
JAVA
제네릭과 컬렉션

1 제네릭과 컬렉션

1. 제네릭 클래스

2. 제네릭 메소드

3. 컬렉션

제네릭이란?

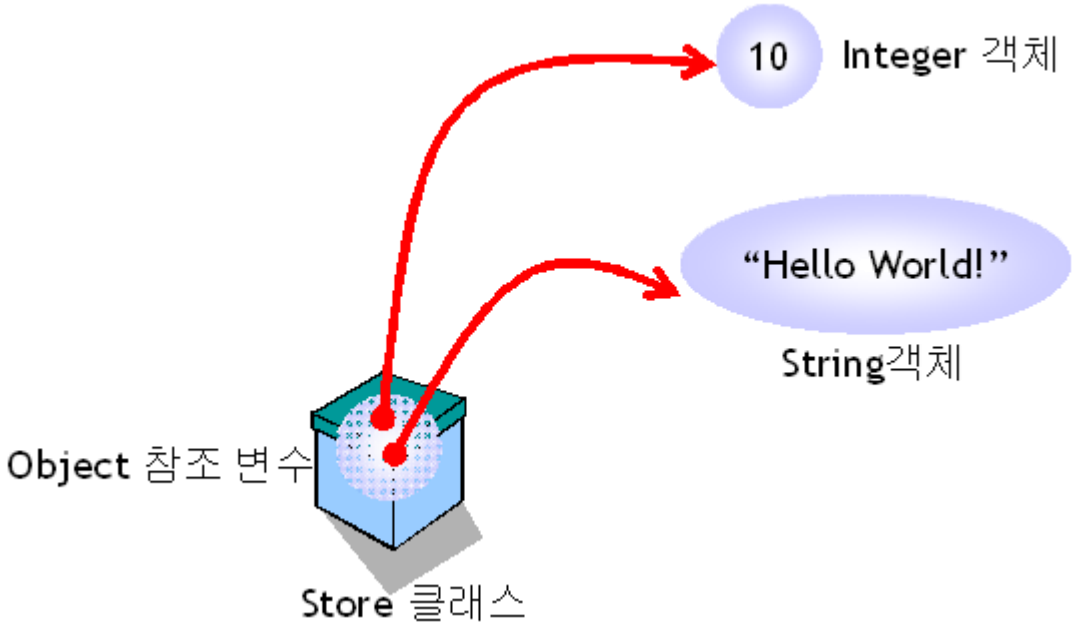
제네릭 프로그래밍(generic programming)-자바 버전 1.5

- 클래스를 정의하면서 사용할 변수의 자료형을 설계할 때 (코드 작성시) 결정하는 것이 아니라 컴파일 할 때 자료형을 검사해서 자료형에 유연하면서 안정성까지 고려할 수 있도록 제공하는 기술이다.

- 제네릭은 어떠한 자료형을 기반으로도 인스턴스의 생성이 가능하도록 자료형에 일반적인 클래스를 정의하는 문법(재사용이 가능)

기존의 방법

일반적인 객체를 처리하려면 Object 참조 변수를 사용
Object 참조 변수는 어떤 객체이던지 참조할 수 있다.
예제로 하나의 데이터를 저장하는 Store클래스를 살펴보자



기존의 방법(코드)

```
public class Store {
        private String data;

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

기존의 방법(코드)

```
public class Store {
        private Integer data;

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

자료형에 따라 같은 역할을 수행하는 클래스가 여러 번 중되어 선언되어야 하므로 비효율적이다.

기존의 방법(코드)

```
public class Store {
        private Object data;

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

이러한 문제점을 해결하기 위해 다형성을 적용해 다음과 같이 적용할 수 있다.

기존의 방법(코드)

```
public void set(Object data) { this.data = data; }
```

```
Store store = new Store();

Integer val = new Integer(10);
store.set(val);
Integer i = (Integer)store.get();

String str = "Java";
store.set(str);
String s =(String)store.get();
```

```
public Object get() { return data; }
```

데이터를 접근 할 때마다 항상 원하는 타입으로 **형변환**을 하여야 한다는 점이다.

기존의 방법(코드)

```
Store store = new Store();

String str = "Java";
store.set(str);

System.out.println("값: " + (Integer)box.get());
```

구현면에서는 효율적이고 편리하지만, 자료형에 매우 불안정한 구조가 된다.

제네릭을 이용한 버전

제네릭 기법을 이용하게 되면 앞의 문제들을 모두 해결할 수 있다. 제네릭 클래스(generic class)에서는 타입을 변수로 표시한다. 이것을 **타입매개변수(type parameter)**라고 하는데 타입매개변수는 객체 생성 시에 프로그래머에 의하여 결정된다.

```
private Object data;

private  T data;
```

T라는 이름이 매개변수화된 자료형임을 나타낸다.

제네릭을 이용한 버전

- 제네릭 타입이란?
  - 타입을 파라미터로 가지는 클래스와 인터페이스
  - 선언 시 클래스 또는 인터페이스 이름 뒤에 “<>” 부호 붙임
  - “<>” 사이에는 타입 파라미터가 위치한다.

- 타입 파라미터
  - 일반적으로 **대문자 알파벳 한 문자로 표현**한다.
  - 실행 클래스에서 타입 파라미터 자리에 구체적인 타입을 지정해 주어야 한다.

타입 매개 변수의 표기

한가지 주의할 점은 제네릭 클래스는 여러 개의 타입매개 변수를 가질 수 있으나 타입의 이름은 클래스나 인터페이스 안에서 유일하여야 한다. 이것은 변수의 이름과 타입의 이름을 구별할 수 있게 하기 위함이다.

- E - Element
- K - Key
- N - Number
- T - Type
- V - Value

제네릭을 이용한 버전

```
public class Store<T> {
        private T data;

        public void set(T data) {
                this.data = data;
        }
        public T get() {
                return data;
        }
}
```

제네릭을 이용한 버전

타입매개변수의 값은 객체를 생성할 때 구체적으로 결정된다.

- 문자열을 저장하려면 다음과 같이 선언
  - Store<String> store = new Store<String>();
- 정수를 저장하려면 다음과 같이 선언
  - Store<Integer> store = new Store<Integer>();

제네릭 버전의 사용

```
Store<String> store = new Store<String>();
String str = "java";
store.set(str);

String s = store.get();
System.out.println(s);
```

Store<String>은 클래스의 형이 아니라 캐스팅을 판단하기 위한 표시이다.

제네릭 버전의 사용

만약 Store<String>에 정수 타입을 추가하려고 하면 컴파일러가 컴파일 단계에서 오류를 감지할 수 있다.

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

제네릭을 사용하는 코드의 이점

- 제네릭을 사용하는 코드의 이점
  - 컴파일 시 강한 타입 체크 가능
    : 실행 시 타입 에러가 나는 것 방지

  - 컴파일 시에 미리 타입을 강하게 체크해서 에러 사전 방지

제네릭과 다형성

만약 어떤 클래스가 다른 클래스로부터 상속되었다면 수퍼 클래스의 참조 변수에 서브 클래스의 참조 변수를 대입할 수 있다.

```
Object o = new Object();
Integer i = new Integer(20);
o=i;
```

```
Store<Object> store = new Store<Object>();
store.set(new Integer(10));
store.set(new Double(10.1));
```

기본자료형은 제네릭 클래스의 인스턴스 생성에 사용할 수 없다.

제네릭 만들기

제네릭 클래스와 인터페이스

- 클래스나 인터페이스 선언부에 일반화된 타입 추가

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

value의 타입은 T

제네릭 클래스 MyClass 선언, 타입 매개 변수 T

T 타입의 값 value를 필드 value에 지정

T 타입의 값 value 리턴

- 제네릭 클래스 레퍼런스 변수 선언

```
MyClass<String> s;
List<Integer> list;
Vector<String> vs;
```

제네릭 객체 생성 – 구체화(specialization)

구체화

- 제네릭 타입의 클래스에 구체적인 타입을 대입하여 객체 생성
- 컴파일러에 의해 이루어짐

```
MyClass<String> s = new MyClass<String>(); // 제네릭 타입 T에 String 지정
s.set("hello");
System.out.println(s.get()); // "hello" 출력

MyClass<Integer> n = new MyClass<Integer>(); // 제네릭 타입 T에 Integer 지정
n.set(5);
System.out.println(n.get()); // 숫자 5 출력
```

- 구체화된 MyClass<String>의 소스 코드

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


T가 String 으로 구체화

```
public class MyClass<String> {
   private String value; // 변수 value의 타입은 String
   public void set(String value) {
      this.value = value; // String 타입의 값 value를 value지정
   }
   public String get() {
      return value; // String 타입의 값 value을 리턴
   }
}
```

구체화 오류

타입 매개 변수에 기본 타입은 사용할 수 없음

```
Vector<int> vi = new Vector<int>();                    // 컴파일 오류. int 사용 불가
```



```
Vector<Integer> vi = new Vector<Integer>(); // 정상 코드
```

예제 7-9 : 제네릭 스택 만들기

스택을 제네릭 클래스로 작성하고, String과 Integer형 스택을 사용하는 예를 보여라.

```
class GStack<T> {
   int tos;
   Object [] stck;
   public GStack() {
      tos = 0;
      stck = new Object [10];
   }
   public void push(T item) {
      if(tos == 10)
         return;
      stck[tos] = item;
      tos++;
   }
   public T pop() {
      if(tos == 0)
         return null;
      tos--;
      return (T)stck[tos];
   }
}
```

```
public class MyStack {
   public static void main(String[] args) {
      GStack<String> stringStack = new GStack<String>();
      stringStack.push("seoul");
      stringStack.push("busan");
      stringStack.push("LA");

      for(int n=0; n<3; n++)
         System.out.println(stringStack.pop());


      GStack<Integer> intStack = new GStack<Integer>();
      intStack.push(1);
      intStack.push(3);
      intStack.push(5);

      for(int n=0; n<3; n++)
         System.out.println(intStack.pop());
   }
}
```

```
LA
busan
seoul
5
3
1
```

제네릭과 배열

제네릭에서 배열의 제한

- 제네릭 클래스 또는 인터페이스의 배열을 허용하지 않음

```
GStack<Integer>[] gs = new GStack<Integer>[10];
```

- 제네릭 타입의 배열도 허용되지 않음

```
T[] a = new T[10];
```

•앞 예제에서는 Object 타입으로 배열 생성 후 실제 사용할 때 타입 캐스팅

```
return (T)stck[tos]; // 타입 매개 변수 T타입으로 캐스팅
```

- 타입 매개변수의 배열에 레퍼런스는 허용

```
public void myArray(T[] a) {....}
```

제네릭 메소드

매개변수 타입과 리턴 타입으로 타입 파라미터를 갖는 메소드를 제네릭 메서드라고 한다.

- 제네릭 메소드 선언 방법
  - 리턴 타입 앞에 “<>” 기호를 추가하고 타입 파라미터 기술
  - 타입 파라미터를 리턴 타입과 매개변수에 사용

```
public <타입파라미터,...> 리턴타입 메서드명(매개변수,...){   }
```

```
public static <T> T getLast(T[] arr) { ... }
```

제네릭 메소드

- 제네릭 메소드 호출하는 두 가지 방법

1. 명시적으로 구체적 타입 지정
리턴타입 변수 = <구체적인타입>메서드명(인수);

2. 인수값을 보고 구체적 타입을 추정
리턴타입 변수 = 메서드명(인수);

제네릭 메소드

메소드 getLast()는 일반 클래스 안에 정의되어 있다. 그러나 <T>를 가지고 있으므로 제네릭 메소드이다.

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

배열의 마지막 원소를 반환하는 메소드

제네릭 메소드의 사용

```
String[]   language= { "C++", "C#", "JAVA" };
String  last = GenericMethod.<String>getLast(language);
```

메소드 호출시는 <String>는 생략하여도 된다. 왜냐하면 컴파일러는 이미 타입 정보를 알고 있기 때문이다. 즉 다음과 같이 호출하여도 된다.

