LẬP TRÌNH TỔNG QUÁT

NỘI DUNG TRONG CHƯƠNG

- *Lập trình tổng quát: lớp tổng quát, tạo đối tượng từ lớp tổng quát, phương thức tổng quát, giới hạn kiểu*
- *Cấu trúc dữ liệu trong Java: danh sách, tập hợp, hàng đợi, bản đồ*

Lập trình tổng quát là một kỹ thuật quan trọng, nó phù hợp với các dự án cần viết mã nhiều và có sự tương đồng trong việc xử lý đối tượng, dữ liệu. Lập trình tổng quát giúp giảm số lượng dòng lệnh cần viết và tiết kiệm được thời gian và chi phí bảo trì mã lệnh sau này. Lập trình tổng quát là phương tiện để tạo ra các cấu trúc dữ liệu giúp thao tác với dữ liệu được dễ dàng hơn. Chương lập trình tổng quát này cũng giới thiệu luôn về khung làm việc bộ sưu (Collection Framework) tập mạnh mẽ trong Java bao gồm: Danh sách (List), Tập hợp (Set), và Bản đồ (Map)

GIỚI THIỆU

Có những tình huống khi các phương thức và lớp không phụ thuộc vào kiểu dữ liệu mà chúng hoạt động. Ví dụ, thuật toán tìm một phần tử trong mảng có thể xử lý mảng chuỗi, số nguyên hoặc lớp tùy chỉnh. Mảng lưu trữ nội dung gì không quan trọng: thuật toán luôn giống nhau.

Để mã lệnh viết ra có thể làm việc với nhiều kiểu dữ liệu khác nhau, có hai cách:

- Sử dụng kế thừa
- Lập trình tổng quát

Sử dụng kế thừa để tạo ra những đoạn mã có thể làm việc với nhiều kiểu đối tượng khác nhau là cách lập trình cổ điển, hiện nay không còn sử dụng nhiều.

Tạo một lớp MyList có nội dung như sau:

```
1  private Object[] elementData;
2  MyList() {
3     elementData = new Object[10];
4  }
5  public Object get(int i) {
6     return elementData[i];
7  }
8  public void add(Object o) {
9     //...
10 }
```

Viết phương thức main sử dụng MyList

```
1  public static void main(String[] args) {
2     MyList files = new MyList();
3     files.add("a.txt");
4     String filename = (String) files.get(0);
```

```
5     files.add(new File("a.txt"));
6     files.add("a.txt");
7  }
```

Có hai vấn đề khi sử dụng kế thừa, thứ nhất là mỗi khi hàm get trả về thì **cần thực hiện việc ép kiểu**:

```
1  MyList files = new MyList();
2  String filename = (String) files.get(0);
```

Vấn đề thứ hai là hàm add **không thực hiện việc kiểm tra kiểu**, ví dụ trong trường hợp dưới đây, tham số của hàm có thể là đối tượng File, cũng có thể là chuỗi.

```
1  files.add(new File(". . ."));
2  files.add(". . .");
```

LẬP TRÌNH TỔNG QUÁT

Lớp tổng quát

Lớp tổng quát (Generic class) là lớp làm việc với kiểu dữ liệu tổng quát (kiểu không cần xác định trước)

Sử dụng cú pháp sau để khai báo lớp tổng quát:

```
1  class Tên_Lớp<Kiểu_1, Kiểu_2, Kiểu 3…>{
2  }
```

Trong ví dụ sau, lớp **GenericClass** có một tham số loại duy nhất được đặt tên **T**. Giả định rằng loại **T** là "kiểu gì đó" và viết phần thân của lớp sẽ làm việc với kiểu dữ liệu này:

```
1  class GenericClass<T> {
2     private T t;
3     public GenericClass(T t) {
4         this.t = t;
5     }
6     public T get() {
7         return t;
8     }
9     public T set(T t) {
10        this.t = t;
11        return this.t;
12    }
13 }
```

Sau khi được khai báo, một tham số kiểu có thể được sử dụng bên trong thân lớp như một kiểu thông thường. Ví dụ trên sử dụng tham số kiểu T như: Kiểu cho trường (type for a field), Kiểu đối số phương thức tạo (constructor argument type), Kiểu đối số phương thức hoặc kiểu trả về

Các tên tham số kiểu được sử dụng phổ biến nhất là:

- **T** – Kiểu chung.

- **E** – Phần tử (được sử dụng rộng rãi bởi các bộ sưu tập khác nhau).
- **K** – Khóa.
- **V** – Giá trị.
- **N** – Con số.
- **S**, **U**, **V** ... – Các kiểu khác.

Có một quy ước đặt tên hạn chế các lựa chọn tên tham số kiểu đối với các chữ cái viết hoa đơn. Nếu không theo quy ước này, sẽ rất khó để phân biệt giữa biến kiểu và tên lớp thông thường.

Tạo đối tượng từ lớp tổng quát

Để tạo một đối tượng từ một lớp tổng quát, cần chỉ định "đối số kiểu":

```
GenericClass<Integer> obj1 = new GenericClass<>(10);
GenericClass<String> obj2 = new GenericClass<>("abc");
```

Điều quan trọng cần lưu ý là "đối số kiểu" phải là kiểu tham chiếu. Các kiểu dữ liệu cơ sở như **int** hoặc **double** là các đối số kiểu không hợp lệ. Cặp dấu ngoặc nhọn <> được gọi một cách không chính thức là Toán tử kim cương (diamond operator). Đôi khi, việc khai báo một biến với kiểu chung có thể dài dòng và khó đọc. Bắt đầu từ Java 10, có thể viết var thay vì một kiểu cụ thể, khi đó kiểu sẽ tự động được xác định dựa trên kiểu giá trị được gán.

```
var obj3 = new GenericClass<>("abc");
```

Sau khi đã tạo một đối tượng với đối số kiểu được chỉ định, có thể gọi các phương thức của lớp nhận hoặc trả về tham số kiểu:

```
Integer number = obj1.get();
String string = obj2.get();
System.out.println(obj1.set(20));
System.out.println(obj2.set("def"));
```

Nếu một lớp có nhiều tham số kiểu, cần chỉ định tất cả chúng khi tạo các đối tượng:

```
GenericClass<Type1, Type2, ..., TypeN> obj = new GenericClass<>(...);
```

PHƯƠNG THỨC TỔNG QUÁT

Một **phương thức tổng quát (generic method)** có thể nằm bên trong một lớp tổng quát, hoặc một lớp thường. Ví dụ viết phương thức **main()** trong một lớp thường và khai báo ba mảng:

```
public static void main(String[] args) {
    Double[] doubles = {1.2, 3.5, 2.5, 3.0, 6.0, 2.2};
    Integer[] integers = {1, 0, -4, 3, 2, 3, 4, -1, 6, 2, 4};
    String[] names = {"Minh", "Thu", "Tuấn Anh"};
}
```

Xây dựng 1 phương thức tổng quát có tên là **getMiddle():**

```
1  public static <E> E getMiddle(E[] a) {
2     return a[a.length / 2];
3  }
```

Sử dụng phương thức tổng quát vừa tạo:

```
1  public static void main(String[] args) {
2     //..
3     System.out.println(getMiddle(doubles));
4     System.out.println(getMiddle(integers));
5     System.out.println(getMiddle(names));
6  }
```

Kiểm tra kết quả in ra màn hình:

```
1  3.0
2  3
3  Thu
```

GIỚI HẠN KIỂU

Sử dụng cú pháp **<T extends A>** để giới hạn kiểu T phải là lớp con của A. Ví dụ:

```
1  public static <T extends Number> T getSum(T[] a) {
2      Double sum = 0.0;
3      for (T num : a) {
4              sum = sum + num.doubleValue() ;
5      }
6      return (T) sum;
7  }
```

Sử dụng cú pháp **<T extends B & C>** để giới hạn kiểu T phải là lớp triển khai từ 2 giao diện B và C.

Hàm trên sẽ báo lỗi ở dòng **num.doubleValue().** Sửa lại bằng cách bổ sung extends Number.

```
1  public static <T extends Number> T getSum(T[] a) {
2     Double sum = 0.0;
3     for (T num : a) {
4         sum = sum + num.doubleValue();
5     }
6     return (T) sum;
7  }
```

Sử dụng phương thức nói trên:

```
1  public static void main(String[] args) {
2     Double[] doubles = {1.2, 3.5, 2.5, 3.0, 6.0, 2.2};
3     Integer[] integers = {1, 0, -4, 3, 2, 3, 4, -1, 6, 2, 4};
4     String[] names = {"Peter", "David", "Harry"};
5     System.out.println(getSum(doubles));
```

```
6     System.out.println(getSum(integers));
7  }
```

Chạy chương trình và kiểm tra kết quả hiển thị ra màn hình:

```
1  18.4
2  20.0
```

Sử dụng cú pháp **<T extends B & C>** để giới hạn kiểu T phải là lớp triển khai từ hai giao diện B và C.

