OPEN STREAM

# 213-02. コレクション

Make IT your Success

# 目次

# この講義について

**目的：**

配列とコレクションフレームワークの違いについて学び、コレクションフレームワークの使用方法、利点について学ぶ。

**ゴール：**

主なコレクションフレームワーク(List,Set,Map)や、拡張for文を使用してプログラムが組める。

オートボクシングを知り、適切に利用できる。

キャストについて理解し、利用できる。

# コレクションフレームワーク

同じ種類（データ型）の複数のデータを扱う場合、

配列を使用すると作成時に要素数を指定しなければならなかった。

コレクションフレームワークは要素の追加・削除など、要素総数含め、要素全体に対する操作などを簡単に扱える仕組み。

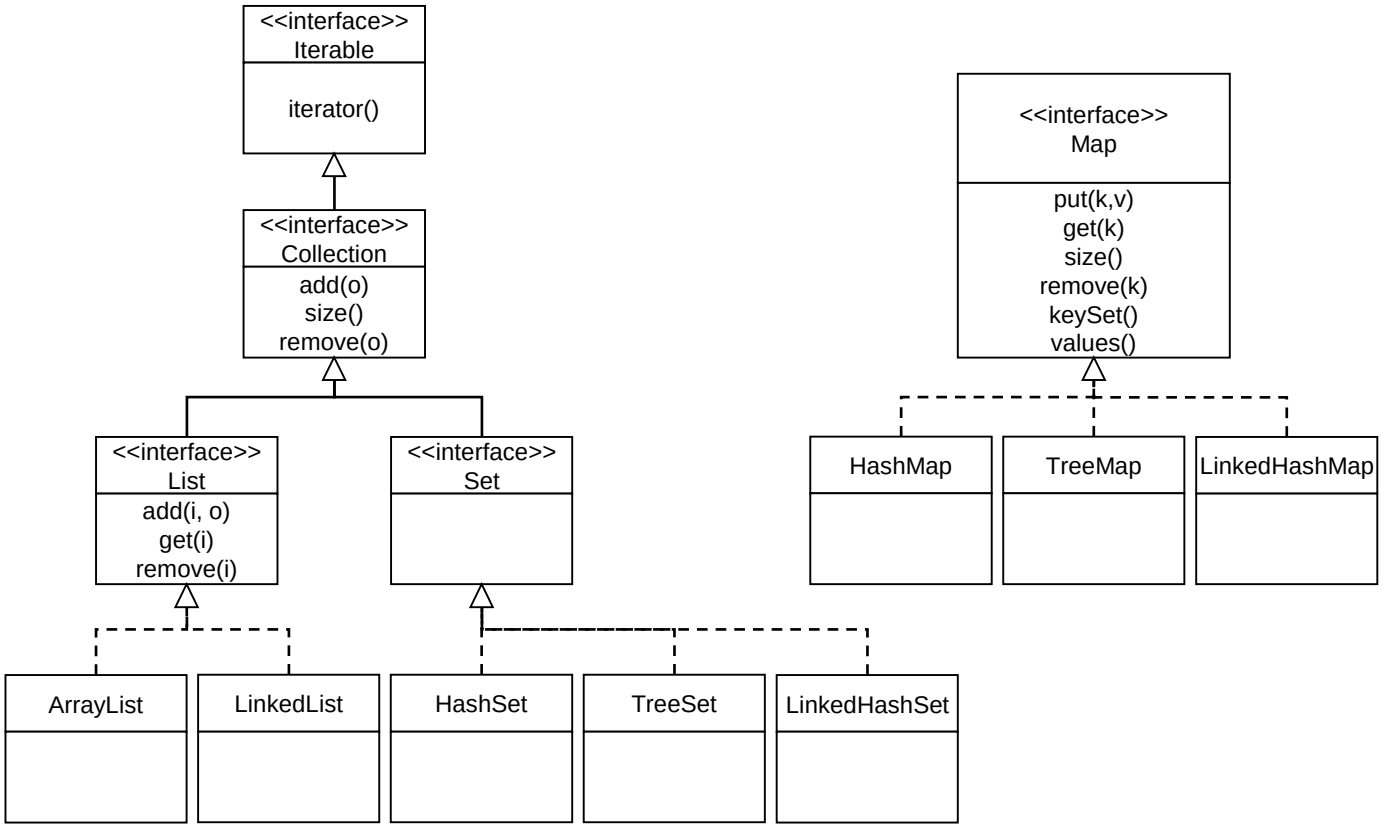
JavaがコレクションフレームワークのAPI（クラス・インタフェース群）を提供している。

