자바 프로그래밍

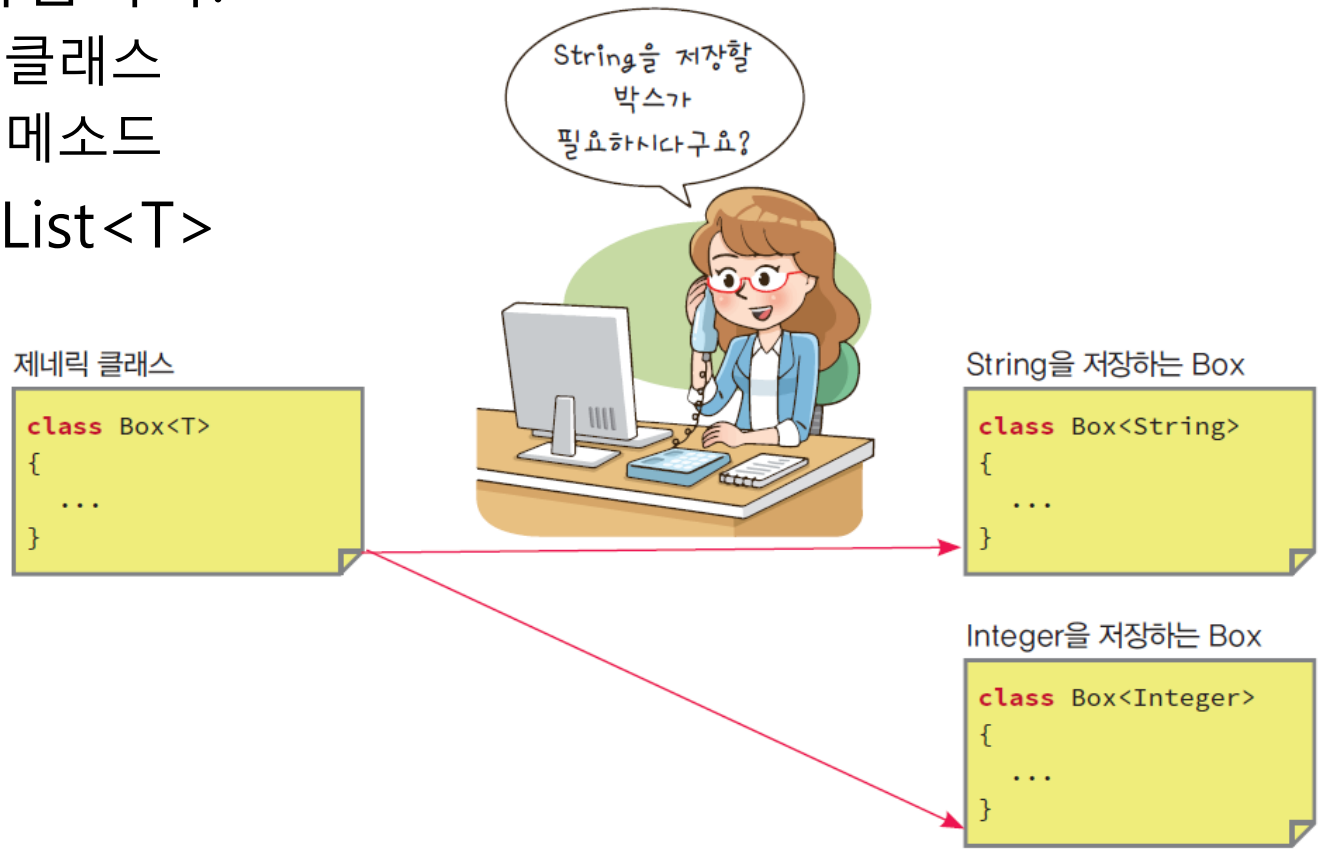
제15장 제네릭과 컬렉션

학습 내용

학습목차

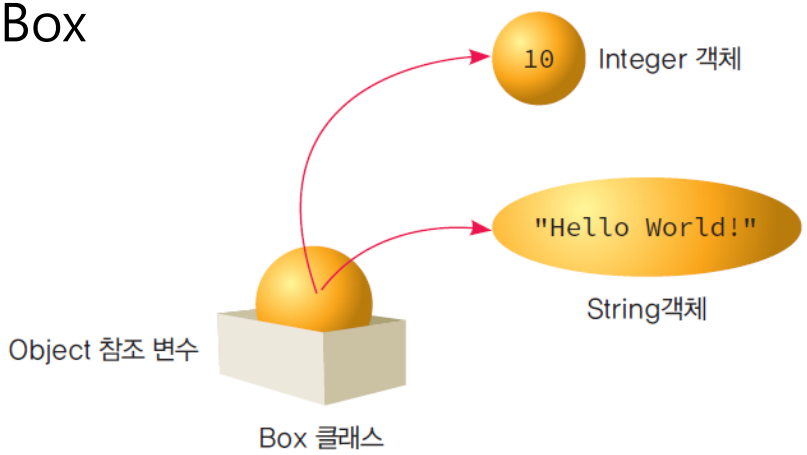
제네릭이란?

- 제네릭 프로그래밍(generic programming)이란 다양한 종류의 데이터를 처리할 수 있는 클래스와 메소드를 작성하는 기법이다.
  - 제네릭 클래스
  - 제네릭 메소드
- 예) ArrayList<T>



예제

- 여러 타입의 객체 하나를 저장할 박스를 만들어보자.
  1. 객체 타입마다 별도의 박스 클래스를 정의하는 방법
     - String을 저장할 StringBox, Integer를 저장할 IntegerBox
  2. 모든 타입의 객체를 저장하도록 Object 타입을 사용하는 방법
     - Object를 저장할 Box



  3. 제네릭을 이용하는 방법
     - 하나의 제네릭 박스 클래스를 정의하면 여러 타입의 박스 클래스를 사용할 수 있다.
     - Box<T>를 정의하고, Box<String>, Box<Integer>를 사용

방법 1 : 타입마다 별도의 박스 정의

- String 객체를 저장할 박스

```
public class StringBox {
        private String data;
        public void set(String data) {  this.data = data;      }
        public String get()              {  return data;          }
}
```

- Integer 객체를 저장할 박스

```
public class IntegerBox {
        private Integer data;
        public void set(Integer data) {  this.data = data;     }
        public Integer get()              {  return data;          }
}
```

방법 2 : Object 타입에 대해 박스 정의

- Object 객체를 저장할 박스

```
public class Box {
        private Object data;
        public void set(Object data) {  this.data = data;      }
        public Object get()              {  return data;          }
}
```

방법 2 : 문제점

- Object 객체를 저장할 박스에 String 객체와 Integer 객체를 저장했다가 꺼내보자.

```
public class Box {
        private Object data;
        public void set(Object data) {  this.data = data;      }
        public Object get()              {  return data;          }
}
```

```
Box bs = new Box();                    // Object 객체를 저장할 박스 생성
bs.set("Hello World!");                // String 객체를 저장
String s = (String) bs.get();          // Object 타입을 String으로 형변환

Box bi = new Box();                    // Object 객체를 저장할 박스 생성
bi.set(new Integer(10));               // Integer 객체를 저장
Integer i = (Integer) bi.get( );       // Object 타입을 Integer로 형변환
```

