**Java: Lập trình hàm,**

**Biểu thức Lambda, Xử lý lười trên nhóm**

1. Giới thiệu
2. Hàm toán học
3. Lập trình hàm với Java
4. Biểu thức lambda
5. Thư viện Java hỗ trợ lập trình hàm: java.util.function
6. Xử lý lười trên nhóm

**1- Giới thiệu**

Trong các chương trình máy tính, người lập trính thường phải đối diện với các hiện thực:

* Các hàm tính toán để tìm ra kết quả xứ lý từ dữ liệu đầu vào.
* Các xử lý bao gồm một chuỗi liên tiếp trên từng phần tử của một nhóm đã có,
* Trích ra một nhóm con của một nhóm đã có dựa trên một điều kiện cho trước.

Bài viết này có thể giúp các bạn hiện thực các hàm tính toán linh động, ngắn gọn hơn cũng như giớn thiệu đến các bạn một phương pháp xử lý “lười hơn- lazy seeking” trên các nhóm.

**2- Hàm toán học**

**Hàm, ánh xạ, toán học** là một xử lý trên từng phần tử của tập dữ liệu đầu vào (được gọi là miền trị của hàm – function’s domain, D) để có được duy nhất một dữ liệu đầu ra ( tập đầu ra được gọi là tập ảnh, function’s image, I). Dữ liệu đầu vào của hàm được gọi là biến hay tham số của hàm.

|  |  |  |
| --- | --- | --- |
| Số biến/ tham số | Thí dụ | Số tập hợp có liên quan |
| 1 | f: x∈Z -> y∈R, y = f(x)=x2+3.5 | 2: Z, R |
| 2 | f: (x,y)∈ ZxN -> t∈R, t = x3+ y + 0.7 | 3: Z, N, R |

Cách tìm kết quả của hàm: Từ tham số đầu vào, biểu thức của hàm phải được tính toán (evaluate/ eval) để có được kết quả.

**Hàm vị từ- Predicate**: Trong lý thuyết ngôn ngữ học, vị từ/ vị ngữ/ thuộc ngữ là cụm từ mô tả tính chất, hoạt động của chủ từ. Một câu (statement, proposition, mệnh đề) mô tả 1 sự kiện có cấu trúc

**cchủ từ + vị từ.**

Thí dụ: Helen is beautiful. Chủ từ: “Helen”, vị từ: “is beautiful”.

Toán logic bậc nhất/ toán mệnh đề (propositional logic, logic mệnh đề) giúp xác định đúng sai của mệnh đề cùng với sự kết hợp của các mệnh đề bằng các liên từ (LOGIC OPERATORS: AND/OR/NOT).

Như vậy, predicate là một hàm **f: Object -> B (**B: Boolean set)

Vị từ là nền tảng toán học của các ngôn ngữ lập trình logic, thí dụ Prolog.

**3-Lập trình hàm với Java**

**Lập trình hàm** (functional programming) là một phương pháp lập trình mà một đơn vị của chương trình là một hàm. Một chương trình là một chuỗi kết hợp của các hàm được thực thi. Hàm thực thi trước có thể là tham số cho hàm thực thi sau đó. Một số ngôn ngữ hỗ trợ lập trính hàm: Python, Lisp,Java script, Clojure, Elm, Haskell,…

Lập trình hàm đã trở thành một xu hướng lập trình hiện dại. Nhiều ngôn ngử lập trình chỉ thị bao gồm cả lập trình hướng đối tượng (imperative/ OO programming language) như Java (từ JDK 8), C++, C3, Python,… cũng đã hỗ trợ lập trình hàm để người dùng có thể kết hợp các tác vụ được diễn đạt bằng lập trình hàm vào code chung của chương trình.

**Các bước lập trình hàm trong Java (xem lại *minh họa 01*)**

Bước 1: Tạo một interface có chứa CHỈ MỘT HÀNH VI TRỪU TƯỢNG

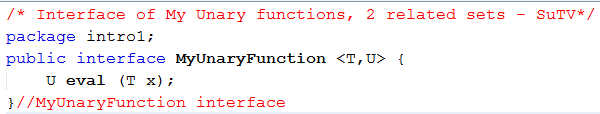
Dĩ nhiên interface này có thể có thêm một số hành vi cụ thể. Interface này trở thành khai báo cho một họ các hàm có cùng tham số và cùng kiểu dữ liệu đầu ra.

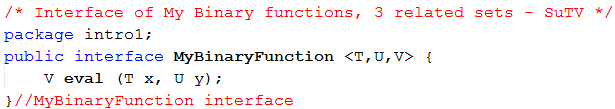
Bước 2: Sử dụng interface này như một kiểu hàm

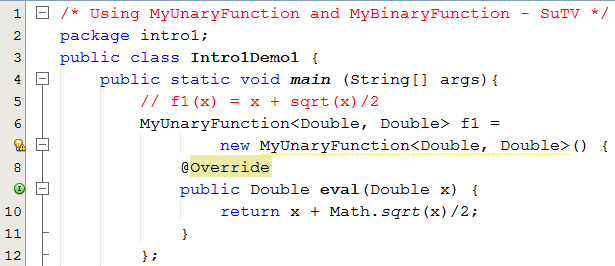
Cụ thể hóa hành vi trừu tượng bằng một lớp hoặc sử dụng lớp vô danh để cụ thể hóa hành vi trừu tượng của interface nảy.

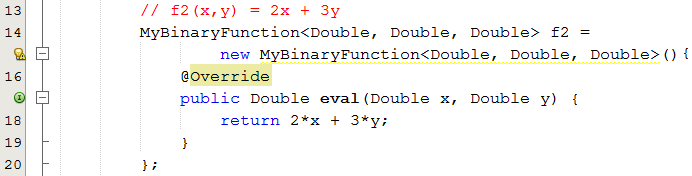
**Minh họa 1: Intro1Demo1.class**

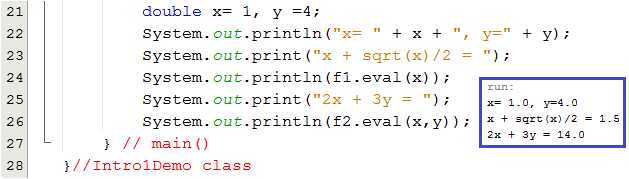
Trong minh họa dưới đây, hàm một biến (2 tập hợp có liên quan) được khái quát hóa bằng interface MyUnaryFunction, hàm 2 biến (3 tập hợp có liên quan) được khái quát hóa bằng interface MyBinaryFunction. Tring main(), sử dụng đối tượng thuộc lớp vô danh để định nghĩa trực tiếp hai hàm.











