11. 컬렉션과 제네릭

이 장에서는 자바에서 객체들을 저장할 수 있는 동적인 배열공간을 갖는 컬렉션 프레임워크(Collection Framework)과 제네릭(Generic) 기능에 대해 설명하기로 하겠습니다.

자바 초기 버전에서는 객체를 저장할 수 있도록 Vector, Enumeration 클래스 등 간단한 클래스만 제공되었습니다. 그 후 JDK 1.2 버전에서 많은 컬렉션 인터페이스와 클래스들이 추가되었고 이를 컬렉션 프레임워크라는 이름으로 사용하고 있습니다. 이 장에서는 객체를 저장하는 다양한 방법과 컬렉션의 종류들에 대해서 설명하고 있습니다.

Java SE 5(JDK 1.5) 버전에서는 제네릭 기능이 도입되었습니다. 이로 인해 컬렉션 객체에 형 안정성(Type Safety)을 제공할 수 있게 되었습니다. 이 장에서는 제네릭이 무엇이며, 이를 사용했을 때의 장점은 무엇인지 설명하고 있습니다.

11장의 주요 내용입니다.

- Set
- List
- Map
- Iterator
- Comparator
- Comparable
- 객체 동등 비교
- JDK1.1의 Collections
- 제네릭과 형 안정성

11.1. Collection Framework

11.1.1. Collection

자바 프로그램에서 객체를 객체들의 모음으로 관리할 수 있는 방법 제공하는 클래스들을 컬렉션 프레임워크라고 부릅니다. 그리고 객체를 저장할 수 있는 자료구조 클래스들을 컬렉션이라고 합니다. 컬렉션은 배열과 유사하지만 데이터를 저장/조회/수정/삭제 작업을 쉽게 처리할 수 있으며, 동적인 크기를 갖는 장점이 있습니다. 이들 컬렉션 계열은 Collection, Set, List, Map 등의 인터페이스가 있으며 이를 구현한 클래스를 이용하면 객체를 저장할 수 있습니다.

자바의 컬렉션 프레임워크들에는 배열구조(객체들을 배열처럼 인덱스를 이용하여 관리), 리스트구조(순차적 접근 가능), 맵 구조(객체 저장 시 key/value 쌍으로 저장) 등이 있습니다. 이들 클래스들은 기존의 배열에 비해 높은 성능을 보장하며, 저장된 객체들을 관리할 수 있는 다양한 메서드들이 포함되어 있습니다.

컬렉션에 저장된 객체를 엘리먼트(Element)라 부르며 Set, List, Map은 각각 Collection 인터페이스를 상속받습니다. Collection 인터페이스는 순서가 없고 중복을 허락하는 구조입니다. Set 인터페이스는 순서가 없고, 중복도 허락하지 않는 구조를 갖습니다. Set 관련 클래스들은 데이터를 저장할 때 저장하는 순서를 보장하지 않으며, 같은 엘리먼트(Element)가 두 개 이상 저장될 수 없습니다. List는 입력되는 엘리먼트(Element)들 사이에 순서가 정해져 있으며, 입력되는 순서가 다르면 같은 데이터도 저장할 수 있도록 중복을 허락하는 구조를 갖습니다. List 관련 클래스들은 엘리먼트를 저장하는 순서대로 조회할 수 있으며, 같은 엘리먼트(Element)를 두 개 이상 저장할 수 있습니다.

Collection, Set, List, Map은 모두 인터페이스이기 때문에 직접 객체를 생성하지 못하며 구현된 클래스를 이용하여 객체를 생성합니다. 구현된 클래스들 중에서 Set 계열에는 HashSet클래스, List 계열에는 ArrayList와 LinkedList가 주로 사용됩니다. 이들 인터페이스 또는 클래스들은 java.util 패키지에 있기 때문에 사용하기 위해서는 import 문이 필요합니다.

인터페이스	순서	중복	구현된 클래스
Collection	X	O	
Set	X	X	HashSet TreeSet
List	O	O	ArrayList LinkedList

Collection 인터페이스의 주요 메서드는 다음과 같습니다.

리턴타입	메서드와 설명
boolean	add(E e) 컬렉션에 엘리먼트를 추가합니다. 만일 추가되지 않을 경우 false를 리턴합니다.
void	clear() 컬렉션의 모든 엘리먼트를 삭제합니다.
boolean	contains(Object o) 컬렉션이 주어진 객체를 포함하면 true를 리턴합니다.
boolean	isEmpty() 컬렉션이 비어있으면 true를 리턴합니다.
Iterator<E>	iterator() 이 컬렉션의 iterator 객체를 반환합니다.
boolean	remove(Object o) 주어진 엘리먼트 객체를 제거합니다.
int	size() 컬렉션의 엘리먼트의 개수를 반환합니다.
Object[]	toArray() 이 컬렉션의 모든 엘리먼트들을 배열로 반환합니다.
<T> T[]	toArray(T[] a) 이 컬렉션의 모든 엘리먼트들을 주어진 타입의 배열로 반환합니다.

11.1.2. Set

Set 인터페이스는 중복된 데이터의 저장을 허용하지 않습니다. 이미 저장된 엘리먼트와 같은 엘리먼트를 저장하려고 했을 때 add() 메서드는 false를 반환합니다. Set 인터페이스를 상속 받은 인터페이스에는 SortedSet 인터페이스가 있습니다. Set은 순서를 보장하지 않지만 SortedSet은 저장된 엘리먼트들을 오름차순으로 정렬하여 보관할 수 있습니다. Set 인터페이스를 구현한 클래스들에는 AbstractSet, ConcurrentSkipListSet, CopyOnWriteArraySet, EnumSet, HashSet, JobStateReasons, LinkedHashSet, TreeSet 클래스가 있습니다. SortedSet 인터페이스를 구현한 클래스는 ConcurrentSkipListSet 클래스와 TreeSet 클래스가 있습니다.

Set 인터페이스의 주요 메서드는 다음과 같습니다.

리턴타입	메서드와 설명
boolean	add(E e) 세트에 엘리먼트를 추가합니다. 만일 추가되지 않을 경우 false를 리턴합니다.
void	clear() 세트의 모든 엘리먼트를 삭제합니다.
boolean	contains(Object o) 세트가 주어진 객체를 포함하면 true를 리턴합니다.

boolean	isEmpty() 세트 비어있으면 true를 리턴합니다.
Iterator<E>	iterator() 이 세트 iterator 객체를 반환합니다.
boolean	remove(Object o) 세트에서 주어진 엘리먼트 객체를 제거합니다.
int	size() 세트의 엘리먼트의 개수를 반환합니다.
Object[]	toArray() 이 세트의 모든 엘리먼트들을 배열로 반환합니다.
<T> T[]	toArray(T[] a) 이 세트의 모든 엘리먼트들을 주어진 타입의 배열로 반환합니다.

HashSet

HashSet 클래스는 해시 테이블에 저장된 데이터를 사용하기 위한 Set 계열 컬렉션 클래스입니다. 해시 테이블 저장소는 데이터를 저장하기 위해 유일(Unique)한 Key와 Value를 한 세트로 매핑하여 저장합니다. 이 때 키는 프로그래머가 지정하는 것이 아니고 자동으로 처리됩니다. HashSet 클래스는 Set 인터페이스를 구현한 클래스이므로 저장된 엘리먼트의 순서를 보장하지 않습니다. 저장된 엘리먼트는 hashcode 로 구분되어 저장되기 때문에 index를 통해 엘리먼트에 접근할 수 없습니다. 이렇게 HashSet 클래스는 해싱(hashing) 기술을 사용하여 엘리먼트들을 저장하게 됩니다.

해싱을 이용하면 많은 양의 데이터를 관리할 때 추가/삭제/검색 등에 있어서 순차적으로 데이터를 관리하는 것 보에 비하여 속도가 향상됩니다. HashSet은 정렬기능을 지원하지 않습니다. 정렬 기능을 갖는 Set 계열 클래스에는 TreeSet 클래스가 있습니다.

다음 프로그램은 HashSet 클래스를 사용하여 엘리먼트를 저장하는 Set의 사용 예를 보인 것입니다.

HashSetExample.java

```
1:  import java.util.*;
2:
3:  public class HashSetExample {
4:
5:      public static void main(String args[]) {
6:
7:          Set set = new HashSet();
8:
9:          set.add("three");
10:         set.add("one");
11:         set.add("two");
```

```
12:         set.add("four");
13:         set.add("five");
14:         set.add(new Integer(4));
15:         boolean isAdded = set.add("five");
16:
17:         System.out.println(set);
18:         System.out.println(isAdded);
19:
20:         System.out.println(set.size());
21:
22:         set.remove("two");
23:         System.out.println(set);
24:
25:         set.clear();
26:         System.out.println(set);
27:
28:         if (set.isEmpty()) {
29:            System.out.println("set is Empty");
30:         }
31:     }
32: }
```

```
Console
<terminated> HashSetExample [Java Application] C:₩Pr
[4, four, one, three, two, five]
false
6
[4, four, one, three, five]
[]
set is Empty
```

코드를 작성하면 Set과 관련된 부분에서 경고 메시지를 볼 수 있습니다. Set 인터페이스와 HashSet 클래스에 Generic 기능을 사용하지 않아 형 안정성을 제공하지 않기 때문에 보이는 경고 메시지입니다. 아직 Generic을 배우지 않았으므로 무시하고 넘어가세요.

TreeSet

TreeSet 클래스는 Set 인터페이스와 SortedSet 인터페이스를 구현한 클래스이기 때문에 정렬 기능을 지원합니다. 클래스 이름에서 알 수 있듯이 트리 형태로 데이터를 관리합니다. 엘리먼트가 저장될 때에 오름차순으로 정렬되어 저장됩니다. 정렬 기능을 포함하기 때문에 검색 속도가 떨어지지는 않습니다.

다음 프로그램은 TreeSet 클래스를 사용하여 엘리먼트를 저장하는 Set의 사용 예를 보인 것입니다.

