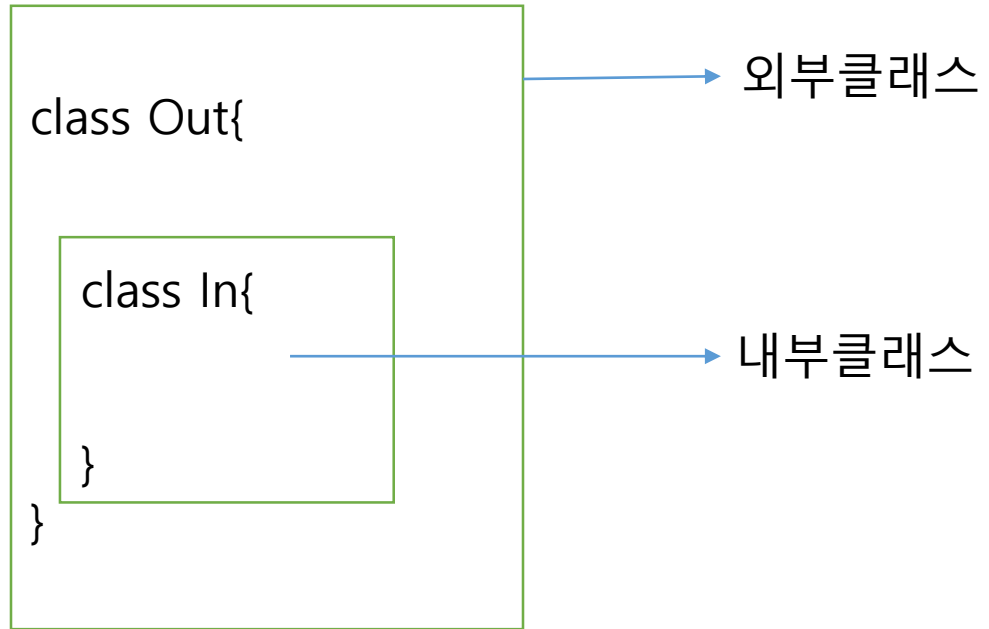


내부클래스



인스턴스 내부클래스
정적 내부클래스
지역내부클래스
익명내부클래스

```
//외부클래스
class OutClass{
    private int num=100;
    private InClass inclass;

    public OutClass() {
        inclass = new InClass();
    }

    public void OutMethod() {
        inclass.inMethod();
    }

    //내부 클래스
    class InClass{
        int inNum=200;

        void inMethod() {
            System.out.println(num);
            System.out.println(inNum);
        }
    }
}
```

```
public class InnerTest {

    public static void main(String[] args) {

        OutClass out = new OutClass();
        out.OutMethod();

        OutClass.InClass inclass=out.new InClass();
        inclass.inMethod();

    }

}
```

```

class OuterClass{

    int outNum =100;

    // 메서드내에서 내부클래스 , 지역내부클래스
    public Runnable getRunnable(int i) {
        int localNum=20;

        class MyRunnable implements Runnable{

            @Override
            public void run() {
                System.out.println( outNum);
                System.out.println(localNum);
                System.out.println(i);
            }
        }
        return new MyRunnable();
    }
}

```

```

public class LocalInnerTest {

    public static void main(String[] args) {

        OuterClass outer = new OuterClass();
        Runnable runnable =outer.getRunnable(10);
        runnable.run();
    }

}

```

```

class OuterClass2{
    int outNum =100;

    // 메서드내에서 내부클래스 , 익명클래스
    public Runnable getRunnable(int i) {
        int localNum=20;
        //익명클래스

        Runnable runnable = new Runnable(){
            @Override
            public void run() {
                System.out.println( outNum);
                System.out.println(localNum); System.out.println(i);
            }
        };

        return runnable;
    }
}

```

```

public class LocalInnerAnonymouseTest2 {

    public static void main(String[] args) {

        OuterClass2 outer = new OuterClass2();
        Runnable runnable =outer.getRunnable(10);
        runnable.run();
    }
}

```

제네릭(Generic)이란?

여러참조형이
쓰일 수 있는 곳에
특정자료형을 지정하지 않고
클래스나, 매서드를 정의한 후 (제네릭클래스, 제네릭매서드)

사용하는 시점에 어떤 자료형을 사용할 것인지를 지정하는 방식이다.

