RECORDS

Records are, first of all, a hack. They are more or less an afterthought to the language and can have their share of inconveniences. I'll cover that later. They're still pretty useful whenever you have a small data structure where you want to access the attributes by name directly. As such, Erlang records are a lot like structs in C (if you know C.)

They're declared as module attributes in the following manner:

```
-module(records).
-compile(export_all).

-record(robot, {name,
type=industrial,
hobbies,
details=[]}).
```

So here we have a record representing robots with 4 fields: name, type, hobbies and details. There is also a default value for type and details, `industrial` and `[]`, respectively. Here's how to declare a record in the modulerecords:

```
first_robot() ->
#robot{name="Mechatron",
type=handmade,
details=["Moved by a small man inside"]}.
```

And running the code:

```
1> c(records).
{ok,records}
2> records:first_robot().
{robot,"Mechatron",handmade,undefined,
["Moved by a small man inside"]}
```

Woops! Here comes the hack! Erlang records are just syntactic sugar on top of tuples. Fortunately, there's a way to make it better. The Erlang shell has a command `rr(Module)` that lets you load record definitions from *`Module`*:

```
3> rr(records).
[robot]
4> records:first_robot().
#robot{name = "Mechatron",type = handmade,
hobbies = undefined,
details = ["Moved by a small man inside"]}
```

Ah there! This makes it much easier to work with records that way. You'll notice that in `first_robot/0`, we had not defined the `hobbies` field and it had no default value in its declaration. Erlang, by defaults, sets the value to`undefined` for you.

To see the behavior of the defaults we set in the `robot` definition, let's compile the following function:

```
car_factory(CorpName) ->
#robot{name=CorpName, hobbies="building cars"}.
```

And run it:

```
5> c(records).
{ok,records}
6> records:car_factory("Jokeswagen").
#robot{name = "Jokeswagen",type = industrial,
hobbies = "building cars",details = []}
```

And we have an industrial robot that likes to spend time building cars.

Note: The function `rr()` can take more than a module name: it can take a wildcard (like`rr("*")`) and also a list as a second argument to specify which records to load.
There are a few other functions to deal with records in the shell: `rd(Name, Definition)` lets you define a record in a manner similar to the `-record(Name, Definition)` used in our module. You can use `rf()` to 'unload' all records, or `rf(Name)` or `rf([Names])` to get rid of specific definitions.
You can use `rl()` to print all record definitions in a way you could copy-paste into the module or use `rl(Name)` or `rl([Names])` to restrict it to specific records.
Finally, `rp(Term)` lets you convert a tuple to a record (given the definition exists).

Writing records alone won't do much. We need a way to extract values from them. There are basically two ways to do this. The first one is with a special 'dot syntax'. Assuming you have the record definition for robots loaded:

```
5> Crusher = #robot{name="Crusher", hobbies=["Crushing
people","petting cats"]}.
#robot{name = "Crusher",type = industrial,
hobbies = ["Crushing people","petting cats"],
details = []}
6> Crusher#robot.hobbies.
["Crushing people","petting cats"]
```

Ugh, not a pretty syntax. This is due to the nature of records as tuples. Because they're just some kind of compiler trick, you have to keep keywords around defining what record goes with what variable, hence the `#robot` part of`Crusher#robot.hobbies`. It's sad, but there's no way out of it. Worse than that, nested records get pretty ugly:

```
7> NestedBot = #robot{details=#robot{name="erNest"}}.
#robot{name = undefined,type = industrial,
hobbies = undefined,
details = #robot{name = "erNest",type = industrial,
hobbies = undefined,details = []}}
8> (NestedBot#robot.details)#robot.name.
"erNest"
```

And yes, the parentheses are mandatory.

Update:

Starting with revision R14A, it is now possible to nest records without the parentheses. The`NestedBot` example above could also be written as `NestedRobot#robot.details#robot.name`and work the same.

To further show the dependence of records on tuples, see the following:

```
9> #robot.type.
3
```

What this outputs is which element of the underlying tuple it is.

