7. 컬렉션 프레임워크 - HashSet

#0.강의/1.자바로드맵/4.자바-중급2편

직접 구현하는 Set1 - MyHashSetV1

지금까지 학습한 내용을 기반으로 해시 알고리즘을 사용해서 Set 자료구조를 다시 구현해보자.
그 전에 Set의 정의를 다시 한번 복습해보자.

Set은 중복을 허용하지 않고, 순서를 보장하지 않는 자료 구조이다.

이전에 구현한 성능이 O(n)으로 느린 `MyHashSetV0`를 다시 한번 확인해보자.
MyHashSetV0의 단점

- `add()`로 데이터를 추가할 때 셋에 중복 데이터가 있는지 전체 데이터를 항상 확인해야 한다. 따라서 O(n)으로 입력 성능이 나쁘다. 중복 데이터가 있는지 검색 O(n) + 데이터 입력 O(1) ➜ O(n)
- `contains()`로 데이터를 찾을 때는 셋에 있는 모든 데이터를 찾고 비교해야 하므로 평균 O(n)이 걸린다.

MyHashSetV0의 문제는 데이터를 추가할 때 중복 데이터가 있는지 체크하는 부분에서 성능이 O(n)으로 좋지 않다는 점이다. 왜냐하면 이때 중복 데이터가 있는지 모든 데이터를 다 찾아봐야 하기 때문이다.
물론 데이터를 찾을 때도 모두 순서대로 전체 데이터를 확인해야 하므로 평균 성능이 O(n)으로 좋지 않다.
이렇게 성능이 느린 `MyHashSetV0`를 해시 알고리즘을 사용해서 평균 O(1)로 개선해보자.

`Set`을 구현하는 방법은 단순하다. 인덱스가 없기 때문에 단순히 데이터를 저장하고, 데이터가 있는지 확인하고, 데이터를 삭제하는 정도면 충분하다. 그리고 `Set`은 중복을 허용하지 않기 때문에 데이터를 추가할 때 중복 여부만 체크하면 된다.

- `add(value)` : 셋에 값을 추가한다. 중복 데이터는 저장하지 않는다.
- `contains(value)` : 셋에 값이 있는지 확인한다.

- `remove(value)` : 셋에 있는 값을 제거한다.

해시 알고리즘을 사용하도록 개선된 MyHashSetV1

```
package collection.set;

import java.util.Arrays;
import java.util.LinkedList;

public class MyHashSetV1 {

    static final int DEFAULT_INITIAL_CAPACITY = 16;

    LinkedList<Integer>[] buckets;

    private int size = 0;
    private int capacity = DEFAULT_INITIAL_CAPACITY;

    public MyHashSetV1() {
        initBuckets();
    }

    public MyHashSetV1(int capacity) {
        this.capacity = capacity;
        initBuckets();
    }

    private void initBuckets() {
        buckets = new LinkedList[capacity];
        for (int i = 0; i < capacity; i++) {
            buckets[i] = new LinkedList<>();
        }
    }

    public boolean add(int value) {
        int hashIndex = hashIndex(value);
        LinkedList<Integer> bucket = buckets[hashIndex];
        if (bucket.contains(value)) {
            return false;
        }

        bucket.add(value);
        size++;
```

```
            return true;
        }

        public boolean contains(int searchValue) {
            int hashIndex = hashIndex(searchValue);
            LinkedList<Integer> bucket = buckets[hashIndex];
            return bucket.contains(searchValue);
        }

        public boolean remove(int value) {
            int hashIndex = hashIndex(value);
            LinkedList<Integer> bucket = buckets[hashIndex];
            boolean result = bucket.remove(Integer.valueOf(value));
            if (result) {
                size--;
                return true;
            } else {
                return false;
            }
        }

        private int hashIndex(int value) {
            return value % capacity;
        }


        public int getSize() {
            return size;
        }

        @Override
        public String toString() {
            return "MyHashSetV1{" +
                    "buckets=" + Arrays.toString(buckets) +
                    ", size=" + size +
                    '}';
        }
    }
```

- `buckets` : 연결 리스트를 배열로 사용한다.
  - 배열안에 연결 리스트가 들어있고, 연결 리스트 안에 데이터가 저장된다.
  - 해시 인덱스가 충돌이 발생하면 같은 연결 리스트 안에 여러 데이터가 저장된다.
- `initBuckets()`

    - 연결 리스트를 생성해서 배열을 채운다. 배열의 모든 인덱스 위치에는 연결 리스트가 들어있다.
- 초기 배열의 크기를 생성자를 통해서 전달할 수 있다.
    - 기본 생성자를 사용하면 `DEFAULT_INITIAL_CAPACITY`의 값인 16이 사용된다.
- `add()` : 해시 인덱스를 사용해서 데이터를 보관한다.
- `contains()` : 해시 인덱스를 사용해서 데이터를 확인한다.
- `remove()` : 해시 인덱스를 사용해서 데이터를 제거한다.

```
package collection.set;

public class MyHashSetV1Main {

    public static void main(String[] args) {
        MyHashSetV1 set = new MyHashSetV1(10);
        set.add(1);
        set.add(2);
        set.add(5);
        set.add(8);
        set.add(14);
        set.add(99);
        set.add(9); //hashIndex 중복
        System.out.println(set);

        //검색
        int searchValue = 9;
        boolean result = set.contains(searchValue);
        System.out.println("set.contains(" + searchValue + ") = " + result);

        //삭제
        boolean removeResult = set.remove(searchValue);
        System.out.println("removeResult = " + removeResult);
        System.out.println(set);
    }
}
```

실행 결과

```
MyHashSetV1{buckets=[[], [1], [2], [], [14], [5], [], [], [8], [99, 9]],
size=7}
bucket.contains(9) = true
```

```
removeResult = true
MyHashSetV1{buckets=[[], [1], [2], [], [14], [5], [], [], [8], [99]], size=6}
```

이미 해시 알고리즘을 학습할 때 대부분 작성해본 코드를 객체로 변경한 것이어서 이해하는데 크게 어려움은 없을 것이다.

- **생성**: `new MyHashSetV1(10)`을 사용해서 배열의 크기를 10으로 지정했다. (여기서는 기본 생성자를 사용하지 않았다.)
- **저장**: 실행 결과를 보면 `99`, `9`의 경우 해시 인덱스가 9로 충돌하게 된다. 따라서 배열의 같은 9번 인덱스 위치에 저장된 것을 확인할 수 있다. 그리고 그 안에 있는 연결 리스트에 `99`, `9`가 함께 저장된다.
- **검색**: `9`를 검색하는 경우 해시 인덱스가 `9`이다. 따라서 배열의 `9`번 인덱스에 있는 연결 리스트를 먼저 찾는다. 해당 연결 리스트에 있는 모든 데이터를 순서대로 비교하면서 `9`를 찾는다.
  - 먼저 `99`와 `9`를 비교한다. ➜ 실패
  - 다음으로 `9`와 `9`를 비교한다. ➜ 성공

`MyHashSetV1`은 해시 알고리즘을 사용한 덕분에 등록, 검색, 삭제 모두 평균 O(1)로 연산 속도를 크게 개선했다.

남은 문제

해시 인덱스를 사용하려면 데이터의 값을 배열의 인덱스로 사용해야 한다. 그런데 배열의 인덱스는 0, 1, 2 같은 숫자만 사용할 수 있다. "A", "B"와 같은 문자열은 배열의 인덱스로 사용할 수 없다.

다음 예와 같이 숫자가 아닌 문자열 데이터를 저장할 때, 해시 인덱스를 사용하려면 어떻게 해야할까?

예)

```
MyHashSetV1 set = new MyHashSetV1(10);
set.add("A");
set.add("B");
set.add("HELLO");
```

문자열 해시 코드

지금까지 해시 인덱스를 구할 때 숫자를 기반으로 해시 인덱스를 구했다. 해시 인덱스는 배열의 인덱스로 사용해야 하므로 0, 1, 2, 같은 숫자(양의 정수)만 사용할 수 있다. 따라서 문자를 사용할 수 없다.

문자 데이터를 기반으로 숫자 해시 인덱스를 구하려면 어떻게 해야 할까?

다음 코드를 통해 문자를 숫자로 변경하는 방법을 알아보자.

```
package collection.set;

public class StringHashMain {

    static final int CAPACITY = 10;

    public static void main(String[] args) {
        //char
        char charA = 'A';
        char charB = 'B';
        System.out.println(charA + " = " + (int)charA);
        System.out.println(charB + " = " + (int)charB);

        //hashCode
        System.out.println("hashCode(A) = " + hashCode("A"));
        System.out.println("hashCode(B) = " + hashCode("B"));
        System.out.println("hashCode(AB) = " + hashCode("AB"));

        //hashIndex
        System.out.println("hashIndex(A) = " + hashIndex(hashCode("A")));
        System.out.println("hashIndex(B) = " + hashIndex(hashCode("B")));
        System.out.println("hashIndex(AB) = " + hashIndex(hashCode("AB")));
    }

    static int hashCode(String str) {
        char[] charArray = str.toCharArray();
        int sum = 0;
        for (char c : charArray) {
            sum += c;
        }
        return sum;
    }

    static int hashIndex(int value) {
        return value % CAPACITY;
    }
}
```

실행 결과

```
A = 65
B = 66
hash(A) = 65
hash(B) = 66
hash(AB) = 131
hashIndex(A) = 5
hashIndex(B) = 6
hashIndex(AB) = 1
```

모든 문자는 본인만의 고유한 숫자로 표현할 수 있다. 예를 들어서 `'A'`는 `65`, `'B'`는 `66`으로 표현된다. 가장 단순하게 `char`형을 `int`형으로 캐스팅하면 문자의 고유한 숫자를 확인할 수 있다.
그리고 "AB"와 같은 연속된 문자는 각각의 문자를 더하는 방식으로 숫자로 표현하면 된다. 65 + 66 = 131이다.

참고: 컴퓨터는 문자를 직접 이해하지는 못한다. 대신에 각 문자에 고유한 숫자를 할당해서 인식한다. 다음 ASCII 코드 표를 참고하자. 예를 들어 'A'라는 문자는 숫자 65로 'B'라는 문자를 숫자 66으로 인식한다. 그리고 문자를 저장할 때도 이런 표를 사용해서 문자를 숫자로 변환해서 저장한다.
이 부분을 더 자세히 이해하려면 인코딩이라는 개념을 알아야 하는데, 인코딩은 별도로 다룬다. 지금은 모든 문자가 고유한 숫자를 가지고 있다는 것 정도만 알아두자.

