14장
함수형 프로그래밍

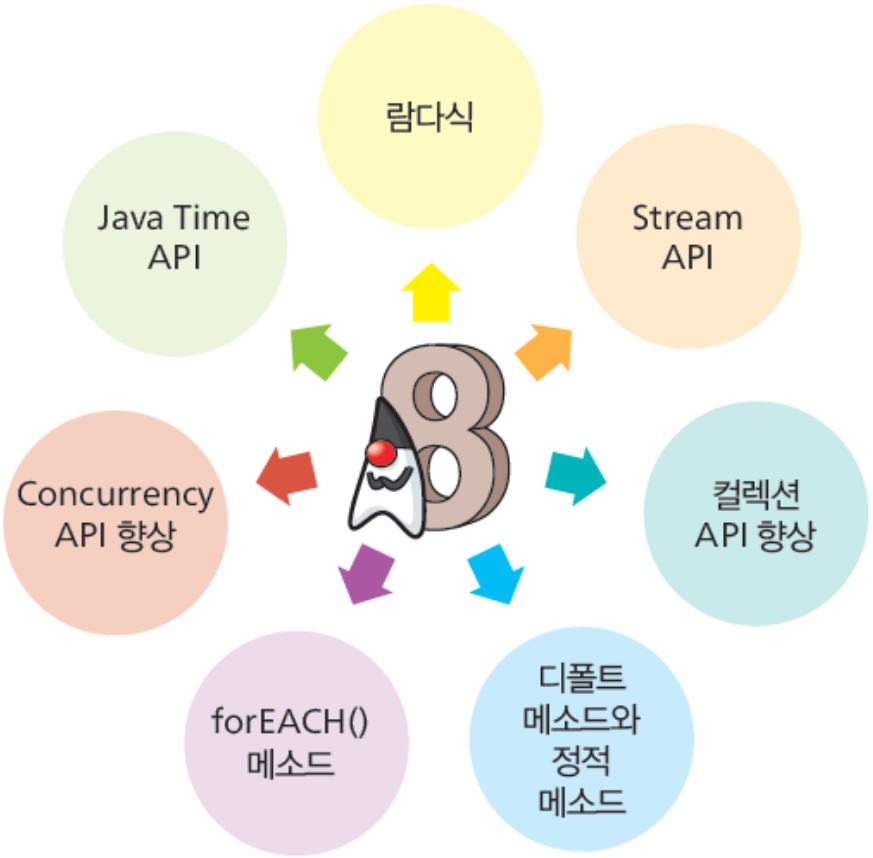
박숙영
blue@sookmyung.ac.kr

14장의 목표

1. 함수형 프로그래밍을 이해하고 사용할 수 있나요?
2. 스트림 API를 사용하여 컬렉션에서 특정 조건을 만족하는 데이터를 추려낼 수 있나요?
3. 원하는 동작을 포장하여서 메소드에 전달할 수 있나요?
4. 람다식으로 이름없는 메소드를 작성하고 전달할 수 있나요?

함수형 프로그래밍의 소개

- 함수형 프로그래밍의 지원은 Java 8부터 시작되었다.



함수형 프로그래밍의 소개

- 함수형 프로그래밍을 잘 사용하면 아주 쉽게 프로그램을 작성할 수 있다. 예를 들어, 문자열들을 저장하고 있는 리스트가 있고, 이것을 문자열의 길이에 따라서 정렬하려고 한다.

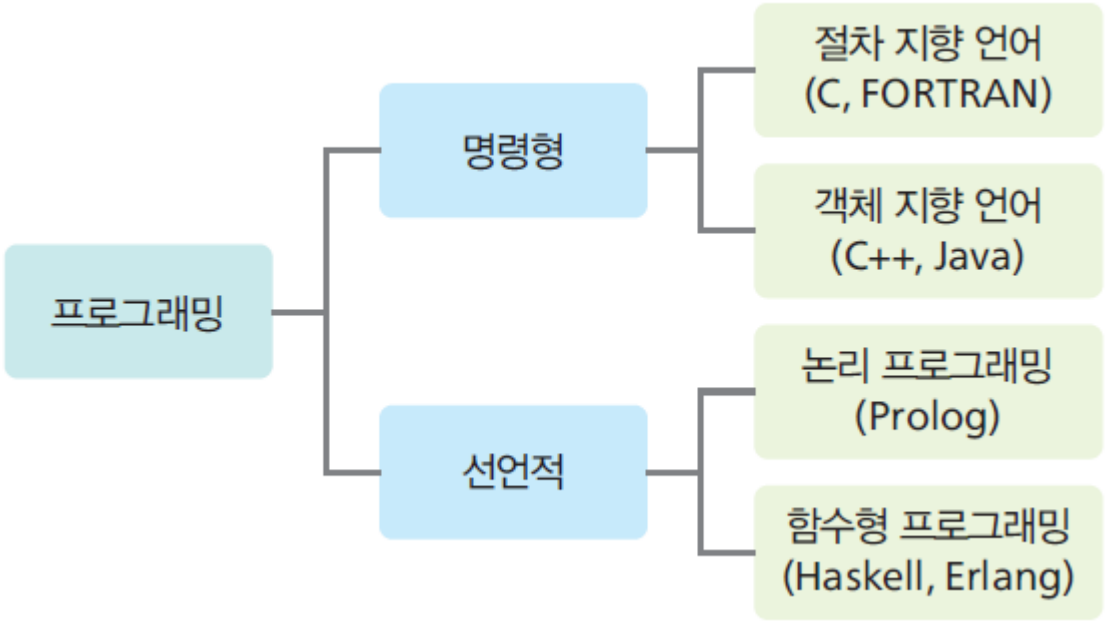
예전 스타일

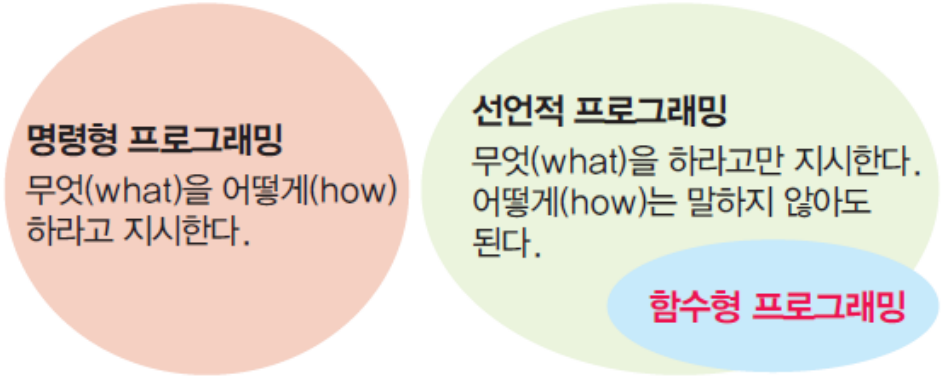
```
Collections.sort (mylist, new Comparator <String> () {
   public int compare (String s1, String s2) {
      return (s1.length()-s2.length());
   }
});
```

