

제12장

# 컬렉션 프레임워크

구디아카데미 ▷ 민경태 강사

# 학습목표

1. Wrapper 클래스에 대해서 알 수 있다.
2. 제네릭에 대해서 알 수 있다.
3. 컬렉션 프레임워크의 개념과 구조에 대해서 알 수 있다.
4. List<E> 인터페이스에 대해서 알 수 있다.
5. Set<E> 인터페이스에 대해서 알 수 있다.
6. Iterator<E> 인터페이스에 대해서 알 수 있다.
7. Map<K, V> 인터페이스에 대해서 알 수 있다.

```
each: function(e, t, n) {  
    o = e.length,  
    a = M(e);  
    if (n) {  
        if (a) {  
            for (; o > i; i++)  
                if (r = t.apply(e[i], n), r ===  
            ) else  
                for (i in e)  
                    if (r = t.apply(e[i], n), r ===  
        } else if (a) {  
            for (; o > i; i++)  
                if (r = t.call(e[i], i, e[i])  
        } else  
            for (i in e)  
                if (r = t.call(e[i], i, e[i])  
    return e  
},  
trim: b && !b.call("\uffff\u00a0") ?  
    return null == e ? "" : b.call(  
} : function(e) {  
    return null == e ? "" : (e +  
},  
makeArray: function(e, t) {  
    var n = t || [];  
    return null != e && (M(Obj  
}),  
isArray: function(e, t, n) {  
    var r;  
    if (t) {  
        if (n) return m.c  
        for (n = t.length  
            if (n in t  
    }  
}
```

# 목차

1. Wrapper 클래스
2. 제네릭
3. 컬렉션 프레임워크의 개념과 구조
4. List<E> 인터페이스
5. Set<E> 인터페이스
6. Iterator<E> 인터페이스
7. Map<K, V> 인터페이스

```

each: function(e, t, n) {
    var r, i = 0,
        o = e.length,
        a = M(e);
    if (n) {
        if (a) {
            for (; o > i; i++)
                if (r = t.apply(e[i], n), r === !1) break;
        } else
            for (i in e)
                if (r = t.apply(e[i], n), r === !1) break;
    } else if (a) {
        for (; o > i; i++)
            if (r = t.call(e[i], e[i]), r === !1) break;
    } else
        for (i in e)
            if (r = t.call(e[i], e[i]), r === !1) break;
    return e;
},
trim: b && !b.call("\uffff\u00a0") ? function(e) {
    return null == e ? "" : b.call(e);
} : function(e) {
    return null == e ? "" : (e + "").replace(C, "");
},
makeArray: function(e, t) {
    var n = t || [];
    return null != e && (M(Object(e)) ? x.merge(n, "string"
),
isArray: function(e, t, n) {
    var r;
    if (t) {
        if (n) return m.call(t, e, n);
        for (r = t.length, r = r ? 0 > n ? Math.max(0, r + n) : n : 0; r in t && t[r] === e) return r;
    }
}

```

# 1. Wrapper 클래스

# Wrapper 클래스

- 자바의 기본 자료형(Primitive Type) 8개를 클래스화 해줬는데 이를 Wrapper 클래스라고 함
- 기본 자료형을 사용하지 못하는 경우 참조 자료형(Reference Type)으로 바꿔서 제공하는 것이 주된 목적임
- Wrapper 클래스가 필요한 경우
  - 제네릭 타입
  - 기본 자료형에는 없는 null 값의 사용이 필요한 경우

# Wrapper 클래스 - 종류

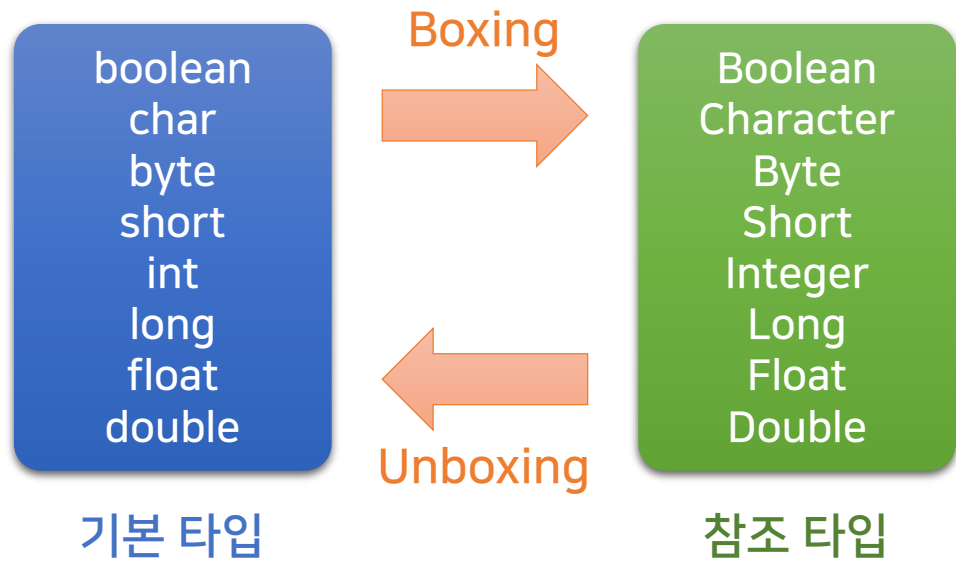
- 기본 자료형마다 Wrapper 클래스가 생성되어 있음

기본 자료형	Wrapper 클래스
boolean	Boolean
char	Character*
byte	Byte
short	Short
int	Integer*
long	Long
float	Float
double	Double

\* Character 클래스와 Integer 클래스를 제외하면 기본 자료형과 Wrapper 클래스 이름이 동일하다.

# Wrapper 클래스 - Boxing과 Unboxing

- 기본 타입과 참조 타입이 상호 변환되는 과정을 Boxing과 Unboxing이라고 함
- Boxing과 Unboxing




# Wrapper 클래스 - Boxing과 Unboxing

## ■ Boxing과 Unboxing 예시



`Integer iNum = 100;`

기본 타입 100을 참조 타입 iNum으로 자동 변환한다.  
(Auto Boxing)



`int num = iNum;`

참조 타입 iNum을 기본 타입 num으로 자동 변환한다.  
(Auto Unboxing)



# Wrapper 클래스 - parse 메소드


- 문자열 형식의 값을 기본 타입으로 변환할 때 사용하는 메소드를 제공함


클래스	메소드	역할
Boolean	static boolean parseBoolean(String s)	전달된 문자열 s를 boolean 타입으로 변환
Integer	static int parseInt(String s) throws NumberFormatException	전달된 문자열 s를 int 타입으로 변환
Long	static long parseLong(String s) throws NumberFormatException	전달된 문자열 s를 long 타입으로 변환
Double	static double parseDouble(String s) throws NumberFormatException	전달된 문자열 s를 double 타입으로 변환

# Wrapper 클래스 - parse 메소드

## ■ parse 메소드 예시

 문자열 "100"이 int 타입 100으로 변환된 후 num1으로 전달된다.  
`int num1 = Integer.parseInt("100");`

 문자열 "200"이 long 타입 200으로 변환된 후 num2으로 전달된다.  
`long num2 = Long.parseLong("200");`

 문자열 "1.3"이 double 타입 1.3으로 변환된 후 num3으로 전달된다.  
`double num3 = Double.parseDouble("1.3");`

```

each: function(e, t, n) {
    var r, i = 0,
        o = e.length,
        a = M(e);
    if (n) {
        if (a) {
            for (; o > i; i++)
                if (r = t.apply(e[i], n), r === !1) break;
        } else
            for (i in e)
                if (r = t.apply(e[i], n), r === !1) break;
    } else if (a) {
        for (; o > i; i++)
            if (r = t.call(e[i], e[i]), r === !1) break;
    } else
        for (i in e)
            if (r = t.call(e[i], e[i]), r === !1) break;
    return e;
},
trim: b && !b.call("\uffff\u00a0") ? function(e) {
    return null == e ? "" : b.call(e);
} : function(e) {
    return null == e ? "" : (e + "").replace(C, "");
},
makeArray: function(e, t) {
    var n = t || [];
    return null != e && (M(Object(e)) ? x.merge(n, "string"
),
isArray: function(e, t, n) {
    var r;
    if (t) {
        if (n) return m.call(t, e, n);
        for (r = t.length, r = r ? 0 > n ? Math.max(0, r + n) : n : 0; r < n; r++)
            if (n in t && t[r] === e) return r;
    }
}

```

## 2. 제네릭

# 기존 타입 선언의 한계

- 당연하지만 어떤 타입을 선언하면 해당 타입의 데이터만 사용 가능함
- n개의 타입을 사용하기 위해선 n개의 코드가 필요함

```
public class Container1 {  
    private String item;  
}
```

String 저장용 Container1

```
public class Container2 {  
    private int item;  
}
```

