Javaプログラミング コレクション編

Quick, Nishio

目 次

第 1 章 List **1**
- 1.1 データの追加と取得 1
- 1.2 データ数の取得と削除 2
- 1.3 要素に順次アクセスする方法その 1 3
- 1.4 要素に順次アクセスする方法その 2 4
- 1.5 要素をソートする方法その 1 6
- 1.6 要素をソートする方法その 2 6
- 1.7 要素をソートする方法その 3 9
- 1.8 並列処理に対応したリスト 11
  - 1.8.1 リストの間違った利用例 12
  - 1.8.2 synchronizedList メソッドによるリストの同期 13
  - 1.8.3 スローダウンとデッドロック問題 16
  - 1.8.4 CopyOnWriteArrayList によるスローダウン回避 19
- 1.9 処理速度の検証 21
  - 1.9.1 ArrayList と LinkedList における追加，挿入の処理能力 . . . 21
  - 1.9.2 get と Iterator の処理能力 23
  - 1.9.3 CopyOnWriteArrayList の処理能力 26
  - 1.9.4 検証に使用したソースコード 27
- 1.10 演習問題 30

第 2 章 Set **33**
- 2.1 HashSet へのデータの追加 33
- 2.2 TreeSet へのデータの追加 34
- 2.3 contains メソッド 34
- 2.4 containsAll メソッド 35
- 2.5 retainAll メソッド 36
- 2.6 Set へのクラスの追加 37
- 2.7 Set へのクラスの追加・改良版 40
- 2.8 実践的サンプルプログラム 42
- 2.9 演習問題 44
- 2.10 演習問題解答例 44

第3章　Map **47**
3.1 HashMap へのデータの追加 47
3.2 すべての値を取り出す 1 48
3.3 すべての値を取り出す 2 48
3.4 １つのキーで複数の値を登録する 50
3.5 １つのキーで複数の値を登録する（クラスを使った方法） 51

第1章　List

この章では配列を扱うデータ構造 List について学んでいきます.

List には主に ArrayList と LinkedList が存在しています．どちらも配列を扱う List ですが，LinkedList の方が ArrayList に比べ挿入，削除が高速です．よって頻繁に挿入や削除を行うことが分かっている場合，LinkedList を使うべきです.

ArrayList と LinkedList の違いはパフォーマンスだけで，使い方はどちらも同じです.

1.1　データの追加と取得

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

public class ArrayList1 {
  public static void main(String[] args) {

    List<String> myList = new ArrayList<String>();
    myList.add("Hello ");
    myList.add("World");

    System.out.println(myList);

    String s = myList.get(1);
    System.out.println(s);
  }
}
```

実行結果

[Hello , World]
World

解説

```
List<String> myList = new ArrayList<String>();
```

この部分で String 型を格納することができる ArrayList ”myList”を生成しています．add 関数はデータを追加する際使用します.

データを取得する際は get 関数を使用します．`String s = myList.get(1);` を呼び出すことで，myList の 2 番目の要素を取り出しています．

1.2 データ数の取得と削除

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

public class ArrayList2 {

  private static final String[] colors =
          { "RED", "GREEN", "BLUE" , "BLACK" , "WHITE"};

  public static void main(String[] args) {

    List<String> myList = new ArrayList<String>();

    for(String color : colors)
      myList.add(color);

    System.out.println("myList size is " + myList.size());
    System.out.println("myList is " + myList);

    myList.remove(3);
    myList.remove("WHITE");

    System.out.println("myList is " + myList);
  }
}
```

実行結果

```
myList size is 5
myList is [RED, GREEN, BLUE, BLACK, WHITE]
myList is [RED, GREEN, BLUE]
```

解説

```
for(String color : colors)
    myList.add(color);
```

この for-each 文は

```
for(int i = 0; i < colors.length; i++)
    myList.add(colors[i]);
```

の省略形です．

size 関数は要素数を取得する際使用します．colors には 5 つの文字列が格納されているので，size 関数は 5 を返します．

remove は指定した要素を削除する関数です．

```
myList.remove(3);
```

で，4 番目の要素 BLACK を削除しています．インデクスを指定する以外に，

```
myList.remove("WHITE");
```

としても要素を消すことができます．

1.3　要素に順次アクセスする方法その 1

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

public class ArrayList_Iterator2 {
  public static void main(String[] args) {
    List<String> myList = new ArrayList<String>();

    myList.add("Bさん");
    myList.add("Aさん");
    myList.add("Cさん");

    //反復子の生成
    Iterator<String> i = myList.iterator();

    while(i.hasNext()){
      String name = i.next();
      System.out.println(name);
    }

  }
}
```

実行結果

```
B さん
A さん
C さん
```

解説

追加した要素にアクセスする方法として次のコードが考えられます．

```
String name =  myList.get(0);
```

このようにしても，myList の先頭要素である，”B さん ”という名前を取り出すことができます．しかし，データに順次アクセスしていく場合，get 関数を使用する方法は一般的ではありません．通常は，反復子 (iterator) を介してコレクションの要素にアクセスしていきます.

```
Iterator<String> i = myList.iterator(); //反復子を生成
```

ここで iterator のメソッドを紹介します.

```
boolean hasNext()
```

hasNext メソッドは次の要素が存在する場合 true を返し，そうでない場合は false を返す.

```
object next()
```

next メソッドは次の要素を返し，返したあと次の要素へと進む.

今回は，まず hasNext() メソッドで次の要素があるか確認し，ある場合は next() メソッドでその要素を取り出しています.

1.4 要素に順次アクセスする方法その 2

```
1  public class Member {
2
3    private String name; //Memberの名前
4    private int age; //Memberの年齢
5
6    public String getName() {
7      return name;
8    }
9    public void setName(String name) {
10     this.name = name;
11   }
12   public int getAge() {
13     return age;
14   }
15   public void setAge(int age) {
16     this.age = age;
17   }
18
19   public Member(String name, int age){
20     this.name = name;
21     this.age = age;
22   }
23 }
```

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

public class ArrayList_Iterator {

  public static void main(String[] args) {

    List<Member> member = new ArrayList<Member>();

    //Memberの登録
    member.add(new Member("Bさん", 21));
    member.add(new Member("Aさん", 20));
    member.add(new Member("Cさん", 22));

    //反復子を生成
    Iterator<Member> i = member.iterator();
    while(i.hasNext()){
                  // 要素を返し, 次の要素へと進む
      Member m = i.next();
      System.out.print("Name : " + m.getName());
      System.out.println(" Age : " + m.getAge());
    }
  }
}
```

実行結果

Name : B さん Age : 21
Name : A さん Age : 20
Name : C さん Age : 22

解説

Member クラスは name(名前) と age(年齢) の値を保持しています.

```
member.add(new Member("B さん", 21));
```

上記のコードで，member に　名前：B さん　年齢：21 歳という要素を追加しています．同様に A さん，B さんの情報も追加していきます.

追加した要素にアクセスする方法として次のコードが考えらます.

```
String name = member.get(0).getName();
```

しかし，通常は get 関数を使用せず，反復子を介してコレクションの要素にアクセスしていきます.

1.5　要素をソートする方法その 1

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

public class ArrayList_Sort {

  public static void main(String[] args) {
    List<String> myList = new ArrayList<String>();

    myList.add("Bさん");
    myList.add("Aさん");
    myList.add("Cさん");

    System.out.println("unsorted " + myList);
              //アルファベット順にsortする
    Collections.sort(myList);
    System.out.println("sorted " + myList);
  }
}
```

実行結果

unsorted [B さん, A さん, C さん]
sorted [A さん, B さん, C さん]

解説

上記のコードは，String 型の ArrayList をアルファベット順にソートするものです．ArrayList には sort メソッドは実装されていないので，java.util.Collections クラスを使用してソートを行います．このクラスには，コレクション関連のメソッドがいくつも定義されています．

```
Collections.sort(myList);
```

とすることで，アルファベット順にデータがソートされます．

1.6　要素をソートする方法その 2

§要素に順次アクセスする方法その 2 で登場したプログラムコードにおいて，

```
Collections.sort(member);
```

とすると次のようなエラーが発生してしまいます．

制約の不一致: 型 Collections の総称メソッド sort(List<T>) は引数 (List<Member>) に適用できません．推測される型 Member は，制約付きパラメーター<T extends Comparable<? super T>> の代替として有効ではありません．

このエラーコードから，Comparable という比較するためのインターフェースが実装されていなければならないということが読み取れます．

§要素をソートする方法その 1 のプログラムでは扱うデータ型が String でした．String クラスは Comparable インターフェースを内部で実装しているので，Collections.sort() メソッドを使って String のリストをソートすることが可能でした．

Comparable インターフェースのメソッドはただひとつ compareTo のみです．String は Comparable を implements して compareTo メソッドを実装していました．

つまり，Member クラスにも Comparable というインターフェースを継承させて compareTo メソッドを実装しなければなりません．

ここで String クラスに備わっている compareTo メソッドについて説明します．

```
public int compareTo(String anotherString)
```

この関数は 2 つの文字列を辞書式に比較します．

辞書式とはどのような順番でしょうか次のようなコードを書いて確認してみます．

```
1  public class ComparableTest {
2
3    public static void main(String[] args) {
4
5      String piyo = "BBB";
6
7      String hoge = "AAA";
8      String foo = "BBB";
9      String bar = "CCC";
10
11     int x = piyo.compareTo(hoge); //piyoとhogeの関係
12     int y = piyo.compareTo(foo); //piyoとfooの関係
13     int z = piyo.compareTo(bar); //piyoとbarの関係
14
15     System.out.println("piyo と hoge の関係は :" + x);
16     System.out.println("piyo と foo の関係は :" + y);
17     System.out.println("piyo と bar の関係は :" + z);
18   }
19 }
```

実行結果

piyo と hoge の関係は :1
piyo と foo の関係は :0
piyo と bar の関係は :-1

この実行結果から分かることは，compareTo メソッドは,

```
"BBB" < "AAA" なら -1
"BBB" == "BBB"   なら 0
"BBB" > "CCC" なら 1
```

を返すようです.

このことから，次の特性を備えた int 型を返すことが分かります.

- 自分の Object < 比較する Object ならば　負の値
- 自分の Object == 比較する Object ならば　0
- 自分の Object > 比較する Object ならば　正の値

sort メソッドは，リストの配列がどのようにソートされるべきかを compareTo メソッドを使って決定しています.

よって自作のクラス Member にも compareTo メソッドを作り，

- 自分 < 相手　なら負の値
- 自分 == 相手　なら 0
- 自分 > 相手　なら正の値

という処理を実装しなければなりません.

では，具体的なプログラムを見ていきましょう.