**Nhận xét**

* Với mỗi hàm khác nhau trong cùng một nhóm (thí dụ hàm một biến), để sử dụng, chúng ta cần implement lại interface tương ứng bằng code. Câu hỏi đặt ra là: “Có cách nào viết code đơn giản hơn không?” – Trả lời: Có, dùng biểu thức lambda (được trình bày ở dưới) sẽ giúp hiện thức hàm đơn giản hơn.

**Cách Java biên dịch hàm bằng kỹ thuật lớp vô danh**

Mỗi đối tượng thuộc lớp vô danh (minh họa trên là một ví dụ) sẽ được trình biên dịch Java tạo ra một lớp RIÊNG có tên là MainClass$x trong đó x là số thự tự lớp vô danh được trình biên dịch phát hiện khi dịch chương trình nguồn. Điều này đã được giới thiệu trong phần lập trình Java cơ bản.

**4- Biểu thức Lambda**

**Biều thức lambda trong toán học:** Lambda calculus, λ-calculus, là một hệ thống toán học giúp mô tả hình thức (khái quát hóa) cho một hàm. λx.f mang ý nghĩa là có một biểu thức f giúp thay thế trị của x để cho kết quả của biều thức.

***Thí dụ***:

λx.f : x 🡪 5x+3. Khi cho x = 2, phép thế trị cho kết quả là 13. Để diễn đạt ngắn gọn, chúng ta gọi biểu thức 5x+3 là f.

λ(x,y).f : (x,y) 🡪 5x+y. (1,2) 🡪 7

**Biều thức lambda trong lập trình máy tính:** Biểu thức Lambda được nhiều ngôn ngữ lập trình hỗ trợ (Java hỗ trợ biểu thức lambda từ phiên bản JDK 8), bao gồm code thư viện, cú pháp, cơ chế dịch, để người dùng đã quen với cách diễn đạt tính toán bằng biểu thức lambda có thể dùng trực tiếp trong code chương trình với cú pháp tương tự như diễn đạt của các hàm toán học. Cú pháp Lambda trở thành công cụ chính trong phương pháp lập trình hàm. Nhìn chung, biểu thức Lambda trong ngôn ngữ lập trình vẫn giữ nguyên cú pháp chính của lambda toán học:

(tham số cách nhau dấu phẩy) -> biểu thức trả trị;

(tham số cách nhau dấu phẩy) -> { nhóm phát biểu };

()-> { nhóm phát biểu }; // lambda không tham số

Thí dụ 1: () -> { System.out.println(“Hello.”);

System.out.println(“From Lambda!”); };

Thí dụ 2: (x) -> 5\*x + sqrt(x)/2;

Thí dụ 3: (x, y) -> 5\*x + y;

Thí dụ 4: (x, y, z) -> (5\*x + y)/z;

**Ích lợi của biều thức lambda trong lập trình máy tính:**

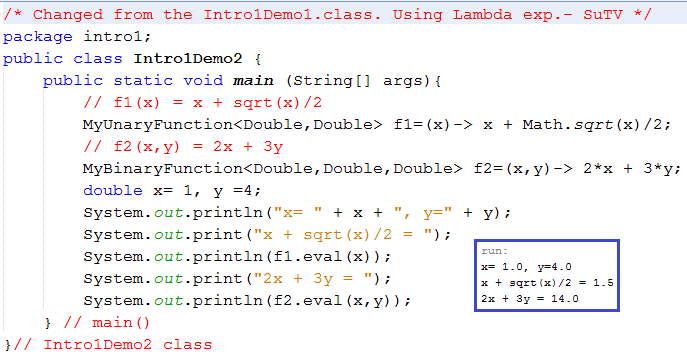
* Phục vụ cách phân tích chương trình dựa trên các hàm tính toán. Việc cụ thể hóa một hàm rất linh động vì người dùng có thể tự định nghĩa một hàm mà không cần phải tường minh khai báo tên hàm (hàm ẩn danh).

**Hạn chế của biều thức lambda trong lập trình máy tính:**

* Cú pháp lambda hơi khó hiểu đối với những người chưa quen thuộc với nó do đã quên cách biểu diễn hàm trong toán học. Do vậy, lambda expression có thể gây ra những khó khăn nhất định trong nhóm phát triển phần mềm (code/ review).
* Mỗi ngôn ngữ có các hỗ trợ biều thức Lambda khác nhau nên cần nắm vững cú pháp lambda của ngôn ngữ.
* Người lập trình quá quen thuộc với lập trình chỉ thị nên việc chuyển dạng diễn đạt sang cú pháp lambda có thể gặp khó khăn nhất định..

**Minh họa 2: Intro1Demo2.class**

Trong minh họa này, Intro1Demo1.class được viết lại để dùng cú pháp lambda.



**Cách Java biên dịch các biểu thức Lambda**

Java vẫn phải dùng kỹ thuật lớp vô danh để biên dịch Java các biểu thức Lambda NHƯNG KHÔNG TẠO RA LỚP VÔ DANH RIÊNG CHO BIỂU THỨC LAMBDA MÀ THÔNG TIN VỀ LỚP VÔ DANH NÀY ĐƯỢC ĐỂ TRONG LỚP CHỨA NÓ. Các bạn có thể kiểm tra các tập tin kết quả biên dịch.

**Những định nghĩa và khuyến nghị về lập trình hàm**

**Các khái niệm chính trong lập trình hàm:**

1. **Function**: Hàm là đối tượng cơ bản.
2. **Pure Functions**: Hàm thuần túy là hàm chỉ nhận tham số đầu vào từ bên ngoài để xứ lý nhưng không làm thay đổi tham số này cũng như các biến bên ngoài hàm.
3. **Higher order functions:** Hàm có thứ bậc cao hơn là hàm có tham số đầu vào lại là kết quả của một hàm khác. Nghĩa là các hàm lồng nhau.