방법 3: 제네릭을 이용한 방법

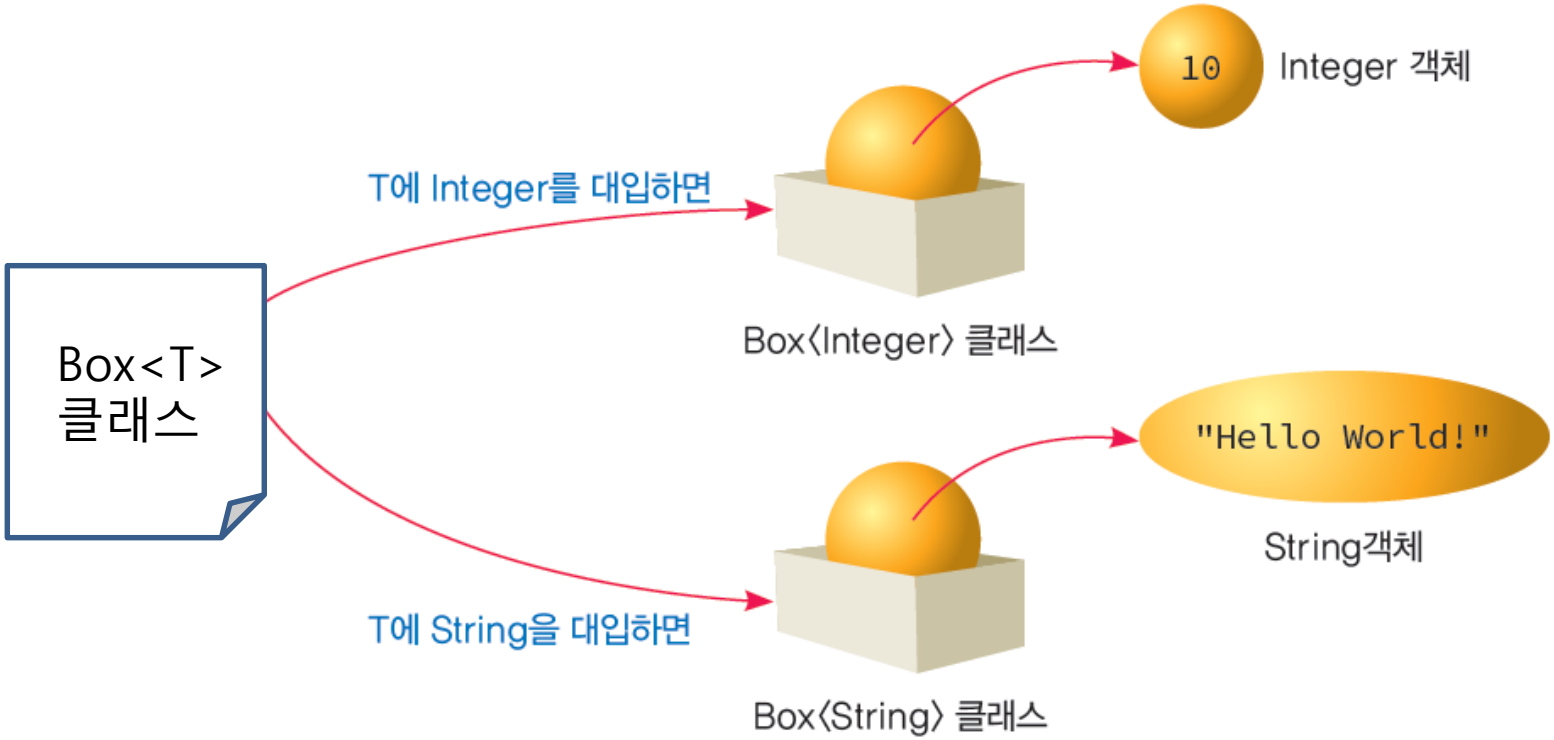
- 제네릭 클래스(generic class)
  - 제네릭 클래스는 클래스를 정의할 때 타입을 변수로 표시한다. 이것을 타입 매개변수(type parameter)라고 한다.

T는 type parameter

```
public class Box<T> {      // 제네릭 클래스 정의
        private T data;
        public void set(T data) {  this.data = data;  }
        public T get()             {  return data;        }
}
```

제네릭 클래스의 타입 매개변수

- Box<T> 클래스의 타입 매개변수 T는 Box<T> 객체 생성 시 프로그래머에 의해 결정된다.



제네릭을 이용한 방법

- 제네릭 클래스를 이용하여 여러 타입의 박스를 생성해보자.
  - Box<String> 타입의 객체, Box<Integer> 타입의 객체

```
public class Box<T> {     // 제네릭 클래스 정의
        private T data;
        public void set(T data) {  this.data = data;  }
        public T get()             {  return data;        }
}
```

type parameter T에 String를 대입

```
Box<String> bs = new Box<String>(); // String 객체를 저장할 박스 생성
bs.set("Hello World!");
String s = bs.get();

Box<Integer> bi = new Box<Integer>(); // Integer 객체를 저장할 박스 생성
bi.set(new Integer(10));
Integer i = bi.get( );
```

type parameter T에 Integer를 대입

LAB: SimplePair 클래스 작성하기

```
public class SimplePairTest {
    public static void main(String[] args) {
        SimplePair<String> pair = new SimplePair<String>("apple", "tomato");
        System.out.println(pair.getFirst());
        System.out.println(pair.getSecond());
    }
}
```

```
apple
tomato
```

SimplePair p = new SimplePair("apple", "tomato");
처럼 타입 매개변수 없이 제네릭 클래스 사용하는 것을 raw type이라고 한다. 이런 방식은 가능한 피할 것

타입 매개변수에 기초 자료형을 사용할 수 없다.

```
SimplePair<int> a;          // 오류!
SimplePair<Integer> b;      // OK!
```

SOLUTION

```
public class SimplePair<T> {
        private T data1;
        private T data2;
        public SimplePair(T data1, T data2) {
                this.data1 = data1;
                this.data2 = data2;
        }
        public T getFirst() {          return data1;      }
        public T getSecond() {      return data2;      }
        public void setFirst(T data1) {       this.data1 = data1;          }
        public void setSecond(T data2) {   this.data2 = data2;          }
}
```

다중 타입 매개변수(Multiple Type Parameters)

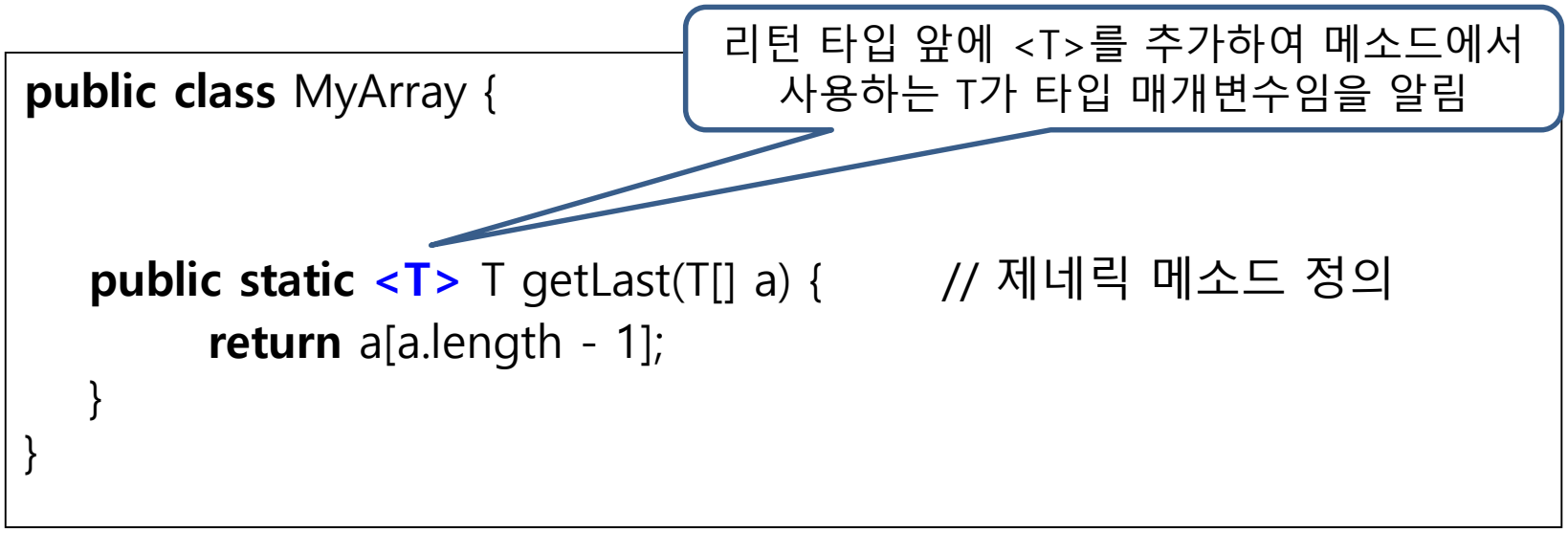
```
public class OrderedPair<K, V> {
    private K key;
    private V value;
    public OrderedPair(K key, V value) {
            this.key = key;
            this.value = value;
    }
    public K getKey()           { return key;       }
    public V getValue()         { return value;     }
}
```

```
public class OrderedPairTest {
    public static void main(String[] args) {
        OrderedPair<String, Integer> p1 = new OrderedPair<String, Integer>("kim", 23);
        OrderedPair<String, String> p2 = new OrderedPair<String, String>("Korea", "Seoul");
        System.out.println(p1.getKey() + " " + p1.getValue());
        System.out.println(p2.getKey() + " " + p2.getValue());
    }
}
```

```
kim 23
Korea Seoul
```

