제네릭(Generic) 타입

Contents

1절. 왜 제네릭을 사용해야 하는가?

❖ 제네릭(Generic) 타입이란?

- ‘컴파일 단계’에서 ‘잘못된 타입 사용될 수 있는 문제’ 제거 가능
- 자바5부터 새로 추가
- 컬렉션, 람다식(함수적 인터페이스), 스트림 등에서 널리 사용
- 제네릭을 모르면 API 도큐먼트 해석 어려우므로 학습 필요

1절. 왜 제네릭을 사용해야 하는가?

❖ 제네릭을 사용하는 코드의 이점

- 컴파일 시 강한 타입 체크 가능
  - 실행 시 타입 에러가 나는 것 방지
  - 컴파일 시에 미리 타입을 강하게 체크해서 에러 사전 방지
- 타입변환 제거 가능

```
List list = new ArrayList();
list.add("hello");
String str = (String) list.get(0);
```

```
List<String> list = new ArrayList<String>();
list.add("hello");
String str = list.get(0);
```

2절. 제네릭 타입

❖ 제네릭 타입이란?

- 타입을 파라미터로 가지는 클래스와 인터페이스
- 선언 시 클래스 또는 인터페이스 이름 뒤에 “<>” 부호 붙임
- “<>” 사이에는 타입 파라미터 위치

- 타입 파라미터
  - 일반적으로 대문자 알파벳 한 문자로 표현
  - 개발 코드에서는 타입 파라미터 자리에 구체적인 타입을 지정해야

2절. 제네릭 타입

❖ 제네릭 타입 사용 여부에 따른 비교

- 제네릭 타입을 사용하지 않은 경우
  - Object 타입 사용 → 빈번한 타입 변환 발생 → 프로그램 성능 저하

```
public class Box {
   private Object object;
   public void set(Object object) { this.object = object; }
   public Object get() { return object; }
}
```

```
Box box = new Box();
box.set("hello");                  //String 타입을 Object 타입으로 자동 타입 변환해서 저장
String str = (String) box.get();   //Object 타입을 String 타입으로 강제 타입 변환해서 얻음
```

2절. 제네릭 타입

❖ 제네릭 타입 사용 여부에 따른 비교

- 제네릭 타입을 사용하지 않은 경우
  - Object 타입 사용 → 빈번한 타입 변환 발생 → 프로그램 성능 저하

```
public class Box {
   private Object object;
   public void set(Object object) { this.object = object; }
   public Object get() { return object; }
}
```

```
Box box = new Box();
box.set("hello");                  //String 타입을 Object 타입으로 자동 타입 변환해서 저장
String str = (String) box.get();   //Object 타입을 String 타입으로 강제 타입 변환해서 얻음
```

2절. 제네릭 타입

❖ 제네릭 타입 사용 여부에 따른 비교

- 제네릭 타입 사용한 경우
  - 클래스 선언할 때 타입 파라미터 사용
  - 컴파일 시 타입 파라미터가 구체적인 클래스로 변경

```
public class Box<T> {
   private T t;
   public T get() { return t; }
   public void set(T t) { this.t = t; }
}
```

```
Box<String> box = new Box<String>();
```

```
public class Box<String> {
   private String t;
   public void set(String t) { this.t = t; }
   public String get() { return t; }
}
```

```
Box<String> box = new Box<String>();
box.set("hello");
String str = box.get();
```

```
Box<Integer> box = new Box<Integer>();
```

```
public class Box<Integer> {
   private Integer t;
   public void set(Integer t) { this.t = t; }
   public Integer get() { return t; }
}
```

```
Box<Integer> box = new Box<Integer>();
box.set(6);
int value = box.get();
```

3절. 멀티 타입 파라미터

❖ 제네릭 타입은 두 개 이상의 타입 파라미터 사용 가능

- 각 타입 파라미터는 콤마로 구분
  - Ex) class<K, V, …> { … }
  - interface<K, V, …> { … }

```
public class Product<T, M> {
    private T kind;
    private M model;

    public T getKind() { return this.kind; }
    public M getModel() { return this.model; }

    public void setKind(T kind) { this.kind = kind; }
    public void setModel(M model) { this.model = model; }
}
```

```
Product<Tv, String> product = new Product<Tv, String>();
```