**Các quy tắc NÊN tuân thủ để có hàm thuần túy (pure function)**

1. **No state**: Hàm thuần túy là hàm **không trạng thái** nghĩa là hàm này độc lập với các dữ liệu (biến) bên ngoài hàm. Các code xử lý tham số rồi sinh ra kết quả mới là XONG. Mỗi hàm nên trả trị để đầu ra của hàm này có thể là đầu vào của hàm sau.
2. **No side effects**: Không gây hiệu ứng lề nghĩa là hàm **không làm thay đổi các biến bên ngoài hàm**.
3. **Immutable variables**: Các biến phải có tính **bất biến nghĩa là hàm không làm thay đổi tham số**. Chính đặc điểm này làm cho hàm trở thành “thuần túy- pure”.
4. **Favour recursion over looping**: Nên dùng kỹ thuật đệ quy để diễn đạt tác vụ lặp..

Các minh họa sẽ được giới thiệu ở phần sau.

**5- Thư viện Java hỗ trợ lập trình hàm: java.util.function**

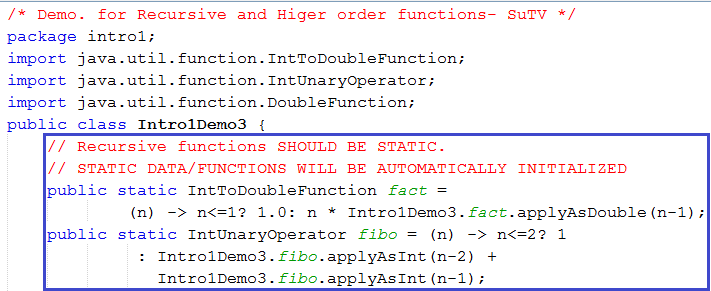
Gói thư viện java.util.function cung cấp rất nhiều các interface cho chúng ta sữ dụng trong lập trình hàm. Chi tiết về các interface này có thể được xem trong Java ducumetation.

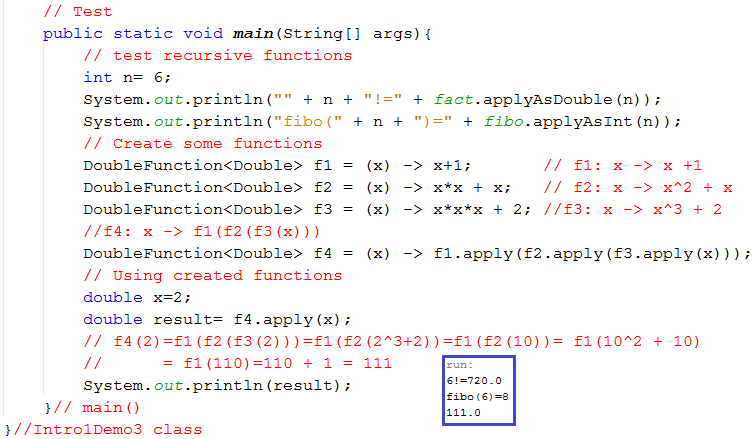
***Các interface cơ bản trong gói java.util.function***

|  |  |  |
| --- | --- | --- |
| **Nhóm Interfaces** | **Hàm** | **Giải thích** |
| Function<T,R>  BiFunction<T,U,R> | f: T 🡪 R  f: T, U 🡪 R | R apply (T)  Hành vi đã hiện thực cho việc kết hợp các hàm:  andThen(…), compose(…) |
| IntFunction  LongFunction DoubleFunction | f: int🡪 <R>  f: long 🡪 <R>  f: double 🡪 <R> | Khai báo có dạng:  public interface **LongFunction<R>**  Tên hành vi trừu tượng: R apply(param)  Input: Param thuộc kiểu int/long/double tương ứng.  Output: Có kiểu số nào đó R (cần ép kiểu khi sử dụng) |
| IntToLongFunction  IntToDoubleFunction  LongToIntFunction  LongToDoubleFunction  DoubleToIntFunction  DoubleToLongFunction  ToIntFunction<T>  ToLongFunction<T>  ToDoubleFunction<T>  ToIntBiFunction<T,U>  ToLongBiFunction<T,U>  ToDoubleBiFunction<T,U> | f: int🡪long  f: int 🡪 double  f: long 🡪 int  f: long 🡪 double  f: double 🡪 int  f: double 🡪 long  f: Type 🡪 int  f: Type 🡪 long  f: Type 🡪 double  f: T,U 🡪 int  f: T,U 🡪 long  f: T, U 🡪 double | Interface cho các hàm chuyển kiều  ResultType applyAsXXX(paraType)  ResultType applyAsXXX(T, U) |
| UnaryOperator<T,R>  BinaryOperator<T,T,T>  IntUnaryOperator  IntBinaryOperator  LongUnaryOperator  LongBinaryOperator  DoubleUnaryOperator  DoubleBinaryOperator | Op: T 🡪 R  Op: T, T 🡪 T  Op:int🡪int  Op: int, int 🡪 int  … | Interface cho các toán tử một/ hai toán hạng  ResultType applyAsXXX(paramType)  ResultType applyAsXXX(paraType left, paraType right)  Hành vi đã hiện thực cho việc kết hợp các hàm: andThen(…), compose(…) |
| Predicate<T>  BiPredicate<T,U> IntPredicate  LongPredicate  DoublePredicate | f: Type 🡪 Boolean  f: T,U 🡪 Boolean  f: Int 🡪 Boolean  f: Long 🡪 Boolean  f: Double 🡪 Boolean | Hành vi trừu tượng: boolean test (param)  Các hành vi đã cụ thể mô tả toán tự vị từ :  and (…)/ or()/ negate()/ isEqual() |
| Supplier<T>  BooleanSupplier  IntSupplier  LongSupplier  DoubleSupplier |  | Cung cấp khả năng hiện thực cách truy xuất một phần tử từ một dòng dữ liệu đã có.  Hành vi trừu tượng:  T get(void)  Type getAsXXX(void) |
| Consumer<T>  IntConsumer  LongConsumer  DoubleConsumer  BiConsumer<T,U>  ObjDoubleConsumer<T>  ObjIntConsumer<T>  ObjLongConsumer<T> |  | Cung cấp khả năng hiện thực một xử lý nào đó khi nhận được một đối tượng từ một dòng dữ liệu đã có.  Hành vi trừu tượng:  void accept (T obj)  void accept (T obj1, U obj2) |

**Minh họa 3: Intro1Demo03.class**

Minh họa cách hiện thực hàm đệ quy và hàm lồng nhau- Intro1Demo3.class





**6- Xử lý lười trên nhóm**

Khi thao tác trên nhóm, chúng ta thường phải đối diện với các tác vụ: Thêm phần tử, tìm phần tử, xóa phần tử, cập nhật phần tử, duyệt danh sách. Ngoài các tác vụ thêm/xóa phần tử, các tác vụ **trung gian** khác như duyệt, trích lọc, xử lý dữ liệu của mỗi phần tử được kháo quát và mô hình hóa thành các interface và các lớp hỗ trợ trong thư viện **java.util.stream.** Nhờ các công cụ này, các thao tác trên nhóm được hiện thực một cách nhẹ nhàng hơn- người lập trình có cơ hội “lười hơn – lazy seeking, lazy programming”.