제네릭 메소드

- 일반 클래스의 메소드에서도 타입 매개변수를 사용하여 제네릭 메소드(generic method)를 정의할 수 있다.
- 이 경우에는 타입 매개변수의 범위(scope)가 메소드 내부로 제한된다.




```
public class MyArray {


    public static <T> T getLast(T[] a) {        // 제네릭 메소드 정의
        return a[a.length - 1];
    }
}
```

제네릭 메소드 호출

```
public class MyArray {
    public static <T> T getLast(T[] a) {        // 제네릭 메소드 정의
        return a[a.length - 1];
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        String[] language = { "C++", "C#", "JAVA" };
        String last = MyArray.getLast(language);
        System.out.println(last);
    }
}
```

실제 자료형을 넣어주어도 됨
MyArray.<String>getLast(language)

JAVA

LAB

- 배열의 i번째 요소와 j번째 요소를 교환하는 swap() 메소드를 제네릭 메소드로 작성해보자.

```
public class MyArray {
        public static <T> void swap(T[] a, int i, int j) { // 제네릭 메소드
                T tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
        }
}
```

```
public class Main {
        public static void main(String[] args) {
                String[] language = { "C++", "C#", "JAVA" };
                MyArray.swap(language, 1, 2);
                for(String value : language)
                        System.out.println(value);
        }
}
```

LAB

- 정수 배열, 실수 배열, 문자 배열을 모두 출력할 수 있는 제네릭 메소드 printArray()를 작성해보자.

```
public class GenericMethodTest {
        public static void main(String[ ] args) {
                Integer[] iArray = { 10, 20, 30, 40, 50 };
                Double[] dArray = { 1.1, 1.2, 1.3, 1.4, 1.5 };
                Character[] cArray = { 'K', 'O', 'R', 'E', 'A' };
                printArray(iArray);
                printArray(dArray);
                printArray(cArray);
        }
        public static <T> void printArray(T[] array) {
                for (T element : array) {
                        System.out.printf("%s ", element);
                }
                System.out.println();
        }
}
```

한정된 타입 매개변수

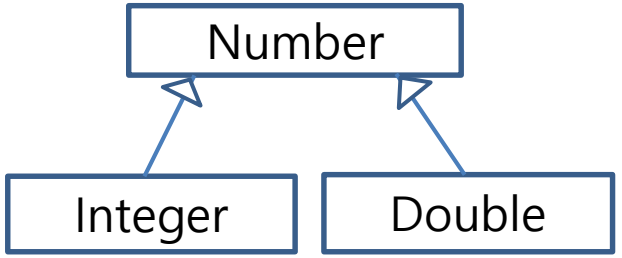
- 타입 매개변수에 대입할 타입의 종류를 제한할 수 있다.

타입 매개변수 T를 Comparable의 하위 클래스인 T만 허용

```
public class MyArray {
    public static <T extends Comparable> T getMax(T[] a) {
        if (a == null || a.length == 0)
            return null;
        T largest = a[0];
        for (int i = 1; i < a.length; i++)
            if (largest.compareTo(a[i]) < 0)
                largest = a[i];
        return largest;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        String[] list = { "xyz", "abc", "def" };
        String max = MyArray.getMax(list);
        System.out.println(max);
    }
}
```

제네릭과 상속

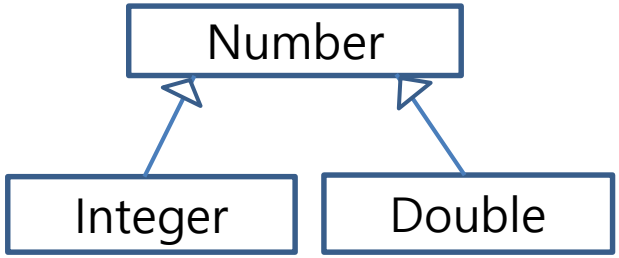


- 자바 라이브러리에서 Number 클래스를 상속받아서 Integer와 Double 클래스를 정의하고 있다.
  - 타입 매개변수를 Number로 하여 박스를 생성했으면 Integer 객체, Double 객체도 저장할 수 있다.

```
public class Box<T> {
        private T data;
        public void set(T data) {  this.data = data;  }
        public T get()              {  return data;       }
}
```

```
Box<Number> box = new Box<Number>();
box.set(new Integer(10));          // Integer 객체 저장
box.set(new Double(10.1));         // Double 객체 저장
```

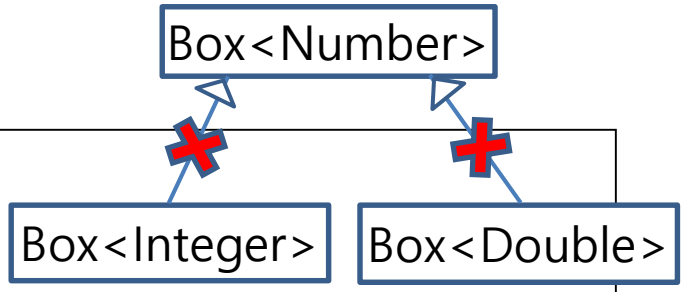
제네릭과 상속



- 다음과 같은 Box<Number> 타입의 매개변수로 Box<Integer> 객체를 받을 수 있을까?

```
void boxPrint(Box<Number> box) {
        System.out.println(box.get());
}
```