새로운 스타일

```
mylist.sort(comparing(String::length));
```

프로그래밍 패러다임 분류





명령형 프로그래밍 방법의 예

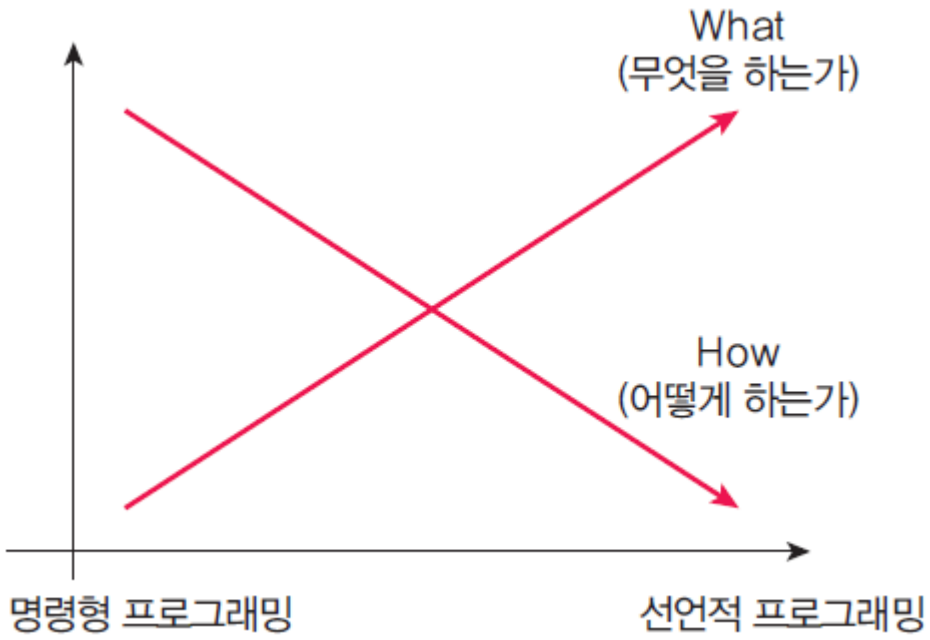
- 정수가 ArrayList에 저장되어 있다고 가정하자. ArrayList에서 짝수만 추려내고 싶다.

```
public class Imperative {
        public static void main(String args[]) {
                List<Integer> list = List.of(12, 3, 16, 2, 1, 9, 7, 20);
                List<Integer> even = new ArrayList<>();

                for(Integer e : list) {          // 짝수를 찾는다.
                        if(e%2 == 0 ) {
                                even.add(e);
                        }
                }
                for(Integer e : even) {          // 찾은 짝수를 출력한다.
                        System.out.println(e);
                }
        }
}
```

```
12
16
2
20
```

명령형 프로그래밍

- 명령형 프로그래밍은 작업을 어떻게(how) 수행하느냐를 중시한다. 즉 먼저 이것을 수행한 다음, 다음에 이것을 수행하라고 말해주는 프로그래밍이다.



함수형 프로그래밍

```
public class Test {
        public static void main(String args[]) {
                List<Integer> list = List.of(12, 3, 16, 2, 1, 9, 7, 20);
                list.stream()
                        .filter(e -> e % 2 == 0)
                        .forEach(System.out::println);
        }
}
```

```
12
16
2
20
```

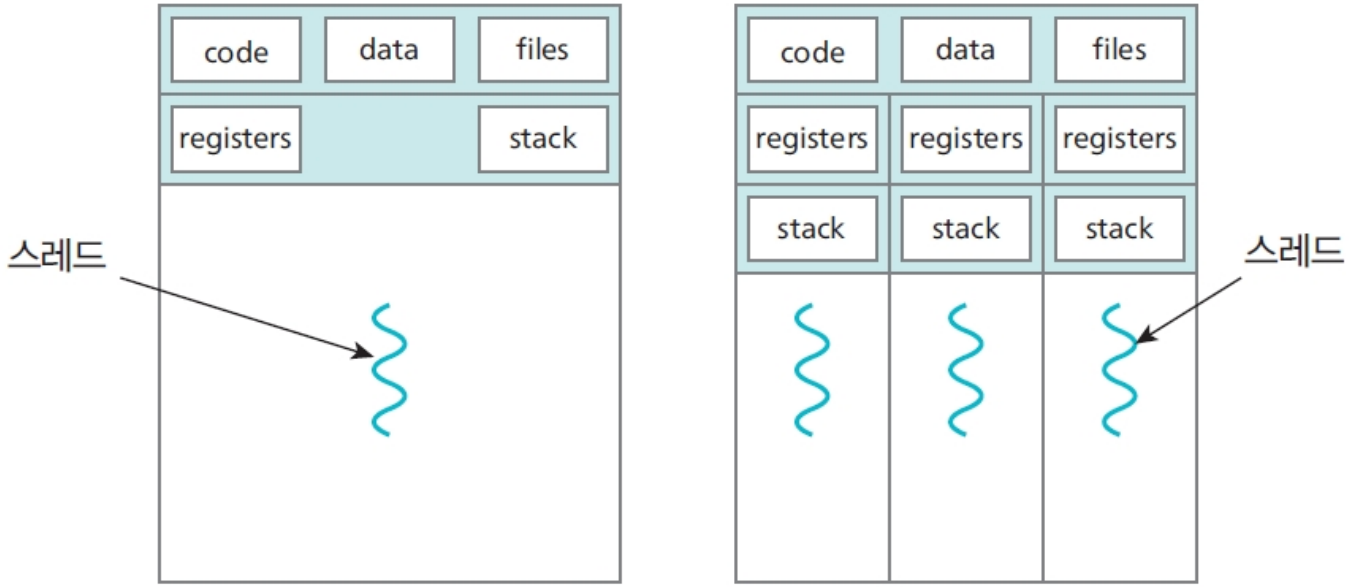
- stream() : 리스트 안의 원소들을 하나씩 추출하는 메소드
- filter(e->e%2==0): 들어오는 정수 중에서 짝수만을 추려내는 메소드

선언적 프로그래밍

- 선언적 프로그래밍은 해야 할 일(what)에 집중한다. 함수형 프로그래밍에서는 함수들이 계속 적용되면서 작업이 진행된다. 함수형 프로그램은 명령문이 아닌 수식이나 함수 호출로 이루어진다.
- 함수형 프로그래밍은 1930년대의 람다 수학에 근간을 두고 있다. 이들 함수들이 구현되는 세부적인 방법은 라이브러리가 담당한다. 이 방법의 가장 큰 장점은 함수 호출이 문제 설명처럼 읽히고, 그 이유 때문에 코드가 수행하는 작업을 이해하려고 할 때, 보다 명확하게 알 수 있다는 점이다.

멀티코어 시대의 함수형 프로그래밍

- 함수형 프로그래밍은 병렬 처리가 쉽다. 최근 CPU는 모두 멀티코어를 장착하고 있고 함수형 프로그래밍에서는 부작용 없는 순수 함수만을 사용하기 때문에 코어를 여러 개 사용하여도 서로 간에 복잡한 문제가 발생하지 않는다



함수란 무엇인가?

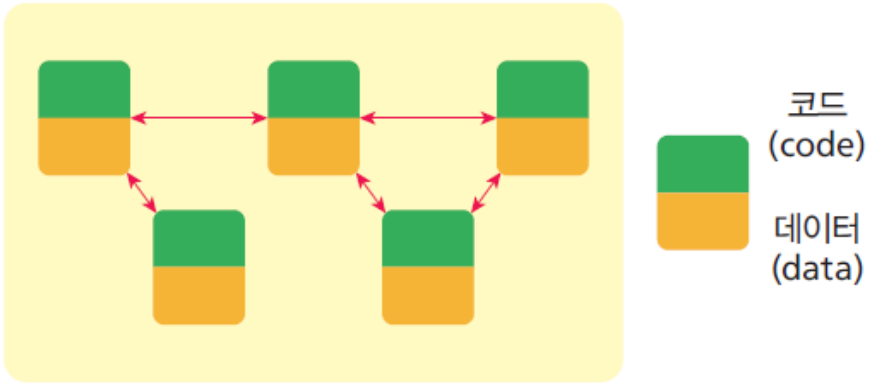
- 함수형 프로그래밍에서 함수는 순수 함수(pure function)라고 한다.
  - 순수 함수(pure function): 부작용이 없는 함수
  - 순수 함수는 스레드에 대하여 안전하고, 병렬적인 계산이 가능하다.

