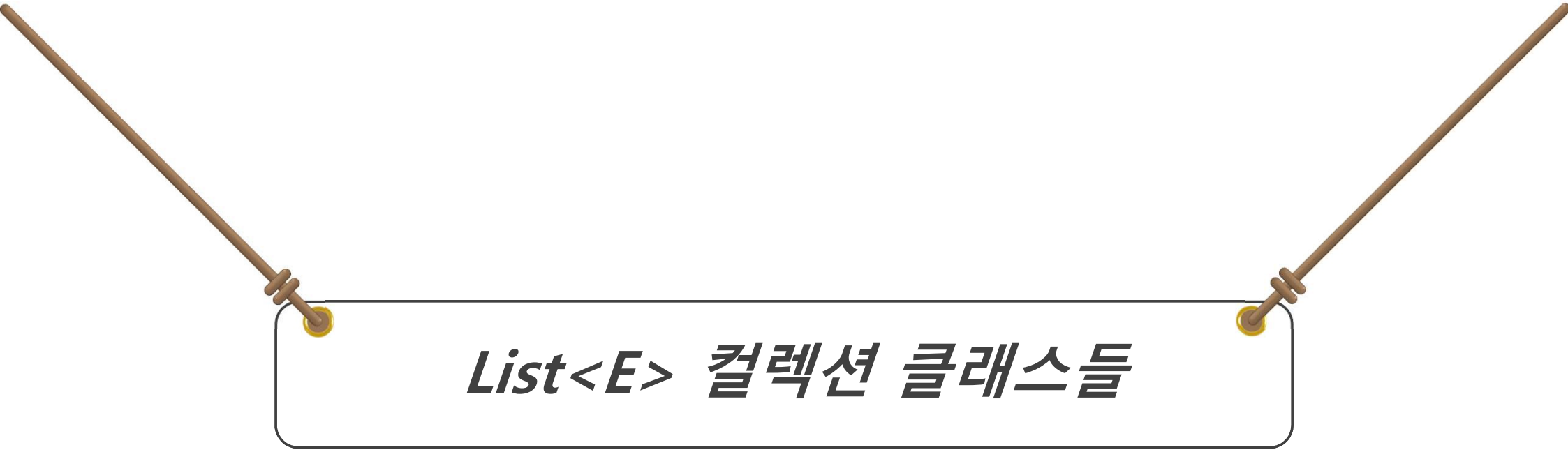
List<E> 컬렉션 클래스들

List<E> 컬렉션 클래스들

- ArrayList<E> 클래스
  - List<E> 인터페이스를 구현하는 대표적인 컬렉션 클래스
    - ArrayList<E> : 배열 기반 자료구조, 배열을 이용하여 인스턴스 저장
    - LinkedList<E> : 리스트 기반 자료구조, 리스트를 구성하여 인스턴스 저장
  - 위 두개의 클래스는 기능적으로 완전 동일하지만 인스턴스를 저장하는 방식에 차이가 있어 장단점 존재
  - List<E> 인터페이스를 구현하는 컬렉션 클래스들이 갖는 공통적인 특성
    - 인스턴스의 저장 순서 유지
    - 동일한 인스턴스의 중복 저장 허용

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

public class ArrayListCollection {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Toy");
        list.add("Box");
        list.add("Robot");
        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }
        System.out.println();
        list.remove(0);
        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }
    }
}
```

List<E> 컬렉션 클래스들

- List, ArrayList 클래스 사용을 위해 import 필요
- ArrayList<E> 인스턴스의 생성문 : List<String> list = new ArrayList<>();
- ArrayList<E> 형 참조변수가 아닌 List<E>형 참조변수를 선언한 이유
  - 코드에 유연성을 주기 위해 -> 클래스의 교체 용이
  - List<String> list = new ArrayList<>(); -> List<String> list = new LinkedList<>();
- for문과 인스턴스의 get(인덱스값)을 통해 list의 데이터 호출 가능
- add 메소드 : 인스턴스에 데이터 추가
- remove 메소드 : 인덱스에 해당하는 데이터 삭제
- 내부적으로 배열을 생성해서 인스턴스를 저장하는데, 필요시 그 배열의 길이를 스스로 추가
- 단, 한번 생성된 배열은 길이를 늘릴 수 없으므로 배열의 길이를 늘린다는 것은 더 긴 배열로의 교체를 의미하며 이는 성능의 저하로 이루어 짐
- 만약, 배열의 길이가 계산된다면 public ArrayList(int initialCapacity) 생성자를 통해 배열의 길이를 정하여 성능 향상 가능

List<E> 컬렉션 클래스들

- LinkedList<E> 클래스