```
1  public class Member2 implements Comparable<Member2>{ // #1
2
3    private String name; //Memberの名前
4    private int age; //Memberの年齢
5
6    public String getName() {
7      return name;
8    }
9    public void setName(String name) {
10     this.name = name;
11   }
12   public int getAge() {
13     return age;
14   }
15   public void setAge(int age) {
16     this.age = age;
17   }
18
19   public Member2(String name, int age){
20     this.name = name;
21     this.age = age;
22   }
23
24   public int compareTo(Member2 o) { // #2
25     return  this.getName().compareTo(o.getName());
26   }
27
28 }
```

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

public class ArrayList_Sort2 {

  public static void main(String[] args) {

    List<Member2> member = new ArrayList<Member2>();

    //Memberの登録
    member.add(new Member2("Bさん", 21));
    member.add(new Member2("Aさん", 20));
    member.add(new Member2("Cさん", 22));

    //Member2で決めた方法でソートする
    Collections.sort(member);

    //反復子を生成
    Iterator<Member2> i = member.iterator();
    while(i.hasNext()){
              // 要素を返し，次の要素へと進む
      Member2 m = i.next();
      System.out.print("Name : " + m.getName());
      System.out.println(" Age : " + m.getAge());
    }

  }
}
```

実行結果

Name : A さん Age : 20
Name : B さん Age : 21
Name : C さん Age : 22

解説

Member2 クラスの 1 の行では，Member2 クラス同士を比較できるように Comparable インターフェースを実装することを宣言しています．そして，compareTo メソッドを実装することで要素の比較を行うことができます．

今回は，名前順にソートしたいので自分の名前と比較相手の名前を比べています．なお，String 型同士を比較しているので String の compareTo メソッドを呼び出しています．

1.7 要素をソートする方法その 3

comparable インターフェースを実装した場合は，ソート順序は 1 通りしか作成できません．ここでは，ソート条件を複数選択できるような実装を考えていきた

いと思います.

Collections に存在する sort メソッドを調べると，次の 2 つがあることに気づきます.

```
public static void sort(List list)
public static void sort(List list,  Comparator c)
```

前節までは前者の sort を使ってきましたが，ここでは後者の sort を使うこととします.

```
1  import java.util.ArrayList;
2  import java.util.Collections;
3  import java.util.Comparator;
4  import java.util.Iterator;
5  import java.util.List;
6
7  public class ArrayList_Sort3 {
8
9    public static void main(String[] args) {
10
11     List<Member2> member = new ArrayList<Member2>();
12
13     //Memberの登録
14     member.add(new Member2("Bさん", 21));
15     member.add(new Member2("Aさん", 20));
16     member.add(new Member2("Cさん", 22));
17
18                 //昇順にソート  #1
19     Collections.sort(member, new NameSort());
20                  //反復子を生成
21     Iterator<Member2> i = member.iterator();
22     while(i.hasNext()){
23                         // 要素を返し，次の要素へと進む
24       Member2 m = i.next();
25       System.out.print("Name : " + m.getName());
26       System.out.println(" Age : " + m.getAge());
27     }
28
29     System.out.println("----------------------");
30                  //降順にソート  #2
31     Collections.sort(member, new NameReverseSort());
32     i = member.iterator(); //イテレータを初期化する
33
34     while(i.hasNext()){
35                         // 要素を返し，次の要素へと進む
36       Member2 m = i.next();
37       System.out.print("Name : " + m.getName());
38       System.out.println(" Age : " + m.getAge());
39     }
40
41   }
42 }
43
44 //昇順にソートするクラス
45 class NameSort implements Comparator<Member2>{  //#3
46
47   public int compare(Member2 o1, Member2 o2) {
48     return  o1.getName().compareTo(o2.getName());
49   }
50 }
51
```

```
52  //降順にソートするクラス
53  class NameReverseSort implements Comparator<Member2>{
54
55    public int compare(Member2 o1, Member2 o2) {
56      return  o1.getName().compareTo(o2.getName()) * (-1); //#4
57    }
58  }
```

実行結果

```
Name : Aさん Age : 20
Name : Bさん Age : 21
Name : Cさん Age : 22
----------------------
Name : Cさん Age : 22
Name : Bさん Age : 21
Name : Aさん Age : 20
```

解説

sort メソッドの第 2 引数には，ソートの条件が定義されているクラスを渡しています.

Comparator インターフェースは compare というメソッド１つしか持っていません．使い方自体は Comparable インターフェースと非常によく似ています.

プログラム中の#1 では第 2 引数に，昇順にソートするクラスを渡しています．また#2 の行では逆に，降順にソートするクラスを渡しています.

#3 では昇順にソートするクラスを実装しています．#4 では降順にするように，compareTo メソッドが返す結果にマイナスを掛け，結果を反転させています.

このようにソートしたいインスタンスのクラスとは別のクラスを作成し，sort メソッドの第 2 引数に渡すクラスを変えることによって，多数のソート順序を作成することができます.

1.8 並列処理に対応したリスト

この節では，並列実装されたリストを紹介していきます．ArrayList や LinkedList は，複数のスレッドから同時にアクセス，変更された場合，エラーが発生します.

ここではスレッドセーフ（同時にアクセスされてもエラーの発生しないよう）なリストの作り方について解説していきます.

1.8.1 リストの間違った利用例

リストの間違った利用例を紹介します．次のコードは複数のスレッドから同時に ArrayList である list に読み込み，変更を行っています．

```
import java.util.ArrayList;
import java.util.Iterator;


public class Practice{

  class AccessTest implements Runnable {

    public void run() {

      try {
        for(;;) {
          Iterator<Integer> iter = list.iterator();
          while( iter.hasNext() ) {
            //list内のデータを取得
            int tmp = iter.next();
            //50ms停止
            Thread.sleep(50);
          }
          System.out.println("データ取得完了");
        }
      }catch(Exception e) {
        //エラー情報を表示
        System.out.println("Error @AccessTest");
        e.printStackTrace();
      }
    }
  }

  class UpdateTest implements Runnable {

    public void run() {
      try{
        for(;;) {
          for(int i=0; i < 100; i++) {
            list.add(i);
          }
          Iterator<Integer> iter = list.iterator();
          while( iter.hasNext() ) {
            //list内のデータを取得
            iter.next();
            //nextで取得したデータを削除
            iter.remove();
            //50ms停止
            Thread.sleep(50);
          }
          System.out.println("データ更新完了");
        }
      }catch(Exception e) {
        System.out.println("Error @UpdateTest");
        e.printStackTrace();
      }
    }
  }

  public Practice() {
    AccessTest access = new AccessTest();
    UpdateTest update = new UpdateTest();
    Thread accessThread = new Thread(access);
```

```
60      Thread updateThread = new Thread(update);
61
62      //AccessTestとUpdateTestの処理開始
63      System.out.println("アクセス開始");
64      accessThread.start();
65      updateThread.start();
66    }
67
68    public static void main(String args[]) {
69      new Practice();
70    }
71
72    private ArrayList<Integer> list = new ArrayList<Integer>();
73  }
```

実行結果

```
アクセス開始
データ取得完了
データ取得完了
データ取得完了
データ取得完了
データ取得完了
Error @AccessTest
java.util.ConcurrentModificationException
at java.util.AbstractList$
                Itr.checkForComodification(AbstractList.java:372)
at java.util.AbstractList$Itr.next(AbstractList.java:343)
at Practice$AccessTest.run(Practice.java:20)
at java.lang.Thread.run(Thread.java:619)
データ更新完了
データ更新完了
```

ConcurrentModificationException が発生し，AccessTest の処理は止まってしまいます．このエラーは複数個所から同時にリストの中身を書き換え，読み込んだ場合に発生します．

ArrayList は同期化されてはいません．同期化する場合，つまり複数スレッドから同時にアクセス，変更されてもエラーを発生させないためには，synchronizedList を利用します．次節でその使い方について説明します．

1.8.2 synchronizedList メソッドによるリストの同期

List を同期化する場合，Collections.synchronizedList メソッドを利用します．このメソッドは，ArrayList だけでなく，LinkedList でも同様に使用することができ

ます.

では，前節のプログラムを同期化してみましょう.

