List

リストは各要素がコレクション内で明示的な位置を持っているコレクションです。List 内の各オブジェクトの位置は、オブジェクトの追加順序によって決定できます。オブジェクトをリストに編成しているコレクションクラスは **List インタフェース** を実装しています。

ArrayList

ArrayList クラスは最も広く使われているList クラスです。
ArrayList クラスの操作の大多数は高速であり、操作に要する時間がリスト内のオブジェクトの数に左右されません。
このような高速な操作としては、リストの末尾へのオブジェクトの追加とリストの末尾からのオブジェクトの削除、リストのサイズの取得、リスト内の指定位置にあるオブジェクトの取得、リスト内の指定位置へのオブジェクトの設定が挙げられます。
それ以外の操作では、操作に要する時間がリスト内のオブジェクトの数に比例します。

ArrayList サンプル

```
import java.util.*;

class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("AAA");
        list.add("BBB");
        list.add("CCC");

        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }
    }
}
```

拡張forループを使って全ての要素を参照する場合は、下記のように記述します。

```
for (String s : list) {
    System.out.println(s);
}
```

ArrayList は配列を扱う一般的なクラスで、下記のようなメソッドがあります。

- list.add(o)　　- オブジェクト o を配列の末尾に追加する
- list.add(n, o)　　- オブジェクトを n で指定した場所に追加する
- list.get(n)　　- n番目の要素を得る
- list.remove(n)　　- n番目の要素を削除する
- list.set(n, o)　　- n番目の要素をオブジェクト o で置き換える
- list.size()　　- 要素の個数を得る
- list.indexOf(o)　　- オブジェクト o と等しい要素のインデックスを探す
- list.contains(o)　　- オブジェクト o と等しい要素があるか調べる

LinkedList

LinkedList クラスの大多数の操作は、ArrayList クラスのものに比べて長い時間を要します。ただし、LinkedList クラスでは、リストの途中にオブジェクトを追加したり、リストの途中からオブジェクトを削除したり、といった挿入や削除する操作は一定の時間で高速に行われます。

これらの操作を多用するのでない限り、一般にはArrayListを選ぶほうが有利でしょう。

LinkedList クラスはQueueやDeque、スタックを編成するのにも便利です。

LinkedList サンプル

```
import java.util.*;

class LinkedListTest {
   public static void main(String[] args) {
      LinkedList<String> list = new LinkedList<>();
      list.add("AAA");
      list.add("BBB");
```

```
        list.add("CCC");

        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }
    }
}
```

拡張forループを使って全ての要素を参照する場合は、下記のように記述します。

```
        for (String s : list) {
            System.out.println(s);
        }
```

挿入や削除を頻繁に行う場合はLinkedList の方が高速ですが、get() による参照などは ArrayList の方が高速です。

Queue

　コレクションをキューとして編成するクラスは **Queue インタフェース** を実装しています。
キューはその中に含まれるオブジェクトに順序を課します。キューには順序の末尾にオブジェクトを追加する操作と、順序の先頭からオブジェクトを削除する操作が備わっています。
キューではオブジェクトが追加された順序をキュー内容の順序付けに使用するのが最も一般的です。
オブジェクトの追加順序を使用するキューは、先入れ先だし構造と呼ばれることもあります。

LinkedList の利用

　LinkedList クラスを使用すると、オブジェクトをキューに編成して、オブジェクトが追加されたときの順序を維持できます。add メソッドと remove メソッドの実行に要する時間は短く、この時間はキュー内のオブジェクトの数に左右されません。

Set

セットは重複のないコレクションです。オブジェクトを追加しようとしたセットに既にそのオブジェクトが含まれていると何も起こりません。オブジェクトをセットに編成しているコレクションクラスは **Set インタフェース** を実装しています。

Set インタフェースを実装したクラスは、要素を特定の順序に保つ必要がありません。あるコレクションオブジェクトのクラスで Set インタフェースが実装されているという以外に、そのオブジェクトについて何もわからなければ、そのコレクションのイテレータでコレクションの内容がどんな順序で反復処理されるのか予測することはできません。

HashSet

HashSet はオブジェクトのセットの管理に最もよく使われるクラスです。
HashSet に対するオブジェクトの追加、削除およびHashSet 内のオブジェクトの検索は高速で行えます。HashSet のパラメータが適切に調整されていれば、これらの操作の所要時間はセット内のオブジェクトの数に左右されません。
HashSet クラスはセット内のオブジェクトを特定の順序に保ちません。

