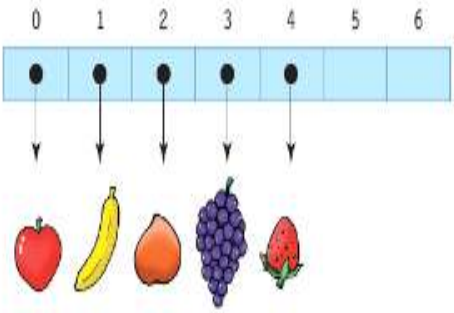
컬렉션(collection)의 개념

3

- 컬렉션
  - 요소(element)라고 불리는 가변 개수의 객체들의 저장소
    - 객체들의 컨테이너라고도 불림
    - 요소의 개수에 따라 크기 자동 조절
    - 요소의 삽입, 삭제에 따른 요소의 위치 자동 이동
  - 고정 크기의 배열을 다루는 어려움 해소
  - 다양한 객체들의 삽입, 삭제, 검색 등의 관리 용이

배열(array)



- 고정 크기 이상의 객체를 관리할 수 없다.
- 배열의 중간에 객체가 삭제되면 응용프로그램에서 자리를 옮겨야 한다.

컬렉션(collection)

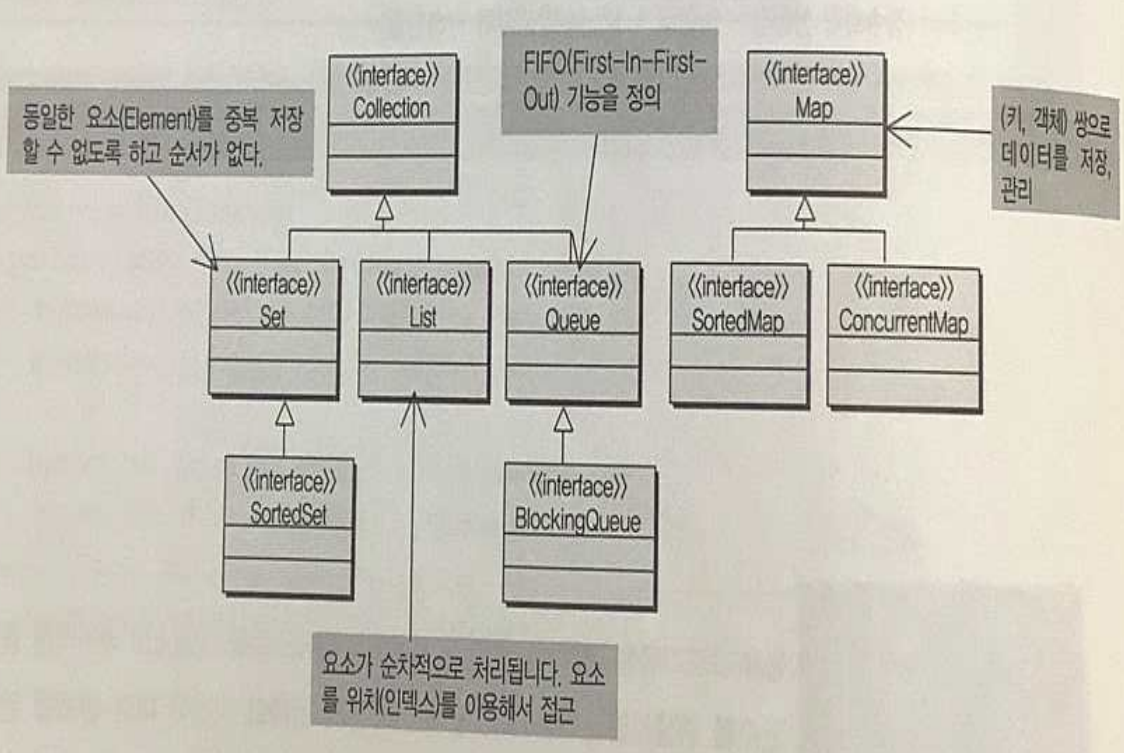
- 가변 크기로서 객체의 개수를 염려할 필요 없다.
- 컬렉션 내의 한 객체가 삭제되면 컬렉션이 자동으로 자리를 옮겨준다.

컬렉션 클래스

자바에서는 다양한 자료구조를 컬렉션(Collection)이라는 클래스로 제공합니다. 자바 컬렉션 클래스 라이브러리는 몇 개의 인터페이스와 실제 구현 클래스, 알고리즘으로 구성되어 있습니다.

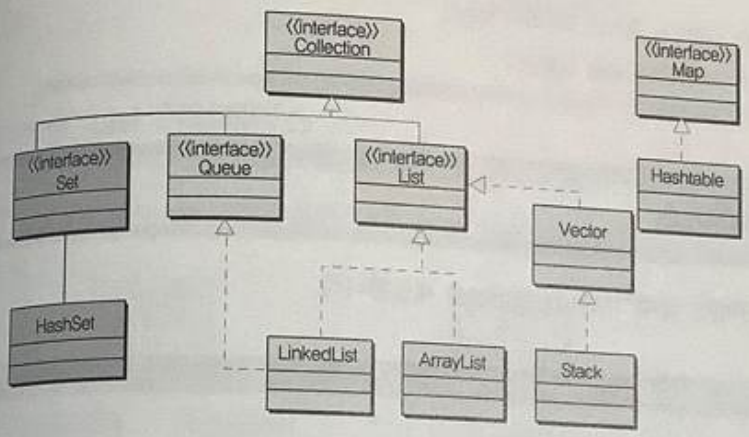
인터페이스

자바에서는 유사한 컬렉션 클래스들끼리 접근하는 방식을 일관성 있게 하기 위해서 다음과 같이 인터페이스로 구분해 두었습니다.



