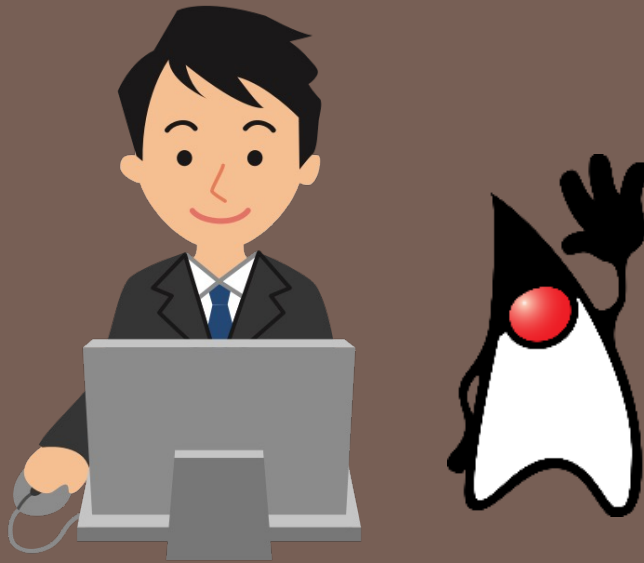


# 파워자바(개정3판)

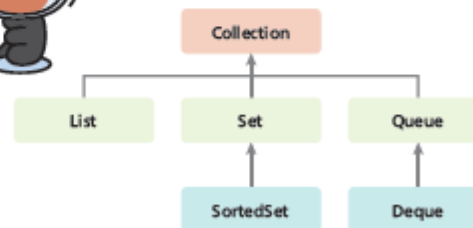


## 13장 제네릭과 컬렉션



# 13장의 목표

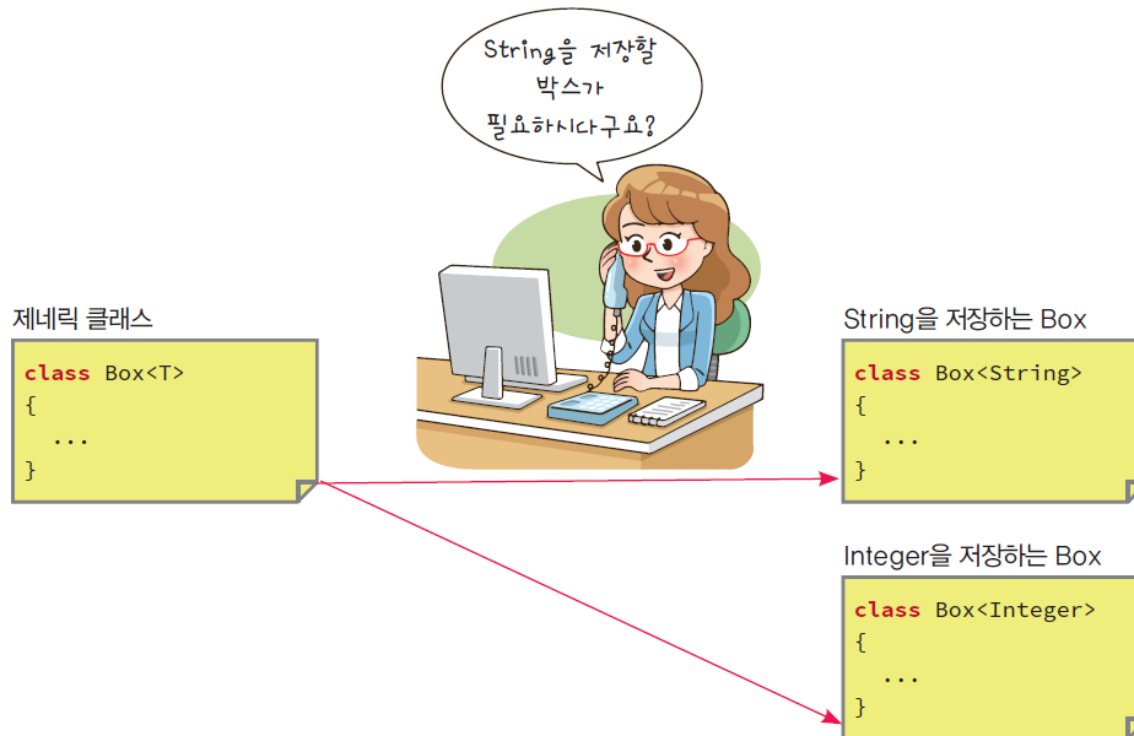
1. 어떤 자료형에서도 동작하는 메소드를 제네릭으로 작성할 수 있나요?
2. ArrayList를 사용하여 데이터를 저장하고 처리할 수 있나요?
3. Map을 사용하여서 키와 값을 묶어서 저장할 수 있나요?
4. Collections 클래스가 제공하는 sort()와 같은 메소드를 사용할 수 있나요?





# 제네릭이란?

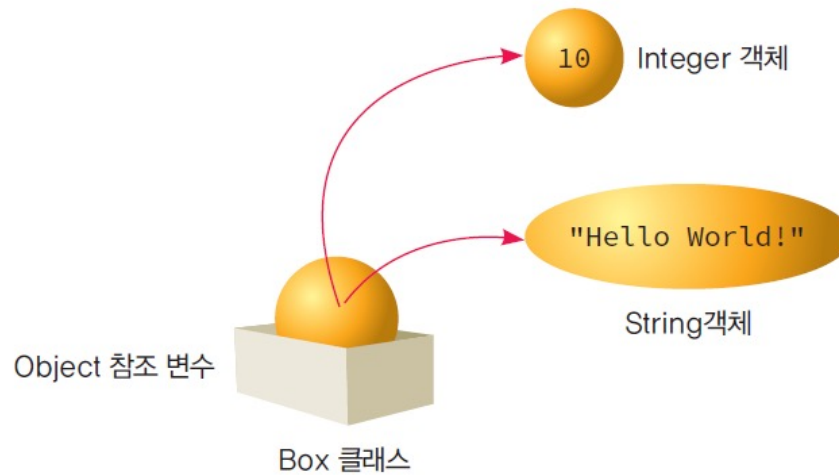
- 제네릭 프로그래밍(**generic programming**)이란 다양한 종류의 데이터를 처리할 수 있는 클래스와 메소드를 작성하는 기법이다





# 기존의 방법

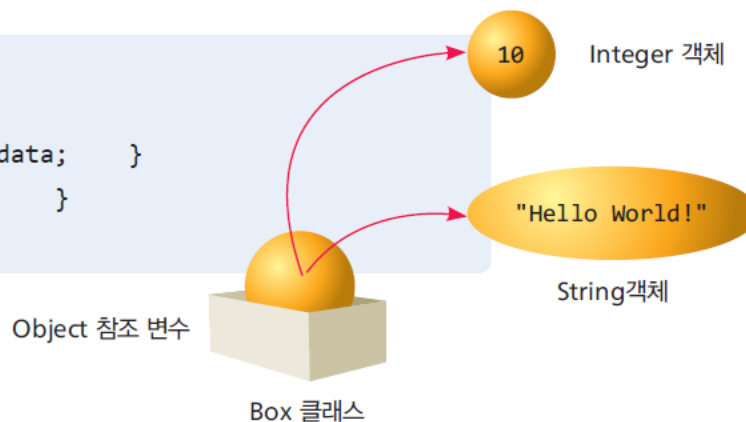
- 일반적인 객체를 처리하려면 **Object** 참조 변수를 사용
- **Object** 참조 변수는 어떤 객체이든지 참조할 수 있다.
- 예제로 하나의 데이터를 저장하는 **Box** 클래스를 살펴보자.