Các công cụ thư viện này xoay quanh một interface trung tâm với khai báo **java.util.stream.Stream** giúp thao tác lên nhóm dữ liệu đầu vào theo dạng đường ống (pipelining).

**6.1- Khái niệm căn bản về dòng- stream**

***Dòng - Stream****: Một cơ chế thao tác MỘT CHIỀU lên một kho chứa theo từng đơn vị dữ liệu.*

|  |  |  |  |
| --- | --- | --- | --- |
| ***Kho chứa*** | ***Ví tiền*** | ***Bàn phám*** | ***Khối code*** |
| Thao tác | Tiều xài theo từng tờ | Lấy từng đơn vị dữ liệu kết thúc bằng phím ENTER | Lấy từng câu lệnh |

*Minh họa một số stream*

**Đặc điểm của mô hình stream**:

Stream tương tự như một dây chuyền sản xuất bao gồm một dãy (sequence) các đối tượng làm việc theo cơ chế đường ống giúp xử lý nhóm dữ liệu đầu vào với các tính chất sau:

- Duyệt từng đối tượng trong tập dữ liệu nguồn (collection) thông qua bản sao của từng phần tử (dữ liệu nguồn không đổi – immutable).

- Thực thi tuần tự các xử lý trên bản sao của dữ liệu nguồn (bảo tồn dữ liệu nguồn) để cho từng kết quả. Kết quả của xử lý trước là đầu vào của xử lý sau.

- Khi đã thao tác xong, stream sẽ đóng lại và mọi thao tác sau khi streamn đã đóng sẽ gây lỗi lúc thực thi.

**Một thí dụ:**

Giả sử chúng ta cần 3 thao tác trên tập dữ liệu nguồn, P1, P2, P3. Cơ chế xử lý đường ống được tiến hành như sau:

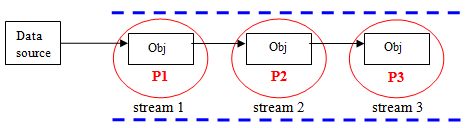
Bước 1: Khởi tạo stream từ Data source

Bước 2: Lặp copy từng đối tượng từ data source để xử lý theo trình tự:

2.1- Thực thi tác vụ P1.

2.2- Thực thi tác vụ P2.

2.3- Thực thi tác vụ P3.



*Minh họa cơ chế xử lý tuần tự trong đường ống các stream*

**Công dụng của Stream**: Thư viện về stream trong các ngôn ngữ lập trình đã cài đặt rất nhiều các tiện ích thao tác trên nhóm giúp giảm bớt công sức lập trình (lazy seeking) chẳng hạn như: Cập nhật dữ liệu/Trích lọc các phần tử thỏa điều kiện cho trước/ Loại bỏ các phần tử trùng lắp/…

**6.2- Dòng tuần tự và dòng song song**

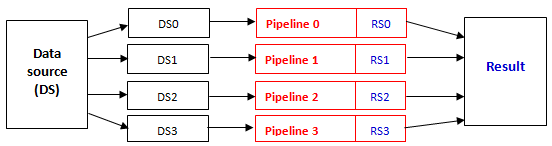
**Dòng đơn tuần tự (sequentail stream)**: Data source chỉ được xử lý bởi một đường ống. Dòng tuần tự thường là dòng mặc định.

**Dòng song song (parallel stream)**: Dòng sẽ được nhân bản thành nhiều stream chạy trên môi trường CPU đa nhân đa luồng. Data source được cắt ra thành nhiều phần, mỗi phần sẽ được xử lý bởi một stream riêng. Dỏng song song có thể được dùng khi tập dữ liệu đầu vào quá lớn (vài triệu phần tử chẳng hạn). Việc nhân bản dòng (có mấy dòng) và phân chia tập dữ liệu đầu vào thường không do người lập trình quyết định.

**Chi phí cơ bản khi dùng dòng song song**

1. Chi phí phân chia tập dữ liệu ban đầu thành các tập con ( chi phí chia ArrayList < TreeMap, HashSet < LinkedList)
2. Chi phí cho việc nối tập kết quả.

Nhiều khi chi phí cơ bản nêu trên này lại lớn hơn chi phí của cơ chế stream tuần tự.



