Set<E> 컬렉션 클래스들

Set<E> 컬렉션 클래스들

- Set<E>을 구현하는 클래스의 특성과 HashSet<E> 클래스
  - Set<E> 인터페이스를 구현하는 제네릭 클래스의 두 가지 특성
    - 저장 순서가 유지되지 않음
    - 데이터의 중복 저장을 허용하지 않음
  - Set<E> 인터페이스를 구현하는 대표 클래스 HashSet<E>

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class SetCollectionFeature {
    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.println(itr.next());
        }
        // for-each문을 이용한 전체 출력
        for (String s : set) {
            System.out.println(s);
        }
    }
}
```

Set<E> 컬렉션 클래스들

- HashSet<E>가 데이터(인스턴스)가 동일하다고 판단하는 기준 확인 필요

```
public class Num {
    private int num;

    public Num(int num) {
        this.num = num;
    }

    @Override
    public String toString() {
        return String.valueOf(this.num);
    }
}
```

```
public class HashSetEqualityOne {
    public static void main(String[] args) {
        HashSet<Num> set = new HashSet<>();
        set.add(new Num(7799));
        set.add(new Num(9955));
        set.add(new Num(7799));
        System.out.println("인스턴스 수 : " + set.size());
        for (Num n : set) {
            System.out.println(n);
        }
    }
}
```

- "set.add(new Num(7799));"는 동일한 데이터를 저장하고 있지만 중복된 데이터로 처리하지 않고 다른 인스턴스로 간주
- HashSet<E>는 Object 클래스에 정의되어 있는 "public boolean equals(Object obj)", "public in hashCode()" 메소드의 호출 결과를 통해 동일 인스턴스 여부를 판단
- 즉 데이터가 아닌 인스턴스의 동일성 여부 판단

Set<E> 컬렉션 클래스들

- 해쉬 알고리즘과 hashCode 메소드
  - HashSet<E> 클래스를 잘 활용하기 위해서는 간단하게나마 해쉬 알고리즘을 이해하여야 함
  - "num % 3"에서 num이 3, 5, 7, 12, 25, 31일 때 그 수들을 분류하면 아래 그림과 같이 연산 결과 0, 1, 2를 기준으로 분류 가능

연산결과0	연산결과1	연산결과2
3, 12	7, 25, 31	5

  - 정수 5의 존재 여부를 확인하는 가장 효율적인 방법
    - 정수들이 3으로 나눈 나머지를 기준으로 나뉘므로, 존재 여부의 확인 대상인 5를 3으로 나머지 연산을 하여, 같은 부류 탐색
    - 위 과정을 거치게 되면 연산 결과가 0, 1인 부류는 탐색 대상에서 제외 -> 해쉬 알고리즘을 사용하는 이유
  - 정수 5의 존재 여부를 확인하는 과정 : 두 단계를 거쳐서 탐색을 진행하기 때문에 탐색 속도가 매우 빠름
    - 탐색 1단계 : 정수 5의 해쉬 값을 계산하여 탐색 부류를 결정
    - 탐색 2단계 선택된 부류 내에서 정수 5가 존재하는지 확인
  - HashSet<E> 클래스는 위의 두 단계를 거쳐서 동일 인스턴스의 존재 여부 확인
    - Object 클래스에 정의된 hashCode 메소드의 반환 값을 기반으로 부류 결정
    - 선택된 부류 내에서 equals 메소드를 호출하여 동등 비교

Set<E> 컬렉션 클래스들

- Object 클래스에 정의되어 있는 hashCode와 equals 메소드의 정의 방식
  - hashCode 메소드는 인스턴스가 저장된 주소값을 기반으로 반환 값이 만들어지도록 정의되어 있음
  - 인스턴스가 다르면 Object 클래스의 hashCode 메소드는 다른 값 반환
  - 인스턴스가 다르면 Object 클래스의 equals 메소드는 false 반환
- 즉, Object 클래스의 hashCode와 equals 메소드는 저장하고 있는 데이터가 아닌 인스턴스의 동등 여부 판단
- 데이터를 기준으로 동등 여부를 판단하려면 hashCode와 equals 메소드를 아래와 같이 오버라이딩 하여 판단

```
public class Num {
    private int num;
    public Num(int num) { this.num = num; }
    @Override
    public int hashCode() { return num % 3; }
    @Override
    public boolean equals(Object obj) {
        if(num == ((Num)obj).num) return true;
        else return false;
    }
    @Override
    public String toString() { return String.valueOf(this.num); }
}
```

```
public class HashSetEqualityOne {
    public static void main(String[] args) {
        HashSet<Num> set = new HashSet<>();
        set.add(new Num(7799));
        set.add(new Num(9955));
        set.add(new Num(7799));
        System.out.println("인스턴스 수 : " + set.size());
        for (Num n : set) {
            System.out.println(n);
        }
    }
}
```

Set<E> 컬렉션 클래스들

- hashCode 메소드의 다양한 정의
  - 둘 이상의 값을 지니는 클래스 내용 비교를 위한 hashCode와 equals 메소드의 정의