# 기존의 방법

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



```
Box b = new Box();
```

```
b.set("Hello World!");           // ① 문자열 객체를 저장  
String s = (String)b.get();      // ② Object 타입을 String 타입으로 형변환
```

```
b.set(new Integer(10));          // ③ 정수 객체를 저장  
Integer i = (Integer)b.get( );   // ④ Object 타입을 Integer 타입으로 형변환
```

```
b.set("Hello World!");  
Integer i = (Integer)b.get( );    // 오류! 문자열을 정수 객체로 형변환
```

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be  
cast to java.lang.Integer at GenericTest.main(GenericTest.java:10)
```



# 제네릭을 이용한 방법

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

T는 타입을 의미한다.

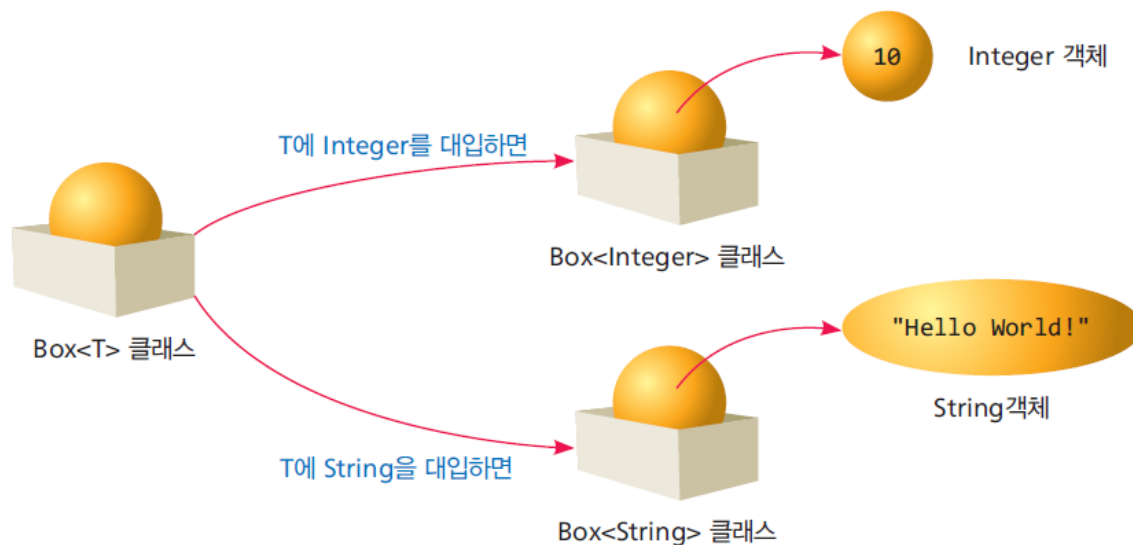


그림 13.2 Box 클래스에 저장하는 데이터의 타입은 객체 생성 시에 결정된다.



# 제네릭을 이용한 방법

```
Box<String> b = new Box<String>();  
b.set("Hello World!");    // 문자열 저장  
String s = Box.get();
```

```
Box<String> stringBox = new Box<String>();  
stringBox.set(new Integer(10));    // 정수 타입을 저장하려고 하면 컴파일 오류!
```

The method set(String) in the type Box<String> is not applicable for the arguments (Integer) at GenericTest.main(GenericTest.java:27)



# 제네릭 메소드

- 하지만 일반 클래스의 메소드에서도 타입 매개 변수를 사용하여서 제네릭 메소드를 정의할 수 있다.
- 이 경우에는 타입 매개 변수의 범위가 메소드 내부로 제한된다.

```
public class MyArrayAlg {
```

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

```
}
```

제네릭 메소드 정의





# 제네릭 메소드

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

JAVA



# 예제: 제네릭 메소드 작성하기

- 다음 코드와 같이 정수 배열, 실수 배열, 문자 배열을 모두 출력할 수 있는 제네릭 메소드 `printArray()`를 작성하여 보자.

```
10 20 30 40 50  
1.1 1.2 1.3 1.4 1.5  
K O R E A
```



# 예제: 제네릭 메소드 작성하기

```
public class GenericMethodTest {  
    public static <T> void printArray(T[] array) {  
        for (T element : array) {  
            System.out.printf("%s ", element);  
        }  
        System.out.println();  
    }  
    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);  
    }  
}
```



# 중간점검

1. 데이터를 Object 참조형 변수에 저장하는 것이 왜 위험할 수 있는가?
2. Box 객체에 Rectangle 객체를 저장하도록 제네릭을 이용하여 생성하여 보라.