*Hình minh họa dòng song song 4 luồng*

**6.3- Gói thư viện java.util.stream**

***Các interface cơ bản trong gói java.util.stream***

|  |
| --- |
| **Class Hierarchy**   * java.lang.[**Object**](file:///C:\Program%20Files\Java\docs\api\java\lang\Object.html)   + java.util.stream.[**Collectors**](file:///C:\Program%20Files\Java\docs\api\java\util\stream\Collectors.html) – Lớp gom dữ liệu sau xử lý   + java.util.stream.[**StreamSupport**](file:///C:\Program%20Files\Java\docs\api\java\util\stream\StreamSupport.html) – Lớp hỗ trợ giúp khởi tạo stream   **Interface Hierarchy: Interface cho các khả năng**   * java.lang.[**AutoCloseable**](file:///C:\Program%20Files\Java\docs\api\java\lang\AutoCloseable.html) – Khả năng tự động đóng sau khi hoàn tất tác vụ   + java.util.stream.[**BaseStream**](file:///C:\Program%20Files\Java\docs\api\java\util\stream\BaseStream.html)<T,S> – Stream cơ bản     - java.util.stream.[**DoubleStream**](file:///C:\Program%20Files\Java\docs\api\java\util\stream\DoubleStream.html) – Khả năng của stream chứa các dữ liệu double     - java.util.stream.[**IntStream**](file:///C:\Program%20Files\Java\docs\api\java\util\stream\IntStream.html) – Khả năng của stream chứa các dữ liệu int     - java.util.stream.[**LongStream**](file:///C:\Program%20Files\Java\docs\api\java\util\stream\LongStream.html) – Khả năng của stream chứa các dữ liệu long     - java.util.stream.[**Stream**](file:///C:\Program%20Files\Java\docs\api\java\util\stream\Stream.html)<T> – Khả năng của Stream chứa các dữ liệu object * java.util.stream.[**Collector**](file:///C:\Program%20Files\Java\docs\api\java\util\stream\Collector.html)<T,A,R> – Khả năng của các đối tượng đi gom dữ liệu * java.util.function.[**Consumer**](file:///C:\Program%20Files\Java\docs\api\java\util\function\Consumer.html)<T> – Khả năng xử lý dữ liệu nhận được   + java.util.stream.[**Stream.Builder**](file:///C:\Program%20Files\Java\docs\api\java\util\stream\Stream.Builder.html)<T>– Khả năng copy data source vào buffer của stream * java.util.function.[**DoubleConsumer**](file:///C:\Program%20Files\Java\docs\api\java\util\function\DoubleConsumer.html) – Khả năng xử lý dữ liệu nhận được là một double   + java.util.stream.[**DoubleStream.Builder**](file:///C:\Program%20Files\Java\docs\api\java\util\stream\DoubleStream.Builder.html) – Khả năng tạo stream các số double * java.util.function.[**IntConsumer**](file:///C:\Program%20Files\Java\docs\api\java\util\function\IntConsumer.html) – Khả năng xử lý dữ liệu nhận được là một int   + java.util.stream.[**IntStream.Builder**](file:///C:\Program%20Files\Java\docs\api\java\util\stream\IntStream.Builder.html) – Khả năng tạo stream các số int * java.util.function.[**LongConsumer**](file:///C:\Program%20Files\Java\docs\api\java\util\function\LongConsumer.html) – Khả năng xử lý dữ liệu nhận được là một long   + java.util.stream.[**LongStream.Builder**](file:///C:\Program%20Files\Java\docs\api\java\util\stream\LongStream.Builder.html) – Khả năng tạo stream các số long |

Vì mỗi lớp mô tả nhóm có cấu trúc lưu trữ khác nhau nên mỗi lớp quản lý nhóm (ArrayList, TreeMap, HashSet, LinkedList) đều đã có hành vi tạo stream phù hợp đồng thời các hành vi được khai báo trong các interface nên trên đã được cụ thể hóa.

***Các thao tác thông dụng trên stream:***

Các tác vụ trên stream được chia thành 3 loại: tác vụ kết thúc/ trung gian/ optional.

|  |  |
| --- | --- |
| Tạo stream từ nhóm dữ liệu | Dùng hành vi ***stream(), parallelStream()*** của lớp thư viện thuộc nhóm Collection (ArrayList, Vector, LinkedList, TreeSet ). Mỗi loại Collection có cấu trúc tổ chức vật lý riêng nên cơ chế tạo Stream cũng khác nhau. |
| Các tác vụ kết thúc (terminal operations) | Tác vụ kết thúc là các tác vụ sau khi thực thi sẽ đóng. Thí dụ  Lấy số phần tử:count(),  Lấy cực trị: max(), min()  Gom trị như tính tổng, nối chuỗi, …: reduce(…)  Gom trị kết quả: toArray(),collect()  Duyệt từng phần tử: iterator(), forEach(), forEachOrdered() |
| Các tác vụ không kết thúc (intermediate, non-terminal operations) | Tác vụ không kết thúc là các tác vụ sẽ trả về một stream mới để stream mới có thể xử lý tiếp trong đường ống. Các tác vụ không kết thúc thông dụng:  Chuyển dạng dữ liệu (map),  Trích lọc phần tử (filter) dựa trên một điều kiện (hàm Predicate)  Chia dữ liệu của phần tử thành các phần nhỏ hơn, split chuỗi, (flatMap)  Loại bỏ các phần tử trùng lắp (distinct)  Hạn chế số phần tử sẽ được lấy (limit) |
| Các tác vụ optional | Các tác vụ này trả về một đối tượng Optional giúp có thể kiểm tra có thể có trong các hiện thực khác nhau. Thí dụ: findXXX(), parallel(), peek(),… |

Các lớp **java.util.Optional, java.util.OptionalInt, java.util.OptionalLong, java.util.OptionalDouble** giúp quản lý tình trạng **null** của các đối tượng trong đó cho phép gán trị mặc định trong tình huống tham chiếu null.

***Cú pháp Java khi truy xuất stream:***

Câu lệnh diễn đạt đường ống với 3 tác vụ trong một dòng:

**streamName.operation1(param).operation2(param).operation3(param);**

***Ý nghĩa của câu lệnh***: streamName thực hiện tác vụ operation1 trên dữ liệu nguồn, trả về một đối tượng đầu ra là một stream thứ hai. Stream thứ hai thực hiện tác vụ operation2 trên dữ liệu được cung cấp từ stream trước đó và trả về stream thứ 3 ….

Để dễ đọc , người ta thường viết câu lệnh đường ống này theo từng dòng, mỗi dòng lệnh là một xử lý. Đừng quên dấu ‘;’ ở xử lý cuối.

ResultType result = (TypeCastIfNeeded)streamName

.firstOperation1(param)

.secondOperation2(param)

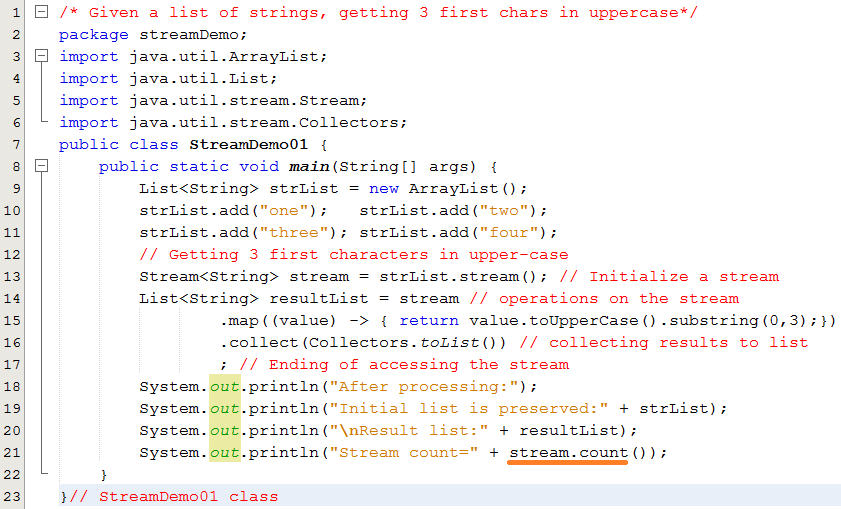
. . .

