컬렉션 프레임워크
(Collection Framework)

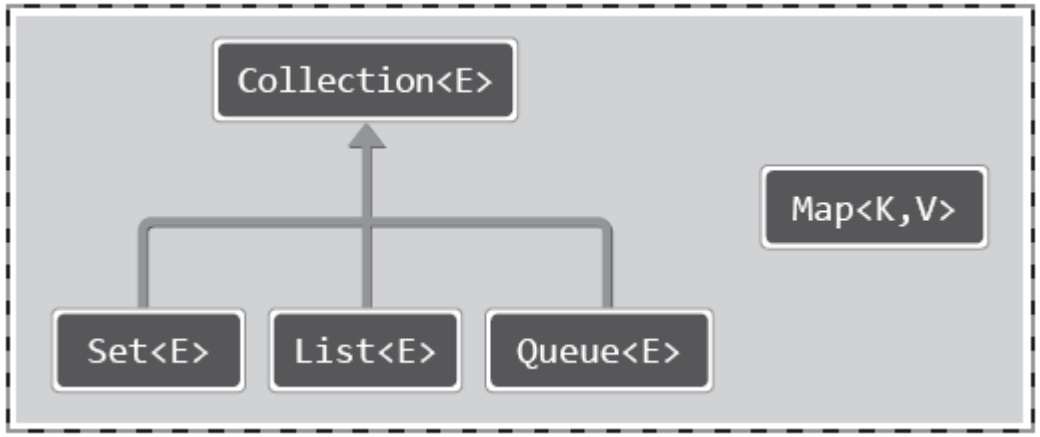
☞ 컬렉션 프레임워크의 기본적인 이해

- 프레임워크가 의미하는 바는 다음과 같다.
  - ➢ 잘 정의된, 약속된 구조와 골격
- 자바의 컬렉션 프레임워크
  - ➢ 인스턴스의 저장과 참조를 위해 잘 정의된, 클래스들의 구조
- 컬렉션 프레임워크가 제공하는 기능의 영역
  - ➢ 자료구조와 알고리즘

자바의 컬렉션 프레임워크는 별도의 구현과 이해 없이 자료구조와 알고리즘을 적용할 수 있도록 설계된 클래스들의 집합이다. 그러나 자료구조의 이론적인 특성을 안다면, 보다 적절하고 합리적인 활용이 가능하다.

☞ 컬렉션 프레임워크의 기본골격

=> 컬렉션 프레임워크의 인터페이스 구조



✓ Collection<E> 인터페이스를 구현하는 제네릭 클래스
: 인스턴스 단위의 데이터 저장 기능 제공(배열과 같이 단순 인스턴스 참조 값 저장)

✓ Map<K, V>
: key-value 구조의 인스턴스 저장 기능 제공

☞ ArrayList<E>, LinkedList<E>

- List<E> 인터페이스를 구현하는 대표적인 제네릭 클래스
  - ➢ ArrayList<E>, LinkedList<E>
- List<E> 인터페이스를 구현 클래스의 인스턴스 저장 특징
  - ➢ 동일한 인스턴스의 중복 저장을 허용한다.
  - ➢ 인스턴스의 저장순서가 유지된다.

```
public static void main(String[] args)
{
    ArrayList<Integer> list=new ArrayList<Integer>();

    /* 데이터의 저장 */
    list.add(new Integer(11));
    list.add(new Integer(22));
    list.add(new Integer(33));

    /* 데이터의 참조 */
    System.out.println("1차 참조");
    for(int i=0; i<list.size(); i++)
        System.out.println(list.get(i));   // 0이 첫 번째

    /* 데이터의 삭제 */
    list.remove(0);     // 0이 전달되었으므로 첫 번째 데이터 삭제
    System.out.println("2차 참조");
    for(int i=0; i<list.size(); i++)
        System.out.println(list.get(i));
}
```