3. 타입 매개 변수 T를 가지는 Point 클래스를 정의하여 보라. Point 클래스는 2차원 공간에서 점을 나타낸다.
4. 제네릭 메소드 sub()에서 매개 변수 d를 타입 매개 변수를 이용하여서 정의하여 보라.

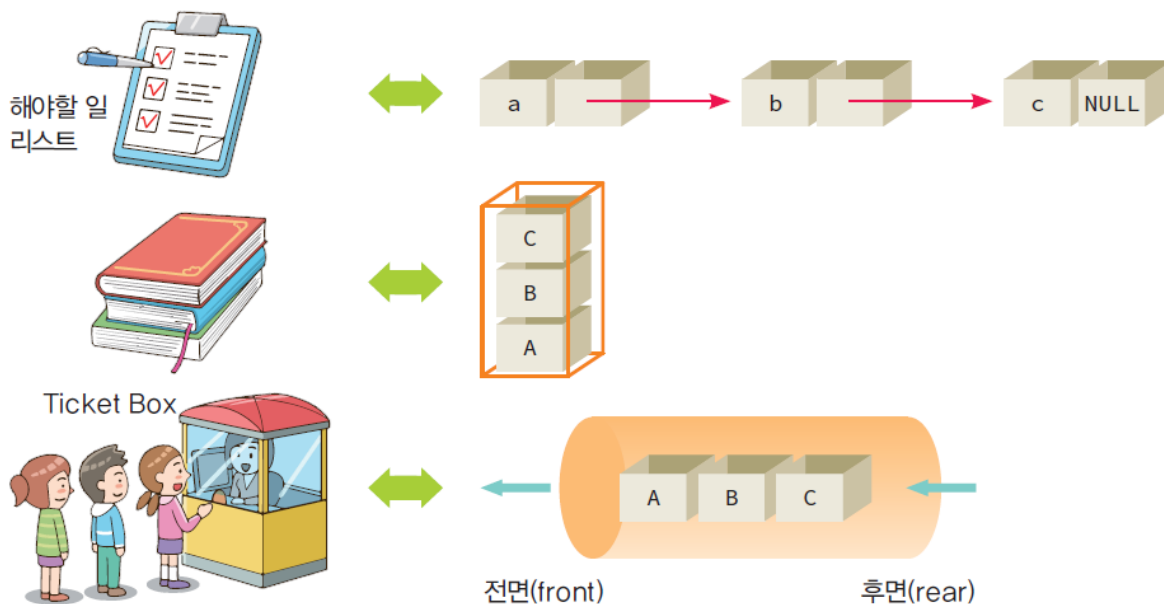


중간점검



# 컬렉션

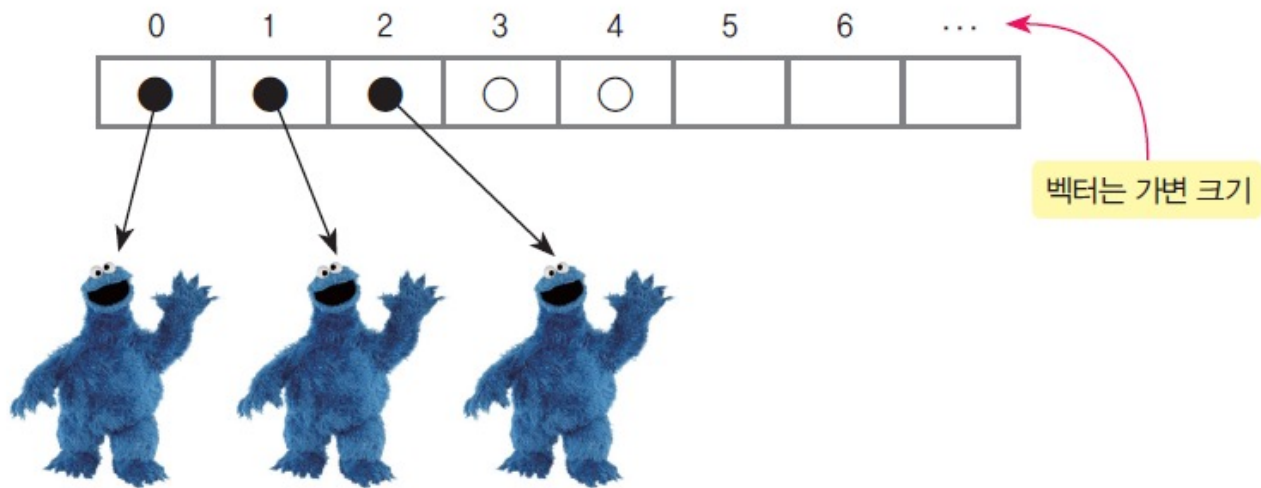
- 컬렉션(collection)은 자바에서 자료 구조를 구현한 클래스
- 자료 구조로는 리스트(list), 스택(stack), 큐(queue), 집합(set), 해쉬 테이블(hash table) 등이 있다.





# 컬렉션의 역사

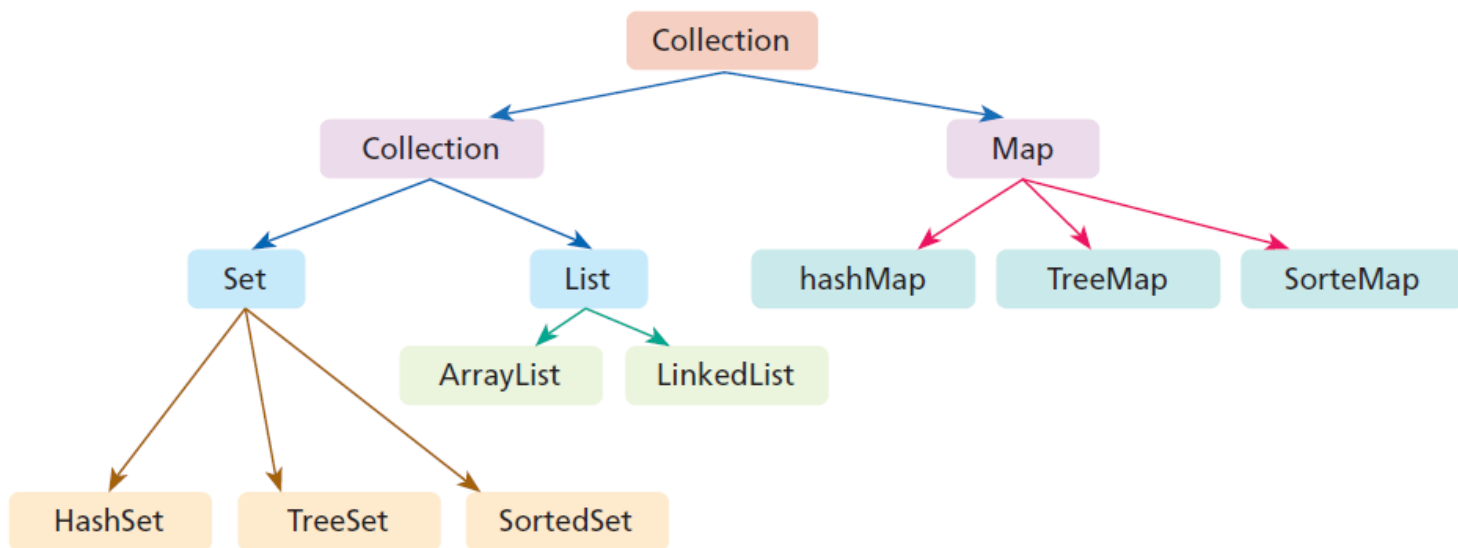
- 초기 버전: Vector, Stack, HashTable, Bitset, Enumeration이 그것이다.
- 버전 1.2부터는 풍부한 컬렉션 라이브러리가 제공
  - 인터페이스와 구현을 분리
  - (예) List 인터페이스를 ArrayList와 LinkedList 클래스가 구현





# 컬렉션의 종류

- 자바는 컬렉션 인터페이스와 컬렉션 클래스로 나누어서 제공한다. 자바에서는 컬렉션 인터페이스를 구현한 클래스도 함께 제공하므로 이것을 간단하게 사용할 수도 있고 아니면 각자 필요에 맞추어 인터페이스를 자신의 클래스로 구현할 수도 있다.





# 컬렉션 인터페이스

표 13.1 컬렉션 인터페이스

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





# 컬렉션의 특징

- 컬렉션은 제네릭을 사용한다.
- 컬렉션에는 `int`나 `double`과 같은 기초 자료형은 저장할 수 없다. 클래스만 가능하다. 기초 자료형을 클래스로 감싼 래퍼 클래스인 `Integer`나 `Double`은 사용할 수 있다.
  - `Vector<int> list = new Vector<int>();` // 컴파일 오류!
  - `Vector<Integer> list = new Vector<Integer>();` // OK !
- 기본 자료형을 저장하면 자동으로 래퍼 클래스의 객체로 변환된다. 이것을 오토박싱(auto boxing)이라고 한다.



# 컬렉션 인터페이스의 주요 메소드

표 13.2 Collection 인터페이스의 메소드

메소드	설명
<code>boolean isEmpty()</code> <code>boolean contains(Object obj)</code> <code>boolean containsAll(Collection&lt;?&gt; c)</code>	공백 상태이면 true 반환 obj를 포함하고 있으면 true 반환
<code>boolean add(E element)</code> <code>boolean addAll(Collection&lt;? extends E&gt; from)</code>	원소를 추가
<code>boolean remove(Object obj)</code> <code>boolean removeAll(Collection&lt;?&gt; c)</code> <code>boolean retainAll(Collection&lt;?&gt; c)</code> <code>void clear()</code>	원소를 삭제
<code>Iterator&lt;E&gt; iterator()</code> <code>Stream&lt;E&gt; stream()</code> <code>Stream&lt;E&gt; parallelStream()</code>	원소 방문
<code>int size()</code>	원소의 개수 반환
<code>Object[] toArray()</code> <code>&lt;T&gt; T[] toArray(T[] a)</code>	컬렉션을 배열로 변환



# 컬렉션의 모든 요소 방문하기

컬렉션 중 하나인 **ArrayList**에 문자열을 저장하였다가 꺼내는 코드를 다양한 방법으로 구현해보자.