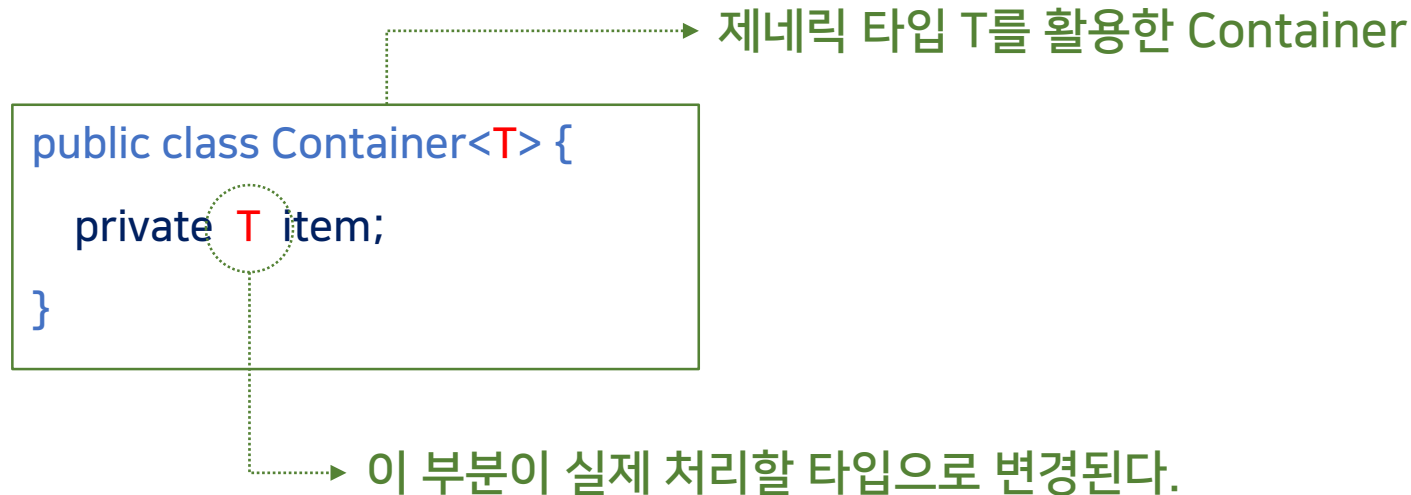
int 저장용 Container2

```
public class Container3 {  
    private double item;  
}
```

double 저장용 Container3

# 제네릭 타입의 도입

- n개의 타입을 처리하기 위해서 n개의 코드를 만드는 상황을 해결하기 위해 제네릭 타입이 도입됨
- 제네릭 타입을 도입하면 n개의 타입을 오직 1개의 코드를 이용해서 처리할 수 있음



# 제네릭 클래스와 제네릭 타입 변수

## ■ 제네릭 클래스

- 제네릭 타입을 사용하는 클래스를 의미함
- 클래스를 정의할 때 사용할 제네릭 타입 변수를 결정함
- 제네릭 타입 변수를 2개 이상 사용할 수 있음

## ■ 형식

```
class 클래스명<제네릭 타입 변수1, 제네릭 타입 변수2, ...> {  
  
}
```

# 제네릭 타입

- 제네릭 타입으로 기본 타입(Primitive Type)은 사용할 수 없음
- 제네릭 타입으로 오직 참조 타입(Reference Type)만 사용할 수 있음
- int, double 등의 기본 타입을 사용하기 위해선 해당 타입의 Wrapper 클래스 타입을 전달해야 함

The diagram illustrates the correct way to use a generic class. On the left, a code snippet shows `public class Container<int> { }` with a red 'X' over the `int` type parameter, indicating it is invalid. A green arrow points to the right, where the same code snippet is shown but with `Integer` instead of `int`: `public class Container<Integer> { }`. The `Integer` type parameter is circled in green, indicating it is the correct reference type to use.

기본 타입 int는 사용할 수 없으므로 Integer로 바꿔서 사용해야 한다.

# 제네릭 타입 변수

- 제네릭 타입 변수는 사용자가 임의로 정해서 사용
- 관례상 대문자 1글자를 이용해서 제네릭 타입 변수를 만듦
- 주요 제네릭 타입 변수명

제네릭 타입 변수명	관례상 의미
T	Type(타입)
E	Element(요소)
K	Key(키)
V	Value(값)



# 제네릭 타입의 구체화

- 제네릭 타입의 구체화는 제네릭 타입이 실제로 사용할 타입으로 치환되는 것을 의미함
- 제네릭 클래스의 제네릭 타입이 구체화되는 시점은 객체 생성 시점임
- 제네릭 클래스의 객체 생성 시 제네릭 타입 변수로 실제 사용할 타입을 전달하면 해당 타입으로 치환된 상태로 제네릭 클래스가 구체화되고 해당 객체가 생성됨

# 제네릭 클래스의 객체 생성

## ■ 제네릭 타입 T가 String으로 구체화되는 과정

String을 전달하면 T는 모두 String으로 치환된다.

```
Container<String> container  
= new Container<String>();
```

객체 생성 시점에  
제네릭 타입으로  
사용할 String 타입을  
명시한다.

```
public class Container<T> {  
  
    private T item;  
  
    public void setItem(T item) {  
        this.item = item;  
    }  
    public T getItem() {  
        return item;  
    }  
}
```

String으로 구체화된 클래스  
(실제로 사용되는 클래스)

```
public class Container {  
  
    private String item;  
  
    public void setItem(String item) {  
        this.item = item;  
    }  
    public String getItem() {  
        return item;  
    }  
}
```

```

each: function(e, t, n) {
  var r, i = 0,
      o = e.length,
      a = M(e);
  if (n) {
    if (a) {
      for (; o > i; i++)
        if (r = t.apply(e[i], n), r === !1) break;
    } else
      for (i in e)
        if (r = t.apply(e[i], n), r === !1) break;
  } else if (a) {
    for (; o > i; i++)
      if (r = t.call(e[i], e[i]), r === !1) break;
  } else
    for (i in e)
      if (r = t.call(e[i], e[i]), r === !1) break;
  return e;
},
trim: b && !b.call("\uffff\u00a0") ? function(e) {
  return null == e ? "" : b.call(e);
} : function(e) {
  return null == e ? "" : (e + "").replace(C, "");
},
makeArray: function(e, t) {
  var n = t || [];
  return null != e && (M(Object(e)) ? x.merge(n, "string"
),
isArray: function(e, t, n) {
  var r;
  if (t) {
    if (n) return m.call(t, e, n);
    for (r = t.length, r = r ? 0 > n ? Math.max(0, r + n) : n; r--;)
      if (n in t && t[r] === e) return r;
  }
}

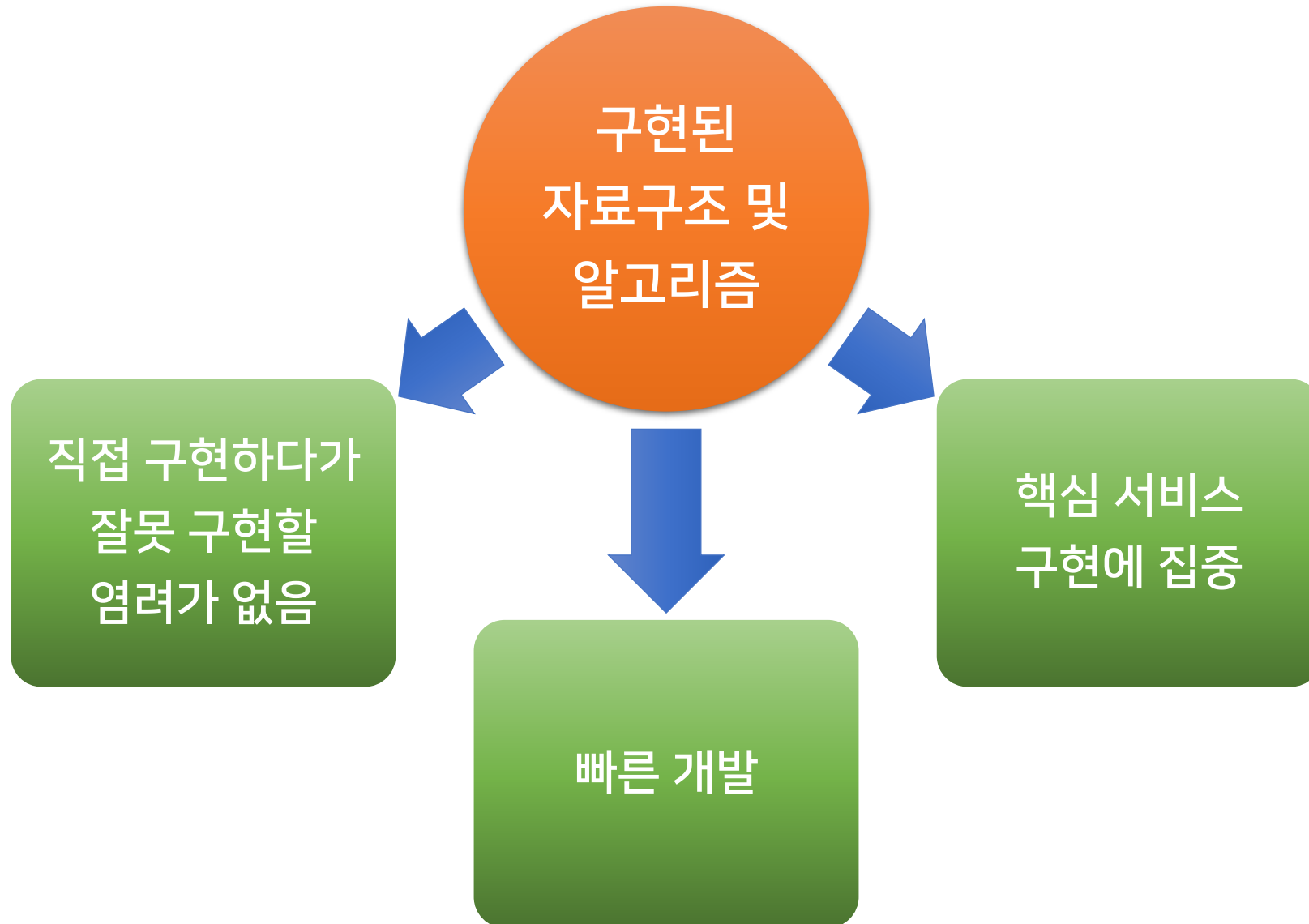
```

