

8장 기본 패키지

- java.lang
 - ▣ 자바 language 패키지
 - 스트링, 수학 함수, 입출력 등 자바 프로그래밍에 필요한 기본적인 클래스와 인터페이스
 - ▣ 자동으로 import. import 문 필요 없음
- java.util
 - ▣ 자바 유틸리티 패키지
 - 날짜, 시간, 벡터, 해시맵 등과 같은 다양한 유틸리티 클래스와 인터페이스 제공
- java.io
 - ▣ 키보드, 모니터, 프린터, 디스크 등에 입출력을 할 수 있는 클래스와 인터페이스 제공
- java.awt
 - ▣ 자바 GUI 프로그래밍을 위한 클래스와 인터페이스 제공
- javax.swing
 - ▣ 자바 GUI 프로그래밍을 위한 스윙 패키지

- Java.lang 패키지의 인터페이스
 - Comparable
 - Runnable
- Java.util 패키지의 인터페이스
 - Collection
 - Comparator
 - List
- Java.util 패키지의 클래스
 - Arrays 클래스
 - Date
 - Calendar
 - Random

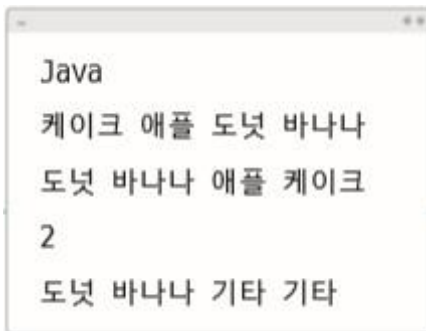
Arrays 클래스(java.util 패키지)

■ Arrays 클래스

- Arrays 클래스가 제공하는 주요 **정적 메서드**

메서드	설명
List asList(배열)	배열을 리스트로 변환한다.
int binarySearch(배열, 키)	배열에서 키 값이 있는 인덱스를 반환한다.
배열 copyOf(배열, 길이)	원본 배열을 길이만큼 복사한다.
배열 copyOfRange(배열, 시작, 끝)	원본 배열을 지정한 영역만큼 복사한다.
boolean equals(배열, 배열)	두 배열의 동일 여부를 비교한다.
void fill(배열, 값)	배열을 지정된 값으로 저장한다.
void fill(배열, 시작, 끝, 값)	배열의 지정된 영역에 지정된 값을 저장한다.
void sort(배열)	배열을 오름차순으로 정렬한다.

- 예제 : [sec03/ArraysDemo](#)



```
package sec03;

import java.util.Arrays;

public class ArraysDemo {
    public static void main(String[] args) {
        char[] a1 = { 'J', 'a', 'v', 'a' };
        char[] a2 = Arrays.copyOf(a1, a1.length);
        System.out.println(a2);

        String[] sa = { "케이크", "애플", "도넛", "바나나" };
        print(sa);
        Arrays.sort(sa); // 정렬 기준이 모호하지 않을 경우
        print(sa);

        System.out.println(Arrays.binarySearch(sa, "애플"));

        Arrays.fill(sa, 2, 4, "기타");
        print(sa);
    }
    static void print(Object[] oa) {
        for (Object o : oa)
            System.out.print(o + " ");
        System.out.println();
    }
}
```

Arrays 클래스(java.util 패키지)

■ 배열 간의 정렬

- 배열 또는 컬렉션 내에 객체(Object)들이 저장되어 정렬 기준이 모호하여 비교가 되지 않을 경우에는 프로그래머가 정렬에 대한 기준을 다음 두 가지 방법 중 하나를 구현하면서 제시해야 한다.

[방법1]

java.lang.Comparable 인터페이스	
인터페이스	설명
public interface Comparable<T>	이 인터페이스를 구현하는 각 클래스의 객체에 정렬 기준을 의미한다. 이는 클래스의 자연적 정렬 기준이라고 불리며 compareTo() 메서드는 자연적 비교 메서드라고도 불린다. 이 인터페이스를 구현하는 객체는 Arrays.sort() 나 Collections.sort() 등에 의해 자동적으로 정렬을 이룰 수 있다.

[방법2]

Comparable 인터페이스의 메서드		
반환형	메서드명	설명
int	compareTo(T o)	인터페이스를 구현한 현재 객체와 인자로 전달된 객체가 정렬을 위한 비교를 한다.

java.util.Comparator 인터페이스	
인터페이스	설명
public interface Comparator<T>	Comparator를 정렬 메서드인 Arrays.sort()나 Collections.sort() 등에 건네 주면 정렬순서를 정확하게 제어할 수 있다. 또 Comparator를 사용하면 TreeSet 또는 TreeMap이라고 하는 자료구조의 순서를 제어할 수도 있다

Comparator 인터페이스의 메서드		
반환형	메서드명	설명
int	compare(T o1, T o2)	정렬을 위해 인자로 전달된 두 개의 객체를 비교한다.
boolean	equals(Object obj)	인자로 전달된 obj 객체가 현재의 Comparator와 동일한지를 비교한다.

=>Comparator는 외부의 Arrays.sort()와 같은 메서드에서 객체 비교를 하여 정렬의 순서가 정해지므로 객체의 정렬 기준을 compare()메서드를 재정의하여 정해 주어야 한다. 인자로 전달된 o1객체가 더 크다면 양수(1)를 반환하고 반대로 o1객체가 o2객체보다 더 작다면 음수(-1)를 반환 해야 한다. 그리고 두 객체가 같다면 0을 반환하는 규칙으로 재정의의 해야 하는 것이다.

10장 람다식과 함수인터페이스

활용

목차

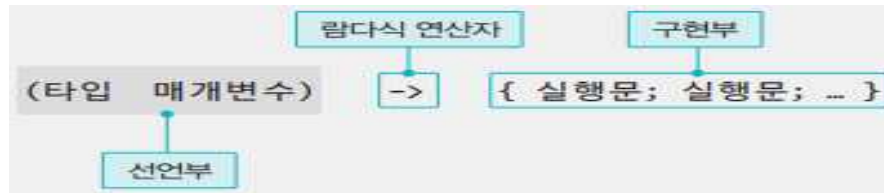
- 람다식 기초
- 란다식 유의 사항과 활용
- 함수형 인터페이스 활용

람다식 필요성

■ 람다식 의미(자바 8부터는 함수형 프로그래밍 기법인 람다식을 지원)

- 메서드를 람다식으로 표현하면 **메서드의 이름이 없어지므로**, 람다식을 '**익명함수(anonymous function)**'라고 함 (메서드를 포함하는 익명 구현 객체를 전달할 수 있는 코드)
- **함수를 변수처럼 사용할 수 있는 개념으로 하나의 식(expression)으로 표현한것**

■ 람다식 문법



■ 람다식 작성

- 람다식에 선언된 매개변수의 타입은 추론이 가능한 경우는 생략할 수 있음, 반환타입이 없는 이
유도 항상 추론이 가능하기 때문
- 익명함수답게 메서드에서 이름과 반환타입을 제거, 선언부와 몸통사이에 ->를 추가



람다식 필요성

■ 람다식 필요성

- 예제와 같이 Rectangle 클래스를 정의하면 사각형 객체끼리 비교할 수 없어 정렬할 수 없다.
- 예제 10-1 : [sec01/etc/ComparableDemo](#)

//예제10-1 : 정렬할 수 없는 클래스

```
package sec01.etc;
import java.util.Arrays;
class Rectangle {
    private int width, height;
    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }
    public int findArea() {
        return width * height;
    }
    public String toString() {
        return String.format("사각형[폭=%d, 높이=%d]",
            width, height);
    }
}
public class ComparableDemo {
    public static void main(String[] args) {
        Rectangle[] rectangles = { new Rectangle(3, 5),
            new Rectangle(2, 10), new Rectangle(5, 5) };
        Arrays.sort(rectangles); //정렬 기준이 없어 오류 발생
        for (Rectangle r : rectangles)
            System.out.println(r);
    }
}
```

```
Exception in thread "main" java.lang.ClassCastException: sec01.Rectangle ...
...
at sec01.etc.ComparableDemo.main(ComparableDemo.java:25)
```

■ 정렬 메서드 구현

새로운 기준으로 비교한다면 새로운 메서드를 구현해야 한다.

(a) flag가 의미하는 비교 기준으로 정렬

(b) 비교 기준마다 다른 메서드로 정렬

■ 문제점

- 복잡하고 가독성이 떨어진다.
- Rectangle 클래스에 색상, 사각형 번호와 같은 다른 속성도 있다면 → 정렬 메서드의 수정 혹은 새로운 메서드 추가 필요

람다식 필요성

■ 객체 비교 및 정렬(방법1)

- 자바는 비교할 수 있는 객체 생성을 위해 **Comparable 인터페이스**(java.lang 패키지)를 제공

=> 이 인터페이스를 구현하는 객체는 Arrays.sort()나 Collections.sort()등에 의해 자동적으로 정렬됨

```
public interface Comparable <T> {  
    int compareTo(T o);  
}
```

=> compareTo()메서드를 비교 메서드라고 함

반환형	메서드명	설명
int	compareTo(T o)	인터페이스를 구현한 현재 객체와 인자로 전달된 객체가 정렬을 위한 비교를 한다.

- java.util 패키지의 Arrays 클래스는 sort()라는 정적 메서드를 제공

```
static void Arrays.sort(Object[] a);
```

배열 원소가 Comparable 타입이어야 한다.

람다식 필요성

■ 객체 비교 및 정렬(방법1)

- 예제와 같이 Rectangle 클래스를 정의하면 사각형 객체끼리 비교할 수 없어 정렬할 수 없다.

- 예제10-1 : [sec01/etc/ComparableDemo](#)

```
//예제10-1 : 정렬할 수 없는 클래스
package sec01.etc;
import java.util.Arrays;
class Rectangle {
    private int width, height;
    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }
    public int findArea() {
        return width * height;
    }
    public String toString() {
        return String.format("사각형[폭=%d, 높이=%d]",
width, height);
    }
}
public class ComparableDemo {
    public static void main(String[] args) {
        Rectangle[] rectangles = { new Rectangle(3, 5),
            new Rectangle(2, 10), new Rectangle(5, 5) };
        Arrays.sort(rectangles); //정렬 기준이 없어 오류 발생
        for (Rectangle r : rectangles)
            System.out.println(r);
    }
}
```

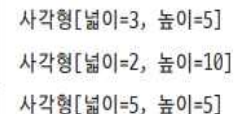
```
public interface Comparable <T> {
    int compareTo(T o);
}
```

```
static void Arrays.sort(Object[] a);
```

배열 원소가 Comparable 타입이어야 한다.