- 자바 7부터는 다이아몬드 연산자 사용해 간단히 작성과 사용 가능

```
Product<Tv, String> product = new Product<>();
```

3절. 멀티 타입 파라미터

❖ 제네릭 타입은 두 개 이상의 타입 파라미터 사용 가능

```
public class Product<T, M> {
    private T kind;
    private M model;

    public T getKind() {
        return kind;    }
    public M getModel() {
        return model;   }
    public void setModel(M model) {
        this.model = model; }
    public void setKind(T kind) {
        this.kind = kind;   }
}

class Tv{
    public void tvPrn() {
        System.out.println("TV 종류");    }
}
class Car{
    public void carPrn() {
        System.out.println("Car 종류");   }
}
```

```
public class ProductEx {
    public static void main(String[] args) {
        Product<Tv, String> prod1 = new Product<>();
        prod1.setKind(new Tv());
        prod1.setModel("스마트TV");

        Tv tv = prod1.getKind();
        String tvModel = prod1.getModel();
        tv.tvPrn();
        System.out.println("==> " + tvModel);

        Product<Car, String> prod2 = new Product<>();
        prod2.setKind(new Car());
        prod2.setModel("그랜져");

        Car car = prod2.getKind();
        String carModel = prod2.getModel();
        car.carPrn();
        System.out.println("==> " + carModel);
    }
}
```

4절. 제네릭 메소드

❖ 제네릭 메소드

- 매개변수 타입과 리턴 타입으로 타입 파라미터를 갖는 메소드
- 제네릭 메소드 선언 방법
  - 리턴 타입 앞에 “<>” 기호를 추가하고 타입 파라미터 기술
  - 타입 파라미터를 리턴 타입과 매개변수에 사용

```
public <타입파라미터,...> 리턴타입 메소드명(매개변수,...) { ... }
```

```
public <T> Box<T> boxing(T t) { ... }
```

- 제네릭 메소드 호출하는 두 가지 방법

```
리턴타입 변수 = <구체적타입> 메소드명(매개값);      //명시적으로 구체적 타입 지정
리턴타입 변수 = 메소드명(매개값);                  //매개값을 보고 구체적 타입을 추정
```

```
Box<Integer> box = <Integer>boxing(100);      //타입 파라미터를 명시적으로 Integer 로 지정
Box<Integer> box = boxing(100);               //타입 파라미터를 Integer 으로 추정
```

4절. 제네릭 메소드

❖ 제네릭 메소드

```
public class Util {
    public static <T> Box<T> boxing(T t){
        Box<T> box = new Box<T>();
        box.setT(t);
        return box;
    }
}
class Box<T>{
    private T t;

    public T getT() { return t; }
    public void setT(T t) { this.t = t; }
}
```

```
public class UtilEx {
    public static void main(String[] args) {
        Box<Integer> box1 = Util.boxing(100);
        int intVal = box1.getT();
        System.out.println("box1 = " + intVal);

        Box<String>  box2 = Util.boxing("홍길동");
        String strVal = box2.getT();
        System.out.println("box2 = " + strVal);
    }
}
```

컬렉션 프레임워크

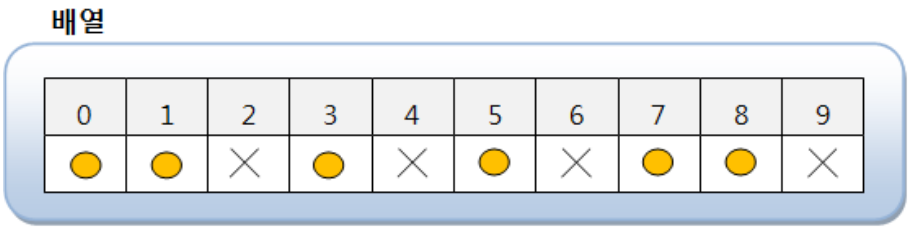
Contents

1절. 컬렉션 프레임워크 소개

❖ 컬렉션 프레임워크(Collection Framework)