コレクションフレームワークAPIもクラス(インタフェース)も、参照型のデータになる。

主なコレクションフレームワークのリストとセット、マップを解説する。

# コレクションフレームワークのクラス図

## クラスの継承図



# コレクションインタフェース(Collection)

Collectionとは

- Listインターフェース、Setインターフェースのスーパーインターフェース
- 複数の要素を管理、操作するための機能をまとめたインターフェース

主なメソッド

- add(o),size(),remove(o),iterator()

よく利用する代表的なコレクションのリストとマップを解説します。

# リストインタフェース(List)

## リスト

配列に似たコレクション。
配列同様に添字を使って各要素にアクセス出来る。
要素の追加・削除が可能。（配列では無理）
配列同様に要素は順序付けられている。
複数のnull値を許可。
Collectionインターフェースを実装。

## 代表的なリストインタフェースの実装：

- ArrayList　要素の参照は高速。 要素の追加・削除は低速。
- LinkedList　要素の参照は低速。 要素の追加・削除は高速。

※常にArrayListを使っていれば良いわけではない。
※配列がそうであったように要素値に同じもの（重複）があっても良い。
※要素値に重複を許可したくないときはセットというコレクションを使う。

# リストインタフェース(List)

主なメソッド

- add(o) …… 要素を追加する
- add(i, o) …… 指定された位置(index)に要素を追加する
- get(i) …… 指定された位置(index)の要素を返す
- remove(i) …… 指定された位置(index)の要素を削除する
- remove(o) …… 引数の要素が存在する時、その要素を削除する
- size() …… 要素の数を返す
- iterator() …… イテレータを返す

# セットインタフェース(Set)

## セット

要素の追加・削除が可能。

重複した参照は保持できない

要素は順序付けられていない。

Collectionインターフェースを実装。

代表的なセットインタフェースの実装：

HashSet　データ構造はハッシュテーブル。

TreeSet　データ構造は平衡木。

# セットインタフェース(Set)

主なメソッド

- add(o) …… 要素がSet内に無い場合、要素を追加する
- addAll(c) …… 引数のコレクション内の全ての要素について、要素がSet内に無い場合、要素を追加する
- remove(o) …… 要素がSet内にある場合、要素を削除する
- removeAll(c) …… 引数のコレクション内の全ての要素を削除する
- contains(o) …… 要素が存在する場合、trueを返す
- containsAll(o) …… 引数のコレクション内の全ての要素をが存在する場合、trueを返す
- isEmpty() …… セットに要素が存在しない場合、trueを返す
- size() …… 要素の数を返す
- iterator() …… イテレータを返す

# マップインタフェース(Map)

## マップ

キー(key)と要素(value)による組み合わせを保持している。
キーを指定することで要素にアクセスすることが出来る。
キー値は重複することは出来ない。
要素値は重複しても良い。
複数のnull値を許可。
MapインタフェースはCollectionインターフェースとは別。

代表的なマップインタフェースの実装：
HashMap
LinkedHashMap

# マップインタフェース(Map)

主なメソッド

- put(key,value) …… キーと値で要素を追加する
- get(key) …… キーに対応する値を返す
- size() …… 要素の数を返す
- remove(key) …… キーに対応する要素を削除する
- keySet() …… すべてのキーのSetオブジェクトを返す
- values() …… すべての値のCollectionを返す

