ÉTUDES FOR ERLANG

Companion exercises for Introducing Erlang

J. David Eisenberg

O'REILLY®

Études for Erlang

J. David Eisenberg

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Études for Erlang

by J. David Eisenberg

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Simon St. Laurent **Interior Designer:** David Futato

March 2013: First Edition

Revision History for the First Edition:

2013-03-20: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449366452* for release details.

ISBN: 978-1-449-36645-2

[]

Table of Contents

Études for Erlang

Welcome to *Études for Erlang*. In this book, you will find descriptions of programs that you can write in Erlang. The programs will usually be short, and each one has been designed to provide practice material for a particular Erlang programming concept. These programs have not been designed to be of considerable difficulty, though they may ask you to stretch a bit beyond the immediate material and examples that you find in the book Introducing Erlang.

This book is open source, so if you'd like to contribute, make a correction, or otherwise participate in the project, check out oreillymedia/etudes-for-erlang on GitHub for details. If we accept your work, we'll add you to the contributors chapter.

The online version of the book is at Études for Erlang on O'Reilly Labs.

Contributor Guidelines

If you're considering making a contribution, here are some guidelines to keep in mind:

Creative Commons license.
: All contributions made to this site are required to be made under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. This means that by making a content contribution, you are agreeing that it is licensed to us and to others under this license. If you do not want your content to be available under this license, you should not contribute it.

Submit only your own work.
: You warrant that all work that you contribute to this site is your original work, except for material that is in the public domain or for which you have obtained permission. Feel free to draw from your own existing work (blogs, articles, talks, etc.), so long as you are happy with the Creative Commons license.

Your submission may not be accepted.
: Be aware that we may not be able to accept your contribution.

Keep your title pithy and to the point.
: The title should only be a 2 to 10 words long if possible and should summarize or capture the essence of the advice. Keep your discussion between 400 and 500 words.

Volunteers only.
: Contributions are made on a volunteer basis — in other words, contributors are not paid for their contributions. The contributions will be made easily available to everyone on the Web for free. However, remember that those of you whose contributions are chosen for publication will get your name attached to your work and your bio published next to it. Any item you contribute you can also reuse in any form you wish, such as in a blog posting.

Only submit a pull request when you consider your work complete.
: Please submit your work once it is complete. Once you make a pull request, the editor will review the submission and (possibly) suggest some changes. Reducing work in progress makes it easier for you to see your own progress and for others to see the progress of the whole project.

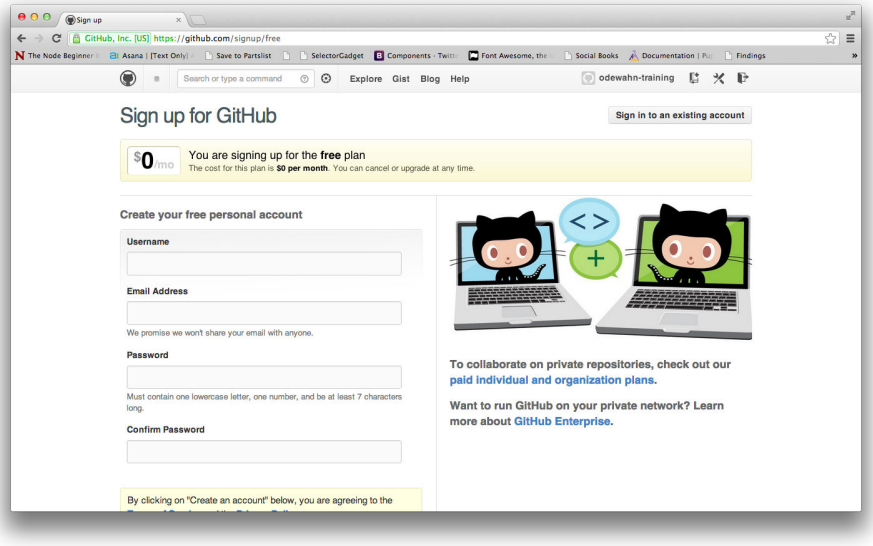
Check spelling, word count, and formatting.
: Such checking seems obvious part, but it is worth a reminder — sometimes it seems that it is honored more in the breach than in the observance. US spelling is used for the contributions, which should be between 400 and 500 words in length. Formatting can be checked by looking at the saved page in GitHub. If it looks right there, it's probably right.

How to Contribute

If you're new to git and GitHub and just want to keep things as simple as possible, this tutorial will give you a quick and easy way to make your contribution. Here are the steps you'll need to follow:

Create a GitHub account

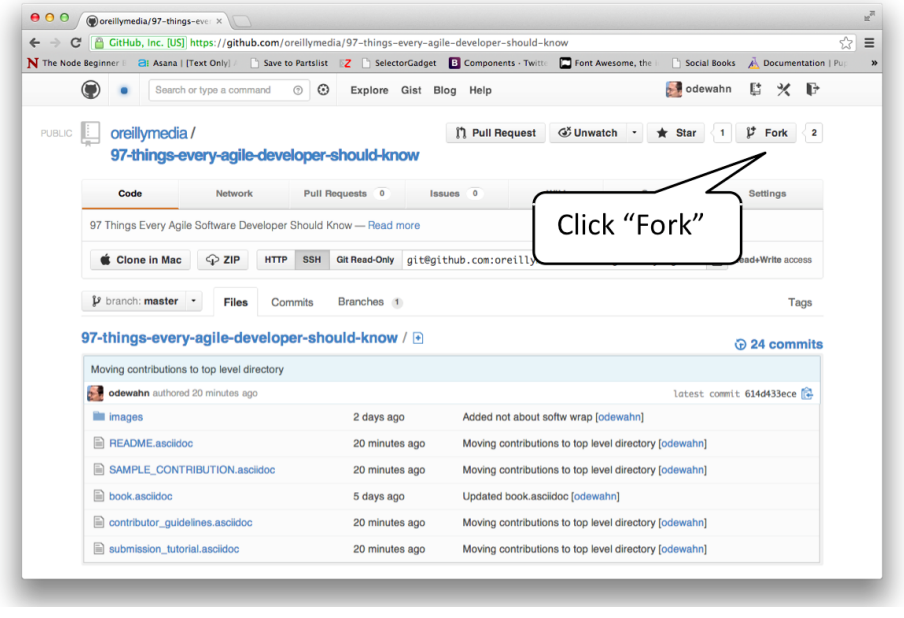
To create and edit a page or to comment on an existing page, you will need to create an account on GitHub. If you don't have one already, then go to the GitHub Signup page. It's free.



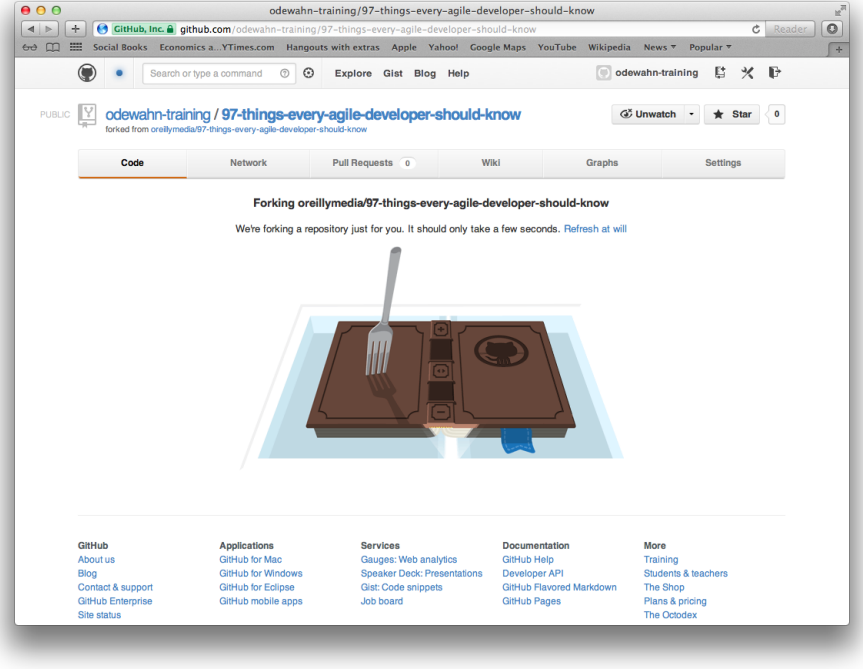
GitHub has excellent tools for collaborating and workflow management, and will be the primary way we communicate with you over the course of the project.

Copy ("fork") the project repository to your account

Once you've got an account, fork (GitHub lingo for copying) the main project to your account. To do this, go to the Etudes for Erlang repository on GitHub and click the "Fork" button at the upper right hand side of the screen.



The following screen will appear while GitHub copies the repository to your account:

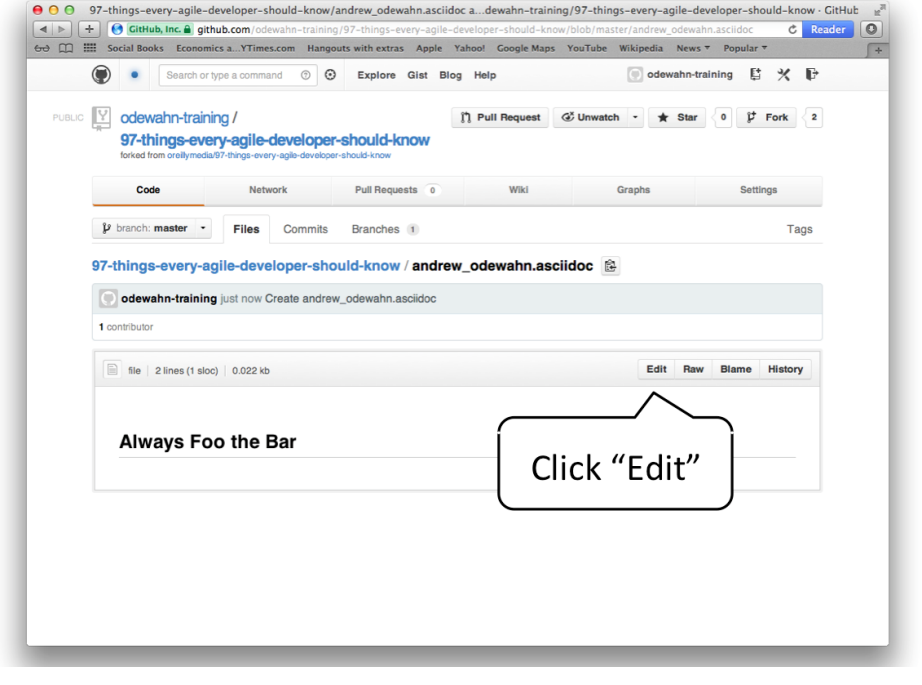


Edit your file using AsciiDoc

Once you've got the file created, you can start editing it at your leisure. Remember to:

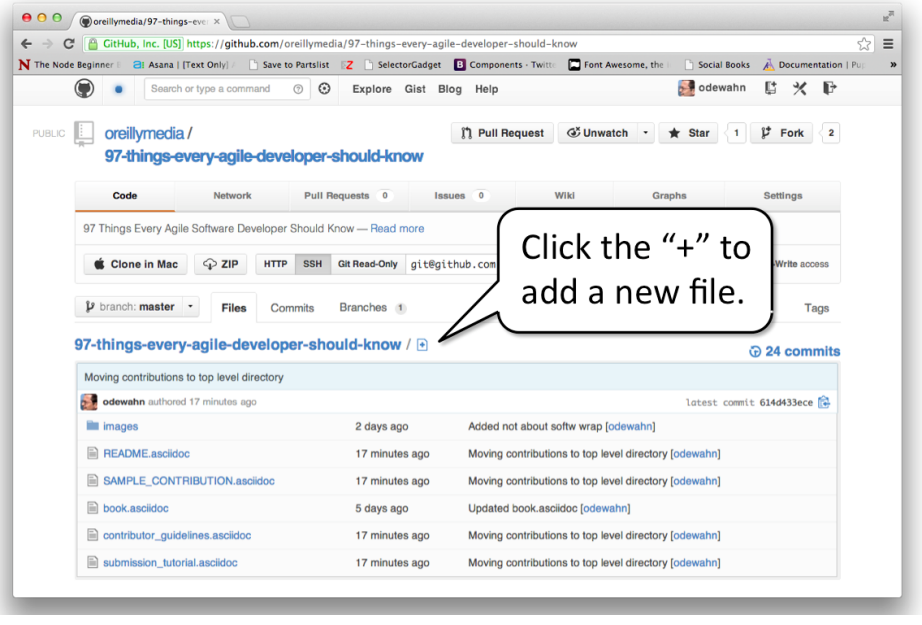
- Mark up your text using AsciiDoc, which is similar to Markdown and other basic wiki-like markup formats.
- Change the line wrapping from "No Wrap" to "Soft Wrap." Otherwise, all your text will appear on a single line.

To edit the file, all you have to do is click its name in the directory listing in GitHub and then press the "Edit" button.



If you want to add an entirely new topic area, you'll need to create a new file in GitHub. To do this, click the "+" button next to the directory name to create a new file

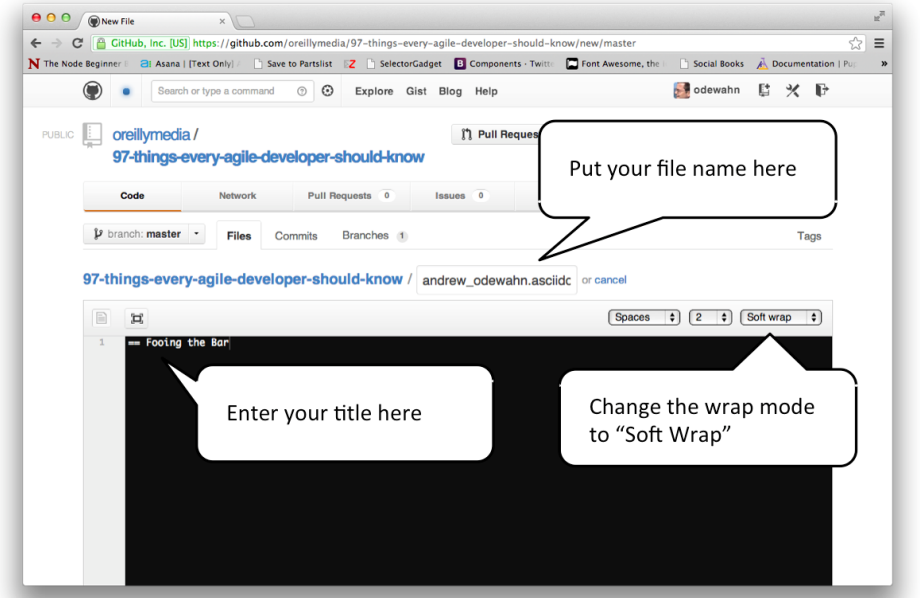
Locate the small "+" sign next to the repository name. (A tooltip will appear that says "Create a new file here" when you hover your mouse above it.) Click the "\+" button:



In the new screen, you'll need to:

- Enter a name for the file. Name the file according to the general topic area, and be sure to include the extension ".asciidoc" at the end. For example, "foo_and_bar.asciidoc".
- Enter the chapter title in the editing box; it should be prefaced with two "==" signs. For example, "== Always Foo the Bar"
- Once you've entered the filename and title, the "Commit Changes" button at the bottom of the screen will activate. Click the button to save your file.

You will see something like this:



Double check your submission and add your biography

Before you submit your request, make sure that you have:

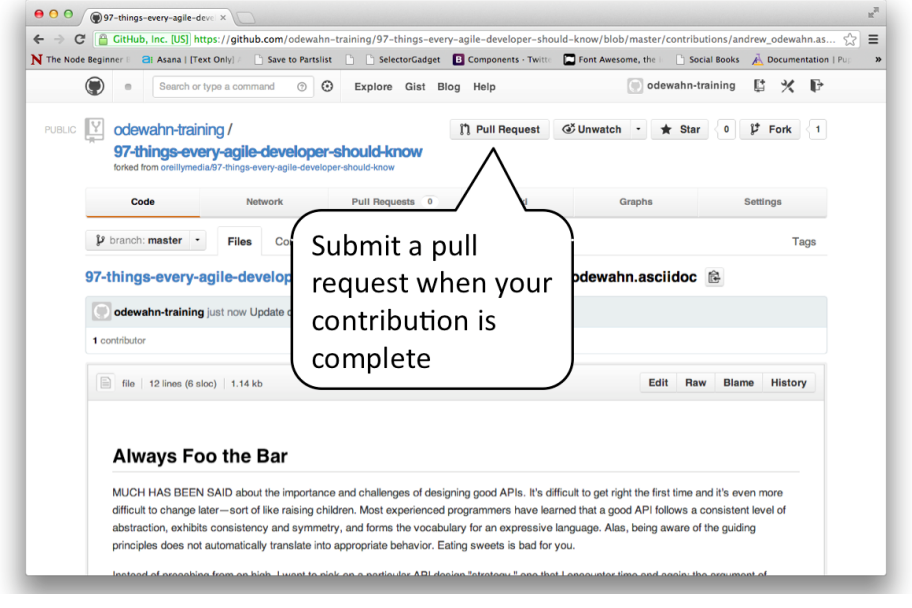
- Run a spell check
- Make sure it's 400-500 words in length
- Add your name and a short biography
- Check the formatting to make sure it looks OK

Your biography should look like this:

```
.About the Author
[NOTE]
****
Name::
   Nicola Tesla
Biography::
   Nicola Tesla is an inventor, electrical engineer, mechanical engineer, physicist, and futurist
****
```

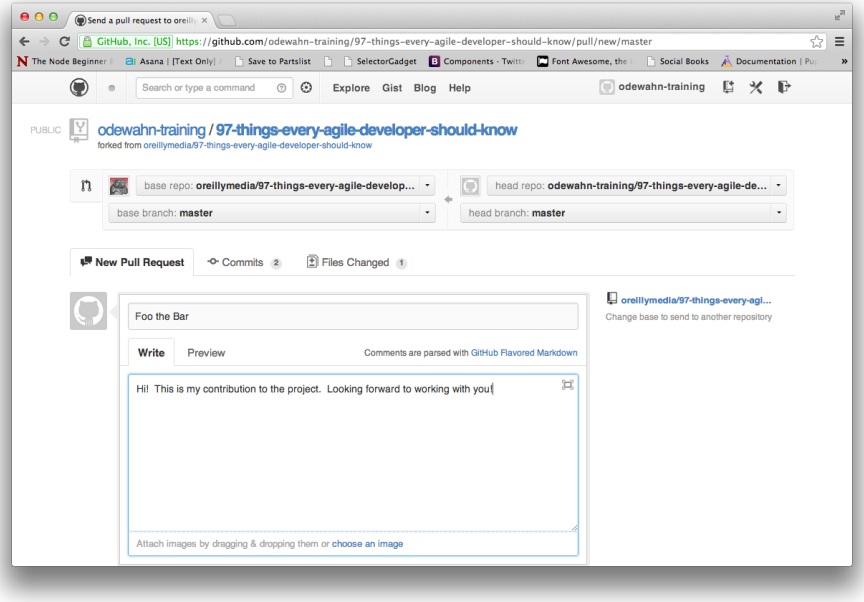
Submit a pull request

Once you've completed and double checked your submission, you're ready to submit it back to O'Reilly. This is done by creating a "pull request" that will trigger the review process.



When you submit the pull request, you'll also be able to submit some additional information that will help us track your work:

- A title. Please enter your name and the title of the contribution. For example, "Andrew Odewahn: Always Foo the Bar"
- A comment. Tell us a little bit about your contribution, as well as anything else you think we should know.



Engage in back-and-forth

Once you submit your pull request, the project's maintainers will begin a back and forth with you in the discussion features. You might be asked to make some revisions, shorten it, add additional elements, and so forth.

Preface: What's an étude?

An étude, according to Wikipedia, is "an instrumental musical composition, usually short and of considerable difficulty, usually designed to provide practice material for perfecting a particular musical skill."

What are Études for Erlang?

In this book, you will find descriptions of programs that you can compose (write) in Erlang.The programs will usually be short, and each one has been designed to provide practice material for a particular Erlang programming concept. Unlike musical études, these programs have not been designed to be of considerable difficulty, though they may ask you to stretch a bit beyond the immediate material and examples that you find in the book Introducing Erlang.

How This Book was Written

While reading the early release version of *Introducing Erlang*, I began by copying the examples in the book. (This always helps me learn the material better.) I then began experimenting with small programs of my own to make sure I really understood the concepts. As I continued writing my own examples, I thought they might be useful to other people as well. I contacted Simon St. Laurent, the author of Introducing Erlang, and he liked the idea of having these companion exercises and suggested naming them études. At some point, the études took on a life of their own, and you are reading the result now.

I was learning Erlang as I was creating the solutions to the études, following the philosophy that "the first way that works is the right way." Therefore, don't be surprised if you see some fairly naïve code that an expert Erlang programmer would never write.

Working with Other Books

Although this was based on *Introducing Erlang*, you can use it with other Erlang books. A note at the beginning of each chapter will point you to relevant sections in other books. The books listed through are:

- Erlang Programming, by Francesco Cesarini, and Simon Thompson (O'Reilly Media, 2009).
- Programming Erlang, by Joe Armstrong (Pragmatic Programmers, 2007).
- Erlang and OTP in Action by Martin Logan, Eric Merritt, and Richard Carlsson (Manning, 2010).
- Learn You Some Erlang for Great Good! by Fred Hebert (No Starch Press, 2013) - also available at *http://learnyousomeerlang.com/*.

