클릭하세요 자바

5부 자바에서 자주 사용되는 클래스

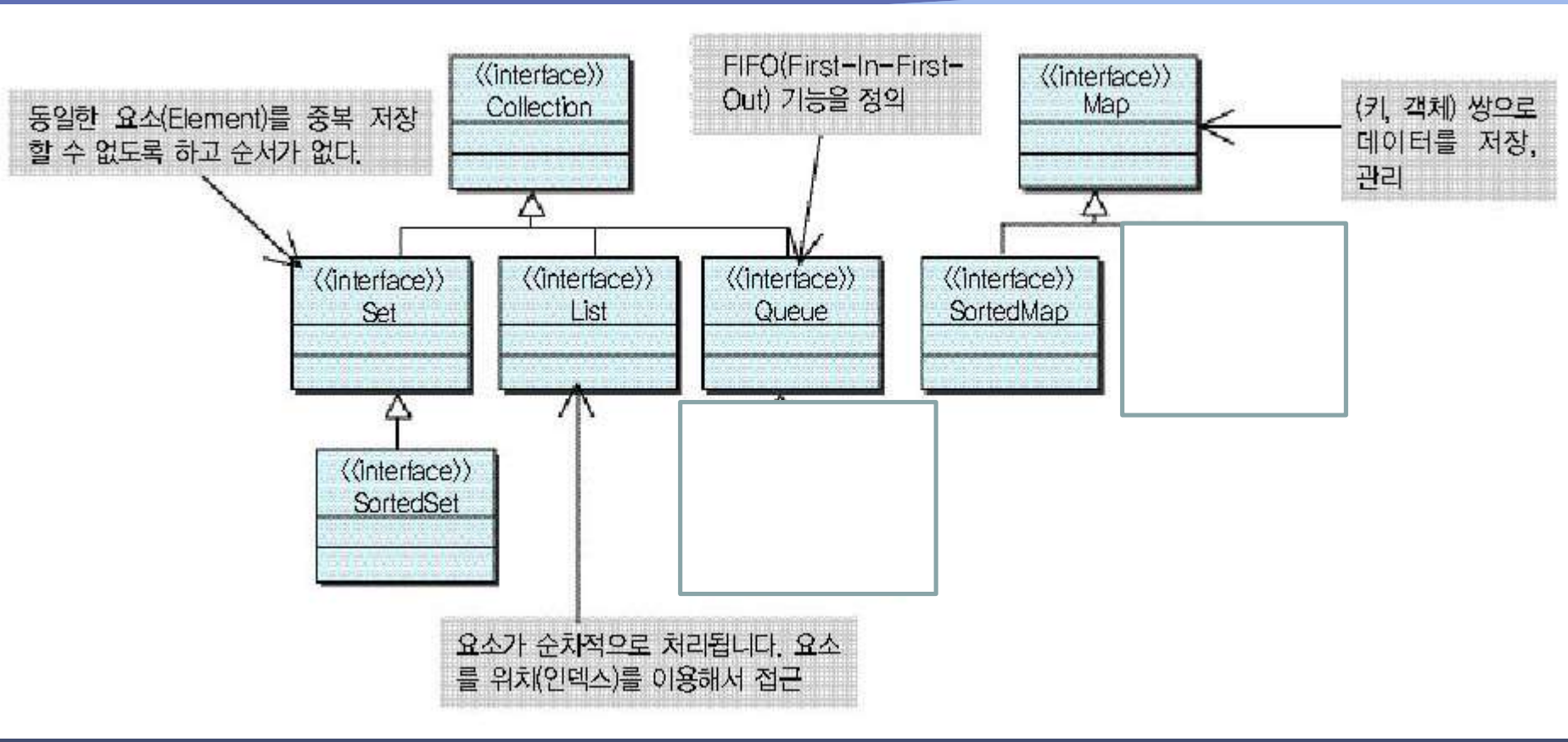
– 24장 컬렉션 클래스와 제네릭

최문환

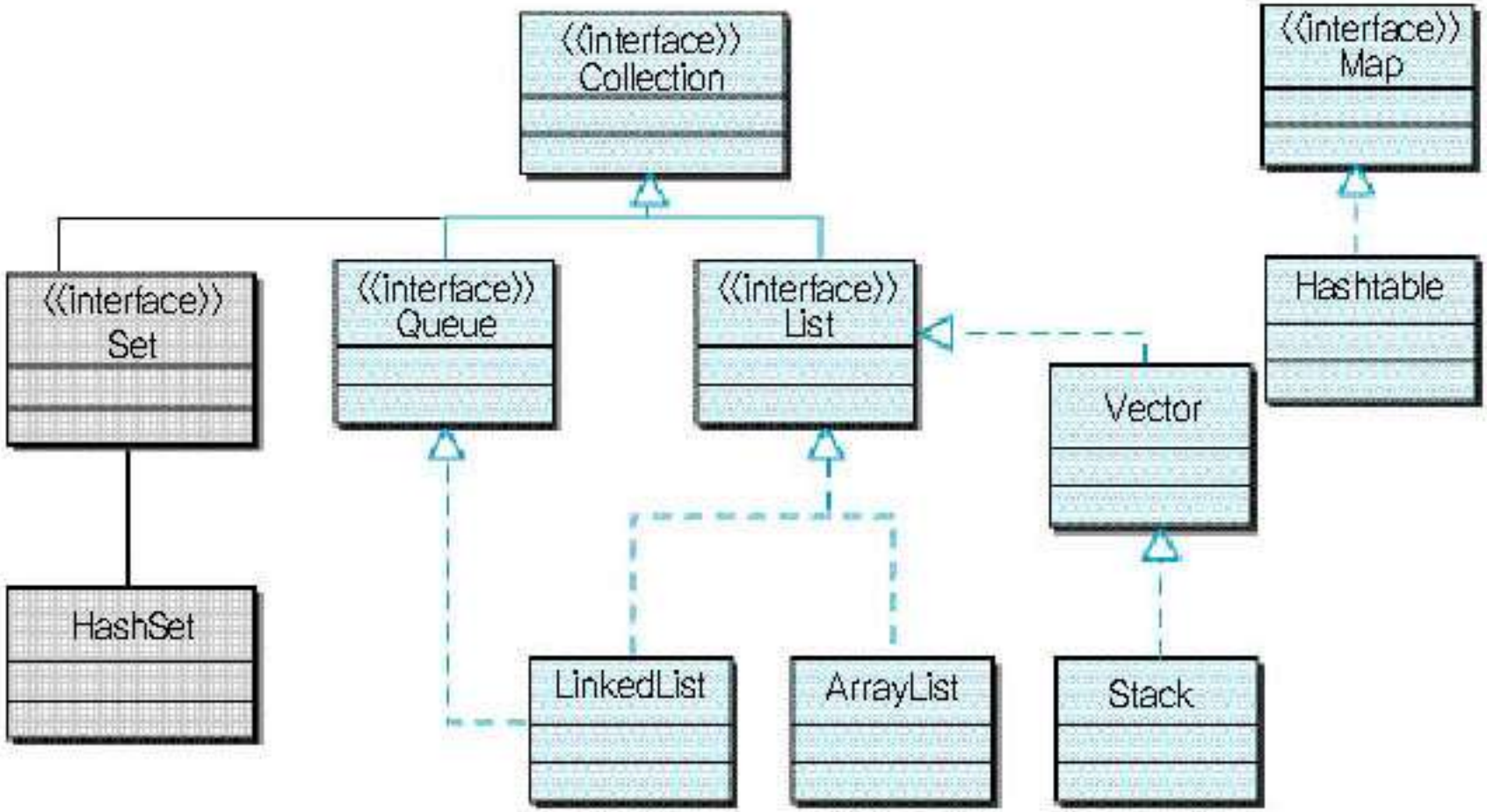
24장 컬렉션과 제네릭

1. 컬렉션
2. Set 인터페이스
3. List 인터페이스
4. Stack 클래스
5. Queue 인터페이스와 LinkedList 클래스
6. Map 인터페이스와 Hashtable 클래스
7. 제네릭이란?

1. 컬렉션


동일한 요소(Element)를 중복 저장할 수 없도록 하고 순서가 없다.
<<interface>>
Collection
FIFO(First-In-First-Out) 기능을 정의
<<interface>>
Map
(키, 객체) 쌍으로 데이터를 저장, 관리
<<interface>>
Set
<<interface>>
List
<<interface>>
Queue
<<interface>>
SortedMap
<<interface>>
SortedSet
요소가 순차적으로 처리됩니다. 요소를 위치(인덱스)를 이용해서 접근

구현 컬렉션 계층도


«interface»
Collection
«interface»
Map
«interface»
Set
«interface»
Queue
«interface»
List
Hashtable
Vector
HashSet
LinkedList
ArrayList
Stack

▪ Set 인터페이스

<예제> 자료를 순서 없이 처리하는 Set 객체

```
001:import java.util.*;
002:class Collections01 {
003:  public static void main(String[] args) {
004:
005:    Set set = new HashSet();   //해쉬 셋 객체를 생성한다.
006:    System.out.println(" 요소의 개수->" + set.size());
007:    set.add("하나");              //해쉬 셋 객체에 요소를 추가한다.
008:    set.add(2);
009:    set.add(3.42);
010:    set.add("넷");
011:    set.add("five");
012:    set.add(6);
013:    System.out.println(" 요소의 개수->" + set.size());
014:    System.out.println(set);  //set 객체 값을 출력한다.
015:  }
016:}
```