제네릭은
제네릭이 아닌 것을 제네릭으로 변경해 보는 방법으로 제네릭을 익히는 것이
가장 빨리 제네릭을 이해하는 길이다

지네릭스

지네릭스란?

- 컴파일시 타입을 체크해 주는 기능 (compile-time type check) jdk 1.5
- 객체의 타입 안정성을 높이고 형변환의 번거로움을 줄여줌
- 실행시 발생하는 **ClassCastException** 예외를 컴파일시점에서 체크할 수 있게 함으로써 실행시 발생하는 오류를 막을 수 있게 함

```
ArrayList list = new ArrayList();
list.add( new Score(100,90) );
list.add( new Score(90,80));
list.add( new User("홍길동"));
```

```
( (Score) list.get(2)).getKor() ; // =>
```

실행을 시켜야 오류를 발생함 (컴파일시점에 오류를 발생하지 않는다)

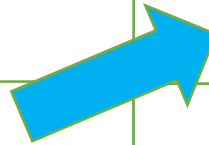
지네릭클래스 만들 때 : <T>

지네릭클래스 제한하기 : <T extends Material >

매개변수에서 지네릭 와일드문자 사용하기 : < ? extends T>
< ? super T >

타입 변수
T가 아닌 다른 것도 가능함

```
public class TreeDPrinterPlastic {  
    private Plastic material;  
  
    public void setMaterial( Plastic material) {  
        this.material = material;  
    }  
  
    public void pirnt() {  
        System.out.println( "3D print use :" + material);  
    }  
}
```



```
public class TreeDPrinterPowder {  
    private Powder material;  
  
    public void setMaterial( Powder material) {  
        this.material = material;  
    }  
  
    public void pirnt() {  
        System.out.println( "3D print use :" + material);  
    }  
}
```

```
public class GenericPrinter<T> {  
    private T material;  
  
    public void setMaterial( T material) {  
        this.material = material;  
    }  
  
    public void print() {  
        System.out.println ( "3D print:" + material);  
    }  
}
```


```
public class TestMain{  
    public static void main(String[] args) {  
  
        GenericPrinter<Powder> printer = new GenericPrinter<>();  
        printer.setMaterial(new Powder());  
        printer.print();  
  
        GenericPrinter<Plastic> printer = new GenericPrinter<>();  
        printer.setMaterial(new Plastic());  
        printer.print();  
  
    }  
}
```


제한된 제네릭 클래스 만들기

```
public class GenericPrinterMaterial<T extends Material> {  
    private T material;  
  
    public void setMaterial( T material) {  
        this.material = material;  
    }  
  
    public void print() {  
        System.out.println ( "3D print:" + material);  
    }  
  
    public static void main(String[] args) {  
  
        GenericPrinterMaterial<Powder> printer = new GenericPrinterMaterial<>();  
        printer.setMaterial(new Powder());  
        printer.print();  
  
        GenericPrinterMaterial<Plastic> printer2 = new GenericPrinterMaterial<>();  
        printer2.setMaterial(new Plastic());  
        printer2.print();  
  
        //error 발생  
        // GenericPrinter2<Water> printer3 = new GenericPrinter2<>();  
        //printer3.setMaterial(new Water());  
        //printer3.print();  
  
    }  
}
```

<>안에 들어 올 수 있는 class type이 제한 되었다.
Material을 상속받은 클래스만 가능하다

```
public class ArrayListTest {  
  
    public static void main(String[] args) {  
        ArrayList<String> list = new ArrayList<>();  
        list.add("one");  
        list.add("two");  
        print( list);  
  
        ArrayList<Object> list2 = new ArrayList<>();  
        list.add("one");  
        list.add("two");  
  
        print ( list2);  
        //print2( list2);  
  
    }  
  
    public static void print( ArrayList< ? super String> list) {  
        list.forEach( str -> System.out.println(str));  
    }  
  
    public static void print2( ArrayList< String> list) {  
        list.forEach( str -> System.out.println(str));  
    }  
  
}
```



와일드카드
하나의 참조변수로 서로 다른 타입이 대
입된 제네릭객체를 다루기 위한것

```

class Person {
    String name;

    // 생성자 오버로딩
    Person(String name) {
        this.name = name;
    }

    public String toString() {
        return name;
    }
}

// Person 상속 Man 클래스
class Man extends Person {
    Man( String name){
        super(name);
    }
}

// Person 상속 Woman 클래스
class Woman extends Person {
    Woman( String name){
        super(name);
    }
}

