자료구조: 2022년 1학기 [실습]

제 6 주:

리스트 성능비교

강 지 훈
jhkang@cnu.ac.kr
충남대학교 컴퓨터융합학부

실습 목표

실습 목표

- 이론적 관점
  - 리스트의 다양한 구현 방법에 따른 차이점
    - 기능 별 장단점을 이해한다.
  - 프로그램의 성능 측정 방법

- 구현적 관점
  - 다양한 리스트를 구현
  - 리스트의 각 구현에 따른 성능을 측정한다.
  - 실체 측정 값들이 이론적 예측과 일치하는지 확인한다.

수행 시간 측정 방법

시간 비교

- System.nanoTime()
  - 자바에서 현재 시간을 얻어오는 시스템 메소드
  - 단위 : nano(=10^{-9}) second
  - Long 타입으로 저장
  - 예시

```
long startTime = System.nanoTime() ;
함수_실행() ;
long stopTime = System.nanoTime() ;
long duration= (stopTime - startTime) ;
System.out.println(" 수행 시간 = " + duration + " nano sec") ;
```

  - 자료의 입력 크기가 작을 경우, 요즈음 컴퓨터의 성능이 좋아 mili second (또는 micro second) 단위로는 확인이 불가능하여 JDK 1.5 이후 추가됨.

실험 데이터 생성

난수를 이용하자 !

- 난수 (Random Number)
  - 아무 규칙 없이, 무작위적으로 얻어지는 수
  - 하나의 난수를 얻고 나서, 그 다음 난수를 얻을 때 이전 난수와 어떠한 연관관계도 없이 무작위적으로 얻어져야 한다.

난수를 이용한 실험 데이터 생성 [O]

```
import java.util.*;
public class ……
{
    ……

    public void  doSomething (……) {
        ……
        Random  random = new Random() ;
        int  data[MaxSize] ;
        int   i = 0 ;

        while ( i < MaxSize ) {
            data[i] = random.nextInt( MAX_SIZE ) ;
            i++ ;
        }
        ……
    }
    ……
}
```

실험 데이터를 무작위로 MaxSize 만큼 생성하여 배열 "data[]" 에 저장하려고 한다.

난수를 이용한 실험 데이터 생성 [4]

```
.....
Random random = new Random();
int data[MaxSize];
int i = 0 ;

while ( i < MaxSize ) {
    data[i] = random.nextInt( MAX_SIZE );
    i++ ;
}
.....
```

난수 생성을 하는 객체를 생성한다.

"nextInt()"
정수형 난수를 얻는다.

"MAX_SIZE"
0~(Maxsize-1) 사이의 정수를 얻게 해 준다.

유사 난수란?

- 난수 (Random Number)
  - 아무 규칙 없이, 무작위적으로 얻어지는 수
  - 하나의 난수를 얻고 나서 그 다음 난수를 얻을 때, 이전 난수와 어떠한 연관관계도 없이 무작위적으로 얻어져야 한다.

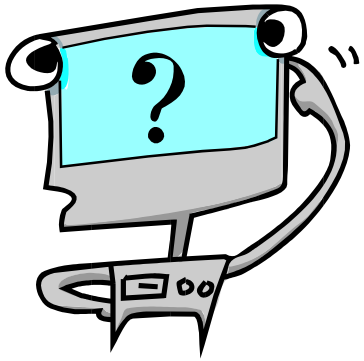
- 유사 난수 (Pseudo Random Number)
  - 엄격히 말해서, 난수는 계속 무작위적으로 수가 얻어져야 한다.
  - nextInt ()를 이용하여 계속 무작위적으로 다음 수를 얻을 수 있을까?

난수 대신 유사난수 ?

- 난수
  - 5, 2, 1, 4, 6, 3, 1, 5, 6, 3, 4, 2, ...

- 유사난수
  - 5, 2, 1, 4, 6, 3, 5, 2, 1, 4, 6, 3, 5, 2, 1, 4, 6, 3,.....

반복되는 sequence가
매우 길다면?

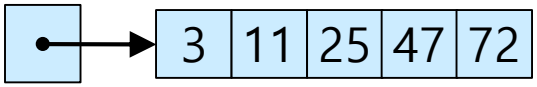


<단계 1>

과제에서 해결할 문제

문제

- 리스트의 구현에 따른 성능을 측정한다.
  - 먼저 Sorted Array List 에 대한 성능 측정을 수행하여 본다.
    - Sorted Array List: 오름차순 (Non-decreasing order)