.lastOperation**;** // last operation will be result value

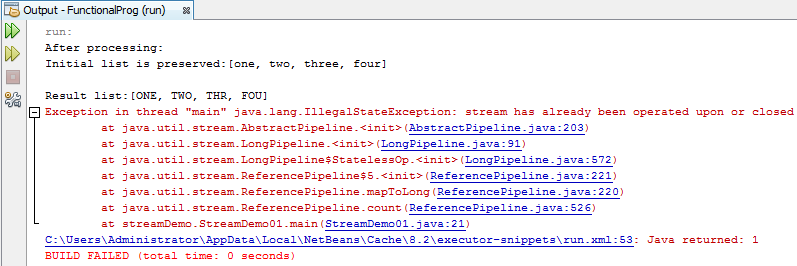
**6.4- Các Minh hoa lập trình với Stream**

***Minh họa 4:***

Dùng Stream để trích ra 3 ký tự đầu dạng chữ hoa trong một danh sách các chuỗi.



**Kết quả:**

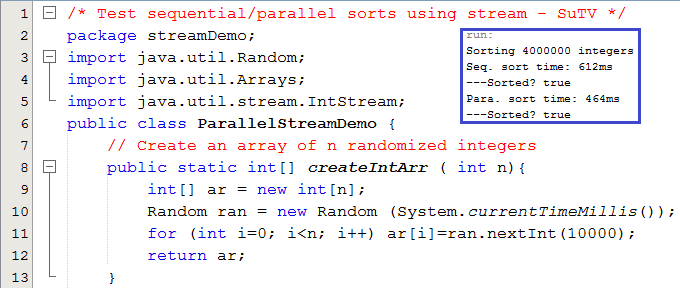


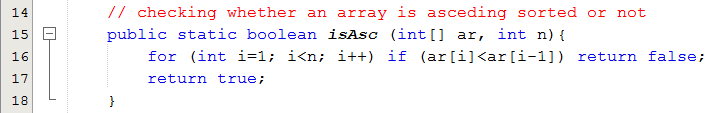
***Chương trình trên đã minh họa:***

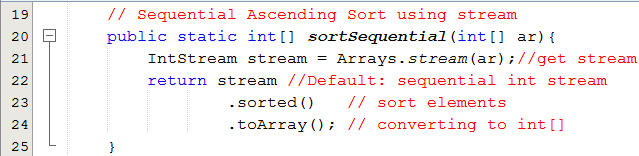
* Stream xử lý dữ liệu để cho kết quả mới, bảo tồn dữ liệu ban đầu.
* Sau khi khởi tạo Stream (dòng 13), các thao tác của stream được tiến hành (dòng 14, 15, 16) và kết thúc có tự đóng stream (dấu ‘;’ cuối dòng 17).
* Sau khi dòng đã đóng, truy xuất stream sẽ gây lỗi lúc thực thi (dòng 21).

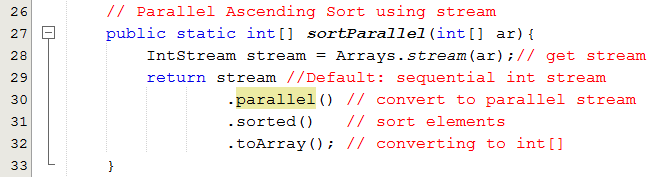
***Minh họa 5:***

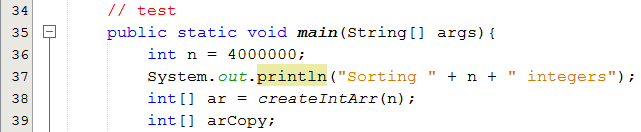
Đo thời gian sắp xếp một mảng int có 4 triệu phần tử sử dụng IntStream tuần tự và IntStream song song.

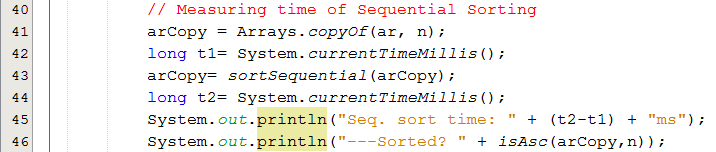


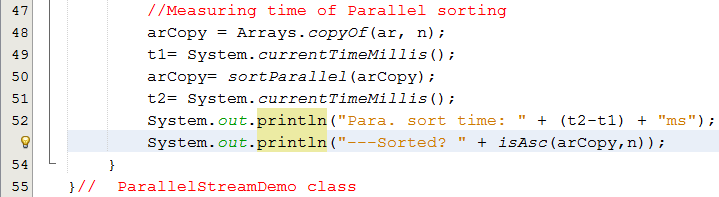










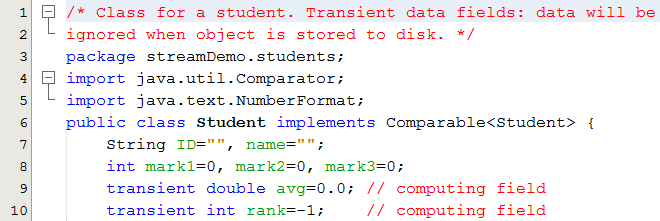


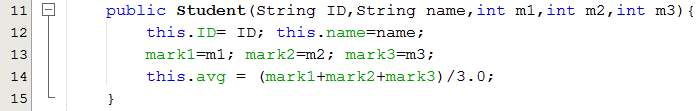
**Nhận xét**:

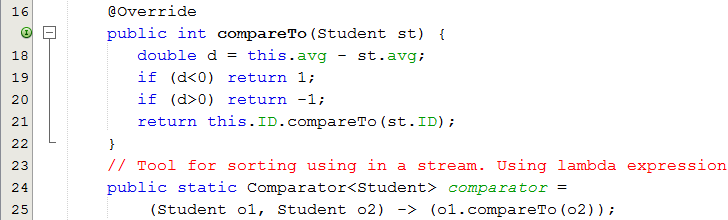
* Số luồng sẽ dùng, thứ tự các luồng thực thi, cách cắt dữ liệu nguồn do máy ảo quyết định
* Khi dữ liệu ít, đừng dùng luồng song song vì chi phí overhead không đáng phải tiêu tốn.

