IF140303 - Modul Praktikum Pengembangan Aplikasi Web

Universitas Pradita

Powered by ChatGPT

Alfa Yohannis

October 3, 2024

Contents

Chapter 1

Pengenalan Elixir

1.1 Elixir

Elixir adalah bahasa pemrograman fungsional yang dirancang untuk membangun aplikasi yang scalable dan maintainable. Dikembangkan oleh José Valim, Elixir berjalan di atas Erlang Virtual Machine (BEAM) dan memanfaatkan keunggulan teknologi Erlang untuk manajemen proses dan concurrent programming. Bahasa ini sering digunakan dalam pengembangan aplikasi berskala besar, seperti sistem web yang memerlukan performa tinggi dan kemampuan untuk menangani banyak koneksi secara bersamaan.

1.1.1 Mengapa Elixir Ada

Elixir diciptakan untuk mengatasi beberapa keterbatasan yang ada pada bahasa pemrograman lain, khususnya dalam konteks aplikasi yang memerlukan skalabilitas tinggi dan keandalan yang kuat. Berikut adalah beberapa alasan utama mengapa Elixir dikembangkan:

- **Mengatasi Keterbatasan Bahasa Lain:** José Valim, pengembang Elixir, merasa bahwa bahasa pemrograman yang ada saat itu tidak sepenuhnya memenuhi kebutuhan aplikasi modern yang memerlukan concurrency, fault tolerance, dan scalability. Elixir dirancang untuk mengatasi kekurangan ini dengan memanfaatkan kekuatan Erlang.

- **Memanfaatkan Infrastruktur Erlang:** Elixir dibangun di atas Erlang Virtual Machine (BEAM), yang sudah terbukti andal dalam menangani aplikasi dengan banyak koneksi secara bersamaan dan dalam situasi yang memerlukan toleransi kesalahan. Dengan memanfaatkan BEAM, Elixir mewarisi kekuatan concurrency dan fault tolerance dari Erlang, tetapi dengan sintaksis yang lebih modern dan fitur tambahan.

- **Produktivitas Pengembang:** Elixir dirancang untuk meningkatkan produktivitas pengembang dengan menyediakan fitur-fitur seperti metaprogramming dan sintaksis yang bersih dan intuitif. Ini memungkinkan pengembang untuk menulis kode yang lebih mudah dibaca dan dikelola, serta mempercepat pengembangan aplikasi.

- **Pengembangan Aplikasi Web Modern:** Seiring dengan meningkatnya kebutuhan untuk aplikasi web yang real-time dan dinamis, Elixir menawarkan solusi yang efektif dengan framework seperti Phoenix. Phoenix mendukung real-time communication dan pengembangan aplikasi web yang responsif, menjadikannya pilihan yang menarik untuk pengembangan web modern.

- **Kebutuhan Scalability dan Fault Tolerance:** Dalam dunia teknologi yang terus berkembang, aplikasi perlu mampu menanggapi peningkatan beban kerja dan potensi kegagalan sistem. Elixir memberikan alat dan struktur untuk membangun aplikasi yang dapat dengan mudah diskalakan dan dikelola, bahkan dalam lingkungan yang penuh tantangan.

Dengan mengatasi masalah-masalah ini, Elixir menyediakan platform yang kuat dan fleksibel untuk pengembangan aplikasi yang memerlukan performa tinggi, keandalan, dan kemudahan dalam pengelolaan.

1.1.2 Sejarah Elixir

Elixir dikembangkan oleh José Valim dan pertama kali diperkenalkan pada tahun 2011. Valim, yang sebelumnya dikenal sebagai kontributor utama untuk framework web Ruby on Rails, memiliki visi untuk menciptakan bahasa pemrograman yang menggabungkan kekuatan concurrency dan fault tolerance dari Erlang dengan sintaksis modern dan fitur-fitur baru yang mendukung produktivitas pengembang.

Beberapa tonggak penting dalam sejarah Elixir adalah:

- **2011:** José Valim mengumumkan Elixir sebagai proyek open-source. Tujuan awalnya adalah untuk mengatasi keterbatasan bahasa pemrograman yang ada, dengan memanfaatkan infrastruktur Erlang untuk membangun aplikasi yang lebih scalable dan maintainable.
- **2014:** Elixir mencapai versi 1.0, menandakan kestabilan dan kematangan bahasa tersebut untuk digunakan dalam produksi. Versi ini memperkenalkan berbagai fitur penting serta integrasi dengan alat dan library yang mendukung pengembangan aplikasi modern.
- **2015:** Framework web Phoenix, yang dibangun dengan Elixir, diluncurkan. Phoenix menawarkan fitur-fitur canggih seperti live view dan real-time capabilities, menjadikannya pilihan populer untuk pengembangan aplikasi web yang dinamis dan interaktif.
- **2018:** Elixir semakin banyak diadopsi dalam berbagai sektor industri, dari fintech hingga telekomunikasi, berkat kemampuannya dalam menangani beban kerja tinggi dan memastikan keandalan sistem. Komunitas Elixir terus berkembang dengan dukungan dari berbagai konferensi, meetup, dan kontribusi komunitas.
- **2021 dan seterusnya:** Elixir terus berkembang dengan pembaruan dan peningkatan, memperkenalkan fitur-fitur baru seperti improved tooling, pengembangan library, dan dukungan untuk teknologi terbaru. Komunitas Elixir terus aktif, mendukung adopsi dan perkembangan bahasa ini di berbagai aplikasi dan industri.

Dengan latar belakang sejarah ini, Elixir telah berkembang menjadi bahasa pemrograman yang solid dan inovatif, menawarkan solusi yang kuat untuk tantangan dalam pengembangan aplikasi modern.

1.1.3 Keunggulan Elixir

- **Concurrency dan Parallelism:** Elixir menyediakan model concurrency yang kuat dan efisien melalui aktor (processes) yang ringan dan dapat berkomunikasi satu sama lain dengan menggunakan message passing. Ini memungkinkan aplikasi untuk menangani ribuan proses secara bersamaan dengan efisiensi tinggi.
- **Fault Tolerance:** Mengadopsi prinsip ”let it crash” dari Erlang, Elixir memungkinkan penanganan kesalahan yang robust dengan strategi supervision tree. Hal ini memastikan bahwa aplikasi tetap beroperasi meskipun beberapa bagian mengalami kegagalan.
- **Scalability:** Elixir dirancang untuk mendukung scaling horizontal dan vertikal dengan mudah. Sistem yang dibangun dengan Elixir dapat berjalan pada berbagai node dan terdistribusi, serta mampu mengelola beban kerja yang meningkat.
- **Metaprogramming:** Elixir mendukung metaprogramming, yang memungkinkan developer untuk menulis kode yang menghasilkan kode lain pada saat kompilasi. Ini memberikan fleksibilitas dan kemampuan untuk mengembangkan DSL (Domain Specific Languages) serta memperluas bahasa sesuai kebutuhan.

- **Pengembangan Web:** Elixir sering digunakan dalam pengembangan aplikasi web modern dengan framework seperti Phoenix, yang menyediakan fitur-fitur canggih seperti real-time communication (WebSocket) dan komponen komputasi yang terdistribusi.

1.1.4 Kelemahan Elixir

Meskipun Elixir menawarkan banyak keunggulan, ada beberapa kelemahan yang perlu diperhatikan:

- **Kurva Belajar:** Bagi pengembang yang tidak familiar dengan pemrograman fungsional atau Erlang, Elixir dapat memiliki kurva belajar yang curam. Konsep-konsep seperti immutability, recursion, dan model concurrency mungkin memerlukan waktu untuk dipahami dan diterapkan secara efektif.
- **Ekosistem yang Terbatas:** Meskipun ekosistem Elixir berkembang pesat, masih terdapat beberapa kekurangan dalam hal library dan alat dibandingkan dengan bahasa pemrograman yang lebih mapan seperti JavaScript atau Python. Hal ini dapat mempengaruhi ketersediaan solusi atau integrasi dengan alat pihak ketiga.
- **Kinerja untuk Tugas CPU-Intensif:** Walaupun Elixir sangat baik dalam menangani concurrency dan I/O-bound tasks, kinerjanya dalam tugas CPU-intensive dapat menjadi masalah. Aplikasi yang memerlukan perhitungan berat atau algoritma kompleks mungkin tidak seefisien dalam Elixir jika dibandingkan dengan bahasa pemrograman lain yang lebih dioptimalkan untuk kinerja tersebut.
- **Komunitas dan Dokumentasi:** Meskipun komunitas Elixir aktif dan mendukung, dokumentasi dan sumber daya pembelajaran mungkin tidak sebanyak yang tersedia untuk bahasa pemrograman yang lebih populer. Hal ini dapat membuat pengembang baru merasa kesulitan untuk menemukan informasi atau dukungan yang mereka butuhkan.
- **Integrasi dengan Sistem Lama:** Mengintegrasikan Elixir dengan sistem lama atau infrastruktur yang tidak dirancang untuk mendukung aplikasi berbasis Elixir bisa menjadi tantangan. Hal ini sering kali memerlukan usaha tambahan dalam hal integrasi dan pemeliharaan.

Memahami kelemahan ini penting bagi pengembang untuk membuat keputusan yang terinformasi tentang penggunaan Elixir dalam proyek mereka dan untuk mengelola potensi masalah yang mungkin muncul.

1.2 Instalasi Elixir

Untuk mulai menggunakan Elixir, Anda perlu menginstalnya di sistem operasi yang Anda gunakan. Berikut adalah panduan untuk menginstal Elixir di Windows, Mac, dan Ubuntu/Linux.

1.2.1 Instalasi di Windows

1. Kunjungi halaman [download Elixir](https://elixir-lang.org/install.html) dan unduh installer Windows yang sesuai.
2. Jalankan installer yang telah diunduh dan ikuti petunjuk di layar untuk menyelesaikan instalasi.
3. Setelah instalasi selesai, buka Command Prompt atau PowerShell dan verifikasi instalasi dengan menjalankan perintah berikut:

```
elixir --version
```

1.2.2 Instalasi di Mac

1. Pastikan Anda memiliki [Homebrew](https://brew.sh/) terinstal. Jika belum, instal Homebrew dengan perintah berikut di Terminal:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com
/Homebrew/install/HEAD/install.sh)"
```

2. Instal Elixir menggunakan Homebrew dengan perintah berikut:

```
brew install elixir
```

3. Setelah instalasi selesai, verifikasi dengan menjalankan perintah berikut di Terminal:

```
elixir --version
```

1.2.3 Instalasi di Ubuntu/Linux

1. Tambahkan repositori Elixir dan instal paket yang diperlukan dengan perintah berikut:

```
sudo apt update
sudo apt install -y esl-erlang
sudo apt install -y elixir
```

2. Setelah instalasi selesai, verifikasi dengan menjalankan perintah berikut di Terminal:

```
elixir --version
```

1.3 Membuat Proyek Elixir dan Membukanya di VS Code

Untuk memulai pengembangan dengan Elixir, langkah pertama adalah membuat proyek Elixir baru dan kemudian membuka proyek tersebut di Visual Studio Code (VS Code).

1.3.1 Membuat Proyek Elixir Baru

1. Buka terminal atau command prompt.
2. Navigasi ke direktori di mana Anda ingin membuat proyek baru.
3. Buat proyek Elixir baru dengan perintah berikut:

```
    mix new nama_proyek
```

4. Masuk ke direktori proyek yang baru dibuat:

```
    cd nama_proyek
```

1.3.2 Membuka Proyek di Visual Studio Code

1. Pastikan Anda sudah menginstal [Visual Studio Code](https://code.visualstudio.com/).
2. Buka VS Code dan pilih menu `File > Open Folder`.
3. Navigasi ke direktori proyek Elixir yang telah Anda buat, lalu klik `Open`.
4. Alternatifnya, Anda bisa membuka proyek langsung dari terminal dengan perintah berikut:

