Academic Year	Module	Task
2025	5CS022/HJ1: Distributed and Cloud Systems Programming	3

Name: Swoyam Pokharel
ID: 2431342
Group: 26
Tutor: Ms. Jenny Rajak

Table Of Contents

Question: **3**

Solution: **4**

- Understanding The Question: 4
- Preparation and Initial Setup: 4
  - Akka Initialization & File Setup: 4
    - Screenshots Showing The Directory Structure 4
    - Screenshot of the `build.sbt` file from the provided template 5
    - Adding all .jar files to path 6
    - Screenshot Of files with minimal code 7
    - Screenshot of the output. 7
- Final Code: 8
  - Deposit: 8
    - Screenshot of deposit class 8
  - Withdrawal: 9
    - Screenshot of the withdrawal class 9
- BankAccount: 9
    - Screenshot of the BankAccount class 10
- Main: 11
    - Screenshot for Main class with Proof Of Concept Code 11
    - Screenshot of the output 12
    - Screenshot of the updated main class 13
    - Screenshot of the output of the updated main class 14
    - Screenshot of main with specified seed 14
    - Screenshot showing the effect of seeding piped through bat for readability 15

Question:

1. Using the Akka Actor framework create a simulation of a bank account with multiple concurrent deposits and withdrawals, over a specified number of transactions.

You should do the following:

- Create a BankAccount class as the Actor to hold the account balance, and to respond to deposit and withdrawal message.
- Create a Deposit class to serve as the message sent to the BankAccount class to deposit money into the account.
- Create a Withdrawal class to serve as the message sent to the BankAccount class to withdraw money from the account.
- Create a Main class to serve as the simulation running program.
- When the Main class starts running, it should create an Actor of the BankAccount class.
- The BankAccount actor should initialise its balance to £100 on startup and print this balance to the console output.
- The Main program should then create 10 random values between -1000 to 1000.
- If a value greater than zero, the Main program should create a Deposit message with that amount, and send it to the BankAccount actor, which would then add it to its current balance, and then print out the new balance on the console output.
- If a value less than zero, the Main program should create a Withdrawal message with that amount, and send it to the BankAccount actor, which would then subtract it from its current balance, and then print out the new balance on the console output.
- After all, 10 transactions have been processed, the program should terminate.

This part of the task will contribute 30% of the marks to the Portfolio.

Solution:

Understanding The Question:

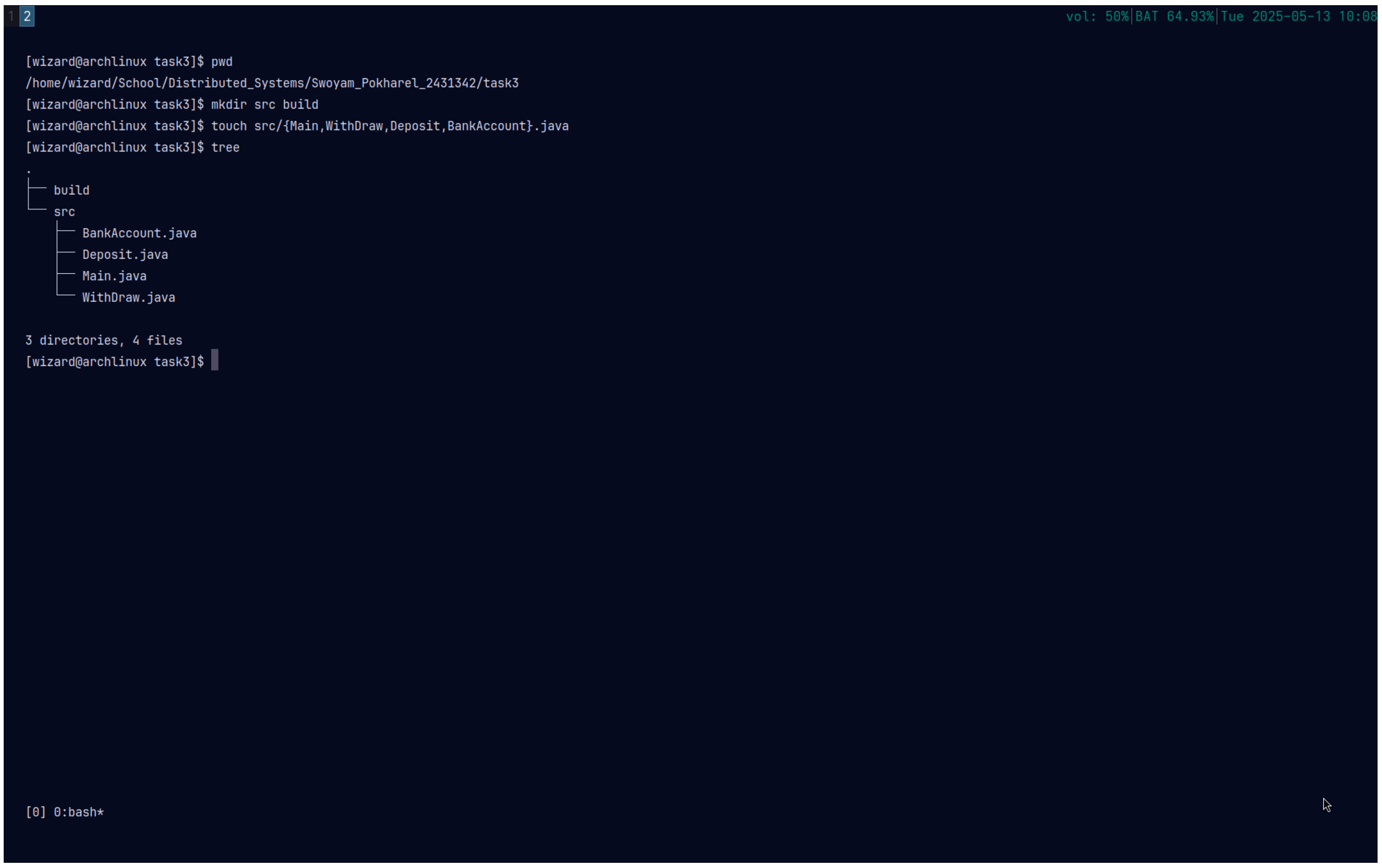
So, the question wants an Akka Actor program to simulate a bank account with multiple concurrent deposits and withdrawals, over a specified number of transactions. It wants:

- A `BankAccount` class as the Actor that holds the account balance, and respond to deposits and withdrawals. So this is where the main chunk of the logic for our program will reside in.
  - The `BankAccount` should be initialized with `$100`
  - It shouldn't withdraw with insufficient funds.
- A `Deposit` class, that serves as the message for `BankAccount` to signal it to deposit `x` amount of money.
- A `Withdrawal` class to serve as the message for `BankAccount` to signal it to withdraw `x` amount of money.
- The `Main` class should initialize the Actor System, and generate random values between `-1000` to `1000`.
  - If the randomly generated number is negative, the `Main` class should send a withdrawal request to the `BankAccount` class.
  - If the generated number is positive, the `Main` class should send a deposit request to the `BankAccount` class.
- The program should terminate after 10 transactions have been processed.

Preparation and Initial Setup:

Akka Initialization & File Setup:

Since we only want 4 files, the file setup is going to be pretty simple:



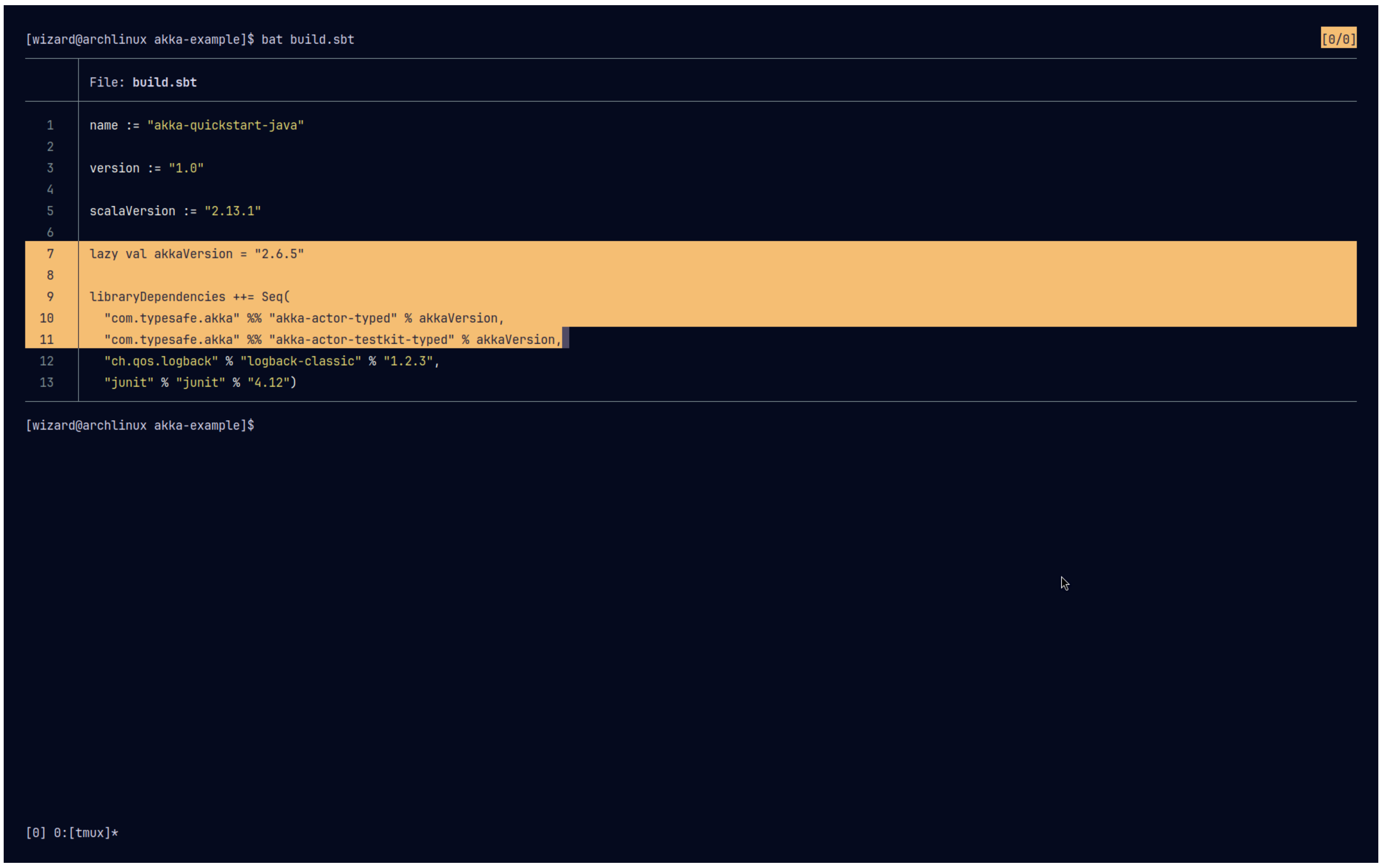
```
1 2                                                              vol: 50%|BAT 64.93%|Tue 2025-05-13 10:08

[wizard@archlinux task3]$ pwd
/home/wizard/School/Distributed_Systems/Swoyam_Pokharel_2431342/task3
[wizard@archlinux task3]$ mkdir src build
[wizard@archlinux task3]$ touch src/{Main,WithDraw,Deposit,BankAccount}.java
[wizard@archlinux task3]$ tree
.
├── build
└── src
    ├── BankAccount.java
    ├── Deposit.java
    ├── Main.java
    └── WithDraw.java

3 directories, 4 files
[wizard@archlinux task3]$

[0] 0:bash*
```