- 부작용(side effect)
  - 함수가 실행하면서 외부의 변수를 변경한다는 의미

객체 지향 프로그래밍과 함수형 프로그래밍

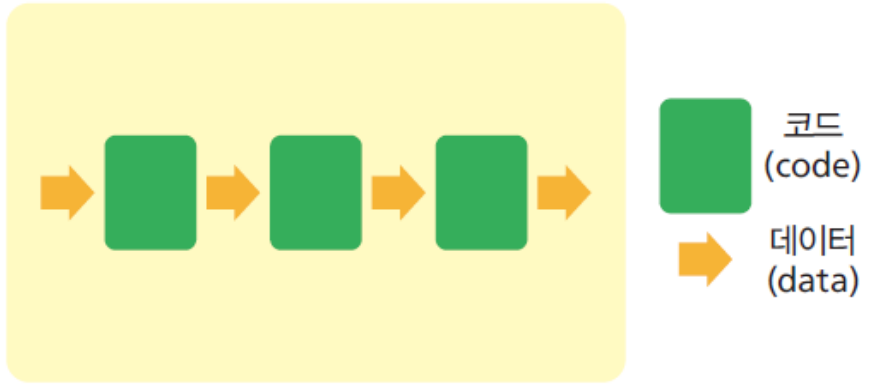
객체 지향 프로그래밍(OOP)	함수형 프로그래밍
OOP는 객체에 기반을 두고 있다.	함수 호출을 기본 프로그래밍 블록으로 강조한다.
OOP는 명령형 프로그래밍 모델에 따른다.	선언적 프로그래밍과 밀접하게 연결되어 있다.
병렬 처리를 지원하지 않는다.	병렬 처리를 지원한다.
기본적인 요소는 객체와 메소드이다.	기본적인 요소는 변수와 순수 함수이다.

객체지향 프로그래밍



객체지향 프로그래밍에서는 객체가
코드와 데이터를 캡슐화한다.

함수형 프로그래밍



함수형 프로그래밍에서는 함수들
사이로 데이터가 흘러가게 된다.

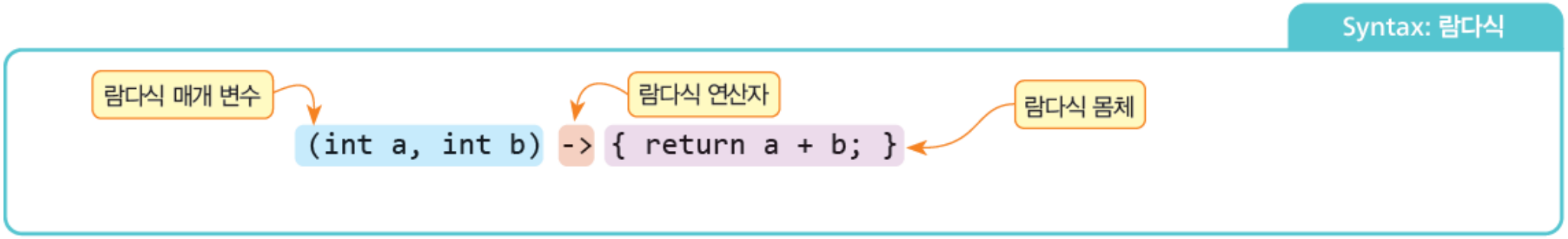
함수의 1급 시민 승격

- Java 8 이전에는 함수는 값(value)이 아니었다. 즉 함수를 변수에 저장할 수 없었고, 매개변수로 전달할 수도 없었다.

- Java 8에서는 함수가 1급 시민(*모든 연산이 허용된 엔티티*)으로 승격되었다. 즉 **함수**가 **값**이 된 것이다. 함수가 값이 되면 다음과 같은 일들이 가능해진다.
  - 함수도 변수에 저장할 수 있다.
  - 함수를 매개 변수로 받을 수 있다.
  - 함수를 반환할 수 있다.

람다식이란?

- 람다식(lambda expression)
  - 나중에 실행될 목적으로 다른 곳에 전달될 수 있는 코드 블록.



람다식

- 람다식은 0개 이상의 매개 변수를 가질 수 있다.
- 화살표 ->는 람다식에서 매개 변수와 몸체를 구분한다.
- 매개 변수의 형식을 명시적으로 선언할 수 있다. 또는 문맥에서 추정될 수 있다.
  - (int a)는 (a)와 동일하다.
  - 빈 괄호는 매개 변수가 없음을 나타낸다. 예를 들어 () -> 69와 같이 표현한다.
- 단일 매개 변수이고 타입이 유추가 가능한 경우에는 괄호를 사용할 필요가 없다. 예를 들어 a -> return a*a와 같이 표현한다.
- 몸체에 하나 이상의 문장이 있으면 중괄호 { }로 묶어야 한다.

람다식의 예

```
() -> System.out.println("Hello World");

(String s) -> { System.out.println(s); }

() -> 69

() -> { return 3.141592; };

(String s) -> s.length()

(Car c) -> c.getprice() > 150

(int x, int y) -> {
    System.out.print("결과값:");
    System.out.println(x + y);
}

(Car c1, Car c2) -> c1.getPrice().compareTo(c2.getPrice())
```

람다식의 활용

- 1. 자바에서 그래픽 사용자 인터페이스 코드를 작성할 때, 함수 몸체를 전달하고 싶은 경우

```
// 이전의 방법
button.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent e) {
      System.out.println("버튼 클릭!");
   }
});
```

```
// 람다식을 이용한 방법
button.addActionListener( (e) -> {
   System.out.println("버튼 클릭!");
});
```

람다식의 활용

- 2. 자바에서 스레드를 작성하려면 먼저 Runnable 인터페이스를 구현하는 클래스부터 작성하여 야 한다.

```
// 이전의 방법
new Thread(new Runnable() {
   @Override
   public void run() {
      System.out.println("스레드 실행");
   }
}).start();
```

```
// 람다식을 이용한 방법
new Thread( () -> System.out.println(
                    "스레드 실행 ") ).start();
```

- 3. 람다식을 사용하면 배열의 모든 요소를 출력하는 코드에서 forEach()와 같은 함수형 프로그래밍을 사용할 수 있다.

```
// 이전의 방법
List<Integer> list = Arrays.asList(
                                1, 2, 3, 4, 5);
for(Integer n: list) {
   System.out.println(n);
}
```

```
// 람다식을 이용한 방법
List<Integer> list = Arrays.asList(
                                1, 2, 3, 4, 5);
list.forEach(n -> System.out.println(n));
```

Lab: 람다식 활용

- Timer 클래스를 사용하여서 1초에 한 번씩 "beep"를 출력하는 프로그램을 람다식을 이용하여 작성해보자.

```
beep
beep
beep
...
```

```
class MyClass implements ActionListener {
        public void actionPerformed(ActionEvent event) {
                System.out.println("beep");
        }
}
public class CallbackTest {
        public static void main(String[] args) {
                ActionListener listener = new MyClass();
                Timer t = new Timer(1000, listener);
                t.start();
                for (int i = 0; i < 1000; i++) {
                        try {
                                Thread.sleep(1000);
                        }
                        catch (InterruptedException e) {
                        }
                }
        }
}
```

Sol: 람다식 활용

```
import javax.swing.Timer;

public class CallbackTest {
        public static void main(String[] args) {

                Timer t = new Timer(1000, event -> System.out.println("beep"));
                t.start();

                for (int i = 0; i < 1000; i++) {
                        try {
                                Thread.sleep(1000);
                        }
                        catch (InterruptedException e) {
                        }
                }
        }
}
```

동작 매개 변수화

- 함수형 프로그래밍에서 핵심적인 사항은 **함수를 다른 함수의 인수로 전달**하는 것이다. 함수(즉 코드가 들어 있는 블록)를 다른 함수로 전달하는 것이 왜 필요할까?