ASCII 코드 표

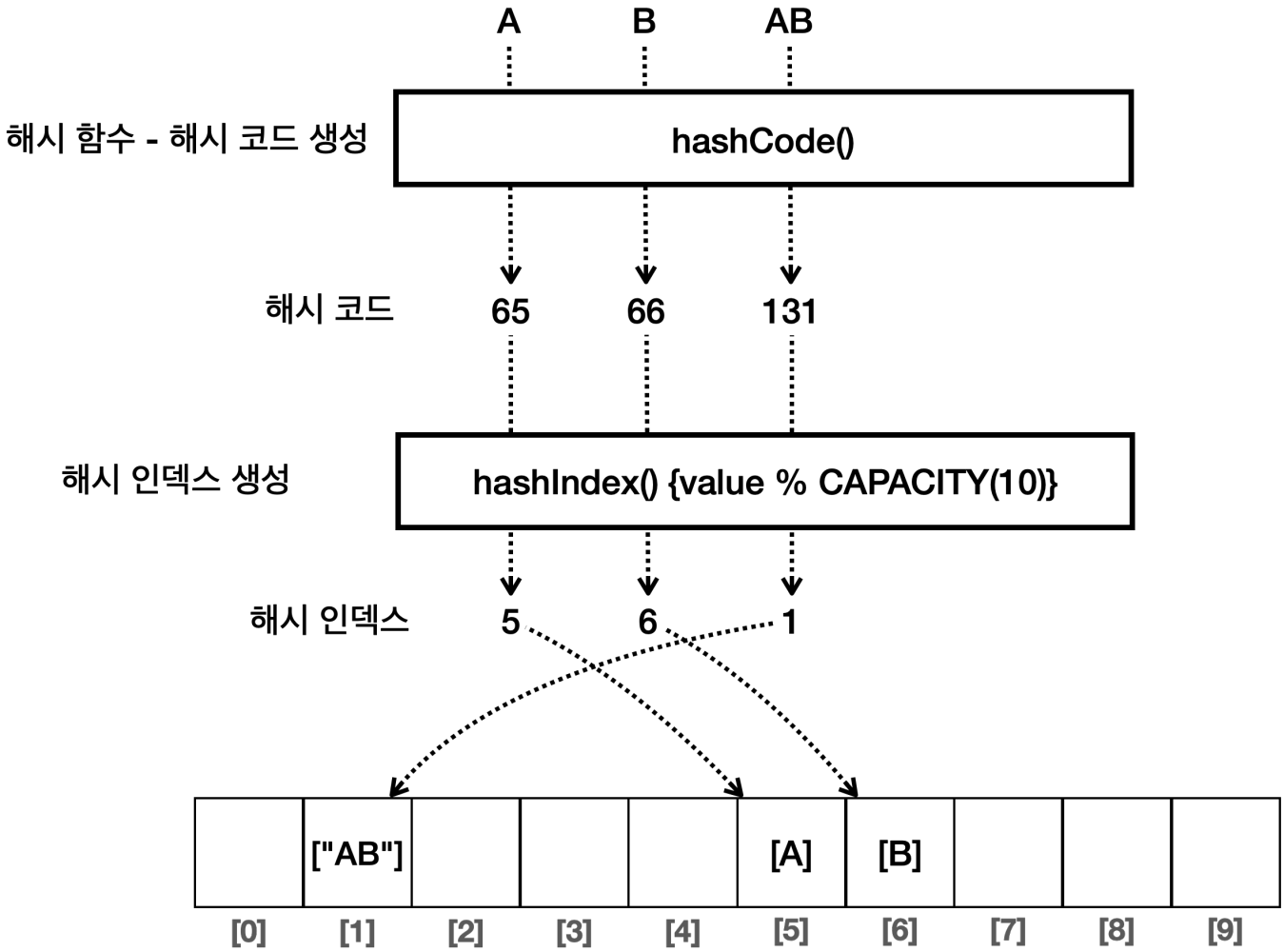
10진수	문자	10진수	문자	10진수	문자
32	(공백)	65	A	97	a
33	!	66	B	98	b
34	"	67	C	99	c
35	#	68	D	100	d
36	$	69	E	101	e
37	%	70	F	102	f
38	&	71	G	103	g
39	'	72	H	104	h
40	(	73	I	105	i
41	)	74	J	106	j

42	*	75	K	107	k
43	+	76	L	108	l
44	,	77	M	109	m
45	-	78	N	110	n
46	.	79	O	111	o
47	/	80	P	112	p
48	0	81	Q	113	q
49	1	82	R	114	r
50	2	83	S	115	s
51	3	84	T	116	t
52	4	85	U	117	u
53	5	86	V	118	v
54	6	87	W	119	w
55	7	88	X	120	x
56	8	89	Y	121	y
57	9	90	Z	122	z

해시 코드와 해시 인덱스

여기서는 `hashCode()`라는 메서드를 통해서 문자를 기반으로 고유한 숫자를 만들었다. 이렇게 만들어진 숫자를 해시 코드라 한다.

여기서 만든 해시 코드는 숫자이기 때문에 배열의 인덱스로 사용할 수 있다. 전체 과정을 그림으로 살펴보자.



- `hashCode()` 메서드를 사용해서 문자열을 해시 코드로 변경한다. 그러면 고유한 정수 숫자 값이 나오는데, 이것을 해시 코드라 한다.
- 숫자 값인 해시 코드를 사용해서 해시 인덱스를 생성한다.
- 이렇게 생성된 해시 인덱스를 배열의 인덱스로 사용하면 된다.

용어 정리

해시 함수(Hash Function)

- 해시 함수는 임의의 길이의 데이터를 입력으로 받아, 고정된 길이의 해시값(해시 코드)을 출력하는 함수이다.
  - 여기서 의미하는 고정된 길이는 저장 공간의 크기를 뜻한다. 예를 들어서 `int`형 `1`, `100`은 둘다 4byte를 차지하는 고정된 길이는 뜻한다.
- 같은 데이터를 입력하면 항상 같은 해시 코드가 출력된다.
- 다른 데이터를 입력해도 같은 해시 코드가 출력될 수 있다. 이것을 해시 충돌이라 한다.
  - 해시 충돌의 예
  - "BC" ➜ B(66) + C(67) = 133
  - "AD" ➜ A(65) + D(68) = 133

해시 코드(Hash Code)

해시 코드는 데이터를 대표하는 값을 뜻한다. 보통 해시 함수를 통해 만들어진다.

- 데이터 `A`의 해시 코드는 `65`
- 데이터 `B`의 해시 코드는 `66`
- 데이터 `AB`의 해시 코드는 `131`

해시 인덱스(Hash Index)

- 해시 인덱스는 데이터의 저장 위치를 결정하는데, 주로 해시 코드를 사용해서 만든다.
- 보통 해시 코드의 결과에 배열의 크기를 나누어 구한다.

요약하면, 해시 코드는 데이터를 대표하는 값, 해시 함수는 이러한 해시 코드를 생성하는 함수, 그리고 해시 인덱스는 해시 코드를 사용해서 데이터의 저장 위치를 결정하는 값을 뜻한다.

정리

문자 데이터를 사용할 때도, 해시 함수를 사용해서 정수 기반의 해시 코드로 변환한 덕분에, 해시 인덱스를 사용할 수 있게 되었다. 따라서 문자의 경우에도 해시 인덱스를 통해 빠르게 저장하고 조회할 수 있다.

여기서 핵심은 해시 코드이다.
세상의 어떤 객체든지 정수로 만든 해시 코드만 정의할 수 있다면 해시 인덱스를 사용할 수 있다.
그렇다면 문자 뿐만 아니라 내가 직접 만든 `Member`, `User`와 같은 객체는 어떻게 해시 코드를 정의할 수 있을까? 자바의 `hashCode()` 메서드에 대해 알아보자.

자바의 hashCode()

해시 인덱스를 사용하는 해시 자료 구조는 데이터 추가, 검색, 삭제의 성능이 O(1)로 매우 빠르다. 따라서 많은 곳에서 자주 사용된다. 그런데 앞서 학습한 것 처럼 해시 자료 구조를 사용하려면 정수로 된 숫자 값인 해시 코드가 필요하다.
자바에는 정수 `int`, `Integer` 뿐만 아니라 `char`, `String`, `Double`, `Boolean`등 수 많은 타입이 있다. 뿐만 아니라 개발자가 직접 정의한 `Member`, `User`와 같은 사용자 정의 타입도 있다.
이 모든 타입을 해시 자료 구조에 저장하려면 모든 객체가 숫자 해시 코드를 제공할 수 있어야 한다.

Object.hashCode()

자바는 모든 객체가 자신만의 해시 코드를 표현할 수 있는 기능을 제공한다. 바로 `Object`에 있는 `hashCode()` 메서드이다.

```
public class Object {
    public int hashCode();
}
```

- 이 메서드를 그대로 사용하기 보다는 보통 재정의(오버라이딩)해서 사용한다.
- 이 메서드의 기본 구현은 객체의 참조값을 기반으로 해시 코드를 생성한다.
- 쉽게 이야기해서 객체의 인스턴스가 다르면 해시 코드도 다르다.