- 컬렉션
  - 사전적 의미로 요소(객체)를 수집해 저장하는 것

- 배열의 문제점
  - 저장할 수 있는 객체 수가 배열을 생성할 때 결정
    → 불특정 다수의 객체를 저장하기에는 문제
  - 객체 삭제했을 때 해당 인덱스가 비게 됨
    → 낱알 빠진 옥수수 같은 배열
    → 객체를 저장하려면 어디가 비어있는지 확인해야



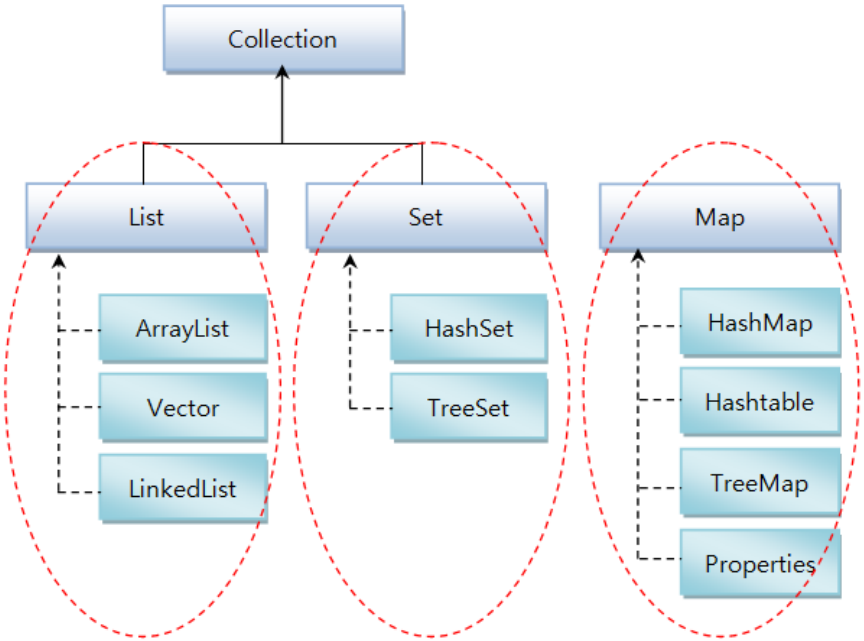
1절. 컬렉션 프레임워크 소개

❖ 컬렉션 프레임워크(Collection Framework)

- 객체들을 효율적으로 추가, 삭제, 검색할 수 있도록 제공되는 컬렉션 라이브러리
- java.util 패키지에 포함
- 인터페이스를 통해서 정형화된 방법으로 다양한 컬렉션 클래스 이용