Screenshots Showing The Directory Structure

Here, we simply create 2 folders, `build` and `src`. The `build` is to hold the compiled program and the `src` will hold the source code. Inside `src` we create 4 files that are required.

Moving on to installing Akka, I personally have to do a bit of a setup considering that I'm on Linux and I don't use neither intellij nor eclipse, so the provided templates won't come to use for me.

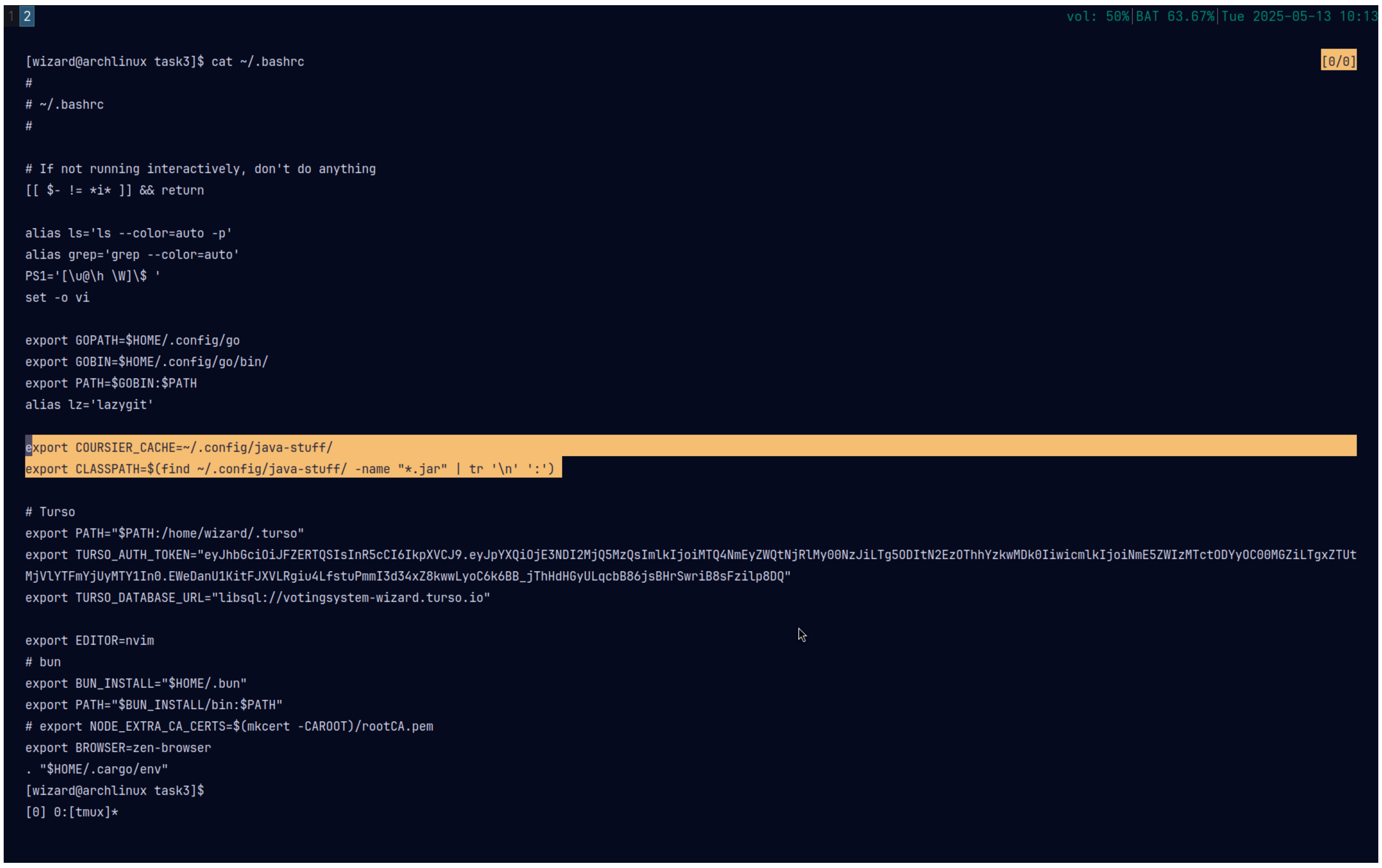
To install Akka, I first needed to know the exact version of Akka that we are supposed to be using, by taking a look inside the `build.sbt` file from the provided templates



```
[wizard@archlinux akka-example]$ bat build.sbt                                  [0/0]
       File: build.sbt

       name := "akka-quickstart-java"

       version := "1.0"

       scalaVersion := "2.13.1"

       lazy val akkaVersion = "2.6.5"

       libraryDependencies ++= Seq(
         "com.typesafe.akka" %% "akka-actor-typed" % akkaVersion,
         "com.typesafe.akka" %% "akka-actor-testkit-typed" % akkaVersion,
         "ch.qos.logback" % "logback-classic" % "1.2.3",
         "junit" % "junit" % "4.12")

[wizard@archlinux akka-example]$

[0] 0:[tmux]*
```

Screenshot of the `build.sbt` file from the provided template

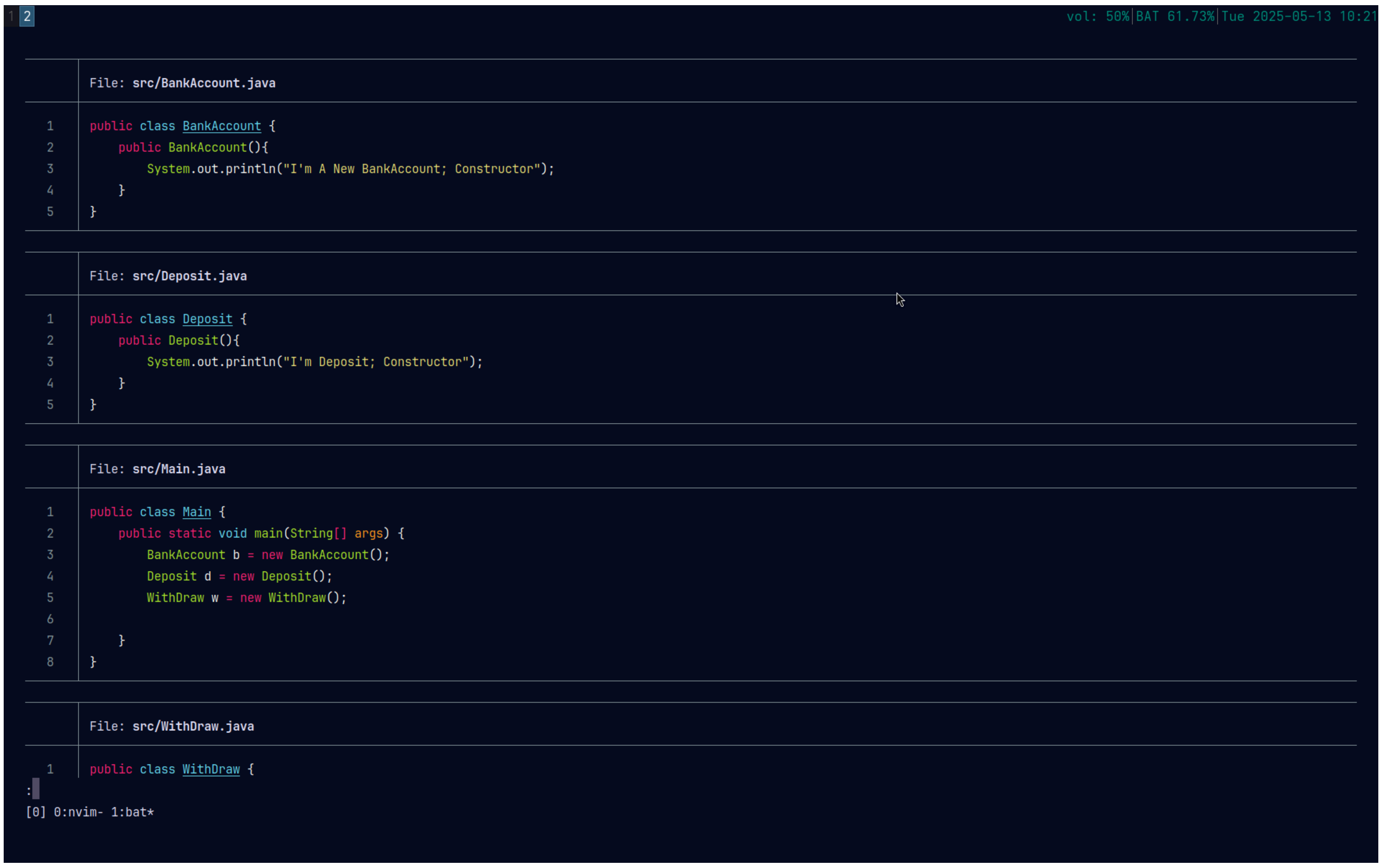
Here, we can see that the version of Akka is `2.6.5`; specifically the typesafe version of Akka, `akka-actor-typed_2.13:2.6.5`. With that information, now I'll install Akka using `coursier`. After which, I've added a bash command to find every jar file inside the directory that `coursier` installs in; and added them to the classpath so that java can directly find those. This makes it convenient for me as now I won't have to manually link `jars`.