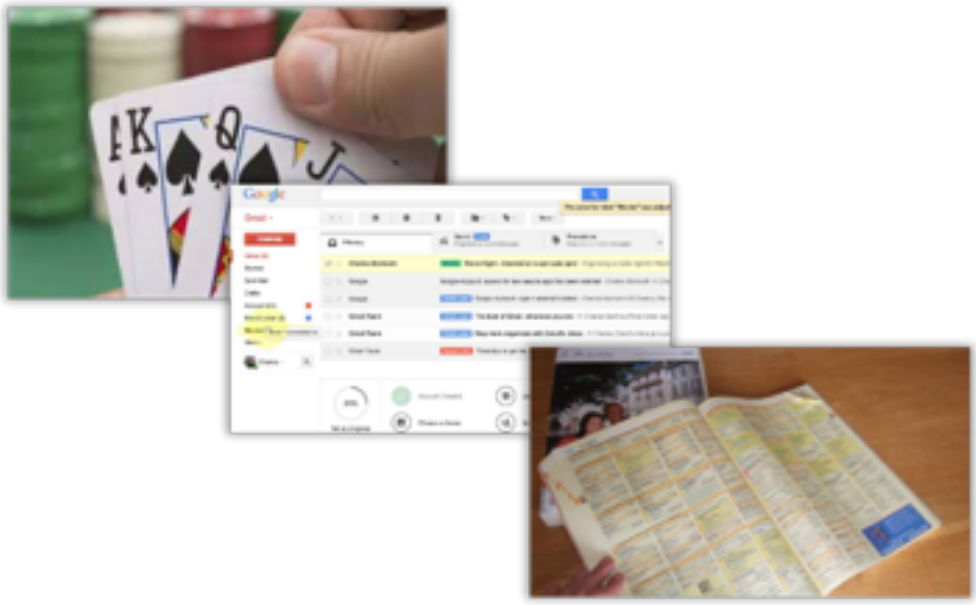
```
<T extends Comparable & Serializable>
```

COLLECTIONS FRAMEWORK

Collections là gì

Một collection là một đối tượng được sử dụng để chứa hoặc nhóm các đối tượng khác (thường là cùng kiểu).

- Một bộ bài
- Một thư mục email
- Một danh bạ điện thoại



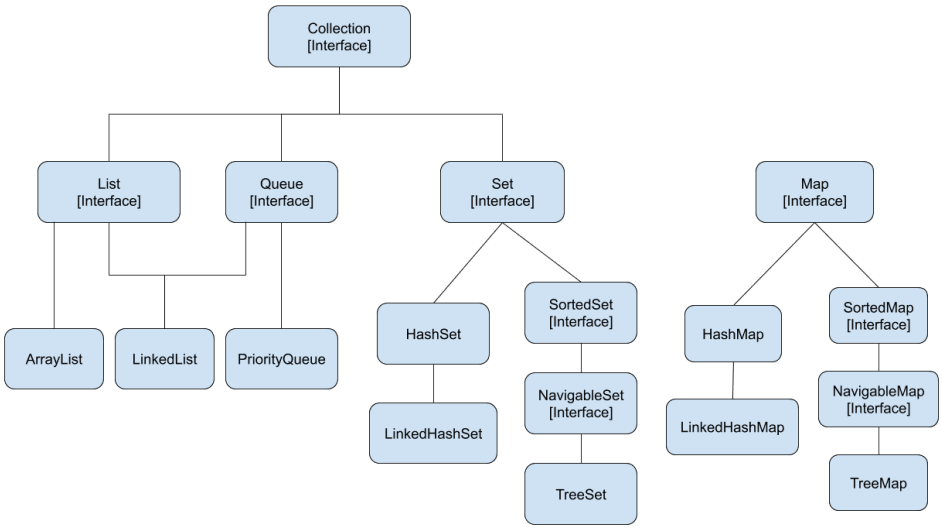
Hình 6-1 Ví dụ minh họa về các tập hợp

Cấu trúc dữ liệu

Java cung cấp **khung làm việc với các tập hợp** (**collections framework)** bao gồm các lớp và giao diện cho các cấu trúc dữ liệu thường được sử dụng lại như:

- Danh sách (List)
- Tập hợp (Set)

- Hằng đợi (Queue)
- Bản đồ (Map)
- ...



Hình 6-2 Các lớp chính khung làm việc Collections

Giao diện **Collection<E>** đại diện cho một tập hợp trừu tượng, là một vùng chứa các đối tượng cùng loại. Nó cung cấp một số phương thức chung cho tất cả các loại bộ sưu tập khác. Các giao diện **List<E>**, **Set<E>**, **Queue<E>**, **SortedSet<E>**, và **Deque<E>** đại diện cho các loại tập hợp khác nhau. Không thể trực tiếp tạo một đối tượng của chúng vì chúng chỉ là giao diện mà chỉ có thể tạo đối tượng từ những lớp con của chúng. Ví dụ, lớp **ArrayList**, đại diện cho một mảng có thể thay đổi kích thước, là đại diện chính của giao diện **List<E>**. Một giao diện khác là **Map<K,V>** đại diện cho một map (bản đồ) để lưu trữ các cặp khóa-giá trị trong đó **K** là loại chìa khóa và **V** là loại được lưu trữ giá trị. Trong thế giới thực, một ví dụ điển hình về map là danh bạ điện thoại trong đó các khoá là tên và các giá trị là điện thoại của họ. Các **Map<K,V>** giao diện không phải là một loại phụ của **Collection** giao diện, nhưng **map** thường được coi là tập hợp vì chúng là một phần của collection framework và có các phương pháp tương tự.

DANH SÁCH (LIST)

Danh sách là gì

Danh sách (List) là kiểu gần nhất với mảng, ngoại trừ kích thước của chúng có thể được thay đổi động trong khi kích thước của mảng bị hạn chế. Hơn nữa, danh sách cung cấp hành vi nâng cao hơn so với mảng.

Danh sách cung cấp các phương thức:

- **E set(int index, E element)** thay thế phần tử ở vị trí được chỉ định trong danh sách này bằng phần tử được chỉ định và trả về phần tử đã được thay thế;
- **E get(int index)** trả về phần tử ở vị trí đã chỉ định trong danh sách;
- **int indexOf(Object obj)** trả về chỉ mục của lần xuất hiện đầu tiên của phần tử trong danh sách hoặc **-1** nếu không có chỉ số này;

- **int lastIndexOf(Object obj)** trả về chỉ mục của lần xuất hiện cuối cùng của phần tử trong danh sách hoặc -1 nếu không có chỉ số này;
- **List<E> subList(int fromIndex, int toIndex)** trả về một danh sách phụ của danh sách này từ **fromIndex** (bao gồm) đến **toIndex** (loại trừ).

Không thể tạo một thể hiện của **List** nhưng có thể tạo một thể hiện của một trong các cách triển khai của nó: **ArrayList** hoặc **LinkedList** hoặc một danh sách bất biến và sau đó sử dụng nó thông qua giao diện **List**. Làm việc với danh sách thông qua giao diện **List** được coi là thực hành tốt trong lập trình vì mã của sẽ không phụ thuộc vào các cơ chế bên trong của một triển khai cụ thể.

Cách đơn giản nhất để tạo một danh sách là gọi **of** phương pháp của **List**.

```
List<String> emptyList = List.of(); // 0 phần tử
List<String> names = List.of("Tuấn Anh", "Dương Minh", "Thu Hương"); // 3 phần tử
List<Integer> numbers = List.of(0, 1, 1, 2, 3, 5, 8, 13);  // 8 phần tử
```

Nó trả về một **danh sách bất biến** có chứa tất cả các phần tử đã truyền hoặc một danh sách trống. Sử dụng phương pháp này thuận tiện khi tạo hằng số danh sách hoặc kiểm tra một số mã.

```
List<String> daysOfWeek = List.of(
            "Monday",
            "Tuesday",
            "Wednesday",
            "Thursday",
            "Friday",
            "Saturday",
            "Sunday"
);
System.out.println(daysOfWeek.size()); // 7
System.out.println(daysOfWeek.get(1)); // Tuesday
System.out.println(daysOfWeek.indexOf("Sunday")); // 6
List<String> weekDays = daysOfWeek.subList(0, 5);
System.out.println(weekDays); // [Monday, Tuesday, Wednesday, Thursday, Friday]
```

Vì nó là bất biến nên chỉ những phương thức không thay đổi các phần tử trong danh sách mới hoạt động. Những phương thức khác sẽ ném ra một ngoại lệ.

```
daysOfWeek.set(0, "Funday"); // throws UnsupportedOperationException
daysOfWeek.add("Holiday");   // throws UnsupportedOperationException
```

Một cách khác để tạo là sử dụng phương thức asList

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```

Duyệt danh sách

Có ba cách để duyệt danh sách. Xét ví dụ danh sách sau:

```
List<String> names = List.of("Tuấn Anh", "Dương Minh", "Thu Hương");
```

Sử dụng vòng lặp "for i"

```
for (int i = 0; i < names.size(); i += 2) {
    System.out.println(names.get(i));
}
```

Sử dụng vòng lặp "for-each"

```
for (String name : names) {
    System.out.println(name);
}
```