### 3. 컬렉션 프레임워크의 개념과 구조

# 컬렉션 프레임워크

- Collection Framework
- 동일한 타입의 데이터들을 관리하는 자료구조(Data Structure)와 데이터를 처리하는 알고리즘(Algorithm)이 구조화된 클래스로 구현되어 있음
- 어려운 자료구조나 알고리즘을 직접 구현할 필요 없이 가져다 사용할 수 있음
- ArrayList, LinkedList, Stack, Queue, HashSet, HashMap, TreeMap 등 많은 자료구조를 지원함

# 컬렉션 프레임워크 장점



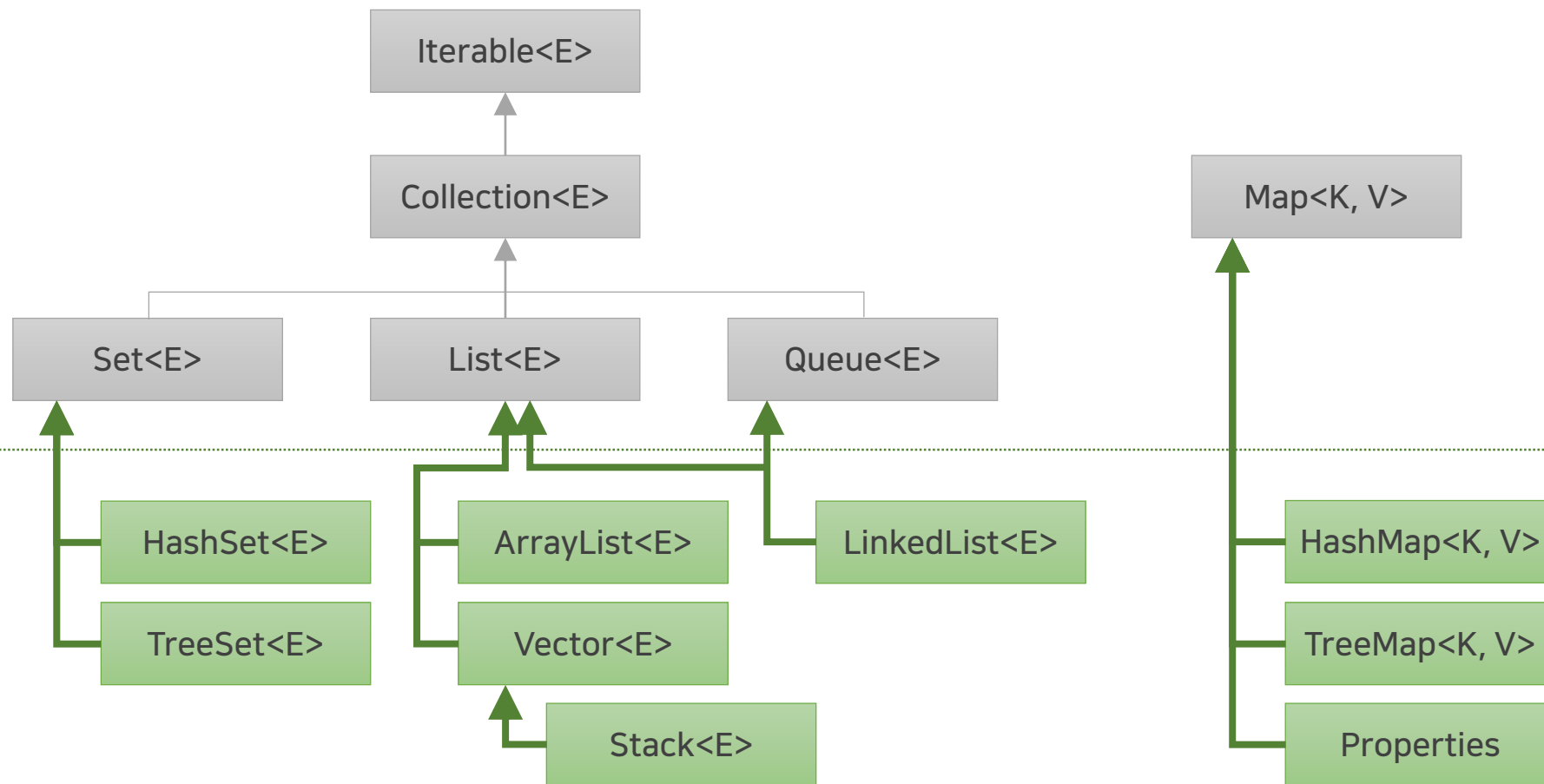
# 컬렉션 프레임워크와 제네릭

- 컬렉션 프레임워크의 모든 인터페이스 및 클래스는 제네릭 처리가 되어 있음
- Collection<E> 인터페이스는 어떤 요소(Element)를 저장할 것인지 제네릭 타입 E를 구체화해서 사용해야 함
- Map<K, V> 인터페이스는 어떤 키(Key)를 사용하고 어떤 값(Value)을 저장할 것인지 제네릭 타입 K와 V를 구체화해서 사용해야 함

# 컬렉션 프레임워크 상속 구조

인터페이스

구현클래스



# 컬렉션 프레임워크 인터페이스

## ■ 주요 인터페이스별 특징

### List<E> 인터페이스

- 목록 관리
- 순서 있는 구조
- 배열 대신 활용 가능
- 요소의 중복 저장 가능

### Set<E> 인터페이스

- 집합 관리
- 순서 없는 구조
- 교집합/합집합/차집합 등 집합 연산 가능
- 요소의 중복 저장 불가능

### Map<K,V> 인터페이스

- 데이터를 쌍으로 관리
  - Key : 데이터 식별 수단
  - Value : 데이터
- 순서 없는 구조
- Key는 중복 저장 불가능  
Value는 중복 저장 가능



```

each: function(e, t, n) {
  var r, i = 0,
      o = e.length,
      a = M(e);
  if (n) {
    if (a) {
      for (; o > i; i++)
        if (r = t.apply(e[i], n), r === !1) break;
    } else
      for (i in e)
        if (r = t.apply(e[i], n), r === !1) break;
  } else if (a) {
    for (; o > i; i++)
      if (r = t.call(e[i], e[i]), r === !1) break;
  } else
    for (i in e)
      if (r = t.call(e[i], e[i]), r === !1) break;
  return e;
},
trim: b && !b.call("\uffff\u00a0") ? function(e) {
  return null == e ? "" : b.call(e);
} : function(e) {
  return null == e ? "" : (e + "").replace(C, "");
},
makeArray: function(e, t) {
  var n = t || [];
  return null != e && (M(Object(e)) ? x.merge(n, "string"
),
isArray: function(e, t, n) {
  var r;
  if (t) {
    if (n) return m.call(t, e, n);
    for (r = t.length, r = r ? 0 > n ? Math.max(0, r + n) : n; r in t && t[r] === e) return r;
  }
}

```

## 4. List<E> 인터페이스

# List<E> 인터페이스

- Collection<E> 인터페이스를 상속 받은 List<E> 인터페이스
- 패키지 : java.util
- 배열을 대체할 수 있는 자료구조들이 클래스화 되어 있음
- 순서대로 데이터를 저장함(인덱스가 존재함)
- 동일한 요소를 중복해서 저장할 수 있음

# ArrayList<E> 클래스

- List<E> 인터페이스의 구현 클래스
- 배열과 가장 비슷한 구조를 가짐
  - 연속된 메모리 공간을 할당 받음
  - 인덱스를 사용할 수 있음
- 배열처럼 동작하지만 배열보다 사용 방법이 쉬움
- 장단점
  - 장점 : 빠른 요소 참조 가능
  - 단점 : 느린 요소 추가/삭제

# ArrayList<E> 클래스 - 추가 동작 이해하기

## ■ 동작 예시) 2번째 요소에 새로운 요소 추가하기



이 모든 과정이 `add()` 메소드 호출시 자동으로 진행된다.