# 繰り返し処理

配列もコレクションも一言で表現すると「データの集まり」。

「データの集まり」を繰り返し処理する便利な方法が用意されている。

## Enumeration イニュメレーションとIterator イテレータ

「データの集合」から1つずつ要素を取り出し繰り返し処理出来る仕組み。

EnumerationとIteratorは似ているが、Enumerationは古くからある列挙型のインタフェースで、Iteratorの方が多く使われる。

Iteratorも、次に説明する拡張for文の登場であまり使われなくなった。

## Iteratorの使い方

```
Iterator iterator = list.iterator();
while (iterator.hasNext()) {
	String str = iterator.next();
	System.out.println(str);
}
```

## 拡張for文

Javaのバージョン5で登場。

「データの集合」から1つずつ要素を取り出すのが簡単に記述できる。

- Java.lang.Iterableインタフェースを実装したオブジェクト
- 配列のインスタンス

が拡張for文が使える条件

書き方：

```
for(データ型 変数名 : データの集まりの変数名){
    文;
}
```

データの集合から要素を取り出し変数に入れてくれる。

拡張for文の中では変数名で要素にアクセス出来る。

# ジェネリクス

コレクションは「参照型」の変数を保持することが出来ますが、
コレクションの変数宣言時・作成時に「参照型」のデータ型を指定することで
プログラムの誤りをコンパイル時に検出できるようにする仕組み。

書き方の例：

```
ArrayList list = new ArrayList();
↓
ArrayList<String> list = new ArrayList<String>();
```

# プリミティブ型とラッパークラス

Javaはオブジェクト指向の言語のため、あらゆるものを「参照型」の変数で扱いたい場面がある。
しかし変数には「基本型（プリミティブ型）」もある。

配列は「基本型（プリミティブ型）」でも作成出来るが、コレクションでは「基本型（プリミティブ型）」で作成出来ない。

「基本型（プリミティブ型）」をラップして（包んで）「参照型」の変数として扱えるラッパークラスがある。

# プリミティブ型とラッパークラス

## 主なラッパークラス

| 基本型 | ラッパークラス |
| --- | --- |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| char | Char |
| boolean | Boolean |

# プリミティブ型とラッパークラス

## ラッパークラス

「基本型（プリミティブ型）」にそれぞれ対応したラッパークラスが用意されている。

※ラッパークラスは値を変更できない。別の値にしたいときは別のオブジェクトを作成する必要がある。

```
Integer number = new Integer(“1”);      // Java9からは非推奨

Integer number = Integer.valueOf(“1”); // Java9からはvalueOfを使う
                                        // キャッシュを見てくれる
```

※Java8でもキャッシュを見てくれているのでvalueOf使いましょう

# プリミティブ型とラッパークラス

## オートボクシング

「基本型（プリミティブ型）」も「参照型」の変数として扱えるようになり便利になった反面、
「基本型（プリミティブ型）」と「参照型」の相互変換が必要になってしまったが、
Javaのバージョン5からこの相互変換を自動で行ってくれるようになった。

## キャストとは

データ型を変換することをキャストという。

小さな型を大きな型にする→拡大変換（ワイドニング）
大きな型を小さな型にする→縮小変換（ナローイング）※要注意

暗黙で行われることもあるが（ワイドニングのみ）、キャスト演算子で明示的に行うことも出来る。

```
// ワイドニング例
byte b = 20;
int i = (int)b;  // 明示的に行う
long l = b;      // 暗黙で行われる
```

```
// ナローイング例
long l = 20L;
int i = (int)l;
byte b = l;      // コンパイルエラー
```

## ナローイングの注意点

入りきらない場合は無理やり切り捨てられる。
浮動小数点型から整数型への変換も「ナローイング変換」。

```
long l = Long.MAX_VALUE; // l = 9223372036854775807
byte b = (byte)l;        // b = -1

double d = 123.45;       // d = 123.45
long val = (long)d;      // val = 123
```

# キャスト

参照型についてキャスト演算子でワイドニングした時は問題ない。
例：ArrayList → List(ワイドニングOK)

参照型についてキャスト演算子でナローイングした時はインスタンスの状態による。
例：　ArrayList → List(ワイドニングOK) → ArrayList(ナローイングOK)
→ LinkedList(ナローイングNG)
ナローイングできない時はClassCastExceptionが発生する。（例外で取り上げる）