Acknowledgments

Many thanks to Simon St. Laurent, who wrote *Introducing Erlang*. His book not only got me to begin to understand functional programming, but also made me realize that it was a lot of fun. Simon also felt that the exercises I was writing for myself could be useful to others, and he encouraged me to continue developing them.

Have any suggested topics?

Suggest topics you'd like to see covered here, or just go add them yourself at oreillymedia/etudes-for-erlang on GitHub.

CHAPTER 1

Getting Comfortable with Erlang

You can learn more about working with `erl` in Chapter 2 of *Erlang Programming*, Chapters 2 and 6 of *Programming Erlang*, Section 2.1 of *Erlang and OTP in Action*, and Chapter 1 of *Learn You Some Erlang For Great Good!*.

Étude 1-1: Experimenting with Errors

The first chapter of Introducing Erlang encourages you to play around with the interacative `erl` system. In this étude, keep using `erl`, but purposely make errors.

Try leaving out parentheses in arithmetic expressions. Try putting numbers next to each other without an operator between them. Try adding `"adam"` to `12`. Make up variable names that you are sure Erlang wouldn't ever accept.

That way, you'll get a feel for the sort of error messages Erlang produces and not be as baffled when you get errors that you aren't expecting.

CHAPTER 2

Functions and Modules

You can learn more about working with functions and modules in Chapters 2, 3, and 9 of *Erlang Programming*, Chapter 3 of *Programming Erlang*, Sections 2.3, 2.5, and 2.7 of *Erlang and OTP in Action*, and Chapters 2 and 3 of *Learn You Some Erlang For Great Good!*. There's more on documentation in Chapter 18 of *Erlang Programming* and types in Chapter 30 of *Learn You Some Erlang For Great Good!*.

Étude 2-1: Writing a Function

Write a module with a function that takes the length and width of a rectangle and returns (yields) its area. Name the module `geom`, and name the function `area`. The function has arity 2, because it needs two pieces of information to make the calculation. In Erlang-speak: write function `area/2`.

Here is some sample output.

```
1> c(geom).
{ok,geom}
2> geom:area(3,4).
12
3> geom:area(12,7).
84
```

See a suggested solution in Appendix A.

Étude 2-2: Documenting a Module

Document the `geom` module you wrote in Étude 2-1. See a suggested solution in Appendix A.

Étude 2-3: Documenting a Function

Document the `area/2` function, and create an *overview.edoc* file to complete the documentation of the application you've written. See a suggested solution in Appendix A.

CHAPTER 3

Atoms, Tuples, and Pattern Matching

You can learn more about working with atoms, tuples, and pattern matching in Chapter 2 of *Erlang Programming*, Chapter 2 of *Programming Erlang*, Sections 2.2 and 2.4 of *Erlang and OTP in Action*, and Chapters 1 and 3 of *Learn You Some Erlang For Great Good!*.

Étude 3-1: Pattern Matching

Use atoms and pattern matching to make your `area` function calculate the area of a rectangle, triangle, or ellipse. If your parameters are `Shape`, `A` and `B`, the area for the atom `rectangle` is `A * B`, where `A` and `B` represent the length and width. For a `triangle` atom, the area is `A * B / 2.0`, with `A` and `B` representing the base and height of the triangle. For an `ellipse` atom, the area is `math:pi() * A * B`, where `A` and `B` represent the major and minor radiuses.

Here is some sample output.

```
1> c(geom).
{ok,geom}
2> geom:area(rectangle, 3, 4).
12
3> geom:area(triangle, 3, 5).
7.5
4> geom:area(ellipse, 2, 4).
25.132741228718345
```

See a suggested solution in Appendix A.

Étude 3-2: Guards

Even though you won't get an error message when calculating the area of a shape that has negative dimensions, it's still worth guarding your `area/3` function. You will want two guards for each pattern to make sure that both dimensions are greater than or equal to zero. Since **both** have to be non-negative, use commas to separate your guards.

Here is some sample output.

```
1> c(geom).
{ok,geom}
2> geom:area(rectangle, 3, 4).
12
3> geom:area(ellipse, 2, 3).
18.84955592153876
4> geom:area(triangle, 4, 5).
10.0
5> geom:area(square, -1, 3).
** exception error: no function clause matching geom:area(square,-1,3) (geom.erl, line 18)
```

See a suggested solution in Appendix A.

Étude 3-3: Underscores

If you enter a shape that `area/3` doesn't know about, or if you enter negative dimensions, Erlang will give you an error message. Use underscores to create a "catch-all" version, so that anything at all that doesn't match a valid rectangle, triangle, or ellipse will return zero. This goes against the Erlang philosophy of "let it fail," but let's look the other way and learn about underscores anyway.

Here is some sample output.

```
1> geom:area(rectangle, 3, 4).
12
2> geom:area(pentagon, 3, 4).
0
3> geom:area(hexagon, -1, 5).
0
4> geom:area(rectangle, 1, -3).
0
```

See a suggested solution in Appendix A.

Étude 3-4: Tuples as Parameters

Add an `area/1` function that takes a tuple of the form `{shape,number,number}` as its parameter. Export it instead of `area/3`. The `area/1` function will call the private `area/3` function.

Here is some sample output.

```
1> c(geom).
{ok,geom}
2> geom:area({rectangle, 7, 3}).
21
3> geom:area({triangle, 7, 3}).
10.5
4> geom:area({ellipse, 7, 3}).
65.97344572538566
```

See a suggested solution in Appendix A.

CHAPTER 4

Logic and Recursion

You can learn more about working with logical flow and recursion in Chapter 3 of *Erlang Programming*, Chapter 3 of *Programming Erlang*, Sections 2.6 and 2.15 of *Erlang and OTP in Action*, and Chapters 3 and 5 of *Learn You Some Erlang For Great Good!*.

Étude 4-1: Using `case`

Change the `area/3` function that you wrote in Étude 3-2 so that it uses a `case` instead of pattern matching. Use a guard on the function definition to ensure that the numeric arguments are both greater than zero.

See a suggested solution in Appendix A.

Étude 4-2: Recursion

This is a typical exercise for recursion: finding the greatest common divisor (GCD) of two integers. Instead of giving Euclid's method, we'll do this with a method devised by Edsger W. Dijkstra. The neat part about Dijkstra's method is that you don't need to do any division to find the result. Here is the method.

To find the GCD of integers M and N:

- If M and N are equal, the result is M.
- If M is greater than N, the result is the GCD of M - N and N
- Otherwise M must be less than N, and the result is the GCD of M and N - M.

Write a function `gcd/2` in a module named `dijkstra` that implements the algorithm. This program is an ideal place to use Erlang's `if` construct. Here is some sample output.

```
1> c(dijkstra).
{ok,dijkstra}
2> dijkstra:gcd(12, 8).
4
3> dijkstra:gcd(14, 21).
7
4> dijkstra:gcd(125, 46).
1
5> dijkstra:gcd(120, 36).
12
```

See a suggested solution in Appendix A.

The next two exercises involve writing code to raise a number to an integer power (like 2.5^3 or 4^{-2}) and finding the *n*th root of a number, such as the cube root of 1728 or the fifth root of 3.2.

These capabilities already exist with the `math:pow/2` function, so you may wonder why I'm asking you to re-invent the wheel. The reason is not to replace `math:pow/2`, but to experiment with recursion by writing functions that can be expressed quite nicely that way.

Étude 4-3: Non-Tail Recursive Functions

Create a module named `powers` (no relation to Francis Gary Powers), and write a function named `raise/2` which takes parameters `X` and `N` and returns the value of X^N.

Here's the information you need to know to write the function:

- Any number to the power 0 equals 1.
- Any number to the power 1 is that number itself — that stops the recursion.
- When `N` is positive, X^N is equal to `X` times $X^{(N-1)}$ — there's your recursion.
- When `N` is negative, X^N is equal to `1.0 / X`^{N}

Note that this function is *not* tail recursive. Here is some sample output.

```
1> c(powers).
{ok,powers}
2> powers:raise(5, 1).
5
3> powers:raise(2, 3).
8
4> powers:raise(1.2, 3).
1.728
5> powers:raise(2, 0).
1
6> powers:raise(2, -3).
0.125
```

See a suggested solution in Appendix A.

Étude 4-4: Tail Recursion with an Accumulator

Practice the "accumulator trick." Rewrite the `raise/2` function for `N` greater than zero so that it calls a helper function `raise/3` This new function has `X`, `N`, and an `Accumulator` as its parameters.

Your `raise/2` function will return 1 when `N` is equal to 0, and will return `1.0 / raise(X, -N)` when N is less than zero.

When `N` is greater than zero, `raise/2` will call `raise/3` with arguments `X`, `N`, and 1 as the Accumulator.

The `raise/3` function will return the `Accumulator` when `N` equals 0 (this will stop the recursion).

Otherwise, recursively call `raise/3` with `X`, `N - 1`, and `X` times the `Accumulator` as its arguments.

The `raise/3` function *is* tail recursive.

Étude 4-5: Recursion with a Helper Function

In this exercise, you will add a function `nth_root/2` to the `powers` module. This new function finds the *n*th root of a number, where *n* is an integer. For example, `nth_root(36, 2)` will calculate the square root of 36, and `nth_root(1.728, 3)` will return the cube root of 1.728.

The algorithm used here is the Newton-Raphson method for calculating roots. (See *http://en.wikipedia.org/wiki/Newton%27s_method* for details).

You will need a helper function `nth_root/3`, whose parameters are `X`, `N`, and an approximation to the result, which we will call `A`. `nth_root/3` works as follows:

- Calculate `F` as $(A^N - X)$
- Calculate `Fprime` as $N * A^{(N - 1)}$
- Calculate your next approximation (call it `Next`) as `A - F / Fprime`
- Calculate the change in value (call it `Change`) as the absolute value of `Next - A`
- If the `Change` is less than some limit (say, 1.0e-8), stop the recursion and return `Next`; that's as close to the root as you are going to get.
- Otherwise, call the `nth_root/3` function again with `X`, `N`, and `Next` as its arguments.

For your first approximation, use `X / 2.0`. Thus, your `nth_root/2` function will simply be this:

```
nth_root(X, N) -> nth_root(X, N, X / 2.0)
```

Use `io:format` to show each new approximation as you calculate it. Here is some sample output.

```
1> c(roots).
{ok,roots}
2> roots:nth_root(27, 3).
Current guess is 13.5
Current guess is 9.049382716049383
Current guess is 6.142823558176272
Current guess is 4.333725614685509
Current guess is 3.3683535855517652
Current guess is 3.038813723595138
Current guess is 3.0004936436555805
Current guess is 3.000000081210202
Current guess is 3.000000000000002
3.0
```

See a suggested solution in Appendix A.

CHAPTER 5
Strings

You can learn more about working with strings in Chapter 2 of *Erlang Programming*, Sections 2.11 and 5.4 of *Programming Erlang*, Section 2.2.6 of *Erlang and OTP in Action*, and Chapter 1 of *Learn You Some Erlang For Great Good!*.

Étude 5-1: Validating Input

The Erlang philosophy is "let it crash"; this makes a great deal of sense for a telecommunications system (which is what Erlang was first designed for). Hardware is going to fail. When it does, you just replace it or restart it. The person using the phone system is unaware of this; her phone just continues to work.

This philosophy, however, is not the one you want to employ when you have (atypical for Erlang) programs that ask for user input. You want to those to crash infrequently and catch as many input errors as possible.

In this étude, you will write a module named `ask_area`, which prompts you for a shape and its dimensions, and then returns the area by calling `geom:area/3`, which you wrote in Étude 4-1.

Your module will ask for the first letter of the shape (in either upper or lower case), then the appropriate dimensions. It should catch invalid letters, non-numeric input, and negative numbers as input. Here is some sample output.

```
1> c(ask_area).
{ok,ask_area}
2> c(geom).
{ok,geom}
3> ask_area:area().
R)ectangle, T)riangle, or E)llipse > r
Enter width > 4
```

```
Enter height > 3.7
14.8
4> ask_area:area().
R)ectangle, T)riangle, or E)llipse > T
Enter base > 3
Enter height > 7
10.5
5> ask_area:area().
R)ectangle, T)riangle, or E)llipse > x
Unknown shape x
ok
6> ask_area:area().
R)ectangle, T)riangle, or E)llipse > r
Enter width > -3
Enter height > 4
Both numbers must be greater than or equal to zero.
ok
7> ask_area:area().
R)ectangle, T)riangle, or E)llipse > e
Enter major axis > three
Enter minor axis > 2
Error in first number.
```

Here are the functions that I needed to write in order to make this program work.

`char_to_shape/1`
: Given a character parameter (`R`, `T`, or `E` in either upper or lower case), return an atom representing the specified shape (`rectangle`, `triangle`, `ellipse`, or `unknown` if some other character is entered).

`get_number/1`
: Given a string as a prompt, displays the string "`Enter` *`prompt`* `>` " and returns the number that was input. Your function should accept either integers or floats. Fun fact: `string:to_float/1` *requires* a decimal point; if you just enter input like "`3`", you will receive `{error,no_float}` for your efforts. That means that you should try to convert to float first, and if that fails, try a conversion to integer. It was at this point that I felt like the guy who is beating his head against a wall, and, when asked, "Why are you doing that?" responds, "Because it feels so good when I stop."

`get_dimensions/2`
: Takes two prompts as its parameters (one for each dimension), and calls `get_number/1` twice. Returns a tuple {*`N1, N2`*} with the dimensions.

`calculate/3`
: Takes a shape (as an atom) and two dimensions as its parameters. If the shape is `unknown`, or the first or second dimension isn't numeric, or either number is negative, the function displays an appropriate error message. Otherwise, the function calls `geom:area/3` to calculate the area of the shape.

See a suggested solution in Appendix A.

Étude 5-2: Using the `re` Module

Write a module named `dates` that contains a function `date_parts/1`, which takes a string in ISO date format (`"yyyy-mm-dd"`) and returns a list of integers in the form `[yyyy, mm, dd]`. This function does not need to do any error checking.

You'll use the `re:split/3` function from Erlang's regular expression (`re`) module to accomplish the task. How, you may ask, does that function work? Ask Erlang! The command `erl -man re` will give you the online documentation for the `re` module.

Scroll down the resulting page until you find `split(Subject, RE, Options) -> Split List` and read the examples.

When you write the `-spec` for this function (you *have* been writing documentation for your functions, haven't you?), the type you will use for the parameter is `string()`.

You can see a complete list of the built-in types at *http://www.erlang.org/doc/reference_manual/typespec.html*

Yes, I know this étude seems pointless, but trust me: I'm going somewhere with this. Stay tuned.

See a suggested solution in Appendix A.

CHAPTER 6
Lists

You can learn more about working with lists in Chapter 2 of *Erlang Programming*, Sections 2.10 and 3.5 of *Programming Erlang*, Section 2.2.5 of *Erlang and OTP in Action*, and Chapter 1 of *Learn You Some Erlang For Great Good!*.

Étude 6-1: Recursive Iteration through a List

In a module named `stats`, write a function named `minimum/1`. It takes a list of numbers as its argument and returns the smallest value. This function already exists in the `lists` module (`lists:min/1`), but it's a good exercise in learning about recursion.

Here's the pseudocode.

- Call function `minimum/2`, which takes the list as its first argument and the "smallest number so far" (the *current candidate*) as its second argument. The starting value will be the head of the original number list passed to `minimum/1`.
- When the list passed to `minimum/2` is empty, the final result is the current candidate. This stops the recursion.
- If the list passed to `minimum/2` is not empty, then see if the head of the list is less than the current candidate.
  - If so, call `minimum/2` with the tail of the list as the first argument and the list head (the new "smallest number") as the second argument.
  - If not, call `minimum/2` with the tail of the list as the first argument and the current candidate (still the "smallest number") as the second argument.

Unlike most examples in Introducing Erlang, passing an empty list to this function will make it crash. That's a reasonable thing to do, as an empty list can't really be said to have a minimum value.

```
1> c(stats).
{ok,stats}
2> N = [4, 1, 7, -17, 8, 2, 5].
[4,1,7,-17,8,2,5]
3> stats:minimum(N).
-17
4> stats:minimum([]).
** exception error: bad argument
     in function  hd/1
        called as hd([])
     in call from stats:minimum/1 (stats.erl, line 15)
5> stats:minimum([52.46]).
52.46
```

See a suggested solution in Appendix A.

Étude 6-2: Iteration through Lists (More Practice)

Add two more functions to the `stats` module:

`maximum/1`, which is just the same as `minimum/1`, but don't forget—as I did—to reverse the direction of your test for "smaller" to become a test for "larger." (This function also already exists as `lists:max/1`.)

`range/1`, which takes a list of numbers as its argument and returns a list of two numbers: the minimum and maximum entries in the list.

```
1> c(stats).
{ok,stats}
2> N = [4, 1, 7, -17, 8, 2, 5].
[4,1,7,-17,8,2,5]
3> stats:maximum(N).
8
4> stats:range(N).
[-17,8]
```

See a suggested solution in Appendix A.

Étude 6-3: Accumulating the Sum of a List

Add a function `julian/1` to the `dates` module that you wrote in Étude 5-2. Given a string in ISO format (`"yyyy-mm-dd"`), it returns the Julian date: the day of the year.

Here is some sample output.

```
1> c(dates).
{ok,dates}
2> dates:julian("2012-12-31").
366
3> dates:julian("2013-12-31").
365
4> dates:julian("2012-02-05").
36
5> dates:julian("2013-02-05").
36
6> dates:julian("1900-03-01").
60
7> dates:julian("2000-03-01").
61
126> dates:julian("2013-01-01").
1
```

The `julian/1` function defines a 12-item list called `DaysPerMonth` that contains the number of days in each month, splits the date into the year, month, and day (using the `date_parts/1` function you wrote in Étude 5-2, and then calls helper function `julian/5` (yes, 5).

The `julian/5` function does all of the work. Its arguments are the year, month, day, the list of days per month, and an accumulated total, which starts at zero. `julian/5` takes the head of the days per month list and adds it to the accumulator, and then calls `julian/5` again with the tail of the days per month list and the accumulator value as its last two arguments.

Let's take, as an example, the sequence of calls for April 18, 2013:

```
julian(2013, 4, 18, [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31], 0).
julian(2013, 4, 18, [28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31], 31).
julian(2013, 4, 18, [31, 30, 31, 30, 31, 31, 30, 31, 30, 31], 59).
julian(2013, 4, 18, [30, 31, 30, 31, 31, 30, 31, 30, 31], 90).
```

At this point, the accumulator has all the days up through the beginning of April, so the last call to `julian/5` just adds the 18 remaining days and yields 108 as its result.

You know you are doing the last call when you have "used up" the first *month-1* items in the list of days per month. That happens `when` the month number is greater than `(13 - length(`*`days_per_month_list`*`))`.

Of course, there's still the problem of leap years. For non-leap years, the last call to `julian/5` adds the number of days in the target month. For leap years, the function must add the number of days in the target month plus one—but only if the month is after February.

I'll give you the code for the `is_leap_year/1` function for free; it returns `true` if the given year is a leap year, `false` otherwise.

```
is_leap_year(Year) ->
  (Year rem 4 == 0 andalso Year rem 100 /= 0)
  orelse (Year rem 400 == 0).
```

See a suggested solution in Appendix A.

Interlude: "Mistakes were made."

As I was writing the next two études, I tried, despite the examples in the book, to avoid using `lists:reverse/1`. I thought, "Why *can't* I add items to the end of a list using the cons (vertical bar; `|`) notation?" Here's why.

I decided to do a simple task: take a list of numbers and return a list consisting of the squares of the numbers. I tried adding new items to the end of the list with this code:

```
-module(bad_code).
-export([squares/1]).

squares(Numbers) -> squares(Numbers, []).

squares([], Result) -> Result;

squares([H | T], Result) -> squares(T, [Result | H * H ]).
```

The resulting list was in the correct order, but it was an improper list.

```
1> c(bad_code).
{ok,bad_code}
2> bad_code:squares([9, 4.22, 5]).
[[[[]|81]|17.8084]|25]
```

That didn't work. Wait a minute—the book said that the right hand side of the cons (`|`) operator should be a list. "OK, you want a list?" I thought. "I've got your list right here." (See the last line of the code, where I wrap the new item in square brackets.)

```
squares2(Numbers) -> squares2(Numbers, []).

squares2([], Result) -> Result;

squares2([H | T], Result) -> squares2(T, [Result | [H * H] ]).
```

There. That should do the trick.

```
6> c(bad_code).
{ok,bad_code}
7> bad_code:squares2([9, 4.22, 5]).
[[[[],81],17.8084],25]
```

The result was better, but only slightly better. I didn't have an improper list any more, but now I had a list of list of list of lists. I could fix the problem by changing one line to flatten the final result.

```
squares2([], Result) -> lists:flatten(Result);
```

That worked, but it wasn't a satisfying solution.

- The longer the original list, the more deeply nested the final list would be,
- I still had to call a function from the `lists` module, and
- A look at *http://www.erlang.org/doc/efficiency_guide/listHandling.html* showed that `lists:flatten` is a very expensive operation.

In light of all of this, I decided to use `lists:reverse/1` and write the code to generate a proper, non-nested list.

```
-module(good_code).
-export([correct_squares/1]).

correct_squares(Numbers) -> correct_squares(Numbers, []).

correct_squares([], Result) -> lists:reverse(Result);

correct_squares([H | T], Result) ->
  correct_squares(T, [H * H | Result]).
```

```
1> c(good_code).
{ok,good_code}
2> good_code:correct_squares([9, 4.22, 5]).
[81,17.8084,25]
```

Success at last! The moral of the story?