```
    code .
```

5. Setelah proyek terbuka, Anda dapat mulai mengembangkan kode Elixir di dalamnya.

Dengan mengikuti langkah-langkah di atas, Anda akan siap untuk memulai pengembangan proyek Elixir menggunakan VS Code.

1.4 Perbedaan Pemrograman Berorientasi Objek dan Pemrograman Fungsional

Pemrograman Berorientasi Objek (OOP) dan Pemrograman Fungsional (FP) merupakan dua paradigma pemrograman yang memiliki prinsip dan pendekatan yang berbeda dalam menyusun kode dan mengelola data.

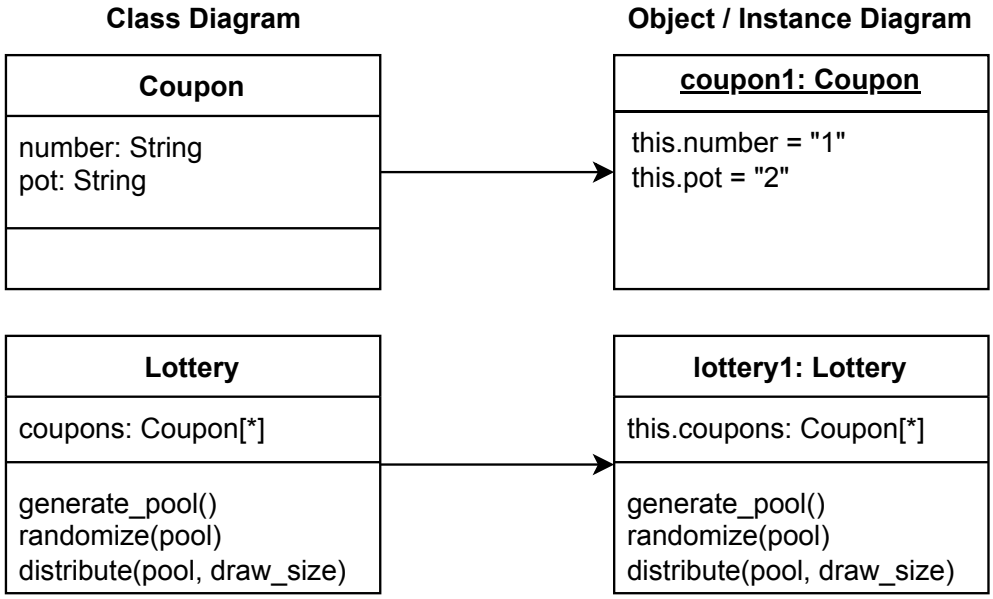


Figure 1.1: Pada gambar di atas metode `generate_pool` menginisialisasi `coupons` pada objek `lottery1` dan `randomize` mengubah *state* dari `coupons`.

1.4.1 Pemrograman Berorientasi Objek (OOP)

OOP menekankan pada pembuatan objek yang merupakan representasi dari entitas dunia nyata. Objek-objek ini memiliki **atribut** (data) dan **metode** (fungsi) yang mengoperasikan data tersebut. Salah satu karakteristik utama dari OOP adalah bahwa **objek dapat mengubah state-nya**, yakni nilai dari atribut-atribut yang dimilikinya bisa berubah selama program berjalan. Misalnya, pada Gambar 1.1, metode `generate_pool` menginisialisasi `coupons` pada objek `lottery1` dan `randomize` mengubah *state* dari `coupons`. Beberapa konsep utama dalam OOP adalah:

- **Kelas dan Objek**: Kelas adalah cetak biru dari objek. Objek adalah instansiasi dari kelas.
- **Enkapsulasi**: Pengelompokan data dan fungsi dalam objek, memungkinkan kontrol akses terhadap data.

- **Pewarisan**: Mekanisme untuk membuat kelas baru berdasarkan kelas yang sudah ada, mewarisi atribut dan metode.
- **Polimorfisme**: Kemampuan objek untuk diperlakukan sebagai bentuk lain dari objek yang berbeda, memungkinkan metode yang sama digunakan untuk tipe objek yang berbeda.

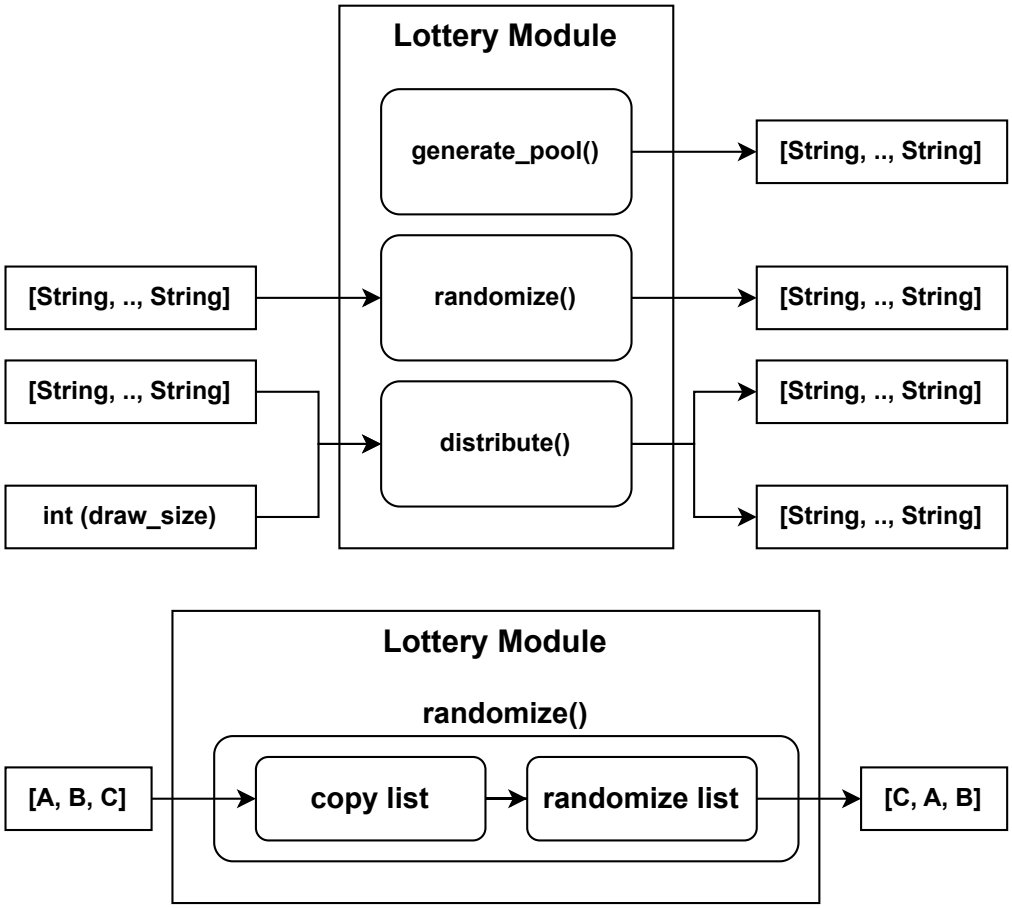


Figure 1.2: Pada gambar di atas, `coupons = [A, B, C]` digandakan terlebih dahulu (`copy list`), kemudian diacak (`randomize list`) menghasilkan `coupons = [C, A, B]`.

1.4.2 Pemrograman Fungsional (FP)

Pemrograman fungsional didasarkan pada konsep **fungsi matematika** yang tidak memiliki efek samping, sehingga fungsi akan selalu memberikan hasil yang sama untuk input yang sama. **FP tidak mengubah state dari nilai**, melainkan setiap operasi pada data menghasilkan **salinan baru** dari data tersebut dengan modifikasi yang diinginkan. Sebagai contoh, Pada Gambar 1.2, `coupons = [A, B, C]` digandakan terlebih dahulu (`copy list`), kemudian diacak (`randomize list`) menghasilkan `coupons = [C, A, B]`. Beberapa konsep kunci dari FP adalah:

- **Fungsi Murni**: Fungsi yang tidak memodifikasi keadaan luar dan selalu memberikan hasil yang sama untuk argumen yang sama.
- **Immutability**: Data tidak dapat diubah setelah didefinisikan, yang meningkatkan prediktabilitas program. Setiap modifikasi menghasilkan salinan baru dari data.

- **Rekursi**: Pengulangan dilakukan melalui pemanggilan fungsi yang berulang, bukan melalui loop imperatif.
- **First-Class Functions**: Fungsi diperlakukan sebagai nilai yang dapat diteruskan sebagai parameter, dikembalikan sebagai hasil, atau disimpan dalam variabel.

Pemrograman fungsional lebih fokus pada transformasi data dengan fungsi, tanpa mengubah data secara langsung, sehingga mengurangi bug yang disebabkan oleh efek samping dan membuat program lebih mudah diuji.

1.4.3 Perbandingan OOP dan FP

- OOP menekankan pada objek yang menyimpan state dan metode yang mengubah state tersebut, sedangkan FP menekankan pada fungsi murni yang **tidak mengubah state** melainkan **menghasilkan salinan baru** dari data sebagai hasil transformasi.
- Pada OOP, perubahan state adalah hal umum, sementara pada FP, data bersifat immutable, dan perubahan dihasilkan dengan mengkopi nilai input dan memodifikasinya.
- OOP biasanya menggunakan loop imperatif untuk iterasi, sementara FP menggunakan rekursi dan fungsi seperti `map`, `filter`, dan `reduce`.
- OOP lebih cocok untuk sistem yang melibatkan interaksi antar entitas (seperti GUI atau game), sedangkan FP lebih efisien untuk perhitungan matematis dan manipulasi data.

Kedua paradigma ini dapat saling melengkapi dan sering digunakan secara bersamaan dalam berbagai bahasa pemrograman modern.

1.5 String di Elixir

String adalah salah satu tipe data dasar di Elixir yang digunakan untuk merepresentasikan teks. String di Elixir didefinisikan dengan menggunakan tanda kutip ganda (`"..."`). Setiap string di Elixir adalah UTF-8 encoded binary, yang memungkinkan penggunaan karakter dari berbagai bahasa.

1.5.1 Operasi Dasar pada String

Elixir menyediakan berbagai operasi yang dapat dilakukan pada string, seperti menggabungkan, membandingkan, dan memanipulasi string. Beberapa operasi dasar termasuk:

- **Penggabungan String**: Anda dapat menggabungkan dua atau lebih string menggunakan operator `<>`.
- **Interpolasi String**: Anda dapat menyisipkan nilai ekspresi atau variabel ke dalam string dengan menggunakan `#{...}`.
- **Mengukur Panjang String**: Fungsi `String.length/1` dapat digunakan untuk mendapatkan jumlah karakter dalam string.
- **Mengubah Huruf Besar/Kecil**: Fungsi seperti `String.upcase/1` dan `String.downcase/1` digunakan untuk mengubah huruf string menjadi huruf besar atau kecil.

1.5.2 Contoh Kode

Berikut adalah beberapa contoh kode yang menunjukkan cara bekerja dengan string di Elixir:

```elixir
# Menggabungkan dua string
greeting = "Hello, " <> "world!"
IO.puts(greeting)  # Output: Hello, world!

# Interpolasi string
name = "Elixir"
message = "Welcome to #{name} programming!"
IO.puts(message)  # Output: Welcome to Elixir programming!

# Mengukur panjang string
len = String.length("Hello")
IO.puts(len)  # Output: 5

# Mengubah string menjadi huruf besar
IO.puts(String.upcase("elixir"))  # Output: ELIXIR

# Mengubah string menjadi huruf kecil
IO.puts(String.downcase("ELIXIR"))  # Output: elixir
```

Dalam contoh-contoh di atas, berbagai operasi dasar string seperti penggabungan, interpolasi, dan perubahan huruf besar/kecil diperlihatkan. Ini adalah beberapa contoh sederhana yang menunjukkan kekuatan dan fleksibilitas dalam bekerja dengan string di Elixir.

1.6 List di Elixir

List adalah salah satu tipe data utama di Elixir yang digunakan untuk menyimpan kumpulan elemen. List di Elixir diwakili oleh tanda kurung siku `[]` dan dapat menyimpan elemen-elemen dengan tipe data yang berbeda-beda, termasuk angka, string, dan bahkan list lainnya.

1.6.1 Operasi Dasar pada List

Elixir menyediakan berbagai operasi yang dapat dilakukan pada list, seperti menambahkan elemen, menggabungkan list, dan mengakses elemen tertentu. Beberapa operasi dasar termasuk:

- **Menambah Elemen**: Anda dapat menambahkan elemen ke dalam list menggunakan operator kons `[head | tail]` atau dengan fungsi `List.insert_at/3`.
- **Menggabungkan List**: Dua atau lebih list dapat digabungkan menggunakan operator `++`.
- **Mengakses Elemen**: Elemen dalam list dapat diakses dengan menggunakan notasi indeks atau dengan pola pencocokan (pattern matching).
- **Menghitung Panjang List**: Fungsi `length/1` digunakan untuk mendapatkan jumlah elemen dalam list.
- **Mencari Elemen dalam List**: Fungsi seperti `Enum.member?/2` digunakan untuk memeriksa apakah sebuah elemen ada dalam list.

1.6.2 Contoh Kode

Berikut adalah beberapa contoh kode yang menunjukkan cara bekerja dengan list di Elixir:

```elixir
# Membuat list baru
list = [1, 2, 3, 4, 5]
IO.inspect(list)  # Output: [1, 2, 3, 4, 5]

# Menambahkan elemen ke list
new_list = [0 | list]
IO.inspect(new_list)  # Output: [0, 1, 2, 3, 4, 5]
```

```elixir

# Menggabungkan dua list
combined_list = [1, 2, 3] ++ [4, 5, 6]
IO.inspect(combined_list)  # Output: [1, 2, 3, 4, 5, 6]

# Mengakses elemen pertama dan sisa list
[head | tail] = list
IO.puts("Head: #{head}")  # Output: Head: 1
IO.inspect(tail)          # Output: [2, 3, 4, 5]

# Menghitung panjang list
IO.puts("Length: #{length(list)}")  # Output: Length: 5

# Memeriksa apakah sebuah elemen ada dalam list
IO.puts(Enum.member?(list, 3))  # Output: true
```

Dalam contoh-contoh di atas, berbagai operasi dasar seperti menambah elemen, menggabungkan list, dan mengakses elemen diperlihatkan. List di Elixir sangat fleksibel dan banyak digunakan dalam pemrograman fungsional untuk menyimpan dan memanipulasi kumpulan data.

1.7 Modul `Enum` di Elixir

Modul `Enum` di Elixir menyediakan fungsi-fungsi untuk bekerja dengan koleksi data yang enumerable, seperti list, map, dan range. Fungsi-fungsi dalam modul `Enum` adalah sangat berguna untuk manipulasi dan transformasi data secara fungsional.

1.7.1 Fungsi `Enum.shuffle/1`

Fungsi `Enum.shuffle/1` digunakan untuk mengacak urutan elemen-elemen dalam sebuah list. Fungsi ini mengembalikan list baru dengan elemen-elemen yang telah diacak.

```elixir
list = [1, 2, 3, 4, 5]
shuffled_list = Enum.shuffle(list)
IO.inspect(shuffled_list)  # Output: [3, 1, 4, 5, 2] (hasil acak)
```

1.7.2 Fungsi `Enum.member?/2`

Fungsi `Enum.member?/2` digunakan untuk memeriksa apakah sebuah elemen ada dalam koleksi enumerable. Fungsi ini mengembalikan `true` jika elemen ditemukan, dan `false` jika tidak.

```elixir
list = [1, 2, 3, 4, 5]
is_member = Enum.member?(list, 3)
IO.puts(is_member)  # Output: true
```

1.7.3 Fungsi `Enum.split/2`

Fungsi `Enum.split/2` digunakan untuk membagi koleksi enumerable menjadi dua bagian berdasarkan jumlah elemen yang ditentukan. Fungsi ini mengembalikan tuple yang berisi dua list.

```elixir
list = [1, 2, 3, 4, 5]
{first_part, second_part} = Enum.split(list, 2)
IO.inspect(first_part)   # Output: [1, 2]
IO.inspect(second_part)  # Output: [3, 4, 5]
```

1.7.4 Fungsi `Enum.map/2`

Fungsi `Enum.map/2` menerapkan fungsi yang diberikan kepada setiap elemen dalam koleksi enumerable, menghasilkan koleksi baru dengan hasil-hasil tersebut.

```
list = [1, 2, 3, 4, 5]
squared_list = Enum.map(list, fn x -> x * x end)
IO.inspect(squared_list)  # Output: [1, 4, 9, 16, 25]
```

1.7.5 Fungsi `Enum.filter/2`

Fungsi `Enum.filter/2` memilih elemen-elemen dari koleksi enumerable yang memenuhi kondisi yang ditentukan oleh fungsi yang diberikan. Hasilnya adalah koleksi baru dengan elemen-elemen yang memenuhi syarat.

```
list = [1, 2, 3, 4, 5]
even_numbers = Enum.filter(list, fn x -> rem(x, 2) == 0 end)
IO.inspect(even_numbers)  # Output: [2, 4]
```

1.7.6 Fungsi `Enum.reduce/3`

Fungsi `Enum.reduce/3` secara berulang menerapkan fungsi yang diberikan kepada elemen-elemen dalam koleksi, mengakumulasi hasilnya. Fungsi ini sering digunakan untuk agregasi nilai-nilai.

```
list = [1, 2, 3, 4, 5]
sum = Enum.reduce(list, 0, fn x, acc -> x + acc end)
IO.puts(sum)  # Output: 15
```

Modul `Enum` sangat berguna untuk memanipulasi dan bekerja dengan koleksi data di Elixir. Dengan menggunakan fungsi-fungsi seperti `Enum.shuffle/1`, `Enum.member?/2`, `Enum.split/2`, `Enum.map/2`, `Enum.filter/2`, dan `Enum.reduce/3`, Anda dapat melakukan berbagai operasi pada data dengan cara yang efisien dan ekspresif.

1.8 Pengulangan dengan `for` di Elixir

Elixir menyediakan cara yang sangat elegan dan sederhana untuk melakukan pengulangan dengan menggunakan ‘for‘ loop. ‘for‘ loop di Elixir dapat digunakan untuk mengiterasi elemen-elemen dalam sebuah list, map, atau struktur lainnya. Selain itu, ‘for‘ loop juga dapat digunakan untuk nested looping, yaitu pengulangan bersarang.

1.8.1 Single Looping

Pengulangan tunggal menggunakan ‘for‘ di Elixir cukup mudah. Sebagai contoh, kita ingin mengiterasi sebuah list angka dan mengalikan setiap angkanya dengan 2.

```
numbers = [1, 2, 3, 4, 5]

doubled_numbers = for n <- numbers do
n * 2
end

IO.inspect(doubled_numbers)
# Output: [2, 4, 6, 8, 10]
```

Dalam contoh di atas, ‘for n <- numbers‘ akan mengiterasi setiap elemen dalam list ‘numbers‘, kemudian ‘n * 2‘ akan mengalikan setiap elemen dengan 2, dan hasilnya disimpan dalam list baru ‘doubled_numbers‘.

1.8.2 Nested Looping

'for' loop juga bisa digunakan untuk melakukan pengulangan bersarang (nested loop). Sebagai contoh, kita dapat membuat kombinasi dari dua list yang berbeda.

