5. 컬렉션 프레임워크 - List

#1.인강/0.자바/4.자바-중급2편

리스트 추상화1 - 인터페이스 도입

자바 기본편에서 학습한 다형성과 OCP 원칙을 가장 잘 활용할 수 있는 곳 중에 하나가 바로 자료 구조이다.

자료 구조에 다형성과 OCP 원칙이 어떻게 적용되는지 알아보자.

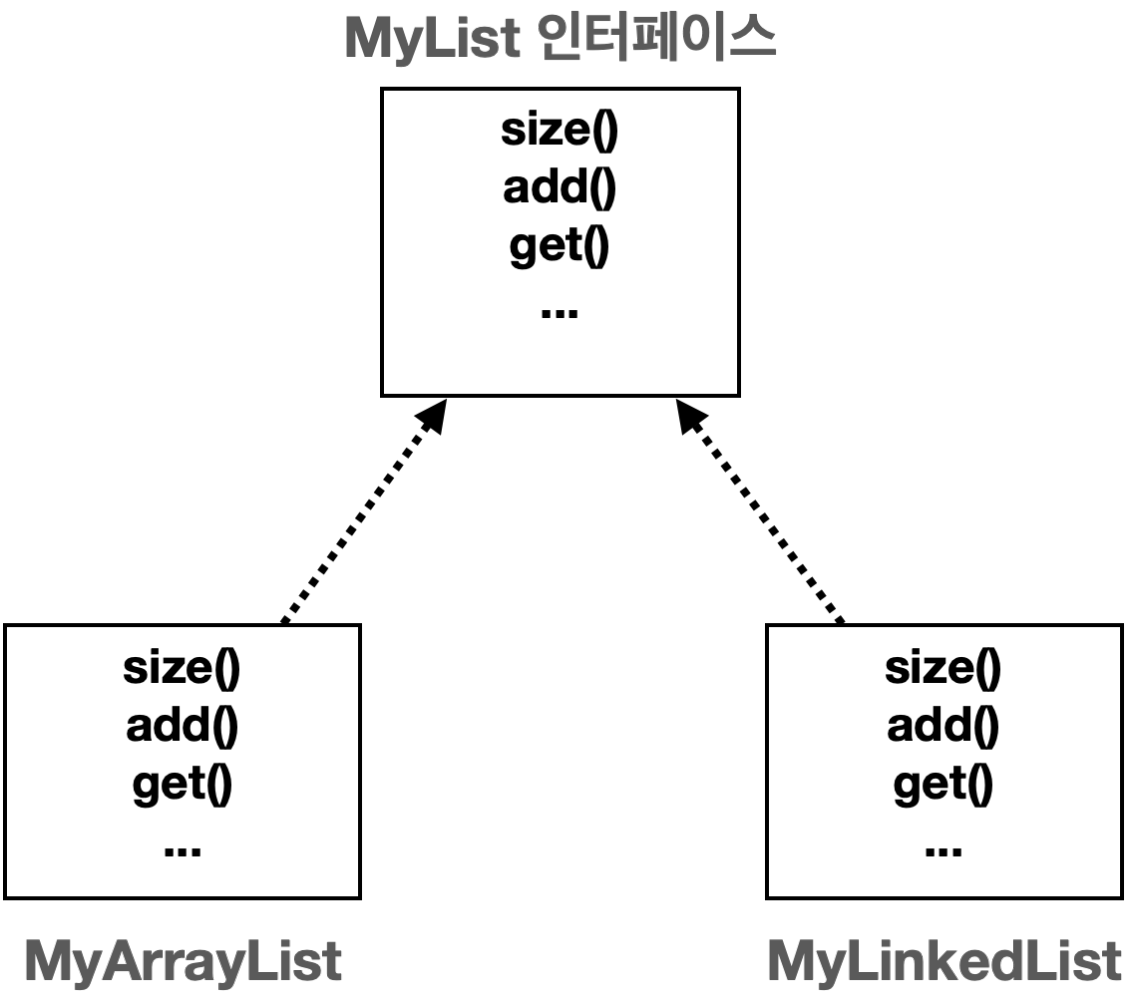
List 자료 구조

순서가 있고, 중복을 허용하는 자료 구조를 리스트(`List`)라 한다.

우리가 지금까지 만든 `MyArrayList`와 `MyLinkedList`는 내부 구현만 다를 뿐 같은 기능을 제공하는 리스트이다.

물론 내부 구현이 다르기 때문에 상황에 따라 성능은 달라질 수 있다. 핵심은 사용자 입장에서 보면 같은 기능을 제공한다는 것이다.

이 둘의 공통 기능을 인터페이스로 뽑아서 추상화하면 다형성을 활용한 다양한 이득을 얻을 수 있다.



같은 기능을 제공하는 메서드를 `MyList` 인터페이스로 뽑아보자.

지금부터 `collection.list` 패키지를 사용한다. 패키지 위치에 주의하자.

```
package collection.list;

public interface MyList<E> {

    int size();

    void add(E e);

    void add(int index, E e);

    E get(int index);

    E set(int index, E element);

    E remove(int index);
```

```
    int indexOf(E o);

}
```

collection.array.MyArrayListV4 코드를 복사해서 MyList 인터페이스를 구현하는 MyArrayList를 만들자.

```
package collection.list;

import java.util.Arrays;

public class MyArrayList<E> implements MyList<E> {
    //...
}
```

collection.link.MyLinkedListV3 코드를 복사해서 MyList 인터페이스를 구현하는 MyLinkedList를 만들자.

```
package collection.list;

public class MyLinkedList<E> implements MyList<E> {
    //...
}
```

- 메서드 이름이 같기 때문에 문제가 발생하지는 않을 것이다. 만약 메서드 정보가 다르다면 오류가 발생할 수 있다. 이때는 `MyList` 인터페이스에 맞추자.
- 추가로 재정의한 메서드에 `@Override` 애노테이션도 넣어주자.

전체 코드

```
package collection.list;

import java.util.Arrays;

public class MyArrayList<E> implements MyList<E> {

    private static final int DEFAULT_CAPACITY = 5;

    private Object[] elementData;
```

```
    private int size = 0;

    public MyArrayList() {
        elementData = new Object[DEFAULT_CAPACITY];
    }

    public MyArrayList(int initialCapacity) {
        elementData = new Object[initialCapacity];
    }

    @Override
    public int size() {
        return size;
    }

    @Override
    public void add(E e) {
        if (size == elementData.length) {
            grow();
        }
        elementData[size] = e;
        size++;
    }

    @Override
    public void add(int index, E e) {
        if (size == elementData.length) {
            grow();
        }
        shiftRightFrom(index);
        elementData[index] = e;
        size++;
    }

    //요소의 마지막부터 index까지 오른쪽으로 밀기
    private void shiftRightFrom(int index) {
        for (int i = size; i > index; i--) {
            elementData[i] = elementData[i - 1];
        }
    }

    @Override
    @SuppressWarnings("unchecked")
```

```
    public E get(int index) {
        return (E) elementData[index];
    }

    @Override
    public E set(int index, E element) {
        E oldValue = get(index);
        elementData[index] = element;
        return oldValue;
    }

    @Override
    public E remove(int index) {
        E oldValue = get(index);
        shiftLeftFrom(index);

        size--;
        elementData[size] = null;
        return oldValue;
    }

    //요소의 index부터 마지막까지 왼쪽으로 밀기
    private void shiftLeftFrom(int index) {
        for (int i = index; i < size - 1; i++) {
            elementData[i] = elementData[i + 1];
        }
    }

    @Override
    public int indexOf(E o) {
        for (int i = 0; i < size; i++) {
            if (o.equals(elementData[i])) {
                return i;
            }
        }
        return -1;
    }

    private void grow() {
        int oldCapacity = elementData.length;
        int newCapacity = oldCapacity * 2;
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
```

```
    @Override
    public String toString() {
        return Arrays.toString(Arrays.copyOf(elementData, size)) + " size=" +
size + ", capacity=" + elementData.length;
    }

}
```

```
package collection.list;

public class MyLinkedList<E> implements MyList<E> {
    private Node<E> first;
    private int size = 0;

    @Override
    public void add(E e) {
        Node<E> newNode = new Node<>(e);
        if (first == null) {
            first = newNode;
        } else {
            Node<E> lastNode = getLastNode();
            lastNode.next = newNode;
        }
        size++;
    }

    private Node<E> getLastNode() {
        Node<E> x = first;
        while (x.next != null) {
            x = x.next;
        }
        return x;
    }

    @Override
    public void add(int index, E e) {
        Node<E> newNode = new Node<>(e);
        if (index == 0) {
            newNode.next = first;
            first = newNode;
```

```
        } else {
            Node<E> prev = getNode(index - 1);
            newNode.next = prev.next;
            prev.next = newNode;
        }
        size++;
    }

    @Override
    public E set(int index, E element) {
        Node<E> x = getNode(index);
        E oldValue = x.item;
        x.item = element;
        return oldValue;
    }

    @Override
    public E remove(int index) {
        Node<E> removeNode = getNode(index);
        E removedItem = removeNode.item;
        if (index == 0) {
            first = removeNode.next;
        } else {
            Node<E> prev = getNode(index - 1);
            prev.next = removeNode.next;
        }
        removeNode.item = null;
        removeNode.next = null;
        size--;
        return removedItem;
    }

    @Override
    public E get(int index) {
        Node<E> node = getNode(index);
        return node.item;
    }

    private Node<E> getNode(int index) {
        Node<E> x = first;
        for (int i = 0; i < index; i++) {
            x = x.next;
        }
```

```
        return x;
    }

    @Override
    public int indexOf(E o) {
        int index = 0;
        for (Node<E> x = first; x != null; x = x.next) {
            if (o.equals(x.item))
                return index;
            index++;
        }
        return -1;
    }

    @Override
    public int size() {
        return size;
    }

    @Override
    public String toString() {
        return "MyLinkedListV1{" +
                "first=" + first +
                ", size=" + size +
                '}';
    }

    private static class Node<E> {
        E item;
        Node<E> next;

        public Node(E item) {
            this.item = item;
        }

        @Override
        public String toString() {
            StringBuilder sb = new StringBuilder();
            Node<E> temp = this;
            sb.append("[");
            while (temp != null) {
                sb.append(temp.item);
                if (temp.next != null) {
```

```
                sb.append("->");
            }
            temp = temp.next;
        }
        sb.append("]");
        return sb.toString();
    }
}
}
```

리스트 추상화2 - 의존관계 주입

`MyArrayList`를 활용해서 많은 데이터를 처리하는 `BatchProcessor` 클래스를 개발하고 있다고 가정하자. 그런데 막상 프로그램을 개발하고 보니 데이터를 앞에서 추가하는 일이 많은 상황이라고 가정해보자. 데이터를 앞에서 추가하거나 삭제하는 일이 많다면 `MyArrayList` 보다는 `MyLinkedList`를 사용하는 것이 훨씬 효율적이다.