```

```

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

        // Person
        ArrayList<Person> listP = new ArrayList<>();
        listP.add(new Person("이사람"));
        listP.add(new Person("김사람"));
        printData(listP);

        // Man
        ArrayList<Man> listM = new ArrayList<>();
        listM.add(new Man("공유"));
        listM.add(new Man("스티브잡스"));
        printData(listM);

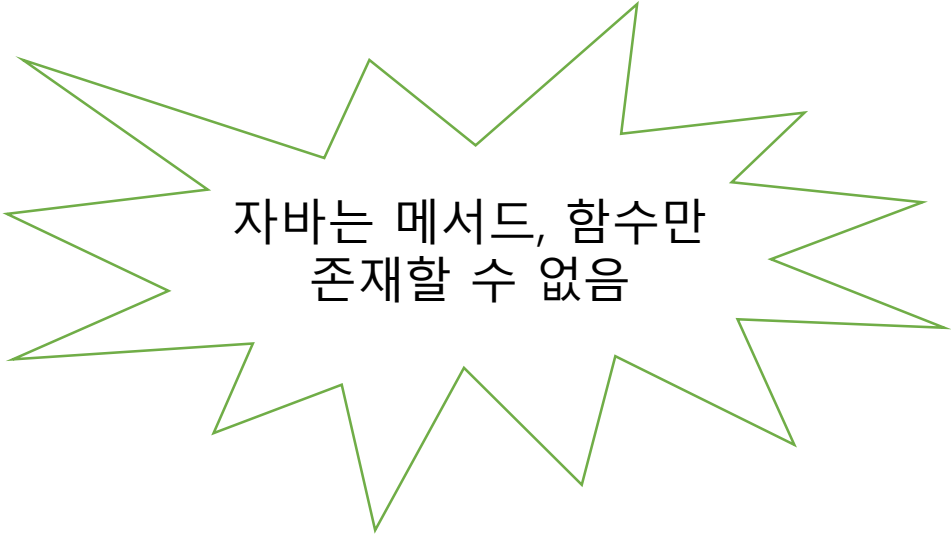
        // Woman
        ArrayList<Woman> listW = new ArrayList<>();
        listW.add(new Woman("아이유"));
        listW.add(new Woman("김연아"));
        // printData(listW); → Man 클래스의 상위 클래스가 아니기 때문에 메소드 호출 불가
    }

    // Man 클래스와 그 상위 클래스로 생성된 인스턴스만 매개변수로 전달 가능
    public static void printData(ArrayList<? super Man> list) {
        for (Object obj : list) {
            System.out.println(obj);
        }
    }
}

```

? super Man
? extends Man
?





자바는 메서드, 함수만
존재할 수 없음

람다식 lambda

함수형프로그램을 가능하게 해 주는것이 람다이다

<<익명메서드>>
익명객체가 생성되고 익명객체의 메서드가 실행됨

람다식

(lambda expression)

- . 자바에서 함수형 프로그래밍(functional programming)을 구현하는 방식
- . 자바 8부터 지원
- . 클래스를 생성하지 않고 함수의 호출만으로 기능을 수행 (익명클래스의 익명객체가 생성됨)

함수형 프로그램

순수함수(pure function)를 구현하고 호출함으로써 외부 자료에 부수적인 영향을 주지 않고
매개 변수만을 사용하도록 만든 함수
함수를 기반으로 구현

입력 받은 자료를 기반으로 수행되고 외부에 영향을 미치지 않으므로 병렬처리등에 가능

람다식 구현하기

- . 익명함수 만들기
- . 매개변수와 매개변수를 활용한 실행문으로 구현
(매개변수) -> { 실행문; }

두 수를 입력 받아 더하는 함수 add

```
int add(int x, int y){  
    return x+y ;  
}
```

$(x,y) \rightarrow \{ \text{return } x+y ; \}$

$(x,y) \rightarrow x+y$

함수의 이름과 반환형을 없애고 $\rightarrow \{ \text{실행문; } \}$

```
public class ExlambdaAddTest {  
  
    public static void main(String[] args) {  
  
        AddInterface instance = (x,y) -> x+y ;  
  
        AddInterface instance2= new AddInterface() {  
  
            @Override  
            public int add(int num1, int num2) {  
                return num1 + num2;  
            }  
        };  
  
        int result = instance.add(5, 7);  
        int result2 = instance2.add(5,7);  
  
        System.out.println(result);  
        System.out.println(result2);  
  
    }  
}
```

```
@FunctionalInterface  
public interface AddInterface {  
    int add(int num1, int num2);  
  
}
```