  - 성능 측정을 할 행위
    - 데이터의 삽입
    - 최대값 검색

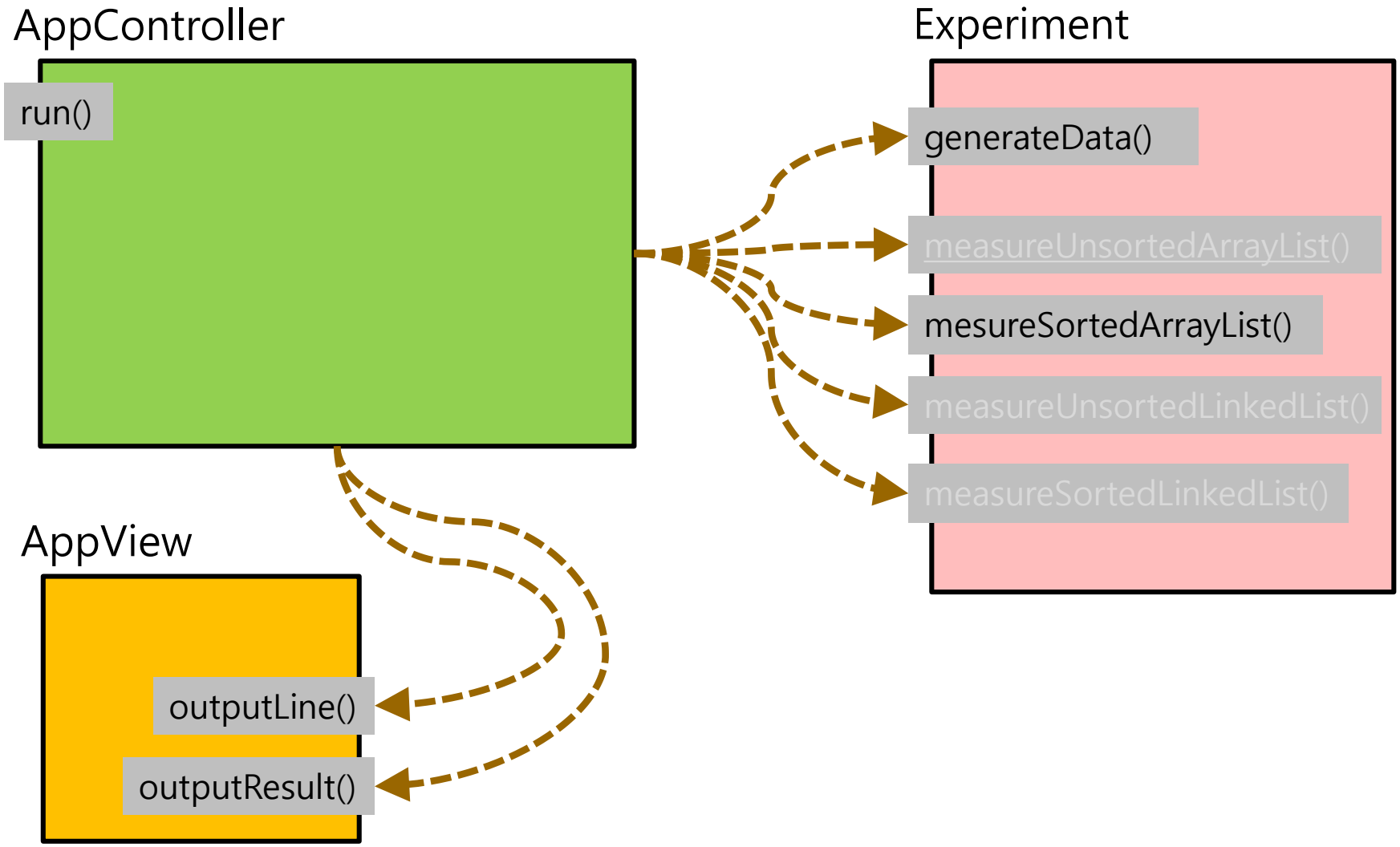
입출력

- 입출력
  - 입력:
    - 없음
    - 필요한 데이터는 프로그램에서 생성
  - 출력: 성능 측정 결과:
    - 데이터 크기 변화에 따른 성능 측정 결과
      - 10000, 20000, 30000, 40000, 50000

출력의 예

```
<<< 리스트의 성능 측정 프로그램을 시작합니다 >>>
! 리스트의 구현에 따른 성능의 차이를 알아봅니다: (단위: Micro Second)

<Sorted Array List>
[크기: 10000]  삽입:   60730,  최대값:     388
[크기: 20000]  삽입:  274788,  최대값:     806
[크기: 30000]  삽입:  689040,  최대값:    1341
[크기: 40000]  삽입: 1257857,  최대값:    1694
[크기: 50000]  삽입: 2169945,  최대값:    2258

<<< 리스트의 성능 측정 프로그램을 종료합니다 >>>
```

이 과제에서 필요한 객체는?



이 과제에서 필요한 Class 는?

- AppController
- AppView

- Model
  - Experiment
  - MeasuredResult
  - UnsortedArrayList<E extends Comparable<E>>
  - SortedArrayList<E extends Comparable<E>>
  - UnsortedLinkedList<E extends Comparable<E>>
  - SortedLinkedList<E extends Comparable<E>>
  - Coin implements Comparable<Coin>
  - LinkedNode<E>

<단계 1>의 구현

main()

main()을 위한 class

```
public class _DS06_학번_이름 {
    public static void main (String[] args)
    {
        AppController  appController = new AppController() ;
            // AppController 가 실질적인 main class 이다.
        appController.run() ;
            // 여기 main()에서는 앱 실행이 시작되도록 해주는 일이 전부이다.
    }
}
```

Class “AppController”

AppController: 인스턴스 변수, Getter/Setter

```
public class AppController {
  // 비공개 인스턴스 변수들
  private Experiment  _experiment ;

  // Getter/Setter
  private Experiment experiment() {
        return this._experiment ;
  }
  private void setExperiment (Experiment newExperiment) {
        this._experiment = newExperiment ;
  }

  // 생성자
  public AppController() {
        this.setExperiment (new Experiment()) ;
        this.experiment().generateData();
  }

  // 비공개함수의 구현
  .....

  // 공개함수의 구현
  .....
} // End of class "AppController"
```

비공개 인스턴스 변수
(Private instance variable)
- 어떠한 경우에도 인스턴스 변수를 "public" 으로 선언하지 않는다.

인스턴스 변수를 위한 Getter/Setter:
- 언제나 모든 인스턴스 변수에 대해 getter/setter 를 만든다.
- Class 내부에서만 사용할 경우에는 private 으로 선언한다.
- Class 내부에서도 직접 instance variable 을 사용하지 않고, 반드시 getter/setter 를 통해서 사용하기로 한다. (이유는?)