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


public class Practice{

  class AccessTest implements Runnable {

    public void run() {

      try {
        for(;;) {
          synchronized( list ) {
            Iterator<Integer> iter = list.iterator();
            while( iter.hasNext() ) {
              //list内のデータを取得
              int tmp = iter.next();
              //50ms停止
              Thread.sleep(50);
            }
            System.out.println("データ取得完了");
            //50ms停止
            Thread.sleep(50);
          }
        }
      }catch(Exception e) {
        //エラー情報を表示
        System.out.println("Error @AccessTest");
        e.printStackTrace();
      }
    }
  }

  class UpdateTest implements Runnable {

    public void run() {
      try{
        for(;;) {
          synchronized( list ) {
            for(int i=0; i < 100; i++) {
              list.add(i);
              //50ms停止
              Thread.sleep(50);
            }
            Iterator<Integer> iter = list.iterator();
            while( iter.hasNext() ) {
              //list内のデータを取得
              iter.next();
              //nextで取得したデータを削除
              iter.remove();
              //50ms停止
              Thread.sleep(50);
            }
          }
          System.out.println("データ更新完了");
        }
      }catch(Exception e) {
        System.out.println("Error @UpdateTest");
        e.printStackTrace();
      }
```

```
63      }
64    }
65
66    public Practice() {
67      AccessTest access = new AccessTest();
68      UpdateTest update = new UpdateTest();
69      Thread accessThread = new Thread(access);
70      Thread updateThread = new Thread(update);
71
72      //AccessTestとUpdateTestの処理開始
73      System.out.println("アクセス開始");
74      accessThread.start();
75      updateThread.start();
76    }
77
78    public static void main(String args[]) {
79      new Practice();
80    }
81
82    private List<Integer> list =
83      Collections.synchronizedList( new ArrayList<Integer>() );
84  }
```

実行結果

アクセス開始
データ取得完了
データ取得完了
データ更新完了
データ取得完了
データ更新完了
データ取得完了
データ取得完了
データ取得完了
データ更新完了
データ取得完了
・
・
・

今回は前回のようにエラーは発生しませんでした．

ArrayList は Collections.synchronizedList メソッドを用いて初期化することで，同期化することができます．また，Iterator を使って list にアクセスする際には，synchronized 修飾子を使い，list が他のスレッドからアクセスできないよう，ロックをかける必要があります．

今回は AccessTest クラスと UpdateTest クラスの run メソッドは，どちらも Iterator を使ってデータにアクセスしているので，両方とも synchronized 修飾子を使用しています．

synchronized 修飾子は，データにアクセスする際にロックを掛け，他のスレッドからアクセスできないようにするものです. また，synchronized ブロックから抜けた後，データをアンロック（つまり別のスレッドからアクセス可能な状態に）します.

また，既にロックされていたデータをロックすると，待ち状態が発生します (図 1.1).

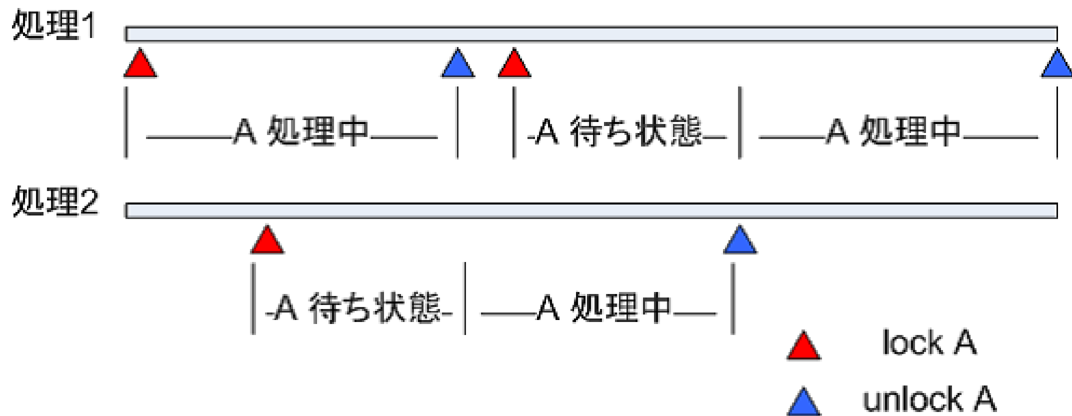

図 1.1: データのロックとアンロック

1.8.3 スローダウンとデッドロック問題

前節のコードの一部を以下のように変更してみました.

```
class AccessTest implements Runnable {

  public void run() {

  try {
     for(;;) {
        System.out.println("start @accessTest");
        //50ms停止
        Thread.sleep(50);
        synchronized( list ) {
          Iterator<Integer> iter = list.iterator();
          while( iter.hasNext() ) {
            //list内のデータを取得
            int tmp = iter.next();
            //50ms停止
            Thread.sleep(50);
          }
          System.out.println("データ取得完了");
        }
     }
    }catch(Exception e) {
      //エラー情報を表示
      System.out.println("Error @AccessTest");
      e.printStackTrace();
```

```
25          }
26      }
27  }
28
29  class UpdateTest implements Runnable {
30
31      public void run() {
32          try{
33              for(;;) {
34                  System.out.println("start @updateTest");
35                  for(int i=0; i < 100; i++) {
36                      list.add(i);
37                      //50ms停止
38                      Thread.sleep(50);
39                  }
40                  while( list.size() != 0 ) {
41                      list.remove(0);
42                      //50ms停止
43                      Thread.sleep(50);
44                  }
45                  System.out.println("データ更新完了");
46              }
47          }catch(Exception e) {
48              System.out.println("Error @UpdateTest");
49              e.printStackTrace();
50          }
51      }
52  }
```

大きく手を加えたのは UpdateTest クラスの run メソッド内です．データへのアクセスを行っていないので，synchronized 修飾子は必要ありません．さて，この場合どのような問題が発生するでしょうか．

実行結果

```
アクセス開始
start @accessTest
start @updateTest
データ取得完了
start @accessTest
データ取得完了
start @accessTest
データ取得完了
start @accessTest
データ取得完了
start @accessTest
データ取得完了
start @accessTest
データ取得完了
start @accessTest
データ取得完了
start @accessTest
・
・
・
```

私の使用している計算機では，2 分立っても UpdateTest 内の run メソッドは処理を完了しませんでした．

これは，AccessTest クラスの run メソッドが list をロックしているために発生しています．UpdateTest がデータを更新するタイミングは，AccessTest がデータをアンロックしてから，またロックするタイミングのわずかな間しかありません．これでは，いくら待ってもデータの更新は完了しません．

このように，synchronized の使い方を誤ると，システムの性能が大幅に低下してしまうスローダウンに陥ってしまいます．

また，最悪の場合，デッドロックが発生してしまうかもしれません．synchronized を使う際には，システム設計は十分注意しなければなりません．

さて，このような問題を解決するリストに，CopyOnWriteArrayList があります．使い方は次節で説明します．

1.8.4 CopyOnWriteArrayList によるスローダウン回避

CopyOnWriteArrayList はデータ配列のコピーを内部で作成することで，スレッド間の干渉を排除しています.

前節のプログラムを CopyOnWriteArrayList を使って書き直してみます．ちなみに，このクラスは ArrayList であり，LinkedList 版はありません.

```
1  import java.util.ArrayList;
2  import java.util.Collections;
3  import java.util.Iterator;
4  import java.util.List;
5  import java.util.concurrent.CopyOnWriteArrayList;
6
7
8  public class Practice{
9
10   class AccessTest implements Runnable {
11
12     public void run() {
13
14       try {
15         for(;;) {
16           System.out.println("start @accessTest");
17
18           Iterator<Integer> iter = list.iterator();
19           while( iter.hasNext() ) {
20             //list内のデータを取得
21             int tmp = iter.next();
22             //50ms停止
23             Thread.sleep(50);
24           }
25           System.out.println("データ取得完了");
26
27         }
28       }catch(Exception e) {
29         //エラー情報を表示
30         System.out.println("Error @AccessTest");
31         e.printStackTrace();
32       }
33     }
34   }
35
36   class UpdateTest implements Runnable {
37
38     public void run() {
39       try{
40         for(;;) {
41           System.out.println("start @updateTest");
42           for(int i=0; i < 100; i++) {
43             list.add(i);
44             //50ms停止
45             Thread.sleep(50);
46           }
47           while( list.size() != 0 ) {
48             list.remove(0);
49             //50ms停止
50             Thread.sleep(50);
51           }
52           System.out.println("データ更新完了");
53         }
54       }catch(Exception e) {
55         System.out.println("Error @UpdateTest");
56         e.printStackTrace();
```

```
57            }
58          }
59        }
60
61        public Practice() {
62          AccessTest access = new AccessTest();
63          UpdateTest update = new UpdateTest();
64          Thread accessThread = new Thread(access);
65          Thread updateThread = new Thread(update);
66
67          //AccessTestとUpdateTestの処理開始
68          System.out.println("アクセス開始");
69          accessThread.start();
70          updateThread.start();
71        }
72
73        public static void main(String args[]) {
74          new Practice();
75        }
76
77        private List<Integer> list =
78          new CopyOnWriteArrayList<Integer>();
79      }
```

実行結果