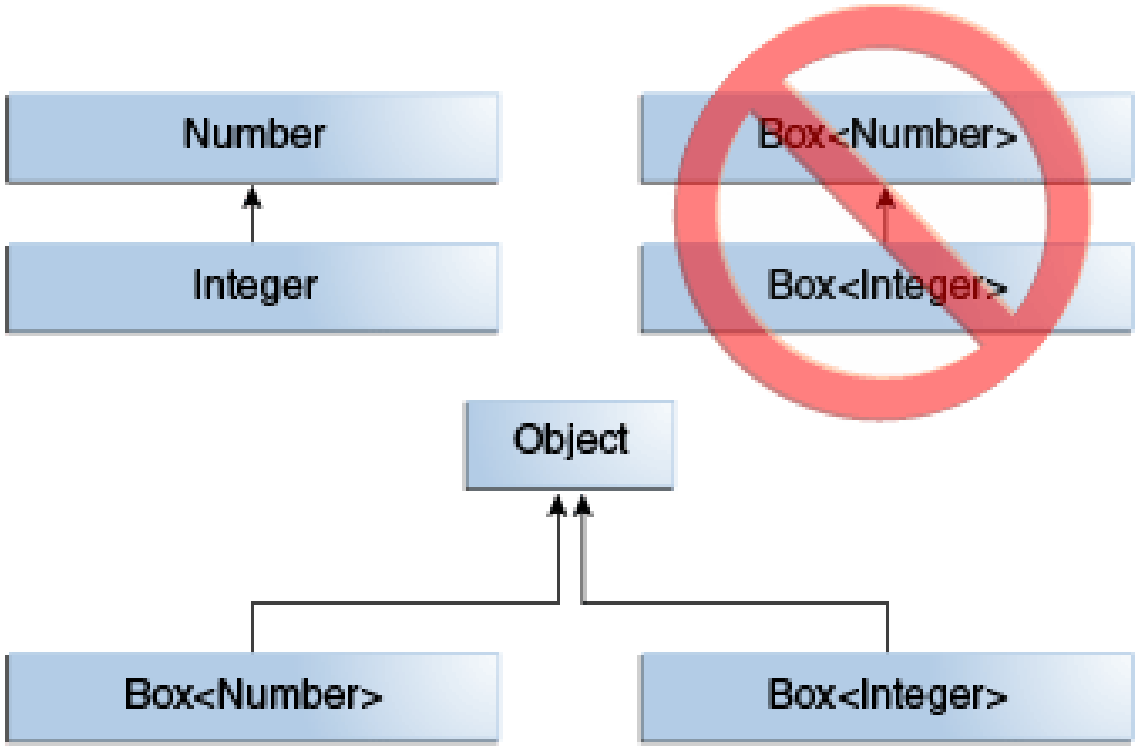
- 답: 받을 수 없다.




```
boxPrint(new Box<Number>()); // OK!

boxPrint(new Box<Integer>());  // 에러!
boxPrint(new Box<Double>());  // 에러!
```

제네릭과 상속

- Integer가 Number의 자식이지만,
- Box<Integer>는 Box<Number>의 자식이 아니다.

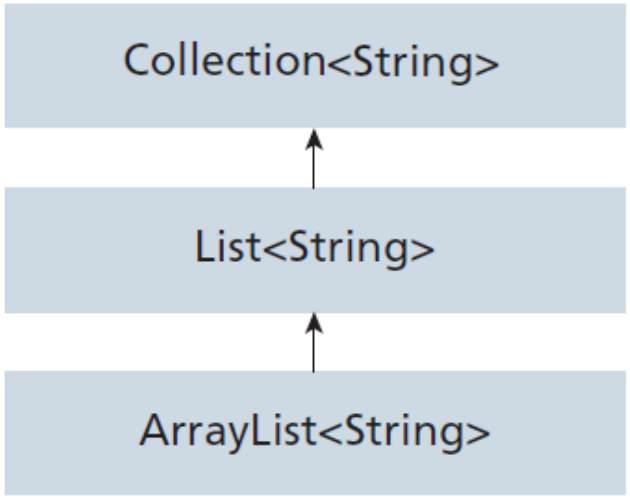


제네릭 클래스의 상속

- 제네릭 클래스들 간의 관계가 다음과 같으면

```
ArrayList<E> implements List<E> {   ...   }
List<E> extends Collection<E> {...}
```

- 다음과 같은 클래스 계층구조를 얻을 수 있다.



와일드 카드

- 제네릭을 사용하는 코드에서 물음표(?)는 와일드 카드(wildcard)라고 불린다.
- 와일드 카드는 어떤 타입이든 나타낼 수 있다.
- 와일드 카드는 다양한 곳에서 사용할 수 있다.
  - 매개변수 타입을 나타낼 때
  - 필드 타입을 나타낼 때
  - 지역변수 타입을 나타낼 때
- 와일드 카드에 상한/하한을 둘 수 있다.
  - 상한이 있는 와일드 카드
  - 제한 없는 와일드 카드
  - 하한이 있는 와일드 카드

상한이 있는 와일드 카드

- 상한이 있는 와일드 카드(Upper Bounded Wildcard)
  - 어떤 클래스 A의 자손 클래스들은 <? extends A>로 표시한다.

```
import java.util.List;
public class MyList {
    public static double sumOfList(List<? extends Number> list) {
        double s = 0.0;
        for (Number n : list)
            s += n.doubleValue();
        return s;
    }
}
```

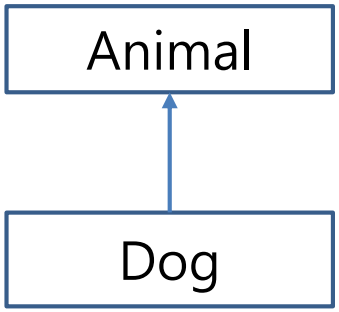
<? extends Number>는 Number의 자손이면 무엇이든 가능하다는 뜻

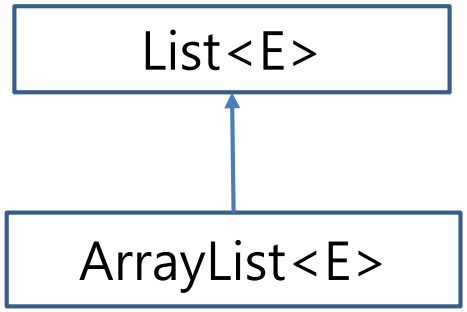
따라서 list의 타입으로 List<Number>, List<Double>, List<Integer> 등 가능

```
List<Integer> li = Arrays.asList(1, 2, 3);
System.out.println(MyList.sumOfList(li));

List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
System.out.println(MyList.sumOfList(ld));
```

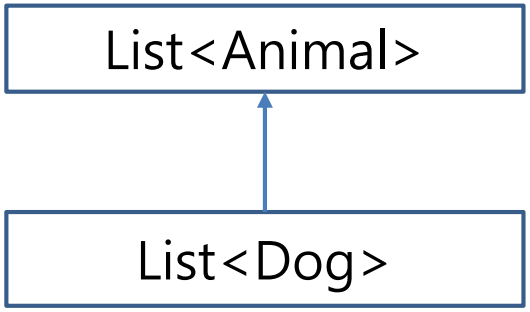
```
6.0
7.0
```

계층 관계가 다음과 같을 때, (1) ~(4) 중 잘못된 것은?

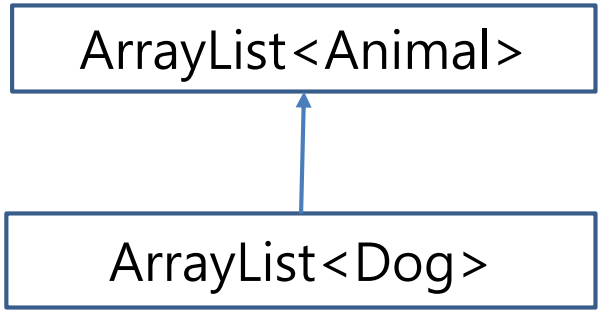




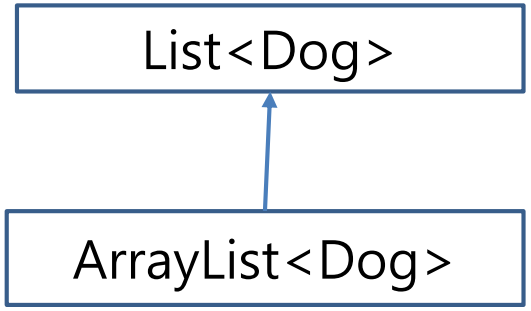
(1)



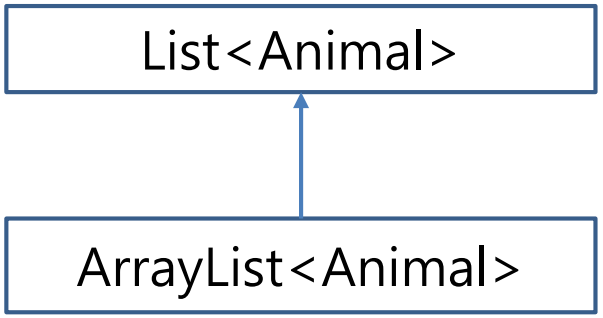
(2)



