

3.2 Java Collections API

- Collections API

- 函数式接口

- 泛型

目录

1

Collections API

2

函数式接口

3

泛型

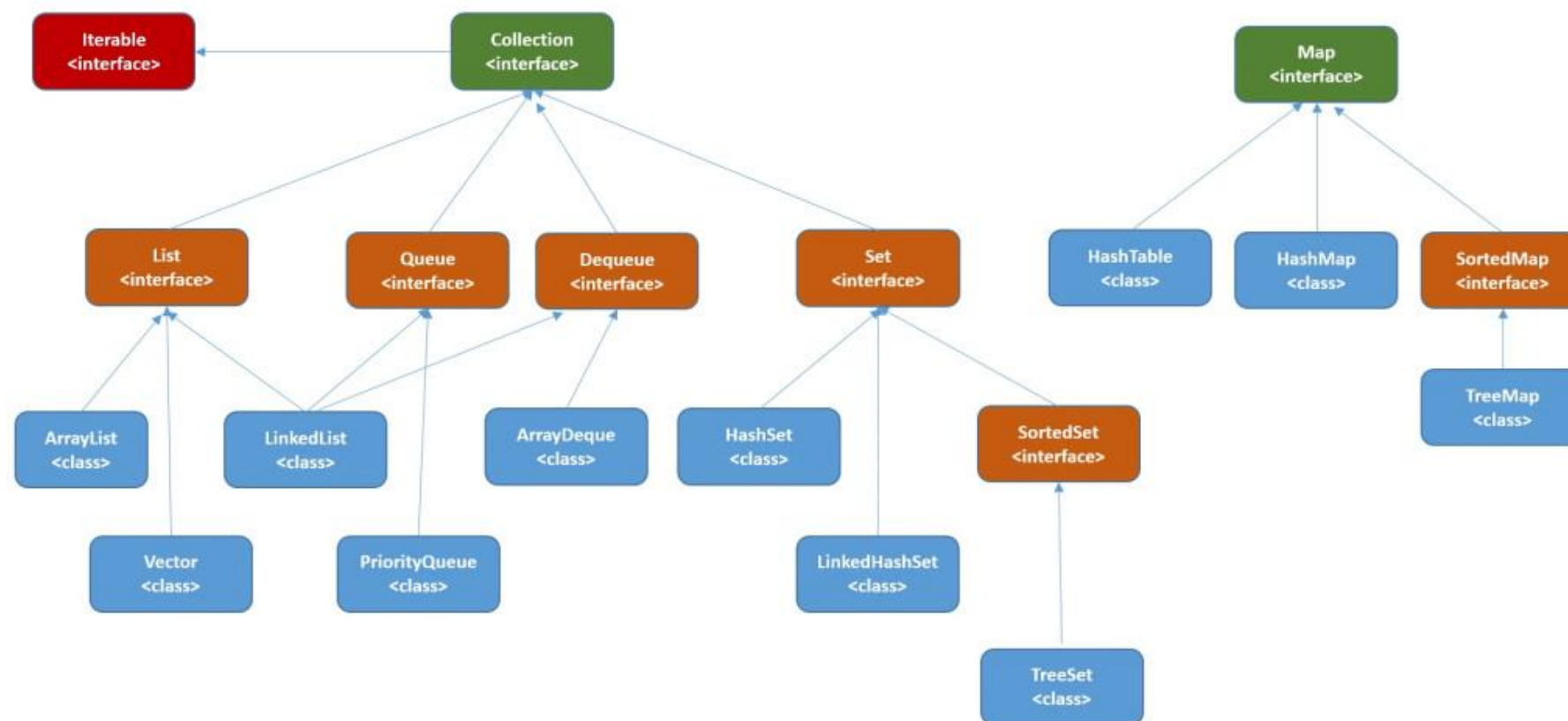
Java Collections API

- 上一节介绍了很多常用的*抽象数据类型*和*数据结构*，但在实际使用的过程中，我们通常不需要自己实现这些结构，而是使用编程语言提供的应用程序接口（API）。
- Java 提供了 **Collections API**，为我们实现了很多抽象类型和数据结构。
- Java 还帮我们实现了一些和这些数据结构有关的算法和复杂操作，比如*排序*等。
- Collections API 提供的类都可以在 `java.util` 包中找到。