Sử dụng phương thức forEach

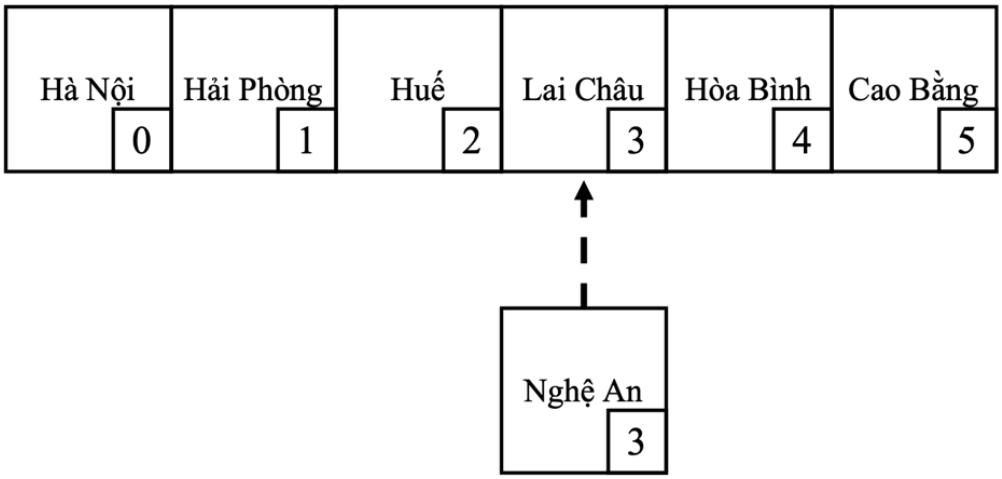
```
names.forEach(System.out::println);
```

Danh sách mảng ArrayList

Khi cần sử dụng danh sách có thể thay đổi, có thể sử dụng một trong hai cách triển khai có thể thay đổi thường được sử dụng của giao diện List.

```
List<Integer> numbers = new ArrayList<>();
numbers.add(15);
numbers.add(10);
numbers.add(20);
System.out.println(numbers); // [15, 10, 20]
numbers.set(0, 30);
System.out.println(numbers); // [30, 10, 20]
```

Logic của một **ArrayList** rất đơn giản: bởi vì bên trong nó là một mảng, các hoạt động **get(int index)** và **set(int index, E element)** sẽ nhanh chóng. Nói cách khác, độ phức tạp về thời gian để truy cập bởi một chỉ mục là O(1) . Nếu sử dụng hoạt động **add(E e)**, một phần tử mới sẽ được thêm vào cuối ArrayList. Điều này cũng sẽ nhanh chóng và cần thời gian cố định để thực hiện, vì vậy độ phức tạp sẽ như nhau: O(1) . Nhưng nếu muốn chèn một phần tử vào **ArrayList** bằng cách sử dụng **add(int index, E element)** hoạt động tình hình sẽ khác nhau. Để đưa phần tử mới vào một chỉ mục cụ thể, **ArrayList** cần di chuyển tất cả các phần tử tiếp theo.



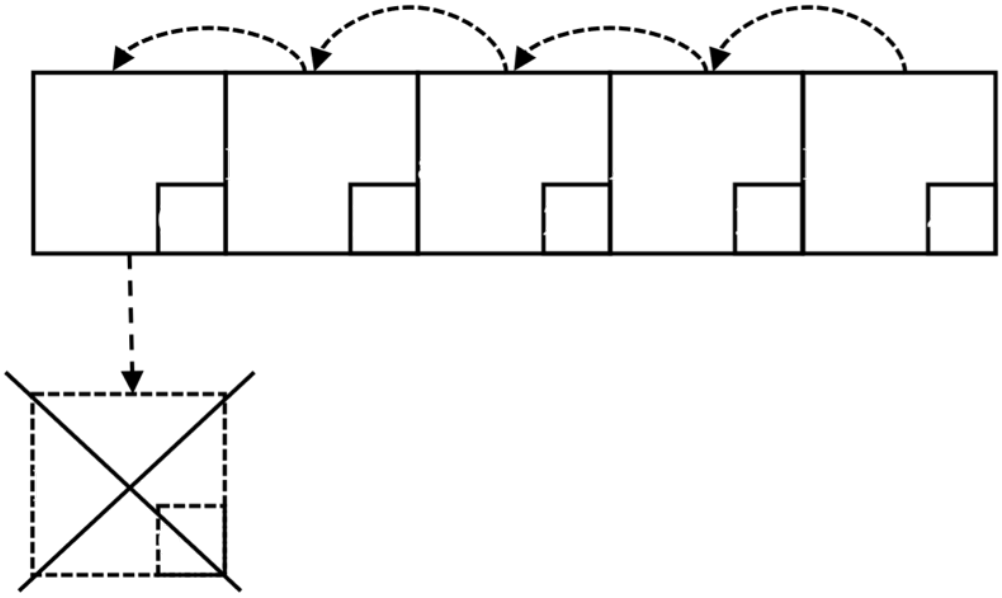
Hình 6-3 ArrayList lưu trữ các phần tử theo dạng mảng

Trong ví dụ, sau khi chèn, phần tử **"Array"** sẽ có chỉ số là 4 và phần tử **"List!"** sẽ có chỉ số là 5. Nếu có nhiều phần tử phía sau thì thao tác này sẽ khá lâu với độ phức tạp về thời gian là O(n).

Khi muốn thêm một phần tử mới vào một ArrayList đầy (full), trước hết, một mảng mới có kích thước lớn hơn sẽ được tạo. Sau đó, tất cả các phần tử hiện có sẽ được sao chép vào mảng mới đó. Và chỉ khi đó phần tử mới sẽ được thêm vào. Vì vậy, độ phức tạp thời gian tồi tệ nhất sẽ là O(n).

Một **ArrayList** là một mảng có thể thay đổi kích thước, vì vậy khi lấp đầy kích thước ban đầu của nó, nó sẽ trở nên lớn hơn và điều đó sẽ xảy ra lặp đi lặp lại.

Hoạt động cuối cùng là **remove(int index)**. Khi muốn xóa một phần tử, tất cả các phần tử tiếp theo sẽ phải được di chuyển.

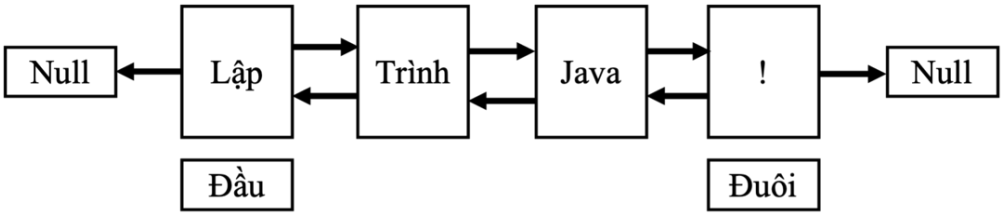


Hình 6-4 Dịch các phần tử trong ArrayList khi xóa một phần tử

Do việc di chuyển tất cả các phần tử tiếp theo, thao tác này sẽ không hiệu quả lắm đối với một danh sách dài, với độ phức tạp về thời gian là O(n).

Danh sách liên kết LinkedList

Một cách triển khai có thể thay đổi khác của giao diện **List** là lớp **LinkedList**. Đại diện cho một danh sách được liên kết kép dựa trên các nút được kết nối. Tất cả các thao tác lập chỉ mục vào danh sách sẽ duyệt qua danh sách từ đầu hoặc từ cuối, tùy theo điều kiện nào gần với chỉ mục được chỉ định.



Hình 6-5 LinkedList tổ chức dữ liệu theo danh sách liên kết

LinkedList không hỗ trợ truy cập nhanh theo chỉ mục. Nhưng nó có thể tiếp cận một phần tử bằng chỉ mục của nó. Để làm điều này LinkedList quyết định đầu hay đuôi gần chỉ mục đó hơn. Và sau đó đi từ đầu hoặc đuôi, LinkedList đi qua tất cả các phần tử trước khi nó đến phần tử cần thiết.

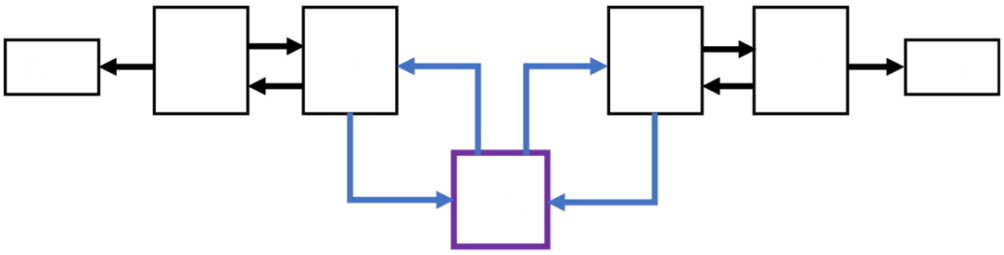
Để có được phần tử "Trình" với chỉ số 1, LinkedList sẽ bắt đầu từ đầu. Trong ví, LinkedList sẽ chỉ cần đi qua một liên kết: giữa phần tử "Lập" và phần tử "Java". Nhưng nếu có một LinkedList dài và chỉ mục yêu cầu nằm ở giữa, các hoạt động này get(int index) và set(int index, E element) sẽ mất rất nhiều thời gian. Vì vậy, trong một tình huống xấu, độ phức tạp về thời gian sẽ là O(n) .

```
1  List<Integer> numbers = new LinkedList<>();
2  numbers.add(10);
3  numbers.add(20);
4  numbers.add(30);
5  System.out.println(numbers); // [10, 20, 30]
```

Thêm một phần tử vào cuối **LinkedList** là một hoạt động nhanh chóng. **LinkedList** sẽ kết nối phần tử mới với phần tử cuối cùng và sẽ làm cho phần tử này thành cái đuôi mới của danh sách. Vì vậy, nó luôn mất thời gian cố định và sự phức tạp của **add(E e)** là O(1).