구현 클래스

위 인터페이스들을 구체적으로 구현한 클래스입니다.

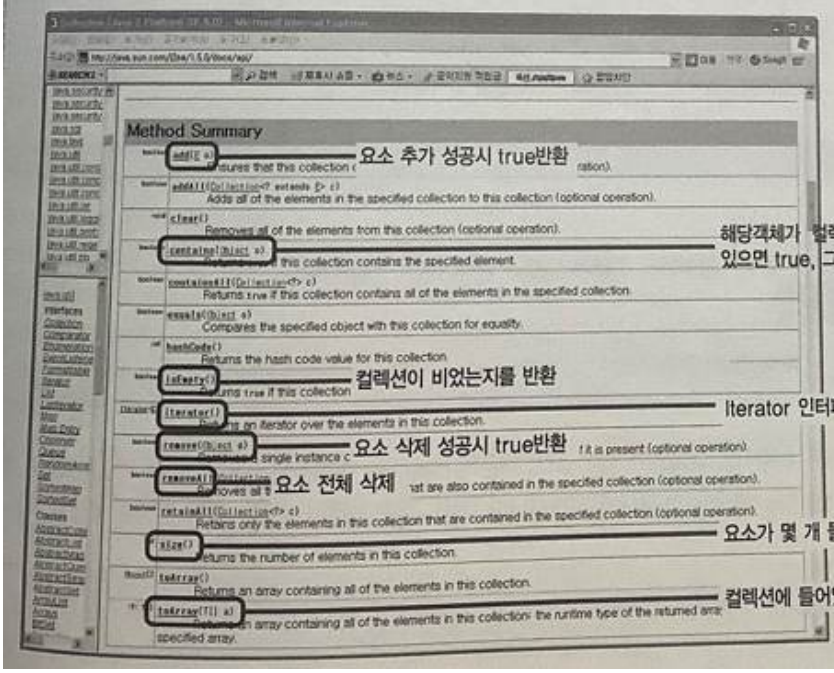


알고리즘

Collections 인터페이스로 구현된 객체들을 검색(binarySearch) 또는 정렬(sort) 섞을(shuffle) 수 있도록 java.util.Collections 인터페이스에 유용한 메서드가 제공됩니다.

Collection 인터페이스의 주요 메서드

다음은 Collection 인터페이스의 메서드입니다. 주요 메서드는 부가 설명을 했습니다.



Collection 인터페이스는 배열처럼 데이터를 여러 개 저장할 수 있는데, 배열과의 차이점은 서로 다른 형태의 자료를 요소로 가질 수 있다는 점입니다. 또 하나의 특징은 기억공간이 실행 시에 자동 증가 될 수 있다는 점입니다. Collections 인터페이스는 Set과 List로 나뉘는데 각 각 다음과 같은 특징이 있습니다.

:: Set : 순서가 없는 집합으로 중복을 허용하지 않는다.

:: List : 순서가 있는 집합으로 중복을 허용한다.

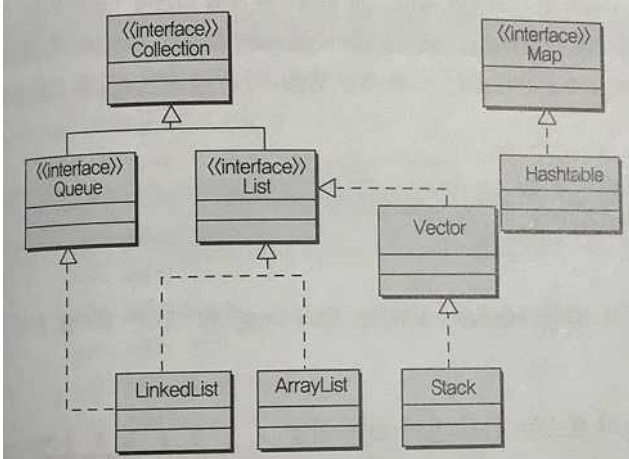
Set 인터페이스

Set 인터페이스를 구현한 클래스로는 HashSet을 제공합니다.

예제. 자료를 순서 없이 처리하는 Set 객체 - [파일 이름 : Collections01.java]

List 인터페이스

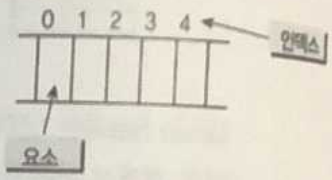
List 인터페이스로 구현된 클래스로는 ArrayList, Vector, Stack, LinkedList가 있습니다.



List는 Set과는 달리 요소가 순차적으로 처리된다고 하였습니다. List 인터페이스가 Set 인터페이스와 어떤 점에서 다른지 살펴보기 위해서 다음 예제를 작성해봅시다. 다음 예제는 List 인터페이스로 구현된 클래스 중에서 ArrayList를 이용한 예입니다.

예제. 자료를 순서 있게 처리하는 List 객체 - [파일 이름 : Collections02.java]

List 인터페이스로 구현한 클래스들의 요소는 위치(인덱스)를 이용해서 접근합니다. List 인터페이스는 Collection 인터페이스로부터 상속되었으므로 Collection 인터페이스에서 제공되는 메서드를 모두 사용할 수 있으며 다음과 같은 메서드가 더 추가되었습니다.



다음에 제시되는 메서드를 살펴보면 List 인터페이스가 요소의 위치로 접근 가능하기에 인덱스로 요소를 찾는 메서드가 대부분입니다.

