컬렉션

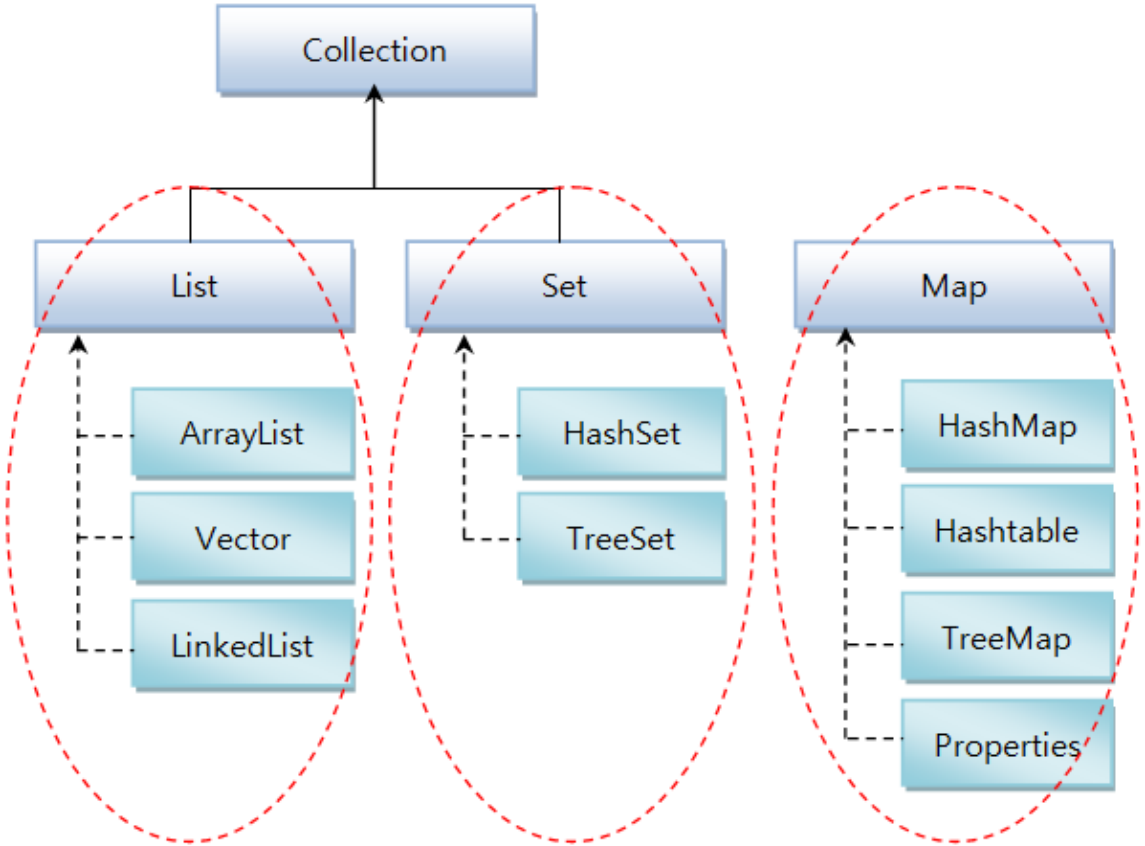
컴퓨터공학전공
박요한

수업내용

- 컬렉션 프레임워크의 이해
- Collection<E> 인터페이스
  - ✓List
  - ✓Set
  - ✓Queue
- Map(K,V) 인터페이스
  - ✓Hash

컬렉션 프레임워크 소개

- 컬렉션 프레임워크의 주요 인터페이스



List 컬렉션

- List 컬렉션의 특징
  - ✓특징
    - 인덱스로 관리
    - 중복해서 객체 저장 가능
  - ✓구현 클래스
    - ArrayList
    - LinkedList
- ArrayList vs LinkedList
  - ✓

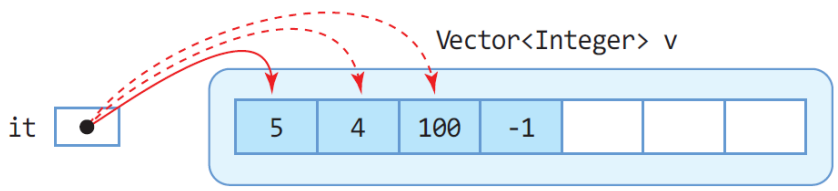
컬렉션의 순차 검색을 위한 Iterator

- Iterator<E> 인터페이스
  - ✓ Vector<E>, ArrayList<E>, LinkedList<E>가 상속받는 인터페이스
    - 리스트 구조의 컬렉션에서 요소의 순차 검색을 위한 메소드 포함
  - ✓ Iterator<E> 인터페이스 메소드

