

3.2 Java コレクション API

- コレクション API
- 関数型インタフェース
- ジェネリクス

目次

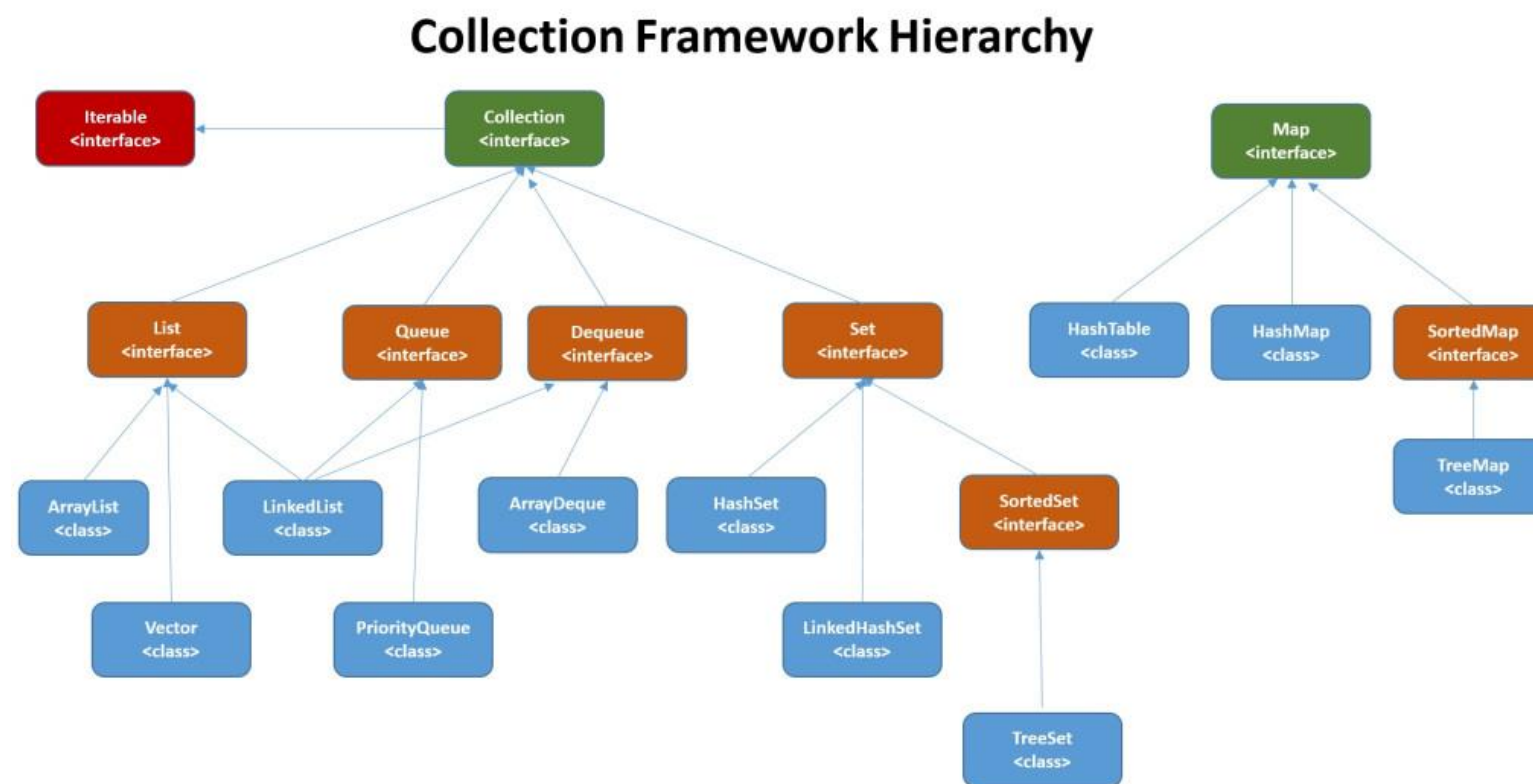
- 1 コレクション API
- 2 関数型インタフェース
- 3 ジェネリクス

Java Collections API

- 前節では、多くの一般的な抽象データ型とデータ構造を紹介した。実際には、これらの構造を自分で実装する必要はなく、プログラミング言語が提供する**アプリケーション・プログラミング・インタフェース**[Application Programming Interface, API]を利用するのが一般的です。
- Java は **Collections API** を提供し、多くの抽象的な型とデータ構造を実装しています。
- また、Java はこれらのデータ構造に関連するアルゴリズムや複雑な操作（ソートなど）を実装しました。
- Collections API が提供するクラスは、全て **java.util** パッケージ内にあります。

Collection と Map

- Collections の主な機能は、以下の 2 つのインタフェースとそのサブクラスによって提供されます：**Collection** と **Map**。前に紹介したほとんどの抽象型は Collection インタフェースを実装しています。**連想配列**だけは、2 つのデータ型（キーと値）の定義が必要なため、別途 Map インタフェースを実装しています。



List

- **List** はリストの抽象データ型を代表するクラスです。
- List は、リストに対するアクセス、挿入、削除などの一般的な操作を提供します：
 - `add(item)` : リストの末尾にデータを挿入。
 - `add(i, item)` : リストのインデックス i にデータを挿入。
 - `remove(i)` : インデックス i に対応する要素を削除。
 - `remove(item)` : `item` がリスト内に存在すれば、それを削除。

- `get(i)` : インデックス i の要素を取得。
- `set(i, item)` : インデックス i の要素を $item$ に設定。
- `size()` : リストの長さを取得します。
- それ以外にもメソッドがたくさんあります。全てのメソッドは、公式ドキュメントで閲覧できます：

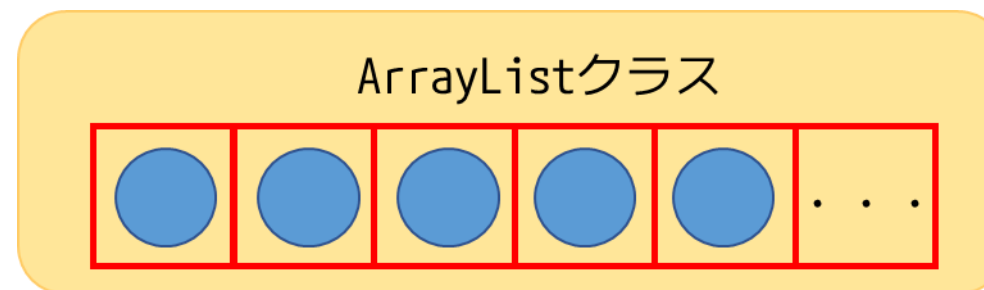
 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/List.html>

