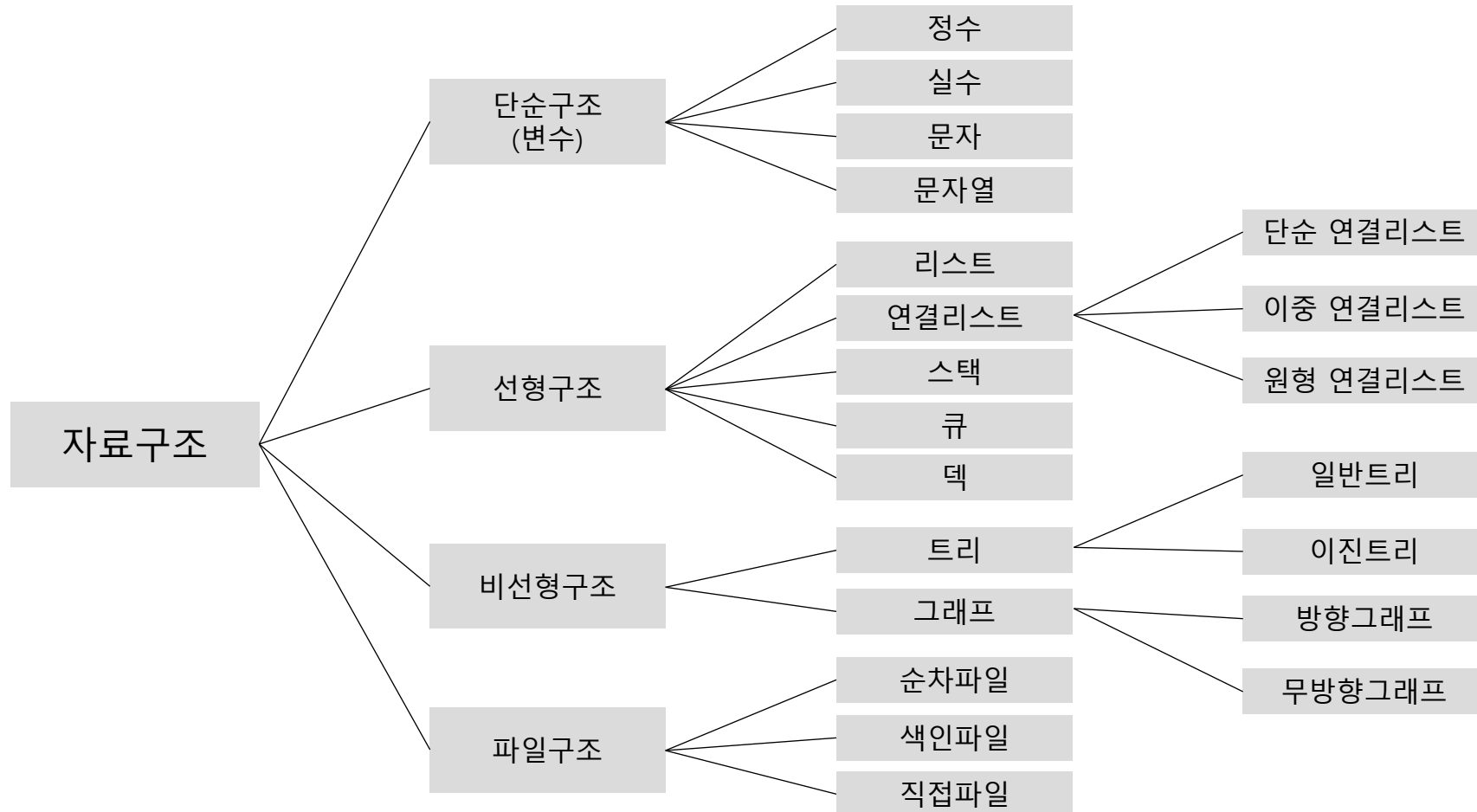


컬렉션 (Collection)