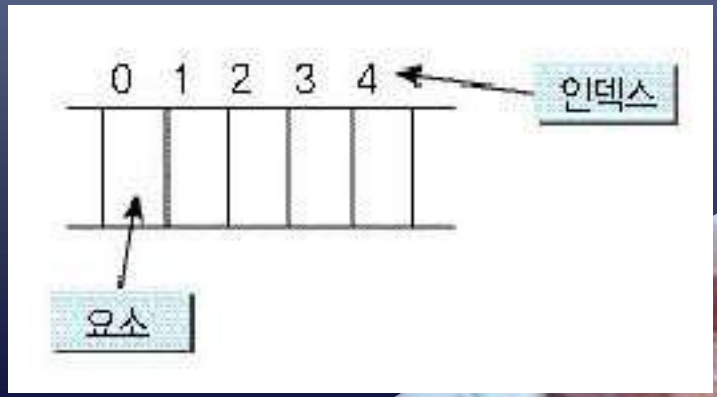
List 인터페이스

<예제> 자료를 순서 있게 처리하는 List 객체

```
001:import java.util.*;
002:class Collections02 {
003:  public static void main(String[] args) {
004:
005:    List list=new ArrayList(); //ArrayList 객체 객체를 생성한다.
006:
007:    list.add("하나");          //ArrayList  객체에 요소를 추가한다.
008:    list.add(2);
009:    list.add(3.42);
010:    list.add("넷");
011:    list.add("five");
012:    list.add(6);
013:
014:    System.out.println(list);
015:  }
016:}
```

List 인터페이스

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)	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

<예제> 리스트 객체의 요소를 인덱스로 접근

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

Iterator 인터페이스

<예제> 리스트 객체의 요소를 반복자로 접근

```
001:import java.util.*;
002:class Collections03 {
003:  public static void main(String[] args) {
004:    List list = new ArrayList();
005:    list.add("하나");
006:    list.add(2);
007:    list.add(3.42);
008:    list.add("넷");
009:    list.add("five");
010:    list.add(6);
011:    //Collections 인터페이스에서 제공하는 iterator() 메서드로 반복자를 구해온다.
012:    Iterator elements=list.iterator();
013:    while(elements.hasNext())          //요소가 있다면
014:      System.out.println(elements.next()); //요소를 얻어내어 출력
015:  }
016}
```

Enumeration 인터페이스

<예제> Enumeration 인터페이스 사용방법

```
001:import java.util.*;
002:public class EnumerationTest01 {
003:  public static void main(String[] args){
004:    Vector vec =  new Vector();
005:    for(int i=1; i<5; i++)
006:      vec.add(new Integer(i*10));
007:    Enumeration enu = vec.elements(); //벡터 요소들에 대한 Enumeration 객체를 반환
008:    while( enu.hasMoreElements())          //벡터에 요소가 있으면
009:    System.out.println( enu.nextElement() );  //요소를 얻어낸다.
010:  }
011:}
```

Vector 클래스

★ 생성자

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.

Vector 클래스

★ 주요 메서드

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> 벡터 이용하기

```
001:import java.util.*;
002:public class VectorTest01 {
003:  public static void main(String[] args) {
004:    //4개의 요소를 저장할 수 있는 벡터 객체 생성, 3개씩 증가
005:    Vector vec =  new Vector(4, 3);
006:    System.out.println("벡터의 크기는 " + vec.size());
007:    System.out.println("벡터의 용량은 " + vec.capacity
008:    for(int i=0; i<5; i++)
009:      vec.add(i*10);
010:    System.out.println("벡터의 크기는 " + vec.size());
011:    System.out.println("벡터의 용량은 " + vec.capacity());
012:    System.out.println("첫 번째 요소는 " + vec.firstElement());
013:    System.out.println("두 번째 요소는 " + vec.get(1));
014:    System.out.println("마지막 요소는 " + vec.lastElement());
015:
016:   System.out.println("₩n >> 저장된 요소 전체 출력 <<" );
017:   for(int i=0; i<vec.size(); i++)
018:     System.out.print("    " + vec.get(i));
019:   System.out.println();
020:  }
021:}
```