1절. 컬렉션 프레임워크 소개

❖ 컬렉션 프레임워크의 주요 인터페이스



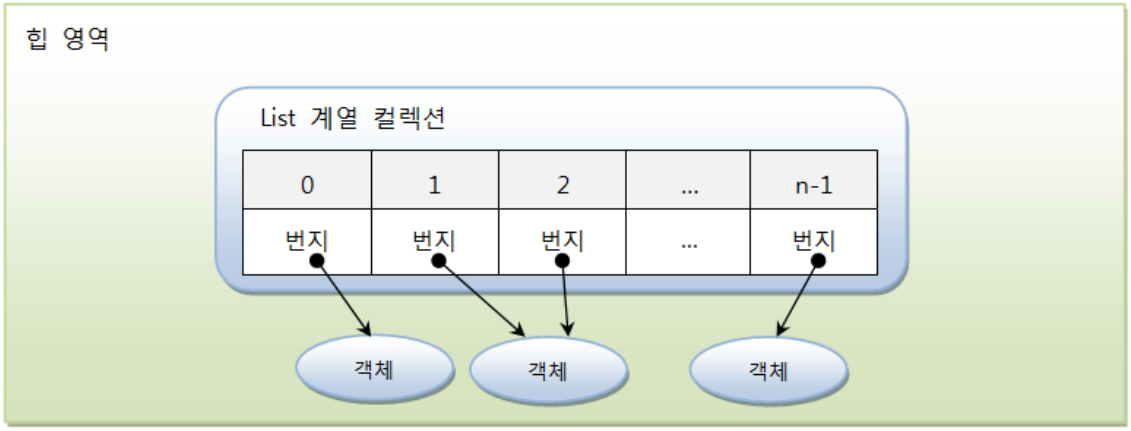
인터페이스 분류		특징	구현 클래스
Collection	List 계열	- 순서를 유지하고 저장 - 중복 저장 가능	ArrayList, Vector, LinkedList
	Set 계열	- 순서를 유지하지 않고 저장 - 중복 저장 안됨	HashSet, TreeSet
Map 계열		- 키와 값의 쌍으로 저장 - 키는 중복 저장 안됨	HashMap, Hashtable, TreeMap, Properties

2절. List 컬렉션

❖ List 컬렉션의 특징 및 주요 메소드

- 특징
  - 인덱스로 관리
  - 중복해서 객체 저장 가능

- 구현 클래스
  - ArrayList
  - Vector
  - LinkedList



2절. List 컬렉션

❖ List 컬렉션의 특징 및 주요 메소드

▪ 주요 메소드

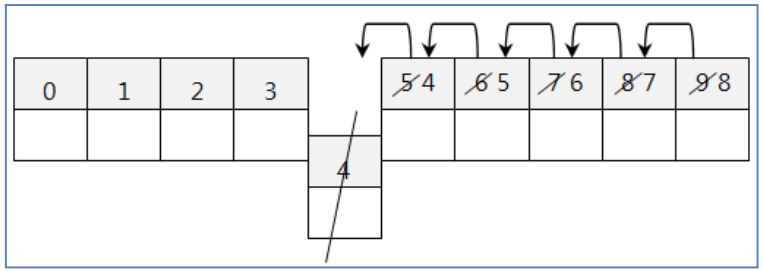
기능	메소드	설명
객체 추가	boolean add(E e)	주어진 객체를 맨끝에 추가
	void add(int index, E element)	주어진 인덱스에 객체를 추가
	set(int index, E element)	주어진 인덱스에 저장된 객체를 주어진 객체로 바꿈
객체 검색	boolean contains(Object o)	주어진 객체가 저장되어 있는지 여부
	E get(int index)	주어진 인덱스에 저장된 객체를 리턴
	isEmpty()	컬렉션이 비어 있는지 조사
	int size()	저장되어있는 전체 객체수를 리턴
객체 삭제	void clear()	저장된 모든 객체를 삭제
	E remove(int index)	주어진 인덱스에 저장된 객체를 삭제
	boolean remove(Object o)	주어진 객체를 삭제

2절. List 컬렉션

❖ ArrayList

- 저장 용량(capacity)
  - 초기 용량 : 10 (따로 지정 가능)
  - 저장 용량을 초과한 객체들이 들어오면 자동적으로 늘어남. 고정도 가능

- 객체 제거
  - 바로 뒤 인덱스부터 마지막 인덱스까지 모두 앞으로 1씩 당겨짐

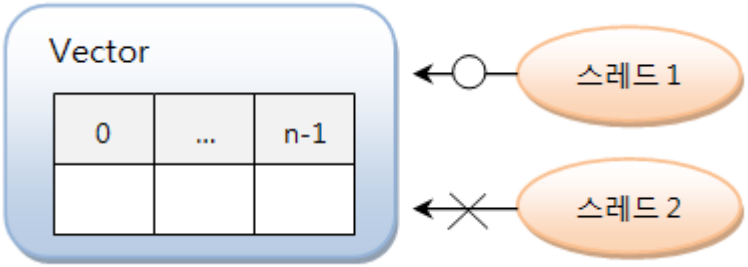


2절. List 컬렉션

❖ Vector

```
List<E> list = new Vector<E>();
```

- ArrayList와 동일한 내부 구조를 가짐
- 특징
  - Vector는 동기화(synchronization)된 메소드로 구성
    - 복수의 스레드가 동시에 Vector에 접근할 수 없음
    - 멀티 스레드 환경에서 안전하게 객체를 추가, 삭제 가능