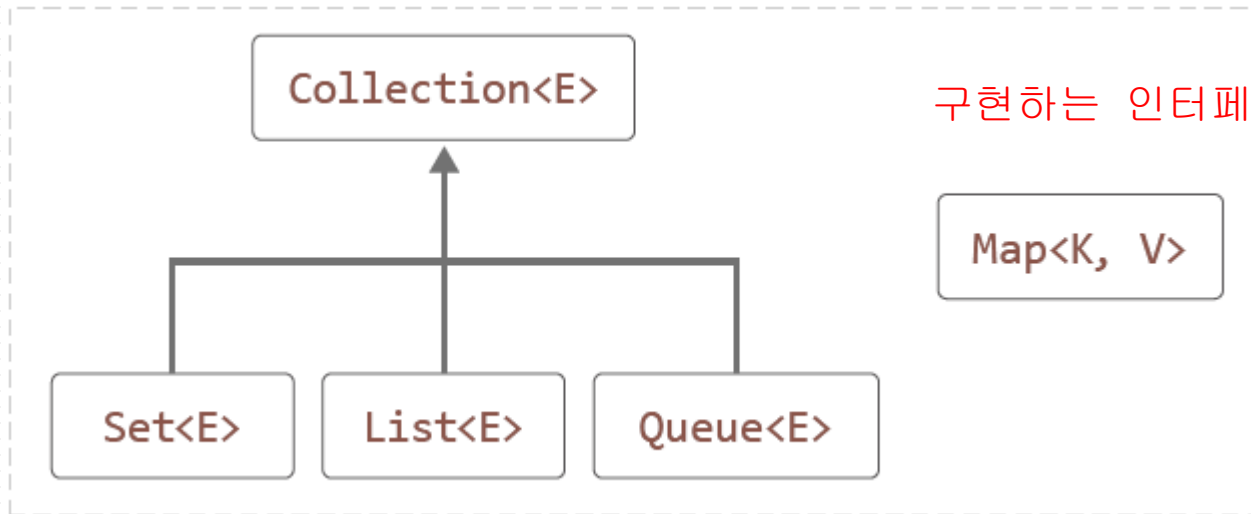
▶ 자료구조

데이터(자료)를 메모리에서 구조적으로 처리하는 방법론이다.



컬렉션 프레임워크

컬렉션 프레임워크의 골격에 해당하는 인터페이스들



구현하는 인터페이스에 따라 사용방법과 특성이 결정된다.

자료구조 및 알고리즘을 구현해 놓은 일종의 라이브러리!
제네릭 기반으로 구현이 되어 있다.

▶ 배열의 문제점 & 컬렉션의 장점

✓ 배열의 문제점

1. 한 번 크기를 지정하면 변경할 수 없다.

- 공간 크기가 부족하면 예러가 발생 → 할당 시 넉넉한 크기로 할당하게 됨 (메모리 낭비)
- 필요에 따라 공간을 늘리거나 줄일 수 없음

2. 배열에 기록된 데이터에 대한 중간 위치의 추가, 삭제가 불편하다.

- 추가, 삭제할 데이터부터 마지막 기록된 데이터까지 하나씩 뒤로 밀어내고 추가해야 함
(복잡한 알고리즘)

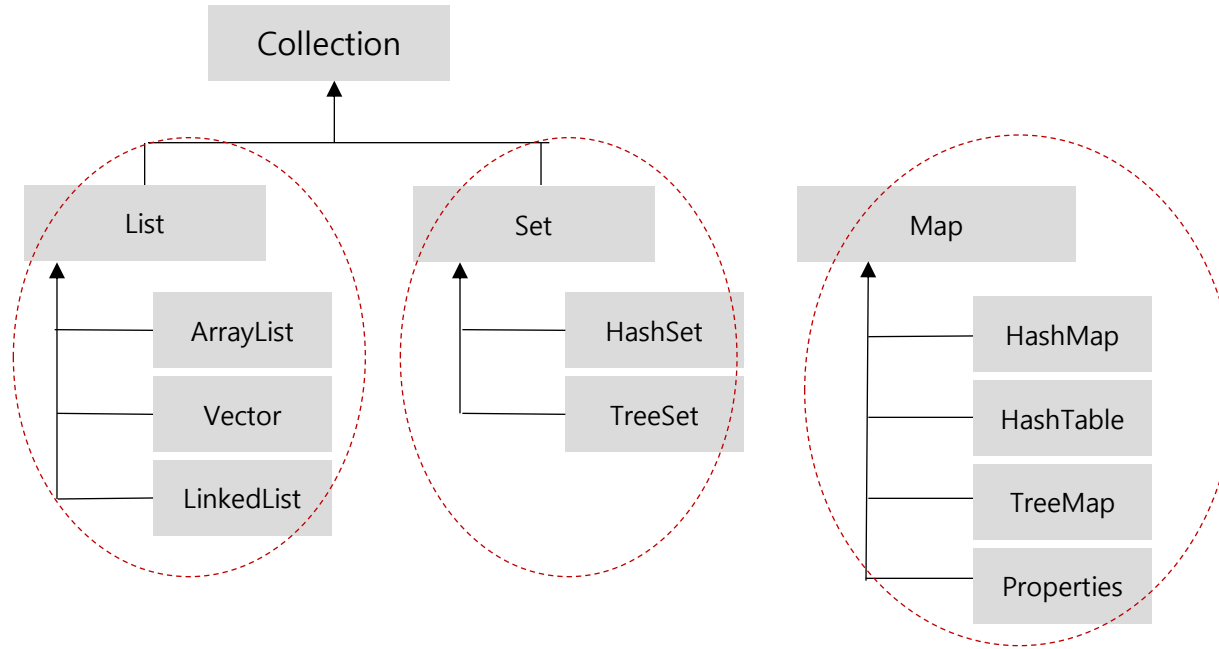
3. 한 타입의 데이터만 저장 가능하다.

