매칭을 사용하면 매우 간결하고 명확한 코드를 구현할 수 있다.

스칼라의 와일드 카드 case는 자바의 default:같은 역할을 한다.

**switch와 다른 점**

-match도 expression이다. 즉, 값을 반환한다.

-명시적으로 break; 를 해야 하는 switch와 달리 match를 찾은 이후엔 계속 matching을 시도하지 않는다.

-어떤 패턴에도 매칭되지 않으면 MatchError 예외를 던진다. 예외를 피하려면 case \_ => 를 명시하는 방법을 써야 한다. (자바는 default)

**14.5 기타 정보**

**14.5.1 캐싱 또는 기억화**

트리 형식의 토포로지를 갖는 네트워크 범위 내에 존재하는 노드의 수를 계산하는 computeNumberOfNodes 라는 부작용 없는 메서드가 있다고 가정하자.

다힝해 불변이지만 재귀적으로 탐색해야 하므로 노드 계산 비용이 비싸다.

🡺표준적인 해결책으로 기억화 라는 기법이 있다. 기억화는 메서드에 래퍼로 캐시(HashMap)을 추가하는 기법이다.

1)래퍼가 호출되면 인수, 결과 쌍이 캐시에 존재하는지 먼저 확인한다.

2)캐시에 값이 존재하면 캐시에 저장된 값을 즉시 반환한다.

3)캐시에 값이 존재하지 않으면 computeNumberOfNodes를 호출해서 결과를 계산한 다음에 새로운 인수, 결과 쌍을 캐시에 저장하고 결과를 반환한다.

다음은 캐싱을 사용하는 예제 코드다.

public class Caching {  
 final Map<NumericShaper.Range, Integer> numberOfNodes = new HashMap<>();  
  
 Integer computeNumberOfNodesUsingCache(NumericShaper.Range range) {  
 Integer result = numberOfNodes.get(range);  
 if (result != null) {  
 return result;  
 }  
 result = computeNumberOfNodesUsingCache(range);  
 numberOfNodes.put(range, result);  
 return result;  
 }  
}

computeNumberOfNodesUsingCache는 참조 투명성을 갖는다. 하지만 numberOfNodes는 공유된 가변 상태며 HashMap은 동기화 되지 않았으므로 스레드 안정성이 없는 코드이다.

ConcurrentHashMap을 사용할 수 있지만 다중 코어에서 numberOfNodes를 동시에 호출하면 성능이 크게 저하될 수 있다.

🡺가장 좋은 것은 동ㅅ이성과 가변 상태가 만나는 상황을 완전히 없애는 것이다.

**14.5.2 같은 객체를 반환함은 무엇을 의미하는가?**

참조 투명성이냐 아니냐를 토론하자면 얘기가 아주 길어진다. 일반적으로 함수형 프로그래밍에서는 데이터가 변겨오디지 않으므로 같다는 의미는 == (레퍼런스가 같음)이 아니라 구조적인 값이 같다는 것을 의미한다.

**14.5.3 콤비네이터**

함수형 프로그래밍에서는 두 함수를 인수로 받아 다른 함수를 반환하는 등 함수를 조합하는 고차원 함수를 많이 사용하게 된다. 이처럼 함수를 조합하는 기능을 콤비네이터라고 부른다.

**CHAPTER 15**

**스칼라 패턴 매칭**

def example(a: Int) {

a match {

case 1 => print("One")

case 2 => print("Two")

case \_ => print("Other")

}

}

match를 이용하며 match는 Any타입에서 동작가능.

match는 Any타입에서 동작하지만 match왼쪽에 있는 타겟객체의 타입으로 매칭타입이 제한됨. case문에 매치되는 것이 없으면 MatchError 예외가 발생하며 5번줄의 case \_ => ... 문을 통해서 와일드카드 매칭을 할 수 있음.

여러가지 타입에 맞춰 패턴을 매칭할 수 있다.

object MatchTest2 extends App {  
 def matchTest(x: Any): Any = x match {  
 case 1 => "one"  
 case "two" => 2  
 case y: Int => "scala.Int"  
 }  
 println(matchTest("two"))  
}

**스칼라 Object (동반자 객체)**

static대신 object를 사용해서 싱글턴 객체로 처리할 수 있다.

class Dog(name: String) {  
 def bark = println("bark! bark!")  
 def getName = name  
}  
  
object Dog {  
 def apply(name: String) = new Dog(name)  
}

**스칼라 불변과 가변**

-Array는 항상 가변

-List 는 항상 불변

-셋과 맵은 가변과 불변 타입 모두 제공

scala.collection.Set // 기본 불변

scala.collection.immutable.Set // 불변과 가변 셋을 따로 지정 가능  
scala.collection.mutable.Set

**튜플**

-튜플을 사용하면 클래스를 정의하지 않고도 여러 아이템을 쉽게 한데 묶을 수 있다.

scala> val hostPort = ("localhost", 80)  
hostPort: (String, Int) = (localhost, 80)

-첫번째 원소는 0번이 아니고 1번이다

(역주: 1번부터 시작한 것은 함수언어의 전통에 따른 것이다).

-튜플은 패턴 매칭과 잘 들어맞는다.

scala> hostPort.\_1  
res0: String = localhost  
  
scala> hostPort.\_2  
res1: Int = 80

**메서드 시그니처(JLS) : 메소드 이름 + 파라미터**

**메서드 타입 (JLS) : 파라미터 + 반환값, 예외 타입**

**🡺(JVMS의 메서드 시그니처와 같다)**

<https://github.com/HomoEfficio/dev-tips/blob/master/%EC%9A%95%20%EB%82%98%EC%98%A4%EB%8A%94%20%EC%9E%90%EB%B0%94%20%EC%8A%A4%ED%8E%99%20-%20%EB%A9%94%EC%84%9C%EB%93%9C%20%EC%8B%9C%EA%B7%B8%EB%84%88%EC%B2%98.md>

<https://homoefficio.github.io/2017/02/19/Java8-%EB%9E%8C%EB%8B%A4-%EA%B4%80%EB%A0%A8-%EC%8A%A4%ED%8E%99-%EC%A0%95%EB%A6%AC/>

**메서드 타입(JLS) and 함수 디스크립터**

-메서드 타입 **: boolean test(T t)**

-함수 디스크립터 : **T -> boolean** 과 같은 형태로 표현할 수 있음

-함수 디스크립터를 통해 메서드 타입을 추론할 수 있음.