(3)



(4)



1 ~ 4 중 컴파일 에러 발생하는 것은?

```
import java.util.*;
class Animal { }
class Dog extends Animal { }
public class Main {
    static void dogMethod(List<Dog> list) { }
    static void animalMethod(List<Animal> list) { }
    static void myMethod(List<? extends Animal> list) { }
    public static void main(String[] args) {
        List<Dog> dogList = new ArrayList<Dog>();
        dogMethod(dogList);                    // 1
        animalMethod(dogList);                 // 2
        myMethod(dogList);                     // 3
        List<Animal> animalList = new ArrayList<Animal>();
        myMethod(animalList);                  // 4
    }
}
```

제한 없는 와일드 카드

- 제한 없는 와일드 카드(Unbounded Wildcard)
  - 단순히 <?> 로 표시하며, 모든 타입에 매치된다.
  - 예를 들면 List<?>

```
import java.util.List;
public class MyList {
    public static void printList(List<?> list) {
        for (Object elem : list)
            System.out.print(elem + " ");
        System.out.println();
    }
}
```

```
List<Integer> li = Arrays.asList(1, 2, 3);
MyList.printList(li);

List<String> ls = Arrays.asList("one", "two", "three");
MyList.printList(ls);
```

```
1 2 3
one two three
```

하한이 있는 와일드 카드

- 하한이 있는 와일드 카드(Lower Bounded Wildcard)
  - 어떤 클래스 A의 조상 클래스들은 <? super A>로 표시

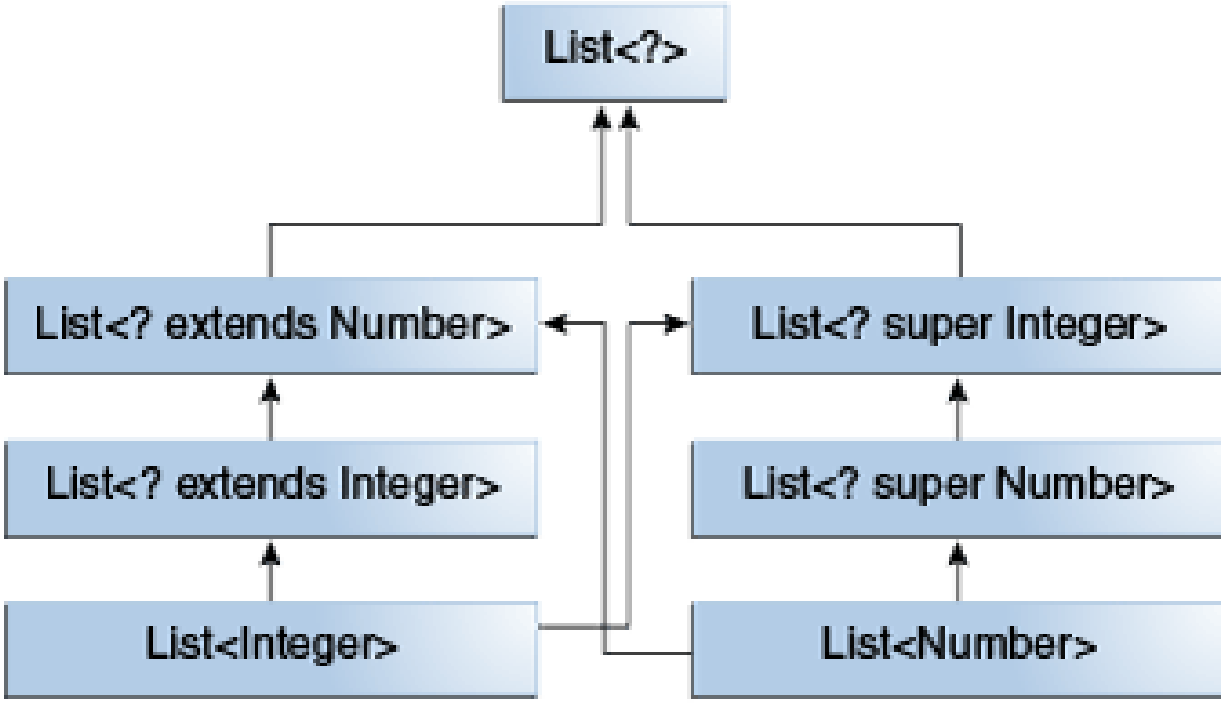
<? super Integer>는 Integer의 조상이면 무엇이든 가능하다는 뜻

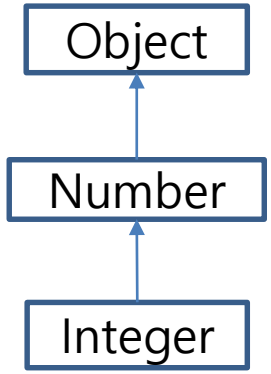
따라서 list의 타입으로 List<Object>, List<Number>, List<Integer> 가 가능

```
import java.util.List;

public class MyList {
    public static void addNumbers(List<? super Integer> list) {
        for (int i = 1; i <= 10; i++) {
            list.add(i);
        }
}
```

제네릭 클래스 계층구조

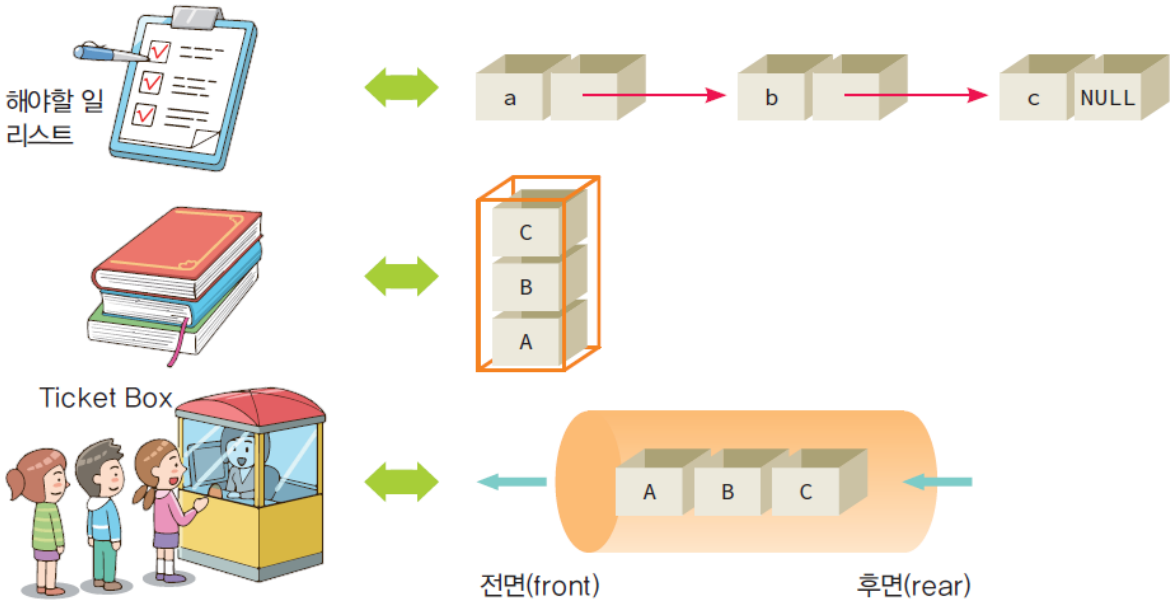
- 와일드 카드를 사용한 제네릭 클래스에서 나타날 수 있는 상속 관계





컬렉션

- 컬렉션(collection)은 자바에서 자료구조(data structure)를 구현한 클래스들을 칭하는 용어이다.
- 자료구조로는 리스트(list), 스택(stack), 큐(queue), 집합(set), 해쉬 테이블(hash table) 등이 있다.