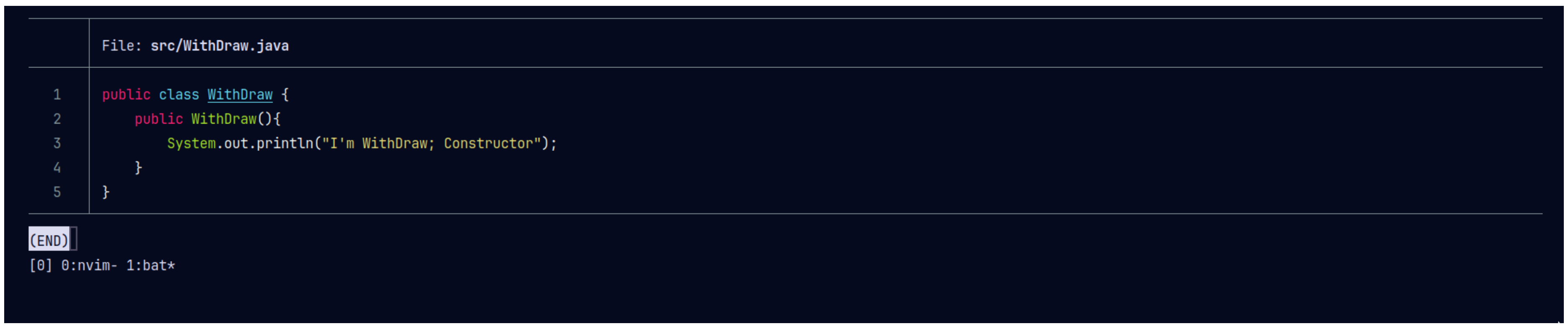
```
1 2                                                      vol: 50%|BAT 63.67%|Tue 2025-05-13 10:13

[wizard@archlinux task3]$ cat ~/.bashrc                                                  [0/0]
#
# ~/.bashrc
#

# If not running interactively, don't do anything
[[ $- != *i* ]] && return

alias ls='ls --color=auto -p'
alias grep='grep --color=auto'
PS1='[\u@\h \W]\$ '
set -o vi

export GOPATH=$HOME/.config/go
export GOBIN=$HOME/.config/go/bin/
export PATH=$GOBIN:$PATH
alias lz='lazygit'

export COURSIER_CACHE=~/.config/java-stuff/
export CLASSPATH=$(find ~/.config/java-stuff/ -name "*.jar" | tr '\n' ':')

# Turso
export PATH="$PATH:/home/wizard/.turso"
export TURSO_AUTH_TOKEN="eyJhbGciOiJFZERTQSIsInR5cCI6IkpXVCJ9.eyJpYXQiOjE3NDI2MjQ5MzQsImlkIjoiMTQ4NmEyZWQtNjRlMy00NzJiLTg5ODItN2EzOThhYzkwMDk0IiwicmlkIjoiNmE5ZWIzMTctODYyOC00MGZiLTgxZTUt
MjVlYTFmYjUyMTY1In0.EWeDanU1KitFJXVLRgiu4LfstuPmmI3d34xZ8kwwLyoC6k6BB_jThHdHGyULqcbB86jsBHrSwriB8sFzilp8DQ"
export TURSO_DATABASE_URL="libsql://votingsystem-wizard.turso.io"

export EDITOR=nvim
# bun
export BUN_INSTALL="$HOME/.bun"
export PATH="$BUN_INSTALL/bin:$PATH"
# export NODE_EXTRA_CA_CERTS=$(mkcert -CAROOT)/rootCA.pem
export BROWSER=zen-browser
. "$HOME/.cargo/env"
[wizard@archlinux task3]$
[0] 0:[tmux]*
```

Adding all .jar files to path

Now that Akka is set up, I'll start adding minimal code to each file; just to lay the foundation and make sure everything compiles as a basic proof of concept.

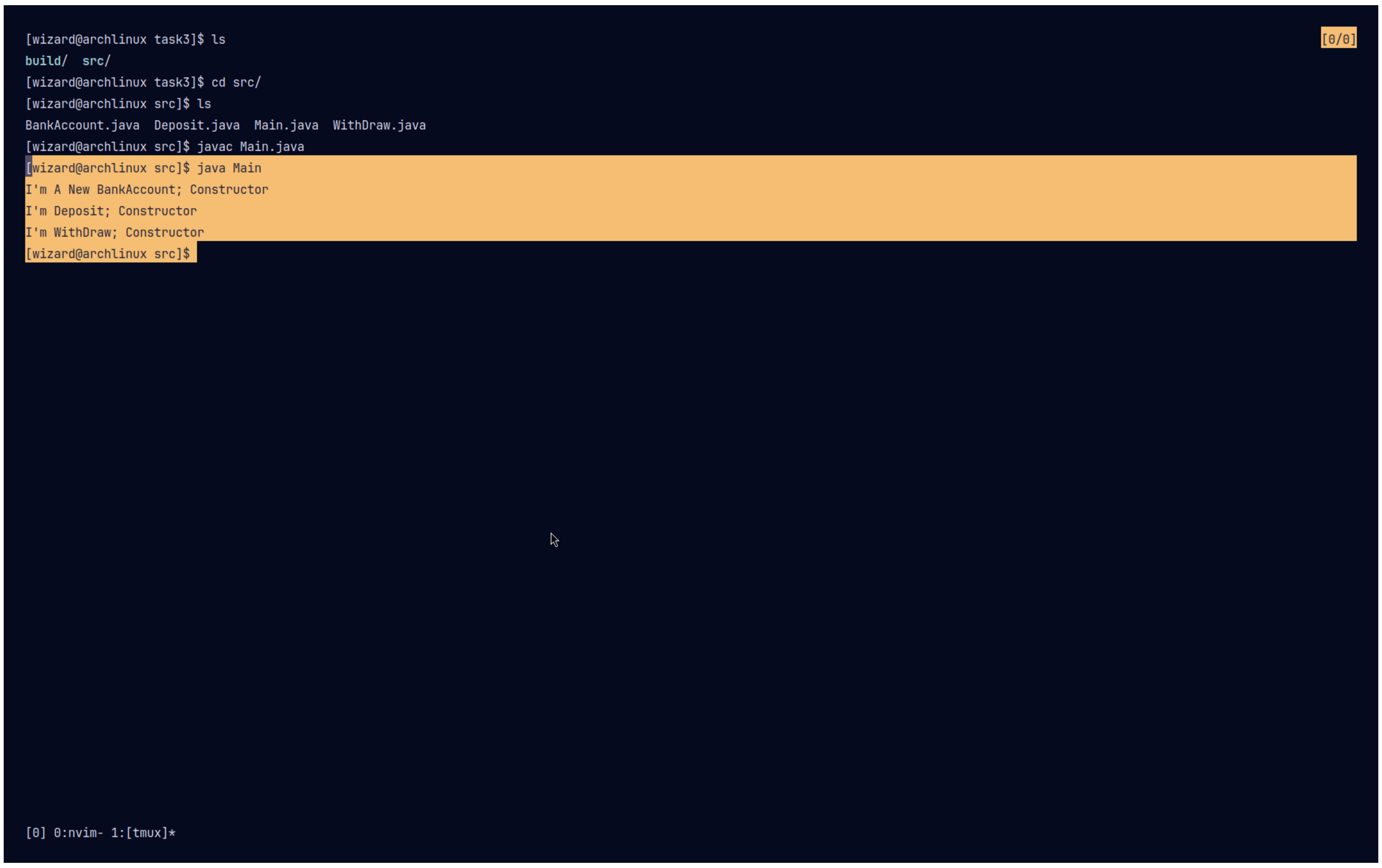


```
1 2                                                      vol: 50%|BAT 61.73%|Tue 2025-05-13 10:21

File: src/BankAccount.java

public class BankAccount {
    public BankAccount(){
        System.out.println("I'm A New BankAccount; Constructor");
    }
}

File: src/Deposit.java

public class Deposit {
    public Deposit(){
        System.out.println("I'm Deposit; Constructor");
    }
}

File: src/Main.java

public class Main {
    public static void main(String[] args) {
        BankAccount b = new BankAccount();
        Deposit d = new Deposit();
        WithDraw w = new WithDraw();

    }
}

File: src/WithDraw.java

public class WithDraw {
:
[0] 0:nvim- 1:bat*
```



```
File: src/WithDraw.java

public class WithDraw {
    public WithDraw(){
        System.out.println("I'm WithDraw; Constructor");
    }
}

(END)
[0] 0:nvim- 1:bat*
```

Screenshot Of files with minimal code



```
[wizard@archlinux task3]$ ls                                                    [0/0]
build/  src/
[wizard@archlinux task3]$ cd src/
[wizard@archlinux src]$ ls
BankAccount.java  Deposit.java  Main.java  WithDraw.java
[wizard@archlinux src]$ javac Main.java
[wizard@archlinux src]$ java Main
I'm A New BankAccount; Constructor
I'm Deposit; Constructor
I'm WithDraw; Constructor
[wizard@archlinux src]$

[0] 0:nvim- 1:[tmux]*
```