```
colors = ["red", "green", "blue"]
shapes = ["circle", "square"]

combinations = for color <- colors, shape <- shapes do
{color, shape}
end

IO.inspect(combinations)
# Output: [{"red", "circle"}, {"red", "square"}, {"green", "circle"}, {"green
    ", "square"}, {"blue", "circle"}, {"blue", "square"}]
```

Dalam contoh di atas, 'for color <- colors, shape <- shapes' akan mengiterasi setiap elemen dari list 'colors' dan 'shapes' untuk menghasilkan semua kombinasi yang mungkin antara warna dan bentuk. Hasilnya disimpan dalam list 'combinations'.

Dengan 'for' loop, Elixir menawarkan cara yang kuat dan fleksibel untuk mengelola pengulangan, baik untuk kasus tunggal maupun yang lebih kompleks seperti pengulangan bersarang.

1.9 The Lottery Module

Modul `Lottery` menyediakan fungsionalitas untuk mengelola sistem undian. Modul ini mencakup fungsi-fungsi untuk membuat, mengacak, memeriksa angka, dan mendistribusikan angka dalam pool undian. Berikut adalah kode lengkap untuk modul `Lottery`.

Listing 1.1: Complete Lottery Module

```
defmodule Lottery do
@moduledoc """
This module provides functionalities for managing a lottery system.
It includes functions for creating, shuffling, checking for numbers, and
    distributing numbers within the lottery pool.
"""

@spec greet() :: <<_::80>>
@doc """
Returns a greeting message.

## Examples

iex> Lottery.greet()
"Good luck!"
"""
def greet do
"Good luck!"
end

@spec generate_pool() :: [<<_::24, _::_*16>>, ...]
@doc """
Generates a pool of lottery numbers with different pots.

## Returns

- A list of lottery numbers with their respective pot numbers.

## Examples

iex> Lottery.generate_pool()
```

```
  ["Number 1 in Pot 1", "Number 2 in Pot 1", ...]
  """
  def generate_pool do
  numbers = ["Number 1", "Number 2", "Number 3", "Number 4", "Number 5", "
      Number 6"]
  pots = ["Pot 1", "Pot 2", "Pot 3", "Pot 4"]

  # Creates a pool by combining numbers and pots.
  for pot <- pots, number <- numbers do
  "#{number} in #{pot}"
  end
  end

  @doc """
  Randomizes the order of numbers in the pool.

  ## Parameters

  - pool: The list of lottery numbers to be shuffled.

  ## Returns

  - A new list with the numbers shuffled.

  ## Examples

  iex> Lottery.randomize(["Number 1 in Pot 1", "Number 2 in Pot 2"])
  ["Number 2 in Pot 2", "Number 1 in Pot 1"]
  """
  def randomize(pool) do
  Enum.shuffle(pool)
  end

  @spec contains?(any(), any()) :: boolean()
  @doc """
  Checks if a specific number is included in the pool.

  ## Parameters

  - pool: The list of lottery numbers.
  - number: The number to check for.

  ## Returns

  - `true` if the number is in the pool, otherwise `false`.

  ## Examples

  iex> Lottery.contains?(["Number 1 in Pot 1"], "Number 1 in Pot 1")
  true
  """
  def contains?(pool, number) do
  Enum.member?(pool, number)
  end

  @doc """
  Distributes the pool into two parts based on the specified draw size.

  ## Parameters

  - pool: The list of lottery numbers to be split.
```

```
  - draw_size: The number of numbers to include in the first part.

  ## Returns

  - A tuple with two lists: the first list containing `draw_size` numbers, and
      the second list containing the remaining numbers.

  ## Examples

  iex> Lottery.distribute(["Number 1 in Pot 1", "Number 2 in Pot 2"], 1)
  {["Number 1 in Pot 1"], ["Number 2 in Pot 2"]}
  """
  def distribute(pool, draw_size) do
  Enum.split(pool, draw_size)
  end
  end
```

1.10 Panduan Menjalankan Kode Elixir di Command Prompt

Setelah Anda membuat modul `Lottery` seperti di atas, Anda dapat menjalankan dan menguji fungsinya menggunakan `iex` (Interactive Elixir) di command prompt. Berikut adalah langkah-langkahnya:

1.10.1 Menggunakan `iex -S mix` dan Perintah `recompile`

Perintah `iex -S mix` digunakan untuk memulai shell interaktif Elixir dan memuat konfigurasi serta dependensi proyek yang telah didefinisikan dalam file `mix.exs`. Dengan menggunakan perintah ini, Anda dapat:

- Mengakses modul dan fungsi yang telah Anda buat dalam proyek Elixir.
- Menjalankan dan menguji kode Elixir secara interaktif.
- Memantau hasil dan output dari kode yang dijalankan dalam konteks proyek Anda.

Contoh Penggunaan `iex -S mix`:

1. **Buka Terminal atau Command Prompt** dan arahkan ke direktori proyek Elixir Anda.
2. **Jalankan perintah berikut** untuk memulai shell interaktif dengan konfigurasi proyek:

```
  iex -S mix
```

3. **Gunakan Modul dan Fungsi** yang telah Anda definisikan dalam proyek.

Selain itu, selama pengembangan, Anda mungkin melakukan perubahan pada kode sumber. Untuk memuat ulang kode yang telah diubah tanpa keluar dari shell, Anda dapat menggunakan perintah ‘recompile‘:

Contoh Penggunaan `recompile`:

1. Setelah memulai `iex -S mix`, lakukan perubahan pada kode sumber Anda. 2. Dalam shell `iex`, jalankan perintah berikut untuk mengkompilasi ulang kode:

```
  recompile
```

Perintah ini akan mengkompilasi ulang file yang telah diubah dan memuat ulang modul yang bersangkutan, sehingga Anda dapat langsung melihat perubahan tanpa harus memulai ulang shell interaktif.

Perintah `iex -S mix` dan `recompile` memudahkan pengembangan dan debugging dengan menyediakan lingkungan yang memungkinkan Anda untuk bereksperimen dan menguji kode dengan cepat.

1.10.2 Menjalankan Kode di `iex`

1. Buka terminal atau command prompt.
2. Navigasi ke direktori tempat file `lottery.ex` berada. Misalnya:

```
cd path/to/your/project
```

3. Mulai sesi `iex` dengan perintah berikut:

```
iex
```

4. Di dalam sesi `iex`, muat modul `Lottery` dengan perintah:

```
c("lottery.ex") # atau path ke lottery.ex
```

5. Anda sekarang bisa memanggil fungsi-fungsi dalam modul `Lottery`. Misalnya:

```
Lottery.greet()
Lottery.generate_pool()
Lottery.randomize(["Number 1 in Pot 1", "Number 2 in Pot 2"])
```

1.10.3 Melakukan Reload Setelah Perubahan Kode

Jika Anda mengubah kode di file `lottery.ex`, Anda perlu memuat ulang modul tersebut di `iex` untuk melihat perubahan. Berikut caranya:

1. Simpan perubahan di file `lottery.ex`.
2. Kembali ke sesi `iex` yang sedang berjalan.
3. Muat ulang modul dengan perintah:

```
r(Lottery)
```

4. Fungsi-fungsi dalam modul `Lottery` sekarang akan mencerminkan perubahan yang baru saja Anda buat.

Dengan panduan ini, Anda dapat menjalankan dan menguji modul `Lottery` serta memperbaruinya secara interaktif di `iex`.

1.11 Penjelasan Detail Modul Lottery

1.11.1 Definisi Modul

Modul `Lottery` didefinisikan menggunakan kata kunci `defmodule`. Modul ini bertanggung jawab untuk mengelola sistem undian dan mencakup fungsi-fungsi untuk membuat, mengacak, memeriksa, dan mendistribusikan angka dalam pool undian.

Listing 1.2: Definisi Modul Lottery

```
defmodule Lottery do
@moduledoc """
This module provides functionalities for managing a lottery system.
It includes functions for creating, shuffling, checking for numbers, and
    distributing numbers within the lottery pool.
"""
```

1.11.2 Fungsi Salam

Fungsi `greet/0` mengembalikan pesan salam sederhana. Fungsi ini merupakan contoh dasar dari fungsi tanpa parameter dan memiliki satu nilai balik.

Listing 1.3: Fungsi Salam

```
@spec greet() :: <<_::80>>
@doc """
Returns a greeting message.

## Examples

iex> Lottery.greet()
"Good luck!"
"""
def greet do
"Good luck!"
end
```

1.11.3 Fungsi Pembentukan Pool

Fungsi `generate_pool/0` menghasilkan pool angka undian yang dikombinasikan dengan nomor pot. Fungsi ini mengembalikan daftar yang setiap angkanya dihubungkan dengan pot.

Listing 1.4: Fungsi Pembentukan Pool

```
@spec generate_pool() :: [<<_::24, _::_*16>>, ...]
@doc """
Generates a pool of lottery numbers with different pots.

## Returns

- A list of lottery numbers with their respective pot numbers.

## Examples

iex> Lottery.generate_pool()
["Number 1 in Pot 1", "Number 2 in Pot 1", ...]
"""
def generate_pool do
numbers = ["Number 1", "Number 2", "Number 3", "Number 4", "Number 5", "
    Number 6"]
pots = ["Pot 1", "Pot 2", "Pot 3", "Pot 4"]

# Creates a pool by combining numbers and pots.
for pot <- pots, number <- numbers do
"#{number} in #{pot}"
end
end
```

1.11.4 Fungsi Pengacakan

Fungsi `randomize/1` menerima pool angka undian dan mengembalikan daftar baru dengan angka yang diacak. Fungsi ini menggunakan `Enum.shuffle/1` untuk mengacak urutan.

Listing 1.5: Fungsi Pengacakan

```
@doc """
Randomizes the order of numbers in the pool.

```

```
  ## Parameters

  - pool: The list of lottery numbers to be shuffled.

  ## Returns

  - A new list with the numbers shuffled.

  ## Examples

  iex> Lottery.randomize(["Number 1 in Pot 1", "Number 2 in Pot 2"])
  ["Number 2 in Pot 2", "Number 1 in Pot 1"]
  """
  def randomize(pool) do
  Enum.shuffle(pool)
  end
```

1.11.5 Fungsi Pemeriksaan Angka

Fungsi `contains?/2` memeriksa apakah angka tertentu termasuk dalam pool. Fungsi ini mengembalikan `true` jika angka ditemukan dan `false` jika tidak.

Listing 1.6: Fungsi Pemeriksaan Angka

```
  @spec contains?(any(), any()) :: boolean()
  @doc """
  Checks if a specific number is included in the pool.

  ## Parameters

  - pool: The list of lottery numbers.
  - number: The number to check for.

  ## Returns

  - `true` if the number is in the pool, otherwise `false`.

  ## Examples

  iex> Lottery.contains?(["Number 1 in Pot 1"], "Number 1 in Pot 1")
  true
  """
  def contains?(pool, number) do
  Enum.member?(pool, number)
  end
```

1.11.6 Fungsi Distribusi

Fungsi `distribute/2` membagi pool menjadi dua bagian berdasarkan ukuran undian yang ditentukan. Fungsi ini mengembalikan tuple yang berisi dua daftar: satu dengan angka yang dipilih dan satu lagi dengan angka sisanya.

Listing 1.7: Fungsi Distribusi

```
  @doc """
  Distributes the pool into two parts based on the specified draw size.

  ## Parameters

  - pool: The list of lottery numbers to be split.
```

```
- draw_size: The number of numbers to include in the first part.

## Returns

- A tuple with two lists: the first list containing `draw_size` numbers, and
    the second list containing the remaining numbers.

## Examples

iex> Lottery.distribute(["Number 1 in Pot 1", "Number 2 in Pot 2"], 1)
{["Number 1 in Pot 1"], ["Number 2 in Pot 2"]}
"""
def distribute(pool, draw_size) do
Enum.split(pool, draw_size)
end
```

1.12 Latihan

Bagian ini menyediakan latihan untuk memperdalam pemahaman mengenai String, List, modul ‘Enum‘, serta pengulangan dengan ‘for‘ loop di Elixir. Selesaikan latihan di bawah ini dan uji kode yang Anda tulis di lingkungan ‘iex‘.

1.12.1 Latihan String

Buatlah sebuah fungsi `greet_person/1` yang menerima sebuah nama dalam bentuk string dan mengembalikan pesan sapaan seperti ”Hello, [nama]!”.

```
defmodule StringExercise do
def greet_person(name) do
"Hello, " <> name <> "!"
end
end

# Contoh penggunaan:
StringExercise.greet_person("Elixir")
# Output: "Hello, Elixir!"
```

1.12.2 Latihan List

Buatlah sebuah fungsi `sum_list/1` yang menerima sebuah list angka dan mengembalikan jumlah dari semua angka tersebut.

```
defmodule ListExercise do
def sum_list(numbers) do
Enum.sum(numbers)
end
end

# Contoh penggunaan:
ListExercise.sum_list([1, 2, 3, 4, 5])
# Output: 15
```

1.12.3 Latihan Modul `Enum`

Gunakan fungsi `Enum.shuffle/1` untuk mengacak urutan elemen dalam sebuah list.

```elixir
defmodule EnumExercise do
def shuffle_list(list) do
Enum.shuffle(list)
end
end

# Contoh penggunaan:
EnumExercise.shuffle_list([1, 2, 3, 4, 5])
# Output: [3, 5, 1, 4, 2] (urutan dapat berbeda setiap kali)
```

1.12.4 Latihan `for` Loop Tunggal

Buatlah sebuah `for` loop yang mengiterasi angka dari 1 hingga 10 dan mencetak angka-angka tersebut.

```elixir
for n <- 1..10 do
IO.puts(n)
end

# Output:
# 1
# 2
# 3
# 4
# 5
# 6
# 7
# 8
# 9
# 10
```

1.12.5 Latihan `for` Loop Bersarang

Buatlah sebuah `for` loop bersarang yang menghasilkan semua pasangan karakter dari dua list huruf.

```elixir
for letter1 <- ["A", "B", "C"], letter2 <- ["X", "Y", "Z"] do
{letter1, letter2}
end

# Output:
# [{"A", "X"}, {"A", "Y"}, {"A", "Z"}, {"B", "X"}, {"B", "Y"}, {"B", "Z"}, {"
    C", "X"}, {"C", "Y"}, {"C", "Z"}]
```

1.12.6 Latihan Membuat Sistem Manajemen Inventaris

Modul ini mengelola inventaris barang dengan fungsi-fungsi untuk menambahkan barang, menghapus barang, memeriksa ketersediaan barang, dan mendistribusikan barang ke berbagai lokasi.

Listing 1.8: Modul Manajemen Inventaris

```elixir
defmodule Inventory do
@moduledoc """
Modul ini menyediakan fungsionalitas untuk mengelola sistem inventaris.
Ini mencakup fungsi untuk menambahkan barang, menghapus barang, memeriksa
    ketersediaan barang, dan mendistribusikan barang ke lokasi yang berbeda.
"""

@spec add_item(String.t(), integer()) :: :ok
```

```
8   @doc """
9   Menambahkan barang baru ke inventaris.
10
11  ## Parameter
12
13  - item: Nama barang.
14  - quantity: Jumlah barang yang akan ditambahkan.
15
16  ## Examples
17
18  iex> Inventory.add_item("Laptop", 10)
19  :ok
20  """
21  def add_item(item, quantity) do
22  IO.puts("Menambahkan #{quantity} #{item} ke inventaris.")
23  :ok
24  end
25
26  @spec remove_item(String.t(), integer()) :: :ok
27  @doc """
28  Menghapus barang dari inventaris.
29
30  ## Parameter
31
32  - item: Nama barang.
33  - quantity: Jumlah barang yang akan dihapus.
34
35  ## Examples
36
37  iex> Inventory.remove_item("Laptop", 5)
38  :ok
39  """
40  def remove_item(item, quantity) do
41  IO.puts("Menghapus #{quantity} #{item} dari inventaris.")
42  :ok
43  end
44
45  @spec check_availability(String.t()) :: integer()
46  @doc """
47  Memeriksa ketersediaan barang dalam inventaris.
48
49  ## Parameter
50
51  - item: Nama barang yang ingin diperiksa.
52
53  ## Returns
54
55  - Jumlah barang yang tersedia.
56
57  ## Examples
58
59  iex> Inventory.check_availability("Laptop")
60  10
61  """
62  def check_availability(item) do
63  IO.puts("Memeriksa ketersediaan #{item}.")
64  10
65  end
66
67  @spec distribute_items([String.t()], String.t()) :: :ok
68  @doc """
```

```
Mendistribusikan barang ke lokasi yang berbeda.