```
String a[] = new String[] { "A", "B", "C", "D", "E" };  
List<String> list = Arrays.asList(a);
```

① 전통적인 **for** 구문을 사용할 수 있다.

```
for (int i=0; i<list.size(); i++)  
    System.out.println(list.get(i));
```

② **for-each** 구문을 사용할 수 있다.

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



# ③ 반복자(Iterator)를 사용할 수 있다.

- 반복자는 특별한 타입의 객체로 컬렉션의 원소들을 접근하는 것이 목적이다. **ArrayList** 뿐만 아니라 반복자는 모든 컬렉션에 적용할 수 있다.

메소드	설명
hasNext()	아직 방문하지 않은 원소가 있으면 true를 반환
next()	다음 원소를 반환
remove()	최근에 반환된 원소를 삭제

```
String s;  
Iterator e = list.iterator();  
while(e.hasNext())  
{  
    s = (String)e.next();           // 반복자는 Object 타입을 반환!  
    System.out.println(s);  
}
```



## ④ Stream 라이브러리를 이용하는 방법

- 아마도 가장 간단한 방법이다. **forEach** 메소드와 람다식을 사용한다. 이 방법은 14장에서 자세하게 설명된다.

```
list.forEach((n) -> System.out.println(n));
```



# 중간점검



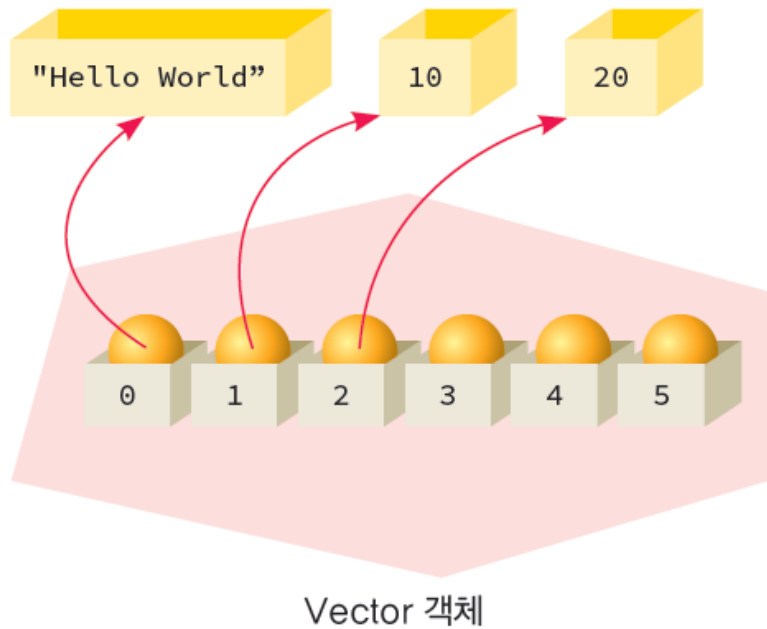
## 중간점검

1. 컬렉션에는 어떤 것들이 있는가?
2. 컬렉션 클래스들은 어디에 이용하면 좋은가?
3. Collection 인터페이스의 각 메소드들의 기능을 자바 API 웹페이지를 이용하여서 조사하여 보자.



# 컬렉션의 예: Vector 클래스

- Vector 클래스는 `java.util` 패키지에 있는 컬렉션의 일종으로 가변 크기의 배열(dynamic array)을 구현하고 있다.

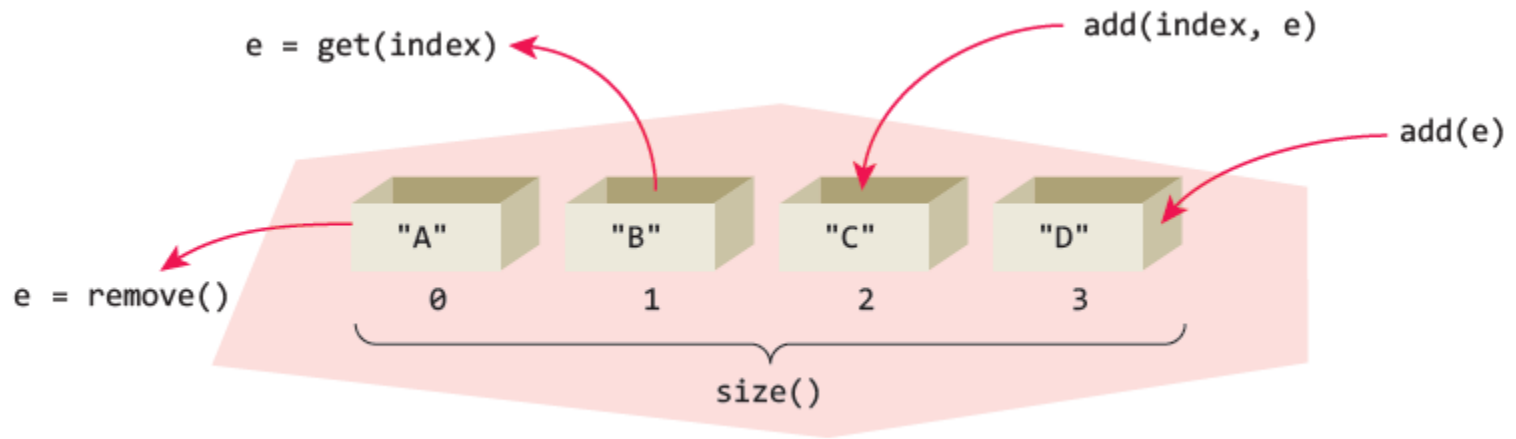


Vector의 크기는  
자동으로  
관리됩니다.





# 벡터의 메소드







# 예제

```
public class VectorTest {  
    public static void main(String[] args) {  
  
        Vector vc = new Vector();  
        vc.add("Hello World!");  
        vc.add(new Integer(10));  
        vc.add(20);  
  
        System.out.println("vector size: " + vc.size());  
  
        for (int i = 0; i < vc.size(); i++) {  
            System.out.println("vector element " + i + ": " + vc.get(i));  
        }  
        String s = (String)vc.get(0);  
    }  
}
```

```
vector size: 3  
vector element 0: Hello World!  
vector element 1: 10  
vector element 2: 20
```



# 제네릭 기능을 사용하는 벡터

```
import java.util.*;

public class VectorExample1 {
    public static void main(String args[]) {

        Vector<String> vec = new Vector<String>(2);
        vec.add("Apple");
        vec.add("Orange");
        vec.add("Mango");

        System.out.println("벡터의 크기: "+vec.size());
        Collections.sort(vec);
        for(String s: vec)
            System.out.print(s + " ");
    }
}
```

벡터의 크기: 3  
Apple Mango Orange



# 예제: 객체를 벡터에 저장하기

- 몬스터를 나타내는 클래스 **Monster**를 정의하고 **Monster** 객체를 몇 개 생성하여서 벡터에 저장한다. 저장된 **Monster** 객체를 꺼내서 출력해보자.

벡터의 크기: 3

[{Mon1,100.0}, {Mon2,200.0}, {Mon3,300.0}]



# 예제: 객체를 벡터에 저장하기

```
class Monster {
    String name;
    double hp;
    public Monster(String name, double hp){
        this.name = name;
        this.hp = hp;
    }
    public String toString(){ return "{"+name+","+hp+""; }
}

public class VectorExample2 {

    public static void main(String args[]) {
        Vector<Monster> list = new Vector<>();

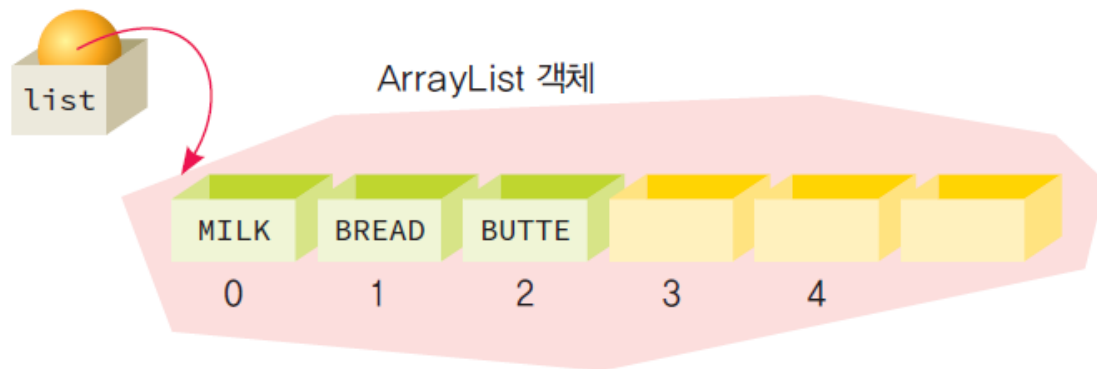
        list.add(new Monster("Mon1", 100));
        list.add(new Monster("Mon2", 200));
        list.add(new Monster("Mon3", 300));

        System.out.println("벡터의 크기: "+list.size());
        System.out.print(list);
    }
}
```



# ArrayList

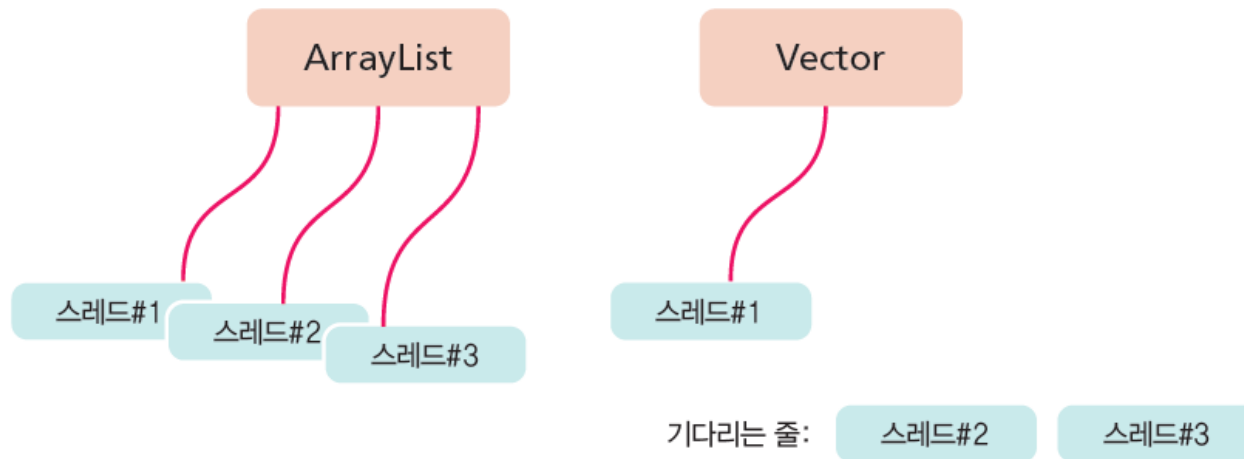
- ArrayList를 배열(Array)의 향상된 버전 또는 가변 크기의 배열이라고 생각하면 된다.
- ArrayList의 생성
  - ArrayList<String> list = **new** ArrayList<String>();
- 원소 추가
  - list.add( "MILK" );
  - list.add( "BREAD" );
  - list.add( "BUTTER" );





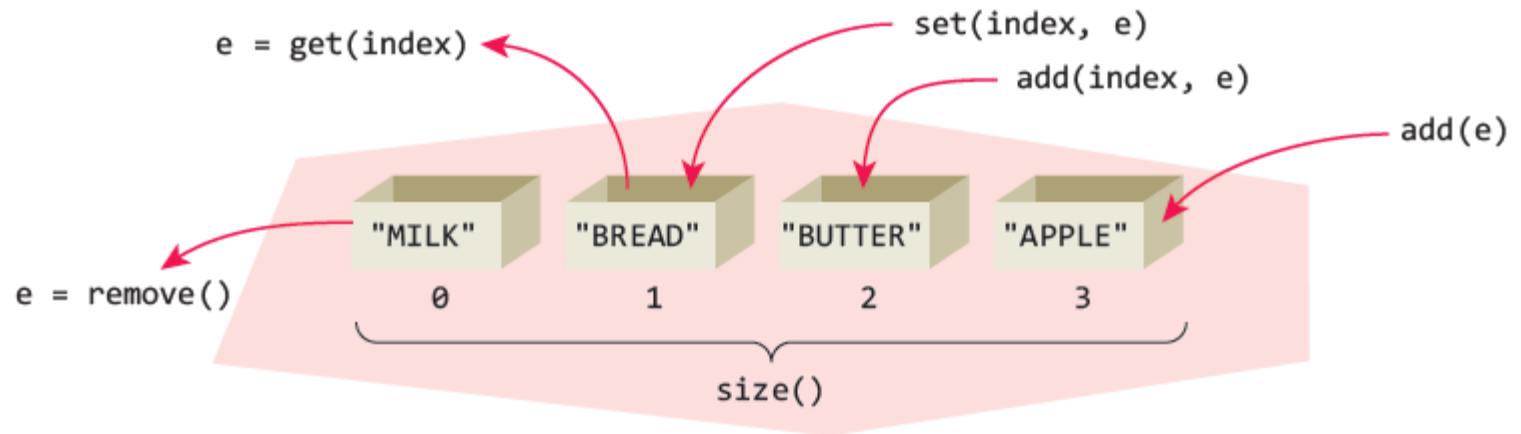
# Vector vs ArrayList

- **Vector**는 스레드 간의 동기화를 지원하는데 반하여 **ArrayList**는 동기화를 하지 않기 때문에 **Vector**보다 성능은 우수하다





# ArrayList의 기본 연산



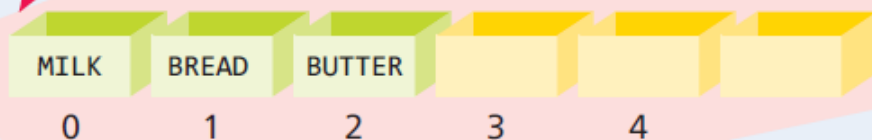


# ArrayList 기본 연산