코드를 구현해보자.

```
package collection.set.member;

import java.util.Objects;

public class Member {

    private String id;

    public Member(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Member member = (Member) o;
        return Objects.equals(id, member.id);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id);
    }

    @Override
    public String toString() {
```

```
        return "Member{" +
                "id='" + id + '\'' +
                '}';
    }
}
```

- IDE가 제공하는 자동 완성 기능을 사용해서 `equals()`, `hashCode()`를 생성하자.
- 여기서는 `Member`의 `id`값을 기준으로 equals 비교를 하고, hashCode도 생성한다.

```
package collection.set;

import collection.set.member.Member;

public class JavaHashCodeMain {

    public static void main(String[] args) {
        //Object의 기본 hashCode는 객체의 참조값을 기반으로 생성
        Object obj1 = new Object();
        Object obj2 = new Object();
        System.out.println("obj1.hashCode() = " + obj1.hashCode());
        System.out.println("obj2.hashCode() = " + obj2.hashCode());

        //각 클래스마다 hashCode를 이미 오버라이딩 해두었다.
        Integer i = 10;
        String strA = "A";
        String strAB = "AB";
        System.out.println("10.hashCode = " + i.hashCode());
        System.out.println("'A'.hashCode = " + strA.hashCode());
        System.out.println("'AB'.hashCode = " + strAB.hashCode());

        //hashCode는 마이너스 값이 들어올 수 있다.
        System.out.println("-1.hashCode = " + Integer.valueOf(-1).hashCode());

        //둘은 같을까 다를까?, 인스턴스는 다르지만, equals는 같다.
        Member member1 = new Member("idA");
        Member member2 = new Member("idA");

        //equals, hashCode를 오버라이딩 하지 않은 경우와, 한 경우를 비교
        System.out.println("(member1 == member2) = " + (member1 == member2));
        System.out.println("member1 equals member2 = " +
member1.equals(member2));
        System.out.println("member1.hashCode() = " + member1.hashCode());
```

```
        System.out.println("member2.hashCode() = " + member2.hashCode());
    }
}
```

실행 결과

```
obj1.hashCode() = 762218386
obj2.hashCode() = 796533847

10.hashCode = 10
'A'.hashCode = 65
'AB'.hashCode = 2081
-1.hashCode = -1

(member1 == member2) = false
member1 equals member2 = true
member1.hashCode() = 104101
member2.hashCode() = 104101
```

Object의 해시 코드 비교

- `Object`가 기본으로 제공하는 `hashCode()`는 객체의 참조값을 해시 코드로 사용한다. 따라서 각각의 인스턴스마다 서로 다른 값을 반환한다.
- 그 결과 `obj1`, `obj2`는 서로 다른 해시 코드를 반환한다.

자바의 기본 클래스의 해시 코드

- `Integer`, `String` 같은 자바의 기본 클래스들은 대부분 내부 값을 기반으로 해시 코드를 구할 수 있도록 `hashCode()` 메서드를 재정의해 두었다.
- 따라서 데이터의 값이 같으면 같은 해시 코드를 반환한다.
- 해시 코드의 경우 정수를 반환하기 때문에 마이너스 값이 나올 수 있다.

동일성과 동등성 복습

잠깐 동일성과 동등성에서 학습한 내용을 복습해보자.

`Object`는 동등성 비교를 위한 `equals()` 메서드를 제공한다.

자바는 두 객체가 같다는 표현을 2가지로 분리해서 사용한다.

- **동일성(Identity)**: `==` 연산자를 사용해서 두 객체의 참조가 동일한 객체를 가리키고 있는지 확인
- **동등성(Equality)**: `equals()` 메서드를 사용하여 두 객체가 논리적으로 동등한지 확인

쉽게 이야기해서 동일성(Identity)은 물리적으로 같은 메모리에 있는 객체인지 참조값을 확인하는 것이고, 동등성은 논리적으로 같은지 확인하는 것이다.
동일성은 자바 머신 기준이고 메모리의 참조가 기준으로 물리적이다. 동등성은 보통 사람이 생각하는 논리적인 것에 기준을 맞춘다. 따라서 논리적이다.

동등성은 예를 들면 같은 회원 번호를 가진 회원 객체가 2개 있다고 가정해보자.

```
User a = new User("id-100")
User b = new User("id-100")
```

이런 경우 물리적으로 다른 메모리에 있는 객체이지만, 논리적으로는 같다고 표현할 수 있다.
따라서 동일성은 다르지만, 동등성은 같다.

문자의 경우도 마찬가지이다.

```
String s1 = new String("hello");
String s2 = new String("hello");
```

이 경우 물리적으로는 각각의 "hello" 문자열이 다른 메모리에 존재할 수 있지만, 논리적으로는 같은 `"hello"` 라는 문자열이다.

직접 구현하는 해시 코드

`Member` 의 경우 회원의 `id` 가 같으면 논리적으로 같은 회원으로 표현할 수 있다. 따라서 회원 `id` 를 기반으로 동등성을 비교하도록 `equals` 를 재정의해야 한다.
여기에 `hashCode()` 도 같은 원리가 적용된다. 회원의 `id` 가 같으면 논리적으로 같은 회원으로 표현할 수 있다. 따라서 회원 `id` 를 기반으로 해시 코드를 생성해야 한다.

Member의 hashCode() 구현

- `Member` 는 `hashCode()` 를 재정의했다.
- `hashCode()` 를 재정의할 때 `Objects.hash()` 에 해시 코드로 사용할 값을 지정해주면 쉽게 해시 코드를 생성할 수 있다.
- `hashCode()` 를 재정의하지 않으면 `Object` 가 기본으로 제공하는 `hashCode()` 를 사용하게 된다. 이것은 객체의 참조값을 기반으로 해시 코드를 제공한다. 따라서 회원의 `id` 가 같아도 인스턴스가 다르면 다른 해시 코드를 반환하게 된다.
- `hashCode()` 를 `id` 를 기반으로 재정의한 덕분에 인스턴스가 달라도 `id` 값이 같으면 같은 해시 코드를 반환한

다.

- 따라서 인스턴스가 다른 `member1`, `member2` 둘다 같은 해시 코드를 반환하는 것을 확인할 수 있다.

정리

자바가 기본으로 제공하는 클래스 대부분은 `hashCode()`를 재정의해두었다.

객체를 직접 만들어야 하는 경우에 `hashCode()`를 재정의하면 된다.

`hashCode()`만 재정의하면 필요한 모든 종류의 객체를 해시 자료 구조에 보관할 수 있다.

정리하면 해시 자료 구조에 데이터를 저장하는 경우 `hashCode()`를 구현해야 한다.

이제 `hashCode()`를 사용해서 모든 데이터 타입을 저장할 수 있는 `MyHashSetV2`를 만들어보자.

직접 구현하는 Set2 - MyHashSetV2

`MyHashSetV1`은 `Integer` 숫자만 저장할 수 있었다. 여기서는 모든 타입을 저장할 수 있는 `Set`을 만들어보자.

자바의 `hashCode()`를 사용하면 타입과 관계없이 해시 코드를 편리하게 구할 수 있다.

```
package collection.set;

import java.util.Arrays;
import java.util.LinkedList;

public class MyHashSetV2 {

    static final int DEFAULT_INITIAL_CAPACITY = 16;

    private LinkedList<Object>[] buckets;

    private int size = 0;
    private int capacity = DEFAULT_INITIAL_CAPACITY;

    public MyHashSetV2() {
        initBuckets();
    }

    public MyHashSetV2(int capacity) {
```

```
        this.capacity = capacity;
        initBuckets();
    }

    private void initBuckets() {
        buckets = new LinkedList[capacity];
        for (int i = 0; i < capacity; i++) {
            buckets[i] = new LinkedList<>();
        }
    }

    public boolean add(Object value) {
        int hashIndex = hashIndex(value);
        LinkedList<Object> bucket = buckets[hashIndex];
        if (bucket.contains(value)) {
            return false;
        }

        bucket.add(value);
        size++;
        return true;
    }

    public boolean contains(Object searchValue) {
        int hashIndex = hashIndex(searchValue);
        LinkedList<Object> bucket = buckets[hashIndex];
        return bucket.contains(searchValue);
    }

    public boolean remove(Object value) {
        int hashIndex = hashIndex(value);
        LinkedList<Object> bucket = buckets[hashIndex];
        boolean result = bucket.remove(value);
        if (result) {
            size--;
            return true;
        } else {
            return false;
        }
    }

    private int hashIndex(Object value) {
        //hashCode의 결과로 음수가 나올 수 있다. abs()를 사용해서 마이너스를 제거한다.
```

```
        return Math.abs(value.hashCode()) % capacity;
    }

    public int getSize() {
        return size;
    }

    @Override
    public String toString() {
        return "MyHashSetV2{" +
                "buckets=" + Arrays.toString(buckets) +
                ", size=" + size +
                ", capacity=" + capacity +
                '}';
    }
}
```

private LinkedList<Object>[] buckets

- `MyHashSetV1`은 `Integer` 숫자만 저장할 수 있었다. 여기서는 모든 타입을 저장할 수 있도록 `Object`를 사용한다.
- 추가로 저장, 검색, 삭제 메서드의 매개변수도 `Object`로 변경했다.

hashIndex()