## Parameter

- items: Daftar barang yang akan didistribusikan.
- location: Lokasi tujuan distribusi.

## Examples

iex> Inventory.distribute_items(["Laptop", "Mouse"], "Gudang A")
:ok
"""
def distribute_items(items, location) do
IO.puts("Mendistribusikan barang ke #{location}.")
:ok
end
end
```

1.12.7 Latihan Membuat Sistem Pendaftaran Kelas

Modul ini mengelola pendaftaran siswa untuk kelas dengan fungsi-fungsi untuk mendaftar siswa, membatalkan pendaftaran, memeriksa pendaftaran, dan mengatur jadwal kelas.

Listing 1.9: Modul Pendaftaran Kelas

```
defmodule ClassRegistration do
@moduledoc """
Modul ini menyediakan fungsionalitas untuk mengelola sistem pendaftaran kelas.
Ini mencakup fungsi untuk mendaftar siswa, membatalkan pendaftaran, memeriksa pendaftaran, dan mengatur jadwal kelas.
"""

@spec register_student(String.t(), String.t()) :: :ok
@doc """
Mendaftar siswa ke kelas.

## Parameter

- student: Nama siswa.
- class: Nama kelas yang akan diikuti.

## Examples

iex> ClassRegistration.register_student("Alice", "Matematika")
:ok
"""
def register_student(student, class) do
IO.puts("Mendaftar #{student} ke kelas #{class}.")
:ok
end

@spec cancel_registration(String.t(), String.t()) :: :ok
@doc """
Membatalkan pendaftaran siswa dari kelas.

## Parameter

- student: Nama siswa.
- class: Nama kelas yang akan dibatalkan.

```

```
  ## Examples

  iex> ClassRegistration.cancel_registration("Alice", "Matematika")
  :ok
  """
  def cancel_registration(student, class) do
  IO.puts("Membatalkan pendaftaran #{student} dari kelas #{class}.")
  :ok
  end

  @spec check_registration(String.t()) :: [String.t()]
  @doc """
  Memeriksa kelas yang diikuti oleh siswa.

  ## Parameter

  - student: Nama siswa yang ingin diperiksa.

  ## Returns

  - Daftar kelas yang diikuti oleh siswa.

  ## Examples

  iex> ClassRegistration.check_registration("Alice")
  ["Matematika", "Fisika"]
  """
  def check_registration(student) do
  IO.puts("Memeriksa pendaftaran untuk #{student}.")
  ["Matematika", "Fisika"]
  end

  @spec schedule_class(String.t(), String.t()) :: :ok
  @doc """
  Mengatur jadwal untuk kelas.

  ## Parameter

  - class: Nama kelas.
  - schedule: Jadwal kelas.

  ## Examples

  iex> ClassRegistration.schedule_class("Matematika", "Senin, 09:00")
  :ok
  """
  def schedule_class(class, schedule) do
  IO.puts("Mengatur jadwal kelas #{class} ke #{schedule}.")
  :ok
  end
  end
```

1.13 Soal: Mengembangkan Sistem Kuis dalam Elixir

Pada latihan ini, Anda diminta untuk membuat sebuah modul Elixir yang bernama `Quiz` dengan tujuan untuk mengelola sistem kuis. Modul ini harus memiliki fungsi-fungsi berikut:

1. `generate_questions/0`: Fungsi ini bertugas untuk menghasilkan daftar pertanyaan kuis. Setiap pertanyaan harus memiliki opsi jawaban yang berbeda-beda.

2. `randomize_questions/1`: Fungsi ini menerima daftar pertanyaan yang dihasilkan dari fungsi sebelumnya dan mengacak urutan pertanyaan tersebut.

3. `check_answer/3`: Fungsi ini bertugas untuk memeriksa apakah jawaban yang diberikan oleh peserta kuis sesuai dengan jawaban yang benar. Parameter yang diterima adalah daftar pertanyaan, jawaban peserta, dan jawaban yang benar.

4. `score_quiz/2`: Fungsi ini menerima daftar jawaban peserta dan daftar jawaban yang benar, kemudian menghitung skor akhir berdasarkan jumlah jawaban yang benar.

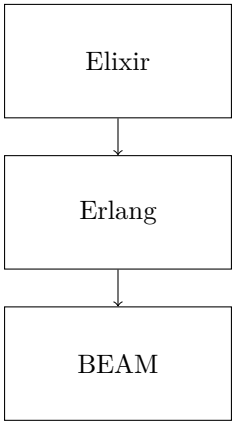
Tugas: Implementasikan modul `Quiz` dalam bahasa Elixir, dan berikan contoh cara memanggil fungsi-fungsi tersebut di dalam `iex`.

Chapter 2

Pattern Matching

2.1 Hubungan antara Elixir, Erlang, dan BEAM

Diagram berikut menggambarkan hubungan antara Elixir, Erlang, dan BEAM:



2.1.1 Elixir

Elixir adalah bahasa pemrograman yang modern dan dinamis, dikembangkan untuk memenuhi kebutuhan aplikasi terdistribusi yang skalabel dan fault-tolerant. Meskipun Elixir memiliki sintaks yang berbeda dan modern, ia bergantung sepenuhnya pada ekosistem Erlang untuk menjalankan aplikasinya.

2.1.2 Erlang

Erlang adalah bahasa pemrograman yang mendasari Elixir. Ketika kode Elixir dikompilasi, ia diubah menjadi bytecode Erlang. Ini memungkinkan Elixir untuk memanfaatkan seluruh ekosistem Erlang, termasuk pustaka, alat, dan framework yang sudah ada. Dengan kata lain, Elixir adalah lapisan di atas Erlang yang menyediakan sintaks dan fitur tambahan sambil tetap menggunakan fondasi yang kuat dari Erlang.

2.1.3 BEAM

BEAM (Bogdan/Björn's Erlang Abstract Machine) adalah mesin virtual yang menjalankan bytecode Erlang, termasuk kode yang ditulis dalam Elixir. BEAM dirancang untuk mendukung concurrency, fault-tolerance, dan distribusi yang dibutuhkan oleh aplikasi Elixir dan Erlang. BEAM adalah komponen inti yang membuat Elixir dan Erlang mampu menangani jutaan proses secara efisien.

2.1.4 Hubungan dalam Diagram

Diagram di atas menunjukkan bagaimana Elixir bergantung pada Erlang dan akhirnya dijalankan di atas BEAM. Ketika seorang pengembang menulis kode dalam Elixir, kode tersebut pertama-tama diterjemahkan menjadi bytecode Erlang. Selanjutnya, bytecode tersebut dijalankan oleh BEAM, yang mengelola eksekusi program secara efisien. Ini berarti meskipun Elixir dan Erlang adalah bahasa yang berbeda, mereka berbagi mesin runtime yang sama, yaitu BEAM, yang membuat mereka sangat kompatibel dan interoperabel.

2.2 Pattern Matching di Elixir

Pattern matching adalah salah satu fitur paling kuat dan fundamental dalam bahasa pemrograman Elixir. Fitur ini memungkinkan pengembang untuk mencocokkan struktur data dan mengekstrak nilai-nilai dari struktur tersebut secara deklaratif. Tidak seperti bahasa pemrograman imperatif di mana variabel diinisialisasi dengan nilai, dalam Elixir, pattern matching berfungsi sebagai alat untuk membandingkan dan mengurai data.

2.2.1 Dasar-Dasar Pattern Matching

Pada dasarnya, pattern matching menggunakan operator `=` untuk mencocokkan sisi kiri dan sisi kanan dari ekspresi. Jika keduanya cocok, maka Elixir akan mengikat nilai dari sisi kanan ke variabel di sisi kiri.

```
iex> x = 1
1

iex> 1 = x
1
```

Dalam contoh di atas, nilai 1 di sisi kanan diikat ke variabel `x`. Karena sisi kiri dan kanan dari ekspresi `1 = x` cocok, Elixir hanya mengembalikan `1`.

2.2.2 Pattern Matching dengan Tuple

Pattern matching sangat berguna ketika bekerja dengan struktur data yang lebih kompleks seperti tuple.

```
iex> {a, b, c} = {1, 2, 3}
{1, 2, 3}

iex> a
1

iex> b
2

iex> c
3
```

Dalam contoh di atas, tuple `{1, 2, 3}` dicocokkan dengan pola `{a, b, c}`, sehingga nilai-nilai di dalam tuple diikat ke variabel `a`, `b`, dan `c`.

2.2.3 Pattern Matching dengan List

Pattern matching juga dapat digunakan dengan list, termasuk penggunaan `head` dan `tail` untuk mencocokkan bagian pertama dari list dan sisa elemennya.

```
iex> [head | tail] = [1, 2, 3]
[1, 2, 3]

```

```
iex> head
1

iex> tail
[2, 3]
```

Di sini, `head` mendapatkan nilai pertama dari list, sedangkan `tail` mendapatkan list yang tersisa.

2.2.4 Menggunakan Pattern Matching dalam Fungsi

Pattern matching dapat digunakan dalam definisi fungsi untuk membuat kode yang lebih bersih dan mudah dibaca.

```
defmodule Example do
def greet({first_name, last_name}) do
"Hello, #{first_name} #{last_name}!"
end
end

iex> Example.greet({"John", "Doe"})
"Hello, John Doe!"
```

Pada contoh ini, fungsi `greet/1` menerima tuple yang terdiri dari `first_name` dan `last_name`. Nilai-nilai ini langsung dicocokkan dengan pola yang didefinisikan dalam parameter fungsi.

2.2.5 Pattern Matching dengan Pengkondisian

Berikut adalah contoh penggunaan `case` dengan pattern matching di dalam modul dan fungsi Elixir:

```
defmodule Example do
  def match_pattern(tuple) do
    case tuple do
    {1, x, 3} -> IO.puts("x adalah #{x}")
    _ -> IO.puts("Tidak ada kecocokan")
    end
  end
end

# Pemanggilan fungsi
Example.match_pattern({1, 2, 3})
```

Pada contoh ini, modul `Example` berisi fungsi `match_pattern/1` yang melakukan pattern matching terhadap tuple yang diterima sebagai argumen. Jika tuple sesuai dengan pola `{1, x, 3}`, maka nilai `x` dicetak. Jika tidak, pesan `"Tidak ada kecocokan"` akan ditampilkan.

```
x adalah 2
```

Pattern matching di Elixir memungkinkan pemrograman yang lebih deklaratif dan ekspresif. Dengan kemampuan untuk mencocokkan dan mengurai struktur data, pattern matching menjadi salah satu fitur esensial yang mempermudah pengembangan aplikasi dalam Elixir.

2.3 Pattern Pipe Operator di Elixir

Pipe operator (`|>`) adalah salah satu fitur yang sangat berguna di Elixir, yang memungkinkan chaining atau penggabungan dari beberapa operasi menjadi satu alur yang mudah dibaca. Pipe operator mengambil output dari ekspresi sebelumnya dan meneruskannya sebagai argumen pertama ke fungsi berikutnya.

2.3.1 Dasar-Dasar Pipe Operator

Pada dasarnya, pipe operator memungkinkan kita untuk menulis kode yang lebih rapi dan berurutan daripada harus menyusun fungsi secara bersarang (nested).

```
iex> "hello" # tekan alt + Enter untuk pindah ke baris berikutnya
  |> String.upcase()
  |> String.reverse()
"OLLEH"
```

Dalam contoh di atas, string `"hello"` pertama-tama diubah menjadi huruf kapital dengan `String.upcase/1`, kemudian hasilnya diteruskan ke `String.reverse/1` yang membalikkan urutan karakter. Tanpa pipe operator, kode ini akan terlihat seperti berikut:

```
iex> String.reverse(String.upcase("hello"))
"OLLEH"
```

2.3.2 Pipe Operator dengan List

Pipe operator juga sering digunakan dengan fungsi-fungsi yang memanipulasi list, seperti pada contoh berikut:

```
iex> [1, 2, 3, 4, 5] # tekan alt + Enter untuk pindah ke baris berikutnya
  |> Enum.map(&(&1 * 2))
  |> Enum.filter(&(&1 > 5))
[6, 8, 10]
```

Pada contoh ini, list `[1, 2, 3, 4, 5]` pertama-tama dikalikan dengan 2 menggunakan `Enum.map/2`, kemudian hasilnya disaring dengan `Enum.filter/2` untuk hanya menyertakan angka yang lebih besar dari 5.

2.3.3 Penjelasan '&1', '&2', dan Seterusnya

Dalam Elixir, '&' digunakan untuk membuat anonymous functions atau fungsi tanpa nama. Dalam anonymous function, '&1', '&2', dan seterusnya adalah placeholder untuk argumen yang diterima oleh fungsi tersebut.