```
list.add( "MILK" );  
list.add( "BREAD" );  
list.add( "BUTTER" );
```



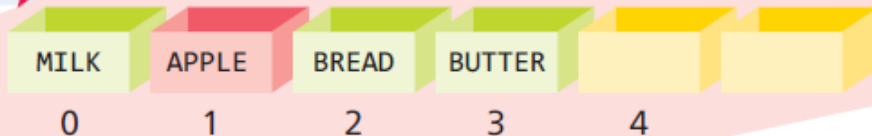
ArrayList 객체



```
list.add( 1, "APPLE" );
```



ArrayList 객체

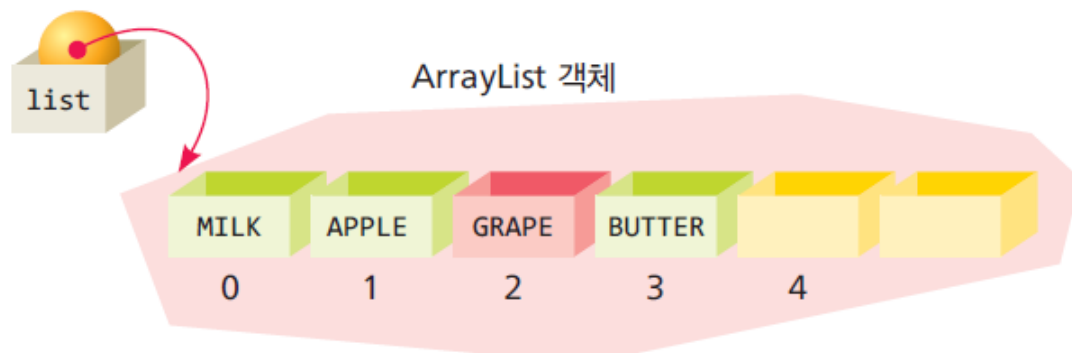




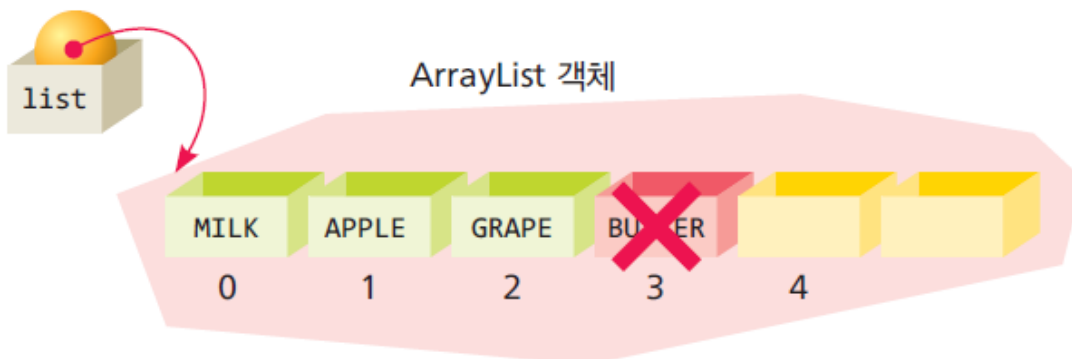


# ArrayList 기본 연산

```
list.set( 2, "GRAPE" ); // 인덱스 2의 원소를 "GRAPE"로 대체
```



```
list.remove( 3 ); // 인덱스 3의 원소를 삭제한다.
```





# 예제: 객체를 ArrayList에 저장하기

- 2차원 공간의 한 점을 나타내는 **Point** 클래스 객체를 저장하는 **ArrayList**를 생성해보자.

```
[(0,0), (4,0), (3,5), (-1,3), (13,2)]
```



# 예제: 객체를 ArrayList에 저장하기

```
class Point {  
    int x, y;  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public String toString(){ return "("+x+","+y+""); }  
}  
  
public class ArrayListTest {  
    public static void main(String args[]) {  
  
        ArrayList<Point> list = new ArrayList<>();  
  
        list.add(new Point(0, 0));  
        list.add(new Point(4, 0));  
        list.add(new Point(3, 5));  
        list.add(new Point(-1, 3));  
        list.add(new Point(13, 2));  
  
        System.out.println(list);  
    }  
}
```



# 예제: 문자열을 ArrayList에 저장

- 문자열을 ArrayList에 저장하고, indexOf() 메소드를 이용하여 특정 문자열을 찾아보자. indexOf()는 Array-List 안에 저장된 데이터를 찾아서 인덱스를 반환하는 메소드이다.

Mango의 위치:2



# 예제: 문자열을 ArrayList에 저장

```
import java.util.ArrayList;

public class ArrayListTest2 {
    public static void main(String[] args) {

        ArrayList<String> list = new ArrayList<>();
        list.add("Apple");

        list.add("Banana");
        list.add("Mango");
        list.add("Pear");
        list.add("Grape");

        int index = list.indexOf("Mango");