ArrayList<E>는 이름이 의미하듯이 배열 기반으로 데이터를 저장한다.

```
1차 참조
11
22
33
2차 참조
22
33
```

☞ LinkedList<E>

- 데이터의 저장방식
  - ➢ 이름이 의미하듯이‘리스트’라는 자료구조를 기반으로 데이터를 저장한다.
- 사용방법
  - ➢ ArrayList<E>의 사용방법과 거의 동일하다! 다만, 데이터를 저장하는 방식에서 큰 차이가 있을 뿐이다.
  - ➢ 대부분의 경우 ArrayList<E>를 대체할 수 있다.

```
public static void main(String[] args)
{
    LinkedList<Integer> list=new LinkedList<Integer>();

    /* 데이터의 저장 */
    list.add(new Integer(11));
    list.add(new Integer(22));
    list.add(new Integer(33));

    /* 데이터의 참조 */
    System.out.println("1차 참조");
    for(int i=0; i<list.size(); i++)
        System.out.println(list.get(i));

    /* 데이터의 삭제 */
    list.remove(0);
    System.out.println("2차 참조");
    for(int i=0; i<list.size(); i++)
        System.out.println(list.get(i));
}
```

```
1차 참조
11
22
33
2차 참조
22
33
```

☞ ArrayList<E>와 LinkedList<E>의 차이점

ArrayList<E>의 특징, 배열의 특징과 일치한다.

• 저장소의 용량을 늘리는 과정에서 많은 시간이 소요된다.	ArrayList<E>의 단점
• 데이터의 삭제에 필요한 연산과정이 매우 길다.	ArrayList<E>의 단점
• 데이터의 참조가 용이해서 빠른 참조가 가능하다.	ArrayList<E>의 장점

LinkedList<E>의 특징, 리스트 자료구조의 특징과 일치한다.

• 저장소의 용량을 늘리는 과정이 간단하다.	LinkedList<E>의 장점
• 데이터의 삭제가 매우 간단하다.	LinkedList<E>의 장점
• 데이터의 참조가 다소 불편하다.	LinkedList<E>의 단점

☞ Iterator를 이용한 인스턴스의 순차적 접근

Iterator<E> 인터페이스

- Collection<E> 인터페이스에는 iterator라는 이름의 메소드가 다음의 형태로 정의
  - Iterator<E> iterator( ) { . . . . }
- iterator 메소드가 반환하는 참조값의 인스턴스는 Iterator<E> 인터페이스를 구현하고 있다.
- iterator 메소드가 반환한 참조 값의 인스턴스를 이용하면, 컬렉션 인스턴스에 저장된 인스턴스의 순차적 접근이 가능함.
- iterator 메소드의 반환형이 Iterator<E>이니, 반환된 참조 값을 이용해서 Iterator<E>에 선언된 함수들만 호출하면 된다.

Iterator<E> 인터페이스에 정의된 메소드

• boolean hasNext( )	참조할 다음 번 요소(element)가 존재하면 true를 반환
• E next()	다음 번 요소를 반환
• void remove()	현재 위치의 요소를 삭제

☞ Iterator의 사용 예

```
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();   // iterator 메소드가 생성하는 인스턴스를 가리켜 '반복자'라 한다.

    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());
}
```

```
반복자를 이용한 1차 출력과 "Third" 삭제
First
Second
Third
Fourth

"Third" 삭제 후 반복자를 이용한 2차 출력
First
Second
Fourth
```

☞ '반복자'를 사용하는 이유

- 반복자를 사용하면, 컬렉션 클래스의 종류에 상관없이 동일한 형태의 데이터 참조방식을 유지할 수 있다.
- 따라서, 컬렉션 클래스의 교체에 큰 영향이 없다.
- 컬렉션 클래스 별 데이터 참조방식을 별도로 확인할 필요가 없다.

```
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());
}
```