```
import java.util.LinkedList;
import java.util.List;

public class LinkedListCollection {
    public static void main(String[] args) {
        List<String> list = new LinkedList<>();
        list.add("Toy");
        list.add("Box");
        list.add("Robot");
        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }
        System.out.println();
        list.remove(0);
        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }
    }
}
```

- LinkedList<E>는 "연결 리스트(Linked List)"라는 자료구조를 기반으로 디자인된 클래스
- ArrayList<E>와 달리 저장 공간을 계속 추가할 수 있기 때문에 인스턴스의 저장 공간을 미리 마련할 필요 없음
- 다음과 같은 두 가지 특성을 가짐
  - 인스턴스 저장 : 공간을 하나 추가하고 인스턴스 저장
  - 인스턴스 삭제 : 해당 인스턴스를 저장하고 있는 공간 삭제

List<E> 컬렉션 클래스들

- ArrayList<E> vs LinkedList<E>
  - ArrayList<E>의 장단점
    - 단점 : 저장 공간을 늘리고 삭제하는 과정에서 많은 연산이 필요할 수 있어 느릴 수 있음
    - 장점 : 저장된 인스턴스의 참조 빠름
  - LinkedList<E>의 장단점
    - 단점 : 저장된 인스턴스의 참조 과정이 복잡하여 느릴 수 있음
    - 장점 : 저장 공간을 늘리고 삭제하는 과정 단순하며 간단
  - 두 클래스의 특성을 고려하여 클래스 선택

List<E> 컬렉션 클래스들

- 저장된 인스턴스의 순차적 접근 방법 1 : enhanced for문(for–each 문)의 사용
  - 컬렉션 클래스를 활용하는데 있어 "저장된 인스턴스들에 순차적 접근"은 보편적이고 중요한 작업
  - 특정 인스턴스를 검색할 때, 저장된 인스턴스 전부를 대상으로 탐색을 진행해야 하므로 for으로 접근할 수 있음
  - for문으로 접근하는 것보다 나은 방법이 for-each 문 사용

```
public class EnhancedForCollection {
    public static void main(String[] args) {
        List<String> list = new LinkedList<>();
        list.add("Toy");
        list.add("Box");
        list.add("Robot");
        for (String s : list) {
            System.out.println(s);
        }
        System.out.println();
        list.remove(0);
        for (String s : list) {
            System.out.println(s);
        }
    }
}
```

  - for-each 문의 순차적 접근의 대상이 되려면 "public interface Iterable<T>" 인터페이스를 구현해야 하며, ArrayList<E>, LinkedList<E> 클래스는 위의 인터페이스를 구현하고 있음
  - public interface Collection<E> extends Iterable<E>

List<E> 컬렉션 클래스들

- 저장된 인스턴스의 순차적 접근 방법 2
  - Collection<E>는 Iterable<T>를 상속하며 Collection<E>를 구현하는 자바의 제네릭 클래스는 "Iterator<T> iterator" 추상 메소드를 모두 구현
  - 위 메소드는 반복자(Iterator)를 반환
  - 반복자 : 저장된 인스턴스들을 순차적으로 참조할 때 사용하는 인스턴스
  - "Iterator<String> itr = list.iterator();" 코드를 통해 반복자 획득
  - Iterator<E>의 메소드
    - E next() : 다음 인스턴스의 참조 값 반환
    - boolean hasNext() : next 메소드 호출 시 참조 값 반환 가능 여부 확인
    - void remove() : next 메소드 호출을 통해 반환했던 인스턴스 삭제

List<E> 컬렉션 클래스들

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class IteratorCollection {
    public static void main(String[] args) {
        List<String> list = new LinkedList<>();
        list.add("Toy");
        list.add("Box");
        list.add("Robot");
        list.add("Box");
        Iterator<String> itr = list.iterator();
        while(itr.hasNext()) {
            String str = itr.next();
            System.out.println(str);
        }
        System.out.println();
        itr = list.iterator();
        while(itr.hasNext()) {
            String str = itr.next();
            if (str.equals("Box")) {
                itr.remove();
            }
        }
        itr = list.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

- for-each문은 컴파일 과정에서 반복자를 이용하는 코드로 자동으로 수정
  - for(String s : list) { System.out.println(s) }
  - for(Iterator<String> itr = list.iterator( ); itr.hasNext( );) { System.out.printlm(itr.next( )) }

List<E> 컬렉션 클래스들

- 배열보다는 컬렉션 인스턴스가 좋다 : 컬렉션 변환
  - ArrayList<E>는 배열을 기반으로 인스턴스를 저장하므로 배열과 특성이 거의 유사
  - 대부분의 경우 인스턴스의 저장과 삭제가 편리하고, 반복자의 사용이 가능하므로 ArrayList<E> 사용이 훨씬 유리
  - 배열처럼 "선언과 동시에 초기화"를 할수 없어서 초기에 데이터 저장이 조금 번거롭지만 아래와 같은 컬렉션 인스턴스 생성 허용
    - List<String> list = Arrays.asList("Toy", "Robot", "Box") -> 인자로 전달된 인스턴스들을 저장한 컬렉션 인스턴스의 생성, 반환
  - 이렇게 생성된 컬렉션 인스턴스는 새로운 인스턴스의 추가나 삭제가 불가능하므로 새로운 인스턴스의 추가나 삭제가 필요하다면 아래 생성자를 기반으로 ArrayList<E> 인스턴스 생성
    - public ArrayList(Collection<? extends E> c) {...} : Collection<E>를 구현한 컬렉션 인스턴스를 인자로 전달 받음

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class AsListCollection {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("Toy", "Box", "Robot",
"Box");
        list = new ArrayList<>(list);
        for (String s : list) {
            System.out.println(s);
        }
        System.out.println();
        for (int i = 0; i < list.size(); i++) {
            if (list.get(i).equals("Box")) {
                list.remove(i);
            }
        }
        for (String s : list) {
            System.out.println(s);
        }
    }
}
```