AppController: 생성자

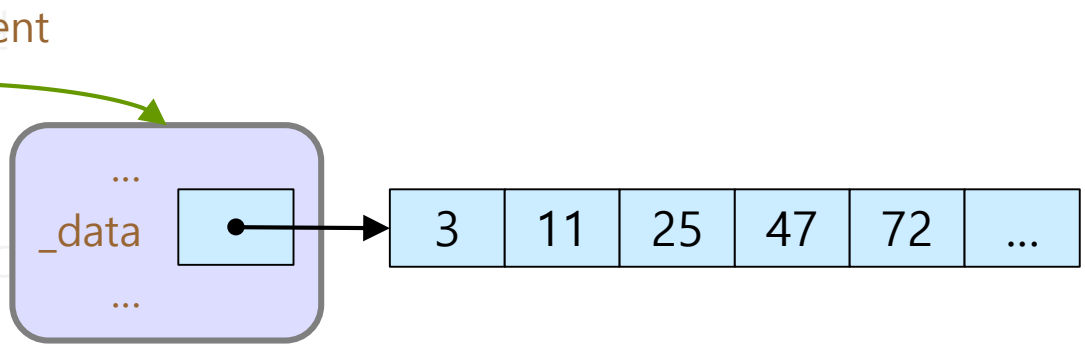
```
public class AppController {
   // 비공개 변수들
   private Experiment  _experiment;

   // Getters/Setters
   private Experiment experiment() {
        return this._experiment;
   }
   private void setExperiment (Experiment newExperiment) {
        this._experiment = newExperiment;
   }

   // 생성자
   public AppController() {
        this.setExperiment (new Experiment()) ;
        this.experiment().generateData();
   }

   // 비공개함수의 구현
   ……

   // 공개함수의 구현
   ……
} // End of class "AppController"
```

실험 객체를 생성한다.

AppController: 인스턴스 변수, 생성자

```
public class AppController {
  // 비공개 변수들
  private Experiment  _experiment ;

  // Getters/Setters
  private Experiment experiment() {
      return  this._experiment ;
  }
  private void setExperiment (Experiment newExperiment) {
      this._experiment = newExperiment ;
  }

  // 생성자
  public AppController() {
      this.setExperiment (new Experiment()) ;
      this.experiment().generateData();
  }
```

실험 객체에게 성능 측정에 사용할 데이터를 생성하게 한다.



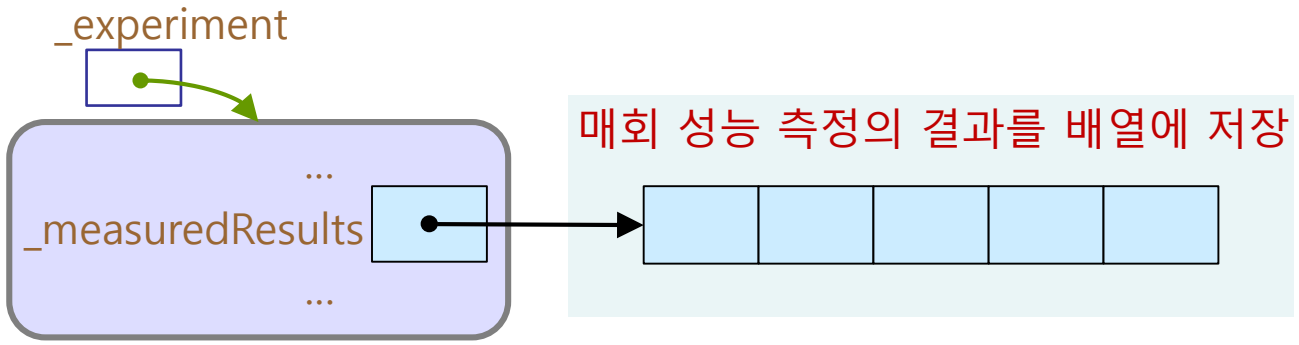
AppController: run()

```
public class AppController {
      ......

      // 공개함수의 구현
       public void run() {
            AppView.outputLine ("<<<리스트 성능 측정 프로그램을 시작합니다.>>>") ;
            AppView.outputLine ("! 리스트의 구현에 따른 시간의 차이를 알아봅니다: (단위: Micro Second)") ;

            AppView.outputLine ("") ;
            AppView.outputLine ("<Sorted Array List>") ;
            this.experiment().measureForSortedArrayList() ;
            this.showExperimentResults() ;

            AppView.outputLine("<<<리스트 성능 측정 프로그램을 종료합니다.>>>") ;
      }
```

AppController: run()

```
public class AppController {
      ......

      // 공개함수의 구현
       public void run() {
            AppView.outputLine ("<<<리스트 성능 측정 프로그램을 시작합니다.>>>") ;
            AppView.outputLine ("! 리스트의 구현에 따른 시간의 차이를 알아봅니다: (단위: Micro Second)") ;

            AppView.outputLine ("") ;
            AppView.outputLine ("<Sorted Array List>") ;
            this.experiment().measureForSortedArrayList() ;
            this.showExperimentResults();

            AppView.outputLine("<<<리스트 성능 측정 프로그램을 종료합니다.>>>") ;
      }
```

실험 객체에게 "SortedArrayList" 에 대한 성능 측정을 실행하게 한다.



AppController: run()

```
public class AppController {
      ......

      // 공개함수의 구현
       public void run() {
            AppView.outputLine ("<<<리스트 성능 측정 프로그램을 시작합니다.>>>") ;
            AppView.outputLine ("! 리스트의 구현에 따른 시간의 차이를 알아봅니다: (단위: Micro Second)") ;

            AppView.outputLine ("");
            AppView.outputLine ("<Sorted Array List>") ;
            this.experiment().measureSortedArrayList() ;
            this.showExperimentResults() ;

            AppView.outputLine("<<<리스트 성능 측정 프로그램을 종료합니다.>>>") ;
      }
```

실험 결과를 출력한다.