リストの作成

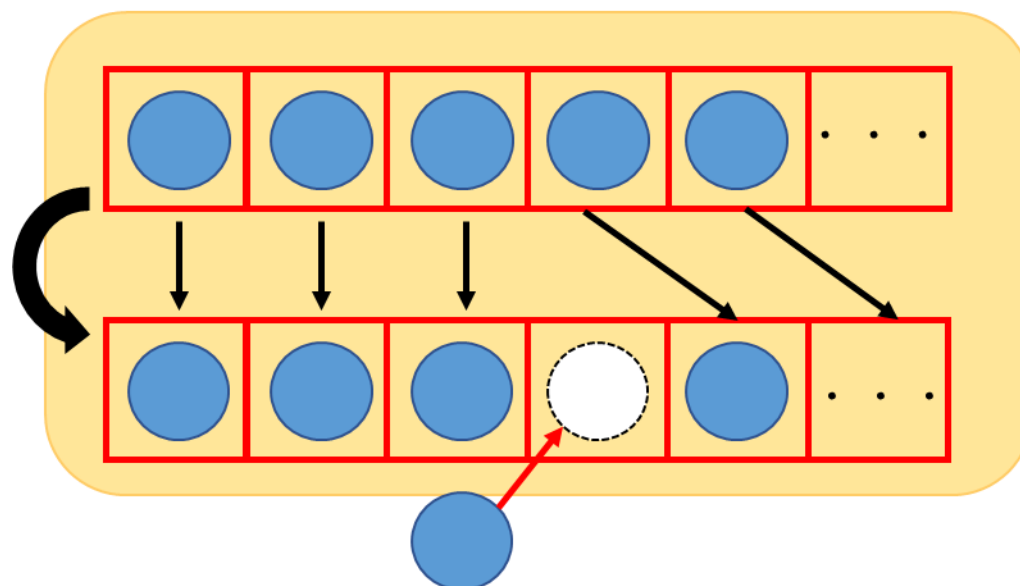
- List はリストを代表する**抽象的**な型で、Java では**インタフェース**として定義されています。したがって、**List のインスタンスを直接作成することはできず**、List を実装したデータ構造クラスのみ作成することができます。
- Java では、**ArrayList**（配列によって実装）、**LinkedList**（連結リストによって実装）など、リストの実装クラスが多数用意されています。

ArrayListクラスとは

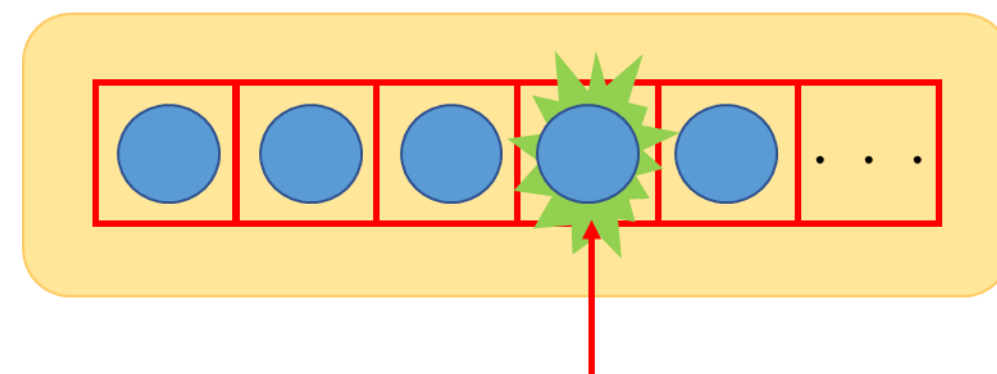
- 決まった要素数に対して、要素の「参照」を「高速」で行うことができる
- 要素の挿入・追加には、配列のコピーが発生する
→処理が遅い



要素の挿入

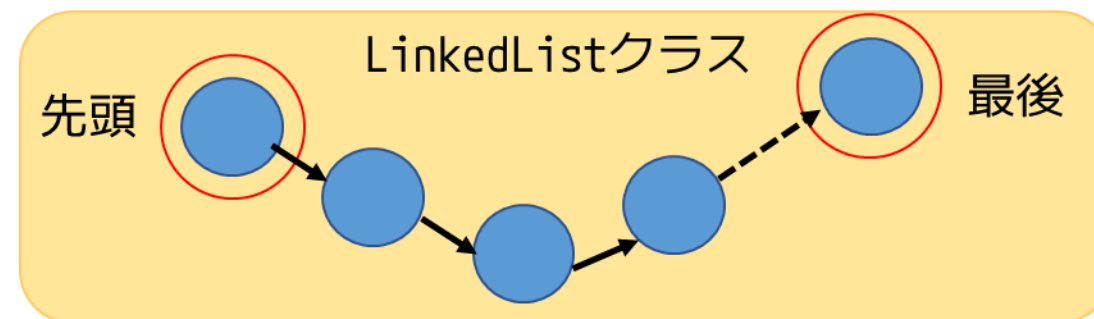


要素の参照

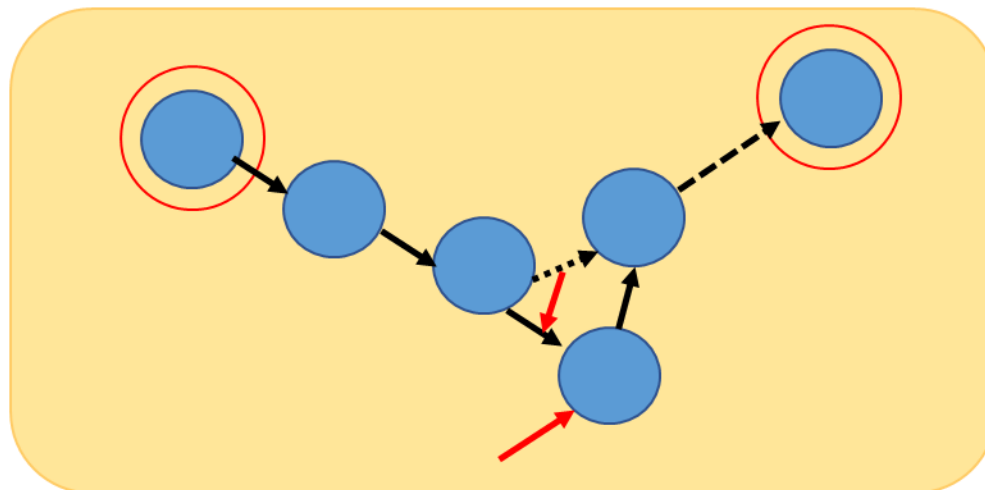


LinkedListクラスとは

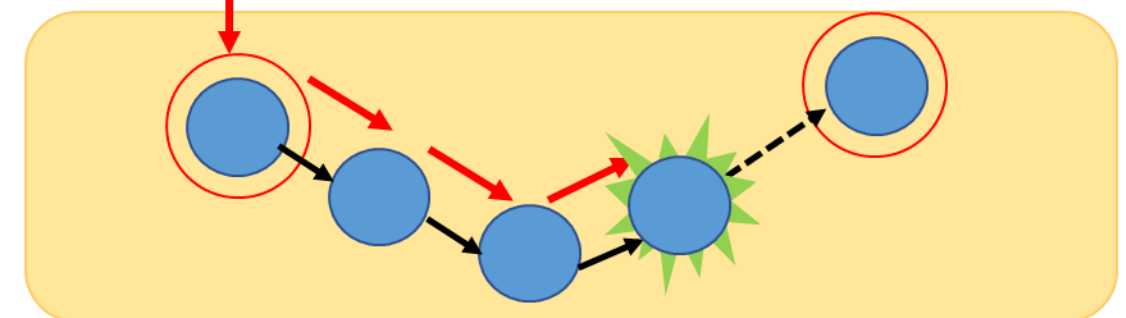
- 各要素をLinkでつなぐ構造のため、要素の追加は「Linkのつなぎかえ」でOK→処理が早い
- 要素の参照は、最初から見ていかないとインデクスが見つからない→参照が遅い



要素の挿入



要素の参照



リストの実装クラス

- これらのクラスのコンストラクタを使って、直接リストが作成できます。ArrayList を例として説明すると：

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

- クラス名の後ろの「**<String>**」は、リストが String 型のデータを含むことを意味します。これは総称型と呼ばれる特殊な文法で、後で説明します（[➡ § 3.2.3](#)）。

