**BỘ NÔNG NGHIỆP VÀ PHÁT TRIỂN NÔNG THÔN**

**PHÂN HIỆU ĐẠI HỌC THỦY LỢI**



**Tên đề tài:**

**Các vấn đề về luồng trong Java thông qua việc mô phỏng Hệ Thống Giao Dịch Ngân Hàng**

**Giảng viên hướng dẫn: Nguyễn Thị Kim Phụng**

**Sinh viên thực hiện: Nguyễn Văn Khánh Duy - 2051067158**

**Nguyễn Hoài Nguyệt An - 2051067525**

**Nguyễn Phạm Xuân Hiền - 2051067165**

**Nguyễn Phùng Hưng - 2051067550**

**Trần Văn Tân - 2051067560**

**TP.HCM, 25 tháng 9 năm 2022**

**MỤC LỤC**

1. Giới thiệu đề tài

2. Nội dung chính của đề tài

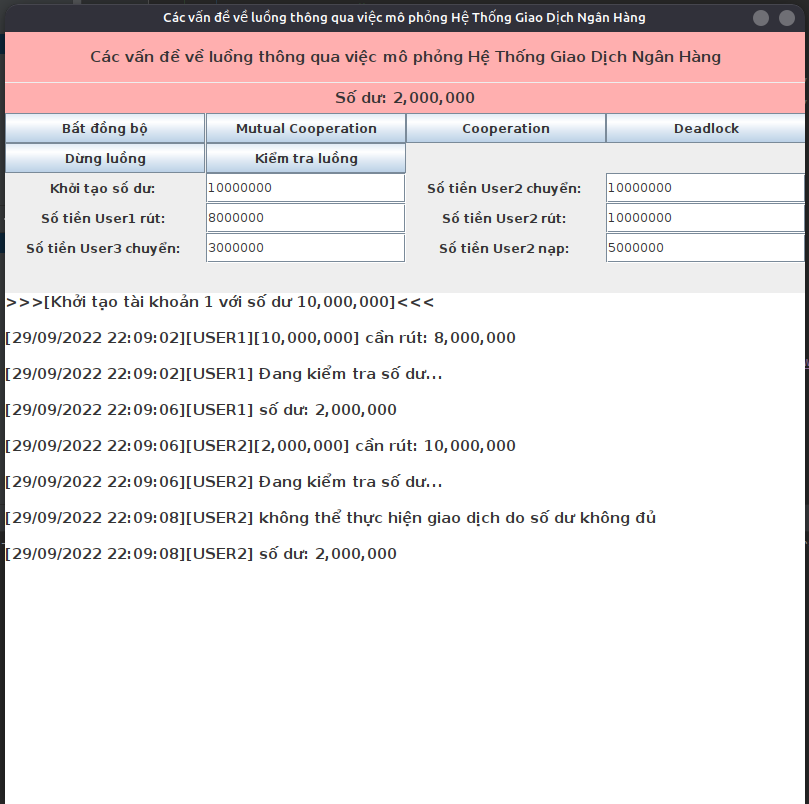
3. Miêu tả chương trình

4. Miêu tả giao diện chương trình

5. Tài liệu tham khảo

**Các vấn đề về luồng thông qua việc mô phỏng Hệ Thống Giao Dịch Ngân Hàng**

**Mã nguồn:** <https://github.com/duynguyen02/banking-system-with-thread>



**1. Giới thiệu đề tài**

**Tiểu trình (thread)**

Một tiến trình có thể tạo nhiều tiểu trình, mỗi tiểu trình thực hiện một chức năng nào đó và thực

thi đồng thời cũng bằng cách chia sẻ CPU. Các tiểu trình trong cùng một tiến trình dùng chung

không gian địa chỉ tiến trình nhưng có con trỏ lệnh, tập các thanh ghi và stack riêng. Một tiểu

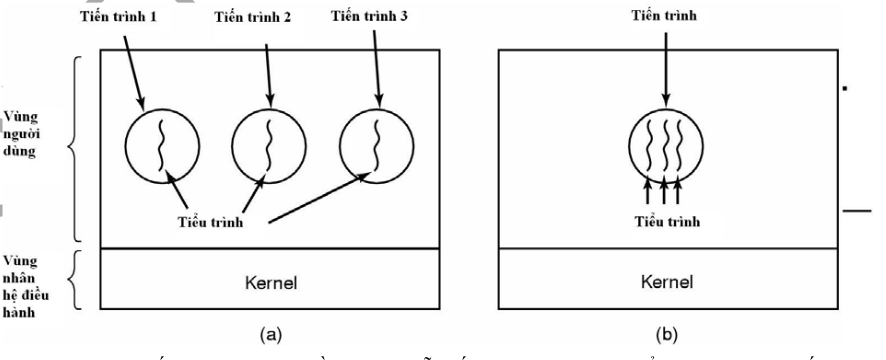
trình cũng có thể tạo lập các tiến trình con, và nhận các trạng thái khác nhau như một tiến trình.

**Liên lạc giữa các tiểu trình:**

Các tiến trình chỉ có thể liên lạc với nhau thông qua các cơ chế do hệ điều hành cung cấp. Các tiểu

trình liên lạc với nhau dễ dàng thông qua các biến toàn cục của tiến trình. Các tiểu trình có thể do

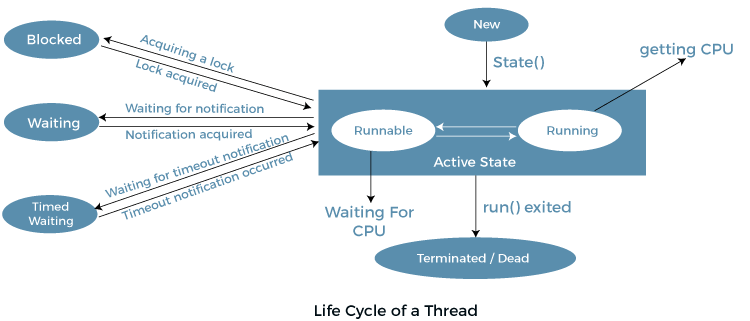
hệ điều hành quản lý hoặc hệ điều hành và tiến trình cùng phối hợp quản lý.



Hình 3.2: a) ba tiến trình thực thi đồng thời, mỗi tiến trình chỉ có một tiểu trình. b) một tiến trình

có ba tiểu trình, việc hoạt động đồng thời của các tiểu trình là do tiến trình quản lý.

**Vòng đời của một tiểu trình:**

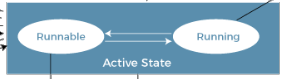


a/ New: tiểu trình mới được tạo đang ở trong bộ nhớ tạm trên đĩa cứng.



b/ Runnable: tiểu trình trong bộ nhớ và chờ được cấp phát CPU.

c/ Running: tiểu trình trong bộ nhớ đang thực thi.



d/ Blocked, waiting, timed\_waiting: Trong quá trình tiểu trình đang chạy, nếu có bất kỳ tác động nào, ngoại trừ làm kết thúc vòng đời của Thread, nó sẽ vào trạng thái **BLOCKED**, hoặc **WAITING**, hoặc **TIMED\_WAITING**, ở giai đoạn này, tiểu trình trong bộ nhớ chờ được cấp phát tài nguyên, hoặc chờ thao tác nhập/xuất hoàn tất hoặc chờ một sự kiện nào đó.

e/ Terminated: tiểu trình trong bộ nhớ hoàn tất xử lý.

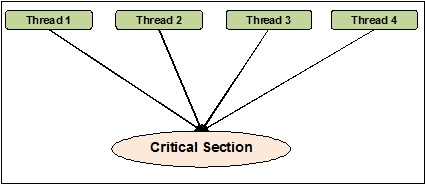


**Đồng bộ hóa luồng:**

Đồng bộ hóa luồng là việc thực hiện đồng thời hai hoặc nhiều luồng trong việc chia sẻ các tài nguyên quan trong . Các luồng nên được đồng bộ hóa để tránh xung đột sử dụng tài nguyên. Nếu không, xung đột có thể phát sinh khi các luồng chạy song song cố gắng sửa đổi một biến chung cùng một lúc.

**Critical Section trong đồng bộ hóa và sự tranh chấp tài nguyên dùng chung:**

Khi hơn 1 luồng truy cập vào cùng một đoạn mã thì đoạn mã đó gọi là Critical Section . Critical Section chứa các biến và tài nguyên cần được đồng bộ để duy trì tính nhất quán giữa các biến dữ liệu, nói một cách đơn giản Critical Section là tập hợp các câu lệnh truy cập đến tài nguyên dùng chung chẳng hạn như tệp tin, các cổng io, dữ liệu toàn cục,… Trong concurrent programming, nếu một luồng cố gắng thay đổi giá trị của một dữ liệu được chia sẻ trong khi một luồng khác cố gắng đọc giá trị, kết quả sẽ không thể đoán được chuyện gì sẽ xảy ra.



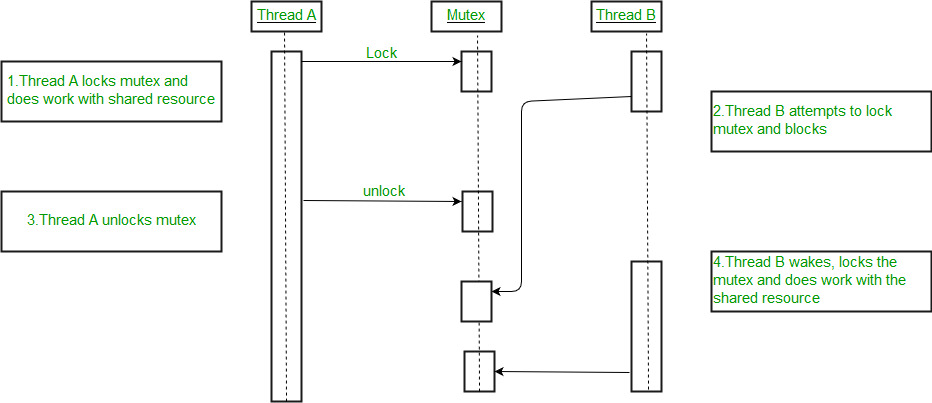
**Các phương thức đồng bộ hóa luồng:**

1. Mutual Exclusion (Loại trừ lẫn nhau):

- đây là một thuộc tính của quá trình đồng bộ hóa được biểu thị rằng: không thể nào tồn tại 2 luồng đồng thời trong một Critical Section trong bất kỳ thời điểm nào.

- Cơ chế đồng bộ của Mutual Exclusion (Monitor Lock) : mỗi một Critical Section sẽ được một đối tương là Monitor bảo hộ. Khi các luồng muốn sử dụng Critical Section, các luồng đó phải đăng ký qua Monitor để có được Lock và mỗi Monitor chỉ có một Lock, luồng nào nhận được Lock sẽ được phép sử dụng Critical Section. Khi luồng nắm giữ Lock hoàn thành, Lock sẽ được chuyển cho luồng tiếp theo đang trong danh sách hàng đợi của Monitor, cứ như vậy Lock sẽ được truyền nhau cho từng luồng còn đợi trong Monitor.

2. Cooperation (Cộng tác):



- đây là phương thức đồng bộ trong đó luồng đang nắm giữ Critical Section có quyền tạm dừng và trao lại Lock của Critical Section nó đang nắm giữ cho một luồng khác đến khi luồng đó thực hiện xong công việc của mình và trao trả lại Lock cho nó tiếp tục thực hiện công việc bị tạm dừng của mình.

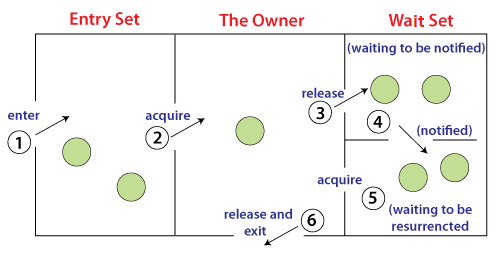
- cơ chế:

+ wait: luồng đang nắm giữa Lock phải giả phóng Lock trả Lock lại cho Monitor đồng thời luồng sẽ rơi vào trạng thái ngủ đợi cho một luồng khác “đánh thức dậy” hoặc tự động dậy sau một khoảng thời gian nhất định.

+notify: luồng đang trong trạng thái ngủ được đánh thức để tiếp tục thực hiện công việc của nó. Nếu có nhiều luồng trong trạng thái ngủ, thì sẽ được đánh thực ngẫu nhiên một trong các luồng.

+notifyAll: tất cả các luồng đang trong trạng thái ngủ sẽ được đánh thức.

- quá trình giao tiếp giữa các luồng:



+ (1) luồng đang yêu cầu Lock

+ (2) Lock đã được nắm giữa bởi luồng

+ (3) luồng sẽ vào trạng thái ngủ nếu wait được gọi, nếu không thì nó sẽ trả lại Lock và thoát ra ngoài.

+ (4) nếu notify (hoặc notifyAll) được gọi, luồng được chuyển vào trạng thái được đánh thức.

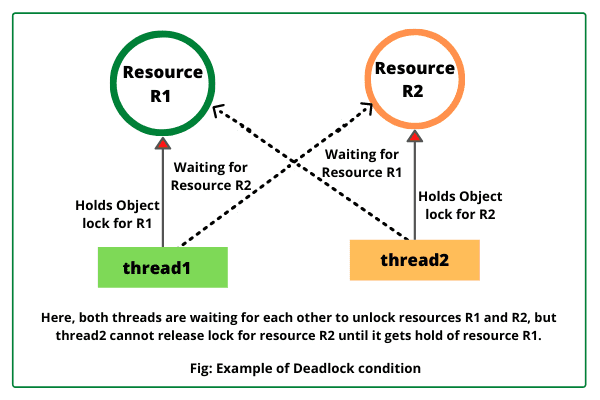
+ (5) bây giờ luồn có thể nhận lại Lock

+ (6) sau khi hoàn thành Task, luồng trả lại Lock cho Monitor và thoát ra ngoài.

**Deadlock:**

- Deadlock là tình huống xảy ra khi một luồng đang chờ đợi nhận Lock đang được nắm giữ bởi một luồng khác và luồng đó cũng đang chờ đợi nhận Lock đang được nắm giữ của luồng còn lại.

- Khi cả hai luồng chờ đợi lẫn nhau để cả hai cùng nhả Lock cho nhau, nhưng không không luồng nào chịu nhả Lock dẫn đến Lock của cả hai luồng không được giải phóng cho đến khi quá trình thực hiện của nó không hoàn thành. (tức là Lock chỉ được nhả khi cả hai luồng thực hiện xong, nhưng để thực hiện xong thì cả 2 luồng phải cần lấy Lock lẫn nhau).



**Điều kiện để xảy ra Deadlock:**

- Mutual Exclusion(Loại trù lẫn nhau): Có ít nhất một tài nguyên phải được giữ ở chế độ không thể chia sẻ và do đó, chỉ có thể được sử dụng bởi một luồng. Nếu một luồng khác yêu cầu nó, nó sẽ đợi cho đến khi tài nguyên nhàn rỗi.

-Hold & wait (giữ và chờ): điều kiện này xảy ra khi một luồng nắm giữ tài nguyên và đợi một tài nguyên đang được nắm giữ bởi một luồng khác.

-No Preemption(không có quyền ưu tiên) tài nguyên chỉ giải phóng sau khi luồng hoàn thành xong công việc.

- Circular wait (chu trình đợi theo vòng): điều kiện này xảy ra khi mỗi luồng đợi tài nguyên do một luồng trước đó nắm giữ và luồng cuối cùng đợi tài nguyên do luồng đầu tiên nắm giữa (theo cơ chế vòng tròn).

**Các phòng tránh deadlock:**

Không có giải pháp chính xác để khắc phục vấn đề bế tắc trong một chương trình. Nó phụ thuộc vào logic được sử dụng bởi lập trình viên. Lập trình viên nên tạo chương trình của mình theo cách mà nó không hình thành Deadlock. Có thể ngăn chặn Deadlock trong các điều kiện xảy ra Deadlock bao gồm:

- Mutual Exclusion: Deadlock sẽ không bao giờ xảy ra nếu tất cả tài nguyên được chia sẻ bởi nhiều luồng.

-Hold and Wait Condition:điều kiện này có thể bị loại bỏ nếu một luồng đang nắm giữ tài nguyên bị cấm chờ thêm một số tài nguyên nhất định. Tức là ta có thể khai báo từ đầu tất cả tài nguyên mà luồng sẽ sử dụng.

- No Preemption Condition: điều kiện này có thể bị loại bỏ nếu một luồng nắm giữ một tài nguyên nhất định bị từ chối yêu cầu thêm. Luồng đó phải mở khóa tài nguyên ban đầu của nó. Nếu cần thiết, hãy yêu cầu chúng một lần nữa cùng với tài nguyên bổ sung.