        System.out.println("Mango의 위치:"+index);    // 1
    }
}
```



# 참고

불행하게도 자바에서는 배열, ArrayList, 문자열 객체의 크기를 알아내는 방법이 약간 다르다.

- 배열: `array.length`
- ArrayList: `arrayList.size()`
- 문자열: `string.length()`

참고





# 중간점검

1. ArrayList가 기존의 배열보다 좋은 점은 무엇인가?
2. ArrayList의 부모 클래스는 무엇인가?
3. 왜 인터페이스 참조 변수를 이용하여서 컬렉션 객체들을 참조할까?
4. ArrayList 안의 객체들을 반복 처리하는 방법들을 모두 설명하라.

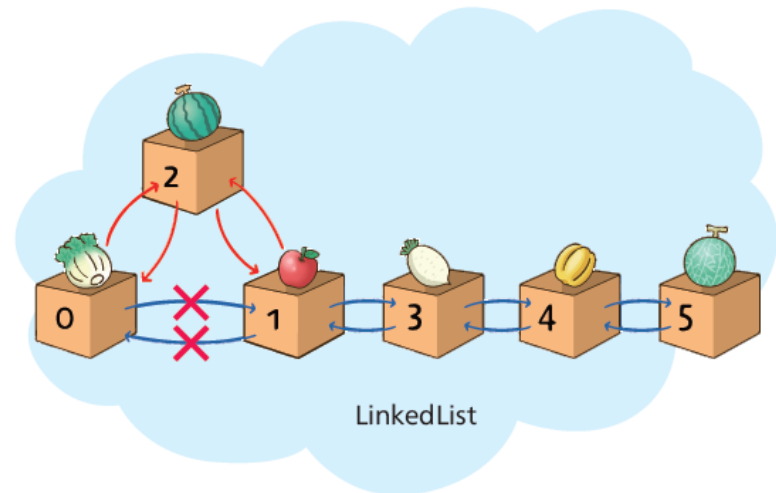
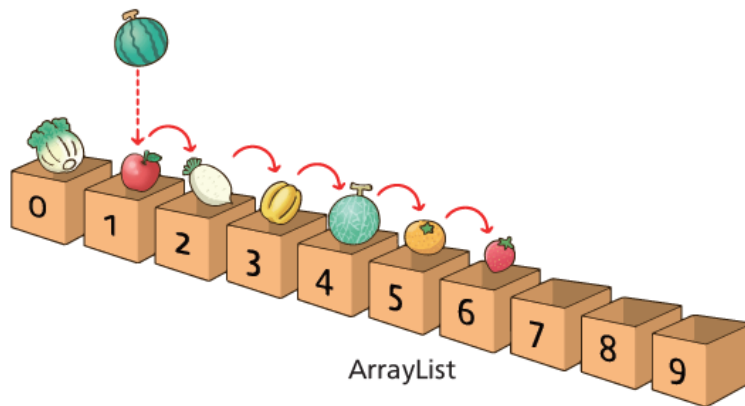


중간점검



# LinkedList

- 빈번하게 삽입과 삭제가 일어나는 경우에 사용

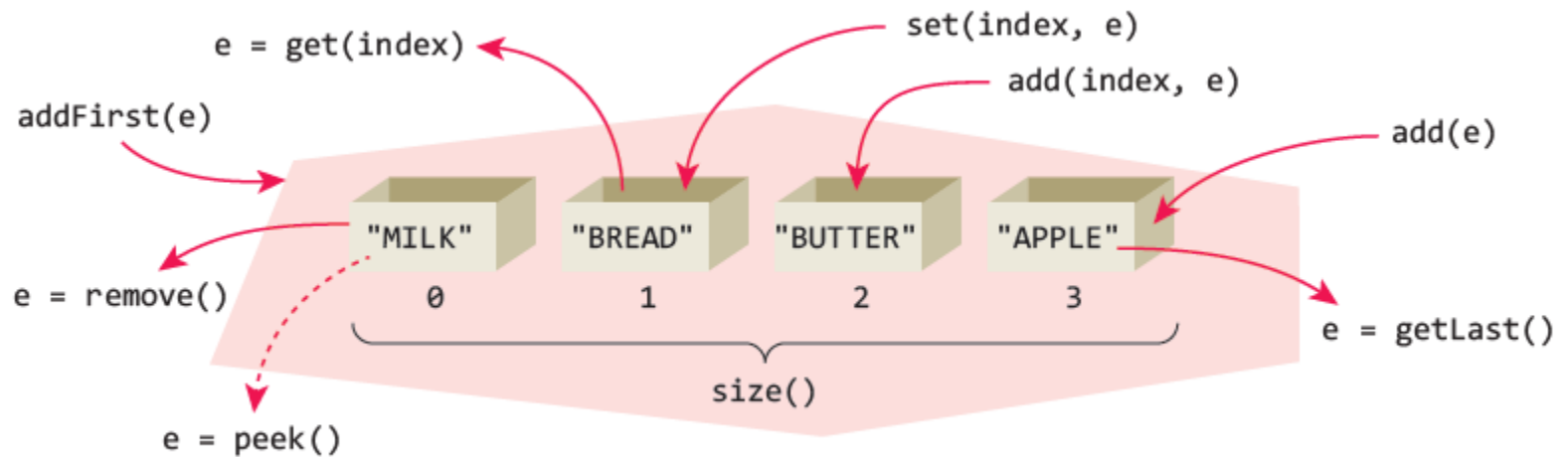


**그림 13.6** 배열의 중간에 데이터를 삽입하려면 원소들을 이동하여야 한다.  
연결 리스트 중간에 삽입하려면 링크만 수정하면 된다.





# LinkedList 기본 연산





# 예제:

```
import java.util.*;

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

        list.add("MILK");
        list.add("BREAD");
        list.add("BUTTER");
        list.add(1, "APPLE");    // 인덱스 1에 "APPLE"을 삽입
        list.set(2, "GRAPE");    // 인덱스 2의 원소를 "GRAPE"로 대체
        list.remove(3);          // 인덱스 3의 원소를 삭제한다.

        for (int i = 0; i < list.size(); i++)
            System.out.print(list.get(i)+" ");
    }
}
```

MILK APPLE GRAPE



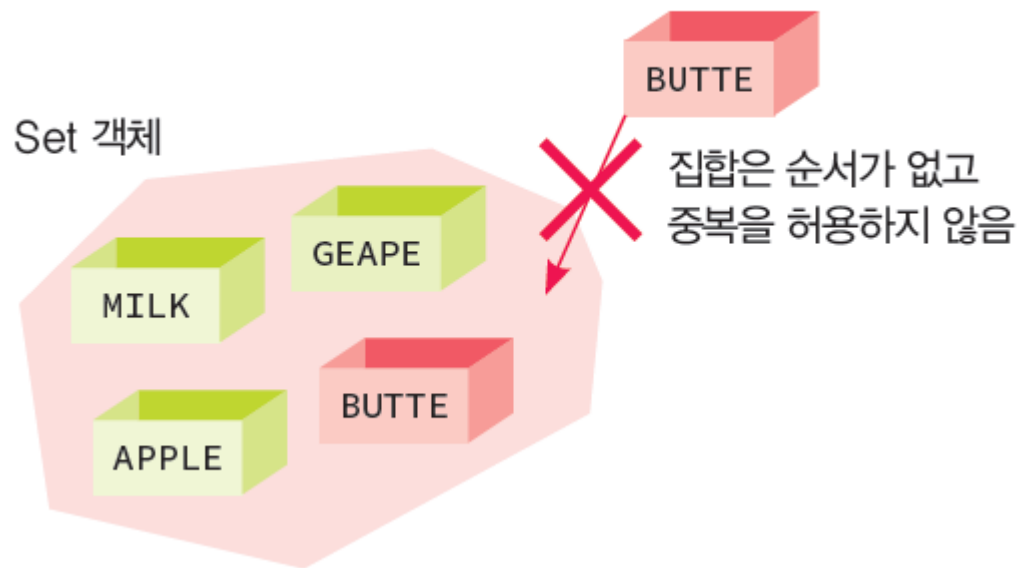
# ArrayList vs LinkedList

- ArrayList는 인덱스를 가지고 원소에 접근할 경우, 항상 일정한 시간만 소요된다. ArrayList는 리스트의 각각의 원소를 위하여 노드 객체를 할당할 필요가 없다. 또 동시에 많은 원소를 이동하여야 하는 경우에는 `System.arraycopy()`를 사용할 수 있다.
- 만약 리스트의 처음에 빈번하게 원소를 추가하거나 내부의 원소 삭제를 반복하는 경우에는 **LinkedList**를 사용하는 것이 낫다. 이들 연산들은 **LinkedList**에서는 일정한 시간만 걸리지만 **ArrayList**에서는 원소의 개수에 비례하는 시간이 소요된다.



# Set

- 집합(Set)은 원소의 중복을 허용하지 않는다.





# Set 인터페이스를 구현하는 방법

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



# 예제: 문자열을 Set에 저장하기

- HashSet을 사용하여 문자열을 저장해보자. `contains()` 메소드도 사용해보자. `contains()`는 집합 안에 데이터가 있는지, 없는지 여부를 반환한다.

[Ham, Butter, Cheese, Milk, Bread]  
Ham도 포함되어 있음



# 예제: 문자열을 Set에 저장하기

```
import java.util.*;

public class SetTest {
    public static void main(String args[]) {
        HashSet<String> set = new HashSet<String>();

        set.add("Milk");
        set.add("Bread");
        set.add("Butter");
        set.add("Cheese");
        set.add("Ham");
        set.add("Ham");