메서드	설명
int indexOf(Object o)	전달인자로 준 객체를 앞에서부터 찾아 해당 위치를 반환. 못 찾으면 -1 반환
int lastIndexOf(Object o)	객체를 마지막 위치부터 찾음
E get(int index)	위치의 객체를 반환
E set(int index, E element)	index 위치의 요소를 element로 주어진 객체를 대체한다.
void add(int index, E element)	index 위치에 element로 주어진 객체를 저장한다. 해당 위치의 객체는 뒤로 밀려난다.
E remove(int index)	index 위치의 객체를 지운다.

ArrayList 클래스

자바에서의 배열은 생성할 당시에 정한 배열의 길이를 늘이거나 줄일 수 없습니다. 배열을 생성할 때 지정한 길이를 벗어나는 인덱스를 접근하려고 하면 에러가 발생합니다. 하지만, ArrayList 클래스는 원소가 가득 차게 되면 자동적으로 저장영역을 늘려줍니다. 즉, ArrayList 클래스는 가변 길이의 배열이라고 할 수 있습니다. ArrayList 클래스의 또 다른 특징은 다양한 객체를 원소로 저장할 수 있다는 점입니다. HashSet 클래스와 차이점이라면 ArrayList 클래스는 원하는 요소를 인덱스로 접근할 수 있다는 점입니다. get 메서드에 인덱스를 지정하여 해당 위치의 요소를 얻어냅니다.

예제. 리스트 객체의 요소를 인덱스로 접근 - [파일 이름 : Collections04.java]

Iterator 인터페이스

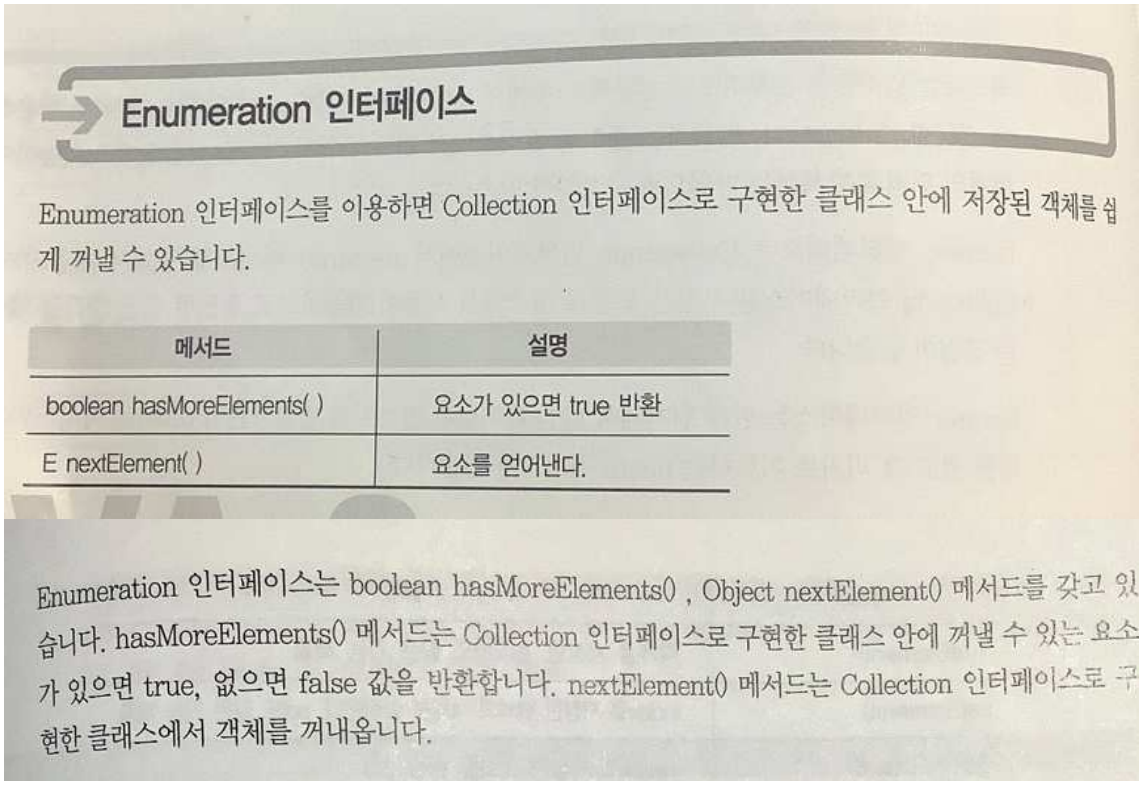
리스트는 원소들을 순차적으로 접근하기 위해서 for문을 이용했지만, 반복자(iterator)를 이용하여 원소에 접근할 수 있습니다. 반복자는 주어진 컬렉션에서 현재의 위치 그리고 다음단계로 이동하여 그곳을 현재의 위치로 반복하는 방법으로 구성되어 있습니다.

Iterator 인터페이스는 Collections 인터페이스에서 iterator() 메서드를 통해서 제공되기 때문에, Collection 인터페이스를 구현한 모든 클래스에서 사용이 가능하므로 동일한 접근 방식을 제공해 준다는 장점이 있습니다.

Iterator 인터페이스는 현재 위치에서 다음 데이터가 있는지를 반환하는 hasNext() 메서드와 다음 데이터를 얻고 그 위치로 이동하는 next() 메서드가 있습니다.

메서드	설명
boolean hasNext()	요소가 있으면 true 반환 없으면 false 반환
E next()	요소를 얻어낸다.

예제. 리스트 객체의 요소를 반복자로 접근 - [파일 이름 : Collections03.java]



Enumeration 인터페이스

Enumeration 인터페이스를 이용하면 Collection 인터페이스로 구현한 클래스 안에 저장된 객체를 쉽게 꺼낼 수 있습니다.