데이터를 앞에서 추가하거나 삭제할 때 빅오 비교

- `MyArrayList` : O(n)
- `MyLinkedList` : O(1)

다음 예시 코드를 보자.

구체적인 MyArrayList에 의존하는 BatchProcessor 예시

```
public class BatchProcessor {

    private final MyArrayList<Integer> list = new MyArrayList<>(); //코드 변경

    public void logic(int size) {
        for (int i = 0; i < size; i++) {
            list.add(0, i); //앞에 추가
        }
    }

}
```

`MyArrayList`를 사용해보니 성능이 너무 느려서 `MyLinkedList`를 사용하도록 코드를 변경해야 한다면 `BatchProcessor`의 내부 코드도 `MyArrayList`에서 `MyLinkedList`를 사용하도록 함께 변경해야 한다.

구체적인 MyLinkedList에 의존하는 BatchProcessor 예시

```
public class BatchProcessor {

    private final MyLinkedList<Integer> list = new MyLinkedList<>(); //코드 변경

    public void logic(int size) {
        for (int i = 0; i < size; i++) {
            list.add(0, i); //앞에 추가
        }
    }

}
```

`BatchProcessor`는 구체적인 `MyArrayList` 또는 `MyLinkedList`를 사용하고 있다. 이것을 `BatchProcessor`가 구체적인 클래스에 의존한다고 표현한다. 이렇게 구체적인 클래스에 직접 의존하면 `MyArrayList`를 `MyLinkedList`로 변경할 때 마다 여기에 의존하는 `BatchProcessor`의 코드도 함께 수정해야 한다.

`BatchProcessor`가 구체적인 클래스에 의존하는 대신 추상적인 `MyList` 인터페이스에 의존하는 방법도 있다.

추상적인 MyList에 의존하는 BatchProcessor 예시

```
public class BatchProcessor {

    private final MyList<Integer> list;

    public BatchProcessor(MyList<Integer> list) {
        this.list = list;
    }

    public void logic(int size) {
        for (int i = 0; i < size; i++) {
            list.add(0, i); //앞에 추가
        }
    }
```

```
}
```

```
main() {
    new BatchProcessor(new MyArrayList());  //MyArrayList를 사용하고 싶을 때
    new BatchProcessor(new MyLinkedList()); //MyLinkedList를 사용하고 싶을 때
}
```

그리고 `BatchProcessor`를 생성하는 시점에 생성자를 통해 원하는 리스트 전략(알고리즘)을 선택해서 전달하면 된다.
이렇게 하면 `MyList`를 사용하는 클라이언트 코드인 `BatchProcessor`를 전혀 변경하지 않고, 원하는 리스트 전략을 런타임에 지정할 수 있다.

정리하면 다형성과 추상화를 활용하면 `BatchProcessor` 코드를 전혀 변경하지 않고 리스트 전략(알고리즘)을 `MyArrayList`에서 `MyLinkedList`로 변경할 수 있다.

실제 코드를 작성해보자.

```
package collection.list;

public class BatchProcessor {

    private final MyList<Integer> list;

    public BatchProcessor(MyList<Integer> list) {
        this.list = list;
    }

    public void logic(int size) {
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            list.add(0, i); //앞에 추가
        }
        long endTime = System.currentTimeMillis();
        System.out.println("크기: " + size + ", 계산 시간: " + (endTime -
startTime) + "ms");
    }

}
```

- `logic(int size)` 메서드는 매우 복잡한 비즈니스 로직을 다룬다고 가정하자. 이 메서드는 `list`의 앞 부분에 데이터를 추가한다.
- 어떤 `MyList list`의 구현체를 선택할 지는 실행 시점에 생성자를 통해서 결정한다.
- 생성자를 통해서 `MyList` 구현체가 전달된다.
  - `MyArrayList`의 인스턴스가 들어올 수도 있고, `MyLinkedList`의 인스턴스가 들어올 수도 있다.
- 이것은 `BatchProcessor`의 외부에서 의존관계가 결정되어서 `BatchProcessor` 인스턴스에 들어오는 것 같다. 마치 의존관계가 외부에서 주입되는 것 같다고 해서 이것을 **의존관계 주입**이라 한다.
- 참고로 생성자를 통해서 의존관계를 주입했기 때문에 생성자 의존관계 주입이라 한다.

의존관계 주입

- Dependency Injection, 줄여서 DI라고 부른다. 의존성 주입이라고도 부른다.

```
package collection.list;

public class BatchProcessorMain {

    public static void main(String[] args) {
        MyArrayList<Integer> list = new MyArrayList<>();
        //MyLinkedList<Integer> list = new MyLinkedList<>();

        BatchProcessor processor = new BatchProcessor(list);
        processor.logic(50_000);
    }
}
```

- `BatchProcessor`의 생성자에 `MyArrayList`를 사용할지, `MyLinkedList`를 사용할지 결정해서 넘겨야 한다.
- 이후에 `processor.logic()`을 호출하면 생성자로 넘긴 자료 구조를 사용해서 실행한다.

MyArrayList - 실행 결과

```
크기: 50000, 계산 시간: 1361ms
```

다음과 같이 주석을 변경해서 `MyLinkedList`로 바꾸고 코드를 실행해보자.

```
//MyArrayList<Integer> list = new MyArrayList<>();
MyLinkedList<Integer> list = new MyLinkedList<>();
```

MyLinkedList - 실행 결과

```
크기: 50000, 계산 시간: 2ms
```

`MyLinkedList`를 사용한 덕분에 O(n) ➜ O(1)로 훨씬 더 성능이 개선 된 것을 확인할 수 있다. 데이터가 증가할수록 성능의 차이는 더 벌어진다.

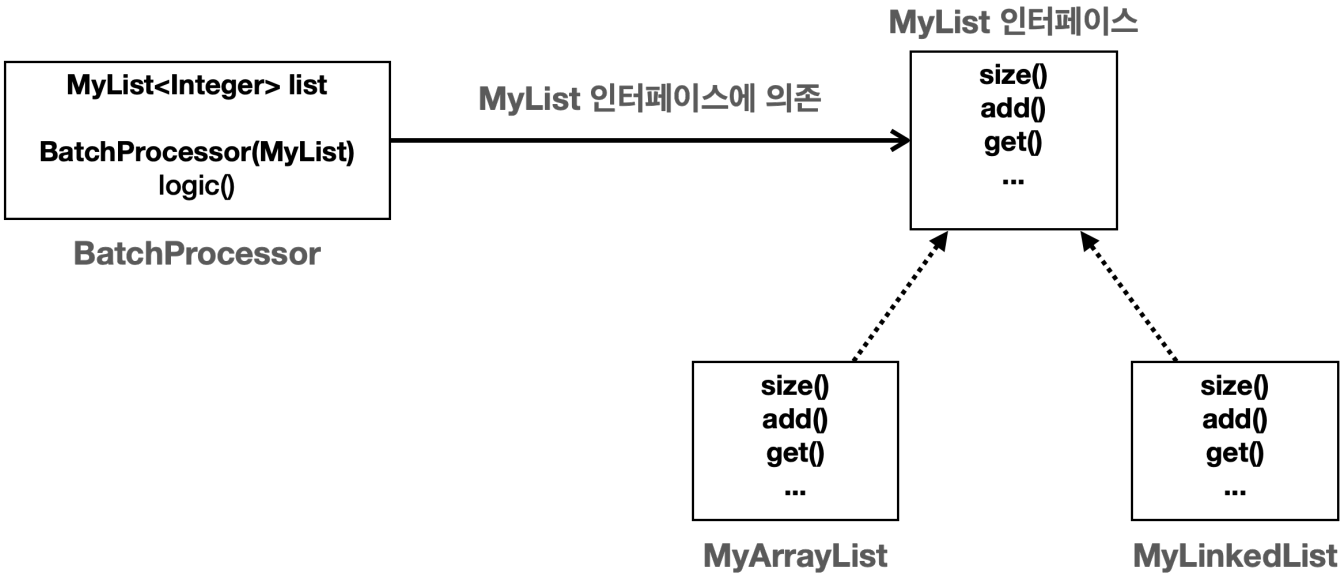
리스트 추상화3 - 컴파일 타임, 런타임 의존관계

의존관계는 크게 컴파일 타임 의존관계와 런타임 의존관계로 나눌 수 있다.