<예제> 벡터 요소 검색과 삭제

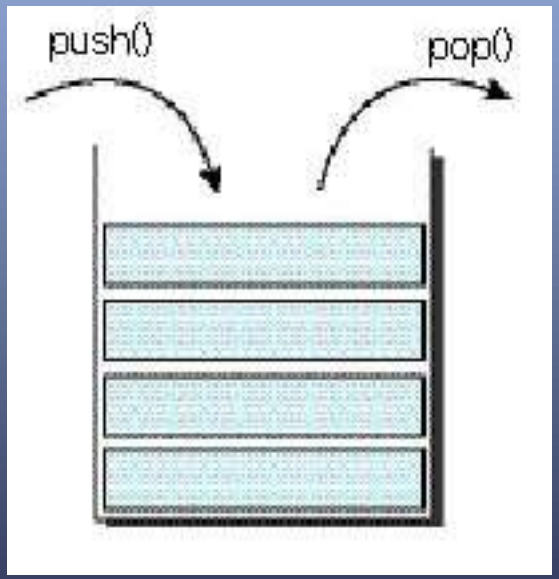
```
001:import java.util.*;
002:public class VectorTest02 {
003:   public static void main(String[] args) {
004:     //디폴트 생성자로 선언된 벡터객체의 초기 용량은 10, 증가량은 1
005:     Vector vec =  new Vector( );
006:     double[] arr = new double[] { 38.6, 9.2, 45.3, 6.1, 4.7, 1.6 };
007:
008:     for(int i=0; i<6; i++)
009:       vec.add(arr[i]);     //벡터에 실수형 데이터 추가하기
010:
011:     System.out.println("₩n >> 요소 전체 출력 <<" );
012:     for(int i=0; i<vec.size(); i++)
013:       System.out.print("    " + vec.get(i));
014:     System.out.println();
015:
016:     double searchData=6.1;  //1.1
017:     int index=vec.indexOf(searchData
018:     if(index != -1)   //찾았다면 (index가 -1이 아니면 찾은 것임)
019:       System.out.println( "₩n 검색 성공 : " + index );
020:     else   //index가 -1이면 찾지 못한 것임
021:       System.out.println( "₩n 검색 실패 : " + index  );
```

<예제> 벡터 요소 검색과 삭제

```
023:    double delData=45.3;
024:    if(vec.contains(delData)){   //delData가 벡터에 포함되어 있다면
025:      vec.remove(delData);     //해당 요소 삭제
026:      System.out.println( "₩n 삭제 완료 " );
027:    }
028:
029:    System.out.println("₩n >> 요소 전체 출력 <<" );
030:    for(int i=0; i<vec.size(); i++)
031:      System.out.print("   " + vec.get(i));
032:    System.out.println();
033:  }
034:}
```

Stack 클래스

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.

Stack 클래스

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

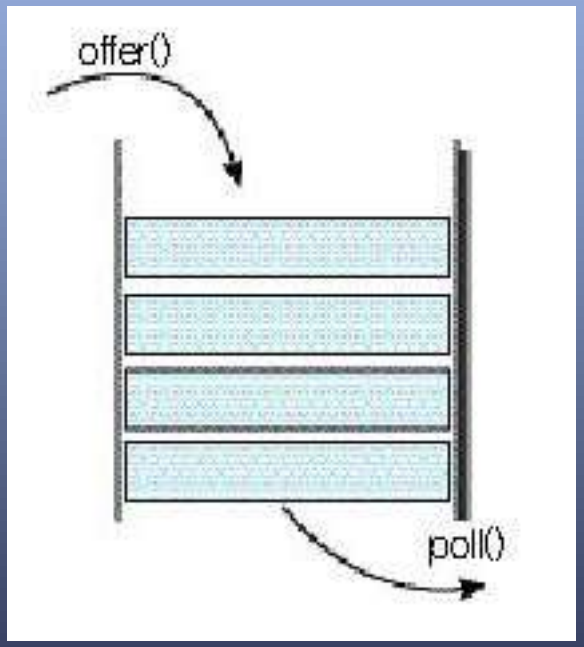
메서드	설명
pop()	스택의 맨 위에서 객체를 제거하고 그 객체를 반환한다.
push (E item)	스택의 맨 위에 객체를 추가한다.
peek()	스택의 맨 위에 있는 객체를 반환한다. 이 때 객체를 스택에서 제거하지는 않는다.
Boolean empty()	현재 스택이 비어있는지를 확인한다. isEmpty도 동일한 기능

<예제> LIFO 구조의 스택 다루기

```
001:import java.util.*;
002:public class StackTest01 {
003:  public static void main(String[] args) {
004:    Stack myStack = new Stack();   //스택 객체 생성
005:    myStack.push("1-자바");        //스택에 객체를 추가한다.
006:    myStack.push("2-C++");
007:    myStack.push("3-API");
008:    myStack.push("4-MFC");
009:
010:    while(!myStack.isEmpty()) //현재 스택이 비어있지 않다면
011:      System.out.println( myStack.pop()); //스택에 저장된 객체를 꺼내온다.
012:  } //LIFO 구조이므로 역순으로 출력된다.
013:}
```