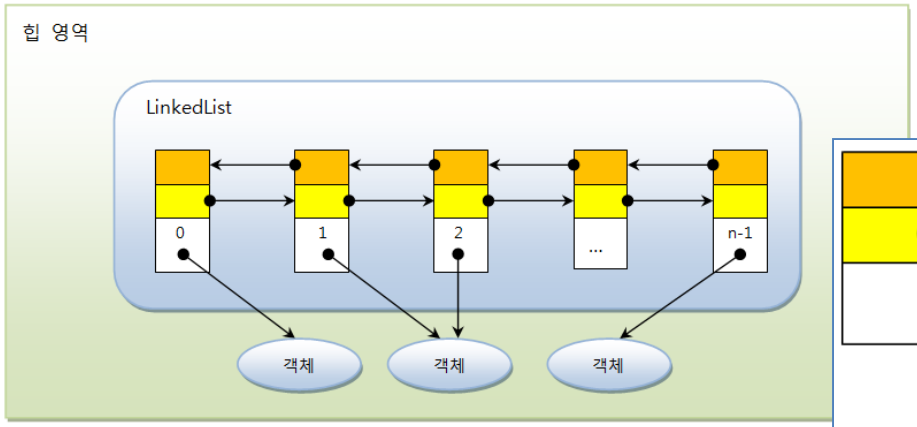


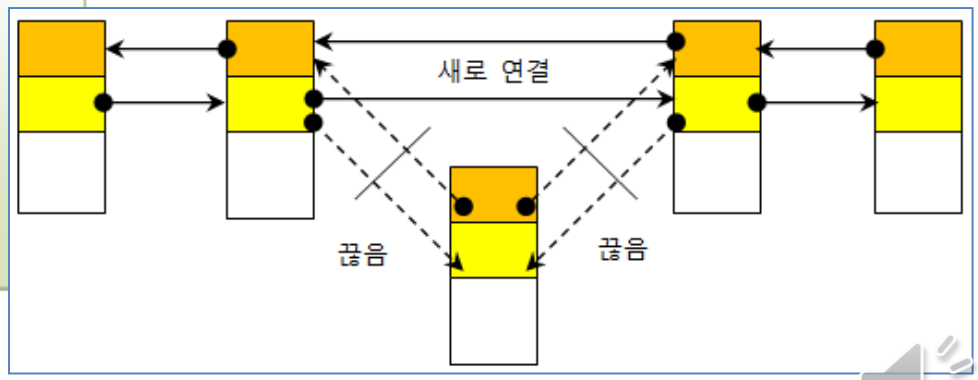
2절. List 컬렉션

❖ LinkedList

```
List<E> list = new LinkedList<E>();
```

- 특징
  - 인접 참조를 링크해서 체인처럼 관리
  - 특정 인덱스에서 객체를 제거하거나 추가하게 되면 바로 앞뒤 링크만 변경
  - 빈번한 객체 삭제와 삽입이 일어나는 곳에서는 ArrayList보다 좋은 성능