- **컴파일 타임(compile time)**: 코드 컴파일 시점을 뜻한다.
- **런타임(runtime)**: 프로그램 실행 시점을 뜻한다.

컴파일 타임 의존관계 vs 런타임 의존관계

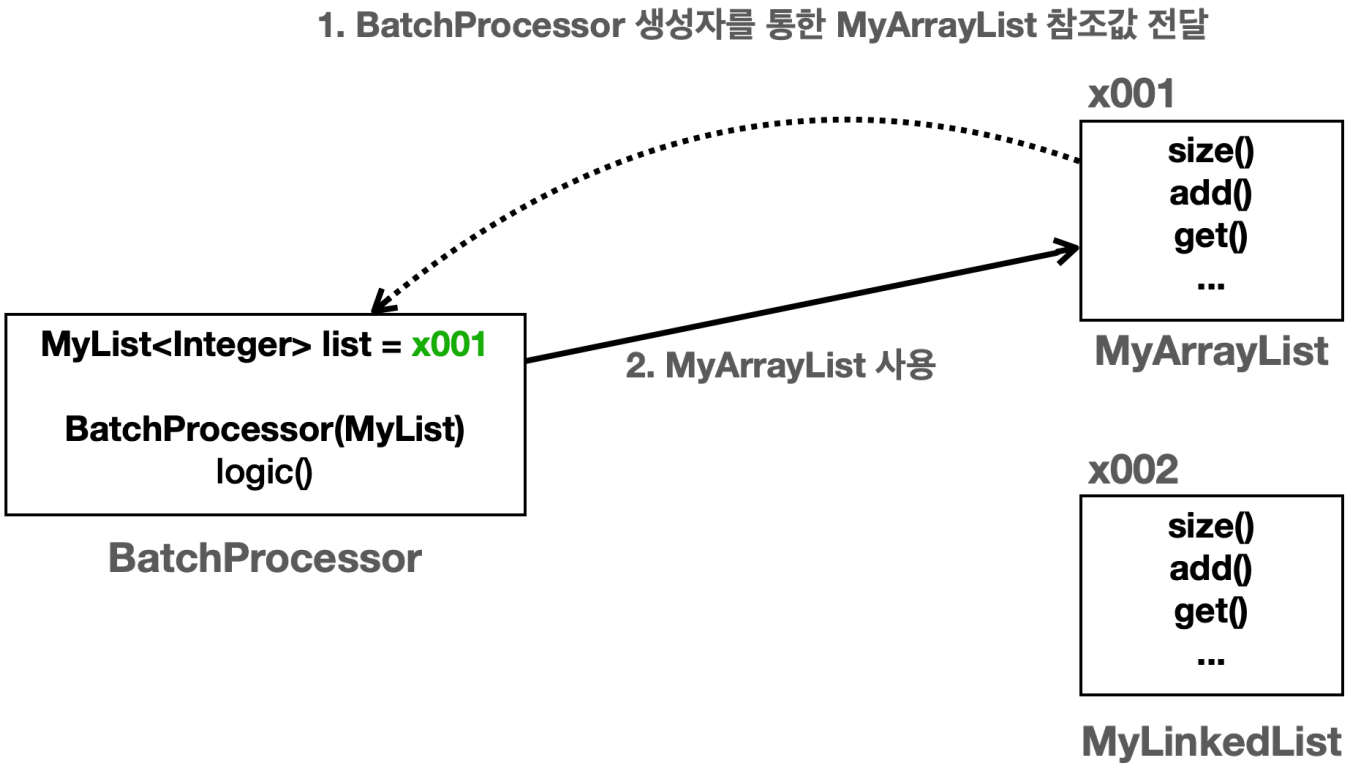
컴파일 타임 의존관계



- 컴파일 타임 의존관계는 자바 컴파일러가 보는 의존관계이다. 클래스에 모든 의존관계가 다 나타난다.
- 쉽게 이야기해서 클래스에 바로 보이는 의존관계이다. 그리고 실행하지 않은 소스 코드에 정적으로 나타나는 의존관계이다.
- `BatchProcessor` 클래스를 보면 `MyList` 인터페이스만 사용한다. 코드 어디에도 `MyArrayList`나

`MyLinkedList` 같은 정보는 보이지 않는다. 따라서 `BatchProcessor`는 `MyList` 인터페이스에만 의존한다.

런타임 의존관계



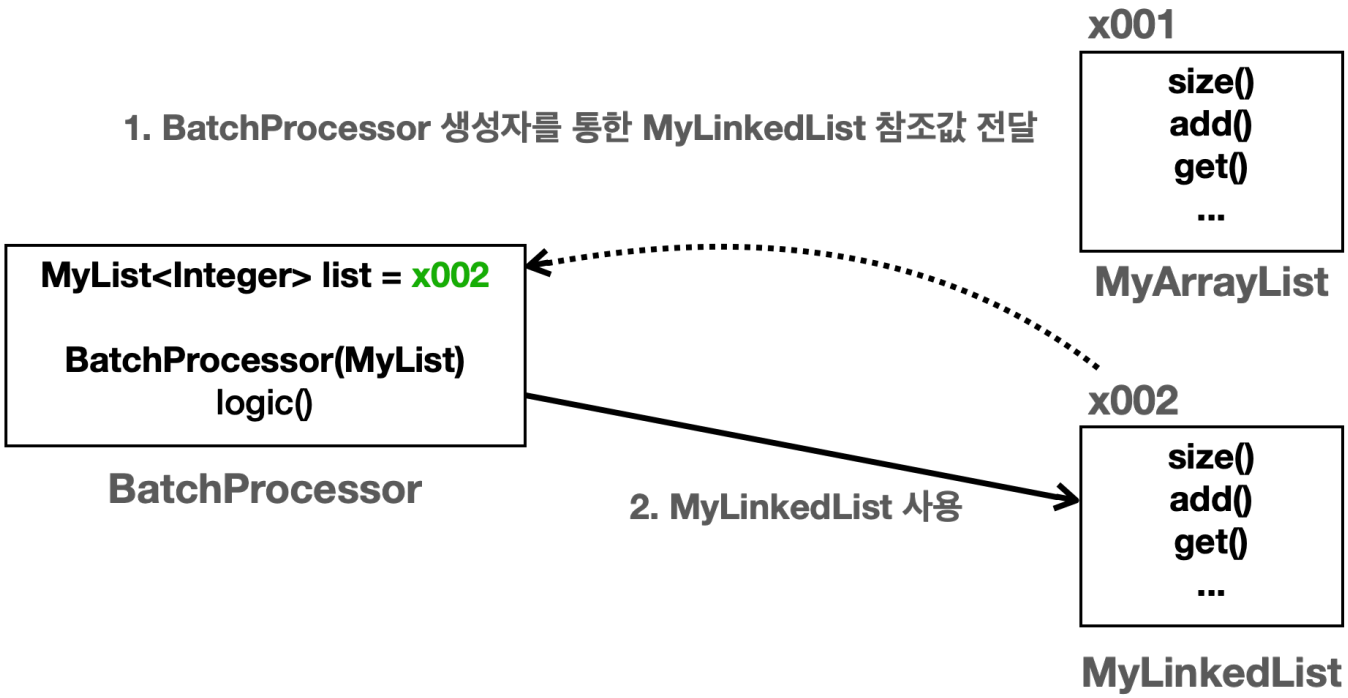
- 런타임 의존관계는 실제 프로그램이 작동할 때 보이는 의존관계다. 주로 생성된 인스턴스와 그것을 참조하는 의존관계이다.
- 쉽게 이야기해서 프로그램이 실행될 때 인스턴스 간에 의존관계로 보면 된다.
- 런타임 의존관계는 프로그램 실행 중에 계속 변할 수 있다.

다음과 같이 코드를 작성하고 실행한다고 가정하자.

```
MyArrayList<Integer> list = new MyArrayList<>();
BatchProcessor processor = new BatchProcessor(list);
processor.logic(50_000);
```

- `BatchProcessor` 인스턴스의 `MyList list`는 생성자를 통해 `MyArrayList(x001)` 인스턴스를 참조한다.
  - `BatchProcessor` 인스턴스에 `MyArrayList(x001)` 의존관계를 주입한다.
- 따라서 이후 `logic()`을 호출하면 `MyArrayList` 인스턴스를 사용하게 된다.

런타임 의존관계



다음과 같이 코드를 작성하고 실행한다고 가정하자.

```
MyLinkedList<Integer> list = new MyLinkedList<>();
BatchProcessor processor = new BatchProcessor(list);
processor.logic(50_000);
```

- `BatchProcessor` 인스턴스의 `MyList list`는 생성자를 통해 `MyLinkedList(x002)` 인스턴스를 참조한다.
  - `BatchProcessor` 인스턴스에 `MyLinkedList(x002)` 의존관계를 주입한다.
- 따라서 이후 `logic()`을 호출하면 `MyLinkedList` 인스턴스를 사용하게 된다.