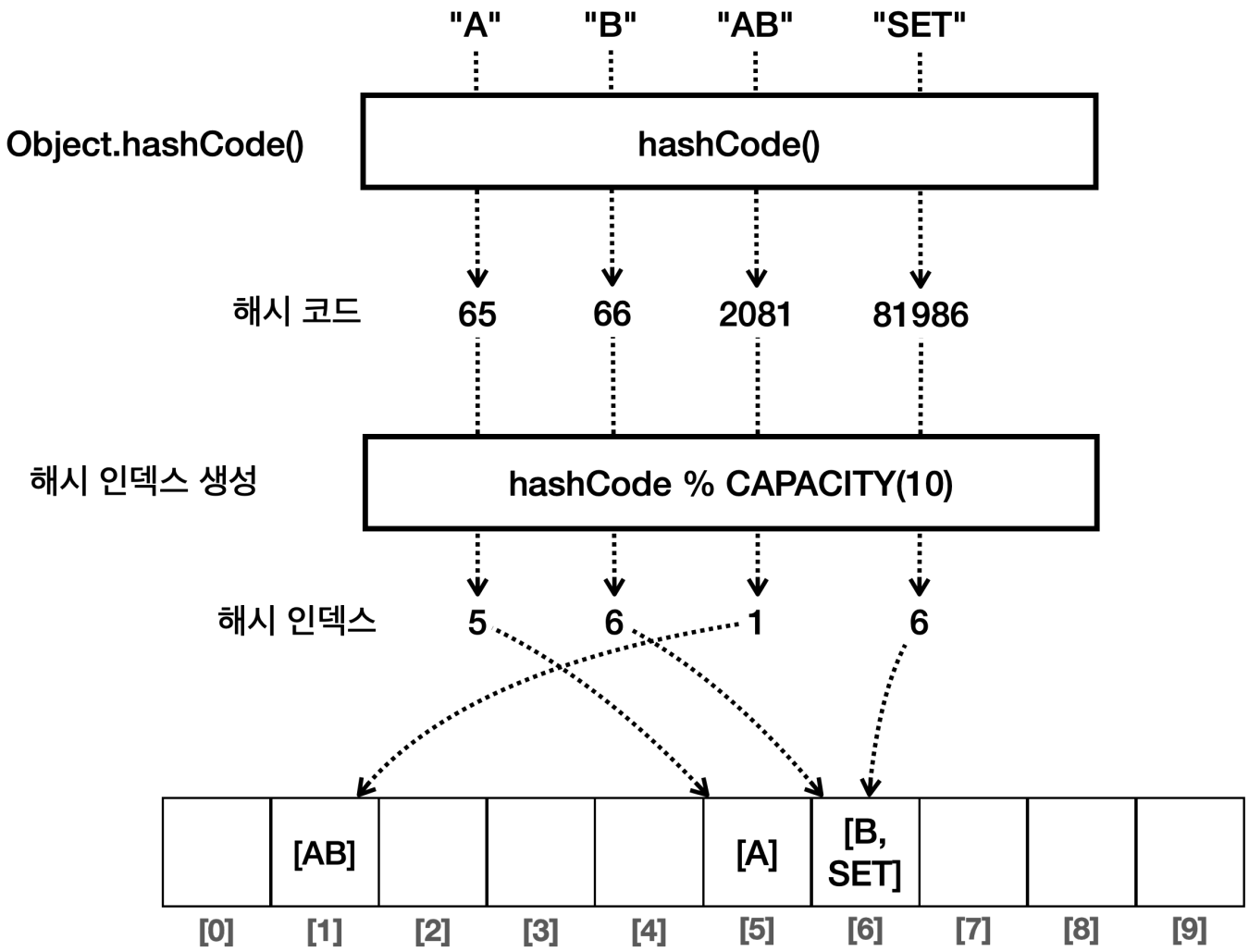
- `hashIndex()` 부분이 변경되었다.
- 먼저 `Object`의 `hashCode()`를 호출해서 해시 코드를 찾는다. 그리고 찾은 해시 코드를 배열의 크기(`capacity`)로 나머지 연산을 수행한다. 이렇게 해시 코드를 기반으로 해시 인덱스를 계산해서 반환한다.
- `Object`의 `hashCode()`를 사용한 덕분에 모든 객체의 `hashCode()`를 구할 수 있다. 물론 다형성에 의해 오버라이딩 된 `hashCode()`가 호출된다.
- `hashCode()`의 실행 결과로 음수가 나올 수 있는데, 배열의 인덱스로 음수는 사용할 수 없다. `Math.abs()`를 사용하면 마이너스를 제거해서 항상 양수를 얻을 수 있다.

```
package collection.set;

public class MyHashSetV2Main1 {

    public static void main(String[] args) {
        MyHashSetV2 set = new MyHashSetV2(10);
```

```
        set.add("A");
        set.add("B");
        set.add("C");
        set.add("D");
        set.add("AB");
        set.add("SET");
        System.out.println(set);

        System.out.println("A.hashCode=" + "A".hashCode());
        System.out.println("B.hashCode=" + "B".hashCode());
        System.out.println("AB.hashCode=" + "AB".hashCode());
        System.out.println("SET.hashCode=" + "SET".hashCode());

        //검색
        String searchValue = "SET";
        boolean result = set.contains(searchValue);
        System.out.println("set.contains(" + searchValue + ") = " + result);
    }
}
```

실행 결과

```
MyHashSetV2{buckets=[[], [AB], [], [], [], [A], [B, SET], [C], [D], []],
size=6, capacity=10}
A.hashCode=65
B.hashCode=66
AB.hashCode=2081
SET.hashCode=81986
bucket.contains(SET) = true
```



- 자바의 `String`은 `hashCode()`를 재정의해 두었다. 우리는 이 값을 사용하면 된다.
- `hashIndex(Object value)`에서 `value.hashCode()`를 호출하면 실제로는 `String`에서 재정의한 `hashCode()`가 호출된다.
- 이렇게 반환된 해시 코드를 기반으로 해시 인덱스를 생성한다.
- 참고로 자바의 해시 함수는 단순히 문자들을 더하기만 하는 것이 아니라 더 복잡한 연산을 사용해서 해시 코드를 구한다. 이 부분은 뒤에서 설명한다.

직접 구현하는 Set3 - 직접 만든 객체 보관

직접 만든 객체를 Set에 보관

`MyHashSetV2`는 `Object`를 받을 수 있다. 따라서 직접 만든 `Member`와 같은 객체도 보관할 수 있다.
여기서 주의할 점은 직접 만든 객체가 `hashCode()`, `equals()` 두 메서드를 반드시 구현해야 한다는 점이다.

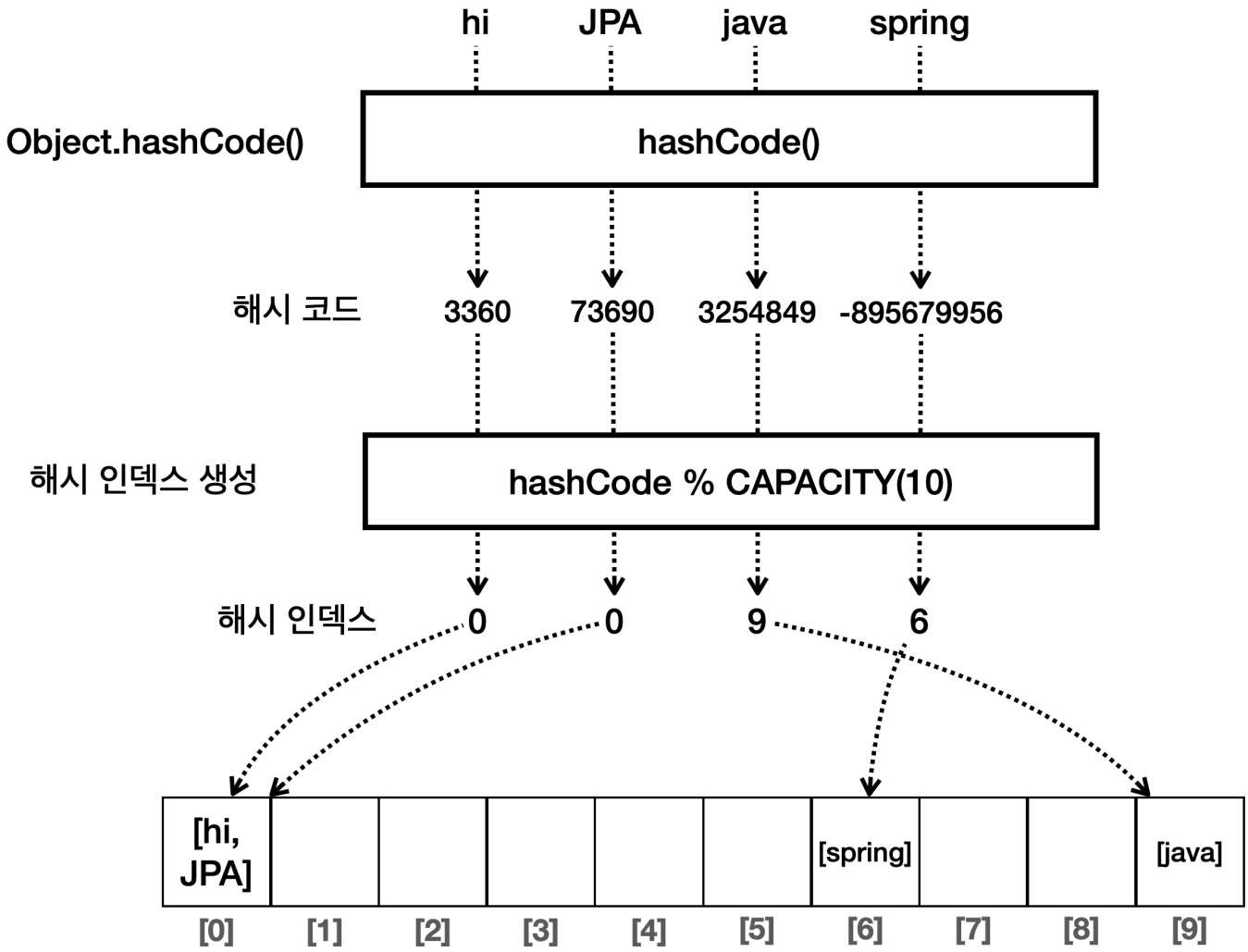
```java
package collection.set;

import collection.set.member.Member;

public class MyHashSetV2Main2 {

    public static void main(String[] args) {
        MyHashSetV2 set = new MyHashSetV2(10);
        Member hi = new Member("hi");
        Member jpa = new Member("JPA"); //대문자 주의!
        Member java = new Member("java");
        Member spring = new Member("spring");

        System.out.println("hi.hashCode() = " + hi.hashCode());
        System.out.println("jpa.hashCode() = " + jpa.hashCode());
        System.out.println("java.hashCode() = " + java.hashCode());
        System.out.println("spring.hashCode() = " + spring.hashCode());

        set.add(hi);
        set.add(jpa);
        set.add(java);
        set.add(spring);
        System.out.println(set);

        //검색
        Member searchValue = new Member("JPA");
        boolean result = set.contains(searchValue);
        System.out.println("set.contains(" + searchValue + ") = " + result);
    }
}
```

- 예제에서 "JPA"가 대문자로 만들어진 것에 주의하자

실행 결과

```
hi.hashCode() = 3360
jpa.hashCode() = 73690
java.hashCode() = 3254849
spring.hashCode() = -895679956

MyHashSetV2{buckets=[[Member{id='hi'}, Member{id='JPA'}], [], [], [], [], [],
[Member{id='spring'}], [], [], [Member{id='java'}]], size=4, capacity=10}
```

```
//검색
bucket.contains(Member{id='JPA'}) = true
```



- `Member`의 `hashCode()`를 `id` 값을 기반으로 재정의해 두었다.
- `hashIndex(Object value)`에서 `value.hashCode()`를 호출하면 실제로는 `Member`에서 재정의한 `hashCode()`가 호출된다.
- 이렇게 반환된 해시 코드를 기반으로 해시 인덱스를 생성한다.

equals() 사용처

그렇다면 `equals()`는 언제 사용할까?

"JPA"를 조회할 때 해시 인덱스는 0이다. 따라서 배열의 `0`번 인덱스를 조회한다.

여기에는 `[hi, JPA]`라는 회원 두 명이 있다. 이것을 하나하나 비교해야 한다. 이때 `equals()`를 사용해서 비교한다.

