1. OOP – Hướng đối tượng
   1. Đối tượng (Object)

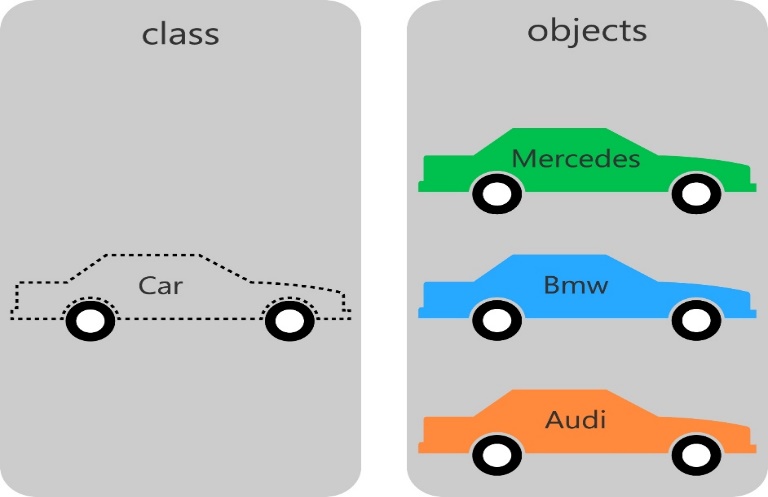
* Đối tượng có thể hiểu là những khái niệm bên ngoài như: Con người, Xe máy, Nhà cửa,…
* Một đối tượng sẽ bao gồm thuộc tính và phương thức.
* Thuộc tính: Là những thông tin riêng của đối tượng. VD: Đối tượng con người có họ tên, giới tính, quê quán,…
* Phương thức: Là những thao tác, hành động mà đối tượng có thể thực hiện được. Phương thức giống hàm trong lập trình hướng cấu trúc nhưng phương thức chỉ được khai báo trong class.

VD: Đối tượng con người có thể chạy, nhảy, ăn, uống,…

* Kiểu dữ liệu nguyên thủy (boolean, char, byte, short,..) khi ta thay đổi giá trị của một biến thì các biến còn lại không thay đổi giá trị.
* Kiểu dữ liệu tham chiếu (array, list, Object,…hoặc class) khi ta thay đổi giá trị của một biến thì các biến còn lại sẽ thay đổi giá trị theo

1.2 Lớp (Class)

* Lớp là người dùng định nghĩa cho của đối tượng, ta sẽ xây dựng lớp để tạo ra những đối tượng khác nhau.



* Nói cách khác class là bản thiết kế để tạo ra các đối tượng. Mà các đối tượng đều phải có những thuộc tính của class
* Cú pháp khai báo class

<Pham\_vi\_truy\_cap> **class** <ten\_lop>

{

<Pham\_vi\_truy\_cap> <Cac\_thanh\_phan\_cua\_lop>

}

Trong đó :

<Pham\_vi\_truy\_cap> : Là các mức độ truy cập: Public, Protected, Private,…

<ten\_lop> : Là tên của class

<Cac\_thanh\_phan\_cua\_lop> : Là biến hoặc phương thức trong lớp

1.2.1 Phương thức

- Là những thao tác, hành động mà đối tượng có thể thực hiện được. Phương thức giống hàm trong lập trình hướng cấu trúc nhưng phương thức chỉ được khai báo trong class.

- Cú pháp:

<Pham\_vi\_truy\_cap> <tu\_khoa> <du\_lieu\_tra\_ve> <name> (Tham\_so){

<Chuong\_trinh>

}

Trong đó:

+ <Pham\_vi\_truy\_cap> : public, protected, private,.

+ <tu\_khoa> : Gồm các từ khóa final, static,…

+ <du\_lieu\_tra\_ve> : Trả về kiểu int, long, double, hoặc tên 1 class nào đó. Nếu không có trả về ta dùng kiểu void. Để trả về dữ liệu ta dùng từ khóa return trong phương thức

+ <name> : Tên của phương thức

+ (Tham\_so) : Là những tham số muốn truyền vào phương thức

+ <Chuong\_trinh> : Các chương trình code trong phương thức

1.2.2 Phương thức khởi tạo

- Khi ta tạo 1 đối tượng Persion a = **new** Persion(); thì phương thức khởi tạo của đối tượng Persion(); sẽ cùng tên với class.

- Ta thường phải chỉnh sửa phương thức khởi tạo cho hợp với yêu cầu.

- Sau khi khởi tạo ta cần phải gán giá trị tương ứng cho các đối tượng a

- Tuy nhiên trong quá trình phát triển phần mềm người lập trình chưa chắc có thể biết hết toàn bộ code bên trong class. Vì vậy khi khởi tạo một đối tượng ta cần cung cấp thông tin về đối tượng đó ngay.

VD:

//Phương thức khởi tạo

**public** Persion(String name, **int** age, **float** height) {

**this**.name = name; //this.name là để gọi đến biến name của class dùng để phân biệt biến name của phương thức

**this**.age = age;

**this**.height = height;

}

//Khai báo đối tượng

Persion a = **new** Persion("Quang",26,1.68f);

* 1. Các phạm vi truy cập trong lập trình hướng đối tượng
* Có 4 loại phạm vi truy cập: Private, (Default), Protected, Public
  + 1. Private
* Chỉ cho phép truy cập nội bộ trong 1 class
  + 1. Default
* Đây là phạm vi truy cập mặc định khi ta không đặt mức truy cập nào cho thuộc tính. Ở dạng mặc định thì phạm vi truy cập chỉ ở trong nội bộ của package mà thôi. Mức truy cập này cũng giống mức truy cập protected.
  + 1. Public
* Là mức truy cập mà nó có thể truy cập ở mọi mức độ
  1. Từ khóa Static
* Dùng để khai báo một biến tĩnh. Biến đó có thể lưu thông tin chung cho tất cả các đối tượng.
* Thuộc tính static là thuộc tính mà chúng ta có thể gọi mà không cần khai báo đối tượng cho nó. Và được gọi ra bằng cú pháp <ten\_class>.<ten\_thuoc\_tinh>
* Phương thức static là phương thức mà ta cũng không cần phải tạo đối tượng mới mà vẫn có thể gọi nó. Nó rất phù hợp cho những class thư viện viết sẵn
* Khối Static được sử dụng cho mục đích khởi tạo giá trị các biến static. Khối sẽ được thực hiện khi lớp chứa nó được load vào trong bộ nhớ. Trong một lớp có thể có nhiều khối tùy ý. Các khối này sẽ chạy cùng nhau và chạy trước cả chương trình main của lớp đó
  1. Từ khóa This
* Từ khóa this dùng để ánh xạ đối tượng hiện tại.
  + Ánh xạ đối tượng khi cần sử dụng.
  + Gọi phương thức từ lớp hiện tại.
  + Gọi lại phương thức khởi tạo. Một phương thức cùng một tên nhưng có nhiều biến truyền vào gọi là overloading.
  + Trả về đối tượng (instance) của lớp hiện tại

1. Kế thừa

II.1 Định nghĩa

* Kế thừa có nghĩa là một lớp sẽ thừa hưởng lại thuộc tính và phương thức của lớp trước đó. Mục đích của kế thừa là nhằm tái sử dụng code đã viết trước đó. Thuật tiện cho việc bảo trì và nâng cấp cấu trúc
* Cấu trúc:

Class A extends B{

<phuong\_thuoc\_ke\_thua\_tu\_lop\_cha>{

Super(gia\_tri\_truyen\_vao);

<code>;

}

}

* Cha có thì con có, con có chưa chắc cha đã có.
* Từ khóa super mục đích chính là truy cập các phương thức của lớp cha.
  1. Setter và Getter
* Định nghĩa: Là hai phương thức để cập nhật hoặc lấy giá trị của thuộc tính. Đặc biệt là những thuộc tính private. Settet và getter cần thiết cho việc truy cập những luồng thông tin quan trọng thường cần giá trị chính xác.
* Cấu trúc:

Setter

Public void set<ten\_thuoc\_tinh>(<tham\_so\_gia\_tri\_moi>){

This.<ten\_thuoc\_tinh> = <tham\_so\_gia\_tri\_moi>;

}

Getter

<kieu\_du\_lieu\_cua\_thuoc\_tinh>get<ten\_thuoc\_tinh>(){

Return this.<ten\_thuoc\_tinh>;

}

* Khi đã dùng getter và setter thì thuộc tính nên để private. Vì setter và getter nhằm quản lý truy cập của thuộc tính, thì ta không nên để thuộc tính truy cập ở mức dễ dàng.
* Hãy cẩn thận với kiểu dữ liệu tham chiếu. Ta dùng phương thức clone(). Đây là phương thức hỗ trợ của Java, nó sẽ tạo một bản sao rồi trả bản sao đó cho đối tượng được gán.

VD: Trong phương thức getter hoặc setter thay vì this.array = array thì ta dùng this.array = array.clone()

* Thuộc tính có kiểu dữ liệu String thì ta dùng setter với getter bình thường. Vì bản chất khi tạo 1 kiểu dữ liệu tham chiếu cho string thì thực chất nó vẫn tạo đối tượng string mới.
* Tạo phương thức clone:

Public <Ten\_class> clone(){

//Khai báo đối tượng;

//return đối\_tượng;

}

* 1. Overriding và Overloading
     1. Overriding (Ghi đè)
* Có nghĩa là hai phương thức giống nhau về tên và tham số truyền vào. Một phương thức ở lớp cha, còn phương thức còn lại ở lớp con. Overriding cho phép lớp con có thể thực hiện riêng biệt cho phương thức mà lớp cha đã cung cấp.
* Nếu không muốn lớp con overriding lại phương thức nào đó thì ta sử dụng từ khóa final trước kiểu trả về của phương thức ở lớp cha.
  + 1. Overloading (Nạp chồng)
* Có nghĩa là nhiều phương thức trong một lớp có chung tên nhưng khác tham số truyền vào

1. Tính trừu tượng trong Java
   1. Tính trừu tượng

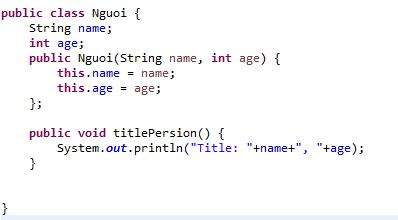
* (Tài liệu)
  + 1. Tính trừu tượng trong lập trình hướng đối tượng
* Tính trừu tượng trong hướng đối tượng là che giấu thông tin thực hiện từ người dùng. Người dùng chỉ biết tính năng đang được cung cấp và thông tin đối tượng.

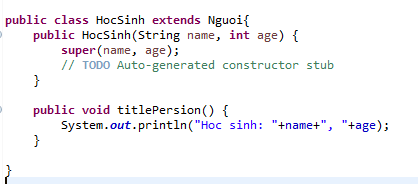
VD: Nếu bạn lập trình ra 1 phần mềm rất lớn. Bạn không thể tưởng tượng hết tất cả các khả năng nó có thể xảy ra trong đầu. Thay vì phải nạp hết vào đầu thì ta trừu tượng hóa nó để tạo ra 1 khung cho biết tính năng mình đang muốn thực hiện thì nó sẽ có kết quả như thế nào. Còn làm như thế nào để ra kết quả như vậy ta tính sau.