```
String  last = GenericMethod.getLast(language);
```

한정된 타입매개변수

타입매개변수로 전달되는 타입의 종류를 제한하고 싶은 경우가 있다. 예를 들어서 특정한 종류의 객체들만을 받게 하고 싶은 경우가 있다.

이런 경우에 사용할 수 있는 것이 **한정된 타입매개변수 (bounded type parameter)**이다.

이 기능을 사용하기 위해서는 extends나 super 라는 키워드를 사용한다.

한정된 타입매개변수

배열 원소 중에서 가장 작은 값을 반환하는 제네릭 메소드 작성

```
public class ArrayData {
        public static <T> 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;
        }
}
```

한정된 타입매개변수

여기서는 한가지 문제가 있다. 만약 T가 compareTo()라고 하는 Comparable 인터페이스를 구현하지 않은 클래스라면 어떻게 되는가? 틀림없이 오류가 발생할 것이다.

따라서 타입매개변수 T가 가리킬 수 있는 클래스 범위를 Comparable 인터페이스를 구현한 클래스로 제한하는 것이 바람직하다.

한정된 타입매개변수

```
public static <T extends Comparable<T>> T
getMax(T[] a) {
        … //<T extends 객체참조형>T는 클래스를 상속하
          //거나 구현하는 클래스의 자료형이 되어야 함을
          //명시하는 문법
}
```

상속 및 구현 관계 이용해
타입 제한

한정된 타입매개변수

코드에서 ? 를 일반적으로 와일드카드(wildcard)라고 부른다. 제네릭 타입을 매개값이나 리턴타입으로 사용할 때 구체적인 타입 대신에 와일드카드를 다음과 같이 세 가지 형태로 사용할 수 있다.

한정된 타입매개변수

<?> 제한없음(객체내부 모든 제네릭 타입은 Object로 인식)

: 타입 파라미터를 대치하는 구체적인 타입으로 모든 클래스나 인터페이스 타입이 올 수 있다.

<? extends 상위타입> 상위 클래스 제한

: 타입 파라미터를 대치하는 구체적인 타입으로 상위 타입이나 하위 타입만 올 수 있다.

<? super 하위타입> 하위 클래스 제한

: 타입 파라미터를 대치하는 구체적인 타입으로 하위 타입이나 상위 타입이 올 수 있다.

자료 구조란

하나의 프로그램은 크게 원하는 산출물을 얻기 위해 작성하는 명령어(코드)와 명령어를 실행하면서 사용하는 데이터로 이루어진다. 프로그램이란 데이터를 표현하고 데이터를 처리하는 것이다. 이러한 데이터를 효율적으로 사용되도록 구조적으로 저장하고 관리하는 것이 자료구조이다. 이를 통해 데이터를 추가, 수정, 삭제, 검색이 효율적으로 이루어진다

프로그램 구성	=	코드	+	데이터

데이터 관리 기술 = 자료구조

알고리즘이란 자료구조가 데이터의 표현과 저장에 대한 방법이라면 알고리즘은 표현되거나 저장된 데이터를 대상으로 '문제를 해결하는 방법'이다. 그러므로 알고리즘과 자료구조는 서로 밀접하게 관련되어 있다.

자료구조와 알고리즘

- 자료구조는 데이터의 저장과 관련이 있는 학문으로 효율적인 데이터 저장방법을 연구하는 학문.
  - 정형화하고 있는 데이터 저장방식
    : 배열, 리스트, 스택, 트리, 큐, 해시 정리

- 알고리즘은 저장된 데이터의 일부 또는 전체를 대상으로 각종 연산을 연구하는 학문.
  - 알고리즘 : 정렬, 탐색 등으로 정리

컬렉션

컬렉션이란 자료구조와 알고리즘을 클래스로 구현해 놓은 것이다.

그래서 컬렉션 관련 클래스들은 많은 양의 인스턴스를 다양한 형태로 저장하는 기능을 제공한다. 컬렉션을 잘 활용하면 다양하고 효율적으로 인스턴스의 저장이 가능하다.

※ 배열의 문제점

- 저장할 수 있는 객체 수가 배열을 생성할 때 결정
  → 불특정 다수의 객체를 저장하기에는 문제

- 객체 삭제했을 때 해당 인덱스가 비게 됨
  → 객체를 저장하려면 어디가 비어있는지 확인해야 함.

컬렉션

보통 컬렉션(collection)이라 하면 기본적으로 객체들의 집합을 의미한다. 객체를 담을 수 있으며 컨테이너(container)라고도 한다. 컬렉션에서는 객체를 컬렉션에 추가하거나 혹은 삭제하거나 검색하는 등의 객체 단위의 연산을 제공한다. 또한 다른 컬렉션과 결합하거나 분리하는 집합 연산도 제공한다.

자바는 컬렉션 관련 java.util 패키지에 자료 구조에 사용하는 인터페이스, 인터페이스로 구현된 클래스, 관련된 알고리즘이 있다. 이들 셋을 컬렉션 프레임워크(Collection Framework)라고 한다.

프레임워크란 구체적이고 체계화된 API(프로그래밍 접근 방식)을 제공하다는 의미이다. 자료구조에 관계없이 추가하려면 add(), 삭제하려면 remove() 같이 표준화된 접근 방식을 제공하는 것을 프레임워크라 한다.

자바에서의 컬렉션

컬렉션 프레임워크
- 컬렉션 인터페이스
- 컬렉션 클래스

프레임워크란 사용 방법을 미리 정해 놓은 라이브러리를 말한다. 자바 컬렉션 프레임워크는 몇 가지 인터페이스를 통해서 다양한 컬렉션 클래스를 이용 할 수 있도록 하고 있다.

- java.util 패키지에 포함.
- 컬렉션 라이브러리들은 모두 제네릭 기능을 지원.
- 인터페이스를 통해서 정형화된 방법으로 다양한 컬렉션 클래스 이용.
- 기본형 데이터를 처리하는 기능이 없다.(기본형의 데이터를 다루려면 Wrapper 클래스를 사용해서 기본형을 참조형으로 변환.

자바의 타입 추론 기능의 진화

Java 7 이전

```
ArrayList<Integer> v = new ArrayList<Integer>(); // Java 7 이전
```

Java 7 이후

- 컴파일러의 타입 추론 기능 추가
- <>(다이어몬드 연산자)에 타입 매개변수 생략