- RTFM (Read the Fabulous Manual).
- Believe what you read.
- If you don't believe what you read, try it and find out.
- Don't worry if you make this sort of mistake. You won't be the first person to do so, and you certainly won't be the last.
- When using cons, "lists come last."

OK. Back to work.

Étude 6-4: Lists of Lists

Dentists check the health of your gums by checking the depth of the "pockets" at six different locations around each of your 32 teeth. The depth is measured in millimeters. If any of the depths is greater than or equal to four millimeters, that tooth needs attention. (Thanks to Dr. Patricia Lee, DDS, for explaining this to me.)

Your task is to write a module named `teeth` and a function named `alert/1`. The function takes a list of 32 lists of six numbers as its input. If a tooth isn't present, it is repre-

sented by the list `[0]` instead of a list of six numbers. The function produces a list of the tooth numbers that require attention. The numbers must be in ascending order.

Here's a set of pocket depths for a person who has had her upper wisdom teeth, numbers 1 and 16, removed. Just copy and paste it.

```
PocketDepths = [[0], [2,2,1,2,2,1], [3,1,2,3,2,3],
[3,1,3,2,1,2], [3,2,3,2,2,1], [2,3,1,2,1,1],
[3,1,3,2,3,2], [3,3,2,1,3,1], [4,3,3,2,3,3],
[3,1,1,3,2,2], [4,3,4,3,2,3], [2,3,1,3,2,2],
[1,2,1,1,3,2], [1,2,2,3,2,3], [1,3,2,1,3,3], [0],
[3,2,3,1,1,2], [2,2,1,1,3,2], [2,1,1,1,1,2],
[3,3,2,1,1,3], [3,1,3,2,3,2], [3,3,1,2,3,3],
[1,2,2,3,3,3], [2,2,3,2,3,3], [2,2,2,4,3,4],
[3,4,3,3,3,4], [1,1,2,3,1,2], [2,2,3,2,1,3],
[3,4,2,4,4,3], [3,3,2,1,2,3], [2,2,2,2,3,3],
[3,2,3,2,3,2]].
```

And here's the output:

```
1> c(teeth).
{ok,teeth}
2> teeth:alert(PocketDepths).
[9,11,25,26,29]
```

See a suggested solution in Appendix A.

Étude 6-5: Random Numbers; Generating Lists of Lists

How do you think I got the numbers for the teeth in the preceding étude? Do you really think I made up and typed all 180 of them? No, of course not. Instead, I wrote an Erlang program to create the list of lists for me, and that's what you'll do in this étude.

In order to create the data for the teeth, I had to generate random numbers with Erlang's `random` module. Try generating a random number uniformly distributed between 0 and 1.0 by typing this in `erl`:

```
1> random:uniform().
0.4435846174457203
```

Now, exit `erl`, restart, and type the same command again. You'll get the same number. In order to ensure that you get different sets of random numbers, you have to *seed* the random number generator with a three-tuple. The easiest way to get a different seed every time you run the program is to use the `now/0` built-in function, which returns a different three-tuple every time you call it.

```
1> now().
{1356,887000,432535}
2> now().
{1356,887002,15527}
```

```
3> now().
{1356,887003,831752}
```

Exit `erl`, restart, it and try these commands. Do this a couple of times to convince yourself that you really get different random numbers. Don't worry about the `undefined`; that's just Erlang's way of telling you that the random number generator wasn't seeded before.

```
1> random:seed(now()).
undefined
2> random:uniform().
0.27846009966109264
```

If you want to generate a random integer between 1 and `N`, use `uniform/1`; thus `random:uniform(10)` will generate a random integer from 1 to 10.

Functions that use random numbers have side effects; unlike the `sin` or `sqrt` function, which always gives you the same numbers for the same input, functions that use random numbers should always give you different numbers for the same input. Since these functions aren't "pure," it's best to isolate them in a module of their own.

In this étude, create a module named `non_fp`, and write a function `generate_teeth/3`. This function has a string consisting of the characters `T` and `F` for its first argument. A `T` in the string indicates that the tooth is present, and a `F` indicates a missing tooth. In Erlang, a string is a list of characters, so you can treat this string just as you would any other list. Remember to refer to individual characters as `$T` and `$F`.

The second argument is a floating point number between 0 and 1.0 that indicates the probability that a tooth will be a good tooth.

These are the helper functions I needed:

`generate_teeth/3`
: The first two arguments are the same as for `generate_teeth/2`; the third argument is the accumulated list. When the first argument is an empty list, the function yields the reverse of the accumulated list.

  Hint: use pattern matching to figure out whether a tooth is present or not. For a non-present tooth, add `[0]` to the accumulated list; for a tooth that is present, create a list of six numbers by calling `generate_tooth/1` with the probability of a good tooth as its argument.

`generate_tooth/1`
: This generates the list of numbers for a single tooth. It generates a random number between 0 and 1. If that number is less than the probability of a good tooth, it sets the "base depth" to 2, otherwise it sets the base depth to 3.

The function then calls `generate_tooth/3` with the base depth, the number 6, and an empty list as its arguments.

`generate_tooth/3`
: The first argument is the base depth, the second is the number of items left to generate, and the third argument is the accumulated list. When the number of items hits zero, the function is finished. Otherwise, it adds a random integer between -1 and 1 to the base depth, adds it to the accumulated list, and does a recursive call with one less item.

See a suggested solution in Appendix A.

CHAPTER 7

Higher Order Functions and List Comprehensions

You can learn more about working with higher order functions in Chapter 9 of *Erlang Programming*, Section 3.4 of *Programming Erlang*, Section 2.7 of *Erlang and OTP in Action*, and Chapter 6 of *Learn You Some Erlang For Great Good!*. List comprehensions are in Chapter 9 of *Erlang Programming*, Section 3.6 of *Programming Erlang*, Section 2.9 of *Erlang and OTP in Action*, and Chapter 1 of *Learn You Some Erlang For Great Good!*.

Étude 7-1: Simple Higher Order Functions

In calculus, the derivative of a function is "a measure of how a function changes as its input changes" (Wikipedia). For example, if an object is traveling at a constant velocity, that velocity is the same from moment to moment, so the derviative is zero. If an object is falling, its velocity changes a little bit as the object starts falling, and then falls faster and faster as time goes by.

You can calculate the rate of change of a function by calculating: `(F(X + Delta) - F(X)) / Delta`, where `Delta` is the interval between measurements. As Delta approaches zero, you get closer and closer to the true value of the derivative.

Write a module named `calculus` with a function `derivative/2`. The first argument is the function whose derivative you wish to find, and the second argument is the point at which you are measuring the derivative.

What should you use for a value of `Delta`? I used `1.0e-10`, as that is a small number that approaches zero.

Here is some sample output.

```
1> c(calculus).
{ok,calculus}
2> F1 = fun(X) -> X * X end.
#Fun<erl_eval.6.82930912>
3> F1(3).
9
4> calculus:derivative(F1, 3).
6.000000496442226
5> calculus:derivative(fun(X) -> 3 * X * X + 2 * X + 1 end, 5).
32.00000264769187
6> calculus:derivative(fun math:sin/1, 0).
1.0
```

- Line 3 is a test to see if the `F1` function works.
- Line 5 shows that you don't have to assign a function to a variable; you can define the function in line.
- Line 6 shows how to refer to a function in another module. You start with the word `fun` followed by the *`module:function/arity`*.

See a suggested solution in Appendix A.

Étude 7-2: List Comprehensions and Pattern Matching

Is it possible to use pattern matching inside a list comprehension? Try it and find out.

Presume you have this list of people's names, genders, and ages:

```
People = [{"Federico", $M, 22}, {"Kim", $F, 45}, {"Hansa", $F, 30},
{"Tran", $M, 47}, {"Cathy", $F, 32}, {"Elias", $M, 50}].
```

Part One

In `erl` (or in a module, if you prefer), write a list comprehension that creates a list consisting of the names of all males who are over 40. Use pattern matching to separate the tuple into three variables, and two guards to do the tests for age and gender.

Part Two

When you use multiple guards in a list comprehension, you get the moral equivalent of `and` for each condition you are testing. If you want an `or` condition, you must test it explicitly. Write a list comprehension that selects the names of all the people who are male *or* over 40. You will need one guard with an `or`; you may also use `orelse`.

Because `or` has higher priority than comparison operators like `<` and `==`, an expression like `X > 5 or X < 12` will generate an error, as Erlang interprets it to mean `X > (5 or X) < 12`. Use parentheses to force the correct evaluation: `(X > 5) or (X < 12)`. If you use `orelse`, which has a lower priority than the comparison operators, you don't need the parentheses, though it doesn't hurt to have them. Another advantage of `orelse` is that it doesn't do any unnecessary comparisons.

Étude 7-3: Using `lists:foldl/3`

Add `mean/1` and `stdv/1` functions to the `stats` module which you created in Étude 6-2 to calculate the mean and standard deviation for a list of numbers.

```
1> c(stats).
{ok,stats}
2> stats:mean([7, 2, 9]).
6.0
3> stats:stdv([7, 2, 9]).
3.605551275463989
```

The formula for the mean is simple; just add up all the numbers and divide by the number of items in the list (which you may find by using the `length/1` function).Use `lists:foldl/3` to calculate the sum of the items in the list.

The following is the algorithm for calculating the standard deviation. Presume that `N` is the number of items in the list.

1. Add up all the numbers in the list (call this the *sum*).
2. Add the squares of the numbers in the list (call this the *sum of squares*).
3. Multiply `N` times the *sum of squares*.
4. Multiply the *sum* times itself.
5. Subtract the result of step 4 from the result of step 3.
6. Divide the result of step 5 by `N * (N - 1)`.
7. Take the square root of that result.

Thus, if your numbers are 7, 2, and 9, `N` would be three, and you would do these calculations:

- The sum is 7 + 2 + 9, or 18.
- The sum of squares is 49 + 4 + 81, or 134.
- `N` times the sum of squares is 134 * 3, or 402.
- The sum times itself is 18 * 18, or 324.

- 402 - 324 is 78.
- 78 divided by (3 * (3 - 1)) is 78 / 6, or 13.
- The standard deviation is the square root of 13, or 3.606.

In your code, you can do steps three through seven in one arithmetic expression. You'd have variables in your expression rather than constants, of course.

```
math:sqrt((3 * 134 - 18 * 18)/(3 * (3 - 1))
```

Use `lists:foldl/3` to calculate the sum and the sum of squares. Bonus points if you can calculate both of them with one call to `lists:foldl/3`. Hint: the argument for the accumulator doesn't have to be a single number. It can be a list or a tuple.

See a suggested solution in Appendix A.

Étude 7-4: Using `lists:split/2`

Use `erl -man lists` to see how the `lists:split/2` function works, or try the following example and see if you can figure it out. Experiment to see what happens if the first argument is zero.

```
1> lists:split(4, [110, 220, 330, 440, 550, 660]).
{[110,220,330,440],[550,660]}
```

Use `lists:split/2` and `lists:foldl/3` to rewrite the `dates:julian/1` function from Étude 6-3. Hint: you'll use those functions when calculating the total number of days up to (but not including) the month in question.

See a suggested solution in Appendix A.

Étude 7-5: Multiple Generators in List Comprehensions

Back to list comprehensions. You can have more than one generator in a list comprehension. Try this in `erl`:

```
1> [X * Y || X <- [3, 5, 7], Y <- [2, 4, 6]].
[6,12,18,10,20,30,14,28,42]
```

Using what you've learned from this example, write a module named `cards` that contains a function `make_deck/0`. The function will generate a deck of cards as a list 52 tuples in this form:

```
[{"A","Clubs"},
 {"A","Diamonds"},
 {"A","Hearts"},
 {"A","Spades"},
 {2,"Clubs"},
 {2,"Diamonds"},
```

```
{2,"Hearts"},
{2,"Spades"},
...
{"K", "Clubs"},
{"K", "Diamonds"},
{"K", "Hearts"},
{"K", "Spades"}]
```

When you run this function, your output will not show the entire list; it will show something that ends like this. Don't freak out.

```
{7,"Clubs"},
{7,"Diamonds"},
{7,[...]},
{7,...},
{...}|...]
```

If you want to see the full list, use this function.

```
show_deck(Deck) ->
  lists:foreach(fun(Item) -> io:format("~p~n", [Item]) end, Deck).
```

See a suggested solution in Appendix A.

Étude 7-6: Explaining an Algorithm

You need a way to shuffle the deck of cards. This is the code for doing a shuffle, taken from the Literate Programs Wiki.

```
shuffle(List) -> shuffle(List, []).
shuffle([], Acc) -> Acc;
shuffle(List, Acc) ->
  {Leading, [H | T]} = lists:split(random:uniform(length(List)) - 1, List),
  shuffle(Leading ++ T, [H | Acc]).
```

Wait a moment. If I've just given you the code, what's the purpose of this étude? I want you to understand the code. The object of this étude is to write the documentation for the algorithm. If you aren't sure what the code does, try adding some `io:format` statements to see what is happening. If you're totally stuck, see the explanation from Literate Programs site.

See a suggested solution in Appendix A.

CHAPTER 8
Processes

You can learn more about working with simple processes in Chapter 4 of *Erlang Programming*, Chapter 8 of *Programming Erlang*, Section 2.13 of *Erlang and OTP in Action*, and Chapters 10 and 11 of *Learn You Some Erlang For Great Good!*.

Étude 8-1: Using Processes to Simulate a Card Game

There is only one étude for this chapter. You're going to write an Erlang program that lets the computer play the card game of "War" against itself.

The Art of War

These are the rules of the game as condensed from Wikipedia, adapted to two players, and simplified further.

Two players each take 26 cards from a shuffled deck. Each person puts her top card face up on the table. Whoever has the higher value card wins that battle, takes both cards, and puts them at the bottom of her stack. What happens the if the cards have the same value? Then the players go to "war." Each person puts the next two cards from their stack face down in the pile and a third card face up. High card wins, and the winner takes all the cards for the bottom of her stack. If the cards match again, the war continues with another set of three cards from each person. If a person has fewer than three cards when a war happens, he puts in all his cards.

Repeat this entire procedure until one person has all the cards. That player wins the game. In this game, aces are considered to have the highest value, and King > Queen > Jack.

War: What is it good for?

Absolutely nothing. Well, almost nothing. War is possibly the most incredibly inane card game ever invented. It is a great way for children to spend time, and it's perfect as an étude because

- it is naturally implementable as processes (players) passing messages (cards)
- there is no strategy involved in the play, thus allowing you to concentrate on the processes and messages

Pay Now or Pay Later

When you purchase an item, if you pay cash on the spot, you often end up paying less than if you used credit. If you are cooking a meal, getting all of the ingredients collected before you start (pay now) is often less stressful than having to stop and go to the grocery store for items you found out you didn't have (pay later). In most cases, "pay now" ends up being less expensive than "pay later," and that certainly applies to most programming tasks.

So, before you rush off to start writing code, let me give you a word of advice: Don't. Spend some time with paper and pencil, away from the computer, and *design* this program first. This is a non-trivial program, and the "extra" time you spend planning it (pay now) will save you a lot of time in debugging and rewriting (pay later). As someone once told me, "Hours of programming will save you minutes of planning."

Trust me, programs written at the keyboard look like it, and that is not meant as a compliment.

Note: This does not mean that you should never use `erl` or write anything at the keyboard. If you are wondering about how a specific part of Erlang works and need to write a small test program to find out, go ahead and do that right away.

Hint: Do your design on paper. Don't try to keep the whole thing in your head. Draw diagrams. Sometimes a picture or a storyboard of how the messages should flow will clarify your thinking. (If your parents ever asked you, "Do I have to draw you a diagram?", you may now confidently answer "Yes. Please do that. It really helps.")

The Design

When I first started planning this, I was going to have just two processes communicating with one another, as it is in a real game. But let's think about that. There is a slight asymmetry between the players. One person usually brings the cards and suggests playing the game. He shuffles the deck and deals out the cards at the beginning. Once that's done, things even out. The game play itself proceeds almost automatically. Neither

player is in control of the play, yet both of them are. It seems as if there is an implicit, almost telepathic communication between the players. Actually, there are no profound metaphysical issues here. Both players are simultaneously following the same set of rules. And that's the point that bothered me—who makes the "decisions" in the program? I decided to sidestep the issue by introducing a third agent, the "dealer," who is responsible for giving the cards to each player at the start of the game. The dealer then can tell each player to turn over cards, make a decision as to who won, and then tell a particular player to take cards. This simplifies the message flow considerably and also fits in nicely with the OTP concepts of supervisors and servers, covered in Chapter 10 of Introducing Erlang.

In my code, the dealer had to keep track of:

- The process IDs of the players (this was a list)
- The current state of play (see the following)
- Cards received from player 1 for this battle
- Cards received from player 2 for this battle
- The number of players who had given the dealer cards so far (0, 1, or 2)
- The pile of cards in the middle of the table

The dealer spawns the players, and then is in one of the following states. I'm going to anthropomorphize and use "me" to represent the dealer.

Pre-battle
: Tell the players to send me cards. If the pile is empty, then it's a normal battle; give me one card each. If the pile isn't empty, then it's a war; give me three cards.

Await battle
: Wait to receive the cards from the players. Add one to the count every time I get a player's cards. When the count reaches two, I'm ready for…

Check Cards
: If either player has sent me an empty list for their cards, then that player is out of cards, so the other player must be the winner.

  If I really have cards from both players, compare them. If one player is a winner, give that player the pile plus the cards currently in play. If the cards match, add the cards currently in play to the pile, and go back to "Pre-battle" state.

Note that this is my implementation; you may find an entirely different and better way to write the program.

Messages Are Asynchronous

Remember that the order in which a process receives messages may not be the same order in which they were sent. For example, if players Andrea and Bertram have a battle, and Andrea wins, you may be tempted to send these messages:

1. Tell Andrea to pick up the two cards that were in the battle.
2. Tell Andrea to send you a card for the next battle.
3. Tell Bertram to send you a card for the next battle.

This works nicely unless Andrea had just thrown her last card down for that battle and message two arrives *before* message one. Andrea will report that she is out of cards, thus losing the game, even though she's really still in the game with the two cards that she hasn't picked up yet.

Hints for Testing

Modify the `cards` module that you wrote in Étude 7-6 to generate a small deck with, say, only four cards in two suits. If you try to play with a full deck, the game could go on for a very, very long time.

Use plenty of calls to `io:format/2` to see what your code is really doing.

See a suggested solution in Appendix A.

CHAPTER 9
Handling Errors

You can learn more about error handling in Chapters 3 and 17 of *Erlang Programming*, Chapter 4 and Section 18.2 of *Programming Erlang*, Section 2.8 and Chapters 5 and 7 of *Erlang and OTP in Action*, and Chapters 7 and 12 of *Learn You Some Erlang For Great Good!*.

Étude 9-1: `try` and `catch`

Update the `stats` module that you wrote in Étude 7-3 so that it will catch errors in the `minimum/1`, `maximum/1`, `mean/1` and `stdv/1` functions.

Here is some sample output.

```
1> c(stats).
{ok,stats}
2> stats:minimum([]).
{error,badarg}
3> stats:mean([]).
{error,badarith}
4> stats:mean(["123", 456]).
{error,badarith}
5> stats:stdv([]).
{error,badarith}
```

See a suggested solution in Appendix A.

Étude 9-2: Logging Errors

Write a module named `bank` that contains a function `account/1`. The function takes a numeric `Balance`, which gives the current balance in the account in imaginary dollars.

The function will repeatedly ask for a transaction (deposit, withdraw, balance inquiry, or quit). If a deposit or withdrawal, it asks for the amount to deposit or withdraw, and then does that transaction. If a deposit is more than $10,000, the deposit may be subject to hold.

Provide output to the customer, and also use `error_logger` to write to a log file (which, in this case, will go to your terminal). Choose any form of input prompts and feedback and logging messages that you deisre. Handle the following situtations:

- Deposits and withdrawals cannot be negative numbers (error)
- Deposits of $10,000 or more might be subject to hold (warning)
- All other transactions are successful (informational)

Use `get_number/1` from Étude 5-1 to allow either integer or float input.

Here is sample output. Due to Erlang's asynchronous nature, the user prompts and logging are often interleaved in the most inconvenient places.

```
1> c(bank).
{ok,bank}
2> bank:account(2000).
D)eposit, W)ithdraw, B)alance, Q)uit: D
Amount to deposit: 300
Your new balance is 2300
D)eposit, W)ithdraw, B)alance, Q)uit:
=INFO REPORT==== 26-Jan-2013::06:42:52 ===
Successful deposit 300
W
Amount to withdraw: -200
Withdrawals may not be less than zero.
=ERROR REPORT==== 26-Jan-2013::06:42:56 ===
Negative withdrawal amount -200
D)eposit, W)ithdraw, B)alance, Q)uit: D
Amount to deposit: 15000
Your deposit of $15000 may be subject to hold.
=ERROR REPORT==== 26-Jan-2013::06:43:05 ===
Excessive deposit 15000
Your new balance is 17300
D)eposit, W)ithdraw, B)alance, Q)uit: W
Amount to withdraw: 32767
You cannot withdraw more than your current balance of 17300.

=ERROR REPORT==== 26-Jan-2013::06:43:17 ===
Overdraw 32767 from balance 17300
D)eposit, W)ithdraw, B)alance, Q)uit: W
Amount to withdraw: 150.25
Your new balance is 17149.75

=INFO REPORT==== 26-Jan-2013::06:43:29 ===
Successful withdrawal 150.25
```

```
D)eposit, W)ithdraw, B)alance, Q)uit: B
D)eposit, W)ithdraw, B)alance, Q)uit:
=INFO REPORT==== 26-Jan-2013::06:43:35 ===
Balance inquiry 17149.75
X
Unknown command X
D)eposit, W)ithdraw, B)alance, Q)uit: Q
true
```

See a suggested solution in Appendix A.