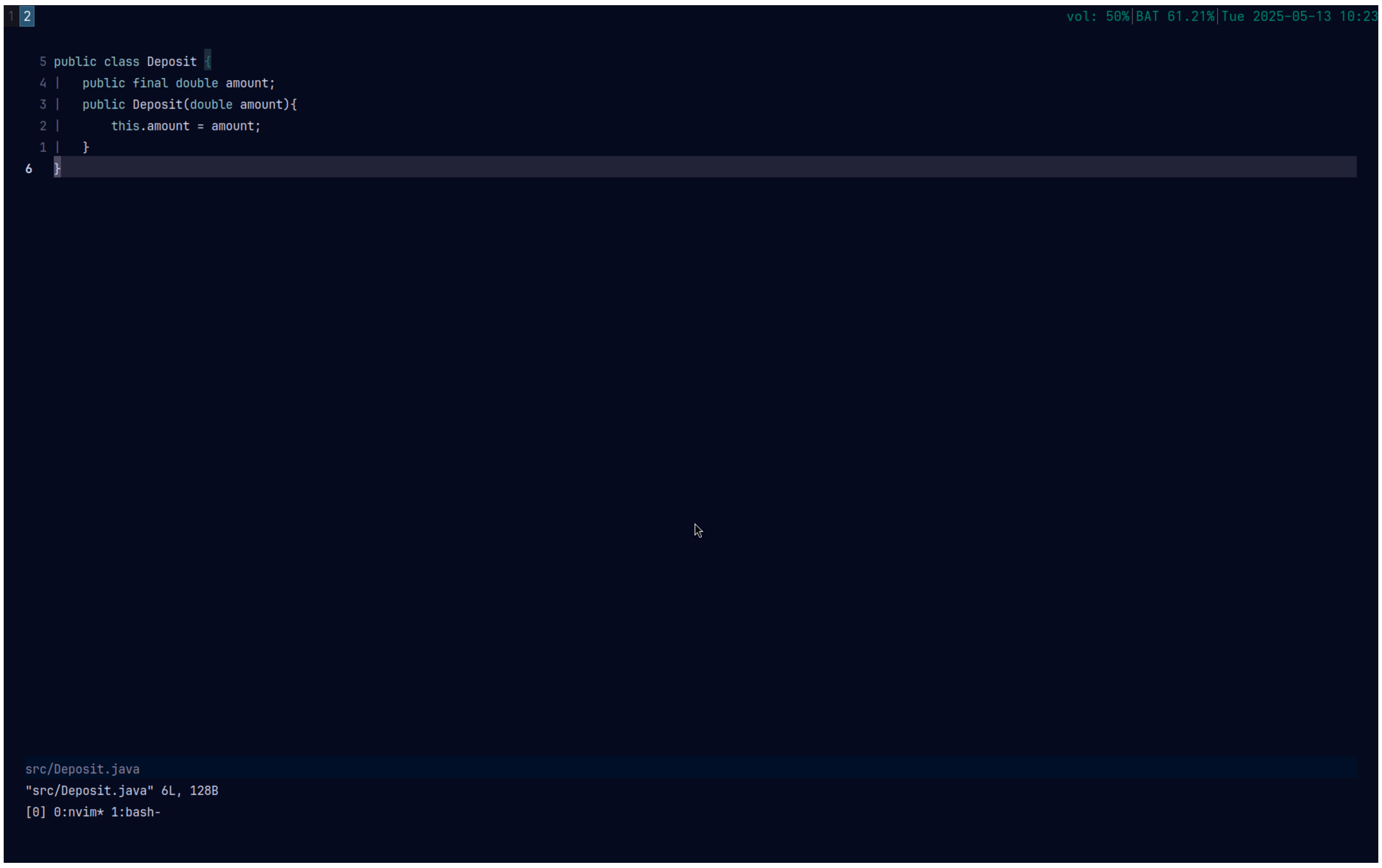
Screenshot of the output.

Now that everything is up and running, I proceed to tackle the actual task.

Final Code:

Deposit:

Since the `Deposit` class is simply a message that signals `BankAccount` class to deposit `X` amount; the code for this will be pretty simple:



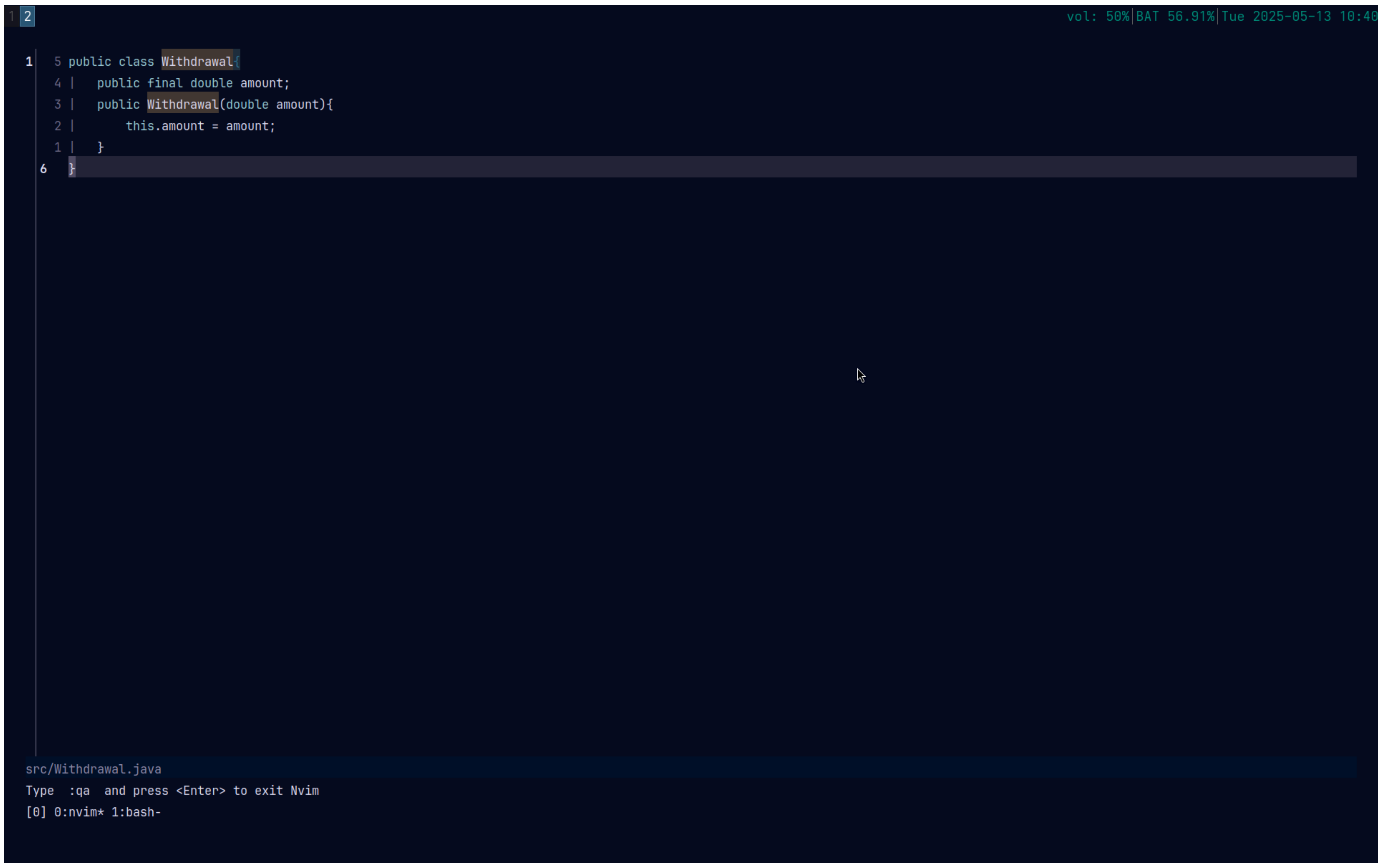
```
vol: 50%|BAT 61.21%|Tue 2025-05-13 10:23

public class Deposit {
    public final double amount;
    public Deposit(double amount){
        this.amount = amount;
    }
}

src/Deposit.java
"src/Deposit.java" 6L, 128B
[0] 0:nvim* 1:bash-
```

Screenshot of deposit class

All I've here is a constructor that simply sets the value of the public `amount` variable to whatever was passed in to the constructor. The variable `amount` is `final` so that it's effectively immutable; as `final` makes it so that the value can be only set once, in our case only in the constructor.

Withdrawal:

The `Withdrawal` class is pretty similar to the deposit class, the only distinction being the class name:



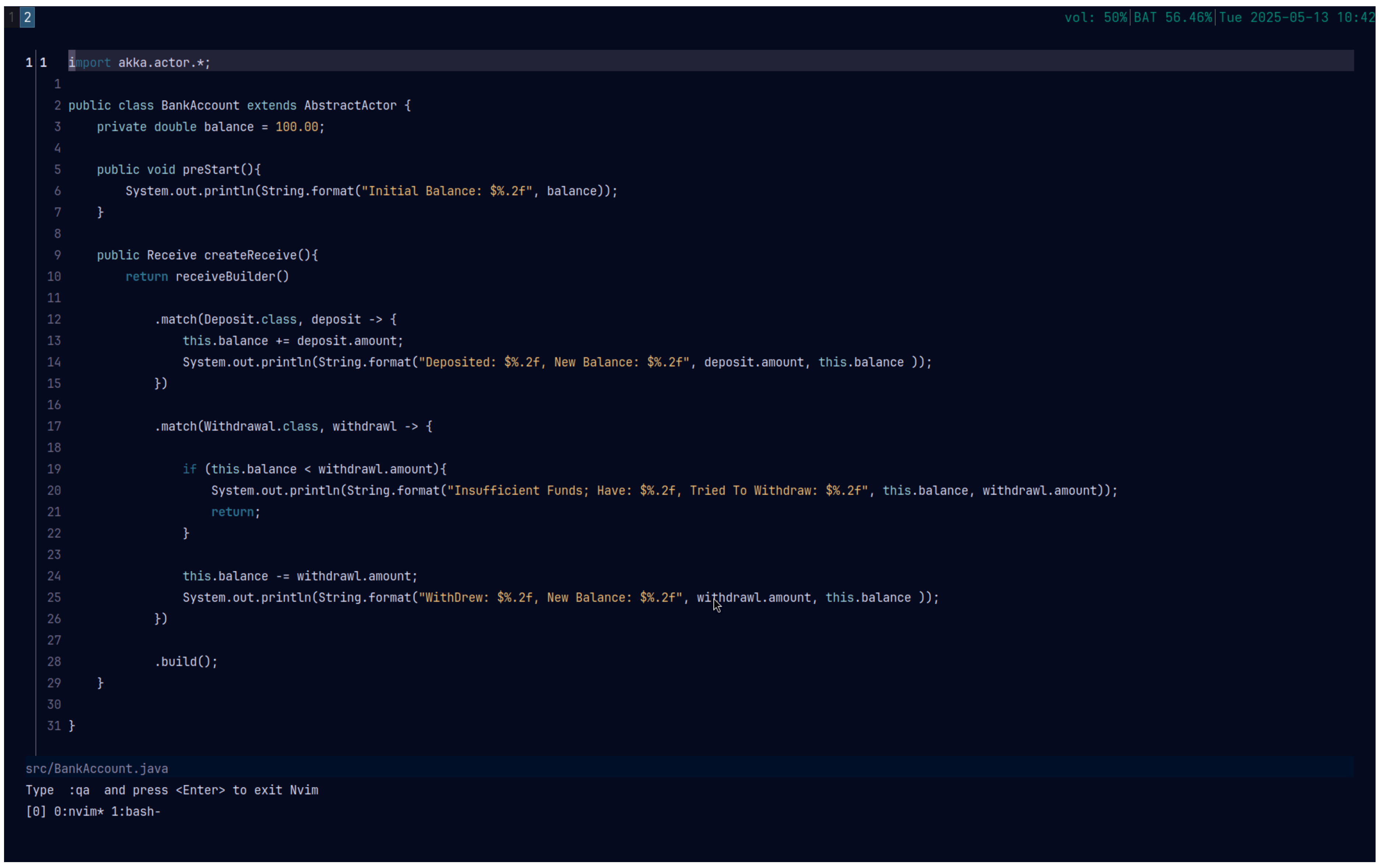
```
1 2                                                  vol: 50%|BAT 56.91%|Tue 2025-05-13 10:40

public class Withdrawal{
    public final double amount;
    public Withdrawal(double amount){
        this.amount = amount;
    }
}

src/Withdrawal.java
Type  :qa  and press <Enter> to exit Nvim
[0] 0:nvim* 1:bash-
```

