DESCRIPTION: For the final project, we will take a look at Erlang. Note that while Erlang is a functional language, in the same style as Clojure, our focus here is the concurrency model provided by Erlang. In particular, this assignment will require you to gain some familiarity with the concept of message passing. In fact, Erlang does this more effectively than any other modern programming language.

That said, there is a "twist" with the project. The course is called "Comparative Programming Languages" for a reason. The purpose is not just to learn something about other languages but to get a better sense of how problems can be solved differently, depending on the language used. Often, a well-chosen language can make the job much easier and much more intuitive.

So, in addition to implementing the application in Erlang, you will implement the same application using Java, arguably the most popular imperative language in use today. While Java will be far more comfortable for many of you, it is a general purpose language that was not designed specifically for concurrency (though it has always provided support for this).

In short, the project emphasizes the "comparative" element in the course's title. Note, however, that this does not mean that the project is massive in size. The application itself is not large, so neither the Erlang program nor the Java program will require a huge amount of source code. Instead, you will have to look at the problem differently in the two cases.

DETAILS: In the description below, we will describe the project in terms of Erlang. A short section on the Java version will be given at the end.

Your objective is to model a simple banking environment. Specifically, you will be given a small number of customers, each of whom will contact a set of banks to request a number of loans. Eventually, they will either receive all of the money they require or they will end up without completely meeting their original objective. The application will display information about the various banking transactions before it finishes. That's it.

So now for the details. To begin, you will need a handful of customers and banks. These will be supplied in a pair of very simple text files – *customers.txt* and *banks.txt*. While Erlang provides many file primitives for processing disk files, the process is not quite as simple as Clojure's slurp() function. So the two files will contain records that are already pre-formatted. In other words, they are ready to be read directly into standard Erlang data structures.

An example *customers.txt* file would be:

```
{jill,450}.
{joe,157}.
{bob,100}.
{sue,125}.
```

An example *banks.txt* file would be:

```
{rbc,800}.
{bmo,700}.
{ing,200}.
```

In other words, each file simply contains a set of erlang tuples. You will see that each label is associated with a number. For customers, this is the total funds that they are hoping to obtain. For banks, the number represents their total financial resources that can be used for loans.

To read these files, all you have to use is the `consult()` function in the `file` module. This will load the contents of either file directly into an erlang structure (a list of tuples). Note that NO error checking is required. The text files are guaranteed to exist and contain valid data.

In addition, however, please keep in mind that these are just samples. The test files will have different customer/bank names and may also contain a different numbers of customers or banks (but no more than 10 of either).

So your job now is to take this information and create an application that models the banking environment. Because customers and banks are distinct entities in this world, each will be modeled as a separate task/process. When the application begins, it will therefore generate a new process for each customer and each bank. Because you do not know how many customers or banks there will be, or even their names, you cannot "hard code" this phase of the application.

The customer and bank tasks will then start up and wait for contact (you may want to make each new task sleep for a 100 milliseconds or so, just to make sure that all tasks have been created and are ready to be used). So the banking mechanism works as follows:

1. Each customer wants to borrow the amount listed in the input file. At any one time, however, they can only request a maximum of 50 dollars. When they make a request, they will therefore choose a random dollar amount between 1 and 50 for their current loan.
2. When they make a request, they will also randomly choose one of the banks as the target.
3. Before each request, a customer will wait/sleep a random period between 10 and 100 milliseconds. This is just to ensure that one customer doesn't take all the money from the banks at once.
4. So the customer will make the request and wait for a response from the bank. It will not make another request until it gets a reply about the current request.
5. The bank can accept or reject the request. It will reject the request if the loan would reduce its current financial resources below 0. Otherwise, it grants the loan and notifies the customer.
6. If the loan is granted, the customer will deduct this amount from its total loan requirement and then randomly choose a bank (possibly the same one) and make another request (again, between 1 and 50 dollars).
7. If the loan is rejected, however, the customer will remove that bank from its list of potential lenders, and then submit a new request to the remaining banks.
8. This process continues until customers have either received all of their money or they have no available banks left to contact.

And that's it.

Of course, we need a way to demonstrate that all of this has worked properly. To begin, it is important to understand that this is a multi-process Erlang program. The "master" process will be the initial process that, in turn, spawns processes for each of the customers (the master process is like the class containing "main" in Java). So, in our little example above, there will be 8 processes in total: the master, 4 customers and 3 banks.

To confirm the validity of the program, we need a series of info messages to be printed to the screen. These include:

1. A customer will indicate that they are making a loan request.
2. Banks will indicate whether they have accepted or denied a given request.
3. Before the program ends, customers will indicate if they have reached their goal or not.
4. Before the program ends, banks will indicate their remaining funds.

IMPORTANT: The "master" process is the only process that should display anything to the screen. So all info must be sent to the master process, where it will be displayed. No customer or bank process should ever print anything.

Below, we see partial output for our running example (including input data):

```
** Customers and loan objectives **
jill: 450
joe: 157
bob: 100
sue: 125

** Banks and financial resources **
rbc: 800
bmo: 700
ing: 200


joe requests a loan of 40 dollar(s) from ing
bob requests a loan of 13 dollar(s) from bmo
jill requests a loan of 19 dollar(s) from bmo
bmo approves a loan of 13 dollars from bob
sue requests a loan of 43 dollar(s) from rbc
bob requests a loan of 21 dollar(s) from ing
ing approves a loan of 40 dollars from joe

…

rbc approves a loan of 29 dollars from sue
bob requests a loan of 4 dollar(s) from bmo
ing denies a loan of 27 dollars from jill
ing approves a loan of 4 dollars from bob

…

bob has reached the objective of 100 dollar(s). Woo Hoo!
jill has reached the objective of 450 dollar(s). Woo Hoo!
ing has 5 dollar(s) remaining.
sue was only able to borrow 98 dollar(s). Boo Hoo!
```

It should be obvious that that the loan requests and loan decision should match up, though you should keep in mind that the actual printing can occur in any order.

Moreover, the calculations should be accurate at the end:

1. The total of the loans should be equal to the resources used by the banks
2. No bank should have a negative balance.
3. No customer should have more money than they needed.

One final thing. Because of the random pause before each loan request, as well as the random loan amounts in the loan requests, the output should be different each time you run the program, even for the same input.