```
ArrayList<Integer> v = new ArrayList<>(); // Java 7부터 추가, 가능
```

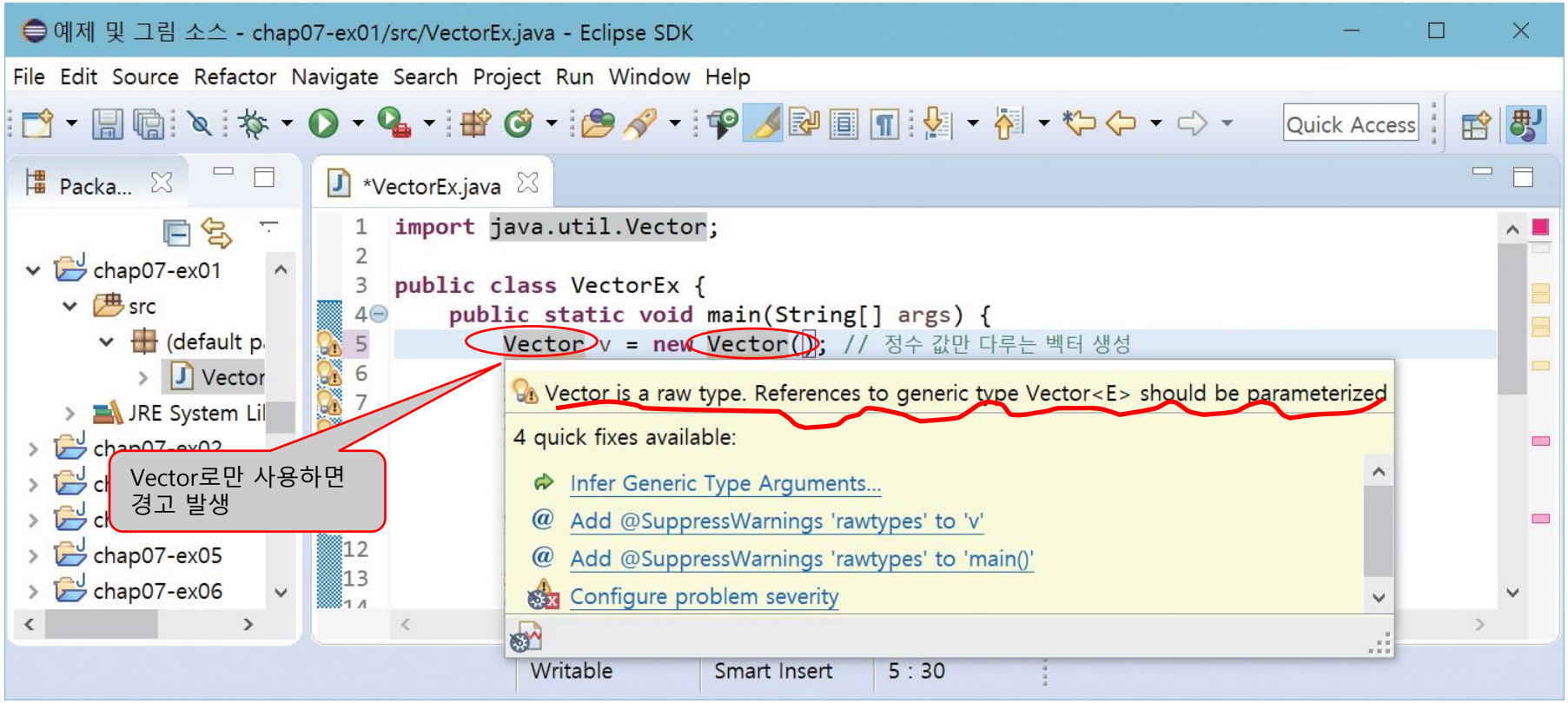
Java 10 이후

- var 키워드 도입, 컴파일러의 지역 변수 타입 추론 가능

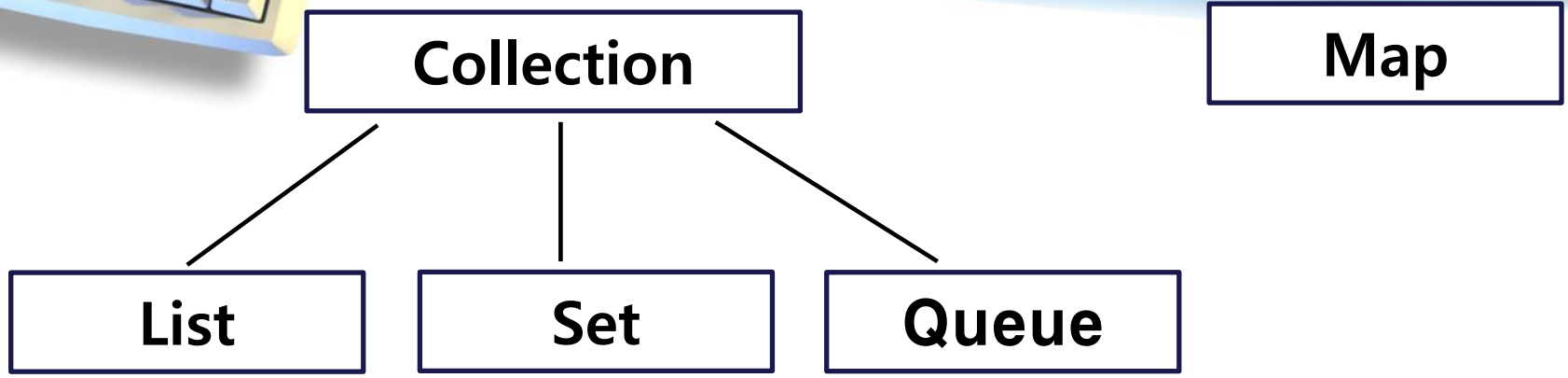
```
var v = new ArrayList<Integer>(); // Java 10부터 추가, 가능
```

타입 매개 변수 사용하지 않는 경우 경고 발생

Vector<Integer>로 타입 매개 변수를 사용하여야 함



컬렉션 인터페이스

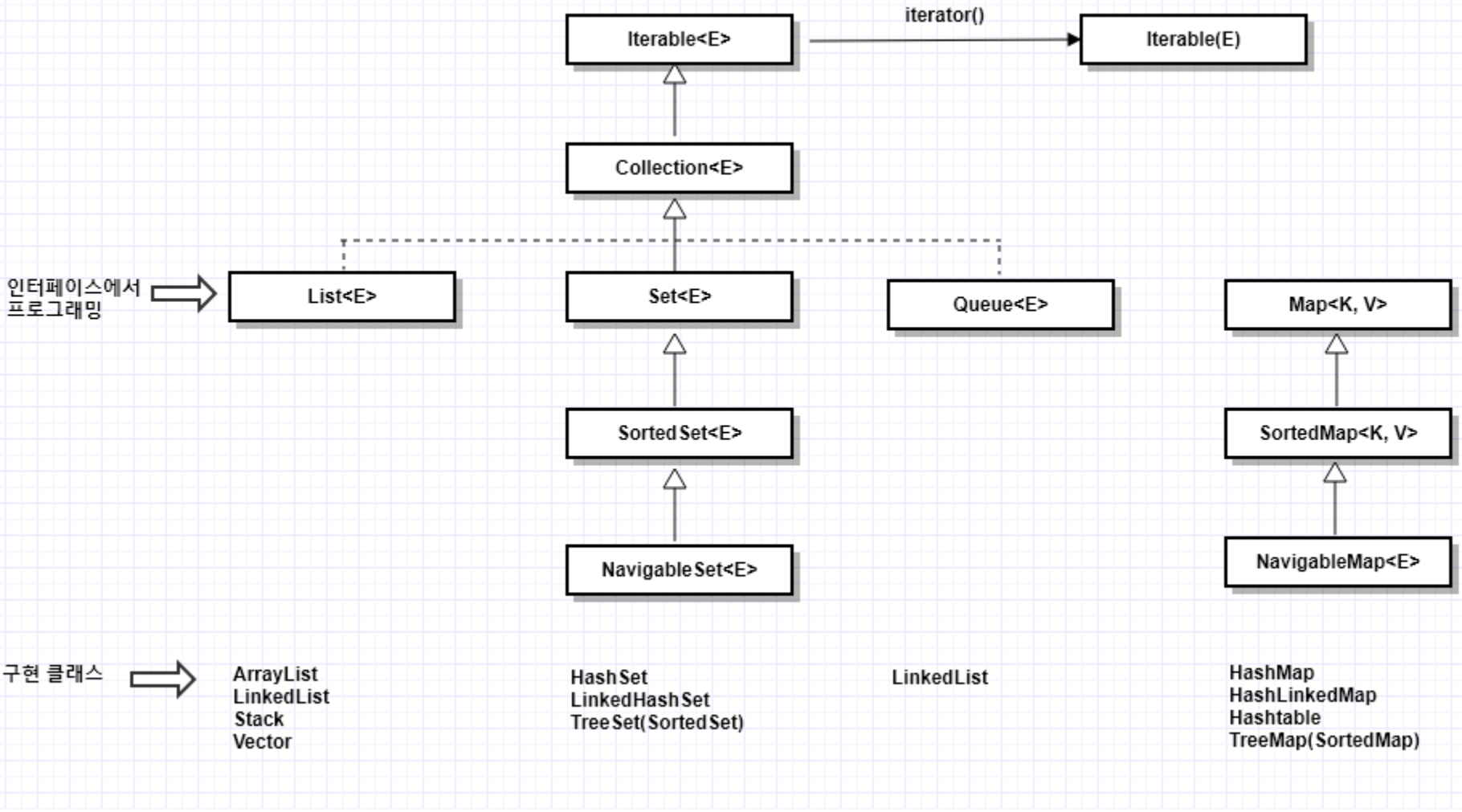


- 인터페이스: 컬렉션에서 공통으로 제공해야 하는 연산들을 추상적으로 정의하고, 구체적인 구현은 구현 클래스에 위임한다.
- 구현 클래스: 컬렉션을 실제로는 어떤 자료구조를 적용해 구현하느냐에 따라 컬렉션의 종류가 달라진다.
- 알고리즘: 컬렉션마다 유용하게 사용할 수 있는 메서드를 의미한다.

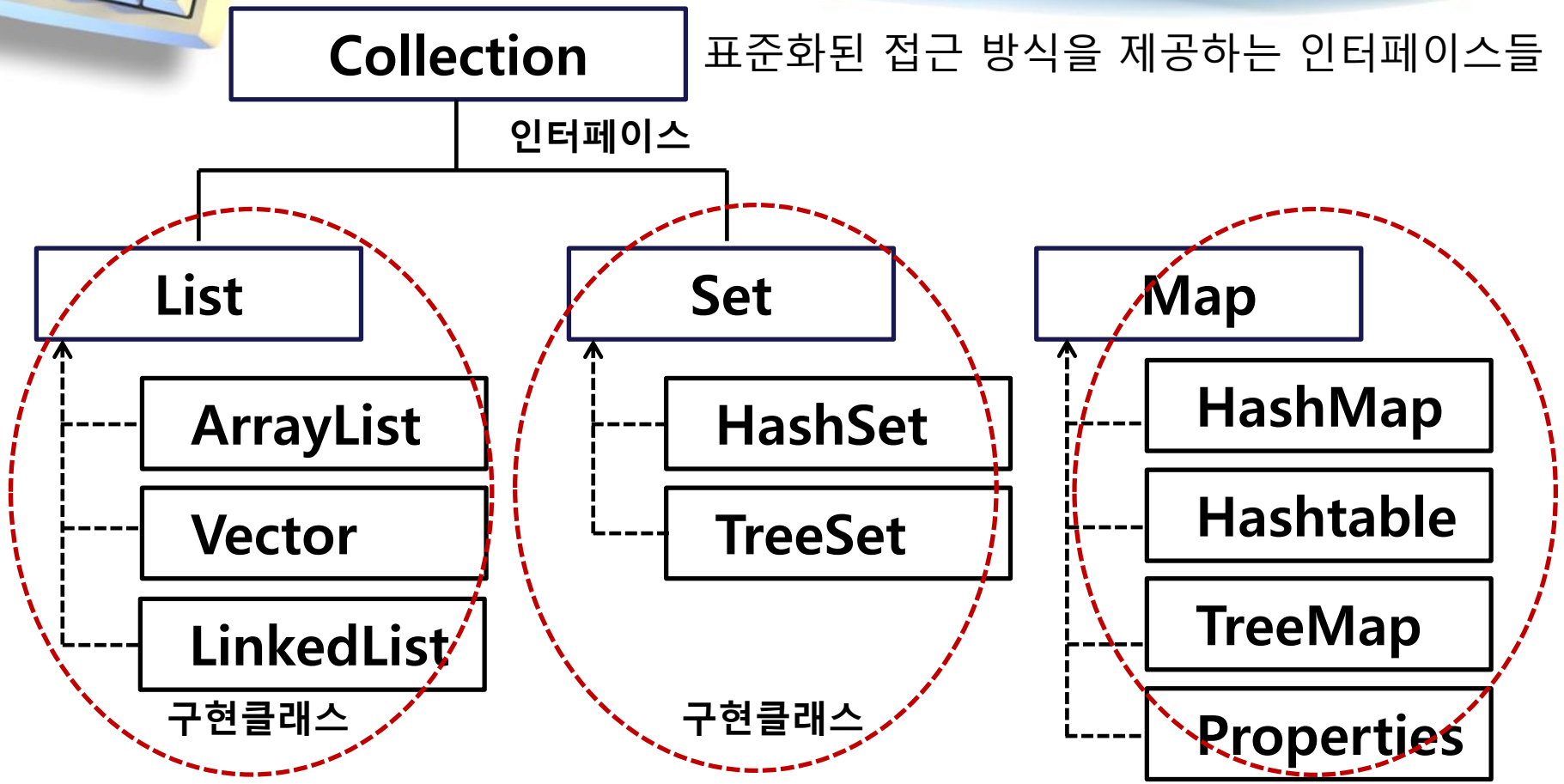
컬렉션 인터페이스

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

컬렉션 인터페이스


Iterable<E>
iterator()
Iterable(E)
Collection<E>
인터페이스에서
프로그래밍
List<E>
Set<E>
Queue<E>
Map<K, V>
SortedSet<E>
SortedMap<K, V>
NavigableSet<E>
NavigableMap<E>
구현 클래스
ArrayList
LinkedList
Stack
Vector
HashSet
LinkedHashSet
TreeSet(SortedSet)
LinkedList
HashMap
HashLinkedMap
Hashtable
TreeMap(SortedMap)

컬렉션 인터페이스



알고리즘: 정렬이나 탐색과 같은 알고리즘을 제공(메서드로 제공)

컬렉션 인터페이스

Collection은 거의 모든 컬렉션 인터페이스의 부모 인터페이스에 해당한다. 모든 컬렉션 클래스들이 Collection 인터페이스를 구현하고 있기 때문에 Collection에 들어 있는 메소드들은 거의 대부분 컬렉션 클래스에서 사용할 수 있다.

기능	메서드	설명
객체 추가	boolean add(E e)	주어진 객체 맨 끝에 추가
	void add(int index, E element)	주어진 인덱스에 객체를 추가
	E set(int index, E element)	주어진 인덱스에 주어진 객체로 변경

인터페이스가 제공하는 메소드

기능	메서드	설명
객체 검색	boolean contains(Object o)	주어진 객체가 저장되어 있는지 여부
	E get(int index)	주어진 인덱스에 저장된 객체를 리턴
	boolean isEmpty()	컬렉션이 비어 있는지 여부
	int size()	저장되어 있는 전체 객체 수를 리턴

인터페이스가 제공하는 메소드

기능	메서드	설명
객체 삭제	void clear()	저장된 모든 객체를 삭제
	E remove(int index)	주어진 인덱스에 저장된 객체를 삭제
	boolean remove(Object o)	주어진 객체를 삭제

List 인터페이스

리스트(List)는 순서를 가지는 원소들의 모임으로 중복된 원소를 가질 수 있다. 리스트의 가장 큰 특징은 위치를 사용하여 원소에 접근한다는 점이다.

자바에서 리스트는 인터페이스인 List에 의하여 정의된다. 인터페이스 List는 ArrayList, LinkedList, Vector 등의 클래스에 의하여 구현된다. ArrayList와 Vector는 배열처럼 데이터가 저장될 때마다 인덱스가 부여되며, LinkedList는 데이터가 저장될 때 이전에 저장된 데이터와 이후에 저장된 데이터의 정보를 포함한다.

List 컬렉션으로 구현된 클래스는 객체 자체를 저장하고 있는 것이 아니라 객체의 참조값를 참조하게 된다.

ArrayList 클래스

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

Java™ Platform Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.util

Class ArrayList<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

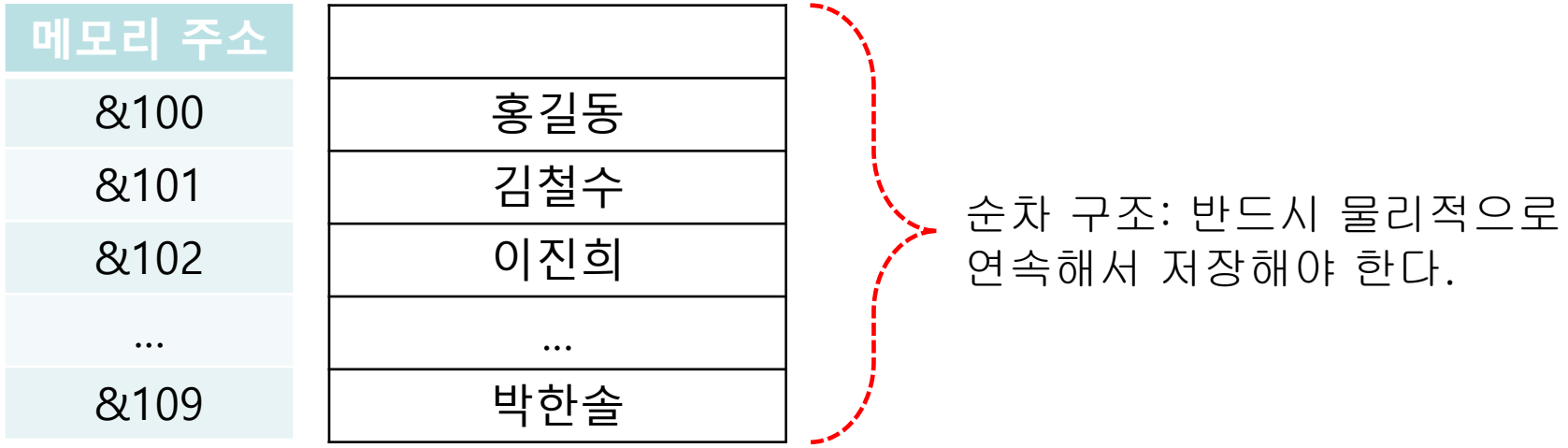
Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

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

ArrayList 클래스

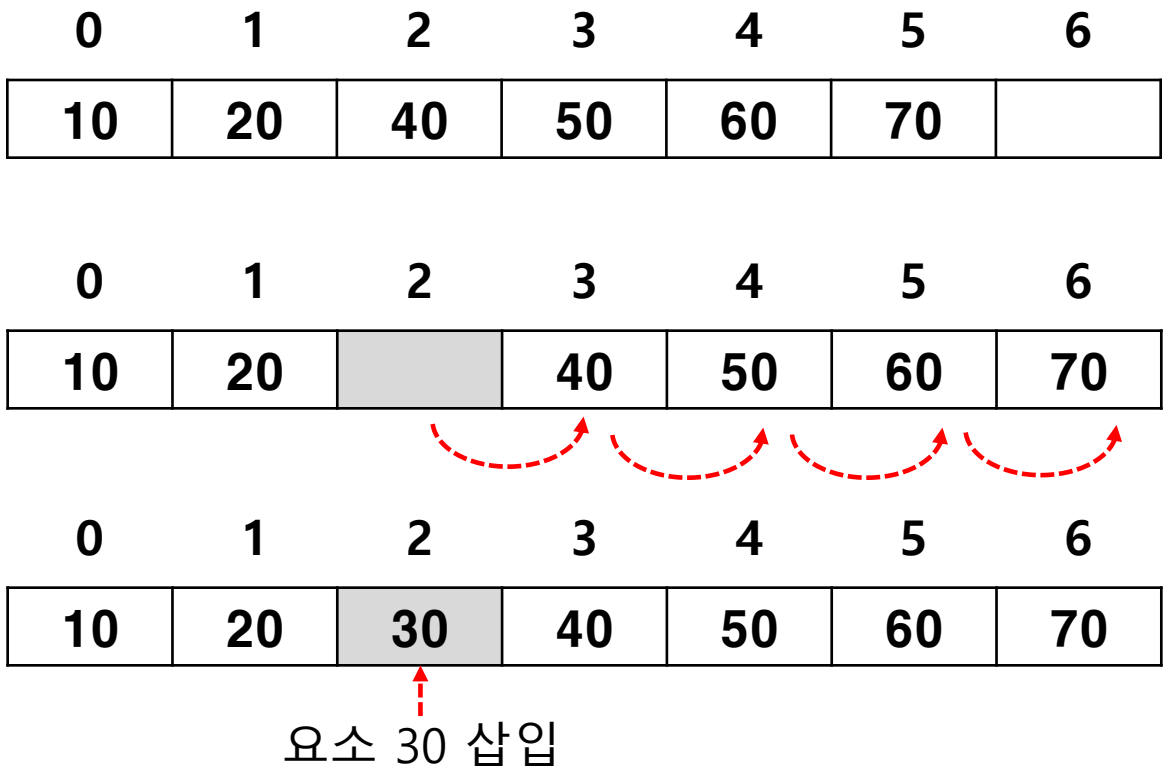
ArrayList 클래스는 List 인터페이스의 구현 클래스이다. ArrayList 클래스는 선형 구조 배열을 이용해 자료를 관리한다. 선형 구조는 데이터를 구조화하는 기본 표현 방식으로 순차 구조방식(ArrayList)와 연결 구조 방식(LinkedList)이 있다. **항목 간의 논리적인 순서와 메모리에 저장되는 물리적 순서가 같은 구조를 순차 구조**라 하며, 이는 배열의 원리가 동일하다. (ArrayList를 배열을 기반으로 하며 내부적으로 배열을 이용해서 인스턴스의 참조값을 저장한다.)



ArrayList 클래스

기본 생성자로 ArrayList 객체를 생성하면 내부에 10개의 객체를 저장할 수 있는 초기 용량(capacity)을 가지게 된다. 저장되는 객체 수가 늘어나면 용량이 자동으로 증가한다.

ArrayList에 객체를 추가하면 인덱스 0부터 차례대로 저장된다.



ArrayList 클래스

- ArrayList 인스턴스 생성
  ```
  ArrayList<String> list = new ArrayList<String>();
  ```

- 원소 추가, 변경, 삭제
  - list.add("JDBC");
  - list.add("Servlet/JSP");
  - list.add("Database");
  - list.set(2, "iBATIS");
  - list.remove(1);

- 원소 접근(모든 요소에 접근은 반복문을 이용하여 처리)
  ```
  String s = list.get(0);
  ```

ArrayList 클래스

크기가 10인 배열이 생성된다. 그러나 이 배열은 ArrayList 객체이므로 일반 배열과는 다르게 크기를 변경할 수 있다.

0	1	2	3	4	5	6	7	8	9
JDBC	Servlet/JSP	Database							

0	1	2	3	4	5	6	7	8	9
JDBC	Servlet/JSP	iBATIS							

0	1	2	3	4	5	6	7	8	9
JDBC	iBATIS								

ArrayList 클래스

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(new Integer(23));
list.add(new Integer(94));
list.add(new Integer(62));
list.add(new Integer(45));

System.out.println("--------Before------");
System.out.println(list.toString());
System.out.println("--------After--------");
list.add(2, new Integer(78));
System.out.println(list.toString());
```

```
--------Before--------
[23, 94, 62, 45]
--------After--------
[23, 94, 78, 62, 45]
```