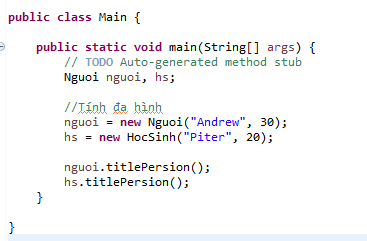
* + 1. Tính trừu tượng trong Java
  + Lớp trừu tượng (Asbtract Class)
    - Lớp trừu tượng là lớp được khai báo mà không thể tạo ra đối tượng từ lớp đó. Ta sẽ tạo những lớp con kế thừa lớp trừu tượng.
    - Mục đích của lớp trừu tượng là tạo ra lớp chung cho những lớp có liên quan với nhau kế thừa.

VD: Khi xây dựng phần mềm quản lý nhà trường. Thì những lớp Sinh viên, giảng viên, cán bộ,... có những thuộc tính và phương thức chung như: Họ tên, giới tính, chức vụ,.. Thì ta sẽ tạo lớp trừu tượng là lớp Persion để khai báo những thuộc tính chung đó. Khi phát triển chương trình ta có thể tạo ra các lớp đối tượng từ lớp con kế thừa từ lớp con người đó. Không thể cho tạo đối tượng từ lớp con người được.

* + Phương thức trừu tượng
    - Các phương thức trừu tượng là các phương thức chỉ định nghĩa mà không có chương trình bên trong, lớp con kế thừa phải bắt buộc override nó để sử dụng.
    - Phương thức trừu tượng có ý nghĩa định nghĩa phương thức bắt buộc phải có trong lớp con kế thừa
    - Nên cho các thuộc tính ở lớp trừu tượng ở mức protected để lớp con có thể kế thừa dễ dàng hơn thay vì private
  1. Tính đa hình
     + VD: Nguoi hs = new HocSinh();
     + Lúc này đối tượng hs được tạo sẽ là thể hiện của lớp đối tượng HocSinh() vì thế nó sẽ gọi được các thuộc tính và phương thức của lớp HocSinh()









* 1. Interface
     1. VD:
     + Khi chúng ta muốn quản lý danh sách về hành động (running). Trong dự án có rất nhiều các lớp đối tượng như quản lý về sinh viên, xe cộ, động vật, … Bây ta muốn khi gọi đến phương thức running(); nó sẽ in ra tất cả những thằng động vật, xe cộ, sinh viên,… đều cùng thực hiện thương thức running(); đó.
     + Như vậy ta không thể dùng kế thừa được vì bản chất nó là những class riêng biệt nhau.
     + Giải pháp của ta là sẽ gom tất cả các phương thức running(); ở các class vào một class chung. Sau đó tất cả các class động vật, xe cộ, sinh viên,… kế thừa lại. Người ta gọi là interface
     1. Định nghĩa
     + Interface là một kiểu dữ liệu tham chiếu trong java. Nó là tập hợp các phương thức abstract.
     + Khi 1 class kế thừa interface thì nó sẽ kế thừa những phương thức abstract của interface đó.
     1. Đặc điểm
     + Không thể khởi tạo nên không có phương thức khởi tạo
     + Tất cả các phương thức trong interface luôn ở dạng public abstract mà không cần khai báo.
     + Các thuộc tính trong interface luôn ở dạng public static final mà không cần khai báo, yêu cầu phải có giá trị
     1. Mục đích
     + Là để thay thế đa kế thừa của lớp trong các ngôn ngữ lập trình khác.
     + Interface sẽ giúp đồng bộ hóa và thống nhất trong việc đồng bộ hóa và trao đổi thông tin.
     + Khi đặt tên của interface nên cho thêm chữ I vào trước tên đó.
     + Chỉ có thể extends 1 class cha nhưng có thể implement bao nhiêu interface cũng được.
     + Nếu một lớp abstract mà implement một interface nào đó thì class con của lớp abstract đó mặc dù không implement trực tiếp nhưng vẫn phải override phương thức của interface đó

1. Try-catch-finally
   1. Exception – Ngoại lệ
      1. Định nghĩa
      * Exception là lỗi sinh ra trong quá trình phát triển dự án
      * Các lỗi thường gặp là: Lỗi logic, Lỗi khai báo,…
      * Khi gặp Exception thì nó sẽ dừng chương trình ngay ở đoạn đó và nó sẽ không xử lý tiếp các phần sau.
   2. Try- catch-finally

**try** {

**for** (**int** i = 0; i <= arr.length; i++) {

System.***out***.println("Phan tu ["+(i+1)+"]: ");

arr[i] = sc.nextInt();

}

} **catch** (Exception e) {

// **TODO**: handle exception

e.printStackTrace();

}**finally** {

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

}

- Thử chạy code ở try. Nếu code chạy không gặp lỗi thì nó sẽ chạy đến finally và nó sẽ chạy đến lệnh tiếp theo.

- Nếu code trong try nó đoạn gây ra lỗi thì nó sẽ đi vào mục catch. Khi mà catch xử lý xong thì nó sẽ chạy đến finally.

- Như vậy finally được gọi khi chương trình gặp lỗi hoặc không gặp lỗi

- Catch thì được gọi khi chương trình gặp lỗi.

- Exception e : Có thể xử lý được mọi lỗi của trương trình

- Finally thường được sử dụng trong các kết nối tới database. Khi chúng ta ở connect tới database và có những truy vấn database. Nếu những truy vấn database đó bị lỗi mà không có thằng finally thì ta sẽ không đóng được được kết nối database. Sẽ dẫn tới làm chậm hệ thống vì sẽ có rất nhiều lệnh kết nối database được mở.

- Những cái cần giải phóng tài nguyên thì người ta thường sử dụng finally để dù có lỗi gì thì ta cũng không bị tốn tài nguyên.

* 1. Tự tạo try-catch

- VD:

**public** **class** Exception\_Main {

**public** **static** **float** divide(**int** x, **int** y) {

**return** x/y;

}

**public** **static** **float** plus(**float** x, **float** y) {

**return** x+y;

}

}

**public** **class** Test {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

**float** sum = Exception\_Main.*divide*(5, 0);

System.***out***.println("Sum: "+sum);

}

}

- Khi chương trình Test được chạy thì nó sẽ xảy ra lỗi

Exception in thread "main" java.lang.ArithmeticException: / by zero

at Exception\_Debug\_Package.Exception\_Main.divide(Exception\_Main.java:7)

at Exception\_Debug\_Package.Test.main(Test.java:7)

- Do người dùng cố tình nhập số 0 cho mẫu số vì thế mà lỗi by zero xảy ra.

- Khi đóng gói chương trình thì người ta muốn cho người lập trình biết được là chỗ đó có khả năng gặp lỗi luôn. Thì ta sẽ tự tạo Exception riêng của cúng ta.

**public** **class** ByZeroException **extends** Exception{

**float** x;

**float** y;

**public** ByZeroException(**float** x, **float** y) {

**this**.x = x;

**this**.y = y;

}

@Override

**public** String getMessage() {

**return** "Exception by zero ("+x+"/"+y+")";

}

}

- Gọi Exception vừa tạo vào hàm

**public** **class** Exception\_Main {

**public** **static** **float** divide(**int** x, **int** y) **throws** ByZeroException {

**if** (y==0) {

**throw** **new** ByZeroException(x, y);

}

**return** x/y;

}

}

- Try-catch vào hàm thực thi

**public** **class** Test {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

**float** sum;

**try** {

sum = Exception\_Main.*divide*(5, 0);

System.***out***.println("Sum: "+sum);

} **catch** (ByZeroException e) {

// **TODO** Auto-generated catch block

System.***out***.println(e.getMessage());

}

}

}

1. Debug
   1. Định nghĩa:

- Khi chạy nhiều chương trình. Tự nhiên chương trình gặp lỗi mà người lập trình không biết ở đâu. Thì ta sẽ sử dụng kỹ thuật debug

* 1. Kỹ thuật debug java

- Tìm hiểu thêm trên Internet

* 1. StringBuilder vs StringBuffer

- StringBuider là nó sẽ cấp phát bộ nhớ khi mình add chuỗi vào. Trong thực tiễn ta hay dùng StringBuilder hơn vì nó đỡ tốn bộ nhớ hơn.

- StringBuffer là nó sẽ cấp phát trước 1 không gian nhớ để add chuỗi vào cho nhanh

1. Collection

- Collection trong java là một root interface trong hệ thống cấp bậc Collection. Java Collection cung cấp nhiều interface (Set, List, Queue, Deque vv) và các lớp (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet vv).

* 1. Stack (Ngăn xếp)

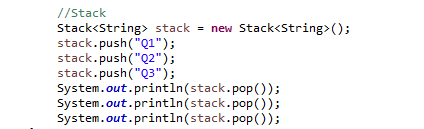
- Những phần tử được add vào stack thì sẽ theo nguyên tắc add trước nhưng lấy ra sau.

- Các hàm thường gặp:

+ pop(): Lấy dữ liệu ra

+ push(E item): Đưa dữ liệu vào

- Minh họa:



- Kết quả hiển thị



* 1. Queues (Hàng đợi)

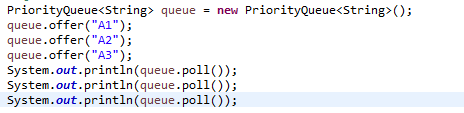
- Cách gọi phần tử theo nguyên tắc xếp hàng tức là phần tử vào trước thì được gọi trước. Phần tử vào sau thì được gọi sau.

- Các hàm thường gặp:

+ poll(): Lấy dữ liệu ra

+ offer(E item): Đưa dữ liệu vào

Minh họa:



- Kết quả hiển thị



* 1. Vector

Vector<Kieu\_du\_lieu> arr = new Vector<>();

- Giống hệt như ArrayList

- Khac biệt với ArrayList là khi add dữ liệu vào mảng thì nó sẽ tự động cấp phát 10 dữ liệu vào mảng. Sau khi ta dùng hết bộ nhớ vào Vector thì nó lại tự động cấp phát 10 dữ liệu nữa.

+ Ưu điểm: Nó sẽ làm cho Vector add dữ liệu vào nhanh hơn so với ArrayList

+ Nhược điểm: Nó sẽ làm tốn tài nguyên bộ nhớ.

* 1. LinkedList

LinkedList<Kieu\_du\_lieu> arr = new LinkedList<>();

- Khai báo giống kiểu con trỏ. Tức là nó sẽ quản lý địa chỉ ô nhớ và giá trị của các ô nhớ đó.

* 1. ArrayList

ArrayList<Kieu\_du\_lieu> arr = new ArrayList<>();

Bản chất của ArrayList là một mảng nhưng khi ta add thêm dữ liệu mới vào nó sẽ copy dữ liệu cũ vào một mảng mới và add dữ liệu thêm vào mảng mới đó và trả về mảng mới.