Screenshot of the withdrawal class

Same idea as the `Deposit` class, `Withdrawal` has a `final amount` field that's set once in the constructor.
Only thing worth noting is I fixed a typo and renamed the file from `WithDraw.java` as evident here to `Withdrawal.java.`

BankAccount:

This class is the bulk of our codebase:



```
1 2                                                    vol: 50%|BAT 56.46%|Tue 2025-05-13 10:42

import akka.actor.*;

public class BankAccount extends AbstractActor {
    private double balance = 100.00;

    public void preStart(){
        System.out.println(String.format("Initial Balance: $%.2f", balance));
    }

    public Receive createReceive(){
        return receiveBuilder()

            .match(Deposit.class, deposit -> {
                this.balance += deposit.amount;
                System.out.println(String.format("Deposited: $%.2f, New Balance: $%.2f", deposit.amount, this.balance ));
            })

            .match(Withdrawal.class, withdrawl -> {

                if (this.balance < withdrawl.amount){
                    System.out.println(String.format("Insufficient Funds; Have: $%.2f, Tried To Withdraw: $%.2f", this.balance, withdrawl.amount));
                    return;
                }

                this.balance -= withdrawl.amount;
                System.out.println(String.format("WithDrew: $%.2f, New Balance: $%.2f", withdrawl.amount, this.balance ));
            })

            .build();
    }

}

src/BankAccount.java
Type  :qa  and press <Enter> to exit Nvim
[0] 0:nvim* 1:bash-
```

Screenshot of the BankAccount class

I extend `AbstractActor` to define how the actor handles messages, simplifying the implementation by abstracting away the boilerplate of the Actor interface. This also gives us built-in access to useful features like sender info, lifecycle hooks, and actor context.

Speaking of the lifecycle hooks, I make use of the `preStart()` hook to display the initial balance, which is always `$100`.

Next, I populate the `createReceive()` method. This is where we define the actor's response to incoming messages by overriding it to return a Receive object.

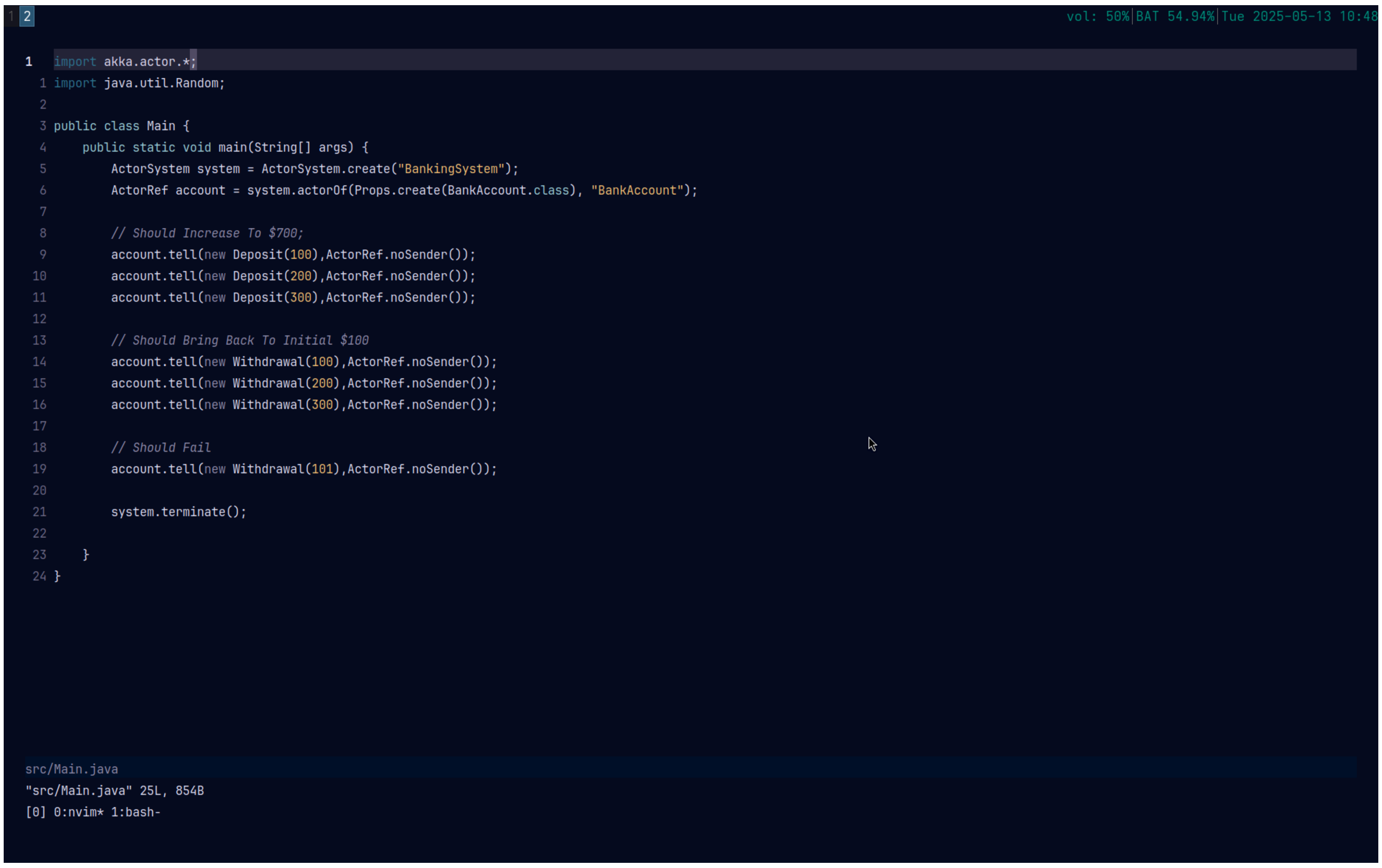
Within `createReceive()`, I use receiveBuilder() to match specific message types like `Deposit` and `Withdrawal`.
For the `Deposit` message, I pass it to a lambda function, aliased as `deposit` and simply add the `deposit.amount` to the current balance, displaying a message with the new balance. The `deposit.amount` comes from the `amount` variable we defined here with public access.

And for the `Withdrawal` message, I pass it to another lambda function, aliased as `withdrawal` and simply substract the `withdrawal.amount` to the current balance, displaying a message with the new balance. The `withdrawal.amount` comes from the `amount` variable we defined here with public access. In the case that we don't have enough funds to withdraw, we simply display that we have insufficient funds with relevant info and return from the function.

And finally we call the `.build()` to finalize the configuration and build the actual `Receive` object.

Main:

For our final class, all we need to do is initialize the `Akka` environment and generate 10 random transactions for `BankAccount` class to process.