▶ 배열의 문제점 & 컬렉션의 장점

✓ 컬렉션의 장점

1. 저장하는 크기의 제약이 없다.
2. 추가, 삭제, 정렬 등의 기능 처리가 간단하게 해결된다.
 - 자료를 구조적으로 처리 하는 자료구조가 내장되어 있어 알고리즘 구현이 필요 없음
3. 여러 타입의 데이터가 저장 가능하다.
 - 객체만 저장할 수 있기 때문에 필요에 따라 기본 자료형을 저장해야 하는 경우
Wrapper클래스 사용

▶ 컬렉션의 주요 인터페이스



인터페이스 분류		특징	구현 클래스
Collection	List 계열	<ul style="list-style-type: none"> - 순서를 유지하고 저장 - 중복 저장 가능 	ArrayList, Vector, LinkedList
	Set계열	<ul style="list-style-type: none"> - 순서를 유지하지 않고 저장 - 중복 저장 안됨 	HashSet, TreeSet
Map 계열		<ul style="list-style-type: none"> - 키와 값의 쌍으로 저장 - 키는 중복 저장 안됨 	HashMap, Hashtable, TreeMap, Properties

List<E> 인터페이스를 구현하는 컬렉션 클래스들

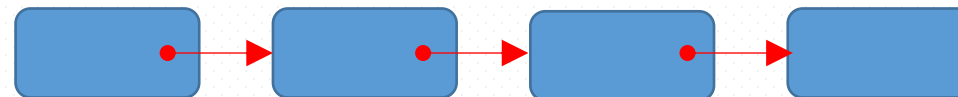
List<E> 인터페이스

List<E> 인터페이스를 구현하는 대표적인 컬렉션 클래스 둘은 다음과 같다.

- `ArrayList<E>` 배열 기반 자료구조, 배열을 이용하여 인스턴스 저장
- `LinkedList<E>` 리스트 기반 자료구조, 리스트를 구성하여 인스턴스 저장

List<E> 인터페이스를 구현하는 컬렉션 클래스들의 공통 특성

- 인스턴스의 저장 순서 유지
- 동일 인스턴스의 중복 저장을 허용한다.



리스트 자료구조의 구성

ArrayList<E> 클래스

```
public static void main(String[] args) {  
    List<String> list = new ArrayList<>(); // 컬렉션 인스턴스 생성
```

```
    // 컬렉션 인스턴스에 문자열 인스턴스 저장  
    list.add("Toy");  
    list.add("Box");  
    list.add("Robot");
```

```
    // 저장된 문자열 인스턴스의 참조  
    for(int i = 0; i < list.size(); i++)  
        System.out.print(list.get(i) + '\t');  
    System.out.println();
```

```
    list.remove(0); // 첫 번째 인스턴스 삭제
```

```
    // 첫 번째 인스턴스 삭제 후 나머지 인스턴스들을 참조  
    for(int i = 0; i < list.size(); i++)  
        System.out.print(list.get(i) + '\t');  
    System.out.println();
```

```
}
```

배열 기반 자료구조이지만 공간의 확보 및 확장은
ArrayList 인스턴스가 스스로 처리한다.

```
명령 프롬프트  
C:\JavaStudy>java ArrayListCollection  
Toy      Box      Robot  
Box      Robot  
C:\JavaStudy>
```