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 인터페이스와 LinkedList 클래스

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

```
001:import java.util.*;
002:class LinkedListTest {
003:  public static void main(String[] args) {
004:    LinkedList myQue = new LinkedList();   //큐 객체를 생성한다.
005:    myQue.offer("1-자바");                 //큐에 객체를 넣는다.
006:    myQue.offer("2-C++");
007:    myQue.offer("3-API");
008:    myQue.offer("4-MFC");
009:
010:    while(myQue.peek() != null)        //큐가 비어있지 않다면
011:      System.out.println(myQue.poll()); //큐에서 데이터를 꺼내온다.
012:  }
013: }
```


Map 인터페이스와 Hashtable 클래스

key(키)	value(값)
"사과"	"Apple"
"딸기"	"StrawBerry"
"포도"	"Grapes"

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

<예제> 해쉬 테이블 다루기

```
001:import java.util.*;
002:class HashTableTest{
003:  public static void main(String[] args) {
004:    Hashtable ht= new Hashtable();
005:    // 해쉬 테이블에 키/데이터를 입력한다.
006:    ht.put("사과", "Apple");
007:    ht.put("딸기", "Strawberry");
008:    ht.put("포도", "Grapes");
009:    // 해쉬 테이블의 값을 키를 이용하여 얻는다.
010:    String Val = (String)ht.get("포도");
011:    if(Val != null)
012:      System.out.println("포도-> " + Val);
013:
014:    Enumeration Enum = ht.keys();  //해쉬 테이블의 키 요소들에 대한 Enumeration 객체 반환
015:    while(Enum.hasMoreElements()){     //키 요소가 있으면
016:      Object k = Enum.nextElement();   //키 요소를 얻어낸다.
017:      Object v = ht.get(k);            //키 요소로 주어진 값을 이용하여 데이터를 검색한다.
018:      System.out.println(k + " : "+ v ); //키 요소와 데이터를 출력한다.
019:    }
020:  }
021:}
```

제네릭이 필요하게 된 배경

```
001:import java.util.*;
002:class Collections04_A {
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:}
```


제네릭이 필요하게 된 배경

벡터 객체가 원소를 Object 형으로 업 캐스팅해서 관리함으로 인하여 다운 캐스팅을 해야 하는 예

```
001:import java.util.*;
002:class Collections05 {
003:  public static void main(String[] args) {
004:    Vector vec = new Vector();     //벡터 객체는 다양한 자료들을 저장할 수 있으므로
005:    vec.add("Apple");    //내부적으로 요소들을 Object형으로 관리하고 있음
006:    vec.add("banana");
007:    vec.add("oRANGE");
008:    String temp; //toUpperCase() 메서드를 사용하기 위해서 String 변수 선언
009:    for(int i=0; i<vec.size(); i++){
010:      //temp=vec.get(i);         //컴파일 상의 에러 메시지 발생
011:      temp=(String) vec.get(i);//Object 형으로 관리되는 요소를 가져다 사용할 때 다운 캐스팅해야함
012:      System.out.println(temp.toUpperCase());
013:    }
014:  }
015:}
```

제네릭타입

<예제> 제네릭을 사용하여 다운 캐스팅을 하지 않아도 되는 예

```
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:}
```

확장 for문

<예제> 제네릭으로 생성한 벡터에 확장 for문 사용한 예

```
001:import java.util.*;
002:class Collections07 {
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:
009:    for(String temp : vec)
010:      System.out.println(temp.toUpperCase());
011:  }
012:}
```

제네릭타입

<예제> 제네릭을 이용한 해쉬 테이블

```
001:import java.util.*;
002:class HashTableTest02{
003:  public static void main(String[] args) {
004:    Hashtable<String, String> ht= new Hashtable<String, String>( );
005:    // 해쉬 테이블에 키/데이터를 입력한다.
006:    ht.put("사과", "Apple");
007:    ht.put("딸기", "StrawBerry");
008:    ht.put("포도", "Grapes");
009:    // 해쉬 테이블의 값을 키를 이용하여 얻는다.
010:    String Val = ht.get("포도");
011:    if(Val != null) {
012:      System.out.println("포도-> " + Val);
013:    }
014:    Enumeration<String> Enum = ht.keys();
015:    while(Enum.hasMoreElements()){
016:      String  k = Enum.nextElement();
017:      String  v = ht.get(k);
018:      System.out.println(k + " : "+ v );
019:    }
020:  }
021:}
```

제네릭타입

<예제> 제네릭을 사용하여 다운 캐스팅을 하지 않아도 되는 예

```
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:}
```

제네릭이란?

```
001:class TestClass {
002:  private int member;
003:  public void setValue(int value){
004:    member = value;
005:  }
006:   public int getValue( ){
007:    return member;
008:  }
009:}
010:class GenericTest01{
011:  public static void main(String[] args) {
012:     TestClass obj01=new TestClass();
013:     obj01.setValue(3);
014:     System.out.println("되돌리는 값은->" + obj01.getValue( ));
015:     obj01.setValue(3.4);     //실제 전달되는 값이 실수형이면 에러 발생
016:     System.out.println("되돌리는 값은->" + obj01.getValue( ));
017:     obj01.setValue("이해할 수 있다."); //실제 전달되는 값이 문자열형이면 에러 발생
018:     System.out.println("되돌리는 값은->" + obj01.getValue( ));
019:  }
020:}
```