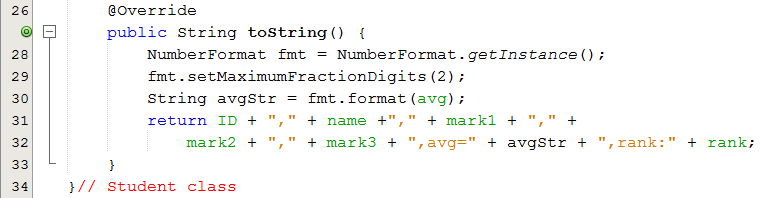
***Minh họa 6:***

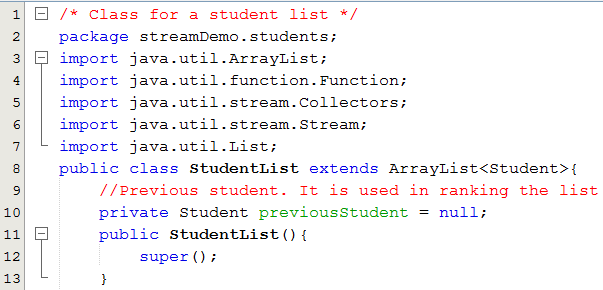
Dùng Stream để sắp xếp, xếp hạng , trích ra danh sách khen thưởng cho một danh sách học sinh.

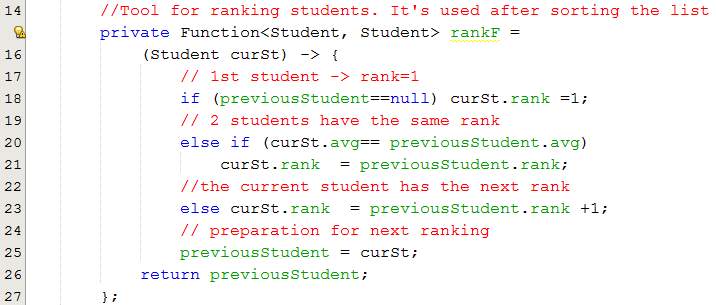


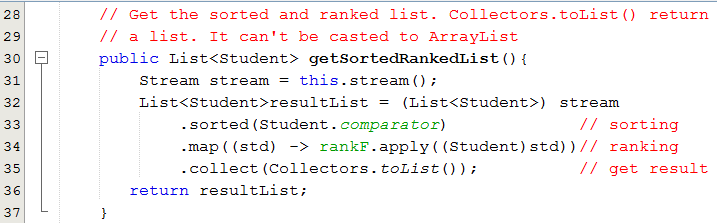


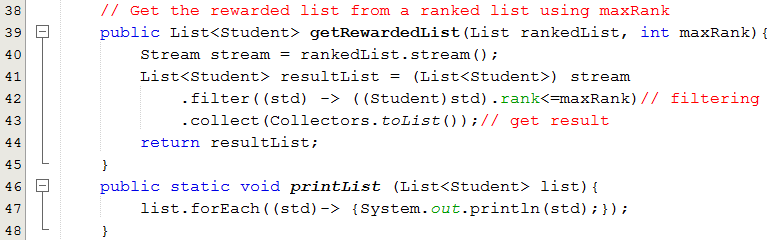


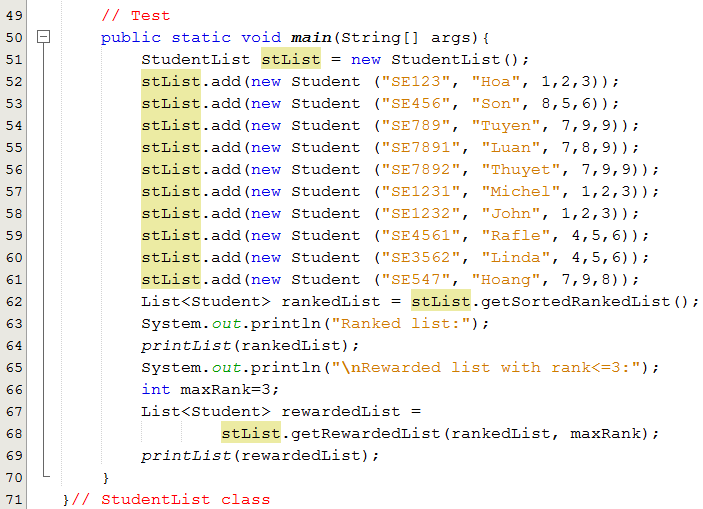




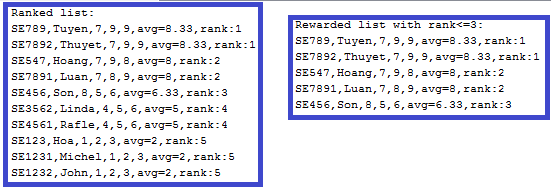






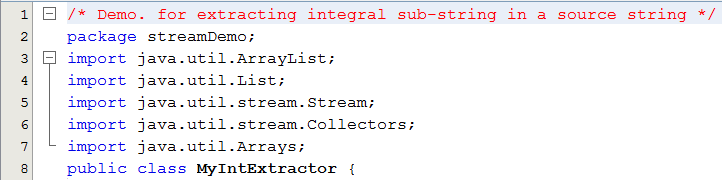


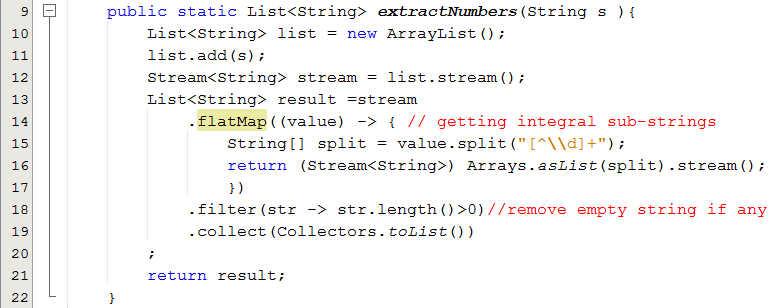
**Kết quả**

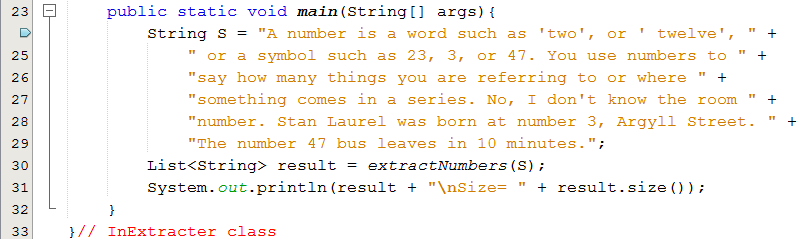


***Minh họa 7:***

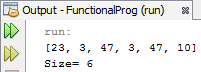
Dùng Stream để trích ra các số nguyên trong một văn bản.