메서드	설명
boolean hasMoreElements( )	요소가 있으면 true 반환
E nextElement( )	요소를 얻어낸다.

Enumeration 인터페이스는 boolean hasMoreElements() , Object nextElement() 메서드를 갖고 있습니다. hasMoreElements() 메서드는 Collection 인터페이스로 구현한 클래스 안에 꺼낼 수 있는 요소가 있으면 true, 없으면 false 값을 반환합니다. nextElement() 메서드는 Collection 인터페이스로 구현한 클래스에서 객체를 꺼내옵니다.

예제. Enumeration 인터페이스 사용방법 - [파일 이름 : EnumerationTest01.java]

Vector 클래스

Vector 클래스는 이미 설명한 ArrayList와 여러 면에서 유사합니다. Vector 클래스 역시 원소가 가득 차게 되면 자동적으로 저장영역을 늘려주는 가변 길이의 배열입니다. 벡터 클래스는 다음과 같은 생성자를 가집니다.

메서드	설명
public Vector()	디폴트 생성자로 빈 벡터 객체를 생성한다.
public Vector(int initialCapacity)	initialCapacity로 지정한 크기의 벡터 객체를 생성한다.
public Vector(int initialCapacity, int capacityIncrement)	initialCapacity로 지정한 크기의 벡터 객체를 생성하되, 새로운 요소가 추가되어 원소가 늘어나야 하면 capacityIncrement만큼 늘어난다.

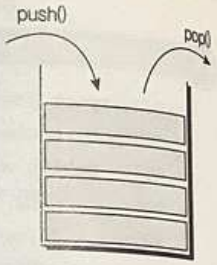
벡터 클래스에서 제공하는 주요 메서드는 다음과 같습니다.

메서드	설명
addElement()	객체를 저장함. add()도 같은 기능 제공
setElement()	index로 지정한 위치의 객체를 설정한다. set도 같은 기능 제공
get(int index)	index로 지정된 요소를 반환한다.
firstElement()	Vector의 첫 번째 요소에 반환한다.
lastElement()	Vector의 마지막 요소를 반환한다.
void trimToSize()	Vector의 용량을 현재 크기로 줄여준다.
int capacity()	벡터의 용량을 반환
Enumeration elements()	벡터 요소들에 대한 Enumeration 객체를 반환

예제. 백터 이용하기 - [파일 이름 : VectorTest01.java]

예제. 백터 요소 검색과 삭제 - [파일 이름 : VectorTest02.java]

Stack 클래스



스택은 한 개 이상의 데이터를 저장할 수 있는 기억공간입니다. 이런 면에서 보면 배열과 매우 유사하지만 아주 큰 차이점이 있습니다. 일반 배열은 첨자로 어느 위치에나 데이터를 랜덤하게 저장할 수 있지만 스택은 항상 순차적으로 저장됩니다. 하지만 스택은 top이라고 불리는 한쪽 끝에서만 삽입 삭제가 행해집니다. 반대쪽은 막혀있다고 보고 입출력을 하지 않습니다.

그러므로 입, 출구가 하나인 기억장소에 데이터를 순차적으로 저장하였다가 가져다 쓰다 보니 맨 처음(First)에 입력(Input)된 값이 가장 나중(Last)에 출력(Output)되고 가장 나중(Last)에 입력(Input)된 것이 가장 먼저(First) 출력(Output)됩니다. 이를 영어로 표현하면 Last Input First Output이고 약어로는 LIFO라 합니다. FILO는 First Input Last Output의 약어입니다.

이러한 모든 개념은 프로그래머가 정할 수 없습니다. 이미 일반화된 개념으로 확립된 것을 자바에서는 Stack 클래스로 제공합니다. Stack 클래스 인스턴스를 생성하기 위해서는 다음과 같이 new 연산자를 이용하여 Stack 클래스의 생성자를 호출합니다.

```
Stack s = new Stack();
```

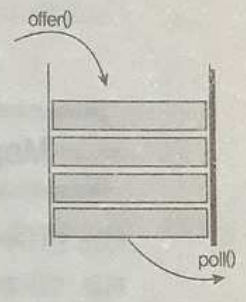
다음은 Stack 클래스의 주요 메서드입니다.

메서드	설명
pop()	스택의 맨 위에서 객체를 제거하고 그 객체를 반환한다.
push (E item)	스택의 맨 위에 객체를 추가한다.
peek()	스택의 맨 위에 있는 객체를 반환한다. 이 때 객체를 스택에서 제거하지는 않는다.
Boolean empty()	현재 스택이 비어있는지를 확인한다. isEmpty도 동일한 기능

예제. LIFO 구조의 스택 다루기 - [파일 이름 : StackTest01.java]

Queue 인터페이스와 LinkedList 클래스

큐는 FIFO(First-In-First-Out) 구조로 알려져 있는 자료구조입니다. 은행에서 업무를 보기 위해서 번호표를 뽑습니다. 먼저 번호표를 뽑은 사람이 먼저 일을 볼 수 있듯이 먼저 들어간 데이터 먼저 나오게 되는 구조입니다. 들어가는 곳과 나가는 곳이 다르기 때문에 큐에 먼저 들어간 데이터가 먼저 나오게 됩니다.



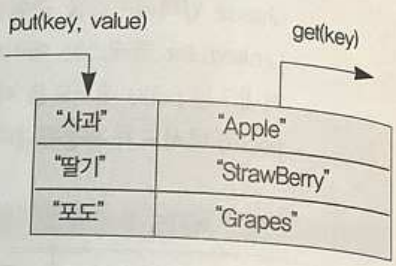
Queue 인터페이스를 구현한 클래스로는 LinkedList 클래스가 있습니다. LinkedList 클래스는 데이터를 저장하기 위한 offer() 메서드와 데이터를 꺼내오는 poll() 메서드를 사용합니다. 데이터를 꺼낼 때 해당 요소를 지우지 않고 값만 얻어낼 수 있는 peek() 메서드가 제공됩니다.