왼편은 앞서 소개한 예제이다. 그런데, 이 예제는 반복자를 사용했기 때문에, LinkedList<E>가 어울리지 않아서, 컬렉션 클래스를 HashSet<E>로 변경해야 할 때, 다음과 같이 변경이 매우 용이하다.

```
LinkedList<String> list
                = new LinkedList<String>( );
```

```
HashSet<String> set
                = new HashSet<String>( );
```

☞ 컬렉션 클래스를 이용한 정수의 저장

```
ArrayList<int> arr = new ArrayList<int>( ); //error
LinkedList<int> link = new LinkedList<int>( ); //error
```

기본 자료형 정보를 이용해서 제네릭 인스턴스 생성 불가능! 따라서 Wraper 클래스를 기반으로 컬렉션 인스턴스를 생성한다.

```
public static void main(String[] args)
{
    LinkedList<Integer> list=new LinkedList<Integer>();
    list.add(10);       // Auto Boxing
    list.add(20);       // Auto Boxing
    list.add(30);       // Auto Boxing

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

    while(itr.hasNext())
    {
        int num=itr.next();     // Auto Unboxing
        System.out.println(num);
    }
}
```

```
10
20
30
```

Auto Boxing과 Auto Unboxing의 도움으로 정수 단위의 데이터 입출력이 매우 자연스럽다!

☞ Set<E> 인터페이스의 특성과 HashSet<E> 클래스

- List<E>를 구현하는 클래스들과 달리 Set<E>를 구현하는 클래스들은 데이터의 저장순서를 유지하지 않는다.
- List<E>를 구현하는 클래스들과 달리Set<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?

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

☞ 동일 인스턴스의 판단기준 관찰을 위한 예

```
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 정의되어있는 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());
}
```

```
저장된 데이터 수 : 3
20
10
20
```

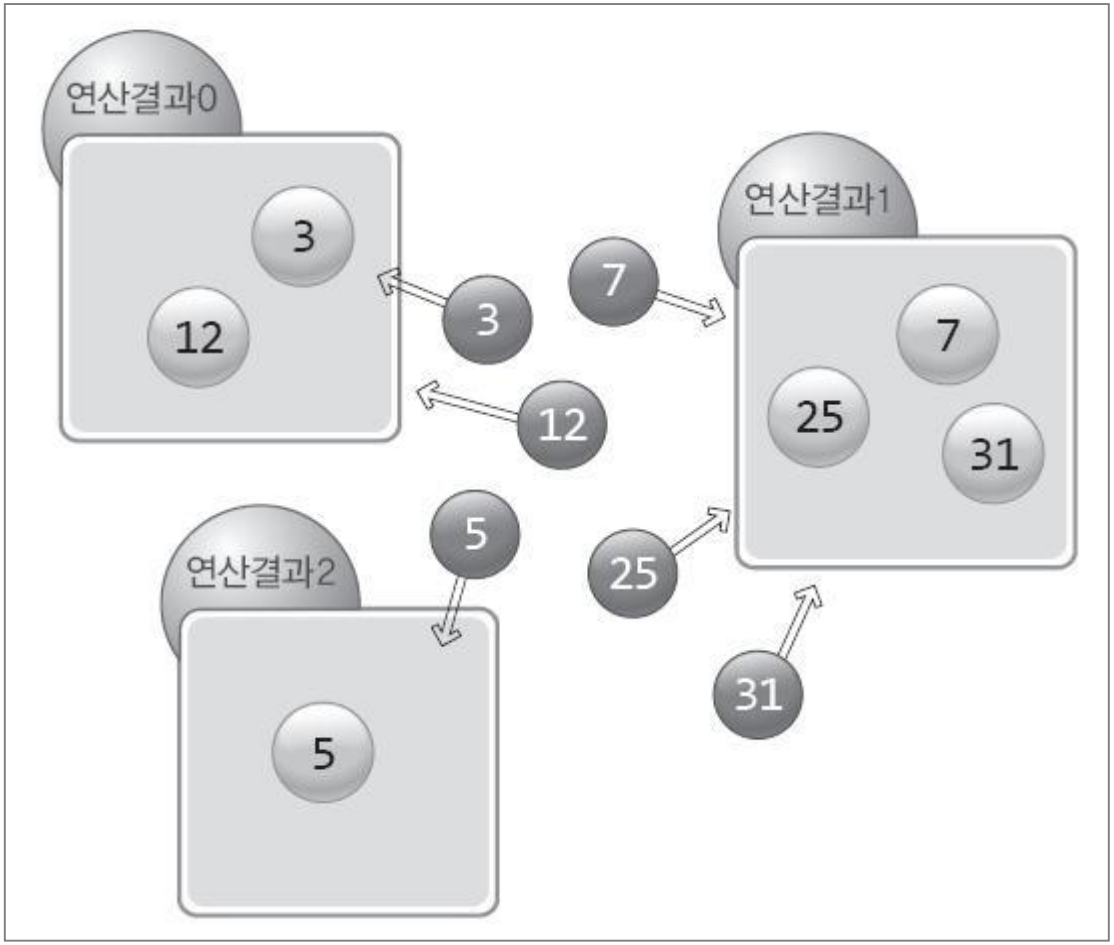
실행결과를 보면, 동일 인스턴스의 판단기준이 별도로 존재함을 알 수 있다.