- 예제10-2 : [sec01/ComparableDemo](#)

```
//예제10-2 : 정렬할 수 있는 클래스(비교 기준이 하나인 경우)
package sec01;
import java.util.Arrays;
class Rectangle implements Comparable<Rectangle> {
    private int width, height;
    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }
    public int findArea() {
        return width * height;
    }
    public String toString() {
        return String.format("사각형[폭=%d, 높이=%d]",
width, height);
    }
    public int compareTo(Rectangle o) { //메서드오버라이딩
        return findArea() - o.findArea(); //넓이로 비교
    } //객체를 비교해 작으면 음수(-1), 같으면 0, 크면 양수(1)
}
public class ComparableDemo {
    public static void main(String[] args) {
        Rectangle[] rectangles = { new Rectangle(3, 5),
            new Rectangle(2, 10), new Rectangle(5, 5) };
        Arrays.sort(rectangles); //오류 발생 하지 않음
        for (Rectangle r : rectangles)
            System.out.println(r);
    }
}
```



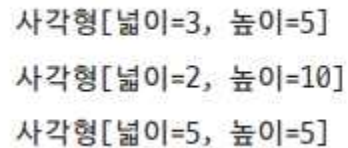
사각형[넓이=3, 높이=5]
사각형[넓이=2, 높이=10]
사각형[넓이=5, 높이=5]

람다식 필요성

■ 객체 비교 및 정렬(방법2)

- 예제10-2 : [sec01/ComparableDemo](#)

```
public interface Comparable <T> {  
    int compareTo(T o);  
}
```



사각형[넓이=3, 높이=5]
사각형[넓이=2, 높이=10]
사각형[넓이=5, 높이=5]

```
static void Arrays.sort(Object[] a);
```

배열 원소가 Comparable 타입이어야 한다.

- 문제점 : 객체를 정렬하기 위하여 모든 클래스에 Comparable 인터페이스(java.lang 패키지)를 구현해야 한다고 하면 문제점 생김

=> 객체끼리 비교할 기준이 여러 가지라면 각 비교 기준마다 Comparable 구현 클래스를 따로 정의해야함

=> 비교 기준을 포함할 클래스가 최종 클래스(final class)라면 Comparable 구현 클래스를 정의할

- 수 없다.
- 해결책 : Comparator 인터페이스(java.util 패키지) 와 sort()

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

```
static void Arrays.sort(T[] a, Comparator<? super T> c);
```

```
Object[] sort(Object[] array, 객체 비교 방식 ) {
```

정렬하기

면적으로 비교하기

둘레로 비교하기

```
}
```

Comparator<? super T>

=> ?는 미지 타입

=> super은 자식 클래스 제한

- 예제10-3 : [sec01/ComparatorDemo](#)

람다식 필요성

■ 객체 비교 및 정렬(방법2)

- 객체끼리 비교할 기준이 여러 가지라면 **Comparator 인터페이스**(java.util 패키지) 와 **sort()** 이용

예제10-3 : [sec01/ComparatorDemo](#)

-익명 클래스는 한번만 사용되는 클래스로 이름이 없는 클래스이며 클래스 선언과 객체 생성 실행문을 하나로 합친 것

```
//예제10-3 : Comparator 인터페이스의 활용, 익명구현 객체
package sec01;
import java.util.Arrays;
import java.util.Comparator;

public class ComparatorDemo {
    public static void main(String[] args) {
        String[] strings = { "로마에 가면 로마법을 따르라.",
                             "시간은 금이다.", "펜은 칼보다 강하다." };

        Arrays.sort(strings, new Comparator<String>() {
            //부모 클래스 객체 생성
            public int compare(String first, String second) {
                return first.length() - second.length();
            }
            // 자식클래스 선언으로 한번만 사용되고
            // 자식클래스 이름 없는 익명 클래스로 구현
        });

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

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}

static void Arrays.sort(T[] a, Comparator<? super T> c);
```

예제10-4 : [sec01/ComparableDemo](#)

-람다식은 익명 구현 객체 대신에 동작을 나타내는 코드만 추출하여 표현하는 방식

-람다식은 1개의 추상 메서드로 구성된 함수형 인터페이스여야 효현 가능하다.

=> 더욱 간결하고 이해하기 쉬운 코드로 표현

```
//예제10-4 : 람다식을 이용한 문자열 길이 순서 정렬
package sec01;
import java.util.Arrays;

public class Lambda1Demo {
    public static void main(String[] args) {
        String[] strings = { "로마에 가면 로마법을 따르라.", "
        시간은 금이다.", "펜은 칼보다 강하다." };

        Arrays.sort(strings, (first, second) -> first.length() -
        second.length());

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

Comparator<? super T>
 => ?는 미지 타입
 => super은 자식 클래스 제한

시간은 금이다.
 펜은 칼보다 강하다.
 로마에 가면 로마법을 따르라.

quiz_1_식별자 ;객체 비교 및 정렬

예제10-2를 참고하여 Rectangle 객체를 넓이와 가로(width)로 비교하는 프로그램 작성하시오.

[quiz_1_1_식별자] Comparator 인터페이스와 익명 구현 객체
[quiz_1_2_식별자] 람다식 표현

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}  
  
static void Arrays.sort(T[] a, Comparator<? super T> c);
```

```
//예제10-2 : 정렬할 수 있는 클래스(비교 기준이 하나인 경우)  
package sec01;  
import java.util.Arrays;  
class Rectangle implements Comparable<Rectangle> {  
    private int width, height;  
    public Rectangle(int width, int height) {  
        this.width = width;  
        this.height = height;    }  
    public int findArea() {  
        return width * height;    }  
    public String toString() {  
        return String.format("사각형[폭=%d, 높이=%d]",  
width, height);  
    }  
    public int compareTo(Rectangle o) {//메서드오버라이딩  
        return findArea() - o.findArea(); //넓이로 비교  
    } //객체를 비교해 작으면 음수(-1), 같으면 0, 크면 양수(1)  
}  
public class ComparableDemo {  
    public static void main(String[] args) {  
        Rectangle[] rectangles = { new Rectangle(3, 5),  
            new Rectangle(2, 10), new Rectangle(5, 5) };  
        Arrays.sort(rectangles); //오류 발생 하지 않음  
        for (Rectangle r : rectangles)  
            System.out.println(r);  
    } }  
}
```

//Comparator 인터페이스의 활용, 익명구현 객체

//(비교 기준이 여러개인 경우)

```
package w12_식별자;  
import java.util.Arrays;  
import java.util.Comparator;  
class Rectangle {  
    private int width, height;  
    public Rectangle(int width, int height) {  
        this.width = width;  
        this.height = height;    }  
    ???  
    public int findArea() {  
        return width * height;    }  
    public String toString() {  
        return String.format("사각형[폭=%d, 높이=%d]", width,  
height);  
    } }  
public class quiz_1_1_식별자 {  
    public static void main(String[] args) {  
        Rectangle[] rectangles = { new Rectangle(3, 5),  
            new Rectangle(2, 10), new Rectangle(5, 5) };  
        Arrays.sort(???);  
        System.out.println("넓이 비교 ");  
        for (Rectangle r : rectangles)  
            System.out.println(r);  
        Arrays.sort(???);  
        System.out.println("너비(width) 비교");  
        for (Rectangle r : rectangles)  
            System.out.println(r.getWidth());    } }  
}
```

넓이 비교
사각형[폭=3, 높이=5]
사각형[폭=2, 높이=10]
사각형[폭=5, 높이=5]
너비(width) 비교
2
3
5

람다식 유의 사항과 활용 quiz_2_식별자

■ 람다식 유의 사항

- 람다식 외부에서 선언된 변수와 동일한 이름의 변수를 람다식에서 선언할 수 없다.
- 람다식에 사용된 지역변수는 final이다.
- 람다식의 this 키워드는 람다식을 실행한 외부 객체를 의미한다.
- 예제 10-8 : [sec02/UseThisDemo](#)



```
sec01.UseThisDemo$1@5b464ce8
UseThisDemo
```

```
package sec02;
interface UseThis { void use(); }

public class UseThisDemo {
    public void lambda() {
        String hi = "Hi!";

        UseThis u1 = new UseThis() { //익명 클래스
            public void use() {
                System.out.println(this); //UserThis의 자식인 익명 개체
                hi = hi + " Lambda."; //지역변수 final이여서 오류
            }
        };
        u1.use();

        UseThis u2 = () -> { //람다식
            System.out.println(this); //UserThisDemo 객체 의미
            hi = hi + " Lambda."; //지역변수 final이여서 오류
        };
        u2.use();
    }

    public String toString() {
        return "UseThisDemo";
    }

    public static void main(String[] args) {
        int one = 1;
        new UseThisDemo().lambda();
        // Comparator<String> c = (one, two) -> one.length() - two.length();
    }
}
```

람다식 유의 사항과 활용

■ 람다식 활용

- 다음과 같은 문제를 해결

- ① 디젤 자동차만 모두 찾아보자.
- ② 10년보다 오래된 자동차만 모두 찾아보자.
- ③ 10년보다 오래된 디젤 자동차만 모두 찾아보자.
- ④ 디젤 자동차를 출력하되 모델과 연식만 나타나도록 출력하자.
- ⑤ 10년보다 오래된 자동차를 출력하되 모델, 연식, 주행거리만 나타나도록 출력하자.

- 필요한 인터페이스 : [sec02/CarPredicate](#), [sec02/CarConsumer](#)

- 문제 해결을 위하여 필요한 인터페이스와 메서드

- ①②③은 요구조건이 서로 달라 하나의 메서드로 구현하기 어렵지만 **인터페이스(CarPredicate)**를 사용하면 편리하다. 인터페이스에 포함될 **추상 메서드(test)**는 **자동차가 요구 조건에 적합한지 아닌지를 반환**하면 되므로 매개변수는 자동차이고 반환 타입이 boolean으로 하면 된다.