메서드	설명
boolean offer(E o)	큐에 객체를 넣는다.
E poll()	큐에서 데이터를 꺼내온다. 만일 큐가 비어있다면 null을 반환한다.
E peek()	큐의 맨 위에 있는 객체를 반환한다. 이 때 객체를 큐에서 제거하지는 않는다. 그리고 큐가 비어있다면 null을 반환한다.

예제. FIFO 구조의 큐 다루기 - [파일 이름 : LinkedListTest.java]

Map 인터페이스와 Hashtable 클래스

Map 인터페이스는 해쉬 테이블처럼 (키, 객체) 쌍으로 데이터를 저장 관리합니다. Map 인터페이스를 구현한 클래스로 Hashtable 클래스가 있습니다. 지금까지의 컬렉션은 배열처럼 인덱스로 데이터에 접근했습니다. 지금 학습할 해쉬 테이블은 키(key)를 통한 검색으로 데이터에 접근합니다.



해쉬 테이블에 정보를 기록하기 위해서는 put 메서드로 키와 데이터 값을 지정해야 합니다. 저장된 데이터를 검색하기 위해서는 get 메서드에 키를 지정하여 검색합니다.

해쉬 테이블이 제공하는 메서드는 주로 데이터를 추가/검색/삭제하는 기능을 제공합니다.

메서드	설명
put(key, value)	value 데이터를 key 키를 이용하여 해쉬 테이블에 저장한다.
Object get(Object key)	key로 주어진 값을 이용하여 데이터를 검색한다.
remove(Object key)	key로 주어진 값을 이용하여 해쉬 테이블에서 해당 데이터를 삭제한다.
void clear()	해쉬 테이블에 들어있는 키와 해당 객체를 모두 삭제한다.
Enumeration keys( )	해쉬 테이블의 키 요소들에 대한 Enumeration 객체 반환

해쉬 테이블은 지금까지 인덱스로 데이터를 접근하는 배열과는 달리 키(key)를 통한 검색으로 데이터에 접근합니다. 해쉬 테이블은 키와 그에 대응되는 데이터를 저장하는 구조로 되어 있으며, 데이터를 찾을 때에 키를 주고 그 키에 대응되는 데이터를 얻어냅니다. 이러한 특징 때문에 해쉬 테이블은 벡터와 달리 검색을 쉽고 빠르게 할 수 있습니다.

예제. 해쉬 테이블 다루기 - [파일 이름 : HashTableTest.java]

제네릭이 필요하게 된 배경

제네릭이란 컬렉션에서 데이터를 안전성을 위해서 JDK5.0 에 새롭게 추가된 기능입니다. 제네릭에 대해서 학습하기에 앞서 제네릭이 나오게 된 배경부터 살펴보도록 하겠습니다. 자바에서는 데이터를 편리하게 관리하기 위한 컬렉션을 제공합니다.

이런 컬렉션 클래스에는 다양한 형태의 자료들을 담을 수 있습니다. 우리가 컬렉션을 학습하기 위해서 다루었던 예제를 살펴봅시다.

예제 벡터 객체의 요소를 인덱스로 접근-[파일 이름 : Collections04.java]

```
001:import java.util.*;
002:class Collections04 {
003:  public static void main(String[] args) {
004:    Vector vec = new Vector();
005:    vec.add("하나");
006:    vec.add(2);
007:    vec.add(3.42);
008:    vec.add("넷");
009:    vec.add("five");
010:    vec.add(6);
011:    for(int i=0; i<vec.size(); i++)
012:      System.out.println( i + "번째 요소는" + vec.get(i));
013:  }
014:}
```

실행결과

이미 다루었던 예제이므로 생략

벡터 객체는 다양한 형태의 원소로 저장할 수 있습니다. add 메서드의 전달인자가 모든 객체의 최상위 클래스인 Object 형으로 정의되어 있기 때문에 다양한 원소를 add 메서드 하나로 추가할 수 있는 것입니다. 어떤 객체를 추가하더라도 add 메서드는 Object 형으로 관리됩니다.

그렇다면 다양한 형태의 객체를 추가하기 위해서 벡터 객체가 add 메서드의 전달된 값을 업 캐스팅해서 Object형으로 관리하는 것이 편리한 것이기만 한 것일까요? 다음 예제는 벡터 객체가 원소를 Object 형으로 업 캐스팅해서 관리함으로 인하여 생기는 불편함을 보여주는 예입니다.

다음은 벡터 객체에 저장된 문자열을 대문자로 변환하여 출력하는 예입니다. 이런 예의 경우에도 다운 캐스팅해야 한다는 약간의 불편함이 있습니다. 다음은 벡터에 영문자로 구성된 단어를 저장하고 벡터에 저장된 요소를 뽑아내서 대문자로 변경하는 예제를 살펴보겠습니다.