Collection 和 Map

- Collections API 的主要功能由以下两个接口和它们的子类提供：**Collection** 和 **Map**。大部分我们之前介绍的抽象类型都实装 **Collection** 接口，而关联数组（Map）则实装单独的 **Map** 接口，因为它需要定义两个数据类型（键和值）。

Collection Framework Hierarchy



List

- **List** 是代表列表的抽象类型。
- List 提供了一些列表的常用操作，包括访问、插入、删除数据等：
 1. **add**(item): 把数据插入列表的末尾。
 2. **add**(i, item): 把数据插入列表的索引 i 的位置。
 3. **remove**(i): 删除列表中索引为 i 的元素。
 4. **remove**(item): 删除列表中的 item（如果它存在）。
 5. **get**(i): 获得索引为 i 的元素。
 6. **set**(i, item): 将索引为 i 的值设为 item。
 7. **size**(): 获得列表的长度。
 - ...
- 所有方法都可以在官方文档中查阅：
 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/List.html>

列表的创建

- **List** 是一个代表列表的抽象类型，在 Java 中被定义为一个接口。因此，我们不能直接创建 List 的实例，只能创建实现它的数据结构类的实例。
- Java 提供了很多列表的实现，比如 ArrayList（基于数组）和 LinkedList（基于链表）。
- 我们可以直接使用这些类的构造器创建列表：

```
List<String> nameList = new ArrayList<String>();
```
- 类名后加上的 **<String>** 代表列表中装的是 String 类型的数据。这是一种被称作泛型的特殊语法，我们会在之后（➡ § 3.2.3）讲解。



创建列表的简单方法

- 我们也可以使用 **List.of** 方法简单地创建一个列表并添加初始值:

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

- 注意: 使用这种方法创建的列表是 *不可变的 (Immutable)* , 对其执行添加或删除操作将会报错:

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

- 可以使用列表类的构造器将其转换成可变的:

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

列表的遍历

- 我们之前介绍过 **for-each** 循环语法，它可以用来简单地遍历一个数组。实际上，**for-each** 循环也可以用来遍历列表：

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


包装类

- 代表泛型的尖括号“<>”中只能使用引用类型。如果想要创建基本类型的列表，需要使用对应的**包装类**[\[ラッパークラス\]](#)。包装类基本上就是基本类型的“引用类型版”。
- 比如，int 类型对应的包装类是 **Integer** 类，因此可以通过以下方法创建一个整数列表：


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

包装类一览

- 下表列出了 Java 中所有基本类型对应的包装类：

基本类型	包装类
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Stack

- **Stack** 是代表 **栈** 的抽象类型。
- Stack 提供了一些栈的常用操作：
 1. **push**(item): 把数据放入栈顶（压栈）。
 2. **pop**(item): 获得并删除栈顶元素（出栈）。
 3. **peek**(item): 获得栈顶元素，但不删除。
 4. **size**(): 获得栈的大小。...
- 所有方法都可以在官方文档中查阅：
 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Stack.html>


栈的创建和使用

- Java 的 Stack 类本身就实现了栈，因此直接使用 Stack 的构造器即可创建一个栈：

```
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]
```



Queue

- **Queue** 是代表 *队列* 的抽象类型。
- Queue 提供了一些队列的常用操作：
 1. **offer**(item): 把数据放入队尾。
 2. **poll**(item): 获得并删除队首元素。
 3. **peek**(item): 获得队首元素，但不删除。
 4. **size**(): 获得队列的大小。...
- 所有方法都可以在官方文档中查阅：
 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Queue.html>