CHAPTER 10

Storing Structured Data

You can learn more about working with records in Chapter 7 of *Erlang Programming*, Section 3.9 of *Programming Erlang*, Section 2.11 of *Erlang and OTP in Action*, and Chapter 9 of *Learn You Some Erlang For Great Good!*. ETS and DETS are in Chapter 10 of *Erlang Programming*, Chapter 15 of *Programming Erlang*, Section 2.14 and Chapter 6 of *Erlang and OTP in Action*, and Chapter 25 of *Learn You Some Erlang For Great Good!*. Mnesia is covered in Chapter 13 of *Erlang Programming*, Chapter 17 of *Programming Erlang*, Section 2.7 of *Erlang and OTP in Action*, and Chapter 29 of *Learn You Some Erlang For Great Good!*.

Étude 10-1: Using ETS

In honor of Erlang's heritage as a language designed for telephony applications, this étude will set up a small database that keeps track of phone calls.

Part One

Create a file named `phone_records.hrl` that defines a record with these fields:

- Phone number
- Starting date (month, day, and year)
- Starting time (hours, minutes, and seconds)
- End date (month, day, and year)
- End time (hours, minutes, and seconds)

You may name the record whatever you wish, and you may use any field names you wish.

Part Two

In a module named `phone_ets`, create an ETS table for phone calls by reading a file. The function that does this will be named `setup/1`, and its argument will be the name of the file containing the data.

Copy the following text into a file named `call_data.csv` and save the file in the same directory where you did part one.

```
650-555-3326,2013-03-10,09:01:47,2013-03-10,09:05:11
415-555-7871,2013-03-10,09:02:20,2013-03-10,09:05:09
729-555-8855,2013-03-10,09:00:55,2013-03-10,09:02:18
729-555-8855,2013-03-10,09:02:57,2013-03-10,09:03:56
213-555-0172,2013-03-10,09:00:59,2013-03-10,09:03:49
946-555-9760,2013-03-10,09:01:20,2013-03-10,09:03:10
301-555-0433,2013-03-10,09:01:44,2013-03-10,09:04:06
301-555-0433,2013-03-10,09:05:17,2013-03-10,09:07:53
301-555-0433,2013-03-10,09:10:05,2013-03-10,09:13:14
729-555-8855,2013-03-10,09:04:40,2013-03-10,09:07:29
213-555-0172,2013-03-10,09:04:26,2013-03-10,09:06:00
213-555-0172,2013-03-10,09:06:59,2013-03-10,09:10:35
946-555-9760,2013-03-10,09:03:36,2013-03-10,09:04:23
838-555-1099,2013-03-10,09:00:43,2013-03-10,09:02:44
650-555-3326,2013-03-10,09:05:48,2013-03-10,09:09:08
838-555-1099,2013-03-10,09:03:43,2013-03-10,09:06:26
838-555-1099,2013-03-10,09:07:54,2013-03-10,09:10:10
301-555-0433,2013-03-10,09:14:07,2013-03-10,09:15:08
415-555-7871,2013-03-10,09:06:15,2013-03-10,09:09:32
650-555-3326,2013-03-10,09:10:12,2013-03-10,09:13:09
```

So, how do you read a file? Take just the first three lines, and put them into a file called `smallfile.csv`, then do the following commands from `erl`

```
1> {ResultCode, InputFile} = file:open("smallfile.csv", [read]).
{ok,<0.33.0>}
2> io:get_line(InputFile, "").
"650-555-3326,2013-03-10,09:01:47,2013-03-10,09:05:11\n"
3> io:get_line(InputFile, "").
"415-555-7871,2013-03-10,09:02:20,2013-03-10,09:05:09\n"
4> io:get_line(InputFile, "").
"729-555-8855,2013-03-10,09:00:55,2013-03-10,09:02:18\n"
5> io:get_line(InputFile, "").
eof
6> file:open("nosuchfile", [read]).
{error,enoent}
```

In the preceding example, lines 1 through 5 show how to open a file and read it. You can tell you are at the end of file when you get an atom (`eof`) instead of a list (remember, Erlang strings are lists). Line 6 shows what happens if you try to open a file that doesn't exist.

The phone number is the key for this data. Since there are multiple calls per phone number, you will need a `bag` type table. To get the individual items from each line, use `re:split/2`, much as you did in Étude 5-2.

Part Three

Write functions to summarize the number of minutes for a single phone number (`summary/1`) or for all phone numbers. (`summary/0`). These functions return a list of tuples in the form:

`[{`*`phoneNumber1, minutes`*`]},{`*`phoneNumber2, minutes`*`}, …]`

You could write your own code to do time and date calculations to figure out the duration of a phone call, but there's a limit on how much you really want to re-invent the wheel, especially with something as complex as calendar calculations. Consider, for example, a call that begins on 31 December 2013 at 11:58:36 p.m. and ends on 1 January 2014 at 12:14:22 p.m. I don't even want to think about calls that start on 28 February and go to the next day.

So, instead, use the `calendar:datetime_to_gregorian_seconds/1` function to convert a date and time to the number of seconds since the year zero. (I swear I am not making this up.) The argument to this function is a tuple in the form:

```
{{year, month, day}, {hours, minutes, seconds}} %% for example
{{2013, 07, 14}, {14, 49, 21}}
```

Round up any number of seconds to the next minute for ech call. Thus, if a call lasts 4 minutes and 6 seconds, round it up to 5 minutes. Hint: add 59 to the total number of seconds before you `div 60`.

Now might be the time to rewrite part two so that your dates and times are stored in the appropriate format. That way, you do the conversion from string to tuple only once, instead of every time you ask for a summary.

Here is the sample output.

```
1> c(phone_ets).
{ok,phone_ets}
2> phone_ets:setup("call_data.csv").
ok
3> phone_ets:summary("415-555-7871").
[{"415-555-7871",7}]
4> phone_ets:summary().
[{"946-555-9760",3},
 {"415-555-7871",7},
 {"729-555-8855",6},
```

```
{"301-555-0433",12},
{"213-555-0172",9},
{"650-555-3326",11}]
```

See a suggested solution in Appendix A.

Étude 10-2: Using Mnesia

I have good news and bad news. First, the bad news. Mnesia is *not* a relational detabase management system. If you try to use a query list comprehension to join three tables, Erlang will complain that joins with more than two tables are not efficient.

Now, the good news. While trying to find a way around this, I discovered something about query list comprehensions that is really pretty neat, and I'm happy to share it with you.

In this étude, you will use add a table of customer names and use Mnesia query list comprehensions to join data from those tables when producing a summary.

Part One

You will need to add a record for customers to `phone_records.hrl`. Its fields will be:

- Phone Number (this is the key)
- Customer's last name
- Customer's first name
- Customer's middle name
- Rate paid per minute (float)

Again, you may name the record whatever you wish, and you may use any field names you wish.

Part Two

In a module named `phone_mnesia`, create the Mnesia tables for the two files. The function that does this will be named `setup/2`, and its arguments will be the names of the file containing the data.

Use the phone call data from Étude 10-1, and use this data for the customers. Put it in a file named `customer_data.csv` or whatever other name you wish.

```
213-555-0172,Nakamura,Noriko,,0.12
301-555-0433,Ekberg,Erik,Engvald,0.07
415-555-7871,Alvarez,Alberto,Agulto,0.15
650-555-3326,Girard,Georges,Gaston,0.10
729-555-8855,Tran,Truong,Thai,0.09
```

```
838-555-1099,Smith,Samuel,Steven,0.10
946-555-9760,Bobrov,Bogdan,Borisovitch,0.14
```

You could write two functions that all open a file, read data, split it into fields, write the data to the Mnesia table, and then keep going until end-of-file. These would share a lot of common code. Instead, try writing just one function that does the reading, and pass a higher-order function to it to do the appropriate "split-and-write" operation.

When I solved this problem, my `fill_table/5` function took these arguments:

- The name of the table (an atom)
- The name of the file to read (a string)
- The function that adds the data (a higher-order `fun`)
- The `record_info` for the field
- The type of table. The phone call data is a `bag`, the customer data is a `set`.

Part Three

Write a function named `summary/3` that takes a last name, first name, and middle name. It produces a tuple that contains the person's phone number, total number of minutes, and total cost for those minutes.

Here is some sample output.

```
1> c(phone_mnesia).
{ok,phone_mnesia}
2> phone_mnesia:setup("call_data.csv", "customer_data.csv").
{atomic,ok}
3> phone_mnesia:summary("Smith", "Samuel", "Steven").
[{"838-555-1099",9,0.9}]
4> phone_mnesia:summary("Nakamura", "Noriko", "").
[{"213-555-0172",9,1.08}]
```

As promised, here's the good news about query list comprehensions. In this module, you need to access the customer table to match the phone number to the name when collecting the calls for the customer. You also need to access the customer table in order to access the customer's rate per minute. You don't want to have to write the specification for the guards on the customer table twice.

As Introducing Erlang notes, "you can use the `qlc:q` function to hold a list comprehension and the `qlc:e` function to process it." Specifically, the `qlc:q` function returns a *query handle* which you can evaluate and which you can also use in place of a list name in a query list comprehension.

Here's an example. Let's say you have tables of people and their pets. In the `pet` table, the `owner_id` references the `id_number` of someone in the `person` table.

```
-record(person,
  {id_number, name, age, gender, city, amount_owed}).
-record(animal,
  {id_number, name, species, gender, owner_id}).
```

You could do a query like this to find a specific set of people, and then to find information about their pets:

```
get_info() ->
  People = mnesia:transaction(
    fun() -> qlc:e(
      qlc:q( [ P ||
        P <- mnesia:table(person),
        P#person.age >= 21,
        P#person.gender == "M",
        P#person.city == "Podunk"]
        )
      )
    end
  ),

  Pets = mnesia:transaction(
    fun() -> qlc:e(
      qlc:q( [{A#animal.name, A#animal.species, P#person.name} ||
        P <- mnesia:table(person),
        P#person.age >= 21,
        P#person.gender == "M",
        P#person.city == "Podunk",
        A <- mnesia:table(animal),
        A#animal.owner_id == P#person.id_number])
      )
    end
  ),
  [People, Pets].
```

To avoid duplicating the list and guards for the person table, you can make a query list handle for that query and use it again in the animal search. Note that you don't have to be in a transacation to create a query handle, but you must be in a transaction to process it.

```
get_info_easier() ->

  %% "Pre-process" the list comprehension for finding people

  QHandle = qlc:q( [ P ||
    P <- mnesia:table(person),
    P#person.age >= 21,
    P#person.gender == "M",
    P#person.city == "Podunk"]
  ),

  %% Evaluate it to retrieve the people you want
```

```
People = mnesia:transaction(
  fun() -> qlc:e( QHandle ) end
),

%% And use the handle again when retrieving
%% information about their pets

Pets = mnesia:transaction(
  fun() -> qlc:e(
    qlc:q( [{A#animal.name, A#animal.species, P#person.name} ||
      P <- QHandle,
      A <- mnesia:table(animal),
      A#animal.owner_id == P#person.id_number])
    )
  end
),
[People, Pets].
```

See a suggested solution in Appendix A.

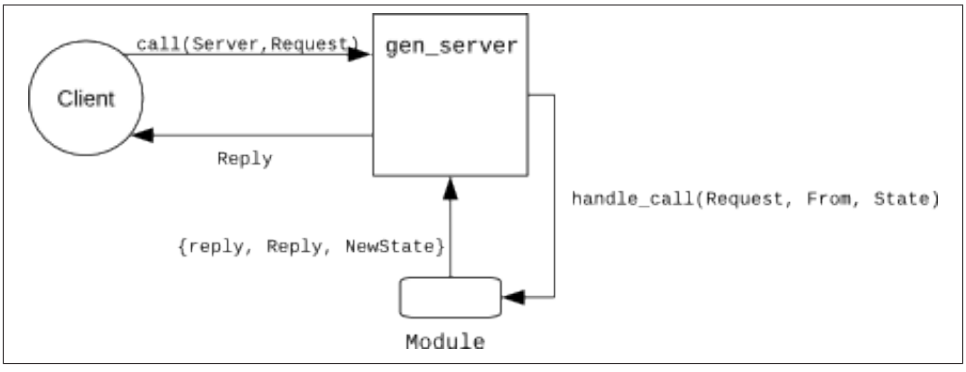


Figure 11-1. Processing a call in `gen_server`

CHAPTER 11
Getting Started with OTP

In order to help me understand how the `gen_server` behavior works, I drew the diagram shown in Figure 11-1.

The client does a `gen_server:call(Server, Request)`. The server will then call the `handle_call/3` function that you have provided in the `Module` that you told `gen_server` to use. `gen_server` will send your module the client's `Request`, an identifier telling who the request is `From`, and the server's current `State`.

Your `handle_call/3` function will fulfill the client's `Request` and send a `{reply, Reply, NewState}` tuple back to the server. It, in turn, will send the `Reply` back to the client, and use the `NewState` to update its state.

In *Introducing Erlang* and in the next two études, the client is you, using the shell. The module that handles the client's call is contained within the same module as the `gen_server` framework, but, as the preceding diagram shows, it does not have to be.

You can learn more about working with OTP basics in Chapters 11 and 12 of *Erlang Programming*, Chapters 16 and 18 of *Programming Erlang*, Chapter 4 of *Erlang and OTP in Action*, and Chapters 14 through 20 of *Learn You Some Erlang For Great Good!*.

Étude 11-1: Get the Weather

In this étude, you will create a weather server using the `gen_server` OTP behavior.This server will handle requests using a four-letter weather station identifier and will return a brief summary of the weather. You may also ask the server for a list of most recently accessed weather stations.

Here is some sample output:

```
1> c(weather).
{ok,weather}
2> weather:start_link().
{ok,<0.42.0>}
3> gen_server:call(weather, "KSJC").
{ok,[{location,"San Jose International Airport, CA"},
     {observation_time_rfc822,"Mon, 18 Feb 2013 13:53:00 -0800"},
     {weather,"Overcast"},
     {temperature_string,"51.0 F (10.6 C)"}]}
4> gen_server:call(weather, "KITH").
{ok,[{location,"Ithaca / Tompkins County, NY"},
     {observation_time_rfc822,"Mon, 18 Feb 2013 16:56:00 -0500"},
     {weather,"A Few Clouds"},
     {temperature_string,"29.0 F (-1.6 C)"}]}
5> gen_server:call(weather,"NONE").
{error,404}
6> gen_server:cast(weather, "").
Most recent requests: ["KITH","KSJC"]
```

Obtaining Weather Data

To retrieve a web page, you must first call `inets:start/0`; you will want to do this in your `init/1` code. Then, simply call `httpc:request(`*`url`*`)`, where *`url`* is a string containing the URL you want. In this case, you will use the server provided by National Oceanic and Atmospheric Administration. This server accepts four-letter weather station codes and returns an XML file summarizing the current weather at that station. You request this data with a URL in the form

`http://w1.weather.gov/xml/current_obs/`*`NNNN`*`.xml`

where *`NNNN`* is the station code.

If the call to `httpc:request/1` fails you will get a tuple of the form {`error,`*`information`*}.

If it succeeds, you will get a tuple in the form:

```
{ok,{{"HTTP/1.1",code,"code message"},
     [{"HTTP header attribute","value"},
      {"Another attribute","another value"}],
     "page contents"}}
```

where *code* is the return code (200 means the page was found, 404 means it's missing, anything else is some sort of error).

So, let's say you have successfully retrieved a station's data. You will then get page content that contains something like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet href="latest_ob.xsl" type="text/xsl"?>
<current_observation version="1.0"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://www.weather.gov/view/current_observation.xsd">
        <credit>NOAA's National Weather Service</credit>
        <credit_URL>http://weather.gov/</credit_URL>
        <image>
                <url>http://weather.gov/images/xml_logo.gif</url>
                <title>NOAA's National Weather Service</title>
                <link>http://weather.gov</link>
        </image>
        <suggested_pickup>15 minutes after the hour</suggested_pickup>
        <suggested_pickup_period>60</suggested_pickup_period>
        <location>San Jose International Airport, CA</location>
        <station_id>KSJC</station_id>
        <latitude>37.37</latitude>
        <longitude>-121.93</longitude>
        <observation_time>Last Updated on Feb 18 2013, 11:53 am PST</observation_time>
  <observation_time_rfc822>Mon, 18 Feb 2013 11:53:00 -0800</observation_time_rfc822>
        <weather>Overcast</weather>
        <temperature_string>50.0 F (10.0 C)</temperature_string>
        <temp_f>50.0</temp_f>
        <temp_c>10.0</temp_c>
        <relative_humidity>77</relative_humidity>
        <wind_string>Calm</wind_string>
        <wind_dir>North</wind_dir>
        <wind_degrees>0</wind_degrees>
        <wind_mph>0.0</wind_mph>
        <wind_kt>0</wind_kt>
        <pressure_string>1017.7 mb</pressure_string>
        <pressure_mb>1017.7</pressure_mb>
        <pressure_in>30.05</pressure_in>
        <dewpoint_string>43.0 F (6.1 C)</dewpoint_string>
        <dewpoint_f>43.0</dewpoint_f>
        <dewpoint_c>6.1</dewpoint_c>
        <visibility_mi>10.00</visibility_mi>
        <icon_url_base>http://forecast.weather.gov/images/wtf/small/</icon_url_base>
        <two_day_history_url>http://www.weather.gov/data/obhistory/KSJC.html</two_day_history_url>
```

```
        <icon_url_name>ovc.png</icon_url_name>
        <ob_url>http://www.weather.gov/data/METAR/KSJC.1.txt</ob_url>
        <disclaimer_url>http://weather.gov/disclaimer.html</disclaimer_url>
        <copyright_url>http://weather.gov/disclaimer.html</copyright_url>
        <privacy_policy_url>http://weather.gov/notice.html</privacy_policy_url>
</current_observation>
```

Parsing the Data

You now have to parse that XML data. Luckily, Erlang comes with the `xmerl_scan:string/1` function, which will parse your XML into a rather imposing-looking tuple. Here is what it looks like for a very simple bit of XML:

```
1> XML = "<pets><cat>Misha</cat><dog>Lady</dog></pets>".
"<pets><cat>Misha</cat><dog>Lady</dog></pets>"
3> Result = xmerl_scan:string(XML).
{{xmlElement,pets,pets,[],
     {xmlNamespace,[],[]},
     [],1,[],
     [{xmlElement,cat,cat,[],
                  {xmlNamespace,[],[]},
                  [{pets,1}],
                  1,[],
                  [{xmlText,[{cat,1},{pets,1}],1,[],"Misha",text}],
                  [],
                  "/home/david/etudes/code/ch11-01",
                  undeclared},
      {xmlElement,dog,dog,[],
                  {xmlNamespace,[],[]},
                  [{pets,1}],
                  2,[],
                  [{xmlText,[{dog,2},{pets,1}],1,[],"Lady",text}],
                  [],undefined,undeclared}],
     [],
     "/home/david/etudes/code/ch11-01",
     undeclared},
     []}
```

Ye cats! How you do work with that?! First, put this at the top of your code so that you can use `xmerl`'s record definitions:

```
-include_lib("xmerl/include/xmerl.hrl").
```

You can see all the details of the records at *http://erlang.googlecode.com/svn-history/r160/trunk/lib/xmerl/include/xmerl.hrl*

Then, copy and paste this into your code. You could figure it out on your own, but that would take away from setting up the server, which is the whole point of this étude.

```
%% Take raw XML data and return a set of {key, value} tuples

analyze_info(WebData) ->
```

```
  %% list of fields that you want to extract
  ToFind = [location, observation_time_rfc822, weather, temperature_string],

  %% get just the parsed data from the XML parse result
  Parsed = element(1, xmerl_scan:string(WebData)),

  %% This is the list of all children under <current_observation>
  Children = Parsed#xmlElement.content,

  %% Find only XML elements and extract their names and their text content.
  %% You need the guard so that you don't process the newlines in the
  %% data (they are XML text descendants of the root element).
  ElementList = [{El#xmlElement.name, extract_text(El#xmlElement.content)}
    || El <- Children, element(1, El) == xmlElement],

  %% ElementList is now a keymap; get the data you want from it.
  lists:map(fun(Item) -> lists:keyfind(Item, 1, ElementList) end, ToFind).


%% Given the parsed content of an XML element, return its first node value
%% (if it's a text node); otherwise return the empty string.

extract_text(Content) ->
  Item = hd(Content),
  case element(1, Item) of
    xmlText -> Item#xmlText.value;
    _ -> ""
  end.
```

Set up a Supervisor

Finally, you can easily crash the server by handing it a number instead of a string for the station code. Set up a supervisor to restart the server when it crashes.