정리

- `MyList` 인터페이스의 도입으로 같은 리스트 자료구조를 그대로 사용하면서 원하는 구현을 변경할 수 있게 되었다.
- `BatchProcessor`에서 다음과 같이 처음부터 `MyArrayList`를 사용하도록 컴파일 타임 의존관계를 지정했다면 이후에 `MyLinkedList`로 수정하기 위해서는 `BatchProcessor`의 코드를 변경해야 한다.

```
public class BatchProcessor {
    private final MyArrayList<Integer> list = new MyArrayList(); //코드 변경 필요
```

```
}
```

- `BatchProcessor` 클래스는 구체적인 `MyArrayList`나 `MyLinkedList`에 의존하는 것이 아니라 추상적인 `MyList`에 의존한다. 따라서 런타임에 `MyList`의 구현체를 얼마든지 선택할 수 있다.
- `BatchProcessor`에서 사용하는 리스트의 의존관계를 클래스에서 미리 결정하는 것이 아니라, 런타임에 객체를 생성하는 시점으로 미룬다. 따라서 런타임에 `MyList`의 구현체를 변경해도 `BatchProcessor`의 코드는 전혀 변경하지 않아도 된다.
- 이렇게 생성자를 통해 런타임 의존관계를 주입하는 것을 **생성자 의존관계 주입** 또는 줄여서 **생성자 주입**이라 한다.
- 자바 기본편에서 학습한 OCP 원칙을 지켰다. 클라이언트 코드의 변경 없이, 구현 알고리즘인 `MyList` 인터페이스의 구현을 자유롭게 확장할 수 있다.
- **클라이언트 클래스는 컴파일 타임에 추상적인 것에 의존하고, 런타임에 의존 관계 주입을 통해 구현체를 주입받아 사용함으로써, 이런 이점을 얻을 수 있다.**

> **전략 패턴(Strategy Pattern)**
> 디자인 패턴 중에 가장 중요한 패턴을 하나 뽑으라고 하면 전략 패턴을 뽑을 수 있다. 전략 패턴은 알고리즘을 클라이언트 코드의 변경 없이 쉽게 교체할 수 있다. 방금 설명한 코드가 바로 전략 패턴을 사용한 코드이다. `MyList` 인터페이스가 바로 전략을 정의하는 인터페이스가 되고, 각각의 구현체인 `MyArrayList`, `MyLinkedList`가 전략의 구체적인 구현이 된다. 그리고 전략을 클라이언트 코드(`BatchProcessor`)의 변경 없이 손쉽게 교체할 수 있다.

직접 구현한 리스트의 성능 비교

직접 구현한 리스트들의 성능을 비교해보자.

```
package collection.list;

public class MyListPerformanceTest {

    public static void main(String[] args) {
        int size = 50_000;
        System.out.println("==MyArrayList 추가==");
        addFirst(new MyArrayList<>(), size);
        addMid(new MyArrayList<>(), size);
        MyArrayList<Integer> arrayList = new MyArrayList<>(); //조회용 데이터로 사
```

```
용
        addLast(arrayList, size);

        System.out.println("==MyLinkedList 추가==");
        addFirst(new MyLinkedList<>(), size);
        addMid(new MyLinkedList<>(), size);
        MyLinkedList<Integer> linkedList = new MyLinkedList<>(); //조회용 데이터로
사용
        addLast(linkedList, size);

        int loop = 10000;
        System.out.println("==MyArrayList 조회==");
        getIndex(arrayList, loop, 0);
        getIndex(arrayList, loop, size / 2);
        getIndex(arrayList, loop, size - 1);

        System.out.println("==MyLinkedList 조회==");
        getIndex(linkedList, loop, 0);
        getIndex(linkedList, loop, size / 2);
        getIndex(linkedList, loop, size - 1);

        System.out.println("==MyArrayList 검색==");
        search(arrayList, loop, 0);
        search(arrayList, loop, size / 2);
        search(arrayList, loop, size - 1);

        System.out.println("==MyLinkedList 검색==");
        search(linkedList, loop, 0);
        search(linkedList, loop, size / 2);
        search(linkedList, loop, size - 1);
    }

    private static void addFirst(MyList<Integer> list, int size) {
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            list.add(0, i);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("앞에 추가 - 크기: " + size + ", 계산 시간: " + (endTime
- startTime) + "ms");
    }

    private static void addMid(MyList<Integer> list, int size) {
```

```
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            list.add(i / 2, i);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("평균 추가 - 크기: " + size + ", 계산 시간: " + (endTime
- startTime) + "ms");
    }

    private static void addLast(MyList<Integer> list, int size) {
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            list.add(i);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("뒤에 추가 - 크기: " + size + ", 계산 시간: " + (endTime
- startTime) + "ms");
    }

    private static void getIndex(MyList<Integer> list, int loop, int index) {
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < loop; i++) {
            list.get(index);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("index: " + index + ", 반복: " + loop + ", 계산 시간: "
+ (endTime - startTime) + "ms");
    }

    private static void search(MyList<Integer> list, int loop, int findValue)
{
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < loop; i++) {
            list.indexOf(findValue);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("findValue: " + findValue + ", 반복: " + loop + ", 계
산 시간: " + (endTime - startTime) + "ms");
    }
}
```

실행 결과

```
==MyArrayList 추가==
앞에 추가 - 크기: 50000, 계산 시간: 1349ms
평균 추가 - 크기: 50000, 계산 시간: 638ms
뒤에 추가 - 크기: 50000, 계산 시간: 2ms

==MyLinkedList 추가==
앞에 추가 - 크기: 50000, 계산 시간: 2ms
평균 추가 - 크기: 50000, 계산 시간: 1066ms
뒤에 추가 - 크기: 50000, 계산 시간: 2169ms

==MyArrayList 조회==
index: 0, 반복: 10000, 계산 시간: 0ms
index: 25000, 반복: 10000, 계산 시간: 0ms
index: 49999, 반복: 10000, 계산 시간: 0ms

==MyLinkedList 조회==
index: 0, 반복: 10000, 계산 시간: 1ms
index: 25000, 반복: 10000, 계산 시간: 438ms
index: 49999, 반복: 10000, 계산 시간: 873ms

==MyArrayList 검색==
findValue: 0, 반복: 10000, 계산 시간: 0ms
findValue: 25000, 반복: 10000, 계산 시간: 115ms
findValue: 49999, 반복: 10000, 계산 시간: 222ms

==MyLinkedList 검색==
findValue: 0, 반복: 10000, 계산 시간: 0ms
findValue: 25000, 반복: 10000, 계산 시간: 492ms
findValue: 49999, 반복: 10000, 계산 시간: 983ms
```

실행 결과는 시스템 환경 마다 다를 수 있다. 참고로 맥북 Pro M2 MAX로 실행한 결과이다.

직접 만든 배열 리스트와 연결 리스트 - 성능 비교 표

기능	배열 리스트	연결 리스트
앞에 추가(삭제)	O(n) - 1369ms	O(1) - 2ms
평균 추가(삭제)	O(n) - 651ms	O(n) - 1112ms
뒤에 추가(삭제)	O(1) - 2ms	O(n) - 2195ms

인덱스 조회	O(1) - 1ms	O(n) - 평균 438ms
검색	O(n) - 평균 115ms	O(n) - 평균 492ms

추가, 삭제

- 배열 리스트는 인덱스를 통해 추가나 삭제할 위치를 O(1)로 빠르게 찾지만, 추가나 삭제 이후에 데이터를 한칸씩 밀어야 한다. 이 부분이 O(n)으로 오래 걸린다.
- 연결 리스트는 인덱스를 통해 추가나 삭제할 위치를 O(n)으로 느리게 찾지만, 실제 데이터의 추가는 간단한 참조 변경으로 빠르게 O(1)로 수행된다.

앞에 추가(삭제)

- 배열 리스트: 추가나 삭제할 위치는 찾는데 O(1), 데이터를 한칸씩 이동 O(n) ➜ O(n)
- 연결 리스트: 추가나 삭제할 위치는 찾는데 O(1), 노드를 변경하는데 O(1) ➜ O(1)

평균 추가(삭제)

- 배열 리스트: 추가나 삭제할 위치는 찾는데 O(1), 인덱스 이후의 데이터를 한칸씩 이동 O(n/2) ➜ O(n)
- 연결 리스트: 추가나 삭제할 위치는 찾는데 O(n/2), 노드를 변경하는데 O(1) ➜ O(n)

뒤에 추가(삭제)

- 배열 리스트: 추가나 삭제할 위치는 찾는데 O(1), 이동할 데이터 없음 ➜ O(1)
- 연결 리스트: 추가나 삭제할 위치는 찾는데 O(n), 노드를 변경하는데 O(1) ➜ O(n)

인덱스 조회

- 배열 리스트: 배열에 인덱스를 사용해서 값을 O(1)로 찾을 수 있음
- 연결 리스트: 노드를 인덱스 수 만큼 이동해야함 O(n)

검색

- 배열 리스트: 데이터를 찾을 때 까지 배열을 순회 O(n)
- 연결 리스트: 데이터를 찾을 때 까지 노드를 순회 O(n)

시간 복잡도와 실제 성능

- 이론적으로 `MyLinkedList`의 평균 추가(중간 삽입) 연산은 `MyArrayList`보다 빠를 수 있다. 그러나 실제 성능은 요소의 순차적 접근 속도, 메모리 할당 및 해제 비용, CPU 캐시 활용도 등 다양한 요소에 의해 영향을 받는다.
- `MyArrayList`는 요소들이 메모리 상에서 연속적으로 위치하여 CPU 캐시 효율이 좋고, 메모리 접근 속도가 빠르다.
- 반면, `MyLinkedList`는 각 요소가 별도의 객체로 존재하고 다음 요소의 참조를 저장하기 때문에 CPU 캐시 효

율이 떨어지고, 메모리 접근 속도가 상대적으로 느릴 수 있다
- `MyArrayList`의 경우 `CAPACITY`를 넘어서면 배열을 다시 만들고 복사하는 과정이 추가된다. 하지만 한번에 2배씩 늘어나기 때문에 이 과정은 가끔 발생하므로, 전체 성능에 큰 영향을 주지는 않는다.

