1. Collection

* Trong Java, collections (tập hợp) là một phần quan trọng trong lập trình, cho phép bạn lưu trữ và quản lý nhóm đối tượng.
* Các lớp và giao diện này nằm trong gói java.util
* Dưới đây là các loại collection phổ biến trong Java

### **List**

* **ArrayList**: Danh sách có kích thước thay đổi, cho phép truy cập nhanh bằng chỉ số.
* **LinkedList**: Danh sách liên kết, hiệu quả hơn cho các hoạt động chèn và xóa.
* **Vector**: Giống như ArrayList nhưng đồng bộ hóa, thường ít được sử dụng hơn.
* **Stack**: Đối tượng dạng ngăn xếp, hỗ trợ các thao tác như push (thêm) và pop (loại bỏ).

### **b. Set**

* **HashSet**: Không lưu trữ các phần tử trùng lặp và không đảm bảo thứ tự của các phần tử.
* **LinkedHashSet**: Giữ nguyên thứ tự chèn của các phần tử và không lưu trữ các phần tử trùng lặp.
* **TreeSet**: Sắp xếp các phần tử theo thứ tự tự nhiên hoặc theo một comparator cụ thể và không lưu trữ các phần tử trùng lặp.

### **c. Queue**

* **PriorityQueue**: Một hàng đợi ưu tiên, các phần tử được sắp xếp theo thứ tự tự nhiên hoặc theo một comparator.
* **LinkedList**: Cũng có thể được sử dụng như một hàng đợi (queue) vì nó triển khai giao diện Queue.
* **ArrayDeque**: Cung cấp hàng đợi và ngăn xếp (deque) hiệu quả hơn LinkedList.

### **d. Map**

* **HashMap**: Không lưu trữ các khóa trùng lặp và không đảm bảo thứ tự của các cặp khóa-giá trị.
* **LinkedHashMap**: Giữ nguyên thứ tự của các cặp khóa-giá trị như chúng đã được chèn vào.
* **TreeMap**: Sắp xếp các cặp khóa-giá trị theo thứ tự của khóa hoặc theo một comparator cụ thể.
* **Hashtable**: Tương tự như HashMap nhưng đồng bộ hóa, thường ít được sử dụng hơn.

### **e. Deque (Double-ended Queue)**

* **ArrayDeque**: Hỗ trợ thao tác thêm và loại bỏ phần tử từ cả hai đầu của hàng đợi.

### **f. Specialized Collections**

* **EnumSet**: Một tập hợp đặc biệt cho các giá trị của kiểu enum, hiệu quả hơn so với các tập hợp thông thường.
* **EnumMap**: Một bản đồ đặc biệt cho các giá trị của kiểu enum, hiệu quả về bộ nhớ.

2. Ví dụ về hashmap

-Giúp lưu trữ các cặp khóa-giá trị mà không có phần tử trùng lặp và không giữ thứ tự của các phần tử.

-Ví dụ lưu trữ các thuộc tính cấu hình của ứng dụng với tên thuộc tính là khóa và giá trị cấu hình là giá trị.

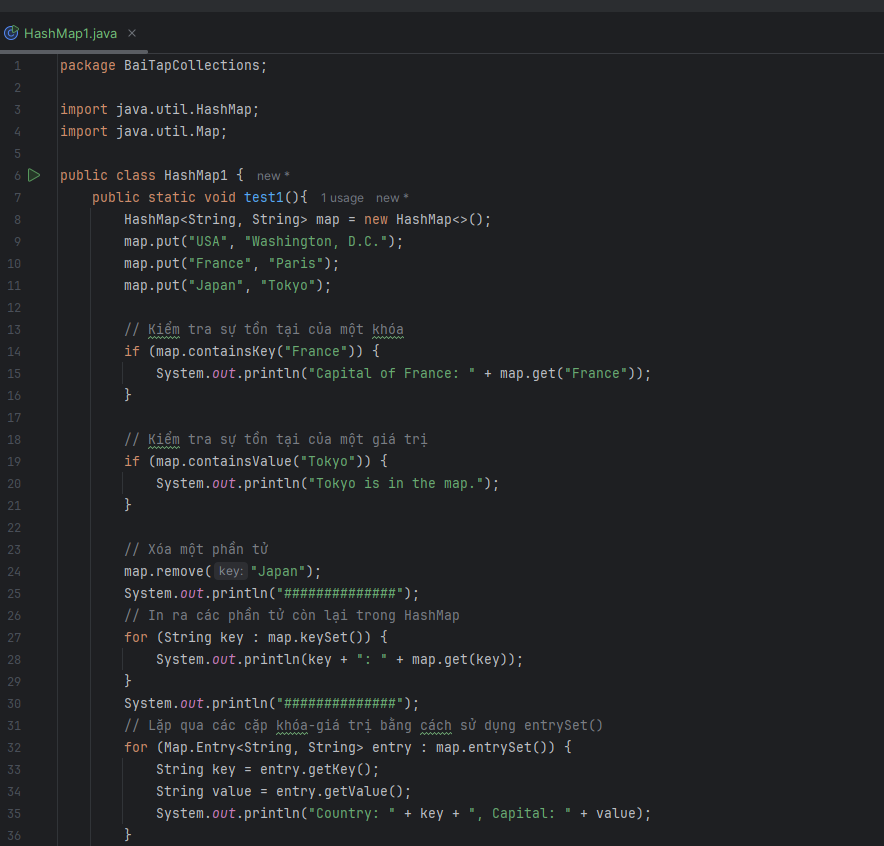
-Ví dụ lưu trữ các phiên làm việc của người dùng với mã phiên làm khóa và thông tin phiên làm giá trị.