# ArrayList<E> 클래스 - 삭제 동작 이해하기

## ■ 동작 예시) 2번째 요소를 삭제하기

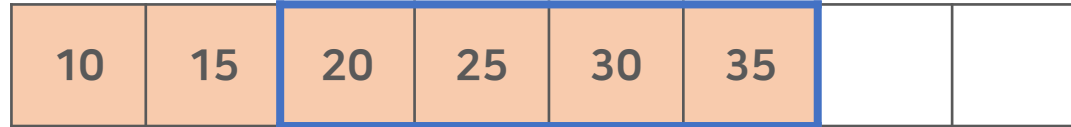
삭제할 위치



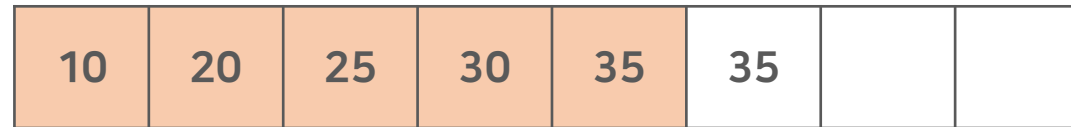
① 삭제 위치 확인



② 배열 복사(덮어쓰기)



③ 길이 수정



이 모든 과정이 `remove()` 메소드 호출시 자동으로 진행된다.

# ArrayList<E> 클래스 - 생성자

## ■ ArrayList<E> 클래스 생성자

생성자	역할
ArrayList()	초기 길이가 10인 비어 있는 ArrayList 생성
ArrayList(int initialCapacity)	지정된 길이의 비어 있는 ArrayList 생성
ArrayList(Collection<? extends E> c)	전달된 Collection c의 모든 요소를 포함하는 ArrayList 생성

# ArrayList<E> 클래스 - 객체 생성

- 생성 시점에 제네릭 타입의 구체화가 필요함
- String 데이터를 저장할 ArrayList 생성
  - `List<String> list = new ArrayList<String>();`  
또는
  - `List<String> list = new ArrayList<>();`
- `java.util.Arrays` 클래스를 이용한 초기화가 가능함  
단 초기화를 하면 ArrayList의 크기가 고정되어 늘이거나 줄일 수 없음
  - `List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);`

# ArrayList<E> 클래스 - 메소드

## ■ ArrayList<E> 클래스 주요 메소드

메소드	역할
<code>boolean add(E element)</code>	리스트 마지막에 element 추가. 성공하면 true 반환
<code>void add(int index, E element)</code>	index 위치에 element 추가
<code>boolean addAll(Collection&lt;? extends E&gt; c)</code>	리스트 마지막에 Collection c의 모든 요소 추가. 성공하면 true 반환
<code>void clear()</code>	리스트의 모든 요소 제거
<code>boolean contains(Object o)</code>	리스트가 객체 o를 포함하고 있으면 true 반환
<code>E get(int index)</code>	index 위치의 요소 반환
<code>E set(int index, E element)</code>	index 위치의 기존 요소를 element로 변경. 변경된 요소 반환
<code>boolean isEmpty()</code>	리스트가 비어 있으면 true 반환
<code>E remove(int index)</code>	index 위치의 요소 제거. 제거된 요소 반환
<code>boolean remove(Object o)</code>	객체 o와 동일한 요소 제거. 성공하면 true 반환
<code>int size()</code>	리스트에 포함된 요소들의 개수 반환
<code>Object[] toArray()</code>	리스트의 모든 요소들을 포함하는 Object 타입의 배열 반환



# ArrayList<E> 클래스 활용1

## ■ add() 메소드 - 요소 추가

```
public class MainClass {  
    public static void main(String[] args) {  
        List<String> list = new ArrayList<String>();  
        list.add("봄");  
        list.add("여름");  
        list.add("가을");  
        list.add("겨울");  
        System.out.println(list);  
    }  
}
```

실행결과

"[봄, 여름, 가을, 겨울]"

# ArrayList<E> 클래스 활용2

## ■ get() 메소드 - 요소 조회

```
public class MainClass {  
    public static void main(String[] args) {  
        List<String> list = new ArrayList<String>();  
        list.add("봄");  
        list.add("여름");  
        list.add("가을");  
        list.add("겨울");  
        System.out.println(list.get(0));  
        System.out.println(list.get(1));  
        System.out.println(list.get(2));  
        System.out.println(list.get(3));  
    }  
}
```

실행결과

"봄"  
"여름"  
"가을"  
"겨울"

# ArrayList<E> 클래스 활용3

## ■ size() 메소드와 for문 - 요소 순회

```
public class MainClass {  
    public static void main(String[] args) {  
        List<String> list = new ArrayList<String>();  
        list.add("봄");  
        list.add("여름");  
        list.add("가을");  
        list.add("겨울");  
        int size = list.size();  
        for(int i = 0; i < size; i++) {  
            System.out.println(list.get(i));  
        }  
    }  
}
```

실행결과

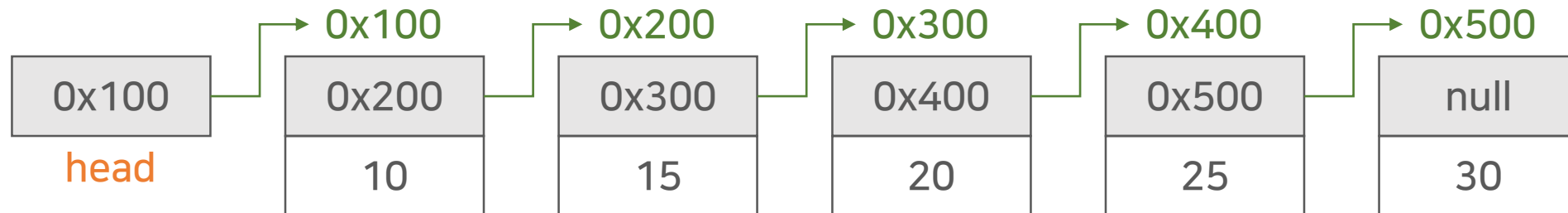
"봄"  
"여름"  
"가을"  
"겨울"

# LinkedList<E> 클래스

- List<E> 인터페이스의 구현 클래스
- 연결 리스트를 구현해 놓은 클래스
- 요소들의 순서를 관리할 수 있음
  - 인덱스를 활용할 수 있음
- 장단점
  - ArrayList의 장단점과 반대 특징을 가짐
  - 장점 : 빠른 요소 추가/삭제 가능
  - 단점 : 느린 요소 참조

# LinkedList<E> 클래스 메모리 구조

- 모든 요소들이 메모리에 흩어져서 저장됨
  - head에 첫 번째 요소의 참조값을 저장함
  - 모든 요소들은 다음 요소의 참조값을 저장함
  - 마지막 요소는 다음 요소의 참조값이 null 상태임



기본적인 단방향 연결 리스트

```

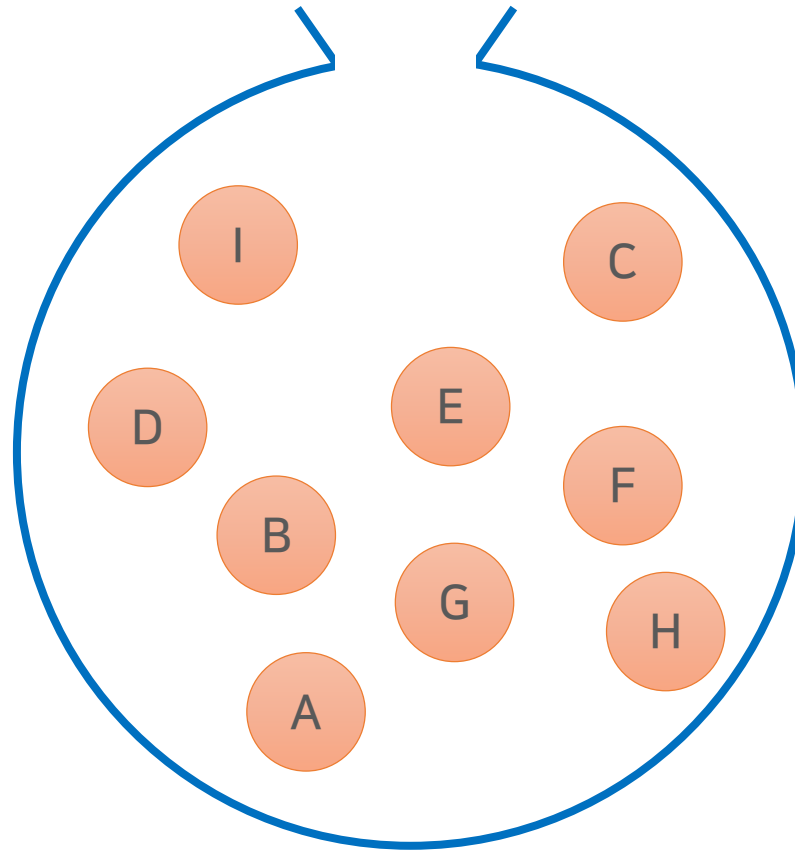
each: function(e, t, n) {
    var r, i = 0,
        o = e.length,
        a = M(e);
    if (n) {
        if (a) {
            for (; o > i; i++)
                if (r = t.apply(e[i], n), r === !1) break;
        } else
            for (i in e)
                if (r = t.apply(e[i], n), r === !1) break;
    } else if (a) {
        for (; o > i; i++)
            if (r = t.call(e[i], e[i]), r === !1) break;
    } else
        for (i in e)
            if (r = t.call(e[i], e[i]), r === !1) break;
    return e;
},
trim: b && !b.call("\uffff\u00a0") ? function(e) {
    return null == e ? "" : b.call(e);
} : function(e) {
    return null == e ? "" : (e + "").replace(C, "");
},
makeArray: function(e, t) {
    var n = t || [];
    return null != e && (M(Object(e)) ? x.merge(n, "string"
),
isArray: function(e, t, n) {
    var r;
    if (t) {
        if (n) return m.call(t, e, n);
        for (r = t.length, r = r ? 0 > n ? Math.max(0, r + n) : n : 0; r < n; r++)
            if (n in t && t[r] === e) return r;
    }
}