- ④⑤⑥조건에도 CarPredicate와 유사한 방법으로 출력할 정보로 인터페이스(CarConsumer)를 사용하고 추상메서드는 매개변수는 자동차이고 반환 타입이 출력정보이므로 void인 추상 메서드를 가진 인터페이스를 사용할 수 있다.

```
package sec02; //①②③
public interface CarPredicate { //자동차 요구조건
    boolean test(Car car); }
```

```
package sec02; //④⑤⑥
public interface CarConsumer { //자동차 출력정보
    void apply(Car car); }
```

- 자동차를 나타낸 클래스로 Car 클래스 내부에 List<Car> cars = Arrays.asList(new Car("소나타", true, 18, 210000),...) 처럼 List<Car> 타입으로 10개의 Car 객체를 미리 포함해 둔다.(11장에서)

- ①②③을 해결하기 위한 메서드 정의

```
List<Car> findCars(List<Car> all, CarPredicate p)
```



- ④⑤⑥을 해결하기 위한 메서드 정의

- 예제 : [sec02/Car](#), [sec02/CarDemo](#)

```
void printCars(List<Car> all, CarConsumer c)
```

람다식 유의 사항과 활용 quiz_3_식별자

● 필요한 인터페이스(예제10-9 ~ 10-10) : [sec02/CarPredicate](#), [sec02/CarConsumer](#)

● 예제10-11, 10-12 : [sec02/Car](#), [sec02/CarDemo](#)

```
package sec02;
public interface CarPredicate { //①②③//자동차 요구조건
    boolean test(Car car); }
```

```
package sec02;
import java.util.Arrays;
import java.util.List;
public class Car {
    private String model;
    private boolean gasoline;
    private int age;
    private int mileage;
    public Car(String model, boolean gasoline, int age, int mileage) {
        this.model = model;
        this.gasoline = gasoline;
        this.age = age;
        this.mileage = mileage; }
    public String getModel() { return model; }
    public boolean isGasoline() { return gasoline; }
    public int getAge() { return age; }
    public int getMileage() { return mileage; }
    public String toString() {
        return String.format("Car(%s, %s, %d, %d)", model, gasoline,
age, mileage);
    }
    public static final List<Car> cars = Arrays.asList(
        new Car("소나타", true, 18, 210000),
        new Car("코란도", false, 15, 200000),
        new Car("그랜저", true, 12, 150000),
        new Car("싼타페", false, 10, 220000),
        new Car("아반테", true, 10, 70000),
        new Car("에쿠스", true, 6, 100000),
        new Car("그랜저", true, 5, 80000),
        new Car("소나타", true, 2, 35000),
        new Car("쏘렌토", false, 1, 10000),
        new Car("아반테", true, 1, 7000));
}
```

- ① 디젤 자동차만 모두 찾아보자.
- ② 10년보다 오래된 자동차만 모두 찾아보자.
- ③ 10년보다 오래된 디젤 자동차만 모두 찾아보자.
- ④ 디젤 자동차를 출력하되 모델과 연식만 나타내도록 출력하자.
- ⑤ 10년보다 오래된 자동차를 출력하되 모델, 연식, 주행거리만 나타내도록 출력하자.

```
package sec02;
public interface CarConsumer { //④⑤⑥ //자동차 출력정보
    void apply(Car car); }
```

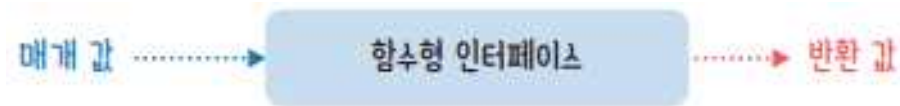
```
package sec02;
import java.util.ArrayList;
import java.util.List;
public class CarDemo {
    public static void main(String[] args) {
        List<Car> dieselCars = findCars(Car.cars, c -> !c.isGasoline());
        System.out.println("디젤 자동차 = " + dieselCars);
        List<Car> oldCars = findCars(Car.cars, c -> c.getAge() > 10);
        System.out.println("오래된 자동차 = " + oldCars);
        List<Car> oldDieselCars = findCars(Car.cars, c -> c.getAge() > 10
&& !c.isGasoline());
        System.out.println("오래된 디젤 자동차 = " + oldDieselCars);

        System.out.print("디젤 자동차 = ");
        printCars(dieselCars, c -> System.out.printf("%s(%d) ",
c.getModel(), c.getAge()));
        System.out.print("\n오래된 자동차 = ");
        printCars(oldCars, c -> System.out.printf("%s(%d, %d) ",
c.getModel(), c.getAge(), c.getMileage()));
    }
    public static List<Car> findCars(List<Car> all, CarPredicate cp) {
        List<Car> result = new ArrayList<>(); //요구조건 맞는 객체 저장위해
        for (Car car : all) {
            if (cp.test(car))
                //test()메서드의 구현 내용은 초록색 람다식으로 제공
                result.add(car);
        }
        return result; }
    public static void printCars(List<Car> all, CarConsumer cc) { for
        (Car car : all) {
            cc.apply(car);
            //apply()메서드의 구현 내용은 위 하늘색 람다식으로 제공
        }
    }
}
```


함수형 인터페이스 응용

■ 함수형 인터페이스

- 의미 : 추상 메서드가 1개만 있는 인터페이스
- 분류



- java.util.function 패키지가 제공하는 함수형 인터페이스 종류

종류	매개 값	반환 값	메서드	의미
Predicate	있음	boolean	test()	매개 값을 조사하여 논릿값으로 보낸다.
Consumer	있음	void	accept()	매개 값을 소비한다.
Supplier	없음	있음	get()	반환 값을 공급한다.
Function	있음	있음	apply()	매개 값을 반환 값으로 매핑한다.
Operator	있음	있음	apply()	매개 값을 연산하여 반환 값으로 보낸다.

- 이외에도 BiPredicate와 같은 다양한 변종도 있음
- java.util.function 패키지가 제공하는 함수형 인터페이스는 추상 메서드 외 다양한 디폴트 메서드나 정적 메서드도 제공

함수형 인터페이스 응용 quiz_4_식별자

■ Predicate 인터페이스 유형

- 매개값을 가지며 논리값을 반환하는 test()라는 추상메서드를 가진 함수형 인터페이스로 자동차 예제에서 사용한 CarPredicate를 일반화한 인터페이스



- Bi, Double, Int, Long을 접두어로 붙인 변종이 있다.
- Test()메서드 외에 공통적으로 디폴트 메서드인 and(), or(), negate()와 정적메서드 isEqual()이 있다.
- Predicate 유형은 다음과 같이 정의

```
Predicate<T> p = t -> { T 타입 t 객체를 조사하여 논리값으로 반환하는 실행문; };
```

- 예제10-13 : [sec03/PredicateDemo](#)

```
출수
1 혹은 짝수
true
false
false
```

```
package sec03;

import java.util.function.BiPredicate;
import java.util.function.IntPredicate;
import java.util.function.Predicate;

public class PredicateDemo {
    public static void main(String[] args) {
        IntPredicate even = x -> x % 2 == 0;
        System.out.println(even.test(3) ? "짝수" : "홀수");

        IntPredicate one = x -> x == 1;
        IntPredicate oneOrEven = one.or(even);
        System.out.println(oneOrEven.test(1) ?
            "1 혹은 짝수" : "1이 아닌 홀수");

        Predicate<String> p = Predicate.isEqual("Java Lambda");
        System.out.println(p.test("Java Lambda"));
        System.out.println(p.test("JavaFX"));

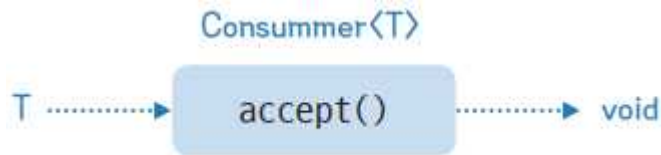
        BiPredicate<Integer, Integer> bp = (x, y) -> x > y;
        System.out.println(bp.test(2, 3));
    }
}
```

=> 예제10-12의 자동차 예제에서 CarPredicate인터페이스를 정의하지 않고 Predicate<Car>를 사용하면 정상적으로 동작

함수형 인터페이스 응용 quiz_5_식별자

■ Consumer 인터페이스 유형

- 주어진 매개 값을 소비만 하고 반환 값이 없는 accept()라는 추상 메서드를 가진 함수형 인터페이스



- Bi, Double, Int, Long, ObjDouble, ObjInt, ObjLong를 접두어로 붙인 변종이 있다.
- 디폴트 메서드로 andThen()을 포함한다.
- Consumer 유형은 다음과 같이 정의

```
Consumer<T> c = t -> { T 타입 t 객체를 사용한 후 void를 반환하는 실행문; }
```

- 예제 10-14 : [sec03/ConsumerDemo](#)

```
java functional interface
Java : Lambda
150
10 * 10 = 100
10 + 10 = 20
```