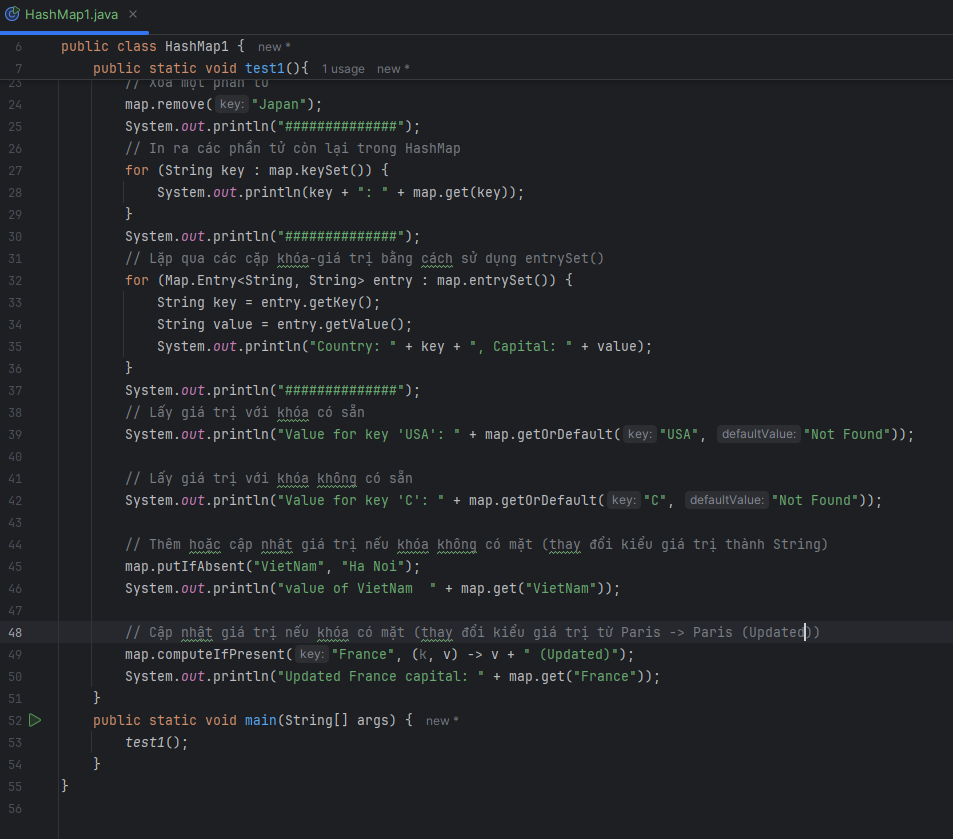
- Ví dụ đếm số lần xuất hiện của các từ trong một văn bản.

- Ví dụ đếm số lần xuất hiện của các số trong một danh sách.

3. Tốc độ

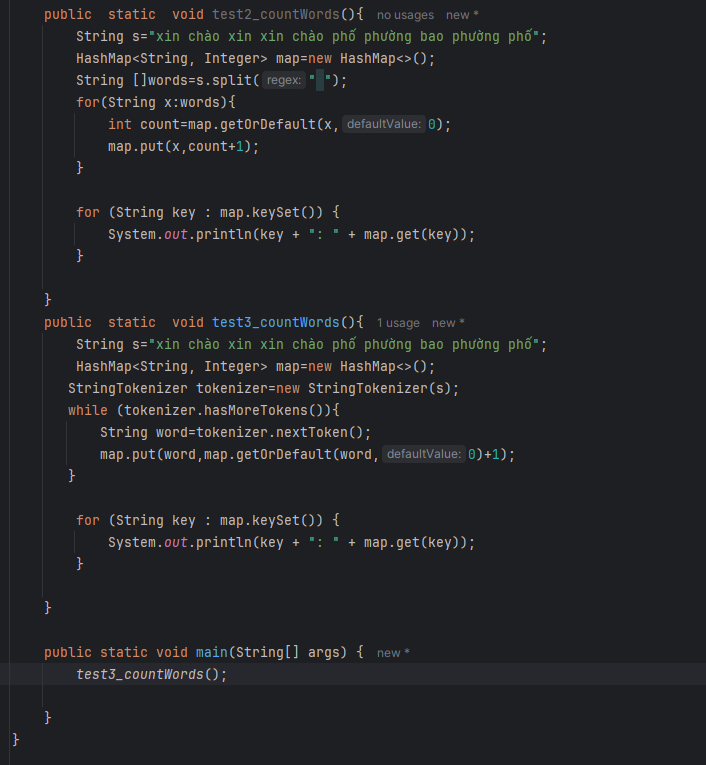
* HashMap cung cấp thời gian tìm kiếm gần như hằng số (O(1)) cho các phần tử. Nó rất hữu ích khi bạn cần thực hiện các phép toán như tìm kiếm, chèn, hoặc xóa các phần tử trong thời gian nhanh. Ví dụ:
* **Lưu trữ các bản ghi cần truy xuất nhanh chóng, như trong các hệ thống caching.**
* **Xử lý các dữ liệu lớn mà bạn cần tìm kiếm hoặc cập nhật nhanh chóng.**
* **HashMap không duy trì thứ tự của các phần tử. Nếu thứ tự là quan trọng, bạn có thể cần sử dụng LinkedHashMap hoặc TreeMap.**
* **HashMap sử dụng bảng băm (hash table) để lưu trữ các cặp khóa-giá trị. Bảng băm là một cấu trúc dữ liệu mà trong đó, các phần tử được lưu trữ tại các vị trí cụ thể trong một mảng (bảng) dựa trên giá trị băm (hash value) của khóa.**