정리하면 이론적으로 `MyLinkedList`가 평균 추가(중간 삽입)에 있어 더 효율적일 수 있지만, 현대 컴퓨터 시스템의 메모리 접근 패턴, CPU 캐시 최적화 등을 고려할 때 `MyArrayList`가 실제 사용 환경에서 더 나은 성능을 보여주는 경우가 많다.

배열 리스트 vs 연결 리스트

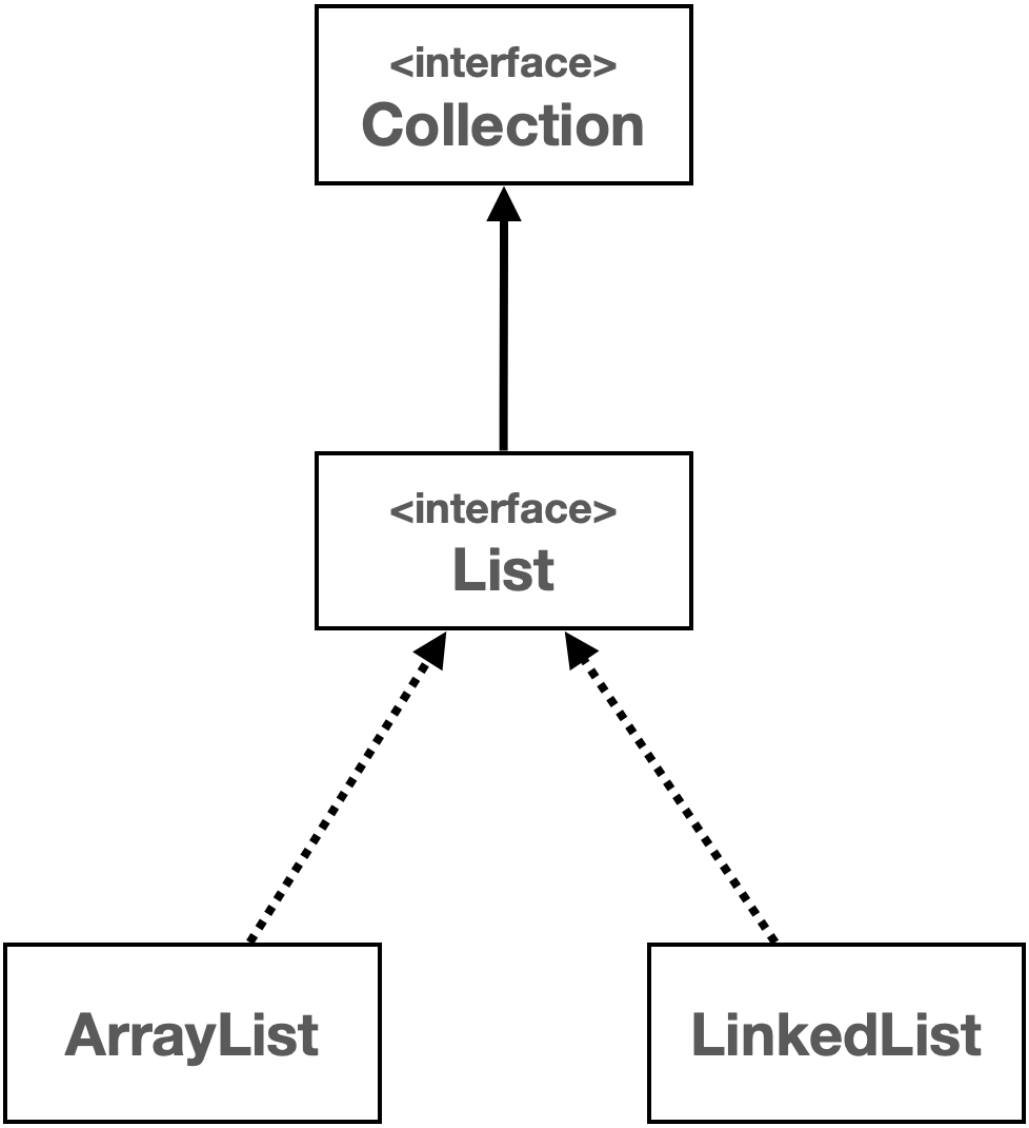
대부분의 경우 배열 리스트가 성능상 유리하다. 이런 이유로 실무에서는 주로 배열 리스트를 기본으로 사용한다.
만약 데이터를 앞쪽에서 자주 추가하거나 삭제할 일이 있다면 연결 리스트를 고려하자.

자바 리스트

List 자료 구조

순서가 있고, 중복을 허용하는 자료 구조를 리스트라 한다. 자바의 컬렉션 프레임워크가 제공하는 가장 대표적인 자료 구조가 바로 리스트이다. 리스트와 관련된 컬렉션 프레임워크는 다음 구조를 가진다.

컬렉션 프레임워크 - 리스트



Collection 인터페이스

`Collection` 인터페이스는 `java.util` 패키지의 컬렉션 프레임워크의 핵심 인터페이스 중 하나이다. 이 인터페이스는 자바에서 다양한 컬렉션, 즉 데이터 그룹을 다루기 위한 메서드를 정의한다. `Collection` 인터페이스는 `List`, `Set`, `Queue`와 같은 다양한 하위 인터페이스와 함께 사용되며, 이를 통해 데이터를 리스트, 세트, 큐 등의 형태로 관리할 수 있다. 자세한 내용은 뒤에서 다룬다.

List 인터페이스

`List` 인터페이스는 `java.util` 패키지에 있는 컬렉션 프레임워크의 일부다. `List`는 객체들의 순서가 있는 컬렉션을 나타내며, 같은 객체의 중복 저장을 허용한다. 이 리스트는 배열과 비슷하지만, 크기가 동적으로 변화하는 컬렉션을 다룰 때 유연하게 사용할 수 있다.

`List` 인터페이스는 `ArrayList`, `LinkedList`와 같은 여러 구현 클래스를 가지고 있으며, 각 클래스는 `List` 인터페이스의 메서드를 구현한다.

List 인터페이스의 주요 메서드

메서드	설명
`add(E e)`	리스트의 끝에 지정된 요소를 추가한다.
`add(int index, E element)`	리스트의 지정된 위치에 요소를 삽입한다.
`addAll(Collection<? extends E> c)`	지정된 컬렉션의 모든 요소를 리스트의 끝에 추가한다.
`addAll(int index, Collection<? extends E> c)`	지정된 컬렉션의 모든 요소를 리스트의 지정된 위치에 추가한다.
`get(int index)`	리스트에서 지정된 위치의 요소를 반환한다.
set(int index, E element)	지정한 위치의 요소를 변경하고, 이전 요소를 반환한다.
`remove(int index)`	리스트에서 지정된 위치의 요소를 제거하고 그 요소를 반환한다.
`remove(Object o)`	리스트에서 지정된 첫 번째 요소를 제거한다.
`clear()`	리스트에서 모든 요소를 제거한다.
`indexOf(Object o)`	리스트에서 지정된 요소의 첫 번째 인덱스를 반환한다.
`lastIndexOf(Object o)`	리스트에서 지정된 요소의 마지막 인덱스를 반환한다.
`contains(Object o)`	리스트가 지정된 요소를 포함하고 있는지 여부를 반환한다.
`sort(Comparator<? super E> c)`	리스트의 요소를 지정된 비교자에 따라 정렬한다.
`subList(int fromIndex, int toIndex)`	리스트의 일부분의 뷰를 반환한다.
`size()`	리스트의 요소 수를 반환한다.
`isEmpty()`	리스트가 비어있는지 여부를 반환한다.
`iterator()`	리스트의 요소에 대한 반복자를 반환한다.
`toArray()`	리스트의 모든 요소를 배열로 반환한다.
`toArray(T[] a)`	리스트의 모든 요소를 지정된 배열로 반환한다.