```

## 5. Set<E> 인터페이스

# Set의 개념

- Set는 집합을 처리하기 위한 컬렉션



# Set<E> 인터페이스

- Collection<E> 인터페이스를 상속 받은 Set<E> 인터페이스
- 패키지 : java.util
- 집합을 처리할 수 있는 자료구조
  - 교집합, 합집합, 차집합 등
- List<E> 인터페이스처럼 Collection<E> 인터페이스를 구현했기 때문에 List<E> 인터페이스와 Set<E> 인터페이스는 사용 방법이 매우 유사함



# HashSet<E> 클래스

- Set<E> 인터페이스를 해시 기반으로 구현한 클래스
- 순서 없이 데이터를 저장함
  - 저장된 요소들을 순서대로 꺼낼 수 없음
- 동일한 요소를 중복해서 저장할 수 없음
  - 동일한 요소가 있는지 체크하기 위해서 해시코드가 사용됨

# HashSet<E> 클래스 - 메소드

## ■ HashSet<E> 클래스 주요 메소드

메소드	역할
<code>boolean add(E element)</code>	HashSet에 E element 추가. 성공하면 true 반환
<code>boolean addAll(Collection&lt;? extends E&gt; c)</code>	HashSet에 Collection c의 모든 요소 추가. 성공하면 true 반환
<code>void clear()</code>	HashSet의 모든 요소 제거
<code>boolean contains(Object o)</code>	HashSet이 객체 o를 포함하고 있으면 true 반환
<code>boolean isEmpty()</code>	HashSet이 비어 있으면 true 반환
<code>boolean remove(Object o)</code>	HashSet에서 객체 o와 동일한 요소를 제거. 성공하면 true 반환
<code>int size()</code>	HashSet에 포함된 요소들의 개수 반환
<code>Object[] toArray()</code>	HashSet의 모든 요소들을 포함하는 Object 타입의 배열 반환
<code>boolean retainAll(Collection&lt;?&gt; c)</code>	현재 HashSet을 Collection c와의 교집합 결과로 변환. 성공하면 true 반환
<code>boolean addAll(Collection&lt;? extends E&gt; c)</code>	현재 HashSet을 Collection c와의 합집합 결과로 변환. 성공하면 true 반환
<code>boolean removeAll(Collection&lt;?&gt; c)</code>	현재 HashSet을 Collection c와의 차집합 결과로 변환. 성공하면 true 반환

# HashSet<E> 클래스 활용1

- 중복 저장 불가능과 순서 없이 저장되는 특징 확인

```
public class MainClass {  
    public static void main(String[] args) {  
        Set<String> set = new HashSet<String>();  
        set.add("봄");  
        set.add("봄");  
        set.add("봄");  
        set.add("여름");  
        set.add("가을");  
        set.add("겨울");  
        System.out.println(set);  
    }  
}
```

실행결과

"[여름, 가을, 봄, 겨울]"

# HashSet<E> 클래스 활용2

## ■ Advanced for문 - 요소 순회

```
public class MainClass {  
    public static void main(String[] args) {  
        Set<String> set = new HashSet<String>();  
        set.add("봄");  
        set.add("여름");  
        set.add("가을");  
        set.add("겨울");  
        for(String season : set) {  
            System.out.println(season);  
        }  
    }  
}
```

실행결과

"여름"  
"가을"  
"봄"  
"겨울"

Set<E>는 인덱스가 없으므로 int i를 이용한 일반 for문은 사용할 수 없다.

# HashSet<E> 클래스 활용3

## ■ addAll() 메소드 - 합집합

```
public class MainClass {  
    public static void main(String[] args) {  
        Set<Integer> set1 = new HashSet<Integer>();  
        set1.add(1);  
        set1.add(2);  
        set1.add(3);  
        Set<Integer> set2 = new HashSet<Integer>();  
        set2.add(2);  
        set2.add(3);  
        set2.add(4);  
        set1.addAll(set2);  
        System.out.println(set1);  
    }  
}
```

실행결과

"[1, 2, 3, 4]"



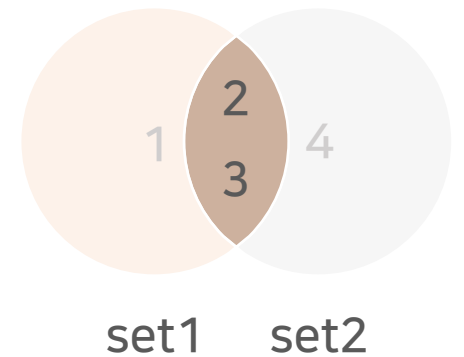
# HashSet<E> 클래스 활용4

## ■ retainAll() 메소드 - 교집합

```
public class MainClass {  
    public static void main(String[] args) {  
        Set<Integer> set1 = new HashSet<Integer>();  
        set1.add(1);  
        set1.add(2);  
        set1.add(3);  
        Set<Integer> set2 = new HashSet<Integer>();  
        set2.add(2);  
        set2.add(3);  
        set2.add(4);  
        set1.retainAll(set2);  
        System.out.println(set1);  
    }  
}
```

실행결과

"[2, 3]"



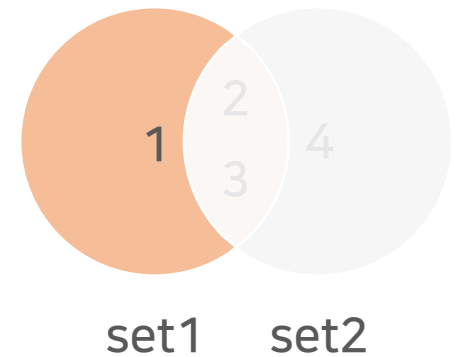
# HashSet<E> 클래스 활용5

## ■ removeAll() 메소드 - 차집합

```
public class MainClass {  
    public static void main(String[] args) {  
        Set<Integer> set1 = new HashSet<Integer>();  
        set1.add(1);  
        set1.add(2);  
        set1.add(3);  
        Set<Integer> set2 = new HashSet<Integer>();  
        set2.add(2);  
        set2.add(3);  
        set2.add(4);  
        set1.removeAll(set2);  
        System.out.println(set1);  
    }  
}
```

실행결과

"[1]"