함수형 인터페이스

람다식을 선언하기 위한 인터페이스

익명함수와 매개변수만으로 구현되므로 단 하나의 메서드만을 가져야 함
(두 개 이상의 메서드인 경우 어떤 메서드의 호출인지 모호해짐)

@FunctionalInterface 애노테이션

함수형 인터페이스라는 의미, 여러 개의 메서드를 선언하면 에러남


```

public class Ex01lambda {
    public static void main(String[] args) {

        MyInterfacImp f = new MyInterfacImp();
        f.specMethod();

        //익명클래스 사용하는 방식
        MyInterface f2 = new MyInterface() {
            @Override
            public void specMethod() {
                System.out.println("기능을 구현합니다");
            }
        };
        f2.specMethod();
    }

}

class MyInterfacImp implements MyInterface{

    @Override
    public void specMethod() {
        System.out.println("기능을 구현합니다.");
    }

}

```

```

@FunctionalInterface
public interface MyInterface {
    public abstract void specMethod();
}

```

```

public class Ex01lambda {
    public static void main(String[] args) {

        MyInterfacelmp f = new MyInterfacelmp();
        f.specMethod();

        //익명클래스 사용하는 방식
        MyInterface f2 = new MyInterface() {
            @Override
            public void specMethod() {
                System.out.println("기능을 구현합니다");
            }
        };
        f2.specMethod();

        //람다식
        MyInterface f3 = ()-> System.out.println("기능을 구현합니다.");
        f3.specMethod();
    }
}

class MyInterfacelmp implements MyInterface{
    @Override
    public void specMethod() {
        System.out.println("기능을 구현합니다.");
    }
}

```

```

@FunctionalInterface
public interface MyInterface {
    public abstract void specMethod();
}

```

매개변수로 전달하는 람다식

```
class Calc{  
    public static void calcAdd( AddInterface f )  
    {  
        int result= f.add( 5,7);  
        System.out.println(result);  
    }  
}
```

```
public class ExlambdaAddTest {  
    public static void main(String[] args) {  
        Calc.calcAdd( (x,y) -> x+y );  
    }  
}
```

```
@FunctionalInterface  
public interface AddInterface {  
    int add(int num1, int num2);  
  
}
```

메서드	람다식
<pre>int max(int a, int b){ return a>b ? a : b; }</pre>	<pre>(a,b) -> a>b ? a:b</pre>
<pre>int printVar(String name, int i) { System.out.println(name + "=" + i); }</pre>	<pre>(name,i) -> System.out.println(name + ":" + i)</pre>
<pre>int squre(int x){ return x *x ; }</pre>	<pre>x-> x *x</pre>
<pre>int getRandom(){ return (int) (Math.random() *6); }</pre>	<pre>() -> (int) (Math.random() *6)</pre>

java.util.function 패키지

함수형인터페이스	메서드	설명
java.lang.Runnable	void run()	매개변수도 없고 반환값도 없음
Supplier<T>	T get()	매개변수가 없고, 반환값만 있음
Consumer<T>	void accept(T t)	매개변수만 있고 반환값이 없음
Function<T,R>	R apply(T t)	하나의 매개변수를 받아서 결과를 반환
Predicate<T>	Boolean test(T t)	조건식을 표현하는데 사용함 매개변수는 하나 반환 타입은 Boolean

알맞는 함수형 인터페이스(`java.util.function`패키지)를 연결하세요

- **Predicate<Integer>**
- **Supplier<Integer>**
- **Function<Integer, Integer>**
- **Consumer<Integer>**

- **f= () -> (int) (Math.random()* 100) +1 ;**
- **f= i -> System.out.println(i);**
- **f= i -> i%2 ==0 ;**
- **f= i -> i -> i+2*3;**

```
public class ExFunctionalInterfaceUtil {  
  
    public static void main(String[] args) {  
  
        Predicate<String> isEmptyStr = s -> s.length() == 0;  
        String s = "ioioioio";  
  
        if(isEmptyStr.test(s))  
            System.out.println("this is an empty String");  
        else  
            System.out.println(s);  
    }  
}
```