```
アクセス開始
start @accessTest
データ取得完了
start @accessTest
データ取得完了
start @accessTest
データ取得完了
start @updateTest
start @accessTest
データ取得完了
start @accessTest
・
・
データ更新完了
start @updateTest
データ取得完了
start @accessTest
・
・
```

今度は更新作業が前節のコードとは比べ物にならないほど早くに完了しました. また，CopyOnWriteArrayList を使用した場合，synchronized 修飾子は使用する

必要はありません．このようにデッドロックやスローダウンの心配をせずに使える所が CopyOnWriteArrayList の利点です．

ただし，いくつか問題もあります．まず，CopyOnWriteArrayList はデータの更新（追加や削除）が ArrayList や synchronizedList を使った場合に比べとても遅いです．どれくらい遅いかは，処理速度の検証の節で比較しています．

また，Iterator を使ってデータに手を加えることはできません．つまり，Iterator を介して add や remove 作業は行えません．もし行った場合には，UnsupportedOperationException が発生します．

1.9 処理速度の検証

1.9.1 ArrayList と LinkedList における追加，挿入の処理能力

ArrayList と LinkedList の処理能力の違いに関して調べてみたいと思います．ArrayList と LinkedList の違いは，LinkedList の方が ArrayList に比べ挿入，削除が高速であるという点です．今回はデータの追加，挿入の処理速度について検証してみます．

まずは ArrayList と LinkedList へのデータの追加について検証してみます．データの追加とは，リストの最後に要素を追加することを指しています．

追加するデータ数を変えて処理速度を計測していきます．なお，使用した計算機の性能は CPU athlon64 3800+(2.41GHz), RAM 1GByte，WindowsXP です．

実験結果は表 1.1 及び図 1.2 のようになりました．

表 1.1: データ追加における処理速度の違い

データ数	ArrayList[ms]	LinkedList[ms]
10000	0	0
50000	16	16
100000	16	31
500000	188	297
1000000	313	610

グラフの横軸はデータ数，縦軸は計算時間 (単位 ms) となっています．結果より，データの追加は LinkedList の方が若干ではあるが遅いようです．

次にデータの挿入速度について検証してみましょう．実験結果は表 1.2 および図 1.3 のようになりました．

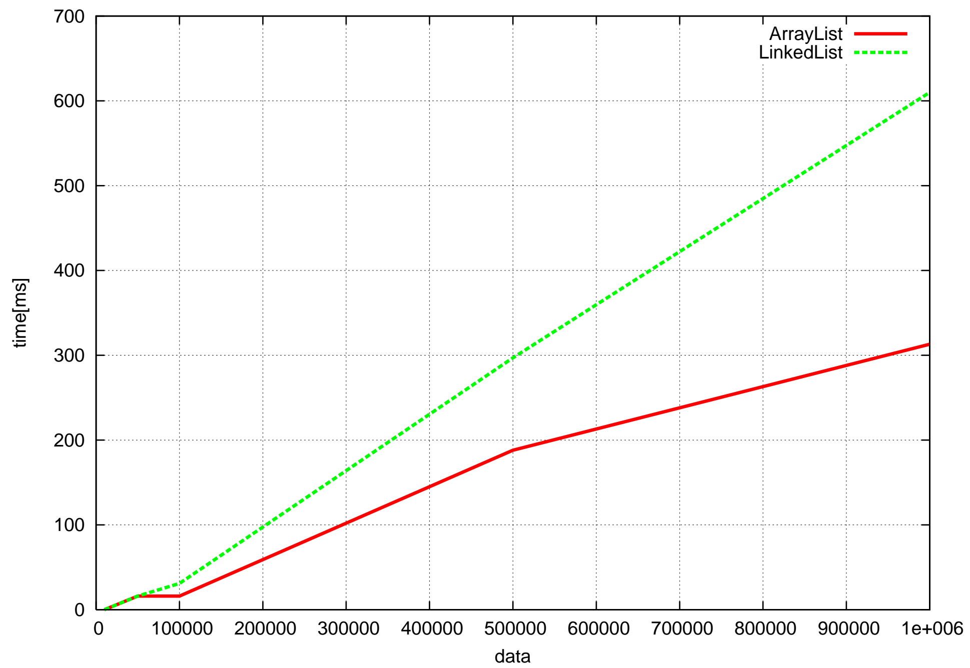

図 1.2: データの追加

表 1.2: データ挿入における処理速度の違い

データ数	ArrayList[ms]	LinkedList[ms]
1000	0	0
5000	16	0
10000	46	0
20000	187	15
30000	516	16
40000	953	16
50000	1500	16

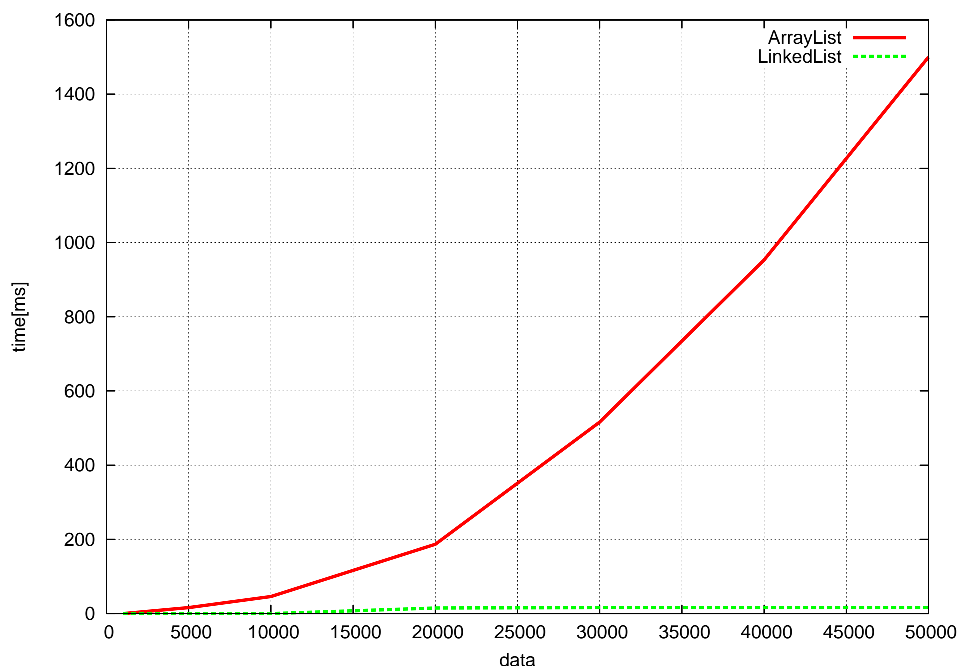

図 1.3: データの挿入

こちらは明らかに LinkedList の方が計算速度が速いことが読み取れます．データ数が 10000 個を過ぎたあたりから ArrayList は急激に処理速度が遅くなっています．50000 個のデータ挿入では，LinkedList に比べ，処理に 100 倍もの差がありました．

それに対し，LinkedList の計算時間はデータの追加とほぼ同じでした．

結論

データの追加だけを行うのなら ArrayList を，挿入，削除操作を行うのなら LinkedList を使うべきである．

1.9.2 get と Iterator の処理能力

データに順次アクセスをしていく場合は，get メソッドを使うよりも Iterator を介してアクセスするほうが一般的です．Iterator は指定したコレクションにあった最適なデータアクセス方法を提供するものです．今回は ArrayList と LinkedList に get または Iterator を使いデータに順次アクセスし，その処理速度を比較してみます．

まずは，ArrayListで検証してみます．データ数を変えて処理速度を計測していきます．なお，使用した計算機の性能はCPU athlon64 3800+(2.41GHz), RAM 1GByte, WindowsXPです．実験結果は表1.3及び図1.4となりました．

表1.3: ArrayListにおけるgetとiteratorの処理速度の違い

データ数	get[ms]	Iterator[ms]
10000	0	0
50000	0	0
100000	0	0
500000	15	31
1000000	31	47

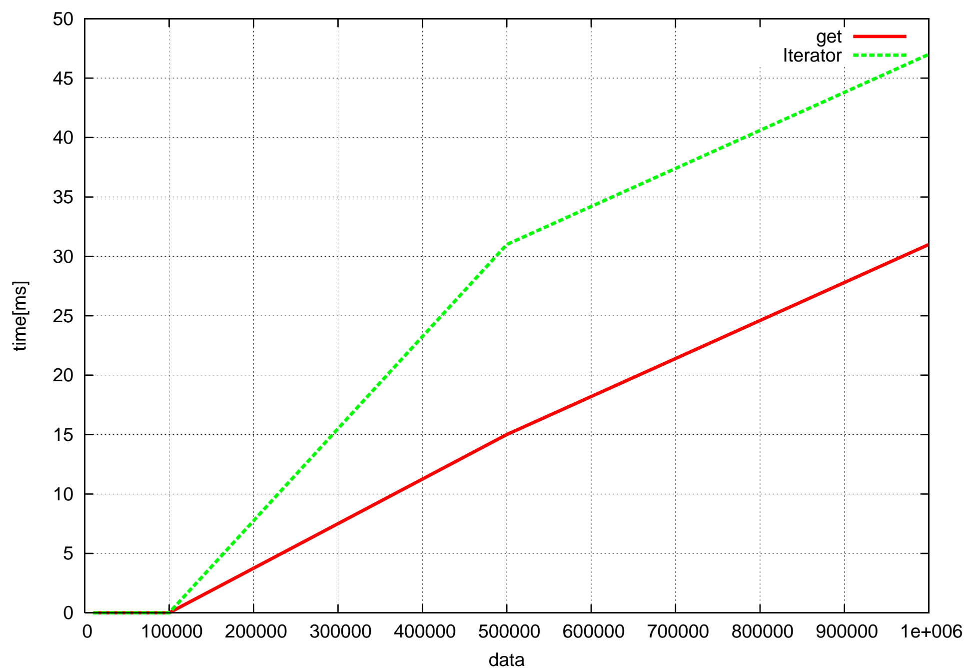

図1.4: ArrayListにおけるgetとiteratorの処理速度

グラフは横軸がデータ数，縦軸が処理速度[ms]となっています．若干ですが，getの方が早いように見えます．しかし，この程度の差は無視できる範囲でしょう．

次にLinkedListについて検証してみます．実験結果は表1.4及び図1.5となりました．

表 1.4: LinkedList における get と iterator の処理速度の違い

データ数	get[ms]	Iterator[ms]
1000	0	0
5000	16	0
10000	141	0
15000	328	0
20000	1484	0
30000	3718	0
40000	7313	0

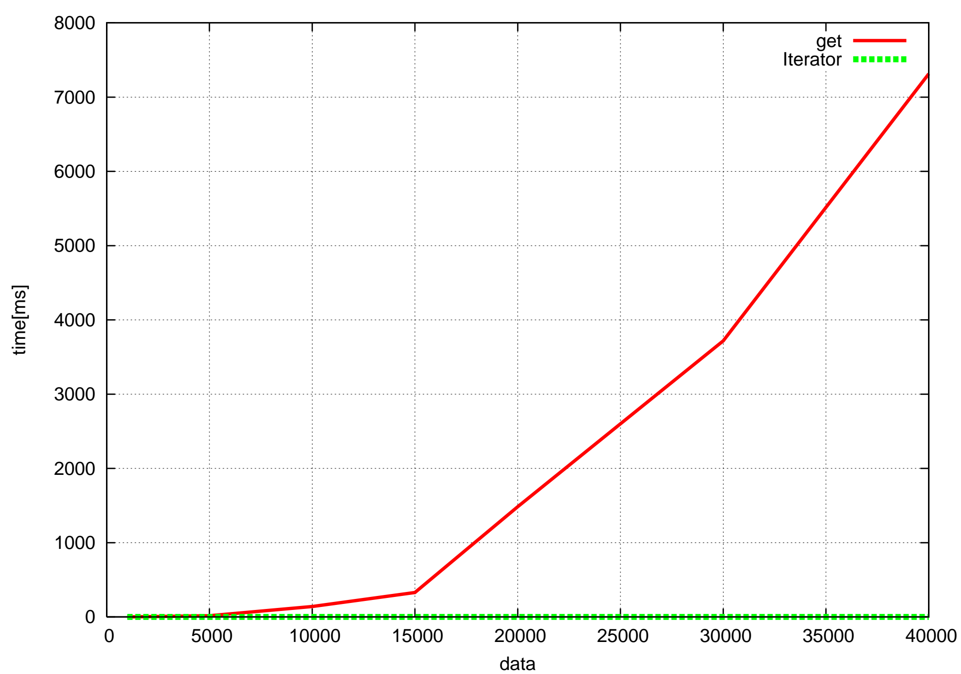

図 1.5: LinkedList における get と iterator の処理速度

こちらは明らかにIteratorを使用した方が早いことがわかります．データ数が増えれば増えるほど差が大きく開いていきます．

以上の実験結果よりgetを使用するよりもIteratorを使用してデータにアクセスした方がよいことが分かります．ArrayListはgetの方が処理速度が高速だからgetを使ったほうがいい，なんてことはありません．処理速度は百万個のデータがあっても10msほどしか変わりません．それよりも，もし使用しているデータ構造をArrayListからLinkedListに変更した場合，処理速度は大幅に低下してしまいます．

結論

データに順次アクセスする場合はIteratorを使うべきである．

1.9.3 CopyOnWriteArrayListの処理能力

CopyOnWriteArrayListのデータ更新速度をArrayList及びsynchronizedListと比較してみたいと思います．今回はデータの追加の処理速度について検証してみます．

データの追加とは，リストの最後に要素を追加することを指しています．追加するデータ数を変えて処理速度を計測していきます．なお，使用した計算機の性能はCPU athlon64 3800+(2.41GHz), RAM 1GByte，WindowsXPです．

実験結果は表1.5及び図1.6となりました．

ArrayList及びsynchronizedListと比較してCopyOnWriteArrayListは著しくデータ更新の処理能力が低いことが分かります．このため，大量のデータを処理する際にはCopyOnWriteArrayListを使用すべきではありません．

結論

大量のデータを扱う場合にはsynchronizedListを使うべきである．

表1.5: CopyOnWriteArrayListのデータ追加速度

データ数	ArrayList[ms]	SyncArrayList[ms]	CopyArrayList[ms]
1000	0	0	16
5000	0	0	109
10000	0	0	422
20000	0	0	1765
30000	15	16	4656

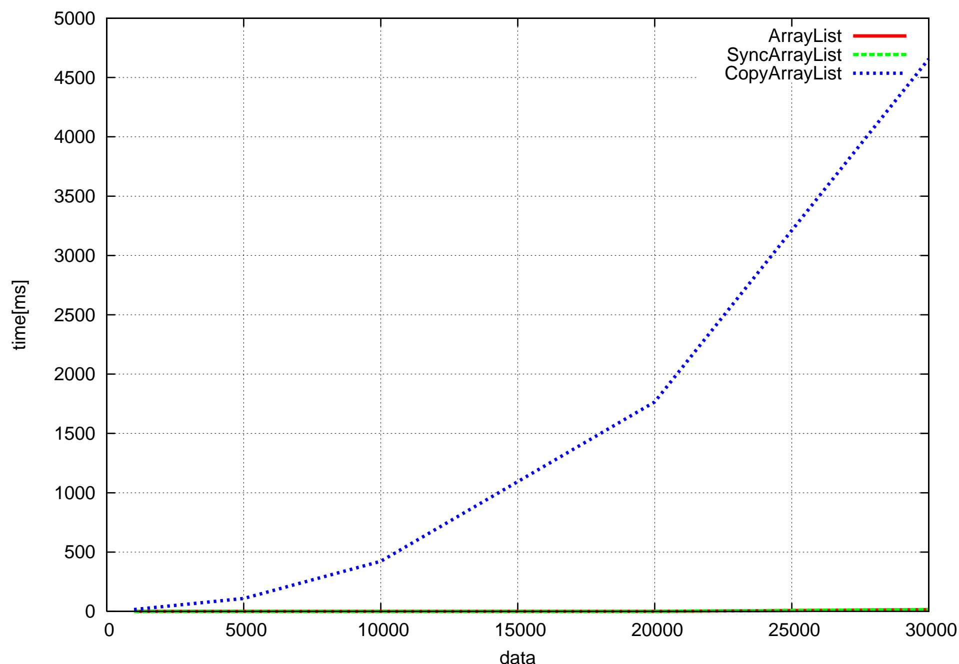

図 1.6: CopyOnWriteArrayList の処理速度

1.9.4 検証に使用したソースコード

ArrayList と LinkedList における追加，挿入の処理能力

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


class Practice{

  public void arrayListTest() {
    List<Integer> myList = new ArrayList<Integer>();
    long start, stop, diff;

    start = System.currentTimeMillis();
    for(int i=0; i < DATASIZE; i++ ) {
      myList.add(i);
    }
    stop = System.currentTimeMillis();
    diff = stop - start;
    System.out.println("実行時間 : " + diff + " ms");
  }

  public void linkedListTest() {
    List<Integer> myList = new LinkedList<Integer>();
    long start, stop, diff;

    start = System.currentTimeMillis();
    for(int i=0; i < DATASIZE; i++ ) {
      myList.add(i);
```

```
28      }
29      stop = System.currentTimeMillis();
30      diff = stop - start;
31      System.out.println("実行時間 : " + diff + " ms");
32    }
33
34    public void arrayListTest2() {
35      List<Integer> myList = new ArrayList<Integer>();
36      long start, stop, diff;
37
38      myList.add(0);
39
40      start = System.currentTimeMillis();
41      for(int i=0; i < DATASIZE; i++ ) {
42        myList.add(1,i);
43      }
44      stop = System.currentTimeMillis();
45      diff = stop - start;
46      System.out.println("実行時間 : " + diff + " ms");
47    }
48
49    public void linkedListTest2() {
50      List<Integer> myList = new LinkedList<Integer>();
51      long start, stop, diff;
52
53      myList.add(0);
54
55      start = System.currentTimeMillis();
56      for(int i=0; i < DATASIZE; i++ ) {
57        myList.add(1,i);
58      }
59      stop = System.currentTimeMillis();
60      diff = stop - start;
61      System.out.println("実行時間 : " + diff + " ms");
62    }
63
64    public static void main(String[] args) {
65      Practice p = new Practice();
66      //p.arrayListTest2();
67      p.linkedListTest2();
68      //p.arrayListTest();
69      //p.linkedListTest();
70    }
71    //データ数
72    private static final int DATASIZE = 50000;
73  }
```