=> 예제 10-12의 자동차 예제에서 CarConsumer 인터페이스를 정의하지 않고 Consumer<Car>를 사용하면 정상적으로 동작

```
package sec03;
import java.util.function.BiConsumer;
import java.util.function.Consumer;
import java.util.function.IntConsumer;
import java.util.function.ObjIntConsumer;
public class ConsumerDemo {
    public static void main(String[] args) {
        Consumer<String> c1 = x ->
        System.out.println(x.toLowerCase());
        c1.accept("Java Functional Interface");

        BiConsumer<String, String> c2 = (x, y) ->
        System.out.println(x + " : " + y);
        c2.accept("Java", "Lambda");

        ObjIntConsumer<String> c3 = (s, x) -> {
            int a = Integer.parseInt(s) + x;
            System.out.println(a);
        };
        c3.accept("100", 50);

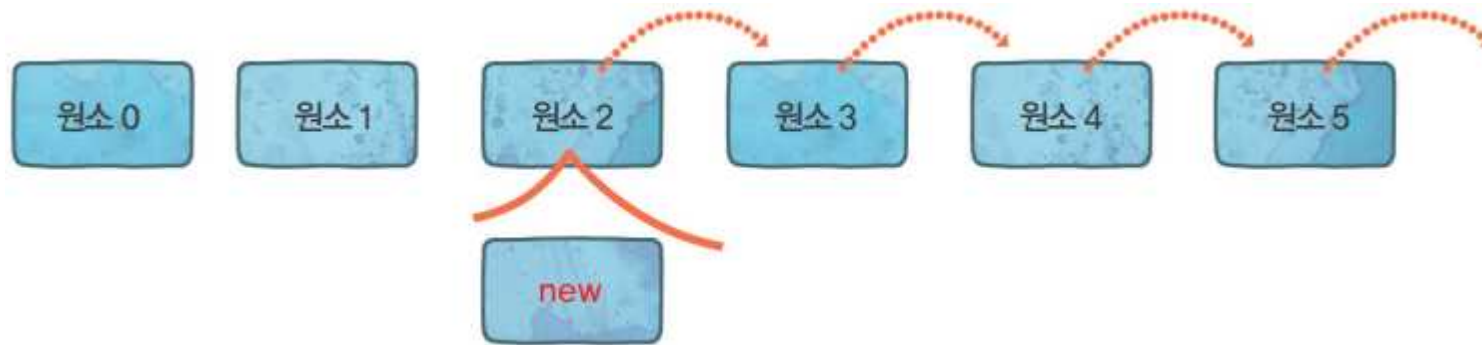
        IntConsumer c4 = x -> System.out.printf("%d * %d
        = %d\n", x, x, x * x);
        IntConsumer c5 = c4.andThen(x ->
        System.out.printf("%d + 10 = %d", x, x + 10));
        c5.accept(10);
    }
}
```

11장 컬렉션 프레임워크

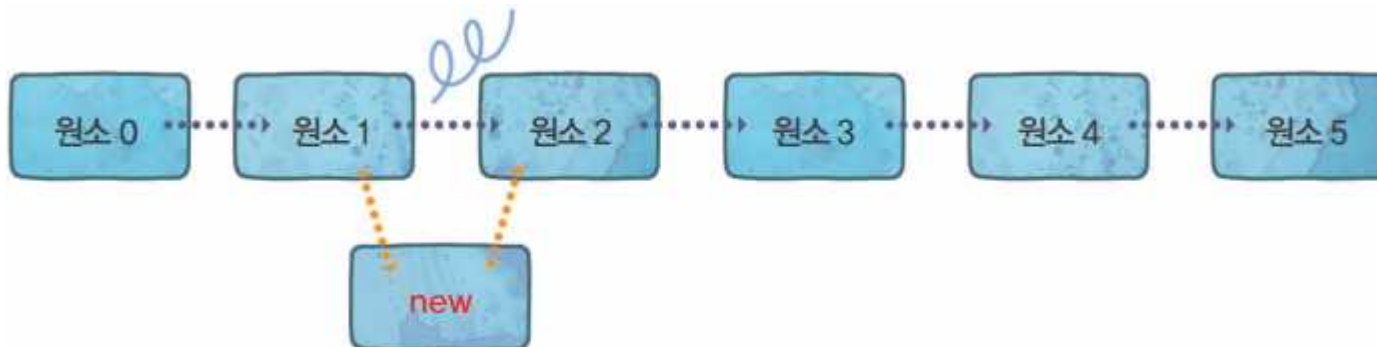
컬렉션 프레임워크 기초

■ 필요성

- 유사한 객체를 여러 개 저장하고 조작해야 할 때가 빈번
- 고정된 크기의 배열의 불편함



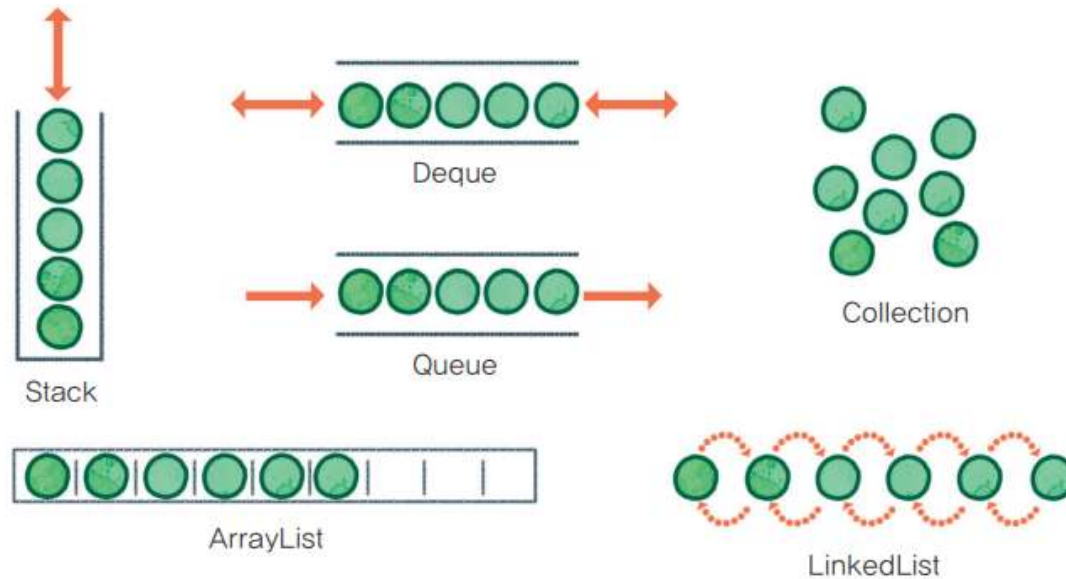
- 연결 리스트를 사용하면 배열보다 쉽게 추가된다.



컬렉션 프레임워크 기초

■ 의미(Java Collection Framework (JCF))

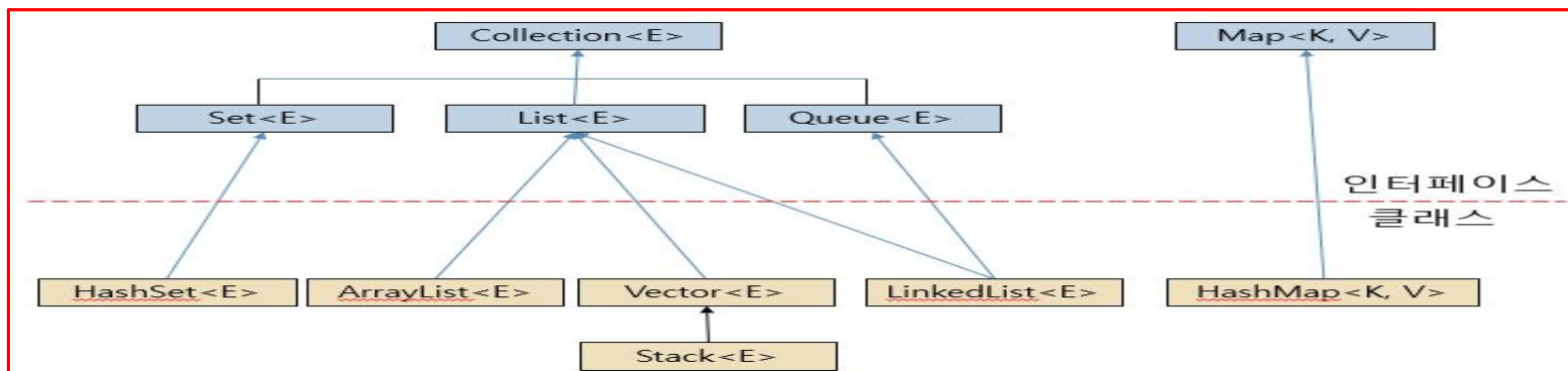
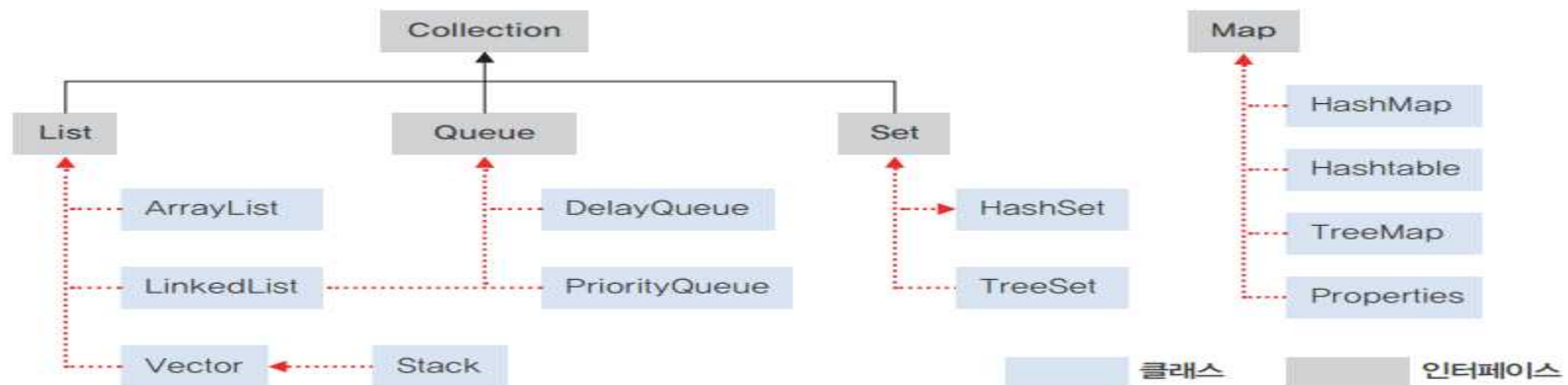
- 유사한 객체의 집단을 효율적으로 관리할 수 있도록 컬렉션 프레임워크를 제공
- 컬렉션 : 데이터를 한곳에 모아 편리하게 저장 및 관리하는 가변 크기의 객체 컨테이너
- 컬렉션 프레임워크 : 객체를 한곳에 모아 효율적으로 관리하고 편리하게 사용할 수 있도록 제공하는 환경