따라서 해시 자료 구조를 사용할 때는 `hashCode()`는 물론이고, `equals()`도 반드시 재정의해야 한다. 참고로 자바가 제공하는 기본 클래스들은 대부분 `hashCode()`, `equals()`를 함께 재정의해 두었다.

equals, hashCode의 중요성1

해시 자료 구조를 사용하려면 `hashCode()`도 중요하지만, 해시 인덱스가 충돌할 경우를 대비에서 `equals()`도 반드시 재정의해야 한다. 해시 인덱스가 충돌할 경우 같은 해시 인덱스에 있는 데이터들을 하나하나 비교해서 찾아야한다. 이때 `equals()`를 사용해서 비교한다.

참고
해시 인덱스가 같아도 실제 저장된 데이터는 다를 수 있다. 따라서 특정 인덱스에 데이터가 하나만 있어도 `equals()`로 찾는 데이터가 맞는지 검증해야 한다.
앞의 예에서 "hi"라는 회원과 "JPA"라는 회원의 해시 인덱스는 둘다 0으로 같다.
만약 "hi"라는 회원만 저장했다고 가정하자. 이렇게 되면 0번 인덱스에는 하나의 데이터만 보관되어 있다. 이때 "JPA"라는 회원을 찾는다면 같은 0번 인덱스에서 찾는다. 이때 `equals()`를 사용해서 "JPA"라는 회원이 맞는지 검증해야 한다. 0번 인덱스에는 "hi"라는 회원만 있으므로 데이터를 찾는데 실패한다.

따라서 해시 자료 구조를 사용하려면 반드시 `hashCode()`와 `equals()`를 구현해야한다.
지금부터 `hashCode()`, `equals()`를 제대로 구현하지 않으면 어떤 문제가 발생하는지 알아보자.
참고로 자바가 제공하는 `String`, `Integer` 같은 기본 클래스들은 대부분 `hashCode()`, `equals()`가 재정의되어 있다.

Object의 기본 기능

- `hashCode()` : 객체의 참조값을 기반으로 해시 코드를 반환한다.
- `equals()` : `==` 동일성 비교를 한다. 따라서 객체의 참조값이 같아야 `true`를 반환한다.

클래스를 만들 때 `hashCode()`, `equals()`를 재정의하지 않으면, 해시 자료 구조에서 `Object`가 기본으로 제공하는 `hashCode()`, `equals()`를 사용하게 된다. 그런데 `Object`가 기본으로 제공하는 기능은 단순히 인스턴스의 참조를 기반으로 작동한다.

지금부터 hashCode, equals를 제대로 구현하지 않은 경우 해시 자료 구조를 사용할 때 어떤 문제들이 발생하는지 하나씩 알아보자.

- hashCode, equals를 모두 구현하지 않은 경우
- hashCode는 구현했지만 equals를 구현하지 않은 경우
- hashCode와 equals를 모두 구현한 경우

hashCode, equals를 모두 구현하지 않은 경우

```
package collection.set.member;

public class MemberNoHashNoEq {

    private String id;

    public MemberNoHashNoEq(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }

    @Override
    public String toString() {
        return "MemberNoHashNoEq{" +
                "id='" + id + '\'' +
                '}';
    }
}
```

- `hashCode()`, `equals()`를 재정의하지 않았다. 따라서 `Object`의 기본 기능을 사용한다.

```
package collection.set.member;

import collection.set.MyHashSetV2;

public class HashAndEqualsMain1 {

    public static void main(String[] args) {
        //중복 등록
        MyHashSetV2 set = new MyHashSetV2(10);
        MemberNoHashNoEq m1 = new MemberNoHashNoEq("A");
        MemberNoHashNoEq m2 = new MemberNoHashNoEq("A");
        System.out.println("m1.hashCode() = " + m1.hashCode());
        System.out.println("m2.hashCode() = " + m2.hashCode());
        System.out.println("m1.equals(m2) = " + m1.equals(m2));

        set.add(m1);
        set.add(m2);
```

```
        System.out.println(set);

        //검색 실패
        MemberNoHashNoEq searchValue = new MemberNoHashNoEq("A");
        System.out.println("searchValue.hashCode() = " +
searchValue.hashCode());
        boolean contains = set.contains(searchValue);
        System.out.println("contains = " + contains);
    }
}
```

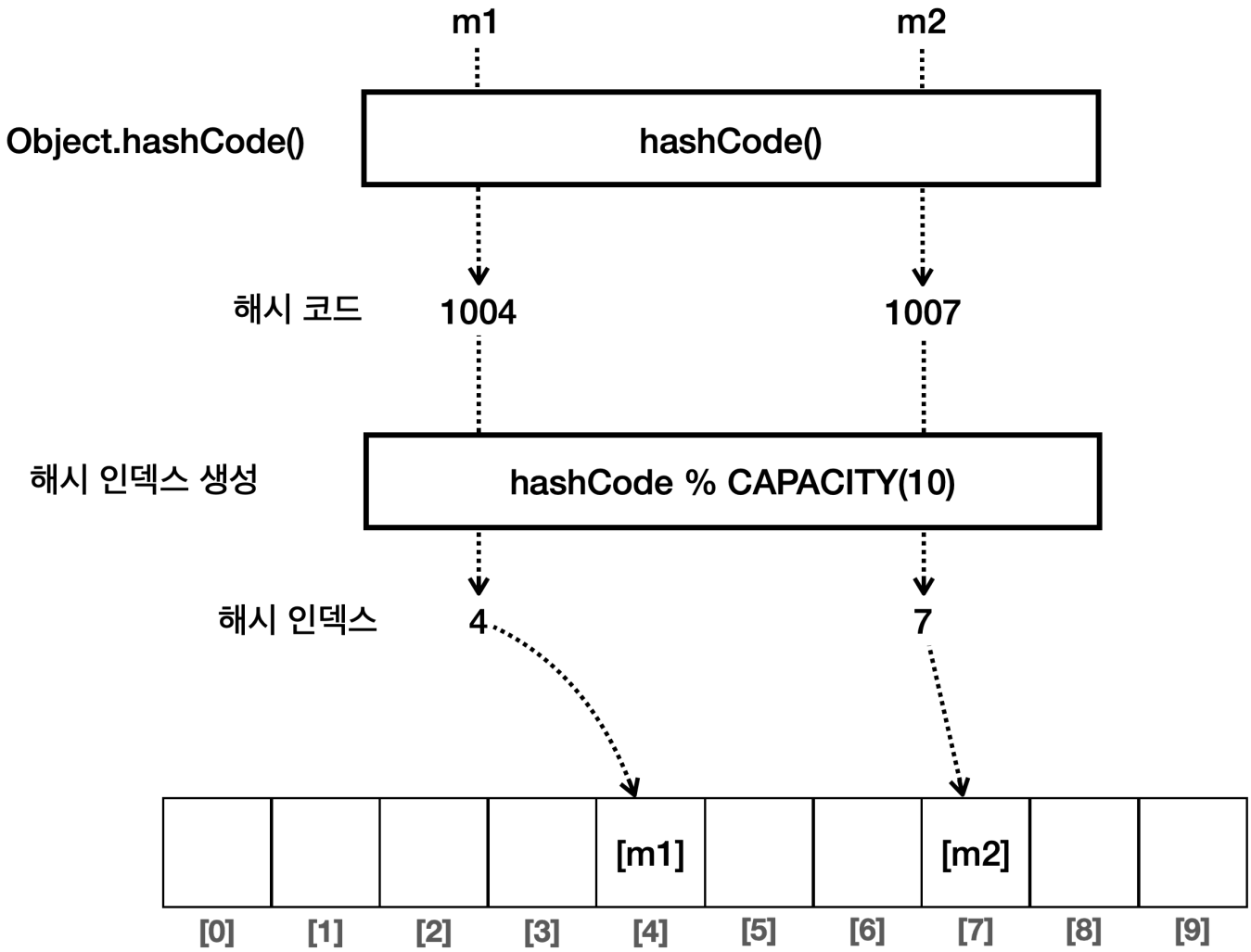
실행 결과

```
m1.hashCode() = 1004 //인스턴스의 참조이므로 변한다.
m2.hashCode() = 1007 //인스턴스의 참조이므로 변한다.
m1.equals(m2) = false
MyHashSetV2{buckets=[[MemberNoHashNoEq{id='A'}], [], [], [],
[MemberNoHashNoEq{id='A'}], [], [], [], [], []], size=2, capacity=10}
searchValue.hashCode() = 1008
contains = false
```

`m1.hashCode()`, `m2.hashCode()`는 `Object`의 기본 기능을 사용하기 때문에 객체의 참조값을 기반으로 해시 코드를 생성한다. 따라서 실행할 때 마다 값이 달라질 수 있다. 여기서는 `m1=1004`, `m2=1007`이라고 가정하겠다.