get と Iterator の処理能力

```
1   import java.util.ArrayList;
2   import java.util.Iterator;
3   import java.util.LinkedList;
4   import java.util.List;
5
6   class Practice{
7
8     public void arrayListTest() {
9       List<Integer> myList = new ArrayList<Integer>();
10      long start, stop, diff;
11
12      for(int i=0; i < DATASIZE; i++ ) {
```

```
13        myList.add(i);
14      }
15
16      start = System.currentTimeMillis();
17      for(int i=0; i < DATASIZE; i++) {
18        int j = myList.get(i);
19      }
20      stop = System.currentTimeMillis();
21
22      diff = stop - start;
23      System.out.println("実行時間 : " + diff + " ms");
24    }
25
26    public void linkedListTest() {
27      List<Integer> myList = new LinkedList<Integer>();
28      long start, stop, diff;
29
30      for(int i=0; i < DATASIZE; i++ ) {
31        myList.add(i);
32      }
33
34      start = System.currentTimeMillis();
35      for(int i=0; i < DATASIZE; i++) {
36        int j = myList.get(i);
37      }
38      stop = System.currentTimeMillis();
39
40      diff = stop - start;
41      System.out.println("実行時間 : " + diff + " ms");
42    }
43
44    public void arrayListTest2() {
45      List<Integer> myList = new ArrayList<Integer>();
46      long start, stop, diff;
47
48      for(int i=0; i < DATASIZE; i++ ) {
49        myList.add(i);
50      }
51
52      Iterator<Integer> iter = myList.iterator();
53      start = System.currentTimeMillis();
54
55      while(iter.hasNext()){
56        int j = iter.next();
57      }
58      stop = System.currentTimeMillis();
59
60      diff = stop - start;
61      System.out.println("実行時間 : " + diff + " ms");
62    }
63
64    public void linkedListTest2() {
65      List<Integer> myList = new LinkedList<Integer>();
66      long start, stop, diff;
67
68      for(int i=0; i < DATASIZE; i++ ) {
69        myList.add(i);
70      }
71
72      Iterator<Integer> iter = myList.iterator();
73      start = System.currentTimeMillis();
74
75      while(iter.hasNext()){
76        int j = iter.next();
77      }
```

```
78      stop = System.currentTimeMillis();
79
80      diff = stop - start;
81      System.out.println("実行時間 : " + diff + " ms");
82    }
83
84    public static void main(String[] args) {
85      Practice p = new Practice();
86      //p.arrayListTest2();
87      p.linkedListTest2();
88      //p.arrayListTest();
89      //p.linkedListTest();
90    }
91
92    private static final int DATASIZE = 20000;
93  }
```

CopyOnWriteArrayList の処理能力

```
1   import java.util.ArrayList;
2   import java.util.Collections;
3   import java.util.List;
4   import java.util.concurrent.CopyOnWriteArrayList;
5
6   public class Practice{
7
8     public void addListTest(List<Integer> test) {
9       List<Integer> myList = test;
10      long start, stop, diff;
11
12      start = System.currentTimeMillis();
13      for(int i=0; i < DATASIZE; i++ ) {
14        myList.add(i);
15      }
16      stop = System.currentTimeMillis();
17      diff = stop - start;
18      System.out.println("実行時間 : " + diff + " ms");
19    }
20
21    public static void main(String[] args) {
22      Practice p = new Practice();
23      //p.addListTest(new ArrayList<Integer>() );
24      //p.addListTest(Collections.synchronizedList(new ArrayList<Integer>()));
25      p.addListTest(new CopyOnWriteArrayList<Integer>() );
26    }
27    //データ数
28    private static final int DATASIZE = 1000;
29  }
```