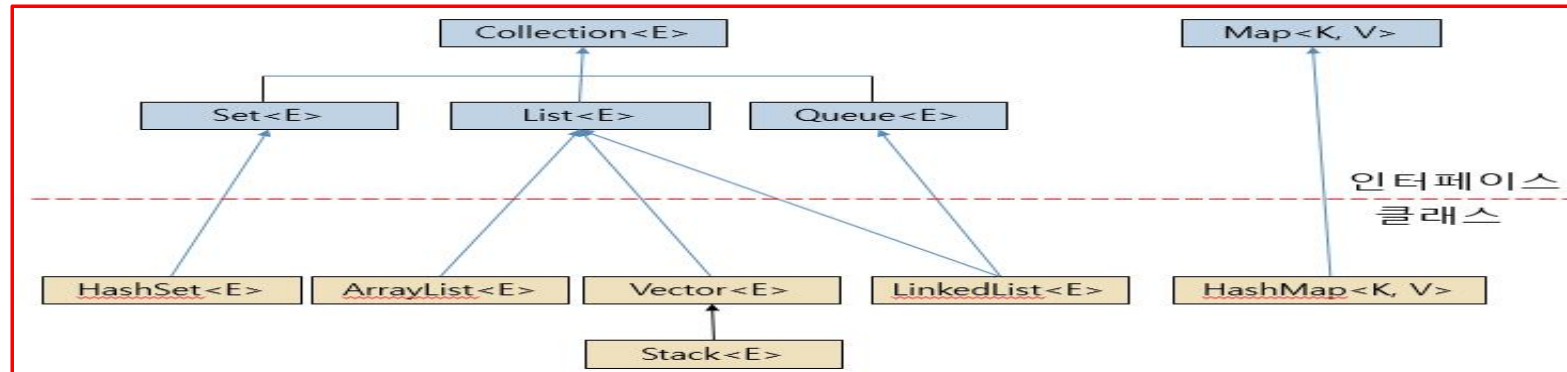
컬렉션 프레임워크 기초

■ 구조

- 컬렉션 프레임워크는 인터페이스와 클래스로 구성
- 인터페이스는 컬렉션에서 수행할 수 있는 각종 연산을 제네릭 타입으로 정의해 유사한 클래스에 일관성 있게 접근하게 함
- 클래스는 컬렉션 프레임워크 인터페이스를 구현한 클래스
- java.util 패키지에 포함 (DelayQueue는 java.util.concurrent 패키지에 포함)



컬렉션 프레임워크 기초



인터페이스	구현 클래스	특징
List	LinkedList Stack Vector ArrayList	순서가 있는 데이터의 집합, 데이터의 중복을 허용한다
Set	HashSet TreeSet	순서를 유지하지 않는 데이터의 집합, 데이터의 중복을 허용하지 않는다.
Map	HashMap TreeMap HashTable Properties	키(key)와 값(value)의 쌍으로 이루어진 데이터의 집합이다. 순서는 유지되지 않고, 키는 중복을 허용하지 않으며 값의 중복을 허용한다.

컬렉션 프레임워크 기초

인터페이스	구현 클래스	특징
List	LinkedList	순서가 있는 데이터의 집합, 데이터의 중복을 허용한다
	Stack	
	Vector	
	ArrayList	
Set	HashSet	순서를 유지하지 않는 데이터의 집합, 데이터의 중복을 허용하지 않는다.
	TreeSet	
Map	HashMap	키(key)와 값(value)의 쌍으로 이루어진 데이터의 집합이다. 순서는 유지되지 않고, 키는 중복을 허용하지 않으며 값의 중복을 허용한다.
	TreeMap	
	HashTable	
	Properties	

List Interface : Collection 인터페이스를 확장한 자료형으로 요소들의 순서를 저장하여 색인(index)을 사용하여 특정 위치에 요소를 삽입하거나 접근할 수 있으며 중복 요소 허용

- ArrayList
 - 상당히 빠르고 크기를 마음대로 조절할 수 있는 배열
 - 단방향 포인터 구조로 자료에 대한 순차적인 접근에 강점이 있음
- Vector
 - ArrayList의 구형버전이며, 모든 메소드가 동기화 되어있음, 잘 쓰이지 않음
- LinkedList
 - 양방향 포인터 구조로 데이터의 삽입, 삭제가 빈번할 경우 빠른 성능을 보장
 - Stack, Queue, 양방향 Queue 등을 만들기 위한 용도로 쓰임

Set Interface : 집합을 정의하며 요소의 중복을 허용하지 않음. 상위 메소드만 사용할

- HashSet
 - 가장 빠른 임의 접근 속도, 순서를 전혀 예측할 수 없음
- LinkedHashSet
 - 추가된 순서, 또는 가장 최근에 접근한 순서대로 접근 가능
- TreeSet
 - 정렬된 순서대로 보관하여 정렬 방법을 지정할 수 있음

Map Interface : Key와 Value의 쌍으로 연관지어 저장하는 객체

- HashMap
 - Map 인터페이스를 구현하기 위해 해시테이블을 사용한 클래스
 - 중복을 허용하지 않고 순서를 보장하지 않음
 - 키와 값으로 null이 허용
- Hashtable
 - HashMap보다는 느리지만 동기화가 지원
 - 키와 값으로 null이 허용되지 않음
- TreeMap
 - 이진검색트리의 형태로 키와 값의 쌍으로 이루어진 데이터를 저장
 - 정렬된 순서로 키 & 값 쌍을 저장하므로 빠른 검색이 가능
 - 저장시 정렬(오름차순)을 하기 때문에 저장시간이 다소 오래 걸림
- LinkedHashMap
 - 기본적으로 HashMap을 상속받아 HashMap과 매우 흡사
 - Map에 있는 엔트리들의 연결 리스트가 유지되므로 입력한 순서대로 반복 가능

Collection 인터페이스

■ Collection 인터페이스와 구현 클래스

인터페이스		특징	구현 클래스
Collection	List	객체의 순서가 있고, 원소가 중복될 수 있다.	ArrayList, Stack, Vector, LinkedList
	Queue	객체를 입력한 순서대로 저장하며, 원소가 중복될 수 있다.	DelayQueue, PriorityQueue, LinkedList
	Set	객체의 순서가 없으며, 동일한 원소를 중복할 수 없다.	HashSet, TreeSet, EnumSet

Collection 인터페이스

■ Collection 인터페이스

● 주요 메서드

메서드	설명
<code>boolean add(E e)</code>	객체를 맨 끝에 추가한다.
<code>void clear()</code>	저장된 모든 객체를 제거한다.
<code>boolean contains(Object o)</code>	명시한 객체의 저장 여부를 조사한다.
<code>boolean isEmpty()</code>	리스트가 비어 있는지 조사한다.
<code>Iterator<E> iterator()</code>	Iterator를 반환한다.
<code>boolean remove(Object o)</code>	명시한 첫 번째 객체를 제거하고, 제거 여부를 반환한다.
<code>int size()</code>	저장된 전체 객체의 개수를 반환한다.
<code>T[] toArray(T[] a)</code>	리스트를 배열로 반환한다.

● 이외에도 Collection 인터페이스는 다음과 같은 유용한 디폴트 메서드를 제공

```
default void forEach(Consumer<? super T> action)
default boolean removeIf(Predicate <? super E> filter)
default <T> T[] toArray(IntFunction<T[]> generator)
```

→ 자바 11부터

Collection 인터페이스

■ 컬렉션의 반복 처리

- Collection 인터페이스는 `iterator()` 메서드를 통하여 반복자를 제공
- `Iterator`는 자바의 컬렉션 프레임워크 (JCF)에서 컬렉션에 저장되어 있는 요소들을 읽어오는 방법을 표준화한 것이다.
- `Iterator`는 반복자로서 객체 지향적 프로그래밍에서 배열이나 그와 유사한 자료구조의 내부요소를 순회하는 객체다.
- 키-값 구조의 Map 컬렉션은 반복자를 제공하지 않는다.
- `Iterator` 인터페이스가 제공하는 주요 메서드

메서드	설명
<code>boolean hasNext()</code>	다음 원소의 존재 여부를 반환한다.
<code>E next()</code>	다음 원소를 반환한다.
<code>default void remove()</code>	마지막에 순회한 컬렉션의 원소를 삭제한다.

- 예제 11-1: [sec02/IteratorDemo](#)



```
package sec02;
import java.util.*;
public class IteratorDemo {
    public static void main(String[] args) {
        Collection<String> list = Arrays.asList("다람쥐", "개구리", "나비");
        //Collection타입 변환

        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext())
            System.out.print(iterator.next() + "-");
        System.out.println();

        while (iterator.hasNext())
            System.out.print(iterator.next() + "+");
        System.out.println();

        iterator = list.iterator();
        while (iterator.hasNext())
            System.out.print(iterator.next() + "=");

    }
}
```

Collection 인터페이스

■ List 컬렉션

- **순서가 있는 객체**를 중복 여부와 상관없이 저장하는 리스트 자료구조를 지원. 배열과 매우 유사하지만 크기가 가변적. 원소의 순서가 있으므로 원소를 저장하거나 읽어올 때 인덱스를 사용

메서드	설명
void add(int index, E element)	객체를 인덱스 위치에 추가한다.
E get(int index)	인덱스에 있는 객체를 반환한다.
int indexOf(Object o)	명시한 객체가 있는 첫 번째 인덱스를 반환한다.
E remove(int index)	인덱스에 있는 객체를 제거한다.
E set(int index, E element)	인덱스에 있는 객체와 주어진 객체를 교체한다.
List<E> subList(int from, int to)	범위에 해당하는 객체를 리스트로 반환한다.