자바 ArrayList

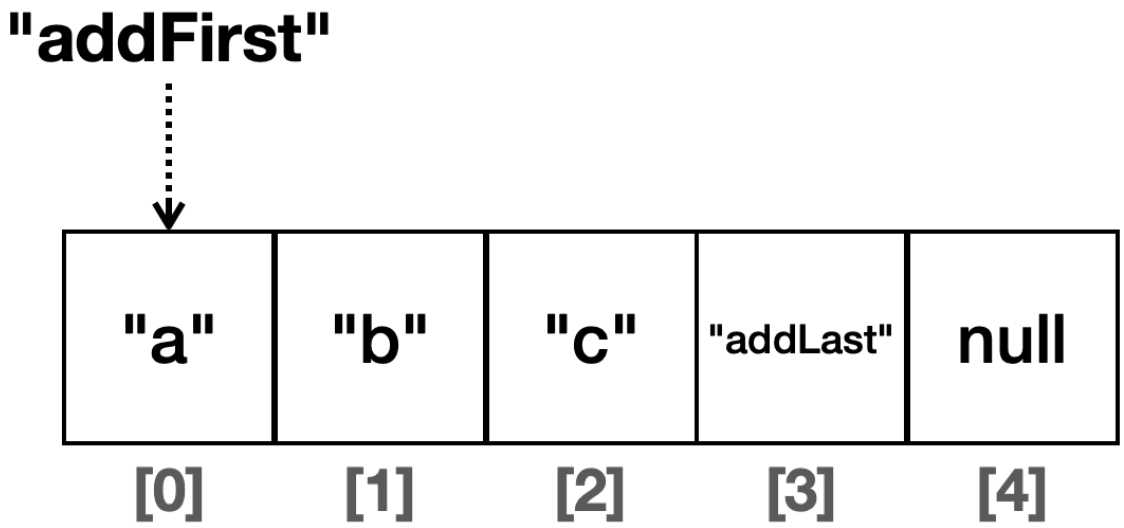
`java.util.ArrayList`

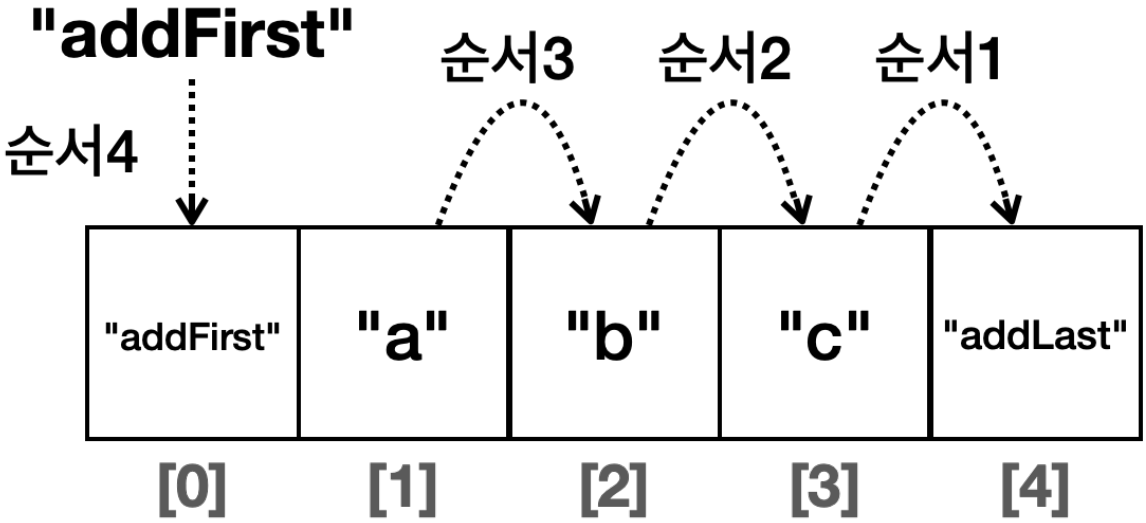
자바가 제공하는 `ArrayList`는 우리가 직접 만든 `MyArrayList`와 거의 비슷하다. 특징은 다음과 같다.

자바 ArrayList의 특징

- 배열을 사용해서 데이터를 관리한다.
- 기본 `CAPACITY`는 10이다.(`DEFAULT_CAPACITY = 10`)
  - `CAPACITY`를 넘어가면 배열을 50% 증가한다.
  - 10 ➜ 15 ➜ 22 ➜ 33 ➜ 49로 증가한다. (최적화는 자바 버전에 따라 달라질 수 있다.)
- 메모리 고속 복사 연산을 사용한다.
  - `ArrayList`의 중간 위치에 데이터를 추가하면, 추가할 위치 이후의 모든 요소를 한 칸씩 뒤로 이동시켜야 한다.
  - 자바가 제공하는 `ArrayList`는 이 부분을 최적화 하는데, 배열의 요소 이동은 시스템 레벨에서 최적화된 메모리 고속 복사 연산을 사용해서 비교적 빠르게 수행된다. 참고로 `System.arraycopy()`를 사용한다.

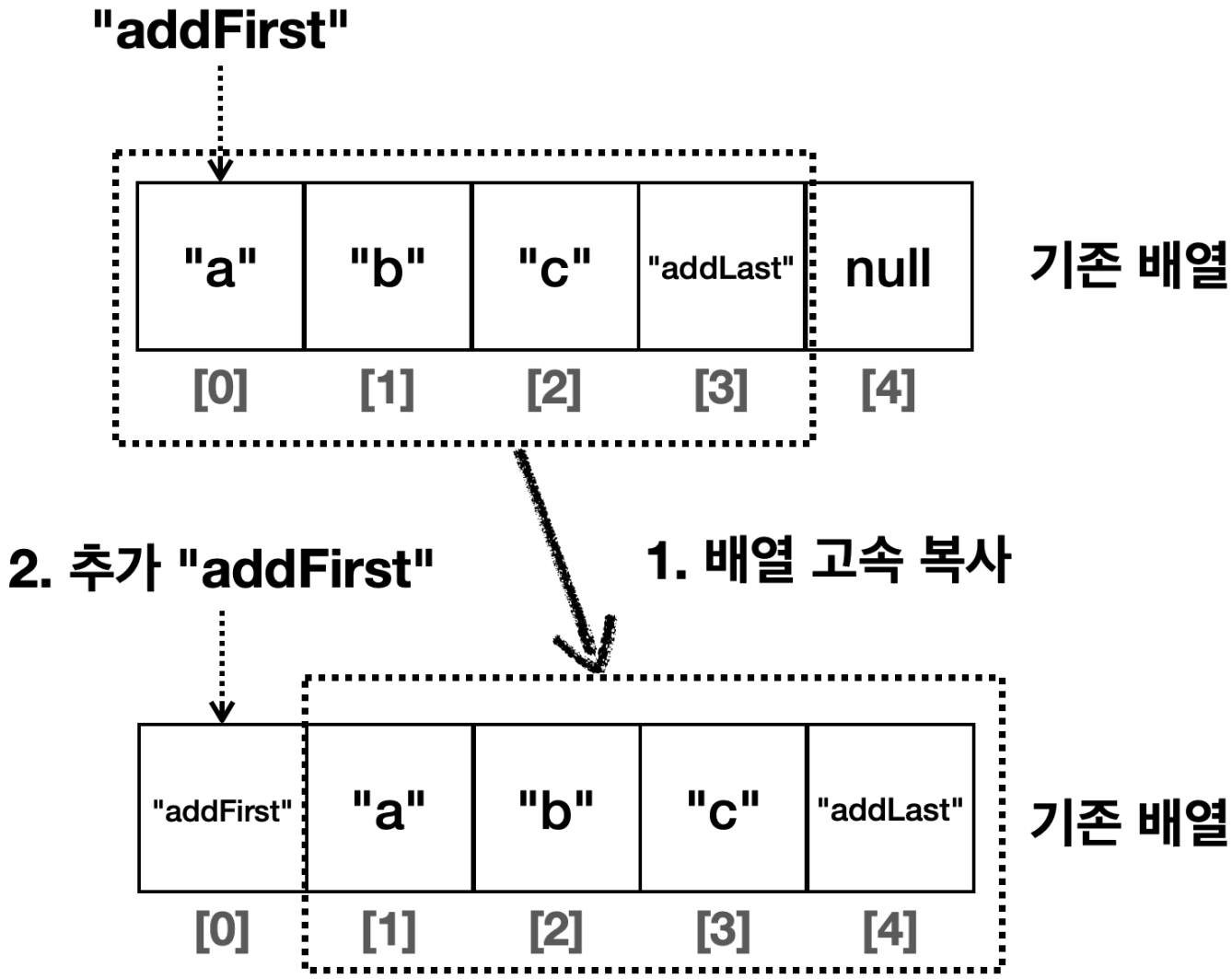
데이터 추가 - 한 칸씩 이동하는 방식





- 데이터를 루프를 돌면서 하나씩 이동해야 하기 때문에 매우 느리다.
- `MyArrayList`에서 사용한 방식이다.

데이터 추가 - 메모리 고속 복사 연산 사용



- 시스템 레벨에서 배열을 한 번에 아주 빠르게 복사한다. 이 부분은 OS, 하드웨어에 따라 성능이 다르기 때문에 정확한 측정이 어렵지만, 한 칸씩 이동하는 방식과 비교하면 보통 수 배 이상의 빠른 성능을 제공한다.

자바 LinkedList

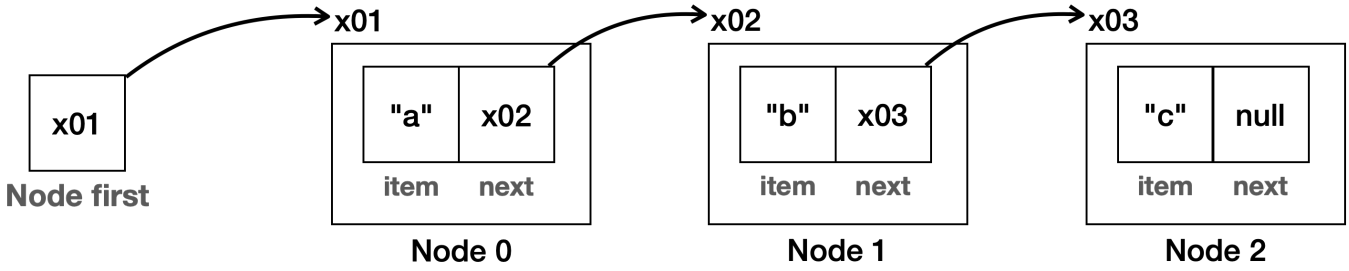
`java.util.LinkedList`

자바가 제공하는 `LinkedList`는 우리가 직접 만든 `MyLinkedList`와 거의 비슷하다. 특징은 다음과 같다.