队列的创建和使用

- Java 提供的队列实现包括 LinkedList, ArrayDeque 等:

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



Set

- **Set** 是代表集合的抽象类型。
- Set 提供了一些集合的常用操作：
 1. **add**(item): 把数据加入集合。
 2. **remove**(item): 删除集合中的 item（如果它存在）。
 3. **contains**(item): 判断某个元素是否属于集合。
 4. **size**(): 获得集合的大小。...
- 所有方法都可以在官方文档中查阅：
 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Set.html>


集合的创建和使用

- Java 提供的常见的集合实现包括 HashSet、TreeSet 等。
- 和列表类似，也可以使用 Set.of 方法方便地创建一个集合。
- 集合也可以通过 for-each 语句遍历：

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



Map

- **Map** 是代表 *关联数组* (字典、映射) 的抽象类型。
- Map 提供了一些关联数组的常用操作：
 1. **put**(key, value): 将关联数组中 key 对应的值设定成 value。
 2. **remove**(key): 删除关联数组中指定的键值对。
 3. **get**(key): 获得指定的键所对应的值。
 4. **containsKey**(key): 判断指定键是否在关联数组中。
 5. **containsValue**(value): 判断指定值是否在关联数组中。
 6. **size**(): 获得关联数组的大小。...
- 所有方法都可以在官方文档中查阅：
 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Map.html>

关联数组的创建和使用

- Java 提供的关联数组实现包括 HashMap、TreeMap 等。
- 和其他类不同，关联数组需要标明 2 个类型：键和值的类型：

```
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 }
```



总结：Collections API 的使用

Sum Up

让我们回顾一下数据结构的决定流程，以及实际在 Java 中的使用方法：

1. 根据实际问题，决定保存数据的**抽象数据类型**。
在 Java 中，我们可以从 **Collection** 接口或 **Map** 接口的子接口里寻找合适的抽象类型。
2. 根据对功能的需求，决定使用哪种**数据结构**进行实装。
在 Java 中，我们可以在抽象接口的文档里查看它的所有实装类。
3. **实装数据结构**。
Collections API 实装了大部分常用结构，如果有特殊需求也可以在互联网上搜索第三方代码库。

Q & A

Question and answer

目录

1

Collections API

2

函数式接口

3

泛型

函数式接口

- 在编程过程中，我们经常会用到一种特殊的接口，它只有 1 个需要实现的方法。这种接口被称为**函数式接口**[\[関数型インターフェース\]](#)。
- 比如，如果想使用 Java List 类的 sort 方法，我们需要告诉 Java 列表中数据的比较方式。Java 需要我们输入一个只包含 compare 方法的接口 Comparator。我们需要实际实装这个接口，然后创建一个实例传入 sort 方法。



Lambda 表达式

- 从刚刚这个例子可以看出，要传入函数式接口的参数，我们必须先实装该接口，再创建该实装类的实例。
- 大部分情况下，方法本身可能只包含几行代码，但我们却需要大费周章地把它实现。
- **Lambda 表达式**[ラムダ式]是一种简洁地实现这一流程（包括实现接口和创建实例）的语法。
- 从本质上来说，lambda 表达式就是一种**把方法作为参数传递**的语法。

Lambda 表达式

- Lambda 表达式的基本语法如下：

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

- 其中，一开始的圆括号“**()**”里写入方法的参数列表，**不需要写参数类型**；花括号“**{}**”里写入方法体；用箭头“**->**”链接。**不需要写方法名和返回类型**。
- 这个表达式实际创建了一个**实现了某函数式接口的类的实例（对象）**。你可以把这个对象直接传入需要函数式接口的函数，也可以先把它存在变量里。