Try 
ListEx.java

リストを作成する簡単な方法

- また、**List.of()** メソッドを使用して、単純なリストを作成し、初期値を追加することも可能です：

```
List<String> nameList = List.of("Alice", "Bob", "Carol");
```

- 注意：このメソッドで作成されたリストは**不可変**^[Immutable]であり、追加や削除の際にエラーが発生します：

```
nameList.add("David"); // => java.lang.UnsupportedOperationException
```

- このようなリストを可変にするためには、ArrayList クラスのコンストラクタを使えばいいです：

```
1 List<String> nameList = new ArrayList<>(List.of("Alice", "Bob"));
2 nameList.add("Carol");
3 System.out.println(nameList; // => [Alice, Bob, Carol]
```

リストの繰り返し処理

- 以前、**for-each** ループの文法について説明しました。実際、for-each ループはリストを繰り返し処理するのにも使えます：

```
1 List<String> nameList = List.of("Alice", "Bob", "Carol");
2 for (String name : nameList) {
3     System.out.println(name); // => Alice Bob Carol
4 }
```

ラッパークラス

- 総称型を代表する山括弧「<>」内は、**参照型のみ**使用可能です。基本型のリストを作成したい場合は、対応する**ラッパークラス**^[Wrapper Class]を使用する必要があります。ラッパークラスは基本型の「参照型バージョン」として理解した方がいいです。
- 例えば、int 型のラッパークラスは Integer クラスです。したがって、整数のリストを作りたい場合は、次のようなコードを書けばいいです：


```
1 List<Integer> ageList = new ArrayList<>();
2 ageList.add(10);
```


ラッパークラス一覧

- 次の表は、Java の全ての基本型（void を除き）に対するラッパークラスの一覧です：

基本型	ラッパークラス
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Stack

- **Stack** は**スタック**を代表する抽象データ型クラスで、同時にその実装クラスでもあります。
- Stack は、スタックに対するいくつかの一般的な操作を提供します：
 - `push(item)` : データをスタックの先頭に追加（プッシュ）。
 - `pop()` : スタックの先頭要素を取得して削除（ポップ）。
 - `peek()` : スタックの先頭要素を削除せずに取得。
 - `size()` : スタックのサイズを取得。
- 全てのメソッドは公式ドキュメントで閲覧できます：
 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Stack.html>


スタックの作成と使用

- Stack クラス自体が実装クラスなので、そのコンストラクタで直接スタックが作成できます：

```
1 Stack<String> nameStack = new Stack<>();  
2 nameStack.push("Alice");  
3 nameStack.push("Bob");  
4 nameStack.push("Carol");  
5 System.out.println(nameStack); // => [Alice, Bob, Carol]
```

Try 
StackEx.java

Queue

- **Queue** は**キュー**を代表する抽象型インタフェースです。
- キューに関するいくつかの一般的な操作を提供します：
 - `offer(item)` : データをキューの末尾に追加。
 - `poll()` : キューの要素を取得して削除。
 - `peek()` : キューの先頭要素を削除せずに取得。
 - `size()` : キューのサイズを取得。
- 全てのメソッドは公式ドキュメントで閲覧できます：
 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Queue.html>


キューの作成と使用

- Java では、LinkedList や ArrayDeque などのクラスでキューを実装しています：

```
1 Queue<String> nameQueue = new LinkedList<>();
2 nameQueue.offer("Alice");
3 nameQueue.offer("Bob");
4 nameQueue.offer("Carol");
5 System.out.println(nameQueue); // => [Alice, Bob, Carol]
```

Try  QueueEx.java

Set

- **Set** は**集合**を代表する抽象型クラスです。
- 集合に対するいくつかの一般的な操作を提供します：
 - `add(item)` : 集合にデータを追加。
 - `remove(item)` : `item` が集合内に存在すれば、それを削除。
 - `contains(item)` : `item` が集合に存在するかどうかを判定。
 - `size()` : 集合のサイズを取得。
- 全てのメソッドは公式ドキュメントで閲覧できます：
 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Set.html>


集合の作成と使用

- 一般的な集合の実装は HashSet、TreeSet などがあります。
- リストと同様に、集合も **Set.of()** メソッドで簡単に作成できます。
- 集合も for-each 文で繰り返し操作できます：

```
1 Set<String> nameSet = Set.of("Alice", "Bob", "Carol");
2 for (String name : nameSet) {
3     System.out.println(name); // => Bob Alice Carol
4 }
```

Try 01011
11010
01011
SetEx.java

Map

- **Map** は**連想配列**を代表する抽象型クラスです。
- 連想配列に対するいくつかの一般的な操作を提供します：
 - `put(key, value)` : `key` に対応する値を `value` に設定。
 - `remove(key)` : 連想配列から指定されたキーと値のペアを削除。
 - `get(key)` : 指定されたキーに対応する値を取得。
 - `containsKey(key)` : 指定されたキーが、連想配列に含まれるかどうかを判定。
 - `containsValue(value)` : 指定された値が、連想配列に含まれるかどうかを判定。
 - `size()` : 連想配列のサイズを取得。
- 全てのメソッドは公式ドキュメントで閲覧できます：
 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Map.html>

連想配列の作成と使用

- HashMap、TreeMap などの連想配列の実装があります。
- 他のクラスと異なり、連想配列を宣言するためには、キーの型と値の型の **2 種類の型** を「<>」に表記する必要があります：

```
1 Map<String, Integer> ages = new java.util.HashMap<>();
2 ages.put("Alice", 3);
3 ages.put("Bob", 5);
4 ages.put("Carol", 2);
5 // => Bob 5 Alice 3 Carol 2
6 for (String name : ages.keySet()) {
7     System.out.println(name + " " + ages.get(name));
8 }
```



まとめ : Collections API を使う流れ

Sum Up

データ構造の決定プロセスと、実際に Java でどのように使用されているかを復習してみましょう：

1. 実際の問題に応じて、データが保存される**抽象データ型**を決定。
Java では、Collection インタフェースや Map インタフェースのサブインタフェース・サブクラスから、適切な抽象型を検索。
2. 実装にどんな**データ構造**を使うか、必要機能に応じて決定。
Java では、インタフェースのドキュメントでそのメソッドを全てチェック可能。
3. データ構造の**実装**。
Collections API は、一般的な構成要素のほとんどを実装しますが、特定の要求がある場合は、インターネットでサードパーティのコードライブラリを検索する道もあります。

