

## 3.2 Java コレクション API

- コレクション API
- 関数型インターフェース
- 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

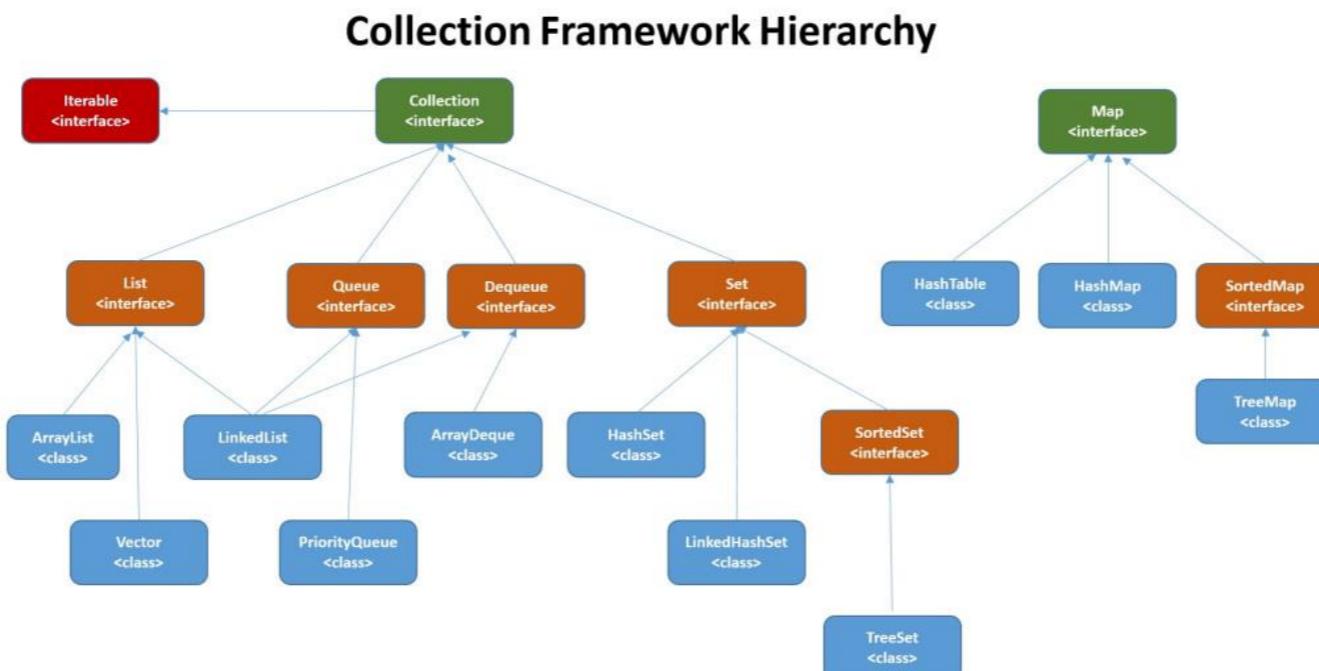
ジェネリクス

# 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・インターフェース**[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 つのインター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。

次へ 

◀ 前へ

- `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:  
 [6](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util>List.html</a></li></ul></div><div data-bbox=)

# リストの作成

- List はリストを代表する**抽象的**な型で、Java ではインター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。

# 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、直接リストを作成できます。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）。



# 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」として理解したらいいです。
- 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 **Stack** 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、 **Stack** 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 **Stack** の先頭に追加（プッシュ）。
  - `pop()`： **Stack** の先頭要素を取得して削除（ポップ）。
  - `peek()`： **Stack** の先頭要素を削除せずに取得。
  - `size()`： **Stack** 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ーフェースです。
- 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  
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、関連配列に含まれるかどうかを判定。
  - `containsValue(value)`: 指定された値が、関連する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 }
```

Try   
MapEx.java

# 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ーフェース・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、インター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。



*Question and answer*



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

# 関数型インターフェース

- プログラミングでは、実装すべきメソッドが 1 つしかない、特殊なインターフェースをよく使います。これは**関数型インターフェース**[Functional Interface]と呼ばれます。関数型インターフェースは、主に「メソッド」を引数としてメソッドに渡したい場合使われます。
- 例えば、Java の List クラスの sort() メソッドを使いたい場合、リスト内のデータをどのように比較するか（どんなデータが大きいのか、どんなデータが小さいのか）を Java に教えなければなりません。
- Java では、compare() メソッドだけを含む Comparator というインターフェース型のデータを sort() メソッドの引数として渡すことが必要です。

次へ 

 前へ

- したがって、我々は `sort()` メソッドを使うためには、まずはこの `Comparator` インタフェースを実装した別のクラスを作って、その中に `compare()` メソッドをオーバーライドして実装して、最後にこのクラスのインスタンスを作らなければなりません。



# ラムダ式

- 先の例からわかるように、関数型インタフェースを引数として渡すには、まずそのインターフェースを実装し、実装したクラスのインスタンスを作成する必要があります。
- 多くの場合、メソッド自体は数行のコードで済むのですが、クラスの定義や、インスタンス化などの作業でコードが余計に膨大になってしまいます。
- **ラムダ式**[Lambda Expression]は、この処理（インターフェースを実装、インスタンスを生成）を簡潔に実装するための文法です。

# 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 }
```