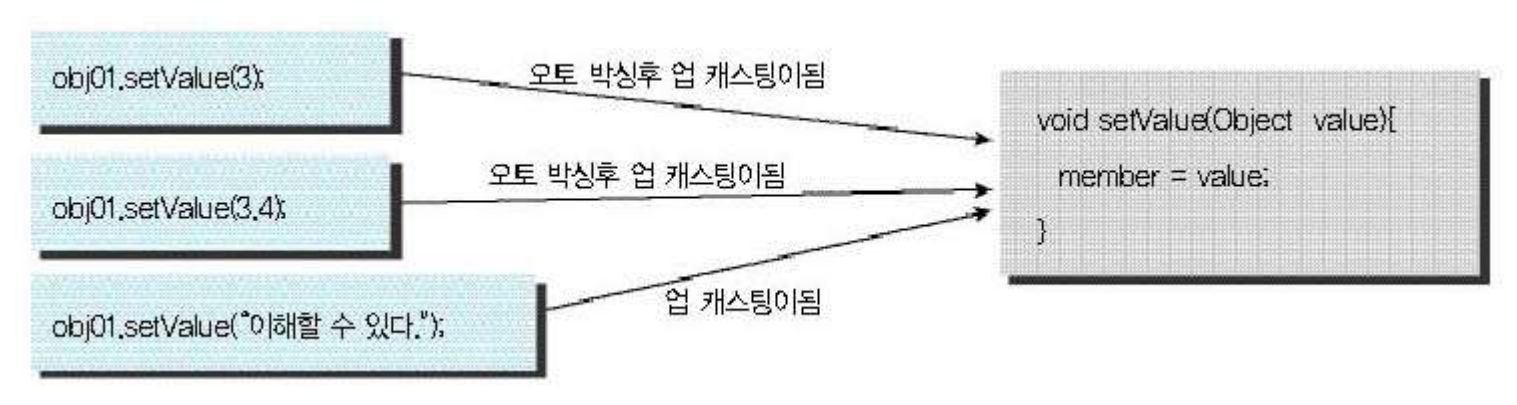
<예제> Object형으로 클래스 설계

```
001:class TestClass {
002:   private Object member;
003:   public void setValue(Object value){
004:     member = value;
005:   }
006:    public Object getValue( ){
007:     return member;
008:   }
009:}
010:class GenericTest02{
011:   public static void main(String[] args) {
012:      TestClass obj01=new TestClass();
013:      obj01.setValue(3);
014:      System.out.println("되돌리는 값은->" + obj01.getValue( ));
015:      obj01.setValue(3.4);
016:      System.out.println("되돌리는 값은->" + obj01.getValue( ));
017:      obj01.setValue("이해할 수 있다.");
018:      System.out.println("되돌리는 값은->" + obj01.getValue( ));
019:   }
020:}
```