1.10 演習問題

問題 1

§要素をソートする方法その 2 の Member2 クラスのコードを改良し，年齢順にソートし結果を表示するプログラムを作成せよ．

問題 2

§要素をソートする方法その 3 の ArrayList_Sort3 クラスを改良し，年齢順に昇順，降順させ表示するプログラムを作成せよ．

第2章　Set

Setは要素の重複を認めない集合を扱います．この章では主にHashSetとTreeSetに関して説明していきます．HashSetとTreeSetの違いは，保持する要素の順序の違いです．HashSetは繰り返しの順序は保障しませんが，TreeSetは順序がソートされた要素を取り出すことができます．

2.1　HashSetへのデータの追加

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class Set_Test {

  public static void main(String[] args) {
    Set<String> set = new HashSet<String>();

    //文字列のsetを作成する
    set.add("Nuko");
    set.add("Quick");
    set.add("Nishio");
    set.add("Nuko");  //#1

    Iterator<String> ite = set.iterator();
    while(ite.hasNext())
      System.out.println(ite.next());
  }
}
```

実行結果

Nuko
Nishio
Quick

解説

setは重複要素を持つことはできません．そのため，ソースコード中の#1の部分では要素を追加しようとしたにもかかわらず挿入されていません．また，実行結

果をみてもわかるように，HashSet はデータを取り出す際，要素の順番が保障されていません.

2.2 TreeSet へのデータの追加

```
import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;


public class Set_Test2 {

  public static void main(String[] args) {
    Set<String> set = new TreeSet<String>();

    //文字列のsetを作成する
    set.add("Nuko");
    set.add("Quick");
    set.add("Nishio");
    set.add("Nuko");

    Iterator<String> ite = set.iterator();
    while(ite.hasNext())
      System.out.println(ite.next());
  }

}
```

実行結果

Nishio
Nuko
Quick

解説

TreeSet を使うことにより順番どおりに取り出せています.

2.3 contains メソッド

Set の中にあるメソッドについて解説していきます.

まずは contains メソッドです．このメソッドは引数で指定された要素が，セットに存在する場合 true を返すものです．具体例を見ていきましょう.

```
import java.util.HashSet;
import java.util.Set;


```

```
5  public class Set_Test3 {
6    public static void main(String[] args) {
7      Set<String> set = new HashSet<String>();
8
9      //要素の追加
10     set.add(new String("A"));
11     set.add(new String("B"));
12     set.add(new String("C"));
13
14     if(set.contains("A"));
15       System.out.println("A は含まれています. ");
16
17     if(set.contains("D"))
18       System.out.println("D は含まれています. ");
19     else
20       System.out.println("D は含まれていません. ");
21   }
22 }
```

実行結果

A は含まれています.
D は含まれていません.

解説

set に A,B,C の要素を追加し，contains メソッドを使い A が入っているか，D が入っているかを判定しています．contains メソッドは指定した要素が Set に含まれていれば true，そうでなければ false を返します．

2.4 containsAll メソッド

containsAll メソッドについて説明していきます．containsAll は引数で指定されたコレクションの要素すべてが，セットに存在する場合 true を返します．

具体例をみていきましょう．

```
1  import java.util.HashSet;
2  import java.util.Set;
3
4  public class Set_Test4 {
5    public static void main(String[] args) {
6      Set<String> set = new HashSet<String>();
7
8      //要素の追加
9      set.add(new String("A"));
10     set.add(new String("B"));
11     set.add(new String("C"));
12
13     Set<String> set2 = new HashSet<String>();
14     set2.add(new String("A"));
15     set2.add(new String("B"));
16
```

```
17	    Set<String> set3 = new HashSet<String>();
18	    set3.add(new String("C"));
19	    set3.add(new String("D"));
20	
21	    System.out.println("set :" + set);
22	    System.out.println("set2 :" + set2);
23	    System.out.println("set3 :" + set3);
24	
25	    if(set.containsAll(set2)) //#1
26	      System.out.println("setの要素の中にset2の要素がすべて含まれています");
27	
28	    if(set.containsAll(set3))  //#2
29	      System.out.println("setの要素の中にset3の要素がすべて含まれています");
30	    else
31	      System.out.println("setの要素の中にset3の要素がすべて含まれてはいません");
32	  }
33	}
```

実行結果

set :[A, B, C]
set2 :[A, B]
set3 :[D, C]
set の要素の中に set2 の要素がすべて含まれています
set の要素の中に set3 の要素がすべて含まれてはいません

解説

#1 では set [A,B,C] のなかに set2 の [A,B] がすべて含まれているので true を返します．一方#2 では set[A,B,C] のなかに Set3[D,C] の C は含まれていますが，D は含まれていないので false を返します.

2.5 retainAll メソッド

retainAll メソッドはセット内の要素のうち，指定されたコレクション内にある要素だけを保持します. つまり，セットから，指定されたコレクション内にない要素をすべて削除します．よって「共通部分」を取ることになります.

具体例を見ていきましょう.

```
1	import java.util.HashSet;
2	import java.util.Set;
3	
4	public class Set_Test5 {
5	  public static void main(String[] args) {
6	    Set<String> set = new HashSet<String>();
7	
8	    //要素の追加
9	    set.add(new String("A"));
10	    set.add(new String("B"));
```

```
11      set.add(new String("C"));
12
13      Set<String> set2 = new HashSet<String>();
14      set2.add(new String("A"));
15      set2.add(new String("B"));
16      set2.add(new String("D"));
17
18      Set<String> intersect = new HashSet<String>(set);  //#1
19      intersect.retainAll(set2);  //#2
20
21      System.out.println("set AND set2 :" + intersect);
22    }
23  }
```

実行結果

```
set AND set2 :[A, B]
```

解説

#1 の行で

```
Set<String> intersect = set;
```

としてはいけません．このようなコードを書き intersect を変更すると，set の内容まで書き変わってしまいます．#2 の行では intersect の内容，つまり [A,B,C] と set2 の内容の [A,B,D] の要素の共通部分をとり，共通でない部分，つまり intersect の要素にある C を削除します．

2.6 Set へのクラスの追加

```
1   public class Member3 {
2
3     private String name; //Memberの名前
4     private int age; //Memberの年齢
5
6     public String getName() {
7       return name;
8     }
9     public void setName(String name) {
10      this.name = name;
11    }
12    public int getAge() {
13      return age;
14    }
15    public void setAge(int age) {
16      this.age = age;
17    }
18
19    public Member3(String name, int age){
20      this.name = name;
21      this.age = age;
```

```
22    }
23
24    public Member3(String name){
25      this.name = name;
26      this.age = 0;
27    }
28
29    public String toString() {
30      String message = this.name + ":" + this.age;
31      return message;
32    }
33  }
```

```
1   import java.util.HashSet;
2   import java.util.Set;
3
4   public class Set_Test6 {
5     public static void main(String[] args) {
6       Set<Member3> member = new HashSet<Member3>();
7
8       //メンバーの追加
9       member.add(new Member3("Aさん",20));
10      member.add(new Member3("Bさん",21));
11      member.add(new Member3("Aさん",22)); //#1
12
13      //Aさん 20歳という要素の追加に失敗した場合には表示する
14      if(!member.add(new Member3("Aさん",20)))  //#2
15        System.out.println("Aさん 20歳の要素の追加に失敗");
16
17      System.out.println(member);
18    }
19  }
```

実行結果

[B さん:21, A さん:22, A さん:20, A さん:20]

解説

予想した結果とは大きく異なるかと思います．まず#1 の行では A さんという名前は同じですが，年齢が違うので要素の追加に成功しても問題ないように思えます．

しかし，#2 の行では　名前も年齢も同じものを追加したにもかかわらず要素の追加に失敗せずに追加されてしまいました．

set は同じ要素を追加できない構造でしたが，ここでは追加されてしまっています．ここで考えなければならない事は，何をもって同じ要素かということを判定できなければなりません．

さて今回は HashSet を使いました．ハッシュを使ったクラス (HashTable, HashMap, HashSet) ではハッシュ値を使いデータの格納，追加，削除をおこなっています．このハッシュ値を生成しているのは hashCode メソッドです．Java の Object クラスに書かれている hashCode の仕様を見ると，次のことが書いてあります．

hashCode メソッド

「オブジェクトのハッシュコード値を返します．このメソッドは，java.util.Hashtable によって提供されるようなハッシュテーブルで使用するために用意されています．equals(Object) メソッドで 2 つのオブジェクトが等価とされた場合，どちらのオブジェクトで hashCode メソッドを呼び出しても結果は同じ整数値にならなければならない. できる限り，Object クラスで定義される hashCode メソッドは，異なるオブジェクトについては異なる整数値を返します．通常，これはオブジェクトの内部アドレスを整数値に変換する形で実装されますが，そのような実装テクニックは JavaTM プログラミング言語では不要です．」

「Java2 プラットフォーム　Se v1.4.0 より引用」

HashCode メソッドで hash 値を作り出しそれをもとにデータの格納，追加，削除を行っていることが見えてきます．また，同じかどうか調べるときは equal メソッドも関係しているということが見えてくると思います.
equals の仕様もみていきましょう.

equals メソッド

「Object クラスの equals メソッドは，もっとも比較しやすいオブジェクトの同値関係を実装します．つまり，すべての参照値 x と y について，このメソッドは x と y が同じオブジェクトを参照する (x==y が true) 場合にだけ true を返します．通常，このメソッドをオーバーライドする場合は，hashCode メソッドを常にオーバーライドして,「等価なオブジェクトは等価なハッシュコードを保持する必要がある」という hashCode メソッドの汎用規約に従う必要があることに留意してください．」

「Java2 プラットフォーム　Se v1.4.0 より引用」

hashCode メソッドは，デフォルトでは異なる（参照先を指している）オブジェクトについては，異なる整数値を返します.
また，要素が同じかどうかの判定は hashCode メソッドで hash 値を計算し，hash 値が同じならば，equals メソッドが呼び出されます.
つまりは，hash 値だけで同じ要素かどうか簡易的な判定を行い，hash 値が同じなら equal メソッドが呼び出され，本当に同じかどうか確かめています.
以上より，ソースコード中の#2 において要素の追加が成功してしまったのは，クラス中の要素は同じかもしれないが，2 つのオブジェクトは異なるものなので，違う hash 値ができ，要素が追加ができてしまった，ということです.
よって，変更すべき点は以下の通りです.

1. デフォルトのハッシュ値計算を使わず，name と age からハッシュ値を求めるように変更する．
2. デフォルトの equal メソッドを使わず，name と age の値が 2 つのオブジェクト間で等しいかどうか判定するように変更する．

以上の 2 点で，追加する要素が重複しているかどうか判定できるようになります．次節では実際にコードを実装してみます．

2.7　Set へのクラスの追加・改良版