Bên cạnh phương thức **add(E e)** có hai phương thức khác: **addFirst(E e)** và **addLast(E e)**. Các phương thức này thêm phần tử vào phần đầu và phần đuôi của **LinkedList** tương ứng. Cơ chế bên trong của chúng hoàn toàn giống với phương thức **add(E e)**. Và độ phức tạp của chúng cũng là O(1) . Phương thức **addLast(E e)** là tương đương với phương thức **add(E e)**. Sự khác biệt là phương thức **add(E e)** trả lại **boolean** và phương thức **addLast(E e)** trả lại **void**.

Để thêm một phần tử **LinkedList** cần đạt được vị trí cần thiết và sau đó chỉ thay đổi các liên kết của các phần tử lân cận mới.



Hình 6-6 Thêm một phần tử vào giữa danh sách

Kết nối phần tử mới với phần tử trước đó và phần tử tiếp theo, và việc chèn được thực hiện. Không cần phải di chuyển các phần tử như trong **ArrayList**.

Nếu thêm các phần tử mới gần phần đầu hoặc phần đuôi thì độ phức tạp về thời gian của nó sẽ là O(1) . Trong hầu hết các tình huống, đó là một hoạt động nhanh. Nhưng nếu danh sách là một danh sách rất dài thì việc tiếp cận phần tử ở giữa sẽ không nhanh như vậy. Vì vậy, trong một tình huống xấu, độ phức tạp về thời gian sẽ là O(n).

Phương thức **remove(int index)** có cơ chế tương tự như phương thức add (thêm). Để xóa phần tử khỏi giữa một **LinkedList** độ phức tạp thời gian sẽ là O(n). **LinkedList** có hai phương pháp hữu ích khác: **removeFirst()** và **removeLast()**. Đối với chúng, **LinkedList** không cần phải duyệt qua các phần tử. Nó chỉ loại bỏ phần tử đầu tiên hoặc cuối cùng và thay đổi phần đầu hoặc phần đuôi tương ứng. Độ phức tạp về thời gian cho cả hai hoạt động là O(1) .

So sánh ArrayList vs LinkedList

Sự khác biệt giữa ArrrayList và LinkedList là ở cách lưu trữ tổ chức các phần tử:

- Một **ArrayList** là một mảng các đối tượng có thể thay đổi kích thước, trong đó mọi phần tử đều có một chỉ mục.
- Một **LinkedList** là một danh sách được liên kết kép dựa trên các nút được kết nối. LinkedList lưu trữ đầu và đuôi của nó.

Bảng so sánh:

Bảng 6-1 So sánh tốc độ của ArrayList và LinkedList

	ArrayList	LinkedList
get(int index)	0(1)	0(n)
set(int index, E element)	0(1)	0(n)
add(E e)	0(1)	0(1)
add((int index, E element)	0(1) + 0(n)	0(n) + 0(1)
Remove(int index)	0(1) + 0(n)	0(n) + 0(1)

ArrayList phổ biến hơn, nhưng trên thực tế, mọi thứ phụ thuộc vào nhiệm vụ đang thực hiện. Đối với tất cả các hoạt động, không thể nói trước lớp nào sẽ hoạt động nhanh hơn. Nó phụ thuộc vào kích thước danh sách là bao nhiêu và đang làm việc với phần nào của danh sách. Điều quan trọng nhất là cần phải hiểu là cách các hoạt động của từng loại.

TẬP HỢP (SET)

Set là gì

Set là tập hợp các phần tử không trùng nhau. Một Set khác đáng kể so với một mảng hoặc một danh sách vì không thể lấy một phần tử theo chỉ mục.

Khi chỉ cần giữ lại các phần tử duy nhất trong một tập hợp, để loại bỏ các phần tử trùng lặp trong một chuỗi hoặc nếu định thực hiện một số phép toán, có thể sử dụng một **Set**. **Set** kế thừa tất cả các phương thức từ giao diện **Collection<E>** giao diện nhưng không thêm cái mới nào. Các phương pháp được sử dụng rộng rãi nhất bao gồm **contains**, **add**, **addAll**, **remove**, **removeAll**, **size**.

Các phương thức **add** và **addAll** chỉ thêm phần tử vào tập hợp nếu những phần tử đó chưa có trong **Set**. Một **Set** luôn chỉ chứa các phần tử duy nhất. Một phương thức đáng được chú ý đặc biệt khi nói về giao diện **Set<E>**, vì nó thường được sử dụng với các tập hợp: **retainAll(Collection<E> coll)**. Nó chỉ giữ lại những phần tử được chứa trong bộ sưu tập được chỉ định. Để bắt đầu sử dụng một tập hợp, cần khởi tạo một trong các cách triển khai của nó: **HashSet**, **TreeSet**, và **LinkedHashSet**. Đây là các tập hợp có thể thay đổi và chúng sử dụng các quy tắc khác nhau để sắp xếp các phần tử và có một số phương pháp bổ sung. Chúng cũng được tối ưu hóa cho các loại hoạt động khác nhau. Cũng có những tập hợp **bất biến**, mà tên của chúng không quan trọng đối với người lập trình. Họ cũng triển khai giao diện **Set<E>**.

Cách đơn giản nhất để tạo một tập hợp là gọi **of** phương pháp của **Set** giao diện. Nó trả về một **Set** bất biến có chứa tất cả các phần tử được truyền vào hoặc một tập hợp rỗng.

```
Set<String> emptySet = Set.of();
Set<String> persons = Set.of("Larry", "Kenny", "Sabrina");
Set<Integer> numbers = Set.of(100, 200, 300, 400);
```

Thứ tự của các phần tử của tập hợp bất biến không cố định:

```
System.out.println(emptySet); // []
System.out.println(persons);  // [Kenny, Larry, Sabrina]
System.out.println(numbers);  // [400, 200, 300, 100] or another order
```

Một trong những hoạt động được sử dụng rộng rãi nhất là kiểm tra xem một tập hợp có chứa một phần tử hay không. Đây là một ví dụ:

```
System.out.println(emptySet.contains("hello")); // false
System.out.println(persons.contains("Sabrina")); // true
System.out.println(persons.contains("John")); // false
System.out.println(numbers.contains(300)); // true
```

Đối với bất biến các tập hợp , chỉ có thể gọi **contains**, **size**, và **isEmpty** các phương pháp. Tất cả những người khác sẽ ném ra ngoại lệ **UnsupportedOperationException** vì họ cố gắng thay đổi tập hợp.

Tiếp theo, chúng ta sẽ xem xét ba cách triển khai có thể thay đổi chính của giao diện **Set**.

Tập hợp băm - HashSet

Lớp **HashSet** đại diện cho một tập hợp được hỗ trợ bởi một bảng băm (Hash table). Nó sử dụng mã băm của các phần tử để lưu trữ chúng một cách hiệu quả.

Ví dụ sau minh họa việc tạo **HashSet** và thêm các quốc gia vào nó (với một bản sao). Kết quả đầu ra không chứa trùng lặp.

```
Set<String> countries = new HashSet<>();
countries.add("India");
countries.add("Japan");
countries.add("Switzerland");
countries.add("Japan");
countries.add("Brazil");
System.out.println(countries);
System.out.println(countries.contains("Switzerland"));
```

Mặc dù về mặt kỹ thuật, thứ tự của **HashSet** phần nào được xác định bởi HashCode. **HashSet** nên được coi như một tập hợp không có thứ tự. Các **HashSet** cung cấp thời gian không đổi O(1) hiệu suất cho các hoạt động cơ bản (**add**, **remove**, và **contains**), giả sử phương thức băm phân tán các phần tử đúng cách giữa các nhóm.Trong thực tế, các tập hợp thường được sử dụng để kiểm tra xem một số phần tử có thuộc về chúng hay không. **HashSet** đặc biệt được khuyến khích cho những trường hợp như vậy vì nó **contains** hoạt động được tối ưu hóa cao.

Tập hợp cây - TreeSet

Lớp **TreeSet** đại diện cho một tập hợp trong đó có sắp xếp các phần tử. Các **TreeSet** triển khai giao diện **SortedSet** mở rộng giao diện cơ sở **Set**. Các giao diện **SortedSet** cung cấp một số phương pháp

mới liên quan đến so sánh các phần tử:

- **Comparator<? super E> comparator()** trả về bộ so sánh được sử dụng để sắp xếp các phần tử trong tập hợp hoặc **null** nếu tập hợp sử dụng thứ tự tự nhiên của các phần tử của nó;
- **SortedSet<E> headSet(E toElement)** trả về một tập hợp con chứa các phần tử nhỏ hơn **toElement**;
- **SortedSet<E> tailSet(E fromElement)** trả về một tập hợp con chứa các phần tử lớn hơn hoặc bằng **fromElement**;
- **SortedSet<E> subSet(E fromElement, E toElement)** trả về một tập hợp con chứa các phần tử trong phạm vi **fromElement**(bao gồm) **toElement**(loại trừ);
- **E first()** trả về phần tử đầu tiên (thấp nhất) trong tập hợp;
- **E last()** trả về phần tử cuối cùng (cao nhất) trong tập hợp.