구체적인 예

- 구체적인 예로 살펴보자. 영업사원은 자동차 재고를 저장하고 검색할 수 있는 애플리케이션을 원하고 있다. 처음에 영업사원은 자동차 재고에서 흰색 자동차를 찾는 기능을 원한다고 하였다. 그러나 다음날 “자동차 가격이 5000만원 이하 자동차도 찾을 수 있죠?” 라고 말할 수 있다. 이틀 후 영업사원은 “색상이 흰색이고 5000만원 이하인 자동차도 찾을 수 있나요?”라고 물어볼 수도 있다. 개발자는 이러한 *변화하는 요구 사항에 부응하면서 최소한의 노력으로 구현 및 유지 관리가 간단한 방법*을 사용해야 한다.

동작 매개 변수화(behavior parameterization)

- 고객의 빈번한 요구 사항 변경을 처리할 수 있는 소프트웨어 개발 패턴이다. 이 방법에서는 사용자의 요구를 담은 코드 블록을 생성하고 이것을 프로그램의 다른 부분에 전달하는 것이다

자동차 영업 사원의 예

- 자동차 재고 리스트에서 특정한 자동차를 선택하는 문제를 여러 가지 방법으로 구현하면서 예전의 방법과 최신의 방법을 비교해보자.

```
private static Car[] carArray  = {
    new Car(1, "BENS SCLASS", "BLACK", 11000),
    new Car(2, "BNW 9", "BLUE", 8000),
    new Car(3, "KEA 9", "WHITE", 7000)
};

private static List<Car> carList = Arrays.asList(carArray);
```

첫 번째 버전: 매개 변수가 없음

- 우리는 filterWhiteCars() 메소드를 작성하려고 한다. 이 메소드는 흰색 자동차만을 추려서 리스트로 만들어서 반환하는 함수이다

```
public static List<Car> filterWhiteCars(List<Car> inventory) {

    List<Car> result = new ArrayList<>();
    for (Car car: inventory){
        if ("WHITE".equals(car.getColor()))
                result.add(car);
    }
    return result;
}
```

두 번째 버전: 값 매개 변수화

- 이때는 색상을 매개 변수화하고 메소드에 색상을 나타내는 매개 변수를 추가하면 좀 더 유연한 코드가 된다.

```
public static List<Car> filterCarByColor(List<Car> inventory,  String color) {
    List<Car> result = new ArrayList<>();
    for (Car car: inventory){
        if ( car.getColor().equals(color) )
            result.add(car);
    }
    return result;
}
```

```
public static List<Car> filterCars(List<Car> inventory, String color,  int price) {
    List<Car> result = new ArrayList<>();
    for (Car car: inventory) {
        if ( (car.getColor().equals(color))  | |  (car.getPrice() <= price) )
                                    result.add(Car);
    }
    return result;
}
```

세 번째 버전: 동작 매개 변수화

- 만약 우리가 원하는 동작을 함수로 전달하면 어떨까? 한 가지 가능한 방법은 자동차의 속성을 검사하여 true, false를 반환하는 함수를 작성하여 메소드로 전달하는 것이다

```
public interface CarPredicate{
    boolean test (Car car);
}
public static List<Car> filterCars(List<Car> inventory,  CarPredicate p) {
    List<Car> result = new ArrayList<>();
    for(Car car: inventory) {
        if(p.test(car))
            result.add(car);
    }
    return result;
}
```

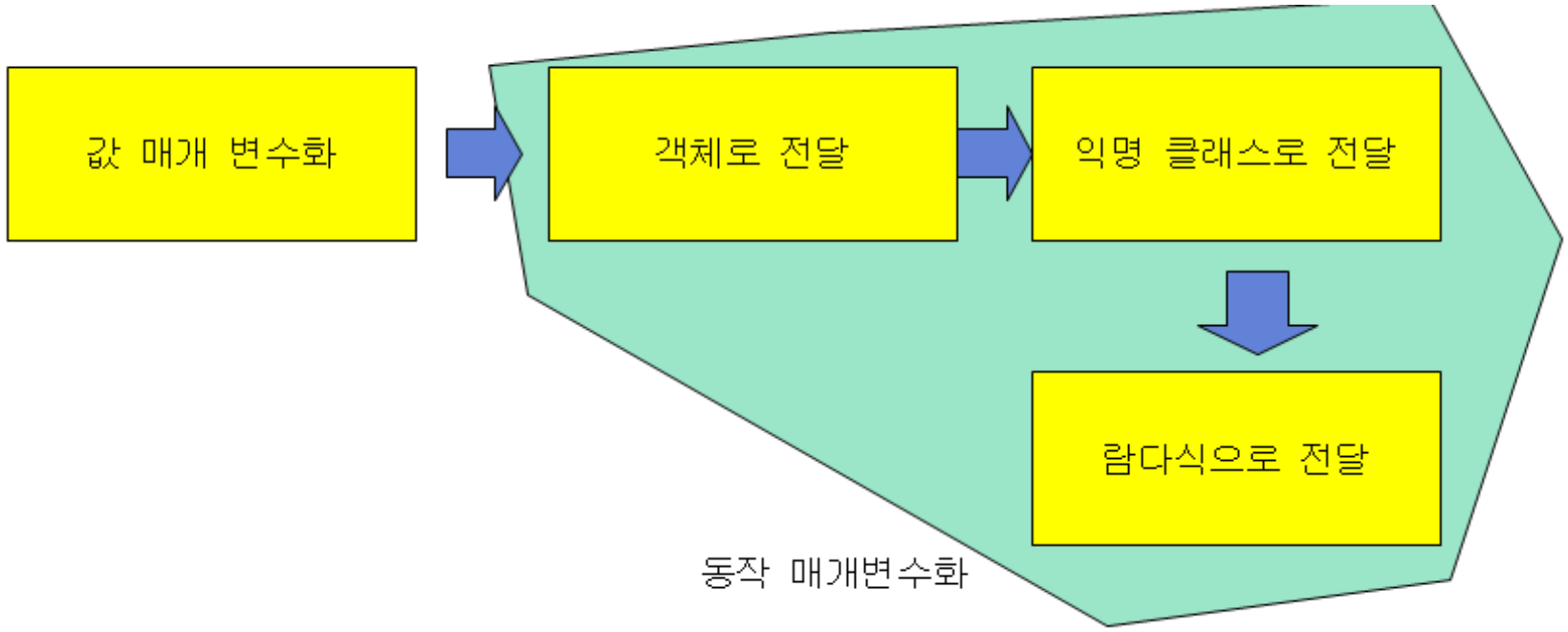
```
public class whiteCheapPredicate implements CarPredicate {
        public boolean test(Car car){
                return "WHITE".equals(car.getColor()) && car.getPrice() <= 5000;
        }
}
List<Car> whiteCheapCars = filterCars(carList, new whiteCheapPredicate());
```

네 번째 버전: 익명 클래스 사용

```
List<Car> whiteCars = filterCars(carList, new CarPredicate() {
    public boolean test(Car car){
        return "WHITE".equals(car.getColor());
    }
});
```

다섯 번째 버전: 람다식 사용

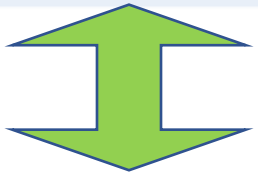
```
List<Car> whiteCars =  filterCars(carList, (Car car) -> "WHITE".equals(car.getColor()));
```