LinkedList<E> 클래스

```
public static void main(String[] args) {  
    List<String> list = new LinkedList<>();    // 유일한 변화!!!
```

```
    // 컬렉션 인스턴스에 문자열 인스턴스 저장  
    list.add("Toy");  
    list.add("Box");  
    list.add("Robot");
```

```
    // 저장된 문자열 인스턴스의 참조  
    for(int i = 0; i < list.size(); i++)  
        System.out.print(list.get(i) + '\t');  
    System.out.println();
```

```
    list.remove(0); // 첫 번째 인스턴스 삭제
```

```
    // 첫 번째 인스턴스 삭제 후 나머지 인스턴스들을 참조  
    for(int i = 0; i < list.size(); i++)  
        System.out.print(list.get(i) + '\t');  
    System.out.println();
```

```
}
```

리스트 기반 자료구조는 열차 칸을 더하고 빼는 형태의
자료구조이다.

인스턴스 저장

인스턴스 삭제

열차 칸을 하나 더한다.

해당 열차 칸을 삭제한다.

ArrayList<E> vs. LinkedList<E>

ArrayList<E>의 단점

- 저장 공간을 늘리는 과정에서 시간이 비교적 많이 소요된다.
- 인스턴스의 삭제 과정에서 많은 연산이 필요할 수 있다. 따라서 느릴 수 있다.

ArrayList<E>의 장점

- 저장된 인스턴스의 참조가 빠르다.

LinkedList<E>의 단점

- 저장된 인스턴스의 참조 과정이 배열에 비해 복잡하다. 따라서 느릴 수 있다.

LinkedList<E>의 장점

- 저장 공간을 늘리는 과정이 간단하다.
- 저장된 인스턴스의 삭제 과정이 단순하다.