Lambda 表达式的简写语法

- 如果只有一个参数，你可以省略圆括号 “()”：

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

- 如果方法体只有一行 return 语句，你可以省略花括号 “{}” 及 return：

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

- 如果方法只有一行且返回值为空，花括号同样可以被省略：

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

Lambda 表达式与匿名内部类

- 我们知道，lambda 表达式实际上是实装了一个接口，并且实例化了这个实装的类。换句话说，我们其实声明了一个新的类，但是没有给他起名字。
- 同时，这还是一个属于它的创建者（外部类）的类。因此，这种类被称为**匿名内部类**[\[匿名内部クラス\]](#)。
- 我们之前（[← § 2.4.5](#)）说过，内部类是可以使用外部类的变量或方法的。因此，我们可以在 lambda 表达式中直接使用外部类的变量或方法。
- 这在某些场景下会很有用，比如创建之后（[➡ § 2.5.1](#)）介绍的线程类。

标准函数接口

- 实际使用的函数式接口不外乎几种常见的形式，比如接受 1 个参数返回 1 个值、接受 2 个参数返回 1 个值等。
- Java 提供了一些通用的函数式接口，我们不需要每次都自己书写。同时，Java 的一些标准 API（如 Collections API）中也会用到这些接口。
- 这些标准接口都被定义在 `java.util.function` 包中：

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

标准接口一览

- 下表列出了一些常见的标准接口：

接口	方法的形式
Function<T, R>	接受 1 个类型 T 的参数，返回类型 R 的值
BiFunction<T, U, R>	接受 2 个类型 T、U 的参数，返回类型 R 的值
Supplier<T>	不接受参数，返回类型 T 的值
Consumer<T>	接受 1 个类型 T 的参数，没有返回值
BiConsumer<T, U>	接受 2 个类型 T、U 的参数，没有返回值
IntFunction<T>	接受 1 个类型 T 的参数，返回 int 类型的值
Predicate<T>	接受 1 个类型 T 的参数，返回 boolean 类型的值

Stream API

- **Stream API** 是 Java 为帮助我们使用 Collections API 中的数据结构提供的一种强大而便捷的方法。
- Stream API 的基本使用流程如下：
 1. 从数据结构（某个 Collection 类的子类）中获得对应的 Stream 对象。
 2. 使用 Stream 类的方法方便地处理数据。
 3. （如果有必要）将 Stream 转换回 Collection。其中，第 2 步涉及到的 Stream 类的方法有很多可以使用 lambda 表达式简洁地表达。
- Stream 类可以简单地理解为一个具有很多功能的特殊的列表，提供了一些对**整个**列表的操作。

获得 Stream 对象

- 对于大部分 Collection 类的子类，如列表、集合等，可以通过它们的 stream 方法简单地获得其 Stream 对象：

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

- 像是 Map 这样的其他数据结构，也可以获得其包含的 Collection 类结构（比如 Map 的 keySet、entrySet 等）来使用 Stream。

使用 Stream 方法


- 有了 Stream 对象后，我们就可以使用 Stream 类提供的方法方便地处理数据。比如，Stream 类提供了 sorted 方法对数据排序，返回的结果将是另一个 Stream，但里面的数据排好了顺序：

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

- 再比如 forEach 方法，它接受一个 Consumer 接口（可以用 lambda 表达式实现），功能是对所有 Stream 中的数据依次执行我们传入的方法。

```
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 方法一览

- 下面列出一些常用的 Stream 方法：
 - **forEach**(consumer): 对每一个元素实行 consumer 方法，没有返回值。
 - **map**(function): 把每一个元素通过 function 方法变换，从而创建一个新的 Stream。
 - **filter**(predicate): 筛选所有满足 predicate 的元素以创建一个新的 Stream。
 - **reduce**(init, operator): 使用累积函数[累積関数]对元素进行运算（比如求和、求积），返回运算的结果。init 指定初始值，operator 指定运算。
 - **max**(comparator): 找出所有元素的最大值。comparator 定义比较大小的方法。
 - **min**(comparator): 找出所有元素的最小值。参数同上。
 - **sorted**(comparator): 对所有元素排序，返回一个新 Stream。参数同上。
 - **distinct**(): 去除重复的元素，返回一个新的 Stream。
- 所有方法的列表可以在官方文档中查阅：
 <https://docs.oracle.com/javase/jp/8/docs/api/java/util/stream/Stream.html>

Stream 的转换

- 某些方法如 `map`、`sorted` 的返回结果仍然是一个 `Stream`。因此，我们可以继续对返回值重复这些方法。如果你最终需要把 `Stream` 转换成其他类型，你可以使用 `toArray` 方法把它转换回数组：