TreeSetExample.java

```
1:  import java.util.TreeSet;
2:
3:  public class TreeSetExample {
4:
5:      public static void main(String[] args) {
6:          TreeSet<String> ts = new TreeSet<String>();
7:
```

```
8:          ts.add("hello");
9:          ts.add("java");
10:         ts.add("aaa");
11:         ts.add("computer");
12:
13:         for(String str : ts) {
14:             System.out.print(str + "\t");
15:         }
16:     }
17: }
```

예제는 저장된 데이터를 알파벳 오름차순으로 정렬된 순서로 출력합니다. TreeSet은 저장한 데이터를 정렬해서 보관합니다. 정렬 방법 변경과 관련된 내용은 Comparable 인터페이스와 Comparator 인터페이스를 설명할 때 언급됩니다.

TreeSet 클래스 뒤에 <String>이 붙은 것은 Generic 기능을 사용한 것입니다. TreeSet에 String 데이터만 저장하겠다는 의미입니다. String이 아닌 다른 객체를 add() 메서드로 추가할 수 없습니다. Generic에 관해서는 이장의 후반부 부분에서 다시 설명됩니다.

11.1.3. List

List 인터페이스는 저장된 엘리먼트들의 순서를 보장합니다. 그리고 동일 엘리먼트를 여러 개 저장할 수 있습니다. List 인터페이스를 구현한 주요 클래스들은 AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector 등이 있습니다. List 계열 클래스들은 순서를 가질 수 있습니다. 엘리먼트가 저장되었을 때 엘리먼트의 위치를 알 수 있다는 것입니다. 엘리먼트의 저장된 위치는 0부터 시작합니다.

List 인터페이스의 주요 메서드는 다음과 같습니다.

리턴타입	메서드와 설명
boolean	add(E e) 리스트에 엘리먼트를 추가합니다. 만일 추가되지 않을 경우 false를 리턴합니다.
boolean	add(int index, E e) 이 리스트의 주어진 index 위치에 엘리먼트를 추가합니다. 만일 추가되지 않을 경우 false를 리턴합니다.
void	clear() 리스트의 모든 엘리먼트를 삭제합니다.
boolean	contains(Object o) 리스트가 주어진 객체를 포함하면 true를 리턴합니다.
E	get(int index) 리스트에서 주어진 index위치의 엘리먼트를 반환합니다.

int	indexOf(Object o) 리스트가 주어진 객체의 위치를 반환합니다. 엘리먼트를 찾지 못할 경우 -1을 반환합니다.
boolean	isEmpty() 리스트가 비어있으면 true를 리턴합니다.
Iterator<E>	iterator() 이 리스트의 iterator 객체를 반환합니다.
ListIterator<E>	listIterator() 이 리스트의 list iterator 객체를 반환합니다.
E	remove(int index) 리스트에서 주어진 위치의 엘리먼트를 제거합니다.
boolean	remove(Object o) 주어진 엘리먼트 객체를 제거합니다.
E	set(int index, E element) 리스트의 index위치 엘리먼트를 주어진 element로 대체합니다.
int	size() 컬렉션의 엘리먼트의 개수를 반환합니다.
Object[]	toArray() 이 컬렉션의 모든 엘리먼트들을 배열로 반환합니다.
<T> T[]	toArray(T[] a) 이 컬렉션의 모든 엘리먼트들을 주어진 타입의 배열로 반환합니다.

ArrayList

ArrayList 클래스는 엘리먼트가 저장되기 위한 공간을 동적으로 할당하는 배열 구조를 갖는 List 계열 컬렉션 클래스입니다. 자바의 배열은 고정된 크기를 지정해야 하며, 배열의 크기를 늘리거나 줄이기 위해서는 새로운 배열을 생성해야 합니다. 그리고 배열은 중간에 새로운 데이터를 삽입하거나 삭제할 때 삽입/삭제가 발생하는 인덱스 뒤의 데이터들을 처리해 줘야 하므로 매우 불편합니다. ArrayList를 사용하면 엘리먼트가 저장되는 크기를 동적으로 할당할 수 있어 필요에 따라 자동으로 크기가 늘어나거나 줄어듭니다. 뿐만 아니라 엘리먼트를 추가하거나 삭제하는 다양한 메서드를 제공합니다.

다음 프로그램은 ArrayList의 사용 예를 보인 것입니다. 여러 메서드들의 사용에 주의해서 보시기 바랍니다.

ArrayListExample.java

```
1:  import java.util.*;
2:
3:  public class ArrayListExample {
4:
5:     public static void main(String[] args)   {
6:        List list = new ArrayList();
7:
8:        list.add("one");
```

```
9:         boolean a=list.add("second");
10:        list.add("3rd");
11:        list.add(new Integer(4));
12:        list.add(new Float(5.0f));
13:        boolean b=list.add("second");
14:        list.add(new Integer(4));
15:        list.add("SECOND");
16:
17:        System.out.println(a);
18:        System.out.println(b);
19:        System.out.println(list);
20:
21:        list.remove(0);
22:        System.out.println(list);
23:
24:        Object o=list.get(1);
25:        System.out.println(o);
26:
27:        int i=list.indexOf("second");
28:        System.out.println("second = "+i);
29:
30:        list.clear();
31:        if (list.isEmpty())          {
32:           System.out.println("list is Empty");
33:        }
34:     }
35: }
```

```
Console
<terminated> ArrayListExample [Java Application] C:₩Program Files₩Java₩j
true
true
[one, second, 3rd, 4, 5.0, second, 4, SECOND]
[second, 3rd, 4, 5.0, second, 4, SECOND]
3rd
second = 0
list is Empty
```

LinkedList

LinkedList 클래스는 Java SE 5(JDK 1.5)버전에 추가된 Queue 인터페이스를 구현하였고, Java SE 6(JDK 1.6)버전에서는 Deque 인터페이스를 구현하면서 많은 새로운 메서드들이 추가되었습니다. LinkedList 클래스는 add() 메서드 외에 addFirst(), addLast() 메서드를 통해서 리스트의 맨 처음이나 마지막에 엘리먼트를 추가할 수 있습니다.
LinkedList는 Queue 인터페이스를 구현했기 때문에 peek() 메서드와 poll() 메서드 등을 사용할 수 있습니다. peek() 메서드는 데이터를 조회한 후에 리스트에서 데이터를 삭제하지 않지만, poll() 메서드는 데이터를 조회한 후에 리스트로부터 데이터를 삭제합니다.
LinkedList는 Deque 인터페이스의 push()와 pop() 메서드를 사용할 수도 있습니다.

다음 프로그램은 LinkedList 예입니다.

LinkedListExample.java

```
1:  import java.util.LinkedList;
2:
```

```
3:  public class LinkedListExample {
4:     public static void main(String[] args) {
5:        LinkedList<String> list = new LinkedList<String>();
6:
7:        list.add("hello");
8:        list.add("java");
9:        list.add("banana");
10:       list.addFirst("apple");
11:       list.addLast("zoo");
12:
13:       System.out.println("list data : " + list);
14:
15:       list.remove();      //head 엘리먼트 삭제
16:       System.out.println("list data after remove() : " + list);
17:
18:       list.remove(2);     //2번 인덱스 엘리먼트 삭제
19:       System.out.println("list data after remove(2) : " + list);
20:
21:       list.set(1, "new element"); //1번째 엘리먼트 변경
22:       System.out.println("list data after set() : " + list);
23:
24:       String str1 = list.peek();   //엘리먼트 조회
25:       System.out.println("str1 : " + str1);
26:       System.out.println("list data after peek() : " + list);
27:
28:       String str2 = list.poll();   //엘리먼트 조회 후 삭제
29:       System.out.println("str2 : " + str2);
30:       System.out.println("list data after poll() : " + list);
31:    }
32: }
```

```
Console
<terminated> LinkedListExample [Java Application] C:₩Program Files₩Java₩jre1.8.0_91₩l
list data : [apple, hello, java, banana, zoo]
list data after remove() : [hello, java, banana, zoo]
list data after remove(2) : [hello, java, zoo]
list data after set() : [hello, new element, zoo]
str1 : hello
list data after peek() : [hello, new element, zoo]
str2 : hello
list data after poll() : [new element, zoo]
```

11.1.4. Map

Map 인터페이스는 저장한 엘리먼트를 대체 키(key) 값으로 지정하여 관리할 수 있는 구조입니다. 즉, 엘리먼트들을 key/value를 하나의 쌍으로 저장하여 관리할 수 있는 컬렉션입니다. 같은 키(key)로 엘리먼트를 두 개 이상 저장할 수는 없지만 키(key)가 다르면 같은 엘리먼트를 저장할 수 있습니다. Map 인터페이스를 구현한 클래스에는 AbstractMap,

Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap 클래스들이 있습니다.

Map 인터페이스의 주요 메서드는 다음과 같습니다.

리턴타입	메서드와 설명
void	clear() map에 저장되어 있는 모든 엘리먼트를 삭제합니다.
boolean	containsKey(Object key) 해당하는 키를 포함하고 있으면 true를 반환합니다.
boolean	containsValue(Object value) 해당하는 객체를 포함하고 있으면 true를 반환합니다.
Set<Map.Entry<K, V>>	entrySet() Map.Entry 객체를 엘리먼트로 갖는 Set 객체를 반환합니다.
V	get(Object key) 키를 갖는 엘리먼트를 반환합니다. 키와 매핑되는 엘리먼트가 없으면 null을 반환합니다.
boolean	isEmpty() 키/값 매핑이 비어있으면 true 반환합니다.
Set<K>	keySet() map 저장된 키들을 Set 객체로 반환합니다.
V	put(K key, V value) 키/값 매핑을 저장합니다.
void	putAll(Map<? extends K,? extends V> m) map에 저장된 모든 키/값 매핑을 현재 맵에 복사 합니다.
V	remove(Object key) 해당 키를 갖는 키/값 매핑을 삭제합니다.
int	size() 키/값 매핑 개수를 반환합니다.
Collection<V>	values() map 이 포함하고 있는 값을 Collection 객체로 반환합니다.

HashMap

HashMap 클래스는 Map 계열 컬렉션 클래스입니다. 이 클래스도 엘리먼트를 저장하기 위해 해시 테이블을 사용합니다. 엘리먼트를 저장할 때 키/값을 매핑하여 저장합니다.

다음 프로그램은 HashMap 클래스의 사용 예를 보인 것입니다.