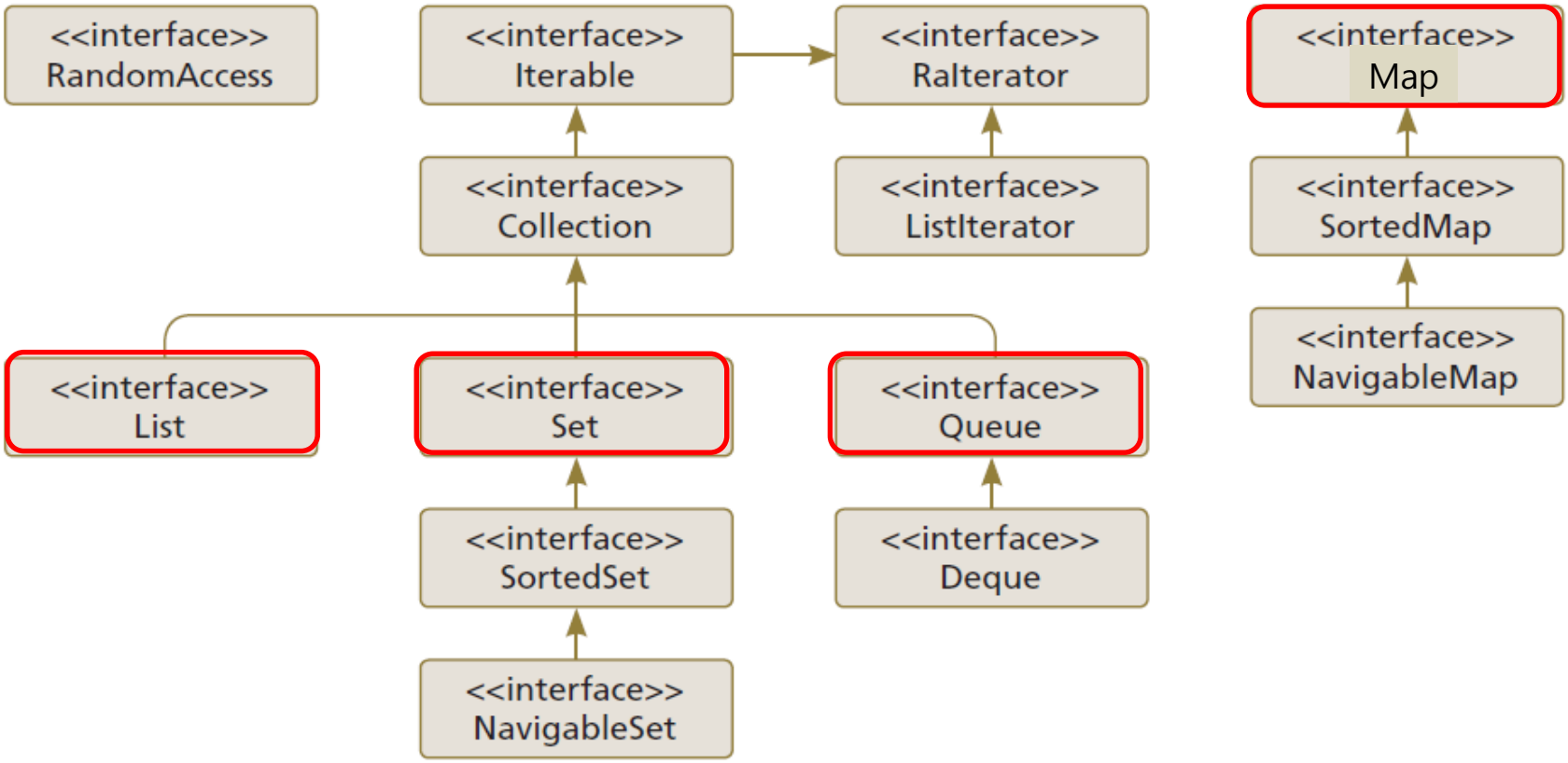
컬렉션 인터페이스와 컬렉션 클래스

- 자바는 컬렉션 인터페이스와 컬렉션 클래스로 나누어 제공한다.
  - 자바는 컬렉션 인터페이스를 구현한 클래스도 함께 제공하므로
  - 간단하게 이 클래스를 사용해도 되고,
  - 각자 필요에 따라 인터페이스를 구현한 자신의 클래스를 정의해도 됨

표 14.1 • 컬렉션 인터페이스

인터페이스	설명
Collection	모든 자료 구조의 부모 인터페이스로서 객체의 모임을 나타낸다.
Set	집합(중복된 원소를 가지지 않는)을 나타내는 자료 구조
List	순서가 있는 자료 구조로 중복된 원소를 가질 수 있다.
Map	키와 값들이 연관되어 있는 사전과 같은 자료 구조
Queue	극장에서의 대기줄과 같이 들어온 순서대로 나가는 자료구조

컬렉션 인터페이스의 계층구조



Collection 인터페이스

표 14.1 • Collection<E> 인터페이스의 메소드

메소드	설명

`boolean isEmpty()`
`boolean contains(Object obj)`
`boolean containsAll(Collection<?> c)`
	공백 상태이면 true 반환
	obj를 포함하고 있으면 true 반환

`boolean add(E element)` `boolean addAll(Collection<? extends E> from)`	원소를 추가한다.

`boolean remove(Object obj)`
`boolean removeAll(Collection<?> c)`
`boolean retainAll(Collection<?> c)`
`void clear()`
	원소를 삭제한다.

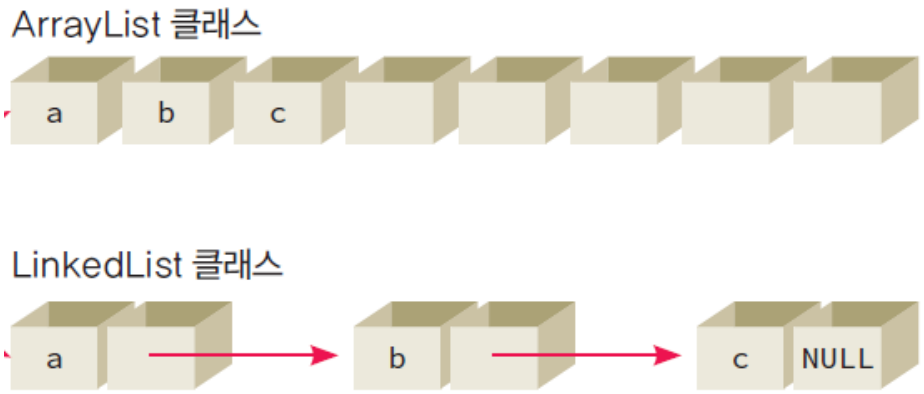
`Iterator<E> iterator()`
`Stream<E> stream()`
`Stream<E> parallelStream()`
	원소 방문

`int size()`	원소의 개수 반환
`Object[] toArray()` `<T> T[] toArray(T[] a)`	컬렉션을 배열로 변환

List 인터페이스

public interface **List<E>** extends Collection<E>

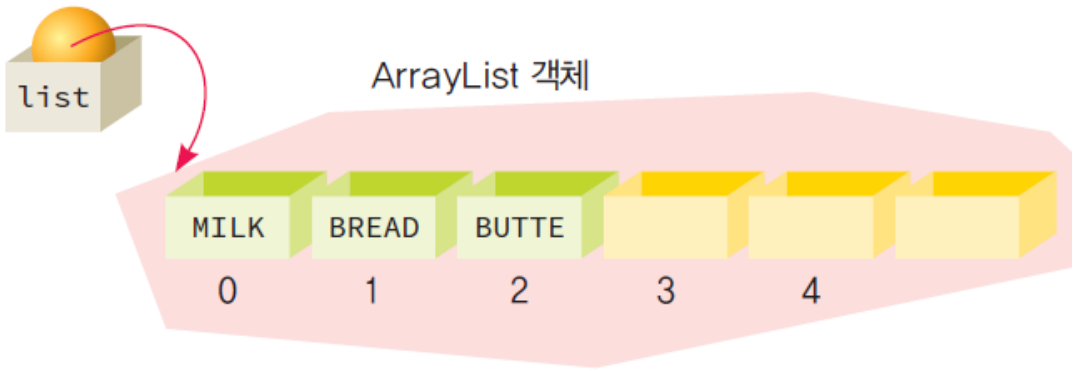
- **리스트(List)**는 순서를 가지는 요소들의 모임으로 중복된 요소를 가질 수 있다.
- List 인터페이스를 구현한 클래스
  - ArrayList
  - LinkedList
  - Vector : 동기화됨(thread-safe implementation)



ArrayList 클래스

public class **ArrayList<E>** ... implements List<E> ...