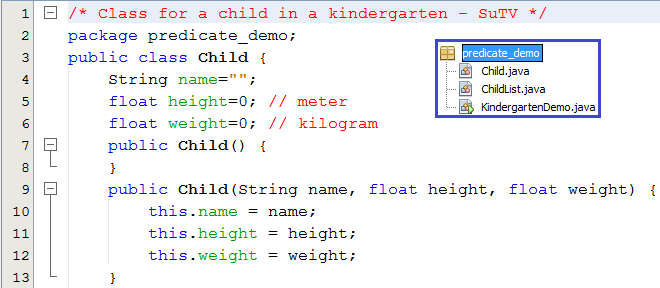


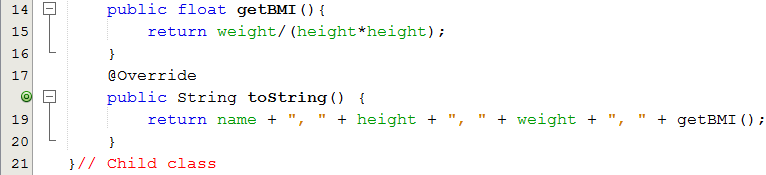
**Result:**

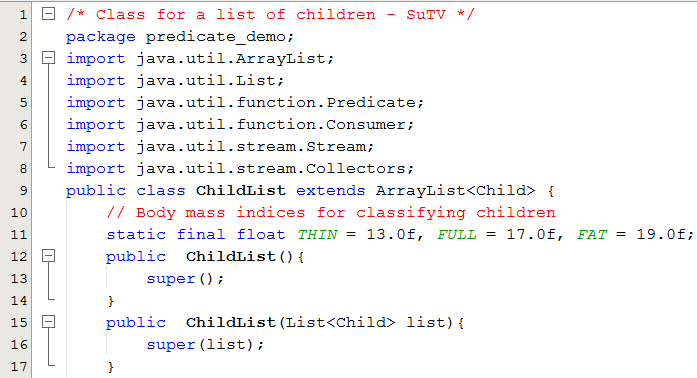
****

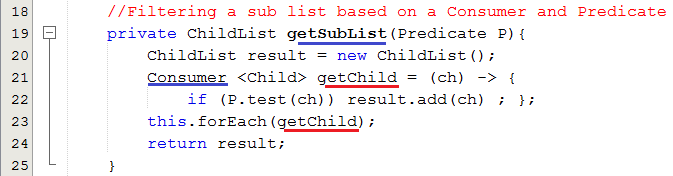
***Minh họa 8:***

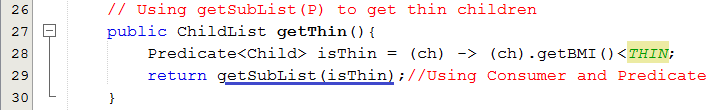
Trích lọc danh sách với interface Predicate/ Consumer và Stream. Môi trường: nhà trẻ.

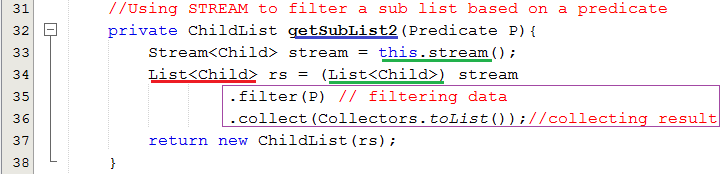


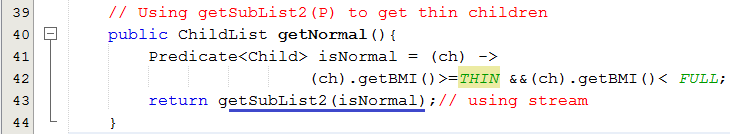


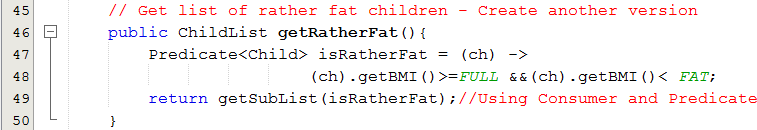


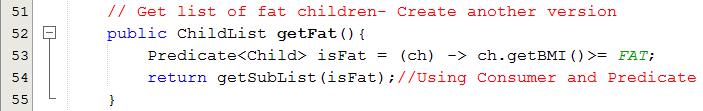


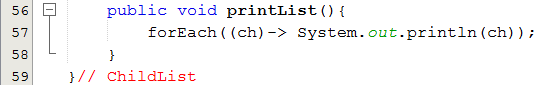


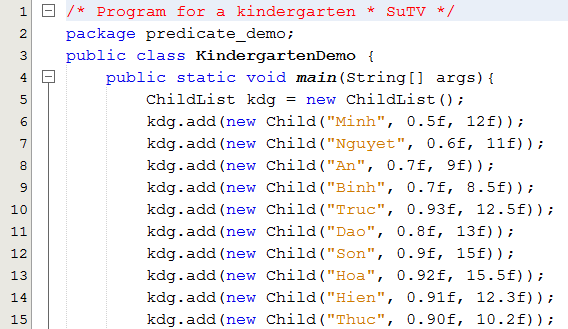


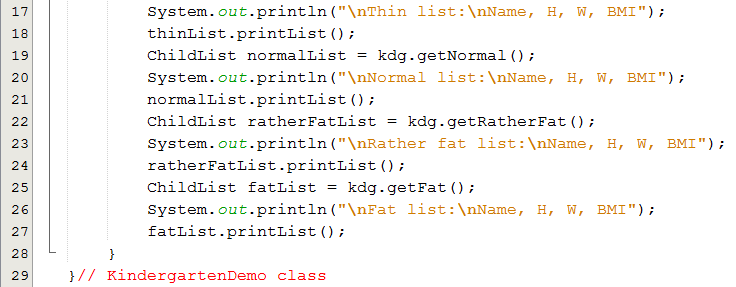












**Kết quả**