▶ List

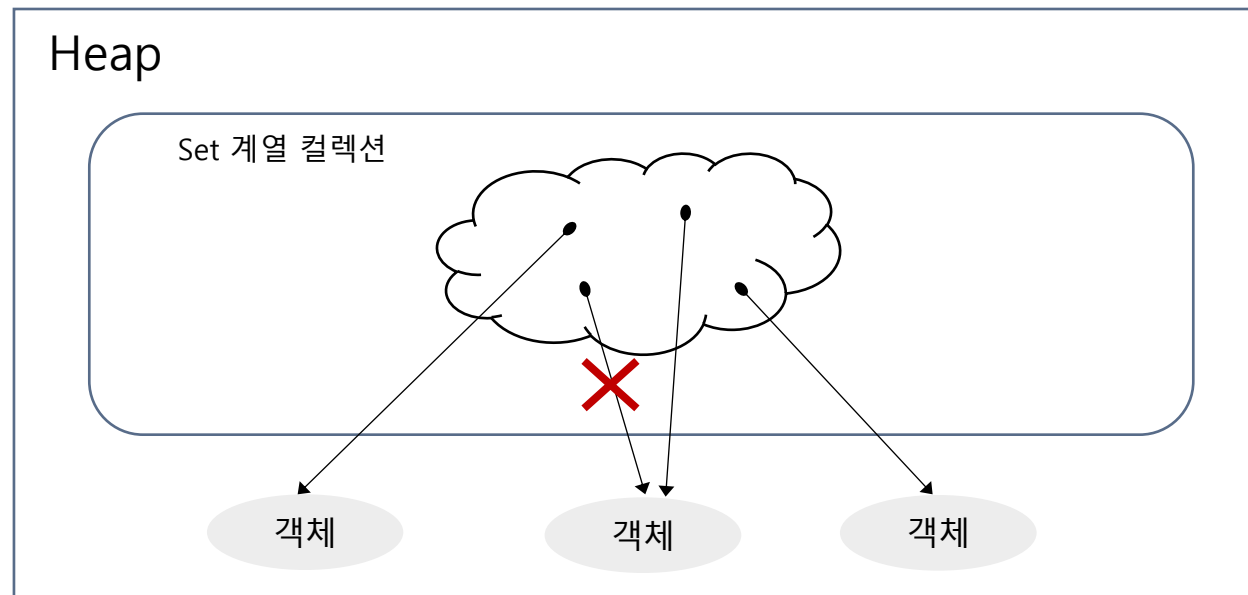
✓ List 계열 주요 메소드

기능	메소드	리턴타입	설명
객체 추가	add(E e)	boolean	주어진 객체를 맨 끝에 추가
	add(int index, E element)	void	주어진 인덱스에 객체를 추가
	addAll(Collection<? extends E> c)	boolean	주어진 Collection타입 객체를 리스트에 추가
	set(int index, E element)	E	주어진 인덱스에 저장된 객체를 주어진 객체로 바꿈
객체 검색	contains(Object o)	boolean	주어진 객체가 저장되어 있는지 여부
	get(int index)	E	주어진 인덱스에 저장된 객체를 리턴
	iterator()	Iterator<E>	저장된 객체를 한번씩 가져오는 반복자 리턴
	isEmpty()	boolean	컬렉션이 비어 있는지 조사
	size()	int	저장되어 있는 전체 객체수를 리턴
객체 삭제	clear()	void	저장된 모든 객체를 삭제
	remove(int index)	E	주어진 인덱스에 저장된 객체를 삭제
	remove(Object o)	boolean	주어진 객체를 삭제

Set<E> 인터페이스를 구현하는 컬렉션 클래스들

▶ Set

저장 순서가 유지되지 않고, 중복 객체도 저장하지 못하게 하는 자료 구조
null도 중복을 허용하지 않기 때문에 1개의 null만 저장
구현 클래스로 HashSet, LinkedSet, TreeSet이 있음



▶ Set

✓ HashSet

Set에 객체를 저장할 때 hash함수를 사용하여 처리 속도가 빠름
동일 객체 뿐 아니라 동등 객체도 중복하여 저장하지 않음

✓ LinkedHashSet


HashSet과 거의 동일하지만 Set에 추가되는 순서를 유지한다는 점이 다름

Set<E>을 구현하는 클래스의 특성과 HashSet<E> 클래스

Set<E> 인터페이스를 구현하는 제네릭 클래스들은 다음 두 가지 특성을 갖는다.

- 저장 순서가 유지되지 않는다.
- 데이터의 중복 저장을 허용하지 않는다.

```
public static void main(String[] args) {  
    Set<String> set = new HashSet<>();  
    set.add("Toy");    set.add("Box");  
    set.add("Robot");    set.add("Box");  
    System.out.println("인스턴스 수: " + set.size());  
  
    // 반복자를 이용한 전체 출력  
    for(Iterator<String> itr = set.iterator(); itr.hasNext(); )  
        System.out.print(itr.next() + '\t');  
    System.out.println();  
  
    // for-each문을 이용한 전체 출력  
    for(String s : set)  
        System.out.print(s + '\t');  
    System.out.println();  
}
```



```
CA. 명령 프롬프트  
C:\JavaStudy>java SetCollectionFeature  
인스턴스 수: 3  
Box    Robot    Toy  
Box    Robot    Toy  
  
C:\JavaStudy>
```

출력 결과를 통해 동일 인스턴스가 저장되지 않음을 알 수 있다.

그렇다면 동일 인스턴스의 기준은?

해쉬 알고리즘의 이해

분류 대상: 3, 5, 7, 12, 25, 31

적용 해쉬 알고리즘: $\text{num} \% 3$

이렇듯 분류를 해 놓으면 탐색의 속도가 매우 빨라진다.
즉 존재 유무 확인이 매우 빠르다.

분류 결과:



`Object` 클래스의 `hashCode` 메소드는 이렇듯 인스턴스들을 분류하는 역할을 한다.

동일 인스턴스에 대한 기준은?

```
public boolean equals(Object obj)
```

`Object` 클래스의 `equals` 메소드 호출 결과를 근거로 동일 인스턴스를 판단한다.

```
public int hashCode()
```

그런데 그에 앞서 `Object` 클래스의 `hashCode` 메소드 호출 결과가 같아야 한다.

정리하면,

두 인스턴스가 `hashCode` 메소드 호출 결과로 반환하는 값이 동일해야 한다.

그리고 이어서 두 인스턴스를 대상으로 `equals` 메소드의 호출 결과 `true`가 반환되면 동일 인스턴스로 간주한다.