AppController: showExperimentResults()

```
public class AppController {
    ......
     // 비공개함수의 구현
    private void  showExperimentResults () {
        MeasuredResult[]  results = this.experiment().measuredResults () ;
        for ( int  i = 0 ; i < this.experiment().numberOfIteration () ; i++ ) {
            AppView.outputResult (
                results[i].size(),
                results[i].durationForAdd() / 1000,      // Nano 를 Micro 로 변환
                results[i].durationForMax() / 1000       // Nano 를 Micro 로 변환
            ) ;
        }
    }
```

Class “AppView”

AppView: scanner

```
public class AppView {
  // 비공개 상수/변수들
  private static Scanner  scanner = new Scanner(System.in) ;

  // 생성자
  private AppView ()
  {
  }

  // 공개함수의 구현
  ……
}
```

이번주 실습에서는 입력 받는 것이 없으므로, 삭제해도 됨.
그렇지만, 지난 주의 것을 복사해 사용할 경우, scanner 를 사용하는 입력 함수가 있다면 그대로 놓아두어도 될 것임.
(물론, 입력 함수는 있다고 해도 사용되지는 않음)

AppView: outputResults()

```
public class AppView {
  // 비공개 상수/변수들
  .....
  // 생성자
  ......

  // 공개 함수의 구현
  public static void outputLine (String aMessage) {...}
  public static void output (String aMessage) {...}

  public static void outputResults
              (int size, long durationForAdd, long durationForMax)
  {
      AppView.outputLine (
          "[크기: "    + String.format("%5d", size) + "]   " +
          "삽입: "     + String.format("%8d", durationForAdd) + ",   " +
          "최대값: " + String.format("%8d", durationForMax)
      ) ;
  }
```

Class "MeasuredResult"

MeasuredResult:

```
public class MeasuredResult {
    // Instance variables
    private int    _size ;
    private long  _durationForAdd ;
    private long  _durationForMax ;

    // Getter/Setter
    public int size() {...}
    public void setSize (int newSize ){...}

    public long durationForAdd() {...}
    public void setDurationForAdd (long newDurationForAdd) {...}

    public long durationForMax() {...}
    public void setDurationForMax (long newDurationForMaX) {...}

    // Constructors
    public MeasuredResult () {
        this (0, 0, 0) ;
    }
    public MeasuredResult
        (int givenSize, long givenDurationForAdd, long givenDurationForMax)
    {
        this.setSize(givenSize) ;
        this.setDurationForAdd (givenDurationForAdd) ;
        this.setDurationForMax (givenDurationForMax) ;
    }
```

시간 측정 결과의 자료형: long
- 우리가 시간 측정에 사용하는 시스템 시계의 시간의 단위는 nano second 이고, 자료형이 "long" 이다. 따라서, 자료형을 "long" 으로 동일하게 맞춘다.
- 이번 성능 측정 결과의 단위 역시 nano second.

실험 자체를 위한 추상자료형

Class "Experiment"

Experiment: 상수, 인스턴스 변수, Getters/Setters

```
public class Experiment {
    // Constants
    private static final int DEFAULT_NUMBER_OF_ITRATION = 5 ;
    private static final int DEFAULT_FIRST_SIZE = 10000 ;  // 첫 번째 실험 데이터 크기
    private static final int DEFAULT_SIZE_INCREMENT = 10000 ; // 실험 데이터 크기 증가량

    // Private instances
    private int          _numberOfIteration ;
    private int          _firstSize ;
    private int          _sizeIncrement ;
    private Coin[]       _data ;
    private MeasuredResult[]  _measuredResults ;

    // Getters/Setters
    public int numberOfIteration() {...}
    public void setNumberOfIteration (int newNumberOfIteration) {...}
    public int firstSize() {...}
    public void setFirstSize(int newFirstSize) {...}
    public int sizeIncrement() {...}
    public void setSizeIncrement (int newSizeIncrement) {...}

    public int maxSize() {...}

    public Coin[] data() {...}
    public void setData (Coin[] newData) {...}
    public MeasuredResult[] measuredResults () {...}
    public void setMeasuredResults (MeasuredResult[] newMeasuredResults) {...}
```

Experiment: 생성자, 공개함수

- 생성자
  - public Experiment()
  - public Experiment(
    int givenNumberOfIteration,
    int givenFirstSize,
    int givenSizeIncrement)

- public int maxSize()
  - 실험 데이터의 최대 크기를 계산하여 돌려준다.
  - 계산 방법: this.firstSize() + this.sizeIncrement() * (this.numberOfIteration() -1)

- public void generateData ()
  - 성능 측정에 필요한 데이터를 생성한다.
  - 난수를 사용한다.
  - 생성된 난수 값을 갖는 Coin 객체를 생성하여 저장한다.

- public MeasuredResult[] measuredResults ()
  - 성능 측정 결과를 돌려준다.

Experiment: 생성자, generateData()

```
public class Experiment {
    ...
    public Experiment() {
        this(DEFAULT_NUMBER_OF_ITRATION, DEFAULT_FIRST_SIZE, DEFAULT_SIZE_INCREMENT);
    }
    public Experiment
        (int givenNumberOfIteration, int givenFirstSize, int givenSizeIncrement)
    {
        this.setNumberOfIteration(givenNumberOfIteration) ;
        this.setFirstSize(givenFirstSize) ;
        this.setSizeIncrement(givenSizeIncrement) ;

        this.setData (new Coin[this.maxSize()]) ; // 실험 데이터를 담을 배열 공간 확보
        this.setMeasuredResults (new MeasuredResult[this.numberOfIteration()]) ;
            // 실험 결과를 저장할 배열 공간 확보
    }

    public void generateData() {
        Random  random = new Random() ;
        for (int i = 0 ; i  < this.maxSize() ; i++ ) {
            int randomCoinValue = random.nextInt(this.maxSize()) ;
            this.data()[i] = new Coin(randomCoinValue) ;
        }
    }
```

생성된 난수 "randomCoinValue" 를 사용하여 Coin 객체를 생성한다.

Experiment: 성능 측정 실행 공개함수

- public void measureForUnsortedArrayList ()
  - Unsorted Array 로 구현한 List 의 성능을 측정한다.
- public void measureForSortedArrayList ()
  - Sorted Array 로 구현한 List 의 성능을 측정한다.

<단계 1> 에서는 이것만 구현하여, 측정에 사용한다.
나머지는 <단계 2> 에서 구현한다.

- public void measureForUnsortedLinkedList ()
  - Unsorted Linked List 로 구현한 List 의 성능을 측정한다.
- Public void measureForSortedLinkedList ()
  - Sorted Linked List 로 구현한 List 의 성능을 측정한다.