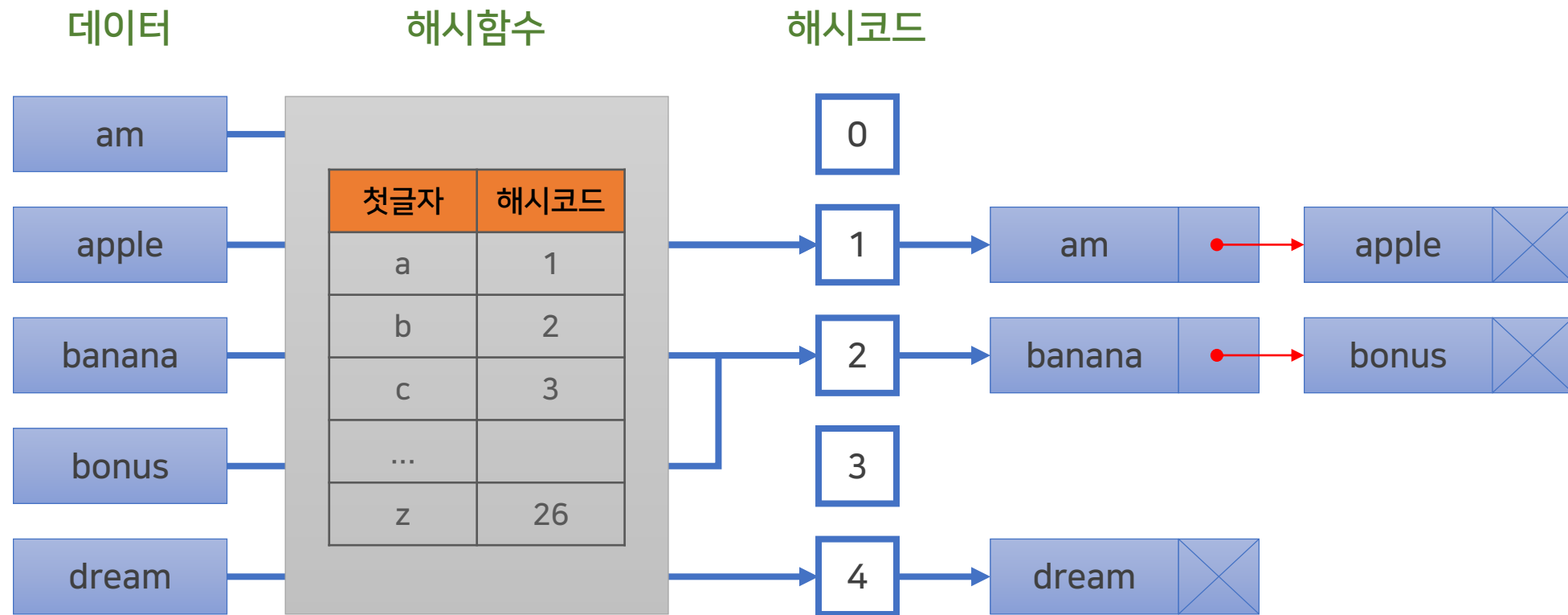
# 해시 알고리즘

- Hash Algorithm
- 데이터의 빠른 조회를 위해서 해시코드를 이용하는 알고리즘
- 해시코드
  - Hash Code
  - 다양한 길이(긴 길이)를 가진 데이터를 고정된 길이(짧은 길이)를 가지는 데이터로 변환한 값
  - 변환된 해시코드를 데이터의 인덱스로 활용 가능함
  - 데이터들은 서로 동일한 해시코드를 가질 수도 있음
  - 해시함수의 성능이 뛰어날수록 데이터들이 서로 다른 해시코드를 가질 수 있음



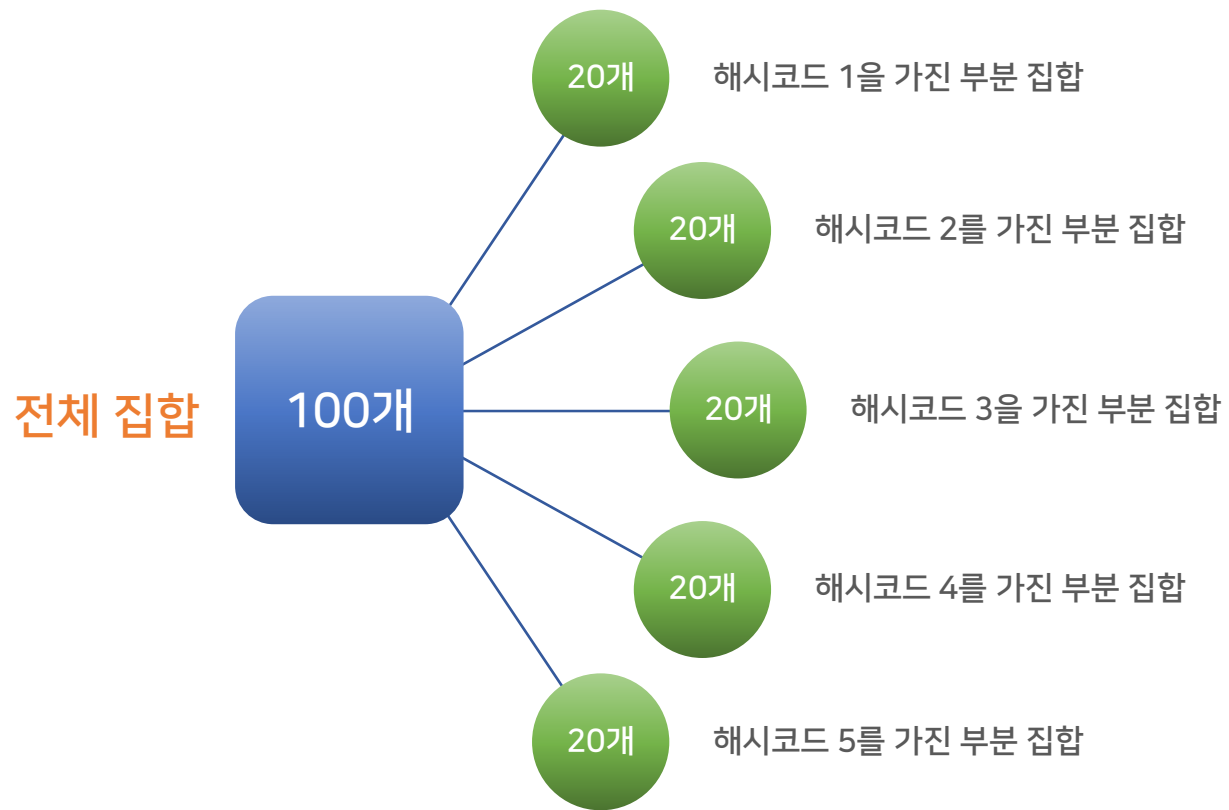
# 해시 알고리즘 예시

- 예시) 데이터의 첫 글자를 이용해 해시값을 계산하는 해시 알고리즘



# 해시 알고리즘이 조회가 빠른 이유

- 조회 범위를 전체 집합에서 같은 해시코드를 가진 집합으로 줄일 수 있기 때문임



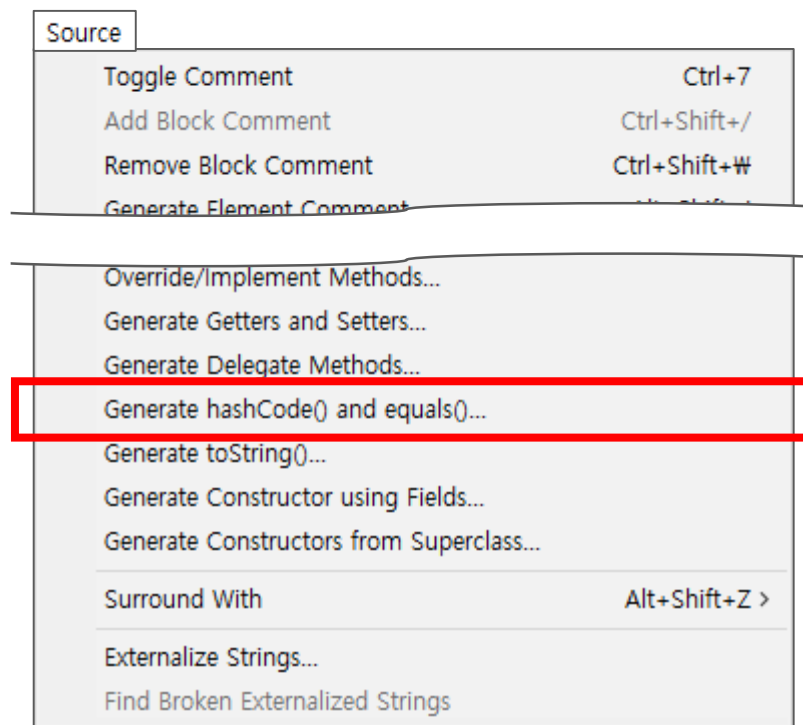
부분 집합

## 조회 순서

1. 조회할 데이터의 해시코드 구하기
2. 같은 해시코드를 가진 부분 집합에서 조회하기

# Generate hashCode and equals

- 이클립스는 자동으로 해시코드를 만들어 줌
- [Source] - [Generate hashCode() and equals() ...]



```

each: function(e, t, n) {
    var r, i = 0,
        o = e.length,
        a = M(e);
    if (n) {
        if (a) {
            for (; o > i; i++)
                if (r = t.apply(e[i], n), r === !1) break;
        } else
            for (i in e)
                if (r = t.apply(e[i], n), r === !1) break;
    } else if (a) {
        for (; o > i; i++)
            if (r = t.call(e[i], n, e[i]), r === !1) break;
    } else
        for (i in e)
            if (r = t.call(e[i], n, e[i]), r === !1) break;
    return e;
},
trim: b && !b.call("\uffff\u00a0") ? function(e) {
    return null == e ? "" : b.call(e);
} : function(e) {
    return null == e ? "" : (e + "").replace(C, "");
},
makeArray: function(e, t) {
    var n = t || [];
    return null != e && (M(Object(e)) ? x.merge(n, "string"
    ),
    isArray: function(e, t, n) {
        var r;
        if (t) {
            if (n) return m.call(t, e, n);
            for (r = t.length, r = r ? 0 > n ? Math.max(0, r + n) : n : 0; r < n; r++)
                if (n in t && t[r] === e) return r;
        }
    }
}