```
1> c(weather_sup).
{ok,weather_sup}
2> {ok, Pid} = weather_sup:start_link().
{ok,<0.38.0>}
3> unlink(Pid).
true
4> gen_server:call(weather, "KGAI").
{ok,[{location,"Montgomery County Airpark, MD"},
     {observation_time_rfc822,"Mon, 18 Feb 2013 17:55:00 -0500"},
     {weather,"Fair"},
     {temperature_string,"37.0 F (3.0 C)"}]}
5> gen_server:call(weather, 1234).
** exception exit: {{badarg,[{erlang,'++',[1234,".xml"],[]},
                             {weather,get_weather,2,[{file,"weather.erl"},{line,43}]},
                             {weather,handle_call,3,[{file,"weather.erl"},{line,23}]},
                             {gen_server,handle_msg,5,
                                         [{file,"gen_server.erl"},{line,588}]},
                             {proc_lib,init_p_do_apply,3,
```

```
                                        [{file,"proc_lib.erl"},{line,227}]}]},
                    {gen_server,call,[weather,1234]}}
     in function  gen_server:call/2 (gen_server.erl, line 180)

=INFO REPORT==== 18-Feb-2013::15:57:19 ===
    application: inets
    exited: stopped
    type: temporary
6>
=ERROR REPORT==== 18-Feb-2013::15:57:19 ===
** Generic server weather terminating
** Last message in was 1234
** When Server state == ["KGAI"]
** Reason for termination ==
** {badarg,[{erlang,'++',[1234,".xml"],[]},
            {weather,get_weather,2,[{file,"weather.erl"},{line,43}]},
            {weather,handle_call,3,[{file,"weather.erl"},{line,23}]},
            {gen_server,handle_msg,5,[{file,"gen_server.erl"},{line,588}]},
            {proc_lib,init_p_do_apply,3,[{file,"proc_lib.erl"},{line,227}]}]}

6> gen_server:call(weather, "KCMI").
{ok,[{location,"Champaign / Urbana, University of Illinois-Willard, IL"},
     {observation_time_rfc822,"Mon, 18 Feb 2013 16:53:00 -0600"},
     {weather,"Overcast and Breezy"},
     {temperature_string,"47.0 F (8.3 C)"}]}
```

See a suggested solution in Appendix A.

Étude 11-2: Wrapper Functions

In the previous étude, you made calls directly to `gen_server`. This is great for experimentation, but in a real application, you do not want other modules to have to know the exact format of the arguments you gave to `gen_server:call/2` or `gen_server:cast/2`. Instead, you provide a "wrapper" function that makes the actual call. In this way, you can change the internal format of your server requests while the interface you present to other users remains unchanged.

In this étude, then, you will provide two wrapper functions `report/1` and `recent/0`. The `report/1` function will take a station name as its argument and do the appropriate `gen_server:call`; the `recent/0` function will do an appropriate `gen_server:cast`. Everything else in your code will remain unchanged. You will, of course, have to add `report/1` and `recent/0` to the `-export` list.

Here's some sample output.

```
1> c(weather).
{ok,weather}
2> weather:start_link().
{ok,<0.45.0>}
3> weather:report("KSJC").
```

```
{ok,[{location,"San Jose International Airport, CA"},
     {observation_time_rfc822,"Tue, 26 Feb 2013 17:53:00 -0800"},
     {weather,"Fair"},
     {temperature_string,"56.0 F (13.3 C)"}]}
4> weather:report("XYXY").
{error,404}
5> weather:report("KCMI").
{ok,[{location,"Champaign / Urbana, University of Illinois-Willard, IL"},
     {observation_time_rfc822,"Tue, 26 Feb 2013 19:53:00 -0600"},
     {weather,"Light Rain Fog/Mist"},
     {temperature_string,"34.0 F (1.1 C)"}]}
6> weather:recent().
Most recent requests: ["KCMI","KSJC"]
```

See a suggested solution in Appendix A.

Étude 11-3: Independent Server and Client

In the previous études, the client and server have been running in the same shell. In this étude, you will make the server available to clients running in other shells.

To make a node available to other nodes, you need to name the node by using the `-name` option when starting `erl`. It looks like this:

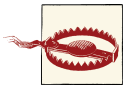
```
michele@localhost $ erl -name serverNode
Erlang R15B02 (erts-5.9.2) [source] [smp:2:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.9.2  (abort with ^G)
(serverNode@localhost.gateway.2wire.net)1>
```

This is a *long name*. You can also set up a node with a short name by using the `-sname` option:

```
michele@localhost $ erl -sname serverNode
Erlang R15B02 (erts-5.9.2) [source] [smp:2:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.9.2  (abort with ^G)
(serverNode@localhost)1>
```



If you set up a node in this way, *any* other node can connect to it and do any shell commands at all. In order to prevent this, you may use the `-setcookie` *`Cookie`* when starting `erl`. Then, only nodes that have the same *Cookie* (which is an atom) can connect to your node.

To connect to a node, use the `net_adm:ping/1` function, and give it the name of the server you want to connect to as its argument. If you connect succesfully, the function will return the atom `pong`; otherwise, it will return `pang`.

Here is an example. First, start a shell with a (very bad) secret cookie:

```
michele@localhost $ erl -sname serverNode -setcookie chocolateChip
Erlang R15B02 (erts-5.9.2) [source] [smp:2:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.9.2  (abort with ^G)
(serverNode@localhost)1>
```

Now, open another terminal window, start a shell with a different cookie, and try to connect to the server node. I have purposely used a different user name to show that this works too.

```
steve@localhost $ erl -sname clientNode -setcookie oatmealRaisin
Erlang R15B02 (erts-5.9.2) [source] [smp:2:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.9.2  (abort with ^G)
(clientNode@localhost)1> net_adm:ping(serverNode@localhost).
pang
```

The server node will detect this attempt and let you know about it:

```
=ERROR REPORT==== 28-Feb-2013::22:41:38 ===
** Connection attempt from disallowed node clientNode@localhost **
```

Quit the client shell, and restart it with a matching cookie, and all will be well.

```
steve@localhost erltest $ erl -sname clientNode -setcookie chocolateChip
Erlang R15B02 (erts-5.9.2) [source] [smp:2:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.9.2  (abort with ^G)
(clientNode@localhost)1> net_adm:ping(serverNode@localhost).
pong
```

To make your weather report server available to other nodes, you need to do these things:

- In the `start_link/0` convenience method, set the first argument to `gen_server:start_link/4` to `{global, ?SERVER}` instead of `{local, ?SERVER}`
- In calls to `gen_server:call/2` and `gen_server:cast/2`, replace the module name `weather` with `{global, weather}`
- Add a `connect/1` function that takes the server node name as its argument. This function will use `net_adm:ping/1` to attempt to contact the server. It provides appropriate feedback when it succeeds or fails.

Here is what it looks like when one user starts the server in a shell.

```
michele@localhost $ erl -sname serverNode -setcookie meteorology
Erlang R15B02 (erts-5.9.2) [source] [smp:2:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.9.2  (abort with ^G)
(serverNode@localhost)1> weather:start_link().
{ok,<0.39.0>}
```

And here's another user in a different shell, calling upon the server.

```
steve@localhost $ erl -sname clientNode -setcookie meteorology
Erlang R15B02 (erts-5.9.2) [source] [smp:2:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.9.2  (abort with ^G)
(clientNode@localhost)1> weather:connect(serverNode@localhost).
Connected to server.
ok
(clientNode@localhost)2> weather:report("KSJC").
{ok,[{location,"San Jose International Airport, CA"},
     {observation_time_rfc822,"Thu, 28 Feb 2013 21:53:00 -0800"},
     {weather,"Fair"},
     {temperature_string,"52.0 F (11.1 C)"}]}
(clientNode@localhost)3> weather:report("KITH").
{ok,[{location,"Ithaca / Tompkins County, NY"},
     {observation_time_rfc822,"Fri, 01 Mar 2013 00:56:00 -0500"},
     {weather,"Light Snow"},
     {temperature_string,"31.0 F (-0.5 C)"}]}
(clientNode@localhost)4> weather:recent().
ok
```

Whoa! What happened to the output from that last call? The problem is that the `weather:recent/0` call does an `io:format/3` call; that output will go to the server shell, since the server is running that code, not the client. Bonus points if you fix this problem by changing `weather:recent/0` from using `gen_server:cast/2` to use `gen_server:call/2` instead to return the recently reported weather stations as its reply.

There's one more question that went through my mind after I implemented my solution: how did I know that the client was calling the `weather` code running on the server and not the `weather` code in its own shell? It was easy to find out: I stopped the server.

```
(serverNode@localhost)2>
User switch command
 --> q
michele@localhost $
```

Then I had the client try to get a weather report.

```
(clientNode@localhost)5> weather:report("KSJC").
** exception exit: {noproc,{gen_server,call,[{global,weather},"KSJC"]}}
     in function  gen_server:call/2 (gen_server.erl, line 180)
```

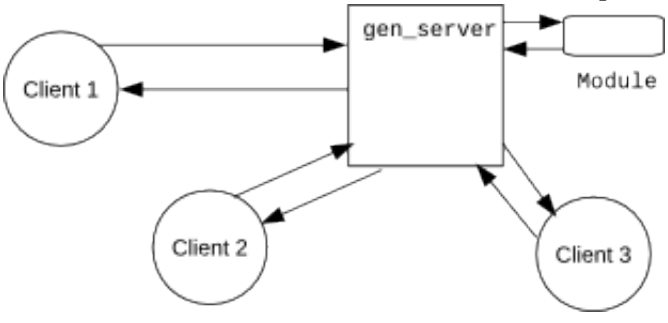
The fact that it failed told me that yes, indeed, the client was getting its information from the server.

See a suggested solution in Appendix A.

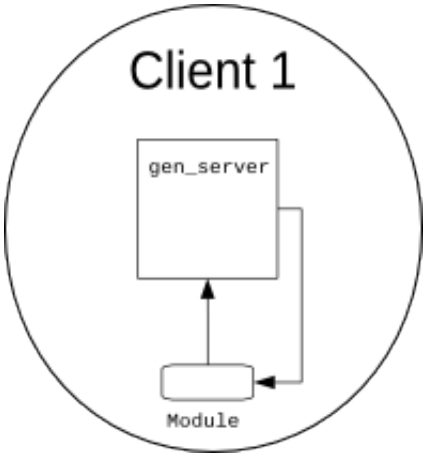
Étude 11-4: Chat Room

In the previous études, the client simply made a call to the server, and didn't do any processing of its own. In this étude, you will create a "chat room" with a chat server and multiple clients, much as you see in Server with multiple clients.

Server with multiple clients.



The interesting part of this program is that the client will *also* be a `gen_server`, as shown in Client as a `gen_server`.



Client as a `gen_server`.

Up until now, you have been using a module name as the first argument to `gen_server:call/2`, and in the previous étude, you used `net_adm:ping/1` to connect to a server.

In this étude, you won't need `net_adm:ping/1`. Instead, you will use a tuple of the form `{Module, Node}` to directly connect to the node you want. So, for example, if you want to make a call to a module named `chatroom` on a node named `lobby@localhost`, you would do something like this:

```
gen_server:call({chatroom, lobby@localhost}, Request)
```

This means you won't need to connect with `net_adm:ping/1`.

Here is my design for the solution. You, of course, may come up with an entirely different and better design.

My solution has two modules, both of which use the `gen_server` behavior.

The `chatroom` Module

The first module, `chatroom`, will keep as its state a list of tuples, one tuple for each person in the chat. Each tuple has the format `{{UserName, UserServer}, Pid}`. The Pid is the one that `gen_server:call` receives in the `From` parameter; it's guaranteed to be unique for each person in chat.

The `handle_call/3` function will accept the following requests.

`{login, UserName, ServerName}`
: Adds the user name, server name, and Pid (which is in the `From` parameter) to the server's state. Don't allow a duplicate user name from the same server.

`logout`
: Removes the user from the state list.

`{say, Text}`
: Sends the given Text to all the other users in the chat room. Use `gen_server:cast/2` to send the message.

`users`
: Returns the list of names and servers for all people currently in the chat room.

`{who, Person, ServerName}`
: Return the profile of the given person/server. (This is "extra credit"; see the following details about the `person` module). It works by calling the `person` module at `ServerName` and giving it a `get_profile` request.

The `person` Module

The other module, `person`, has a `start_link/1` function; the argument is the node name of the chat room server. This will be passed on to the `init/1` function. This is stored in the server's state. I did this because many other calls need to know the chat room server's name, and keeping it in the person's state seemed a reasonable choice.

For extra credit, the state will also include the person's profile, which is a list of `{Key, Value}` tuples.

The `handle_call/3` takes care of these requests:

`get_chat_node`
: Returns the chat node name that's stored in the server's state. (Almost all of the wrapper functions to be described in the following section will need the chat node name.)

`get_profile`
: Returns the profile that's stored in the server's state (extra credit)

`{profile, Key, Value}`
: If the profile already contains the key, replace it with the given value. Otherwise, add the key and value to the profile. Hint: use `lists:keymember/3` and `lists:keyreplace/4`. (extra credit)

Because the chat room server uses `gen_server:cast/2` to send messages to the people in the room, your `handle_cast/3` function will receive messages sent from other users in this form:

```
{message, {FromUser, FromServer}, Text}
```

Wrapper Functions for the `person` module

`get_chat_node()`
: A convenience function to get the name of the chat host node by doing `gen_server:call(person, get_chat_node)`

`login(UserName)`
: Calls the chat room server with a {login, UserName} request. If the user name is an atom, use `atom_to_list/1` to convert it to a string.

`logout()`
: Calls the chat room server with a `logout` request. As you saw in the description of `chatroom`, the server uses the process ID to figure out who should be logged out.

`say(Text)`
: Calls the chat server with a `{say, Text}` request.

`users()`
: Calls the chat server with a `users` request.

`who(UserName, UserNode)`
: Calls the chat server with a `{who, UserName, UserNode}` request to see the profile of the given person. (extra credit)

`profile(Key, Value)`
: A convenience method that calls the `person` module with a `{profile, Key, Value}` request. (extra credit)

Putting it All Together

Here is what the chat room server looks like. The lines beginning with `Recipient list:` are debug output. I have gotten rid of the startup lines from the `erl` command.

```
erl -sname lobby

(lobby@localhost)1> chatroom:start_link().
```

```
{ok,<0.39.0>}
Recipient list: [{"Steve",sales@localhost},{"Michele",marketing@localhost}]
Recipient list: [{"David",engineering@localhost},
                 {"Michele",marketing@localhost}]
Recipient list: [{"David",engineering@localhost},{"Steve",sales@localhost}]
Recipient list: [{"David",engineering@localhost},
                 {"Michele",marketing@localhost}]
```

And here are three other servers talking to one another and setting profile information.

```
erl -sname sales

(sales@localhost)1> person:start_link(lobby@localhost).
Chat node is: lobby@localhost
{ok,<0.39.0>}
(sales@localhost)2> person:login("Steve").
{ok,"Logged in."}
(sales@localhost)3> person:profile(city, "Chicago").
{ok,[{city,"Chicago"}]}
David (engineering@localhost) says: "Hi, everyone."
(sales@localhost)4> person:say("How's things in Toronto, David?").
ok
Michele (marketing@localhost) says: "New product launch is next week."
(sales@localhost)5> person:say("oops, gotta run.").
ok
(sales@localhost)6> person:logout().
ok

erl -sname engineering

(engineering@localhost)1> person:start_link(lobby@localhost).
Chat node is: lobby@localhost
{ok,<0.39.0>}
(engineering@localhost)2> person:login("David").
{ok,"Logged in."}
(engineering@localhost)3> person:profile(city, "Toronto").
{ok,[{city,"Toronto"}]}
(engineering@localhost)4> person:profile(department, "New Products").
{ok,[{department,"New Products"},{city,"Toronto"}]}
(engineering@localhost)5> person:say("Hi, everyone.").
ok
Steve (sales@localhost) says: "How's things in Toronto, David?"
Michele (marketing@localhost) says: "New product launch is next week."
(engineering@localhost)6> person:users().
[{"David",engineering@localhost},
 {"Steve",sales@localhost},
 {"Michele",marketing@localhost}]
Steve (sales@localhost) says: "oops, gotta run."

erl -sname marketing

(marketing@localhost)1> person:start_link(lobby@localhost).
Chat node is: lobby@localhost
{ok,<0.39.0>}
```

```
(marketing@localhost)2> person:login("Michele").
{ok,"Logged in."}
(marketing@localhost)3> person:profile(city, "San Jose").
{ok,[{city,"San Jose"}]}
David (engineering@localhost) says: "Hi, everyone."
Steve (sales@localhost) says: "How's things in Toronto, David?"
(marketing@localhost)4> person:say("New product launch is next week.").
ok
Steve (sales@localhost) says: "oops, gotta run."
(marketing@localhost)5> person:users().
[{"David",engineering@localhost},
 {"Michele",marketing@localhost}]
```

See a suggested solution in Appendix A.

APPENDIX A

Solutions to Études

Here are the solutions that I came up with for the études in this book. Since I was learning Erlang as I wrote them, you may expect some of the code to be naïve in the extreme.

Solution 2-1

Here is a suggested solution for Étude 2-1.

geom.erl

```
-module(geom).
-export([area/2]).

area(L,W) -> L * W.
```

Solution 2-2

Here is a suggested solution for Étude 2-2.

geom.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for calculating areas of geometric shapes.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(geom).
-export([area/2]).

%% @doc Calculates the area of a rectangle, given the
%% length and width. Returns the product
%% of its arguments.
```

```
-spec(area(number(),number()) -> number()).

area(L,W) -> L * W.
```

Solution 2-3

Here is a suggested solution for Étude 2-3.

geom.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for calculating areas of geometric shapes.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(geom).
-export([area/2]).

%% @doc Calculates the area of a rectangle, given the
%% length and width. Returns the product
%% of its arguments.

-spec(area(number(),number()) -> number()).

area(L,W) -> L * W.
```

Solution 3-1

Here is a suggested solution for Étude 3-1.

geom.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for calculating areas of geometric shapes.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(geom).
-export([area/3]).

%% @doc Calculates the area of a shape, given the
%% shape and two of the dimensions. Returns the product
%% of its arguments for a rectangle, one half the
%% product of the arguments for a triangle, and
%% math:pi times the product of the arguments for
%% an ellipse.

-spec(area(atom(), number(),number()) -> number()).

area(rectangle, L,W) -> L * W;
```

```
area(triangle, B, H) -> (B * H) / 2.0;

area(ellipse, A, B) -> math:pi() * A * B.
```

Solution 3-2

Here is a suggested solution for Étude 3-2.

geom.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for calculating areas of geometric shapes.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(geom).
-export([area/3]).

%% @doc Calculates the area of a shape, given the
%% shape and two of the dimensions. Returns the product
%% of its arguments for a rectangle, one half the
%% product of the arguments for a triangle, and
%% math:pi times the product of the arguments for
%% an ellipse. Ensure that both arguments are greater than
%% or equal to zero.

-spec(area(atom(), number(),number()) -> number()).

area(rectangle, L,W) when L >=0, W >= 0 -> L * W;

area(triangle, B, H) when B>= 0, H >= 0 -> (B * H) / 2.0;

area(ellipse, A, B) when A >= 0, B >= 0 -> math:pi() * A * B.
```

Solution 3-3

Here is a suggested solution for Étude 3-3.

geom.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for calculating areas of geometric shapes.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(geom).
-export([area/3]).

%% @doc Calculates the area of a shape, given the
```

```
%% shape and two of the dimensions. Returns the product
%% of its arguments for a rectangle, one half the
%% product of the arguments for a triangle, and
%% math:pi times the product of the arguments for
%% an ellipse. Invalid data returns zero.

-spec(area(atom(), number(),number()) -> number()).

area(rectangle, L,W) when L >=0, W >= 0 -> L * W;

area(triangle, B, H) when B>= 0, H >= 0 -> (B * H) / 2.0;

area(ellipse, A, B) when A >= 0, B >= 0 -> math:pi() * A * B;

area(_, _, _) -> 0.
```

Solution 3-4

Here is a suggested solution for Étude 3-4.

geom.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for calculating areas of geometric shapes.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(geom).
-export([area/1]).

%% @doc Calculates the area of a shape, given a tuple
%% containing a shape and two of the dimensions.
%% Works by calling a private function.

-spec(area({atom(), number(),number()}) -> number()).

area({Shape, Dim1, Dim2}) -> area(Shape, Dim1, Dim2).

%% @doc Returns the product of its arguments for a rectangle,
%% one half the product of the arguments for a triangle,
%% and math:pi times the product of the arguments for
%% an ellipse. Invalid data returns zero.

-spec(area(atom(), number(),number()) -> number()).

area(rectangle, L,W) when L >=0, W >= 0 -> L * W;

area(triangle, B, H) when B>= 0, H >= 0 -> (B * H) / 2.0;

area(ellipse, A, B) when A >= 0, B >= 0 -> math:pi() * A * B;
```

```
area(_, _, _) -> 0.
```

Solution 4-1

Here is a suggested solution for Étude 4-1.

geom.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for calculating areas of geometric shapes.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(geom).
-export([area/3]).

%% @doc Calculates the area of a shape, given the
%% shape and two of the dimensions. Returns the product
%% of its arguments for a rectangle, one half the
%% product of the arguments for a triangle, and
%% math:pi times the product of the arguments for
%% an ellipse.

-spec(area(atom(), number(),number()) -> number()).

area(Shape, A, B) when A >= 0, B >= 0 ->
  case Shape of
    rectangle -> A * B;
    triangle -> (A * B) / 2.0;
    ellipse -> math:pi() * A * B
  end.
```

Solution 4-2

Here is a suggested solution for Étude 4-2.

dijkstra.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Recursive function for calculating GCD
%% of two numbers using Dijkstra's algorithm.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(dijkstra).
-export([gcd/2]).

%% @doc Calculates the greatest common divisor of two
%% integers. Uses Dijkstra's algorithm, which does not
```

```
%% require any division.