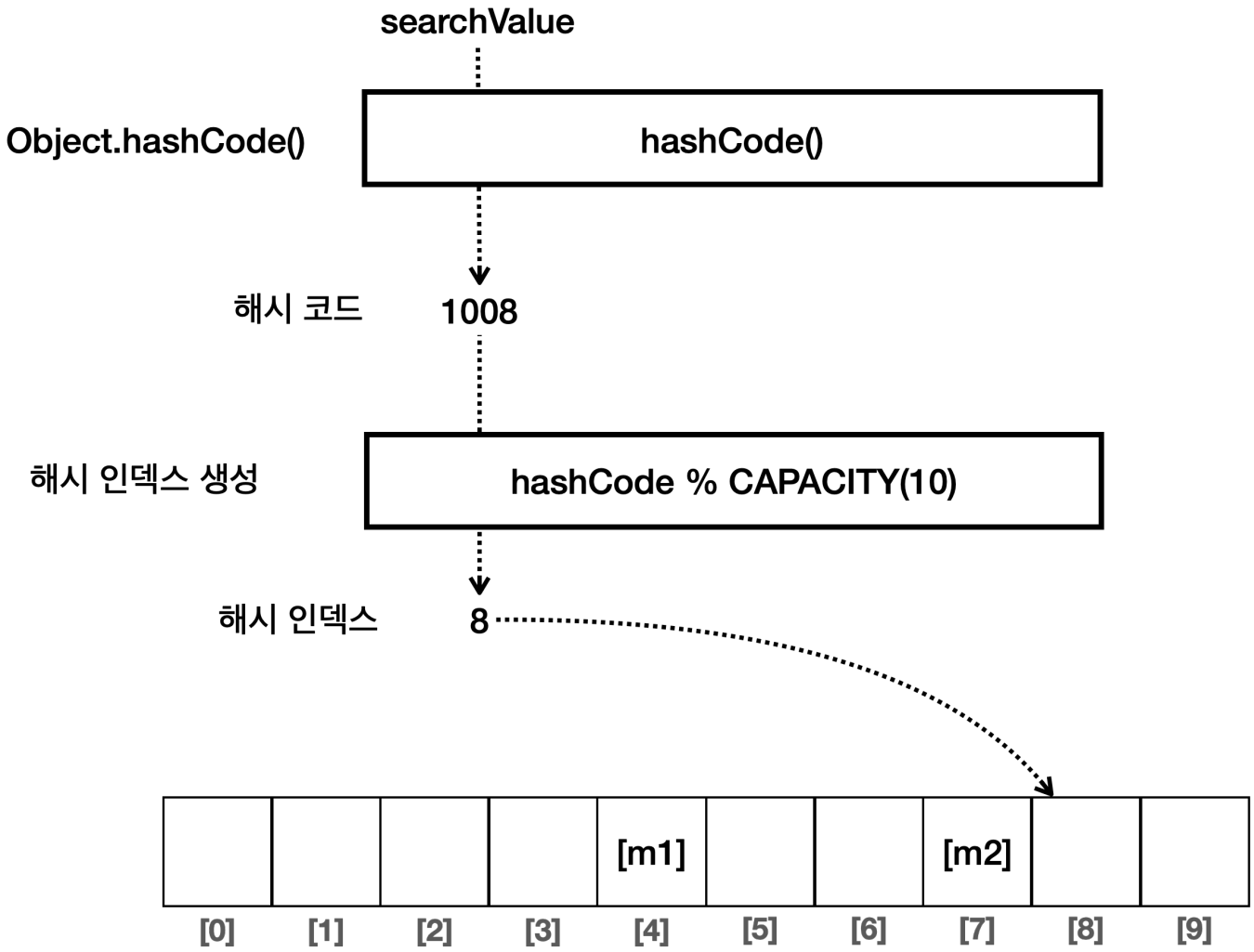
`m1`과 `m2`는 인스턴스는 다르지만 둘다 "A"라는 같은 회원 `id`를 가지고 있다. 따라서 **논리적으로 같은 회원**으로 보아야 한다.

데이터 저장 문제



- `m1` 과 `m2` 의 해시 코드가 서로 다르기 때문에 다른 위치에 각각 저장된다.
- 회원 id가 "A"로 같은 회원의 데이터가 데이터가 중복 저장된다.

데이터 검색 문제



- `MemberNoHashNoEq searchValue = new MemberNoHashNoEq("A")`
- 회원 id가 "A"인 객체를 검색하기 위해 회원 id가 "A"인 객체를 만들었다. 이 객체의 참조값은 `1008`이라 가정하자.
- 데이터를 검색할 때 `searchValue` 객체의 해시 코드는 `1008`이다. 따라서 다른 위치에서 데이터를 찾게 되고, 검색에 실패한다.

equals, hashCode의 중요성2

hashCode는 구현했지만 equals를 구현하지 않은 경우

이번에는 `hashCode()`만 재정의하고, `equals()`는 재정의하지 않으면 어떤 문제가 발생하는 알아보자.

```
package collection.set.member;
```

```java
import java.util.Objects;

public class MemberOnlyHash {

    private String id;

    public MemberOnlyHash(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }

    @Override
    public int hashCode() {
        return Objects.hash(id);
    }

    @Override
    public String toString() {
        return "MemberOnlyHash{" +
                "id='" + id + '\'' +
                '}';
    }
}
```

- `Objects.hash(id)`를 사용해서 `id`를 기준으로 해시 코드를 생성했다.

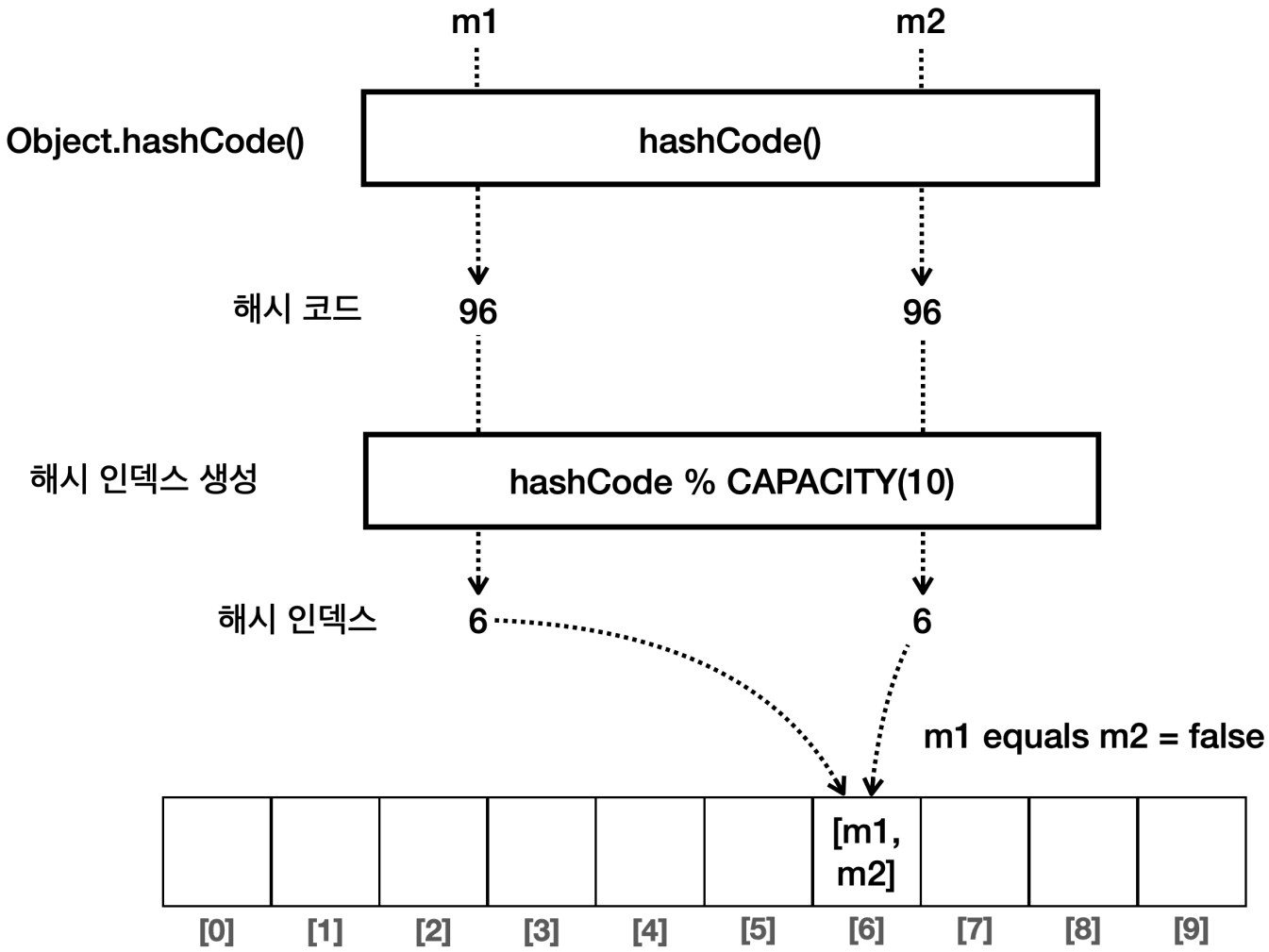
```java
package collection.set.member;

import collection.set.MyHashSetV2;

public class HashAndEqualsMain2 {

    public static void main(String[] args) {
        //중복 등록
        MyHashSetV2 set = new MyHashSetV2(10);
        MemberOnlyHash m1 = new MemberOnlyHash("A");
        MemberOnlyHash m2 = new MemberOnlyHash("A");
        System.out.println("m1.hashCode() = " + m1.hashCode());
        System.out.println("m2.hashCode() = " + m2.hashCode());
```

```
        System.out.println("m1.equals(m2) = " + m1.equals(m2));

        set.add(m1);
        set.add(m2);
        System.out.println(set);

        //검색 실패
        MemberOnlyHash searchValue = new MemberOnlyHash("A");
        System.out.println("searchValue.hashCode() = " +
searchValue.hashCode());
        boolean contains = set.contains(searchValue);
        System.out.println("contains = " + contains);
    }
}
```

실행 결과

```
m1.hashCode() = 96
m2.hashCode() = 96
m1.equals(m2) = false
MyHashSetV2{buckets=[[], [], [], [], [], [], [MemberOnlyHash{id='A'},
MemberOnlyHash{id='A'}], [], [], []], size=2, capacity=10}
searchValue.hashCode() = 96
contains = false
```

데이터 저장 문제



- `hashCode()`를 재정의했기 때문에 같은 `id`를 사용하는 `m1`, `m2`는 같은 해시 코드를 사용한다.
- 따라서 같은 해시 인덱스에 데이터가 저장된다.
- 그런데 `add()` 로직은 중복 데이터를 체크하기 때문에 같은 데이터가 저장되면 안된다.