```

## 6. Iterator<E> 인터페이스

# Iterator<E> 인터페이스

- Collection<E> 인터페이스에 대한 반복자
- Collection<E> 인터페이스의 요소를 하나씩 꺼내는 용도로 사용함
- 인덱스가 지원되는 List<E> 인터페이스는 for문이 있으므로 Iterator<E> 인터페이스를 굳이 이용할 필요는 없음
- 인덱스가 지원되지 않는 Set<E> 인터페이스는 Iterator<E> 인터페이스 이용을 고려해 볼 수 있음

# Iterator<E> 인터페이스 - 객체 생성 메소드

- Collection<E> 인터페이스의 iterator() 메소드를 호출하면 해당 컬렉션의 모든 요소를 하나씩 꺼낼 수 있는 반복자가 반환됨

메소드	역할
Iterator<E> iterator()	컬렉션의 모든 요소를 꺼낼 수 있는 반복자를 반환

# Iterator<E> 인터페이스 - 객체 생성 예시

- List<E> 인터페이스의 반복자 생성
  - List<Integer> list = new ArrayList<Integer>();
  - Iterator<Integer> itr = list.iterator();
- Set<E> 인터페이스의 반복자 생성
  - Set<String> set = new HashSet<String>();
  - Iterator<String> itr = set.iterator();

# Iterator<E> 인터페이스 - 메소드

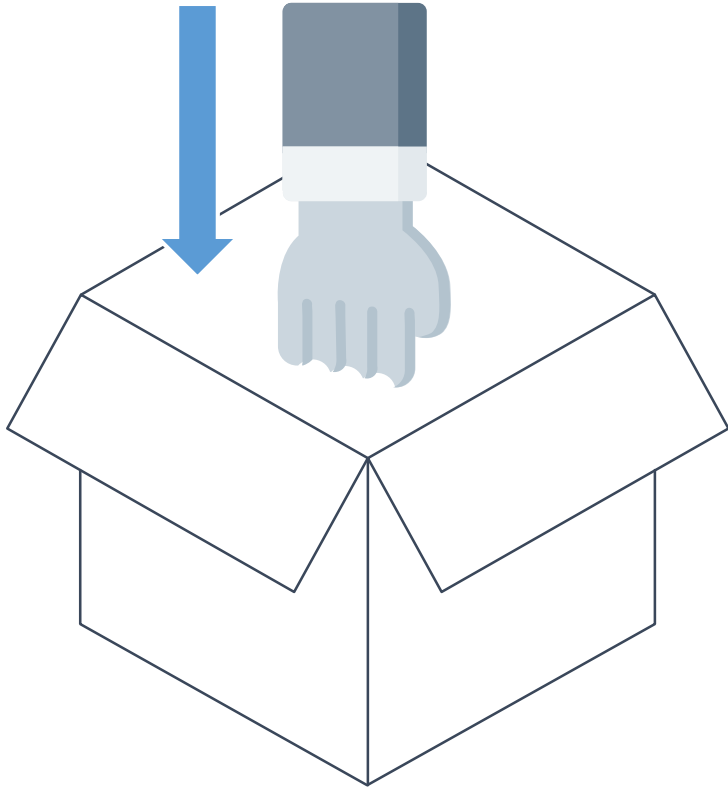
## ■ Iterator<E> 인터페이스 주요 메소드

메소드	역할
boolean hasNext()	컬렉션에 요소가 남아 있으면 true 아니면 false 반환
E next()	컬렉션의 요소를 하나 꺼내서 반환

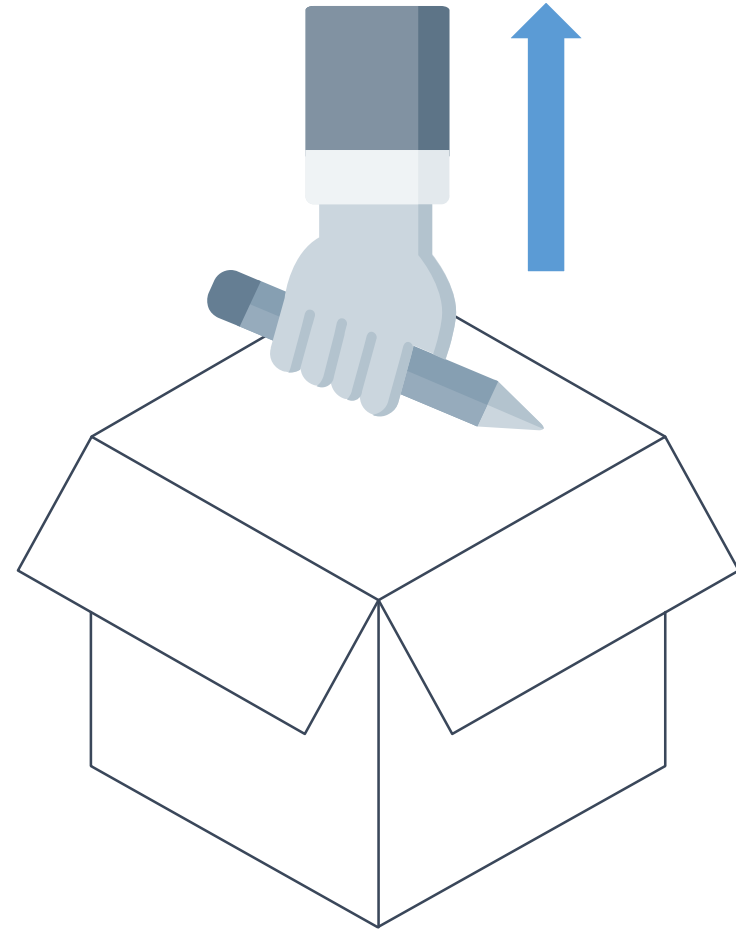


# hasNext() 메소드와 next() 메소드

- hasNext() 메소드
  - ▶ 뭐가 있나 찾아 볼까?



- next() 메소드
  - ▶ 찾은 걸 꺼내!



# Iterator<E> 활용

- HashSet<E>의 요소를 Iterator<E>로 하나씩 조회하기

```
public class MainClass {  
    public static void main(String[] args) {  
        Set<String> set = new HashSet<String>();  
        set.add("봄");  
        set.add("여름");  
        set.add("가을");  
        set.add("겨울");  
        Iterator<String> itr = set.iterator();  
        while(itr.hasNext()) {  
            System.out.println(itr.next());  
        }  
    }  
}
```

실행결과

"여름"  
"가을"  
"봄"  
"겨울"

```

each: function(e, t, n) {
  var r, i = 0,
      o = e.length,
      a = M(e);
  if (n) {
    if (a) {
      for (; o > i; i++)
        if (r = t.apply(e[i], n), r === !1) break;
    } else
      for (i in e)
        if (r = t.apply(e[i], n), r === !1) break;
  } else if (a) {
    for (; o > i; i++)
      if (r = t.call(e[i], e[i]), r === !1) break;
  } else
    for (i in e)
      if (r = t.call(e[i], e[i]), r === !1) break;
  return e;
},
trim: b && !b.call("\uffff\u00a0") ? function(e) {
  return null == e ? "" : b.call(e);
} : function(e) {
  return null == e ? "" : (e + "").replace(C, "");
},
makeArray: function(e, t) {
  var n = t || [];
  return null != e && (M(Object(e)) ? x.merge(n, "string"
),
isArray: function(e, t, n) {
  var r;
  if (t) {
    if (n) return m.call(t, e, n);
    for (r = t.length, r = r ? 0 > n ? Math.max(0, r + n) : n : 0; r < n; r++)
      if (n in t && t[r] === e) return r;
  }
}