- ArrayList는 배열(array)의 향상된 버전 또는 가변 크기의 배열이라고 생각하면 된다.
- ArrayList의 생성 예:
  - ArrayList<String> list = **new** ArrayList<String>();
- 원소 추가
  - list.add("MILK");
  - list.add("BREAD");
  - list.add("BUTTER");



ArrayList 클래스 사용 예

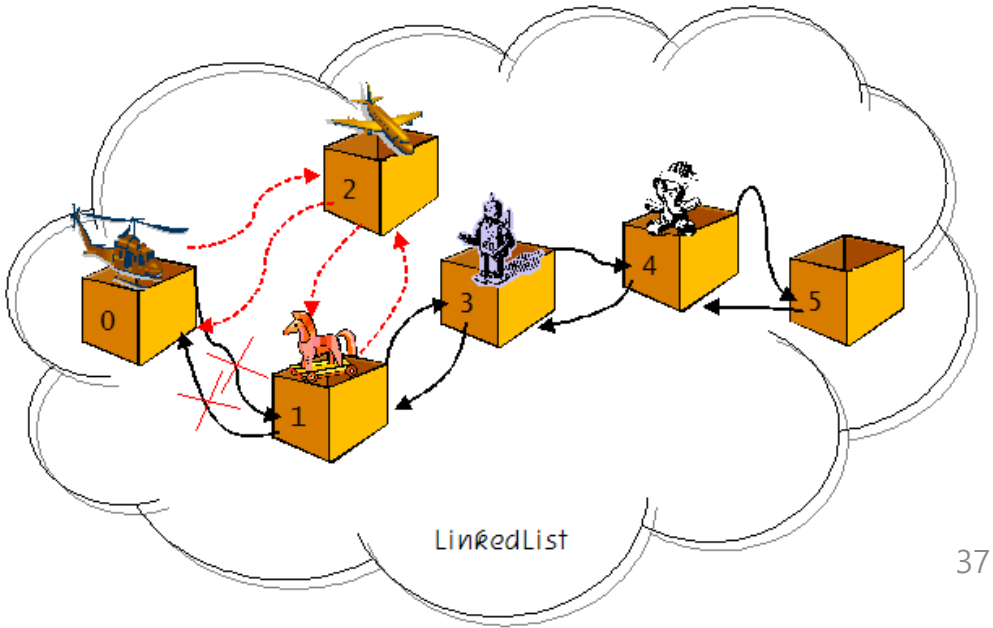
```
List<String> list = new ArrayList<String>();
list.add("MILK");
list.add("BREAD");
list.add("BUTTER");
list.add("APPLE");
list.add(1, "MILK");          // 인덱스 1에 "MILK" 삽입
list.set(2, "GRAPE");         // 인덱스 2의 원소를 "GRAPE"로 대체
list.remove(3);               // 인덱스 3의 원소를 삭제
list.remove("APPLE");         // 원소 "APPLE" 삭제

for(String s : list)
    System.out.println(s);

System.out.println(list.indexOf("MILK"));       //  첫번째 MILK의 인덱스 0
System.out.println(list.lastIndexOf ("MILK")); // 마지막 MILK의 인덱스 1
System.out.println(list.indexOf ("APPLE"));    // 검색 실패시 -1
```

```
MILK
MILK
GRAPE
0
1
-1
```

LinkedList 클래스

public class **LinkedList<E>** ... implements List<E> ...

- 각 원소를 링크로 연결한 연결 리스트로 구현한 리스트임
- 빈번하게 삽입과 삭제가 일어나는 경우에 사용
- LinkedList의 생성 예:
  - LinkedList<String> list = **new** LinkedList<String>();
- 원소 추가
  - list.add("MILK");
  - list.add("BREAD");
  - list.add("BUTTER");



배열을 리스트로 변환하기

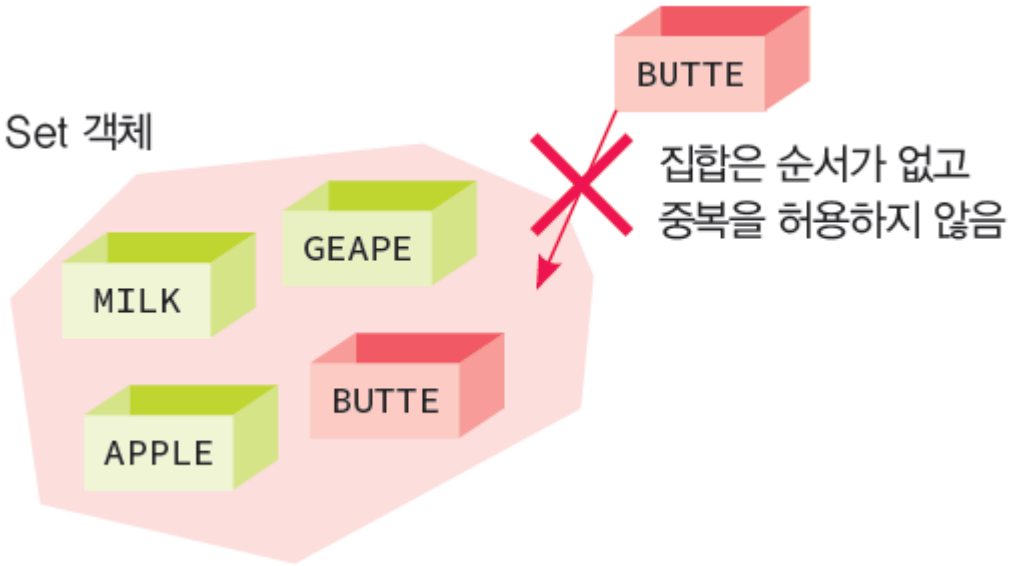
- List를 매개변수로 받는 메소드에 배열을 매개변수로 넘길 수 있나?
  - Arrays.asList() 메소드를 사용하여 배열을 리스트 형태로 변환한 다음 넘기면 된다.
  - 리스트 원소를 변경하면 배열도 변경되고, 그 역도 성립
  - 리스트의 크기는 변경 불가능
- 예:

```
String[] array = new String[5];
List<String> list = Arrays.asList(array);
list.set(0, "hello");
array[1] = "bye";
System.out.println(list);   // 출력: [hello, bye, null, null, null]
```

Set 인터페이스

public interface **Set<E>** extends Collection<E> ...

- **집합(Set)**은 원소의 중복을 허용하지 않는다.