-Circular Wait Condition:

+ cách 1: Một luồng buộc chỉ giữ một tài nguyên tại một thời điểm. Nếu nó cần một tài nguyên khác, trước tiên nó phải giải phóng tài nguyên đó do nó nắm giữ và sau đó yêu cầu một tài nguyên khác.

+ cách 2: thu thập các tài nguyên theo một thứ tự cụ thể và giải phóng chúng theo thứ tự ngược lại để một luồng chỉ có thể tiếp tục nhận được một tài nguyên nếu nó nắm giữ một tài nguyên khác.

**Các vấn đề của luồng thông qua mô phỏng hệ thống giao dịch ngân hàng:**

Các ứng dụng chạy đa luồng đem lại rất nhiều lời ích, giúp tận dụng được các CPU đa nhân để phân công cho cho mỗi nhận vật lý 1 luồng xử lý để tăng tốc độ tính toán của bài toán. Hay hiện tại trong 1 số ứng dụng sử dụng GPU để tính toán hiệu năng cao dựa vào số nhân vô cùng lớn của GPU.

Một hệ thống giao dịch ngân hàng bao gồm các dịch vụ nạp tiền, rút tiền, chuyển tiền, thanh toán,... Để hệ thống được hoạt động ổn định, các dịch vụ sẽ được thực hiện trong các luồng để phục vụ cho nhiều người dùng trong cùng các thời điểm xác định.

Nhưng việc sử dụng luồng sai cách có thể dẫn đến các vấn đề phát sinh làm ảnh hưởng đến hệ thống. Sau đây qua việc phát triển một ứng dụng mô phỏng một hệ thống giao dịch ngân hàng, chúng tôi sẽ đưa ra các vấn đề phát sinh khi sử dụng luồng và các phương hướng giải quyết.

**2. Cài đặt và sử dụng:**

**2.1 Yêu cầu:**

* Ngôn ngữ lập trình: Java
* Nền tảng phát triển: openjdk 11.0.16 2022-07-19
* Công cụ và phần mềm: IntelliJ IDEA 2022.2.1 (Community Edition)

**2.2 Cài đặt và sử dụng:**

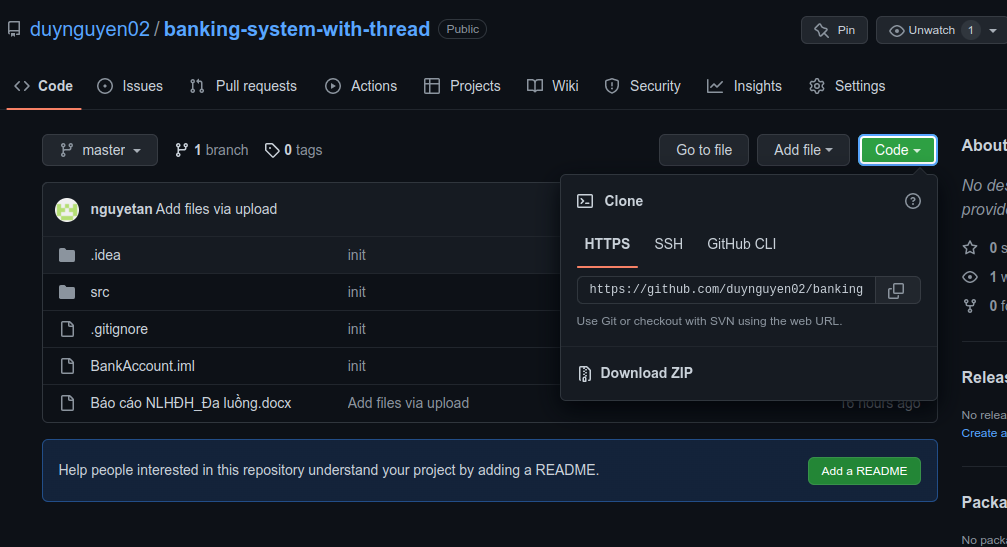
Tải mã nguồn:

* Cách 1:

+ Truy cập vào liên kết: <https://github.com/duynguyen02/banking-system-with-thread>

+ Nhấn vào mục Code -> Download ZIP

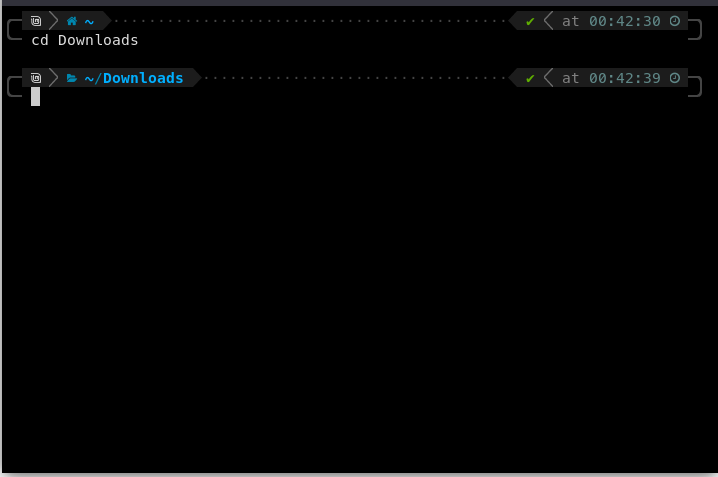
+ Giải nén tệp .zip vừa tải về.



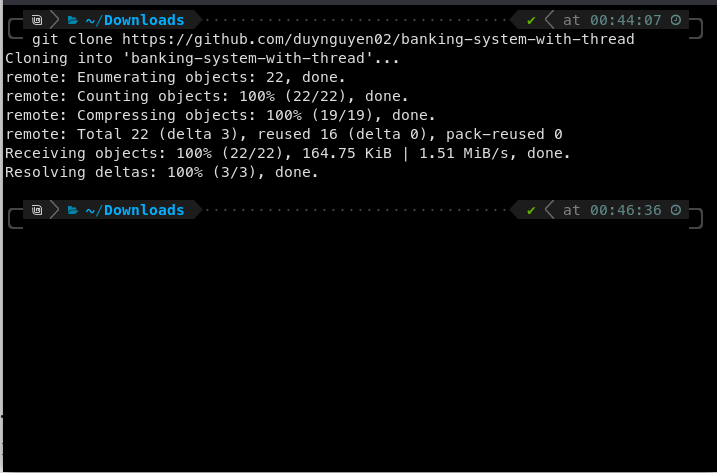
* Cách 2: yêu cầu đã cài đặt Git

+ Mở Terminal hoặc Command prompt

+ Di chuyển đến thư mục muốn lưu trữ mã nguồn



+ Nhập lệnh: git clone <https://github.com/duynguyen02/banking-system-with-thread>

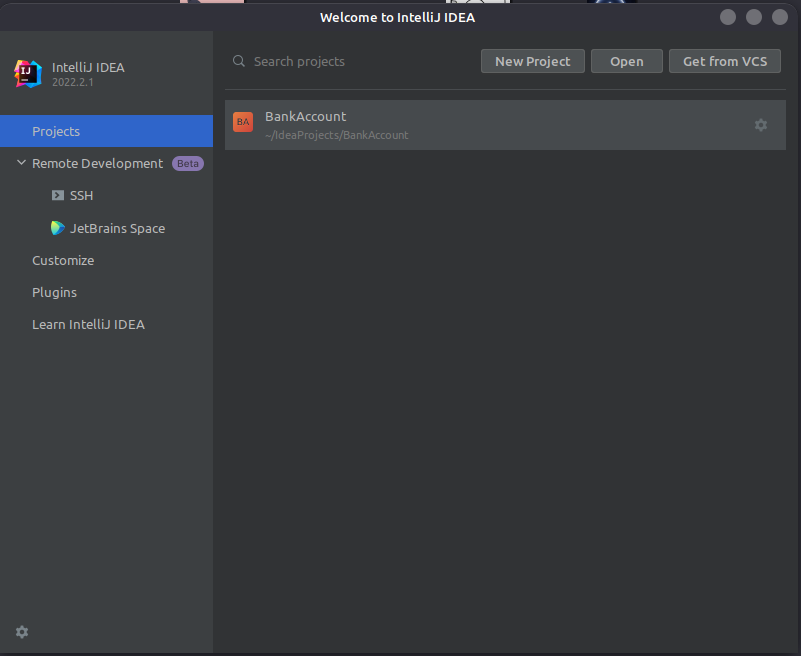


Mở mã nguồn và tiến hành chạy chương trình:

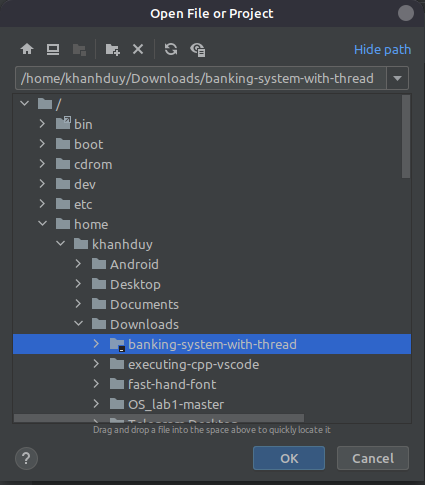
* Khởi động IntelliJ IDEA 2022.2.1 (Community Edition)



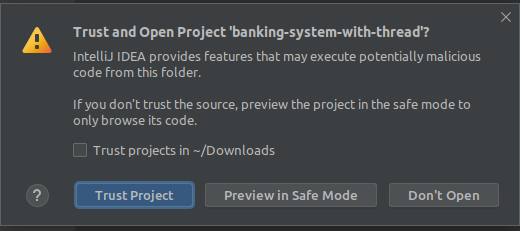
* Chọn Open



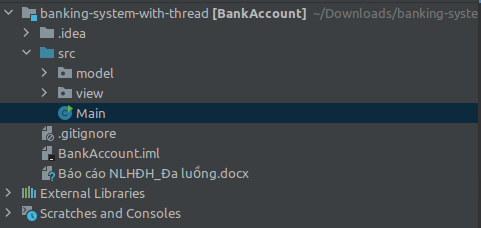
* Chọn thư mục mã nguồn đã tải trước đó (banking-system-with-thread)



* Hộp thoại cảnh báo hiện lên, chọn Trust project



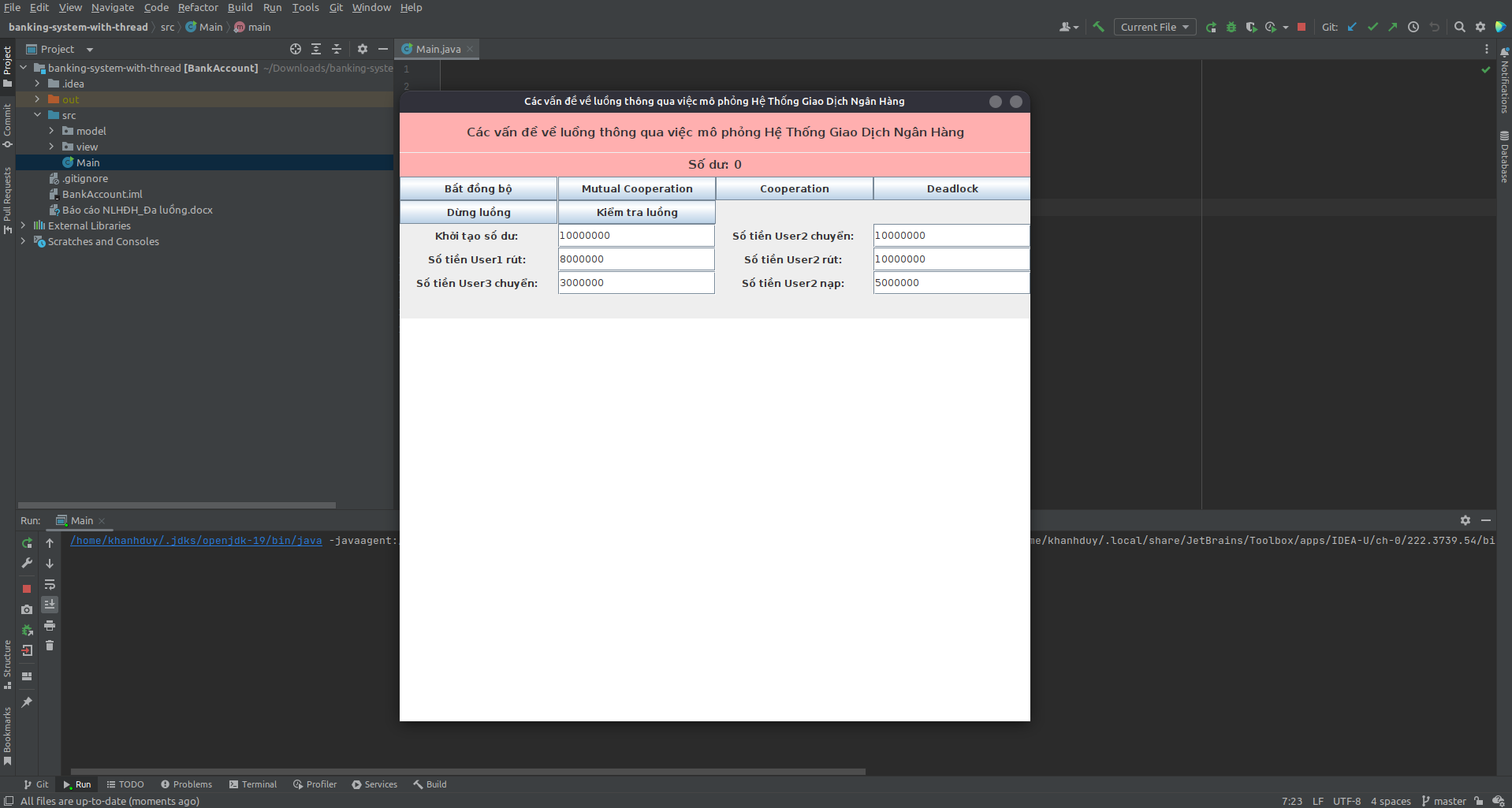
* Sau khi mở thành công mã nguồn, ở phía cây thư mục, click vào mũi tên phía bên trái thư mục src và double click vào file Main



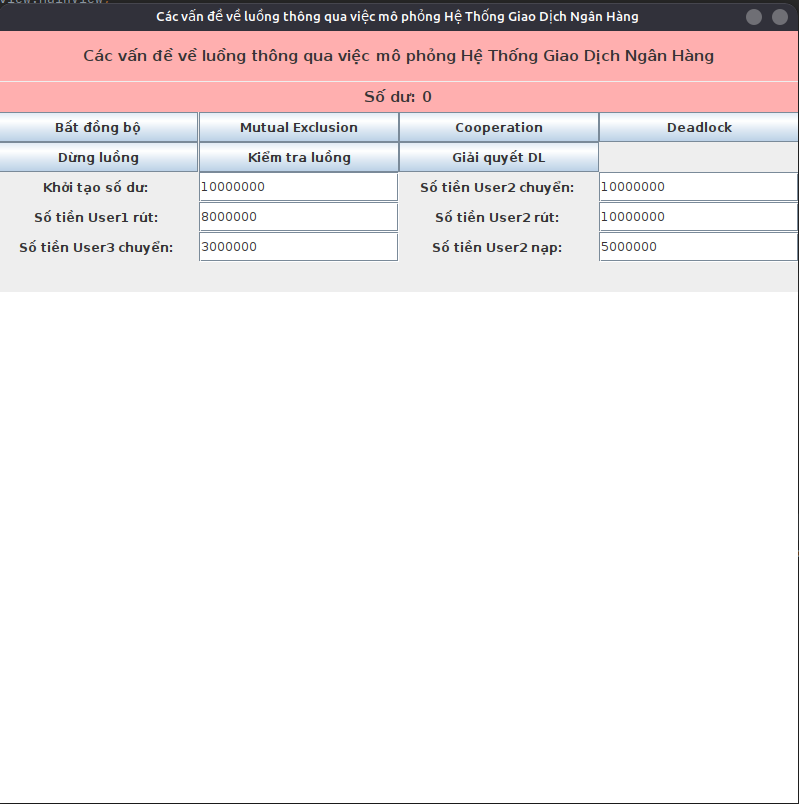
* Nhấn Shift + F10 để chạy chương trình hoặc nút phía bên trên IDE.



* Kết quả sau khi chạy chương trình.



**3. Miêu tả giao diện chương trình:**



Giao diện chính của chương trình

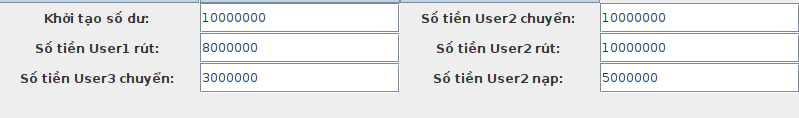


Biểu thị trạng thái của số dư tài khoản khi bị các luồng tác động.