HashMapExample.java

```
1:  import java.util.Date;
2:  import java.util.HashMap;
```

```
3:  import java.util.Map;
4:  import java.util.Set;
5:
6:  public class HashMapExample {
7:     public static void main(String[] args) {
8:        Map maps = new HashMap();
9:
10:       String s1 = new String("홍길동");
11:       maps.put("name", s1);
12:       maps.put("hiredate", new Date());
13:       maps.put("salary", 20000);
14:
15:       System.out.println(maps);
16:
17:       System.out.println();
18:       System.out.println(maps.get("hiredate"));
19:       System.out.println(maps.get("salary"));
20:       System.out.println(maps.get("name"));
21:
22:       System.out.println();
23:       //map안의 엘리먼트를 entrySet() 메서드를 이용하여 조회
24:       Set<Map.Entry<String, Object>> s = maps.entrySet();
25:       for(Map.Entry<String, Object> me : s) {
26:          System.out.println(me.getKey() + " : " + me.getValue());
27:       }
28:
29:       System.out.println();
30:       //keySet() 메서드로 map 키를 리턴받고 get(key) 메서드로 조회
31:       Set<String> ss = maps.keySet();
32:       for(String key : ss) {
33:          System.out.println(key + " :: " + maps.get(key));
34:       }
35:    }
36: }
```

```
Console
<terminated> HashMapExample [Java Application] C:₩Program Files₩Java₩jre1.8.0_91₩bin₩javaw.exe (201
{name=홍길동, salary=20000, hiredate=Sat Jul 16 17:45:19 KST 2016}

Sat Jul 16 17:45:19 KST 2016
20000
홍길동

name : 홍길동
salary : 20000
hiredate : Sat Jul 16 17:45:19 KST 2016

name :: 홍길동
salary :: 20000
hiredate :: Sat Jul 16 17:45:19 KST 2016
```

TreeMap

TreeMap 클래스는 Map 인터페이스와 SortedMap 인터페이스를 구현한 클래스입니다. 정렬 기능이 지원되는 Map 계열 컬렉션 클래스입니다. 이 클래스도 엘리먼트를 저장할 때 키/값을 매핑하여 저장합니다. 데이터를 저장할 대 오름차순으로 저장되며, 데이터 조회 시 HashSet에 비하여 더 빠른 성능을 보장합니다.

다음 프로그램은 TreeMap 클래스의 사용 예를 보인 것입니다.

TreeMapExample.java

```
1:  import java.util.Map;
2:  import java.util.Set;
3:  import java.util.TreeMap;
4:
5:  public class TreeMapExample {
6:
7:      public static void main(String[] args) {
8:
9:          Map<String, Integer> accounts = new TreeMap<String, Integer>();
10:
11:         accounts.put("홍길동", 10000);
12:         accounts.put("이순신", 50000);
13:         accounts.put("정준수", 90000);
14:         accounts.put("허현수", 70000);
15:
16:         System.out.println(accounts);
17:
18:         System.out.println();
19:         System.out.println("허현수: " + accounts.get("허현수"));
20:
21:         System.out.println();
22:         Set<Map.Entry<String, Integer>> s = accounts.entrySet();
23:         for(Map.Entry<String, Integer> me : s) {
24:            System.out.println(me.getKey() + " : " + me.getValue());
25:         }
26:
27:         System.out.println();
28:         Set<String> ss = accounts.keySet();
29:         for(String key : ss) {
30:            System.out.println(key + " :: " + accounts.get(key));
31:         }
32:     }
33: }
```

```
Console
<terminated> TreeMapExample [Java Application] C:₩Program Files₩Java₩jre1.8.0_
{이순신=50000, 정준수=90000, 허현수=70000, 홍길동=10000}

허현수: 70000

이순신 : 50000
정준수 : 90000
허현수 : 70000
홍길동 : 10000

이순신 :: 50000
정준수 :: 90000
허현수 :: 70000
홍길동 :: 10000
```

11.1.5. Iterator

컬렉션에 저장되어 있는 엘리먼트를 검색하는 절차를 Iteration이라 합니다. Iterator 인터페이스는 컬렉션에 저장된 엘리먼트를 순차적으로 하나씩 접근하고자 할 때 사용됩니다. JDK 1.2 이전 버전에는 Enumeration 인터페이스를 사용하였습니다. Iterator 인터페이스는 저장된 엘리먼트를 제거할 수 있는 기능이 추가되었습니다.
Set 계열에서는 Collection 인터페이스에 정의된 iterator() 메서드를 통하여 Iterator 객체를 반환받아 사용하며, List를 사용했을 때는 listIterator() 메서드를 통해 ListIterator 객체를 반환받아 사용할 수 있습니다. ListIterator를 사용하면 엘리먼트를 앞 또는 뒤의 원하는 방향으로 검색할 수 있습니다. Iterator는 인덱스를 사용해서 엘리먼트에 접근하는 것이 아닙니다. 저장된 데이터를 하나씩 순서대로 접근하면서 얻어오거나 삭제할 수 있습니다.

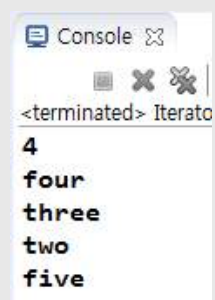
다음은 Iterator 인터페이스의 주요 메서드입니다.

- boolean hasNext() : 다음 엘리먼트를 포함하고 있으면 true를 반환합니다.
- E next() : 다음 엘리먼트를 반환합니다.
- void remove() : iterator에 의해 반환된 마지막 엘리먼트를 삭제합니다.

다음 프로그램은 Iterator를 사용하여 Set 계열 클래스로부터 데이터를 검색하는 예를 보인 것입니다.

IteratorExample.java

```
1:  import java.util.HashSet;
2:  import java.util.Iterator;
3:  import java.util.Set;
4:
5:  public class IteratorExample {
6:
7:      public static void main(String args[]) {
8:
9:          Set set = new HashSet();
10:         set.add("three");
11:         set.add("two");
12:         set.add("four");
13:         set.add("five");
14:         set.add(new Integer(4));
15:
16:         Iterator it = set.iterator();
17:         while( it.hasNext() ) {
18:             System.out.println(it.next());
19:         }
20:     }
21: }
```



11.2. 객체 비교

자바의 기본형 변수의 동등 비교는 == 또는 != 연산자를 이용하면 됩니다. 그리고 크기 비교는 >, >=, <, <= 등의 연산자를 이용하면 됩니다. 그러나 비교 대상이 객체일 경우에는 이등 연산자를 이용해 비교했을 경우 객체의 값을 이용해 비교할 수 없습니다. 이번 절에서는 객체들의 동등 비교 및 크기 비교를 위해 무엇이 필요하고 어떻게 비교하는지 알아보겠습니다.

11.2.1. 객체 동등 비교

Set계열 클래스에는 같은 객체가 두 개 이상 들어갈 수 없다고 했습니다. Set 계열 클래스에 엘리먼트를 add 할 경우 엘리먼트의 hashCode() 메서드와 equals() 메서드를 통해 엘리먼트의 동등 비교가 이루어집니다. 개발자는 새로운 클래스를 정의할 때 객체들의 동등 비교가 될 수 있도록 해 주어야 합니다.

equals() 와 hashCode() 메서드는 Object 클래스에 있는 메서드입니다. 두 메서드의 원형은 다음과 같습니다.

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

equals() 메서드에서는 객체의 멤버를 비교하여 boolean형 값을 리턴 하도록 구현해야 합니다. hashCode() 메서드에서는 멤버변수의 내용이 같으면 동일한 리턴 값을 갖도록 해야 합니다. 물론 haskCode() 메서드는 항상 같은 값을 리턴하면 안됩니다. 즉 'return 1; 은 안 된다.'는 뜻입니다. hashCode() 메서드의 리턴 값이 다르면 두 객체는 다른 객체임을 보장합니다. 그러나 hashCode() 메서드의 리턴 값이 같더라도 두 객체가 같다는 것을 보장하지는 않습니다. hashCode()는 동등비교에 사용할 멤버변수들의 해시코드 값을 ^(XOR) 해서 리턴 하도록 구현할 수 있으며, hashCode()는 객체 동등비교에 equals()와 같이 사용됩니다.

Java SE 7버전 API에서 제공하는 클래스들 중에서 HashMap 클래스의 put() 메서드 구현부를 살펴보면 다음과 같습니다. 객체 동등 비교를 위해 hashCode() 메서드를 통해 객체 동등비교를 수행한 다음 equals() 메서드를 통해 다시 객체 동등 비교를 수행하는 것을 알 수 있습니다. HashSet의 add() 메서드는 HashMap 클래스의 put() 메서드를 호출합니다. 그래서 HashMap 클래스의 put() 메서드를 보여준 것입니다.

```
386:    public V put(K key, V value) {
387:        if (key == null)
388:            return putForNullKey(value);
389:        int hash = hash(key.hashCode());
390:        int i = indexFor(hash, table.length);
```

```
391:        for (Entry<K,V> e = table[i]; e != null; e = e.next) {
392:            Object k;
393:            if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
394:                V oldValue = e.value;
395:                e.value = value;
396:                e.recordAccess(this);
397:                return oldValue;
398:            }
399:        }
400:
401:        modCount++;
402:        addEntry(hash, key, value, i);
403:        return null;
404:    }
```

다음은 Dog 클래스에 equals()메서드와 hashCode() 메서드를 구현한 예입니다.