```
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]
```

- 也可以用 `collect` 方法转换回一个 `Collection` 对象（比如列表）：

```
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]
```



方法引用

- 有时我们的 lambda 表达式只是直接使用一个现成的方法，比如：

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

- 这时我们可以使用 Java 的**方法引用**[\[メソッド参照\]](#)语法，更简洁地传递这个方法。要获得方法引用，使用双冒号运算符 “::”：

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

- 双冒号运算符的使用与普通调用方法类似：对于非静态方法，用 “::” 连接对象名和方法名；对于静态方法，用 “::” 连接类名和方法名。

Q & A

Question and answer

目录

1

Collections API

2

函数式接口

3

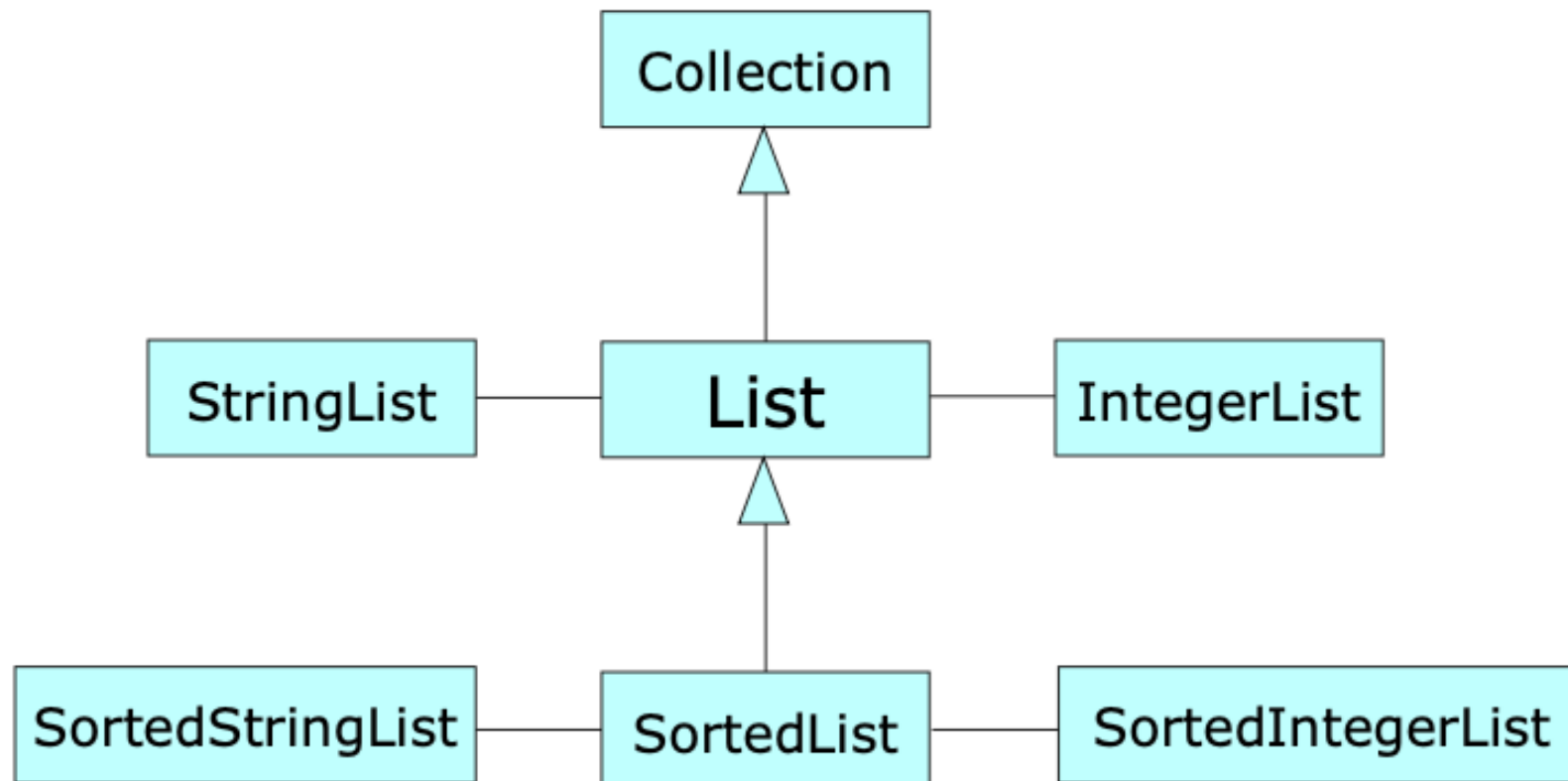
泛型

泛型

- 在学习 Collections API 的过程中，我们已经接触到了**泛型**[ジェネリクス・総称型]使用方法。
- 简单来说，泛型是一种特殊的类，它可以将一个（或多个）类型作为参数。
- 比如，使用 List 需要类型指定一个类型作为参数。我们可以指定 String、Integer 等不同的类型，这样我们就可以使用保存各种**不同类型**数据的列表了。