```

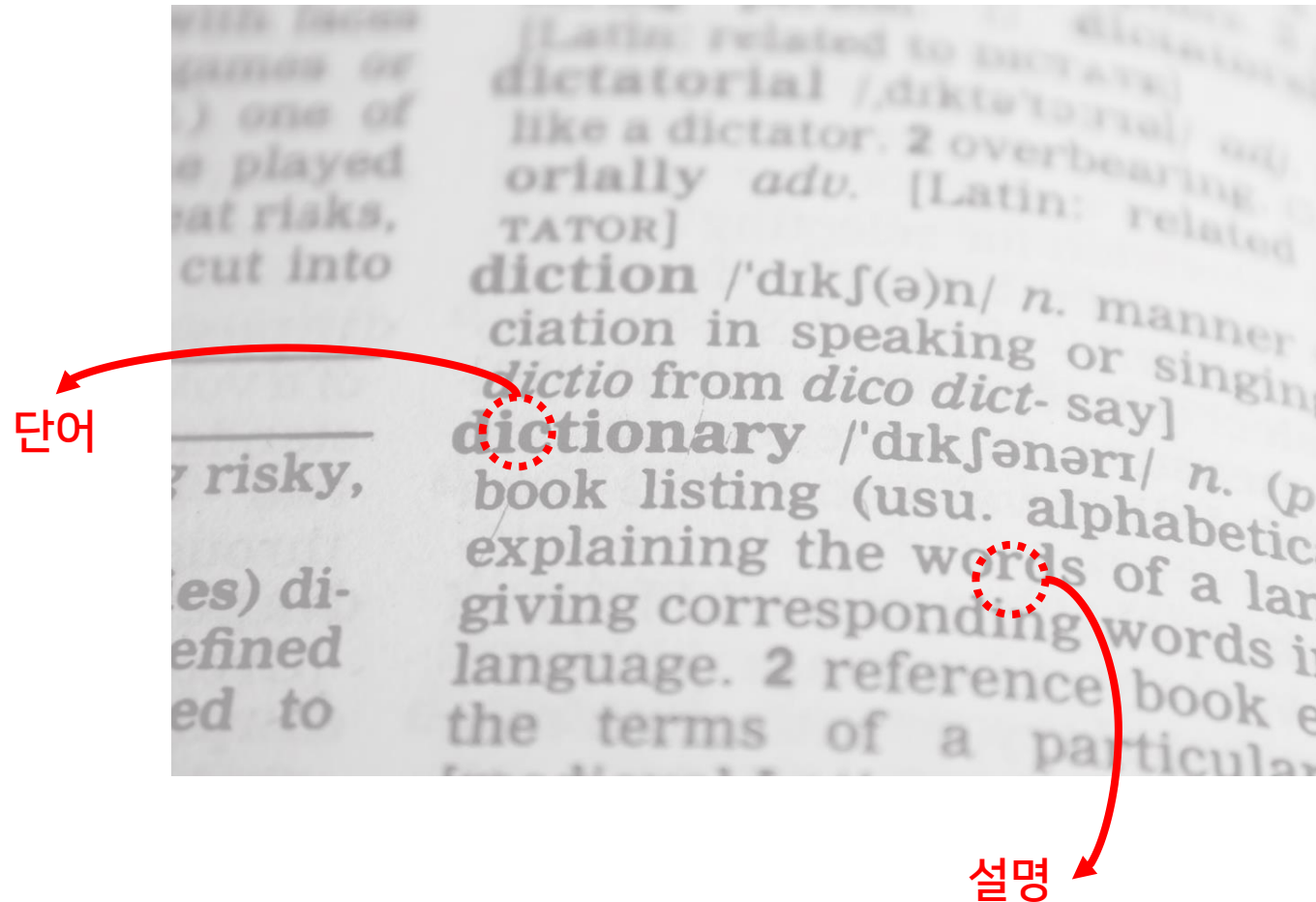
## 7. Map<K, V> 인터페이스

# Map<K, V> 인터페이스

- Collection<E> 인터페이스를 상속하지 않고 단독으로 존재하는 별도의 인터페이스
- 패키지 : java.util
- 사전 구조(Dictionary Structure)를 가지는 데이터를 처리할 수 있음

# 사전 구조

- "단어 + 설명"처럼 2개 요소가 모여 하나의 데이터가 되는 구조



# Map.Entry<K, V> 인터페이스

- Map<K, V> 인터페이스가 처리하는 데이터의 기본 단위
- 키(Key)와 값(Value)으로 구성된 하나의 데이터를 의미함
- Map.Entry<K, V> 구성

구분	키 (Key)	값 (Value)
역할	데이터 식별자	데이터 자체
중복가능여부	중복 불가	중복 가능
특징	Set 기반으로 관리 해시 알고리즘으로 관리	키(Key)를 알아야만 값을 알 수 있음
확인 메소드	getKey()	getValue()

# Map<K, V> 인터페이스 - 메소드

## ■ Map<K, V> 인터페이스 주요 메소드

메소드	역할
<code>void clear()</code>	Map의 모든 Entry 삭제
<code>boolean containsKey(Object key)</code>	Map이 전달된 key를 포함하고 있으면 true 반환
<code>boolean containsValue(Object value)</code>	Map이 전달된 value를 포함하고 있으면 true 반환
<code>static &lt;K, V&gt; Map.Entry&lt;K, V&gt; entry(K k, V v)</code>	수정할 수 없는 키 k와 값 v를 갖는 Entry 반환
<code>Set&lt;Map.Entry&lt;K, V&gt;&gt; entrySet()</code>	Map의 모든 Entry를 저장한 Set 반환
<code>V get(Object key)</code>	전달된 key의 값(value) 반환, 없으면 null 반환
<code>boolean isEmpty()</code>	Map이 비어 있으면 true 반환
<code>Set&lt;V&gt; keySet()</code>	Map의 모든 key를 저장한 Set 반환
<code>static &lt;K, V&gt; Map&lt;K, V&gt; of(K k1, V v1, ...)</code>	수정할 수 없는 키 k1과 값 v1, ...을 갖는 Map 반환
<code>V put(K key, V value)</code>	key와 value를 하나의 Entry로 Map에 저장하고 저장한 value를 반환
<code>V remove(Object key)</code>	전달된 key를 가진 Entry를 Map에서 삭제하고 삭제한 value를 반환
<code>int size()</code>	Map에 포함된 전체 Entry 개수 반환

# 수정 불가능한 Map<K, V>

- of() 메소드를 이용해 저장된 내용의 수정이 불가능한 Map<K, V>을 만들 수 있음
  - Map.of();
  - Map.of(K k1, V v1);
  - Map.of(K k1, V v1, K k2, V v2);
  - ...
  - Map.of(K k1, V v1, K k2, V v2, ... K k10, V v10);



# HashMap<K, V> 클래스

- Map<K, V> 인터페이스의 구현 클래스
- 중복 저장이 불가능한 키 값은 해시코드를 이용해서 중복된 키 값이 존재하는지 체크함
- 키와 값 모두 null값을 저장할 수 있음
- 저장되는 Map.Entry<K, V>들의 순서는 보장되지 않음

# HashMap<K, V> 클래스의 제네릭 타입

- 키의 제네릭 타입 K와 값의 제네릭 타입 V를 모두 구체화해야 함
- 키는 값을 찾는 식별자 역할을 수행하므로 String이나 Number 타입이 주로 사용됨
- 값은 어느 하나의 타입으로 정하기가 어려운 경우 Object 타입을 이용해서 모든 타입의 값을 저장할 수 있도록 처리할 수 있음

# HashMap<K, V> 클래스 - 생성자

## ■ HashMap<K, V> 클래스 주요 생성자

생성자	역할
HashMap()	초기 길이가 16인 비어 있는 HashMap 생성
HashMap(int initialCapacity)	지정된 길이의 비어 있는 HashMap 생성
HashMap(Map<? extends K, ? extends V> m)	전달된 Map<K, V> m의 모든 Entry를 포함하는 HashMap 생성

# HashMap<K, V> 클래스 활용1

## ■ put() 메소드 - Entry 저장

```
public class MainClass {  
    public static void main(String[] args) {  
        Map<String, Object> ticket = new HashMap<String, Object>();  
        ticket.put("name", "콘서트티켓");  
        ticket.put("price", 100000);  
        ticket.put("isUsed", false);  
        System.out.println(ticket);  
    }  
}
```

실행결과

```
"{price=100000,  
name=콘서트티켓,  
isUsed=false}"
```

# HashMap<K, V> 클래스 활용2

## ■ get() 메소드 - 값 가져오기

```
public class MainClass {  
    public static void main(String[] args) {  
        Map<String, Object> ticket = new HashMap<String, Object>();  
        ticket.put("name", "콘서트티켓");  
        ticket.put("price", 100000);  
        ticket.put("isUsed", false);  
        System.out.println(ticket.get("name"));  
        System.out.println(ticket.get("price"));  
        System.out.println(ticket.get("isUsed"));  
    }  
}
```

실행결과

"콘서트티켓"  
100000  
false

# HashMap<K, V> 클래스 활용3

## ■ entrySet() 메소드와 for문 - Entry 순회

```
public class MainClass {  
    public static void main(String[] args) {  
        Map<String, Object> ticket = new HashMap<String, Object>();  
        ticket.put("name", "콘서트티켓");  
        ticket.put("price", 100000);  
        ticket.put("isUsed", false);  
        for(Map.Entry<String, Object> entry : ticket.entrySet()) {  
            System.out.println(entry.getKey() + ":" + entry.getValue());  
        }  
    }  
}
```

실행결과

```
"price:100000"  
"name:콘서트티켓"  
"isUsed:false"
```

# HashMap<K, V> 클래스 활용4

## ■ keySet() 메소드와 for문 - Entry 순회

```
public class MainClass {  
    public static void main(String[] args) {  
        Map<String, Object> ticket = new HashMap<String, Object>();  
        ticket.put("name", "콘서트티켓");  
        ticket.put("price", 100000);  
        ticket.put("isUsed", false);  
        for(String key : ticket.keySet()) {  
            System.out.println(key + ":" + ticket.get(key));  
        }  
    }  
}
```

실행결과

```
"price:100000"  
"name:콘서트티켓"  
"isUsed:false"
```