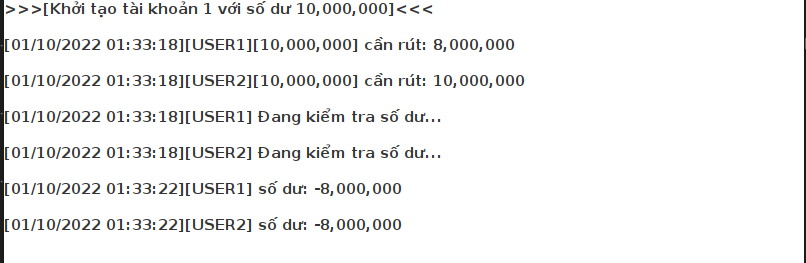


Các nút mô phỏng các vấn đề khi sử dụng luồng, trong đó:

* Bất đồng bộ: mô phỏng vấn đề tranh chấp tài nguyên chung của các luồng
* Mutual Exclusion: mô phỏng việc giải quyết đồng bộ luồng bằng Mutual Exclusion
* Cooperation: mô phỏng tình huống sử dụng Cooperation.
* Deadlock: mô phỏng tình huống xảy ra deadlock
* Dừng luồng: dừng các luồng đang chạy (lưu ý: việc dừng luồng ở đây mang tính chất tương đối vì các luồng vẫn chạy, các luồng đang chạy thật sự bị ngắt khi đối tượng luồng được tạo mới và khởi chạy).
* Kiểm tra luồng: kiểm tra trạng thái của các luồng.
* Giải quyết DL: giải quyết vấn đề deadlock.



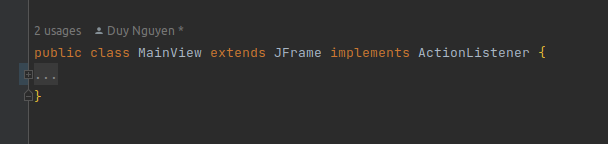
Khởi tạo các giá trị ban đầu cho tài khoản được tạo và các giá trị luồng sẽ tác động.



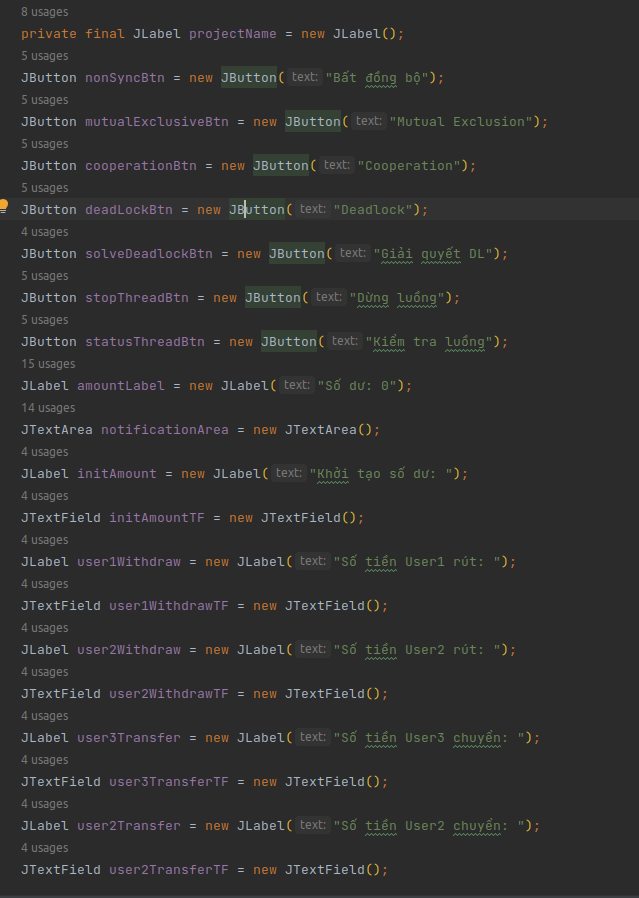
Hiển thị các trạng thái trong quá trình luồng thao tác với tài nguyên.

**4. Miêu tả chương trình**

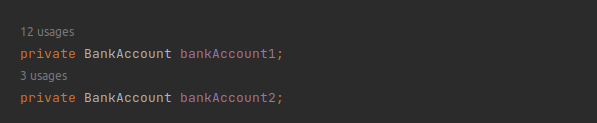
**4.1 Giao diện chương trình:**



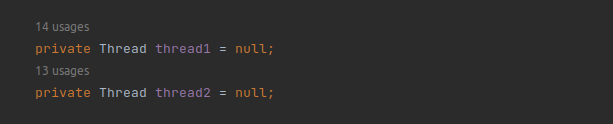
Lớp MainView: lóp định nghĩa giao diện chính cho chương trình



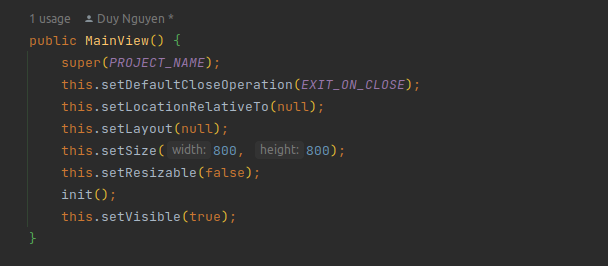
Khởi tạo các thành phần trong giao diện các nút, nhãn, và vùng hiển thị trạng thái ở lớp MainView



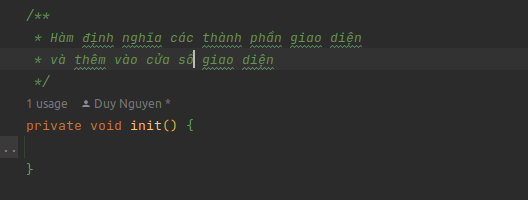
Định nghĩa lớp BankAccount để mô phỏng tại khoản ngân hàng, với biến bankAccount1 được sử dụng cho USER1 và USER2, biến bankAccount2 được sử dụng cho USER3. Chi tiết về lớp BankAccount sẽ được đề cập sau.



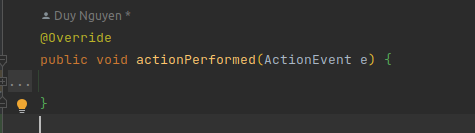
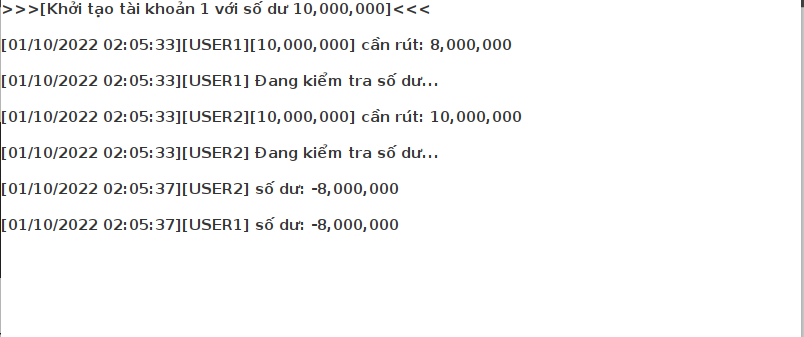
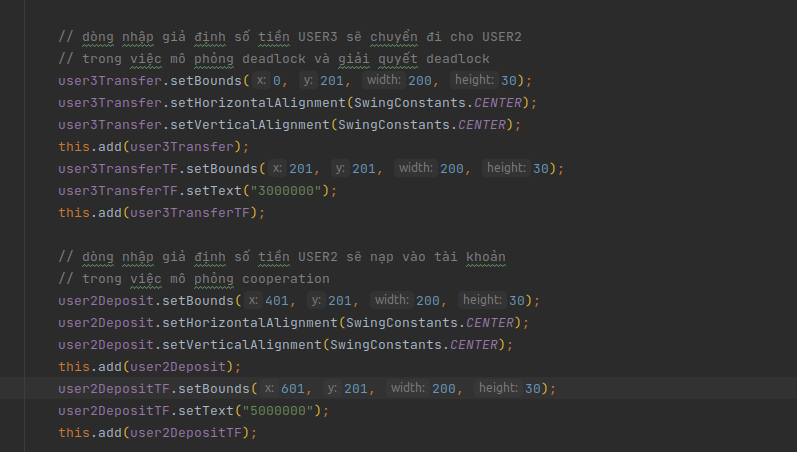
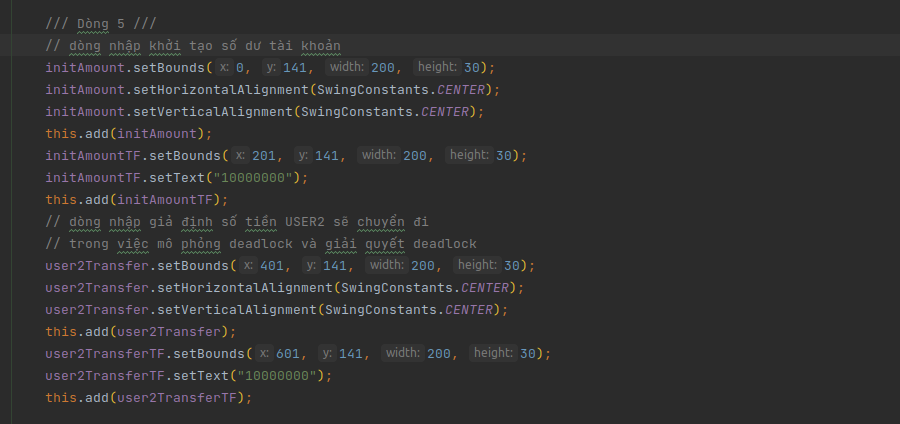
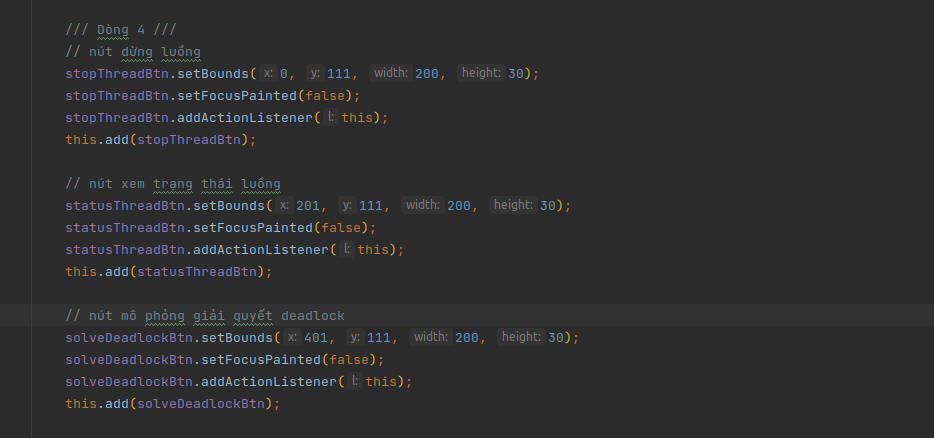
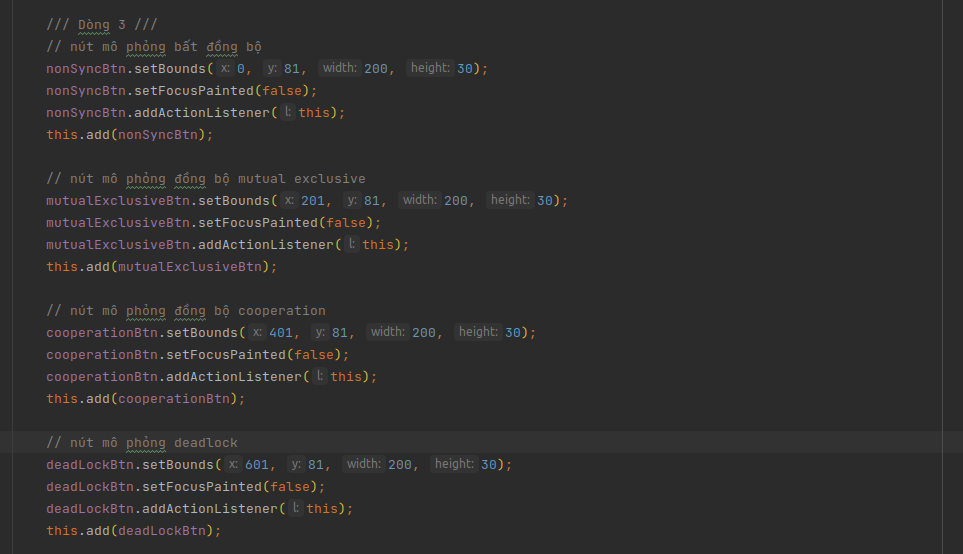
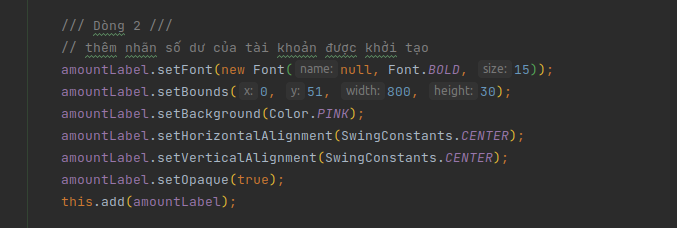
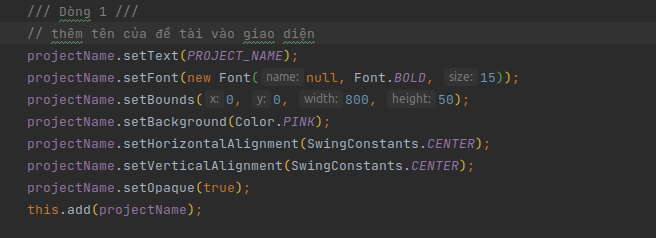
Định nghĩa lớp Thread để mô phỏng các luồng sẽ được sử dụng để chạy các dịch vụ được định nghĩa trong lóp BankAccount.



Định nghĩa Constructor để thiết đặt các thông số cho cửa số giao diện và thêm các thành phần đã định nghĩa ở trên thông qua hàm init()