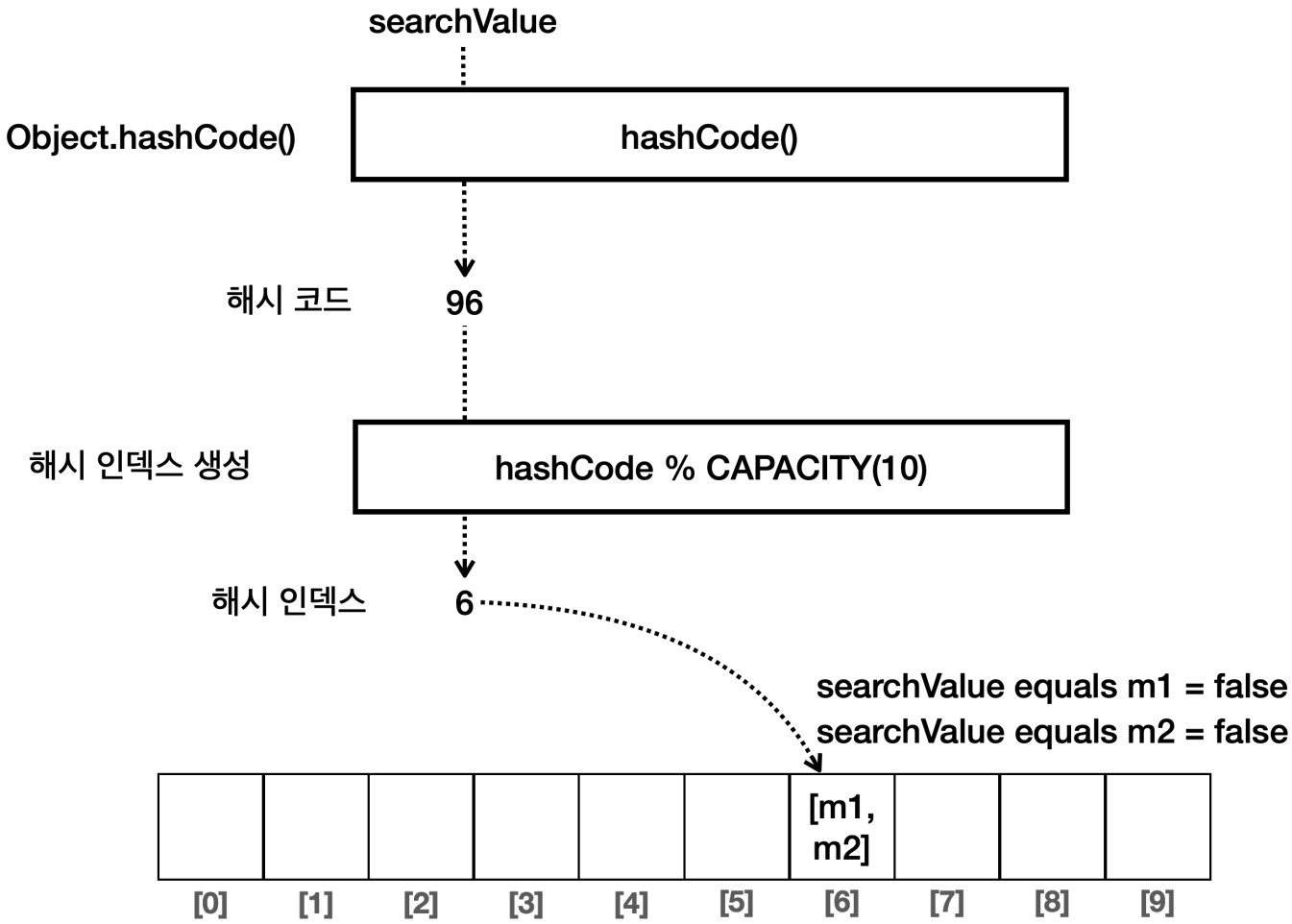
```
public boolean add(Object value) {
    int hashIndex = hashIndex(value);
    LinkedList<Object> bucket = buckets[hashIndex];

    //중복 체크 로직
    if (bucket.contains(value)) {
        return false;
    }

    bucket.add(value);
    size++;
    return true;
}
```

- `bucket.contains()` 내부에서 데이터를 순차 비교할 때 `equals()`를 사용한다.
- 그런데 `MemberOnlyHash`는 `equals()`를 재정의하지 않았으므로 `Object`의 `equals()`를 상속 받아서

사용한다. 따라서 인스턴스의 참조값을 비교한다. 인스턴스가 서로 다른 `m1`, `m2`는 비교에 실패한다.

- `add()` 로직은 중복 데이터가 없다고 생각하고 `m1`, `m2`를 모두 저장한다.
- 결과적으로 같은 회원 `id`를 가진 중복 데이터가 저장된다.

데이터 검색 문제



- `MemberOnlyHash searchValue = new MemberOnlyHash("A")`
- 회원 id가 "A"인 객체를 검색하기 위해 회원 id가 "A"인 객체를 만들었다. 해시 코드가 구현되어 있다.
- `searchValue`는 해시 인덱스 6을 정확히 찾을 수 있다.
- 해시 인덱스에 있는 모든 데이터를 `equals()`를 통해 비교해서 같은 값을 찾아야 한다.
- 다음 코드를 보자. `bucket.contains(searchValue)` 내부에서 연결 리스트에 있는 모든 항목을 `searchValue`와 `equals()`로 비교한다.

```
public boolean contains(Object searchValue) {
    int hashIndex = hashIndex(searchValue);
    LinkedList<Object> bucket = buckets[hashIndex];
    return bucket.contains(searchValue);
}
```

- `MemberOnlyHash`는 `equals()`를 재정의하지 않았으므로 `Object`의 `equals()`를 상속 받아서 사용한다. 따라서 인스턴스의 참조값을 비교한다. 인스턴스가 서로 다른 `searchValue`와 `m1`, `m2`는 비교에 실패한다.
- 결과적으로 데이터를 찾을 수 없다.

hashCode와 equals를 모두 구현한 경우

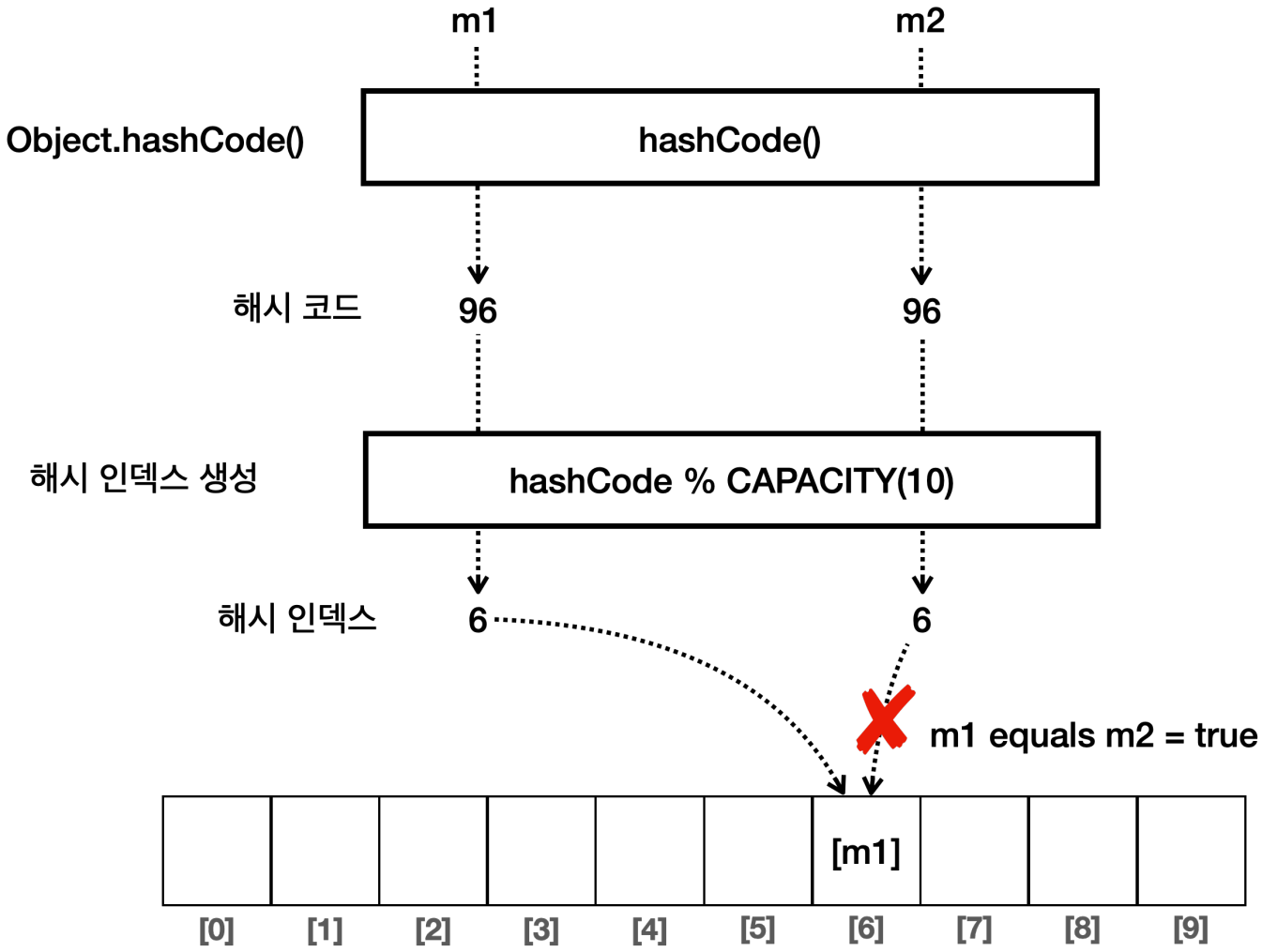
이번에는 `hashCode()`와 `equals()` 모두 재정의한 경우를 살펴보자. 기존에 작성한 `Member` 클래스를 활용하자.

```
package collection.set.member;

import collection.set.MyHashSetV2;

public class HashAndEqualsMain3 {

    public static void main(String[] args) {
        //중복 등록 안됨
        MyHashSetV2 set = new MyHashSetV2(10);
        Member m1 = new Member("A");
        Member m2 = new Member("A");
        System.out.println("m1.hashCode() = " + m1.hashCode());
        System.out.println("m2.hashCode() = " + m2.hashCode());
        System.out.println("m1.equals(m2) = " + m1.equals(m2));

        set.add(m1);
        set.add(m2);
        System.out.println(set);

        //검색 성공
        Member searchValue = new Member("A");
        System.out.println("searchValue.hashCode() = " +
searchValue.hashCode());
        boolean contains = set.contains(searchValue);
        System.out.println("contains = " + contains);
    }
}
```