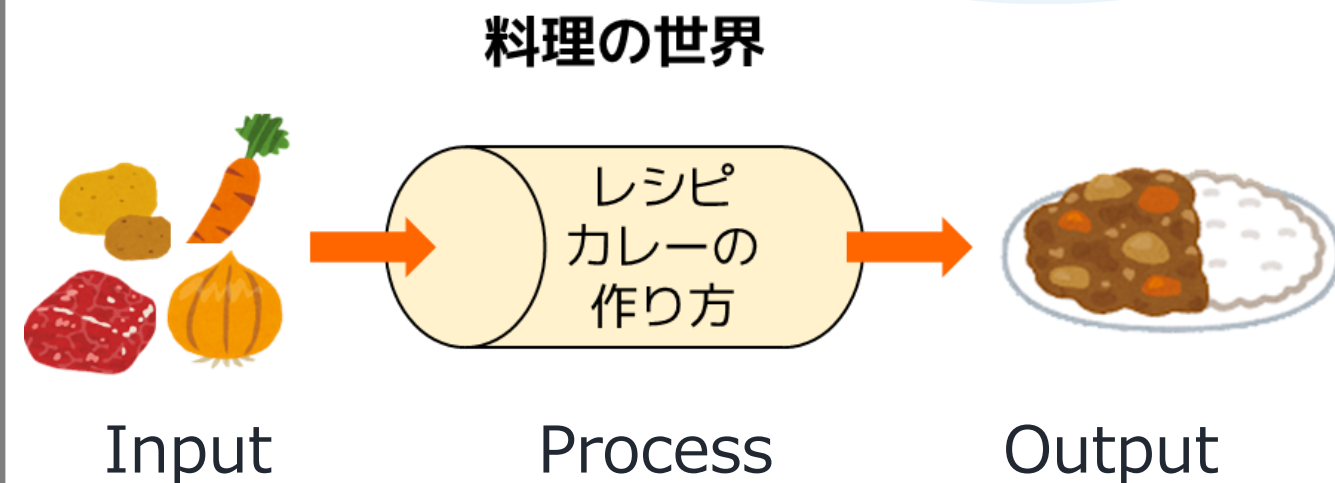
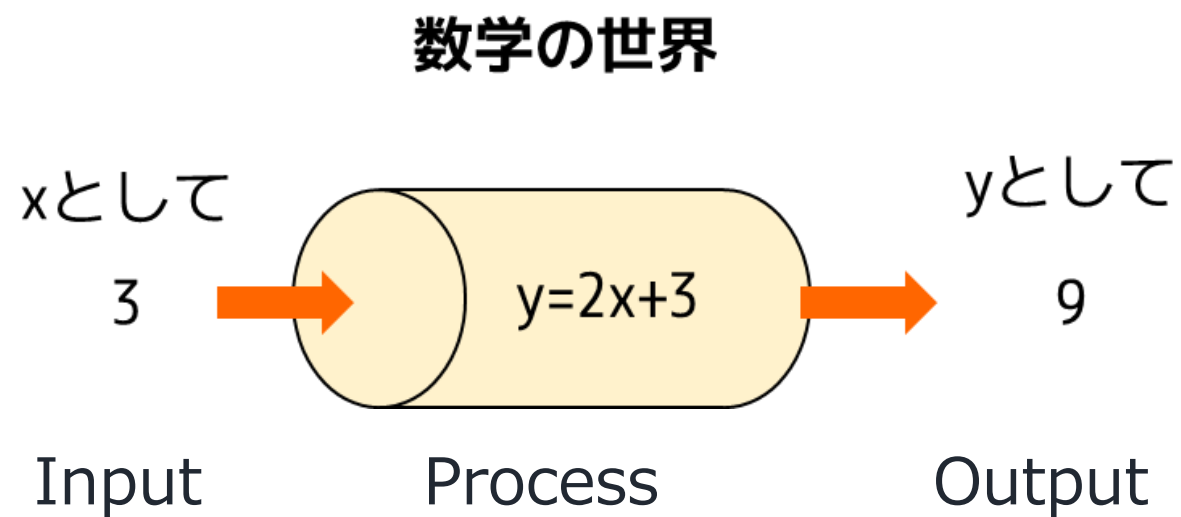
Q&A

目次

- 1 コレクション API
- 2 関数型インタフェース
- 3 ジェネリクス

関数とは

- 何らかの**入力**^[INPUT]を受け取って、何らかの**処理**^[Process]を行い、何らかの**出力**^[Output]を返すもの
- メソッドは関数の一種である。



関数のイメージ

- レシピには、必要な材料（I）、調理の手順（P）が記述されており、調理すれば料理（O）が出来上がる。
- このレシピには、名前がない。「名もなきメニュー」となる
- しかし、このレシピに名前がないが、役にたたないわけではない。
- 材料をレシピに渡せば、料理という出力を生み出せる

<材料>

・卵 ・油 ・小麦 ・鶏肉

<手順>

- 1：卵を混ぜる
- 2：混ぜた卵に鶏肉をつける
- 3：小麦を鶏肉につける
- 4：油であげる
- 5：火が通ったら完成

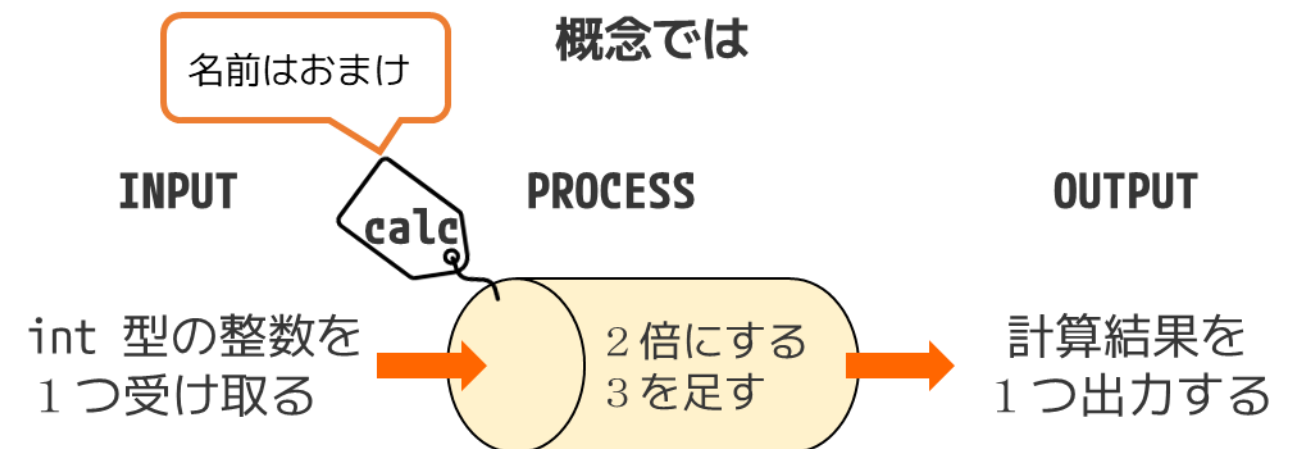
メソッドとの違い

- 関数にとって名前は、重要ではなく必須ではない。
- メソッドは必ず名前を持ちます。
名前がなければ呼び出すことができません。
- 関数にとって重要なのは、
- 「何を入力して受け取り、どんな処理をし、どんな出力をするか」という部分で、その処理のロジックにどのような名前が付けられているかは二の次です、実際に名前を持たない関数もあります。

ソースコードでは名前はどうでもいい

```
int calc(int x) {
    x = x*2;
    x = x+3;
    return x;
}
```