HashSet<E> 인스턴스에 저장할 클래스 정의 예

```
class Num {  
    private int num;  
    public Num(int n) { num = n; }  
  
    @Override  
    public String toString() { return String.valueOf(num); }  
  
    @Override  
    public int hashCode() {  
        return num % 3; // num의 값이 같으면 부류도 같다.  
    }  
  
    @Override  
    public boolean equals(Object obj) { // num의 값이 같으면 true 반환  
        if(num == ((Num)obj).num)  
            return true;  
        else  
            return false;  
    }  
}
```

hashCode 메소드의 다양한 정의의 예

```
class Car {  
    private String model;  
    private String color;  
    . . . .  
  
    @Override  
    public int hashCode() {  
        return (model.hashCode() + color.hashCode()) / 2;  
    }  
    . . . .  
}
```

모든 인스턴스 변수의 정보를 다 반영하여 해쉬 값을 얻으려는 노력이 깃든 문장.

결과적으로 더 세밀하게 나뉘고, 따라서 그만큼 탐색 속도가 높아진다.

해쉬 알고리즘 일일이 정의하기 조금 그렇다면...

```
public static int hash(Object...values)
```

→ `java.util.Objects`에 정의된 메소드, 전달된 인자 기반의 해쉬 값 반환

```
@Override
```

```
public int hashCode() {
```

```
    return Objects.hash(model, color);    // 전달인자 model, color 기반 해쉬 값 반환
```

```
}        전달된 인자를 모두 반영한 해쉬 값을 반환한다.
```

▶ Set

✓ Set 계열 주요 메소드

기능	메소드	리턴타입	설명
객체 추가	add(E e)	boolean	주어진 객체를 맨 끝에 추가
	addAll(Collection<? extends E> c)	boolean	주어진 Collection타입 객체를 리스트에 추가
객체 검색	contains(Object o)	boolean	주어진 객체가 저장되어 있는지 여부
	iterator()	Iterator<E>	저장된 객체를 한번씩 가져오는 반복자 리턴
	isEmpty()	boolean	컬렉션이 비어 있는지 조사
	size()	int	저장되어 있는 전체 객체수를 리턴
객체 삭제	clear()	void	저장된 모든 객체를 삭제
	remove(Object o)	boolean	주어진 객체를 삭제