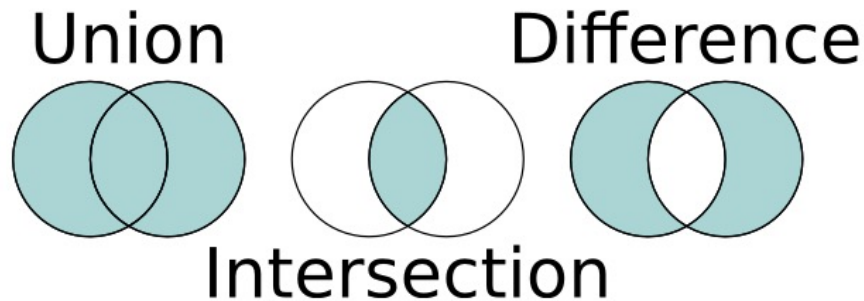
        System.out.println(set);

        if (set.contains("Ham")) {
            System.out.println("Ham도 포함되어 있음");
        }
    }
}
```



# 대량 연산 메소드

- `s1.containsAll(s2)` — 만약 `s2`가 `s1`의 부분 집합이면 참이다.
- `s1.addAll(s2)` — `s1`을 `s1`과 `s2`의 합집합으로 만든다.
- `s1.retainAll(s2)` — `s1`을 `s1`과 `s2`의 교집합으로 만든다.
- `s1.removeAll(s2)` — `s1`을 `s1`과 `s2`의 차집합으로 만든다.







# 예제:

```
Set<Integer> s1 = new HashSet<>(Arrays.asList(1, 2, 3, 4, 5, 7, 9));  
Set<Integer> s2 = new HashSet<>(Arrays.asList(2, 4, 6, 8));
```

```
s1.retainAll(s2);      // 교집합을 계산한다.  
System.out.println(s1);
```

[2, 4]



# 예제: 중복된 단어 검출하기

- 집합은 우리가 잘 알다시피 중복을 허용하지 않는다. 이것을 이용하여 전체 문장에서 중복된 단어를 검출하는 프로그램을 작성할 수 있다.

중복된 단어: 사과

3 중복되지 않은 단어: [토마토, 사과, 바나나]



# 예제: 중복된 단어 검출하기

```
import java.util.*;

public class FindDuplication {

    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        String[] sample = { "사과", "사과", "바나나", "토마토" };
        for (String a : sample)
            if (!s.add(a))
                System.out.println("중복된 단어: " + a);

        System.out.println(s.size() + " 중복되지 않은 단어: " + s);
    }
}
```




# 중간점검



중간점검


1. Set은 어떤 타입의 애플리케이션에 유용한가?
2. Set과 List의 차이점은 무엇인가?



# Map

- Map은 많은 데이터 중에서 원하는 데이터를 빠르게 찾을 수 있는 자료 구조이다.
- 맵은 사전과 같은 자료 구조이다.

```
Map<String, String> map = Map.of("kim", "1234", "park", "pass", "lee", "word");
```

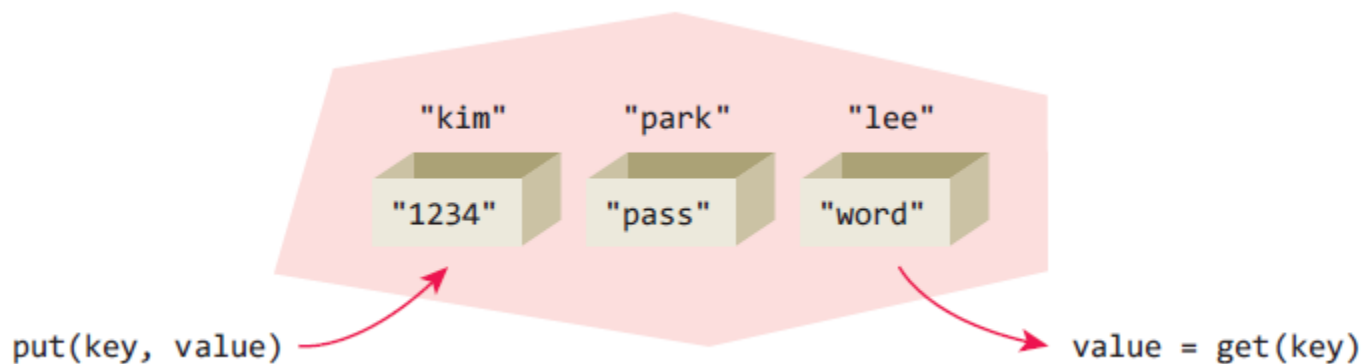


키(key)	값(value)
"kim"	"1234"
"park"	"pass"
"lee"	"word"

그림 13.8 Map의 개념



# Map 기본 연산





# 예제: Map에 학생들의 데이터 저장하기

- 하나의 예로 아이디와 패스워드를 Map에 저장하여 처리하는 코드를 살펴보자.

```
word  
key=lee, value=word  
key=kim, value=1234  
key=park, value=pass  
{choi=password, lee=word, kim=1234, park=pass}
```



# 예제: Map에 학생들의 데이터 저장하기

```
import java.util.HashMap;
import java.util.Map;

public class MapTest {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<String, String>();

        map.put("kim", "1234");
        map.put("park", "pass");
        map.put("lee", "word");

        System.out.println(map.get("lee"));    // 키를 가지고 값을 참조한다.

        for (String key: map.keySet()) {        // 모든 항목을 방문한다.
            String value = map.get(key);
            System.out.println("key=" + key + ", value=" + value);
        }
        map.remove(3);                          // 하나의 항목을 삭제한다.
        map.put("choi", "password");             // 하나의 항목을 대치한다.
        System.out.println(map);
    }
}
```





# Map의 모든 요소 방문하기

① for-each 구문과 keySet()을 사용하는 방법

```
for (String key: map.keySet()) {  
    System.out.println("key=" + key + ", value=" + map.get(key));  
}
```

Java 10 이상의 버전에서는 변수 타입 추론을 사용할 수 있다.

```
for (var key: map.keySet()) {  
    System.out.println("key=" + key + ", value=" + map.get(key));  
}
```



# Map의 모든 요소 방문하기

## ② 반복자를 사용하는 방법

```
Iterator<String> it = map.keySet().iterator();
while (it.hasNext()) {
    String key = it.next();
    System.out.println("key=" + key + ", value=" + map.get(key));
}
```

## ③ Stream 라이브러리를 사용하는 방법

```
map.forEach ( (key, value)-> {
    System.out.println("key=" + key + ", value=" + value);
});
```



# 중간점검



## 중간점검

1. Map의 각 원소들은 \_\_\_\_\_와 \_\_\_\_\_의 두 부분으로 구성되어 있다.
2. Map의 두 가지의 기본적인 연산은 무엇인가?



# 큐(queue)

- 큐는 후단(tail)에서 원소를 추가하고 전단(head)에서 원소를 삭제한다.

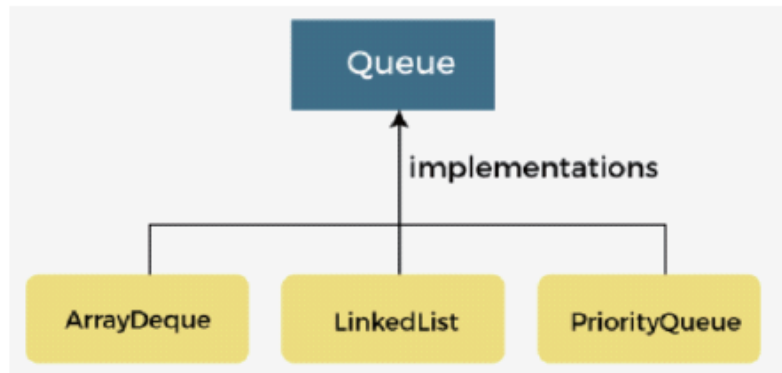


그림 15-11 • 큐



# 큐(queue)

- 자바에서 큐는 **Queue** 인터페이스로 정의되며, 이 **Queue** 인터페이스를 구현한 3개의 클래스가 주어진다.
  - ArrayDeque
  - LinkedList
  - PriorityQueue





# 큐 기본 연산

- **add()** 메소드는 새로운 원소의 추가가 큐의 용량을 넘어서지 않으면 원소를 추가한다.
- **remove()**와 **poll()**는 큐의 처음에 있는 원소를 제거하거나 가져온다. 정확히 어떤 원소가 제거되느냐는 큐의 정렬 정책에 따라 달라진다.



# 예제:

카운트다운 타이머  
구현