引数・戻り値・内容は重要



関数の代入

- これまでは、メソッドという一種の関数を変数に格納したことはなかった。
- メソッドを変数に格納することもできる

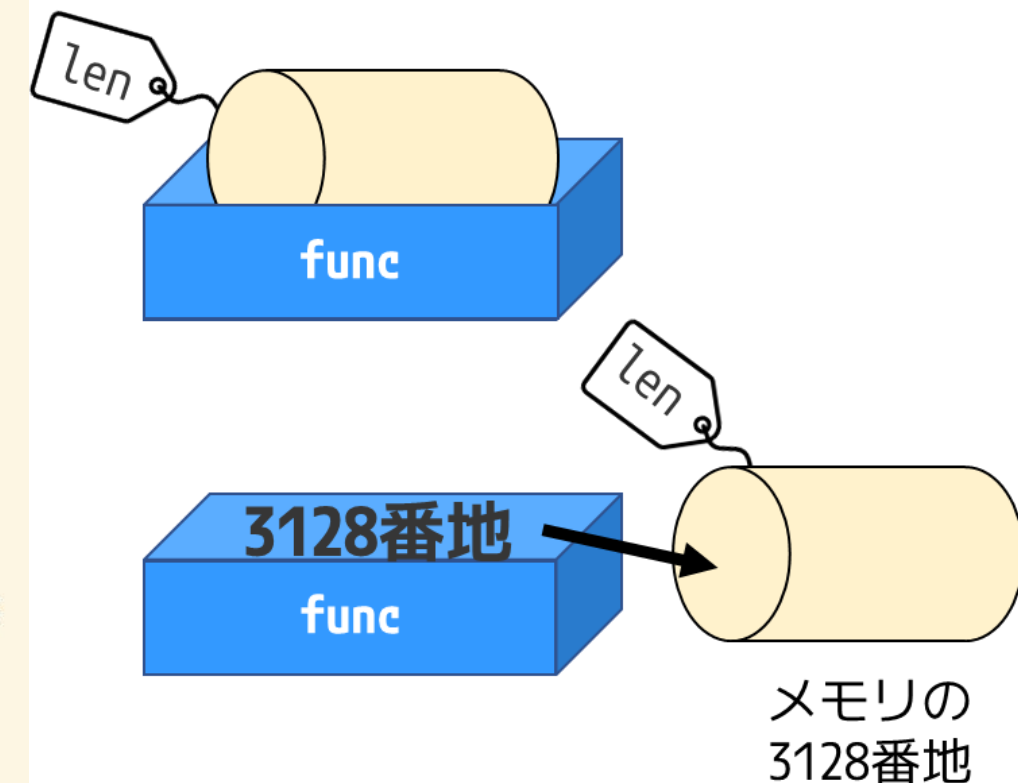
```

1 import java.util.function.Function;
2
3 public class Main {
4
5     // 文字列を受け取って、その文字を返す関数
6     public static Integer len(String s) {
7         return s.length();
8     }
9
10    public static void main(String[] args) {
11        // lenメソッドの処理を変数funcに代入する
12        Function<String, Integer> func = Main::len;
13
14        // 変数funcに格納されている処理内容を引数「HelloWorld」で実行する
15        int a = func.apply("HelloWorld");
16        System.out.println("文字列「HelloWorld」は" + a + "文字です");
17    }
18
19 }

```

funcに代入されているのは、lenメソッドへの参照

誤ったイメージ



Function型とそのバリエーション

- Function型は、`java.util.function.Function`インターフェースとして定義されている。
- 様々な種類の関数オブジェクトへの参照を格納できる。
- `Function<T,R>` : 引数をT、戻り値の型をRとしている
- 引数と戻り値の型さえわかれば使うことができる

しかし...

残念ながらすべての関数オブジェクトをFunction型に代入できるとは限らない。引数が無かったり戻り値がない関数もあるためです。

次へ 

標準的な関数型インタフェース

- Java.util.function パッケージ内には、Function 以外にも様々な関数オブジェクトを格納するためのインタフェースが存在する。それを「標準関数インタフェース」と総称している。
- APIリファレンスを見ると標準関数インタフェースは40種類以上も存在するが、その中からよく使用するものを紹介します。

 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/function/package-summary.html>

次へ 

標準インタフェース

- 一般的な標準インタフェースは以下の通り：

インターフェース	メソッドの形
Function<T, R>	T 型の引数を 1 つ受け取り、R 型の値を返す
BiFunction<T, U, R>	T、U 型の 2 つの引数を受け取り、R 型の値を返す
Supplier<R>	引数を受け取らず、R 型の値を返す
Consumer<T>	T 型引数 1 個を受け取り、戻り値はない
BiConsumer<T, U>	T、U 型の 2 つの引数を受け取り、戻り値はない
IntFunction<T>	T 型の引数を 1 つ取り、int型の値を返す
Predicate<T>	T 型の引数を 1 つ取り、boolean 型の値を返す

オリジナルの関数型インタフェース

- ここまで紹介してきた標準関数インタフェースだけでは、すべてのケースに対応することはできません。
- JAVAでは、私たち自身でオリジナルの**関数型インタフェース**[Functional Interface]を定義することが許されている。
- 具体的には、**1つの抽象メソッドのみ**[SAM:Single Abstract Method]を持つインタフェースは、関数インタフェースとして扱われ、以下の振る舞いが可能となる。
 - 抽象メソッド宣言に記述した引数と型が一致する関数オブジェクトを格納できる。
 - 抽象メソッド名で呼び出すことができる。

次へ 

関数型インタフェースの例

Example



```
1 // 抽象メソッドが1なので関数型インタフェースになる！  
2 interface IntA {  
3     public void methodA();  
4 }
```

Example



```
1 // 抽象メソッドが複数あるので関数型インタフェースにはならない！  
2 interface IntC {  
3     public void methodA();  
4  
5     public void methodB();  
6 }
```


関数型インタフェースの実装

- 今回は、並び変えのComparatorというインタフェースを紹介します。
- 例えば、Collectionsクラスのsortはその名が示す通り、呼び出すだけで要素を順番に並び替えてくれる便利なメソッドです。
- しかしsort()には重要な制約が1つあり、それを意識しないで、下記のようなソースを書くとエラーが起こります。

```
public class NoComparator {

    public static void main(String[] args) {

        List<Cat> list = new ArrayList<Cat>();
        list.add(new Cat("Alice", 12.5));
        list.add(new Cat("Bob", 18.7));
        list.add(new Cat("Mai", 14.5));

        Collections.sort(list);

    }

}
```

Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The method sort(List<T>) in the type Collections is not applicable for the arguments (List<Cat>)

Try 
NoComparator.java

- ただ、並び替えるといっても猫の並び変えには、名前のABC順や体重順等、色々な方法が考えられる。単に並び替えると言われてもJAVAも困ります。
そこで、体重順に並べ替えなさいと、並べ替えたい並べ順をあらかじめ宣言しておくなら話は別です。
- あるクラスについて一般に想定される並べ替え順を自然順序付けといいます。
自然順序が定めてあるクラスであれば、Collections.sortという指示でもエラーがでずに並び変えることが可能です。

Comparatorとは

- Comparatorは、現実世界における「並び替えのルール」(2つのインスタンスの大小を比較する関数) をクラスとして表現したものです。
- `java.util.Comparator` インタフェースを実装し、その唯一のメソッドである`compare()`をオーバーライドして定義します。
- 今回は、Catクラスを作成、体重に基づいて並び変えをしていこうと思います。

手順1 : Catクラスの作成

```
1 public class Cat {  
2     private String name;  
3     private double weight;  
4  
5     public Cat(String name, double weight) {  
6         this.name = name;  
7         this.weight = weight;  
8     }  
9  
10    public String getName() {  
11        return name;  
12    }  
13  
14    public double getWeight() {  
15        return weight;  
16    }  
17  
18    @Override  
19    public String toString() {  
20        return "Cat[" + name + ", weight: " + weight + "];"  
21    }  
22 }
```

手順2：並び替え順を定義

- 体重(weight)を元にソートしたい状況を考える。
- 以下のようなComparatorを定義する。

```

1 import java.util.Comparator;
2
3 public class CatComparator implements Comparator<Cat>{
4     @Override
5     public int compare(Cat cat1, Cat cat2) {
6         if (cat1.getWeight() == cat2.getWeight()) {
7             return 0;
8         } else if (cat1.getWeight() < cat2.getWeight()) {
9             return -1;
10        } else {
11            return 1;
12        }
13    }
14 }