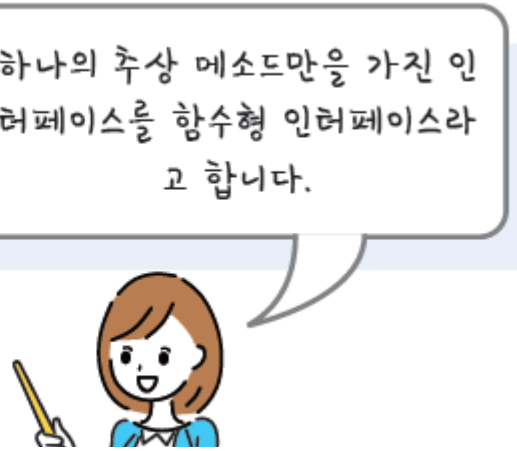


함수형 인터페이스

- 람다식과 함수형 인터페이스는 불가분의 관계에 있다. 컴파일러는 람다식을 어떻게 검사할 수 있을까? 람다식을 올바르게 컴파일하려면 반드시 함수형 인터페이스가 정의되어야 한다.

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```






```
Comparator<Car> byprice =
    (Car a1, Car a2) -> a1.getPrice()-a2.getPrice()  // 자동차의 가격은 정수라고 가정한다.
```

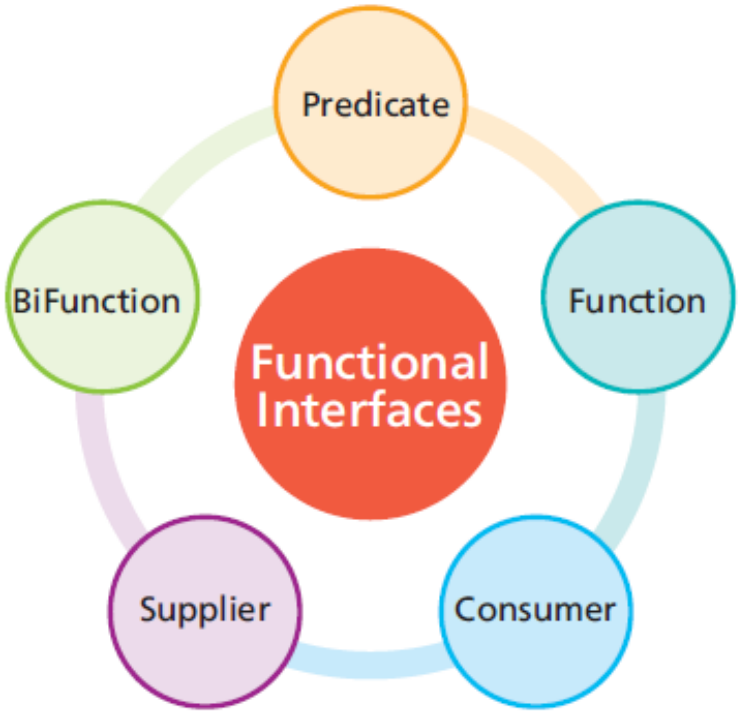
- 즉, 람다식을 사용하려면 누군가가 먼저 람다식을 위한 함수형 인터페이스를 정의해야 한다.

예제: 함수형 인터페이스와 람다식

```
@FunctionalInterface
interface MyMath
{
    int calculate(int x);
}

public class Test
{
    public static void main(String args[])     {
        int value = 9;

        MyMath s = (int x) -> x*x;
        int y = s.calculate(value);
        System.out.println(y);
    }
}
```

미리 만들어져 있는 함수형 인터페이스

- 자바에서는 많이 사용되는 함수형 인터페이스는 java.util.function 패키지로제공한다



함수형 인터페이스	반환형	추상 메소드 이름
Supplier<T>	T	get()
Consumer<T>	void	accept()
BiConsumer<T, U>	void	accept()
Predicate<T>	boolean	test()
BiPredicate<T, U>	boolean	test()
Function<T, R>	R	apply()
BiFunction<T, U, R>	R	apply()
UnaryOperator<T>	T	apply()
BinaryOperator<T>	T	apply()

Function 인터페이스

- Function<T, R> 인터페이스는 특정 객체를 받아서 특정 객체를 반환하는 추상 인터페이스이다. 추상 메소드 apply()는 T 타입의 객체를 입력으로 하고 R 타입의 객체를 반환한다. 다음과 같이 정의되어 있다.

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

```
import java.util.function.Function;

public class FunctionTest {
  public static void main(String[] args) {

    Function<Integer, Integer> f1 = i -> i*4;
    System.out.println(f1.apply(3));

    Function<String, Integer> f2 = s -> s.length();
    System.out.println(f2.apply("Hello"));
  }
}
```

Predicate 인터페이스

- Predicate 인터페이스는 단일 값을 매개 변수로 사용하고 true 또는 false를 반환하는 간단한 함수를 나타낸다. 다음과 같이 사용할 수 있다.

```
Predicate predicate = (v) -> v != null;
```

BiFunction 인터페이스

- BiFunction은 두 개의 인수를 취하고 하나의 객체를 반환하는 함수형 인터페이스이다.

```
BiFunction<Integer, Integer, Integer> func = (x1, x2) -> x1 + x2;
Integer result = func.apply(1, 2);

BiFunction<Integer, Integer, Double> func1 = (x1, x2) -> Math.pow(x1, x2);
Double result2 = func1.apply(10, 2);
```

메소드 참조

- 우리는 앞에서 익명 클래스를 사용하는 대신 람다식을 사용할 수 있다는 것을 배워서 알고 있다. 그러나 때로는 람다식이 실제로는 메소드에 대한 호출일 뿐이다. 아래의 람다식이 하는 일은 단지 println() 호출이다.

```
list.forEach(s -> System.out.println(s));
```

```
s  -> System.out.println(s)
```

↓

```
System.out::println
```

메소드 참조의 형식

익명 클래스에서 메소드 참조까지의 발전사

```
new Consumer<String>() {
    @Override
    public void accept(String s){
        System.out.println(s);
    }
}
```

익명 클래스 사용

```
s->System.out.println(s)
```

람다식 사용

```
System.out::println
```

메소드 참조 사용

람다식과 메소드 참조

- 메소드 참조도 코드 블록을 전달하는 동작 매개 변수화의 한 방법이다. 코드 블록이 하는 일이 단지 메소드 호출 뿐이라면 간단하게 해당 메소드만 보내자는 것이다.

표 14.4 람다식과 대응되는 메소드 참조의 예

람다식	메소드 참조
(Car car) -> Car.getPrice()	Car::getPrice
() -> Thread.currentThread().dumpStack()	Thread.currentThread()::dumpStack
(s) -> System.out.println(s) (s) -> this.isValidName(s)	System.out::println this::isValidName

메소드 참조의 종류

종류	문법	예제
정적 메소드 참조	ContainingClass::staticMethodName	Integer::parseInt
특정 객체의 인스턴스 메소드 참조	ContainingObject::instanceMethodName	System.out::println
특정 유형의 인스턴스 메소드 참조	ContainingType::methodName	String::toUpperCase
생성자 참조	ClassName::new	String::new

정적 메소드 참조

- 람다식이 왼쪽과 같은 형태일 때, 오른쪽처럼 변환할 수 있다.

```
(args) -> ClassName.staticMethod(args)
```