☞ 해시 알고리즘의 이해(데이터의 구분)

3, 5, 7, 12, 25, 31 데이터

num % 3

해시알고리즘 : 해시 알고리즘은 이렇듯 간단하게 디자인 될 수도 있다.



알고리즘적용결과

해시 알고리즘은 데이터의 분류에 사용이 된다. 데이터를 3으로 나머지 연산하였을 때 얻게 되는 반환값을 '해시값'으로 하여 총 세 개의 부류를 구성하였다.

이렇게 분류해 놓으면, 데이터의 검색이 빨라진다. 정수12가 저장되어 있는지 확인한다고 했을때 문제 정수12의 해시값을 구한다. 그 다음에 해시값에 해당하는 부류에서만 정수12의 존재유무를 확인하면 된다.

☞ HashSet<E> 클래스의 동등비교

- 검색1단계

  Object 클래스의 hashCode 메소드의 반환 값을 해시값으로 활용하여 검색의 그룹을 선택한다.

- 검색2단계

  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.

☞ HashSet<E> 클래스의 활용의 예

```
class SimpleNumber
{
    int num;
    . . . . . .
    public int hashCode()
    {
        return num%3;
    }
    public boolean equals(Object obj)
    {
        SimpleNumber comp=(SimpleNumber)obj;
        if(comp.num==num)
            return true;
        else
            return false;
    }
}
```

%3 의 연산이 해시알고리즘이니, 해시 그룹은 세부류로 나뉜다.

인스턴스 변수 num이 같을때 동일 인스턴스로 간주하는 내용이 담겨있다.

```
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());
}
```

```
저장된 데이터 수 : 2
20
10
```

☞ TreeSet<E> 클래스의 이해와 활용

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

```
public static void main(String[] args)
{
    TreeSet<Integer> sTree=new TreeSet<Integer>();
    sTree.add(1);
    sTree.add(2);
    sTree.add(4);
    sTree.add(3);
    sTree.add(2);

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

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

데이터는 정렬되어 저장이 되며, 때문에 iterator 메소드의 호출로 생성된 반복자는 오름차순의 데이터참조를 진행한다.

```
저장된 데이터 수 : 4
1
2
3
4
```

출력순서가 정렬되어 있음에 주목해야 한다! 이것이 TreeSet<E>의 특징이다.

☞ 정렬의 기준을 정하는 Comparable<T> 인터페이스

- TreeSet<E> 인스턴스에 저장이 되려면 Comparable<T> 인터페이스를 구현해야 한다.
- Comparable<T> 인터페이스의 유일한 메소드는 int compareTo(T obj); 이다.
- compareTo 메소드는 다음의 기준으로 구현을 해야 한다.
  - 인자로 전달된 obj가 작다면 양의 정수를 반환해라.
  - 인자로 전달된 obj가 크다면 음의정수를 반환해라.
  - 인자로 전달된 obj와 같다면 0을 반환해라.

'작다', '크다', '같다'의 기준은 프로그래머가 결정!