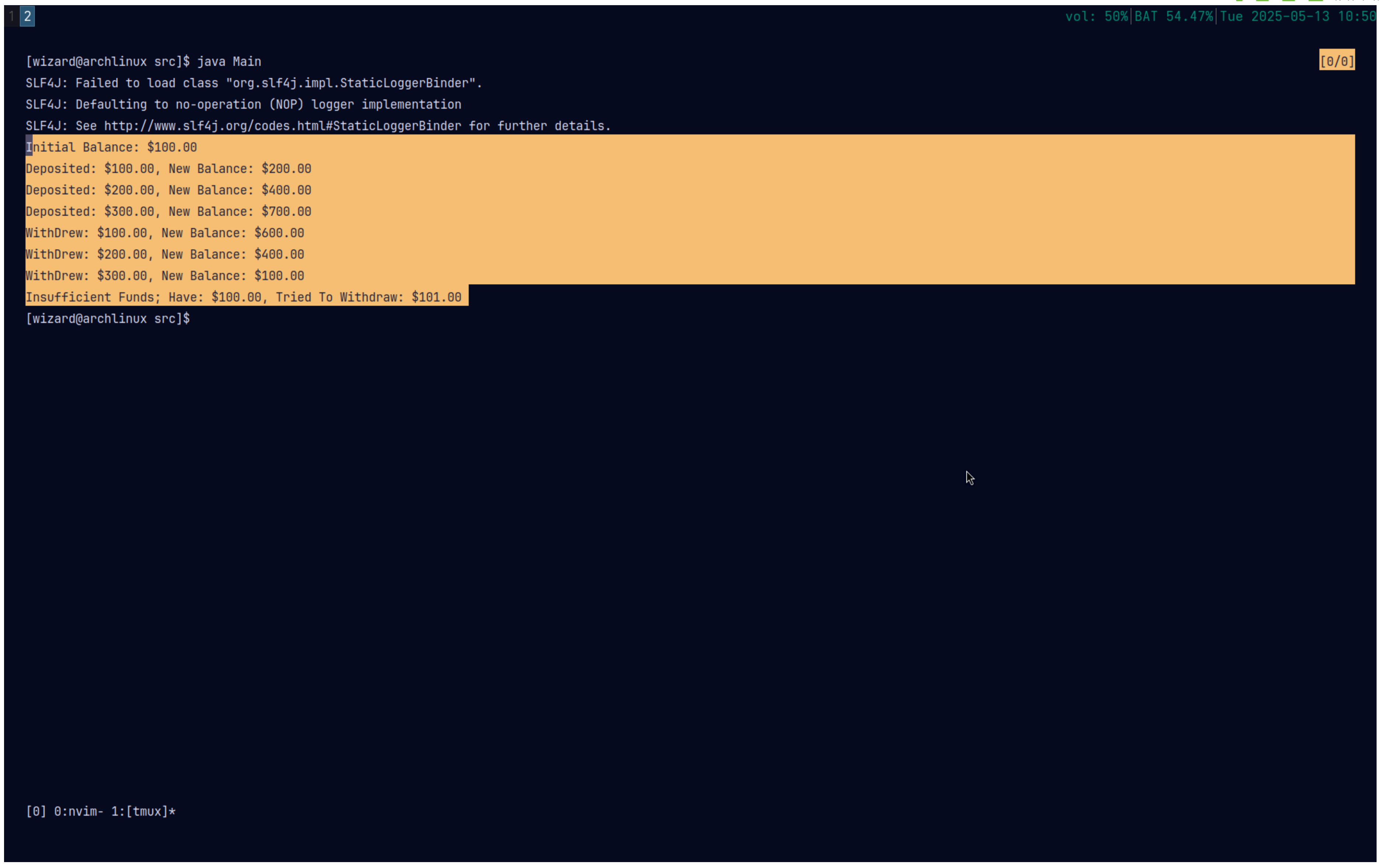


```
1 2                                                     vol: 50%|BAT 54.94%|Tue 2025-05-13 10:48

import akka.actor.*;
import java.util.Random;

public class Main {
    public static void main(String[] args) {
        ActorSystem system = ActorSystem.create("BankingSystem");
        ActorRef account = system.actorOf(Props.create(BankAccount.class), "BankAccount");

        // Should Increase To $700;
        account.tell(new Deposit(100),ActorRef.noSender());
        account.tell(new Deposit(200),ActorRef.noSender());
        account.tell(new Deposit(300),ActorRef.noSender());

        // Should Bring Back To Initial $100
        account.tell(new Withdrawal(100),ActorRef.noSender());
        account.tell(new Withdrawal(200),ActorRef.noSender());
        account.tell(new Withdrawal(300),ActorRef.noSender());

        // Should Fail
        account.tell(new Withdrawal(101),ActorRef.noSender());

        system.terminate();

    }
}

src/Main.java
"src/Main.java" 25L, 854B
[0] 0:nvim* 1:bash-
```

Screenshot for Main class with Proof Of Concept Code

Here, I simply create an instance of the `ActorSystem` and `ActorRef`. `ActorRef` holds the reference to a new `BankAccount`.

For now, I’m not generating random numbers. Instead, I've manually set the transaction amounts to test if everything works as expected. Essentially, I add $600 to the bank account and then withdraw the same amount, leaving the balance at the initial $100. Finally, I then attempt to withdraw $101 to ensure the program prevents withdrawals with insufficient funds.
This approach lets me test all the functionality, and once everything checks out, I'll move on to generating randomized transactions. Running the code now gives us:

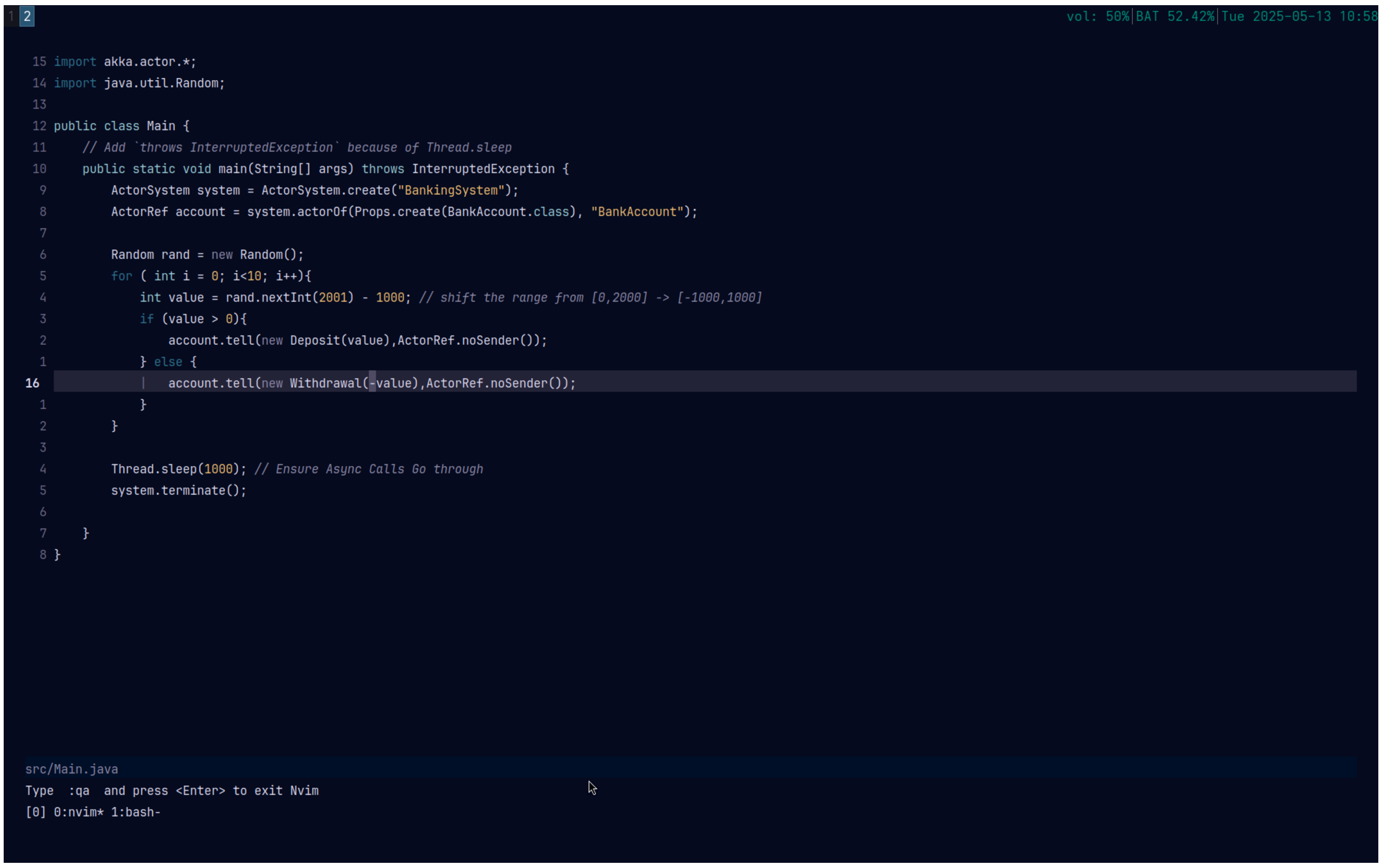


```
1 2                                                    vol: 50%|BAT 54.47%|Tue 2025-05-13 10:50

[wizard@archlinux src]$ java Main                                                       [0/0]
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Initial Balance: $100.00
Deposited: $100.00, New Balance: $200.00
Deposited: $200.00, New Balance: $400.00
Deposited: $300.00, New Balance: $700.00
WithDrew: $100.00, New Balance: $600.00
WithDrew: $200.00, New Balance: $400.00
WithDrew: $300.00, New Balance: $100.00
Insufficient Funds; Have: $100.00, Tried To Withdraw: $101.00
[wizard@archlinux src]$

[0] 0:nvim- 1:[tmux]*
```

Screenshot of the output

Evidently it's verified that the program works as expected. It prints the initial balance, adds in $600 across 3 deposits, withdraws the same $600 again across 3 withdrawals and when we are left with the initial $100, and try to deposit $101 it provides us with an error message.

Now that that works, I proceed to implement the randomized transactions:



```
1 2                                                    vol: 50%|BAT 52.42%|Tue 2025-05-13 10:58

import akka.actor.*;
import java.util.Random;

public class Main {
    // Add `throws InterruptedException` because of Thread.sleep
    public static void main(String[] args) throws InterruptedException {
        ActorSystem system = ActorSystem.create("BankingSystem");
        ActorRef account = system.actorOf(Props.create(BankAccount.class), "BankAccount");

        Random rand = new Random();
        for ( int i = 0; i<10; i++){
            int value = rand.nextInt(2001) - 1000; // shift the range from [0,2000] -> [-1000,1000]
            if (value > 0){
                account.tell(new Deposit(value),ActorRef.noSender());
            } else {
            |   account.tell(new Withdrawal(-value),ActorRef.noSender());
            }
        }

        Thread.sleep(1000); // Ensure Async Calls Go through
        system.terminate();

    }
}

src/Main.java
Type  :qa  and press <Enter> to exit Nvim
[0] 0:nvim* 1:bash-
```