Experiment : measureForSortedArrayList()

```
public void measureForSortedArrayList() {
    // Sorted Array 로 구현한 List 의 성능을 측정한다.

    @SuppressWarnings("unused")
    Coin  maxCoin ;

    long  durationForAdd, durationForMax ;
    long  start, stop ;

    int  dataSize = this.firstSize() ;
    for ( int  iteration = 0 ; iteration < this.numberOfIteration() ; iteration ++ ) {
        SortedArrayList<Coin> listOfCoins = new SortedArrayList<Coin>(dataSize) ;
        durationForAdd = 0 ;
        durationForMax = 0 ;
        for ( int i = 0 ; i < dataSize ;  i++ ) {
            start = System.nanoTime() ;
            listOfCoins.add (this.data()[i]) ;
            stop = System.nanoTime() ;
            durationForAdd += (stop - start) ;

            start = System.nanoTime() ;
            maxCoin = listOfCoins.max() ;
            stop = System.nanoTime() ;
            durationForMax+= (stop - start) ;
        }
        this.measuredResults()[iteration] =
            new MeasuredResult(dataSize, durationForAdd, durationForMax) ;
        dataSize += this.sizeIncrement() ;
    }
}
```

실행시간을 측정할 실행코드

Class "SortedArrayList"

SortedArrayList: 상수, 인스턴스, Getters/Setters

```
public class SortedArrayList<E extends Comparable<E>> {

    // Constants
    private static final int DEFAULT_CAPACITY = 100 ;

    // Private Instances
    private int  _capacity ;
    private int  _size ;
    private E[]  _elements ;

    // Getter/Setter
    public int capacity() {…}
    public void setCapacity(int newCapacity) {…}

    public int size() {…}
    public void setSize(int newSize) {…}

    public E[] elements() {…}
    public void setElements (E[] newElements) {…}

    // Constructors
    ……
```

SortedArrayList: 생성자

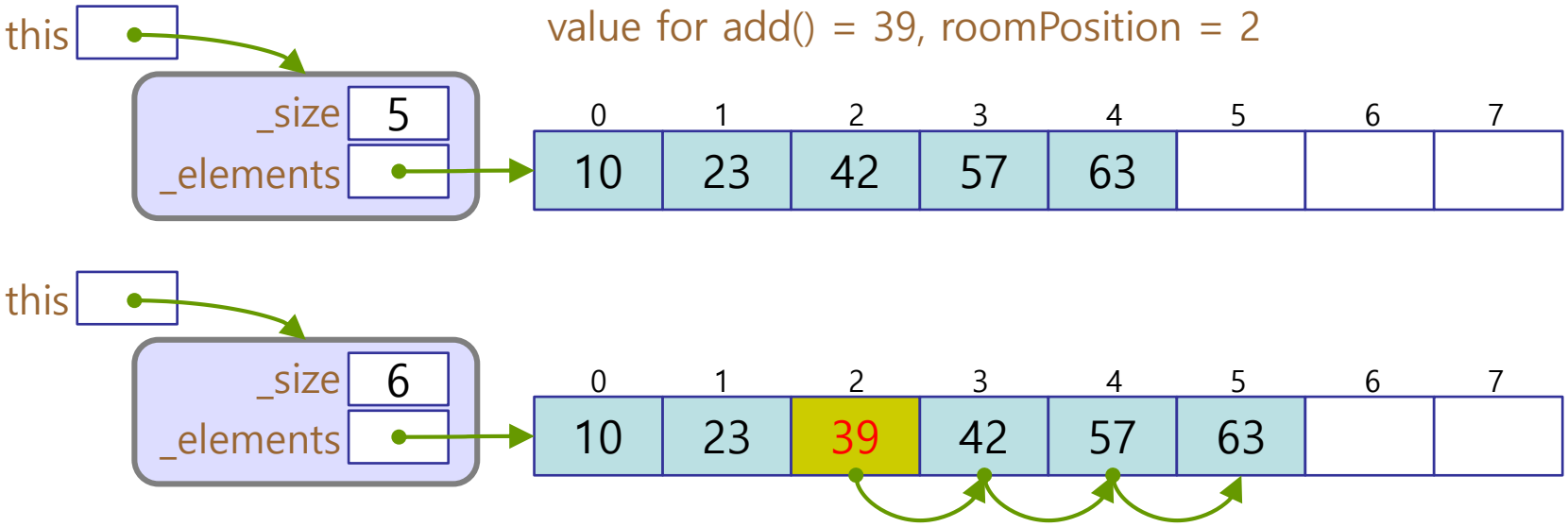
```
public class SortedArrayList<E extends Comparable<E>> {
    ......


    // Constructors
    public SortedArrayList () {
        this (SortedArrayList.DEFAULT_CAPACITY) ;
    }

    @SuppressWarnings("unchecked")
    public SortedArrayList (int givenCapacity) {
        this.setCapacity (givenCapacity) ;
        this.setElements ( (E[]) new Comparable [this.capacity()] );
    }
```

이 class 의 원소 "E" 는 반드시 "Comparable"을 구현하고 있는 것이 사용되어야 한다고 선언:
E 로 주어지는 실제 리스트의 원소의 class 에 Comparable 이 반드시 구현되어 있어야 한다.

"Object" 로 하면 오류가 발생:
- Generic type 을 원소로 하는 배열 객체는 생성할 수 없다.
  즉 "new E[this.capacity()]" 와 같이 하여 생성할 수 없다.
- 이 경우에 보통 원소의 type 으로 class "Object" 를 사용하여 생성.
- 그런데, 여기서는 "Object" 를 사용하면 오류가 발생한다.
- 그 이유는 원소 "E" 가 "Comparable" 을 구현하고 있다고 선언되어 있어서 이다. 즉 class "Object" 는 interface "Comparable" 을 구현하고 있지 않아서 "E[]" 로의 형 변환이 불가능하다.