- 새로운 값이 중간에 삽입되면
  - 메모리 재할당
  - 기존값을 새롭게 할당된 메모리로 복사해 이동.

ArrayList 클래스

컬렉션 라이브러리가 좋은 점은 다양한 연산을 제공한다는 점이다.

indexOf()를 사용하면 특정한 데이터가 저장된 위치를 알 수 있다. 다만 ArrayList는 동일한 데이터도 여러 번 저장될 수 있으므로 맨 처음에 있는 데이터의 위치가 반환된다.

```
int index = list.indexOf("Servlet/JSP");
```

검색을 반대 방향으로 하려면 lastIndexOf()를 사용한다.

```
int index = list.lastIndexOf("Servlet/JSP");
```

배열 : 배열명.length
문자열 : 문자열참조변수 또는 "문자".length()
리스트 구현 클래스 : 리스트명.size()

반복자(Iterator)

ArrayList에 있는 원소에 접근하는 또 하나의 방법은 반복자(iterator)를 사용하는 것이다. 반복자는 특별한 타입의 객체로 컬렉션의 원소들에 접근하는 것이 목적이다.

(객체 집합의 항목들에 차례대로 접근하는데 사용하는 객체이다)

Iterator는 컬렉션에 저장된 데이터의 위치 정보를 포함한 커서가 있어서 인덱스 등을 사용하지 않고 쉽게 데이터에 접근할 수 있다. 이처럼 컬렉션에 저장된 모든 데이터를 순차적으로 접근하여 사용할 목적으로 사용하는 Iterator를 '컬렉션 뷰(Collection view)'라고 한다.

ArrayList 뿐만 아니라 반복자는 모든 컬렉션 클래스에 적용할 수 있다. 반복자는 java.util 패키지에 정의되어 있는 Iterator 인터페이스를 구현해서 사용해야 하는 객체이다.

반복자(Iterator)

Iterator인터페이스에는 다음 3개의 메소드만 정의되어 있다

메소드	설명
hasNext()	참조할 다음 요소(항목)가 존재하면 true를 반환
next()	다음 요소(항목)를 반환
remove()	현재 위치의 요소(항목)를 삭제

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    default void remove() {
  throw new UnsupportedOperationException("remove");
    }
    ...
}
```

반복자(Iterator)

반복자를 사용하기 위해서는 먼저 ArrayList의 iterator()메소드를 호출하여서 반복자 객체를 얻는다.

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>...{ ...
   public Iterator<E> iterator() {
        return new Itr();
    }
   private class Itr implements Iterator<E> {
        ...
        public boolean hasNext() { ... }
        @SuppressWarnings("unchecked")
        public E next() { ... }
...
    }
```

반복자(Iterator)

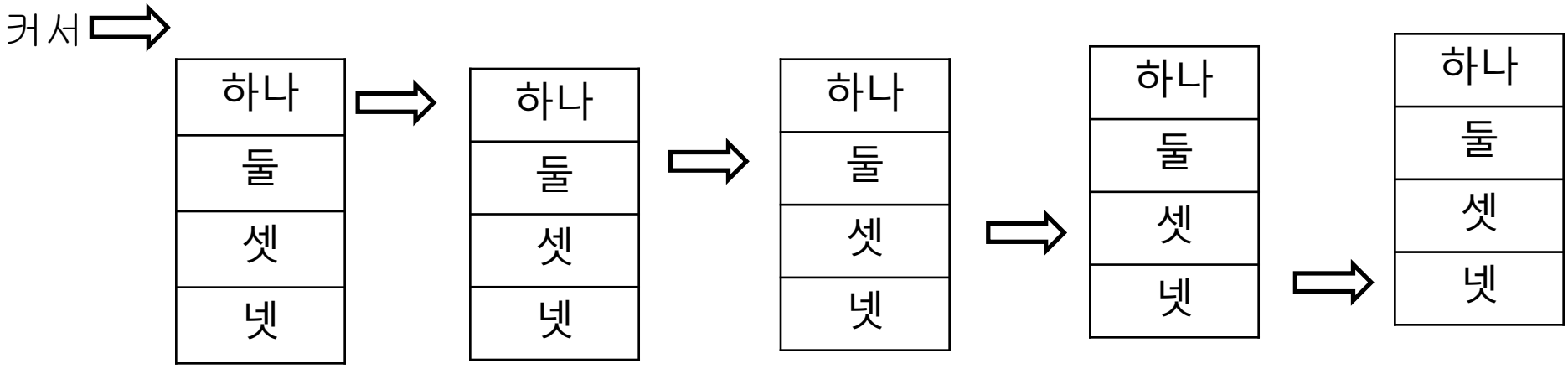
반복자를 사용하기 위해서는 먼저 ArrayList의 iterator()메소드를 호출하여서 반복자 객체를 얻는다. 다음으로 반복자 객체의 hasNext()와 next()메소드를 이용하여서 컬렉션의 각 원소들에 접근하게 된다.

```
ArrayList<String> list = new ArrayList<String>();
list.add("하나");
list.add("둘");
list.add("셋");
list.add("넷");
```

반복자(Iterator)

```
String s;
Iterator<String> e = list.iterator();
while (e.hasNext()){ // 현재 커서 다음에 데이터가 존재하는지 판단
    s = e.next(); // 커서 다음의 데이터를 반환하고 다음 데이터로 이동
    System.out.println(s);
}
```

iterator()메소드가 호출될 때 생성되는 인스턴스를 가리켜 반복자라하고 컬렉션 인스턴스에 저장되어 있는 데이터들을 순차적으로 참조하는데 사용되기 때문이다.



LinkedList 클래스

compact1, compact2, compact3

java.util

Class LinkedList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.AbstractSequentialList<E>
        java.util.LinkedList<E>

Type Parameters:

`E - the type of elements held in this collection`

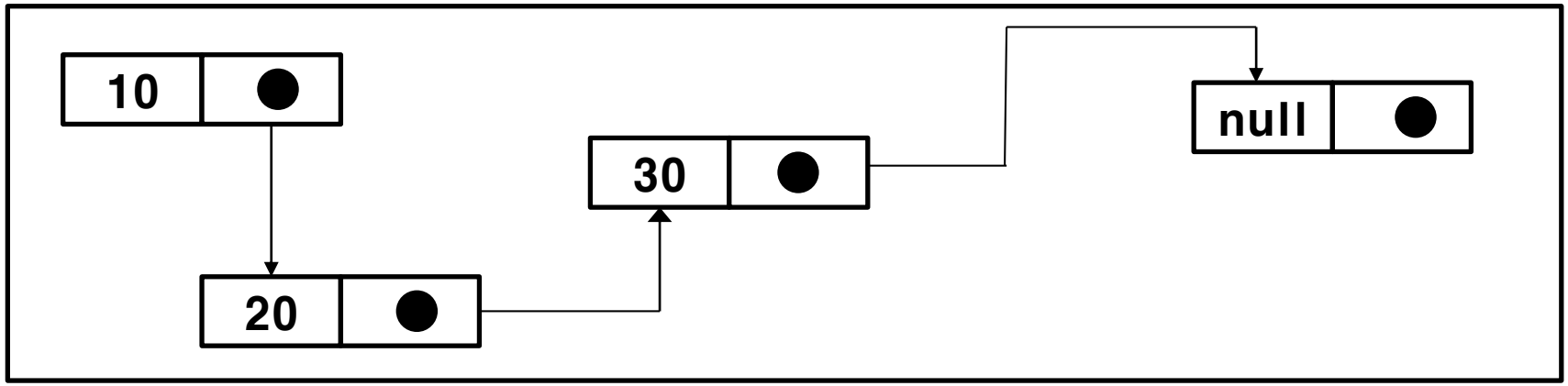
All Implemented Interfaces:

`Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>`

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

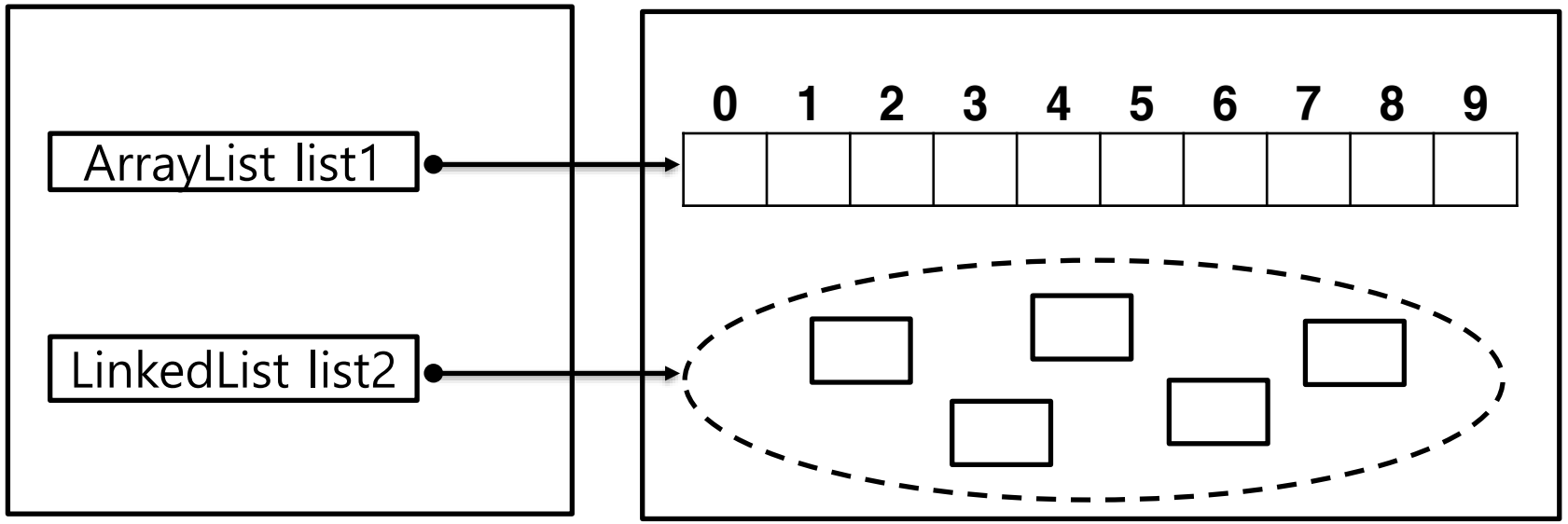
LinkedList 클래스

ArrayList 클래스는 논리적인 순서와 물리적인 순서가 같은 순차 구조라 항목의 위치를 찾아 접근하기 쉽다는 장점은 있다. 하지만 삽입이나 삭제 후에 연속적인 물리 주소를 유지하기 위해서 항목을 이동해야 하는 작업으로 시간이 추가로 필요하다. **삽입/삭제 연산이 많으면 항목이 이동이 많아져서 많은 부하가 생긴다는 단점과 검색속도가 빠르다는 장점이 있다.** 순차 구조 방식에서 연산 지연에 대한 문제와 저장 영역에 대한 문제를 개선한 자료구조 방식이 연결 구조 방식(LinkedList)이다.



LinkedList 클래스

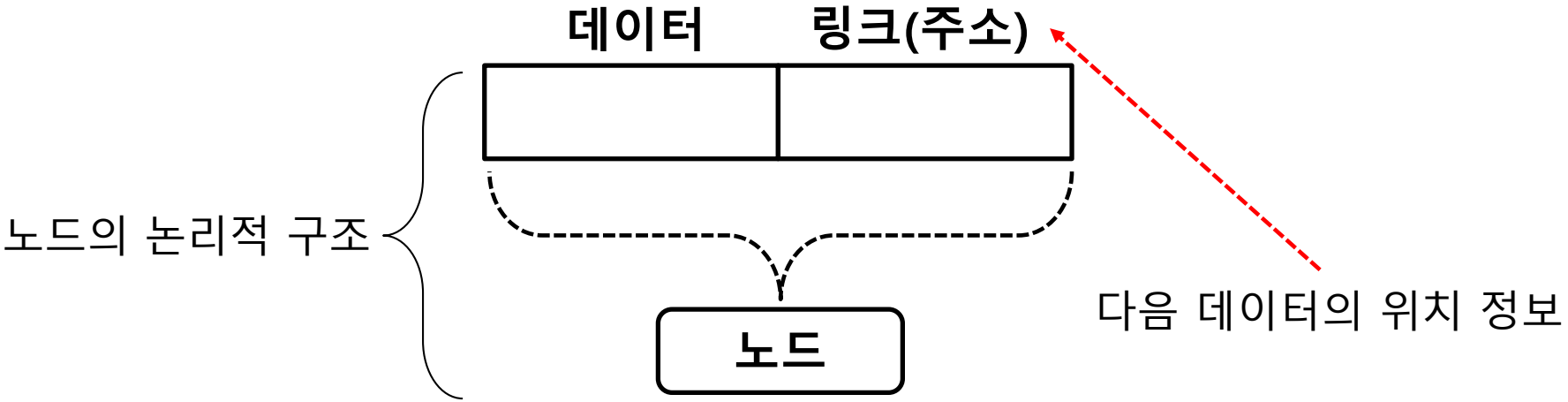
LinkedList는 ArrayList처럼 메모리에 순서대로 데이터를 저장하는 것이 아니라 메모리의 저장할 수 있는 공간이 있다면 위치에 상관없이 저장한다. 즉 ArrayList의 데이터는 연속적으로 저장되지만 LinkedList는 연속적으로 저장되지 않고 임의의 위치에 저장된다. 메모리 효율면에서는 연속적인 공간이 필요없는 LinkedList가 더 유리하고 데이터 검색면에서는 차례대로 접근할 수 있는 ArrayList가 유리하다.



LinkedList 클래스

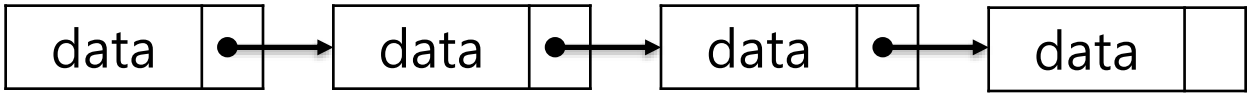
LinkedList 방식에서는 순차 구조 방식에서처럼 항목의 논리적인 순서와 물리적인 순서가 일치하지 않아도 된다.

단 물리적으로 연속적이지 않은 항목을 논리적 순서대로 연결해야 하므로 하나의 항목은 다음 항목에 대한 주소값을 가지고 있어야 한다. 이러한 단위 구조를 **노드(node)**라 한다.

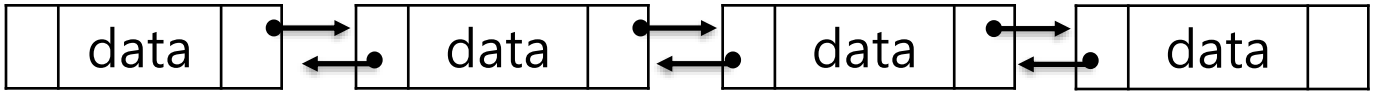


LinkedList 클래스

LinkedList는 단일 링크드리스트(Singly LinkedList)와 이중 링크드리스트(Doubly LinkedList) 두 가지 종류가 있으며 단일 링크드리스트(Singly LinkedList)는 노드가 데이터와 다음 데이터의 정보만 가지는 형태이다.



이중 링크드리스트(Doubly LinkedList)는 노드에서 다음 데이터뿐만 아니라 이전 데이터의 위치 정보도 가지는 구조이다.

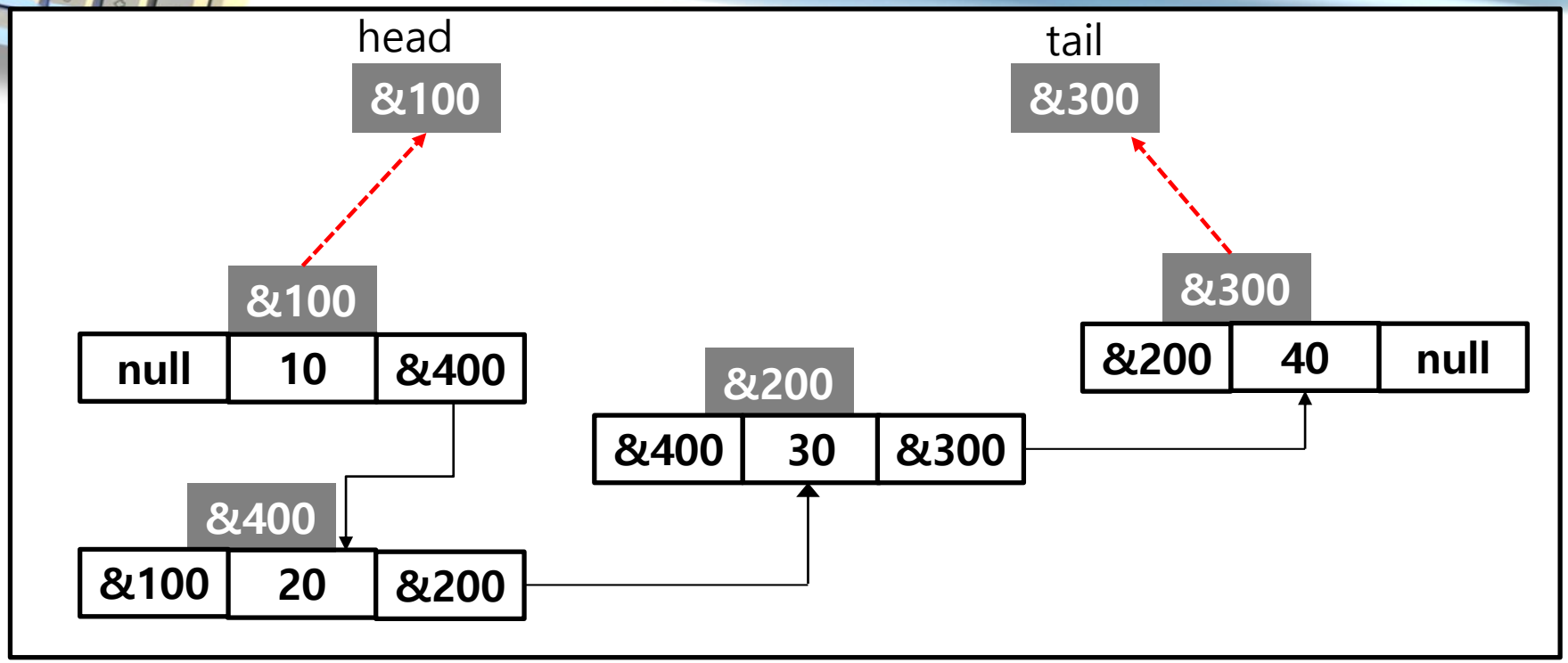


LinkedList 클래스

단일 링크드리스트(Singly LinkedList)는 노드가 다음 데이터의 정보만 가지고 있어서 한 방향으로만 탐색되고, 이중 링크드리스트(Doubly LinkedList)는 노드가 이전 데이터와 다음 데이터의 정보를 가지고 있어서 양방향 탐색을 할 수 있다. 자바에서 사용하는 LinkedList는 이중 링크드리스트이다.

링크드리스트에서 탐색 작업을 위해서는 첫 노드와 마지막 노드의 위치 정보가 필요한데 처음 생성된 노드는 head라고 부르고 가장 마지막에 생성된 노드를 tail라고 부른다.

LinkedList 클래스



참조에 의한 논리적인 연결 구조를 이용하는 LinkedList 클래스는 ArrayList 클래스와 달리, 중간 항목의 삽입과 삭제가 쉽다. 삽입이나 삭제는 위치의 바로 앞에 있는 항목의 링크 값만 변경하면 된다.