- 想一想：如果没有泛型，我们该怎么实现这样能够应对**各种数据的类**？

方法 1：制作专用的类

- 我们可以分别对不同类型的数据制作不同的类：



- 这样会带来怎样的问题？
 - 需要制作的类的数量很多，不便开发和维护。
 - 每次新增一种数据类型，都要单独开发对应的类。

方法 2：使用 Object 类

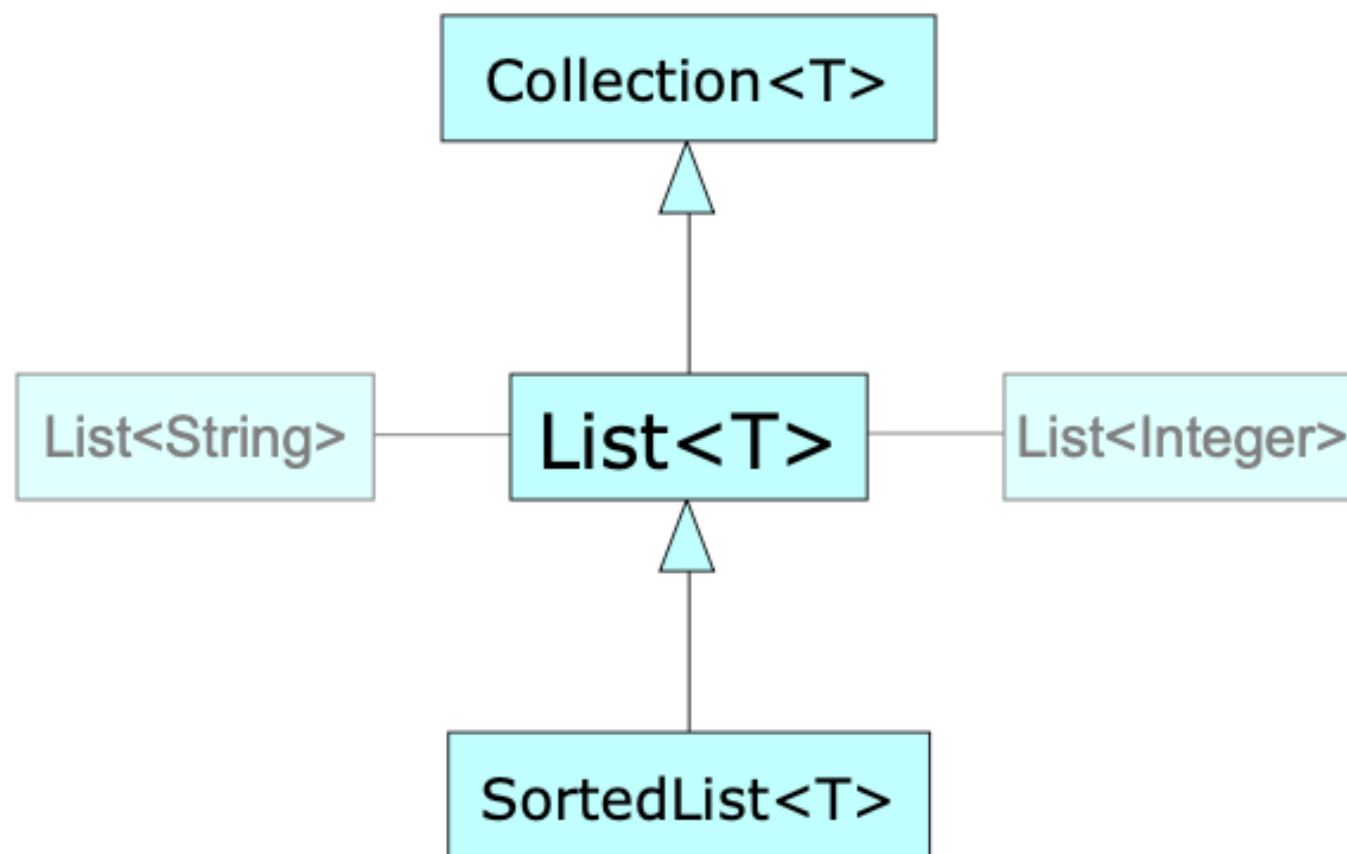
- 我们知道所有引用类型都是 Object 的子类，因此我们可以写一个保存 Object 的 List 类，以存入任意类型的数据：

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

- 这样会带来怎样的问题？
 - 每次取出数据都要进行类型转换才能使用。
 - 同一个 List 里可能会存入不同类型的数据。
 - 可读性降低。

方法 3：泛型

- 有了泛型，我们就可以只写一个 List 类，对任意类型的数据提供一些共通的方法。同时，在继承（或实装）时也只需要编写最小限度的代码。



泛型的声明

- 除了使用 Java 给我们提供的泛型之外，我们也可以创建自己的泛型。要声明泛型，在类（接口）名后加上尖括号“**<>**”，在里面写上作为参数的类型的名称：

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

- 这里的 MyList 类接受一个类型作为参数，并起名叫 T。比如如果用户想创建一个存储 String 的 MyList，就应该把 T 设为 String。
- 如果要接受多个类型，使用逗号“**,**”把它们名称隔开：

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

泛型的使用

- 我们已经在使用 Collections API 时学习过了使用泛型的语法：

```
MyList<String>
```

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

- 包括构造器或其他静态方法的使用：

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

Tips

当 Java 可以判断出参数是什么类型时，可以省略 “<>” 中的内容：

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

泛型中的方法

- 泛型声明中定义的类型参数（前例中的 T、K 等）都可以在类里直接当作一个类型使用。比如，可以把它们作为方法的参数类型或返回值：

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



- 还可以在声明方法时要求用户指定一个类型，这被称为**泛型方法** [ジェネリックメソッド]，此处不作详细介绍。

Q & A

Question and answer

总结

Sum Up

1. Java Collections API:

- ① Collection 和 Map 类;
- ② 抽象数据结构: **List**、Stack、Queue、**Set**、**Map**;
- ③ 上述抽象结构的实装类。

2. 函数式接口和 lambda 表达式:

- ① lambda 表达式的基本语法和简写方法 (方法引用);
- ② Stream API: Stream 方法中 lambda 表达式的使用。

3. 泛型的概念和基本语法。

THANK YOU!