```

比較したいクラスを<>形式で指定



Compareメソッドの解説

- compare メソッドで、

負の数、
0、
正の数 (int)、

を返すことによって、その大小を定義します。

- 第一引数を小さいとする場合は負の数。
- 第一引数を大きいとする場合は正の数。
- どちらでもない場合、すなわち引数の大小が同じの場合、0を返せば良いです。

定義したコンパレータの利用

最後は、Comparatorというインタフェース型のデータをsortメソッドの引数に渡して並び変えをしていきます。

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

```
public class Main {
```

```
    public static void main(String[] args) {
        List<Cat> list = new ArrayList<Cat>();
        list.add(new Cat("Alice", 12.5));
        list.add(new Cat("Bob", 18.7));
        list.add(new Cat("Mai", 14.5));
```

```
        Collections.sort(list, new CatComparator());
        System.out.println(list);
```

```
    }
```

//実行結果

//[Cat[Alice, weight: 12.5], Cat[Mai, weight: 14.5], Cat[Bob, weight: 18.7]]

```
}
```

Collections.sort(list,cmp);

List:並び変えたいコレクション

cmp:コンパレータのインスタンス

第二引数に指定された順序で並び変える

Q&A

匿名クラス

- 今まで、インタフェースや抽象クラスがあったら、サブクラスや実装クラスを作成して利用しますが、プログラムの中で何らかのインタフェース等を利用したい処理が出てくるときに実装クラスを作成しなければいけなくなります。
- 抽象クラスやインタフェースを利用したい箇所がそこだけに限られている場合でも、実装したいメソッドごとに実装クラスを作成することになり、定義するクラスが非常に多くなってしまう。
- その手間を省くために、「匿名クラス」というものを利用することができます。
- 匿名クラスを利用することで、新しくクラスを作成せずに抽象クラスやインタフェースをそのままインスタンス化させるような動作を行うことができます。

次へ



従来のインタフェース利用方法

```
インタフェース {
    抽象メソッド
}
```



実装クラスを作成

```
Aクラス implements インタフェース {
    抽象メソッドをオーバーライド
}
```



実装クラスをインスタンス化

```
利用クラス{
    public static void main(String[] args){
        //実装クラスをインスタンス化
        A a = new A();
        aのメソッドを呼び出し
    }
}
```

匿名クラスを利用した インタフェース利用方法

```
インタフェース {
    抽象メソッド
}
```



実装クラス無しで
インタフェースを
実装したオブジェ
クトを作成、利用
する。

```
利用クラス{
    public static void main(String[] args){
        //匿名クラスでインスタンス化
        A a = new A(){
            抽象メソッドをオーバーライド
        };
        aのメソッドを呼び出し
    }
}
```

匿名クラスは明確なクラスとして定義しないので、匿名クラスを利用したそのスコープ内や利用した場所でのみインスタンス化したオブジェクトを利用できません。



匿名クラスの例

- 匿名クラスを利用して作成したオブジェクトを変数に代入して利用する方法です。
- オブジェクトを格納する変数のデータ型はインタフェースを指定します。

```
interface SampleEx { // SampleExインタフェースの定義
    public void practice(); // 抽象メソッド「practice()」を定義
}

public class AnonymityEx {

    public static void main(String[] args) {
        SampleEx sampleEx = new SampleEx() { // SampleEx型の変数「sampleEx」を定義し、匿名クラスで実装する

            @Override // インタフェースに定義された抽象メソッドのオーバーライド
            public void practice() {
                System.out.println("practiceメソッドのオーバーライド");
            }
        };

        // 定義されたローカル変数の「sampleEx」はスコープ内で呼び出しが可能になります。
        sampleEx.practice();
        // sampleExオブジェクトのpractice()メソッドを呼び出す。
    }
}
```

次へ

- 匿名クラスの書き方を使って、先ほどComparatorを使って作成したソースはどれほど短くなるのか見てみよう！

Try 
AnonymityEx.java

Q&A

ラムダ式

- 先程まで、匿名クラスを学んできましたが、今回学ぶラムダ式を勉強することで、匿名クラスを利用した記述に比べてさらにソースの量を減らすことができます。
- **ラムダ式**^[Lambda Expression]は、処理（インタフェースを実装、インスタンスを生成）を簡潔に実装するための文法です。

関数型インタフェース

```
Interface インタフェース{
    戻り値の型 メソッド名(引数);
}
```

ラムダ式

```
インタフェース 変数 = 引数 -> 処理と戻り値;
```

オーバーライド

ラムダ式の利用

- ラムダ式の基本構文は以下の通り：

```
1 (arg1, arg2) -> {  
2     codes;  
3 }
```

Try 
Lambda.java

- ここで、冒頭の括弧「**()**」はメソッドのパラメータのリストを書くために使われ、**パラメータの型は書きません**。中括弧「**{}**」内にメソッド本体を記述し、矢印「**->**」で繋がれます。**メソッド名や戻り値の型を書く必要はありません**。
- この式は、実際には、ある関数型インタフェースを実装したクラスのインスタンス（オブジェクト）を生成するものです。このオブジェクトは、関数型インタフェースを必要とする関数に直接渡してよい、変数に格納しておいてもよい。

ラムダ式の省略文法

- パラメータが 1 つだけの場合は、括弧「()」が省略可能：

```
1 a -> {  
2     int b = a * 2;  
3     return b;  
4 }
```



- メソッド本体に return 文 1 つしかない場合、「{}」が省略可能：

```
(a, b) -> a + b
```

- メソッドが 1 行だけで戻り値もない時も「{}」が省略可能：

```
a -> System.out.println(a)
```

ラムダ式と匿名内部クラス

- ラムダ式は、実際にはインタフェースを実装し、その実装したクラスをインスタンス化します。言い換えれば、実際には名無しの新しいクラスを宣言しました。
- このクラスは、外部クラス（のオブジェクト）に属する内部クラスです。**匿名内部クラス**[Anonymous Inner Class]と呼ばれます。
- 前に述べたように（ § 2.4.5）、内部クラスはそれを作る外部オブジェクトの変数やメソッドが使用できます。故に、**ラムダ式も外部クラスの変数やメソッドが直接利用できます。**
- これは特定の状況で役に立ちます。例えば、後（ § 2.5.1）紹介するスレッドクラスの作成。

Stream API

- **Stream API** は、Java が Collections API のデータ構造を利用するための強力かつ便利な API です。
- Stream API の基本的な使い方は以下の通りです：
 1. データ構造（Collection クラスのサブクラス）から、対応する Stream オブジェクトを取得。
 2. Stream オブジェクトのメソッドで簡単にデータを操作。
 3. （必要な場合）Stream を Collection に戻す。

ステップ 2 にあたる Stream クラスのメソッドには、ラムダ式で簡潔に表現できるものがあります。

- Stream クラスは、リスト全体に対する便利な操作を多数提供する、特別なリスト、と理解すればいいでしょう。

Stream オブジェクトの取得

- リストや集合などの Collection クラスのほとんど（あるいはそれらのサブクラス、実装クラス）は、その **stream()** メソッドで簡単に Stream オブジェクトが取得できます：

```
1 List<String> names = new ArrayList<>();
2 names.add("Alice");
3 names.add("Bob");
4 Stream<String> stream = names.stream();
```

- Map などの他のデータ構造も、それらが含む Collection のような構造（Map の `keySet()` や `entrySet()` など）を取得することで、Stream で使用できます。