실습: 각 함수형 인터페이스를 구현하시오

Runnable : 실행하기 (입력x, 반환x)
Consumer<T> : 소비하기 (입력o, 반환x)
Supplier<T> : 공급하기 (입력x, 반환o)
Predicate<T> : 판별하기 (입력o, 반환 Boolean)
Function<T,R> : 함수 (입력o, 반환o)

1. Runnable : 버킷리스트 출력하기
2. Consumer : 3만원으로 장보기
3. Supplier : 요리레시피를 출력하고 내가 만든 요리이름 반환하기
4. Predicate : 입력으로 받은 요리가 내가 만든 요리인 경우 true, false반환하기
5. Function : 입력 하나 반환이 있는 함수 만들기
 - 입력으로 들어오는 수의 제곱 반환하기
 - 입력으로 들어오는 수의 범위 안의 난수 반환하기 10-> 10범위 안의 난수가 리턴됨
 - 입력으로 들어오는 금액에 대한 화폐매수 구하기

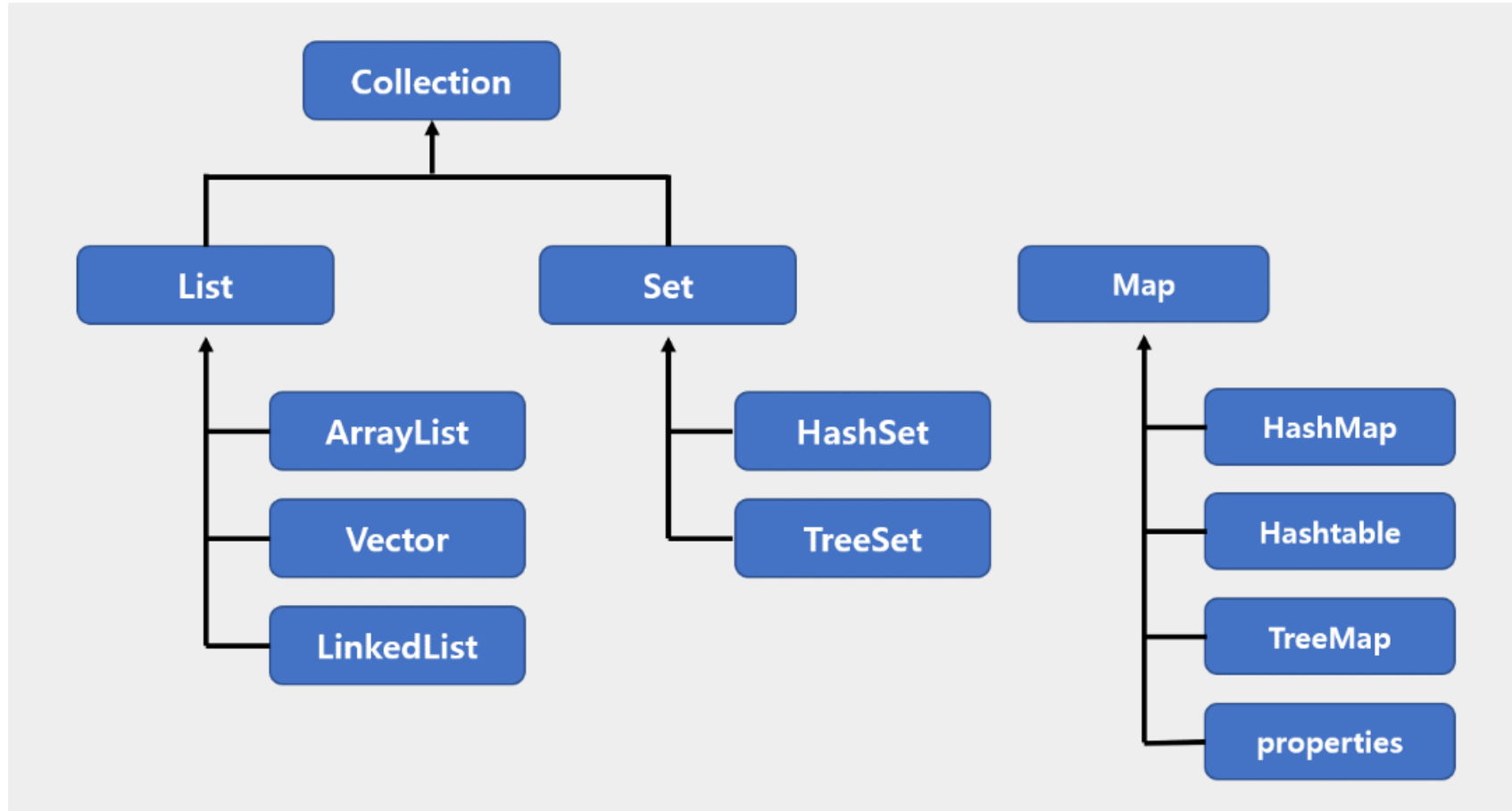
2인 1조 과제

BiConsumer<T,U> : 두 개의 매개변수를 받아서 사용
BiFunction<T,U,R> : 두 개의 매개변수를 받아서 사용하고 리턴함
BiPredicate<T,U> : 두 개의 매개변수를 받아서 true, false 리턴

Collection

표준화된 방법으로 다루겠다

자료구조:
데이터를 저장하는 방법



```
//set , 중복불가능, 순서가 없음  
Set<String> set = new HashSet<>();  
set.add("hi");  
set.add("hi2");  
set.add("hi2");
```

```
//list , 중복허용 , 순서있음,  
List<String> list = new ArrayList<>();  