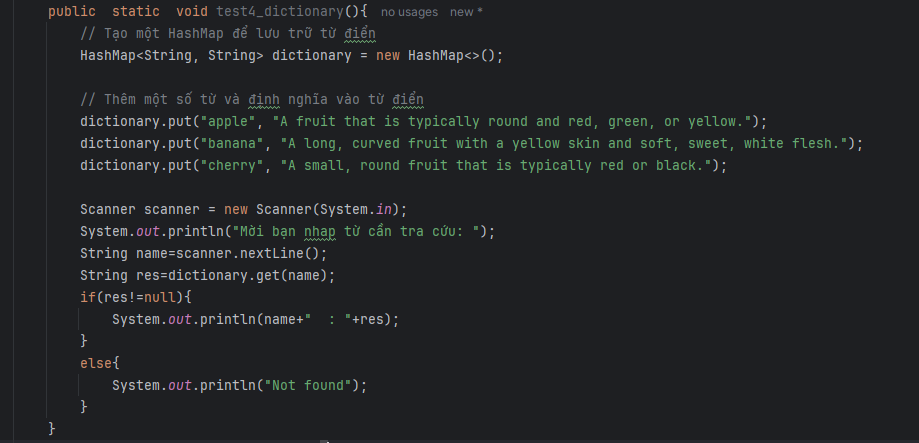


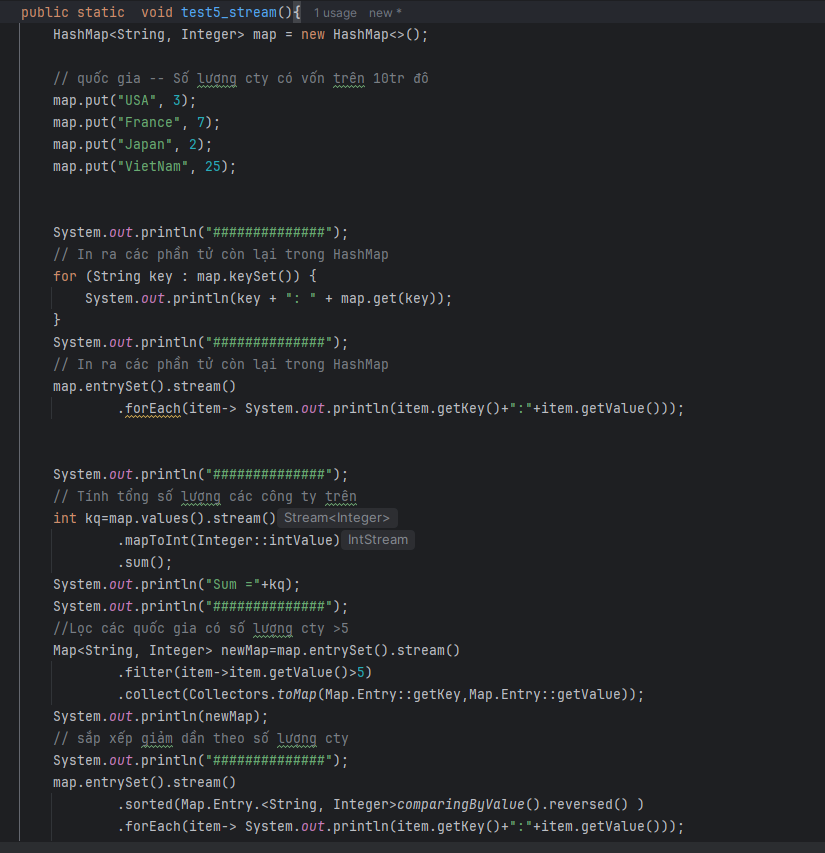


-Lưu ý:

* map.put(key, value) để thêm phần tử vào HashMap.
* map.keySet() trả về một Set chứa tất cả các khóa.
* map.get(key) lấy giá trị tương ứng với khóa.
* map.containsKey(key) kiểm tra sự tồn tại của khóa.
* map.containsValue(value) kiểm tra sự tồn tại của giá trị.
* map.remove(key) xóa phần tử với khóa cụ thể.
* map.entrySet() trả về một Set chứa các Map.Entry đại diện cho các cặp khóa-giá trị.
* entry.getKey() và entry.getValue() lấy khóa và giá trị tương ứng.
* map.getOrDefault(key, defaultValue) trả về giá trị tương ứng với khóa, hoặc giá trị mặc định nếu khóa không tồn tại.
* map.computeIfAbsent(key, mappingFunction) thêm giá trị mới nếu khóa không có mặt.
* map.computeIfPresent(key, remappingFunction) cập nhật giá trị nếu khóa đã có mặt.

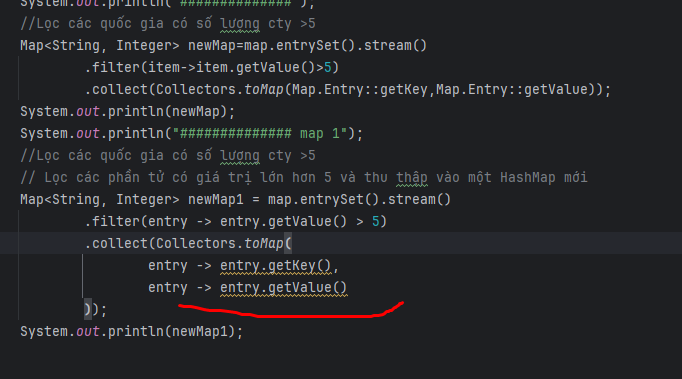






Lưu ý:

* List<String> list = new ArrayList<>();
* **List** là một interface trong Java, định nghĩa các phương thức cho một danh sách, bao gồm các phương thức để thêm, xóa và truy cập phần tử.
* Khai báo biến với kiểu List cho phép bạn dễ dàng thay đổi loại bản triển khai mà không cần thay đổi mã sử dụng. Ví dụ, bạn có thể thay đổi từ ArrayList sang LinkedList hoặc Vector mà không cần thay đổi cách bạn sử dụng biến list.
* Map<String, Integer> map = new HashMap<>();
* **Map** là một interface trong Java. Nó định nghĩa các phương thức mà bất kỳ lớp nào triển khai Map cần phải có.
* Khi bạn khai báo biến map với kiểu là Map, bạn có thể gán bất kỳ thể hiện nào của các lớp triển khai Map cho biến đó (ví dụ: HashMap, TreeMap, LinkedHashMap, v.v.). Điều này cho phép bạn dễ dàng thay đổi loại bản triển khai mà không cần thay đổi mã sử dụng biến đó.



Phần 2: **LinkedHashMap**

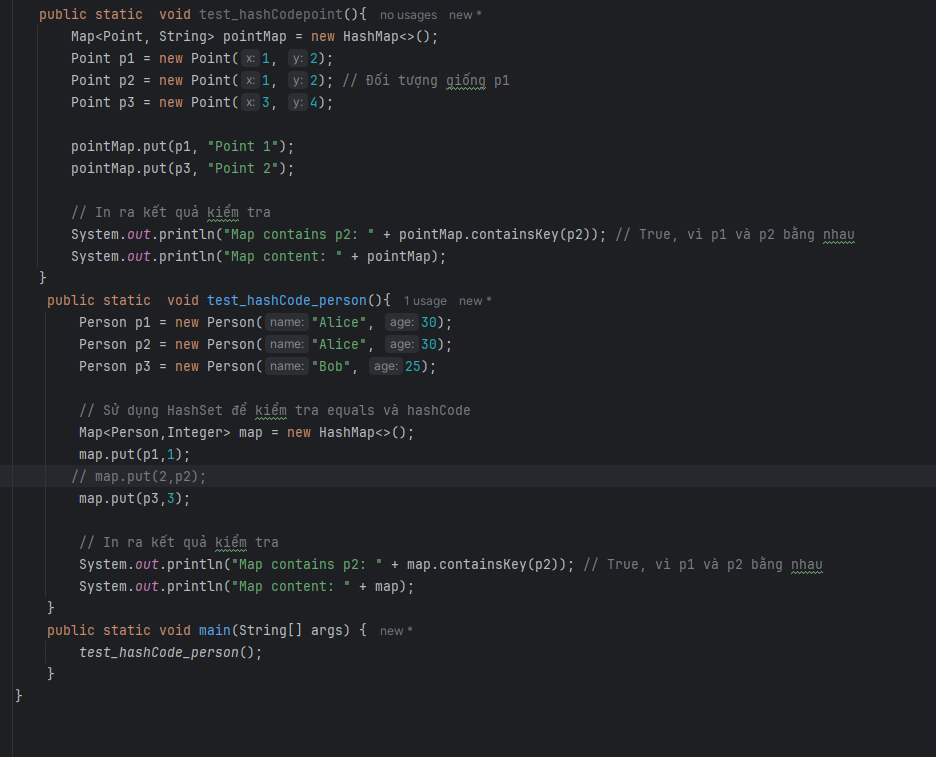
1. **Duy Trì Thứ Tự**:
   * LinkedHashMap duy trì thứ tự của các phần tử theo cách chúng được chèn vào hoặc theo cách chúng được truy cập (nếu được cấu hình cho thứ tự truy cập).
   * Thứ tự này giúp giữ nguyên trình tự chèn hoặc truy cập, điều này hữu ích khi cần giữ thứ tự của các phần tử như khi in ra hoặc xử lý chúng theo thứ tự.
2. **Khóa và Giá Trị**:
   * Giống như HashMap, LinkedHashMap cho phép lưu trữ các cặp khóa-giá trị. Các khóa và giá trị có thể là null.
3. **Hiệu Suất**:
   * LinkedHashMap có hiệu suất tương tự như HashMap khi thực hiện các thao tác thêm, xóa và tìm kiếm phần tử (O(1) trung bình).
   * Tuy nhiên, LinkedHashMap sẽ tiêu tốn thêm một chút bộ nhớ để duy trì cấu trúc liên kết.
4. **Thứ Tự Chèn và Truy Cập**:
   * Khi tạo một LinkedHashMap, bạn có thể chỉ định tham số accessOrder để xác định thứ tự duy trì:
     + **Thứ tự chèn (Insertion Order)**: Đây là mặc định. Các phần tử được lưu theo thứ tự chúng được thêm vào.
     + **Thứ tự truy cập (Access Order)**: Nếu bạn đặt accessOrder là true, LinkedHashMap sẽ duy trì thứ tự của các phần tử theo lần truy cập gần đây nhất, điều này có thể hữu ích cho việc tạo bộ nhớ đệm (cache).
5. 