⇨

```
ClassName::staticMethod
```

(예제)

```
list.forEach(s -> StringUtils.capitalize(s));
```

⇨

```
list.forEach(StringUtils::capitalize);
```

예제

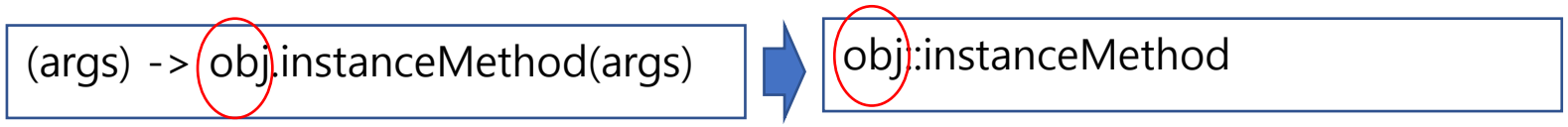
```
import java.util.function.BiFunction;

class Calculator {
        public static int add(int a, int b) {
                return a + b;
        }
}

public class Test {
        public static void main(String[] args) {
                BiFunction<Integer, Integer, Integer> obj = Calculator::add;
                int result = obj.apply(10, 20);
                System.out.println("주어진 수의 덧셈: " + result);
        }
}
```

```
주어진 수의 덧셈: 30
```

특정객체의 인스턴스 메소드 참조

교재 정오표 참고

- 외부에 정의된 특정 객체 obj의 메소드를 호출하는 경우



(예제)

```
public class Test {
        static void print(Supplier<Integer> f) {
                System.out.println(f.get());
        }
        public static void main(String[] args) {
                String s = "Hello World!";
                print(s::length);
        }
}
```

()->s.length() 람다식 대용

람다식 매개변수의 인스턴스 메소드 참조

교재 정오표 참고

- 매개변수 obj의 메소드를 호출하고 다른 매개변수를 인수로 사용하는 경우

(예제)

```
public static void main(String[] args) {
      String[] sArray = { "Kim", "Park", "Lee", "Choi", "Mary" };

      Arrays.sort(sArray, String::compareToIgnoreCase);
}
```

```
(String a, String b)->
a.compareToIgnoreCase(b)
```

예제: 메소드 참조 예제

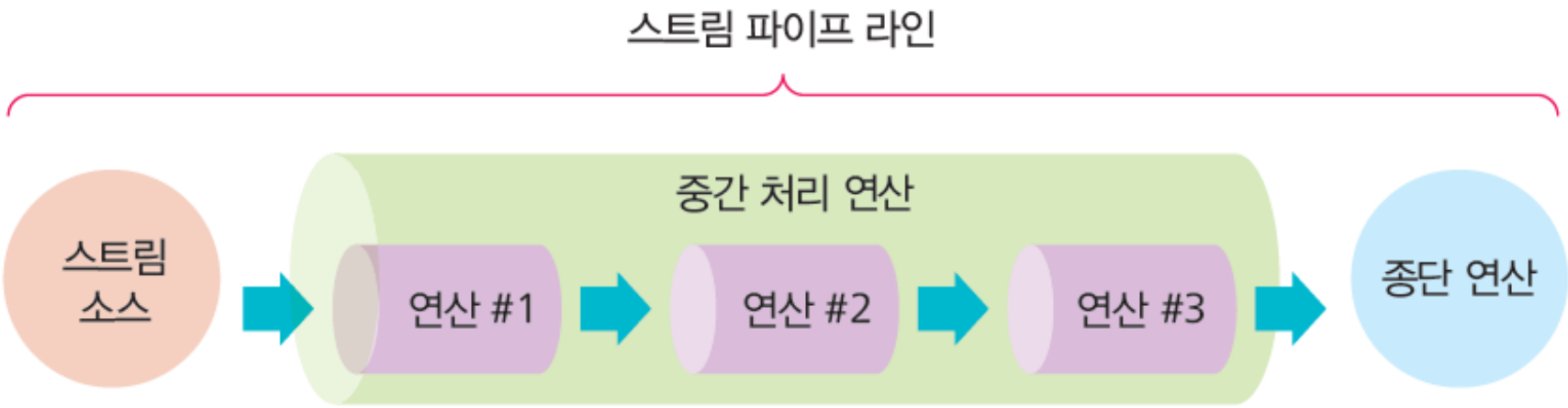
- 예를 들어 특정한 디렉터리 안에서 디렉터리와 파일을 구분하려고 한다고 하자. 우리는 파일이 주어지면 이것이 디렉터리인지 단순한 파일인지를 확인하는 메소드를 작성해야 한다. 다행히도 File 클래스에는 isDirectory() 메소드가 있다. 이 메소드는 File 객체를 받아서 부울 값을 반환한다. 그러나 필터링에 사용하려면 다음과 같이 FileFilter 객체로 감싼 후에, 객체 형태로 메소드에 전달해야 한다.

```
File[] direc = new File(".").listFiles(new FileFilter() {
    public boolean accept(File file) {
        return file.isDirectory();
    }
});
```

```
File [] direc = new File ( "."). listFiles (File :: isDirectory);
```

스트림

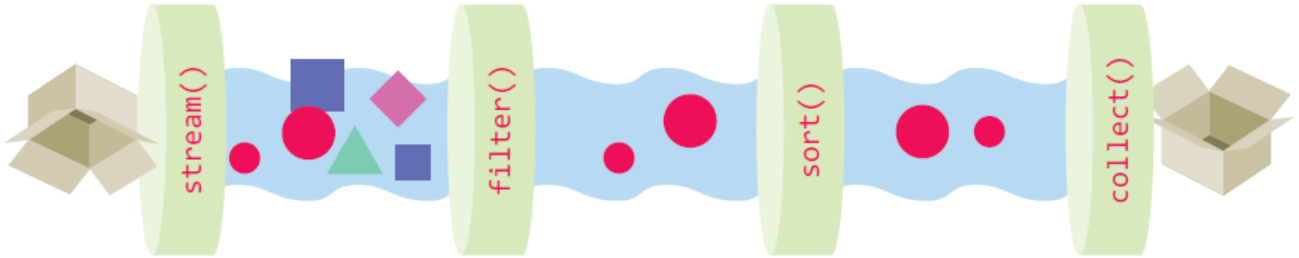
- 여기서의 스트림은 ArrayList와 같은 컬렉션에서 시작되는 스트림을 의미한다. 입출력에서와 같이 스트림은 한 번에 하나씩 생성되고 처리되는 일련의 데이터이다. 스트림 API를 이용하면 메소드는 입력 스트림에서 항목을 하나씩 읽고 처리한 후에, 항목을 출력 스트림으로 쓸 수 있다. 한 메소드의 출력 스트림은 다른 메소드의 입력 스트림이 될 수 있다.



예제: 스트림의 개념

```
public class Test {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("Kim", "Park", "Lee", "Choi", "Chee");    // (1)

        List<String> sublist = list.stream()          // (2) 스트림 생성
            .filter(s -> s.startsWith("C"))          // (3) 스트림 처리
            .sorted()                                // (4) 스트림 처리
            .collect(Collectors.toList());           // (5) 결과 생성

        System.out.println(sublist);                 // (6) 결과 출력
    }
}
```

```
[Chee, Choi]
```