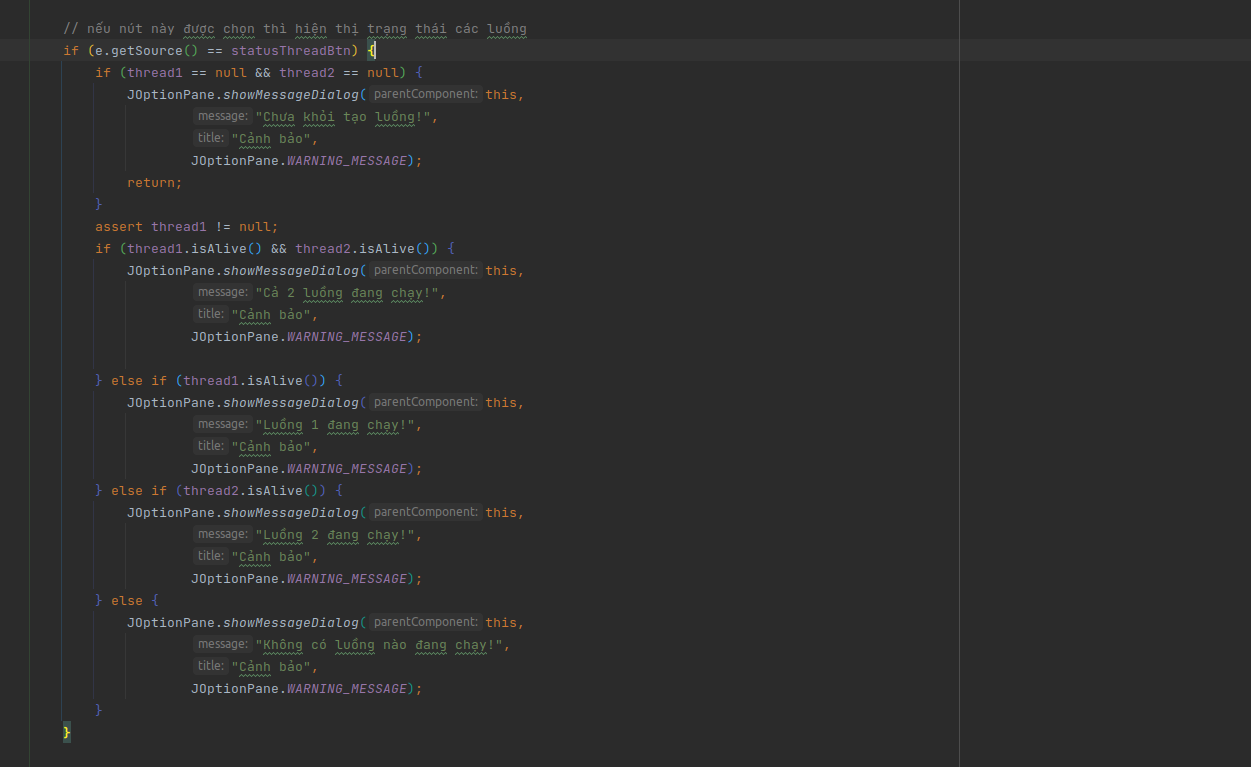
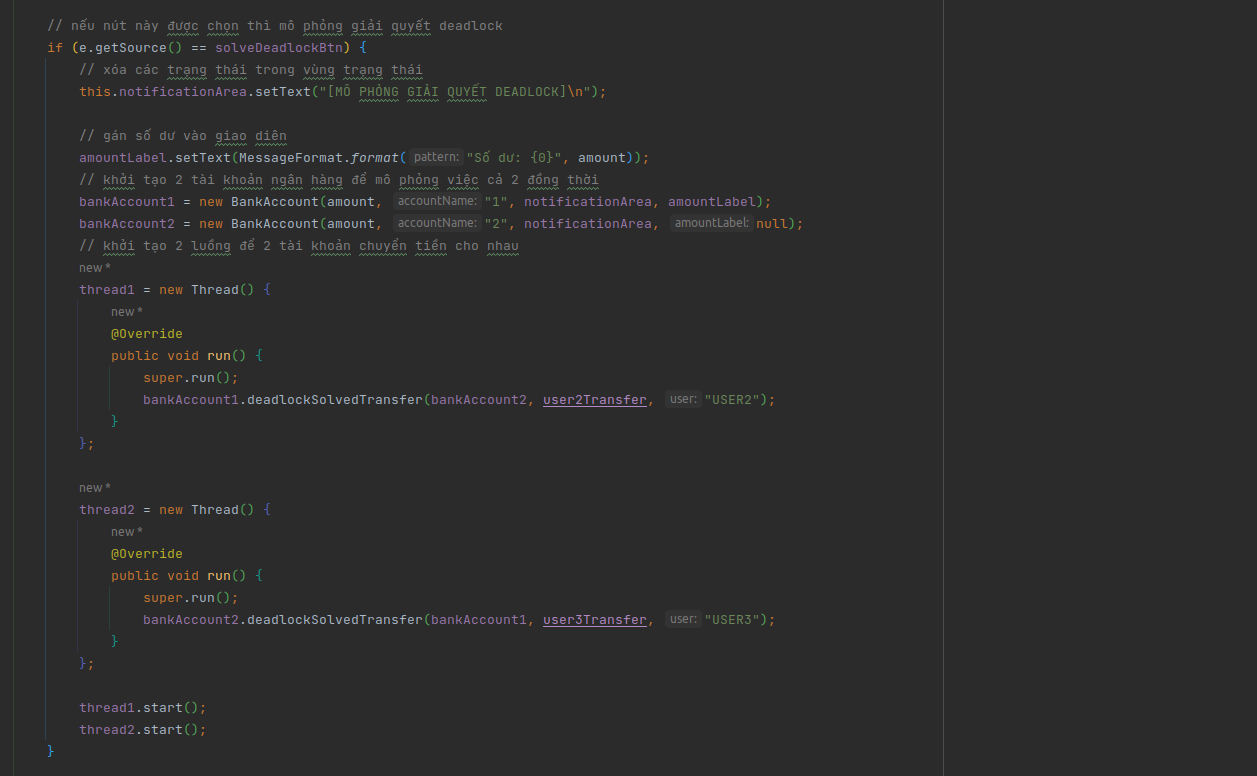
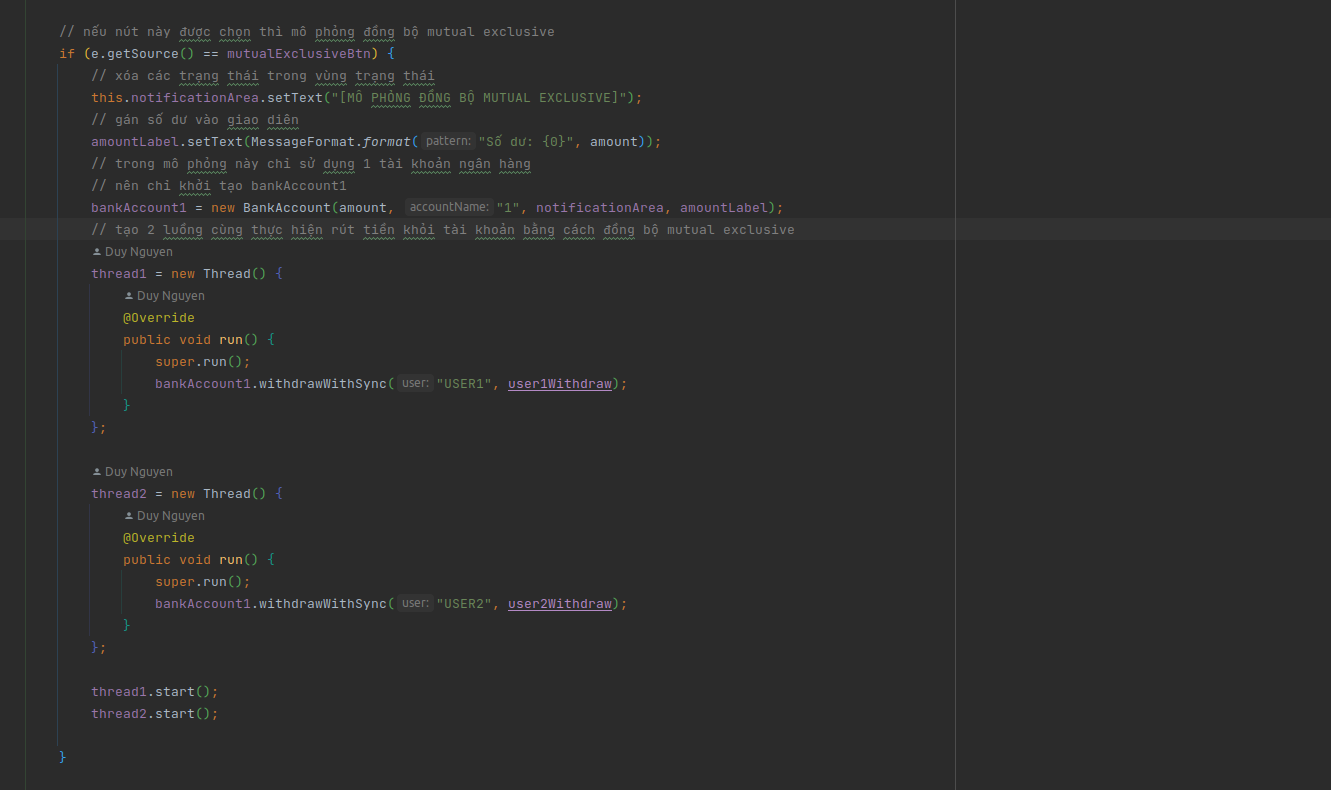
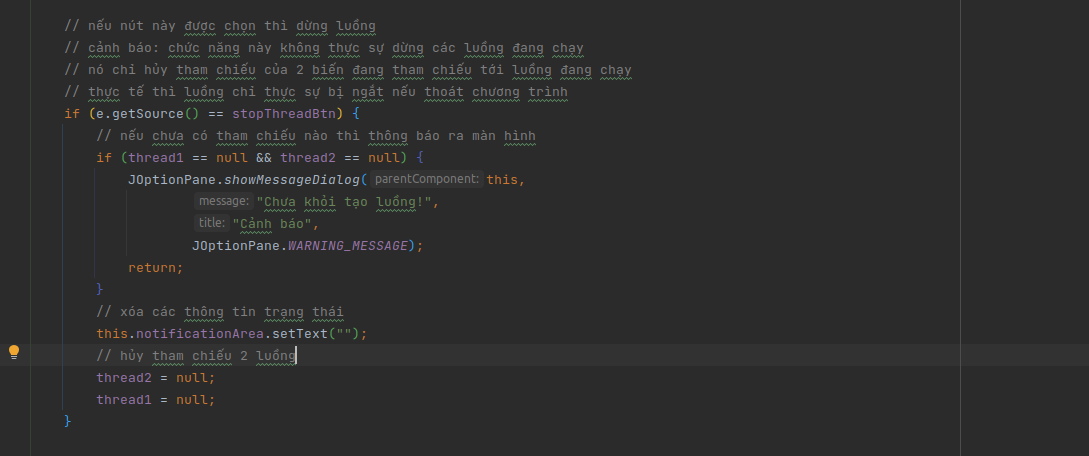
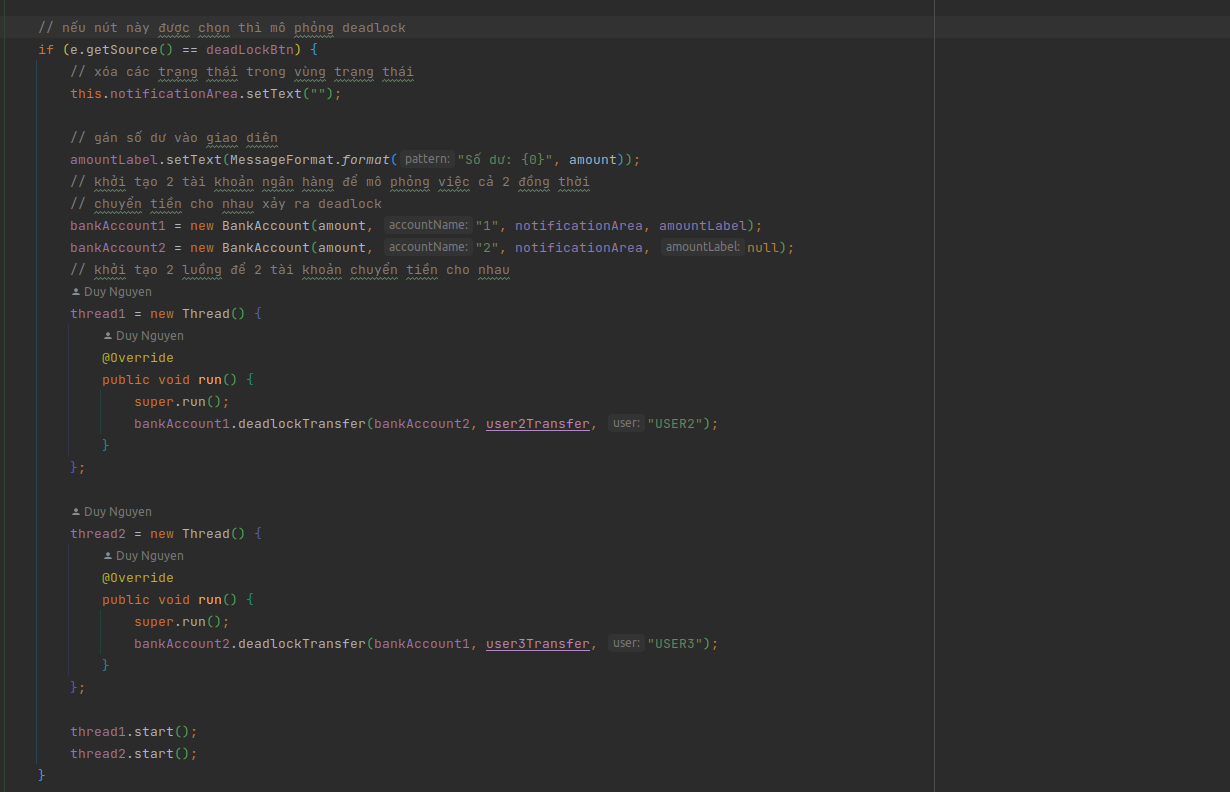
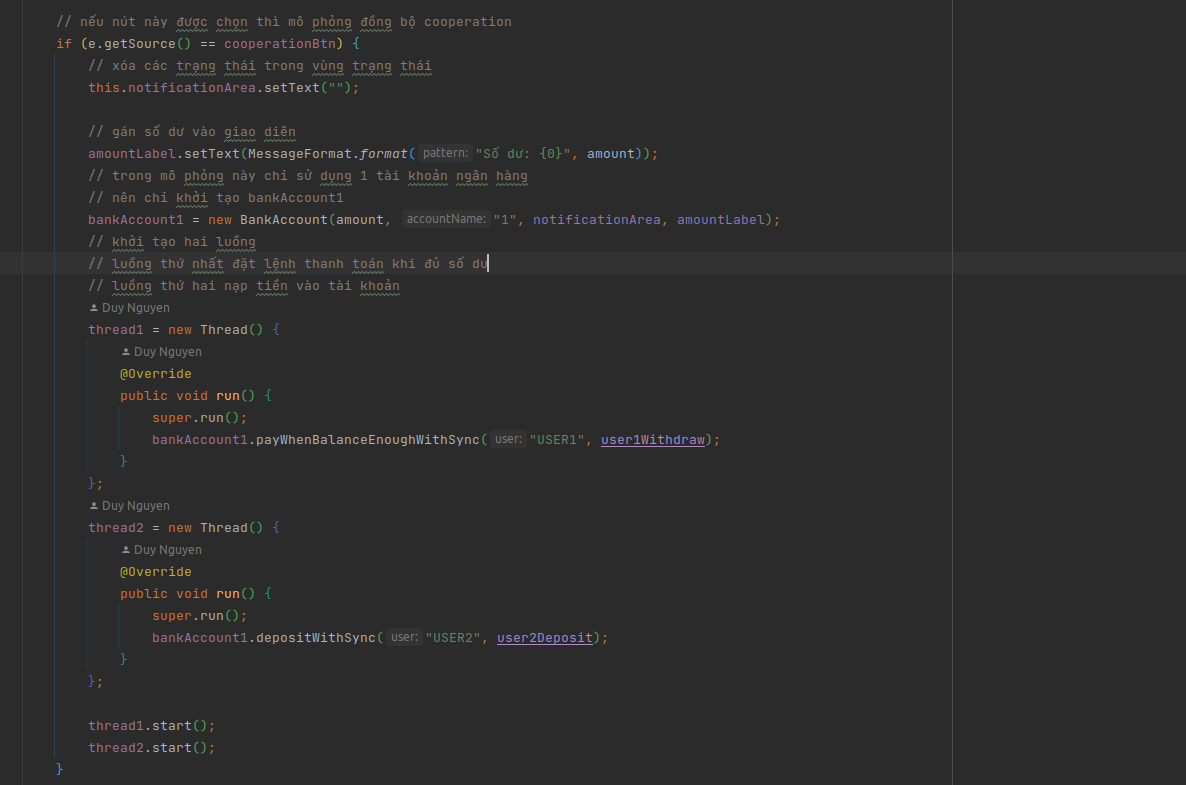
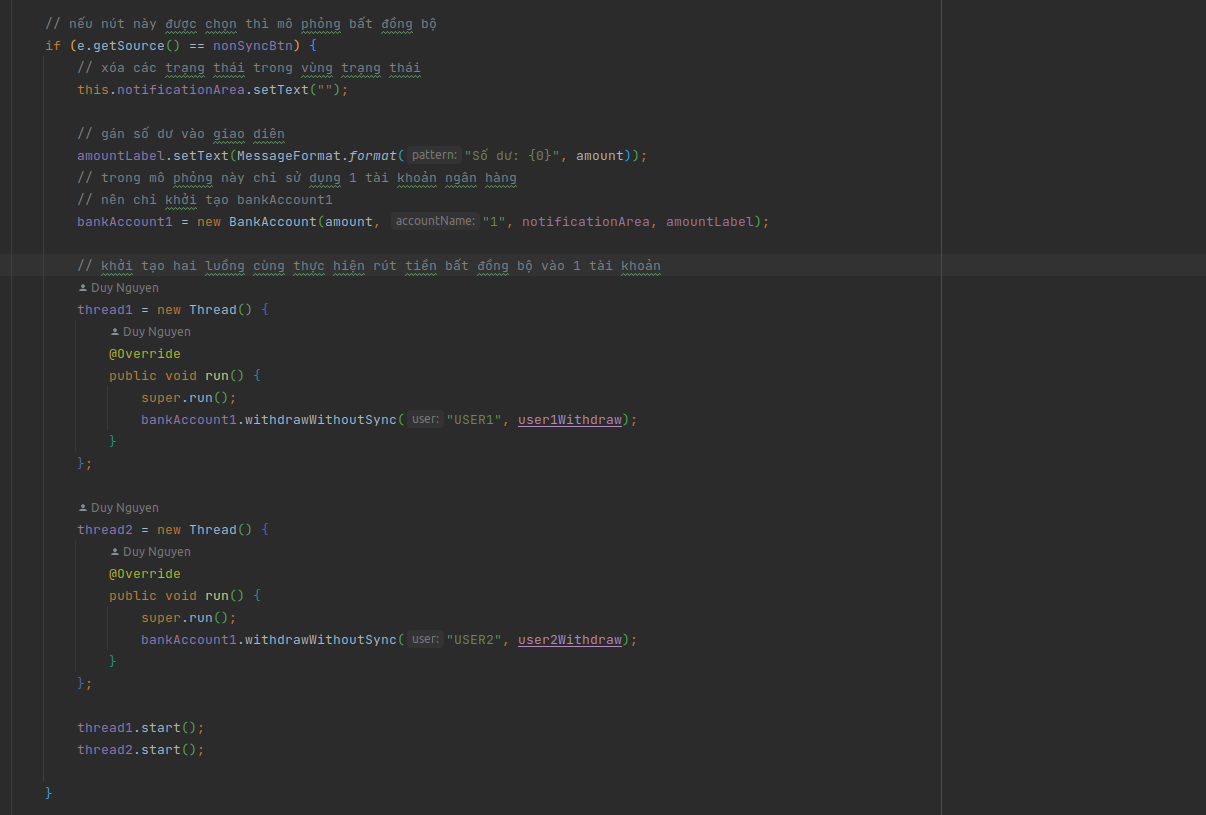
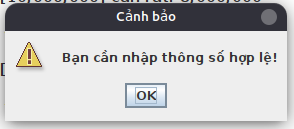
Định nghĩa hàm init(), các định nghĩa trong hàm init():



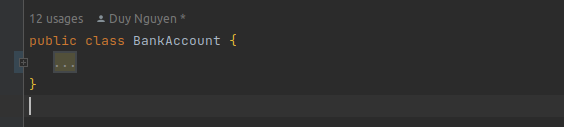
Hàm định nghĩa các thao tác khi click chọn các nút bấm trên giao diện.