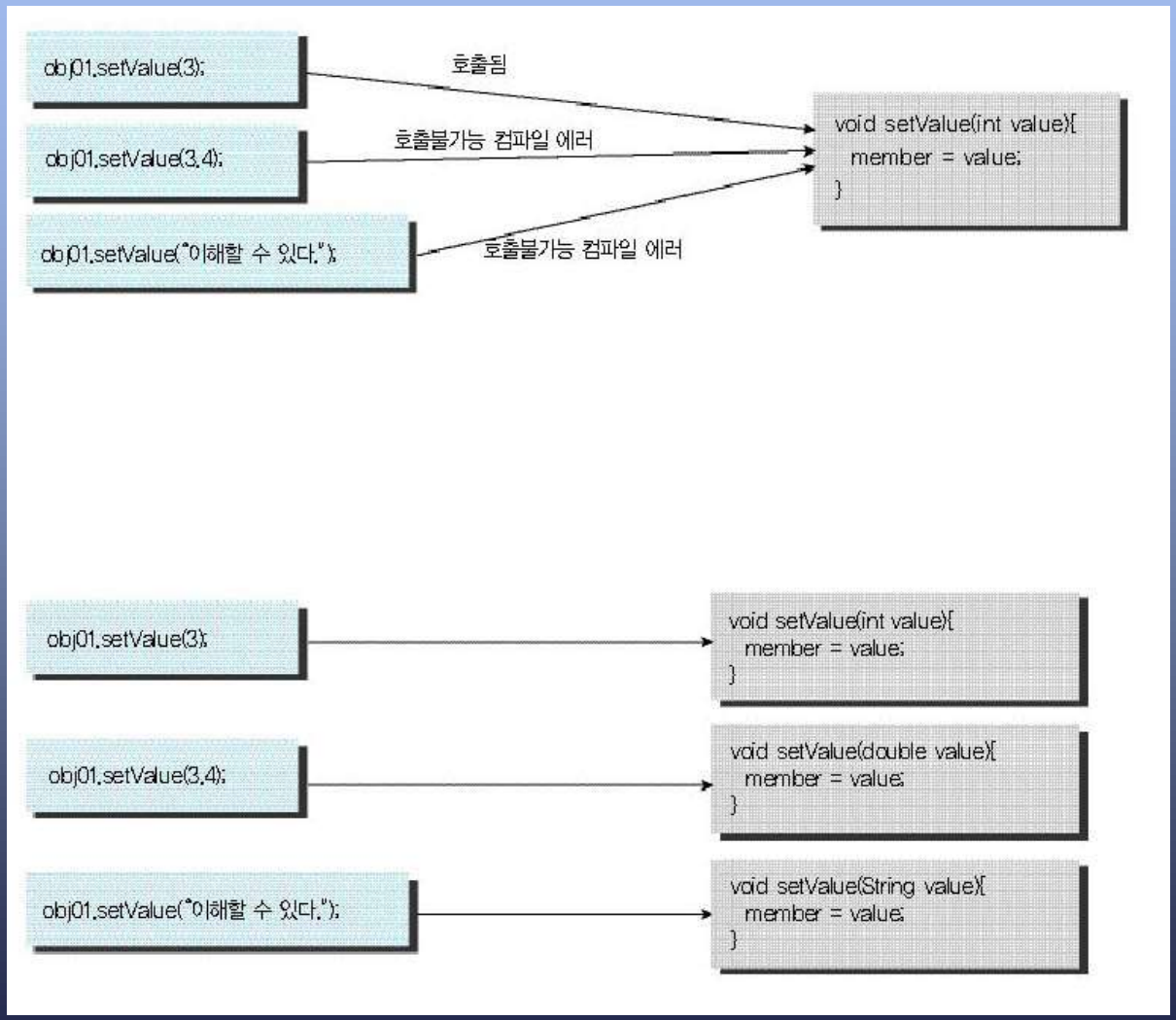
<예제> Object형으로 설계한 클래스의 단점

```
001:class TestClass {
002:  private Object member;
003:  public void setValue(Object value){
004:    member = value;
005:  }
006:   public Object getValue( ){
007:    return member;
008:  }
009:}
010:class GenericTest03{
011:  public static void main(String[] args) {
012:     TestClass obj01=new TestClass();
013:     obj01.setValue("apple");
014:     String temp;
015:     //temp=obj01.getValue( );
016:     temp=(String)obj01.getValue( );
017:     System.out.println(temp.toUpperCase());
018:  }
019:}
```