SortedArrayList: add()

- 배열의 sort 순서에 맞는 위치를 찾아 삽입한다:
  - 먼저, 순서에 맞는 위치를 찾는다.
    - 비교는 compareTo() 사용
  - 그 다음, 뒤에서부터 찾은 위치까지 하나씩 뒤로 미루어 자리를 만든다.
    - makeRoomAt() 사용
  - 찾은 자리에 삽입할 값을 저장한다.



SortedArrayList: add()

- 구현 방법은, 강의 슬라이드와 지난 실습 내용을 참고하여 작성할 것.

- 힌트:
  - 정렬이 되어 있으므로, 삽입할 위치를 먼저 찾아야 한다.
    - 단순히 while-loop 을 사용하거나, 아니면 이진 검색을 한다.
  - 위치를 찾은 다음에, 그 위치 이후의 원소를 모두 뒤로 이동시킨다.
    - makeRoomAt() 을 사용.

SortedArrayList: add()

- 원소 끼리의 대소 관계 비교는?
- 원소 객체를 표현하는 class 에 비교 기능을 추가해야 한다:
  - interface "Comparable" 를 구현
  - class "Coin" 에 다음의 내용을 추가한다.

```
public class Coin implements Comparable<Coin> {

    ……

    @Override
    public int compareTo(Coin aCoin) {
        if ( this.value() < aCoin.value() ) {
            return  -1;
        }
        else if (this.value > aCoin.value() ) {
            return  +1 ;
        }
        else {
            return  0 ;
        }
    }
```

SortedArrayList: max()

- SortedArrayList 에서 가장 큰 값은 맨 뒤에 있다.
  - [this.size()-1] 위치가 된다.
  - Time Complexity: O(1)

```
public E   max() {
      if ( this.isEmpty() ) {
            return   null ;
      }
      else {
            return   this.elements()[this.size()-1] ;
      }
}
```

<단계 1> 의 결과 확인

❑ 〈단계 1〉 출력의 예

```
<<< 리스트의 성능 측정 프로그램을 시작합니다 >>>
! 리스트의 구현에 따른 성능의 차이를 알아봅니다: (단위: Micro Second)

<Sorted Array List>
[크기: 10000]  삽입:   60730,  최대값:     388
[크기: 20000]  삽입:  274788,  최대값:     806
[크기: 30000]  삽입:  689040,  최대값:    1341
[크기: 40000]  삽입: 1257857,  최대값:    1694
[크기: 50000]  삽입: 2169945,  최대값:    2258

<<< 리스트의 성능 측정 프로그램을 종료합니다 >>>
```

데이터를 해석하자

- 출력 예제와 비슷한 결과를 보여주고 있는가?
  - 차이가 난다면, 그 이유는 무엇일까?
- 출력된 성능측정 결과 데이터의 의미를 확인한다.
  - 각 리스트 구현 방법에 대해, 데이터 크기를 증가함에 따라 기능에 대한 성능 측정 결과가, 적정하게 나타나고있는가?

데이터를 해석하자: max()

- 데이터의 크기가 n 일 때: 최대값은 O(1).

- 예측과 실제:
  - 예측: 측정 값은, 크기에 비례하여 증가할 것이다.
  - 실제 측정 값은?

- 분석:
  - 크기 10000 일 때, 실제 측정 값은 388 마이크로 초.
  - 크기 20000 이면, 388*2 => 추정 값 776. 실제 값 806.
  - 크기 30000 이면, 388*3 => 추정 값 1164. 실제 값 1341.
  - 크기 40000 이면, 388*4 => 추정 값 1552. 실제 값 1694.
  - 크기 50000 이면, 388*5 => 추정 값 1940. 실제 값 2258.

데이터를 해석하자: add()

```
<Sorted Array List>
[크기: 10000]  삽입:   60730,
[크기: 20000]  삽입:  274788,
[크기: 30000]  삽입:  689040,
[크기: 40000]  삽입: 1257857,
[크기: 50000]  삽입: 2169945,
```

- 데이터의 크기가 n 일 때: 삽입은 O(n).
- 예측과 실제:
  - 예측: 측정 값은, 크기의 제곱에 비례하여 증가할 것이다.
  - 실제 측정 값은?
- 분석:
  - 크기 10000 일 때, 실제 측정 값은 60730 마이크로 초.
  - 크기 20000 이면, $60730*2^2$ => 추정 값 242920. 실제 값 274788.
  - 크기 30000 이면, $60730*3^2$ => 추정 값 546570. 실제 값 689040.
  - 크기 40000 이면, $60730*4^2$ => 추정 값 917680. 실제 값 1257857.
  - 크기 50000 이면, $60730*5^2$ => 추정 값 1518250. 실제 값 2169945.
  - 실제 값이 추정 값 보다 큰 이유는?
    add() 의 시간 복잡도를 나타내는 다음 식을 보고 이유를 생각해 볼 것.
    - 한번 실행: T(n) = c1*n + c0 = O(n)
    - n 번 실행 n*T(n) = n * (c1*n + c0) = $O(n^2)$

<단계 2>

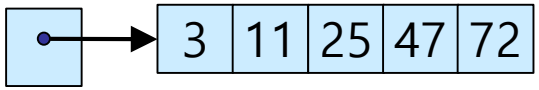
리스트 – 성능비교

과제에서 해결할 문제

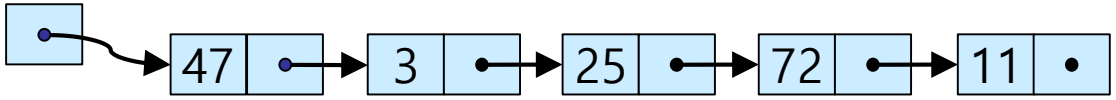
해야 할 과제

- 가방의 구현을 추가하고 성능을 측정하여 비교한다.
  - 측정을 위한 가방 구현 방법들
    - Unsorted Array List

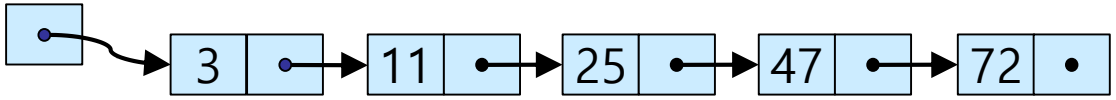
    - Sorted Array List (Non-decreasing order)