-spec(gcd(number(), number()) -> number()).

gcd(M, N) ->
        if
          M == N  -> M;
    M > N -> gcd(M - N, N);
    true -> gcd(M, N - M)
        end.
```

Solution 4-3

Here is a suggested solution for Étude 4-3.

powers.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for raising a number to an integer power
%% and finding the Nth root of a number using Newton's method.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(powers).
-export([raise/2]).

%% @doc Raise a number X to an integer power N.
%% Any number to the power 0 equals 1.
%% Any number to the power 1 is that number itself.
%% When N is positive, X^N is equal to X times X^(N - 1)
%% When N is negative, X^N is equal to 1.0 / X^N

-spec(raise(number(), integer()) -> number()).

raise(_, 0) -> 1;

raise(X, 1) -> X;

raise(X, N) when N > 0 -> X * raise(X, N - 1);

raise(X, N) when N < 0 -> 1 / raise(X, -N).
```

powers_traced.erl

This code contains output that lets you see the progress of the recursion.

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for raising a number to an integer power
%% and finding the Nth root of a number using Newton's method.
%% @copyright 2013 J D Eisenberg
%% @version 0.1
```

```
-module(powers_traced).
-export([raise/2]).

%% @doc Raise a number X to an integer power N.
%% Any number to the power 0 equals 1.
%% Any number to the power 1 is that number itself.
%% When N is positive, X^N is equal to X times X^(N - 1)
%% When N is negative, X^N is equal to 1.0 / X^N

-spec(raise(number(), integer()) -> number()).

raise(_, 0) -> 1;

raise(X, 1) -> X;

raise(X, N) when N > 0 ->
  io:format("Enter X: ~p, N: ~p~n", [X, N]),
  Result = X * raise(X, N - 1),
  io:format("Result is ~p~n", [Result]),
  Result;

raise(X, N) when N < 0 -> 1 / raise(X, -N).
```

Solution 4-4

Here is a suggested solution for Étude 4-4.

powers.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for raising a number to an integer power.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(powers).
-export([raise/2]).

%% @doc Raise a number X to an integer power N.
%% Any number to the power 0 equals 1.
%% Any number to the power 1 is that number itself.
%% When N is positive, X^N is equal to X times X^(N - 1)
%% When N is negative, X^N is equal to 1.0 / X^N

-spec(raise(number(), integer()) -> number()).

raise(_, 0) -> 1;

raise(X, N) when N > 0 ->
  raise(X, N, 1);
```

```
raise(X, N) when N < 0 -> 1 / raise(X, -N).

%% @doc Helper function to raise X to N by passing an Accumulator
%% from call to call.
%% When N is 0, return the value of the Accumulator;
%% otherwise return raise(X, N - 1, X * Accumulator)

raise(_, 0, Accumulator) -> Accumulator;

raise(X, N, Accumulator) ->
  raise(X, N-1, X * Accumulator).
```

powers_traced.erl

This code contains output that lets you see the progress of the recursion.

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for raising a number to an integer power.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(powers_traced).
-export([raise/2]).

%% @doc Raise a number X to an integer power N.
%% Any number to the power 0 equals 1.
%% Any number to the power 1 is that number itself.
%% When N is negative, X^N is equal to 1.0 / X^N
%% When N is positive, call raise/3 with 1 as the accumulator.

-spec(raise(number(), integer()) -> number()).

raise(_, 0) -> 1;

raise(X, N) when N > 0 ->
  raise(X, N, 1);

raise(X, N) when N < 0 -> 1 / raise(X, -N).

%% @doc Helper function to raise X to N by passing an Accumulator
%% from call to call.
%% When N is 0, return the value of the Accumulator;
%% otherwise return raise(X, N - 1, X * Accumulator)

-spec(raise(number(), integer(), number()) -> number()).

raise(_, 0, Accumulator) ->
  io:format("N equals 0."),
  Result = Accumulator,
  io:format("Result is ~p~n", [Result]),
  Result;
```

```
raise(X, N, Accumulator) ->
  io:format("Enter: X is ~p, N is ~p, Accumulator is ~p~n",
    [X, N, Accumulator]),
  Result = raise(X, N-1, X * Accumulator),
  io:format("Result is ~p~n", [Result]),
  Result.
```

Solution 4-5

Here is a suggested solution for Étude 4-5.

powers.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for raising a number to an integer power
%% and finding the Nth root of a number using Newton's method.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(powers).
-export([nth_root/2, raise/2]).

%% @doc Find the nth root of a given number.

-spec(nth_root(number(), integer()) -> number()).

nth_root(X, N) ->
  A = X / 2.0,
  nth_root(X, N, A).

%% @doc Helper function to find an nth_root by passing
%% an approximation from one call to the next.
%% If the difference between current and next approximations
%% is less than 1.0e-8, return the next approximation; otherwise return
%% nth_root(X, N, NextApproximation).

nth_root(X, N, A) ->
  io:format("Current guess is ~p~n", [A]), %% see the guesses converge
  F = raise(A, N) - X,
  Fprime = N * raise(A, N - 1),
  Next = A - F / Fprime,
  Change = abs(Next - A),
  if
    Change < 1.0e-8 -> Next;
    true -> nth_root(X, N, Next)
  end.

%% @doc Raise a number X to an integer power N.
%% Any number to the power 0 equals 1.
%% Any number to the power 1 is that number itself.
%% When N is positive, X^N is equal to X times X^(N - 1)
```

```
%% When N is negative, X^N is equal to 1.0 / X^N

-spec(raise(number(), integer()) -> number()).

raise(_, 0) -> 1;

raise(X, N) when N > 0 ->
  raise(X, N, 1);

raise(X, N) when N < 0 -> 1 / raise(X, -N).

%% @doc Helper function to raise X to N by passing an Accumulator
%% from call to call.
%% When N is 0, return the value of the Accumulator;
%% otherwise return raise(X, N - 1, X * Accumulator)

raise(_, 0, Accumulator) -> Accumulator;

raise(X, N, Accumulator) ->
  raise(X, N-1, X * Accumulator).
```

Solution 5-1

Here is a suggested solution for Étude 5-1.

geom.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for calculating areas of geometric shapes.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(geom).
-export([area/3]).

%% @doc Calculates the area of a shape, given the
%% shape and two of the dimensions. Returns the product
%% of its arguments for a rectangle, one half the
%% product of the arguments for a triangle, and
%% math:pi times the product of the arguments for
%% an ellipse.

-spec(area(atom(), number(), number()) -> number()).

area(Shape, A, B) when A >= 0, B >= 0 ->
  case Shape of
    rectangle -> A * B;
    triangle -> (A * B) / 2.0;
    ellipse -> math:pi() * A * B
  end.
```

ask_area.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions to calculate areas of shape given user input.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(ask_area).
-export([area/0]).

%% @doc Requests a character for the name of a shape,
%% numbers for its dimensions, and calculates shape's area.
%% The characters are R for rectangle, T for triangle,
%% and E for ellipse. Input is allowed in either upper
%% or lower case.

-spec(area() -> number()).

area() ->
  Answer = io:get_line("R)ectangle, T)riangle, or E)llipse > "),
  Shape = char_to_shape(hd(Answer)),
  case Shape of
    rectangle -> Numbers = get_dimensions("width", "height");
    triangle -> Numbers = get_dimensions("base", "height");
    ellipse -> Numbers = get_dimensions("major axis", "minor axis");
    unknown -> Numbers = {error, "Unknown shape " ++ [hd(Answer)]}
  end,

  Area = calculate(Shape, element(1, Numbers), element(2, Numbers)),
  Area.

%% @doc Given a character, returns an atom representing the
%% specified shape (or the atom unknown if a bad character is given).

-spec(char_to_shape(char()) -> atom()).

char_to_shape(Char) ->
  case Char of
    $R -> rectangle;
    $r -> rectangle;
    $T -> triangle;
    $t -> triangle;
    $E -> ellipse;
    $e -> ellipse;
    _ ->  unknown
  end.

%% @doc Present a prompt and get a number from the
%% user. Allow either integers or floats.

-spec(get_number(string()) -> number()).
```

```
get_number(Prompt) ->
  Str = io:get_line("Enter " ++ Prompt ++ " > "),
  {Test, _} = string:to_float(Str),
  case Test of
    error -> {N, _} = string:to_integer(Str);
    _ -> N = Test
  end,
  N.

%% @doc Get dimensions for a shape. Input are the two prompts,
%% output is a tuple {Dimension1, Dimension2}.

-spec(get_dimensions(string(), string()) -> {number(), number()}).

get_dimensions(Prompt1, Prompt2) ->
  N1 = get_number(Prompt1),
  N2 = get_number(Prompt2),
  {N1, N2}.

%% @doc Calculate area of a shape, given its shape and dimensions.
%% Handle errors appropriately.

-spec(calculate(atom(), number(), number()) -> number()).

calculate(unknown, _, Err) -> io:format("~s~n", [Err]);
calculate(_, error, _) -> io:format("Error in first number.~n");
calculate(_, _, error) -> io:format("Error in second number.~n");
calculate(_, A, B) when A < 0; B < 0 ->
  io:format("Both numbers must be greater than or equal to zero~n");
calculate(Shape, A, B) -> geom:area(Shape, A, B).
```

Solution 5-2

Here is a suggested solution for Étude 5-2.

dates.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for splitting a date into a list of
%% year-month-day.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(dates).
-export([date_parts/1]).

%% @doc Takes a string in ISO date format (yyyy-mm-dd) and
%% returns a list of integers in form [year, month, day].

-spec(date_parts(list()) -> list()).
```

```
date_parts(DateStr) ->
  [YStr, MStr, DStr] = re:split(DateStr, "-", [{return, list}]),
  [element(1, string:to_integer(YStr)),
    element(1, string:to_integer(MStr)),
    element(1, string:to_integer(DStr))].
```

Solution 6-1

Here is a suggested solution for Étude 6-1.

stats.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for calculating basic statistics on a list of numbers.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(stats).
-export([minimum/1]).

%% @doc Returns the minimum item in a list of numbers. Fails when given
%% an empty list, as there's nothing reasonable to return.

-spec(minimum(list(number())) -> number()).

minimum(NumberList) ->
  minimum(NumberList, hd(NumberList)).

minimum([], Result) -> Result;

minimum([Head|Tail], Result) ->
  case Head < Result of
    true -> minimum(Tail, Head);
    false -> minimum(Tail, Result)
  end.
```

Solution 6-2

Here is a suggested solution for Étude 6-2.

stats.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for calculating basic statistics on a list of numbers.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(stats).
-export([minimum/1, maximum/1, range/1]).
```

```
%% @doc Returns the minimum item in a list of numbers. Fails when given
%% an empty list, as there's nothing reasonable to return.

-spec(minimum(list(number())) -> number()).

minimum(NumberList) ->
  minimum(NumberList, hd(NumberList)).

minimum([], Result) -> Result;

minimum([Head|Tail], Result) ->
  case Head < Result of
    true -> minimum(Tail, Head);
    false -> minimum(Tail, Result)
  end.

%% @doc Returns the maximum item in a list of numbers. Fails when given
%% an empty list, as there's nothing reasonable to return.

-spec(maximum(list(number())) -> number()).

maximum(NumberList) ->
  maximum(NumberList, hd(NumberList)).

maximum([], Result) -> Result;

maximum([Head|Tail], Result) ->
  case Head > Result of
    true -> maximum(Tail, Head);
    false -> maximum(Tail, Result)
  end.

%% @doc Return the range (maximum and minimum) of a list of numbers
%% as a two-element list.
-spec(range([number()]) -> [number()]).

range(NumberList) -> [minimum(NumberList), maximum(NumberList)].
```

Solution 6-3

Here is a suggested solution for Étude 6-3.

dates.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for splitting a date into a list of
%% year-month-day and finding Julian date.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(dates).
```

```
-export([date_parts/1, julian/1, is_leap_year/1]).

%% @doc Takes a string in ISO date format (yyyy-mm-dd) and
%% returns a list of integers in form [year, month, day].

-spec(date_parts(list()) -> list()).

date_parts(DateStr) ->
  [YStr, MStr, DStr] = re:split(DateStr, "-", [{return, list}]),
  [element(1, string:to_integer(YStr)),
    element(1, string:to_integer(MStr)),
    element(1, string:to_integer(DStr))].

%% @doc Takes a string in ISO date format (yyyy-mm-dd) and
%% returns the day of the year (Julian date).

-spec(julian(string()) -> integer()).

julian(DateStr) ->
  DaysPerMonth = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31],
  [Y, M, D] = date_parts(DateStr),
  julian(Y, M, D, DaysPerMonth, 0).

%% @doc Helper function that recursively accumulates the number of days
%% up to the specified date.

-spec(julian(integer(), integer(), integer(), [integer()], integer) -> integer()).

julian(Y, M, D, MonthList, Total) when M > 13 - length(MonthList) ->
  [ThisMonth|RemainingMonths] = MonthList,
  julian(Y, M, D, RemainingMonths, Total + ThisMonth);

julian(Y, M, D, _MonthList, Total) ->
  case M > 2 andalso is_leap_year(Y) of
    true -> Total + D + 1;
    false -> Total + D
  end.

%% @doc Given a year, return true or false depending on whether
%% the year is a leap year.

-spec(is_leap_year(integer()) -> boolean()).

is_leap_year(Year) ->
  (Year rem 4 == 0 andalso Year rem 100 /= 0)
    orelse (Year rem 400 == 0).
```

Solution 6-4

Here is a suggested solution for Étude 6-4.

teeth.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Show teeth that need attention due to excessive pocket depth.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(teeth).
-export([alert/1]).

%% @doc Create a list of tooth numbers that require attention.

-spec(alert[integer()]) -> [integer()]).

alert(ToothList) -> alert(ToothList, 1, []).

%% @doc Helper function that accumulates the list of teeth needing attention

-spec(alert([integer()], integer(), [integer()]) -> [integer()]).

alert([], _Tooth_number, Result) -> lists:reverse(Result);

alert([Head | Tail ], ToothNumber, Result ) ->
  case stats:maximum(Head) >= 4 of
    true -> alert(Tail, ToothNumber + 1, [ToothNumber | Result]);
    false -> alert(Tail, ToothNumber + 1, Result)
  end.
```

stats.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for calculating basic statistics on a list of numbers.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(stats).
-export([minimum/1, maximum/1, range/1]).

%% @doc Returns the minimum item in a list of numbers. Fails when given
%% an empty list, as there's nothing reasonable to return.

-spec(minimum([number()]) -> number()).

minimum(NumberList) ->
  minimum(NumberList, hd(NumberList)).

minimum([], Result) -> Result;

minimum([Head|Tail], Result) ->
  case Head < Result of
    true -> minimum(Tail, Head);
    false -> minimum(Tail, Result)
```

```
  end.

%% @doc Returns the maximum item in a list of numbers. Fails when given
%% an empty list, as there's nothing reasonable to return.

-spec(maximum([number()]) -> number()).

maximum(NumberList) ->
  maximum(NumberList, hd(NumberList)).

maximum([], Result) -> Result;

maximum([Head|Tail], Result) ->
  case Head > Result of
    true -> maximum(Tail, Head);
    false -> maximum(Tail, Result)
  end.

%% @doc Return the range (maximum and minimum) of a list of numbers
%% as a two-element list.
-spec(range([number()]) -> [number()]).

range(NumberList) -> [minimum(NumberList), maximum(NumberList)].
```

Solution 6-5

Here is a suggested solution for Étude 6-5.

non_fp.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Generate a random set of teeth, with a certain
%% percentage expected to be bad.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(non_fp).
-export([generate_teeth/2, test_teeth/0]).

%% @doc Generate a list of lists, six numbers per tooth, giving random
%% pocket depths. Takes a string where T="there's a tooth there"
%% and F="no tooth"), and a float giving probability that a tooth is good.

-spec(generate_teeth(string(), float()) -> list(list(integer()))).

generate_teeth(TeethPresent, ProbGood) ->
  random:seed(now()),
  generate_teeth(TeethPresent, ProbGood, []).

%% @doc Helper function that adds tooth data to the ultimate result.
```

```
-spec(generate_teeth(string(), float(), [[integer()]]) -> [[integer()]]).

generate_teeth([], _Prob, Result) -> lists:reverse(Result);

generate_teeth([$F | Tail], ProbGood, Result) ->
  generate_teeth(Tail, ProbGood, [[0] | Result]);

generate_teeth([$T | Tail], ProbGood, Result) ->
  generate_teeth(Tail, ProbGood,
  [generate_tooth(ProbGood) | Result]).

-spec(generate_tooth(float()) -> list(integer())).

%% @doc Generates a list of six numbers for a single tooth. Choose a
%% random number between 0 and 1. If that number is less than the probability
%% of a good tooth, it sets the "base depth" to 2, otherwise it sets the base
%% depth to 3.

generate_tooth(ProbGood) ->
  Good = random:uniform() < ProbGood,
  case Good of
    true -> BaseDepth = 2;
    false -> BaseDepth = 3
  end,
  generate_tooth(BaseDepth, 6, []).

%% @doc Take the base depth, add a number in range -1..1 to it,
%% and add it to the list.

generate_tooth(_Base, 0, Result) -> Result;

generate_tooth(Base, N, Result) ->
  [Base + random:uniform(3) - 2 | generate_tooth(Base, N - 1, Result)].

test_teeth() ->
  TList = "FTTTTTTTTTTTTTTFTTTTTTTTTTTTTTT",
  N = generate_teeth(TList, 0.75),
  print_tooth(N).

print_tooth([]) -> io:format("Finished.~n");
print_tooth([H|T]) ->
  io:format("~p~n", [H]),
  print_tooth(T).
```

Solution 7-1

Here is a suggested solution for Étude 7-1.

calculus.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Find the derivative of a function Fn at point X.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(calculus).
-export([derivative/2]).

%% @doc Calculate derivative by classical definition.
%% (Fn(X + H) - Fn(X)) / H

-spec(derivative(function(), float()) -> float()).

derivative(Fn, X) ->
        Delta = 1.0e-10,
        (Fn(X + Delta) - Fn(X)) / Delta.
```

Solution 7-2

Here is a suggested solution for Étude 7-2.

patmatch.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Use pattern matching in a list comprehension.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(patmatch).
-export([older_males/0, older_or_male/0]).

%% @doc Select all males older than 40 from a list of tuples giving
%% name, gender, and age.

-spec(older_males() -> list()).

get_people() ->
  [{"Federico", $M, 22}, {"Kim", $F, 45}, {"Hansa", $F, 30},
  {"Vu", $M, 47}, {"Cathy", $F, 32}, {"Elias", $M, 50}].

older_males() ->
  People = get_people(),
  [Name || {Name, Gender, Age} <- People, Gender == $M, Age > 40].

older_or_male() ->
  People = get_people(),
  [Name || {Name, Gender, Age} <- People, (Gender == $M) orelse (Age > 40)].
```

Solution 7-3

Here is a suggested solution for Étude 7-3.

stats.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for calculating basic statistics on a list of numbers.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(stats).
-export([minimum/1, maximum/1, range/1, mean/1, stdv/1, stdv_sums/2]).

%% @doc Returns the minimum item in a list of numbers. Fails when given
%% an empty list, as there's nothing reasonable to return.

-spec(minimum(list()) -> number()).

minimum(NumberList) ->
  minimum(NumberList, hd(NumberList)).

minimum([], Result) -> Result;

minimum([Head|Tail], Result) ->
  case Head < Result of
    true -> minimum(Tail, Head);
    false -> minimum(Tail, Result)
  end.

%% @doc Returns the maximum item in a list of numbers. Fails when given
%% an empty list, as there's nothing reasonable to return.

-spec(maximum(list()) -> number()).

maximum(NumberList) ->
  maximum(NumberList, hd(NumberList)).

maximum([], Result) -> Result;

maximum([Head|Tail], Result) ->
  case Head > Result of
    true -> maximum(Tail, Head);
    false -> maximum(Tail, Result)
  end.

%% @doc Return the range (maximum and minimum) of a list of numbers
%% as a two-element list.
-spec(range(list()) -> list()).

range(NumberList) -> [minimum(NumberList), maximum(NumberList)].
```

```
%% @doc Return the mean of the list.
-spec(mean(list) -> float()).

mean(NumberList) ->
  Sum = lists:foldl(fun(V, A) -> V + A end, 0, NumberList),
  Sum / length(NumberList).

stdv_sums(Value, Accumulator) ->
  [Sum, SumSquares] = Accumulator,
  [Sum + Value, SumSquares + Value * Value].

stdv(NumberList) ->
  N = length(NumberList),
  [Sum, SumSquares] = lists:foldl(fun stdv_sums/2, [0, 0], NumberList),
  math:sqrt((N * SumSquares - Sum * Sum) / (N * (N - 1))).
```

Solution 7-4

Here is a suggested solution for Étude 7-4.

dates.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for splitting a date into a list of
%% year-month-day and finding Julian date.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(dates).
-export([date_parts/1, julian/1, is_leap_year/1]).

%% @doc Takes a string in ISO date format (yyyy-mm-dd) and
%% returns a list of integers in form [year, month, day].

-spec(date_parts(list()) -> list()).

date_parts(DateStr) ->
  [YStr, MStr, DStr] = re:split(DateStr, "-", [{return, list}]),
  [element(1, string:to_integer(YStr)),
    element(1, string:to_integer(MStr)),
    element(1, string:to_integer(DStr))].

%% @doc Takes a string in ISO date format (yyyy-mm-dd) and
%% returns the day of the year (Julian date).
%% Works by summing the days per month up to, but not including,
%% the month in question, then adding the number of days.
%% If it's a leap year and past February, add a leap day.