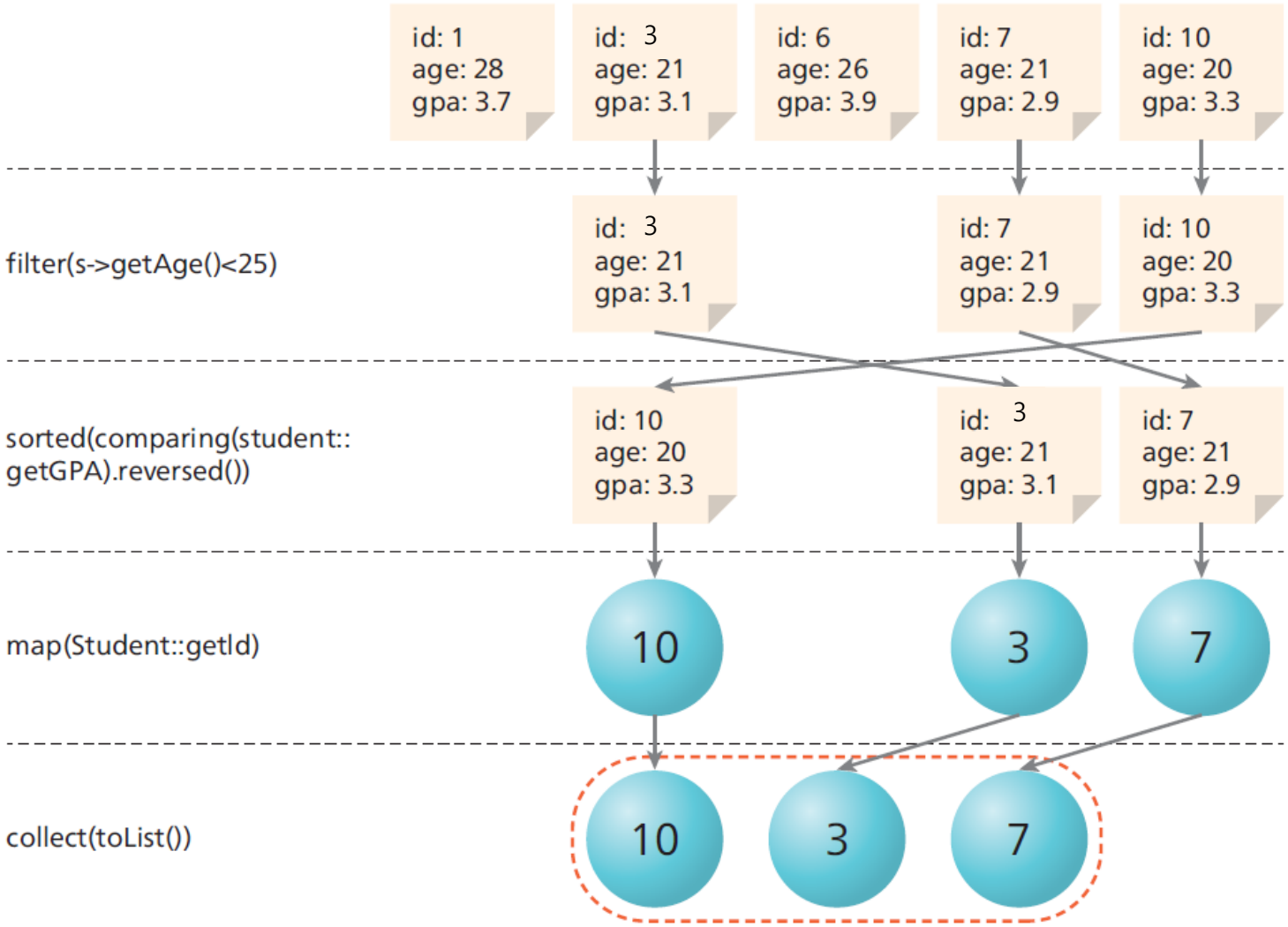


스트림의 장점

- 첫째, 컬렉션에 대한 일반적인 처리 패턴은 "찾기"(예: 가장 높은 평점의 학생 찾기) 또는 "그룹화"와 같은 SQL과 유사한 작업이다. 대부분의 데이터베이스에서는 이러한 작업을 선언적으로 지정할 수 있다.
- 컬렉션에서도 반복문을 사용하지 않고 SQL처럼 선언만 하여서 비슷한 작업을 할 수 있으면 얼마나 좋을까? 스트림 API를 사용하면 다음과 같이 간단하게 표현하는 것이 가능하다.

```
List<Integer> result =
    list.stream()
                .filter(s ->s.getAge() < 25)
                .sorted(comparing(Student::getGPA).reversed())
                .map(Student::getId)
                .collect(Collectors.toList());
```

스트림의 장점


id: 1
age: 28
gpa: 3.7
id: 3
age: 21
gpa: 3.1
id: 6
age: 26
gpa: 3.9
id: 7
age: 21
gpa: 2.9
id: 10
age: 20
gpa: 3.3
filter(s->getAge()<25)
id: 3
age: 21
gpa: 3.1
id: 7
age: 21
gpa: 2.9
id: 10
age: 20
gpa: 3.3
sorted(comparing(student::
getGPA).reversed())
id: 10
age: 20
gpa: 3.3
id: 3
age: 21
gpa: 3.1
id: 7
age: 21
gpa: 2.9
map(Student::getId)
10
3
7
collect(toList())
10
3
7

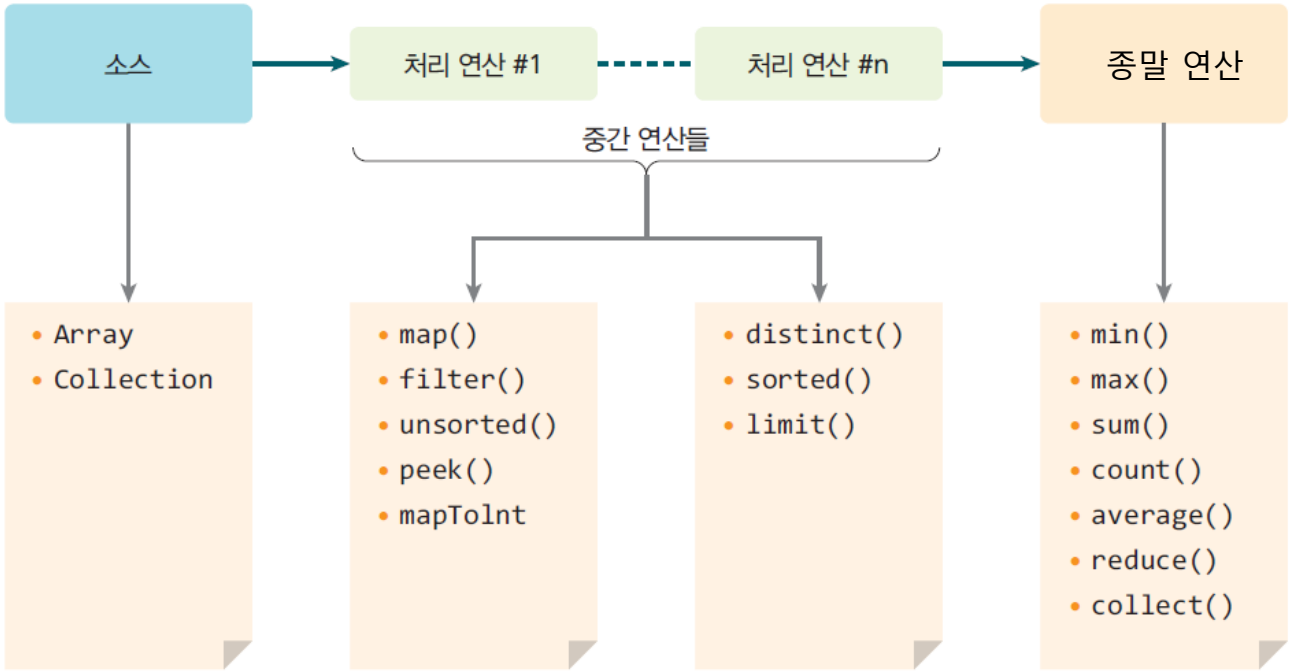
스트림의 장점

- 둘째, 큰 컬렉션을 효율적으로 처리하려면 멀티 코어 아키텍처를 활용하는 것이 좋다. 그러나 병렬 코드를 작성하는 것은 아직도 어렵고 오류가 발생하기 쉽다. 스트림 API를 사용하면 다중 스레드 코드를 한 줄도 작성하지 않고도 다중 코어 아키텍처를 활용할 수 있다. stream() 대신에 parallelStream()을 사용하면 스트림 API가 자동으로 쿼리를 여러 개의 코어를 활용하는 코드로 분해한다.

```
List<Integer> result =
    list.parallelStream()
            .filter(s ->s.getAge() < 25)
            .sorted(comparing(Student::getGPA).reversed())
            .map(Student::getId)
            .collect(toList());
```

스트림 연산

- 생성 단계: 스트림 객체를 생성하는 단계이다. 배열이나 컬렉션을 가지고 스트림을 생성할 수 있다.
- 처리 단계: 입력 데이터를 출력 데이터로 가공하는 연산이다.
- 종말 단계: 처리된 데이터를 모아서 결과를 만드는 연산이다.