    - Unsorted Linked List



    - Sorted Linked List (Non-Decreasing order)



  - 성능 측정을 할 행위
    - 데이터의 삽입
    - 최대값 검색

출력의 예

```
<<< 리스트의 성능 측정 프로그램을 시작합니다 >>>
! 리스트의 구현에 따른 성능의 차이를 알아봅니다: (단위: Micro Second)

[Unsorted Array List]
[크기: 10000]  삽입:      548,  최대값:    48262
[크기: 20000]  삽입:      883,  최대값:   178482
[크기: 30000]  삽입:     1391,  최대값:   408172
[크기: 40000]  삽입:     1966,  최대값:   742515
[크기: 50000]  삽입:     2436,  최대값:  1072971

<Sorted Array List>
[크기: 10000]  삽입:    42374,  최대값:      305
[크기: 20000]  삽입:   220150,  최대값:      639
[크기: 30000]  삽입:   559638,  최대값:     1044
[크기: 40000]  삽입:  1013320,  최대값:     1437
[크기: 50000]  삽입:  1746024,  최대값:     1845

[Unsorted Linked List]
[크기: 10000]  삽입:      486,  최대값:   148561
[크기: 20000]  삽입:     1001,  최대값:   768023
[크기: 30000]  삽입:     1555,  최대값:  1736978
[크기: 40000]  삽입:     2116,  최대값:  3083808
[크기: 50000]  삽입:     2691,  최대값:  4813256

[Sorted Linked List]
[크기: 10000]  삽입:   152617,  최대값:   181562
[크기: 20000]  삽입:   865197,  최대값:  1150824
[크기: 30000]  삽입:  2277392,  최대값:  3332033
[크기: 40000]  삽입:  4384505,  최대값:  6744787
[크기: 50000]  삽입:  7094893,  최대값: 11300642

<<< 리스트의 성능 측정 프로그램을 종료합니다 >>>
```

구현할 내용

Class "AppController"

추가 구현

AppController: run() 수정

```
public class AppController {
    ......
    // 공개함수의 구현
     public void run() {
        AppView.outputLine ("<<<리스트 성능 측정 프로그램을 시작합니다.>>>");
        AppView.outputLine ("! 리스트의 구현에 따른 시간의 차이를 알아봅니다: (단위: Micro Second)");

        // Unsorted Array List 에 대한 측정
        AppView.outputLine ("");
        AppView.outputLine ("<UnSorted Array List>");
        this.experiment().measureForSortedArrayList();
        this.showExperimentResults();

        // Sorted Array List 에 대한 측정
        AppView.outputLine ("");
        AppView.outputLine ("<Sorted Array List>");
        this.experiment().measureForSortedArrayList();
        this.showExperimentResults();

        // Unsorted Linked List 에 대한 측정
        ......

        // Sorted Linked List 에 대한 측정
        ......

        AppView.outputLine("<<<리스트 성능 측정 프로그램을 종료합니다.>>>");
    }
```

추가한다

앞 부분을 참고하여 작성한다

Class Experiment"

구현 추가

다른 리스트 구현에 대한 실험도 같은 방법으로 !

- 다음 메소드를 추가 구현한다:
  - public void measureForUnsortedArrayList () {…}
  - public void measureForUnsortedLinkedList () {…}
  - public void measureForSortedLinkedList () {…}

  - <단계 1>에 만들었던 "measureForSortedArrayList()" 를 참고하여 구현한다.

각각의 구현 별 리스트의 삽입의 구현

각 클래스의 재사용

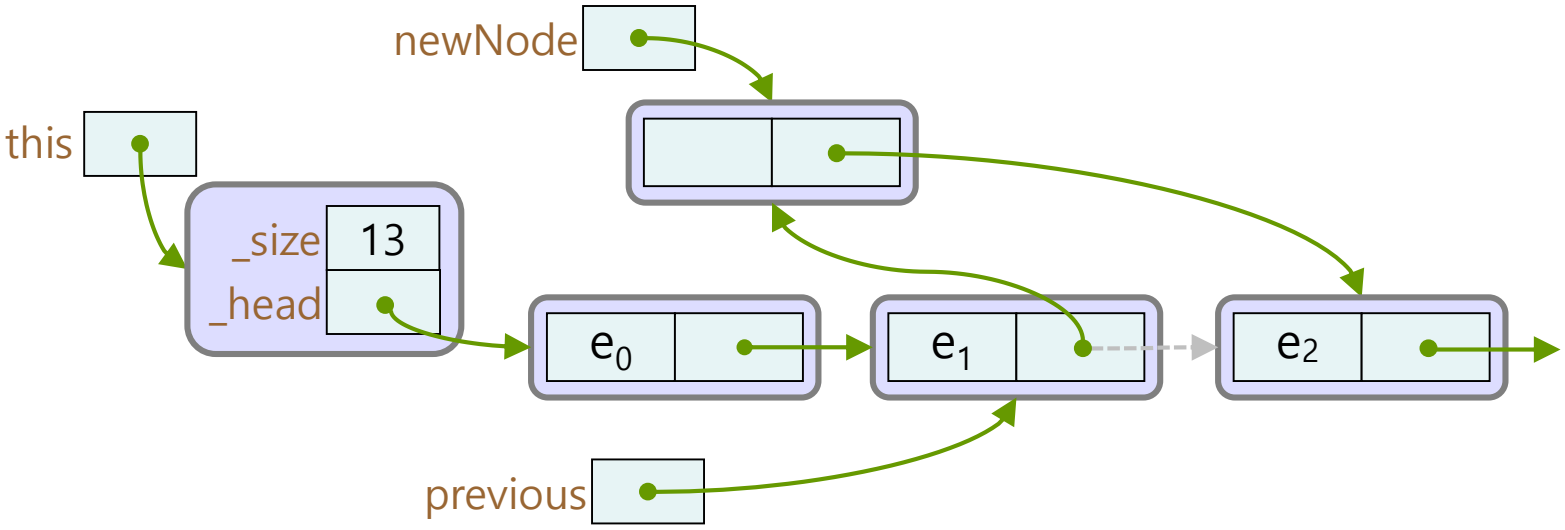
- 이전 실습에서 사용한 class 를 복사해 온다.
  - Coin.java
  - LinkedNode<E>.java
- 기존의 class 를 복사하여 이름을 변경한다.
  - 지난주의 ArrayList.java를 복사하여 다음의 것을 만든다.
    - UnsortedArrayList.java (거의 그대로 사용)
  - 지난주의 LinkedList.java를 복사하여 다음의 두 개를 만든다.
    - UnsortedLinkedList.java (거의 그대로 사용)
    - SortedLinkedList.java (수정하여 사용)
- 새로 만든 class 들의 수정할 부분을 찾아 수정한다.
  - Class 의 모든 method 를 철저하게 점검한다.
  - 각 method 의 구현 방법은 이전의 실습과 강의 슬라이드를 참고할 것.