Dog.java

```
1:  public class Dog extends Object {
2:      String dogId;
3:      String dogName;
4:
5:      public Dog(String dogId, String dogName) {
6:          super();
7:          this.dogId = dogId;
8:          this.dogName = dogName;
9:      }
10:
11:     public String toString() {
12:         return dogId + " : " + dogName;
13:     }
14:
15:     public boolean equals(Object obj) {
16:         if (this == obj) return true;
17:         if (obj == null) return false;
18:
19:         //상속관계 있을 경우 데이터가 같더라도 다른 객체
20:         if (getClass() != obj.getClass()) return false;
21:
22:         Dog d = (Dog)obj;
23:         if(this.dogId.equals(d.dogId) && this.dogName.equals(d.dogName) ) {
24:             return true;
25:         }else {
26:             return false;
27:         }
28:     }
29:
30:     public int hashCode() {
31:         return dogId.hashCode() ^ dogName.hashCode();
32:     }
33:
34: }
```

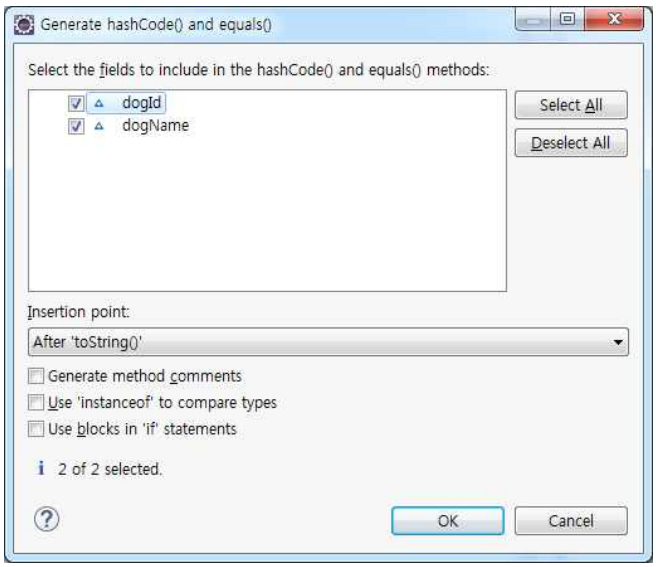
다음 코드는 앞의 Dog 클래스의 인스턴스를 생성한 다음 HashSet에 넣어서 같은 객체가 들어가는지 확인하는 클래스입니다.

ObjectEqualsExample.java

```
1:  import java.util.HashSet;
2:  import java.util.Set;
3:
4:  public class ObjectEqualsExample extends Object {
5:     public static void main(String[] args) {
6:        Dog d1 = new Dog("1234567890", "바둑이");
7:        Dog d2 = new Dog("1234567890", "바둑이");
8:        System.out.println(d1.equals(d2));
9:        System.out.println(d1 == d2);
10:       System.out.println(d1);
11:
12:       Set<Dog> set = new HashSet<Dog>();
13:       System.out.println("d1 객체를 set에 저장 : " + set.add(d1));
14:       System.out.println("d2 객체를 set에 저장 : " + set.add(d2));
15:    }
16: }
```

```
Console
<terminated> ObjectEqualsExample [Java
true
false
1234567890 : 바둑이
d1 객체를 set에 저장 : true
d2 객체를 set에 저장 : false
```

hashCode() 메서드와 equals() 메서드는 이클립스에서 쉽게 작성할 수 있습니다. 소스코드 창에서 오른쪽 마우스 버튼을 클릭하고 Source -> Generate hashCode and equals...를 선택하여 나타난 창 에서객체 동등비교에 사용할 멤버변수들을 체크한 다음 [OK] 버튼을 클릭하면 자동으로 hashCode() 메서드와 equals() 메서드가 추가됩니다.



11.2.2. java.lang.Comparable

Set 계열 클래스에는 TreeSet, 그리고 Map계열 클래스에는 TreeMap 클래스를 이용하면 저장되는 엘리먼트들을 정렬시킬 수 있습니다. 그러나 List 계열 클래스에는 Tree 구조를 갖는 클래스가 존재하지 않습니다. List 계열 컬렉션을 정렬시키기 위해서 Collections 클

래스의 sort() 메서드를 이용할 수 있습니다. 그런데 TreeSet, TreeMap 에 저장되는 엘리먼트 클래스들이나, sort() 메서드의 인자로 전달되는 List에 저장되어 있는 sort의 대상이 되는 클래스는 java.lang.Comparable 인터페이스를 구현한 클래스여야 합니다. Comparable 인터페이스를 구현할 때에는 compareTo() 메서드를 재정의 해야 합니다.

Collections 클래스의 sort() 메서드 원형은 다음과 같습니다.

- static <T extends Comparable<? super T>> void sort(List<T> list)
- static <T> void sort(List<T> list, Comparator<? super T> c)

첫 번째 sort() 메서드는 메서드 인자로 List객체를 받습니다. 그런데 List에 저장된 객체들이 Comparable 인터페이스를 구현한 클래스여야 합니다.
두 번째 sort() 메서드는 인자를 두 개 받습니다. List에 저장된 객체가 Comparable 인터페이스를 구현하지 않았을 경우 비교기(Comparator 인터페이스를 구현한 클래스)를 지정하여 두 객체의 크기비교를 할 수 있습니다.

다음 코드는 ArrayList에 저장되어 있는 엘리먼트를 Collections.sort(List<T> list) 메서드를 이용하여 정렬하는 예입니다. ArrayList에 저장되는 Tiger 클래스는 Comparable 인터페이스를 구현한 클래스입니다.

Tiger.java

```
1:  public class Tiger implements Comparable<Tiger>{
2:      String name;
3:
4:      Tiger(String name) {
5:          this.name = name;
6:      }
7:
8:      public int compareTo(Tiger obj) {
9:          return this.name.compareTo(obj.name);
10:     }
11: }
```

SortExample.java

```
1:  import java.util.ArrayList;
2:  import java.util.Collections;
3:  import java.util.List;
4:
5:  public class SortExample {
6:      public static void main(String[] args){
7:
8:          List<Tiger> list = new ArrayList<>();
9:
10:         list.add(new Tiger("라이거"));
11:         list.add(new Tiger("백호"));
12:         list.add(new Tiger("한라산"));
```

```
13:         list.add(new Tiger("백두"));
14:
15:         Collections.sort(list);
16:
17:         for(Tiger t : list) {
18:             System.out.println(t.name);
19:         }
20:     }
21: }
```

11.2.3. java.util.Comparator

Collections.sort() 메서드를 이용하여 리스트 내의 엘리먼트들을 정렬 시킬 수 있고, TreeSet과 TreeMap은 엘리먼트를 저장할 때 자동으로 정렬시켜 저장한다고 했습니다. 리스트에 저장되는 객체 또는 TreeSet, TreeMap에 저장하는 객체들이 프로그래머가 정의하는 클래스일 경우에 무엇을 기준으로 정렬이 되어야 할지 생각해야 합니다. 이처럼 엘리먼트들을 정렬해야 할 상황이 발생하면 프로그래머는 클래스를 정의할 때 Comparable 인터페이스를 구현하여 compareTo() 메서드에서 객체의 크기 비교를 수행할 수 있도록 하면 된다고 했습니다.

만일, TreeSet 또는 TreeMap에 저장되는 엘리먼트 클래스를 수정할 수 없는 상황이거나 List에 저장된 객체들의 클래스를 수정할 수 없는 상황이 있습니다. 그럴 경우에는 java.util.Comparable 인터페스를 구현하도록 클래스를 수정할 수 없으므로, 정렬을 위한 크기비교를 대신 해주는 비교기(Comparator)라는 것이 필요합니다. 앞에서 설명한 Comparable 인터페이스는 java.lang 패키지에 있습니다. 그러나 Comparable 인터페이스는 java.util 패키지에 정의되어 있기 때문에 import 문장이 필요합니다. 비교기는 Comparator 인터페이스를 구현한 클래스를 이용해 객체들 사이의 크기 비교를 해 줄 수 있습니다.

Comparator 인터페이스의 compare() 메서드는 매개변수로 전달된 두 값을 비교하여 같은 경우에는 0을 리턴하며, 첫 번째 값이 두 번째 값보다 큰 경우에는 양수를, 그렇지 않은 경우 음수를 리턴하게 됩니다. 만일 두 매개변수가 서로 다른 유형일 경우에는 ClassCastException이라는 예외를 던집니다. 그러므로 프로그래머는 컬렉션 계열 클래스 중에서 TreeSet이나 TreeMap에 저장할 엘리먼트 클래스를 선언할 경우 또는 Collections 클래스의 sort(List<T> list, Comparator<? super T> c)메서드를 사용할 경우에는 Comparator 인터페이스를 구현하여 compare() 메서드를 재정의 한 비교기 클래스를 만들어야 합니다.

다음 프로그램은 Employee의 이름을 기준으로 정렬시키기 위하여 Comparator 인터페이스를 구현한 예입니다.

Employee.java

```
1:  public class Employee {
2:     String name;
3:     int salary;
4:
5:     public Employee() {}
6:
7:     public Employee(String name, int salary) {
8:        super();
9:        this.name = name;
10:       this.salary = salary;
11:    }
12:
13:    public String toString() {
14:       return name + ":" + salary;
15:    }
16: }
```

EmployeeComparator.java

```
1:  import java.util.Comparator;
2:
3:  public class EmployeeComparator implements Comparator<Employee> {
4:
5:     public int compare(Employee obj1, Employee obj2) {
6:        return obj1.name.compareTo(obj2.name);
7:     }
8:  }
```

ComparatorExample.java

```
1:  import java.util.TreeSet;
2:
3:  public class ComparatorExample {
4:     public static void main(String[] args) {
5:        Employee e1 = new Employee("홍길동", 20000);
6:        Employee e2 = new Employee("김진숙", 30000);
7:        Employee e3 = new Employee("허현정", 70000);
8:        Employee e4 = new Employee("이순신", 40000);
9:
10:       TreeSet<Employee> list = new TreeSet<Employee>(new EmployeeComparator());
11:       list.add(e1);
12:       list.add(e2);
13:       list.add(e3);
14:       list.add(e4);
15:
16:       for(Employee s : list) {
17:          System.out.println(s);
18:       }
19:    }
20: }
```

```
Console
<terminated> Com
김진숙:30000
이순신:40000
허현정:70000
홍길동:20000
```

실행 결과를 보면 입력한 Employee 엘리먼트들이 이름을 기준으로 정렬된 것을 확인할 수 있습니다. Employee를 salary 순으로 정렬시키고 싶으면 EmployeeComparator 클래스의 6라인을 아래와 같이 바꿔서 실행해 보세요.

EmployeeComparator.java

```
6:   return obj1.salary - obj2.salary;
```

Comparator 클래스를 이용하면 기존 객체를 수정하지 않고도 여러 가지 정렬 방법을 정할 수 있습니다. Comparable 인터페이스를 구현하는 것이 객체의 기본 정렬 방법을 결정하는 것이라면 Comparator 인터페이스를 구현하는 것은 객체의 추가 정렬 방법을 결정할 수 있는 것입니다.