HashSet サンプル

```
import java.util.*;

class HashSetTest {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<>();
        set.add("AAA");
        set.add("BBB");
        set.add("CCC");
        set.add("AAA");

        Iterator it = set.iterator();
        while (it.hasNext()) {
```

```
            System.out.println(it.next());
        }
    }
}
```

拡張forループを使って全ての要素を参照する場合は、下記のように記述します。

```
        for (String s : set) {
            System.out.println(s);
        }
```

上記ソースでは「AAA」を2回 add() していますが、重複要素がひとつに統合されます。また結果の順序は add() した順序に関係なくバラバラになります。

HashSetには下記のようなメソッドがあります。

- set.add(o)　　- オブジェクト o を配列の末尾に追加する
- set.clear()　　- 配列をクリアする
- set.contains(o)　- オブジェクト o と等しい要素があるか調べる
- set.isEmpty()　　- 空かどうか調べる
- set.remove(o)　- オブジェクト o にマッチする要素を削除する
- set.size()　　- 要素の個数を得る

TreeSet

TreeSet クラスはセットの要素を指定の順序でソートされた状態に保ちます。
しかし、要素の追加/削除または要素の検索に要する時間は、HashSetで管理されるセットに比べて一般に長くなります。
これらの操作の所要時間はセット内の要素の数の対数に比例します。

TreeSet サンプル

```
import java.util.*;

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

```
        TreeSet<String> set = new TreeSet<>();
        set.add("CCC");
        set.add("AAA");
        set.add("BBB");
        set.add("AAA");

        Iterator it = set.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

拡張forループを使って全ての要素を参照する場合は、下記のように記述します。

```
        for (String s : set) {
            System.out.println(s);
        }
```

TreeSet も HashSet と同じように使用できます。

要素が自動的にソートされる点が HashSet と異なります。

Map

マップは「キーと値」が対になった要素を持ちます。キーの重複は許可されず、各キーは1つの値のみに対応付けられます。
キーから値を参照するデータ構造を持ったデータの利用に適しています。
Map インタフェース を実装するクラスとしては、HashMap、TreeMapなどが定義されています。

HashMap

Map インタフェースを実装した基本となるクラスです。キーの並び順を保持しないという特性をもっています。Map インタフェースを実装したクラス内では最も高速に動作します。
HashMap クラスは HashTable クラスの後継として定義されたものです。
2つのクラスの違いは、要素の操作を行う際に HashTable クラスが同期化されているのに対して、HashMapクラスが同期化されていない点です。
HashMap クラスは同期化されていない分、高速に動作します。
同期化された処理をHashMap クラスで行う場合は、HashMap クラスに同期処理を実装します。

HashMap サンプル

```
import java.util.*;

class HashSetTest {
    public static void main(String[] args) {
        HashMap<String, String> map = new HashMap<String, String>();
        map.put("Name", “北川”);
        map.put("Age", “33”);

        System.out.println("Name = " + map.get("Name"));
        System.out.println("Age = " + map.get("Age"));
    }
```

```
}
```

すべての値を参照するには下記のように記述します。

```
        Iterator it = map.keySet().iterator();
        while (it.hasNext()) {
            Object o = it.next();
            System.out.println(o + “ =” + map.get(o));
        }
```

拡張forループを使って全てのエントリを参照する場合は、下記のように記述します。

```
        for (Map.Entry<String,String> entry : map.entrySet()) {
            System.out.println(entry.getKey() + " = " + entry.getValue());
        }
```

TreeMap

キーが自然順序、もしくはコンストラクタに指定された Comparator に従って昇順にソートされた要素を持ちます。

TreeMap サンプル

```
mport java.util.*;

class HashSetTest {
    public static void main(String[] args) {
        TreeMap<String, String> map = new TreeMap<String, String>();
        map.put("Name", “北川”);
        map.put("Age", “33”);

        Iterator it = map.keySet().iterator();
        while (it.hasNext()) {
            Object o = it.next();
            System.out.println(o + “ =” + map.get(o));
        }
```

```
    }
}
```

拡張forループを使って全てのエントリを参照する場合は、下記のように記述します。

```
        for (Map.Entry<String,String> entry : map.entrySet()) {
            System.out.println(entry.getKey() + " = " + entry.getValue());
        }
```

TreeMap も HashMap と同じように使用できます。

要素が自動的にソートされる点が HashMap と異なります。