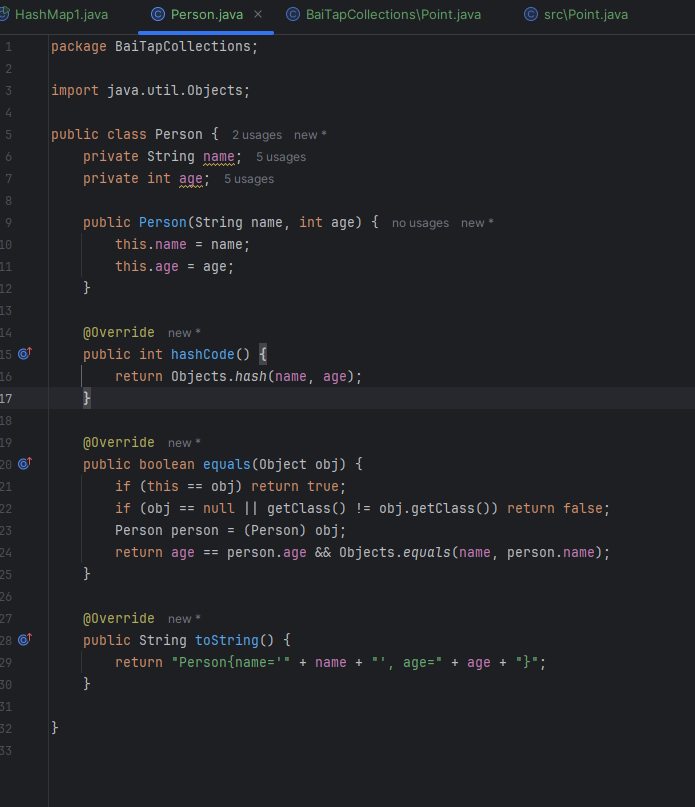
### **Phần 3 TreeMap**

1. **Sắp Xếp Theo Thứ Tự**:
   * TreeMap tự động sắp xếp các phần tử theo thứ tự của các khóa. Mặc định, các khóa được sắp xếp theo thứ tự tự nhiên của chúng (theo thứ tự tăng dần của các đối tượng Comparable), hoặc bạn có thể cung cấp một Comparator tùy chỉnh để xác định thứ tự sắp xếp.
2. **Cấu Trúc Dữ Liệu Cây**:
   * TreeMap dựa trên cây nhị phân tìm kiếm (Red-Black Tree), vì vậy các thao tác thêm, xóa và tìm kiếm phần tử đều có độ phức tạp thời gian trung bình là O(log n), nơi n là số lượng phần tử trong bản đồ.
3. **Không Cho Phép null Key**:
   * TreeMap không cho phép các khóa null. Nếu bạn cố gắng thêm một khóa null, sẽ ném ra NullPointerException. Tuy nhiên, giá trị null là hợp lệ và có thể được sử dụng trong TreeMap.