하지만 LinkedList 클래스는 위치(인덱스)를 이용해 항목에 접근하는 경우 ArrayList 클래스보다 탐색시간이 오래걸린다

LinkedList 클래스

```
import java.util.*;

public class LinkedListTest {
	public static void main(String args[]) {
		LinkedList<String> list = new LinkedList<String>();
		list.add("Galaxy S20");
		list.add("iPhone12");
		list.add("LG G8");

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

항목의 위치를 이용하는 접근하는 경우가 빈번한 경우에는 ArrayList 클래스를, 항목의 삽입과 삭제가 빈번한 경우에는 LinkedList를 사용한 것을 권장한다.

배열을 리스트로 변환하기

List<String> list = Arrays.asList(배열명 또는 배열의 참조값);

- 일반적인 배열을 리스트로 변환한다.

```
import java.util.List;

public class ArraysAsListExample {
        public static void main(String[] args) {
                String[] names = {"홍길동", "김철수", "조미진"};
                List<String> list = Arrays.asList(names);

                for(String name: list) {
                        System.out.println(name);
                 }
        }
}
```

Vector 클래스

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

Java™ Platform Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.util

Class Vector<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.Vector<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

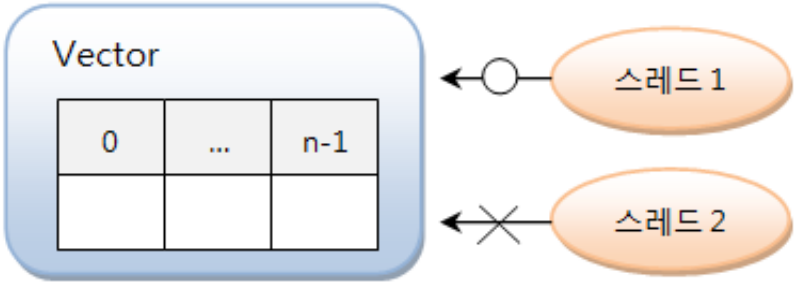
Stack

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

Vector 클래스

Vector와 ArrayList의 차이점은 Vector는 동시 사용을 안전하게 처리할 수 있도록 설계된 클래스이지만 ArrayList는 동시 사용을 허용하지 않는 클래스이다. 즉 Vector는 동기화 처리를 지원하고 ArrayList는 동기화 처리를 지원하지 않는다. 동기화 처리란 여러 곳에서 동시에 하나의 자원에 사용할 때 문제없이 동작하도록 하는 기능이다. 동기화가 필요한 상황에서는 Vector를, 동기화 처리가 필요없는 상황에서는 ArrayList를 사용하는 것이 효율적이다.

- Vector는 동기화(synchronization)를 지원
  - 복수의 스레드가 동시에 Vector에 접근해 객체를 추가, 삭제하더라도 스레드에 안전(thread safe)

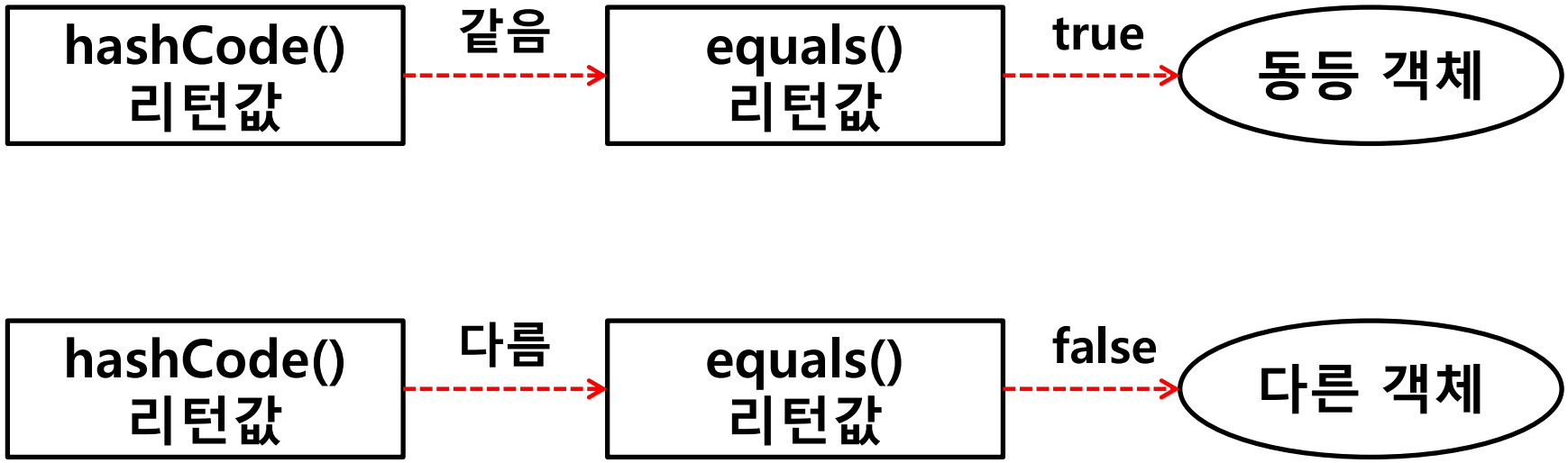


Set 인터페이스

순서에 관계(상관)없이 원소만 저장하고 싶은 경우에 사용할 수 있는 자료 구조가 집합(Set)이다. 집합(Set)은 동일한 객체를 중복해서 가실 수 없다. HashSet은 Set 인터페이스의 구현 클래스이다. 동일한 객체 판단 방법은 **hashCode()** 메서드를 호출해서 해시코드를 얻어낸다. 그리고 이미 저장되어 있는 객체들의 해시코드와 비교한다. 만약 동일한 해시코드가 있다면 다시 **equals()** 메서드로 두 객체를 비교해서 true가 나오면 동일한 객체로 판단하고 중복 저장을 하지 않는다.

- 객체 식별할 하나의 정수값
- 객체의 메모리 번지 이용해 해시코드 만들어 리턴

Set 인터페이스


hashCode()
리턴값
같음
equals()
리턴값
true
동등 객체
hashCode()
리턴값
다름
equals()
리턴값
false
다른 객체

HashSet 클래스

compact1, compact2, compact3

java.util

Class HashSet<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractSet<E>
      java.util.HashSet<E>

Type Parameters:

E - the type of elements maintained by this set

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, Set<E>

Direct Known Subclasses:

JobStateReasons, LinkedHashSet

```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

Set 인터페이스 구현 클래스

HashSet

- HashSet은 원소들의 순서가 일정하지 않다. 비정렬이면서 비순차적인 자료구조.

LinkedHashSet

- HashSet 클래스를 순차구조로 만든 클래스로 모든 항목에 대해 이중 연결 리스트 구조를 갖는다. 순차적이면서 반복 처리인 경우에 객체를 저장하는 순서가 중요하면 이 클래스를 사용하면 된다. (원소들의 순서는 삽입되었던 순서와 같다.)

TreeSet

- HashSet 클래스는 값에 따라서 순서가 결정되며 하지만 HashSet보다는 느리다.

예제

```
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);
  }
}
```

```
HashSet : [Bread, Milk, Butter, Ham, Cheese]
LinkedHashSet(입력된 순서대로 출력) :
            [Milk, Bread, Butter, Cheese, Ham]
TreeSet(알파벳 순으로 정렬) :
            [Bread,  Butter, Cheese,  Ham, Milk]
```

Map 인터페이스

맵(Map, 사전과 같은 자료구조)은 많은 데이터 중에서 원하는 데이터를 빠르게 찾을 수 있는 자료 구조이다. 즉 사전처럼 단어가 있고(키(key)) 단어에 대한 설명(값(value))이 있다. 이와 같이 Map 인터페이스는 키(key)와 값(value)의 한 쌍으로 데이터를 저장하며, 키와 값 모두 객체인 참조형만 가능하다. 즉 키를 값에 매핑하는 자료구조이다.

Map은 중복된 키를 가질 수 없다. 각 키는 오직 하나의 값에만 매핑 될 수 있다. 즉 키 하나에 값 하나가 대응하는 자료구조이다. 키가 제시되면 값을 반환한다.

자바에서는 Map이라는 이름의 인터페이스가 제공되고 이 인터페이스를 구현한 클래스로 HashMap, TreeMap, LinkedHashMap 3가지의 클래스가 제공된다.

Map 인터페이스

Map 인터페이스에는 다음과 같이 정의되어 있다

```
public interface Map<K,V> {
        int size();
        V get(Object key);
        V put(K key, V value);
        Set<K> keySet();
        Set<Map.Entry<K, V>> entrySet();
        interface Entry<K,V> {
                K getKey();
                V getValue();
                V setValue(V value);
                boolean equals(Object o);
                int hashCode();
        }
        boolean equals(Object o);
}
```

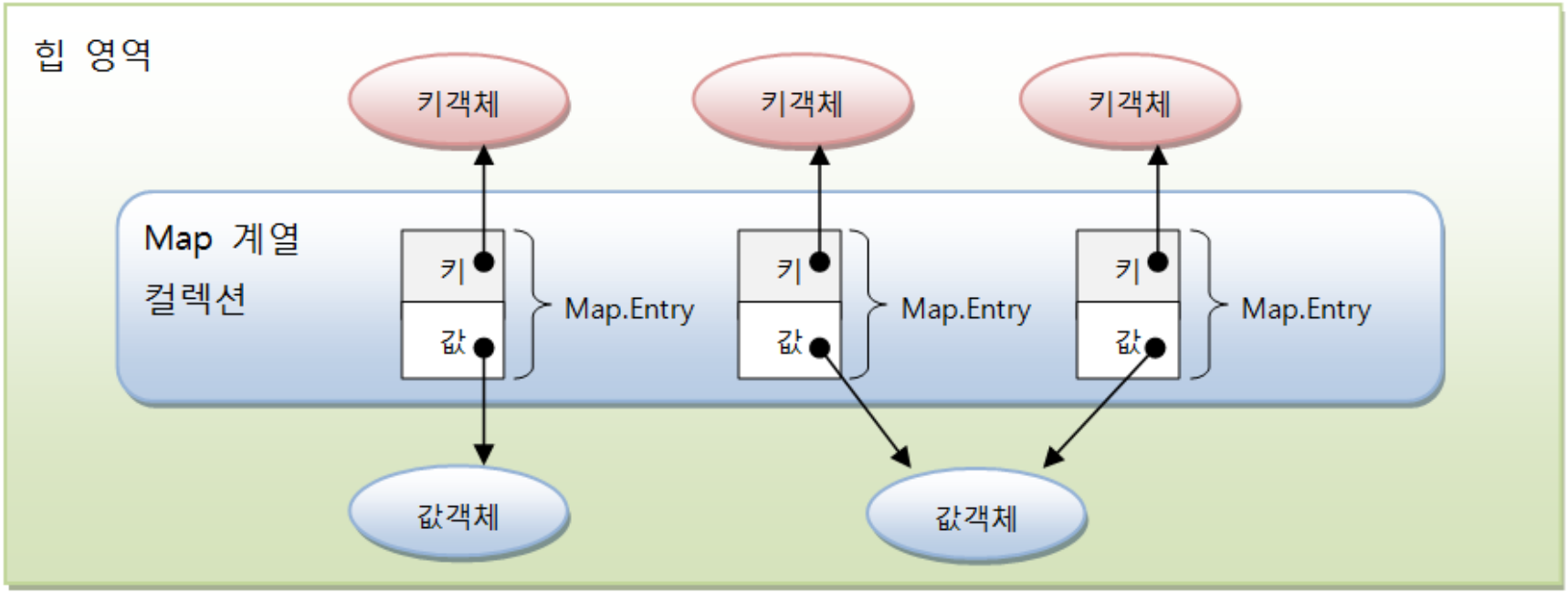
Entry 인터페이스

Entry는 Map 인터페이스에 선언되 내부 인터페이스이다. 내부 클래스와 인터페이스는 외부객체명.내부객체명으로 표현한다. 따라서 Map 인터페이스의 내부 인터페이스로 선언된 Entry는 Map.Entry 로 표현해야 한다.