3절. Set 컬렉션

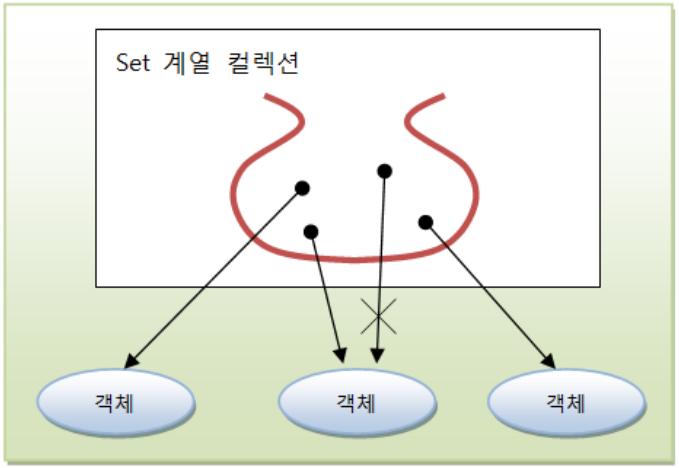
❖ Set 컬렉션의 특징 및 주요 메소드

▪ 특징

- 수학의 집합에 비유
- 저장 순서가 유지되지 않음
- 객체를 중복 저장 불가
- 하나의 null만 저장 가능

▪ 구현 클래스

- HashSet, LinkedHashSet, TreeSet



3절. Set 컬렉션

❖ Set 컬렉션의 특징 및 주요 메소드

▪ 주요 메소드

기능	메소드	설명
객체 추가	boolean add(E e)	주어진 객체를 저장, 객체가 성공적으로 저장되면 true 를 리턴하고 중복 객체면 false 를 리턴
객체 검색	boolean contains(Object o)	주어진 객체가 저장되어 있는지 여부
	isEmpty()	컬렉션이 비어 있는지 조사
	Iterator<E> iterator()	저장된 객체를 한번씩 가져오는 반복자 리턴
	int size()	저장되어있는 전체 객체수 리턴
객체 삭제	void clear()	저장된 모든 객체를 삭제
	boolean remove(Object o)	주어진 객체를 삭제