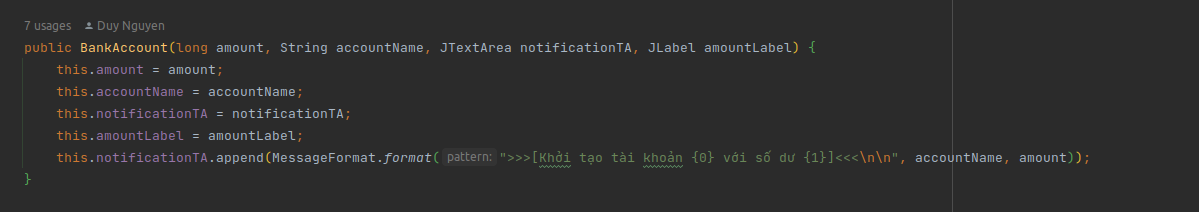
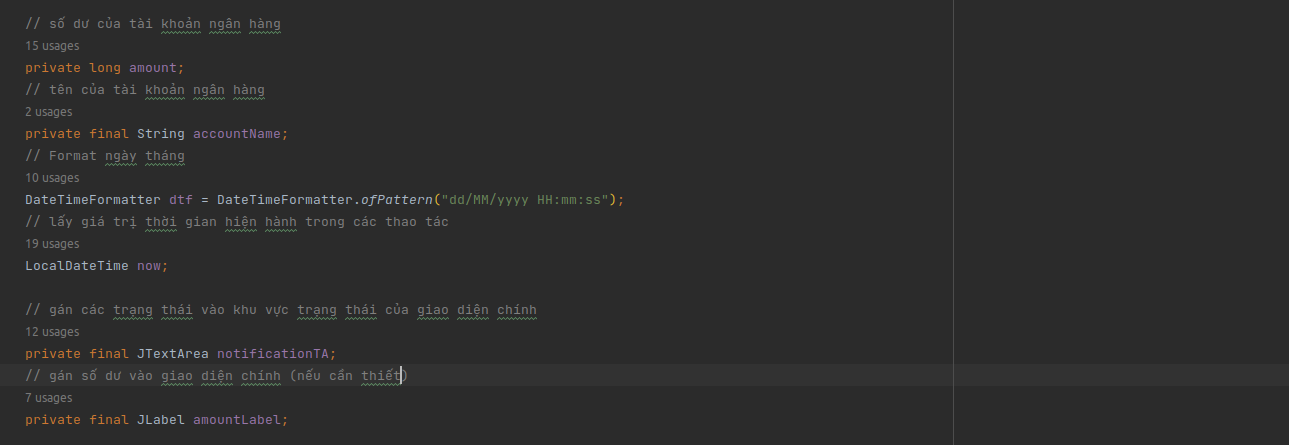
Lấy các giá trị bên trong các ô nhập các giá trị giả định phía trên và thử chuyển tất cả về kiểu dữ liệu long. Nếu xảy ra lỗi sẽ hiển thị hộp thoại thông báo lỗi. Nếu chuyển kiểu dữ liệu thành công thì kiểm tra xem người dùng nhấn vào nút nào để thực hiện mô phỏng tương ứng với nút bấm đó.



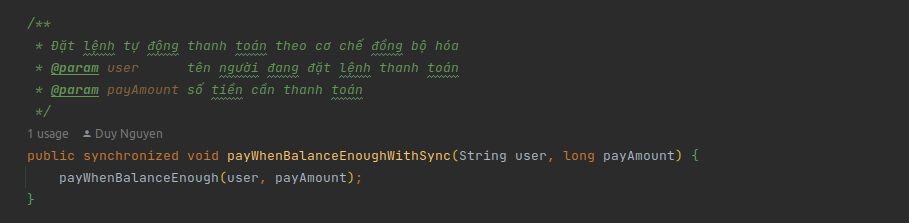
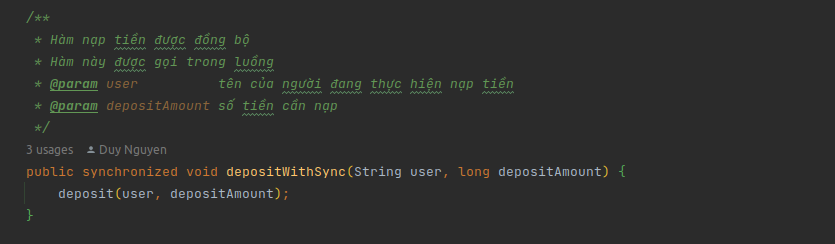
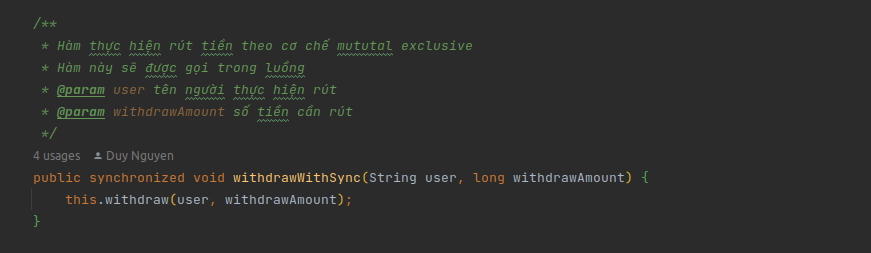
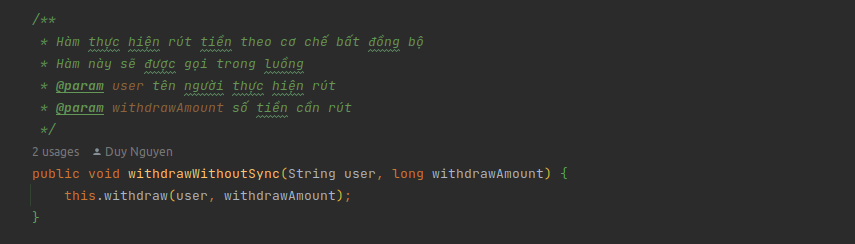
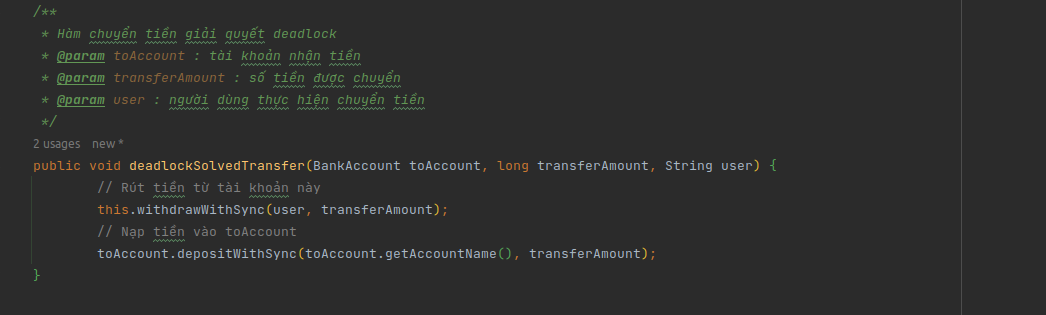
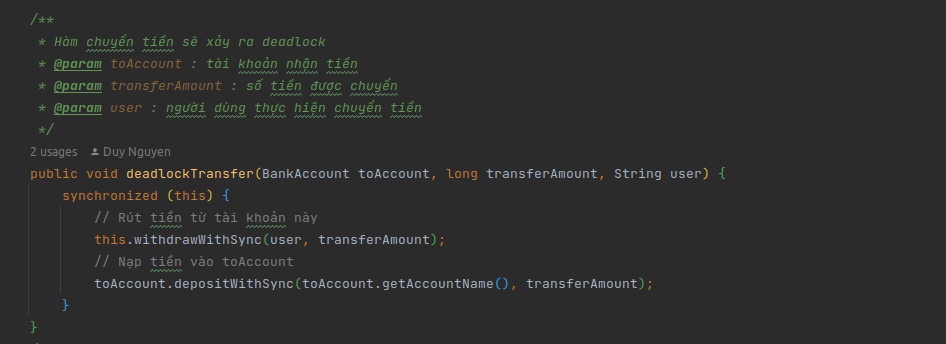
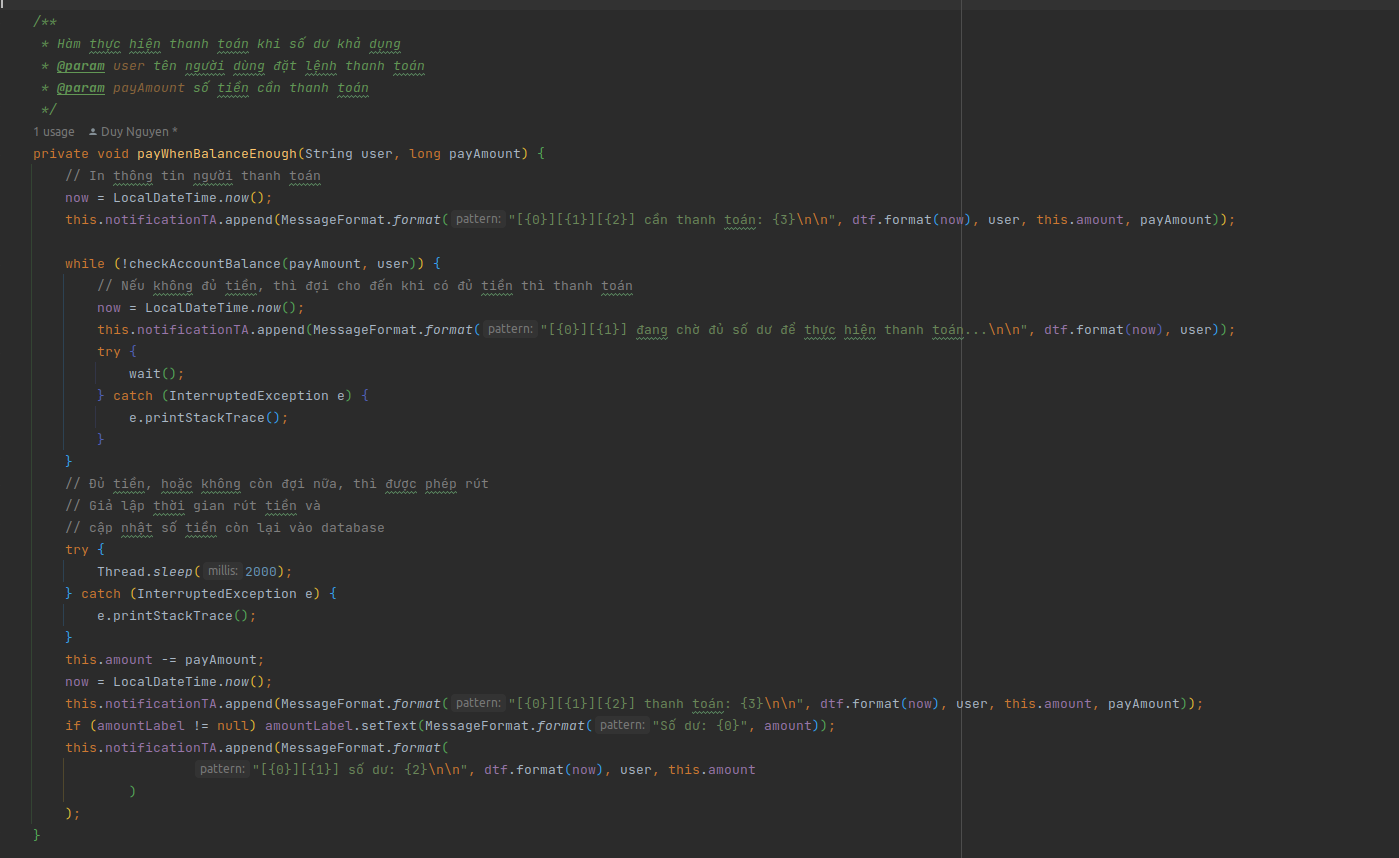
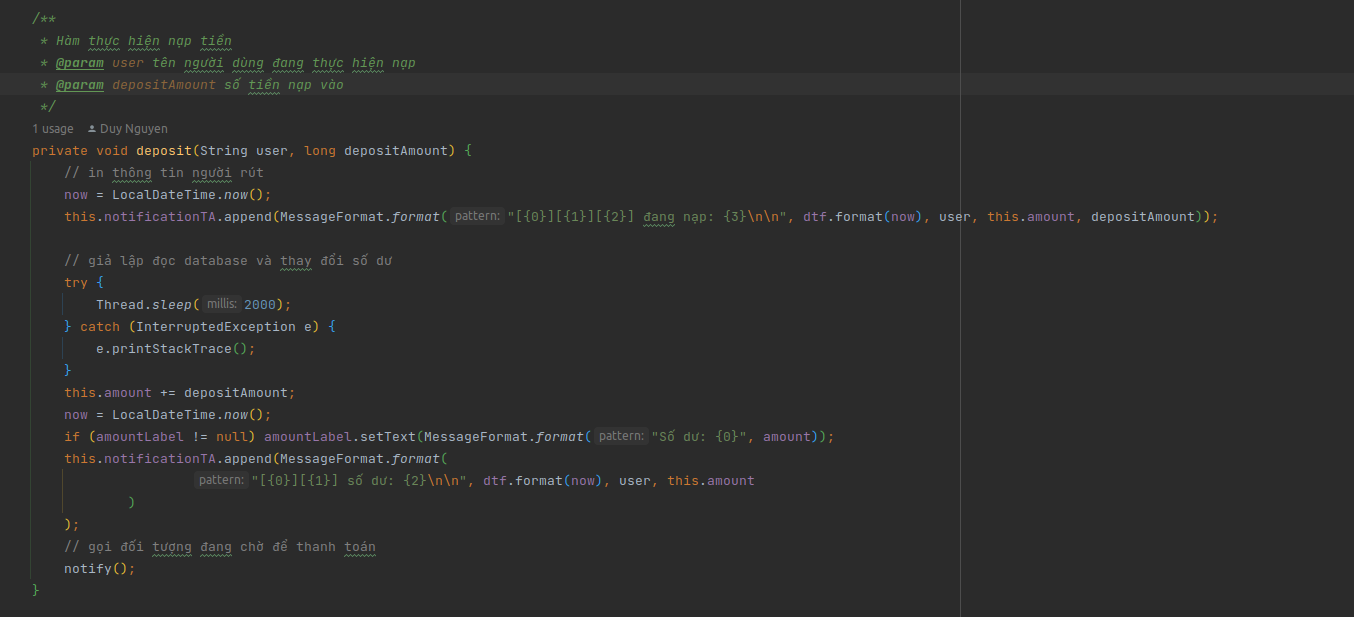
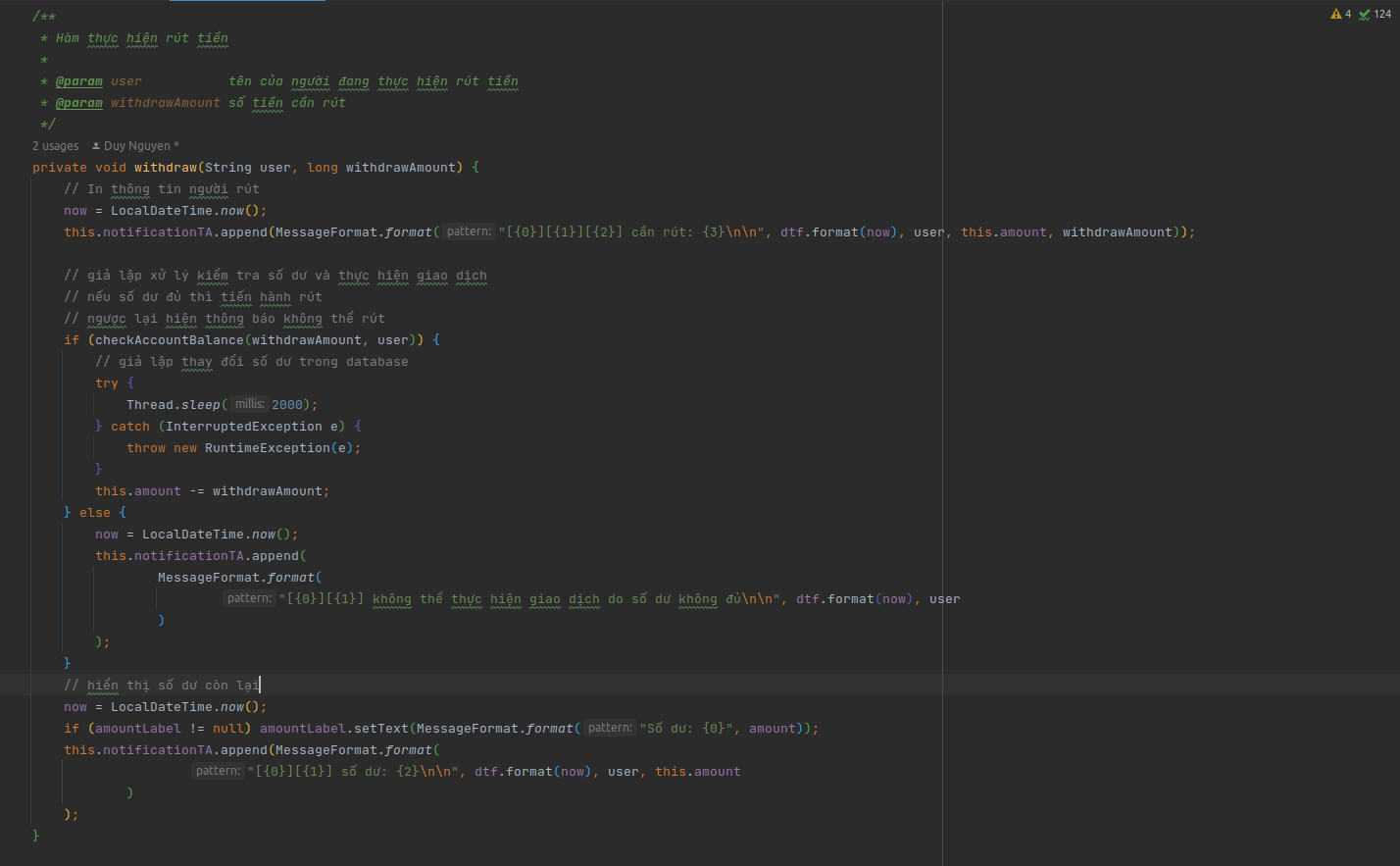
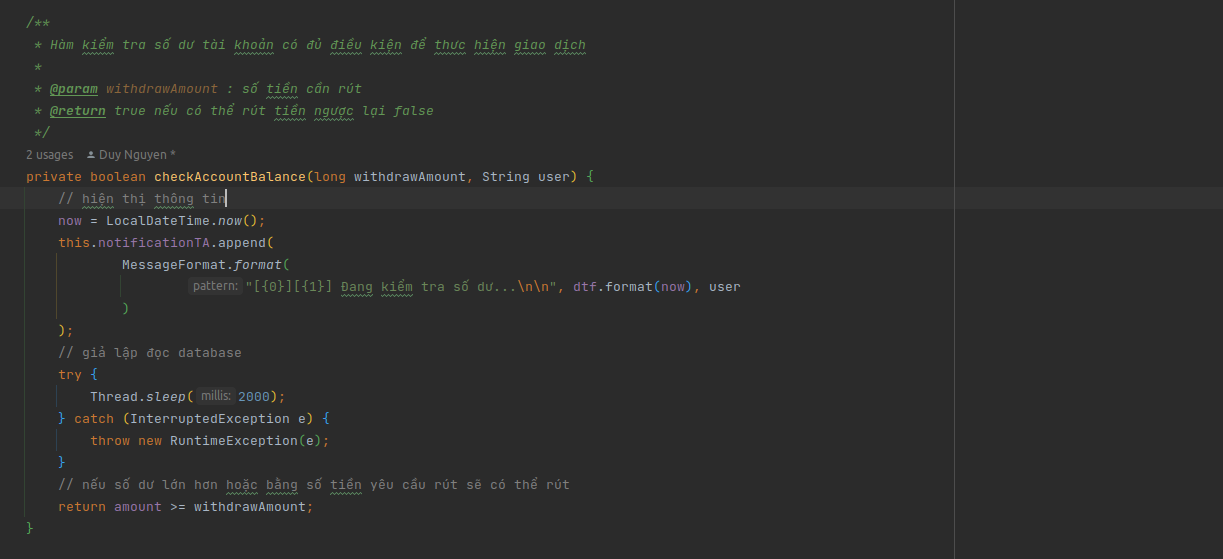
**4.2 Lớp BankAccount và các phương thức giao dịch**