생성단계

```
// 배열에서 만들기
String[] arr = { "Kim", "Lee", "Park" };
Stream<String> s2 = Arrays.stream(arr);

// 컬렉션에서 만들기
List<String> list = Arrays.asList("Kim", "Lee", "Park");
Stream<String> s1 = list.stream();
```

필터링

- 필터링은 조건에 맞는 데이터만을 통과시키는 연산이다. filter() 메소드를 사용하며, 이 메소드는 람다식을 인수로 받는다. 예를 들어서 문자열 중에서 "P"가 포함된 문자열만 통과시키려면 다음과 같은 코드를 사용한다.

```
List<String> list = Arrays.asList("Kim", "Lee", "Park");
Stream<String> s1 = list.stream()
                        .filter(s->s.contains("P"));
```

매핑 연산(map())

- 매핑 연산은 map() 메소드를 사용하며 기존의 데이터를 변형하여서 새로운 데이터로 생성하는 연산이다. 이 메소드도 람다식을 인수로 받는다. 예를 들어서 문자열들을 모두 소문자로 변환하려면 다음과 같은 코드를 사용한다

```
List<String> list = Arrays.asList("Kim", "Lee", "Park");
Stream<String> s1 = list.stream()
                        .map(s->s.toUpperCase());
```

```
Stream<String> s1 = list.stream()
                        .map(String::toUpperCase());
```

정렬 연산(sorted())

- 입력되는 데이터들을 어떤 기준에 따라 정렬하는 연산이다. 정렬 기준은 Comparator 객체가 된다. 기준이 주어지지 않으면 기본 정렬된다. 예를 들어 문자열들을 내림차순으로 정렬하려면 다음과 같은 코드를 사용한다.

```
Stream<String> s1 = list.stream()
                .sorted(Comparator.reverseOrder());
```

축소 연산(reduce())

- reduce()는 스트림의 요소에 대하여 어떤 함수를 가지고 축소 연산을 수행한다. 즉 요소들을 어떤 함수를 이용하여 결합하여서 하나의 값으로 만들 수 있다.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

종말 단계

- 종말 단계에서는 입력 데이터들을 모아서 결과를 생성한다. 여러 가지의 결과를 생성할 수 있도록 다양한 메소드들이 제공된다.

```
IntStream.of(20, 10, 30, 90, 60)                    // 정수를 스트림으로 생성
해주는 문장이다.
                .sorted()
                .collect(Collectors.toList());
```

```
int sum = IntStream.of(20, 10, 30, 90, 60)          // 정수를 스트림으로 생성해주는 문장이다.
            .sum();                                 // 합계를 계산하여 반환한다.

int count = IntStream.of(20, 10, 30, 90, 60)        // 정수를 스트림으로 생성해주는 문장이다.
            .count();                               // 합계를 계산하여 반환한다.
```

forEach() 연산

- forEach() 메소드를 사용하면 **스트림의 각 항목에 대하여** 어떤 특정한 연산을 수행할 수 있다.

```
List<String> list = Arrays.asList("Kim", "Lee", "Park");
Stream<String> s1 = list.stream()
                        .forEach(System.out::println);
```

예제:

- 1부터 8까지를 저장하는 컬렉션을 만들고 이 중에서 짝수만을 골라내는 코드를 스트림 API로 만들어보자. 람다식을 사용해보자.

```
입력데이터 =[1, 2, 3, 4, 5, 6, 7, 8]
실행결과 =[4, 16, 36, 64]
```

```
public class StreamExample1 {
        public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
        System.out.println("입력데이터 ="+numbers);
        List<Integer> result =
           numbers.stream()
                .filter(n -> {
                        return n % 2 == 0;
                    })
                .map(n -> {
                        return n * n;
                    })
                .collect(Collectors.toList());
        System.out.println("실행결과 ="+result);
        }
}
```

예제:

- 스트림은 각 요소에서 정보를 추출하는 데 사용할 수 있다. 단어들의 리스트를 받아서 각 단어의 길이 리스트를 반환하는 코드를 작성해보자.

```
public class StreamExample2 {
		public static void main(String[] args) {
		List<String> words = Arrays.asList("Java", "Stream", "Library");
		System.out.println("입력데이터 ="+words);
		List<Integer> result =     words.stream()
			.map(String::length)
			.collect(Collectors.toList());
		System.out.println("실행결과 ="+result);
		}
}
```

```
입력데이터 =[Java, Stream, Library]
실행결과 =[4, 6, 7]
```

예제:

- 가전 제품들을 ArrayList에 저장하고, 가격이 300만 원 이상인 가전 제품의 이름을 출력하는 프로그램을 작성해보자. 스트림 API를 사용한다.

```
[TV, Air Conditioner]
```

```
class Product{
    int id;
    String name;
    int price;
    public Product(int id, String name, int price) {
                        super();
                        this.id = id;
                        this.name = name;
                        this.price = price;
            }
}
public class StreamTest  {
    public static void main(String[] args) {
        List<Product> list = new ArrayList<Product>();
        list.add(new Product(1,"NoteBook", 100));
        list.add(new Product(2,"TV", 320));
        list.add(new Product(3,"Washing Machine", 250));
        list.add(new Product(4,"Air Conditioner", 500));

        List<String> result =list.stream()
                            .filter(p -> p.price > 300)
                            .map(p->p.name)
                            .collect(Collectors.toList());
        System.out.println(result);
    }
}
```

Mini Project: 스트림 응용하기

- 여러 가지 상품을 생성하고 ArrayList에 저장한 후에 사용자로부터 조건을 받아서 검색하는 프로그램을 작성해보자. 람다식이나 스트림 API, 메소드 참조 등을 적극적으로 사용해보자.

```
상품을 검색하세요.
상품의 이름(*은 모든 상품을 의미): NoteBook
상품의 가격 상한: 5000000

검색된 상품은 HP NoteBook Model 100 입니다.
```

연습문제 1

- 람다식을 메소드 참조로 변환하는 표 채우기

람다식	메소드 참조
x->System.out.println(x);	
(String s)->s.toLowerCase();	
(s1, s2) -> s1.compareTo(s2);	
(v) -> obj.setValue(v);	

연습문제 3

- 다음의 코드를 스트림 API와 메소드 참조, 또는 람다식을 이용하여 바꿔보자.

- a)

```
List<String> list1 = Arrays.asList("Apple", "Banana", "Pear", "Cherry");
List<String> list2 = new ArrayList<>();
for(String string: list1){
    if(string.equals("Apple") || string.equals("Cherry")){
        list2.add(string);
    }
}
```

- b)

```
List<String> list3 = new ArrayList<>();
for(String string:list2){
    list3.add(String + " (Fruits)");
}
```