11.2.4. JDK 1.1 이전의 Collections

Set계열과 List 계열 컬렉션 클래스들 외에도 JDK 1.1 이전의 컬렉션 클래스들이 있습니다. 과거에 작성된 소스코드들은 객체를 저장할 때 Vector 클래스를 주로 사용했었습니다. 요즘은 성능상의 이유로 Set과 특별한 경우가 아니면 List 계열 컬렉션을 사용할 것을 권장합니다.

Vector : List 인터페이스를 구현한 클래스입니다. Vector 클래스는 확장 가능한 객체 배열을 구현합니다. 배열과 마찬가지로 정수 인덱스를 사용하여 액세스 할 수 있는 구성 요소가 들어 있습니다. 그러나 Vector를 만든 후 항목을 추가하거나 제거 할 수 있도록 필요에 따라 Vector의 크기를 늘리거나 줄일 수 있습니다

Stack : Vector클래스의 하위클래스입니다. Stack 클래스는 LIFO (Last-In-First-Out) 스택을 나타냅니다. empty(), push(), pop(), peek() 메서드를 지원합니다.

Hashtable : Map인터페이스를 구현한 클래스입니다. 이 클래스는 키를 값에 매핑하는 해시 테이블을 구현합니다. null 이외의 객체는 키 또는 값으로서 사용할 수 있습니다.
해시 테이블에서 객체를 성공적으로 저장 및 검색하려면 키로 사용되는 객체가 hashCode 메서드와 equals 메서드를 구현해야합니다.

Enumeration : Vector, Stack, Hashtable 등에서 엘리먼트를 조회하기 위한 클래스입니다. Enumeration 인터페이스를 구현하는 객체는 한 번에 하나씩 일련의 요소를 생성합니다. nextElement() 메서드를 연속적으로 호출하면 연속되는 일련의 요소가 반환됩니다.

다음 프로그램은 Person객체를 Vector클래스에 저장하는 예입니다.

Person.java

```
1:  public class Person {
2:     private String name;
3:     private int age;
4:
5:     public Person(String name, int age) {
6:        this.name = name;
7:        this.age = age;
8:     }
9:
10:    public String getName() {
11:       return name;
12:    }
13:    public int getAge() {
14:       return age;
15:    }
16: }
```

VectorExample.java

```
1:  import java.util.*;
2:
3:  public class VectorExample {
4:
5:     public static void main(String[] args) {
6:        Vector v = new Vector();
7:
8:        v.addElement(new Person("홍길동", 29));
9:        v.addElement(new Person("이순신", 30));
10:       v.addElement(new Person("강감찬", 65));
11:
12:       System.out.println(v);
13:       Person p = (Person)v.elementAt(2);
14:       System.out.println(p.getName());
15:       v.remove(1);
16:       System.out.println(v);
17:       System.out.println(v.size());
18:       System.out.println(v);
19:    }
20: }
```

```
Console
<terminated> VectorExample [Java Application] C:₩Program Files₩Java₩jre1.8.0_91₩
[Person@15db9742, Person@6d06d69c, Person@7852e922]
강감찬
[Person@15db9742, Person@7852e922]
2
[Person@15db9742, Person@7852e922]
```

11.3. 제네릭과 형 안정성(Generic & Type Safety)

11.3.1. 제네릭 클래스(generic class)

제네릭 클래스란 클래스 선언에 유형 매개변수가 들어있는 클래스를 뜻합니다. 제네릭은 자바 5.0에서 도입되었습니다. 제네릭이 도입되기 전까지는 모든 컬렉션이 파라미터로 Object 유형의 객체를 받기 때문에 어떤 객체를 넣든 컴파일러는 이를 상관하지 않았습니다. 예를 들면 ArrayList에 야구공(Ball)을 넣을 수도 있고, 자동차(Car)를 넣을 수도 있었습니다. 그러나 반대로 컬렉션에서 객체를 빼 내올 경우에는 리턴 유형이 Object이기 때문에 객체를 넣기 전의 상태로 되돌리기 위해서는 반드시 형 변환이 필요했었습니다. 그러나 제네릭 기능이 도입된 이후에는 제네릭 기능으로 인해 컬렉션에 원하지 않는 객체가 저장되는 것을 들어가거나 반대로 꺼낼 때 저장된 객체 유형으로 형 변환해야 하는 것에 대해 신경 쓰지 않아도 됩니다. 예를 들어 ArrayList<Student> 이라고 하면 ArrayList에는 Student객체만 넣을 수 있고, 객체를 꺼낼 때도 Student 레퍼런스로 나오게 할 수 있습니다.
제네릭을 사용하기 위해서는 컴파일러 버전이 5.0이상이어야 하며, 특정 유형의 변수를 넣는 컬렉션을 선언하기 위해서는 다음과 같이 클래스 이름 뒤에 < > 기호를 사용하여 컬렉션에 넣을 유형을 선언해야 합니다. 제네릭 클래스를 사용하기 위해서 API 문서를 참조 할 수 있습니다. 다음은 컬렉션 중에서 가장 흔하게 쓰이는 ArrayList 클래스의 클래스 선언부를 나타내고 있습니다.

```
public class ArrayList<E>
     extends AbstractList<E>
     implements    List<E>,    RandomAccess,    Cloneable,
Serializable
```

위의 클래스 선언부의 <E>는 엘리먼트(Element)의 약어입니다. 컬렉션에 저장하거나 컬렉션에서 리턴 할 엘리먼트의 유형(type)입니다.

ArrayList에서 유형 매개변수를 사용하기 위해서는 클래스 이름 뒤에 E 대신에 저장된 엘리먼트의 유형을 적으면 되는데 다음과 같이 선언하면 ArrayList에는 String 객체만 넣고 뺄 수 있게 됩니다.

```
ArrayList<String> myList = new ArrayList<String>();
```

Java SE 7(JDK 1.7)부터는 다음 코드와 같이 r-value에 해당하는 부분에 제네릭 유형을

써주지 않아도 됩니다. 이를 Diamond operation 이라고 부릅니다.

```
ArrayList<String> myList = new ArrayList<>();
```

앞의 코드는 컴파일러에 의해서 다음과 같은 식으로 해석됩니다.

```
public class ArrayList<String>
            extends AbstractList<String> ... {
    public boolean add(String o) {
    //생략...
}
```

다음은 ArrayList에 Student 유형의 변수를 넣는 컬렉션을 선언한 예입니다.

```
ArrayList<Student> myList = new ArrayList<Student>();
```

다음은 제네릭 기능을 사용한 예입니다.

GenericExample.java

```
1:  import java.util.ArrayList;
2:
3:  public class GenericExample {
4:
5:     public static void main(String[] args)   {
6:        ArrayList<String> lists=new ArrayList<>();
7:
8:        lists.add("first");
9:        lists.add("second");
10:       lists.add("third");
11: //    list.add(new Integer(4));    //String이 아닌 다른 객체는 저장 못함
12: //    list.add(new Float(5.0f));   //String이 아닌 다른 객체는 저장 못함
13:
14:       String s1 = lists.get(0);    //형변환 하지 않아도 됨
15:       System.out.println(s1);
16:
17:       for(String s : lists) {
18:          System.out.println(s);
19:       }
20:    }
21: }
```

```
Console
<terminated> Gen
first
first
second
third
```

코드에서 ArrayList 선언 시 String만 올 수 있도록 설정하였습니다. 그러므로 String아닌 다른 유형을 넣으려고 하면 오류가 발생하는 것을 알 수 있습니다. 제네릭 기능을 사용하면 컬렉션에 들어있는 엘리먼트를 꺼내올 때에 형 변환을 하지 않아도 됩니다.

11.3.2. 제네릭 메서드(generic method)

제네릭 메서드는 메서드 선언에 유형 매개변수가 포함되어 있는 메서드를 뜻합니다. 메서드에서는 클래스 선언부에서 정의된 유형 매개변수를 사용하는 방법과 클래스 선언부에서 쓰이지 않은 유형 매개변수를 사용하는 방법이 있습니다.

다음은 클래스 선언부에서 정의된 유형 매개변수를 사용하는 방법입니다. 클래스를 정의할 때 'E'를 사용하였기 때문에 add()메서드에서도 'E'를 사용할 수 있습니다.

```
public class ArrayList<E>
      extends AbstractList<E>
      implements List<E>, RandomAccess, Cloneable, Serializable  {
      public boolean add(E o) {
            //...
      }
      //생략
}
```

다음은 클래스 선언부에서 쓰이지 않는 유형 매개변수를 사용하는 방법입니다.

```
public <T extends Animal> void doSomething(ArrayList<T> list)
```

11.4. Typesafe enum

11.4.1. enum

enum은 Java SE 5(JDK 1.5)버전에 추가된 데이터 유형입니다. enum은 수년간 메서드가 없는 변수만 선언된 클래스를 이용하여 enum 값으로 사용되어온 public static final int 선언과 상당히 유사합니다. 클래스를 이용하여 열거형을 표현했을 때와 비교하여 가장 크고 확실하게 개선된 점은 typesafe입니다. 클래스의 int형 상수 변수와는 달리 enum 유형으로 선언된 변수의 값은 다른 유형의 위치에 사용할 수 없습니다. 그 이유는 기존의 방법으로 열거형을 표현했을 때에 선언된 상수 변수들이 모두 int 형이라면 변수의 이름이 다르더라도 컴파일러에게는 모든 것이 똑같아 보이기 때문입니다. 아주 드물게 예외가 있긴 하지만 이 경우에도 enum과 유사한 모든 int 구성을 enum 인스턴스로 교체해야 합니다.
enum은 여러 가지 추가 기능을 제공합니다. 유틸리티 클래스인 EnumMap 및 EnumSet은 특히 enum에 최적화된 표준 컬렉션 클래스들 입니다. 컬렉션에 enum만 포함되는 것을 알고 있다면 HashMap 또는 HashSet 대신 이러한 특정 컬렉션을 사용해야 합니다.

대부분의 경우 코드에서 모든 public static final int를 enum으로 교체할 수 있습니다. enum은 비슷(Comparable)하고, 내부 클래스(또는 내부 enum)일지라도 이에 대한 참조가 똑같아 보이므로 정적으로 가져올 수 있습니다. 유의해야 할 점은 enum을 비교할 때 선언되는 순서가 C언어에서처럼 서수 값을 나타내지 않는 다는 것입니다.

enum의 선언은 클래스 선언과 유사합니다. 패키지를 선언할 수 있으며, 접근 제한 모드를 가질 수 있습니다. 열거 항목들은 콤마(,)로 구분하여 나열합니다. 그리고 열거 항목의 마지막에 세미콜론(;)을 입력하는 것을 잊지 마세요.