메소드	설명
boolean hasNext()	방문할 요소가 남아 있으면 true 리턴
E next()	다음 요소 리턴
void remove()	마지막으로 리턴된 요소 제거

  - ✓ iterator() 메소드 : Iterator 객체 반환
    - Iterator 객체를 이용하여 인덱스 없이 순차적 검색 가능

```
Vector<Integer> v = new Vector<Integer>();
Iterator<Integer> it = v.iterator();
while(it.hasNext()) { // 모든 요소 방문
   int n = it.next(); // 다음 요소 리턴
   …
}
```



```
class IteratorUsage
{
   public static void main(String[] args)
   {
      LinkedList<String> list=new LinkedList<String>();
      list.add("First");
      list.add("Second");
      list.add("Third");
      list.add("Fourth");

      Iterator<String> itr=list.iterator();

      System.out.println("반복자를 이용한 1차 출력과 ₩"Third₩" 삭제");
      while(itr.hasNext())
      {
         String curStr=itr.next();
         System.out.println(curStr);
         if(curStr.compareTo("Third")==0)
            itr.remove();
      }

      System.out.println("₩n₩"Third₩" 삭제 후 반복자를 이용한 2차 출력

      itr=list.iterator();
      while(itr.hasNext())
         System.out.println(itr.next());
   }
}
```

```
class UsefulIterator
{
   public static void main(String[] args)
   {
      HashSet<String> set=new HashSet<String>();
      set.add("First");
      set.add("Second");
      set.add("Third");
      set.add("Fourth");

      Iterator<String> itr=set.iterator();

      System.out.println("반복자를 이용한 1차 출력과 ₩"Third₩" 삭제");
      while(itr.hasNext())
      {
         String curStr=itr.next();
         System.out.println(curStr);
         if(curStr.compareTo("Third")==0)
            itr.remove();
      }

      System.out.println("₩n₩"Third₩" 삭제 후 반복자를 이용한 2차 출력 ");

      itr=set.iterator();
      while(itr.hasNext())
         System.out.println(itr.next());
   }
}
```

멤버를 입력받고 출력, 삭제하는 프로그램

```
public class MemberArrayList {
		private ArrayList<Member> arrayList;
		public MemberArrayList() {
		arrayList = new ArrayList<Member>();
		}

		public void addMember(Member member) {
				arrayList.add(member);
		}
public boolean removeMember(int memberId) {
		for(int i=0; i<arrayList.size(); i++) {
		Member member = arrayList.get(i);
		int tempId = member.getMemberId();
				if(memberId==tempId) {
						arrayList.remove(i);
						return true;
				}
		}
		System.out.println(memberId + "가 존재하지 않습니다.");
		return false;
		}

		public void showAll() {

		for(Member member:arrayList) {
				System.out.println(member);
				}
				System.out.println();
		}
}
```

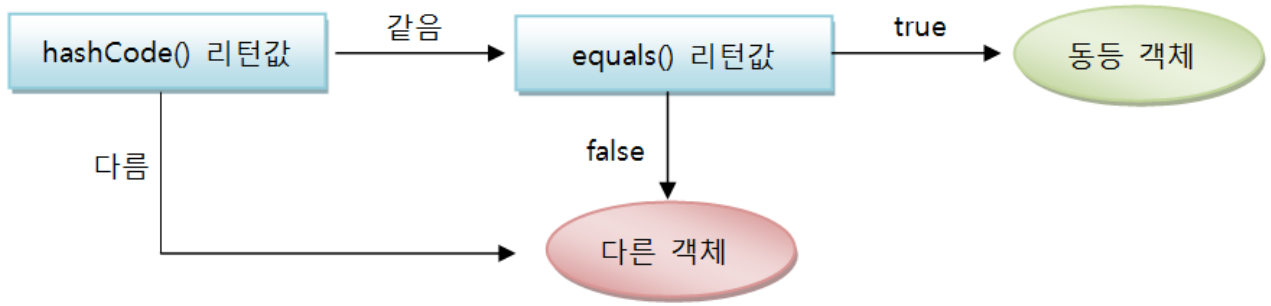
Set 컬렉션

- HashSet

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

✓특징

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



HashSet<E>

- List<E>를 구현하는 클래스들과 달리 Set<E>를 구현하는 클래스들은 데이터의 저장순서를 유지하지 않는다.
- List<E>를 구현하는 클래스들과 달리 Set<E>를 구현하는 클래스들은 데이터의 중복저장을 허용하지 않는다. 단, 동일 데이터에 대한 기준은 프로그래머가 정의
- 즉, Set<E>를 구현하는 클래스는 '집합'의 성격을 지닌다.

```
public static void main(String[] args)
{
    HashSet<String> hSet=new HashSet<String>();
    hSet.add("First");
    hSet.add("Second");
    hSet.add("Third");
    hSet.add("First");

    System.out.println("저장된 데이터 수 : "+hSet.size());

    Iterator<String> itr=hSet.iterator();
    while(itr.hasNext())
        System.out.println(itr.next());
}
```

동일한 문자열 인스턴스는 저장되지 않았다. 그렇다면 동일 인스턴스를 판단하는 기준은?