예제. 백터 객체가 원소를 Object 형으로 업 캐스팅해서 관리함으로 인하여 다운 캐스팅을 해야 하는 예 - [파일 이름 : Collections05.java]

제네릭타입

JDK5.0에서는 제네릭을 제공하여 특정 컬렉션 내부에 지정한 클래스 형태가 아닌 객체는 절대 저장하지 못하도록 합니다. 이미 살펴본 예제에서 사용한 벡터 객체는 제네릭을 사용할 수 있는 클래스입니다. 그러므로 벡터 객체를 생성할 때 특정 클래스형을 지정하면 그 클래스 형만을 요소로 가질 수 있게 됩니다. 제네릭을 사용하기 위해서는 클래스 명 다음에 각괄호를 하고 그 내부에 자료 형을 지정합니다.

```
Vector<String> vec = new Vector<String>( );
```

이렇게 선언된 벡터 객체는 요소를 String 객체로 저장하고 관리합니다. 그러므로 list에 add 메서드나 get 메서드를 사용할 수 있는 요소의 자료형 역시 Object가 아닌 String 형태로 사용 가능하게 됩니다.

예제 제네릭을 사용하여 다운 캐스팅을 하지 않아도 되는 예-[파일이름 : Collections06.java]

```
001:import java.util.*;
002:class Collections06 {
003:  public static void main(String[] args) {
004:    Vector<String> vec = new Vector<String>();
005:    vec.add("Apple");
006:    vec.add("banana");
007:    vec.add("oRANGE");
008:    String temp;
009:    for(int i=0; i<vec.size(); i++){
010:      temp=vec.get(i);
011:      System.out.println(temp.toUpperCase());
012:    }
013:  }
014:}
```

실행결과

```
APPLE
BANANA
ORANGE
```

Vector<String>와 같이 제네릭 형태로 벡터 객체를 생성하였기에 String 객체에 벡터 객체의 get 메서드의 결과값을 저장하기 위해서 다운 캐스팅하지 않아도 된다.

제네릭 클래스를 이용하여 004:처럼 선언한 객체는 다운 캐스팅을 하지 않아도 되지만 반드시 String 형 요소만을 저장할 수 있게 됩니다.

확장 for문

이렇게 제네릭 타입으로 선언한 객체는 확장 for문을 사용할 수 있습니다.

```
009:    for(int i=0; i<list.size(); i++){
010:      temp=list.get(i);
011:      System.out.println(temp.toUpperCase());
012:    }
```

JDK 5.0에서는 for문을 확장하였습니다. 확장 for문은 배열이나 컬렉션과 같이 여러 개체가 모여 있는 경우에 개체의 끝을 체크하지 않고도 하나씩 차례대로 접근할 수 있도록 합니다.

확장 for문의 문법적 형태는 다음과 같습니다.

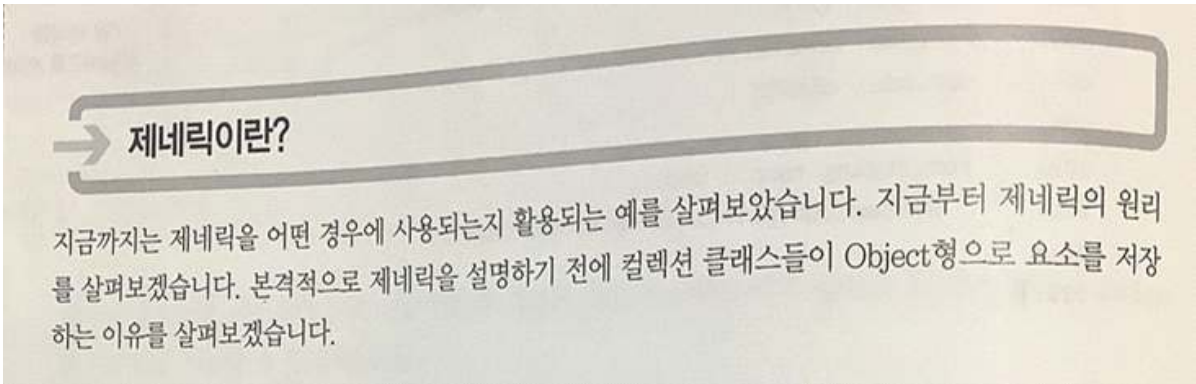
```
for (자료형 접근변수명 : 배열이나 컬렉션 변수명) {
  반복 코드
}
```

접근변수의 자료형은 배열이나 컬렉션의 요소의 자료형과 동일해야 합니다.

예제. 제네릭으로 생성한 백터에 확장 for문 사용한 예 - [파일 이름 : Collections07.java]

해쉬 테이블을 학습하면서 다루었던 예제. 해쉬 테이블 다루기 - [파일 이름 : HashTableTest.java] 파일을 살펴보면서 제네릭을 사용하지 않아 다운 캐스팅을 한 흔적이 보입니다. 제네릭을 사용한 예제로 바꿔 봅시다.

예제. 제네릭을 이용한 해쉬 테이블 - [파일 이름 : HashTableTest02.java]
[파일 이름 : HashTableTest.java] 파일 복사-> HashTableTest02.java



예제. 정해진 자료형만 처리하는 클래스 - [파일 이름 : GenericTest01.java]

예제. Object형으로 클래스 설계 - [파일 이름 : GenericTest02.java]

예제. Object형으로 설계한 클래스의 단점 - [파일 이름 : GenericTest03.java]

자 이제 제네릭에 대한 설명을 할 차례가 온 것 같습니다. 제네릭은 메서드나 클래스의 사용 때에 필요한 데이터 형을 지정할 수 있습니다. 메서드를 정의할 때는 인수의 데이터 형을 결정하지 않습니다. 가상적인 자료형으로 인자를 지정하고 실질적으로 메서드 호출할 때 전달되어지는 실제 값에 의해서 메서드의 인자가 어떤 자료형인지 결정됩니다.