* 1. HashMap

- HashMap<String, String> map = new HashMap<>();

- map.put(“ten\_key”, “value”);

- Khi sử dụng HashMap ta có thể truyền key của nó là một string. Giống với khai báo mảng trong ngôn ngữ PHP

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

HashMap<String, String> map = **new** HashMap<>();

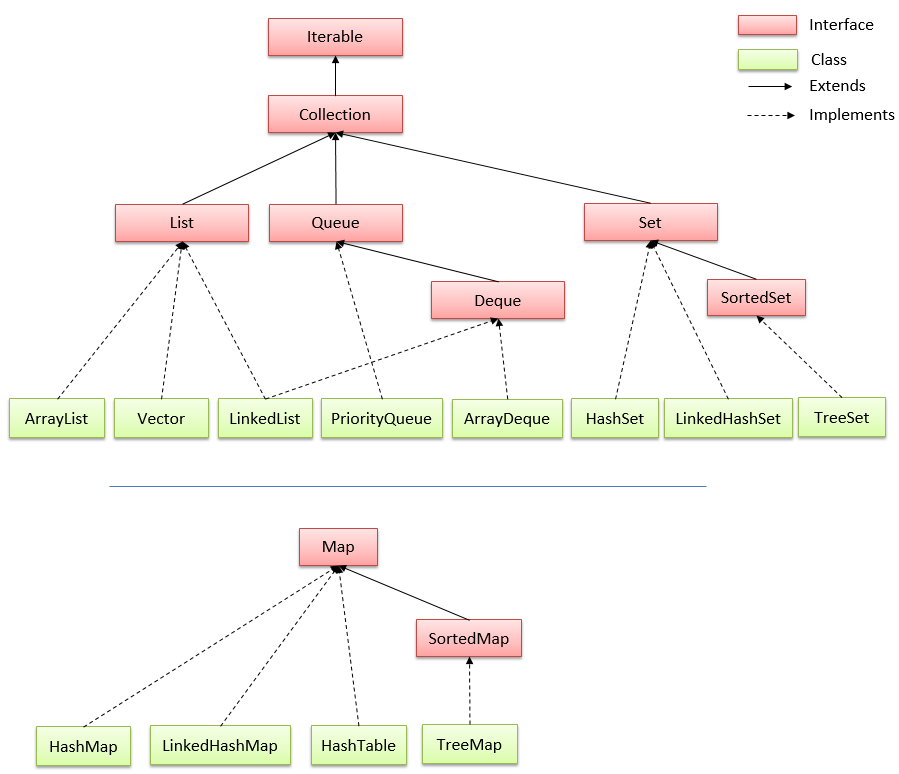
map.put("fullname", "Nguyen Dinh Quang");

map.put("tuoi", "25");

System.***out***.println(map.get("tuoi"));

}

- TreeMap và LinkedHashMap cũng giống như HashMap

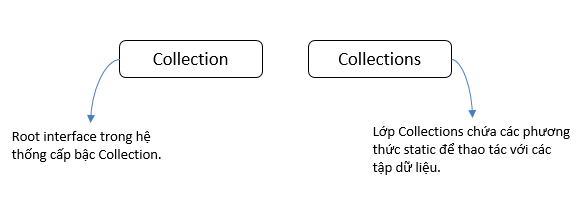


Có nhiều phương thức được khai báo trong interface Collection như sau:

|  |  |
| --- | --- |
| **Phương thức** | **Mô tả** |
| public boolean add(Object element) | Được sử dụng để chèn một phần tử vào collection. |
| public boolean addAll(Collection c) | Được sử dụng để chèn các phần tử collection được chỉ định vào collection gọi phương thức này. |
| public boolean remove(Object element) | Được sử dụng để xóa phần tử từ collection. |
| public boolean removeAll(Collection c) | Được sử dụng để xóa tất cả các phần tử của collection được chỉ định từ collection gọi phương thức này. |
| public boolean retainAll(Collection c) | Được sử dụng để xóa tất cả các thành phần từ collection gọi phương thức này ngoại trừ collection được chỉ định. |
| public int size() | Trả lại tổng số các phần tử trong collection. |
| public void clear() | Loại bỏ tổng số của phần tử khỏi collection. |
| public boolean contains(Object element) | Được sử dụng để tìm kiếm phần tử. |
| public boolean containsAll(Collection c) | ược sử dụng để tìm kiếm collection được chỉ định trong collection. |
| public Iterator iterator() | Trả về một iterator. |
| public Object[] toArray() | Chuyển đổi collection thành mảng (array). |
| public boolean isEmpty() | Kiểm tra nếu collection trống. |
| public boolean equals(Object element) | So sanh 2 collection. |
| public int hashCode() | Trả về số hashcode của collection. |

1. Collections
   1. Định nghĩa

- Collections trong java là một khuôn khổ cung cấp một kiến trúc để lưu trữ và thao tác tới nhóm các đối tượng. Tất cả các hoạt động mà bạn thực hiện trên một dữ liệu như tìm kiếm, phân loại, chèn, xóa,... có thể được thực hiện bởi Java Collections.



* 1. Hfytdyt
  2. hgdgrzt

1. Generic
   1. Định nghĩa

K - key

T - type

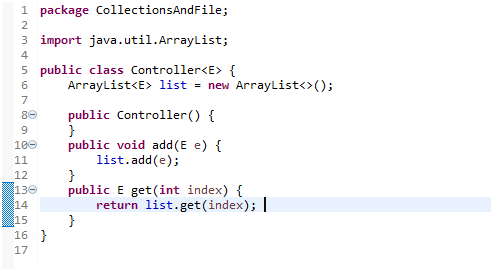
V - value

N - number

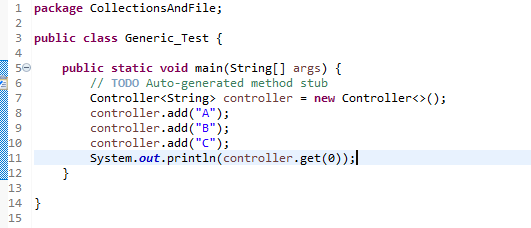
E - elememt

* 1. Create Generic

- Bước 1: Tạo thư viện Controller.java



- Bước 2: Dùng thư viện Controller vừa thiết kế



- Nếu muốn thư viện quản lý các class thì ta sửa **public** **class** Controller<E> thành **public** **class** Controller<E **extends** People>

1. File trong Java
   1. Định nghĩa

- File dùng để lưu trữ dữ liệu.

+) dạng text -> Đọc được

+) dang binary -> Không đọc được

* 1. Tạo File, Gọi File

- Để đọc ghi dữ liệu lên trên file chúng ta cần tới 3 bước:

+) Mở file

+) Đọc ghi dữ liệu

+) Đóng file

- Cú pháp mở một file: File file = **new** File("file.txt");

- Kiểm tra một file đã tồn tại hay chưa:

**if**(file.isFile()) {

System.***out***.println("File da ton tai: "+file.getAbsolutePath());

} **else** {

System.***out***.println("Not");

}

Trong đó: file.getAbsolutePath(): Cú pháp kiểm tra đường dẫn tuyệt đối của file

- Tạo file mới: file.createNewFile();

- Xóa một file: file.delete();

* 1. Đọc file

- Trong file ký tự kết thúc trả về -1 nên để đọc được file ta cần phải kiểm tra điều kiện của vòng lặp while khác -1:

- Cú pháp đọc file:

**int** ch;

**while** ((ch = fileInput.read()) != - 1) {

System.***out***.print((**char**)ch);

}

- Cú pháp đọc từng ký tự một bằng FileInputStream:

FileInputStream fileInput = **null**;

**try** {

fileInput = **new** FileInputStream("file-01.txt");

**int** ch;

**while** ((ch = fileInput.read()) != - 1) {

System.***out***.print((**char**)ch);

**break**;

}

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}**finally** {

**try** {

fileInput.close();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

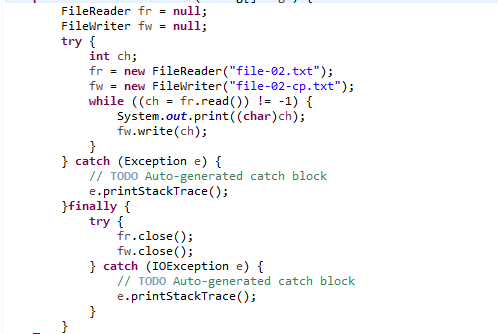
}

- Copy file bằng cách viết bằng đối tượng FileOutputStream:

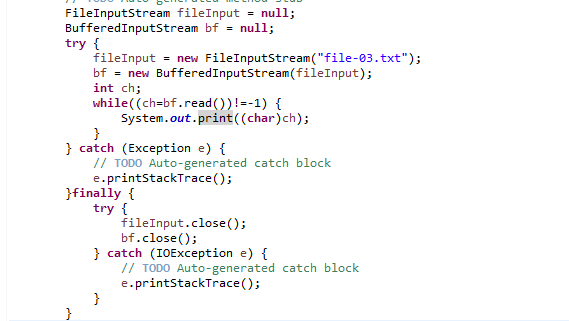


- FileReader và FileWriter

+) FileReader và FileWriter dùng để đọc ghi những file có dạng texr chứ nhưng không thể dùng để đọc ghi file nhị phân được.!



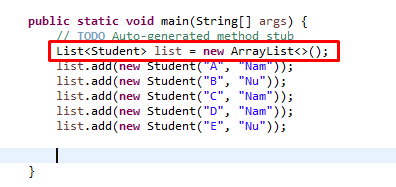
- BufferedInputStream

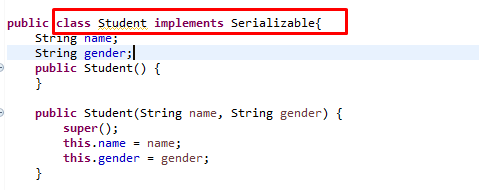


* 1. ObjectOutputStream và ObjectInputStream

- Để lưu dữ liệu của list vào một file ta có thể lưu trực tiếp đối tượng bên trong vào file bằng cách sau.