자바의 LinkedList 특징

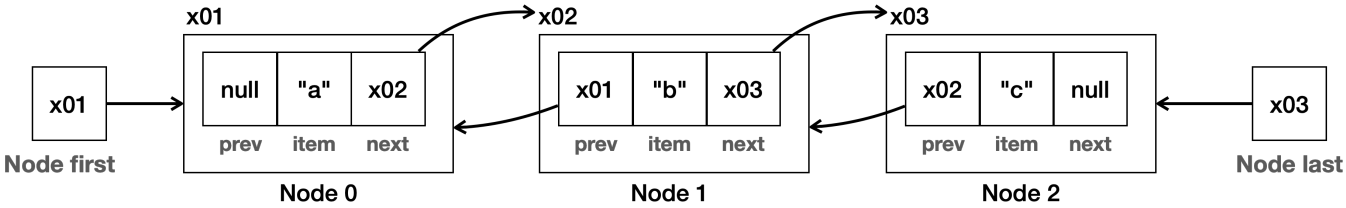
- 이중 연결 리스트 구조
- 첫 번째 노드와 마지막 노드 둘다 참조

단일 연결 리스트



- 우리가 직접 만든 `MyLinkedList`의 노드는 다음 노드로만 이동할 수 있는 단일 연결 구조다. 따라서 이전 노드로 이동할 수 없다는 단점이 있다.

이중 연결 리스트



- 자바가 제공하는 `LinkedList`는 이중 연결 구조를 사용한다. 이 구조는 다음 노드 뿐만 아니라 이전 노드로도 이동할 수 있다.
  - 예) `node.next`를 호출하면 다음 노드로, `node.prev`를 호출하면 이전 노드로 이동한다.
- 마지막 노드에 대한 참조를 제공한다. 따라서 데이터를 마지막에 추가하는 경우에도 O(1)의 성능을 제공한다.
- 이전 노드로 이동할 수 있기 때문에 마지막 노드부터 앞으로, 그러니까 역방향으로 조회할 수 있다.
  - 덕분에 인덱스 조회 성능을 최적화 할 수 있다.
  - 예를 들어 인덱스로 조회하는 경우 인덱스가 사이즈의 절반 이하라면 처음부터 찾아서 올라가고, 인덱스가 사이즈의 절반을 넘으면 마지막 노드 부터 역방향으로 조회해서 성능을 최적화 할 수 있다.

자바의 LinkedList 클래스

```
class Node {
    E item;
    Node next;
    Node prev;
}

class LinkedList {
    Node first; //첫 번째 노드 참조
    Node last; //마지막 노드 참조
    int size;
}
```

자바 리스트의 성능 비교

자바가 제공하는 리스트의 성능을 비교해보자.

기존에 만들었던 `MyListPerformanceTest` 코드를 복사해서 자바의 리스트를 사용하도록 일부를 코드를 수정하면 된다.

수정 코드

- `MyList` ➜ `List`
- `MyArrayList` ➜ `ArrayList`
- `MyLinkedList` ➜ `LinkedList`

```
package collection.list;

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

public class JavaListPerformanceTest {

    public static void main(String[] args) {
        int size = 50_000;
        System.out.println("==ArrayList 추가==");
        addFirst(new ArrayList<>(), size);
        addMid(new ArrayList<>(), size);
        ArrayList<Integer> arrayList = new ArrayList<>(); //조회용 데이터로 사용
        addLast(arrayList, size);

        System.out.println("==LinkedList 추가==");
        addFirst(new LinkedList<>(), size);
        addMid(new LinkedList<>(), size);
        LinkedList<Integer> linkedList = new LinkedList<>(); //조회용 데이터로 사용
        addLast(linkedList, size);

        int loop = 10000;
        System.out.println("==ArrayList 조회==");
        getIndex(arrayList, loop, 0);
        getIndex(arrayList, loop, size / 2);
        getIndex(arrayList, loop, size - 1);
```

```
        System.out.println("==LinkedList 조회==");
        getIndex(linkedList, loop, 0);
        getIndex(linkedList, loop, size / 2);
        getIndex(linkedList, loop, size - 1);

        System.out.println("==ArrayList 검색==");
        search(arrayList, loop, 0);
        search(arrayList, loop, size / 2);
        search(arrayList, loop, size - 1);

        System.out.println("==LinkedList 검색==");
        search(linkedList, loop, 0);
        search(linkedList, loop, size / 2);
        search(linkedList, loop, size - 1);
    }

    private static void addFirst(List<Integer> list, int size) {
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            list.add(0, i);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("앞에 추가 - 크기: " + size + ", 계산 시간: " + (endTime
- startTime) + "ms");
    }

    private static void addMid(List<Integer> list, int size) {
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            list.add(i / 2, i);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("평균 추가 - 크기: " + size + ", 계산 시간: " + (endTime
- startTime) + "ms");
    }

    private static void addLast(List<Integer> list, int size) {
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            list.add(i);
        }
        long endTime = System.currentTimeMillis();
```

```
        System.out.println("뒤에 추가 - 크기: " + size + ", 계산 시간: " + (endTime
- startTime) + "ms");
    }

    private static void getIndex(List<Integer> list, int loop, int index) {
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < loop; i++) {
            list.get(index);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("index: " + index + ", 반복: " + loop + ", 계산 시간: "
+ (endTime - startTime) + "ms");
    }

    private static void search(List<Integer> list, int loop, int findValue) {
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < loop; i++) {
            list.indexOf(findValue);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("findValue: " + findValue + ", 반복: " + loop + ", 계
산 시간: " + (endTime - startTime) + "ms");
    }
}
```

실행 결과

```
==ArrayList 추가==
앞에 추가 - 크기: 50000, 계산 시간: 106ms
평균 추가 - 크기: 50000, 계산 시간: 49ms
뒤에 추가 - 크기: 50000, 계산 시간: 1ms
==LinkedList 추가==
앞에 추가 - 크기: 50000, 계산 시간: 2ms
평균 추가 - 크기: 50000, 계산 시간: 1116ms
뒤에 추가 - 크기: 50000, 계산 시간: 2ms
==ArrayList 조회==
index: 0, 반복: 10000, 계산 시간: 1ms
index: 25000, 반복: 10000, 계산 시간: 0ms
index: 49999, 반복: 10000, 계산 시간: 1ms
==LinkedList 조회==
index: 0, 반복: 10000, 계산 시간: 0ms
index: 25000, 반복: 10000, 계산 시간: 439ms
```

```
index: 49999, 반복: 10000, 계산 시간: 1ms
==ArrayList 검색==
findValue: 0, 반복: 10000, 계산 시간: 0ms
findValue: 25000, 반복: 10000, 계산 시간: 104ms
findValue: 49999, 반복: 10000, 계산 시간: 218ms
==LinkedList 검색==
findValue: 0, 반복: 10000, 계산 시간: 1ms
findValue: 25000, 반복: 10000, 계산 시간: 473ms
findValue: 49999, 반복: 10000, 계산 시간: 945ms
```

직접 만든 배열 리스트와 연결 리스트 - 성능 비교 표

기능	배열 리스트	연결 리스트
앞에 추가(삭제)	O(n) - 1369ms	O(1) - 2ms
평균 추가(삭제)	O(n) - 651ms	O(n) - 1112ms
뒤에 추가(삭제)	O(1) - 2ms	O(n) - 2195ms
인덱스 조회	O(1) - 1ms	O(n) - 평균 438ms
검색	O(n) - 평균 115ms	O(n) - 평균 492ms

자바가 제공하는 배열 리스트와 연결 리스트 - 성능 비교 표

기능	배열 리스트	연결 리스트
앞에 추가(삭제)	O(n) - 106ms	O(1) - 2ms
평균 추가(삭제)	O(n) - 49ms	O(n) - 1116ms
뒤에 추가(삭제)	O(1) - 1ms	O(1) - 2ms
인덱스 조회	O(1) - 1ms	O(n) - 평균 439ms
검색	O(n) - 평균 104ms	O(n) - 평균 473ms

추가, 삭제

- 배열 리스트는 인덱스를 통해 추가나 삭제할 위치를 O(1)로 빠르게 찾지만, 추가나 삭제 이후에 데이터를 한칸씩 밀어야 한다. 이 부분이 O(n)으로 오래 걸린다.
- 연결 리스트는 인덱스를 통해 추가나 삭제할 위치를 O(n)으로 느리게 찾지만, 실제 데이터의 추가는 간단한 참조

변경으로 O(1)로 빠르게 수행된다.

앞에 추가(삭제)

- 배열 리스트: 추가나 삭제할 위치는 찾는데 O(1), 데이터를 한칸씩 이동 O(n) ➜ O(n)
- 연결 리스트: 추가나 삭제할 위치는 찾는데 O(1), 노드를 변경하는데 O(1) ➜ O(1)

평균 추가(삭제)

- 배열 리스트: 추가나 삭제할 위치는 찾는데 O(1), 인덱스 이후의 데이터를 한칸씩 이동 O(n/2) ➜ O(n)
- 연결 리스트: 추가나 삭제할 위치는 찾는데 O(n/2), 노드를 변경하는데 O(1) ➜ O(n)

뒤에 추가(삭제)

- 배열 리스트: 추가나 삭제할 위치는 찾는데 O(1), 이동할 데이터 없음 ➜ O(1)
- 연결 리스트: 추가나 삭제할 위치는 찾는데 O(1), 노드를 변경하는데 O(1) ➜ O(1)
    - 참고로 자바가 제공하는 연결 리스트(`LinkedList`)는 마지막 위치를 가지고 있다.