선언 예:

```
public enum Coin {
    PENNY,
    NICKEL,
    DIME,
    QUARTER;
}
```

사용 예:

```
Coin coin = Coin.DIME;
```

다음 코드는 열거형을 선언한 예입니다. 패키지가 선언되어 있음에 유의하세요. 그리고 접근 모드가 public입니다. 상황에 따라 접근 모드는 달라질 수 있을 것입니다. 그리고 패키

지 이름에서 enum뒤에 _ 문자가 포함되어 있는 이유는 패키지 이름에 키워드는 포함될 수 없기 때문이고, enum과 관련된 예제를 나타내기 위해서 enum_라고 표현한 것입니다.

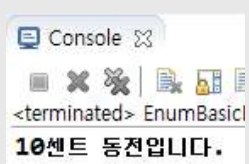
Coin.java

```
1:  public enum Coin {
2:     PENNY,
3:     NICKEL,
4:     DIME,
5:     QUARTER;
6:  }
```

열거형을 반드시 별도의 파일로 작성할 필요는 없습니다. 한 클래스서만 사용된다면 클래스 블록 안에 내부 열거형으로 선언할 수도 있습니다.
다음 코드는 위의 열거형을 사용한 클래스입니다.

EnumBasicExample.java

```
1:  public class EnumBasicExample {
2:     public static void main(String[] args) {
3:        Coin coin = Coin.DIME;
4:
5:        switch(coin) {
6:        case PENNY :
7:           System.out.println("1센트 동전입니다.");
8:           break;
9:        case NICKEL :
10:          System.out.println("5센트 동전입니다.");
11:          break;
12:       case DIME :
13:          System.out.println("10센트 동전입니다.");
14:          break;
15:       case QUARTER :
16:          System.out.println("25센트 동전입니다.");
17:          break;
18:       default :
19:          break;
20:       }
21:    }
22: }
```



앞에서도 언급했지만 자바의 열거형은 C언어처럼 서수를 대신하여 사용할 수 없습니다. 위 코드에서 coin에 숫자 값을 할당 할 수 없으며 마찬가지로 switch 문에 열거형일 경우 case 문에서 정수 값으로 비교할 수 없습니다.

11.4.2. "Hidden" 정적 메서드

작성한 모든 enum 선언에는 values() 메서드와 valueOf() 메서드가 표시됩니다. 이 두 가지 메서드는 Enum 클래스 자체가 아니라 enum 하위 클래스에 대한 정적 메서드이기 때문에 java.lang.Enum에 대한 javadoc에는 표시되지 않습니다.

첫 번째 values()는 enum에 가능한 모든 값의 배열을 반환합니다.
두 번째 valueOf()는 제공된 문자열에 대한 enum을 반환하는데, 원본 코드 선언과 똑같이 일치해야 합니다.

다음 코드는 enum의 정적 메서드 예입니다.

EnumHiddenMethodExample.java

```
1:  public class EnumHiddenMethodExample {
2:     public static void main(String[] args) {
3:        Coin[] coins = Coin.values();
4:
5:        System.out.println(coins);
6:        for(Coin c : coins) {
7:           System.out.println(c.toString());
8:        }
9:
10:       Coin c1 = Coin.valueOf("QUARTER");
11:       System.out.println("c1의 값 : " + c1);
12:       System.out.printf("c1의 값 : %s\n",c1);
13:    }
14: }
```

```
Console
<terminated> EnumHiddenM
[LCoin;@15db9742
PENNY
NICKEL
DIME
QUARTER
c1의 값 : QUARTER
c1의 값 : QUARTER
```

11.4.3. 메서드를 갖는 enum

enum의 좋은 점 중의 하나는 메서드를 가질 수 있다는 것입니다. 메서드를 이용하면 열거 데이터를 원하는 형태의 값으로 사용할 수 있습니다.

다음은 앞에서 선언한 Coin 열거형에 메서드를 선언한 예입니다.

Coin.java

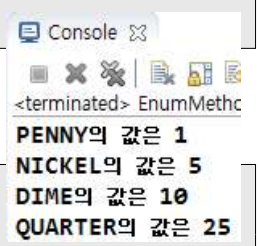
```
1:  public enum Coin {
2:     PENNY,
3:     NICKEL,
4:     DIME,
5:     QUARTER;
6:
7:     int getValue() {
```

```
8:         switch(this) {
9:            case PENNY: return 1;
10:           case NICKEL: return 5;
11:           case DIME: return 10;
12:           case QUARTER: return 25;
13:           default: return 0;
14:        }
15:     }
16: }
```

다음 코드는 위의 열거형을 사용하는 예입니다.

EnumMethodExample.java

```
1:  public class EnumMethodExample {
2:     public static void main(String[] args) {
3:        for(Coin coin : Coin.values()) {
4:           System.out.println(coin + "의 값은 " + coin.getValue());
5:        }
6:     }
7:  }
```



실행 결과는 빈약하지만 이것은 많은 의미를 내포하고 있습니다. C언어에서는 열거형 데이터를 기수형태의 정수 값으로 표현이 가능하지만 자바의 열거형은 열거 데이터를 기수형태의 정수 값으로 사용하지 못하는 단점(?)이 있었습니다. 열거형 메서드는 C언어에서처럼 기수형태의 정수 값뿐만 아니라 사용자가 원하는 형태의 값을 사용할 수 있도록 해줍니다.

11.4.4. enum 추상 메서드

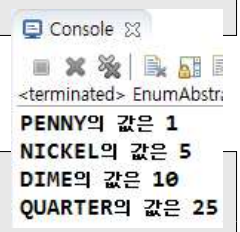
enum에서 추상 메서드를 선언한 다음 각각의 열거 항목에서 추상 메서드를 다르게 구현할 수도 있습니다.

다음은 추상메서드 선언한 후 열거 항목에서 메서드를 각각 다르게 구현한 예입니다.

Coin2.java

```
1:  public enum Coin2 {
2:     PENNY {
3:        int getValue() {
4:           return 1;
5:        }
6:     },
7:     NICKEL {
8:        int getValue() {
9:           return 5;
10:       }
11:    },
```

```
12:     DIME {
13:        int getValue() {
14:           return 10;
15:        }
16:     },
17:     QUARTER {
18:        int getValue() {
19:           return 25;
20:        }
21:     };
22:
23:     abstract int getValue();
24: }
```



다음은 추상메서드를 갖는 열거형을 테스트 하는 코드입니다.

EnumAbstractMethodExample.java

```
1:  public class EnumAbstractMethodExample {
2:     public static void main(String[] args) {
3:        for(Coin2 coin : Coin2.values()) {
4:           System.out.println(coin + "의 값은 " + coin.getValue());
5:        }
6:     }
7:  }
```

11.4.5. 생성자를 갖는 enum

다음의 예에서 보는바와 같이 생성자를 가질 수 있습니다. 생성자를 이용하면 많은 열거 데이터를 원하는 형태의 값으로 사용하기가 더 쉬워집니다.

다음 코드는 열거형에 생성자를 정의한 예입니다.

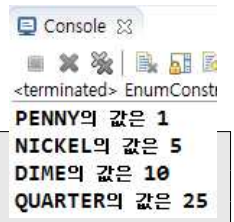
Coin3.java

```
1:  public enum Coin3 {
2:     PENNY(1),
3:     NICKEL(5),
4:     DIME(10),
5:     QUARTER(25);
6:
7:     private int coinValue;
8:
9:     int getValue() {
10:       return coinValue;
11:    }
12:
13:    Coin3(int value) {
14:       coinValue = value;
```

```
15:     }
16: }
```

생성자를 갖는 열거형을 사용하는 방법은 의외로 간단합니다. 생성자라고 해서 보통의 클래스처름 new를 통해 생성하지는 않습니다. 열거형의 생성자는 직접 호출할 필요 없이 열거 데이터만 참조하면 됩니다.

다음은 생성자를 갖는 열거형을 테스트하는 예입니다.



EnumConstructorExample.java

```
1:  public class EnumConstructorExample {
2:     public static void main(String[] args) {
3:        for(Coin3 coin : Coin3.values()) {
4:           System.out.println(coin + "의 값은 " + coin.getValue());
5:        }
6:     }
7:  }
```

11.4.6. Java SE 5(JDK 1.5) 이전 버전에서 열거형 사용

typesafe 하지 않은 열거형 클래스

만일 여러분이 C 또는 C++ 또는 C# 프로그래머라면 아래의 코드들이 더 익숙할 것입니다.

```
typedef enum {
   WHITE,
   BLACK,
   YELLOW
}color;
```

```
typedef enum {
   CAMRY,
   HONDA,
   FORD
}car;
```

C 또는 C++에서는 열거형으로 정의된 변수의 값은 정수형으로 사용할 수 있었을 것입니다. 위의 예에서는 할당되는 값들은 0부터 증가하여 각각, WHITE == 0, BLACK == 1 그리고 YELLOW == 2 값이 할당되었을 것입니다. 이로 인해 switch문에서도 사용할 수 있

었고 다른 제어문들에서 열거형의 값을 정수형으로 대신 사용할 수 있었습니다.