* 전체 객체 대상으로 한 번씩 반복해서 가져오는 반복자(Iterator)를 제공 인덱스로 객체에 접근할 수 없음

Map<K, V> 인터페이스를
구현하는 컬렉션 클래스들

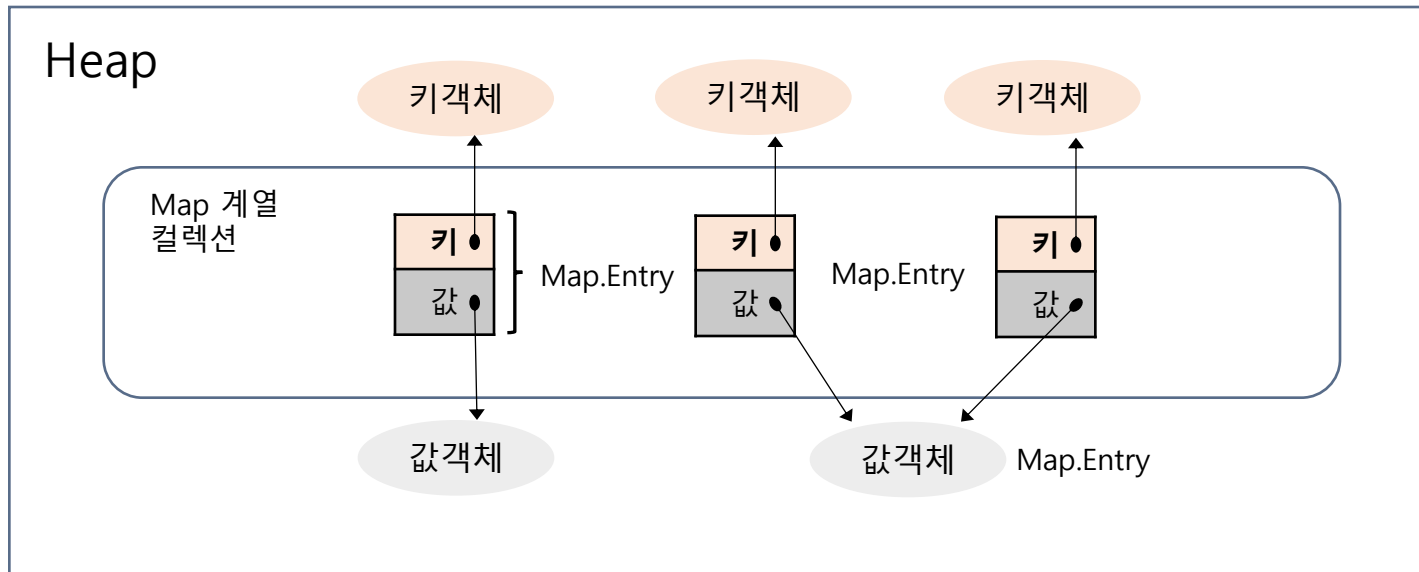
▶ Map

키(key)와 값(value)으로 구성되어 있으며, 키와 값은 모두 객체

키는 중복 저장을 허용하지 않고(Set방식), 값은 중복 저장 가능(List방식)

키가 중복되는 경우, 기존에 있는 키에 해당하는 값을 덮어 씌움

구현 클래스로 HashMap, Hashtable, LinkedHashMap, Properties, TreeMap이 있음



▶ Map

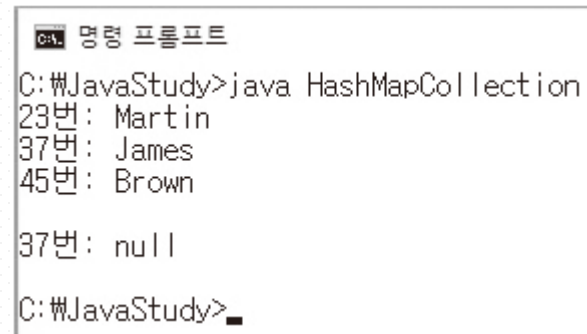
✓ HashMap

키 객체는 hashCode()와 equals()를 재정의해 동등 객체가 될 조건을 정해야 함
때문에 키 타입은 hashCode()와 equals()메소드가 재정의되어 있는 String타입을 주로 사용

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

Key-Value 방식의 데이터 저장과 HashMap<K, V> 클래스

```
public static void main(String[] args) {  
    HashMap<Integer, String> map = new HashMap<>();  
  
    // Key-Value 기반 데이터 저장  
    map.put(45, "Brown");  
    map.put(37, "James");  
    map.put(23, "Martin");  
  
    // 데이터 탐색  
    System.out.println("23번: " + map.get(23));  
    System.out.println("37번: " + map.get(37));  
    System.out.println("45번: " + map.get(45));  
    System.out.println();  
  
    // 데이터 삭제  
    map.remove(37);  
  
    // 데이터 삭제 확인  
    System.out.println("37번: " + map.get(37));  
}
```



```
명령 프롬프트  
C:\JavaStudy>java HashMapCollection  
23번: Martin  
37번: James  
45번: Brown  
  
37번: null  
C:\JavaStudy>
```

HashMap<K, V>의 순차적 접근의 예

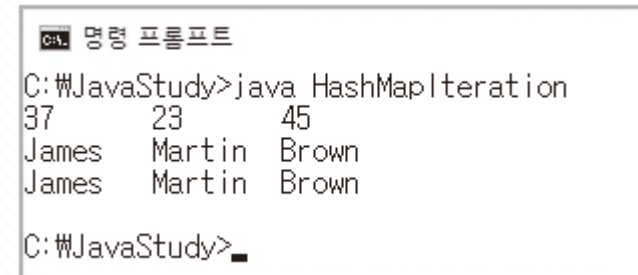
```
public static void main(String[] args) {
    HashMap<Integer, String> map = new HashMap<>();
    map.put(45, "Brown");
    map.put(37, "James");
    map.put(23, "Martin");

    // Key만 담고 있는 컬렉션 인스턴스 생성
    Set<Integer> ks = map.keySet();

    // 전체 Key 출력 (for-each문 기반)
    for(Integer n : ks)
        System.out.print(n.toString() + '\t');
    System.out.println();

    // 전체 Value 출력 (for-each문 기반)
    for(Integer n : ks)
        System.out.print(map.get(n).toString() + '\t');
    System.out.println();

    // 전체 Value 출력 (반복자 기반)
    for(Iterator<Integer> itr = ks.iterator(); itr.hasNext(); )
        System.out.print(map.get(itr.next()) + '\t');
    System.out.println();
}
```



```
명령 프롬프트
C:\JavaStudy>java HashMapIteration
37      23      45
James   Martin  Brown
James   Martin  Brown
C:\JavaStudy>
```

▶ Map

✓ Map 계열 주요 메소드

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