인덱스 조회

- 배열 리스트: 배열에 인덱스를 사용해서 값을 O(1)로 찾을 수 있음
- 연결 리스트: 노드를 인덱스 수 만큼 이동해야함 O(n)

검색

- 배열 리스트: 데이터를 찾을 때 까지 배열을 순회 O(n)
- 연결 리스트: 데이터를 찾을 때 까지 노드를 순회 O(n)

데이터를 추가할 때 자바 ArrayList가 직접 구현한 MyArrayList보다 빠른 이유

- 자바의 배열 리스트는 이때 메모리 고속 복사를 사용하기 때문에 성능이 최적화된다.
- 메모리 고속 복사는 시스템에 따라 성능이 다르기 때문에 정확한 계산은 어렵지만 대략 O(n/10) 정도로 추정하자. 상수를 제거하면 O(n)이 된다. 하지만 메모리 고속 복사라도 데이터가 아주 많으면 느려진다.

시간 복잡도와 실제 성능

- 이론적으로 `LinkedList`의 중간 삽입 연산은 `ArrayList`보다 빠를 수 있다. 그러나 실제 성능은 요소의 순차적 접근 속도, 메모리 할당 및 해제 비용, CPU 캐시 활용도 등 다양한 요소에 의해 영향을 받는다.
    - 추가로 `ArrayList`는 데이터를 한 칸씩 직접 이동하지 않고, 대신에 메모리 고속 복사를 사용한다.
- `ArrayList`는 요소들이 메모리 상에서 연속적으로 위치하여 CPU 캐시 효율이 좋고, 메모리 접근 속도가 빠르다.
- 반면, `LinkedList`는 각 요소가 별도의 객체로 존재하고 다음 요소의 참조를 저장하기 때문에 CPU 캐시 효율이 떨어지고, 메모리 접근 속도가 상대적으로 느려질 수 있다
- `ArrayList`의 경우 `CAPACITY`를 넘어서면 배열을 다시 만들고 복사하는 과정이 추가된다. 하지만 한번에

50%씩 늘어나기 때문에 이 과정은 가끔 발생하므로, 전체 성능에 큰 영향을 주지는 않는다.

정리하면 이론적으로 `LinkedList`가 중간 삽입에 있어 더 효율적일 수 있지만, 현대 컴퓨터 시스템의 메모리 접근 패턴, CPU 캐시 최적화, 메모리 고속 복사 등을 고려할 때 `ArrayList`가 실제 사용 환경에서 더 나은 성능을 보여주는 경우가 많다.

배열 리스트 vs 연결 리스트

대부분의 경우 배열 리스트가 성능상 유리하다. 이런 이유로 실무에서는 주로 배열 리스트를 기본으로 사용한다.
만약 데이터를 앞쪽에서 자주 추가하거나 삭제할 일이 있다면 연결 리스트를 고려하자.

문제와 풀이1

문제1 - 배열을 리스트로 변경하기

문제 설명

- `ArrayEx1`는 배열을 사용한다. 이 코드를 배열 대신에 리스트를 사용하도록 변경하자.
- 다음 코드와 실행 결과를 참고해서 리스트를 사용하는 `ListEx1` 클래스를 만들어라.

```
package collection.list.test.ex1;

public class ArrayEx1 {
    public static void main(String[] args) {
        int[] students = {90, 80, 70, 60, 50};

        int total = 0;
        for (int i = 0; i < students.length; i++) {
            total += students[i];
        }

        double average = (double) total / students.length;
        System.out.println("점수 총합: " + total);
        System.out.println("점수 평균: " + average);
    }
}
```

실행 결과

```
점수 총합: 350
점수 평균: 70.0
```

정답 - 클래스

```java
package collection.list.test.ex1;

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

public class ListEx1 {
    public static void main(String[] args) {
        List<Integer> students = new ArrayList<>();
        students.add(90);
        students.add(80);
        students.add(70);
        students.add(60);
        students.add(50);

        int total = 0;
        for (int i = 0; i < students.size(); i++) {
            total += students.get(i);
        }

        double average = (double) total / students.size();
        System.out.println("점수 총합: " + total);
        System.out.println("점수 평균: " + average);
    }
}
```

문제2 - 리스트의 입력과 출력

문제 설명

사용자에게 `n`개의 정수를 입력받아서 `List`에 저장하고, 입력 순서대로 출력하자.

`0`을 입력하면 입력을 종료하고 결과를 출력한다.

출력시 출력 포멧은 1, 2, 3, 4, 5와 같이 `, `쉼표를 사용해서 구분하고, 마지막에는 쉼표를 넣지 않아야 한다.
실행 결과 예시를 참고하자.
문제는 `ListEx2`에 풀자

실행 결과

```
n개의 정수를 입력하세요 (종료 0)
1
2
3
4
5
0
출력
1, 2, 3, 4, 5
```

정답

```java
package collection.list.test.ex1;

import java.util.ArrayList;
import java.util.Scanner;

public class ListEx2 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        ArrayList<Integer> numbers = new ArrayList<>();

        System.out.println("n개의 정수를 입력하세요 (종료 0)");
        while (true) {
            int input = scanner.nextInt();
            if (input == 0) {
                break;
            }
            numbers.add(input);
        }

        System.out.println("출력");
        for (int i = 0; i < numbers.size(); i++) {
            System.out.print(numbers.get(i));
```

```
            if (i < numbers.size() - 1) {
                System.out.print(", ");
            }
        }
    }
}
```

문제3 - 합계와 평균

사용자에게 `n`개의 정수를 입력받아서 `List`에 보관하고, 보관한 정수의 합계와 평균을 계산하는 프로그램을 작성하자.

`ListEx3`에 작성하자.

실행 결과 예시

```
n개의 정수를 입력하세요 (종료 0)
1
2
3
4
5
0
입력한 정수의 합계: 15
입력한 정수의 평균: 3.0
```

정답

```
package collection.list.test.ex1;

import java.util.ArrayList;
import java.util.Scanner;

public class ListEx3 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        ArrayList<Integer> numbers = new ArrayList<>();
```

```
        System.out.println("n개의 정수를 입력하세요 (종료 0)");
        while (true) {
            int input = scanner.nextInt();
            if (input == 0) {
                break;
            }
            numbers.add(input);
        }

        int sum = 0;
        for (Integer number : numbers) {
            sum += number;
        }
        double average = (double) sum / numbers.size();

        System.out.println("입력한 정수의 합계: " + sum);
        System.out.println("입력한 정수의 평균: " + average);
    }
}
```

문제와 풀이2

문제 - 리스트를 사용한 쇼핑 카트

`ShoppingCartMain` 코드가 작동하도록 `ShoppingCart` 클래스를 완성해라.

`ShoppingCart` 는 내부에 리스트를 사용해야 한다.

Item 클래스

```
package collection.list.test.ex2;

public class Item {

    private String name;
    private int price;
    private int quantity;

    public Item(String name, int price, int quantity) {
```

```
        this.name = name;
        this.price = price;
        this.quantity = quantity;
    }

    public String getName() {
        return name;
    }

    public int getTotalPrice() {
        return price * quantity;
    }
}
```

```
package collection.list.test.ex2;

public class ShoppingCartMain {

    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        Item item1 = new Item("마늘", 2000, 2);
        Item item2 = new Item("상추", 3000, 4);

        cart.addItem(item1);
        cart.addItem(item2);

        cart.displayItems();
    }
}
```

실행 결과

```
장바구니 상품 출력
상품명:마늘, 합계:4000
상품명:상추, 합계:12000
전체 가격 합:16000
```

ShoppingCart - 코드 작성

```
package collection.list.test.ex2;

public class ShoppingCart {
    // 코드 작성
}
```

정답 - ShoppingCart 클래스

```
package collection.list.test.ex2;

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

public class ShoppingCart {

    private List<Item> items = new ArrayList<>();

    public void addItem(Item item) {
        items.add(item);
    }

    public void displayItems() {
        System.out.println("장바구니 상품 출력");
        for (Item item : items) {
            System.out.println("상품명:" + item.getName() + ", 합계:" +
item.getTotalPrice());
        }
        System.out.println("전체 가격 합:" + calculateTotalPrice());
    }

    private int calculateTotalPrice() {
        int totalPrice = 0;
        for (Item item : items) {
            totalPrice += item.getTotalPrice();
        }
        return totalPrice;
    }
```

```
}
```

다음 배열을 사용한 코드와 비교해보면 배열보다 리스트를 사용하는 것이 이점이 더 많은 것을 확인할 수 있다.

배열과 비교한 리스트의 이점

- 자료 구조의 크기가 동적으로 증가한다. 따라서 배열처럼 입력 가능한 크기를 미리 정하지 않아도 된다.
- `itemCount`와 같이 배열에 몇게의 데이터가 추가 되었는지 추적하는 변수를 제거할 수 있다. 리스트는 `size()` 메서드를 통해 입력된 데이터의 크기를 제공한다.

참고 - 배열을 사용한 코드와 비교

```
public class ShoppingCart {
    private Item[] items = new Item[10];
    private int itemCount;

    public void addItem(Item item) {
        if (itemCount >= items.length) {
            System.out.println("장바구니가 가득 찼습니다.");
            return;
        }

        items[itemCount] = item;
        itemCount++;
    }

    public void displayItems() {
        System.out.println("장바구니 상품 출력");
        for (int i = 0; i < itemCount; i++) {
            Item item = items[i];
            System.out.println("상품명:" + item.getName() + ", 합계:" +
item.getTotalPrice());
        }
        System.out.println("전체 가격 합:" + calculateTotalPrice());
    }

    private int calculateTotalPrice() {
        int totalPrice = 0;
        for (int i = 0; i < itemCount; i++) {
            Item item = items[i];
            totalPrice += item.getTotalPrice();
```

```
        }
        return totalPrice;
    }

}
```

정리