- B1: **implements** Serializable trong lớp đối tượng được gọi đến của arrayList

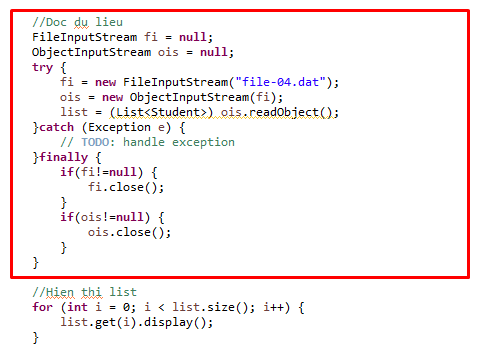




- B2: Tạo ra ObjectOutputStream để ghi dữ liệu vào file

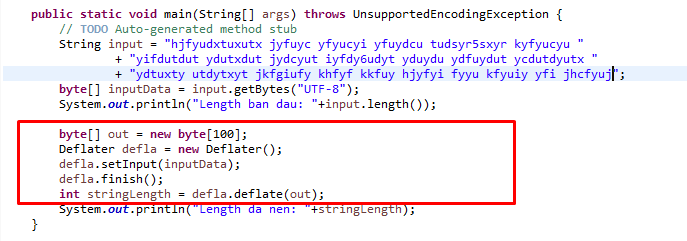


- B3: Tạo ra ObjectInputStream để đọc dữ liệu từ file đã có

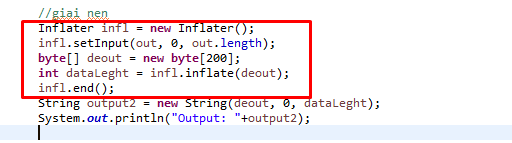


* 1. Nén file và giải nén file

- Nén dữ liệu bằng Deflater



- Giải nén bằng Inflater



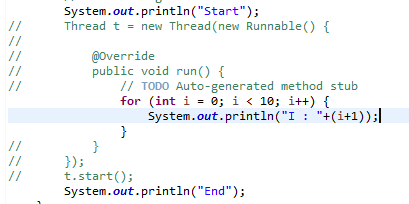
1. Thread (Luồng)
   1. Định nghĩa

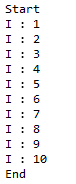
-

* 1. Khai báo luồng trong java

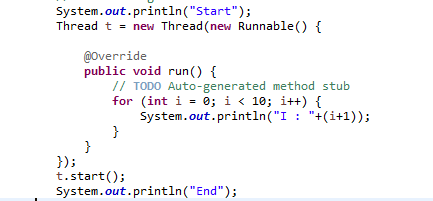
- Cách chạy của các luồng

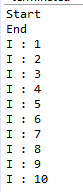
Trước khi sử dụng thread thì code sẽ chạy như sau





Nhưng sau khi sử dụng thread thì code sẽ được chạy lại như sau.





Khi tạo luồng bằng Thread thì tiến trình sẽ được chia làm 2 luồng chạy độc lập nhau. Luồng 1 là luồng chính xử lý các đoạn code

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

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

Và luồng 2 xử lý các đoạn code

Thread t = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

// **TODO** Auto-generated method stub

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("I : "+(i+1));

}

}

});

t.start();

- Khai báo luồng (Cách 1)

Thread t = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

// Code

}

});

t.start();

- Khai báo luồng (Cách 2)

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

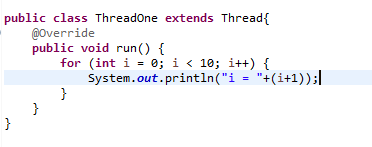
// code

}

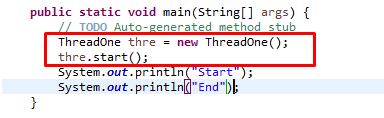
}).start();

- Khai báo luồng độc lập

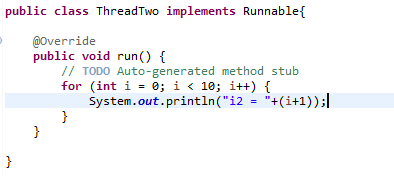
+) Tạo class ThreadOne chứa code chạy tại luồng Thread

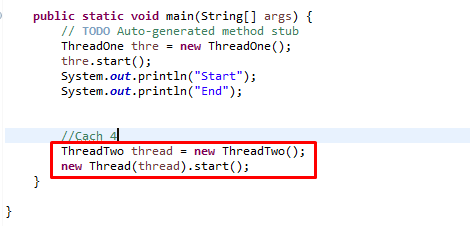


+) Trong class Thread2 khởi tạo đối tượng luồng



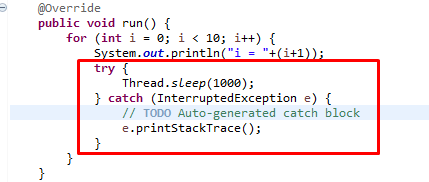
- Cách khai báo luồng thứ 4





* 1. Các lệnh trong xử lý luồng

- Tạm dừng luồng: Thread.*sleep*(millis);



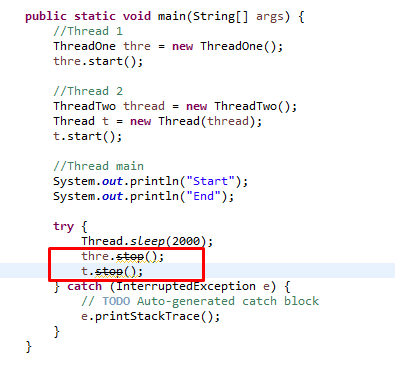
+) Mỗi lần in ra 1 dòng lệnh nó sẽ dừng 1s sau đó lại thực hiện câu lệnh tiếp theo

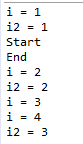
- Lệnh chết luồng: thre.~~stop~~();

+) Khi nhận lệnh stop() thì luồng sẽ bị hủy

+) Khi ta dùng lệnh stop() thì thread sẽ chết hoặc khi nó xử lý hết code thì luồng cũng chết.

+) Luồng đã chết thì không thể start lại được

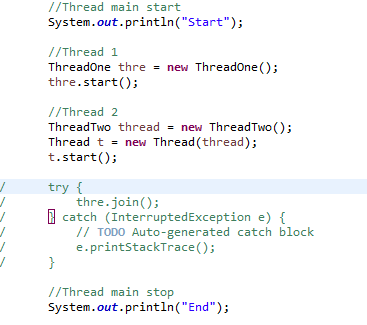


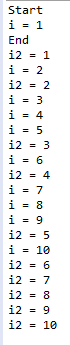


- Nối luồng 1 vào luồng main: thre.join();

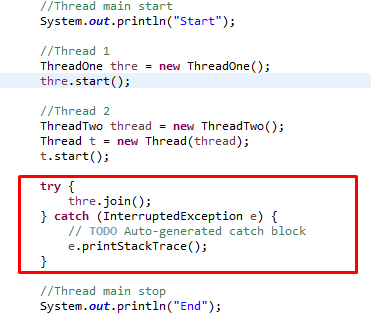
+) Nối luồng là khi để cho các luồng trước đó chạy hết rồi thì luồng kế tiếp mới chạy.

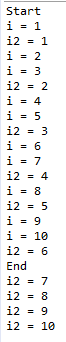
+) Trước khi nối luồng:





+) Sau khi nối luồng Thread 1:



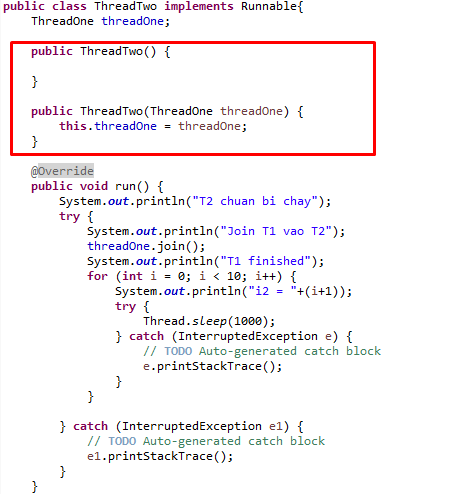


Ta thấy khi áp dụng lệnh join() thì khi thực hiện tiến trình start máy sẽ thực hiện lần lượt các thread 1 và 2. Sau đó phải chờ cho thread 1 chạy xong thì luồng tiếp theo mới được chạy. Như ở trên là thread stop.

- Nối 2 luồng lại với nhau:

+) Nếu muốn nối Thread 1 vào Thread 2 thì ta phải đặt nameThread1.join() vào chính Thread 2.

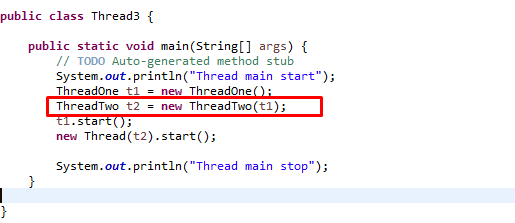
+) Để làm được như vậy ta sẽ đặt một hàm tạo tại Thread2:



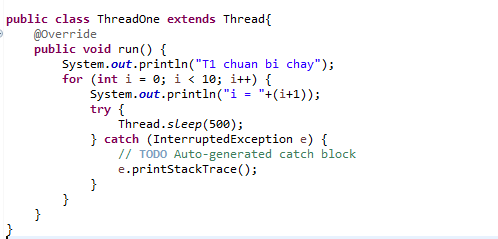
+) Sau đó đặt câu lệnh join vào Thread2



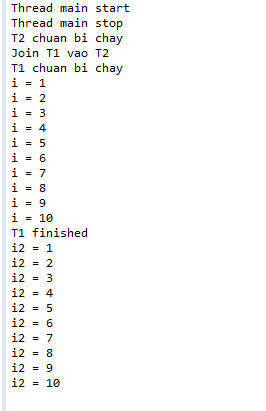
+) Class main



+) Class Thread1



+) Kết quả:



- Bài tập 1:

+) Tạo lớp Thread1 kế thừa Thread gồm các thuộc tính ArrayList<Interger> listData. Thread này thực hiện sau 1s sinh ngẫu nhiên các số tự nhiên chạy từ 0-100 -> Lưu kết quả vào mảng listData.

+) Tạo lớp Thread2 kế thừa Thread gồm các thuôc tính ArrayList<Char> list2. Thread này thực hiện trong 2s thì sinh ngẫu nhiên các ký tự từ a-z và lưu vào mảng list2.

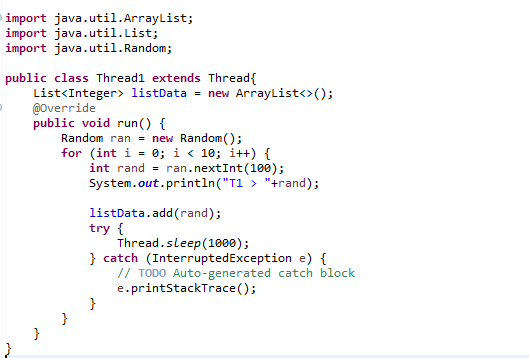
+) Trong phương thức main của lớp Test tạo ra 2 thread t1 và t2 lần lượt từ Thread1 và Thread2.