- 디폴트 메서드

```
default void replaceAll(UnaryOperator<E> operator)
default void sort(Comparator<? super E> c)
```

- 팩토리 메서드(자바 9부터)

```
static <E> List<E> of(E... elements)
```

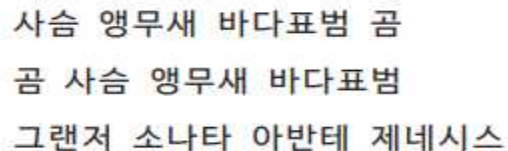
Collection 인터페이스

■ List 컬렉션

- List 타입과 배열 사이에는 다음 메서드를 사용하여 상호 변환

```
public static <T> List<T> asList(T... a) // java.util.Arrays 클래스의 정적 메서드
<T> T[] toArray(T[] a) // java.util.List 클래스의 메서드
```

- Java.util.Arrays 클래스의 정적 메서드 asList()에 의하여 반환되는 List 객체는 크기가 고정된 배열을 List 타입으로 변환했기 때문에 원소를 추가하거나 제거할 수 없다.
- 예제 11-2 : [sec02/ListDemo](#)
- 배열과 List 타입과의 상호 변환 및 List가 제공하는 디폴트 메서드 예제



```
사슴 앵무새 바다표범 곰
곰 사슴 앵무새 바다표범
그랜저 소나타 아반테 제네시스
```

- 대표적인 List 구현 클래스 : ArrayList, Vector

```
package sec02;

import java.util.Arrays;
import java.util.List;

public class ListDemo {
    public static void main(String[] args) {
        String[] animals1 = { "사슴", "호랑이", "바다표범", "곰" };

        List<String> animals2 = Arrays.asList(animals1); // List 타입 변환
        animals2.set(1, "앵무새");
        // animals2.add("늑대"); // List 타입으로 변환으로 원소를 추가 불가

        for (String s : animals2)
            System.out.print(s + " ");
        System.out.println();

        animals2.sort((x, y) -> x.length() - y.length());
        String[] animals3 = animals2.toArray(new String[0]);
        for (int i = 0; i < animals3.length; i++)
            System.out.print(animals3[i] + " ");
        System.out.println();

        List<String> car = List.of("그랜저", "소나타", "아반테", "제네시스");
        // car.set(1, "싼타페");
        car.forEach(s -> System.out.print(s + " "));

        // List<Object> objects = List.of("a", null);
    }
}
```

Collection 인터페이스

■ List 컬렉션

- **ArrayList 클래스** : 동적 배열로 사용
- ArrayList 클래스는 List 인터페이스의 구현 클래스로 원소의 추가 및 제거등을 인덱스로 관리한다.
- Vector 클래스도 ArrayList와 동일한 기능을 제공하지만, ArrayList와 달리 동기화된 메서드로 구현해서 스레드에 안전
- 예제11-3 : [sec02/ArrayListDemo](#)
 - ArrayList에서 사용할 수 있는 각종클래스를 활용한 예제
 - List타입은 ArrayList()생성자로 사용하여 ArrayList타입으로 변환할 수 있다.

```
1
false
true
뉴그랜저 뉴아반테 뉴제네시스 뉴싼타페
true
```

```
package sec02;

import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo {
    public static void main(String[] args) {
        List<String> list = List.of("그랜저", "소나타", "아반테", "제네시스", "소울");

        System.out.println(list.indexOf("소나타"));
        System.out.println(list.contains("싼타페"));

        List<String> cars1 = new ArrayList<>(list);
        cars1.add("싼타페");
        List<String> cars2 = new ArrayList<>(list);
        cars2.remove("제네시스");
        System.out.println(cars1.containsAll(cars2));

        cars1.removeIf(c -> c.startsWith("소"));
        cars1.replaceAll(s -> "뉴" + s);
        cars1.forEach(s -> System.out.print(s + " "));
        System.out.println();

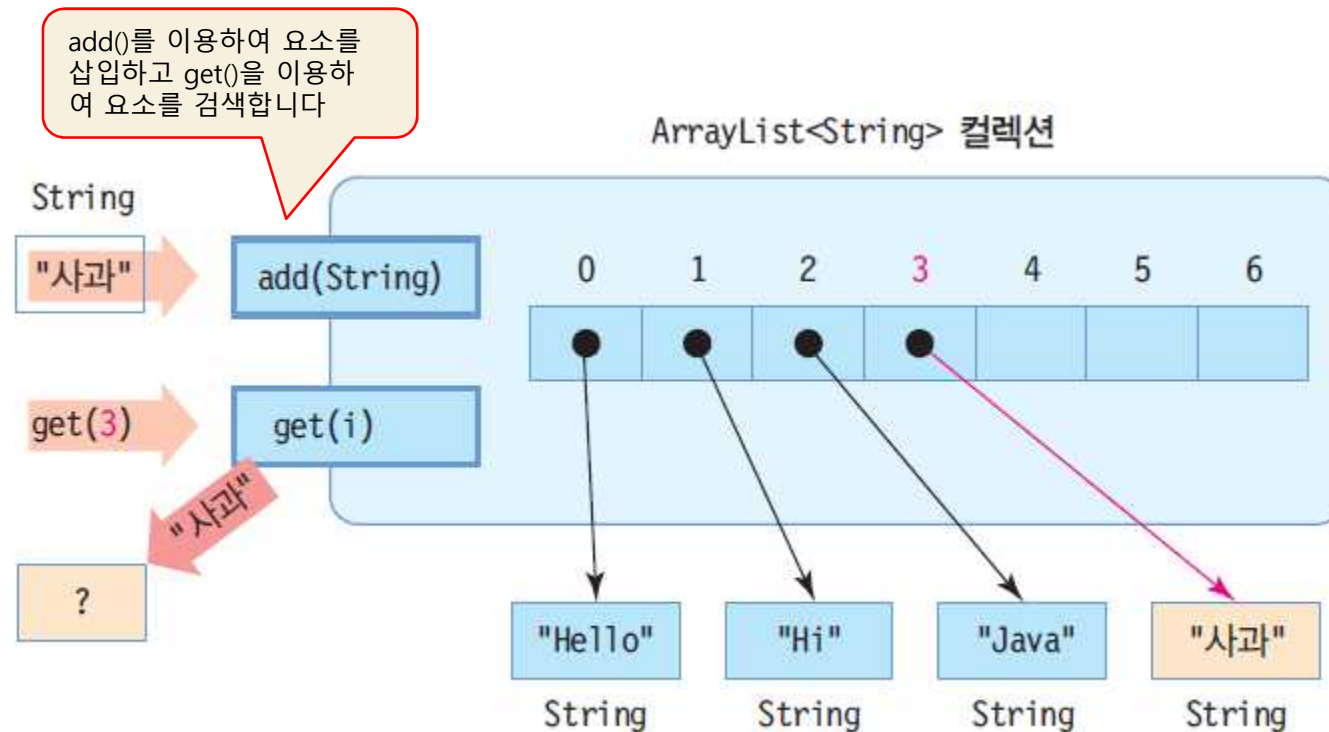
        cars1.clear();
        System.out.println(cars1.isEmpty());
    }
}
```

ArrayList<E> 클래스

- ArrayList<E>의 특성
 - ▣ java.util.ArrayList, 가변 크기 배열을 구현한 클래스
 - <E>에서 E 대신 요소로 사용할 특정 타입으로 구체화
 - ▣ ArrayList에 삽입 가능한 것
 - 객체, null
 - 기본 타입은 박싱/언박싱으로 Wrapper 객체로 만들어 저장
 - ▣ ArrayList에 객체 삽입/삭제
 - 리스트의 맨 뒤에 객체 추가
 - 리스트의 중간에 객체 삽입
 - 임의의 위치에 있는 객체 삭제 가능
 - ▣ 벡터와 달리 스레드 동기화 기능 없음
 - 다수 스레드가 동시에 ArrayList에 접근할 때 동기화되지 않음
 - 개발자가 스레드 동기화 코드 작성

ArrayList<String> 컬렉션의 내부 구성

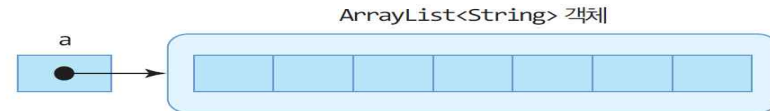
```
ArrayList<String> al = new ArrayList<String>();
```



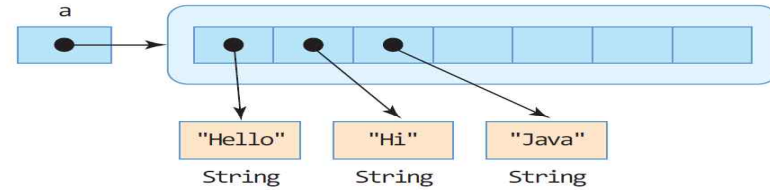
ArrayList<E> 클래스의 주요 메소드

메소드	설명
<code>boolean add(E element)</code>	ArrayList의 맨 뒤에 element 추가
<code>void add(int index, E element)</code>	인덱스 index 위치에 element 삽입
<code>boolean addAll(Collection<? extends E> c)</code>	컬렉션 c의 모든 요소를 ArrayList의 맨 뒤에 추가
<code>void clear()</code>	ArrayList의 모든 요소 삭제
<code>boolean contains(Object o)</code>	ArrayList가 지정된 객체를 포함하고 있으면 true 리턴
<code>E elementAt(int index)</code>	index 인덱스의 요소 리턴
<code>E get(int index)</code>	index 인덱스의 요소 리턴
<code>int indexOf(Object o)</code>	o와 같은 첫 번째 요소의 인덱스 리턴, 없으면 -1 리턴
<code>boolean isEmpty()</code>	ArrayList가 비어있으면 true 리턴
<code>E remove(int index)</code>	index 인덱스의 요소 삭제
<code>boolean remove(Object o)</code>	o와 같은 첫 번째 요소를 ArrayList에서 삭제
<code>int size()</code>	ArrayList가 포함하는 요소의 개수 리턴
<code>Object[] toArray()</code>	ArrayList의 모든 요소를 포함하는 배열 리턴