list.add("hi");  
list.add("hi2");  
list.add("hi2");
```

```
//map 키:값 쌍으로 저장
```

```
Map<String, String> map = new HashMap<>();  
map.put("key1", "test내용1");  
map.put("key2", "test내용2");  
map.put("key3", "test내용3");
```

// List, Set의 부모형 Collection
// Iterator는 데이터를 순회하기 위한 표준된 방법을 제공함

```
Collection<String> collection=null;  
//가능함  
collection = set;  
Iterator<String> i =collection.iterator();  
//  
while( i.hasNext()) {  
    System.out.println( i.next());  
}
```

```
collection= list;  
Iterator<String> i2 =collection.iterator();  
//  
while( i2.hasNext()) {  
    System.out.println( i2.next());  
}
```

스트림

표준화된 방법으로 다루겠다

스트림(stream)

자료의 대상과 관계없이 동일한 연산을 수행
배열, 컬렉션을 대상으로 동일한 연산을 수행 함
일관성 있는 연산으로 자료의 처리를 쉽고 간단하게 함

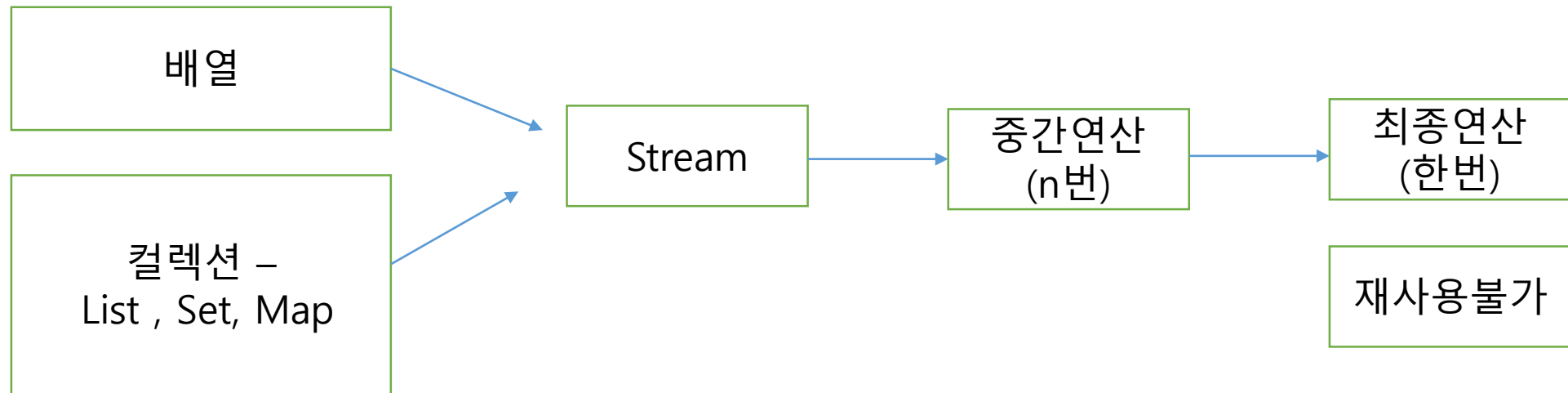
한 번 생성하고 사용한 스트림은 재사용할 수 없음
자료에 대한 스트림을 생성하여 연산을 수행하면 스트림은 소모됨
다른 연산을 위해서는 새로운 스트림을 생성 함
스트림 연산은 기존 자료를 변경하지 않음