Map은 하나의 데이터 키와 값으로 구성되어 있다. Entry는 Map에 저장되는 데이터 단위인 키와 값을 가지는 객체이다. 그래서 **Map은 여러 개의 Entry로 구성된 컬렉션이다.**

Map 인터페이스

- 특징
  - 키(key)와 값(value)으로 구성된 Map.Entry 객체를 저장하는 구조
  - 키와 값은 모두 객체
  - 키는 중복될 수 없지만 값은 중복 저장 가능



Map 컬렉션의 주요 메소드

기능	메서드	설명
객체 추가	V put(K key, V value)	주어진 키와 값을 추가. 저장이 되면 값을 리턴
객체 검색	boolean containsKey (Object key)	주어진 키가 있는지 여부
	boolean containsValue (Object value)	주어진 값이 있는지 여부

Map 컬렉션의 주요 메소드

기능	메서드	설명
객체 검색	V get(Object key)	주어진 키의 값을 리턴
	Set<Map.Entry<K,V>> entrySet()	키와 값의 한 쌍으로 구성된 객체를 Set에 담아서 리턴
	Set<K> keySet()	모든 키를 Set 객체 담아서 리턴
	Collection<V> values()	저장된 모든 값을 collection에 담아서 리턴

HashMap 구현 클래스

compact1, compact2, compact3

java.util

Class HashMap<K,V>

java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.HashMap<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Implemented Interfaces:

Serializable, Cloneable, Map<K,V>

Direct Known Subclasses:

LinkedHashMap, PrinterStateReasons

---

public class **HashMap<K,V>**
extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable

HashMap 구현 클래스

HashMap 구현 클래스는 다음과 같이 정의되어 있다

```
public class HashMap<K,V> extends AbstractMap<K,V>
                    implements Map<K,V>, Cloneable,
Serializable {
    ...
    static class Node<K,V> implements Map.Entry<K,V> {
            final int hash;
            final K key;
            V value;
           ...
            public final K getKey()         { return key; }
            public final V getValue()       { return value; }
            public final String toString() {
                    return key + "=" + value; }
        ...
    }
```

HashMap 구현 클래스

Map 인터페이스에는 다음과 같이 정의되어 있다

```
    public Set<Map.Entry<K,V>> entrySet() {
        Set<Map.Entry<K,V>> es;
        return (es = entrySet) == null ? (entrySet=new EntrySet()):es;
    }

    final class EntrySet extends AbstractSet<Map.Entry<K,V>> {
        public final int size()                 { return size; }
        public final void clear()               { HashMap.this.clear(); }
        public final Iterator<Map.Entry<K,V>> iterator() {
            return new EntryIterator();
        }
          ...
    }
}
```

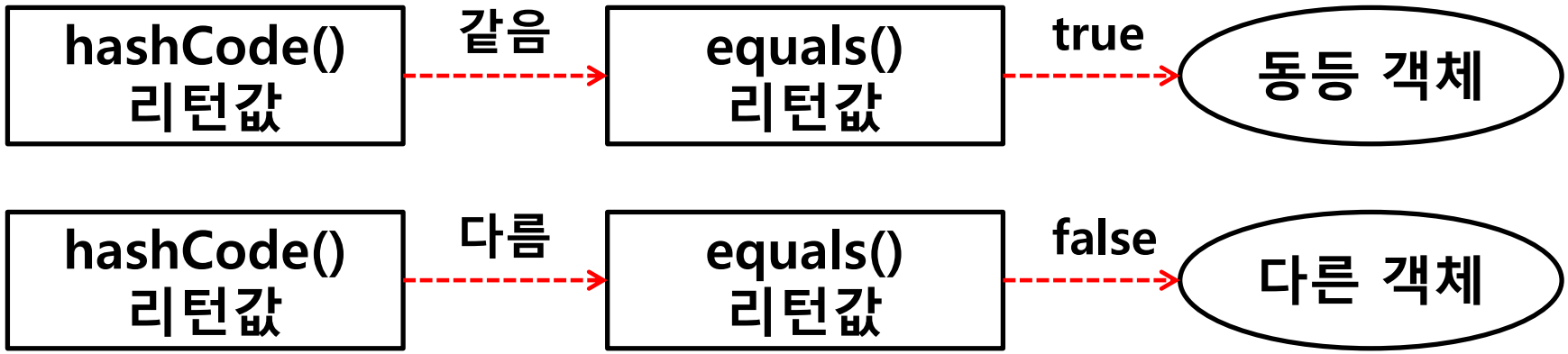
HashMap 구현 클래스

- 특징
  - 키 타입은 String 많이 사용
    - String은 문자열이 같을 경우 동등 객체가 될 수 있도록
    - hashCode()와 equals() 메소드가 재정의되어 있기 때문

```
Map<K, V> map = new HashMap<K, V>();
```

키타입 값타입 키타입 값타입

- 키 객체는 hashCode()와 equals() 를 재정의해 동등 객체가 될 조건을 정해 주면 된다.



Map 인터페이스 구현 클래스

HashMap

- HashMap은 비정렬이면서 비순차적인 맵구조. HashMap 클래스의 키는 키가 되는 객체의 해시코드를 기반으로 한다.

LinkedHashMap

- LinkedHashSet 클래스처럼 객체가 추가되는 순서를 유지한다. 항목의 추가나 삭제에는 HashMap 클래스보다 느리지만, 순차적이면서 반복 처리인 경우 더 빠르다.

TreeMap

- TreeMap 클래스는 정렬 컬렉션 중 하나이다.

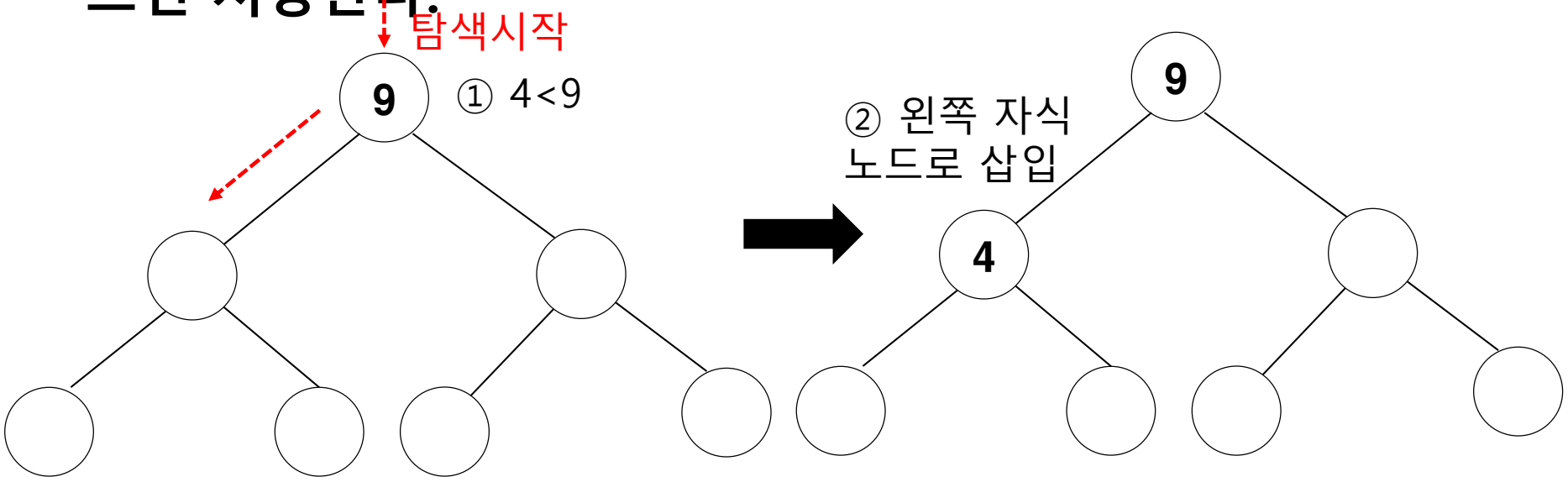
TreeMap 구현 클래스

ArrayList는 메모리의 연속된 위치에 데이터를 순차적으로 저장하고, LinkedList는 메모리에 연속적으로 저장하지 않으며 값과 위치 정보를 가진 노드 형태로 저장한다.

TreeMap은 데이터가 트리 형태로 저장되는 특징이 있다. 트리 형태는 데이터가 저장될 때 기존의 데이터와 새롭게 저장되는 데이터를 비교하여 저장되는 위치를 결정한다. 트리 형태로 저장되는 데이터 역시 '노드'라고 부른다.

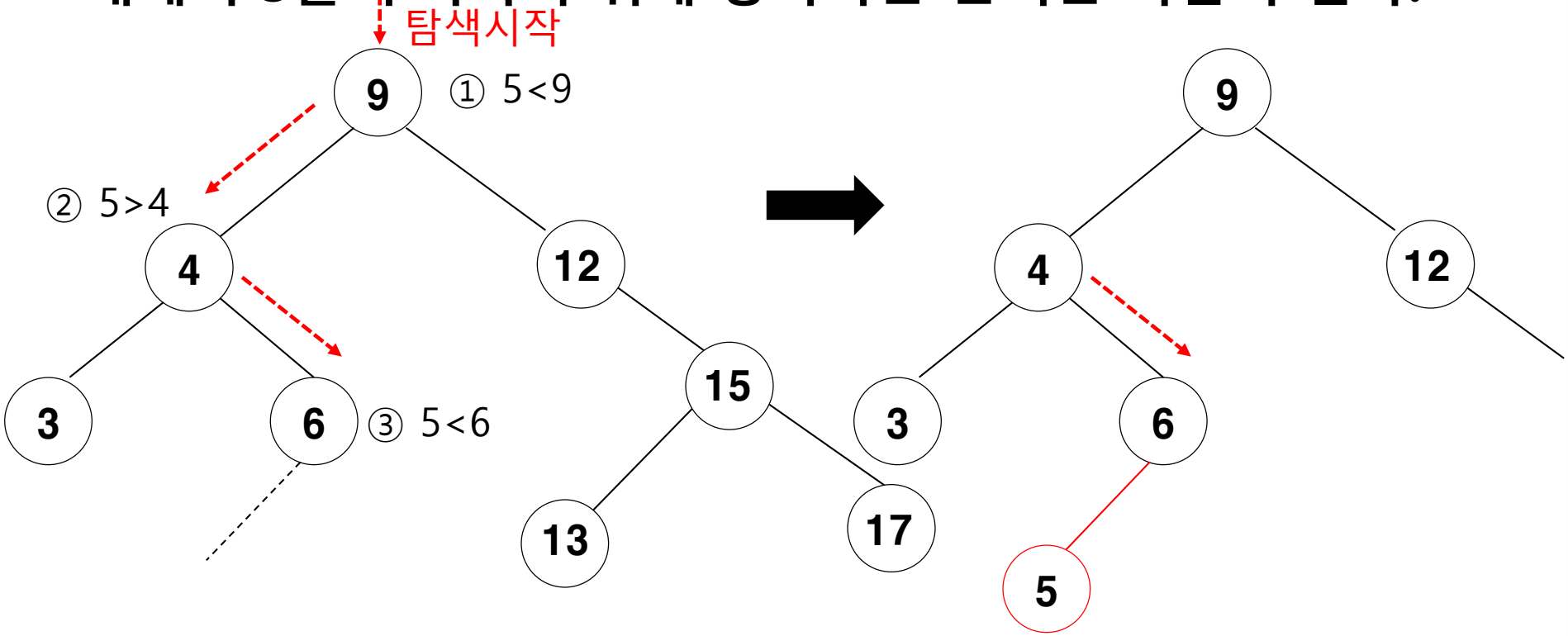
트리 구조

가장 첫번째 위치를 루트(root)라고 하며, 여기서부터 데이터의 저장 또는 탐색 작업을 시작한다. 만약 트리 구조의 컬렉션에 9가 저장된 상태에서 4를 저장하려고 하면 새로 저장할 데이터(4)를 기존의 데이터(9)와 비교해 작으면 왼쪽, 크면 오른쪽으로 이동한다. 이동한 자리에 데이터가 없으면 저장한다.



TreeMap 구현 클래스

트리 구조에 다음과 같은 데이터들이 저장되었다면 이 상태에서 5를 추가하기 위해 동작하는 순서는 다음과 같다.



트리 구조에서 데이터 추가 과정

트리 구조에서 데이터 추가 과정

① 9와 5를 비교한다. 5가 작으니 왼쪽을 선택한다.

② 선택된 위치의 4와 5를 비교한다. 5가 크므로 오른쪽을 선택한다

③ 선택된 위치의 6과 5를 비교한다. 5가 작으므로 왼쪽을 선택한다

④ 선택된 위치에 데이터가 없다. 5를 저장하고 추가 작업을 완료한다

트리 구조는 이와 같은 방식으로 데이터를 저장하므로 정렬이 자동으로 이루어진다. TreeMap은 트리와 맵의 특징이 모두 있다. 즉 데이터가 저장될 때는 키와 값으로 구성된 Entry로 저장되며 저장되는 구조는 트리 형태이다.