Ví dụ sau minh họa một số phương thức:

```
SortedSet<Integer> sortedSet = new TreeSet<>();
sortedSet.add(10);
sortedSet.add(15);
sortedSet.add(13);
sortedSet.add(21);
sortedSet.add(17);
System.out.println(sortedSet); // [10, 13, 15, 17, 21]
System.out.println(sortedSet.headSet(15)); // [10, 13]
System.out.println(sortedSet.tailSet(15)); // [15, 17, 21]
System.out.println(sortedSet.subSet(13,17)); // [13, 15]
System.out.println(sortedSet.first()); // minimum is 10
System.out.println(sortedSet.last());  // maximum is 21
```

HashSet nhanh hơn nhiều so với **TreeSet**: thời gian không đổi so với thời gian log (log-time) cho hầu hết các hoạt động, nó không cung cấp đảm bảo thứ tự đặt hàng. Nếu cần sử dụng các thao tác từ giao diện **SortedSet** hoặc nếu bắt buộc phải lặp lại theo thứ tự giá trị, hãy sử dụng **TreeSet**, nếu không thì, **HashSet** sẽ là một lựa chọn tốt hơn.

BẢN ĐỒ (MAP)

Một **bản đồ (map)** là một tập hợp các cặp khóa-giá trị . Các khóa luôn là duy nhất trong khi các giá trị có thể lặp lại.

Hãy xem ví dụ về danh bạ điện thoại:

```
Keys  : Values
----------------------
Bob   : +1-202-555-0118
James : +1-202-555-0220
Katy  : +1-202-555-0175
```

Map có một số điểm tương đồng với tập hợp và danh sách;

- Các khóa của Map tạo thành một Set, nhưng mỗi khóa có một giá trị liên kết;
- Các khóa của một map tương tự như các chỉ mục của một mảng, nhưng các khóa có thể có bất kỳ loại nào bao gồm số nguyên, chuỗi, v.v.

Giao diện khai báo rất nhiều phương thức để làm việc với **Map**. Một số phương thức tương tự như phương pháp của **Collection**, trong khi những thứ khác là duy nhất đối với Map:

Các phương thức giống như bộ sưu tập:

- **int size()** trả về số phần tử trong Map;
- **boolean isEmpty()** trả lại **true** nếu Map không chứa các phần tử và **false** nếu không thì;
- **void clear()** xóa tất cả các yếu tố khỏi Map.

Tôi hy vọng, những phương pháp này không cần bất kỳ ý kiến nào.

Xử lý khóa và giá trị:

- **V put(K key, V value):** liên kết **value** với **key** được chỉ định và trả về giá trị đã được liên kết trước đó với key hoặc null;
- **V get(Object key)** trả về giá trị được liên kết với khóa hoặc **null** nếu không tìm thấy;
- **V remove(Object key)** loại bỏ ánh xạ cho một key từ Map;
- **boolean containsKey(Object key)** trả lại true nếu Map có chứa key;
- **boolean containsValue(Object value)** trả lại true nếu Map có chứa value.

Các phương thức trả về các tập hợp khác:

- **Set<K> keySet()** trả về một **Set** các khóa có trong Map này;
- **Collection<V> values()** trả về một **Collection** xem các giá trị có trong Map này;
- **Set<Map.Entry<K, V>> entrySet()** trả về một **Set** các mục (liên kết) (**Set** view of the entries (associations)) có trong Map này.

Phương thức nâng cao:

- **V putIfAbsent(K key, V value)** đặt một cặp nếu khóa được chỉ định chưa được liên kết với một giá trị (hoặc được ánh xạ tới null) và trả lại null, nếu không, trả về giá trị hiện tại;
- **V getOrDefault(Object key, V defaultValue)** trả về giá trị mà khóa đã chỉ định được ánh xạ, hoặc **defaultValue** nếu Map này không chứa ánh xạ cho khóa.

Cách đơn giản nhất để tạo Map là gọi phương thức **of** của giao diện **Map**. Phương thức này nhận không hoặc số chẵn đối số trong định dạng key1, value1, key2, value2, ...và trả về một Map bất biến.

```
1 Map<String, String> emptyMap = Map.of();
2 Map<String, String> friendPhones = Map.of(
3             "Minh", "0932932343",
```

```
            "An", "09475983893",
            "Duong", "094759432"
);
```

Bây giờ hãy xem xét một số hoạt động có thể được áp dụng cho các Map bất biến bằng cách sử dụng ví dụ của với **friendPhones**.

Kích thước của một Map bằng với số lượng cặp chứa trong đó.

```
System.out.println(emptyMap.size());
System.out.println(friendPhones.size());
```

Có thể lấy giá trị từ Map bằng khóa của nó:

```
String bobPhone = friendPhones.get("Minh");
String alicePhone = friendPhones.get("An");
String phone = friendPhones.getOrDefault("Duong", "Unknown phone");
```

Cũng có thể kiểm tra xem bản đồ có chứa một khóa hoặc giá trị cụ thể hay không bằng cách sử dụng các phương pháp **containsKey** và **containsValue**.

Truy cập trực tiếp vào tập hợp khóa và bộ sưu tập các giá trị từ bản đồ:

```
System.out.println(friendPhones.keySet());
System.out.println(friendPhones.values());
```

HashMap

Lớp **HashMap** đại diện cho một Map được hỗ trợ bởi một bảng băm. Việc triển khai này cung cấp hiệu suất có thời gian cố định các phương thức **get** và **put** (giả sử phương thức băm phân tán các phần tử đúng cách giữa các nhóm).

Ví dụ sau minh họa một Map các sản phẩm trong đó khóa(key) là mã sản phẩm và giá trị(value) là tên.

```
Map<Integer, String> products = new HashMap<>();
products.put(1000, "Sach");
products.put(2000, "But");
products.put(3000, "Dien Thoai");
System.out.println(products);
System.out.println(products.get(1000));
products.remove(1000);
System.out.println(products.get(1000));

products.putIfAbsent(3000, "Dien Thoai");
System.out.println(products.get(3000));
```

TreeMap

Lớp **TreeMap** đại diện cho một Map cung cấp cho chúng ta sự đảm bảo về thứ tự của các phần tử. Nó tương ứng với thứ tự sắp xếp của các khóa được xác định theo thứ tự tự nhiên của chúng (nếu chúng

triển khai **Comparable** giao diện) hoặc thực hiện Comparator cụ thể.
Lớp này triển khai giao diện **SortedMap** mở rộng giao diện cơ sở **Map**. Nó cung cấp một số phương pháp mới, liên quan đến việc so sánh các khóa:

- **Comparator<? super K> comparator()** trả về trình so sánh được sử dụng để sắp xếp các phần tử trong bản đồ hoặc **null** nếu bản đồ sử dụng thứ tự tự nhiên của các khóa của nó;
- **E firstKey()** trả về khóa đầu tiên (thấp nhất) trong bản đồ;
- **E lastKey()** trả về khóa cuối cùng (cao nhất) trong bản đồ;
- **SortedMap<K, V> headMap(K toKey)** trả về một submap chứa các phần tử có khóa nhỏ hơn **toKey**;
- **SortedMap<K, V> tailMap(K fromKey)** trả về một submap chứa các phần tử có khóa lớn hơn hoặc bằng **fromKey**;
- **SortedMap<K, V> subMap(K fromKey, E toKey)** trả về một submap chứa các phần tử có khóa nằm trong phạm vi **fromKey** (inclusive) **toKey** (exclusive);

Ví dụ dưới đây trình bày cách tạo và sử dụng một đối tượng của **TreeMap**. Bản đồ này chứa các sự kiện, mỗi sự kiện có ngày (khóa) và tiêu đề (giá trị).

LocalDate là một lớp đại diện cho một ngày tháng. Lời kêu gọi của **LocalDate.of(year, month, day)** phương thức tạo đối tượng ngày tháng được chỉ định với **year, month, day** đã cho.

```
1  SortedMap<LocalDate, String> events = new TreeMap<>();
2  events.put(LocalDate.of(2023, 6, 6), "Lập trình Java");
3  events.put(LocalDate.of(2023, 6, 7), "Thiết kế Web");
4  events.put(LocalDate.of(2023, 6, 8), "Cơ sở dữ liệu");
5  LocalDate fromInclusive = LocalDate.of(2023, 6, 7);
6  LocalDate toExclusive = LocalDate.of(2023, 6, 8);
7  System.out.println(events.subMap(fromInclusive, toExclusive));
```

Đầu ra:

```
{2023-06-07= Thiết kế Web, 2013-06-08=Cơ sở dữ liệu}
```

Sử dụng **TreeMap** chỉ khi thực sự cần thứ tự sắp xếp của các phần tử vì việc triển khai này thường kém hiệu quả hơn **HashMap**.