```
import java.util.LinkedList;
import java.util.Queue;

public class QueueTest {
    public static void main(String[] args) {
        Queue<Integer> q = new LinkedList<>();

        for (int i = 0; i < 5; i++)
            q.add(i);
        System.out.println("큐의 요소: " + q);

        int e = q.remove();
        System.out.println("삭제된 요소: " + e);
        System.out.println(q);
    }
}
```

큐의 요소: [0, 1, 2, 3, 4]  
삭제된 요소: 0  
[1, 2, 3, 4]



# 우선순위큐

- 우선 순위큐는 원소들이 무작위로 삽입되었더라도 정렬된 상태로 원소들을 추출한다. 즉 **remove()**를 호출할 때마다 가장 작은 원소가 추출된다.
- 우선 순위큐는 힙(heap)라고 하는 자료 구조를 내부적으로 사용한다.







# 우선순위큐

- 우선 순위 큐는 원소들이 무작위로 삽입되었더라도 정렬된 상태로 원소들을 추출한다. 즉 `remove()`를 호출할 때마다 가장 작은 원소가 추출된다

```
import java.util.*;

public class PriorityQueueTest {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
        pq.add(30);
        pq.add(80);
        pq.add(20);

        System.out.println(pq);
        System.out.println("삭제된 원소: " + pq.remove());
    }
}
```

[20, 80, 30]  
삭제된 원소: 20



# Collections 클래스

- Collections 클래스는 여러 유용한 알고리즘을 구현한 메소드들을 제공한다.
- 정렬(Sorting)
- 섞기(Shuffling)
- 탐색(Searching)



# 정렬

- 정렬은 데이터를 어떤 기준에 의하여 순서대로 나열하는 것이다.

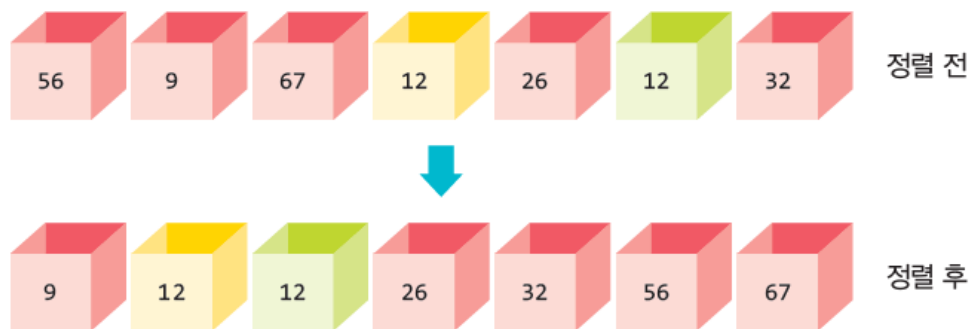


그림 13.8 안정된 정렬

```
List<String> list = new LinkedList<String>();  
list.add("김철수");  
list.add("김영희");  
Collections.sort(list);  
// 리스트 안의 문자열이 정렬된다.
```



# 예제: 정렬

```
import java.util.*;

public class Sort {
    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);
    }
}
```

[i, line, the, walk]



# 예제: 사용자 클래스의 객체 정렬하기

```
import java.util.*;

class Student implements Comparable<Student> {
    int number;
    String name;

    public Student(int number, String name) {
        this.number = number;
        this.name = name;
    }

    public String toString() {        return name;    }

    public int compareTo(Student s) {
        return s.number - number;
    }
}
```



# 예제: 사용자 클래스의 객체 정렬하기

```
public class SortTest {  
    public static void main(String[] args) {  
        Student array[] = {  
            new Student(2, "김철수"),  
            new Student(3, "이철수"),  
            new Student(1, "박철수"),  
        };  
  
        List<Student> list = Arrays.asList(array);  
        Collections.sort(list);  
        System.out.println(list);  
    }  
}
```

[박철수, 김철수, 이철수]



# 섞기

- 섞기(Shuffling) 알고리즘은 정렬의 반대 동작을 한다. 즉 리스트에 존재하는 정렬을 파괴하여서 원소들의 순서를 랜덤하게 만든다.

*Shuffle.java*

```
01  import java.util.*;
02
03  public class Shuffle {
04      public static void main(String[] args) {
05          List<Integer> list = new ArrayList<Integer>();
06          for (int i = 1; i <= 10; i++)
07              list.add(i);
08          Collections.shuffle(list);
09          System.out.println(list);
10      }
11  }
```

[5, 9, 7, 3, 6, 4, 8, 2, 1, 10]



# 탐색

- 탐색(**Searching**)이란 리스트 안에서 원하는 원소를 찾는 것이다. 만약 리스트가 정렬되어 있지 않다면 처음부터 모든 원소를 방문할 수밖에 없다(선형 탐색). 하지만 리스트가 정렬되어 있다면 중간에 있는 원소와 먼저 비교하는 것이 좋다(이진 탐색).

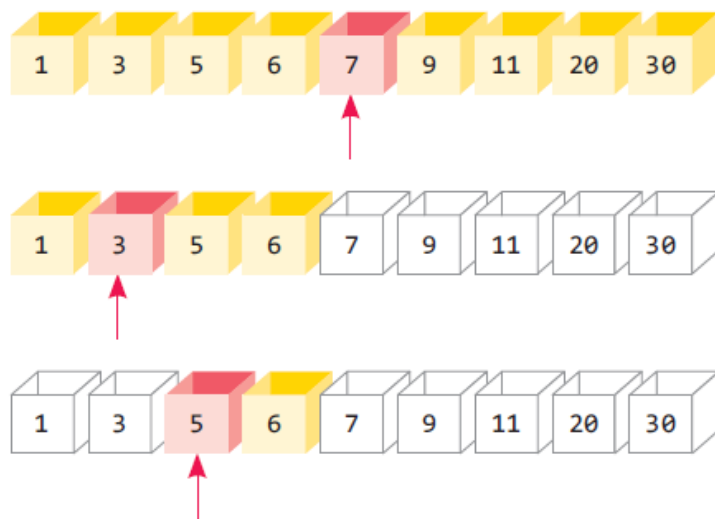


그림 13.9 탐색의 개념





# 예제:

*Search.java*

```
01  import java.util.*;
02
03  public class Search {
04      public static void main(String[] args) {
05          int key = 50;
06          List<Integer> list = new ArrayList<Integer>();
07          for (int i = 0; i < 100; i++)
08              list.add(i);
09          int index = Collections.binarySearch(list, key);
10          System.out.println("탐색의 반환값  = " + index);
11      }
12  }
```

탐색의 반환값 =50



# Lab: 영어 사전 구현

- 여기서는 **Map**을 사용하여 영어 사전을 구현하여 보자. 사용자가 단어를 입력하면 단어의 설명을 보여준다.

영어 단어를 입력하시오:map

단어의 의미는 지도

영어 단어를 입력하시오:school

단어의 의미는 학교

영어 단어를 입력하시오:quit



# Sol: 영어 사전 구현

```
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 단어를 입력하시오:");
            String key = sc.next();
            if( key.equals("quit") ) break;
            System.out.println("단어의 의미는 " + st.get(key));
        } while(true);
    }
}
```



# Mini Project: 카드게임

- 제네릭과 컬렉션을 이용하여서 카드 게임 프로그램을 작성해보자. 먼저 어떤 클래스가 필요할 지를 생각해보자. 카드 게임 세계에서 필요한 것은 “카드”, “덱”, “경기자”이다. 따라서 이것들은 모두 클래스로 작성된다.
  - Card
  - Deck
  - Player





# Summary

- 제네릭은 클래스를 정의할 때, 클래스 안에서 사용되는 자료형(타입)을 구체적으로 명시하지 않고 T와 같이 기호로 적어놓는 것이다.
- 컬렉션(collection)은 자료를 저장하기 위한 구조이다. 많이 사용되는 컬렉션에는 리스트(list), 스택(stack), 큐(queue), 집합(set), 해쉬 테이블(hash table) 등이 있다.
- 벡터(Vector) 클래스는 java.util 패키지에 있는 컬렉션의 일종으로 가변 크기의 배열(dynamic array)을 구현하고 있다.
- ArrayList도 가변 크기의 배열을 구현하는 클래스이다. ArrayList는 동기화를 하지 않기 때문에 Vector보다 성능은 우수하다.
- 순서에는 상관없이 데이터만 저장하고 싶은 경우에는 집합(Set)을 사용할 수 있다. 집합(set)은 동일한 데이터를 중복해서 가질 수 없다.
- Map은 많은 데이터 중에서 원하는 데이터를 빠르게 찾을 수 있는 컬렉션이다. 다른 언어에서는 딕셔너리(dictionary)라고도 한다.
- Map은 키(key)와 값(value)을 쌍으로 저장한다. 각 키는 오직 하나의 값에만 매핑될 수 있다.
- Collections 클래스는 정렬(Sorting), 섞기(Shuffling), 탐색(Searching)과 같은 유용한 알고리즘을 구현한 메소드들을 제공한다.





# Q & A