Stream メソッドの使用

- Stream オブジェクトを取得したら、Stream のメソッドを使って簡単にデータが処理できます。例えば、データをソートするための `sorted()` メソッドが提供され、その結果は別の Stream になりますが、データは順番に並べられます：


```
1 List<Integer> nums = new ArrayList<>(List.of(1,3,4,2));
2 Stream<Integer> stream = nums.stream().sorted();
```

- もう一つの例は `forEach()` メソッドです。Consumer インタフェース（ラムダ式で実装可能）を受け取り、Stream にある全データに対して、渡されたメソッドを実行します：

```
1 List<Integer> nums = new ArrayList<>(List.of(1,3,4,2));
2 // 1 2 3 4
3 nums.stream()
4     .sorted()
5     .forEach(num -> System.out.println(num));
```

Stream のメソッドの一覧

- 一般的な Stream のメソッドを紹介します：
 - `forEach(consumer)` : 各要素に `consumer` メソッドが適用され、戻り値はない。
 - `map(function)` : 各要素を `function` メソッドで変換し、新しい Stream を作成。
 - `filter(predicate)` : 元の全ての要素から、`predicate` を満たすものだけを選んで、新しい Stream を作成。
 - `reduce(init, operator)` : 全ての要素に対する累積的な演算（足し算、掛け算など）を行い、演算結果を返す。`init` は初期値、`operator` は演算の方法を指定します。

- `max(comparator)` : 全要素の最大値を計算。`comparator` はリスト内の要素を比較するためのメソッドを定義（以下同じ）。
- `min(comparator)` : 全要素の最小値を計算。
- `sorted(comparator)` : 全ての要素をソートし、整列された Stream を返す。
- `distinct()` : 重複する要素を削除し、新しいStreamを返します。
- 全てのメソッドの一覧は公式ドキュメントに閲覧できます：
 <https://docs.oracle.com/javase/jp/8/docs/api/java/util/stream/Stream.html>

Stream の変換

- `map()` や `sorted()` などの一部のメソッドはまだ Stream を返すので、これらを連続に使うことができます。Stream を配列に変換したい時は、**`toArray()`** メソッドが使用可能：

```
1 List<String> names = new ArrayList<>(List.of("Bob", "Carol", "Alice"));
2 String[] arr = names.stream().sorted().toArray(String[]::new);
3 System.out.println(Arrays.toString(arr)); // [Alice, Bob, Carol]
```

- また、**`collect()`** メソッドを使用し、Collection オブジェクト（リストなど）に変換することもできます：

```
1 List<String> names = new ArrayList<>(List.of("Bob", "Carol", "Alice"));
2 List<String> list = names.stream().sorted().collect(Collectors.toList());
3 System.out.println(list); // [Alice, Bob, Carol]
```

Try 
Stream.java

メソッド参照

- ラムダ式では、以下のように既成のメソッドを直接使用することもよくあります：

```
a -> System.out.println(a)
```

- このとき、Java の**メソッド参照**[\[Method Reference\]](#)構文を使ってより簡潔にメソッドを渡すことができます。メソッドの参照を取得するにはダブルコロン演算子「**::**」を使用します：

```
1 List<String> names = new ArrayList<>(List.of("Bob", "Carol", "Alice"));
2 // Alice Bob Carol
3 names.stream().sorted().forEach(System.out::println);
```

- ダブルコロン演算子の使い方は、通常のメソッド呼び出しと同様です：静的なメソッドじゃない場合、「**::**」でオブジェクト名とメソッド名を結び、静的メソッドの場合、「**::**」でクラス名とメソッド名を結びます。

Q&A

目次

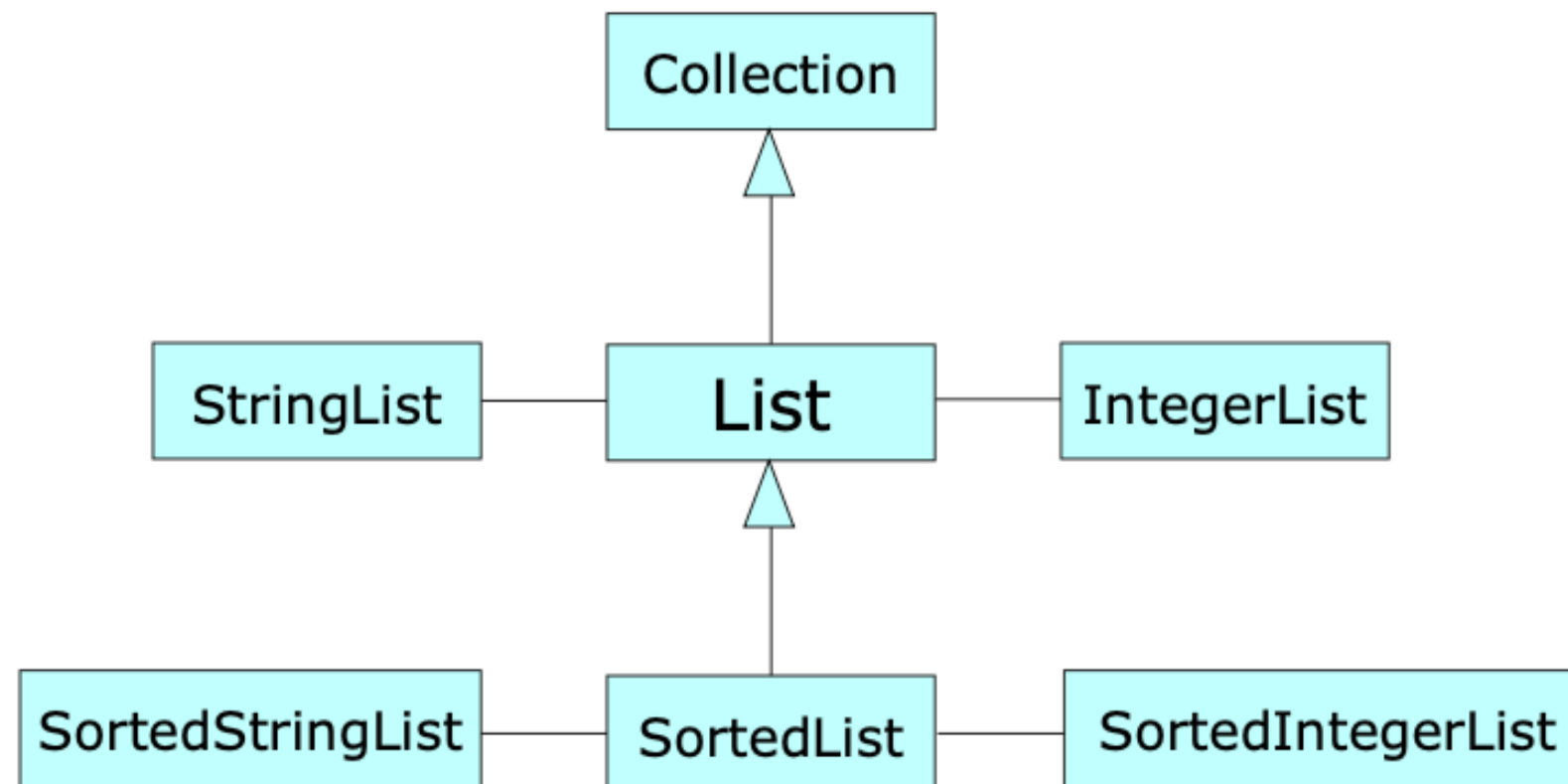
- 1 コレクション API
- 2 関数型インタフェース
- 3 ジェネリクス