그러나 아래의 코드를 보겠습니다.

```
color myColor = FORD;
```

이 코드는 자바에서는 안 되는 일이지만 C언어에서는 가능한 일입니다. 그러나 위의 코드에는 문제가 있습니다. color와 car가 섞여 사용되고 있습니다. 자동차 열거형 값을 색상 열거형 값에 할당하는 경우입니다. 이러한 코드가 C언어에서는 아무런 제약을 받지 않고 컴파일 오류 없이 실행이 된다는 것입니다. 이러한 경우를 보고 "Such type is not safe." 라고 할 수 있습니다.

enum 키워드가 추가되기 이전 버전, 즉, Java SE 5(JDK 1.5) 이전 버전의 자바에서는 열거형을 사용하기 위해서는 여러분은 아래의 코드처럼 클래스를 선언하고 그 안에 변수들을 선언하는 방식으로 열거형을 사용할 수 있을 것입니다.

```
public class PlayingCard {
    public static final int SUIT_CLUBS =0;
    public static final int SUIT_DIAMONDS =1;
    public static final int SUIT_HEARTS =2;
    public static final int SUIT_SPADES =3;
    ...
}
```

위의 코드는 switch문에서 사용하는데 아무런 문제가 없을 것입니다. 그러나 type safe하지 않습니다. 뭔가 미심쩍은 부분이 존재한다는 것입니다. C언어에 익숙한 프로그래머들에게는 그저 좋게만 보이는 코드일 수 있습니다.
다음에서는 클래스가 상수를 이용하여 열거형 자료를 나타낼 때 type safe하도록 표현해 보도록 하겠습니다.

typesafe 한 열거형 사용을 위한 클래스

다음은 Java SE 5(JDK 1.5)이전 버전에서의 열거형을 사용하기 위해서 클래스를 선언할 때 type safe하도록 상수를 선언한 예입니다.

Suit.java

```
1:  public class Suit {
2:      private final String name;
3:
4:      public static final Suit CLUBS = new Suit("clubs");
5:      public static final Suit DIAMONDS = new Suit("diamonds");
6:      public static final Suit HEARTS = new Suit("hearts");
7:      public static final Suit SPADES = new Suit("spades");
8:
9:      private Suit(String name) {
```

```
10:         this.name =name;
11:     }
12:     public String toString() {
13:         return name;
14:     }
15: }
```

우선 생성자가 private로 선언되어 있는 것을 확인하세요. 이 클래스는 final로 선언되어 있지 않아도 자식 클래스를 가질 수 없습니다. 그리고 상수들은 static으로 선언되어 있어 객체 생성 없이 클래스 이름만으로 참조가 가능합니다. 이러한 코드 디자인은 컴파일 시 형 안정성(type safety)을 보장해 줍니다.

이제 여러분은 Suit 클래스를 아래와 같은 방법으로 C언어의 enum처럼 사용할 수 있습니다. 물론 형 안정성을 갖도록 사용하는 것입니다.

```
private static final Suit[] CARD_SUIT = {
    Suit.CLUBS,
    Suit.DIAMONDS,
    Suit.HEARTS,
    Suit.SPADES
};
```

```
Suit suit = CARD_SUIT[0];

if (suit == Suit.CLUBS) {
    ...
}else if (suit == Suit.DIAMONDS) {
    ...
}else if (suit == Suit.HEARTS) {
    ...
}else if (suit == Suit.SPADES) {
    ...
}else{
   throw new NullPointerException("Null Suit");
   //suit ==null
}
```

나중에 여러분이 Suit 클래스를 확장하게 될지라도, 위의 코드가 포함된 클래스는 다시 컴파일을 할 필요가 없을 것입니다.

다음 코드는 Suit 클래스를 형 안정성을 갖도록 사용하는 예입니다.

EnumSuitExample.java

```
1:  public class EnumSuitExample {
2:      private static final Suit[] CARD_SUIT = {
3:          Suit.CLUBS,
4:          Suit.DIAMONDS,
```

```
5:         Suit.HEARTS,
6:         Suit.SPADES
7:     };
8:
9:     public static void main(String[] args) {
10:        for(Suit suit : CARD_SUIT) {
11:           System.out.println(suit);
12:        }
13:
14:        System.out.println();
15:        Suit suit = CARD_SUIT[0];
16:        if(suit == Suit.CLUBS) {
17:           System.out.println("clubs");
18:        }else if(suit == Suit.DIAMONDS) {
19:           System.out.println("diamonds");
20:        }else if(suit == Suit.HEARTS) {
21:           System.out.println("hearts");
22:        }else if(suit == Suit.SPADES) {
23:           System.out.println("spades");
24:        }else {
25:           throw new NullPointerException("Null Suit");
26:        }
27:    }
28: }
```

```
Console
<terminated> Er
clubs
diamonds
hearts
spades

clubs
```

순서 비교가 가능하고 typesafe한 열거형 클래스

Comparable 인터페이스를 구현하면 순서 비교가 가능한 열거형을 만들 수 있습니다. 물론 그러기 위해서는 compareTo() 메서드를 재정의 해야 합니다.

Suit.java

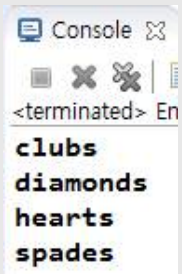
```
1:  public class Suit implements Comparable<Suit> {
2:     private final String name;
3:
4:     public static final Suit CLUBS =new Suit("clubs");
5:     public static final Suit DIAMONDS =new Suit("diamonds");
6:     public static final Suit HEARTS =new Suit("hearts");
7:     public static final Suit SPADES =new Suit("spades");
8:
9:     private static int nextOrdinal =0;
10:
11:    private final int ordinal = nextOrdinal++;
12:
13:    private Suit(String name) {
14:        this.name = name;
15:    }
16:    public String toString() {
```

```
17:         return name;
18:     }
19:
20:     public int compareTo(Suit suit) {
21:        return ordinal - suit.ordinal;
22:     }
23: }
```

다음은 비교 가능하고 type safe 한 열거형을 구현하기 위한 클래스를 테스트 하는 코드입니다. 코드에서 사용된 TreeSet 클래스는 객체를 저장할 수 있는 클래스입니다. 클래스에 대한 자세한 내용은 뒤에서 다시 설명됩니다. 이 예제의 중요한 사항은 TreeSet에 add한 Suit 들이 자동으로 정렬이 되어 있다는 것입니다.

EnumSuitSortExample.java

```
1:  import java.util.TreeSet;
2:
3:  public class EnumSuitSortExample {
4:     public static void main(String[] args) {
5:        Suit clubs = Suit.CLUBS;
6:        Suit diamonds = Suit.DIAMONDS;
7:        Suit hearts = Suit.HEARTS;
8:        Suit spades = Suit.SPADES;
9:
10:       TreeSet<Suit> suits = new TreeSet<>();
11:       suits.add(spades);
12:       suits.add(hearts);
13:       suits.add(diamonds);
14:       suits.add(clubs);
15:
16:       for(Suit suit : suits) {
17:          System.out.println(suit);
18:       }
19:    }
20: }
```



위와 같이 클래스를 이용하여 typesafe한 열거형 데이터를 사용할 수 있지만 enum 형식을 이용하여 열거형을 사용하면 더 많은 이점이 있습니다. JDK버전이 1.4이하에서 컴파일/실행되어야 하는 상황이라면 어쩔 수 없지만 그렇지 않은 상황이라면 enum 형 선언을 하는 것이 더 많은 장점을 제공합니다.

※ 자바의 열거형에 대해서 더 자세하게 알고 싶으면 다음 사이트를 참고하세요.
http://docs.oracle.com/javase/1.5.0/docs/guide/language/enums.html

11.5. Varargs(Variable-length arguments)

11.5.1. 가변인자(Varargs)

varargs는 Variable-length argument의 약어입니다. 우리말로 번역하면 가변인자 또는 가변인수라고 부릅니다. 가변인자의 기능은 Java SE 5(JDK 1.5)이후부터 사용할 수 있습니다. 가변인자를 잘 사용하면 정말로 거추장스러운 코드를 일부 정리할 수 있습니다. 다음과 같은 기본 예제는 String 인수를 다양하게 가진 로그 메서드입니다.

```
Log.log(String code)
Log.log(String code, String arg)
Log.log(String code, String arg1, String arg2)
Log.log(String code, String[] args)
```

가변인자에 관한 설명에서 흥미로운 사실은 앞의 4개의 메서드 예제를 새로운 메서드 예제로 교체하는 경우 얻어지는 호환성입니다. 아래의 코드를 보시면 파라미터 변수의 선언 유형 뒤에 말줄임표(...)가 붙어있습니다.

```
Log.log(String code, String... args)
```

모든 가변인자는 소스 호환적이어서 log() 메서드의 모든 호출자를 재 컴파일 하는 경우 4개의 모든 메서드를 바로 교체할 수 있습니다. 그러나 이전 버전과의 호환성이 필요한 경우 처음 세 개는 그대로 두어야 합니다. 마지막 메서드에서만 String 배열을 가져오면 같은 값이 되므로 가변인자 버전으로 교체할 수 있습니다.

가변인자는 배열을 통해 전달됩니다. 그러므로 다음 예제에서처럼 for 문을 이용해 모든 인자를 쉽게 처리할 수 있습니다.

VariableLengthExample.java

```
1:  public class VariableLengthExample {
2:
3:      public static void main(String[] args) {
4:          log("Hello");
5:          log("VariableLengthExample", "Hello");
6:          log("a", "b", "c");
7:          log();
8:      }
9:
10:     public static void log(String... msg) {
```

```
11:         System.out.print("로그 : ");
12:         for(String message : msg) {
13:             System.out.print(message + ", ");
14:         }
15:         System.out.println();
16:     }
17: }
```

11.5.2. 가변인자 형변환(Casting)

다음 코드는 log메서드의 인자로 들어가는 항목들 중에서 두 개 이상의 인자가 들어갈 경우에는, 첫 번째 항목은 String이고 두 번째는 Exception인 상황을 예상할 수 있습니다. 그러한 상황에 varargs를 이용해서 메서드를 선언하면 다음처럼 구현할 수 있을 것입니다. 그러면 목록의 첫 번째(0번 인덱스) 데이터를 String으로 형 변환해서 사용하고 두 번째(1번 인덱스) 데이터를 예외 클래스로 형 변환해서 사용해야 할 것입니다.