Duyệt bản đồ

Bản đồ thường được duyệt thông qua khóa hoặc giá trị

```
1  Map<String, String> friendPhones = Map.of(
2              "Bob", "+1-202-555-0118",
3              "James", "+1-202-555-0220",
4              "Katy", "+1-202-555-0175"
5  );
6  // In ra tên
```

```
for (String name : friendPhones.keySet()) {
    System.out.println(name);
}
// In ra số điện thoại
for (String phone : friendPhones.values()) {
    System.out.println(phone);
}
```

Nếu muốn in một khóa và giá trị liên quan của nó trong cùng một lần lặp, có thể lấy **entrySet()** và lặp lại nó.

```
for (var entry : friendPhones.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

Mã này in tất cả các cặp khoá/giá trị:

```
Bob: +1-202-555-0118
James: +1-202-555-0220
Katy: +1-202-555-0175
```

Hành vi tương tự có thể đạt được bằng cách sử dụng biểu thức lambda với hai đối số:

```
friendPhones.forEach((name, phone) -> System.out.println(name + ": " + phone));
```

ITERATOR

Các **Iterator <T>** là một cơ chế phổ biến để lặp qua các tập hợp bất kể cấu trúc của chúng.. **iterator** cho phép xóa các phần tử khỏi tập hợp cơ bản nhưng không thể thực hiện việc đó bằng cách sử dụng vòng lặp for-each.

Một số phương thức của giao diện **Iterator <E>**:

- **boolean hasNext()** trả lại **true** nếu vẫn còn phần tử và **false** nếu không;
- **E next()** trả về phần tử tiếp theo trong vòng lặp;
- **void remove()** loại bỏ phần tử cuối cùng được trả về bởi trình lặp này khỏi bộ sưu tập.

Cách sử dụng điển hình bao gồm ba bước:

- Kiểm tra bộ sưu tập có phần tử tiếp theo.
- Chuyển đến phần tử tiếp theo.
- Xử lý phần tử thu được.

```
Set<Long> set = new TreeSet<>();
set.add(10L);
set.add(5L);
set.add(18L);
set.add(14L);
```

```
set.add(9L);
System.out.println(set); // [5, 9, 10, 14, 18]
Iterator<Long> iter = set.iterator();
while (iter.hasNext()) {
    Long current = iter.next();
    if (current < 10L) {
            iter.remove();
    }
}
System.out.println(set); // [10, 14, 18]
```

LỚP TIỆN ÍCH COLLECTIONS

Java Collections framework bao gồm lớp tiện ích **Collections**, chứa một số phương thức tĩnh để tạo và xử lý Collection. Các lập trình viên thường quên mất lớp này và viết lại các phương thức đã có. Rõ ràng, tốt hơn là nên nhớ về lớp này và kiểm tra xem nó có chứa các thao tác cần thực hiện với một tập hợp hay không. Xin đừng nhầm lẫn lớp **Collections** và giao diện **Collection**.

Dưới đây là một số phương thức quan trọng mà bạn có thể sử dụng từ Collections:

- **sort(List<T> list):** Sắp xếp các phần tử trong danh sách theo thứ tự tăng dần, sử dụng phương pháp so sánh mặc định của đối tượng.
- **reverse(List<?> list):** Đảo ngược thứ tự của các phần tử trong danh sách.
- **binarySearch(List<? extends Comparable<? super T>> list, T key):** Tìm kiếm nhị phân một phần tử trong danh sách được sắp xếp và trả về chỉ mục của phần tử đó nếu nó tồn tại; ngược lại, trả về một số âm thể hiện vị trí mà phần tử có thể được chèn để duy trì sự sắp xếp.
- **shuffle(List<?> list):** Xáo trộn ngẫu nhiên các phần tử trong danh sách.
- **addAll(Collection<? super T> c, T... elements):** Thêm các phần tử được chỉ định vào tập hợp.
- **max(Collection<? extends T> coll):** Trả về phần tử lớn nhất trong tập hợp, sử dụng phương pháp so sánh mặc định của đối tượng.
- **min(Collection<? extends T> coll):** Trả về phần tử nhỏ nhất trong tập hợp, sử dụng phương pháp so sánh mặc định của đối tượng.

Một số ví dụ về sắp xếp danh sách:

```
var numbers = new ArrayList<>(List.of(1, 2, 3, 2, 3, 4));
Collections.sort(numbers);  // [1, 2, 2, 3, 3, 4]
Collections.reverse(numbers); // [4, 3, 3, 2, 2, 1]
Collections.shuffle(numbers);
System.out.println(numbers);
```

Một số ví dụ về tính toán số trên danh sách:

```
List<Integer> numbers = List.of(1, 2, 3, 2, 3, 4);
System.out.println(Collections.frequency(numbers, 3)); // 2
System.out.println(Collections.min(numbers)); // 1
System.out.println(Collections.max(numbers)); // 4
```

```
System.out.println(Collections.disjoint(numbers, List.of(1, 2))); // False: Không có phần tử chung
System.out.println(Collections.disjoint(numbers, List.of(5, 6))); // True: Có phần tử chung
```

Nếu bộ sưu tập trống, các phương thức **min** và **max** sẽ ném ra ngoại lệ **NoSuchElementException**. Nhưng **frequency** sẽ chỉ trả về 0.

THỰC HÀNH

Lớp tổng quát Pair

Tạo class Pair có nội dung như sau:

```
public class Pair<T> {
    private T first;
    private T second;
    public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }
    public T getFirst() {
        return first;
    }
    public T getSecond() {
        return second;
    }
    public void setFirst(T first) {
        this.first = first;
    }
    public void setSecond(T second) {
        this.second = second;
    }
}
```

Viết phương thức main Sử dụng class Pair này:

```
public static void main(String[] args) {
    Pair<String> pairString = new Pair("Hello", "World");
    Pair<Integer> pairInteger = new Pair(1, 3);
    Pair<Double> pairDouble = new Pair(-2.5, 1.3);
    Pair pair = new Pair<>("Hello", "World");
    pair = new Pair(1, 3);
    pair = new Pair(-2.5, 1.3);
    System.out.println(pairString.getSecond());
    System.out.println(pair.getFirst());
}
```

Kết quả in ra màn hình:

```
1  World
2  -2.5
```

Thực hành thêm: Hãy viết một phương thức swap đổi chỗ first và second cho nhau.

Giới hạn kiểu

Viết phương thức trả về giá trị min và max của các phần tử trong 1 mảng có kiểu T:

```
 1  public static <T extends Comparable> Pair<T> minmax(T[] a) {
 2     if (a == null || a.length == 0) return null;
 3     T min = a[0];
 4     T max = a[0];
 5     for (int i = 1; i < a.length; i++) {
 6         if (min.compareTo(a[i]) > 0) min = a[i];
 7         if (max.compareTo(a[i]) < 0) max = a[i];
 8     }
 9     return new Pair<>(min, max);
10  }
```

Sử dụng phương thức nói trên để tìm giá trị lớn nhất nhỏ nhất:

```
 1  public static void main(String[] args) {
 2     Integer[] numbers = {2, 3, 5, 2, 5};
 3     LocalDate[] birthdays =
 4             {
 5                     LocalDate.of(1906, 12, 9),
 6                     LocalDate.of(1815, 12, 10),
 7                     LocalDate.of(1903, 12, 3),
 8                     LocalDate.of(1910, 6, 22)
 9             };
10     Pair<Integer> pairMinMaxNum = minmax(numbers);
11     System.out.println("Min:" + pairMinMaxNum.getFirst() + " - Max:" + pairMinMaxNum.
getSecond());
12     Pair<LocalDate> pairMinMaxDay = minmax(birthdays);
13     System.out.println("Min:" + pairMinMaxDay.getFirst() + " - Max:" + pairMinMaxDay.
getSecond());
14  }
```

Kết quả in ra màn hình:

```
1  Min:2 - Max:5
2  Min:1815-12-10 - Max:1910-06-22
```

Danh sách mảng ArrayList

Tạo danh sách mới arrayList:

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

Bổ sung 5 loại quả vào arrayList và in ra:

```
list.add("apples");
list.add("bananas");
list.add("grapes");
list.add("pears");
list.add("black plums");
System.out.println("My array list currently includes: " + list);
```

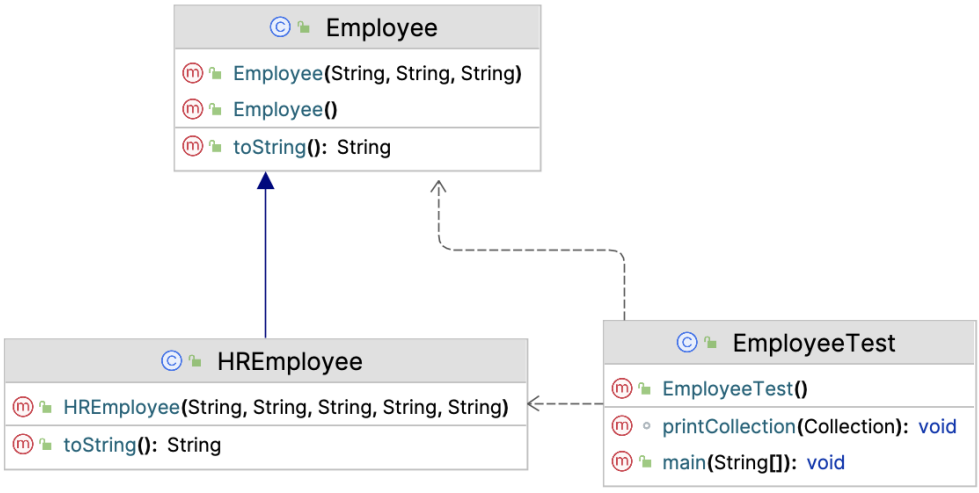
Gọi một số phương thức trên danh sách:

```
list.add(0, "watermelons");
System.out.println("My array list now includes: " + list);
list.add(3, "oranges");
System.out.println("My array list now includes: " + list);
System.out.println("'grapes' is an element in the list: " + list.contains("grapes"));
System.out.println("'apricots' is an element in the list: " + list.contains("apricots"));
System.out.println("Element at index 2 is: " + list.get(2));
System.out.println("Element at index 1 is: " + list.get(1));
System.out.println("The size of our current array list is: " + list.size());
```

Bài tập: Hãy viết thêm mã nguồn để:

- Kiểm tra xem trong danh sách có "lemons" hay không
- Phần tử cuối cùng của danh sách có phải "bananas" hay không
- Đưa "cheries" vào sau "apples"
- Xóa "graphes" nếu có
- Chuyển các ký tự trong danh sách thành chữ viết hoa

Danh sách nhân viên



Tạo lớp Employee

```
public class Employee {
    private String employeeName;
```

```
    private String employeeNo;
    private String employeeDesgn;
    public Employee() {
    }
    public Employee(String eName, String eNo, String eDesgn) {
        employeeName = eName;
        employeeNo = eNo;
        employeeDesgn = eDesgn;
    }
    public String toString() {
        return "Name: " + employeeName + "\tNumber: " + employeeNo + "\tDesignation:
" + employeeDesgn;
    }
}
```

Tạo lớp HREmployee

```
public class HREmployee extends Employee {
    private String deptName;
    private String deptCode;
    public HREmployee(String name, String num, String desig, String dept, String cod
e) {
        super(name, num, desig);
        deptName = dept;
        deptCode = code;
    }
    public String toString() {
        return super.toString() + "\tDepartment Name: " + deptName + "\tDepartment Co
de:" + deptCode;
    }
}
```

Chương trình chạy

```
public class EmployeeTest {
    static void printCollection(Collection collect) {
        for (Object obj : collect)
            System.out.println(obj.toString());
    }
    public static void main(String[] args) {
        ArrayList<Employee> emp = new ArrayList<>(1);
        System.out.println("Danh sách nhân viên: ");
        System.out.println("************************");
        emp.add(0, new Employee("Trần Dương Minh", "A001", "Team Lead"));
        emp.add(1, new Employee("Lê Anh Quân", "A002", "Project Manager"));
        printCollection(emp);
        ArrayList<HREmployee> empHR = new ArrayList<>(1);
        System.out.println("\nHREmployee Class Details: ");
        System.out.println("************************");
        empHR.add(0, new HREmployee("Lê Bảo Anh", "A001", "Project Manager", "Operati
ons", "01"));
        empHR.add(1, new HREmployee("Nguyễn Quốc Cường", "A002", "Team Leader", "Writ
ing", "02"));
```

```
        printCollection(empHR);
    }
}
```

Danh sách sinh viên

Tạo lớp Student

```
public class Student {
    String StudentName;
    String StudentID;
    String description;
    int Studentage;
    public Student() {
    }
    public Student(String pName, String pID, String descr, int age) {
        StudentName = pName;
        StudentID = pID;
        description = descr;
        Studentage = age;
    }
    public String toString() {
        return "Student Name: " + StudentName + "\tStudent ID: " + StudentID + "\tDes
cription: " + description + "\tage: " + Studentage;
    }
}
```

Tạo lớp PeopeList

```
public class PeopleList<T> {
    ArrayList<T> listPeople;
    public PeopleList() {
        listPeople = new ArrayList<T>();
    }
    public void add(T obj) {
        listPeople.add(obj);
    }
    public void display() {
        for (T objPeople : listPeople) {
            System.out.println(objPeople.toString());
        }
    }
    public int getSize() {
        return listPeople.size();
    }
    public boolean checkEmpty() {
        return (listPeople.size() == 0);
    }
    public T grab() {
        if (!checkEmpty()) {
            return listPeople.remove(0);
        } else
            return null;
```

```
    }
}
```

Tạo chương trình chạy

```
public class Program {
    public static void main(String[] args) throws Exception {
        PeopleList<Student> objStudent = new PeopleList<Student>();
        objStudent.add(new Student("Pham Xuan Minh", "123", "Class1", 22));
        objStudent.add(new Student("Duong Mai Huong", "124", "Class2", 20));
        objStudent.add(new Student("Tran Minh Tuan", "125", "Class1", 19));
        System.out.println("Is the list empty? " + objStudent.checkEmpty());
        System.out.println("Size of the list: " + objStudent.getSize());
        System.out.println("Student Details:");
        System.out.println("************");
        objStudent.display();
    }
}
```

Thực hiện chương trình

```
Is the list empty? false
Size of the list: 3
Student Details:
************
Student Name: Pham Xuan Minh        Student ID: 123 Description: Class1     age: 22
Student Name: Duong Mai Huong       Student ID: 124 Description: Class2     age: 20
Student Name: Tran Minh Tuan        Student ID: 125 Description: Class1     age: 19
```

Bài tập: Hãy sửa lại chương trình để các sinh viên in ra theo danh sách của bảng chữ cái (Tên theo ABC)

Danh sách liên kết LinkedList

Tạo class ExLinkedList với phương thức main, trong đó tạo LinkedList:

```
LinkedList<Integer> list = new LinkedList<>();
```

Bổ sung dữ liệu vào danh sách

```
list.add(11);
list.add(22);
list.add(33);
list.add(44);
```

Tạo iterator để in ra danh sách

```
System.out.print("Linked list data: ");
Iterator iterator = list.iterator();
while (iterator.hasNext())
    System.out.print(iterator.next() + " ");
System.out.println();
```

Kiểm tra danh sách rỗng hay không

```
if (list.isEmpty())
   System.out.println("Danh sách liên kết rỗng");
else
   System.out.println("Kích thước danh sách liên kết: " + list.size());
```

Bổ sung phần tử vào đầu danh sách

```
list.addFirst(55);
iterator = list.iterator();
while (iterator.hasNext())
   System.out.print(iterator.next() + " ");
System.out.println(list.size());
```

Bổ sung phần tử vào cuối danh sách

```
list.addLast(66);
iterator = list.iterator();
while (iterator.hasNext())
   System.out.print(iterator.next() + " ");
System.out.println();
System.out.println(list.size());
```

Bổ sung phần tử vào vị trí thứ 3

```
list.add(2, 99);
iterator = list.iterator();
while (iterator.hasNext())
   System.out.print(iterator.next() + " ");
System.out.println();
System.out.println(list.size());
```

In ra phần tử ở đầu và ở cuối

```
System.out.println("Phần tử đầu: " + list.getFirst());
System.out.println("Phần tử cuối: " + list.getLast());
```

Xóa phần tử ở cuối

```
int last = list.removeLast();
iterator = list.iterator();
while (iterator.hasNext()
   System.out.print(iterator.next()+" ");
System.out.println();
System.out.println(list.size());
```

Xóa toàn bộ các phần tử

```
list.clear();
```

```
2  list.size());
```

Hãy chạy thử và giải thích từng kết quả của chương trình in ra màn hình

HashSet và TreeSet

Tạo lớp ExHashSet với phương thức main, trong đó tạo HashSet đặt tên là s:

```
Set<String> s = new HashSet<>();
```

Bổ sung các phần tử vào HashSet và in ra kết quả:

```
 1  s.add("grapes");
 2  s.add("bananas");
 3  s.add("apples");
 4  s.add("pears");
 5  s.add("black plums");
 6  System.out.println(s);
 7  s.add("watermelons");
 8  System.out.println(s);
 9  s.add("bananas");
10  System.out.println(s);
```

Kiểm tra xem set có các phần tử **grapes** và **apricots** hay không

```
 1  System.out.println("'grapes' là một phần tử trong set: " + s.contains("grapes"));
 2  System.out.println("'apricots' là một phần tử trong set: "+  s.contains("apricot
s"));
```

Xoá các phần tử trong HashSet và hiển thị kích thước

```
 1  System.out.println("The set is currently empty: " + s.isEmpty());
 2  System.out.println("The size of set is: " + s.size());
```

Chạy chương trình và kiểm tra kết quả:

```
 1  'grapes' là một phần tử trong set: true
 2  'apricots' là một phần tử trong set: false
```

Bài tập: Hãy viết mã tạo một đối tượng **TreeSet**, trong phương thức tạo của **TreeSet** này đưa đối tượng s vào để copy toàn bộ nội dung **HashSet** vào **TreeSet**. Hãy hiển thị các Item trong **TreeSet** và đánh giá kết quả khác gì so với **HashSet.**

Thực hành với Map

Phần này sẽ thực hành việc sử dụng Hashmap và Treemap. Tạo một lớp ExHashMap, trong đó phương thức main tạo một HashMap chứa khóa và giá trị:

```
HashMap<String, Integer> myMap = new HashMap<String, Integer>();
```

Đưa vào trong HashMap các cặp khóa, giá trị ngẫu nhiên:

```
myMap.put("Apples", (int) (Math.random() * 10));
myMap.put("Bananas", (int) (Math.random() * 10));
myMap.put("Grapes", (int) (Math.random() * 10));
myMap.put("Watermelons", (int) (Math.random() * 10));
```

Khóa Cherries có giá trị giống như Grapes

```
myMap.put("Cherries", myMap.get("Grapes"));
```

Thay khóa Apples chứa giá trị ngẫu nhiên khác:

```
myMap.put("Apples", (int) (Math.random() * 10));
```

Tạo ra một đối tượng mySortedMap

```
TreeMap<String, Integer> mySortedMap = new TreeMap<>(myMap);
System.out.println("my sorted map is: " + mySortedMap);
```

Chạy và in ra kết quả như sau:

```
My sorted map is: {Apples=8, Bananas=2, Cherries=9, Grapes=9, Watermelons=0}
```

Tạo một myMap2, đổi giá trị và khóa

```
HashMap<Integer, String> myMap2 = new HashMap<>();
for (String s : myMap.keySet())
   myMap2.put(myMap.get(s), s);
System.out.println("my second map is: " + myMap2);
```

Tạo mySortedMap2 sắp xếp các myMap2 theo key

```
TreeMap<Integer, String> mySortedMap2 = new TreeMap<>(myMap2);
System.out.println("my second sorted map is: " + mySortedMap2);
System.out.println();
```

Kiểm tra myMap2 xem 3 có phải key

```
System.out.println("my second map contains 3: " + myMap2.containsKey(3));
```

Kiểm tra myMap2 xem "Apples" có phải giá trị

```
System.out.println("my second map contains Apples: " + myMap2.containsValue("Apples"));
```

Thực hiện chương trình. Chạy và kiểm tra kết quả, đầu ra có thể như sau:

```
my sorted map is: {Apples=7, Bananas=7, Cherries=6, Grapes=6, Watermelons=6}
my second map is: {6=Cherries, 7=Bananas}
my second sorted map is: {6=Cherries, 7=Bananas}
```

```
4  my second map contains 3: false
5  my second map contains Apples: false
```

Hãy in thử kích thước của myMap và myMap2

```
1  System.out.println("my map has " + myMap.size() + " keys");
2  System.out.println("my second map has " + myMap2.size() + " keys");
```

Bài tập: Hãy giải thích tại sao kích thước của myMap2 thay đổi trong khi kích thước của MyMap lại cố định với mỗi lần chạy khác nhau

Hãy viết một phương thức **swap** đổi chỗ **first** và **second** cho nhau

CÂU HỎI ÔN TẬP LÝ THUYẾT

1. Đâu là hạn chế của việc lập trình tổng quát sử dụng kế thừa

- A. Phải ép kiểu trong nhiều trường hợp
- B. Mã lệnh dài
- C. Kiểm tra kiểu qua chặt chẽ
- D. Chưa đủ tính tổng quát

2. Trong quy ước đặt tên cho các tham số kiểu, T đại diện cho

- A. Giá trị
- B. Số
- C. Khóa
- D. Kiểu chung

3. Từ khóa var được sử dụng

- A. Từ phiên bản java 1
- B. Từ phiên bản java 4
- C. Từ phiên bản java 8

D. Từ phiên bản java 10

4. Khi tạo đối tượng từ lớp tổng quát

A. Sử dụng một kiểu tổng quát

B. Không cần chỉ định đối số kiểu

C. Cần chỉ định đối số kiểu

D. Chỉ định sử dụng kiểu dữ liệu cơ sở

5. Phương thức tổng quát là phương thức

A. Chỉ có thể nằm trong một lớp tổng quát

B. Có thể đặt trong một lớp thường

C. Bắt buộc phải là phương thức tĩnh

D. Được ghi đè từ một phương thức tổng quát

6. Để giới hạn kiểu khi khai báo một lớp tổng quát sử dụng từ khóa

A. extends

B. limit

C. implements

D. instanceof

7. Khởi tạo một Danh sách (List)

A. Bắt buộc phải khai báo kích thước

Phải khởi tạo các giá trị và không thể thay đổi sau đó

Giống như khởi tạo một đối tượng

D. Cần khai báo kiểu của các phần tử

8. Sự khác sau giữa Set và List là

A. List luôn tối ưu về mặt lưu trữ dữ liệu

B. Set luôn sắp xếp các phần tử

C. List cho phép các phần tử trùng nhau, Set thì không

D. Khác nhau về tốc độ thực hiện

9. Danh sách bất biến (Immutable list) được tạo ra bởi

A. new ArrayList()

B. List.of()

C. List.imutable()

D. LinkedList

10. ArrayList hoạt động nhanh hơn LinkedList khi gọi hàm

A. get

B. add

C. remove

D. search

11. TreeSet có đặc điểm gì

A. Tổ chức các phần tử theo danh sách

B. Luôn sắp xếp các phần tử theo giá trị

C. Cho phép các phần tử trùng nhau

D. Sắp xếp các phần tử theo giá trị băm

12. Hãy cho biết hàm makelist có thể truyền biến t1 t2 kiểu gì?

```
public static <T> List<T> makeList(T t1, T t2) {
    List<T> result = new ArrayList<T>();
    result.add(t1);
    result.add(t2);
    return result;
}
```

A. Dữ liệu cơ sở

B. Đối tượng

C. Số hoặc chuỗi

D. Danh sách

13. Hãy cho biết kết quả thực hiện đoạn mã sau:

```
public abstract class Component {
    private int x, y;
    int Component(int x, int y){
        return x+ y;
    }
    public static void main(String[] args) {
        Component component = new Component(3,4);
        System.out.println(component);

    }
}
```

A. In ra giá trị 7

B. Chương trình đưa ra một ngoại lệ

C. In ra tổng x và y

D. Lỗi biên dịch

14. Hãy cho biết kết quả thực hiện đoạn mã lệnh sau:

```java
class Car {
    public int gearRatio = 8;
    public String accelerate() {
        return "car";
    }
}

public class SportsCar extends Car {
    public int gearRatio = 9;
    public String accelerate() {
        return "accelerate" ;
    }
    public static void main(String[] args) {
        Car car = new SportsCar();
        System.out.println(car.gearRatio + " " + car.accelerate());
    }
}
```

- A. In ra số: 8
- B. In ra dòng chữ: 8 car
- C. In ra dòng chữ: 8 accelerate
- D. Đoạn mã có lỗi biên dịch

15. Hãy cho biết kết quả thực hiện đoạn mã lệnh sau:

```java
List<String> fruits = new ArrayList<String>();
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Strawberry");

for (String fruit : fruits) {
    System.out.println(fruit);
    if ("Apple".equals(fruit)) {
        fruits.remove(fruit);
    }
}
```

- A. Xóa Apple khỏi danh sách
- B. Xóa các chuỗi trừ Apple khỏi danh sách
- C. Xóa toàn bộ các phần tử trong danh sách
- D. Xuất hiện ngoại lệ ConcurrentModificationException

BÀI TẬP TỰ THỰC HÀNH

Chuyển chuỗi thành số

Viết chương trình Java để chuyển đổi một danh sách các chuỗi thành một danh sách các số nguyên. Nếu chuỗi không phải là số nguyên, bỏ qua chuỗi đó và tiếp tục với các chuỗi khác.

Tìm các số lặp lại

Cho một danh sách các số, hãy hiển thị ra những số nào bị lặp lại nhiều hơn 1 lần trong danh sách đó

Tìm phần tử lớn nhất trong Set

Viết phương thức findMax để tìm phần tử lớn nhất trong một Set. Nếu Set rỗng, phương thức sẽ trả về null.

Đếm số lần xuất hiện của từ trong câu

Viết chương trình Java sử dụng Map để đếm số lần xuất hiện của mỗi từ trong một câu.

TÀI LIỆU THAM KHẢO

[1] Core Java: Fundamentals (2021) , Cay Horstmann (Oracle Press Java)

[2] Head First Java: A Brain-Friendly Guide (2022), Kathy Sierra, O'Reilly Media

[3] Java OOP Done Right: Create object oriented code you can be proud of with modern Java Paperback (2019), Mr Alan Mellor, Mellor Books

[4] Murach's Java Programming (5th Edition) (2017), Joe Murach, Mike Murach & Associates

[5]. Java for Absolute Beginners Learn to Program the Fundamentals the Java 9+ Way

[6]. Modern Java Recipes: Simple Solutions to Difficult Problems in Java 8 and 9 (2017), by Ken Kousen, O'Reilly Media

[7] Effective Java (2018), Joshua Bloch, Addison-Wesley Professional