```
class Person implements Comparable<Person>
{
    String name;
    int age;
    . . . . .
    public int compareTo(Person p)
    {
        if(age>p.age)
            return 1;
        else if(age<p.age)
            return -1;
        else
            return 0;
    }
}
```

```
public static void main(String[] args)
{
    TreeSet<Person> sTree=new TreeSet<Person>();
    sTree.add(new Person("Lee", 24));
    sTree.add(new Person("Hong", 29));
    sTree.add(new Person("Choi", 21));

    Iterator<Person> itr=sTree.iterator();
    while(itr.hasNext())
        itr.next().showData();
}
```

```
Choi 21
Lee 24
Hong 29
```

Person 클래스의 comareTo 메소드는 정렬의 기준을 '나이의 많고 적음'으로 구현하였다.

☞ Map<K, V> 인터페이스와 HashMap<K, V> 클래스

- Map<K, V> 인터페이스를 구현하는 컬렉션 클래스는 key-value 방식의 데이터 저장을 한다.
- value는 저장할 데이터를 의미하고, key는value를 찾는 열쇠를 의미한다.
- Map<K, V>를 구현하는 대표적인 클래스로는 HashMap<K, V>와 TreeMap<K, V>가 있다.
- TreeMap<K, V>는 정렬된 형태로 데이터가 저장된다.

```
public static void main(String[] args)
{
    HashMap<Integer, String> hMap=new HashMap<Integer, String>();

    hMap.put(new Integer(3), "나삼번");
    hMap.put(5, "윤오번");
    hMap.put(8, "박팔번");

    System.out.println("6학년 3반 8번 학생 : "+hMap.get(new Integer(8)));
    System.out.println("6학년 3반 5번 학생 : "+hMap.get(5));
    System.out.println("6학년 3반 3번 학생 : "+hMap.get(3));

    hMap.remove(5);     // 5번 학생 전학 감
    System.out.println("6학년 3반 5번 학생 : "+hMap.get(5));
}
```

```
6학년 3반 8번 학생 : 박팔번
6학년 3반 5번 학생 : 윤오번
6학년 3반 3번 학생 : 나삼번
6학년 3반 5번 학생 : null
```

☞ TreeMap<K, V> 클래스의 활용 예

```
public static void main(String[] args)
{
    TreeMap<Integer, String> tMap=new TreeMap<Integer, String>();
    tMap.put(1, "data1");
    tMap.put(3, "data3");
    tMap.put(5, "data5");
    tMap.put(2, "data2");
    tMap.put(4, "data4");

    NavigableSet<Integer> navi=tMap.navigableKeySet();

    System.out.println("오름차순 출력...");
    Iterator<Integer> itr=navi.iterator();
    while(itr.hasNext())
        System.out.println(tMap.get(itr.next()));

    System.out.println("내림차순 출력...");
    itr=navi.descendingIterator();
    while(itr.hasNext())
        System.out.println(tMap.get(itr.next()));
}
```

키들의 모임을 얻어와야
키를 순차적으로 검색할 수 있다.
navigableKeySet 메소드는 키들이 모여있는
컬렉션 인스턴스의 참조 값을 반환!

```
오름차순 출력...
data1
data2
data3
data4
data5
내림차순 출력...
data5
data4
data3
data2
data1
```

위의 예제에서 보이듯이 descendingIterator 메소드는 내림차순의 검색을 위한 반복자를 생성한다.
그리고 NavigableSet<E> 클래스도 Set<E> 클래스를 상속하는 컬렉션클래스이다!