자료에 대한 스트림을 생성하면 별도의 메모리 공간을 사용함으로 기존 자료를 변경하지 않음

스트림에 대한 중간 연산과 최종연산으로 구분됨

스트림에 대한 중간연산은 여러 개 적용 될 수 있지만 최종연산은 마지막에 한 번만 적용됨
최종연산이 호출되어야 중간연산의 결과가 모두 적용됨
이를 지연연산이라고 함

표준화된
방법으로 다루겠다
<진정한 통일>



```
ArrayList<String> list = new ArrayList<>();  
list.add("hi");  
list.add("hello");  
list.add("hello");  
list.add("bye");  
list.add("hello");
```

//hello와 같은게 몇개인지 구하기

```
long cnt = list.stream().filter( str -> str.equals("hello")).count();  
System.out.println( cnt) ;
```

//hello와 같은것만 출력하기

```
list.stream().filter( str -> str.equals("hello")).forEach( str-> System.out.println(str));
```

// 위의결과에서 위에서 2개만 출력

```
list.stream().filter( str -> str.equals("hello")).limit(2).forEach(str-> System.out.println(str));
```

```
IntStream s = new Random().ints(1,46);
s.distinct().limit(6).sorted().forEach( x->System.out.print(x + " ,"));
HashSet<Integer> lottos = new HashSet<Integer>();
Random r = new Random();

//중복제거
for( int i=0; i< 20; i++) {
    lottos.add( r.nextInt(46));
}

System.out.println("size=" + lottos.size());

Object[] lottosArray= lottos.toArray();

Integer[] result= new Integer[6]; // 배열카피

System.arraycopy(lottosArray, 0, result, 0, 6);

for( int i=0 ; i< result.length ;i++ ) {
    System.out.print( result[i] + ",");
}
```


배열을 스트림으로

```
int[] arr = {1,2,3,4,5};  
Arrays.stream(arr).forEach( n-> System.out.println(n) );
```

ArrayList 스트림으로

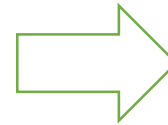
```
ArrayList<String> list = new ArrayList<>();
```

```
list.add("one");  
list.add("two");  
list.add("three");  
list.add("four");
```

```
list.stream().forEach( str -> System.out.println(str);  
list.stream().sorted().forEach( str -> System.out.println(str);
```

```
Class Optional<T>{  
    T value;  
}
```

간접적으로 null 다룸 (NullPointerException 발생)
null를 직접적으로 다룬다면 null 체크를 해야함



Optional 객체 사용

```

package javaprj.optional;
import java.util.ArrayList;
import java.util.Optional;

public class OptionalTest0 {

    public static void main(String[] args) {
        Optional<User> user = searchUserOptional("gy");
        System.out.println( user.orElse(new User()).getId());

        // User user2 = searchUser("gy");
        // System.out.println( user2.getId()); // NullPointerException
    }

}

```

```

private static Optional<User> searchUserOptional(String id) {
    ArrayList<User> users = new ArrayList<>();
    users.add(new User("hong" ,"t11"));
    users.add(new User("kim" ,"t11"));
    users.add(new User("lee" ,"t11"));
    return users.stream().filter( user -> user.getId().equals(id))
        .findFirst();
}

```

```

private static User searchUser(String id) {
    User user=null;
    ArrayList<User> users = new ArrayList<>();
    users.add(new User("hong" ,"t11"));
    users.add(new User("kim" ,"t11"));
    users.add(new User("lee" ,"t11"));

    for( User u: users) {
        if( u.getId().equals(id)) {
            user = u;
        }
    }
    return user; // 못찾으면 null 반환
}

```

```
String str="" ; // char[] str = new char[0];
```

```
String str2=null;
```

```
String str="hello";
```

```
Optional<String> obj = Optional.ofNullable( str );
```