List<E> 컬렉션 클래스들

- 기본 자료형 데이터의 저장과 참조
  - 컬렉션 인스턴스도 기본 자료형의 값은 저장 못함
  - 래퍼 클래스의 도움으로 이들 값의 저장 및 참조 가능
  - 이 과정에서 오토 박싱과 오토 언박싱으로 자연스러운 코드의 구성 가능

```
public class PrimitiveCollection {
    public static void main(String[] args) {
        LinkedList<Integer> list = new LinkedList<>();
        // 저장 과정에서 오토 박싱 진행
        list.add(10);
        list.add(20);
        list.add(30);
        for(int n : list) { // 오토 언박싱 진행
            System.out.println(n);
        }
    }
}
```

List<E> 컬렉션 클래스들

- 연결 리스트만 갖는 양방향 반복자
  - Collection<E>를 구현하는 클래스의 인스턴스는 iterator 메소드의 호출을 통해 반복자 획득
  - List<E>를 구현하는 클래스는 "public ListIterator<E> listIterator()" 메소드를 통해 양방향 반복자 획득
  - 양방향 반복자는 양쪽 방향으로 이동 가능
  - 양방향 반복자의 대표 메소드들
    - E next() : 다음 인스턴스의 참조 값 반환
    - boolean hasNext() : next 메소드 호출 시 참조 값 반환 가능 여부 확인
    - void remove : next 메소드 호출을 통해 반환했던 인스턴스 삭제
    - E previous() : next 메소드와 기능은 같고 방향만 반대
    - boolean hasPrevious() : hasNext 메소드와 기능은 같고 방향만 반대
    - void add(E e) : 인스턴스 추가
    - void set(E e) : 인스턴스 변경

List<E> 컬렉션 클래스들

```
public class ListIteratorCollection {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("Toy", "Box", "Robot",
"Box");
        list = new ArrayList<>(list);
        ListIterator<String> lItr = list.listIterator();    // 양방향
반복자 획득
        String str;
        while(lItr.hasNext()) { // 정방향으로
            str = lItr.next();
            System.out.println(str);
            if(str.equals("Toy")) {
                lItr.add("Toy2");
            }
        }
        System.out.println();
        while (lItr.hasPrevious()) {    // 반대 방향으로
            str = lItr.previous();
            System.out.println(str);
            if (str.equals("Robot")) {
                lItr.add("Robot2");
            }
        }
        System.out.println();
        for (String s : list) {
            System.out.println(s);
        }
    }
}
```

- 예제를 통해 다음과 같은 사실을 알 수 있음
  - next 메소드 호출 후에 add를 하면, 앞서 반환된 인스턴스 뒤에 새 인스턴스 삽입
  - previous 메소드 호출 후에 add를 하면, 앞서 반환된 인스턴스 앞에 새 인스턴스 삽입