```
iex> add = &(&1 + &2)
#Function<6.86581258/2 in :erl_eval.expr/5>
iex> add.(2, 3)
5
```

Pada contoh ini, '&1' dan '&2' mewakili argumen pertama dan kedua dari anonymous function yang dibuat. Fungsi ini menambahkan kedua argumen dan mengembalikan hasilnya.

Pipe operator di Elixir mempermudah penulisan kode yang jelas dan mudah dibaca, terutama ketika menggabungkan serangkaian operasi. Selain itu, kemampuan untuk menggunakan pipe operator dengan fungsi yang menerima banyak parameter serta penggunaan '&1', '&2', dan seterusnya memungkinkan penulisan kode yang lebih fleksibel dan ekspresif.

2.3.4 Pipe Operator dengan Fungsi yang Memiliki Banyak Parameter

Pipe operator juga dapat digunakan dengan fungsi yang memerlukan lebih dari satu parameter. Berikut adalah contohnya:

```
defmodule Example do
def multiply_and_add(x, y, z) do
x * y + z
end
end

iex> 5
```

```
  |> Example.multiply_and_add(2, 3)
13
```

Dalam contoh di atas, angka `5` diteruskan sebagai parameter pertama ke fungsi `multiply_and_add/3`, dan parameter kedua dan ketiga adalah `2` dan `3`. Fungsi ini mengalikan `5` dengan `2` dan menambahkan `3`, menghasilkan `13`.

2.3.5 Contoh dengan Konversi Tipe Data

Dalam Elixir, ketika kita menggunakan pipe operator dengan fungsi yang memerlukan parameter dengan tipe yang berbeda dari output fungsi sebelumnya, kita harus memastikan bahwa data yang diteruskan sesuai dengan tipe yang diharapkan oleh fungsi tersebut. Ini mungkin memerlukan konversi atau pemrosesan data sebelum menggunakan pipe operator.

Misalkan kita memiliki fungsi yang mengharapkan parameter bertipe integer dan fungsi lain yang menghasilkan string. Kita perlu mengkonversi string menjadi integer sebelum meneruskan ke fungsi berikutnya.

Berikut adalah contoh sederhana:

```
defmodule Converter do
def string_to_integer(str) do
String.to_integer(str)
end

def add_five(num) do
num + 5
end
end

iex> "42"
  |> Converter.string_to_integer()
  |> Converter.add_five()
47
```

Pada contoh ini, string `"42"` pertama-tama dikonversi menjadi integer dengan `Converter.string_to_integer/1`, kemudian hasil integer tersebut diteruskan ke `Converter.add_five/1` untuk ditambahkan dengan 5.

2.3.6 Menggunakan Nilai Pipe sebagai Parameter Kedua

Di Elixir, kita dapat menggunakan pipe operator untuk meneruskan nilai dari fungsi pertama sebagai parameter kedua dalam sebuah fungsi dengan memanfaatkan fungsi anonim. Contoh berikut menunjukkan cara melakukannya:

```
defmodule Example do
def hello(greet, name) do
greet <> name
end
end

iex> "world"
  |> (&Example.hello("Hello, ", &1 )).()
"Hello, world"
```

Pada kode di atas, modul `Example` mendefinisikan fungsi `hello/2` yang menggabungkan dua string: `greet` dan `name`. Nilai `"world"` diteruskan melalui pipe operator ke fungsi anonim. Fungsi anonim tersebut menggunakan `"Hello, "` sebagai parameter pertama dan nilai dari pipe (`"world"`) sebagai parameter kedua.

Fungsi `hello/2` kemudian menggabungkan `"Hello, "` dengan `"world"`, menghasilkan string `"Hello, world"`. Dengan pendekatan ini, pipe operator dan fungsi anonim memungkinkan kita untuk dengan mudah mengatur parameter fungsi, termasuk menggunakan nilai dari pipe sebagai parameter kedua. Pendekatan ini meningkatkan fleksibilitas dan keterbacaan kode dalam Elixir.

2.4 Menyimpan dan Memuat Ulang Nilai ke dan dari File

Elixir menyediakan berbagai metode untuk menyimpan dan memuat ulang data dari file. Ini berguna untuk menyimpan konfigurasi, hasil pemrosesan, atau data lainnya secara persisten. Berikut adalah cara melakukannya dengan menggunakan Elixir.

2.4.1 Menyimpan Data ke File

Untuk menyimpan data ke file, Anda dapat menggunakan fungsi ‘File.write/2‘. Fungsi ini menerima nama file dan data yang akan ditulis. Berikut adalah contoh cara menyimpan string ke file:

```
filename = "data.txt"
data = "Hello, Elixir!"

File.write(filename, data)
```

Pada contoh di atas, string `"Hello, Elixir!"` disimpan ke dalam file `"data.txt"`.

2.4.2 Memuat Data dari File

Untuk memuat data dari file, Anda dapat menggunakan fungsi ‘File.read/1‘. Fungsi ini membaca isi file dan mengembalikan hasilnya sebagai string. Berikut adalah contoh cara memuat data dari file:

```
# File: lib/file_reader.ex

defmodule FileReader do
def read_file(filename) do
case File.read(filename) do
{:ok, content} ->
IO.puts("File content: #{content}")
{:error, reason} ->
IO.puts("Failed to read file: #{reason}")
end
end
end
```

Perintah pada `iex` command prompt:

```
FileReader.read_file("data.txt")
```

Pada contoh di atas, fungsi ‘File.read/1‘ membaca isi file `"data.txt"`. Jika berhasil, isi file ditampilkan ke konsol; jika terjadi kesalahan, pesan kesalahan akan ditampilkan.

2.4.3 Menambahkan Dependensi `{:jason, "~ 1.4"}`

Untuk menambahkan library `Jason` ke dalam proyek Elixir, langkah-langkah berikut perlu dilakukan:

1. Buka file `mix.exs` di root direktori proyek Elixir Anda.
2. Di dalam fungsi `deps/0`, tambahkan dependensi `{:jason, "~ 1.4"}`.

Berikut adalah contoh bagaimana menambahkan `Jason` dalam file `mix.exs`:

```
defp deps do
[
{:jason, "~> 1.4"}
]
end
```

- Library `Jason` digunakan untuk melakukan parsing dan encoding JSON dengan performa tinggi di Elixir.
- Setelah menambahkan dependensi tersebut, jalankan perintah `mix deps.get` untuk mengunduh dan menginstal library yang diperlukan.

Dengan langkah-langkah ini, proyek Elixir Anda siap menggunakan `Jason` untuk bekerja dengan data JSON.

2.4.4 Menyimpan dan Memuat Data dengan Format Lain

Jika Anda bekerja dengan data yang lebih kompleks, seperti objek atau struktur data, Anda mungkin ingin menggunakan format lain, seperti JSON atau Erlang Term Storage (ETS). Berikut adalah contoh cara menggunakan format JSON untuk menyimpan dan memuat data:

```
# File: lib/json_file_handler.ex

defmodule JsonFileHandler do
# Save data to a JSON file
def save_json(filename, data) do
File.write(filename, Jason.encode!(data))
end

# Load data from a JSON file
def load_json(filename) do
case File.read(filename) do
{:ok, content} ->
case Jason.decode(content) do
{:ok, decoded_data} ->
IO.inspect(decoded_data)
{:error, reason} ->
IO.puts("Failed to decode JSON: #{reason}")
end
{:error, reason} ->
IO.puts("Failed to read file: #{reason}")
end
end
end
```

Perintah pada `iex` command prompt:

```
filename="data.json"
data='{"greeting": "Hello, Elixir!", "count": 42}'
JsonFileHandler.save_json(filename, data)
JsonFileHandler.load_json(filename)
```

Pada contoh di atas, kita menggunakan pustaka ‘Jason‘ untuk mengkodekan dan mendekode data JSON. Fungsi ‘Jason.encode!/1‘ mengubah data menjadi format JSON, dan ‘Jason.decode/1‘ mengembalikannya ke bentuk asli.

Dengan menggunakan fungsi-fungsi dari modul ‘File‘ dan pustaka tambahan seperti ‘Jason‘, Anda dapat dengan mudah menyimpan dan memuat data dari file di Elixir. Ini memungkinkan pengelolaan data yang persisten dan interoperabilitas dengan berbagai format data.

2.5 Latihan

Pada bagian ini, terdapat beberapa latihan yang bertujuan untuk menguji pemahaman mengenai kode-kode yang telah dipelajari. Setiap latihan berisi kode yang harus dijalankan dan dipahami hasilnya.

2.5.1 Latihan 1: Pattern Matching di Elixir

Latihan pertama ini akan mengeksplorasi bagaimana pattern matching bekerja di Elixir. Anda akan diminta untuk memahami dan menjalankan contoh kode di bawah ini.

```
# Contoh Pattern Matching Sederhana
{x, y, z} = {1, 2, 3}
IO.puts("x = #{x}, y = #{y}, z = #{z}")

# Pattern Matching dengan List
[head | tail] = [1, 2, 3, 4, 5]
IO.puts("Head: #{head}")
IO.inspect(tail)

# Pattern Matching dengan Map
%{name: name, age: age} = %{name: "John", age: 30}
IO.puts("Name: #{name}, Age: #{age}")
```

2.5.2 Latihan 2: Pipe Operator di Elixir

Latihan kedua ini akan berfokus pada penggunaan pipe operator (`|>`) di Elixir. Jalankan dan pahami bagaimana nilai dapat diteruskan dari satu fungsi ke fungsi lainnya.

```
# Contoh Penggunaan Pipe Operator
defmodule PipeExample do
def greet(name), do: "Hello, " <> name
def exclaim(statement), do: statement <> "!"
def emphasize(statement), do: String.upcase(statement)
end

"world"
|> PipeExample.greet()
|> PipeExample.exclaim()
|> PipeExample.emphasize()
|> IO.puts()
```

Selain itu, berikut contoh pipe operator yang memindahkan nilai ke parameter kedua:

```
defmodule Example do
def wrap_in_brackets(prefix, content), do: prefix <> "[" <> content <> "]"
end

"world"
|> (&Example.wrap_in_brackets(&1, "Hello")).()
|> IO.puts()
```

`&1` merujuk pada nilai yang diteruskan melalui pipe operator. Variasi lainnya seperti `&2` merujuk pada parameter kedua, dan seterusnya.

2.5.3 Latihan 3: Membaca File Teks

Latihan ini melibatkan penggunaan modul `FileReader` untuk membaca konten dari sebuah file teks. Pastikan file `data.txt` tersedia dengan beberapa konten di dalamnya.

```
# File: lib/file_reader.ex

defmodule FileReader do
def read_file(filename) do
case File.read(filename) do
{:ok, content} ->
IO.puts("File content: #{content}")
```

```
  {:error, reason} ->
  IO.puts("Failed to read file: #{reason}")
  end
  end
  end
```

```
iex>  FileReader.read_file("data.txt")\part{title}
```

2.5.4 Latihan 4: Menyimpan dan Membaca Data JSON

Latihan ini menggunakan modul `JsonFileHandler` untuk menyimpan dan membaca data dalam format JSON. Anda akan melihat bagaimana data JSON dapat disimpan ke file dan kemudian dimuat kembali.

```
# File: lib/json_file_handler.ex

defmodule JsonFileHandler do
# Save data to a JSON file
def save_json(filename, data) do
File.write(filename, Jason.encode!(data))
end

# Load data from a JSON file
def load_json(filename) do
case File.read(filename) do
{:ok, content} ->
case Jason.decode(content) do
{:ok, decoded_data} ->
IO.inspect(decoded_data)
{:error, reason} ->
IO.puts("Failed to decode JSON: #{reason}")
end
{:error, reason} ->
IO.puts("Failed to read file: #{reason}")
end
end
end
```

```
iex> filename="data.json"
iex> data='{"greeting": "Hello, Elixir!", "count": 42}'
iex> JsonFileHandler.save_json(filename, data)
iex> JsonFileHandler.load_json(filename)
```

2.5.5 Latihan 5: Menggabungkan Semua Konsep

Dalam latihan ini, buatlah sebuah modul yang memanfaatkan pattern matching, pipe operator, serta pembacaan dan penulisan file JSON. Fungsi dalam modul ini:

- Membaca data dari sebuah file JSON.
- Menggunakan pattern matching untuk mengekstrak nilai tertentu dari data JSON yang diambil.
- Memanipulasi data tersebut menggunakan pipe operator.
- Menyimpan hasilnya kembali ke file JSON yang sama.

Berikut adalah struktur dasar untuk memulai:

```elixir
defmodule IntegratedExercise do
def process_file(filename) do
# Membaca file JSON
filename
|> File.read()                # Baca file
|> case do                    # Gunakan pattern matching untuk memproses
    konten
{:ok, content} ->
# Decode JSON dan ekstrak nilai menggunakan pattern matching
case Jason.decode(content) do
{:ok, %{"greeting" => greeting, "count" => count}} ->
# Manipulasi data menggunakan pipe operator
new_count = count + 1
new_data = %{"greeting" => greeting, "count" => new_count}

# Simpan data baru ke file JSON
File.write(filename, Jason.encode!(new_data))
{:error, reason} ->
IO.puts("Failed to decode JSON: #{reason}")
end
{:error, reason} ->
IO.puts("Failed to read file: #{reason}")
end
end
end

# Memanggil fungsi
filename = "data.json"
IntegratedExercise.process_file(filename)
```

Instruksi:

1. Buat file `data.json` dengan struktur: `{"greeting": "Hello, Elixir!", "count": 42}`.
2. Modifikasi fungsi `process_file` untuk melakukan operasi tambahan, seperti menambahkan data baru atau mengubah nilai tertentu.
3. Pastikan hasil akhirnya disimpan kembali ke dalam file JSON yang sama.

2.6 Memperluas Modul Lottery

Dalam bagian ini, kami memperluas fungsionalitas modul `Lottery` dengan menambahkan beberapa fitur baru. Fitur-fitur ini mencakup menyimpan dan memuat pool lotere ke dan dari file, serta membuat tangan lotere yang acak. Kode berikut menunjukkan peningkatan-peningkatan ini:

```elixir
defmodule Lottery do
def generate_pool do
numbers = ["Number 1", "Number 2", "Number 3", "Number 4", "Number 5"]
pots = ["Pot 1", "Pot 2", "Pot 3", "Pot 4"]

# Membuat pool dengan menggabungkan angka dan pot.
for pot <- pots, number <- numbers do
"#{number} in #{pot}"
end
end

def randomize(pool) do
Enum.shuffle(pool)
end

```

```
16  def contains?(pool, number) do
17  Enum.member?(pool, number)
18  end
19
20  def distribute(pool, draw_size) do
21  Enum.split(pool, draw_size)
22  end
23
24  def save_pool(pool, filename) do
25  binary = :erlang.term_to_binary(pool)
26  File.write(filename, binary)
27  end
28
29  def load_pool(filename) do
30  case File.read(filename) do
31  {:ok, binary} -> :erlang.binary_to_term(binary)
32  {:error, _reason} -> "File tersebut tidak ada"
33  end
34  end
35
36  def create_hand(draw_size) do
37  Lottery.generate_pool()
38  |> Lottery.randomize()
39  |> Lottery.distribute(draw_size)
40  end
41  end
```

2.6.1 Deskripsi Fungsi

- `save_pool/2`: Fungsi ini mengkonversi pool lotere menjadi format biner dan menyimpannya ke dalam file. File ini kemudian dapat digunakan untuk mempertahankan status pool.
- `load_pool/1`: Fungsi ini membaca file biner dan mengkonversinya kembali ke dalam format pool lotere. Jika file tidak ditemukan, fungsi ini akan mengembalikan pesan kesalahan.
- `create_hand/1`: Fungsi ini membuat tangan lotere yang acak dengan menggabungkan fungsi-fungsi yang telah didefinisikan sebelumnya. Fungsi ini membuat pool, merandomnya, dan kemudian membaginya sesuai dengan ukuran undian yang ditentukan.

Chapter 3

Dokumentasi dan Unit Test pada Elixir

3.1 Dokumentasi di Elixir

Elixir mendukung dokumentasi yang mudah untuk modul dan fungsi. Dokumentasi ini dapat ditambahkan langsung dalam kode menggunakan komentar khusus yang disebut dengan `@moduledoc` untuk modul dan `@doc` untuk fungsi. Selain itu, Elixir juga menyediakan cara mudah untuk menggenerate dokumentasi dalam bentuk HTML menggunakan perintah `mix docs`.

3.1.1 Dokumentasi pada Level Modul

Untuk mendokumentasikan sebuah modul, digunakan anotasi `@moduledoc`. Ini adalah tempat yang tepat untuk memberikan deskripsi tentang tujuan modul, bagaimana modul tersebut digunakan, serta contoh-contoh jika diperlukan.

```
defmodule ExampleModule do
@moduledoc """
This module is an example of how to document a module in Elixir.

It serves to demonstrate how module documentation can be written
and used as a reference when generating HTML documentation.
"""

# Other functions and logic can be written here
end
```

3.1.2 Dokumentasi pada Level Fungsi

Selain mendokumentasikan modul, Anda juga bisa mendokumentasikan setiap fungsi dengan menggunakan anotasi `@doc`. Dokumentasi fungsi biasanya memberikan penjelasan tentang apa yang dilakukan fungsi tersebut, parameter yang dibutuhkan, dan hasil yang dikembalikan.

```
defmodule ExampleModule do
@moduledoc """
This module is an example of how to document a module in Elixir.
"""

@doc """
This function adds two numbers.