ArrayList 생성 `ArrayList<String> a = new ArrayList<String>(7);`



요소 삽입
`a.add("Hello");`
`a.add("Hi");`
`a.add("Java");`

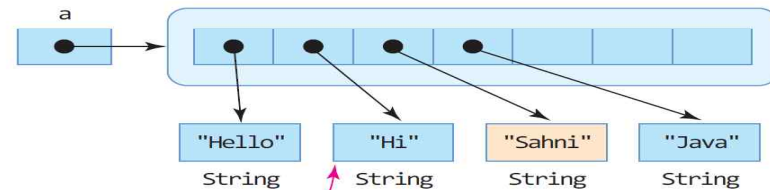


요소 개수 n
용량
`int n = a.size(); // n은 3`
~~`int c = a.capacity(); // capacity() 메소드 없음`~~
오류

n = 3

요소 중간 삽입
`a.add(2, "Sahni");`

오류
~~`a.add(5, "Sahni");`~~
`// a.size()보다 큰 위치에 삽입 불가능. 오류`



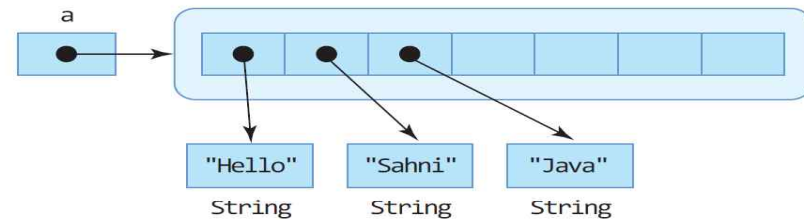
요소 알아내기
`String str = a.get(1);`

"Hi"



요소 삭제
`a.remove(1);`

오류
~~`a.remove(4);`~~ `// 오류`



모든 요소 삭제
`a.clear();`



quiz_6_식별자:문자열 입력받아 ArrayList에 저장

이름을 4개 입력받아 ArrayList에 저장하고 모두 출력한 후 제일 긴 이름을 출력하라.

```
import java.util.*;

public class quiz_6_식별자 {
    public static void main(String[] args) {
        // 문자열만 삽입가능한 ArrayList 컬렉션 생성
        ArrayList<String> a = new ???<String>();

        // 키보드로부터 4개의 이름 입력받아 ArrayList에 삽입
        Scanner scanner = new Scanner(System.in);
        for(int i=0; i<4; i++) {
            System.out.print("이름을 입력하세요>>");
            String s = scanner.next(); // 키보드로부터 이름 입력
            a. ???(s); // ArrayList 컬렉션에 삽입
        }

        // ArrayList에 들어 있는 모든 이름 출력
        for(int i=0; i<a.size(); i++) {
            // ArrayList의 i 번째 문자열 얻어오기
            String name = a. ???(i);
            System.out.print(name + " ");
        }
    }
}
```

```
// 가장 긴 이름 출력
int longestIndex = 0;
for(int i=1; i<a. ???(); i++) {
    if(a.get(longestIndex).length() < a.get(i). ???())
        longestIndex = i;
}
System.out.println("\n가장 긴 이름은 : " +
    a.get(longestIndex));
}
scanner.close();
}
```

이름을 입력하세요>>Mike
이름을 입력하세요>>Jane
이름을 입력하세요>>Ashley
이름을 입력하세요>>Helen
Mike Jane Ashley Helen
가장 긴 이름은 : Ashley

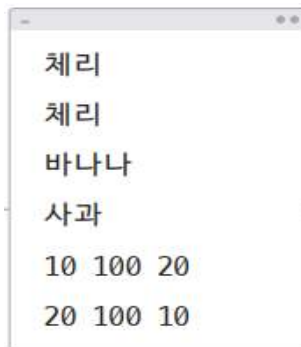
Collection 인터페이스

■ List 컬렉션

- Stack 클래스 : 후입선출 방식으로 객체를 관리하며, Vector의 자식 클래스이다.
대부분의 인덱스가 0부터 시작하지만 Stack 클래스는 1부터 시작한다.

메서드	설명
boolean empty()	스택이 비어 있는지 조사한다.
E peek()	스택의 최상위 원소를 제거하지 않고 엿본다.
E pop()	스택의 최상위 원소를 반환하며, 스택에서 제거한다.
E push(E item)	스택의 최상위에 원소를 추가한다.
int search(Object o)	주어진 원소의 인덱스 값(1부터 시작)을 반환한다.

- 예제11-4 : [sec02/StackDemo](#)



```
package sec02;

import java.util.Stack;

public class StackDemo {
    public static void main(String[] args) {
        Stack<String> s1 = new Stack<>();

        s1.push("사과");
        s1.push("바나나");
        s1.push("체리");

        System.out.println(s1.peek());

        System.out.println(s1.pop());
        System.out.println(s1.pop());
        System.out.println(s1.pop());
        System.out.println();

        Stack<Integer> s2 = new Stack<>();

        s2.add(10);
        s2.add(20);
        s2.add(1, 100);

        for (int value : s2)
            System.out.print(value + " ");
        System.out.println();

        while (!s2.empty())
            System.out.print(s2.pop() + " ");

        }
}
```

Map 인터페이스

■ 특징

- 키와 값, 이렇게 쌍으로 구성된 객체를 저장하는 자료구조
- 맵이 사용하는 키와 값도 모두 객체
- 키는 중복되지 않고 하나의 값에만 매핑되어 있으므로 키가 있다면 대응하는 값을 얻을 수 있다.
- Map 객체에 같은 키로 중복 저장되지 않도록 하려면 Set 객체처럼 키로 사용할 클래스에 대한 hashCode()와 equals() 메서드를 오버로딩해야 한다.
- 구현 클래스 : HashMap, Hashtable, TreeMap, Properties

◦ Map은 Key와 Value의 쌍으로 이루어진 자료구조이다.



Map 인터페이스

■ 주요 메서드

- Map 인터페이스가 제공

메서드	설명
<code>void clear()</code>	모든 매핑을 삭제한다.
<code>boolean containsKey(Object key)</code>	주어진 키의 존재 여부를 반환한다.
<code>boolean containsValue(Object value)</code>	주어진 값의 존재 여부를 반환한다.
<code>Set<Map.Entry<K, V>> entrySet()</code>	모든 매핑을 Set 타입으로 반환한다.
<code>V get(Object key)</code>	주어진 키에 해당하는 값을 반환한다.
<code>boolean isEmpty()</code>	컬렉션이 비어 있는지 여부를 반환한다.
<code>Set<K> keySet()</code>	모든 키를 Set 타입으로 반환한다.
<code>V put(K key, V value)</code>	주어진 키-값을 저장하고 값을 반환한다.
<code>V remove(Object key)</code>	키와 일치하는 원소를 삭제하고 값을 반환한다.
<code>int size()</code>	컬렉션의 크기를 반환한다.
<code>Collection<V> values()</code>	모든 값을 Collection 타입으로 반환한다.

- Map.Entry<K, V> 인터페이스가 제공

메서드	설명
<code>K getKey()</code>	원소에 해당하는 키를 반환한다.
<code>V getValue()</code>	원소에 해당하는 값을 반환한다.
<code>V setValue()</code>	원소의 값을 교체한다.

Map 인터페이스

■ 주요 메서드

- 팩토리 메서드(자바 9부터)

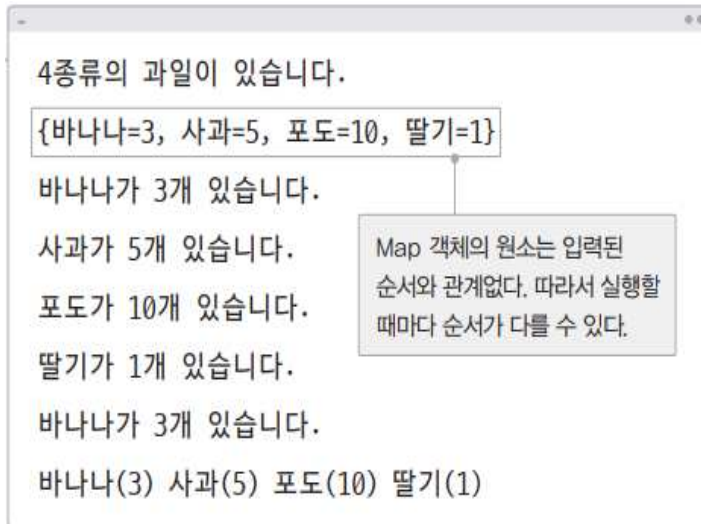
```
static <K, V> Map<K,V> of(K k1, V v1)
```

- 디폴트 메서드

```
default void forEach(BiConsumer action)
```

```
default void replaceAll(BiFunction function)
```

- 예제 11-10 : [sec03/MapDemo](#)



The screenshot shows a Java application window with the following text:

4종류의 과일이 있습니다.
{바나나=3, 사과=5, 포도=10, 딸기=1}
바나나가 3개 있습니다.
사과가 5개 있습니다.
포도가 10개 있습니다.
딸기가 1개 있습니다.
바나나가 3개 있습니다.
바나나(3) 사과(5) 포도(10) 딸기(1)

A callout box points to the map representation, stating: "Map 객체의 원소는 입력된 순서와 관계없다. 따라서 실행할 때마다 순서가 다를 수 있다."

```
package sec03;
```

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

```
public class MapDemo {
```

```
    public static void main(String[] args) {
```

```
        Map<String, Integer> fruits =
```

```
            Map.of("사과", 5, "바나나", 3, "포도", 10, "딸기", 1);
```

```
        System.out.println(fruits.size() + "종류의 과일이 있습니다.");
```

```
        System.out.println(fruits);
```

```
        for (String key : fruits.keySet())
```

```
            System.out.println(key + "가 " + fruits.get(key) + "개 있습니다.");
```

```
        String key = "바나나";
```

```
        if (fruits.containsKey(key))
```

```
            System.out.println(key + "가 " + fruits.get(key) + "개 있습니다.");
```

```
        fruits.forEach((k, n) -> System.out.print(k + "(" + n + ") "));
```

```
        // fruits.put("키위", 2);
```

```
        // fruits.remove("사과");
```

```
        // fruits.clear();
```

```
        //Map.of()에 의하여 생성된 객체는 불변으로 원소의 추가,삭제등은 실행오류
```

```
    }
```

```
}
```

Map 인터페이스

■ HashMap과 Hashtable

- Hashtable은HashMap은 입력 순서와 상관 없이 키-값 객체로 구성된 Map 구현체이다.
- Hashtable은HashMap과 달리 동기화된 메서드로 구현되어 스레드에 안전
- HashMap에서는 키와 값으로 null을 사용할 수 있지만 Hashtable에서는 사용할 수 없다.
- 예제11-11 : [sec03/HashMap1Demo](#)
 - HashMap객체에 원소를 추가 혹은 삭제하거나 모든 키를 Set으로 가져오는 등의 작업을 수행하는 예제

```
package sec03;  
import java.util.HashMap;  
import java.util.Map;
```

```
public class HashMap1Demo {  
    public static void main(String[] args) {  
        Map<String, Integer> map =  
            Map.of("사과", 5, "바나나", 3, "포도", 10, "딸기", 1);  
  
        Map<String, Integer> fruits = new HashMap<>(map);  
        System.out.println("현재 " + fruits.size() + "종류의 과일이 있습니다.");  
        fruits.remove("바나나");  
        System.out.println("바나나를 없앤 후 " + fruits.size() + "종류의 과일이 있습니다.");  
  
        fruits.put("망고", 2);  
        System.out.println("망고를 추가한 후 현재 " + fruits + "가 있습니다.");  
  
        fruits.clear();  
        System.out.println("모두 없앤 후 " + fruits.size() + "종류의 과일이 있습니다.");  
    }  
}
```

현재 4종류의 과일이 있습니다.
바나나를 없앤 후 3종류의 과일이 있습니다.
망고를 추가한 후 현재 {사과=5, 포도=10, 망고=2, 딸기=1}가 있습니다.
모두 없앤 후 0종류의 과일이 있습니다.