One saving feature of records is the possibility to use them in function heads to pattern match and also in guards. Declare a new record as follows on top of the file, and then add the functions under:

```
-record(user, {id, name, group, age}).

%% use pattern matching to filter
admin_panel(#user{name=Name, group=admin}) ->
Name ++ " is allowed!";
admin_panel(#user{name=Name}) ->
Name ++ " is not allowed".

%% can extend user without problem
adult_section(U = #user{}) when U#user.age >= 18 ->
%% Show stuff that can't be written in such a text
allowed;
adult_section(_) ->
%% redirect to sesame street site
forbidden.
```

The syntax to bind a variable to any field of a record is demonstrated in the `admin_panel/1` function (it's possible to bind variables to more than one field). An important thing to note about the `adult_section/1` function is that you need to do `SomeVar = #some_record{}` in order to bind the whole record to a variable. Then we do the compiling as usual:

```
10> c(records).
{ok,records}
11> rr(records).
[robot,user]
12> records:admin_panel(#user{id=1, name="ferd", group=admin
, age=96}).
"ferd is allowed!"
13> records:admin_panel(#user{id=2, name="you", group=users,
 age=66}).
"you is not allowed"
14> records:adult_section(#user{id=21, name="Bill", group=us
ers, age=72}).
allowed
15> records:adult_section(#user{id=22, name="Noah", group=us
ers, age=13}).
forbidden
```

What this lets us see is how it is not necessary to match on all parts of the tuple or even know how many there are when writing the function: we can only match on the age or the group if that's what's needed and forget about all the rest of the structure. If we were to use a normal tuple, the function definition might need to look a bit like`function({record, _, _, ICareAboutThis, _, _}) -> ...`. Then, whenever someone decides to add an element to the tuple, someone else (probably angry about it all) would need to go around and update all functions where that tuple is used.

The following function illustrates how to update a record (they wouldn't be very useful otherwise):

```
repairman(Rob) ->
Details = Rob#robot.details,
NewRob = Rob#robot{details=["Repaired by
repairman"|Details]},
{repaired, NewRob}.
```

And then:

```
16> c(records).
{ok,records}
17> records:repairman(#robot{name="Ulbert", hobbies=["trying
to have feelings"]}).
{repaired,#robot{name = "Ulbert",type = industrial,
hobbies = ["trying to have feelings"],
details = ["Repaired by repairman"]}}
```

And you can see my robot has been repaired. The syntax to update records is a bit special here. It looks like we're updating the record in place (`Rob#robot{Field=NewValue}`) but it's all compiler trickery to call the underlying`erlang:setelement/3` function.

One last thing about records. Because they're pretty useful and code duplication is annoying, Erlang programmers frequently share records across modules with the help of *header files*. Erlang header files are pretty similar to their C counter-part: they're nothing but a snippet of code that gets added to the module as if it were written there in the first place. Create a file named records.hrl with the following content:

```
%% this is a .hrl (header) file.
-record(included, {some_field,
some_default = "yeah!",
unimaginative_name}).
```

To include it in records.erl, just add the following line to the module:

```
-include("records.hrl").
```

And then the following function to try it:

```
included() -> #included{some_field="Some value"}.
```

Now, try it as usual:

```
18> c(records).
{ok,records}
19> rr(records).
[included,robot,user]
20> records:included().
#included{some_field = "Some value",some_default = "yeah!",
unimaginative_name = undefined}
```

Hooray! That's about it for records; they're ugly but useful. Their syntax is not pretty, they're not much but a hack, but they're relatively important for the maintainability of your code.

Note: You will often see open source software using the method shown here of having a project-wide `.hrl` file for records that are shared across all modules. While I felt obligated to document this use, I strongly recommend that you keep all record definitions local, within one module. If you want some other module to look at a record's innards, write functions to access its fields and keep its details as private as possible. This helps prevent name clashes, avoids problems when upgrading code, and just generally improves the readability and maintainability of your code.

Source : http://learnyousomeerlang.com/a-short-visit-to-common-data-structures