실행결과

```
저장된 데이터 수 : 3
Third
Second
First
```

HashSet<E> 동일 인스턴스 판단 기준

```
class SimpleNumber
{
    int num;
    public SimpleNumber(int n)
    {
        num=n;
    }
    public String toString()
    {
        return String.valueOf(num);
    }
}
```

HashSet<E> 클래스의 인스턴스 동등비교 방법

Object 클래스에 정의되어 있는 equals 메소드의 호출결과와 hashCode 메소드의 호출결과를 참조하여 인스턴스의 동등비교를 진행

```
public static void main(String[] args)
{
    HashSet<SimpleNumber> hSet=new HashSet<SimpleNumber>();
    hSet.add(new SimpleNumber(10));
    hSet.add(new SimpleNumber(20));
    hSet.add(new SimpleNumber(20));

    System.out.println("저장된 데이터 수 : "+hSet.size());

    Iterator<SimpleNumber> itr=hSet.iterator();
    while(itr.hasNext())
        System.out.println(itr.next());
}
```

실행결과

```
저장된 데이터 수 : 3
20
10
20
```

실행결과를 보면, 동일 인스턴스의 판단기준이 별도로 존재함을 알 수 있다.

HashSet<E> 동일 인스턴스 판단 기준

- 검색 1단계
  Object 클래스의 hashCode 메소드의 반환 값을 해시 값으로 활용하여 검색의 그룹을 선택한다.
- 검색 2단계
  그룹내의 인스턴스를 대상으로 Object 클래스의 equals 메소드의 반환 값의 결과로 동등을 판단

HashSet<E>의 인스턴스에 데이터의 저장을 명령하면, 우선 다음의 순서를 거치면서 동일 인스턴스가 저장되었는지를 확인한다.

따라서 아래의 두 메소드 적절히 오버라이딩 해야 함.

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

hashCode 메소드의 구현에 따라서 검색의 성능이 달라진다. 그리고 동일 인스턴스를 판단하는 기준이 맞게 equals 메소드를 정의해야 한다.

TreeSet<E>

- TreeSet<E> 클래스는 트리라는 자료구조를 기반으로 데이터를 저장한다.
- 데이터를 정렬된 순서로 저장하며, HashSet<E>와 마찬가지로 데이터의 중복저장 않는다.
- 정렬의 기준은 프로그래머가 직접 정의한다.

Comparable vs Comparator

- TreeSet과 TreeMap은 정렬을 위해 Comparable이나 Comparator 인터페이스를 구현해야 함
- Integer, String 등은 이미 이에 대한 정의를 해둠
  - 아래 코드는 문제 없음

```
public class TreeSetTest {
        public static void main(String[] args) {
                TreeSet<String> tree = new TreeSet();

                tree.add("aaa");
                tree.add("ccc");
                tree.add("bbb");

                System.out.println(tree);
        }
}
```

Comparable vs Comparator

- 아래와 같이 내가 만든 클래스를 제네릭 타입으로 사용하는 경우 Member클래스에서 Comparable이나 Comparator 인터페이스를 구현해 주어야 함

```
private TreeSet<Member> treeSet;

public MemberTreeSet() {
treeSet = new TreeSet<Member>(new Member());
}
```

```
public class Member implements Comparable<Member>, Comparator<Member>{ ……

//Comparable 인터페이스는 compareTo 메소드 구현
public int compareTo(Member member) {
return (this.memberName.compareTo(member.memberName)); } //이름 기준

//Comparator 인터페이스는 compare 메소드 구현
public int compare(Member member1, Member member2) {
return (member1.memberId - member2.memberId); } //아이디 기준

}
```

Comparable vs Comparator

- Integer, String의 경우 이미 Comparable 인터페이스를 구현하고 있기 때문에, 정렬 기준을 바꾸고 싶은 경우 Comparator를 사용하여 나만의 기준을 정해줄 수 있음

```
class MyCompare implements Comparator<String>{

public int compare(String o1, String o2) {
// TODO Auto-generated method stub
return o1.compareTo(o2)*-1; //내림차순 정리
}
}

public class ComparatorTest {
public static void main(String[] args) {
TreeSet<String> tree = new TreeSet(new MyCompare());

tree.add("aaa");
tree.add("bbb");
tree.add("ccc");

System.out.println(tree);  } }
```