## Parameters
- a: the first number.
- b: the second number.
```

```
12
13  ## Example
14
15  iex> ExampleModule.add(2, 3)
16  5
17  """
18  def add(a, b) do
19  a + b
20  end
21  end
```

3.2 Menambahkan `ex_doc` untuk Dokumentasi

Untuk menghasilkan dokumentasi dalam proyek Elixir, Anda perlu menambahkan library `ex_doc`. Library ini memungkinkan Anda untuk menghasilkan dokumentasi dalam berbagai format, termasuk HTML. Berikut adalah langkah-langkah untuk menambahkan `ex_doc` ke dalam proyek Elixir Anda.

3.2.1 Langkah-langkah Menambahkan `ex_doc`

1. Buka file `mix.exs` di proyek Anda, dan tambahkan `ex_doc` ke dalam daftar dependensi di fungsi `deps`. Pastikan Anda menambahkan dependensi ini hanya untuk lingkungan pengembangan (`dev`) dan tidak di runtime.

```
    defp deps do
    [
      {:ex_doc, "~> 0.34"}
    ]
    end
```

2. Setelah menambahkan `ex_doc`, jalankan perintah berikut di terminal untuk mengunduh dan menginstal dependensi:

```
    mix deps.get
```

3. Setelah dependensi berhasil diinstal, Anda dapat menghasilkan dokumentasi untuk proyek Anda dengan perintah berikut:

```
    mix docs
```

4. Dokumentasi yang dihasilkan akan disimpan dalam folder `doc/` di dalam direktori proyek Anda. Anda dapat membuka file `index.html` di browser untuk melihat dokumentasi dalam format HTML.

3.3 Mengenerate Dokumentasi HTML

Untuk menggenerate dokumentasi dalam format HTML, Elixir menyediakan perintah sederhana, yaitu `mix docs`. Perintah ini akan mencari dokumentasi di dalam modul dan fungsi yang ada, lalu mengubahnya menjadi halaman HTML yang mudah dibaca. Langkah-langkahnya adalah sebagai berikut:

1. Pastikan Anda berada di dalam direktori proyek Elixir.
2. Jalankan perintah berikut pada *command prompt*:

```
mix docs
```

3. Dokumentasi HTML akan dihasilkan di dalam folder `doc`.

Dengan mendokumentasikan kode dengan baik dan mengenerate dokumentasi HTML, Anda dapat mempermudah orang lain (atau diri sendiri di masa depan) untuk memahami kode yang Anda tulis.

3.4 Unit Test di Elixir

Unit testing di Elixir dapat dilakukan menggunakan modul bawaan bernama `ExUnit`. Elixir menyediakan fitur untuk membuat test dalam file khusus serta memungkinkan test ditulis dalam dokumentasi fungsi menggunakan anotasi `doctest`. Unit test sangat penting untuk memastikan bahwa fungsi-fungsi dalam kode bekerja sesuai harapan.

3.4.1 Kesimpulan

Dengan menambahkan `ex_doc`, Anda dapat dengan mudah menghasilkan dokumentasi untuk proyek Elixir Anda. Dokumentasi ini membantu memudahkan pemahaman tentang kode yang ditulis dan berfungsi sebagai panduan bagi pengguna lain yang ingin menggunakan atau mengembangkan proyek tersebut.

3.4.2 Unit Test dalam File Khusus

Untuk membuat unit test di Elixir, biasanya digunakan file khusus yang ditempatkan dalam folder `test`. File test ini harus memiliki akhiran `_test.exs`. Di dalam file tersebut, Anda mendefinisikan modul yang menggunakan `ExUnit.Case` sebagai template.

```
defmodule ExampleModuleTest do
use ExUnit.Case

test "adds two numbers" do
assert ExampleModule.add(2, 3) == 5
end

test "subtracts two numbers" do
assert ExampleModule.subtract(5, 2) == 3
end
end
```

3.4.3 Unit Test dalam Dokumentasi Fungsi (Doctest)

Selain menulis test di file khusus, Elixir juga memungkinkan Anda untuk menulis unit test langsung dalam dokumentasi fungsi menggunakan anotasi `doctest`. Test ini secara otomatis dijalankan ketika Anda menjalankan `ExUnit`. Contoh doctest dapat ditulis dalam bagian `@doc` dari suatu fungsi.

```
defmodule ExampleModule do
@moduledoc """
This module demonstrates how to document and test functions.
"""

@doc """
This function adds two numbers.

## Parameters
- a: the first number.
- b: the second number.

## Example
```

```
14
15  iex> ExampleModule.add(2, 3)
16  5
17  """
18  def add(a, b) do
19  a + b
20  end
21
22  @doc """
23  This function subtracts two numbers.
24
25  ## Parameters
26  - a: the first number.
27  - b: the second number.
28
29  ## Example
30
31  iex> ExampleModule.subtract(5, 2)
32  3
33  """
34  def subtract(a, b) do
35  a - b
36  end
37  end
```

Untuk mengaktifkan `doctest`, Anda perlu menambahkannya dalam modul test:

```
1  defmodule ExampleModuleTest do
2  use ExUnit.Case
3  doctest ExampleModule
4  end
```

3.4.4 Menjalankan Test dari Command Prompt

Elixir memudahkan Anda untuk menjalankan test langsung dari command prompt. Ada beberapa opsi untuk menjalankan test tergantung kebutuhan Anda.

Menjalankan Semua Test

Untuk menjalankan semua test yang ada di proyek, cukup gunakan perintah berikut:

```
1  mix test
```

Perintah ini akan mengeksekusi semua test yang ada di folder `test` dan menampilkan hasilnya di terminal.

Menjalankan Test Spesifik untuk Modul Tertentu

Jika Anda hanya ingin menjalankan test untuk modul tertentu, Anda dapat menyebutkan nama file test yang bersangkutan. Misalnya, untuk menjalankan test pada `ExampleModuleTest`, jalankan perintah berikut:

```
1  mix test test/example_module_test.exs
```

Menjalankan Test Spesifik untuk Fungsi Tertentu

Untuk menjalankan test yang spesifik untuk suatu fungsi dalam modul, Anda dapat menggunakan tag line number. Misalnya, jika test untuk fungsi `add` ada pada baris 4 dalam file test, Anda bisa menjalankan perintah berikut:

```
mix test test/example_module_test.exs:4
```

Jika Anda ingin menjalankan test untuk fungsi lain seperti `subtract` yang ada di baris 8, Anda cukup menjalankan:

```
mix test test/example_module_test.exs:8
```

Dengan cara ini, hanya test yang ada pada baris tersebut yang akan dijalankan, sehingga memudahkan dalam proses debugging dan pengujian parsial.

Dengan menggunakan `ExUnit`, Elixir memberikan cara yang mudah dan efisien untuk melakukan unit testing. Anda bisa menulis test baik dalam file khusus maupun langsung dalam dokumentasi fungsi menggunakan `doctest`. Menjalankan test dari command prompt juga fleksibel, memungkinkan Anda untuk menjalankan seluruh test, test untuk modul tertentu, atau test untuk fungsi spesifik.

3.5 Latihan

Bagian ini berisi beberapa latihan yang dirancang untuk membantu Anda mempraktikkan dokumentasi dan unit testing di Elixir. Setiap latihan akan mencakup pembuatan modul, menulis dokumentasi, dan membuat unit test.

3.5.1 Latihan 1: Menulis Dokumentasi Modul

Buatlah sebuah modul Elixir bernama `Calculator` yang memiliki dua fungsi: `multiply/2` dan `divide/2`. Tulislah dokumentasi untuk setiap fungsi menggunakan anotasi `@doc`.

```elixir
defmodule Calculator do
@moduledoc """
A module for basic arithmetic operations.
"""

@doc """
Multiplies two numbers.

## Parameters
- x: the first number.
- y: the second number.

## Example

iex> Calculator.multiply(4, 5)
20
"""
def multiply(x, y) do
x * y
end

@doc """
Divides two numbers.

## Parameters
- x: the numerator.
- y: the denominator.

## Example

iex> Calculator.divide(10, 2)
5
"""
```

```
34  def divide(x, y) do
35  x / y
36  end
37  end
```

3.5.2 Latihan 2: Menulis Unit Test untuk Modul

Buatlah file test bernama `calculator_test.exs` di dalam folder `test` untuk mengetes fungsi `multiply/2` dan `divide/2` dari modul `Calculator`. Sertakan beberapa skenario pengujian.

```
1  defmodule CalculatorTest do
2  use ExUnit.Case
3  doctest Calculator
4
5  test "multiplies two numbers" do
6  assert Calculator.multiply(4, 5) == 20
7  end
8
9  test "divides two numbers" do
10 assert Calculator.divide(10, 2) == 5
11 end
12
13 test "divides by zero raises an error" do
14 assert_raise ArithmeticError, fn -> Calculator.divide(10, 0) end
15 end
16 end
```

3.5.3 Latihan 3: Menjalankan Test untuk Modul

Jalankan unit test yang telah Anda tulis untuk modul `Calculator` menggunakan perintah berikut:

```
1  mix test test/calculator_test.exs
```

3.5.4 Latihan 4: Menjalankan Test untuk Fungsi Tertentu

Tambahkan sebuah test baru dalam `calculator_test.exs` untuk fungsi `multiply/2` dan jalankan test tersebut menggunakan nomor baris.

```
1  defmodule CalculatorTest do
2  use ExUnit.Case
3  doctest Calculator
4
5  test "multiplies two numbers" do
6  assert Calculator.multiply(4, 5) == 20
7  end
8
9  # New test added for specific function
10 test "checks multiply with negative numbers" do
11 assert Calculator.multiply(-4, 5) == -20
12 end
13 end
```

Jika test baru ditambahkan pada baris 6, jalankan perintah berikut:

```
1  mix test test/calculator_test.exs:6
```

3.6 Soal

Bagian ini berisi beberapa soal latihan tambahan yang dirancang untuk mempraktikkan dokumentasi dan unit testing di Elixir. Setiap soal merupakan varian dari contoh-contoh latihan sebelumnya.

3.6.1 Soal 1: Dokumentasi dan Unit Test untuk Modul `Statistics`

Buatlah modul Elixir bernama `Statistics` yang memiliki dua fungsi: `mean/1` dan `median/1`.

1. Dokumentasikan setiap fungsi dengan anotasi `@doc`. Fungsi `mean/1` harus menghitung rata-rata dari sebuah list angka, sementara `median/1` harus menghitung median dari list angka yang sudah diurutkan.
2. Buatlah file test bernama `statistics_test.exs` di dalam folder `test` untuk mengetes fungsi `mean/1` dan `median/1`. Sertakan beberapa skenario pengujian, seperti menghitung rata-rata dan median untuk list dengan berbagai panjang dan nilai.

3.6.2 Soal 2: Dokumentasi dan Unit Test untuk Modul `StringUtils`

Buatlah modul Elixir bernama `StringUtils` yang memiliki dua fungsi: `reverse/1` dan `capitalize/1`.

1. Dokumentasikan setiap fungsi dengan anotasi `@doc`. Fungsi `reverse/1` harus membalikkan string, sementara `capitalize/1` harus mengubah huruf pertama dari string menjadi huruf kapital.
2. Buatlah file test bernama `string_utils_test.exs` di dalam folder `test` untuk mengetes fungsi `reverse/1` dan `capitalize/1`. Sertakan beberapa skenario pengujian, seperti membalikkan string dengan berbagai karakter dan mengubah kapitalisasi string dengan berbagai kasus huruf.

Chapter 4

Struktur Data di Elixir: Atom, Map, Tuple, List, dan Keyword List

4.1 Atom

Atom dalam Elixir adalah konstanta yang nilainya adalah nama itu sendiri. Atom sering digunakan untuk memberi label nilai atau mewakili konsep-konsep tertentu dalam program. Atom diawali dengan titik dua (`:`), diikuti oleh namanya.

4.1.1 Membuat Atom

Atom dibuat dengan menambahkan nama diawali dengan titik dua.

```
:name  # Sebuah atom dengan nilai :name
```

4.1.2 Menggunakan Atom dalam Pattern Matching

Atom umumnya digunakan dalam pattern matching untuk alur kontrol.

```
status = :ok

case status do
:ok -> "Berhasil"
:error -> "Gagal"
end
```

4.1.3 Atom dalam Keyword Lists

Atom biasanya digunakan sebagai kunci dalam keyword lists, yaitu daftar pasangan kunci-nilai.

```
keyword_list = [name: "Alice", age: 30]

IO.puts(keyword_list[:name])  # Output: Alice
```

4.1.4 Atom yang Mewakili Modul dan Fungsi

Dalam Elixir, atom digunakan untuk mewakili nama modul dan referensi fungsi.

```
String.length("hello")  # Di sini, modul String adalah atom

fun = &String.length/1
IO.puts(fun.("world"))  # Output: 5
```

Atom efisien dan ringan, menjadikannya ideal untuk digunakan dalam pattern matching, sebagai label, dan dalam berbagai struktur data seperti keyword lists dan maps. Karena atom bersifat immutable, atom menyediakan cara yang dapat diandalkan untuk merujuk nilai dengan nama di seluruh program.

4.2 Map

Map dalam Elixir adalah struktur data kunci-nilai di mana kunci dapat berupa tipe apa saja. Map tidak terurut, yang berarti pasangan kunci-nilai tidak disimpan dalam urutan tertentu.

4.2.1 Membuat Map

Map dibuat menggunakan sintaks `%`.

```
map = %{"name" => "Alice", "age" => 30}
```

4.2.2 Menambah/Memperbarui Entitas dalam Map

Untuk menambah atau memperbarui entitas dalam map, gunakan fungsi `Map.put/3`.

```
map = %{"name" => "Alice", "age" => 30}
updated_map = Map.put(map, "city", "New York")
```

4.2.3 Menghapus Entitas dari Map

Untuk menghapus entitas dari map, gunakan fungsi `Map.delete/2`.

```
map = %{"name" => "Alice", "age" => 30}
updated_map = Map.delete(map, "age")
```

4.2.4 Mengakses Nilai dalam Map

Nilai dapat diakses menggunakan kuncinya.

```
map = %{"name" => "Alice", "age" => 30}
IO.puts(map["name"])  # Output: Alice
```

4.3 Tuple

Tuple adalah koleksi elemen yang terurut. Tuple bersifat immutable, yang berarti tidak dapat diubah setelah dibuat.

4.3.1 Membuat Tuple

Tuple dibuat menggunakan sintaks `{}`.

```
tuple = {"Alice", 30, "New York"}
```

4.3.2 Mengakses Elemen dalam Tuple

Elemen dalam tuple dapat diakses berdasarkan indeksnya menggunakan pattern matching atau fungsi `elem/2`.

```
tuple = {"Alice", 30, "New York"}
IO.puts(elem(tuple, 0))  # Output: Alice
```

4.3.3 Memperbarui Tuple

Berikut adalah perintah untuk menambahkan elemen baru ke suatu tuple. Akan tetapi, karena tuple bersifat immutable, untuk memperbarui tuple, harus dibuat tuple baru dengan nilai yang diperbarui, seolah-olah elemen baru ditambahkan ke tuple.

```
tuple = {"Alice", 30, "New York"}
updated_tuple = Tuple.append(tuple, "Engineer")
```

4.3.4 Menghapus Elemen dari Tuple

Karena tuple bersifat immutable, elemen tidak dapat dihapus langsung dari tuple. Berikut adalah cara untuk menghapus elemen dari suatu tuple. Walapun demikian, elemen tersebut tidaklah dihapus. Yang terjadi adalah suatu tuple baru dibuat tetappi tidak menyertakan elemen yang ingin dihapus.

```
tuple = {"Alice", 30, "New York"}
updated_tuple = Tuple.delete_at(tuple, 2)
```

4.4 List

List adalah koleksi elemen yang terurut, dan berbeda dengan tuple, list bersifat mutable dan dapat memiliki elemen yang ditambahkan atau dihapus.

4.4.1 Membuat List

List dibuat menggunakan tanda kurung siku.