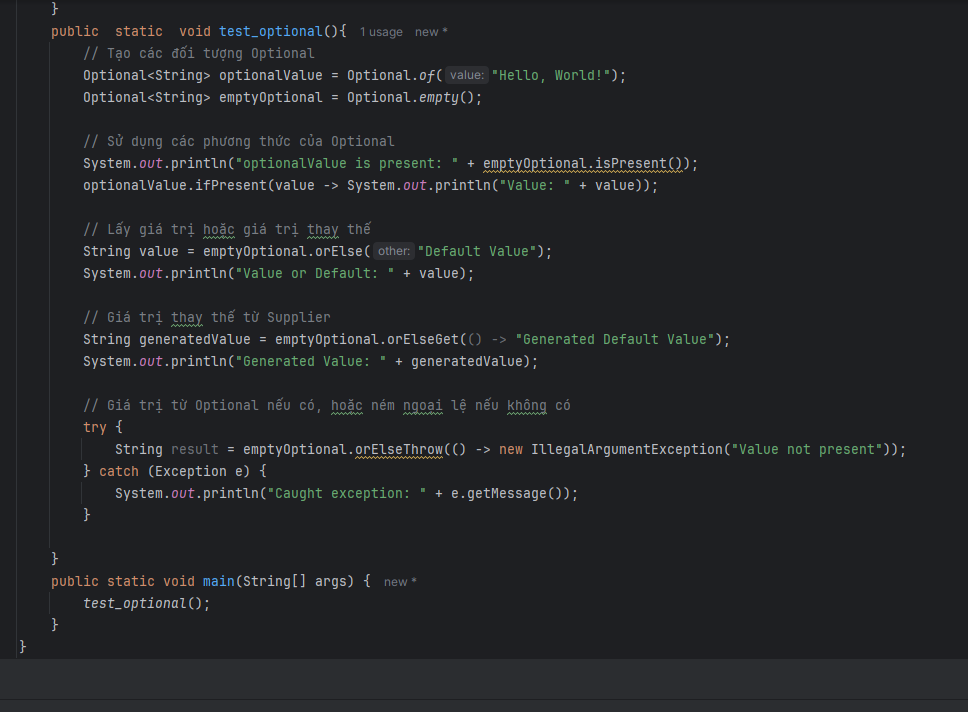


Ví dụ về hashcode







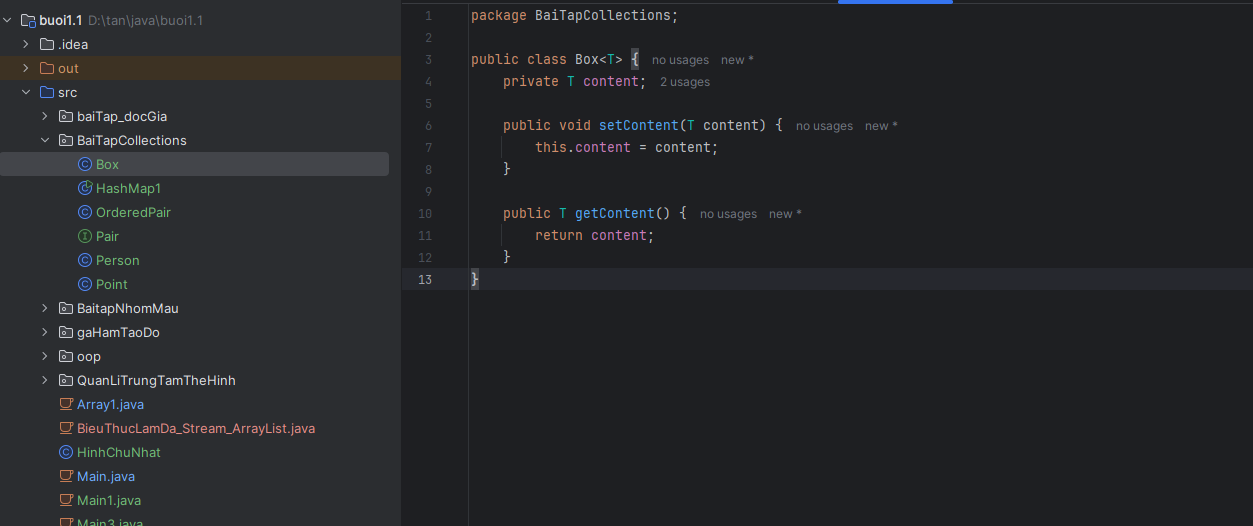
* **hashCode()**: Phương thức dùng để tạo mã băm cho các đối tượng, giúp cấu trúc dữ liệu như HashMap và HashSet xác định vị trí lưu trữ các phần tử.
* **equals()**: Được sử dụng để so sánh hai đối tượng để kiểm tra xem chúng có bằng nhau không.
* **Tính nhất quán**: hashCode() và equals() phải được ghi đè đồng thời để đảm bảo tính nhất quán trong các cấu trúc dữ liệu dựa trên băm.
* Việc triển khai hashCode() và equals() đúng cách rất quan trọng để đảm bảo rằng các đối tượng hoạt động chính xác trong các cấu trúc dữ liệu sử dụng mã băm.
* 