▪ 전체 객체 대상으로 한 번씩 반복해 가져오는 반복자(Iterator) 제공

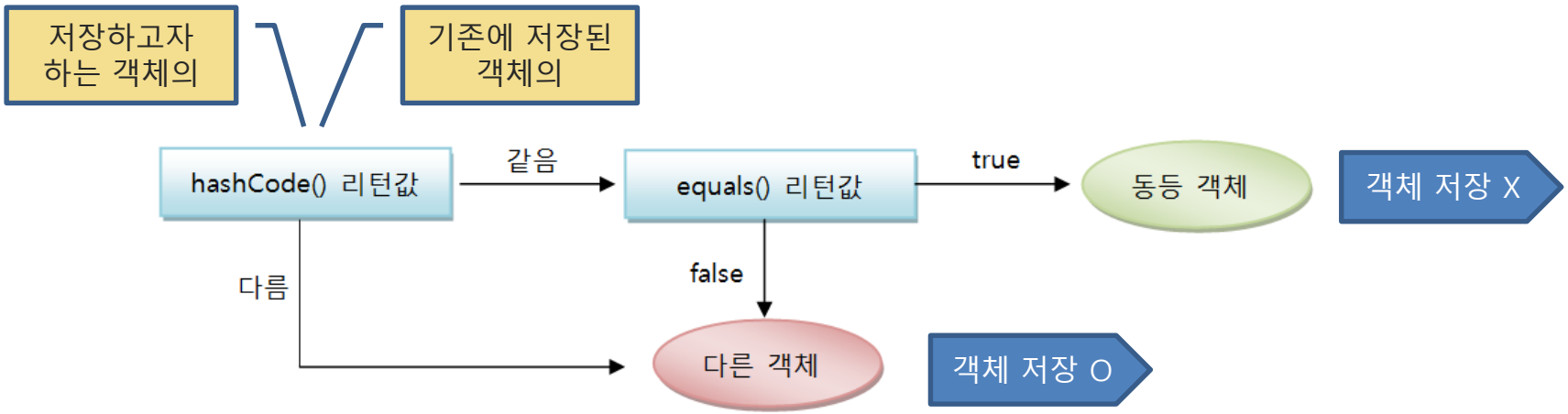
- 인덱스로 객체를 검색해서 가져오는 메소드 없음

3절. Set 컬렉션

❖ HashSet

```
Set<E> set = new HashSet<E>();
```

- **특징**
  - **동일 객체 및 동등 객체는 중복 저장하지 않음**
  - **동등 객체 판단 방법**

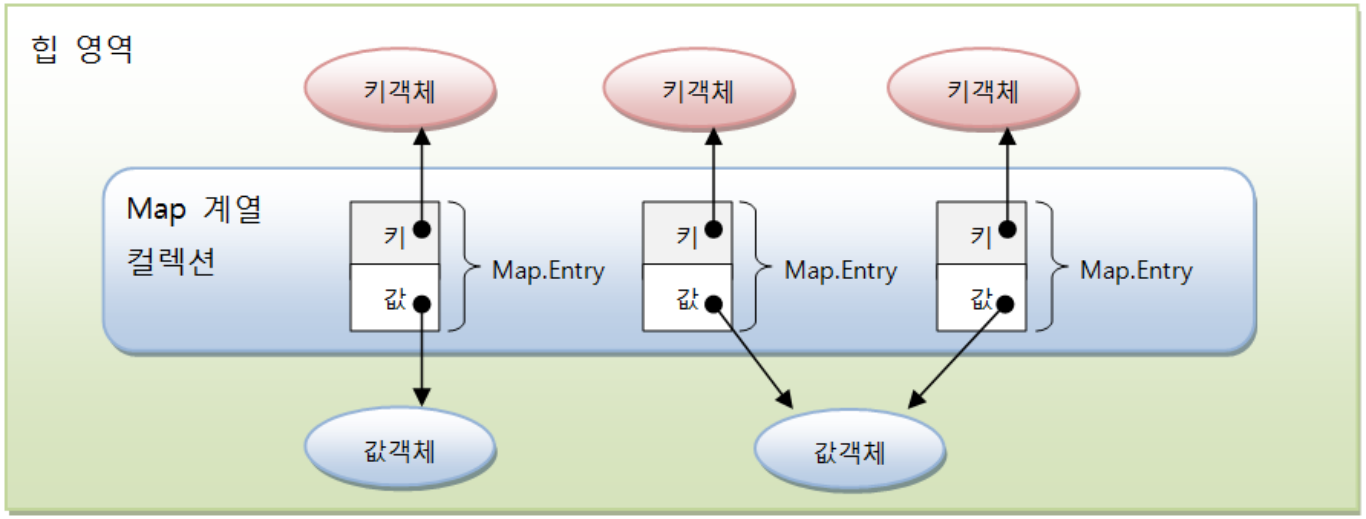


4절. Map 컬렉션

❖ Map 컬렉션의 특징 및 주요 메소드

- 특징
  - 키(key)와 값(value)으로 구성된 Map.Entry 객체를 저장하는 구조
  - 키와 값은 모두 객체
  - 키는 중복될 수 없지만 값은 중복 저장 가능

- 구현 클래스
  - HashMap, Hashtable, LinkedHashMap, Properties, TreeMap



4절. Map 컬렉션

❖ Map 컬렉션의 특징 및 주요 메소드

▪ 주요 메소드

기능	메소드	설명
객체 추가	V put(K key, V value)	주어진 키와 값을 추가, 저장이 되면 값을 리턴
객체 검색	boolean containsKey(Object key)	주어진 키가 있는지 여부
	boolean containsValue(Object value)	주어진 값이 있는지 여부
	Set<Map.Entry<K,V>> entrySet()	키와 값의 쌍으로 구성된 모든 Map.Entry 객체를 Set 에 담아서 리턴
	V get(Object key)	주어진 키의 값을 리턴
	boolean isEmpty()	컬렉션이 비어있는지 여부
	Set<K> keySet()	모든 키를 Set 객체에 담아서 리턴
	int size()	저장된 키의 총 수를 리턴
	Collection<V> values()	저장된 모든 값 Collection 에 담아서 리턴
객체 삭제	void clear()	모든 Map.Entry(키와 값)를 삭제
	V remove(Object key)	주어진 키와 일치하는 Map.Entry 삭제, 삭제가 되면 값을 리턴