```
list = [1, 2, 3, 4, 5]
```

4.4.2 Menambah Elemen ke List

Elemen dapat ditambahkan ke depan list menggunakan operator cons `[ | ]` atau menggunakan fungsi dari modul `List`.

Menggunakan Operator Cons `[ | ]`

Elemen dapat ditambahkan ke depan list dengan operator cons.

```
list = [1, 2, 3]
new_list = [0 | list]
```

Menggunakan `List.insert_at/3`

Fungsi `List.insert_at/3` menyisipkan elemen pada indeks tertentu dalam list. Indeks dimulai dari 0.

```
list = [1, 2, 3]
new_list = List.insert_at(list, 0, 0)  # Menyisipkan 0 pada indeks 0
```

Menggunakan `List.concat/2`

Fungsi `List.concat/2` dapat digunakan untuk menambahkan elemen dengan cara menggabungkan dua list.

```
list = [1, 2, 3]
new_list = List.flatten([[0], list])  # Menggabungkan list dengan list baru
    yang berisi 0
```

Menghapus Elemen dari List

Elemen dapat dihapus dari list menggunakan fungsi `List.delete/2`.

```
list = [1, 2, 3, 4]
updated_list = List.delete(list, 3)
```

4.4.3 Mengakses Elemen dalam List

Ada beberapa cara untuk mendapatkan nilai dari list dalam Elixir. Beberapa metode yang umum digunakan adalah sebagai berikut:

Menggunakan Pattern Matching

Pattern matching adalah metode yang kuat dan sering digunakan untuk mengakses elemen dalam list. Dengan cara ini, elemen dari list dapat diambil dengan mencocokkan pola yang sesuai. Misalnya, mengambil Elemen Pertama:

```
list = [1, 2, 3, 4]

# Mengambil elemen pertama dari list
[head | _tail] = list
IO.puts(head)  # Outputs: 1
```

Menggunakan Fungsi `hd/1`

Fungsi `hd/1` digunakan untuk mendapatkan elemen pertama dari list.

```
list = [1, 2, 3, 4]
IO.puts(hd(list))  # Outputs: 1
```

Menggunakan Fungsi `tl/1`

Fungsi `tl/1` digunakan untuk mendapatkan semua elemen dalam list kecuali elemen pertama.

```
list = [1, 2, 3, 4]
IO.inspect(tl(list))  # Outputs: [2, 3, 4]
```

Mengakses Elemen Berdasarkan Indeks

Untuk mengakses elemen berdasarkan indeks dalam list, Elixir menyediakan beberapa metode:

Menggunakan Fungsi `Enum.at/2`. Fungsi `Enum.at/2` digunakan untuk mendapatkan elemen dari list berdasarkan indeksnya. Indeks mulai dari 0.

```
list = [1, 2, 3, 4]
IO.puts(Enum.at(list, 2))  # Outputs: 3
```

Menggunakan Pattern Matching untuk Mengakses Elemen Berdasarkan Indeks. Dengan menggunakan pattern matching, elemen pada posisi tertentu dapat diakses dengan mendefinisikan pola yang sesuai.

```elixir
defmodule ListUtils do
  def get_element_at([head | _tail], 0), do: head
  def get_element_at([_head | tail], index) when index > 0, do: get_element_at(
      tail, index - 1)
  def get_element_at([], _index), do: nil
end

list = [1, 2, 3, 4]
IO.puts(ListUtils.get_element_at(list, 2))  # Outputs: 3
```

Mendapatkan Indeks Berdasarkan Nilai

Untuk menemukan indeks elemen dalam list berdasarkan nilai tertentu, Anda dapat menggunakan beberapa pendekatan berikut:

Menggunakan Fungsi Enum.find_index/2. Fungsi `Enum.find_index/2` mengembalikan indeks dari elemen pertama yang cocok dengan kondisi yang diberikan oleh fungsi.

```elixir
  list = [10, 20, 30, 40]
  index = Enum.find_index(list, fn x -> x == 30 end)
  IO.puts(index)  # Outputs: 2
```

Menggunakan Pattern Matching untuk Mendapatkan Indeks Berdasarkan Nilai. Untuk menemukan indeks dengan pattern matching, Anda dapat menggunakan rekursi untuk mencari nilai yang sesuai dalam list.

```elixir
  defmodule ListUtils do
  def index_of([], _value, _index), do: nil
  def index_of([value | _tail], value, index), do: index
  def index_of([_head | tail], value, index) do
  index_of(tail, value, index + 1)
  end
  end

  list = [10, 20, 30, 40]
  index = ListUtils.index_of(list, 30, 0)
  IO.puts(index)  # Outputs: 2
```

4.5 Keyword List

Keyword list adalah list dari tuple di mana elemen pertama dari setiap tuple adalah atom, biasanya digunakan untuk melewatkan opsi dalam fungsi.

4.5.1 Membuat Keyword List

Keyword list dibuat menggunakan sintaks yang sama seperti list tetapi dengan tuple yang berisi atom dan nilai.

```elixir
  keyword_list = [name: "Alice", age: 30, city: "New York"]
```

4.5.2 Menambah/Memperbarui Elemen dalam Keyword List

Untuk menambah atau memperbarui nilai dalam keyword list, cukup tambahkan tuple baru dengan nilai yang diperbarui.

```elixir
  keyword_list = [name: "Alice", age: 30]
  updated_keyword_list = [city: "New York" | keyword_list]
```

4.5.3 Mengakses Elemen dalam Keyword List

Nilai dalam keyword list dapat diakses menggunakan kuncinya.

```
keyword_list = [name: "Alice", age: 30]
IO.puts(keyword_list[:name])  # Output: Alice
```

4.5.4 Menghapus Elemen dari Keyword List

Untuk menghapus entri dari keyword list, gunakan fungsi `Keyword.delete/2`.

```
keyword_list = [name: "Alice", age: 30, city: "New York"]
updated_keyword_list = Keyword.delete(keyword_list, :city)
```

4.6 Konversi Antara Struktur Data

Dalam Elixir, seringkali diperlukan untuk mengkonversi antara struktur data yang berbeda seperti tuple, list, dan keyword list. Berikut adalah cara untuk melakukan konversi antara struktur data ini:

4.6.1 Konversi dari Tuple ke List

Untuk mengkonversi tuple menjadi list, gunakan fungsi `Tuple.to_list/1`:

```
tuple = {1, 2, 3, 4}
list = Tuple.to_list(tuple)
IO.inspect(list) # Output: [1, 2, 3, 4]
```

4.6.2 Konversi dari List ke Tuple

Untuk mengkonversi list menjadi tuple, gunakan fungsi `List.to_tuple/1`:

```
list = [1, 2, 3, 4]
tuple = List.to_tuple(list)
IO.inspect(tuple) # Output: {1, 2, 3, 4}
```

4.6.3 Konversi dari Keyword List ke Map

Untuk mengkonversi keyword list menjadi map, gunakan fungsi `Enum.into/2`:

```
keyword_list = [name: "Budi", age: 25]
map = Enum.into(keyword_list, %{})
IO.inspect(map) # Output: %{name: "Budi", age: 25}
```

4.6.4 Konversi dari Map ke Keyword List

Untuk mengkonversi map menjadi keyword list, gunakan fungsi `Map.to_list/1`:

```
map = %{name: "Budi", age: 25}
keyword_list = Map.to_list(map)
IO.inspect(keyword_list) # Output: [name: "Budi", age: 25]
```

4.6.5 Konversi dari List ke Keyword List

Untuk mengkonversi list ke keyword list, setiap elemen list harus berupa tuple dengan dua elemen, di mana elemen pertama adalah atom yang akan menjadi key dan elemen kedua adalah nilai. Gunakan `Enum.into/2` untuk konversi:

```
list = [name: "Budi", age: 25]
keyword_list = Enum.into(list, [])
IO.inspect(keyword_list) # Output: [name: "Budi", age: 25]
```

4.6.6 Konversi dari Keyword List ke List

Untuk mengkonversi keyword list menjadi list, gunakan `Keyword.to_list/1`:

```
keyword_list = [name: "Budi", age: 25]
list = Keyword.to_list(keyword_list)
IO.inspect(list) # Output: [name: "Budi", age: 25]
```

4.6.7 Konversi dari Tuple ke Keyword List

Untuk mengkonversi tuple yang berisi pasangan key-value menjadi keyword list, Anda bisa menggunakan `Tuple.to_list/1` dan kemudian melakukan konversi lebih lanjut:

```
tuple = {:name, "Budi"}
keyword_list = Tuple.to_list(tuple) |> Enum.chunk_every(2) |> Enum.map(fn [k,
    v] -> {k, v} end)
IO.inspect(keyword_list) # Output: [name: "Budi"]
```

4.7 Latihan

4.7.1 Latihan 1: Manipulasi Map

Tujuan: Memahami cara menambah, mengubah, menghapus, dan mengakses data dalam Map.

1. Buatlah sebuah Map yang menyimpan informasi berikut:
   - `name`: ”Budi”
   - `age`: 25
   - `city`: ”Jakarta”
2. Tambahkan key baru `job` dengan nilai ”Engineer”.
3. Update nilai dari key `age` menjadi 26.
4. Hapus key `city` dari Map.
5. Cetak nilai dari key `name` dan `age`.

```
# Buat Map awal
person = %{"name" => "Budi", "age" => 25, "city" => "Jakarta"}

# Tambahkan key baru
person = Map.put(person, "job", "Engineer")

# Update nilai dari key age
person = Map.put(person, "age", 26)

# Hapus key city
```

```
person = Map.delete(person, "city")

# Akses dan cetak nilai
IO.puts("Name: #{person["name"]}")
IO.puts("Age: #{person["age"]}")
```

4.7.2 Latihan 2: Manipulasi Tuple

Tujuan: Memahami cara membuat dan memodifikasi Tuple.

1. Buat sebuah Tuple yang menyimpan data berikut: `"Budi"`, `25`, `"Jakarta"`.
2. Akses dan cetak elemen kedua dari Tuple.
3. Tambahkan elemen baru `"Engineer"` ke dalam Tuple.
4. Hapus elemen kedua (usia) dari Tuple dan cetak Tuple hasil akhir.

```
# Buat Tuple awal
person_tuple = {"Budi", 25, "Jakarta"}

# Akses elemen kedua
IO.puts("Age: #{elem(person_tuple, 1)}")

# Tambahkan elemen baru
person_tuple = Tuple.append(person_tuple, "Engineer")

# Hapus elemen kedua (usia)
person_tuple = person_tuple |> Tuple.to_list() |> List.delete_at(1) |> List.
    to_tuple()

# Cetak Tuple hasil akhir
IO.inspect(person_tuple)
```

4.7.3 Latihan 3: Manipulasi List

Tujuan: Memahami cara menambah, menghapus, dan mengakses elemen dari List.

1. Buat sebuah List berisi angka-angka dari 1 sampai 5.
2. Tambahkan angka 0 di depan List.
3. Hapus angka 3 dari List.
4. Akses dan cetak elemen ketiga dari List.

```
# Buat List awal
numbers = [1, 2, 3, 4, 5]

# Tambahkan angka 0 di depan List
numbers = [0 | numbers]

# Hapus angka 3
numbers = List.delete(numbers, 3)

# Akses elemen ketiga
IO.puts("Elemen ketiga: #{Enum.at(numbers, 2)}")
```

4.7.4 Latihan 4: Manipulasi Keyword List

Tujuan: Memahami cara membuat, menambah, mengubah, menghapus, dan mengakses elemen dari Keyword List.

1. Buat sebuah Keyword List yang menyimpan informasi berikut:
   - `name`: ”Budi”
   - `age`: 25
2. Tambahkan key baru `city` dengan nilai ”Jakarta”.
3. Update nilai dari key `age` menjadi 26.
4. Hapus key `city` dari Keyword List.
5. Cetak nilai dari key `name` dan `age`.

```
# Buat Keyword List awal
person_kw = [name: "Budi", age: 25]

# Tambahkan key baru
person_kw = [city: "Jakarta" | person_kw]

# Update nilai dari key age
person_kw = Keyword.put(person_kw, :age, 26)

# Hapus key city
person_kw = Keyword.delete(person_kw, :city)

# Akses dan cetak nilai
IO.puts("Name: #{person_kw[:name]}")
IO.puts("Age: #{person_kw[:age]}")
```

4.7.5 Latihan 5: Penggunaan Atom

Tujuan: Memahami cara menggunakan atom dalam pattern matching dan sebagai key dalam struktur data.

1. Buat sebuah atom bernama `:status` dengan nilai `:ok`.
2. Gunakan atom `:ok` dan `:error` untuk melakukan pattern matching di dalam `case`.
3. Buatlah sebuah Map dengan key berupa atom, misalnya `:name`, `:age`, dan `:city`. Akses dan cetak nilai dari masing-masing key tersebut.

```
# Buat atom status
status = :ok

# Pattern matching dengan atom
case status do
:ok -> IO.puts("Success")
:error -> IO.puts("Failure")
end

# Buat Map dengan key berupa atom
person_map = %{name: "Budi", age: 25, city: "Jakarta"}

# Akses dan cetak nilai
IO.puts("Name: #{person_map[:name]}")
IO.puts("Age: #{person_map[:age]}")
IO.puts("City: #{person_map[:city]}")
```

4.7.6 Latihan 6: Menggabungkan Semua Konsep

Tujuan: Menerapkan semua konsep yang telah dipelajari dalam satu program.

1. Buat sebuah fungsi `create_person/3` yang menerima tiga argumen: Nama (String), Usia (Integer), Kota (String).
2. Fungsi ini harus mengembalikan Map yang berisi informasi tersebut dengan key berupa atom.
3. Buat fungsi `update_city/2` untuk mengubah kota dari Map hasil dari fungsi `create_person/3`.
4. Buat fungsi `delete_age/1` untuk menghapus key `:age` dari Map.
5. Cetak hasil akhir dari setiap fungsi.