Lớp BankAccount chịu trách nhiệm mô phỏng các tài khoản ngân hàng và định nghĩa các phương thức giao dịch, các kiểu dữ liệu trong lớp BankAccount được định nghĩa bao gồm:



Constructor của lớp BankAccount

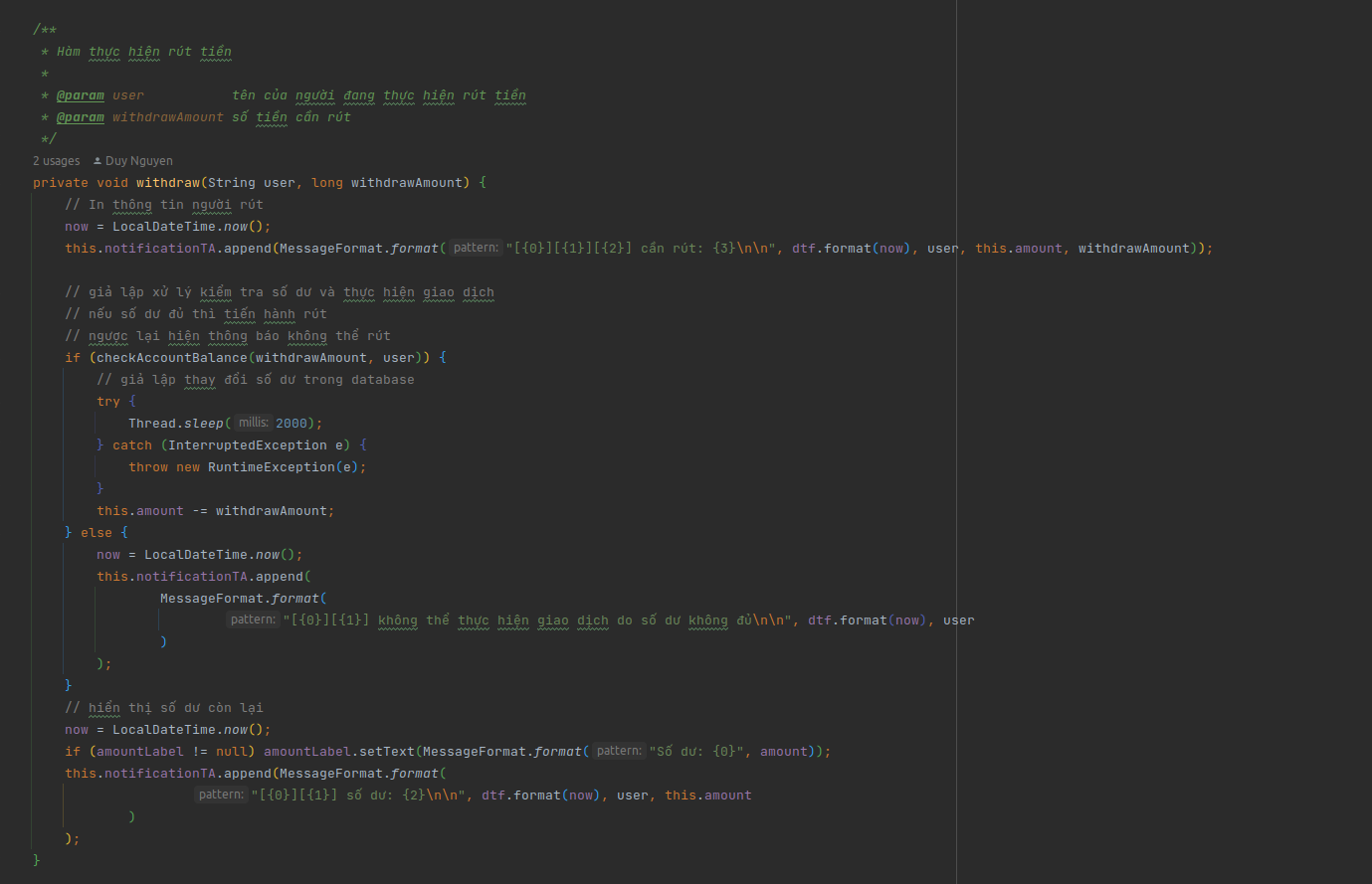


**5. Thực hiện chương trình và mô tả các vấn đề**

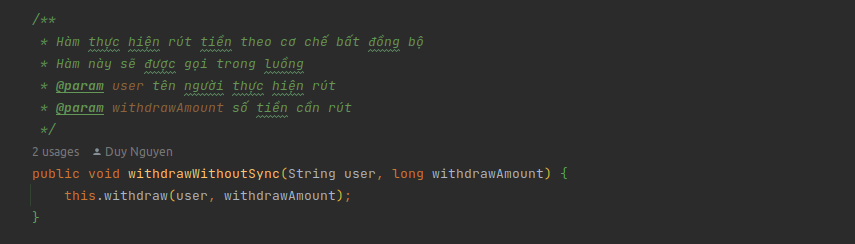
**5.1 Bất đồng bộ:**

Khi các luồng thực hiện đồng thời và cùng sử dụng tài nguyên chung, nếu không được đồng bộ, vấn đề tranh chấp tài nguyên diễn ra, dẫn đến nhiều luồng cùng thực hiện thao tác lên đối tượng này cùng một lúc dẫn đến sai lệch giá trị của đối tượng tài nguyên dùng chung.

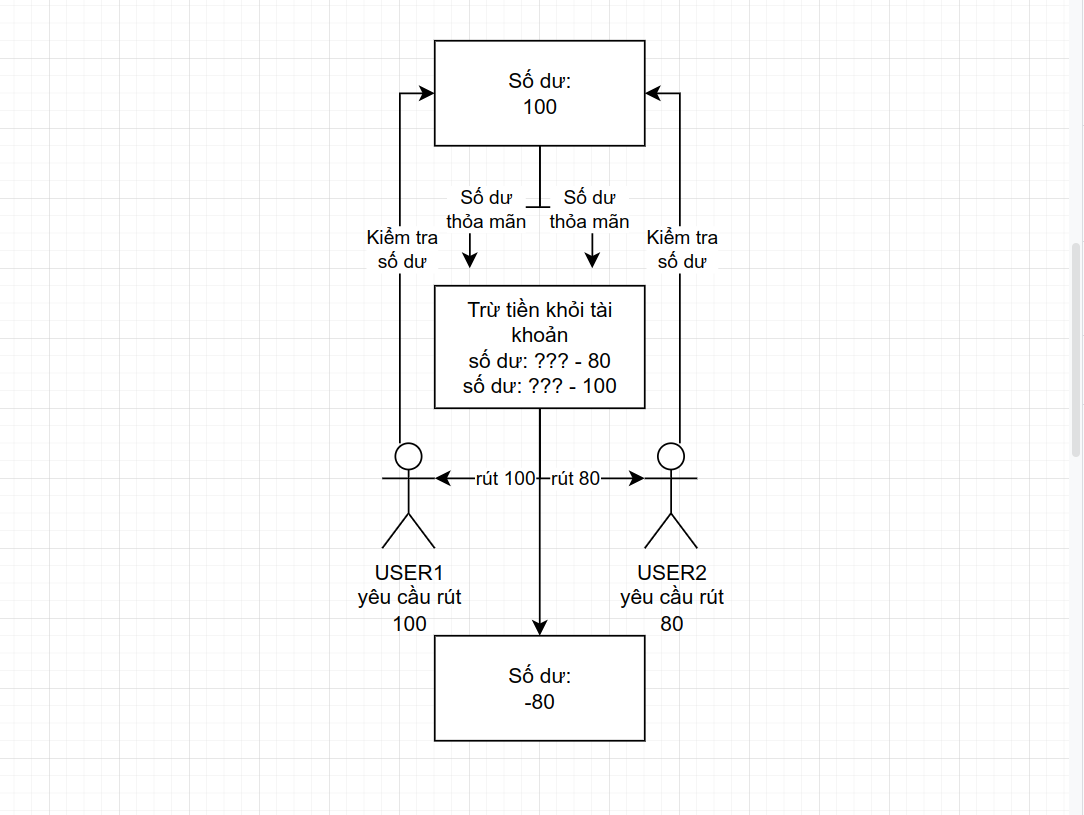
Dẫn chứng ở việc rút tiền ở chung một tài khoản ngân hàng. Khi 2 người dùng cùng thực hiện rút tiền trong cùng một tài khoản ngân hàng cùng một thời điểm, nếu hàm rút tiền không được đồng bộ, hai luồng yêu cầu rút tiền sẽ cùng kiểm tra số dư ở cùng một thời điểm, lúc này việc tranh chấp tài nguyên giữa hai luồng diễn ra, việc kiểm tra số dư bị sai lệch, dẫn tới cả hai người dùng đều có thể rút tiền, mặc dù số dư thực tế không để cho cả hai người cùng rút.



Hàm withdraw thực hiện kiểm tra số dư, nếu đủ điều kiện sẽ thực hiện rút tiền.



Với bất đồng bộ, cả 2 luồng đều có thể gọi đến withdraw cùng một lúc.



Thực hiện trên chương trình mô phỏng:



Nhập giá trị khởi tạo số dư là: 10.000.000



Nhập số tiền USER1 cần rút: 8.000.000

Nhập số tiền USER2 cần rút: 10.000.000



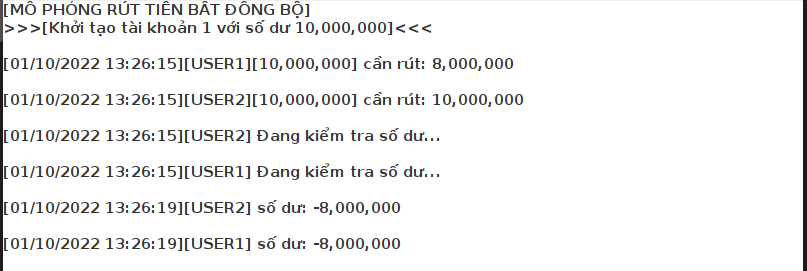
Thực hiện mô phỏng cả hai người dùng cùng rút tiền bằng cách nhấn vào nút Bất đồng bộ.



Lúc này hệ thống sẽ khởi tạo số dư là: 10.000.000



Sau khi cả 2 luồng thực hiện xong việc rút tiền, số dư còn lại sẽ là: -8.000.000



Các trạng thái đã xảy ra trong quá trình rút tiền.

Ở thời điểm [01/10/2022 13:26:15] cả 2 người dùng đều thực hiện yêu cầu rút tiền dẫn đến tại cùng thời điểm đó, cả hai luồng đều thực hiện kiểm tra số dư trong tài khoản.

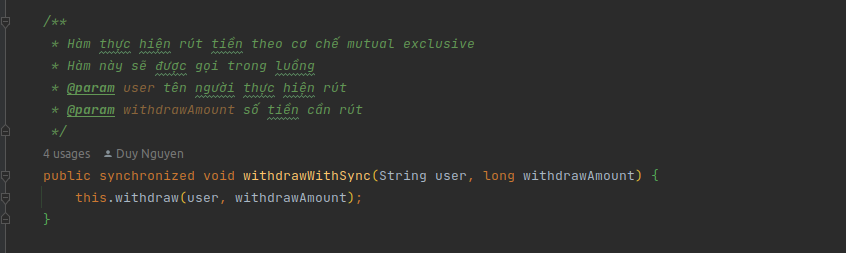
Ở thời điểm [01/10/2022 13:26:19], vùng trạng thái không hiển thị bất cứ thông báo nào khác ngoài thông báo số dư cho cả 2 người dùng, tức là cả 2 người dùng đều rút tiền thành công. Số dư được hiển thị là

-8.000.000, tức là ngân hàng đã bị lỗ 8.000.000.