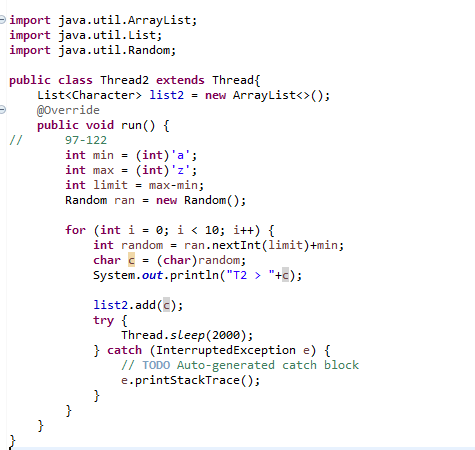
+) Khi t1 đã sinh ra 10 lần -> thực hiện dừng t1

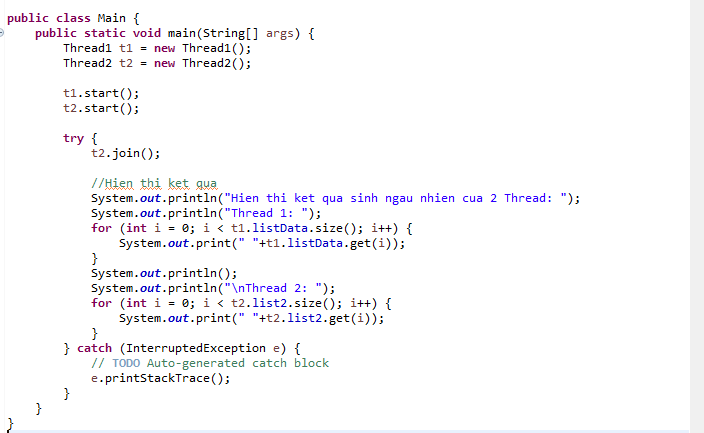
+) Khi t2 đã sinh ra 10 ký tự -> dừng t2

+) In kết quả lên màn hình

- Chữa bài





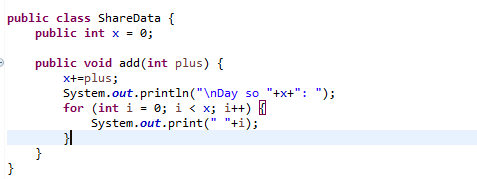


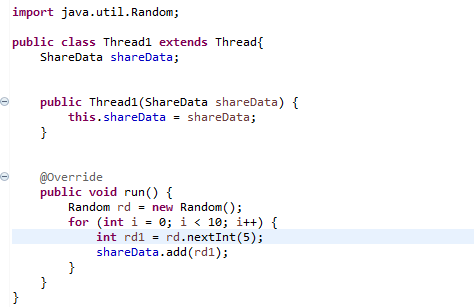
* 1. Đồng bộ luồng

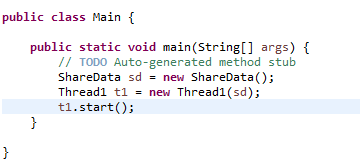
- Dùng trong trường hợp đặt tính bảo mật cao

* 1. Synchronized

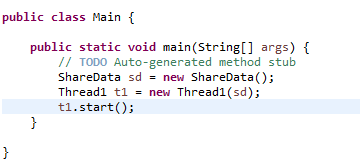
- Xây dựng chương trình demo

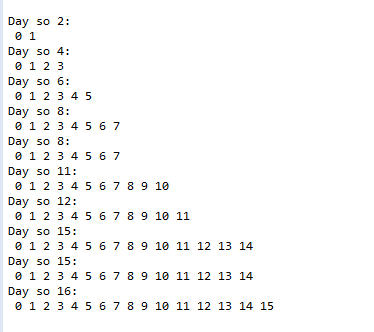




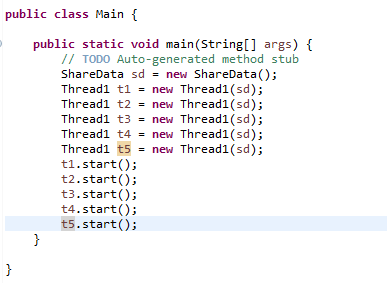


- Khi có một luồng chạy



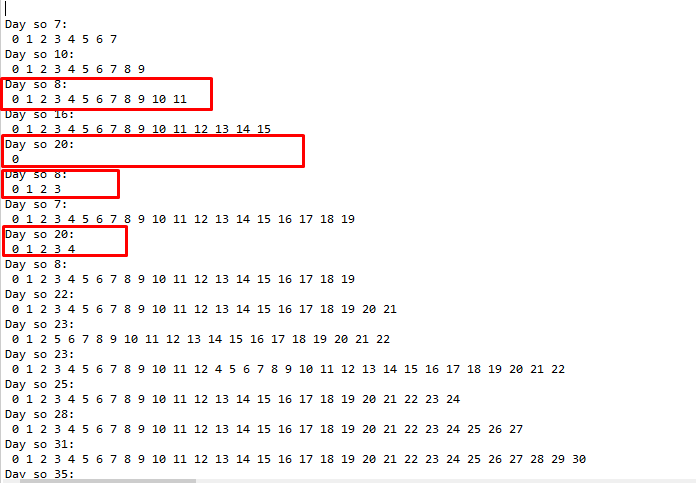


- Khi có nhiều luồng chạy đồng thời



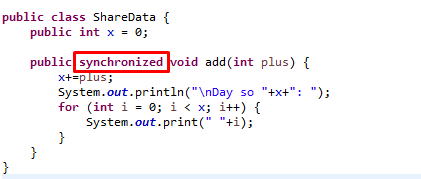
Ở đây ta thấy 5 luồng cùng chạy đồng thời một lúc.

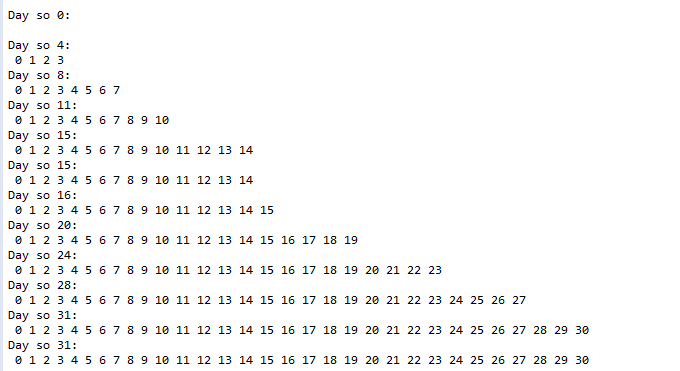
Kết quả là:



Ta nhận thấy trong kết quả nhận được thì dãy số 8 in tận 12 số. Mà dãy số 20 lại chỉ in được 1 số. Như vậy dữ liệu được in ra là không đồng bộ.

- Khi có nhiều luồng chạy đồng thời và chương trình đã được đồng bộ





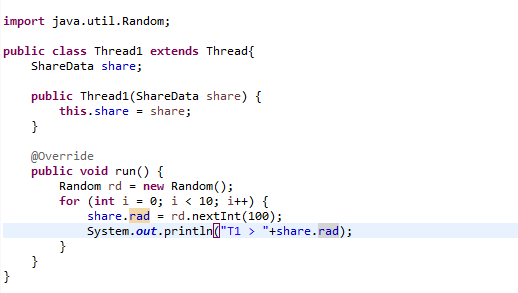
Ở kết quả trên ta thấy dữ liệu đã được đồng bộ.

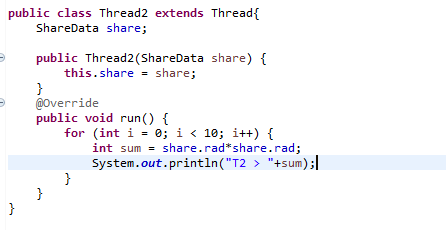
- Đặt Synchronized trước phương thức mà các luồng dữ liệu truy cập vào.

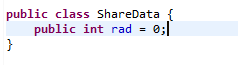
* 1. Wait/NotifyAll

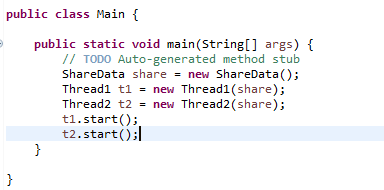
- Bài toán đặt ra: Tạo thread 1 sinh số ngẫu nhiên và thread 2 lấy số từ thread 1 đã sinh ra sau đó bình phương lên. Với điều kiện T1 chạy xong phải để T2 chạy xong T1 mới tiếp tục chạy

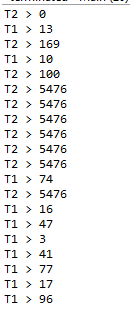
VD: T1 > 2 , T2 > 4, T1 > 3, T2 > 9,…





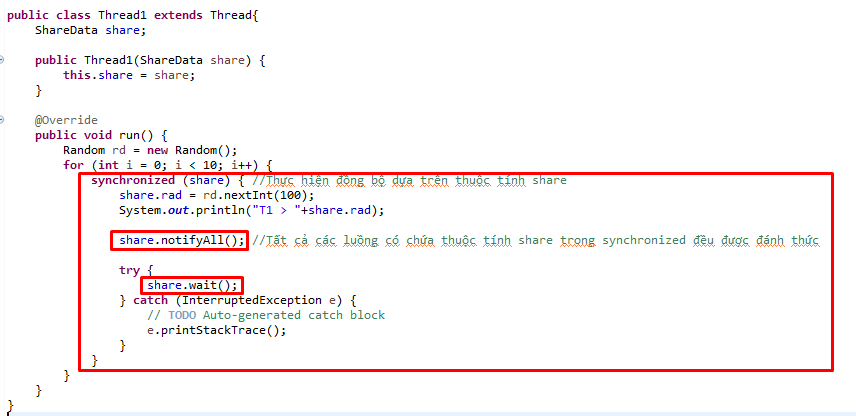


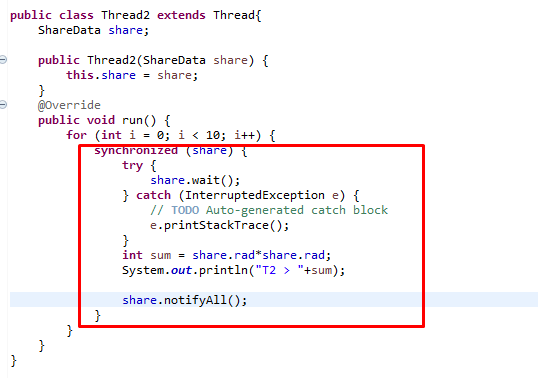




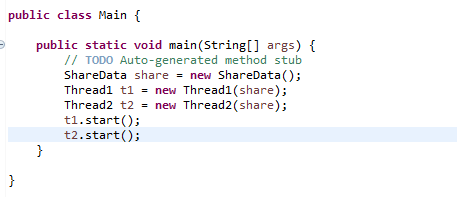
Kết quả chạy không đúng yêu cầu.

- Giải pháp:









+) Đặt **synchronized** (share) trong luồng 1 để thực hiện đồng bộ dựa trên thuộc tính share. Làm tương tự với luồng 2. Trong bài này luồng 1 thực hiện trước. Nên sau lệnh **synchronized** (share) ở luồng 1 ta thực hiện các dòng code chính. Ở luồng 2 đặt lệnh share.wait();. Lệnh này có nghĩa là tất cả chương trình chạy ở luồng 2 đều ở trạng thái chờ.