Comparable과 Comparator

- TreeSet과 TreeMap의 자동 정렬
  - TreeSet의 객체와 TreeMap의 키는 저장과 동시에 자동 오름차순 정렬
  - 숫자(Integer, Double)타입일 경우에는 값으로 정렬
  - 문자열(String) 타입일 경우에는 유니코드로 정렬

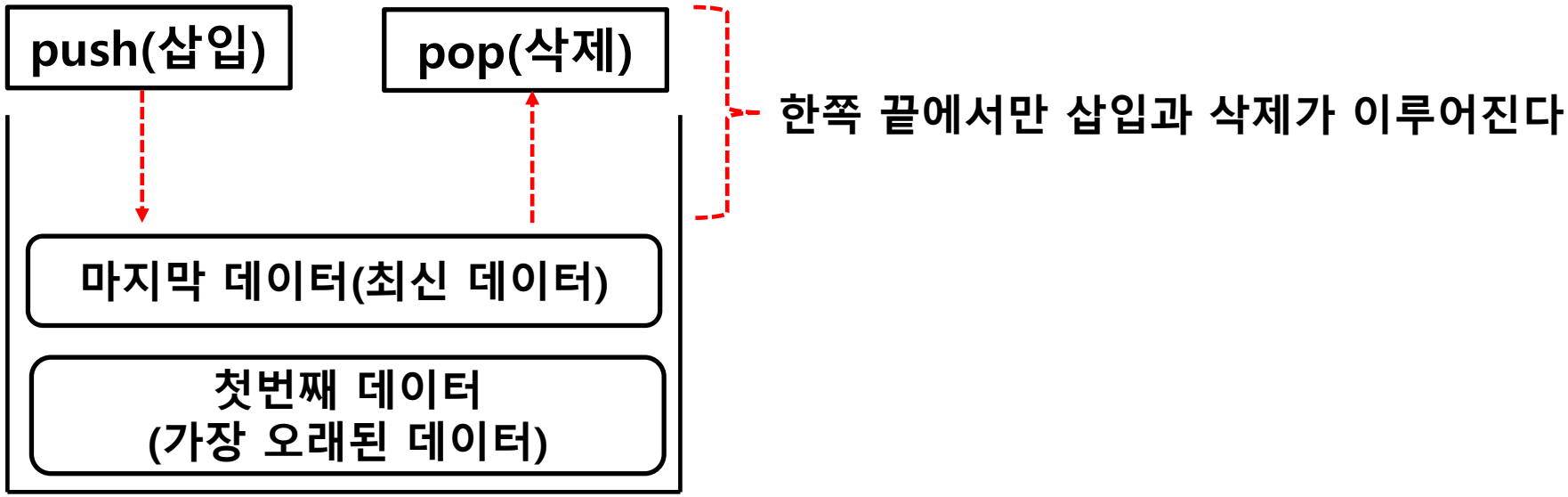
- TreeSet과 TreeMap은 정렬 위해 java.lang.Comparable을 구현 객체를 요구
  - Integer, Double, String은 모두 Comparable 인터페이스 구현
  - Comparable을 구현하고 있지 않을 경우에는 저장하는 순간 ClassCastException 발생

Stack 클래스

- Stack 클래스

한쪽에서 차곡차곡 쌓아 올리는 형태의 자료구조를 말한다. 스택에서 연산은 삽입과 삭제가 가능하며, 한쪽 끝에서만 이루어진다. 스택은 후입선출(LIFO: Last In First Out) 구조이다.

```
Stack<E> stack = new Stack<E>();
```



Stack 클래스

- Stack 클래스의 메서드

리턴타입	메서드	설명
E	push(E item)	주어진 객체를 스택에 저장(삽입).
E	peek()	스택의 맨위 객체를 가져온다. 객체를 스택에서 제거하지 않는다.
E	pop()	스택의 맨위 객체를 가져온다. 객체를 스택에서 제거한다.

Queue 인터페이스

- Queue 인터페이스

스택과 달리 리스트의 한쪽 끝에서는 삽입 작업이 이루어지고 반대쪽 끝에서는 삭제 작업이 이루어진다. 삽입된 순서대로 삭제되는 선입선출(FIFO: First In First Out)의 구조이다.

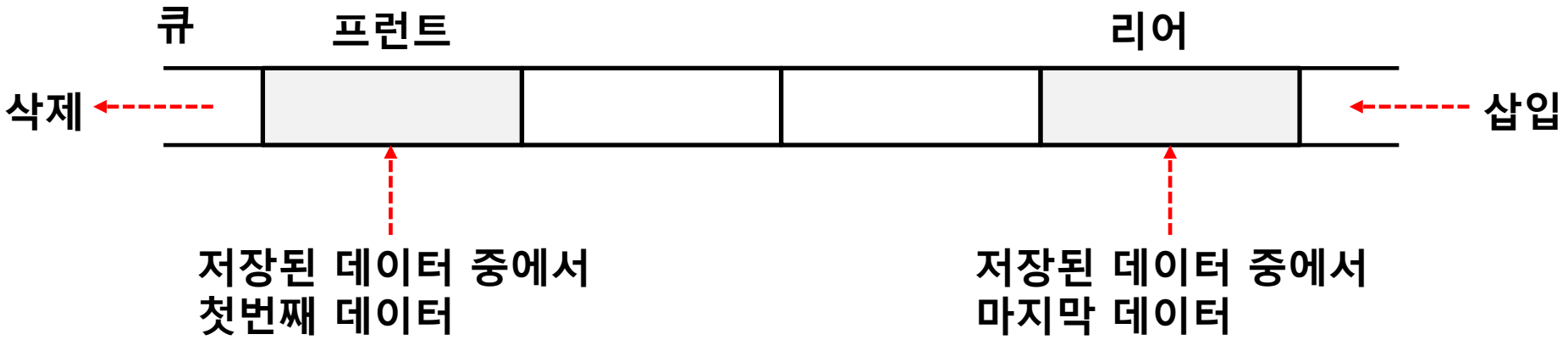
큐의 한쪽 끝은 프런트(front)로 정해 삭제 연산만 수행하도록 제한하고, 반대쪽 끝은 리어(rear)로 정해 삽입 연산만 수행하도록 제한한다. 큐의 리어에서 이루어지는 삽입 연산을 인큐(enqueuer)라고 하고, 프런트에서 이루어지는 삭제 연산을 디큐(dequeuer)라고 한다.

```
Queue<E> queue = new LinkedList<E>();
```

Queue 인터페이스

- Queue 인터페이스의 주요 메서드

리턴타입	메서드	설명
boolean	offer(E e)	주어진 객체를 큐에 저장(삽입).
E	peek()	객체 하나를 가져온다. 객체를 큐에서 제거하지 않는다.
E	poll()	객체 하나를 가져온다. 객체를 큐에서 제거한다.



구현 클래스의 특징

	순차(ordered)	정렬(sorted)
HashMap	안됨	안됨
TreeMap	정렬된 대로 됨	지정된 기준에 의해 정렬
LinkedHashMap	추가된 순서대로	안됨
HashSet	안됨	안됨
TreeSet	정렬된 대로 됨	지정된 기준에 의해 정렬
LinkedHashSet	추가된 순서대로	안됨
ArrayList	인덱스 기준	안됨
LinkedList	인덱스 기준	안됨

컬렉션 프레임워크의 특징

컬렉션 프레임워크	특징
ArrayList 클래스	배열 기반으로 데이터의 추가와 삭제에 불리하다. 순차적인 추가와 삭제는 제일 빠르다. 임의의 항목에 대한 접근성이 좋다.
LinkedList 클래스	연결 구조 기반으로 데이터의 추가와 삭제에 유리하다. 임의의 항목에 대한 접근성이 좋지 않다.
HashMap 클래스	배열 기반과 연결 구조 기반이 결합된 형태로 추가, 삭제, 검색, 접근성 모두 뛰어나다. 검색에는 최고 성능을 보인다.
TreeMap 클래스	연결 구조 기반으로 정렬과 검색(범위 검색)에 적합하다. 검색 성능은 HashMap 클래스보다 떨어진다.
HashSet 클래스	HashMap 클래스를 이용해서 구현한다
TreeSet 클래스	TreeMap 클래스를 이용해서 구현한다
LinkedHashMap LinkedHashSet	HashMap 클래스와 HashSet 클래스에 저장 순서를 유지하도록 기능을 추가한 클래스이다.
Stack 클래스	Vector 클래스를 상속받아 구현한다.
Queue 인터페이스	LinkedList 클래스가 Queue 인터페이스를 구현한 클래스이다

Collections 클래스

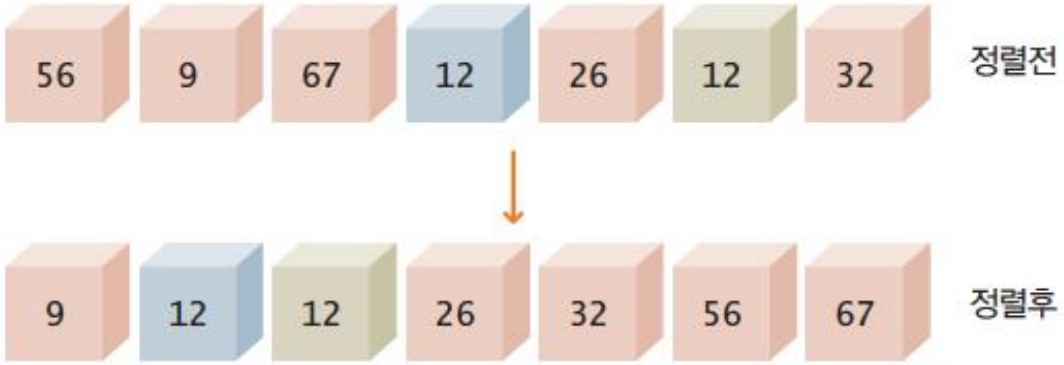
Collections 클래스는 여러 유용한 알고리즘을 구현한 메소드들을 제공한다. 이 메소드들은 제네릭 기술을 사용하여서 작성되었으며 정적 메소드의 형태로 되어 있다. 메소드의 첫번째 매개변수는 알고리즘이 적용되는 컬렉션이 된다.

- 정렬(Sorting)
- 섞기(Shuffling)
- 탐색(Searching)

Collections 클래스

1) 정렬(Sorting)

정렬은 데이터를 어떤 기준에 의하여 순서대로 나열하는 것이다. 정렬알고리즘에는 삽입정렬, 합병정렬, 버블정렬 등의 다양한 방법이 존재한다. Collections 클래스의 정렬은 속도가 비교적 빠르고 안정성이 보장되는 합병정렬을 이용한다. 주어진 데이터를 절반씩 2개의 그룹으로 나누고, 나누어진 각각의 그룹을 다시 절반씩 2개의 그룹으로 나누어, 더 이상 나눌 수 없을 때까지 나눈 후, 나누어진 그룹들을 2개씩 정렬하면서 1개의 그룹이 될 때까지 합병해 가는 방법이다.



Collections 클래스

```
import java.util.*;

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

[i, line, the, walk]

Collections 클래스

2) 섞기(Shuffling)

섞기(Shuffle) 알고리즘은 정렬의 반대 동작을 한다. 즉 원소들의 순서를 랜덤하게 만든다. 이 알고리즘은 특히 게임을 구현할 때 유용하다.

Collections 클래스

```
import java.util.*;

public class Shuffle {
	public static void main(String[] args) {
		List<Integer> list = new ArrayList<Integer>();
			for (int i = 1; i <= 10; i++){
				list.add(i);
			}
			Collections.shuffle(list);
			System.out.println(list);
	}
}
```

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

Collections 클래스

3) 탐색(Searching)

탐색이란 리스트 안에서 원하는 원소를 찾는 것이다. 만약 리스트가 정렬되어 있지 않다면 처음부터 모든 원소를 방문할 수밖에 없다(선형탐색) 하지만 리스트가 정렬되어 있다면 중간에 있는 원소와 먼저 비교하는 것이 좋다(이진탐색) 만약 중간 원소보다 찾고자 하는 원소가 크면 뒷부분에 있고 반대이면 앞부분에 있다. 이런 식으로 하여서 한번 비교할 때마다 문제의 크기를 반으로 줄일 수 있다.

Collections 클래스

```
import java.util.*;

public class Search {
	public static void main(String[] args) {
		int key = 50;
		List<Integer> list = new ArrayList<Integer>();
		for (int i = 0; i < 100; i++) {
			list.add(i);
		}
		int index = Collections.binarySearch(list, key);
		System.out.println("탐색의 반환값 =" + index);
	}
}
```

탐색의 반환값 = 50


Thank You