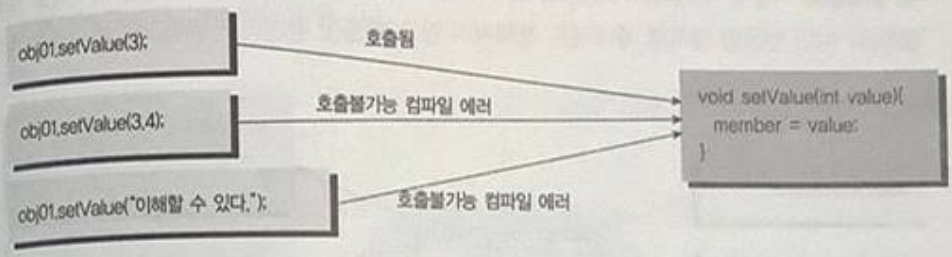
어떤 자료형이 처리될지 미정입니다. 이렇게 아직 정해지지 않은 자료형은 각 괄호로 표현합니다.

```
<T>
```

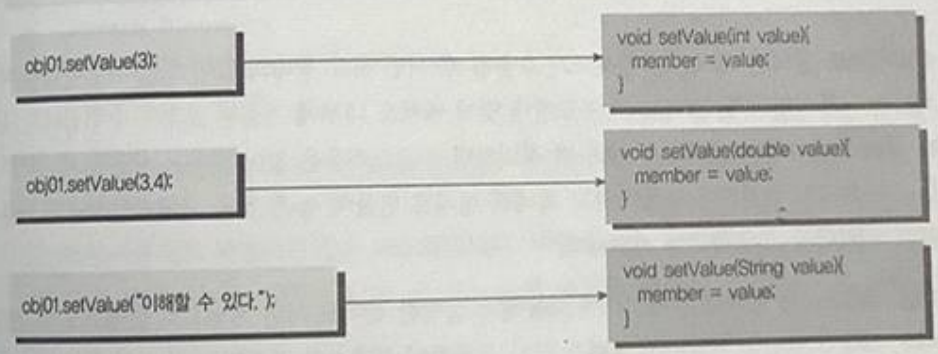
템플릿의 의미로 각 괄호 안에 T라고 기술했지만 E나 다른 표현도 가능합니다. 각 괄호가 중요한 것입니다. 각 괄호로 표시하게 되면 정해지지 않은 자료형이란 의미를 갖습니다.

T가 어떤 형이 될지는 전달되어지는 값에 의해서 결정됩니다. 이것이 바로 제네릭의 출발입니다. 제네릭은 말 그대로 가장 일반적인 자료형에 대한 처리를 해 놓은 것입니다. 일반적이라는 것은 구체적인이란 말과는 반대의 의미 즉, 어떤 것이 될 수도 있다는 의미가 포함됩니다.

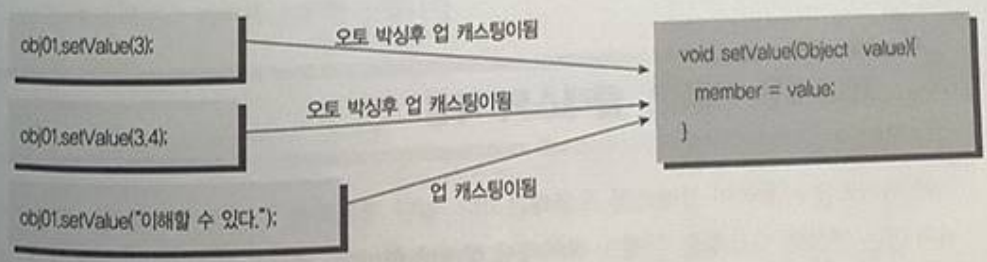
다음은 매개변수가 구체적으로 정해진(제네릭을 제공하지 않는) 자료형을 갖는 메서드의 호출 방식입니다. 한 가지 형태의 자료 형만을 처리하는 클래스입니다.



위와 같이 3가지 방식으로 모두 처리되도록 하기 위해서 클래스가 3번 정의되어야 합니다. 또 다른 자료형의 처리를 원한다면 계속해서 클래스를 정의해야 합니다. 해당 클래스에 구현된 알고리즘은 동일하지만 처리할 데이터의 유형만 달라진다고 해서 클래스를 계속 정의해 나가야 한다면 참 많이 수고스럽겠지요^^. 서로 다른 자료형에 대해서 3번 클래스를 정의한 경우입니다.



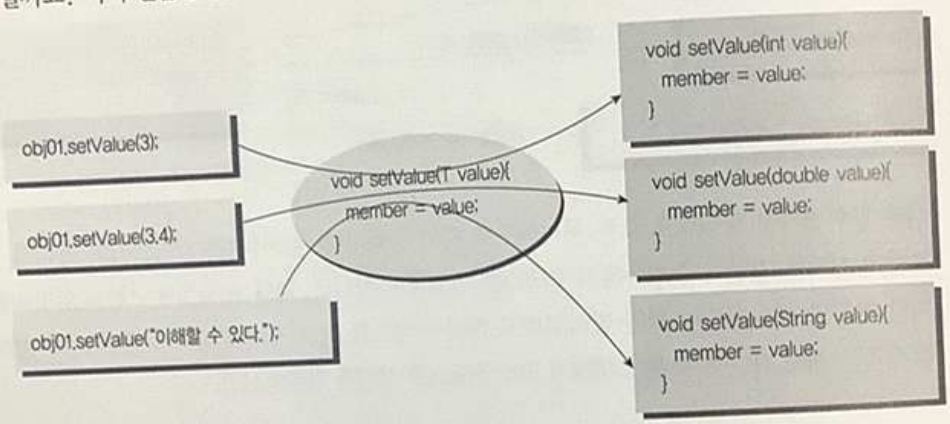
그래서 생각해 낸 것이 Object 형으로 데이터를 처리하도록 클래스를 설계한 것입니다.