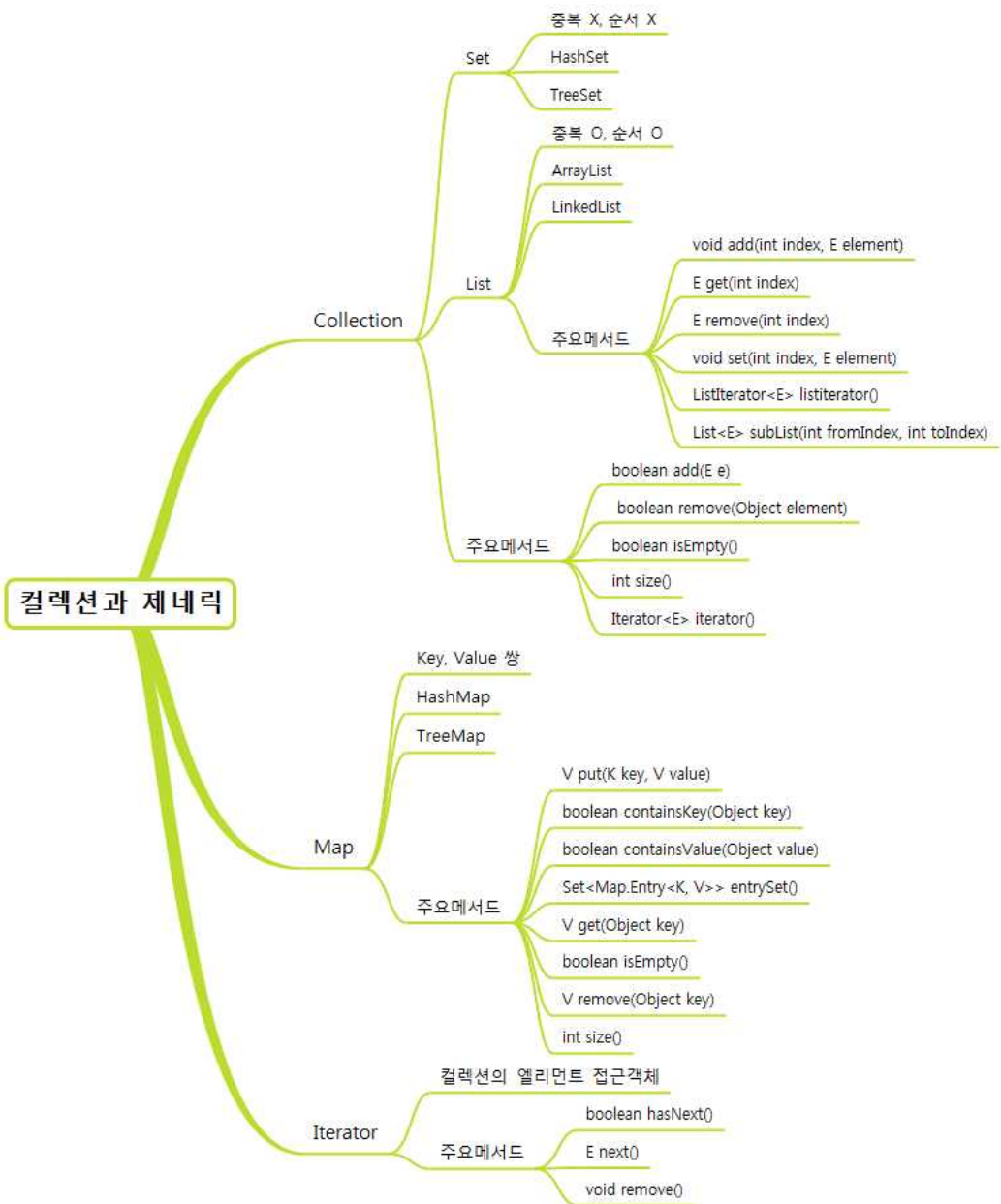
```
Log.log(Object... objects) {
    String message = (String)objects[0];
    if (objects.length > 1) {
        Exception e = (Exception)objects[1];
        // Do something with the exception
    }
}
```

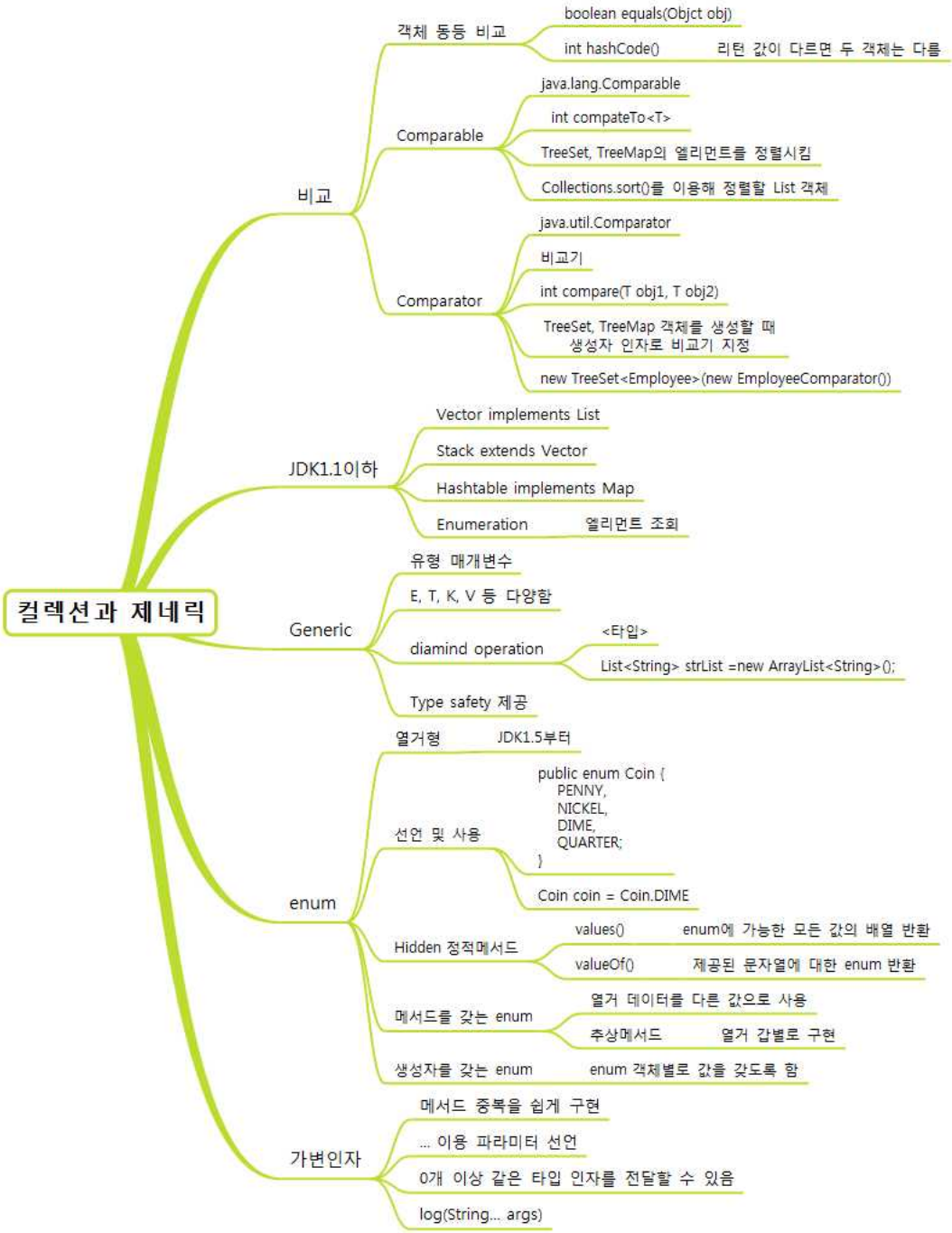
이럴 경우 위의 코드에 비하여 권장하는 메서드 원형은 varargs 파라미터에서 별도로 선언된 String 및 Exception 을 사용하여 다음과 같이 선언하는 것이 더 바람직 할 것입니다.

```
Log.log(String message, Exception e, Object... objects) {
    ...
}
```

지나치게 사용하려고 하지 마세요. 가변인자를 사용하면 '타입 시스템'을 망가뜨릴 수 있습니다. 강력한 형 지정(strong typing) 이 필요한 경우 사용하세요. PrintStream.printf() 메서드는 이러한 규칙에 대한 예외입니다. 이것은 유형 정보를 나중에도 수용할 수 있도록 첫 번째 인수로 제공합니다.

11.6. 마인드맵 정리


컬렉션과 제네릭
Collection
Set
중복 X, 순서 X
HashSet
TreeSet
List
중복 O, 순서 O
ArrayList
LinkedList
주요메서드
void add(int index, E element)
E get(int index)
E remove(int index)
void set(int index, E element)
ListIterator<E> listiterator()
List<E> subList(int fromIndex, int toIndex)
주요메서드
boolean add(E e)
boolean remove(Object element)
boolean isEmpty()
int size()
Iterator<E> iterator()
Map
Key, Value 쌍
HashMap
TreeMap
주요메서드
V put(K key, V value)
boolean containsKey(Object key)
boolean containsValue(Object value)
Set<Map.Entry<K, V>> entrySet()
V get(Object key)
boolean isEmpty()
V remove(Object key)
int size()
Iterator
컬렉션의 엘리먼트 접근객체
주요메서드
boolean hasNext()
E next()
void remove()


컬렉션과 제네릭
비교
객체 동등 비교
boolean equals(Objct obj)
int hashCode()
리턴 값이 다르면 두 객체는 다름
Comparable
java.lang.Comparable
int compateTo<T>
TreeSet, TreeMap의 엘리먼트를 정렬시킴
Collections.sort()를 이용해 정렬할 List 객체
Comparator
java.util.Comparator
비교기
int compare(T obj1, T obj2)
TreeSet, TreeMap 객체를 생성할 때
생성자 인자로 비교기 지정
new TreeSet<Employee>(new EmployeeComparator())
JDK1.1이하
Vector implements List
Stack extends Vector
Hashtable implements Map
Enumeration
엘리먼트 조회
Generic
유형 매개변수
E, T, K, V 등 다양함
diamind operation
<타입>
List<String> strList =new ArrayList<String>();
Type safety 제공
enum
열거형
JDK1.5부터
선언 및 사용
public enum Coin {
PENNY,
NICKEL,
DIME,
QUARTER;
}
Coin coin = Coin.DIME
Hidden 정적메서드
values()
enum에 가능한 모든 값의 배열 반환
valueOf()
제공된 문자열에 대한 enum 반환
메서드를 갖는 enum
열거 데이터를 다른 값으로 사용
추상메서드
열거 갑별로 구현
생성자를 갖는 enum
enum 객체별로 값을 갖도록 함
가변인자
메서드 중복을 쉽게 구현
... 이용 파라미터 선언
0개 이상 같은 타입 인자를 전달할 수 있음
log(String... args)