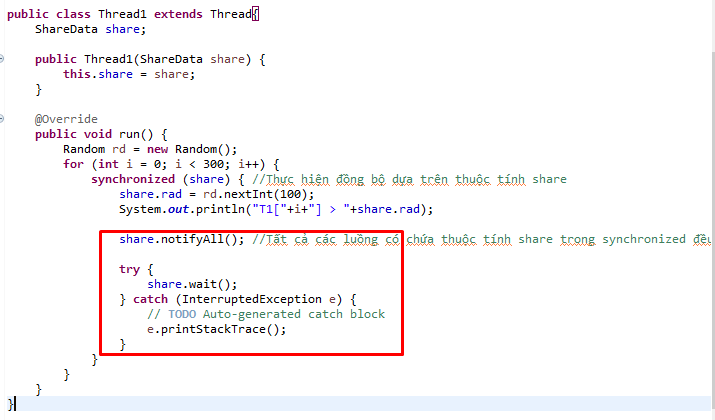
+) Khi luồng 1 chạy lần lượt các dòng code đến dòng share.notifyAll(); nó sẽ đánh thức tất cả các luồng có chứa **synchronized** (share). Tức là lúc này luồng 2 được đánh thức vì luồng 2 cũng có dòng **synchronized** (share)

+) 2 luồng chạy song song và ở luồng 1 nó chạy đến dòng lệnh share.wait(); ngay nên nó sẽ dừng lại. Ở luồng 2 lúc này chạy đến những dòng code để thực hiện chương trình.

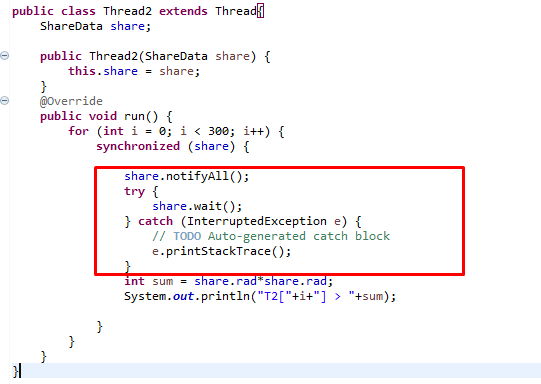
+) Khi luồng 2 chạy đến dòng code share.notifyAll(); thì luồng 1 lại được đánh thức. Và 2 luồng lại chạy song song. Tiếp tục quy trình cho đến khi kết thúc vòng lặp.

-

* 1. Deadlock  
     - Giả sử 2 luồng đồng thời chạy nhiều vòng lặp thì sẽ gặp tình trạng chờ lẫn nhau và đôi khi sẽ xảy ra tình trạng lỗi do luồng 1 hoặc 2 chưa xử lý hết công việc sẽ làm cho luồng còn lại phải chờ và không thể nào chạy code được. Vì thế để tối ưu hơn ta phải sử dụng kỹ thuật deadlock để tránh tình trạng này xảy ra.



+) Tại Thread 1 là luồng xử lý đầu tiên ta nên để notifyAll và wait ở dưới cùng trong synchronized



+) Tại Thread 2 là luồng chờ xử lý ta nên để notifyAll và wait ở trên đầu trong synchronized

+) Khi các luồng đã chạy xong ta phải xem xét xem luồng nào đang ở trạng thái delay. Sau đó vào luồng còn lại đặt

**synchronized** (data) {

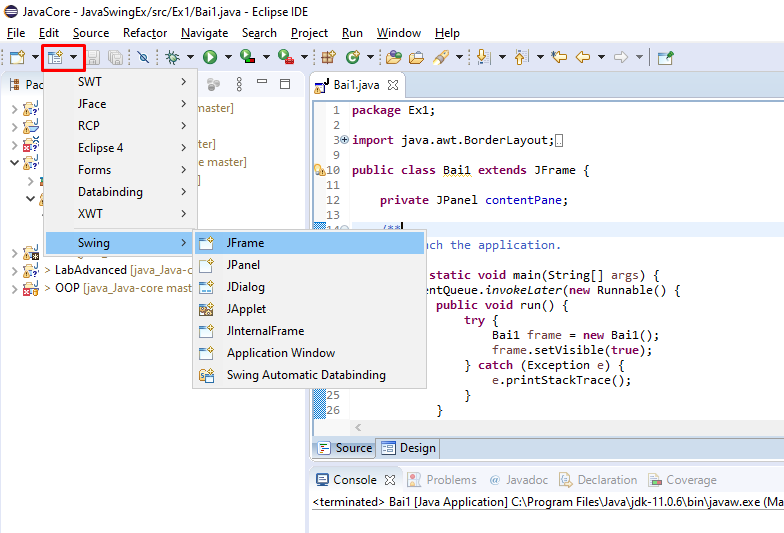
data.notifyAll();

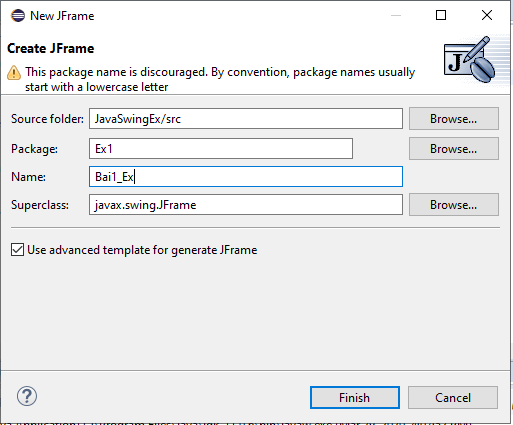
}

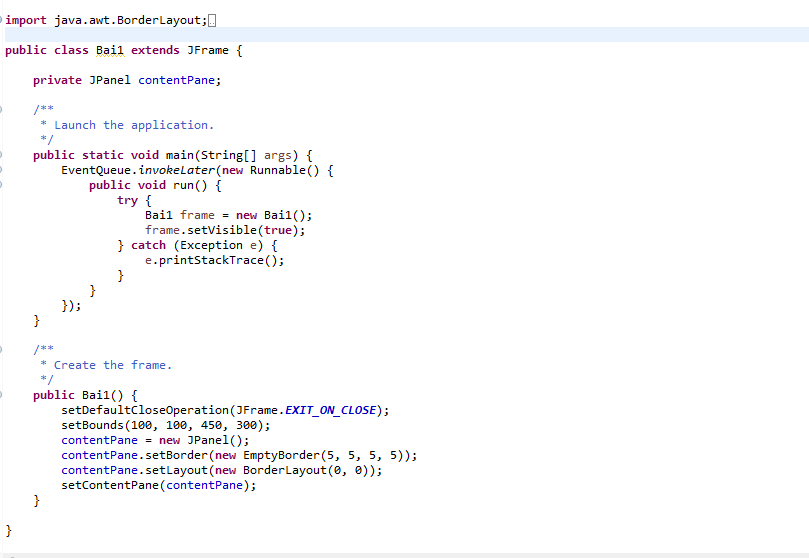
Để mở luồng đang delay để nó chết luồng. Như vậy code mới an toàn.

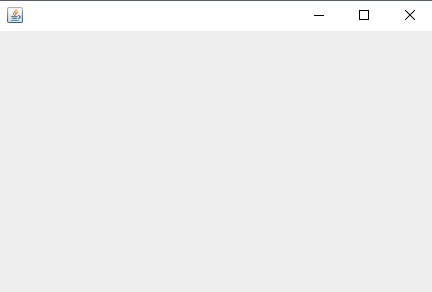
1. Java Swing
   1. Mở giao diện

- Mở giao diện Java Swing trong Eclipse

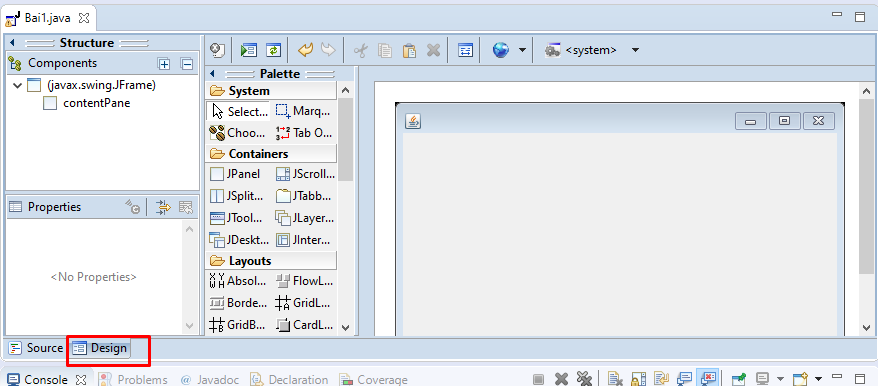




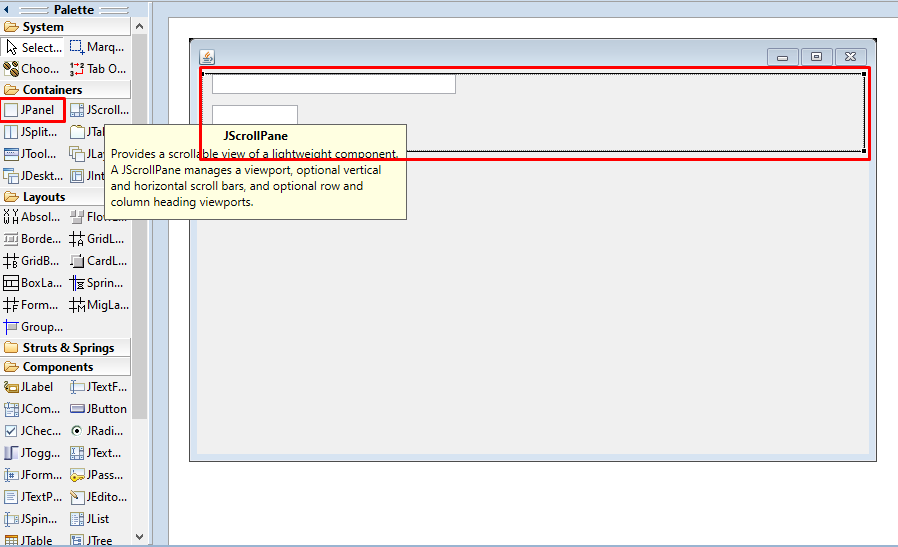




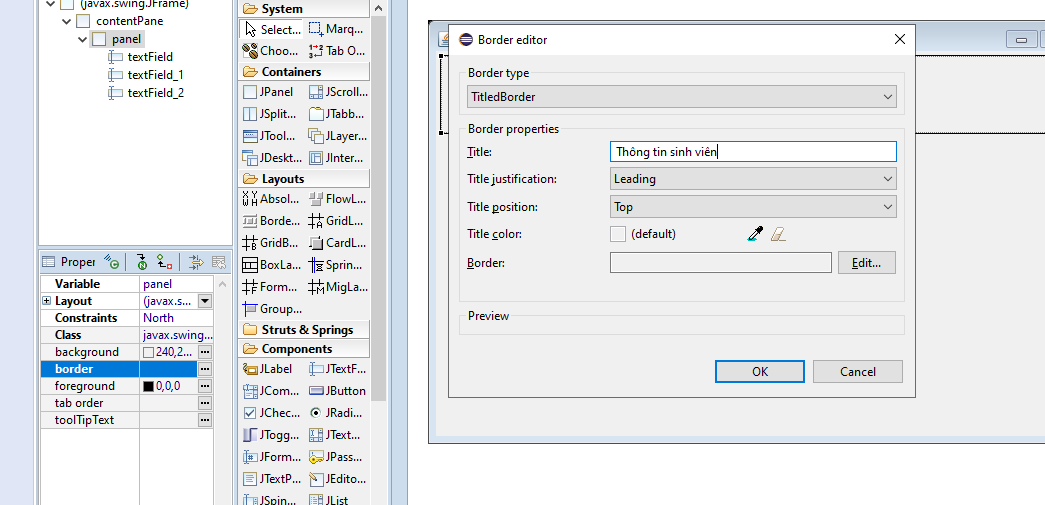
- Mở giao diện design



* 1. Components
     1. Jpanel
     + Giống như container là một khối để chứa các components

