Records and Maps

Records and Maps

- Records
- Records and the Shell
- Maps

Records

- **Records are used to store a fixed number of items**
  - Similar to a C struct or a Pascal record
- **These items are accessed by name**
  - Unlike tuples where they are accessed by position
- **Items can easily be added or removed without affecting the code not using these items**
  - Unlike tuples, where updates are needed everywhere

Records: defining

```
-record(Name, {Field1, [= DefaultValue1],
                ...,
               FieldN, [= DefaultValueN]}).


-record(person, {name, age = 0, phone}).
```

- Fields may be assigned a value when declared
- The default value for a field is the atom **'undefined'**
- Record definitions should be placed in a header file

Records: defining

- All shared record definitions should be placed in include files. Local ones stay in their modules.
- The suffix .hrl is recommended but not enforced
- Include files are added to a module using the -include("File.hrl"). directive, "quotes" included.
- The compiler will look for the include file in the compiler include path list
- By default the include path list includes the current working directory

Records: creating instances

```
Var = #Name{Field1 = Expression1,
            ...,
            FieldN = ExpressionN}

Person = #person{name = "Joe",
                 phone = [1,2,3,4]}
```

- ▶️ If any of the fields are omitted , the default values in the record definition are assigned to them (including 'undefined')
- ▶️ The default value expression is evaluated when the record instance is created

Records: field selectors

```
FieldVar = RecordVar#Name.FieldName

P = #person{name = "Mike"}

Name = P#person.name                    % Name =:= "Mike"
Age  = P#person.age                     % Age =:= 0
```

▶️ Field and record names may not be variables

Records: updating

```
NewVar = OldVar#Name{Field1 = Expression1,
                     ...,
                     FieldN = ExpressionN}

P = #person{name = "Mike"}              % age = 0
NewP = P#person{age = 35}               % age = 35
NewP2 = P#person{name = "Joe"}          % age = 0
```

- Only fields to be changed have to be referred
- Others will return their old values
- Remember that Erlang variables are single assignment!

Records: pattern matching

```
P = #person{name = "Joe", age = 35, phone = [1,2,3,4]}

#person{name = Name, age = 35, phone = Phone} = P

foo(#person{name = "Joe", age = Age}) -> ...
```

- ▶️ Records may be used in pattern matching to extract variables or pick the flow of computation

Records: guards

```
foobar(P) when is_record(P, person),
               P#person.name =:= "Joe" -> ...
```

is the same as:

```
foobar(P = #person{name = "Joe"}) -> ...
```

- Record guards may be used to pick the flow of execution in different clauses
- When using guards to inspect a field of a record, use the record guard as well if **P** will not always be a record of type **person**.

Records: nesting

```
-record(name, {first, last}).

P = #person{name = #name{first = "Robert",
                         last = "Virding"}},

First = (P#person.name)#name.first.
```

- Record fields may contain other nested records
- Fields in nested records are accessed with one operation

Records: internal representation

```
#person{} =:= {person, undefined, 0, undefined}
```

Warning! Never use the tuple representation of records!

- Records are represented as tuples by the run time system
- The precompiler translates the creating, updating and selecting operations on records to operations on tuples
- N fields in the record will result in a tuple with N+1 elements
- The first element is the name of the record

Records: information

- record_info(fields, RecType)
  - returns a list of field names
- record_info(size, RecType)
  - returns the size of the tuple (Fields + 1)
- #RecType.Name returns the position of Name in the tuple
- RecType and Name must be atoms, and they may not be variables bound to atoms
- Record information constructs are handled by the precompiler

Records and the Shell

```
1> rr(person).
[person].
2> rl().
-record(person, {name,
                 age = 0,
                 phone}
ok
3> P = #person{name='Henry',
3>                phone=[0,1,2]}.
#person{name='Henry',
        age=0,
        phone=[0,1,2]}
```

- All record definitions in a module can be loaded using the function **rr/1**
- The records known to the shell can be listed using **rl/0**

Records and the Shell

```
4> rd(request,
      {id,action,data,
       stamp = erlang:timestamp()}).
request
5> #request{}.
#request{id = undefined,
         action = undefined,
         data = undefined,
         stamp = {1151,112955,759346}}
5> #request{}.
#request{id = undefined,
         action = undefined,
         data = undefined,
         stamp = {1151,112955,983613}}
6> rf(request).
ok
```

- Records can be defined using **rd/2**
- Useful for testing and debugging
- A record can also be forgotten like variables by using **rf/0** and **rf/1**
- **erlang:timestamp/0** is evaluated when the record instance is created

Maps

- **Maps are used to store a variable number of items**
  - Similar to a hash table
- **These items are accessed by a key**
  - Keys and items can be arbitrary Erlang terms
- **Items can be added or removed without affecting the code not using these items**

Maps: creating instances

```
Map = #{Key1 => Expression1,
        ...,
        KeyN => ExpressionN}

Person = #{name => "Joe", phone => [1,2,3,4]}
```

- ▶️ Only the given fields will be included in the map

Maps: **field selectors**

```
FieldVar = maps:get(Key, Map)

P = #{name => “Mike”, age => 0}
Name  = maps:get(name, P)                 % Name =:= "Mike"
Age   = maps:get(age, P)                  % Age =:= 0
Phone = maps:get(phone, P)                exception
```

- ▶️ Only works with keys in the map
- ▶️ Generates an exception otherwise
- ▶️ Can use pattern matching as well
  #{name := Name} = P

Maps: updating

```
NewVar = Map#{Field1 => Expression1,
              ...,
              FieldN := ExpressionN}

P = #{name => “Mike”, age => 0}
NewP = P#{age => 35}                    % age = 35
NewP2 = P#{phone := [1,2,3,4]}          exception
```

- => updates existing fields or adds new fields
- := ONLY updates existing fields
- Only fields to be changed have to be referred
- Others will return their old values
- Maps are immutable so new maps are created
- Remember that Erlang variables are single assignment!

Maps: pattern matching

```
P = #{name => "Joe", age => 35, phone => [1,2,3,4]}
#{name := Name, age := 35, phone := Phone} = P

foo(#{name := "Joe", age := Age}) -> ...

foo(P = #{name := "Joe", age := Age}) -> ...
```

- **:=** is used for matching fields
- Maps may be used in pattern matching to extract variables or pick the flow of computation

Maps: nesting

```
P = #{name => #{first => "Robert",last => "Virding"}},

First = maps:get(first, maps:get(name, P))

#{name := #{first := First}} = P
```

▶ Map fields may contain other nested maps

Maps: information

- **maps:keys(Map)**
  - returns a list of keys
- **maps:size(Map)**
  - returns the size of the map
- **maps:is_key(Key, Map)**
  - returns whether the map contains Key
- **maps:from_list(KeyValueList)**
- **maps:to_list(Map)**
  - convert between maps and list of {Key,Value}

Records vs Maps

- ▶️ **Records**
  - Static and compile time checking
  - Faster
  - More difficult to modify
- ▶️ **Maps**
  - Dynamic
  - Easier to modify
  - Easier with large number of keys

Records and Maps

- Records
- Records and the shell
- Maps