```
defmodule Person do
# Fungsi untuk membuat Map dengan atom sebagai key
def create_person(name, age, city) do
%{name: name, age: age, city: city}
end

# Fungsi untuk mengupdate nilai city
def update_city(person, new_city) do
Map.put(person, :city, new_city)
end

# Fungsi untuk menghapus key age
def delete_age(person) do
Map.delete(person, :age)
end
end

# Membuat person
person = Person.create_person("Budi", 25, "Jakarta")

# Mengupdate kota
person = Person.update_city(person, "Bandung")

# Menghapus usia
person = Person.delete_age(person)

# Cetak hasil akhir
IO.inspect(person)
```

4.8 Soal Latihan

4.8.1 Latihan 1: Atom

1. Apa itu atom dalam Elixir? Jelaskan kapan dan mengapa atom digunakan.
2. Buat sebuah fungsi yang menerima atom sebagai argumen, dan melakukan pattern matching untuk mencetak pesan berikut:
   - Jika menerima `:ok`, cetak ”Proses berhasil”.
   - Jika menerima `:error`, cetak ”Proses gagal”.
3. Buatlah tiga atom berbeda, simpan mereka dalam sebuah tuple, dan akses masing-masing elemen dari tuple tersebut.

4.8.2 Latihan 2: Manipulasi Map

1. Buatlah sebuah Map yang menyimpan informasi tentang sebuah buku, yang terdiri dari `title`, `author`, dan `year_published`.
2. Tambahkan sebuah key baru `publisher` ke dalam Map yang telah dibuat.
3. Update nilai dari key `year_published` menjadi tahun terbaru.
4. Hapus key `publisher` dari Map.
5. Bagaimana cara mengakses nilai dari key `author`?

4.8.3 Latihan 3: Manipulasi Tuple

1. Buatlah sebuah Tuple yang menyimpan informasi mengenai sebuah kendaraan, yang terdiri dari `jenis`, `warna`, dan `tahun`.
2. Akses elemen kedua dari Tuple tersebut.
3. Bagaimana cara menambah elemen baru ke dalam Tuple? Ubah tuple agar elemen baru berupa `"plat nomor"` ditambahkan di akhir.
4. Bagaimana cara menghapus elemen kedua dari Tuple tersebut?

4.8.4 Latihan 4: Manipulasi List

1. Buat sebuah List yang berisi lima angka acak.
2. Bagaimana cara menambahkan elemen baru ke depan List?
3. Hapus elemen ketiga dari List tersebut.
4. Jelaskan bagaimana cara mengakses elemen keempat dari List.

4.8.5 Latihan 5: Manipulasi Keyword List

1. Buat sebuah Keyword List yang menyimpan informasi mengenai produk dengan key berupa `:nama`, `:harga`, dan `:stok`.
2. Bagaimana cara menambah key baru `:kategori` ke dalam Keyword List tersebut?
3. Ubah nilai dari key `:harga`.
4. Hapus key `:stok` dari Keyword List.
5. Jelaskan bagaimana cara mengakses nilai dari key `:nama`.

4.8.6 Latihan 6: Menggabungkan Semua Konsep

1. Buat sebuah fungsi yang menerima nama, usia, dan pekerjaan sebagai argumen, dan mengembalikan sebuah Map dengan key berupa atom.
2. Buat sebuah fungsi untuk mengupdate salah satu informasi dalam Map yang dihasilkan dari fungsi di atas.
3. Buat sebuah fungsi untuk menghapus salah satu informasi dari Map tersebut.
4. Buat sebuah fungsi yang menerima argumen berupa Tuple dan List, lalu gabungkan elemen dari keduanya menjadi satu List.
5. Buatlah Keyword List yang menyimpan informasi tentang siswa (nama, nilai, dan kelas), dan lakukan operasi penambahan, pengubahan, dan penghapusan pada Keyword List tersebut.

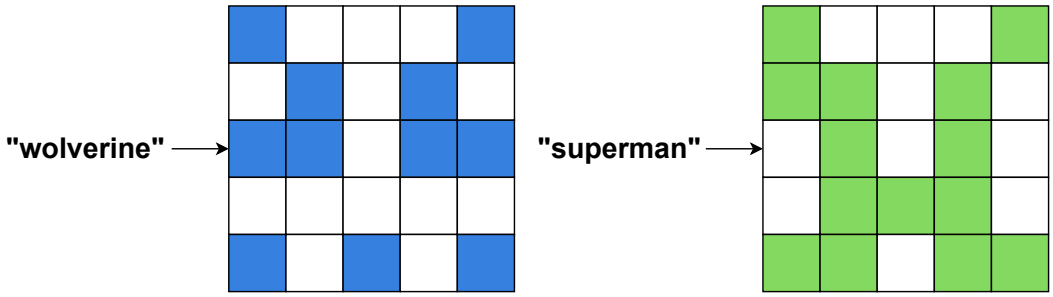
Chapter 5

Generator Avatar dengan Elixir

5.1 Pendahuluan

Pada bab ini, akan dibahas pengembangan generator avatar menggunakan bahasa pemrograman Elixir. Kode ini memanfaatkan konsep inti Elixir seperti `defstruct` untuk mendefinisikan struktur data khusus, `defmodule` untuk membuat modul, serta menggunakan `Application` behavior untuk mengelola siklus hidup aplikasi. Generator avatar ini menerima sebuah input, menghitung hash, memilih warna, dan menghasilkan representasi grid dari avatar. Bagian-bagian berikut akan membahas setiap bagian kode secara lebih rinci.

5.1.1 Avatar Generator



wolverine's md5 hash = [54, 129, 223, 141, 4, 71, 14, 204, 101, 5, 59, 121, 14, 25, 160, 101]

superman's md5 hash = [132, 217, 97, 86, 138, 101, 7, 58, 59, 207, 14, 178, 22, 178, 165, 118]

Pada program generator avatar ini, kata-kata seperti `"wolverine"` dan `"superman"` akan diolah dengan menghitung nilai hash-nya menggunakan algoritma MD5. Hash yang dihasilkan berupa daftar angka yang mewakili nilai-nilai biner dari hasil hash tersebut.

Sebagai contoh:

- Kata `"wolverine"` memiliki hash MD5 yang direpresentasikan sebagai daftar angka:

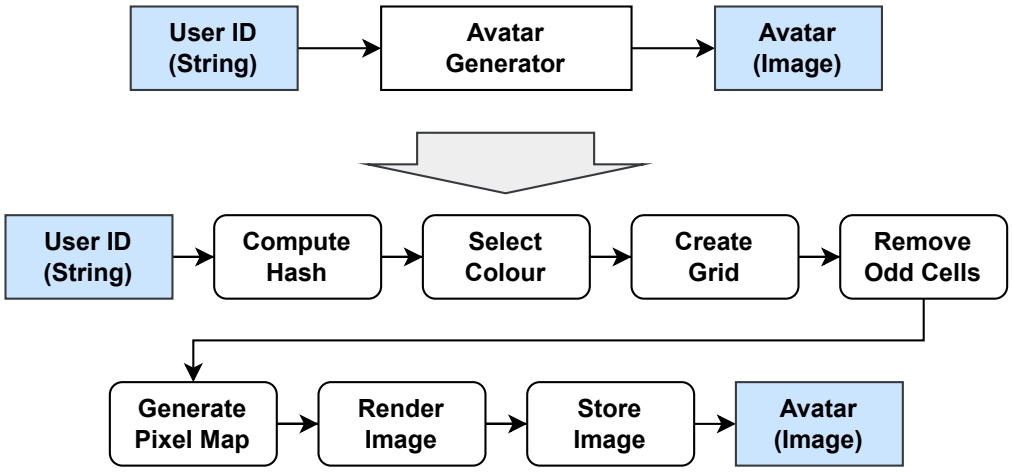
$$[54, 129, 223, 141, 4, 71, 14, 204, 101, 5, 59, 121, 14, 25, 160, 101]$$

- Kata `"superman"` menghasilkan hash MD5 yang direpresentasikan sebagai daftar angka:

$$[132, 217, 97, 86, 138, 101, 7, 58, 59, 207, 14, 178, 22, 178, 165, 118]$$

Hash ini kemudian digunakan untuk membentuk representasi grid dari avatar. Setiap nilai dalam hash akan digunakan untuk menentukan pola atau warna dari avatar yang dihasilkan, sehingga setiap kata input yang berbeda akan menghasilkan avatar yang unik.

5.1.2 Avatar Pipeline



5.1.3 Avatar Computation

1 **54**	2 **129**	3 **223**	2 **129**	1 **54**
4 **141**	5 **4**	6 **71**	5 **4**	4 **141**
7 **14**	8 **204**	9 **101**	8 **204**	7 **14**
10 **5**	11 **59**	12 **121**	11 **59**	10 **5**
13 **14**	14 **25**	15 **160**	14 **25**	13 **14**

The first 3 values are the RGB colour

[54, 129, 223, 141, 4, 71, 14, 204, 101, 5, 59, 121, 14, 25, 160, 101]

5.2 Struktur Modul

Program ini terdiri dari dua modul:

- **Avatar.Image**: Modul yang mendefinisikan struktur untuk menyimpan informasi terkait avatar seperti hash dan warna.
- **AvatarGenerator**: Modul utama yang bertanggung jawab untuk menghasilkan avatar dengan menghitung hash, memilih warna, dan membuat grid.

5.2.1 Mendefinisikan Struktur Avatar Image

Modul `Avatar.Image` mendefinisikan struktur yang akan digunakan untuk menyimpan informasi hash dan warna dari avatar. Struktur ini didefinisikan menggunakan kata kunci `defstruct`.

Listing 5.1: Mendefinisikan struktur Avatar Image di `lib/image.ex`

```
1  defmodule Avatar.Image do
2  defstruct hash: nil, color: nil
3  end
```

Kata kunci `defstruct` digunakan untuk mendefinisikan struktur dengan atribut `hash` dan `color`. Atribut-atribut ini awalnya diatur menjadi `nil` dan akan diisi seiring dengan proses pembuatan avatar.

5.2.2 Modul Generator Avatar

Modul `AvatarGenerator` adalah modul inti yang menangani logika pembuatan avatar. Modul ini menggunakan `Application` behavior, yang memungkinkan untuk mendefinisikan fungsi `start/2` guna menginisialisasi aplikasi.

Listing 5.2: Definisi modul Avatar Generator di `lib/avatar.ex`

```
1  defmodule AvatarGenerator do
2  use Application
```

5.2.3 Pembuatan Avatar

Fungsi `generate/1` adalah fungsi utama yang menangani pembuatan avatar. Fungsi ini menerima sebuah string `input`, menghitung hash, memilih warna, dan membuat representasi grid dari avatar.

Listing 5.3: Fungsi utama untuk pembuatan avatar

```
1  def generate(input) do
2  input
3  |> compute_hash
4  |> select_color
5  |> create_grid
6  end
```

Fungsi `generate/1` menggunakan operator pipa (`|>`) untuk meneruskan hasil dari setiap fungsi ke fungsi berikutnya dalam pipeline. Ini memastikan alur kode yang bersih dan mudah dibaca.

5.2.4 Menghitung Hash

Fungsi `compute_hash/1` menggunakan modul `:crypto` untuk menghasilkan hash MD5 dari string input. Hash biner yang dihasilkan kemudian dikonversi menjadi daftar integer dan diubah ukurannya agar panjangnya menjadi kelipatan tiga.

Listing 5.4: Menghitung hash dari string input

```
1  def compute_hash(input) do
2  hash =
3  :crypto.hash(:md5, input)
4  |> :binary.bin_to_list()
5  |> resize_list
6
7  IO.inspect(hash)
8  %Avatar.Image{hash: hash}
9  end
```

Fungsi pembantu `resize_list/1` digunakan untuk memastikan bahwa panjang daftar hash adalah kelipatan 3. Hal ini diperlukan untuk membuat representasi grid avatar yang simetris.

Listing 5.5: Mengubah ukuran daftar hash

```
def resize_list(list) do
full_chunks_count = div(length(list), 3) * 3
Enum.take(list, full_chunks_count)
end
```

5.2.5 Memilih Warna Avatar

Fungsi `select_color/1` mengekstrak tiga elemen pertama dari daftar hash dan menggunakannya untuk mendefinisikan nilai RGB dari warna avatar.

Listing 5.6: Memilih warna avatar dari hash

```
def select_color(%Avatar.Image{hash: [r, g, b | _tail]} = image) do
%Avatar.Image{image | color: {r, g, b}}
end
```

Warna direpresentasikan sebagai tuple `{r, g, b}`, di mana `r`, `g`, dan `b` masing-masing adalah komponen merah, hijau, dan biru dari warna tersebut.

5.2.6 Membuat Grid Avatar

Fungsi `create_grid/1` mengubah hash menjadi pola grid dengan memecah daftar hash menjadi baris-baris yang masing-masing terdiri dari tiga elemen, kemudian mencerminkan setiap baris untuk menciptakan simetri. Baris-baris ini kemudian dipipihkan dan diberi indeks.

Listing 5.7: Membuat grid avatar

```
@spec create_grid(%Avatar.Image{:hash => any(), optional(any()) => any()}) ::
    [
{any(), integer()}
]
def create_grid(%Avatar.Image{hash: hash} = image) do
hash
|> Enum.chunk_every(3)
|> Enum.map(&reflect_row/1)
|> List.flatten
|> Enum.with_index
end
```

Fungsi pembantu `reflect_row/1` menduplikasi dua elemen pertama dari setiap baris dalam urutan terbalik, memastikan bahwa grid yang dihasilkan adalah simetris.

Listing 5.8: Mencerminkan baris untuk menciptakan simetri

```
def reflect_row(row) do
[ first, second | _tail] = row
row ++ [second, first]
end
```

5.2.7 Memulai Aplikasi

Fungsi `start/2` dipanggil saat aplikasi dimulai. Fungsi ini membuat avatar untuk input sampel, yaitu ”wolverine”, dan mencetak grid yang dihasilkan ke konsol.

Listing 5.9: Memulai aplikasi dan membuat avatar

```
def start(_type, _args) do
result = AvatarGenerator.generate("wolverine")
IO.inspect(result)
{:ok, self()}
end
```

5.3 Menjalankan Aplikasi

Untuk menjalankan aplikasi, konfigurasi pada file `mix.exs` harus menyebutkan modul `AvatarGenerator` sebagai modul awal. Hal ini dilakukan dengan menambahkan atribut `mod` pada fungsi `application`.

Listing 5.10: Menentukan modul awal pada `mix.exs`

```
1  def application do
2  [
3  extra_applications: [:logger],
4  mod: {AvatarGenerator, [] }
5  ]
6  end
```

5.4 Kesimpulan

Bab ini membahas implementasi generator avatar menggunakan Elixir. Kode ini memanfaatkan modul `:crypto` untuk menghitung hash, struktur Elixir untuk menyimpan data avatar, serta berbagai operasi list untuk menciptakan representasi berbasis grid dari avatar. Aplikasi ini diatur menggunakan `Application` behavior sehingga dapat dijalankan sebagai aplikasi mandiri.

Appendix A

Penjelasan Perintah Mix untuk Mengelola Dependencies

Pada bagian ini, akan dijelaskan dua perintah penting dalam Elixir yang digunakan untuk mengelola dependencies dalam proyek menggunakan `mix`.

A.1 `mix deps.unlock --all`

Perintah `mix deps.unlock --all` digunakan untuk membuka kunci (unlock) semua dependencies dalam proyek Elixir. Ketika dependencies diinstal, versi spesifik dari setiap dependency akan dikunci di dalam file `mix.lock`. Perintah ini berguna ketika Anda ingin memperbarui atau menghapus dependencies tanpa terikat pada versi yang sudah dikunci. Dengan membuka kunci semua dependencies, proyek dapat melakukan upgrade ke versi terbaru dari dependencies yang sesuai dengan spesifikasi di `mix.exs`.

A.2 `mix deps.update --all`

Perintah `mix deps.update --all` digunakan untuk memperbarui semua dependencies dalam proyek Elixir ke versi terbaru yang kompatibel berdasarkan spesifikasi di `mix.exs`. Perintah ini akan mengunduh versi terbaru dari setiap dependency yang tersedia dan memperbarui file `mix.lock` dengan informasi versi yang baru. Ini sangat berguna ketika Anda ingin memastikan proyek menggunakan versi dependencies yang paling up-to-date untuk mendapatkan fitur terbaru dan perbaikan bug.

A.3 Kapan Menggunakan Perintah Ini?

Perintah `mix deps.unlock --all` diikuti oleh `mix deps.update --all` biasanya digunakan bersama ketika Anda ingin menghapus semua pembatasan versi dari dependencies yang ada dan memperbarui semuanya ke versi terbaru. Langkah ini sering diambil ketika terjadi masalah kompatibilitas dengan versi dependencies atau ketika ada pembaruan besar yang perlu diadopsi dalam proyek.

A.4 Error: 08:13:29.182 [error] beam/beam_load.c(206): Error loading module 'Elixir.Hex': This BEAM file was compiled for an old version of the runtime system. To fix this, please re-compile this module with ErlangOTP 24 or later.

I would `mix local.hex` and also if you're in a mix project I would `rm -rf _build` deps and see if recompiling helps.