실행 결과

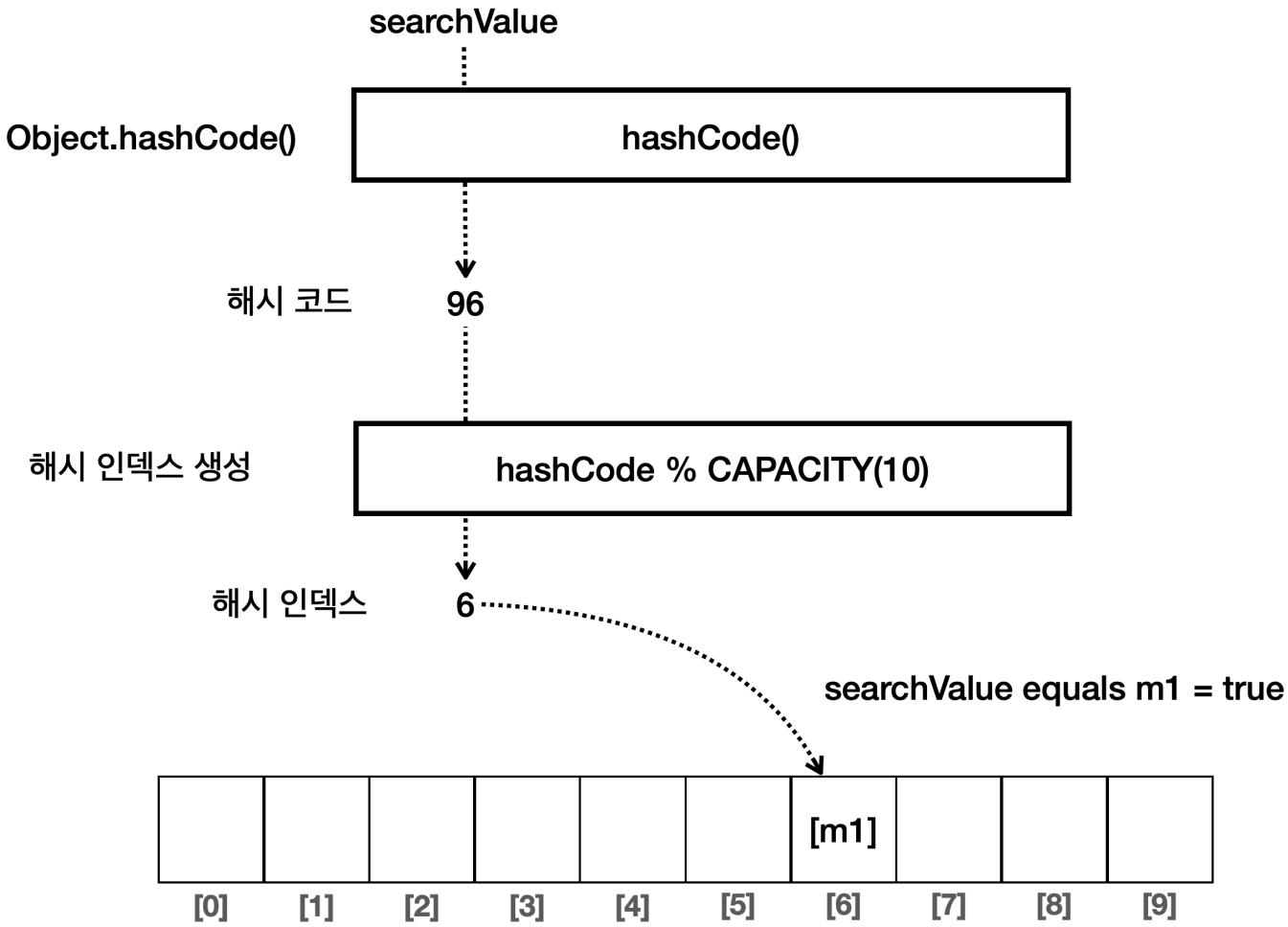
```
m1.hashCode() = 96
m2.hashCode() = 96
m1.equals(m2) = true
MyHashSetV2{buckets=[[], [], [], [], [], [], [Member{id='A'}], [], [], []],
size=1, capacity=10}
searchValue.hashCode() = 96
contains = true
```

데이터 저장



- 처음에 `m1`을 저장한다.
- 다음으로 `m2` 저장을 시도한다. `m1`과 같은 해시 코드를 사용하므로 해시 인덱스도 같다.
- 여기서 중복 데이터를 저장하면 안되므로 `m1`과 `m2`를 `equals` 비교한다. 같은 데이터가 이미 있으므로 `m2`는 저장에 실패한다.
- 결과적으로 중복 데이터가 저장되지 않는다.

데이터 검색



- `searchValue`의 해시 코드로 6번 해시 인덱스를 찾는다.
- `searchValue`와 6번 해시 인덱스 내부의 데이터를 모두 `equals` 비교한다. `searchValue`와 `m1`이 `equals` 비교에 성공하므로 참을 반환한다.

정리

`hashCode()`를 항상 재정의해야 하는 것은 아니다. 하지만 해시 자료 구조를 사용하는 경우 `hashCode()`와 `equals()`를 반드시 함께 재정의해야 한다. 물론 직접 재정의하는 것은 쉽지 않으므로 IDE의 도움을 받자.

참고 - 해시 함수는 해시 코드가 최대한 충돌하지 않도록 설계

다른 데이터를 입력해도 같은 해시 코드가 출력될 수 있다. 이것을 해시 충돌이라 한다.

- 앞서 직접 만든 해시 함수의 해시 코드 충돌의 예
- "BC" ➜ B(66) + C(67) = 133
- "AD" ➜ A(65) + D(68) = 133

해시 함수로 해시 코드를 만들 때 단순히 문자의 숫자를 더하기만 해서는 해시가 충돌할 가능성이 높다.

해시가 충돌하면 결과적으로 같은 해시 인덱스에 보관된다. 따라서 성능이 나빠진다.

자바의 해시 함수는 이런 문제를 해결하기 위해 문자의 숫자를 단순히 더하기만 하는 것이 아니라 내부에서 복잡한 추가 연산을 수행한다. 따라서 자바의 해시 코드를 사용하면 `"AB"`의 결과가 `2081`, `"SET"`의 결과가 `81986`으로 복잡한 계산 결과가 나온다.
복잡한 추가 연산으로 다양한 범위의 해시 코드가 만들어지므로 해시가 충돌할 가능성이 낮아지고, 결과적으로 해시 자료 구조를 사용할 때 성능이 개선된다.

해시 함수는 같은 입력에 대해서 항상 동일한 해시 코드를 반환해야 한다.
좋은 해시 함수는 해시 코드가 한 곳에 뭉치지 않고 균일하게 분포하는 것이 좋다. 그래야 해시 인덱스도 골고루 분포되어서 해시 자료 구조의 성능을 최적화할 수 있다. 이런 해시 함수를 직접 구현하는 것은 쉽지 않다. 자바가 제공하는 해시 함수들을 사용하면 이런 부분을 걱정하지 않고 최적화 된 해시 코드를 구할 수 있다.
하지만 자바가 제공하는 해시 함수를 사용해도 같은 해시 코드가 생성되어서 해시 코드가 충돌하는 경우도 간혹 존재한다.
예)

- `"Aa".hashCode() = 2112`
- `"BB".hashCode() = 2112`

이 경우 같은 해시 코드를 가지기 때문에 해시 인덱스도 같게 된다. 하지만 `equals()`를 사용해서 다시 비교하기 때문에 해시 코드가 충돌하더라도 문제가 되지는 않는다. 그리고 매우 낮은 확률로 충돌하기 때문에 성능에 대한 부분도 크게 걱정하지 않아도 된다.

직접 구현하는 Set4 - 제네릭과 인터페이스 도입

지금까지 만든 해시 셋에 제네릭을 도입해서 타입 안전성을 높여보자.

```
package collection.set;

public interface MySet<E> {
    boolean add(E element);
    boolean remove(E value);
    boolean contains(E value);
}
```

- 핵심 기능을 인터페이스로 뽑았다.
- 이 인터페이스를 구현하면 해시 기반이 아니라 다른 자료 구조 기반의 Set도 만들 수 있다.

```java
package collection.set;

import java.util.Arrays;
import java.util.LinkedList;

public class MyHashSetV3<E> implements MySet<E> {

    static final int DEFAULT_INITIAL_CAPACITY = 16;

    private LinkedList<E>[] buckets;

    private int size = 0;
    private int capacity = DEFAULT_INITIAL_CAPACITY;

    public MyHashSetV3() {
        initBuckets();
    }

    public MyHashSetV3(int capacity) {
        this.capacity = capacity;
        initBuckets();
    }

    private void initBuckets() {
        buckets = new LinkedList[capacity];
        for (int i = 0; i < capacity; i++) {
            buckets[i] = new LinkedList<>();
        }
    }

    @Override
    public boolean add(E value) {
        int hashIndex = hashIndex(value);
        LinkedList<E> bucket = buckets[hashIndex];
        if (bucket.contains(value)) {
            return false;
        }

        bucket.add(value);
        size++;
        return true;
```

```
    }

    @Override
    public boolean contains(E searchValue) {
        int hashIndex = hashIndex(searchValue);
        LinkedList<E> bucket = buckets[hashIndex];
        return bucket.contains(searchValue);
    }

    @Override
    public boolean remove(E value) {
        int hashIndex = hashIndex(value);
        LinkedList<E> bucket = buckets[hashIndex];
        boolean result = bucket.remove(value);
        if (result) {
            size--;
            return true;
        } else {
            return false;
        }
    }

    private int hashIndex(Object value) {
        //hashCode의 결과로 음수가 나올 수 있다. abs()를 사용해서 마이너스를 제거한다.
        return Math.abs(value.hashCode()) % capacity;
    }

    public int getSize() {
        return size;
    }

    @Override
    public String toString() {
        return "MyHashSetV3{" +
                "buckets=" + Arrays.toString(buckets) +
                ", size=" + size +
                ", capacity=" + capacity +
                '}';
    }
}
```

- 이전 코드에서 `Object`로 다루던 부분들을 제네릭의 타입 매개변수 `E`로 변경했다.

```
package collection.set;

public class MyHashSetV3Main {

    public static void main(String[] args) {
        MySet<String> set = new MyHashSetV3<>(10);
        set.add("A");
        set.add("B");
        set.add("C");
        System.out.println(set);

        //검색
        String searchValue = "A";
        boolean result = set.contains(searchValue);
        System.out.println("set.contains(" + searchValue + ") = " + result);
    }
}
```

실행 결과

```
MyHashSetV3{buckets=[[], [], [], [], [], [A], [B], [C], [], []], size=3,
capacity=10}
bucket.contains(A) = true
```

제네릭의 덕분에 타입 안전성이 높은 자료 구조를 만들 수 있었다.