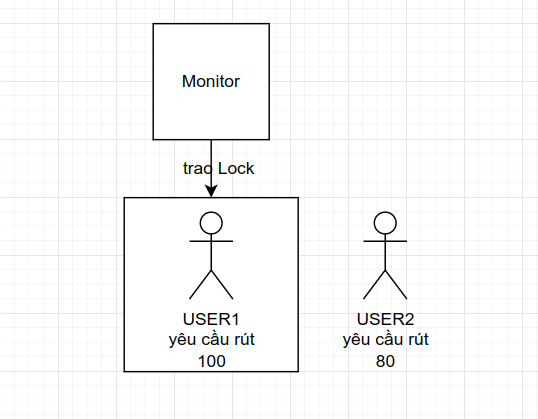
**5.2 Giải quyết vấn đề bằng đồng bộ bằng Mutual Exclusion:**

Đồng bộ Mutual Exclusion là phương pháp đồng bộ luồng giúp cho các luồng không đồng thời can thiệp vào tài nguyên dùng chung bằng cơ chế Monitor & Lock đã được để cập ở các phần trên.

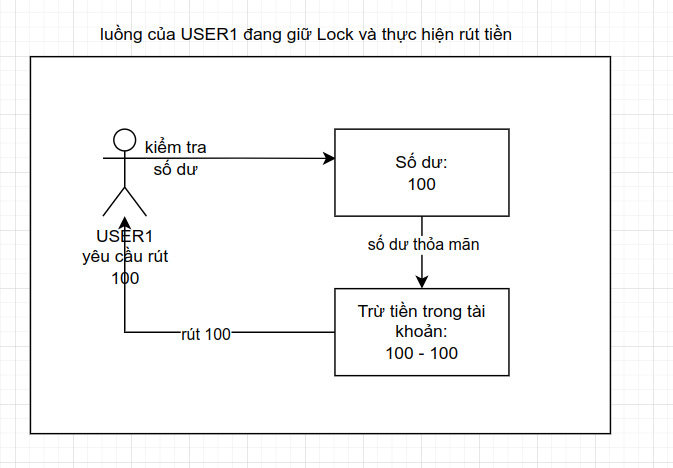
Để giải quyết bài toán bất đồng bộ ở vấn đề được nêu ra ở việc bất đồng bộ ở trên, ta sử dụng từ khóa “synchronized”, việc khai báo “synchronized” giúp cho tài nguyên dùng chung được bảo hộ bởi Monitor, các luồng muốn sử dụng tới các tài nguyên dùng chung được bảo hộ phải được Minitor trao Lock.



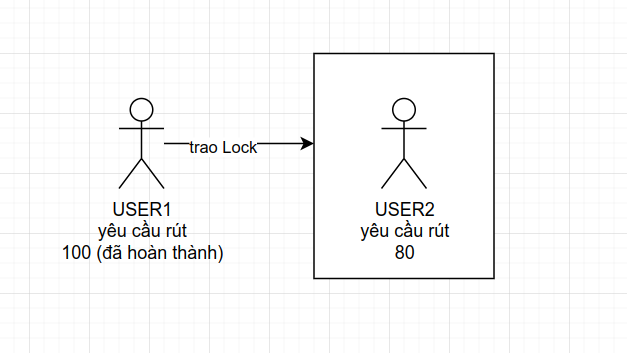
Việc khai báo synchronized cho hàm withdrawWithSync giúp cho tất cả những gì bên trong hàm được bảo hộ bởi Monitor (ở đây hàm withdraw sẽ được bảo hộ).



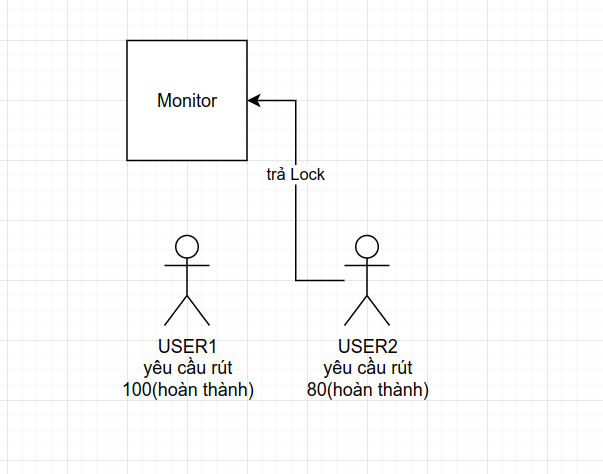
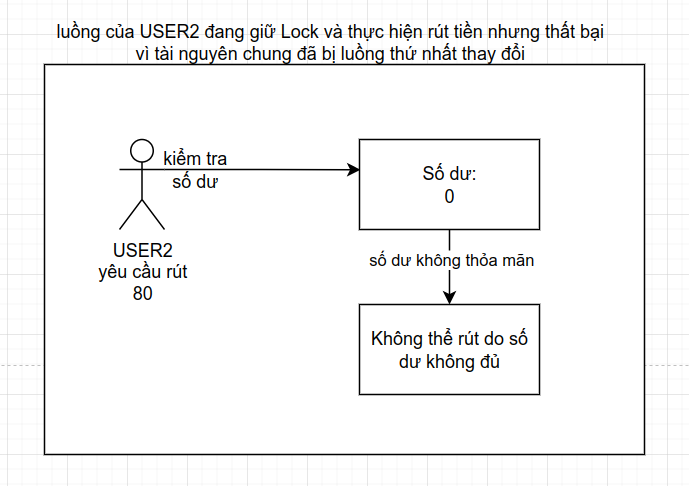
Monitor trao Lock cho luồng đầu tiên.



Sau khi được nắm giữ Lock, luồng đầu tiên sẽ thực hiện rút tiền. Và chỉ mình nó mới có thể truy cập tài nguyên chung trong thời điểm này.



Sau khi luồng thứ nhất hoàn thành công việc của mình, nó sẽ trao lại Lock cho luồng tiếp theo đang nằm trong hàng đợi của Minitor.



Sau khi luồng cuối cùng trong hàng đợi của Minitor hoàn thành công việc của nó, nó sẽ trả Lock lại cho Monitor.

Thực hiện mô phỏng bằng chương trình:



Nhập giá trị khởi tạo số dư là: 10.000.000



Nhập số tiền USER1 cần rút: 8.000.000

Nhập số tiền USER2 cần rút: 10.000.000



Thực hiện mô phỏng cả hai người dùng cùng rút tiền bằng cách nhấn vào nút Mutual Exclusion



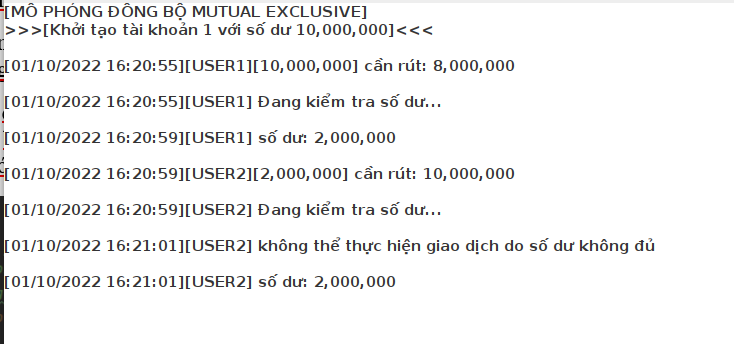
Lúc này hệ thống sẽ khởi tạo số dư là: 10.000.000



Sau khi luồng của USER1 thực hiện rút tiền số dư còn lại là: 2.000.000



Sau khi luồng của USER2 thực hiện rút tiền số dư còn lại là: 2.000.000 do số dư không đáp ứng yêu cầu để rút.



Các trạng thái đã xảy ra trong quá trình rút tiền.

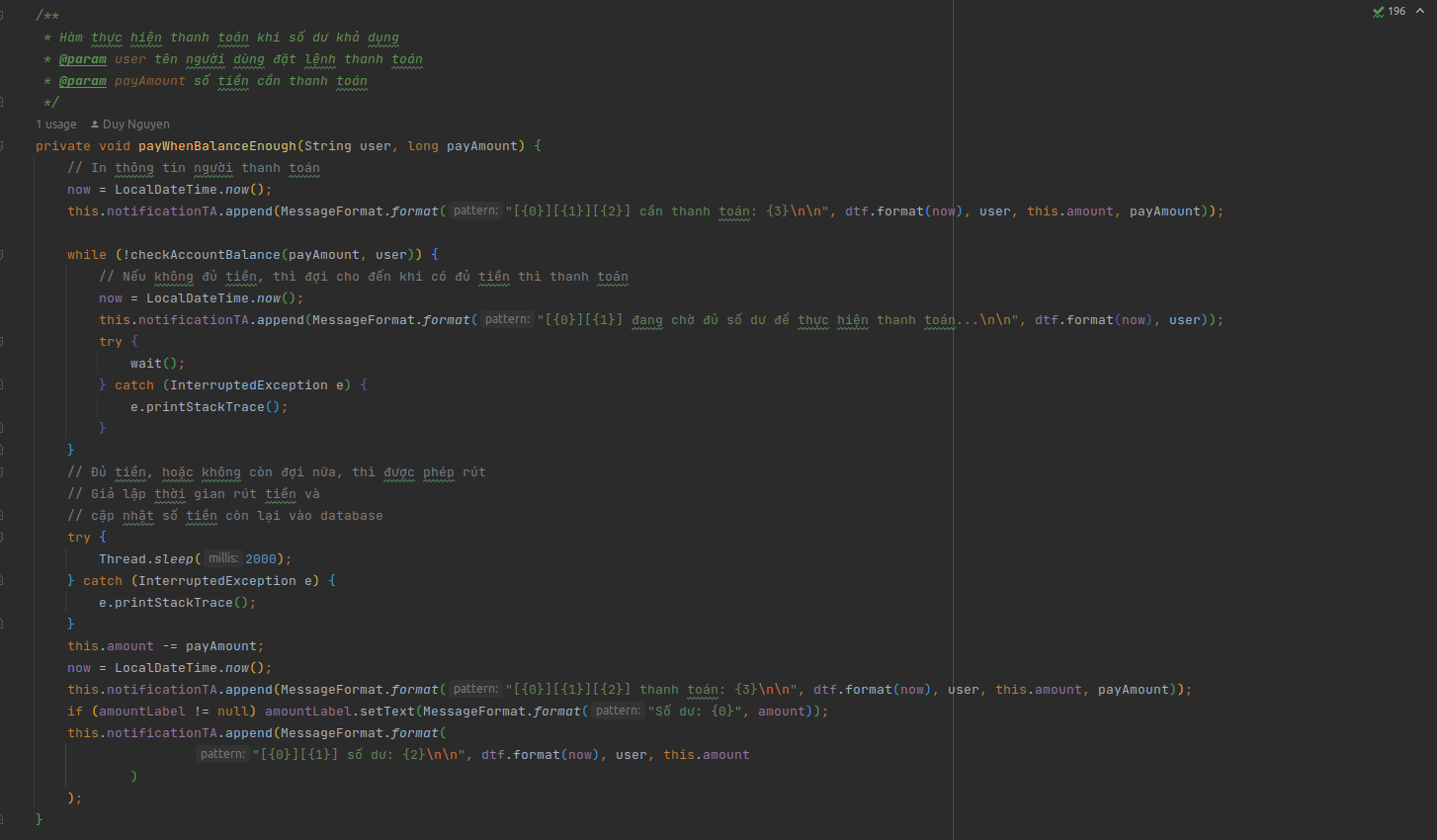
Dù cả hai luồng cùng vào một lúc nhưng tại thời điểm [01/10/2022 16:20:55] chỉ có luồng của USER1 là thực hiện việc kiểm tra số dư do lúc này luồng của USER1 đang nắm giữ Lock.

Tại thời điểm [01/10/2022 16:20:59] sau khi hiển thị số dư (đã có sự thay đổi) tức là luồng của USER1 đã kết thúc cũng là lúc nó trao lại Lock cho luồng của USER2 để luồng của USER2 thực hiện công việc của nó. Tuy nhiên lúc này số dư là 2.000.000 nhưng USER2 cần rút 10.000.000 nên hệ thống không thể thực hiện yêu cầu cảu USER2 nên nó trả lại Lock cho Minitor và kết thúc phiên làm việc của nó.

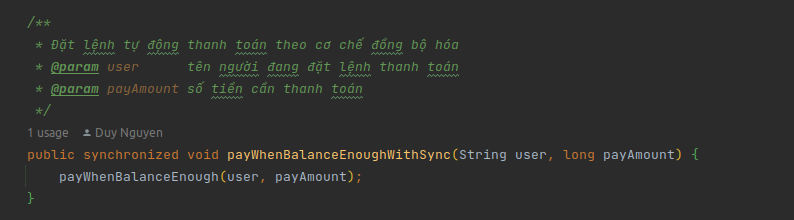
**5.3 Giải quyết vấn đề đồng bộ bằng Cooperation:**

Đồng bộ Cooperation, hay có một số tài liệu gọi là Inter-Thread Communication, mục đích của phương pháp đồng bộ này không ngoài mong muốn tránh các xung đột từ các Thread khi chúng đồng thời cùng sử dụng đến cùng một đối tượng, hay tài nguyên của hệ thống. Khác với phương pháp đồng bộ Mutual Exclusive, phương pháp Mutual Exclusive tạo ra một cơ chế loại trừ, giúp các Thread nào dùng đến tài nguyên trước sẽ được ưu tiên, Thread dùng sau sẽ bị loại trừ và phải đợi. Phương pháp Cooperation có sự “cộng tác” với nhau, cộng tác theo một tinh thần mà một Thread có thể hoàn toàn “nhường” cho Thread khác sử dụng đến tài nguyên mà nó đã “giành” trước, để rồi khi Thread nào đó khác sử dụng xong tài nguyên đó, Thread đó phải “đánh thức” nó dậy để nó tiếp tục công việc trên tài nguyên đó.

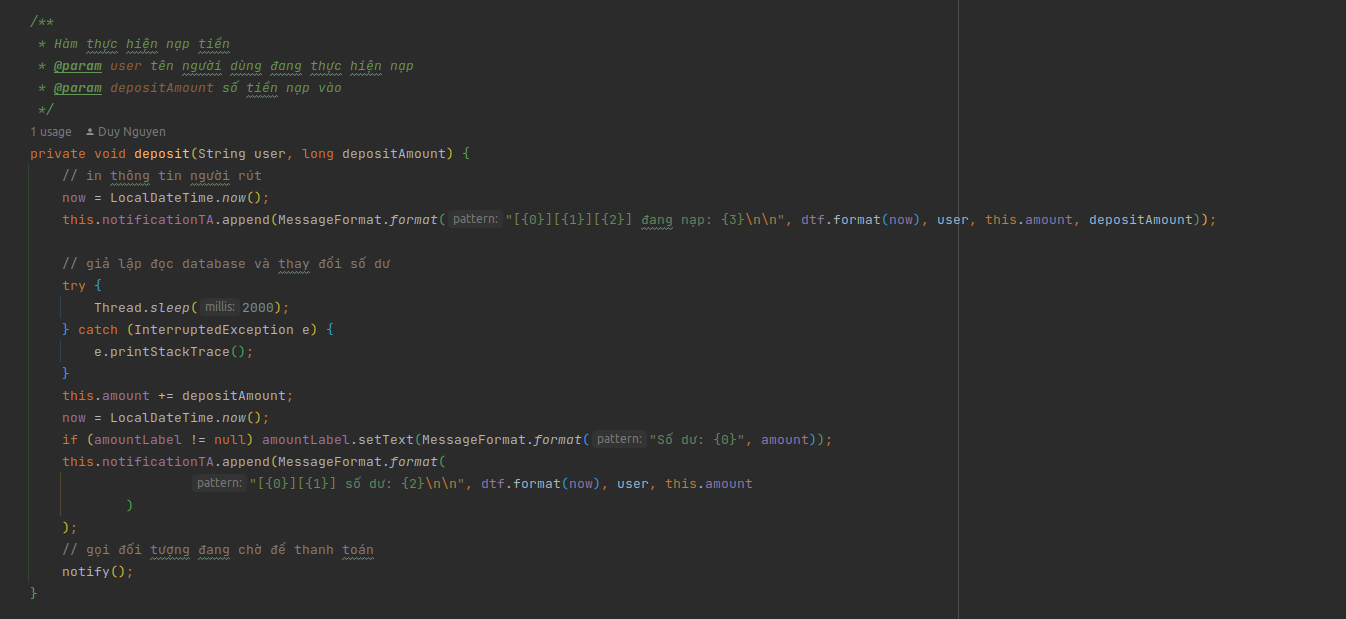
Dẫn chứng ở việc nạp tiền và đặt lệnh thanh toán. Khi một người đặt lệnh thanh toán một hóa đơn, hóa đơn được thanh toán khi và chỉ khi số dư trong tài khoản hiện tại đủ hoặc số dư thay đổi thông qua hình thực nạp tiền hoặc chuyển tiền khiến số dư đủ điều kiện để thanh toán thì công việc đặt lệnh kết thúc.