데이터가 저장될 때 업캐스팅이 일어납니다. 메서드를 호출할 때에는 3이란 기본 자료형을 전달해 주었

는데 Object 형으로 저장되는 이유는 기본 자료형인 3이 Integer 형으로 오토 박싱이 일어나기 때문입니다. 이렇게 저장된 데이터를 가져다 사용할 경우는 어떻게 해야 할까요? 만일 저장된 데이터 형태인 Object형을 그대로 사용한다면 별 문제 없지만, 다운 캐스팅해야 할 일이 생기면 저장할 때 어떤 형태의 데이터를 저장했는지 모른다면 어떤 자료형태로 다운 캐스팅해야 할지 애매모호하게 됩니다.

그래서 여러 자료 형에 대해서 처리 가능하도록 클래스를 만들되 클래스가 처리할 자료형을 객체 생성할 때 정해놓고 사용할 수 있도록 하는 클래스가 필요하게 되었습니다. 그럼 이번 제네릭을 적용한다면 어떨까요? 이미 언급한 것처럼 제네릭은 정해지지 않은 템플릿 형태의 자료형으로 메서드를 정의합니다.



제너릭으로 정의된 메서드는 메서드가 호출될 때 어떤 자료 형태의 값이 오느냐에 따라서 이에 맞는 메서드가 만들어집니다. 그리고 그 자료형에 맞게 메서드 내부에 기술된 로직이 수행됩니다. 동일한 로직에 대해서 자료형만 달리해서 여러 번 정의하던 수고스러움은 없어집니다. 사용할 때 결정되는 자료형으로 구체적인 메서드가 만들어지고 제네릭 형태로 만들어 놓은 것은 구체적인 메서드들을 찍어 내기 위한 형틀(템플릿)에 불과한 것입니다.

이번 예제에서는 제네릭을 메서드에 적용해서 설명한 것이고 이런 개념으로 클래스도 디자인할 수 있습니다. 제네릭 개념을 이용하면 클래스 역시 구체적인 클래스가 정의되는 것이 아니고, 형틀만 만들어 놓습니다. 객체가 생성될 때 구체적인 클래스가 본격적으로 만들어지는 것입니다. 이렇게 탁월한 제네릭으로 어떻게 클래스를 정의하고 메서드를 정의하는지 문법적인 사항을 살펴보도록 하겠습니다.

제네릭을 사용한 클래스의 작성

이제 제네릭을 사용하여 클래스를 작성해봅시다. 일단 제네릭을 지원하기 위해서는 템플릿 형태의 정해지지 않은 가상의 자료형을 클래스 정의에서 정해야 합니다. 보통은 1글자의 영문자 대문자를 사용합니다. 여기서는 T를 사용하였습니다.

```
[접근제어자] [기타제어자] class 클래스이름<T> [extends 상위_클래스] [implements 인터페이스]
{
}
```

예를 들어 〈T〉라는 가상적인 자료형을 갖는 GenericClass이라는 제네릭을 지원하는 클래스를 만들기 위해서는 다음과 같이 정의를 합니다.

```
class GenericClass<T>{
}
```

멤버와 메서드의 정의

〈T〉라는 자료형으로 클래스를 정의한 후에는 클래스 내부에서 T를 일반적인 자료형처럼 이용하면 됩니다. 즉, T로 멤버변수를 선언합니다. 메서드의 매개변수나 반환형에 T를 자료형으로 사용합니다.

```
private T member;
public void setValue(T value){
  member = value;
}
```

T는 아직 아무것도 확정되지 않은 자료형입니다. 해당 클래스를 정의할 때 아무것도 정해지지 않은 T는 어떤 것도 될 수 있다는 뜻도 됩니다. 즉, T는 객체 생성할 때 어떤 자료형으로도 탈바꿈 할 수 있게 됩니다.

제네릭의 이용

이제 정의된 클래스의 T에 대해서 실질적인 자료형을 정해주고 객체 생성을 해봅시다. 다음은 객체 생성 전에 레퍼런스 변수를 선언한 것입니다.

```
GenericClass<Double> obj01;
```

해당 레퍼런스 변수에 new로 객체 생성합시다.

```
obj01=new GenericClass<Double>( );
```

일단 이렇게 되면 Double이 클래스의 정의에 사용되었던 T를 대신하여 코드 전반에 걸쳐서 모두 바뀌게 됩니다.

예제. 제네릭의 레퍼런스 형변환 - [파일 이름 : Collections08.java]

제네릭과 와일드 카드

하지만 제네릭으로 선언된 객체들은 상속 관계에서도 형변환이 암시적으로 이루어지지 않습니다. 하지만 어느 정도의 융통성을 위해서 와일드 카드를 사용합니다.

```
Vector<? extends Object> objlist;
```

이 선언의 의미는 objlist 레퍼런스 선언 시 ? extends 다음에 기술한 Object 클래스 형의 하위 클래스 형으로 선언한 제네릭 레퍼런스 형을 가리킬 수 있다는 의미입니다.

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

즉, String 클래스는 Object 클래스 형 하위 클래스이므로 list를 objlist에 대입하여 사용할 수 있습니다.

예제. 제네릭의 레퍼런스 형변환 - [파일 이름 : Collections08.java]

TIP

제네릭의 또 다른 레퍼런스 형변환과 와일드 카드

제네릭에서 와일드 카드를 이용한 레퍼런스 형변환은 위에서 언급한 형태 말고 다음과 같은 두가지 형태가 더 있습니다.

:: ? : 모든 객체 타입을 다 배치할 수 있다.

:: 〈? super 타입〉 : 명시된 타입의 상위 클래스형을 배치할 수 있다.