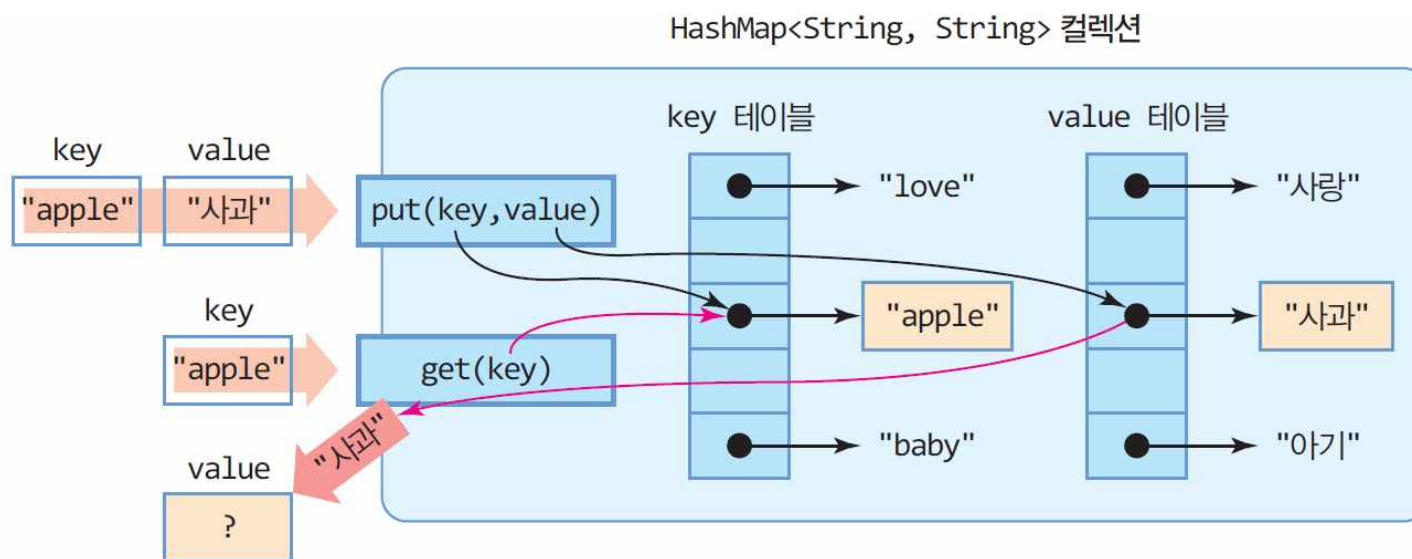
HashMap<K,V>

- HashMap<K,V>
 - ▣ 키(key)와 값(value)의 쌍으로 구성되는 요소를 다루는 컬렉션
 - java.util.HashMap
 - K는 키로 사용할 요소의 타입, V는 값으로 사용할 요소의 타입 지정
 - 키와 값이 한 쌍으로 삽입
 - 키는 해시맵에 삽입되는 위치 결정에 사용
 - 값을 검색하기 위해서는 반드시 키 이용
 - ▣ 삽입, 삭제, 검색이 빠른 특징
 - 요소 삽입 : put() 메소드
 - 요소 검색 : get() 메소드
 - ▣ 예) HashMap<String, String> 생성, 요소 삽입, 요소 검색

```
HashMap<String, String> h = new HashMap<String, String>();  
h.put("apple", "사과"); // "apple" 키와 "사과" 값의 쌍을 해시맵에 삽입  
String kor = h.get("apple"); // "apple" 키로 값 검색. kor는 "사과"
```

HashMap<String, String>의 내부 구성

```
HashMap<String, String> map = new HashMap<String, String>();
```

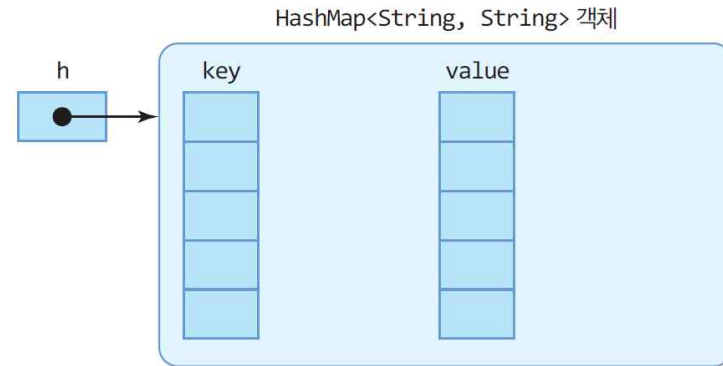


HashMap<K,V>의 주요 메소드

메소드	설명
<code>void clear()</code>	해시맵의 모든 요소 삭제
<code>boolean containsKey(Object key)</code>	지정된 키(key)를 포함하고 있으면 true 리턴
<code>boolean containsValue(Object value)</code>	지정된 값(value)에 일치하는 키가 있으면 true 리턴
<code>V get(Object key)</code>	지정된 키(key)의 값 리턴, 키가 없으면 null 리턴
<code>boolean isEmpty()</code>	해시맵이 비어 있으면 true 리턴
<code>Set<K> keySet()</code>	해시맵의 모든 키를 담은 Set<K> 컬렉션 리턴
<code>V put(K key, V value)</code>	key와 value 쌍을 해시맵에 저장
<code>V remove(Object key)</code>	지정된 키(key)를 찾아 키와 값 모두 삭제
<code>int size()</code>	HashMap에 포함된 요소의 개수 리턴

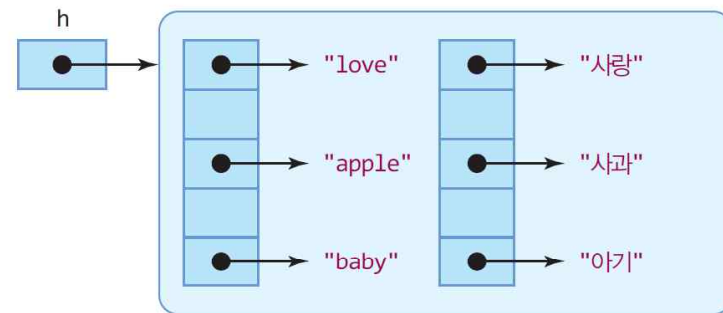
해시맵 생성

```
HashMap<String, String> h =  
new HashMap<String, String>();
```



(키, 값) 삽입

```
h.put("baby", "아기");  
h.put("love", "사랑");  
h.put("apple", "사과");
```



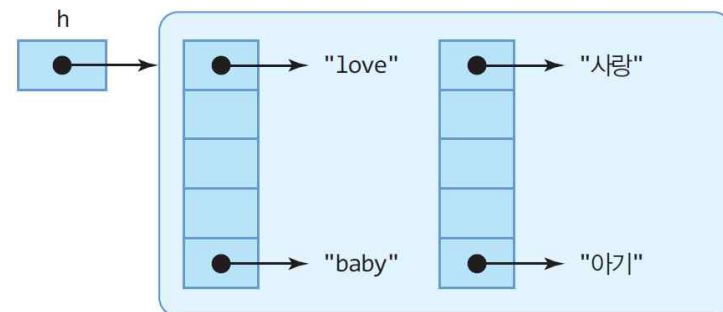
키로 값 읽기

```
String kor = h.get("love");
```

kor = "사랑"

키로 요소 삭제

```
h.remove("apple");
```



요소 개수

```
int n = h.size();
```

n = 2

quiz_7_식별자 :

HashMap을 이용하여 (영어, 한글) 단어 쌍의 저장 검색

(영어, 한글) 단어를 쌍으로 해시맵에 저장하고 영어로 한글을 검색하는 프로그램을 작성하라. "exit"이 입력되면 프로그램을 종료한다.

```
import java.util.*;

public class quiz_7_식별자 {
    public static void main(String[] args) {
        // 영어 단어와 한글 단어의 쌍을 저장하는 HashMap 컬렉션 생성
        HashMap<String, String> dic =
            new ???<String, String>();

        // 3 개의 (key, value) 쌍을 dic에 저장
        ???.put("baby", "아기"); // "baby"는 key, "아기"은 value
        ???.put("love", "사랑");
        ???.put("apple", "사과");

        // 영어 단어를 입력받고 한글 단어 검색. "exit" 입력받으면 종료
        Scanner scanner = new Scanner(System.in);
        while(true) {
            System.out.print("찾고 싶은 단어는?");
            String eng = scanner.next();
            if(???.equals("exit")) {
                System.out.println("종료합니다...");
                break;
            }
        }
    }
}
```

```
// 해시맵에서 '키' eng의 '값' kor 검색
String kor = dic.get(???);
if(kor == null)
    System.out.println(eng +
        "는 없는 단어 입니다.");
else
    System.out.println(kor);
}
scanner.close();
}
```

찾고 싶은 단어는?apple
사과
찾고 싶은 단어는?babo
babo는 없는 단어 입니다.
찾고 싶은 단어는?exit
종료합니다...