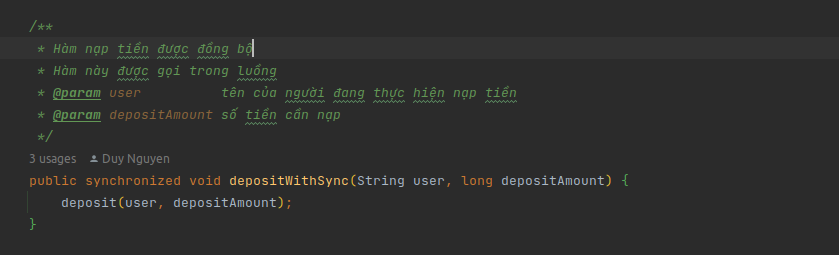
Phương thức này chỉ kết thúc khi nó đủ khả năng để thanh toán số tiền mà nó đã đặt lệnh. Nếu không đủ nó sẽ rơi vào "trạng thái ngủ" thông qua lệnh wait() để tạm thời trao Lock lại cho các luồng khác thực hiện để đáp ứng được "nhu cầu" của nó. Phương thức chỉ có thể thoát khỏi trạng thái ngủ khi một luồng khác “gọi nó dậy”. Vòng lặp “ngủ” và "thức dậy" chỉ kết thúc khi số dư đủ điều kiện thanh toán.



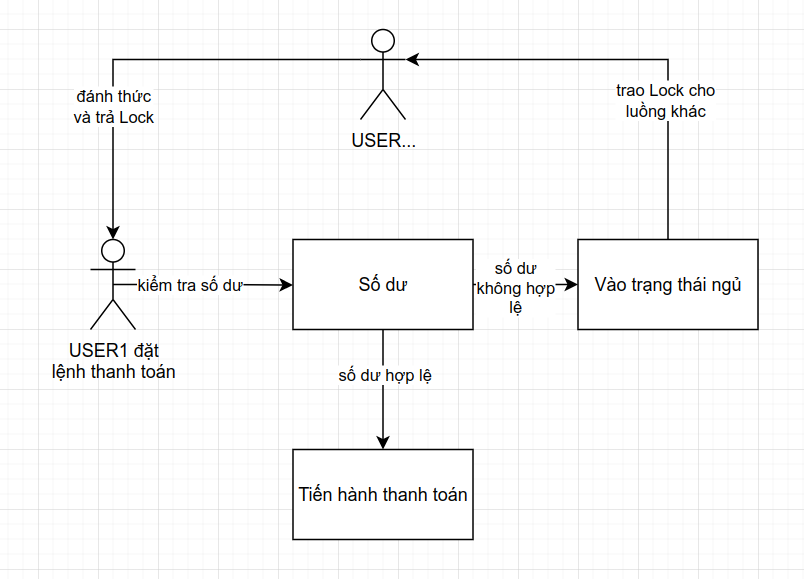
Phương thức đặt lệnh cũng sẽ được đồng bộ hóa để ngăn sự xung đột giữa các luồng đều cùng gọi đến nó.



Phương thức nạp tiền vào tài khoản và đánh thức các luồng “đang ngủ” để trả lại Lock cho nó tiếp tục thực hiện công việc của mình.



Phương thức nạp tiền cũng sẽ được đồng bộ hóa để ngăn sự xung đột giữa các luồng đều cùng gọi đến nó.



Thực hiện mô phỏng bằng chương trình:

Tình huống các luồng đều đáp ứng được yêu cầu của luồng đặt lệnh thanh toán:



Nhập giá trị khởi tạo số dư là: 10.000.000



Nhập số tiền USER1 cần thanh toán: 12.000.000



Nhập số tiền USER2 nạp: 5.000.000



Nhấn nút Cooperation để tiến hành mô phỏng



Lúc này hệ thống sẽ khởi tạo số dư là: 10.000.000

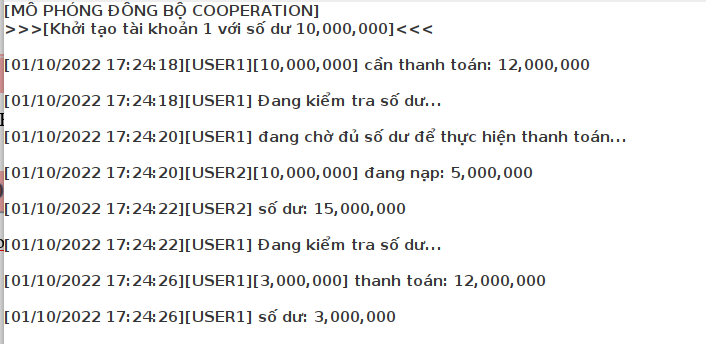
Sau khi luồng đặt lệnh thanh toán của USER1 vào kiểm tra thì nhận thấy số dư không đủ nên vào trạng thái ngủ.



Sau khi luồng nạp tiền của USER2 thực hiện xong số dư có sự thay đổi, nó tiến hành đánh thức luồng đang trong trạng thái ngủ.



Sau khi luồng đặt lệnh thanh toán của USER1 được đánh thức và vào kiểm tra thì nhận thấy số dư đã đủ nên tiến hành thanh toán và số dư thay đổi thành 3.000.000



Thời điểm [01/10/2022 17:24:18], luồng của USER1 thực hiện đặt lệnh thanh toán và tiến hành kiểm tra số dư.

Thời điểm [01/10/2022 17:24:20], vì không thể thực hiện thanh toán, luồng của USER1 rơi vào trạng thái ngủ và tiến hành trao Lock cho luồng của USER2 thực hiện.

Thời điểm [01/10/2022 17:24:22], sau khi nạp tiền vào tài khoản thành công luồng USER2 đánh thức luồng của USER1 để tiến hành kiểm tra số dư.

Thời điểm [01/10/2022 17:24:26] sau khi nhận thấy số dư đã đủ luồng USER1 tiến hành thanh toán và kết thúc phiên làm việc của mình.

Tình huống các luồng không đáp ứng được yêu cầu của luồng đặt lệnh thanh toán:

Tình huống các luồng đều đáp ứng được yêu cầu của luồng đặt lệnh thanh toán:



Nhập giá trị khởi tạo số dư là: 10.000.000



Nhập số tiền USER1 cần thanh toán: 100.000.000



Nhập số tiền USER2 nạp: 5.000.000



Nhấn nút Cooperation để tiến hành mô phỏng



Lúc này hệ thống sẽ khởi tạo số dư là: 10.000.000

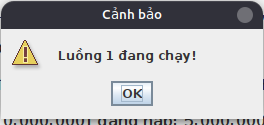
Sau khi luồng đặt lệnh thanh toán của USER1 vào kiểm tra thì nhận thấy số dư không đủ nên vào trạng thái ngủ.



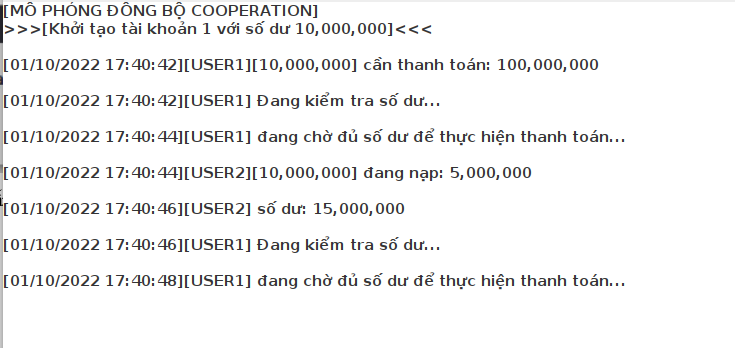
Sau khi luồng nạp tiền của USER2 thực hiện xong số dư có sự thay đổi, nó tiến hành đánh thức luồng đang trong trạng thái ngủ.



Sau khi luồng của USER1 thức dậy nó tiến hành kiểm tra số dư và nó nhận thấy số dư vẫn không đủ và nó tiếp tục rơi vào trạng thái ngủ. Vì công việc của luồng USER1 vẫn chưa kết thúc nên nó vẫn chưa thể vào trạng thái TERMINATED, để kiểm chứng ta nhấn vào nút Kiểm tra luồng và nhận được thông báo như bên dưới.



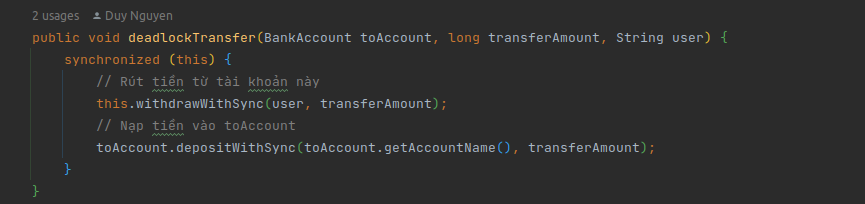
Ta nhận thấy rằng khi số dư vẫn chưa đủ và không còn luồng nào khác chạy, luồng USER1 sẽ luôn trong trạng thái ngủ cho đến khi nào số dư đủ hoặc một luồng khác đánh thức dậy.



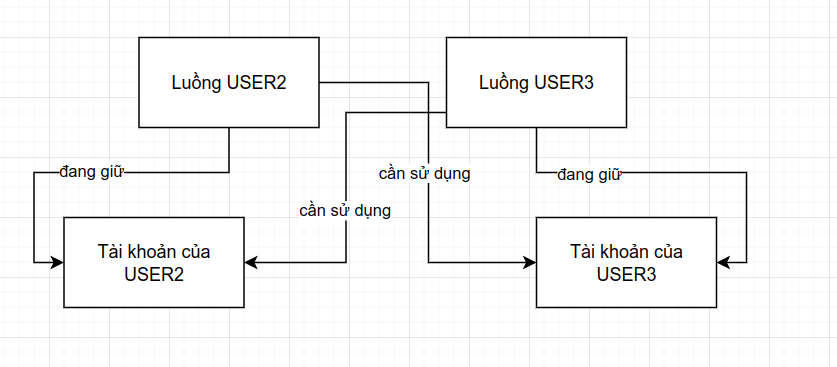
**5.4 Deadlock:**

Deadlock là tình huống xảy ra khi một luồng đang chờ đợi nhận Lock đang được nắm giữ bởi một luồng khác và luồng đó cũng đang chờ đợi nhận Lock đang được nắm giữ của luồng còn lại.

Trong vấn đề chuyển tiền, việc thực hiện sẽ bao gồm thao tác rút tiền (withdraw) khỏi tài khoản A và sau đó sẽ nạp vào tài khoản B (deposit). Nhưng trong một vấn đề nào đó, do quá “cẩn thận” trong việc xây dựng các phương thức đồng bộ, đã vô tình khiến cho việc dẫn đến Deadlock xảy ra trong trường hợp cả 2 tài khoản khác nhau đều muốn chuyển tiền cho nhau.



Việc đồng bộ lớp BankAccount khiến cho sau khi luồng sử dụng đến nó cứ nắm giữ Lock không chịu nhả ra cho luồng còn lại và luồng còn lại cũng tương tự do cả 2 luồng đều chưa kết thúc (đồng bộ khiến cho luồng chỉ nhả Lock khi luồng đó kết thúc).



Thực hiện mô phỏng bằng chương trình:



Nhập giá trị khởi tạo số dư là: 10.000.000



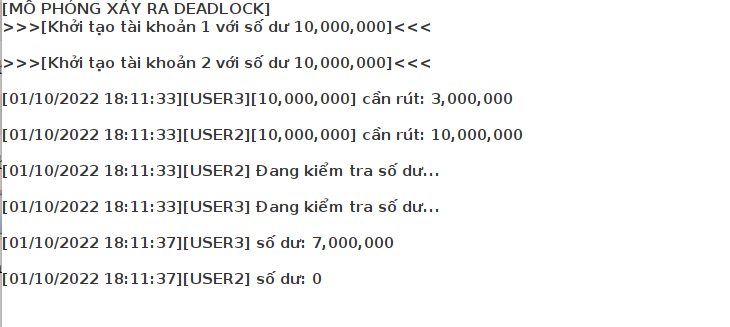
Nhập số tiền USER3 chuyển cho USER2 là 3.000.000



Nhập số tiền USER2 chuyển cho USER3 là 10.000.000



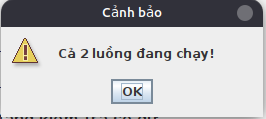
Nhấn nút Deadlock để tiến hành mô phỏng.



Kết quả của quá trình mô phỏng

Thời điểm [01/10/2022 18:11:33] cả 2 người dùng sẽ tiến hành việc chuyển tiền. Đầu tiên họ phải rút tiền của tài khoản của chính mình ra. Vì đây là 2 tài khoản được khởi tạo độc lập nên chắc chắn sẽ không có sự xung đột nào xảy ra.

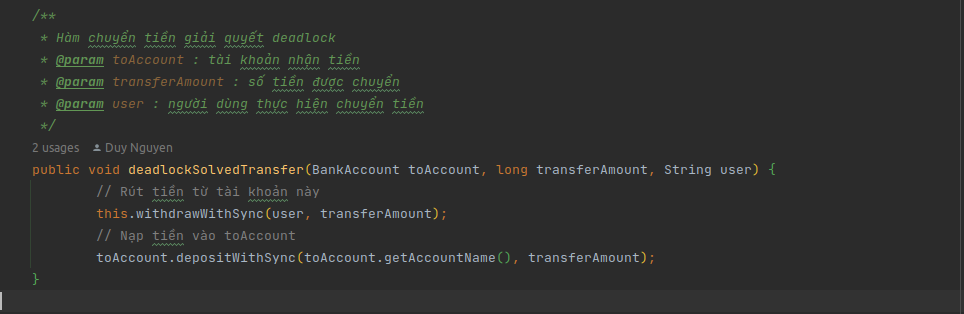
Thời điểm [01/10/2022 18:11:37] cả 2 người dùng đều thành công rút tiền ra khỏi tài khoản của họ. Nhưng khi này họ không thể tiến thành nạp tiền vào tài khoản của đối phương do hai luồng không chịu nhả Lock. Để kiểm chứng ta nhấn nút Kiểm tra luồng.



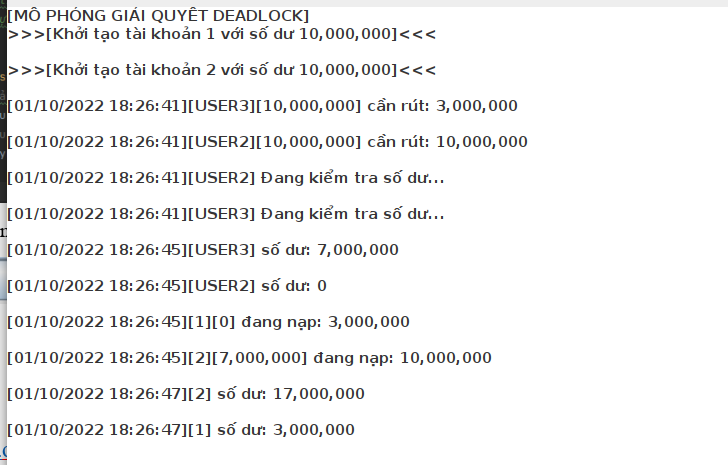
Ta nhận thấy cả hai luồng đều đang chạy mà không có bất kì sự kiện nào diễn ra tiếp theo tức là cả 2 đang trong trạng thái Deadlock.

**5.5 Giải quyết Deadlock:**

Để giải quyết Deadlock cho vấn đề được nếu trên, ta chỉ việc loại bỏ đồng bộ giữa các tài nguyên không thể nào có sự xảy ra tranh chấp.



Thực hiện nhập các giá trị tương tự ở phần Deadlock ở trên và nhấn nút Giải quyết DL:



Kết quả sau khi giải quyết Deadlock

Vào thời điểm [01/10/2022 18:26:45] khi cả 2 người dùng đều rút tiền ra khỏi tài khoản của mình thành công, luồng bắt đầu tiến hành nạp só tiền mà 2 người dùng chuyển cho lẫn nhau vẫn tài khoản tương ứng.

Thời điểm [01/10/2022 18:26:47] khi giao dịch hoàn tất, số dư của [1](tài khoản của USER2) là 3.000.00. Do USER2 đã chuyển cho USER3 10.000.000 và nhận lại từ USER3 3.000.000 về phía của USER3 thì ngược lại và cơ chế cũng tương tự.

**6. Tài liệu tham khảo**

<https://www.geeksforgeeks.org/banking-transaction-system-using-java/>

<https://www.javatpoint.com/life-cycle-of-a-thread>

<https://www.scientecheasy.com/2020/08/deadlock-in-java.html/>

<https://www.geeksforgeeks.org/g-fact-70/>

https://www.geeksforgeeks.org/mutual-exclusion-in-synchronization/

https://www.scientecheasy.com/2020/08/deadlock-in-java.html/

[https://www.javatpoint.com/inter-thread-communication-example#:~:text=Cooperation%20(Inter%2Dthread%20communication),wait](https://www.javatpoint.com/inter-thread-communication-example" \l ":~:text=Cooperation (Inter-thread communication),wait)()