삽입 : add()

- UnsortedArrayList:
  - 배열의 맨 끝에 삽입한다.
  - 시간복잡도: O(1)
- SortedArrayList:
  - 배열의 sort 순서에 맞는 위치를 찾아 삽입한다.
  - 시간복잡도: O(n)
- UnsortedLinkedList:
  - 연결리스트의 맨 앞에 삽입한다.
  - 시간복잡도: O(1)
- SortedLinkedList:
  - 연결리스트의 sort 순서에 맞는 위치를 찾아 삽입한다.
  - 시간복잡도: O(n)

SortedLinkedList: add() [1]

- LinkedList 의 sort 순서에 맞는 위치를 찾아 삽입한다.
  - 먼저, 순서에 맞는 위치의 앞 node 를 찾는다. (그림에서 "previous" node)
  - 새로 삽입할 노드 (그림에서 "newNode") 를 "previous" node 다음에 삽입한다.



SortedLinkedList: add() [2]

```
public boolean  add (E anElement) {
      // 전달 받은 anElement을 배열의 sort 순서에 맞는 위치를 찾아 삽입
      if ( this.isFull() ) {
            return  false ;
      }
       else {
            LinkedNode<E>  nodeForAdd = new LinkedNode<E>(anElement, null) ;
            if ( this.isEmpty() ) {
                  this.setHead(nodeForAdd) ;
            }
            else { // 리스트에는 적어도 하나의 노드가 있다.
                  LinkedNode<E>  current = this.head(); // 현재 비교하는 노드
                  LinkedNode<E>  previous = null // current 의 앞 노드. 삽입을 하려면, 앞 노드를 알아야 한다.
                  while ( current != null ) { // 리스트의 끝에 도달할 때 까지 비교 검색한다.
                        if ( current.element().compareTo(anElement) > 0 ) {
                              // 현재의 원소가 삽입할 anElement 보다 크면
                              break;  // 삽입할 위치를 찾은 것이므로 비교 검색 중지
                        }
                        previous = current ; // 아닐 경우 previous를  current 로 변경
                        current = current.next() ; // current 를 다음 노드로 이동
                  }
                  if ( previous == null ) { // anElement 가 가장 작다. 맨 앞에 삽입한다
                        nodeForAdd.setNext(this.head()) ;
                        this.setHead(nodeForAdd) ;
                  }
                  else {
                        nodeForAdd.setNext(current) ;
                        previous.setNext(nodeForAdd) ;
                  }
            }
            this.setSize(this.size()+1) ; // 크기를 하나 증가시킨다.
            return  true ;
      }
}
```

각각의 구현 별 리스트의 "max()" 의 구현

최대값 원소 찾기 : max()

- UnsortedArrayList:
  - 배열 전체를 조사하여 찾는다.
  - 시간복잡도: O(n)
- SortedArrayList:
  - 최대값 원소는 배열 맨 끝 원소이다.
  - 시간복잡도: O(1)
- UnsortedLinkedList:
  - 연결 체인 전체를 조사하여 찾는다.
  - 시간복잡도: O(n)
- SortedLinkedList:
  - 최대값 원소는 연결 체인의 맨 끝 노드에 있다.
  - 시간복잡도: O(n)

UnsortedArrayList: max ()

- UnsortedArrayList 에서, max()는 다음과 같이 구현할 수 있다.
  - class 선언에서 generic type <E> 가 비교 가능한 것임을 명시:
    - <E extends Comparable <E>> // <E> 객체는 다른 <E> 객체와 비교 가능
  - 원소를 비교할 때에는 E 의 "compareTo()" method 사용.

```
public class  UnsortedArrayList<E extends Comparable<E>> {
    ......
    public E max () {
        if ( this.isEmpty() ) {
            return  null ;
        }
        else {
            E  maxElement = this.elements()[0] ;
            for ( int  i = 1 ; i < this.size() ; i++ ) {
            if ( maxElement.compareTo (this.elements()[i]) < 0 ) {
                maxElement = this.elements()[i] ;
            }
            return  maxElement ;
        }

    }
}
```

기타 기능의 구현

기타 기능

- 지금까지의 강의/실습 내용을 참고하여 작성한다.

요약

확인하자

- 실험 준비와 측정
  - 난수를 이용한 데이터 생성 방법을 이해한다.
  - 시간 측정 방법을 이해한다.
- 리스트의 구현에 따른 성능 차이를 이해한다.
  - 데이터의 크기에 따른 시간의 변화를 이해한다.
  - 동일한 기능이 구현에 따라 어떻게 성능 차이가 나는지 이해한다.

생각해 볼 점

⇒ "리스트의 구현에 따른 성능 차이" 에 대해, 자신의 실험 결과를 분석하고, 그 결과를 각자의 의견과 함께 보고서에 작성하시오.

- <단계 1> 에서 데이터를 분석했던 예제를 참고할 것.

[실습 끝]