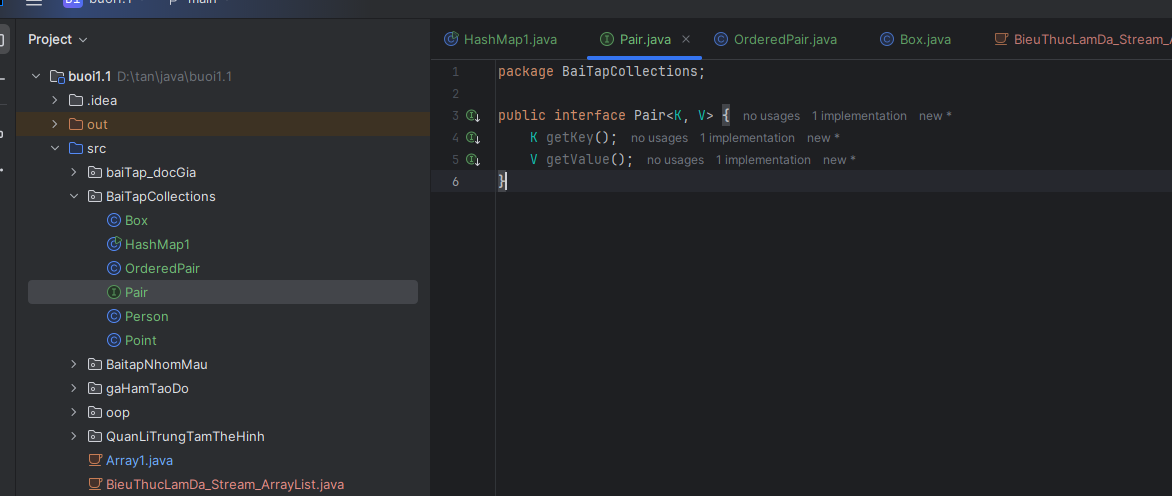
Generic

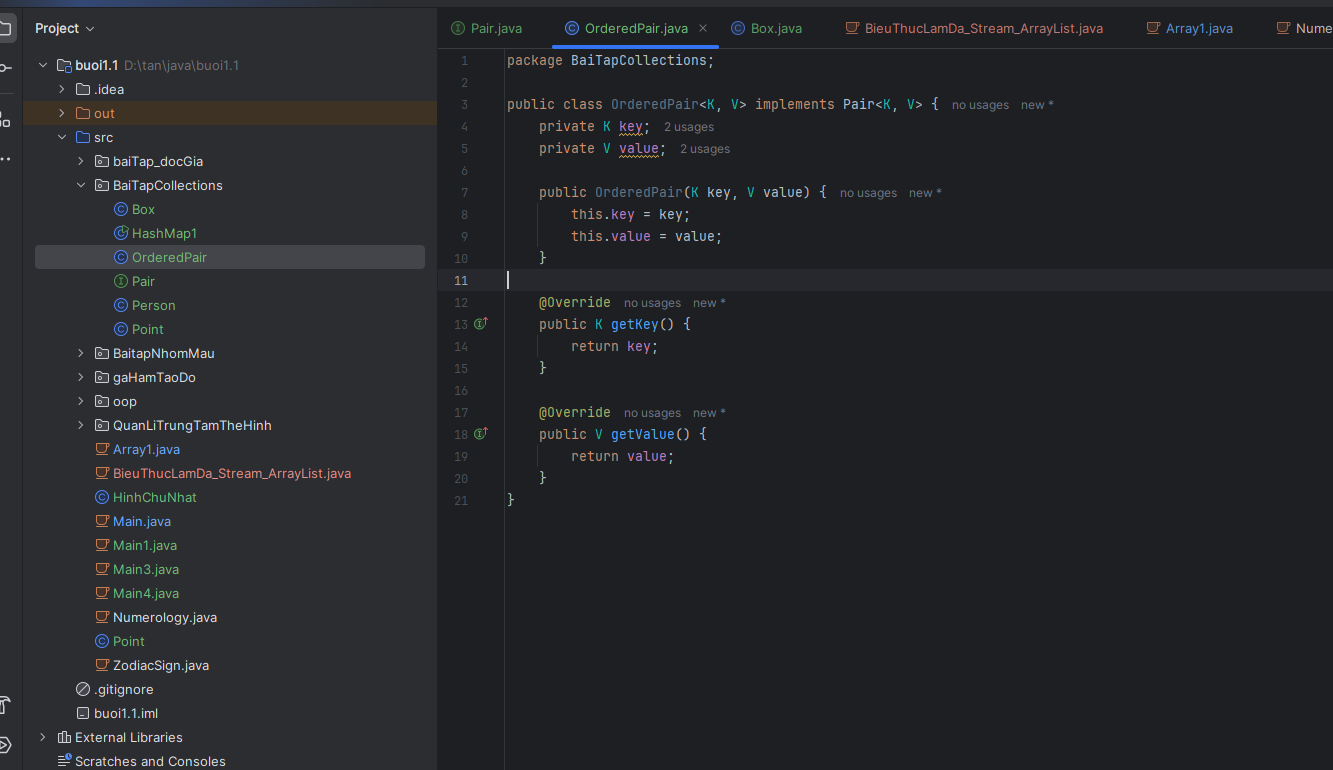
1. Khái niệm:

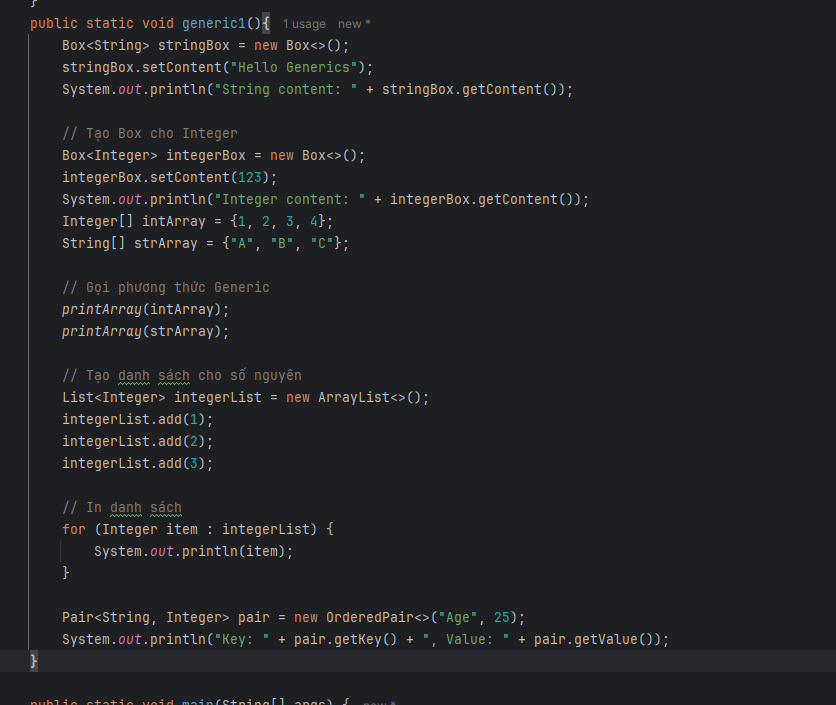
* Generics được giới thiệu từ Java 5
* Generics (generic types) là một tính năng mạnh mẽ cho
* phép bạn viết mã linh hoạt, tái sử dụng, và an toàn hơn bằng cách sử dụng kiểu dữ liệu kiểu tham số
* **Generics** giúp bạn viết mã linh hoạt và tái sử dụng hơn bằng cách cho phép bạn sử dụng kiểu tham số trong lớp, phương thức, và giao diện.
* **Generic Class** và **Generic Method** cho phép bạn làm việc với các kiểu dữ liệu khác nhau mà không phải viết mã lặp lại.

1. Nội dung









public static <T> void printArray(T[] array) {

for (T element : array) {

System.*out*.printf(element+" ");

}

System.*out*.println("");

}

4.Tổng kết:

1. Về quá trình

* **Java 2 (J2SE 1.2)**: Giới thiệu Java Collections Framework.
* **Java 5**: Giới thiệu Generics.
* **Java 8**: Giới thiệu Stream API và các phương thức default cho interface.
* **Java 9**: Giới thiệu Immutable Collections. =>các phương thức factory (List.of(), Set.of(), Map.of()) để dễ dàng tạo các tập hợp bất biến.

=>Các tập hợp bất biến rất hữu ích trong nhiều tình huống, đặc biệt khi làm việc với dữ liệu không nên thay đổi và khi bạn cần đảm bảo tính nhất quán trong ứng dụng của mình

Vd:

List<String> immutableList = List.*of*("Java", "Python", "C++");

// Không thể thêm, xóa hoặc thay đổi phần tử

// immutableList.add("JavaScript"); // Sẽ ném UnsupportedOperationException

for (String language : immutableList) {

System.*out*.println(language);

}

* **Java 11 và sau đó**: Tiếp tục cải tiến hiệu suất và hỗ trợ các tính năng mới.
* Java Collections Framework đã liên tục được cải tiến qua các phiên bản của Java, với các tính năng như generics, Stream API, và immutable collections giúp làm việc với tập hợp dữ liệu trở nên dễ dàng và hiệu quả hơn.

1. **Về Collection** framework

* **List** (Danh sách)
  + - * **ArrayList**
      * **LinkedList**
* **Set** (Tập hợp)
  + - * **HashSet**
      * **LinkedHashSet**
      * **TreeSet**
* **Queue** (Hàng đợi)
  + - * **LinkedList**
      * **PriorityQueue**
* **Map** (Bản đồ)
* **HashMap**
* **LinkedHashMap**
* **TreeMap**

1. Về tính năng  
   **Các Tính Năng Chính của Java Collections Framework**

* **Thứ Tự**: Các lớp như LinkedHashMap và TreeMap hỗ trợ duy trì thứ tự của các phần tử hoặc khóa.
* **Hiệu Suất**: Các lớp như HashMap, HashSet cung cấp hiệu suất tốt cho các thao tác như thêm, xóa và tìm kiếm.
* **Khả Năng Mở Rộng**: Hỗ trợ các cấu trúc dữ liệu mở rộng và tùy chỉnh thông qua các giao diện và lớp cơ sở.