-spec(julian(list()) -> integer()).
```

```
julian(DateStr) ->
  DaysPerMonth = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31],
  [Y, M, D] = date_parts(DateStr),
  {Sublist, _} = lists:split(M - 1, DaysPerMonth),
  Total = lists:foldl(fun(V, A) -> V + A end, 0, Sublist),
  case M > 2 andalso is_leap_year(Y) of
    true -> Total + D + 1;
    false -> Total + D
  end.

is_leap_year(Year) ->
  (Year rem 4 == 0 andalso Year rem 100 /= 0)
    orelse (Year rem 400 == 0).
```

Solution 7-5

Here is a suggested solution for Étude 7-5.

cards.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for playing a card game.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(cards).
-export([make_deck/0, show_deck/1]).

%% @doc generate a deck of cards
make_deck() ->
  [{Value, Suit} || Value <- ["A", 2, 3, 4, 5, 6, 7, 8, 9, 10, "J", "Q", "K"],
    Suit <- ["Clubs", "Diamonds", "Hearts", "Spades"]].

show_deck(Deck) ->
  lists:foreach(fun(Item) -> io:format("~p~n", [Item]) end, Deck).
```

Solution 7-6

Here is a suggested solution for Étude 7-6.

cards.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for playing a card game.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(cards).
-export([make_deck/0, shuffle/1]).
```

```
%% @doc generate a deck of cards
make_deck() ->
  [{Value, Suit} || Value <- ["A", 2, 3, 4, 5, 6, 7, 8, 9, 10, "J", "Q", "K"],
    Suit <- ["Clubs", "Diamonds", "Hearts", "Spades"]].

shuffle(List) -> shuffle(List, []).

%% If the list is empty, return the accumulated value.
shuffle([], Acc) -> Acc;

%% Otherwise, find a random location in the list and split the list
%% at that location. Let's say the list has 52 elements and the random
%% location is location 22. The first 22 elements go into Leading, and the
%% last 30 elements go into [H|T]. Thus, H would contain element 23, and
%% T would contain elements 24 through 52.
%%
%% H is the "chosen element". It goes into the accumulator (the shuffled list)
%% and then we call shuffle again with the remainder of the deck: the
%% leading elements and the tail of the split list.

shuffle(List, Acc) ->
  {Leading, [H | T]} = lists:split(random:uniform(length(List)) - 1, List),
  shuffle(Leading ++ T, [H | Acc]).
```

Solution 8-1

Here is a suggested solution for Étude 8-1.

cards.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for playing card games.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(cards).
-export([make_deck/0, shuffle/1]).

%% @doc generate a deck of cards
-type card()::{string()|integer(), string()}.
-spec(make_deck() -> [card()]).

%%make_deck() ->
%%  [{Value, Suit} || Value <- ["A", 2, 3, 4, 5, 6, 7, 8, 9, 10, "J", "Q", "K"],
%%    Suit <- ["Clubs", "Diamonds", "Hearts", "Spades"]].

make_deck() ->
  [{Value, Suit} || Value <- ["A", 2, 3, 4],
    Suit <- ["Clubs", "Diamonds"]].

%% Do a Fisher-Yates shuffle of a deck
```

```
-spec(shuffle([card()])-> [card()]).

shuffle(List) -> shuffle(List, []).

%% If the list is empty, return the accumulated value.
shuffle([], Acc) -> Acc;

%% Otherwise, find a random location in the list and split the list
%% at that location. Let's say the list has 52 elements and the random
%% location is location 22. The first 22 elements go into Leading, and the
%% last 30 elements go into [H|T]. Thus, H would contain element 23, and
%% T would contain elements 24 through 52.
%%
%% H is the "chosen element". It goes into the accumulator (the shuffled list)
%% and then we call shuffle again with the remainder of the deck: the
%% leading elements and the tail of the split list.

shuffle(List, Acc) ->
  {Leading, [H | T]} = lists:split(random:uniform(length(List)) - 1, List),
  shuffle(Leading ++ T, [H | Acc]).
```

game.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Play the card game "war" with two players.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(game).
-export([play_game/0, dealer/0, player/2, value/1]).

%% @doc create a dealer
play_game() ->
  spawn(game, dealer, []).

dealer() ->
  random:seed(now()),
  DealerPid = self(),
  Deck = cards:shuffle(cards:make_deck()),
  {P1Cards, P2Cards} = lists:split(trunc(length(Deck) / 2), Deck),
  io:format("About to spawn players each with ~p cards.~n",
    [trunc(length(Deck) / 2)]),
  P1 = spawn(game, player, [DealerPid, P1Cards]),
  P2 = spawn(game, player, [DealerPid, P2Cards]),
  io:format("Spawned players ~p and ~p~n", [P1, P2]),
  dealer([P1, P2], pre_battle, [], [], 0, []).

%% The dealer has to keep track of the players' process IDs,
%% the cards they have given to the dealer for comparison,
%% how many players have responded (0, 1, or 2), and the pile
%% in the middle of the table in case of a war.
```

```
dealer(Pids, State, P1Cards, P2Cards, Count, Pile) ->
  [P1, P2] = Pids,
  NCards = if
    Pile == []  -> 1;
    Pile /= [] -> 3
  end,
  case State of
    pre_battle ->
      P1 ! {give_cards, NCards},
      P2 ! {give_cards, NCards},
      dealer(Pids, await_battle, P1Cards, P2Cards, Count, Pile);
    await_battle ->
      receive
        {accept, Pid, Data} ->
          NextCount = Count + 1,
          case Pid of
            P1 -> Next_P1Cards = Data, Next_P2Cards = P2Cards;
            P2 -> Next_P1Cards = P1Cards, Next_P2Cards = Data
          end
      end,
      if
        NextCount == 2 -> NextState = check_cards;
        NextCount /= 2 -> NextState = State
      end,
      dealer(Pids, NextState, Next_P1Cards, Next_P2Cards,
        NextCount, Pile);
    check_cards ->
      Winner = game_winner(P1Cards, P2Cards),
      case Winner of
        0 ->
          io:format("Compare ~p to ~p~n", [P1Cards, P2Cards]),
          NewPile = Pile ++ P1Cards ++ P2Cards,
          case battle_winner(P1Cards, P2Cards) of
             0 -> dealer(Pids, pre_battle, [], [], 0, NewPile);
             1 ->
              P1 ! {take_cards, NewPile},
              dealer(Pids, await_confirmation, [], [], 0, []);
             2 ->
              P2 ! {take_cards, NewPile},
              dealer(Pids, await_confirmation, [], [], 0, [])
           end;
        3 ->
          io:format("It's a draw!~n"),
          end_game(Pids);
        _ ->
          io:format("Player ~p wins~n", [Winner]),
          end_game(Pids)
      end;
    await_war->
      io:format("Awaiting war~n");
    await_confirmation ->
      io:format("Awaiting confirmation of player receiving cards~n"),
```

```
      receive
        {confirmed, _Pid, _Data} ->
        dealer(Pids, pre_battle, [], [], 0, [])
      end
  end.

end_game(Pids) ->
  lists:foreach(fun(Process) -> exit(Process, kill) end, Pids),
  io:format("Game finished.~n").

%% Do we have a winner? If both players are out of cards,
%% it's a draw. If one player is out of cards, the other is the winner.

game_winner([], []) -> 3;
game_winner([], _) -> 2;
game_winner(_, []) -> 1;
game_winner(_, _) -> 0.

battle_winner(P1Cards, P2Cards) ->
  V1 = value(hd(lists:reverse(P1Cards))),
  V2 = value(hd(lists:reverse(P2Cards))),
  Winner = if
    V1 > V2 -> 1;
    V2 > V1 -> 2;
    V1 == V2 -> 0
  end,
  io:format("Winner of ~p vs. ~p is ~p~n", [V1, V2, Winner]),
  Winner = Winner.

player(Dealer, Hand) ->
  receive
    {Command, Data} ->
      case Command of
        give_cards ->
          {ToSend, NewHand} = give_cards(Hand, Data),
          io:format("Sending ~p to ~p~n", [ToSend, Dealer]),
          Dealer!{accept, self(), ToSend};
        take_cards ->
          io:format("~p now has ~p (cards)~n", [self(),
            length(Data) + length(Hand)]),
          NewHand = Hand ++ Data,
          Dealer!{confirmed, self(), []}
      end
  end,
  player(Dealer, NewHand).

%% Player gives N cards from current Hand. N is 1 or 3,
%% depending if there is a war or not.
%% If a player is asked for 3 cards but doesn't have enough,
%% give all the cards in the hand.
%% This function returns a tuple: {[cards to send], [remaining cards in hand]}
```

```
give_cards([], _N) -> {[],[]};
give_cards([A], _N) -> {[A],[]};
give_cards([A, B], N) ->
  if
    N == 1 -> {[A], [B]};
    N == 3 -> {[A, B], []}
  end;
give_cards(Hand, N) ->
  if
    N == 1 -> {[hd(Hand)], tl(Hand)};
    N == 3 ->
      [A, B, C | Remainder] = Hand,
      {[A, B, C], Remainder}
  end.

%% @doc Returns the value of a card. Aces are high; K > Q > J
-spec(value({cards:card()}) -> integer()).

value({V, _Suit}) ->
  if
    is_integer(V) -> V;
    is_list(V) ->
      case hd(V) of
        $J -> 11;
        $Q -> 12;
        $K -> 13;
        $A -> 14
      end
  end.
```

Solution 9-1

Here is a suggested solution for Étude 9-1.

stats.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Functions for calculating basic statistics on a list of numbers.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(stats).
-export([minimum/1, maximum/1, range/1, mean/1, stdv/1, stdv_sums/2]).

%% @doc Returns the minimum item in a list of numbers. Uses
%% try/catch to return an error when there's an empty list,
%% as there's nothing reasonable to return.

-spec(minimum(list()) -> number()).

minimum(NumberList) ->
```

```
  try minimum(NumberList, hd(NumberList)) of
    Answer -> Answer
  catch
    error:Error -> {error, Error}
  end.

minimum([], Result) -> Result;

minimum([Head|Tail], Result) ->
  case Head < Result of
    true -> minimum(Tail, Head);
    false -> minimum(Tail, Result)
  end.

%% @doc Returns the maximum item in a list of numbers. Catches
%% errors when given an empty list.

-spec(maximum(list()) -> number()).

maximum(NumberList) ->
  try
    maximum(NumberList, hd(NumberList))
  catch
    error:Error-> {error, Error}
  end.

maximum([], Result) -> Result;

maximum([Head|Tail], Result) ->
  case Head > Result of
    true -> maximum(Tail, Head);
    false -> maximum(Tail, Result)
  end.

%% @doc Return the range (maximum and minimum) of a list of numbers
%% as a two-element list.
-spec(range(list()) -> list()).

range(NumberList) -> [minimum(NumberList), maximum(NumberList)].

%% @doc Return the mean of the list.
-spec(mean(list()) -> float()).

mean(NumberList) ->
  try
    Sum = lists:foldl(fun(V, A) -> V + A end, 0, NumberList),
    Sum / length(NumberList)
  catch
    error:Error -> {error, Error}
  end.

%% @doc Helper function to generate sums and sums of squares
```

```
%% when calculating standard deviation.

-spec(stdv_sums(number(),[number()]) -> [number()]).

stdv_sums(Value, Accumulator) ->
  [Sum, SumSquares] = Accumulator,
  [Sum + Value, SumSquares + Value * Value].

%% @doc Calculate the standard deviation of a list of numbers.

-spec(stdv([number()]) -> float()).

stdv(NumberList) ->
  N = length(NumberList),
  try
    [Sum, SumSquares] = lists:foldl(fun stdv_sums/2, [0, 0], NumberList),
    math:sqrt((N * SumSquares - Sum * Sum) / (N * (N - 1)))
  catch
    error:Error -> {error, Error}
  end.
```

Solution 9-2

Here is a suggested solution for Étude 9-2.

bank.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Implement a bank account that logs its transactions.
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(bank).
-export([account/1]).

-spec(account(pid()) -> number()).

%% @doc create a client and give it the process ID for an account
account(Balance) ->
  Input = io:get_line("D)eposit, W)ithdraw, B)alance, Q)uit: "),
  Action = hd(Input),

  case Action of
    $D ->
      Amount = get_number("Amount to deposit: "),
      NewBalance = transaction(deposit, Balance, Amount);
    $W ->
      Amount = get_number("Amount to withdraw: "),
      NewBalance = transaction(withdraw, Balance, Amount);
    $B ->
      NewBalance = transaction(balance, Balance);
```

```
    $Q ->
      NewBalance = Balance;
    _ ->
      io:format("Unknown command ~c~n", [Action]),
      NewBalance = Balance
  end,
  if
    Action /= $Q ->
    account(NewBalance);
    true -> true
  end.


%% @doc Present a prompt and get a number from the
%% user. Allow either integers or floats.
get_number(Prompt) ->
  Str = io:get_line(Prompt),
  {Test, _} = string:to_float(Str),
  case Test of
    error -> {N, _} = string:to_integer(Str);
    _ -> N = Test
  end,
  N.

transaction(Action, Balance, Amount) ->
  case Action of
    deposit ->
      if
        Amount >= 10000 ->
          error_logger:warning_msg("Excessive deposit ~p~n", [Amount]),
          io:format("Your deposit of $~p may be subject to hold.", [Amount]),
          io:format("Your new balance is ~p~n", [Balance + Amount]),
          NewBalance = Balance + Amount;
        Amount < 0 ->
          error_logger:error_msg("Negative deposit amount ~p~n", [Amount]),
          io:format("Deposits may not be less than zero."),
          NewBalance = Balance;
        Amount >= 0 ->
          error_logger:info_msg("Successful deposit ~p~n", [Amount]),
          NewBalance = Balance + Amount,
          io:format("Your new balance is ~p~n", [NewBalance])
      end;
    withdraw ->
      if
        Amount > Balance ->
          error_logger:error_msg("Overdraw ~p from balance ~p~n", [Amount,
            Balance]),
          io:format("You cannot withdraw more than your current balance of ~p.~n",
            [Balance]),
          NewBalance = Balance;
        Amount < 0 ->
          error_logger:error_msg("Negative withdrawal amount ~p~n", [Amount]),
```

```
            io:format("Withdrawals may not be less than zero."),
            NewBalance = Balance;
          Amount >= 0 ->
            error_logger:info_msg("Successful withdrawal ~p~n", [Amount]),
            NewBalance = Balance - Amount,
            io:format("Your new balance is ~p~n", [NewBalance])
        end
    end,
    NewBalance.

transaction(balance, Balance) ->
    error_logger:info_msg("Balance inquiry ~p~n", [Balance]),
    Balance.
```

Solution 10-1

Here is a suggested solution for Étude 10-1.

phone_records.hrl

```
-record(phone_call,
  {phone_number, start_date, start_time, end_date, end_time}).
```

phone_ets.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Read in a database of phone calls
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(phone_ets).
-export([setup/1, summary/0, summary/1]).
-include("phone_records.hrl").

%% @doc Create an ets table of phone calls from the given file name.

-spec(setup(string()) -> atom()).

setup(FileName) ->

  %% If the table exists, delete it
  case ets:info(call_table) of
    undefined -> false;
    _ -> ets:delete(call_table)
   end,

  %% and create it anew
  ets:new(call_table, [named_table, bag,
    {keypos, #phone_call.phone_number}]),

  {ResultCode, InputFile} = file:open(FileName, [read]),
```

```
  case ResultCode of
    ok -> read_item(InputFile);
    _ -> io:format("Error opening file: ~p~n", [InputFile])
  end.

%% Read a line from the input file, and insert its contents into
%% the call_table. This function is called recursively until end of file

-spec(read_item(file:io_device()) -> atom()).

read_item(InputFile) ->
  RawData = io:get_line(InputFile, ""),
  if
    is_list(RawData) ->
      Data = string:strip(RawData, right, $\n),
      [Number, SDate, STime, EDate, ETime] =
        re:split(Data, ",", [{return, list}]),
      ets:insert(call_table, #phone_call{phone_number = Number,
        start_date = to_date(SDate), start_time = to_time(STime),
        end_date = to_date(EDate), end_time= to_time(ETime)}),
      read_item(InputFile);
    RawData == eof -> ok
  end.

%% @doc Convert a string in form "yyyy-mm-dd" to a tuple {yyyy, mm, dd}
%% suitable for use with the calendar module.

-spec(to_date(string()) -> {integer(), integer(), integer()}).

to_date(Date) ->
  [Year, Month, Day] = re:split(Date, "-", [{return, list}]),
  [{Y, _}, {M, _}, {D, _}] = lists:map(fun string:to_integer/1,
    [Year, Month, Day]),
  {Y, M, D}.

%% @doc Convert a string in form "hh:mm:ss" to a tuple {hh, mm, ss}
%% suitable for use with the calendar module.

-spec(to_time(string()) -> {integer(), integer(), integer()}).

to_time(Time) ->
  [Hour, Minute, Second] = re:split(Time, ":", [{return, list}]),
  [{H, _}, {M, _}, {S, _}] = lists:map(fun string:to_integer/1,
    [Hour, Minute, Second]),
  {H, M, S}.

%% @doc Create a summary of number of minutes used by all phone numbers.

-spec(summary() -> [tuple(string(), integer())]).

summary() ->
  FirstKey = ets:first(call_table),
```

```
    summary(FirstKey, []).

summary(Key, Result) ->
    NextKey = ets:next(call_table, Key),
    case NextKey of
        '$end_of_table' -> Result;
        _ -> summary(NextKey, [hd(summary(Key)) | Result])
    end.

%% @doc Create a summary of number of minutes used by one phone number.

-spec(summary(string()) -> [tuple(string(), integer())]).

summary(PhoneNumber) ->
    Calls = ets:lookup(call_table, PhoneNumber),
    Total = lists:foldl(fun subtotal/2, 0, Calls),
    [{PhoneNumber, Total}].

subtotal(Item, Accumulator) ->
    StartSeconds = calendar:datetime_to_gregorian_seconds(
        {Item#phone_call.start_date, Item#phone_call.start_time}),
    EndSeconds = calendar:datetime_to_gregorian_seconds(
        {Item#phone_call.end_date, Item#phone_call.end_time}),
    Accumulator + ((EndSeconds - StartSeconds + 59) div 60).
```

generate_calls.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Generate a random set of data for phone calls
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(generate_calls).
-export([make_call_list/1, format_date/1, format_time/1]).

make_call_list(N) ->
    Now = calendar:datetime_to_gregorian_seconds({{2013, 3, 10}, {9, 0, 0}}),
    Numbers = [
        {"213-555-0172", Now},
        {"301-555-0433", Now},
        {"415-555-7871", Now},
        {"650-555-3326", Now},
        {"729-555-8855", Now},
        {"838-555-1099", Now},
        {"946-555-9760", Now}
    ],
    CallList = make_call_list(N, Numbers, []),
    {Result, OutputFile} = file:open("call_list.csv", [write]),
    case Result of
        ok -> write_item(OutputFile, CallList);
        error -> io:format("Error: ~p~n", OutputFile)
    end.
```

```
make_call_list(0, _Numbers, Result) -> lists:reverse(Result);

make_call_list(N, Numbers, Result) ->
  Entry = random:uniform(length(Numbers)),
  {Head, Tail} = lists:split(Entry - 1, Numbers),
  {Number, LastCall} = hd(Tail),
  StartCall = LastCall + random:uniform(120) + 20,
  Duration = random:uniform(180) + 40,
  EndCall = StartCall + Duration,
  Item = [Number, format_date(StartCall), format_time(StartCall),
    format_date(EndCall), format_time(EndCall)],
  UpdatedNumbers = Head ++ [{Number, EndCall} | tl(Tail)],
  make_call_list(N - 1, UpdatedNumbers, [Item | Result]).

write_item(OutputFile, []) ->
  file:close(OutputFile);

write_item(OutputFile, [H|T]) ->
  io:format("~s ~s ~s ~s ~s~n", H),
  io:fwrite(OutputFile, "~s,~s,~s,~s,~s~n", H),
  write_item(OutputFile, T).

format_date(GSeconds) ->
  {Date, _Time} = calendar:gregorian_seconds_to_datetime(GSeconds),
  {Y, M, D} = Date,
  lists:flatten(io_lib:format("~4b-~2..0b-~2..0b", [Y, M, D])).

format_time(GSeconds) ->
  {_Date, Time} = calendar:gregorian_seconds_to_datetime(GSeconds),
  {M, H, S} = Time,
  lists:flatten(io_lib:format("~2..0b:~2..0b:~2..0b", [M, H, S])).
```

Solution 10-2

Here is a suggested solution for Étude 10-2.

phone_records.hrl