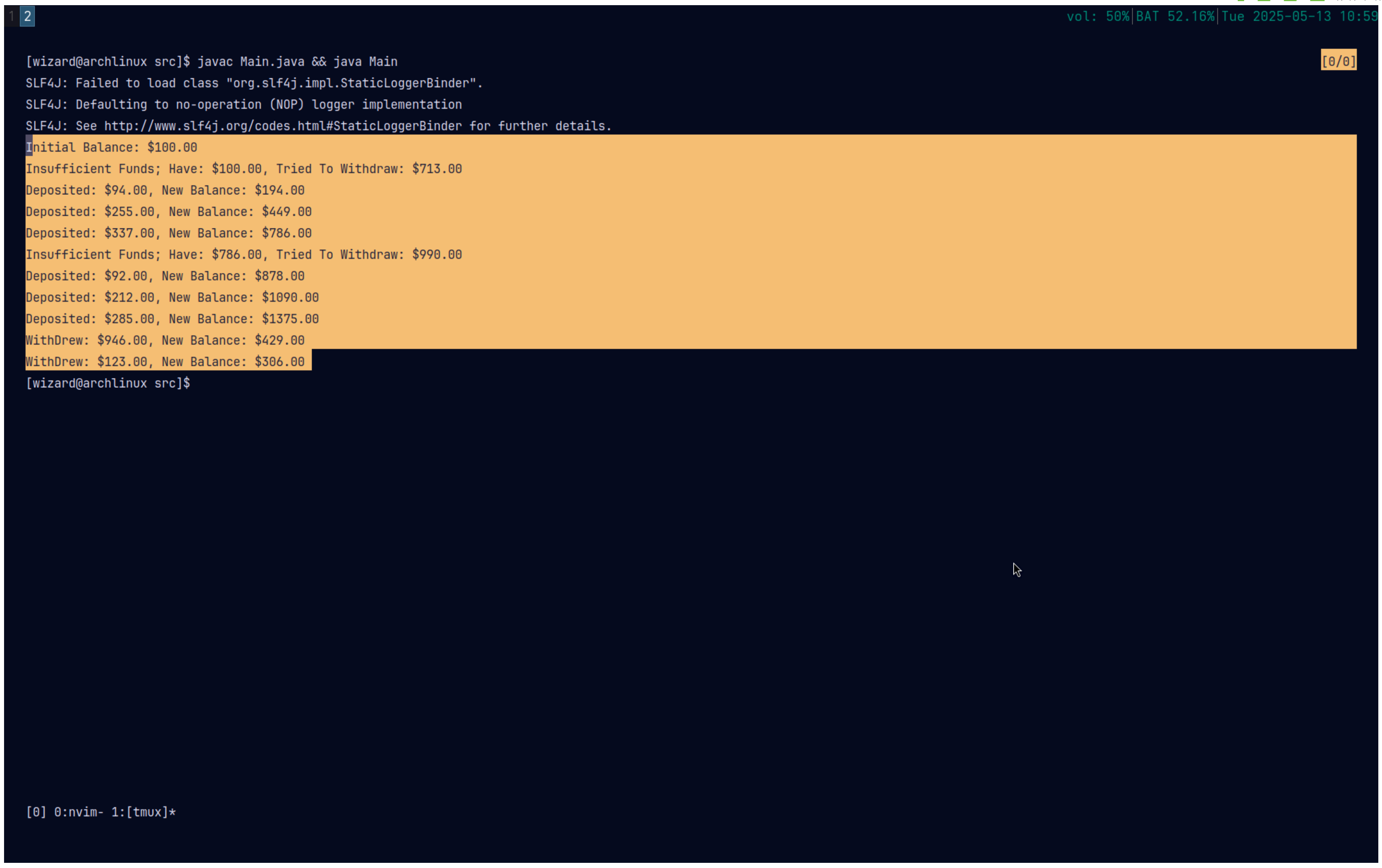
Screenshot of the updated main class

The main class now throws an `InterruptedException` to accommodate the `Thread.sleep(100)` on line 20. I added the `Thread.sleep(100)` because messages are processed asynchronously, and this provides a simple, effective way to ensure all messages are sent and processed before moving forward.

Instead of manually calling `account.tell` multiple times, I now use a `for` loop that runs for 10 iterations. For each iteration, a random number between 0 and 2000 is generated by `rand.nextInt(2001)`. Since `rand.nextInt` only gives a number from 0 (inclusive) to the specified number (exclusive), I subtract 1000 to shift the range from `[0,2000]` to `[-1000, 1000]`.

If the random number is positive, I create a `Deposit` object and send it to the `BankAccount` actor. If it's negative, I convert it to a positive value and create a `Withdrawal` object instead.

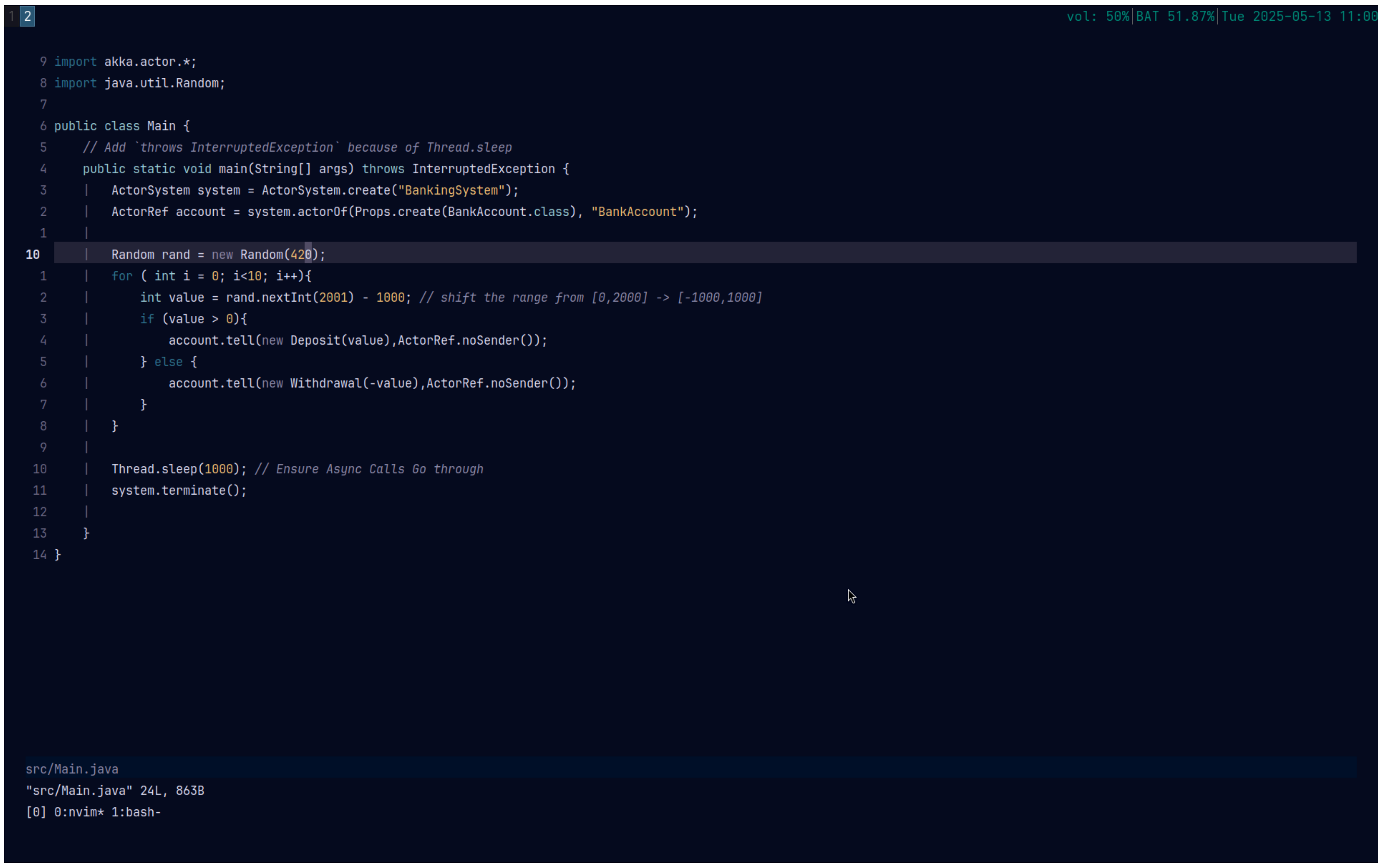
Finally, after the loop, I use `Thread.sleep(1000)` to ensure all asynchronous messages are processed, and then I terminate the actor system.



```
1 2                                                    vol: 50%|BAT 52.16%|Tue 2025-05-13 10:59

[wizard@archlinux src]$ javac Main.java && java Main                                   [0/0]
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Initial Balance: $100.00
Insufficient Funds; Have: $100.00, Tried To Withdraw: $713.00
Deposited: $94.00, New Balance: $194.00
Deposited: $255.00, New Balance: $449.00
Deposited: $337.00, New Balance: $786.00
Insufficient Funds; Have: $786.00, Tried To Withdraw: $990.00
Deposited: $92.00, New Balance: $878.00
Deposited: $212.00, New Balance: $1090.00
Deposited: $285.00, New Balance: $1375.00
WithDrew: $946.00, New Balance: $429.00
WithDrew: $123.00, New Balance: $306.00
[wizard@archlinux src]$

[0] 0:nvim- 1:[tmux]*
```

Screenshot of the output of the updated main class

As we can see, it is now correctly working with 10 randomly generated values. Finally, I specify a random seed, in this case 420 in the `Random()` function.