総称型

- Collections API を学ぶ過程で、**総称型**、または**ジェネリクス**^[Generics]の使い方にちょっとだけ触れました。
- 簡単に言うと、総称型は 1 つ（または 2 つ、3 つ…）の**型をパラメータとして受け取る**ことができる特別なクラスです。
- 例えば、List を使用する場合は、パラメータとして、リストに保存されたデータの型を指定する必要があります。String、Integer などの異なる型を指定できるので、異なる型のデータを保持するリストが利用できます。
- 考えてみてください：総称型がなければ、あらゆる種類のデータを扱えるクラスをどのように実装すればいいのでしょうか？

方法 1：専用クラスの作成

- 最初に考えられるのは、データの種類によって異なるのクラスを作る：



- この方法はどのような問題があるのでしょうか？
 1. 作成するクラスの数が多くて、開発と保守に不便。
 2. 新しいデータ型が追加されるたびに、対応するクラスを個別に開発する必要があります。

方法 2 : Object クラスの使用

- 全ての参照型は Object のサブクラスであることが分かっているなので、任意の型のデータを格納するために、Object のインスタンスを保持する List クラスを書けばいいです :

```

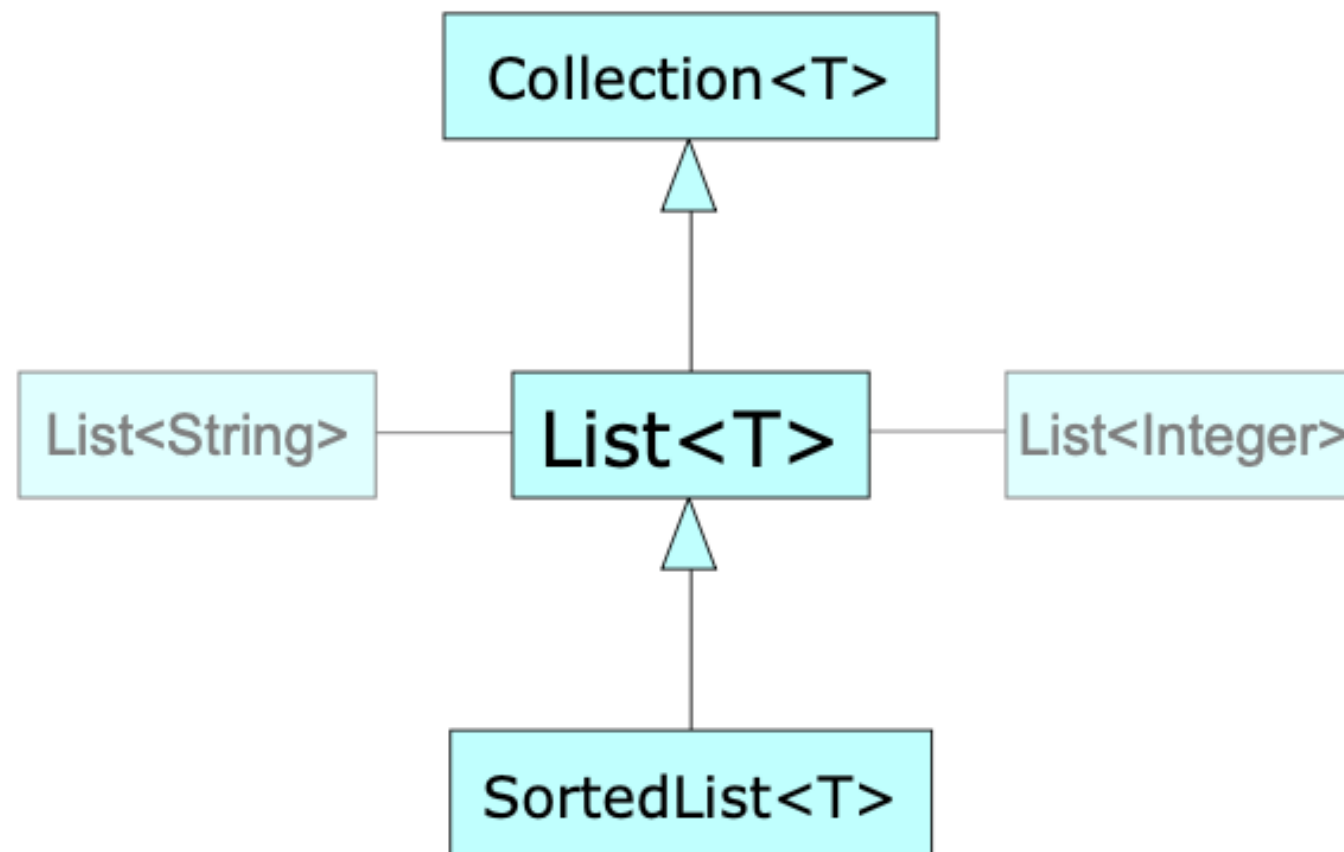
1 public class List {
2     public void add(Object obj) {
3         // ...
4     }
5     public Object get() {
6         // ...
7     }
8 }

```

- しかしこの方法にも問題があります。それは何でしょうか。
 - データを取得するたびに型変換を行う必要があります。
 - 異なる種類のデータが同じリストに格納されてしまいます。
 - 可読性の低下。

方法 3 : 総称型

- 総称型を使えば、どんなデータ型にも共通するメソッドを提供する List クラスを、**1 つだけ書けばいいです**。サブクラスや実装クラスも最小限のコードで書けます。



総称型の宣言

- Java が提供する総称型を利用するだけでなく、自分の総称型を作成することも可能です。総称型を宣言するには、クラス・インタフェース名の後に山括弧「**<>**」を付け、その中にパラメータとして使用する型名を記述します：

```
public class MyList<T> {}
```

- ここでは、MyList クラスはパラメータとして型を受け取り、それを T という名前で覚えます。例えば、String を格納する MyList を作成した場合、T が String になっています。
- 複数のタイプを受け付ける場合は、カンマ「**,**」で名前を区切ります：

```
public class Map<K, V> { }
```

総称型の使用

- Collections API を使用する際に、総称型を使用する方法は既に学びました：

```
MyList<String>
```

```
Map<String, Integer>
```

- コンストラクタやその他の静的メソッドの使用も同じです：

```
MyList<String> names = new MyList<String>();
```

Tips💡

Java がパラメータの型を推測できる場合、
「<>」内の内容は省略可能です：

```
MyList<String> names = new MyList<>();
```

総称型におけるメソッド

- 総称型の宣言で定義された型パラメータ（先の例の T、K など）は、クラス内に、型として直接使用することが可能です。例えば、メソッドのパラメータ型や戻り値として使用できます：

```

1 public class MyList<T> {
2     void add(T data) {
3         // ...
4     }
5     T get() {
6         // ...
7     }
8 }

```

Try  MyList.java

- まだ、ジェネリックメソッド [Generic Method] という高度な構文もありますが、ここでは詳しく説明しません。

Q&A

まとめ

Sum Up



1. Java Collections API :

- ① Collection と Map クラス。
- ② 抽象的なデータ構造 : **List**、Stack、Queue、Set、**Map**。
- ③ 上記の抽象構造の実装のクラス。

2. 関数型インタフェースとラムダ式 :

- ① ラムダ式の基本文法とその省略型、メソッド参照。
- ② Stream API : stream() メソッドでのラムダ式の使用。

3. 総称型の概念と基本文法。



Light in Your Career.
THANK YOU!