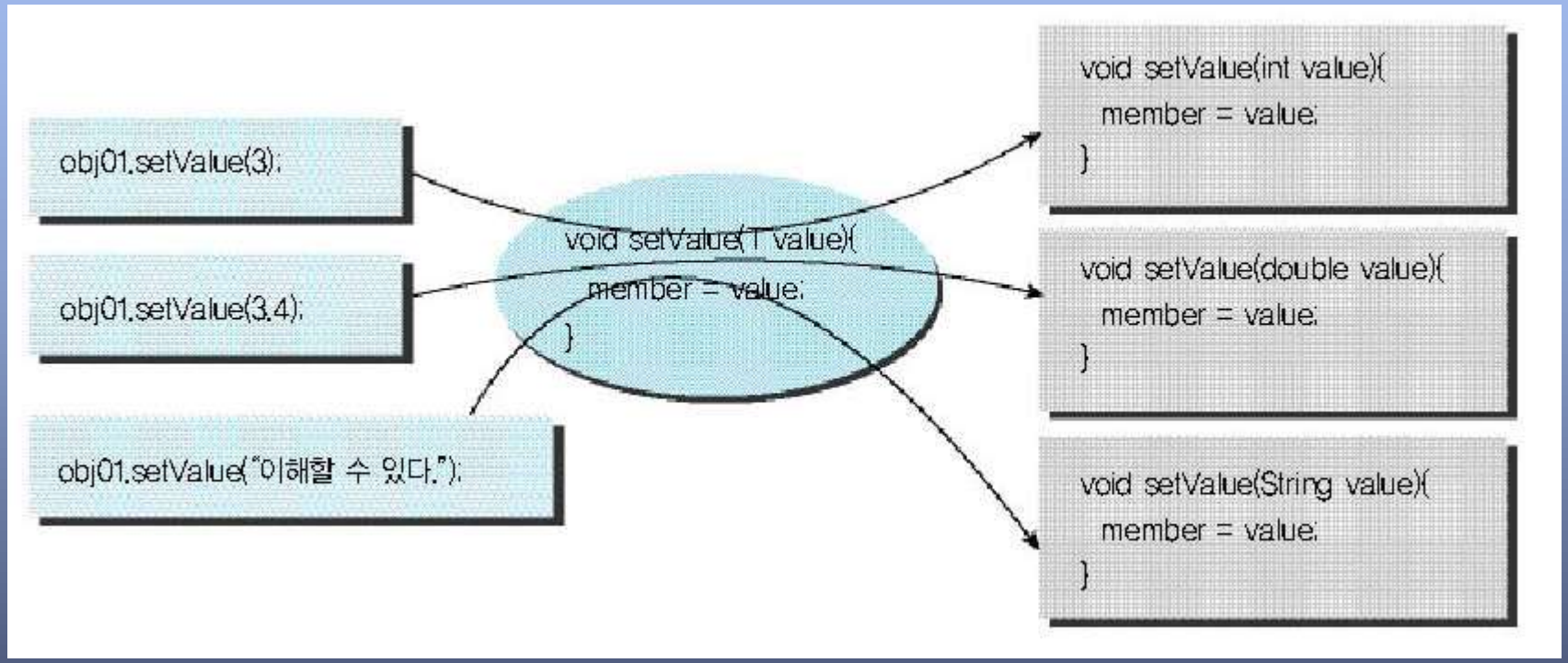

제네릭이란?


obj01.setValue(3);
오토 박싱후 업 캐스팅이됨
obj01.setValue(3.4);
오토 박싱후 업 캐스팅이됨
obj01.setValue("이해할 수 있다.");
업 캐스팅이됨
void setValue(Object value){
member = value;
}

제네릭이란?


obj01.setValue(3);
호출됨
obj01.setValue(3.4);
호출불가능 컴파일 에러
obj01.setValue("이해할 수 있다.");
호출불가능 컴파일 에러
void setValue(int value){
member = value;
}
obj01.setValue(3);
void setValue(int value){
member = value;
}
obj01.setValue(3.4);
void setValue(double value){
member = value;
}
obj01.setValue("이해할 수 있다.");
void setValue(String value){
member = value;
}

제네릭이란?


obj01.setValue(3);
obj01.setValue(3.4);
obj01.setValue("이해할 수 있다.");
void setValue(T value){
member = value;
}
void setValue(int value){
member = value;
}
void setValue(double value){
member = value;
}
void setValue(String value){
member = value;
}

제네릭을 사용한 클래스의 작성

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

}
```

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

★ 멤버와 메서드의 정의

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

제네릭의 이용

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

<예제> 제네릭 클래스 설계

```
001:class GenericClass<T> {
002:  private T member;
003:  public void setValue(T value){
004:    member = value;
005:  }
006:   public T getValue( ){
007:    return member;
008:  }
009:}
```

<예제> 제네릭 클래스 설계

```
010:class GenericTest05{
011:  public static void main(String[] args) {
012:     GenericClass<Double> obj01=new GenericClass<Double>();
013:     obj01.setValue(3.4);
014:     System.out.println("되돌리는 값은->" + obj01.getValue( ));
015:
016:     GenericClass<Integer> obj02=new GenericClass<Integer>( );
017:     obj02.setValue(new Integer(10));
018:     System.out.println("되돌리는 값은->" + obj02.getValue( ));
019:
020:     GenericClass<String> obj03=new GenericClass<String>( );
021:     obj03.setValue("이해할 수 있다.");
022:     System.out.println("되돌리는 값은->" + obj03.getValue( ));
023:  }
024:}
```

<예제> 제네릭의 레퍼런스 형변환

```
001:import java.util.*;
002:class Collections08 {
003:  public static void main(String[] args) {
004:    Vector<String> list = new Vector<String>();
005:    list.add("Apple");
006:    list.add("banana");
007:    list.add("oRANGE");
008:
009:    Vector<Object> objlist;
010:    objlist=list;
011:
012:    for(Object obj : objlist){
013:            System.out.println(obj);
014:        }
015:  }
016:}
```

제네릭와 와일드 카드

```
001:import java.util.*;
002:class Collections09 {
003:  public static void main(String[] args) {
004:
005:   Vector<String> list = new Vector<String>();
006:
007:    list.add("Apple");
008:    list.add("banana");
009:    list.add("oRANGE");
010:
011:    Vector<? extends Object> objlist;
012:    objlist=list;
013:
014:    for(Object obj : objlist)
015:       System.out.println( obj);
016:
017:    System.out.println("==============================");
018:    for(Object obj : objlist){
019:      String temp=(String)obj;  //다운 캐스팅해야 한다.
020:      System.out.println(temp.toUpperCase());
021:    }
022:  }
023:}
```