```
-record(phone_call,
  {phone_number, start_date, start_time, end_date, end_time}).
-record(customer,
  {phone_number, last_name, first_name, middle_name, rate}).
```

phone_mnesia.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Read in a database of phone calls and customers.
%% @copyright 2013 J D Eisenberg
%% @version 0.1
```

```
-module(phone_mnesia).
-export([setup/2, summary/3]).
-include("phone_records.hrl").
-include_lib("stdlib/include/qlc.hrl").

%% @doc Set up Mnesia tables for phone calls and customers
%% given their file names

-spec(setup(string(), string()) -> atom()).

setup(CallFileName, CustomerFileName) ->

  mnesia:create_schema([node()]),
  mnesia:start(),
  mnesia:delete_table(phone_call),
  mnesia:delete_table(customer),

  fill_table(phone_call, CallFileName, fun add_call/1,
    record_info(fields, phone_call), bag),
  fill_table(customer, CustomerFileName, fun add_customer/1,
    record_info(fields, customer), set).

%% @doc Fill the given table with data from given file name.
%% AdderFunction assigns data to fields and writes it to the table;
%% RecordInfo is used when creating the table, as is the TableType.

fill_table(TableName, FileName, AdderFunction, RecordInfo, TableType) ->
  mnesia:create_table(TableName, [{attributes, RecordInfo}, {type, TableType}]),

  {OpenResult, InputFile} = file:open(FileName, [read]),
  case OpenResult of
    ok ->
      mnesia:transaction(
        fun() -> read_file(InputFile, AdderFunction) end);
    _ -> io:format("Error opening file: ~p~n", [FileName])
  end.

%% @doc Read a line from InputFile, and insert its contents into
%% the appropriate table by using AdderFunction.

-spec(read_file(file:io_device(), function()) -> atom()).

read_file(InputFile, AdderFunction) ->
  RawData = io:get_line(InputFile, ""),
  if
    is_list(RawData) ->
      Data = string:strip(RawData, right, $\n),
      ItemList = re:split(Data, ",", [{return, list}]),
      AdderFunction(ItemList),
      read_file(InputFile, AdderFunction);
    RawData == eof -> ok
  end.
```

```
%% Add a phone call record; the data is in an ItemList.

-spec(add_call(list()) -> undefined).

add_call(ItemList) ->
  [Number, SDate, STime, EDate, ETime] = ItemList,
  mnesia:write(#phone_call{phone_number = Number,
        start_date = to_date(SDate), start_time = to_time(STime),
        end_date = to_date(EDate), end_time= to_time(ETime)}).

%% Add a customer record; the data is in an ItemList.

-spec(add_customer(list()) -> undefined).

add_customer(ItemList) ->
  [Phone, Last, First, Middle, Rate] = ItemList,
  mnesia:write(#customer{phone_number = Phone, last_name = Last,
    first_name = First, middle_name = Middle, rate = to_float(Rate)}).

%% @doc Convert a string in form "yyyy-mm-dd" to a tuple {yyyy, mm, dd}
%% suitable for use with the calendar module.

-spec(to_date(string()) -> {integer(), integer(), integer()}).

to_date(Date) ->
  [Year, Month, Day] = re:split(Date, "-", [{return, list}]),
  [{Y, _}, {M, _}, {D, _}] = lists:map(fun string:to_integer/1,
    [Year, Month, Day]),
  {Y, M, D}.

%% @doc Convert a string in form "hh:mm:ss" to a tuple {hh, mm, ss}
%% suitable for use with the calendar module.

-spec(to_time(string()) -> {integer(), integer(), integer()}).

to_time(Time) ->
  [Hour, Minute, Second] = re:split(Time, ":", [{return, list}]),
  [{H, _}, {M, _}, {S, _}] = lists:map(fun string:to_integer/1,
    [Hour, Minute, Second]),
  {H, M, S}.


%% @doc Convenience routine to convert a string to float.
%% In case of an error, return zero.

-spec(to_float(string()) -> float()).

to_float(Str) ->
  {FPart, _} = string:to_float(Str),
  case FPart of
```

```
      error -> 0;
      _ -> FPart
    end.

summary(Last, First, Middle) ->

  QHandle = qlc:q([Customer ||
    Customer <- mnesia:table(customer),
    Customer#customer.last_name == Last,
    Customer#customer.first_name == First,
    Customer#customer.middle_name == Middle ]),

  {_Result, [ThePerson|_]} =
    mnesia:transaction(fun() -> qlc:e(QHandle) end),

  {_Result, Calls} = mnesia:transaction(
    fun() ->
       qlc:e(
        qlc:q( [Call ||
          Call <- mnesia:table(phone_call),
          QCustomer <- QHandle,
          QCustomer#customer.phone_number == Call#phone_call.phone_number
        ]
        )
      )
    end
  ),

  TotalMinutes = lists:foldl(fun subtotal/2, 0, Calls),

  [{ThePerson#customer.phone_number,
    TotalMinutes, TotalMinutes * ThePerson#customer.rate}].

subtotal(Item, Accumulator) ->
  StartSeconds = calendar:datetime_to_gregorian_seconds(
    {Item#phone_call.start_date, Item#phone_call.start_time}),
  EndSeconds = calendar:datetime_to_gregorian_seconds(
    {Item#phone_call.end_date, Item#phone_call.end_time}),
  Accumulator + ((EndSeconds - StartSeconds + 59) div 60).
```

pet_records.hrl

```
-record(person,
  {id_number, name, age, gender, city, amount_owed}).
-record(animal,
  {id_number, name, species, gender, owner_id}).
```

pet_mnesia.erl

```
%% @author J D Eisenberg <jdavid.eisenberg@gmail.com>
%% @doc Read in a database of people and their pets
%% appointments.
```

```
%% @copyright 2013 J D Eisenberg
%% @version 0.1

-module(pet_mnesia).
-export([setup/2, get_info/0, get_info_easier/0]).
-include("pet_records.hrl").
-include_lib("stdlib/include/qlc.hrl").

%% @doc Set up Mnesia tables for phone calls and customers
%% given their file names

-spec(setup(string(), string()) -> atom()).

setup(PersonFileName, AnimalFileName) ->

  mnesia:create_schema([node()]),
  mnesia:start(),
  mnesia:delete_table(person),
  mnesia:delete_table(animal),

  fill_table(person, PersonFileName, fun add_person/1,
    record_info(fields, person), set),
  fill_table(animal, AnimalFileName, fun add_animal/1,
    record_info(fields, animal), set).

%% @doc Fill the given table with data from given file name.
%% AdderFunction assigns data to fields and writes it to the table;
%% RecordInfo is used when creating the table, as is the TableType.

fill_table(TableName, FileName, AdderFunction, RecordInfo, TableType) ->
  mnesia:create_table(TableName, [{attributes, RecordInfo}, {type, TableType}]),

  {OpenResult, InputFile} = file:open(FileName, [read]),
  case OpenResult of
    ok ->
      TransResult = mnesia:transaction(
        fun() -> read_file(InputFile, AdderFunction) end),
        io:format("Transaction result ~p~n", [TransResult]);
    _ -> io:format("Error opening file: ~p~n", [FileName])
  end.

%% @doc Read a line from InputFile, and insert its contents into
%% the appropriate table by using AdderFunction.

-spec(read_file(file:io_device(), function()) -> atom()).

read_file(InputFile, AdderFunction) ->
  RawData = io:get_line(InputFile, ""),
  if
    is_list(RawData) ->
      Data = string:strip(RawData, right, $\n),
      ItemList = re:split(Data, ",", [{return, list}]),
```

```
      AdderFunction(ItemList),
      read_file(InputFile, AdderFunction);
    RawData == eof -> ok
  end.


%% Add a person record; the data is in an ItemList.

-spec(add_person(list()) -> undefined).

add_person(ItemList) ->
  [Id, Name, Age, Gender, City, Owed] = ItemList,
  mnesia:write(#person{id_number = to_int(Id), name = Name,
    age = to_int(Age), gender = Gender, city = City,
    amount_owed = to_float(Owed)}).

%% Add an animal record; the data is in an ItemList.

-spec(add_animal(list()) -> undefined).

add_animal(ItemList) ->
  [Id, Name, Species, Gender, Owner] = ItemList,
  mnesia:write(#animal{id_number = to_int(Id),
    name = Name, species = Species, gender = Gender,
    owner_id = to_int(Owner)}).

%% @doc Convenience routine to convert a string to integer.
%% In case of an error, return zero.

-spec(to_int(string()) -> integer()).

to_int(Str) ->
  {IPart, _} = string:to_integer(Str),
  case IPart of
    error -> 0;
    _ -> IPart
  end.

%% @doc Convenience routine to convert a string to float.
%% In case of an error, return zero.

-spec(to_float(string()) -> float()).

to_float(Str) ->
  {FPart, _} = string:to_float(Str),
  case FPart of
    error -> 0;
    _ -> FPart
  end.

get_info() ->
  People = mnesia:transaction(
```

```
    fun() -> qlc:e(
      qlc:q( [ P ||
        P <- mnesia:table(person),
        P#person.age >= 21,
        P#person.gender == "M",
        P#person.city == "Podunk"]
        )
      )
    end
  ),

  Pets = mnesia:transaction(
    fun() -> qlc:e(
      qlc:q( [{A#animal.name, A#animal.species, P#person.name} ||
        P <- mnesia:table(person),
        P#person.age >= 21,
        P#person.gender == "M",
        P#person.city == "Podunk",
        A <- mnesia:table(animal),
        A#animal.owner_id == P#person.id_number])
      )
    end
  ),
  [People, Pets].

get_info_easier() ->

  %% "Pre-process" the list comprehension for finding people

  QHandle = qlc:q( [ P ||
    P <- mnesia:table(person),
    P#person.age >= 21,
    P#person.gender == "M",
    P#person.city == "Podunk"]
  ),

  %% Evaluate it to retrieve the people you want

  People = mnesia:transaction(
    fun() -> qlc:e( QHandle ) end
  ),

  %% And use the handle again when retrieving
  %% information about their pets

  Pets = mnesia:transaction(
    fun() -> qlc:e(
      qlc:q( [{A#animal.name, A#animal.species, P#person.name} ||
        P <- QHandle,
        A <- mnesia:table(animal),
        A#animal.owner_id == P#person.id_number])
      )
```

```
    end
),
[People, Pets].
```

Solution 11-1

Here is a suggested solution for Étude 11-1.

weather.erl

```
-module(weather).
-behaviour(gen_server).
-include_lib("xmerl/include/xmerl.hrl").
-export([start_link/0]). % convenience call for startup
-export([init/1,
         handle_call/3,
         handle_cast/2,
         handle_info/2,
         terminate/2,
         code_change/3]). % gen_server callbacks
-define(SERVER, ?MODULE). % macro that just defines this module as server

%%% convenience method for startup
start_link() ->
        gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

%%% gen_server callbacks
init([]) ->
  inets:start(),
  {ok, []}.

handle_call(Request, _From, State) ->
  {Reply, NewState} = get_weather(Request, State),
  {reply, Reply, NewState}.

handle_cast(_Message, State) ->
  io:format("Most recent requests: ~p\n", [State]),
  {noreply, State}.

handle_info(_Info, State) ->
  {noreply, State}.

terminate(_Reason, _State) ->
  inets:stop(),
  ok.

code_change(_OldVsn, State, _Extra) ->
  {ok, State}.

%%% Internal functions
```

```
%% Given a 4-letter station code as the Request, return its basic
%% weather information as a {key,value} list. If successful, add the
%% station name to the State, which will keep track of recently-accessed
%% weather stations.

get_weather(Request, State) ->
  URL = "http://w1.weather.gov/xml/current_obs/" ++ Request ++ ".xml",
  {Result, Info} = httpc:request(URL),
  case Result of
    error -> {{Result, Info}, State};
    ok ->
      {{_Protocol, Code, _CodeStr}, _Attrs, WebData} = Info,
      case Code of
        404 ->
          {{error, 404}, State};
        200 ->
          Weather = analyze_info(WebData),
          {{ok, Weather}, [Request | lists:sublist(State, 10)]}
      end
  end.

%% Take raw XML data and return a set of {key, value} tuples

analyze_info(WebData) ->
  %% list of fields that you want to extract
  ToFind = [location, observation_time_rfc822, weather, temperature_string],

  %% get just the parsed data from the XML parse result
  Parsed = element(1, xmerl_scan:string(WebData)),

  %% This is the list of all children under <current_observation>
  Children = Parsed#xmlElement.content,

  %% Find only XML elements and extract their names and their text content.
  %% You need the guard so that you don't process the newlines in the
  %% data (they are XML text descendants of the root element).
  ElementList = [{El#xmlElement.name, extract_text(El#xmlElement.content)}
    || El <- Children, element(1, El) == xmlElement],

  %% ElementList is now a keymap; get the data you want from it.
  lists:map(fun(Item) -> lists:keyfind(Item, 1, ElementList) end, ToFind).


%% Given the parsed content of an XML element, return its first node value
%% (if it's a text node); otherwise return the empty string.

extract_text(Content) ->
  Item = hd(Content),
  case element(1, Item) of
    xmlText -> Item#xmlText.value;
    _ -> ""
  end.
```

weather_sup.erl

```
-module(weather_sup).
-behaviour(supervisor).
-export([start_link/0]). % convenience call for startup

-export([init/1]). % supervisor calls
-define(SERVER, ?MODULE).


%%% convenience method for startup
start_link() ->
  supervisor:start_link({local, ?SERVER}, ?MODULE, []).

%%% supervisor callback
init([]) ->
    RestartStrategy = one_for_one,
    MaxRestarts = 1, % one restart every
    MaxSecondsBetweenRestarts = 5, % five seconds

    SupFlags = {RestartStrategy, MaxRestarts, MaxSecondsBetweenRestarts},

    Restart = permanent, % or temporary, or transient
    Shutdown = 2000, % milliseconds, could be infinity or brutal_kill
    Type = worker, % could also be supervisor

    Weather = {weather, {weather, start_link, []},
                    Restart, Shutdown, Type, [weather]},

    {ok, {SupFlags, [Weather]}}.
```

Solution 11-2

Here is a suggested solution for Étude 11-2. Since the bulk of the code is identical to the code in the previous étude, the only code shown here is the revised `-export` list and the added functions.

weather.erl

```
-export([report/1, recent/0]). % wrapper functions

%% Wrapper to hide internal details when getting a weather report
report(Station) ->
  gen_server:call(?SERVER, Station).

%% Wrapper to hide internal details when getting a list of recently used
%% stations.
recent() ->
  gen_server:cast(?SERVER, "").
```

Solution 11-3

Here is a suggested solution for Étude 11-3. Since the bulk of the code is identical to the previous étude, the only code shown here is the added and revised code.

```
%% @doc Connect to a named server
connect(ServerName) ->
  Result = net_adm:ping(ServerName),
  case Result of
    pong -> io:format("Connected to server.~n");
    pang -> io:format("Cannot connect to ~p.~n", [ServerName])
  end.

%% Wrapper to hide internal details when getting a weather report
report(Station) ->
  gen_server:call({global, weather}, Station).

%% Wrapper to hide internal details when getting a list of recently used
%% stations.
recent() ->
  gen_server:call({global,weather}, recent).

%%% convenience method for startup
start_link() ->
  gen_server:start_link({global, ?SERVER}, ?MODULE, [], []).

%%% gen_server callbacks
init([]) ->
  inets:start(),
  {ok, []}.

handle_call(recent, _From, State) ->
  {reply, State, State};
handle_call(Request, _From, State) ->
  {Reply, NewState} = get_weather(Request, State),
  {reply, Reply, NewState}.

handle_cast(_Message, State) ->
  io:format("Most recent requests: ~p\n", [State]),
  {noreply, State}.
```

Solution 11-4

Here is a suggested solution for Étude 11-4.

`chatroom.erl`

```
-module(chatroom).
-behaviour(gen_server).
-export([start_link/0]). % convenience call for startup
-export([init/1,
```

```
         handle_call/3,
         handle_cast/2,
         handle_info/2,
         terminate/2,
         code_change/3]). % gen_server callbacks

-define(SERVER, ?MODULE). % macro that defines this module as the server

% The server state consists of a list of tuples for each person in chat.
% Each tuple has the format {{UserName, UserServer}, PID of person}

%%% convenience method for startup
start_link() ->
  gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

%%% gen_server callbacks
init([]) ->
  {ok, []}.

%% Check to see if a user name/server pair is unique;
%% if so, add it to the server's state

handle_call({login, UserName, ServerRef}, From, State) ->
  {FromPid, _FromTag} = From,
  case lists:keymember({UserName, ServerRef}, 1, State) of
    true ->
      NewState = State,
      Reply = {error, "User " ++ UserName ++ " already in use."};
    false ->
      NewState = [{{UserName, ServerRef}, FromPid} | State],
      Reply = {ok, "Logged in."}
  end,
  {reply, Reply, NewState};

%% Log out the person sending the message, but only
%% if they're logged in already.

handle_call({logout}, From, State) ->
  {FromPid, _FromTag} = From,
  case lists:keymember(FromPid, 2, State) of
    true ->
      NewState = lists:keydelete(FromPid, 2, State),
      Reply  = {ok, logged_out};
    false ->
      NewState = State,
      Reply = {error, not_logged_in}
  end,
  {reply, Reply, NewState};

%% When receiving a message from a person, use the From PID to
%% get the user's name and server name from the chatroom server state.
%% Send the message via a "cast" to everyone who is NOT the sender.
```

```
handle_call({say, Text}, From, State) ->
  {FromPid, _FromTag} = From,

  case lists:keymember(FromPid, 2, State) of
    true ->
    {value, {{SenderName, SenderServer}, _}} =
      lists:keysearch(FromPid, 2, State),

    % For debugging: get the list of recipients.
    RecipientList = [{RecipientName, RecipientServer} ||
      {{RecipientName, RecipientServer}, _} <- State,
      {RecipientName, RecipientServer} /= {SenderName, SenderServer}],
    io:format("Recipient list: ~p~n", [RecipientList]),

    [gen_server:cast({person, RecipientServer},
      {message, {SenderName, SenderServer}, Text}) ||
      {{RecipientName, RecipientServer}, _} <- State,
     RecipientName /= SenderName];

    false -> ok
  end,
  {reply, ok, State};

%% Get the state of another person and return it to the asker

handle_call({who, _Person, ServerRef}, _From, State) ->
  Reply = gen_server:call({person, ServerRef}, get_profile),
  {reply, Reply, State};

%% Return a list of all users currently in the chat room

handle_call(users, _From, State) ->
  UserList = [{UserName, UserServer} ||
    {{UserName, UserServer}, _} <- State],
  {reply, UserList, State};

handle_call(Request, _From, State) ->
  {ok, {error, "Unhandled Request", Request}, State}.

handle_cast(_Request, State) ->
  {noreply, State}.

handle_info(Info, State) ->
  io:format("Received unknown message ~p~n", [Info]),
  {noreply, State}.

terminate(_Reason, _State) ->
  ok.

code_change(_OldVsn, State, _Extra) ->
  {ok, State}.
```

```
%%% Internal functions
```

person.erl

```
-module(person).
-behaviour(gen_server).
-export([start_link/1]). % convenience call for startup
-export([init/1,
         handle_call/3,
         handle_cast/2,
         handle_info/2,
         terminate/2,
         code_change/3]). % gen_server callbacks

-record(state, {chat_node, profile}).

% internal functions
-export([login/1, logout/0, say/1, users/0, who/2, profile/2]).

-define(CLIENT, ?MODULE). % macro that defines this module as the client

%%% convenience method for startup
start_link(ChatNode) ->
  gen_server:start_link({local, ?CLIENT}, ?MODULE, ChatNode, []).

init(ChatNode)->
  io:format("Chat node is: ~p~n", [ChatNode]),
  {ok, #state{chat_node=ChatNode, profile=[]}}.

%% The server is asked to either:
%% a) return the chat host name from the state,
%% b) return the user profile
%% c) update the user profile

handle_call(get_chat_node, _From, State) ->
  {reply, State#state.chat_node, State};

handle_call(get_profile, _From, State) ->
  {reply, State#state.profile, State};

handle_call({profile, Key, Value}, _From, State) ->
  case lists:keymember(Key, 1, State#state.profile) of
    true -> NewProfile = lists:keyreplace(Key, 1, State#state.profile,
      {Key, Value});
    false -> NewProfile = [{Key, Value} | State#state.profile]
  end,
  {reply, NewProfile,
    #state{chat_node = State#state.chat_node, profile=NewProfile}};

handle_call(_, _From, State) -> {ok, [], State}.
```

```
handle_cast({message, {FromUser, FromServer}, Text}, State) ->
  io:format("~s (~p) says: ~p~n", [FromUser, FromServer, Text]),
  {noreply, State};

handle_cast(_Request, State) ->
  io:format("Unknown request ~p~n", _Request),
  {noReply, State}.

handle_info(Info, State) ->
  io:format("Received unexpected message: ~p~n", [Info]),
  {noreply, State}.

terminate(_Reason, _State) ->
  ok.

code_change(_OldVsn, State, _Extra) ->
  {ok, State}.


% internal functions

%% @doc Gets the name of the chat host. This is a really
%% ugly hack; it works by sending itself a call to retrieve
%% the chat node name from the server state.

get_chat_node() ->
  gen_server:call(person, get_chat_node).

%% @doc Login to a server using a name
%% If you connect, tell the server your user name and node.
%% You don't need a reply from the server for this.

-spec(login(string()) -> term()).

login(UserName) ->
  ChatNode = get_chat_node(),
  if
    is_atom(UserName) ->
      gen_server:call({chatroom, ChatNode},
        {login, atom_to_list(UserName), node()});
    is_list(UserName) ->
      gen_server:call({chatroom, ChatNode},
        {login, UserName, node()});
    true ->
      {error, "User name must be an atom or a list"}
  end.


%% @doc Log out of the system. The person server will send a From that tells
%% who is logging out; the chatroom server doesn't need to reply.

-spec(logout() -> atom()).
```

```
logout() ->
  ChatNode = get_chat_node(),
  gen_server:call({chatroom, ChatNode}, {logout}),
  ok.


%% @doc Send the given Text to the chat room server. No reply needed.

-spec(say(string()) -> atom()).

say(Text) ->
  ChatNode = get_chat_node(),
  gen_server:call({chatroom, ChatNode}, {say, Text}),
  ok.

%% @doc Ask chat room server for a list of users.

-spec(users() -> [string()]).

users() ->
  gen_server:call({chatroom, get_chat_node()}, users).

%% @doc Ask chat room server for a profile of a given person.

-spec(who(string(), atom()) -> [tuple()]).

who(Person, ServerRef) ->
  gen_server:call({chatroom, get_chat_node()}, {who, Person, ServerRef}).

%% @doc Update profile with a key/value pair.

-spec(profile(atom(), term()) -> term()).

profile(Key, Value) ->
  % ask *this* server for the current state
  NewProfile = gen_server:call(person, {profile, Key, Value}),
  {ok, NewProfile}.
```