```
public class Member4 {

  private String name; //Memberの名前
  private int age; //Memberの年齢

  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public int getAge() {
    return age;
  }
  public void setAge(int age) {
    this.age = age;
  }

  public Member4(String name, int age){
    this.name = name;
    this.age = age;
  }

  public Member4(String name){
    this.name = name;
    this.age = 0;
  }

  public String toString() {
    String message = this.name + ":" + this.age;
    return message;
  }

  public int hashCode() {  //#1
    return this.name.hashCode() + this.age ;
  }

  public boolean equals(Object obj) { //#2
    //比較対象のオブジェクトの型が正しいかどうか確認
    if (obj instanceof Member4) {  //#3
      Member4 member = (Member4) obj;
      //名前と年齢が同じなら等しいとみなす
      if(this.name.equals(member.getName()) && this.age == member.getAge())  //#4
        return true;
      else
        return false;
    }else{
```

```
48              return false;
49          }
50      }
51  }
```

```
1   import java.util.HashSet;
2   import java.util.Set;
3
4
5   public class Set_Test7 {
6     public static void main(String[] args) {
7       Set<Member4> member = new HashSet<Member4>();
8
9       //メンバーの追加
10      member.add(new Member4("Aさん",20));
11      member.add(new Member4("Bさん",21));
12      member.add(new Member4("Aさん",22));
13
14      //Aさん　２０歳という要素の追加に失敗した場合には表示する
15      if(!member.add(new Member4("Aさん",20)))
16        System.out.println("Aさん　20歳の要素の追加に失敗");
17
18      System.out.println(member);
19    }
20  }
```

実行結果

A さん　20 歳の要素の追加に失敗
[A さん:20, B さん:21, A さん:22]

解説

今度はうまくいきました．A さん 20 歳という要素は重複して追加することができませんでした．

#1 で hashCode メソッドをオーバーライドし，名前と年齢から hash 値を求めています．今回は分かりやすいようにあえて Hash 値の計算をこのようにしましたが，実際は Eclipse の hashCode の自動生成時に作られるものを使うなど，もっと hash 値がより重ならないものになるようにしたほうがよいです．

#2 では equal メソッドをオーバーライドしています．ここでは Object クラスを引数として受け取っています．

#3 の行で受け取った Object が Member4 型かどうか判定しています．Member4 型なら true を，違う型なら false を返すような実装とします．このように，違う型が引数にとられてしまってもエラーとせず，false を返すようにすべきです．

#4 の行で自分と比較対象の名前と年齢が同じならば true を返すようにしています．

2.8 実践的サンプルプログラム

さて，実際にはSetはどのように使えるのか，次のコードを見て使い方を学んでいきましょう．今までの知識があれば十分理解できると思います．

```
public class Schedule {
  private String date; //日付
  private String subject; //題名

  //コンストラクタ
  public Schedule(String date){
    this.date = date;
    this.subject ="";
  }
  //コンストラクタ
  public Schedule(String date, String subject) {
    this.date = date;
    this.subject = subject;
  }

  //アクセサメソッド
  public String getDate() {
    return date;
  }
  public void setDate(String date) {
    this.date = date;
  }
  public String getSubject() {
    return subject;
  }
  public void setSubject(String subject) {
    this.subject = subject;
  }

  //toStringのオーバーライド
  public String toString() {
    String message = this.date + " の予定は " + this.subject + " です";
    return message;
  }

  //equalsのオーバーライド
  public boolean equals(Object obj) {
    //比較対象のオブジェクトの型が正しいかどうか確認
    if (obj instanceof Schedule) {
      Schedule schedule = (Schedule) obj;
      //日付同士を比較し同じなら等しいとみなす
      if(this.getDate().equals(schedule.getDate()))
        return true;
      else
        return false;
    }else{
      return false;
    }
  }

  //hashCodeのオーバーライド
  public int hashCode() {
    //日付をhash値とする
    return this.date.hashCode();
  }

}
```

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;


public class Set_final {

  public static void main(String[] args) {
    Set<Schedule> schedule = new HashSet<Schedule>();

    //setを初期化する (予定の登録)
    schedule.add(new Schedule("2008-08-05","合宿1日目"));
    schedule.add(new Schedule("2008-08-06","合宿2日目"));
    schedule.add(new Schedule("2008-08-07","合宿3日目"));
    schedule.add(new Schedule("2008-08-08","合宿4日目"));
    schedule.add(new Schedule("2008-08-10","オープン大会"));

    //反復子の生成
    Iterator<Schedule> ite = schedule.iterator();

    //登録されている内容をすべて表示(確認のため)
    while(ite.hasNext()){
      System.out.println(ite.next());
    }

    System.out.println("---------------");

    //特定の日付の情報だけもらうために作る
    Set<Schedule> findSchedule = new HashSet<Schedule>();
    findSchedule.add(new Schedule("2008-08-05"));
    findSchedule.add(new Schedule("2008-08-10"));
    findSchedule.add(new Schedule("2008-08-11"));

    // ----schedule AND findSchedule をする-----
    Set<Schedule> intersect = new HashSet<Schedule>(schedule);
    intersect.retainAll(findSchedule);

    System.out.println("検索結果 :" + intersect);

    //------重複値の追加のテストを行う---------
    Schedule addSchedule = new Schedule("2008-08-08","アルバイト");
    if(schedule.add(addSchedule)){
      System.out.println("無事追加されました");
    } else {
      System.out.println(addSchedule.getDate() + " の予定の追加に失敗しました" +
          " (理由)スケジュールの重複");
    }

    //追加に失敗したことを確認する
    System.out.println(schedule);
  }

}
```

実行結果

2008-08-10 の予定は オープン大会 です
2008-08-08 の予定は 合宿４日目 です
2008-08-07 の予定は 合宿３日目 です
2008-08-06 の予定は 合宿２日目 です
2008-08-05 の予定は 合宿１日目 です

―――――――

検索結果 :[2008-08-10 の予定は オープン大会 です, 2008-08-05 の予定は 合宿１日目 です]
2008-08-08 の予定の追加に失敗しました （理由）スケジュールの重複
[2008-08-10 の予定は オープン大会 です, 2008-08-08 の予定は 合宿４日目 です, 2008-08-07 の予定は 合宿３日目 です, 2008-08-06 の予定は 合宿２日目 です, 2008-08-05 の予定は 合宿１日目 です]

解説

前節の例では Member の名前と年齢が一致していれば同じ要素とみなしていましたが，今回は日付のみが一致していれば同一要素だと判定しています.

2.9 演習問題

問題 1

§実践的サンプルプログラムで扱ったコードを，HashSet から TreeSet に変えてうまく動作するように改良せよ.

2.10 演習問題解答例

```
public class Schedule2 implements Comparable<Schedule2> {

  private String date; //日付
  private String subject; //題名

  //コンストラクタ
  public Schedule2(String date){
    this.date = date;
    this.subject ="";
  }
  //コンストラクタ
  public Schedule2(String date, String subject) {
    this.date = date;
    this.subject = subject;
  }
```

```java

  //アクセサメソッド
  public String getDate() {
    return date;
  }
  public void setDate(String date) {
    this.date = date;
  }
  public String getSubject() {
    return subject;
  }
  public void setSubject(String subject) {
    this.subject = subject;
  }

  //toStringのオーバーライド
  public String toString() {
    String message = this.date + " の予定は " + this.subject + " です";
    return message;
  }


  public int compareTo(Schedule2 o) {
    return this.date.compareTo(o.getDate());
  }

}


--------------------------------------------------

import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;


public class Set_final2 {

  public static void main(String[] args) {
    Set<Schedule2> schedule = new TreeSet<Schedule2>();

    //setを初期化する (予定の登録)
    schedule.add(new Schedule2("2008-08-05","合宿1日目"));
    schedule.add(new Schedule2("2008-08-06","合宿2日目"));
    schedule.add(new Schedule2("2008-08-07","合宿3日目"));
    schedule.add(new Schedule2("2008-08-08","合宿4日目"));
    schedule.add(new Schedule2("2008-08-10","オープン大会"));

    //反復子の生成
    Iterator<Schedule2> ite = schedule.iterator();

    //登録されている内容をすべて表示(確認のため)
    while(ite.hasNext()){
      System.out.println(ite.next());
    }

    System.out.println("---------------");

    //特定の日付の情報だけもらうために作る
    Set<Schedule2> findSchedule = new TreeSet<Schedule2>();
    findSchedule.add(new Schedule2("2008-08-05"));
    findSchedule.add(new Schedule2("2008-08-10"));
    findSchedule.add(new Schedule2("2008-08-11"));

    // ----schedule AND findSchedule をする-----
```

```
81      Set<Schedule2> intersect = new TreeSet<Schedule2>(schedule);
82      intersect.retainAll(findSchedule);
83
84      System.out.println("検索結果 :" + intersect);
85
86      //------重複値の追加のテストを行う---------
87      Schedule2 addSchedule = new Schedule2("2008-08-08","アルバイト");
88      if(schedule.add(addSchedule)){
89        System.out.println("無事追加されました");
90      } else {
91        System.out.println(addSchedule.getDate() + " の予定の追加に失敗しました" +
92            " （理由）スケジュールの重複");
93      }
94
95      //追加に失敗したことを確認する
96      System.out.println(schedule);
97    }
98
99  }
```