Set 인터페이스를 구현한 클래스

- HashSet
  - HashSet은 해쉬 테이블에 원소를 저장하기 때문에 성능 면에서 가장 우수하다.
  - 하지만 원소들의 순서가 일정하지 않은 단점이 있다.
- TreeSet
  - 레드-블랙 트리(red-black tree)에 원소를 저장한다.
  - 원소 값에 따라서 순서가 결정되지만 HashSet보다는 느리다.
- LinkedHashSet
  - 해쉬 테이블과 연결 리스트를 결합한 것으로 원소들의 순서는 삽입 순서와 같다.

HashSet 클래스

```
import java.util.*;

public class SetTest {
        public static void main(String[ ] args) {
                Set<String> set = new HashSet<String>();
                set.add("Milk");
                set.add("Bread");
                set.add("Butter");
                set.add("Cheese");
                set.add("Ham");
                set.add("Ham");
                System.out.println(set);
        }
}
```

```
[Ham, Butter, Cheese, Milk, Bread]
```

Set의 대량 연산 메소드

- s1.containsAll(s2)
  - s2가 s1의 부분 집합이면 참이다.
- s1.addAll(s2)
  - s1을 s1과 s2의 합집합(union)으로 만든다.
- s1.retainAll(s2)
  - s1을 s1과 s2의 교집합(intersection)으로 만든다.
- s1.removeAll(s2)
  - s1을 s1과 s2의 차집합(difference)으로 만든다.

Queue 인터페이스

public interface **Queue<E>** extends Collection<E> ...

- 큐(Queue)는 후단(tail)에서 원소를 추가하고 전단(head)에서 원소를 삭제하는 자료구조이다.

그림 15-11 • 큐

Map 인터페이스

public interface **Map<K,V>**

- 맵(Map)은 사전과 같은 자료구조이다.
  - <키, 값> 쌍을 저장한 후, 키를 제시하면 값을 찾을 수 있다.
  - 중복된 키를 가질 수 없다.

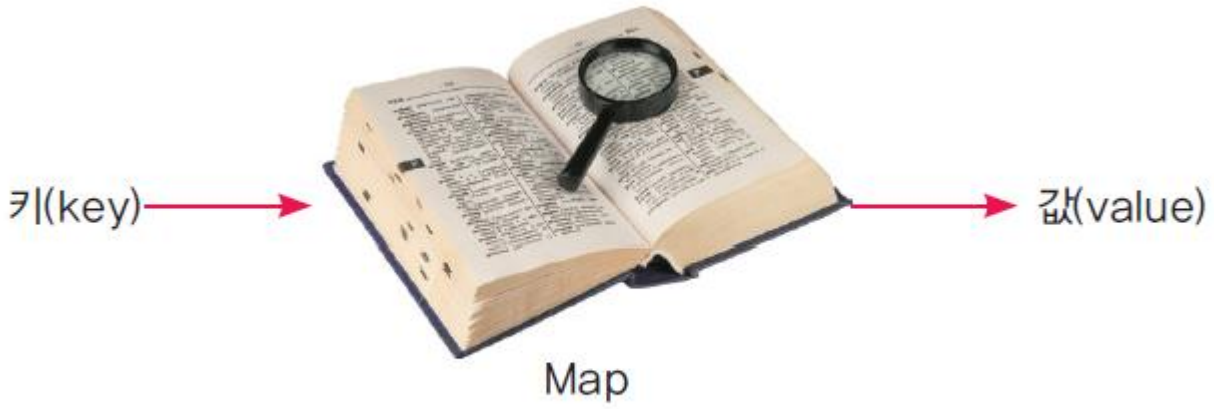


그림 15-17 • Map의 개념

```
import java.util.*;

public class Main {
	public static void main(String[] args) {
		Map<String, Integer> map = new HashMap<String, Integer>();
		map.put("소프", new Integer(41));
		map.put("컴공", new Integer(42));
		map.put("정통", new Integer(42));

		System.out.println(map);                    // 모든 항목을 출력
		map.remove("정통");                          // 키가 정통인 항목을 삭제
		map.put("컴공", new Integer(55));            // 키가 컴공인 값을 대치
		System.out.println(map.get("소프"));         // 키가 소프인 값을 찾음

		for (Map.Entry<String, Integer> s : map.entrySet()) {  // 모든 항목을 방문
			String key = s.getKey();
			Integer value = s.getValue();
			System.out.println("key=" + key + ", value=" + value);
		}
	}
}
```

```
{소프=41, 컴공=42, 정통=42}
41
key=소프, value=41
key=컴공, value=55
```

Collections 클래스

- Collections 클래스는 여러 유용한 알고리즘을 구현한 메소드들을 제공한다.
  - 제네릭 기술 사용
  - 정적 메소드
  - 컬렉션을 매개변수로 받음
- Collections 클래스의 주요 알고리즘
  - 정렬(sorting)
  - 섞기(shuffling)
  - 탐색(searching)

정렬

- 정렬이란 리스트의 데이터를 어떤 기준에 따라 순서대로 나열하는 것이다.

```
import java.util.*;

public class Main {
	public static void main(String[] args) {
		String[] sample = { "i", "walk", "the", "line" };
		List<String> list = Arrays.asList(sample);  // 배열을 리스트로 변경
		Collections.sort(list);
		System.out.println(list);
	}
}
```

list의 원소를 정렬

```
[i, line, the, walk]
```

```
import java.util.*;
public class Main {
     static class Student {
          String name;
          public Student(String name) {
               this.name = name;
          }
          public String toString() {
               return name;
          }
     }
     public static void main(String[] args) {
          List<Student> list = new ArrayList<Student>();
          list.add(new Student("park"));
          list.add(new Student("kim"));
          Collections.sort(list);     // 1
          System.out.println(list);
     }
}
```

라인 1의 컴파일 에러 발생 이유는?

```
import java.util.*;
public class Main {
    static class Student implements Comparable<Student> {
        String name;
        public Student(String name) {  this.name = name;       }
        public String toString() {     return name;          }
        @Override
        public int compareTo(Student other) {
            return name.compareTo(other.name);
        }
    }
    public static void main(String[] args) {
        List<Student> list = new ArrayList<Student>();
        list.add(new Student("park"));
        list.add(new Student("kim"));
        Collections.sort(list);
        System.out.println(list);
    }
}
```

[kim, park]

```
import java.util.*;
public class Main {
      static class Student {
            String name;
            int age;
            public Student(String name, int age) {
                  this.name = name;
                  this.age = age; }
            public String toString() {
                  return name + age;
            }
      }
      public static void main(String[] args) {
            List<Student> list = new ArrayList<Student>();
            list.add(new Student("park", 22));
            list.add(new Student("lee", 20));
            Collections.sort(list, (s1, s2) -> s1.age – s2.age );
            System.out.println(list);
      }
}
```

[lee20, park22]

탐색

- 탐색이란 리스트 안에서 원하는 원소를 찾는 것이다.

```
import java.util.*;

public class Main {
	public static void main(String[] args) {
		int key = 50;
		List<Integer> list = new ArrayList<Integer>();
		for (int i = 0; i < 10; i++)  // 리스트에 0, 10, 20, ..., 90 삽입
			list.add(i * 10);
		int index = Collections.binarySearch(list, key);
		System.out.println(index);
	}
}
```

list에서 key를 이진탐색하여 위치(인덱스)를 리턴

```
5
```

LAB: Map을 이용한 영어사전

```
import java.util.*;
public class EnglishDic {
	public static void main(String[] args) {
		Map<String, String> st = new HashMap<String, String>();
		st.put("map", "지도");
		st.put("java", "자바");
		st.put("school", "학교");

		Scanner sc = new Scanner(System.in);
		do {
			System.out.print("영어 단어 입력(quit 입력시 종료): ");
			String key = sc.next();
			if (key.equals("quit"))
				break;
			System.out.println(st.get(key));
		} while(true);
	}
}
```

```
영어 단어 입력(quit 입력시 종료): map
지도
영어 단어 입력(quit 입력시 종료): school
학교
영어 단어 입력(quit 입력시 종료): quit
```