Data Oriented Design

- ***"If you don't understand the data, you don't understand the problem. This is because all problems are unique and specific to the data you are working with. When the data is changing, your problems are changing. When your problems are changing, the algorithms (data transformations) needs to change with it."***

- Think about this. Every problem you work on is a data transformation problem. Every function you write and every program you run takes some input data and produces some output data. Your mental model of software is, from this perspective, an understanding of these data transformations (i.e., how they are organized and applied in the code base).

- **Uncertainty about the data is not a license to guess but a directive to STOP and learn more.**
- If performance matters, you must have mechanical sympathy for how the hardware and operating system work.
- Minimize, simplify and REDUCE the amount of code required to solve each problem. Do less work by not wasting effort.
- Code that can be reasoned about and does not hide execution costs can be better understood, debugged and performance tuned.
- Efficiency is obtained through algorithms but performance is obtained through data structures and layouts.

Go supports 3 types of data structure Array, Slice and Map

```
## Arrays

Arrays are a special data structure in Go that allow us to allocate contiguous blocks of fixed size memory. A
rrays have some special features in Go related to how they are declared and viewed as types.

## Notes

• If you don't understand the data, you don't understand the problem.
• If you don't understand the cost of solving the problem, you can't reason about the problem.
• If you don't understand the hardware, you can't reason about the cost of solving the problem.
• Arrays are fixed length data structures that can't change.
• Arrays of different sizes are considered to be of different types.
• Memory is allocated as a contiguous block.
• Go gives you control over spacial locality.

## Code Review

[Declare, initialize and iterate](example1/example1.go)
[Different type arrays](example2/example2.go)
[Contiguous memory allocations](example3/example3.go)
[Range mechanics](example4/example4.go)

## Slices -

Slices are an incredibly important data structure in Go. They form the basis for how we manage and manipulate
 data in a flexible, performant and dynamic way. It is incredibly important for all Go programmers to learn h
ow to uses slices.
Reference type
Zero value is set to nil
## Notes

• Slices are like dynamic arrays with special and built-in functionality.
• There is a difference between a slices length and capacity and they each service a purpose.
• Slices allow for multiple "views" of the same underlying array.
• Slices can grow through the use of the built-in function append.
## Code Review

[Declare and Length](example1/example1.go)
[Reference Types](example2/example2.go)
[Appending slices](example3/example3.go)
[Taking slices of slices](example4/example4.go)
[Slices and References](example5/example5.go)
[Strings and slices](example6/example6.go)
[Variadic functions](example7/example7.go)
[Range mechanics](example8/example8.go)
## Maps

Maps provide a data structure that allow for the storage and management of key/value pair data.

## Notes

• Maps provide a way to store and retrieve key/value pairs.
• Reading an absent key returns the zero value for the map's value type.
• Iterating over a map is always random.
• The map key must be a value that is comparable.
• Elements in a map are not addressable.
• Maps are a reference type.
## Code Review

[Declare, write, read, and delete](example1/example1.go)
[Absent key](example2/example2.go)
[Map key restrictions](example3/example3.go)
[Map literals and range](example4/example4.go)
[Sorting maps by key](example5/example5.go)
[Taking an element's address](example6/example6.go)
[Maps are Reference Types](example7/example7.go)
```