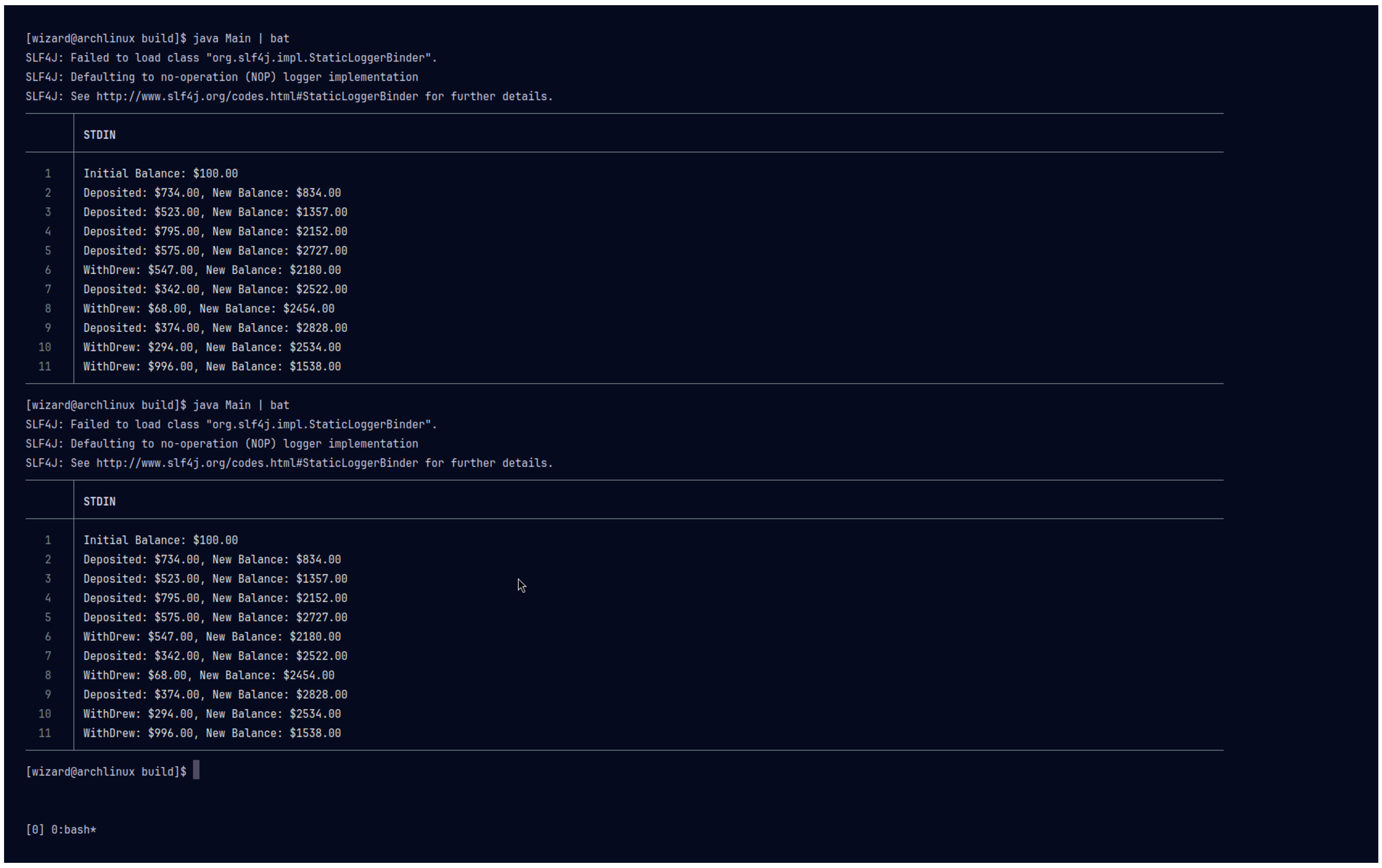


```java
import akka.actor.*;
import java.util.Random;

public class Main {
    // Add `throws InterruptedException` because of Thread.sleep
    public static void main(String[] args) throws InterruptedException {
        ActorSystem system = ActorSystem.create("BankingSystem");
        ActorRef account = system.actorOf(Props.create(BankAccount.class), "BankAccount");

        Random rand = new Random(420);
        for ( int i = 0; i<10; i++){
            int value = rand.nextInt(2001) - 1000; // shift the range from [0,2000] -> [-1000,1000]
            if (value > 0){
                account.tell(new Deposit(value),ActorRef.noSender());
            } else {
                account.tell(new Withdrawal(-value),ActorRef.noSender());
            }
        }

        Thread.sleep(1000); // Ensure Async Calls Go through
        system.terminate();

    }
}
```

```
src/Main.java
"src/Main.java" 24L, 863B
[0] 0:nvim* 1:bash-
```

Screenshot of main with specified seed

Doing this is helpful for reproducibility and verifying the output especially since this is a graded task; even if it technically makes the randomness deterministic.

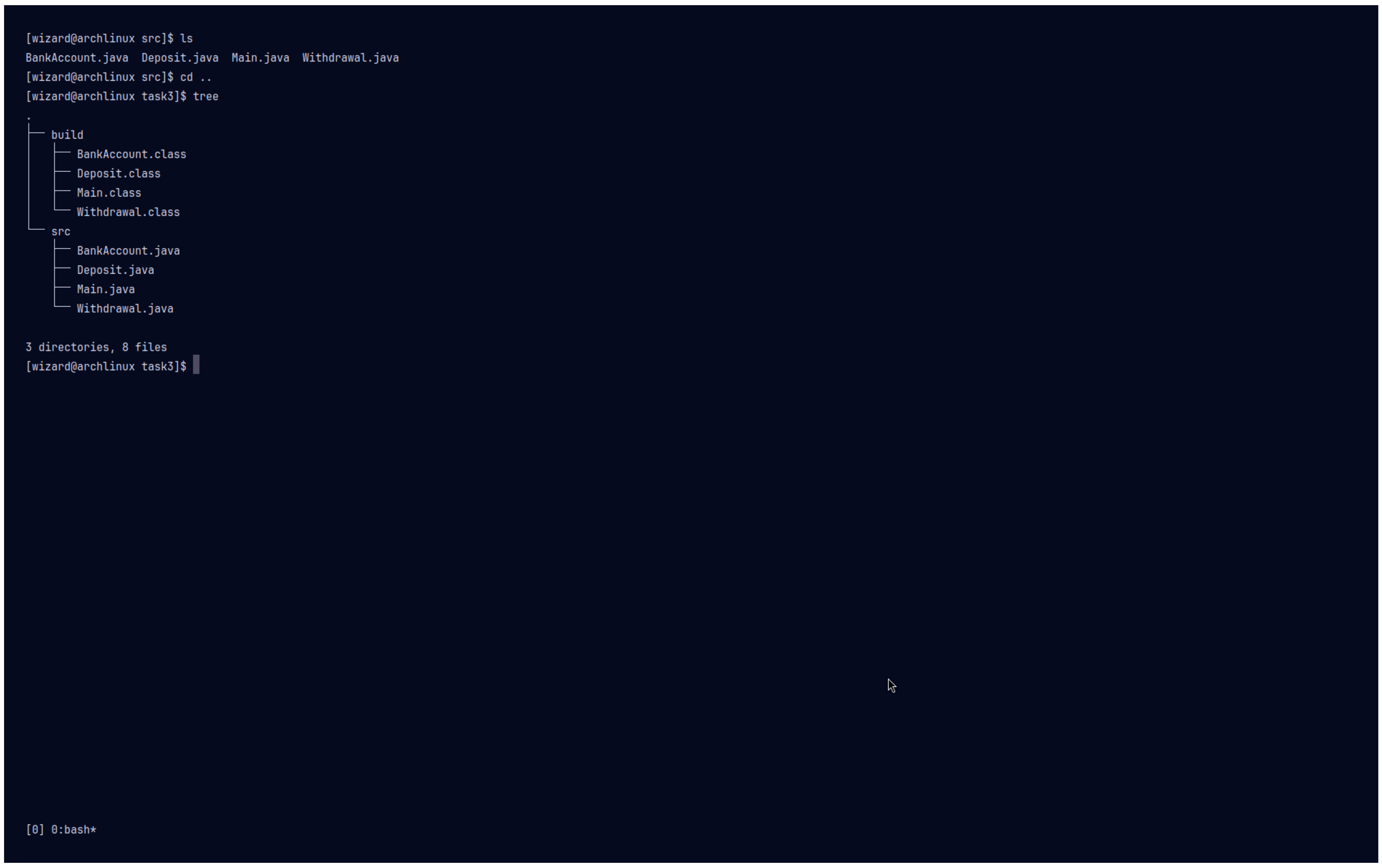


```
[wizard@archlinux build]$ java Main | bat
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
───────┬──────────────────────────────────────────────────
       │ STDIN
───────┼──────────────────────────────────────────────────
   1   │ Initial Balance: $100.00
   2   │ Deposited: $734.00, New Balance: $834.00
   3   │ Deposited: $523.00, New Balance: $1357.00
   4   │ Deposited: $795.00, New Balance: $2152.00
   5   │ Deposited: $575.00, New Balance: $2727.00
   6   │ WithDrew: $547.00, New Balance: $2180.00
   7   │ Deposited: $342.00, New Balance: $2522.00
   8   │ WithDrew: $68.00, New Balance: $2454.00
   9   │ Deposited: $374.00, New Balance: $2828.00
  10   │ WithDrew: $294.00, New Balance: $2534.00
  11   │ WithDrew: $996.00, New Balance: $1538.00
───────┴──────────────────────────────────────────────────
[wizard@archlinux build]$ java Main | bat
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
───────┬──────────────────────────────────────────────────
       │ STDIN
───────┼──────────────────────────────────────────────────
   1   │ Initial Balance: $100.00
   2   │ Deposited: $734.00, New Balance: $834.00
   3   │ Deposited: $523.00, New Balance: $1357.00
   4   │ Deposited: $795.00, New Balance: $2152.00
   5   │ Deposited: $575.00, New Balance: $2727.00
   6   │ WithDrew: $547.00, New Balance: $2180.00
   7   │ Deposited: $342.00, New Balance: $2522.00
   8   │ WithDrew: $68.00, New Balance: $2454.00
   9   │ Deposited: $374.00, New Balance: $2828.00
  10   │ WithDrew: $294.00, New Balance: $2534.00
  11   │ WithDrew: $996.00, New Balance: $1538.00
───────┴──────────────────────────────────────────────────
[wizard@archlinux build]$

[0] 0:bash*
```

Screenshot showing the effect of seeding piped through bat for readability

Now the program retains the same "random sequence" across all the runs. If this is not desirable, simply change `Random(420)` to `Random()` in line number 10.

Finally, I moved the compiled files to the `build` directory and this marks the end of the task.



```
[wizard@archlinux src]$ ls
BankAccount.java  Deposit.java  Main.java  Withdrawal.java
[wizard@archlinux src]$ cd ..
[wizard@archlinux task3]$ tree
.
├── build
│   ├── BankAccount.class
│   ├── Deposit.class
│   ├── Main.class
│   └── Withdrawal.class
└── src
    ├── BankAccount.java
    ├── Deposit.java
    ├── Main.java
    └── Withdrawal.java

3 directories, 8 files
[wizard@archlinux task3]$

[0] 0:bash*
```