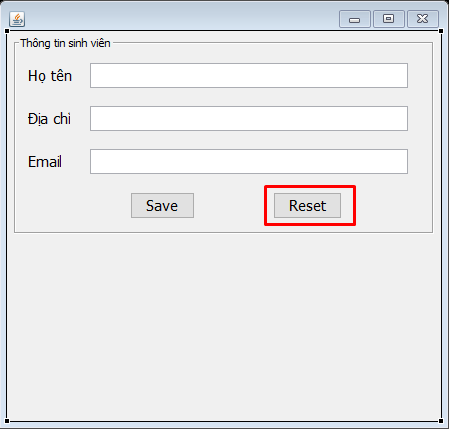


Thay đổi title của Jpanel

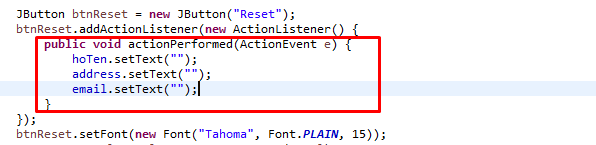


- Click vào components cần thay đổi code là trong phần code sẽ tự gender ra code của phần đó. Và ta chỉ việc sửa lại code cho phần đó mà thôi.

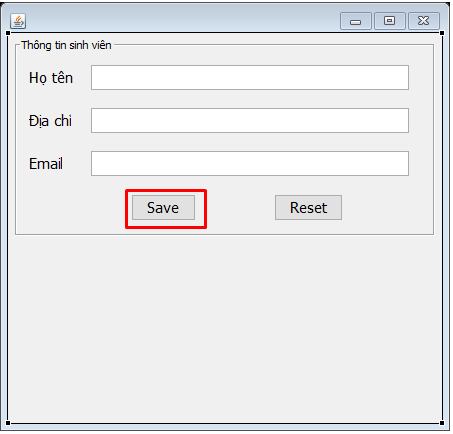
* + 1. Bắt sự kiện nút Reset



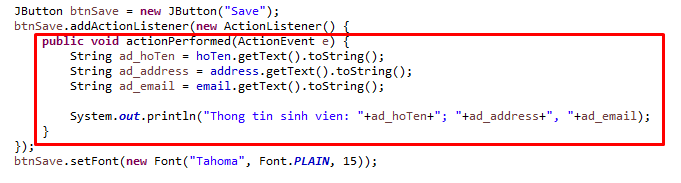
- Code:



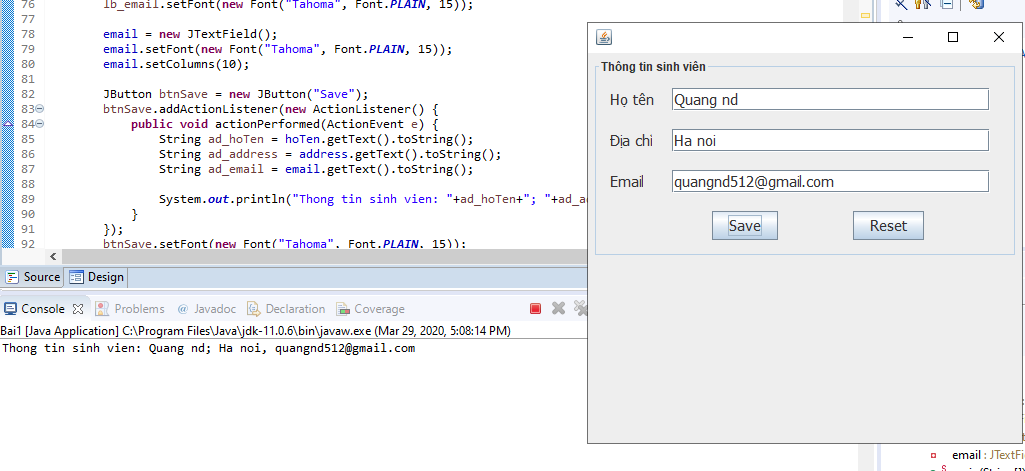
* + 1. Bắt sự kiện nút Save



- Code

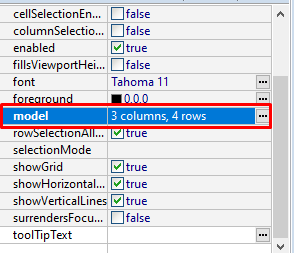


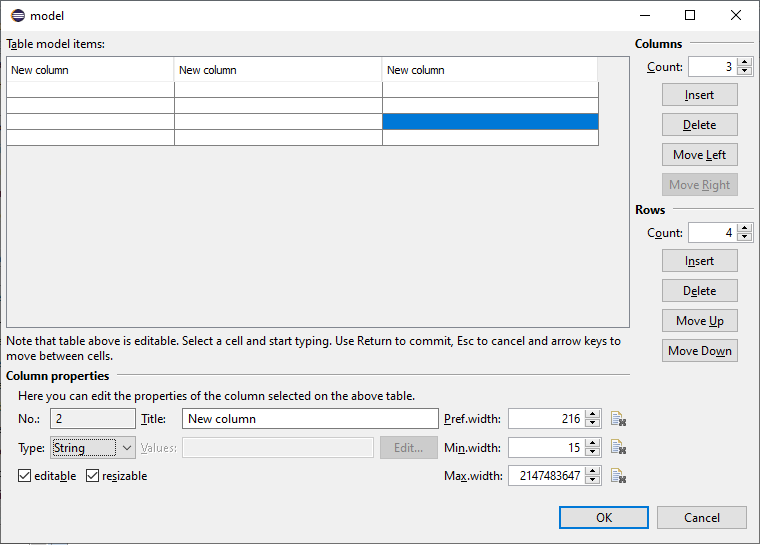
- Kết quả



* + 1. Table trong Swing

- Thay đổi cột và hàng của bảng





- Editable: Cho phép thay đổi dữ liệu trong bảng

- Resizable: Cho phép thay đổi độ dài của các cột

- Để tự động thêm các trường mới vào trong bảng khi ta add dữ liệu vào list ta làm như sau:

* + 1. Hgsttrst
  1. hfchdyt

1. Java FX
2. JDBC
   1. Các bước kết nối Java với MySQL

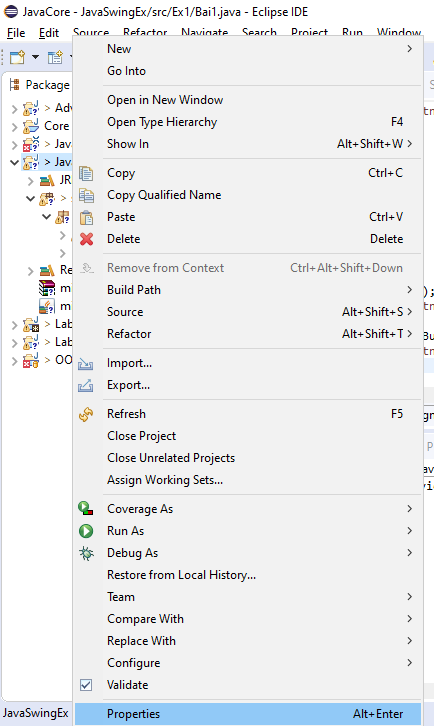
Bước 1: Cài đặt xampp

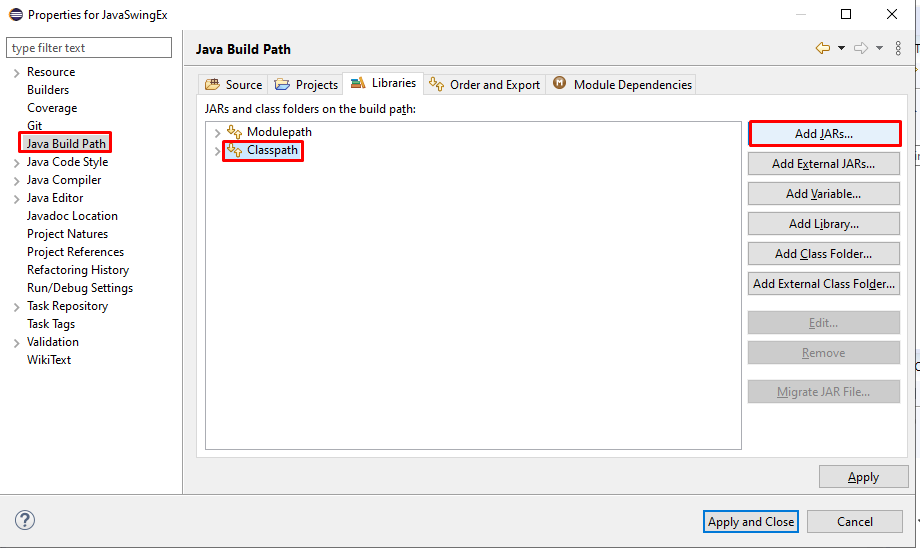
Bước 2: Truy cập vào MySQL trong Xampp tạo Database mới và các tables.

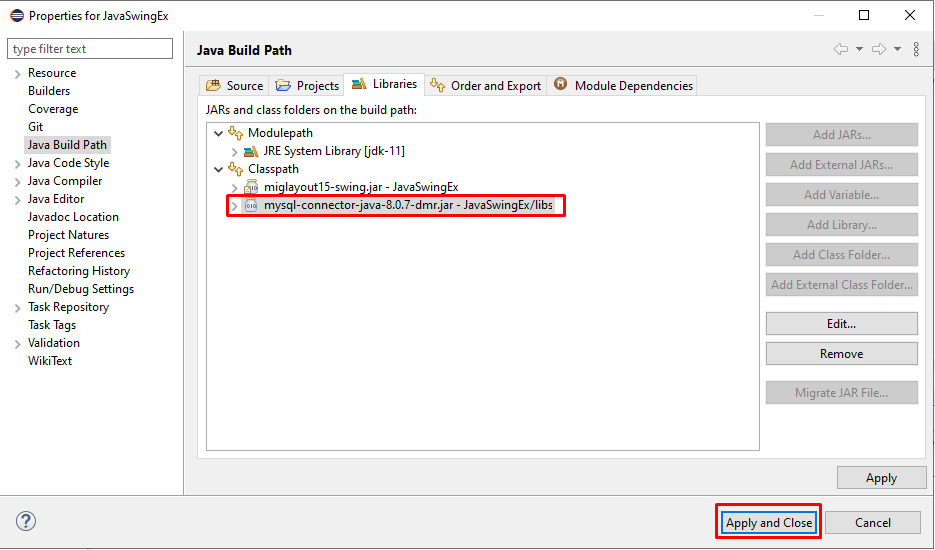
Bước 3: Dowload drivers JDBC (<https://mvnrepository.com/artifact/mysql/mysql-connector-java>)

Bước 4: Tạo thư mục libs trong dự án và bỏ mysql-connector-java-8.0.16.jar vào thư mục đó.