第3章　Map

Map は「キー」と「値」がペアになった要素をもったオブジェクトです。Map ではキーが重複してはいけませんが、異なるキーで値が同じデータは登録することができます。

3.1　HashMap へのデータの追加

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


public class Map_Test {
  public static void main(String[] args) {
    //県名をキーにし、県庁所在地を値とする
    Map<String,String> prefectureInfo = new HashMap<String,String>();

    prefectureInfo.put("東京", "２３区"); //#1
    prefectureInfo.put("山梨", "甲府");
    prefectureInfo.put("埼玉", "さいたま");
    prefectureInfo.put("山梨", "甲府２"); //#3

    System.out.println(prefectureInfo);
    System.out.println("---------------");
    System.out.println("埼玉の県庁所在地は " + prefectureInfo.get("埼玉") + "です。");  //#2
  }
}
```

実行結果

埼玉=さいたま, 東京=２３区, 山梨=甲府２

———————

埼玉の県庁所在地は さいたま です。

解説

#1 で「東京」をキーに。２３区を値として Map に登録します。同様に山梨、埼玉も登録します。#2 でキー「埼玉」に関連付けられた値「さいたま」を呼び出しています。さて#3 では山梨をキーとしたものを２回登録しています。Set の時は値が重複するときは登録が拒否されました。しかし、map では 重複登録が拒否される

のではなく、上書きされることがわかります。上書きされる前のデータはHashMapから削除されるので注意しましょう。

3.2 すべての値を取り出す1

```
import java.util.HashMap;
import java.util.Map;
import java.util.Set;


public class Map_Test2 {
  public static void main(String[] args) {
                    // 県名をキーにし、県庁所在地を値とする
    Map<String,String> prefectureInfo = new HashMap<String,String>();

    prefectureInfo.put("東京", "２３区");
    prefectureInfo.put("山梨", "甲府");
    prefectureInfo.put("埼玉", "さいたま");
    prefectureInfo.put("山梨", "甲府");

    //キーを取得する
    Set<String> keys = prefectureInfo.keySet(); // #1

    // mapの中身をすべて表示する
    for(String key : keys) //#2
      System.out.println(key + " : " + prefectureInfo.get(key));

  }
}
```

実行結果

埼玉 : さいたま
東京 : ２３区
山梨 : 甲府

解説

値を１つのみ取り出すのは成功したのですが、すべての値を取得するときはこの様なプログラムになります。まず、#1 で　keySet メソッドを使いキーをすべて取り出します。このとき、keySet メソッドで返される値が　Set　で今回はキーには String 型を使ったので　Set¡String¿ という感じになります。キーをすべて取得できたなら //#2 でキーを一つずつ取り出し、get で値を取得していくということをしています。

3.3 すべての値を取り出す2

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.TreeSet;


public class Map_Test3 {
  public static void main(String[] args) {
    //番号をキー　名前を値とする
    Map<String,String> memberInfo = new HashMap<String,String>();

    memberInfo.put("003", "Cさん");
    memberInfo.put("005", "Eさん");
    memberInfo.put("001", "Aさん");
    memberInfo.put("002", "Bさん");

    //memberInfoのキーを取得する
    Set<String> keys = memberInfo.keySet(); //#1

    //キーをソートする
    Set<String> sortedKeys = new TreeSet<String>(keys); //#2

    //sortedKeysのイテレータを取得する
    Iterator<String> ite = sortedKeys.iterator(); //#3

    //値をすべて表示する
    while(ite.hasNext()){ //#4
      String key = (String)ite.next();
      String value = (String)memberInfo.get(key);
      System.out.println(key + " : " + value);
    }
  }
}
```

実行結果

001 : A さん
002 : B さん
003 : C さん
005 : E さん

解説

今度は foreach 文を使わず iterator を使い値をすべて取り出してみます。また、取り出す順番も考慮に入れています。#1 では先の例と同じように Map に登録されているキーをすべて取り出します。そして、#2 では取得したキーを TreeSet にいれ、順番をソートします。#3 で、ソートしたキーの iterator を取得し、#4 で値を表示しています。

この例において#2、#3 を取り除くと、値がソートしないまま表示されます。

3.4 １つのキーで複数の値を登録する

たとえば　Tokyo → Shinjuku, Hachioji, Minato　と、一つのキーから複数の値を取り出したいということは多々あると思います。しかし、現段階では標準に C++ STL のような multimap の機能はついていません。ここでは、独自に作成していきます。

```
1  import java.util.HashMap;
2  import java.util.HashSet;
3  import java.util.Iterator;
4  import java.util.Map;
5
6
7  public class Map_Test4 {
8    public static void main(String[] args) {
9
10     //県名をキーに区、市の名前を取り出す。
11     Map<String,HashSet<String>> city = new HashMap<String,HashSet<String>>(); //#1
12     HashSet<String> tokyo = new HashSet<String>(); //#2
13     HashSet<String> yamanashi = new HashSet<String>();
14
15     //東京都と山梨県の区、市を登録する
16     tokyo.add("Shinjuku"); //#3
17     tokyo.add("Hachioji");
18     tokyo.add("Minato");
19     yamanashi.add("Kohu");
20     yamanashi.add("Minamiarupusu");
21     yamanashi.add("Yamanashi");
22
23     //map(city)に件名をキーに区、市のリストを関連付ける
24     city.put("tokyo", tokyo); //#4
25     city.put("yamanashi", yamanashi);
26
27     System.out.println("-------- Tokyo --------");
28     //foreach文でまわしてtokyoに関連付けられた要素を取り出す
29     for(String town: city.get("tokyo")) //#5
30       System.out.println(town);
31
32     System.out.println("-------- Yamanashi --------");
33     //iteratorを使いyamanashiに関連付けられた要素を取り出す
34     Iterator ite = city.get("yamanashi").iterator(); //#6
35     while(ite.hasNext())
36       System.out.println(ite.next());
37
38   }
39 }
```

実行結果

——— Tokyo ———
Shinjuku
Hachioji
Minato
——— Yamanashi ———
Kohu
Minamiarupusu
Yamanashi

解説

さて、一つの値から複数の値を取得するために、Map の構成をキーを String 型にし、値を HashSet にしました。(#1) このプログラムでは県の名前をキーに、町の名前を値にし、県名を指定すると、町の名前が格納された HashSet を取得するということをしています。#2 では個々の県の町の名前を格納する HashSet を作り、#3 で登録をおこなっています。#4 においてキーを「tokyo」に、値を東京の町の名前が登録された HashSet とし、map に登録をおこなっています。そして、#5 では foreach 文を使い「tokyo」に関連付けられた値を取り出し、#6 では iterator を使い「yamanashi」に関連付けられた要素を取り出しています。このようにすることで１つのキーで複数の値を登録、取出しが可能になります。

3.5 １つのキーで複数の値を登録する（クラスを使った方法）

最後に 3.4 をクラスを使った方法にしてみます。

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;

public class Prefecture {
  //件名をキーに区、市の名前を取り出す。
  private Map<String,HashSet<String>> prefecture = new HashMap<String,HashSet<String>>();
  //区市町村の名前を格納する
  HashSet<String> city = null;

  public void add(String prefectureName, String cityName){

    //すでにある県名ならその中に入っているcity名を取り出す
    if(prefecture.containsKey(prefectureName)) //#1
      city = prefecture.get(prefectureName); //#2
    else
      city = new HashSet<String>(); //#3
```

```
18
19      city.add(cityName); //区市町村の名前を追加する //#4
20      prefecture.put(prefectureName, city);//mapを更新する
21    }
22
23    //県名を引数にして区、市の名前をArrayList<String>で返す
24    public HashSet<String> getCityName(String prefectureName){
25      HashSet<String> city = new HashSet<String>();
26      //県名をキーにしてMap prefecture に関連付けられた値（区、市）を返す
27      city = prefecture.get(prefectureName);  //#5
28      return city;
29    }
30  }
31
32  ---------------------------------------
33
34  import java.util.HashSet;
35
36  public class Map_Test5 {
37
38    public static void main(String[] args) {
39      Prefecture prefecture = new Prefecture();
40      HashSet<String> city = new ArrayList<String>();
41
42      //県名をキーに市名を登録する
43      prefecture.add("Tokyo", "Shinjuku");
44      prefecture.add("Tokyo", "Hachioji");
45      prefecture.add("Yamanashi", "Kohu");
46
47      //東京（キー）に関連付けられた市名（値）を取り出す
48      city = prefecture.getCityName("Tokyo");
49      System.out.println(city);
50    }
51  }
```

実行結果

解説

今度はずいぶんと main 文がシンプルになりました。Prefecture の add メソッドの #1 に注目してみましょう。まず、main 文においてはじめて prefecture.add(”Tokyo”, ”Shinjuku”); を行ったときは、#3 の else 文が実行されます。つづいて main 文において prefecture.add(”Tokyo”, ”Hachioji”); を実行すると、今度は#2 が実行されます。ここで#1 はすでに prefecture という map に tokyo（県名）というキーが存在していれば true を存在していなければ false を返しています。このようにすることで、tokyo（県名）というキーが存在していれば、今入っている値 (Shinjuku) を取り出し、#4 でさらに値 (Hachioji) を追加します。最後に map を更新し、add の作業は終了となります。

prefecture の getCityName メソッドでは prefectureName（県名）をキーにして Map prefecture に関連付けられた値（区、市）を返しています。これで完成となります。