```
public class Car {
    private String model;
    private String color;
    public Car(String model, String color) {
        this.model = model;
        this.color = color;
    }
    @Override
    public String toString() {
        return model + " : " + color;
    }
    @Override
    public int hashCode() {
        return (model.hashCode() + color.hashCode()) / 2;
    }
    @Override
    public boolean equals(Object obj) {
        if (model.equals(((Car) obj).model)
                && color.equals(((Car) obj).color)) {
            return true;
        } else {
            return false;
        }
    }
}
```

```
public class HowHashCode {
    public static void main(String[] args) {
        HashSet<Car> set = new HashSet<>();
        set.add(new Car("HY_MD-301", "RED"));
        set.add(new Car("HY_MD-301", "BLACK"));
        set.add(new Car("HY_MD-302", "RED"));
        set.add(new Car("HY_MD-302", "WHITE"));
        set.add(new Car("HY_MD-301", "BLACK"));
        System.out.println("인스턴스 수 : " + set.size());
        for(Car car : set) {
            System.out.println(car.toString());
        }
    }
}
```

Set<E> 컬렉션 클래스들

- Car 클래스는 두 개의 참조변수를 가지고 있으며, hashCode와 equals 메소드를 통해 내용을 비교하도록 오버라이딩 되어 있음
- 클래스를 정의할 때마다 hashCode 메소드를 정의하는 것은 번거로우며, 해쉬 알고리즘의 성능적 측면까지 고려하면서 정의하기는 더 어려움
- 이를 위해 자바에서는 "public static int hash(Object...values)" 메소드 제공
  - java.util.Objects에 정의된 메소드이며, 전달된 인자 기반의 해쉬 값 반환
  - Car 클래스의 hashCode 메소드의 내용을 "return Objects.hash(model, color)"로 수정
  - 특별한 경우를 제외하고 해쉬 알고리즘을 따로 작성하지 않고 위 메소드로 판단하여도 무관

Set<E> 컬렉션 클래스들

- TreeSet<E> 클래스의 이해와 활용
  - "트리(tree)"라는 자료구조를 기반으로 인스턴스 저장
  - 정렬된 상태가 유지되면서 인스턴스 저장(트리라는 자료구조의 특성)

```
public class SortedTreeSet {
    public static void main(String[] args) {
        TreeSet<Integer> tree = new TreeSet<>();
        tree.add(3);
        tree.add(1);
        tree.add(2);
        tree.add(4);
        System.out.println("인스턴스 수 : " + tree.size());
        for(Integer n : tree) {
            System.out.println(n.toString());
        }
        for(Iterator<Integer> itr = tree.iterator(); itr.hasNext();) {
            System.out.println(itr.next().toString());
        }
    }
}
```

  - TreeSet<E> 인스턴스는 정렬 상태를 유지하면서 인스턴스를 저장하기 때문에 반복자는 "인스턴스들의 참조 순서는 오름차순을 기준으로 한다" 특성을 가짐

Set<E> 컬렉션 클래스들

- 아래 클래스의 인스턴스는 무엇이 작은 것이며 무엇이 큰지 구분 필요

```
public class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return name + " : " + age;
    }
}
```

- 수의 경우 대소의 비교 기준이 있지만 위 클래스의 경우 기준을 어떻게 정하느냐에 따라서 오름차순의 나열 결과 바뀜
- 위와 같은 클래스를 정의할 때에는 "public interface Comparable<T>" 인터페이스에 존재하는 추상 메소드 "int compareTo(T o)" 의 구현을 통해 대소의 기준 결정

Set<E> 컬렉션 클래스들

- 인스턴스의 비교 기준을 정의하는 Comparable<T> 인터페이스의 구현 기준
  - Comparable<T> 인터페이스에 존재하는 "int compareTo(T o)" 추상 메소드 정의 방법
    - 인자로 전달된 o가 작다면 양의 정수 반환
    - 인자로 전달된 o가 크다면 음의 정수 반환
    - 인자로 전달된 o와 같다면 0을 반환

```
public class Person implements Comparable<Person> {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() { return name + " : " + age; }
    @Override
    public int compareTo(Person p) { return this.age - p.age; }
}
```

```
public class ComparablePerson {
    public static void main(String[] args) {
        TreeSet<Person> tree = new TreeSet<>();
        tree.add(new Person("YOON", 37));
        tree.add(new Person("HONG", 53));
        tree.add(new Person("PARK", 22));
        for(Person p : tree) {
            System.out.println(p);
        }
    }
}
```

  - 나이를 기준으로 오름차순으로 정렬
  - 만약 나이를 기준으로 내림차순으로 정렬하고 싶다면 compareTo 메소드의 정의를 "return p.age – this.page;"로 수정

Set<E> 컬렉션 클래스들

- 중복된 인스턴스 삭제
  - List<E>를 구현하는 컬렉션 클래스는 인스턴스의 중복 삽입을 허용하지만 중복 제거 방법 존재

```
public class ConvertCollection {
    public static void main(String[] args) {
        List<String> lst = Arrays.asList("Box", "Toy", "Box", "Toy");
        ArrayList<String> list = new ArrayList<>(lst);
        for(String s : list) {
            System.out.println(s.toString());
        }
        System.out.println();
        HashSet<String> set = new HashSet<>(list);
        list = new ArrayList<>(set);
        for(String s : list) {
            System.out.println(s.toString());
        }
    }
}
```