- 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「->」で繋げます。**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。

# 標準的な関数型インターフェース

- 実際に使用されるインターフェースは大体、1つのパラメータを受け取って1つの値を返すものや、2つのパラメータを受け取って1つの値を返すものなど、いくつかの一般的な形なメソッドした使いません。
- 毎回自分で書くのも面倒くさいので、Java ではこのような汎用的な関数インターフェースが多数用意されています。また、これらのインターフェースは、Java の標準的な API（例えば、Collections API）の一部で使用されています。
- これらの**標準的なインターフェース**は、`java.util.function` パッケージで定義されています:  
 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/function/package-summary.html>

# 標準インターフェース

- 一般的な標準インター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<T>	引数を受け取らず、T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

# 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多くあります。

- 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。

次へ 

 前へ

- `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]
```



# 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。



*Question and answer*



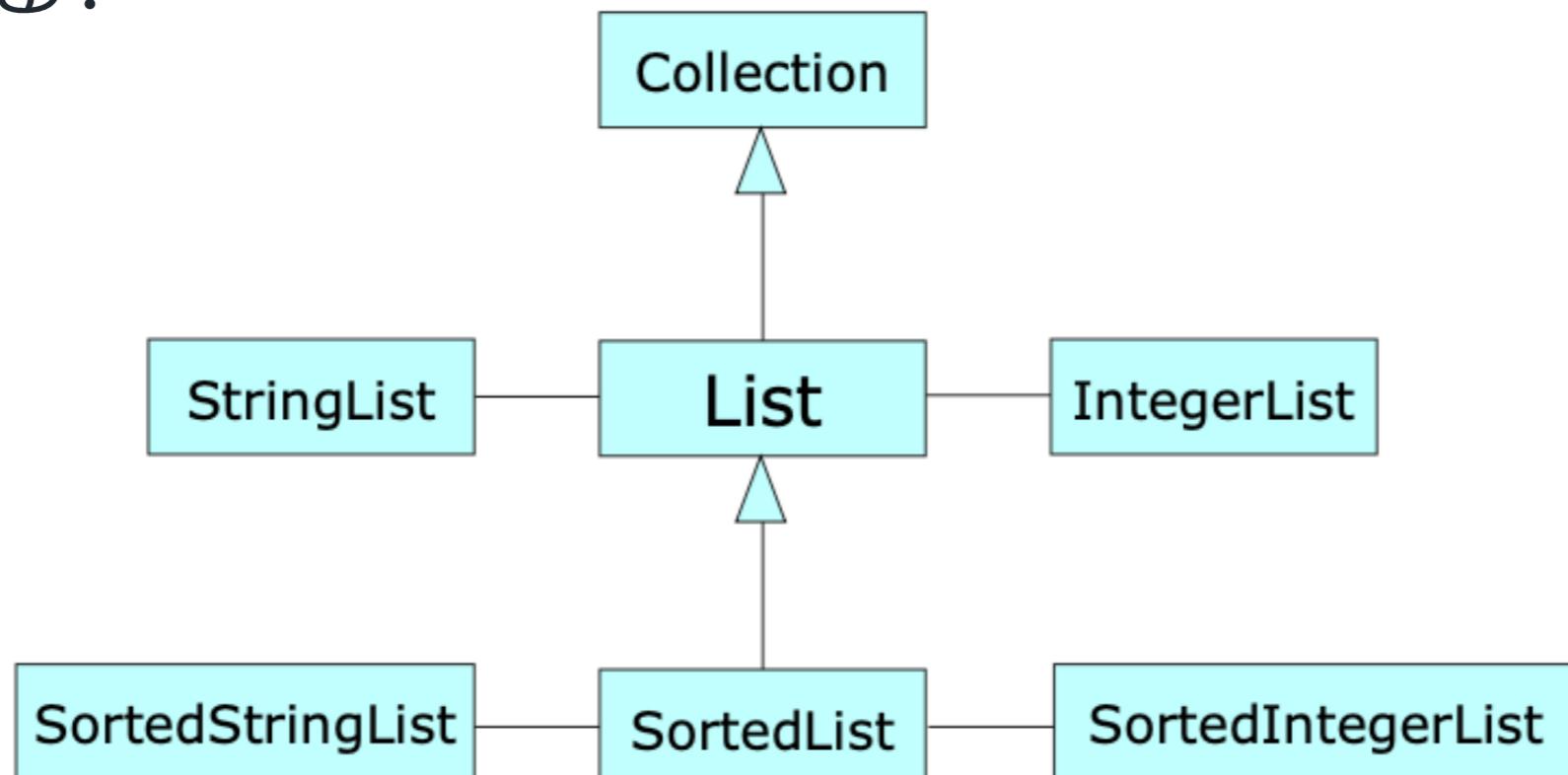
- 1 コレクション API
- 2 関数型インターフェース
- 3 ジェネリクス

# 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?
  - 作成するクラスの数が多くて、開発と保守に不便。
  - 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

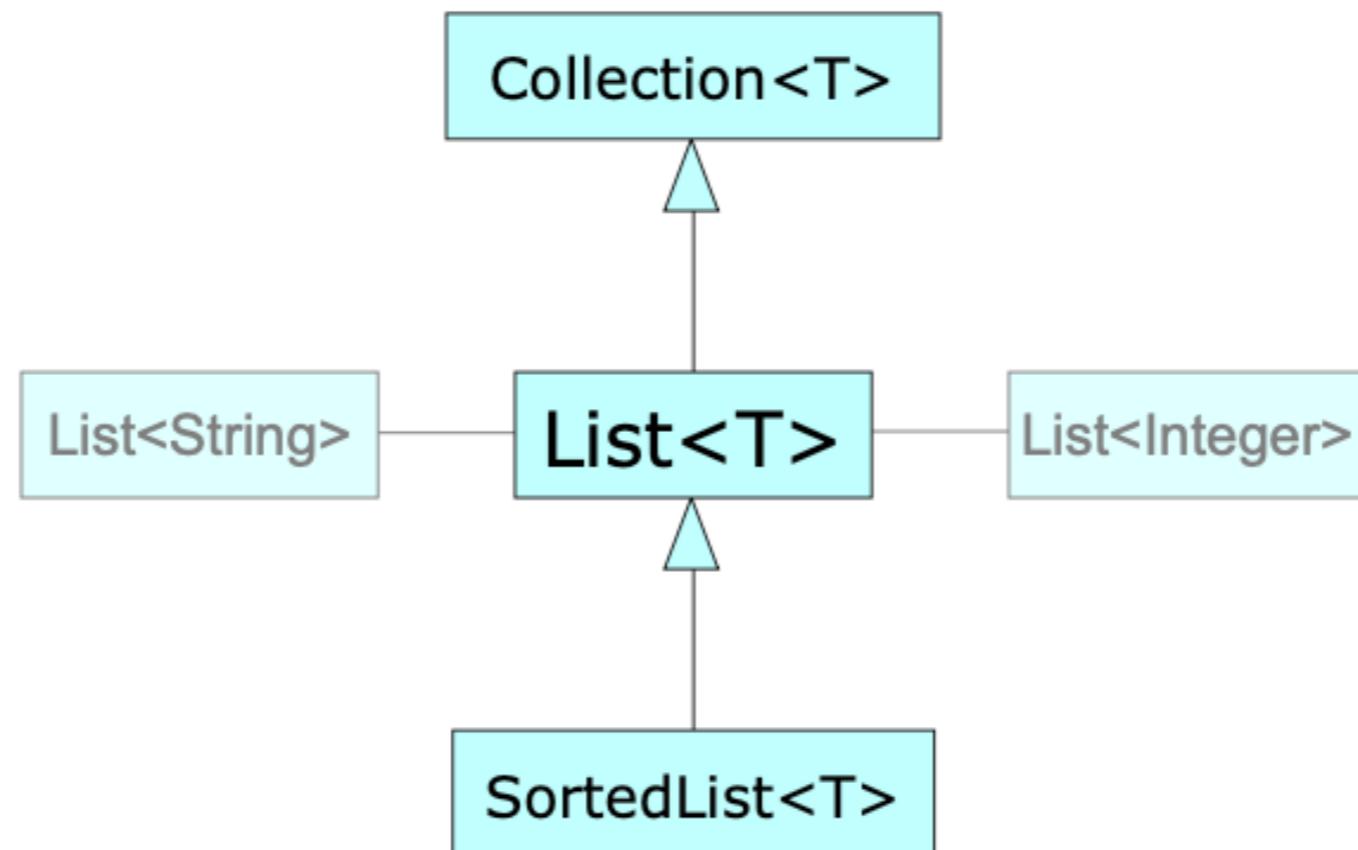
- 全ての参照型は Object のサブクラスであることが分かっているので、任意の型のデータを格納するために、Object のインスタンスを保持する List クラスを書けばいいです：

```
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・インター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「,」で名前を区切れます：

```
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>();
```



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 }
```



- 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。



*Question and answer*

# 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. 関数型インター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。

**THANK YOU!**