Bước 5: Nhúng mysql-connector-java-8.0.7-dmr.jar vào project







Bước 6: Triển khai code

1. Regex trong Java (Biểu thức chính quy)

- Java Regex hoặc Regular Expression (biểu thức chính quy) là một API để định nghĩa một mẫu để tìm kiếm hoặc thao tác với chuỗi. Nó được sử dụng rộng rãi để xác định ràng buộc trên các chuỗi như xác thực mật khẩu, email, kiểu dữ liệu datetime, ...

* 1. Matcher class

|  |  |  |
| --- | --- | --- |
| **No.** | **Phương thức** | **Mô tả** |
| 1 | boolean matches() | kiểm tra xem biểu thức chính quy có khớp với mẫu hay không. |
| 2 | boolean find() | tìm biểu thức tiếp theo khớp với mẫu. |
| 3 | boolean find(int start) | tìm biểu thức tiếp theo khớp với mẫu từ số bắt đầu đã cho. |
| 4 | String group() | trả về chuỗi con phù hợp. |
| 5 | int start() | trả về vị trí bắt đầu của chuỗi con phù hợp. |
| 6 | int end() | trả về vị trí kết thúc của chuỗi con phù hợp. |
| 7 | int groupCount() | trả về tổng số các chuỗi con phù hợp. |

Nó implements interface MatchResult, cung cấp bộ máy xử lý biểu thức chính quy để thao tác với chuỗi ký tự.

* 1. Pattern class

Đây là phiên bản được biên dịch của một biểu thức chính quy. Nó được sử dụng để xác định một mẫu cho bộ máy regex.

|  |  |  |
| --- | --- | --- |
| **No.** | **Phương thức** | **Mô tả** |
| 1 | static Pattern compile(String regex) | biên dịch regex đã cho và trả về thể hiện của Pattern. |
| 2 | Matcher matcher(CharSequence input) | tạo một matcher khớp với đầu vào đã cho với mẫu. |
| 3 | static boolean matches(String regex, CharSequence input) | Nó biên dịch biểu thức chính quy và tìm kiếm các chuỗi con từ chuỗi input phù hợp với mẫu regex. |
| 4 | String[] split(CharSequence input) | chia chuỗi input đã cho thành mảng các kết quả trùng khớp với mẫu đã cho. |
| 5 | String pattern() | trả về mẫu regex. |

* 1. Cú pháp Regex trong Java

1. Các lớp ký tự Regex

|  |  |
| --- | --- |
| **Regex** | **Mô tả** |
| [...] | trả về một ký tự phù hợp |
| [abc] | a, b, hoặc c |
| [^abc] | Bất kỳ ký tự nào ngoại trừ a, b, hoặc c |
| [a-zA-Z] | a tới z hoặc A tới Z |
| [a-d[m-p]] | a tới d, hoặc m tới p: [a-dm-p] |
| [a-z&&[def]] | d, e, hoặc f |
| [a-z&&[^bc]] | a tới z, ngoại trừ b và c: [ad-z] |
| [a-z&&[^m-p]] | a tới z, ngoại trừ m tới p: [a-lq-z] |
| [0-9] | 0 tới 9 |

1. Số lượng ký tự trong Regex

|  |  |  |  |
| --- | --- | --- | --- |
| **Regex** | **Mô tả** | **Pattern** | **Ví dụ** |
| X? | X xảy ra một hoặc không lần | hellos? | **hello**, **hellos**, **hellos**s |
| X+ | X xảy ra một hoặc nhiều lần | Version \w-\w+ | **Version A-b1\_1** |
| X\* | X xảy ra không hoặc nhiều lần | A\*B\*C\* | **AAACC** |
| X{n} | X chỉ xảy ra n lần | \d{4} | **2018**, **2018**9, 201 |
| X{n,} | X xảy ra n hoặc nhiều lần | \d{4,} | **2018**, **20189**, 201 |
| X{y,z} | X xảy ra ít nhất y lần nhưng nhỏ hơn z lần | \d{2,3} | **201**8, **201** |

1. Ký tự đặc biệt trong Regex

|  |  |
| --- | --- |
| **Regex** | **Mô tả** |
| . | Bất kỳ ký tự nào |
| ^ | Có 2 cách sử dụng. 1. Đánh dấu bắt đầu của một dòng, một chuỗi. 2. Nếuu sử dụng trong dấu [...] thì nó có nghĩa là phủ định. |
| $ | Đánh dấu Kết thúc của một dòng |
| \d | Bất kỳ chữ số nào, viết tắt của [0-9] |
| \D | Bất kỳ ký tự nào không phải chữ số, viết tắt của [^0-9] |
| \s | Bất kỳ ký tự trống nào (như dấu cách, tab, xuống dòng, ...), viết tắt của [\t\n\x0B\f\r] |
| \S | Bất kỳ ký tự trống nào không phải ký tự trống, viết tắt của [^\s] |
| \w | Bất kỳ ký tự chữ nào (chữ cái và chữ số), viết tắt của [a-zA-Z\_0-9] |
| \W | Bất kỳ ký tự nào không phải chữ cái và chữ số, viết tắt của [^\w] |
| \b | Ranh giới của một từ |
| \B | Không phải ranh giới của một từ |

1. Ký tự logic trong Regex

|  |  |  |  |
| --- | --- | --- | --- |
| **Regex** | **Mô tả** | **Pattern** | **Ví dụ** |
| | | Hoặc | 22|33 | 33 |
| ( … ) | Group các ký tự và chụp lại | A(nt|pple) | Ant, Apple |
| \1 | Nội dung của Group 1 | r(\w)g\1x | regex |
| \2 | Nội dung của Group 2 | (\d\d)\+(\d\d)=\2\+\1 | 12+65=65+12 |
| (?: … ) | Group không được chụp lại, bạn không thể sử dụng \1 | A(?:nt|pple) | Ant, Apple |

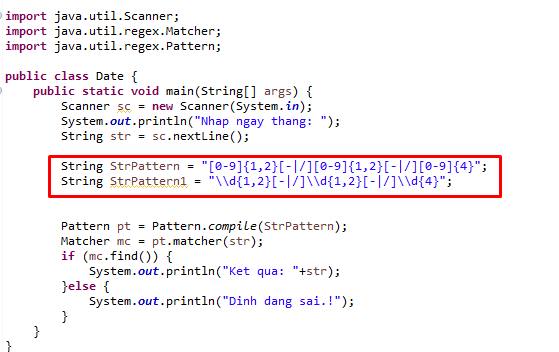
* 1. Ví dụ

- Định dạng ngày tháng

+) Cú pháp: ^[0-9]{1,2}[-|/][0-9]{1,2}[-|/][0-9]{4}$

+) Cách 2 : \\d{1,2}[-|/]\\d{1,2}[-|/]\\d{4}

+) Ý nghĩa: ^ bắt đầu dãy số từ 0 đến 9 [0-9] và có từ 1 đến 2 số {1,2}. Tiếp theo là [-|/] có nghĩa sẽ là dấu – hoặc dấu /. Các ký hiệu khác tương tự cho đến $ là kết thúc chuỗi. Cách 2 ý nghĩa tương tự nhưng dấu \d thay đế cho [0-9] và vì chuỗi đặt trong dấu “” nên cần phải thêm dấu \ cho nên ký tự cần dấu \\d .



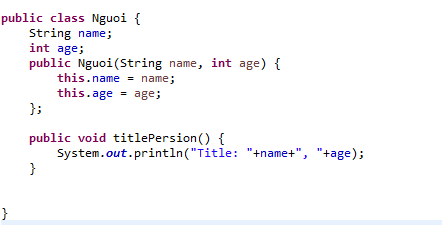
- Định dạng email

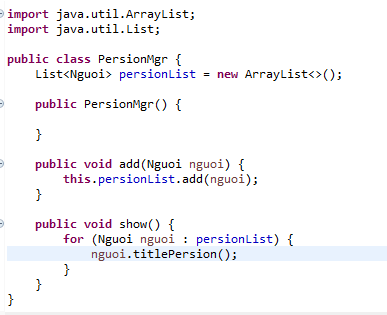
+)

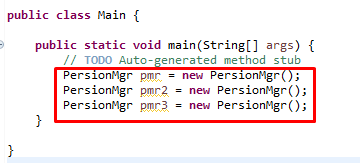
+)

1. Design Pattern
   1. Singleton Pattern

- Bài toán:

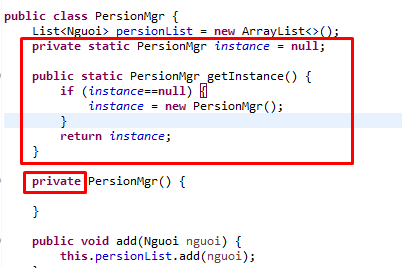


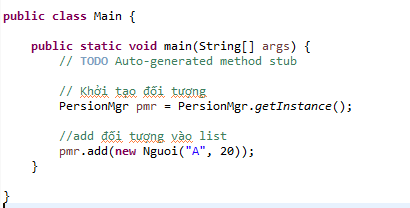




+) Với cách viết thông thường ta có thể tạo được 3 đối tượng khác nhau cho thằng PersionMgr. Bài toán đặt ra là làm sao để ta chỉ có thể tạo ra một đối tượng duy nhất cho thằng PersionMgr để ta dễ dàng quản lý các đối tượng.

- Giải pháp





+) Ở cách trên ta thấy:

1. Hàm tạo được đổi thành mức truy cập private nên sẽ không thể khởi tạo đối tượng trong class khác được.
2. Phương thức getInstance() được khai báo kiểu static. Là phương thức tĩnh nên nó sẽ truy cập trực tiếp thông qua lớp đối tượng.
3. Đối tượng PersionMgr là đối tượng instance và cũng là kiểu static nên nó cũng sẽ chỉ là đối tượng duy nhất.
4. Trong phương thức getInstance() nếu instance == null thì khởi tạo đối tượng. Còn nếu khác thì return ra luôn.

- Ứng dụng

+) Kiểu singleton cho phép chúng ta khai báo một đối tượng của một class để quản lý toàn bộ project. Để quản lý đối tượng dễ dàng hơn.

* 1. JFIYFCIY

- Bài toán

- Giải pháp

- Ứng dụng

* 1. Jfjy

1. Hướng dẫn phát triển dự án
   1. Bước 1: Tạo 1 project
   2. Bước 2: Chuột phải vào project chọn properties ->Build->Packaging->Compress JAR file->ok
   3. Bước 3: Nếu trong dự án có rất nhiều file và nhiều phương thức main thì ta chuột phải vào project chọn properties ->Run->Trong Main class click Browse-> tìm file muốn chạy đầu tiên
   4. Bước 4: Chuột phải vào project chọn Clean and Build
   5. Vậy là thành công.!
   6. Để chạy được dự án. Vào cmd gõ: java –jar “File-du-an.jar”