4절. Map 컬렉션

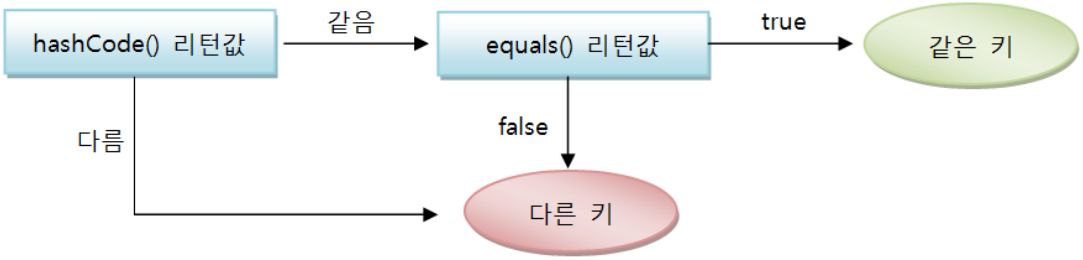
❖ HashMap

▪ 특징

```
Map<K, V> map = new HashMap<K, V>();
```

키 타입 — 값 타입 — 키 타입 — 값 타입

- 키 객체는 hashCode()와 equals() 를 재정의해 동등 객체가 될 조건을 정해야



- 키 타입은 String 많이 사용
  - String은 문자열이 같을 경우 동등 객체가 될 수 있도록 hashCode()와 equals() 메소드가 재정의되어 있기 때문

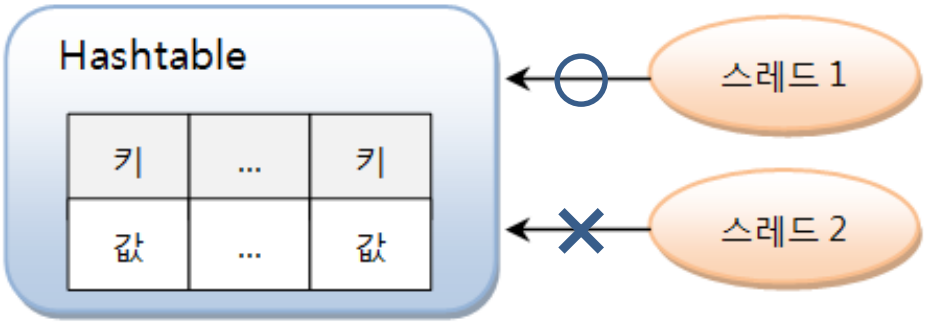
4절. Map 컬렉션

❖ Hashtable

```
Map<K, V> map = new Hashtable<K, V>();
```

키 타입 — 값 타입 — 키 타입 — 값 타입

- 특징
  - 키 객체 만드는 법은 HashMap과 동일
  - Hashtable은 스레드 동기화(synchronization)가 된 상태
    - 복수의 스레드가 동시에 Hashtable에 접근해서 객체를 추가, 삭제 하더라도 스레드에 안전(thread safe)



스레드 동기화 적용됨

과제물

설명이 포함된 소스코드와 실행 결과창을 화면 캡쳐 후 제출
과제 제출 파일명 : 학번+이름

❖ 다음 소스 코드에서 각 빈 칸에 알맞은 설명을 쓰시오.

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListEx {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        int idx = 0;

        //
        list.add("Java");
        list.add("Database");
        list.add("객체지향프로그래밍");

        //
        int size = list.size();
        System.out.println("총 객체수 : " + size);
        System.out.println();

        //
        String item = list.get(2);
        System.out.println("2 : " + item);
        System.out.println();

        //
        for (String s : list)
            System.out.println(idx++ + " : " + s);

        //
        list.remove(1);
        list.remove(1);
        System.out.println();

        //
        idx = 0;
        for (String s : list)
            System.out.println(idx++ + " : " + s);

    }
}
```