Chapter 1. Properties and Variables

```
import Foundation

struct Position {
  let x: Int
  let y: Int
}


var position = Position(x: 1, y: 1)
```

1. Foundation - standard library
2. Structures:
   - largely similar to classes,
   - don’t support **inheritance**
   - passed around by **value** not reference
3. Variables and properties:
   - `let` - constants vs `var` - variables
   - Structure: keyword propertyName: **type annotation**
   - no need for **type annotation** due to **type inference**
   - `position`’s type can’t change - **statically typed** (all types have to be resolved at compile time)

Exercise:

- Let’s try to define `position` variable and initialise it.

Chapter 2. Building Data Types

```
import Foundation

enum Direction {
  case left
  case right
  case up
  case down

  var horizontal: Bool {
      return self == .left || self == .right
  }
}

class Board {
  let size: (width: Int, height: Int)
  let obstacles: [Position]
  var player: Position
  let finish: Position

  init(width: Int, height: Int, obstacles:
[Position], start: Position, finish: Position) {
      self.size = (width, height)
      self.obstacles = obstacles
      self.player = start
      self.finish = finish
  }
}

var board = Board(width: 4, height: 5, obstacles:
[Position(x: 0, y: 4), Position(x: 1, y: 0),
Position(x: 3, y: 2)], start: Position(x: 0, y:
2), finish: Position(x: 1, y: 1))
```

1. Enumerations:
   - can take only one value from finite pool of values
   - in wide support for enum’s value types
   - can implement **protocols**
   - can have **members** like **computed properties** and methods
2. Classes:
   - require initialiser
   - compiler will make sure we initialise all properties before object is created
3. **tuples**, a collection of named elements of static length, can hold different types
4. `array`, sugar syntax, **homogeneous**, structs

Exercise:

- Can someone spot what information are we missing for our `Board`?
- How can we define the `finish` property?

Chapter 3. Functions and Loops

```
var obstacles = [Position(x: 0, y: 4), Position(x:
1, y: 0), Position(x: 3, y: 2)] +
edgesForBoard(ofSize: 4, by: 5)
var board = Board(width: 4, height: 5, finish:
Position(x: 1, y: 1), obstacles: obstacles,
player: Position(x: 0, y: 2))
board
```

```
func verticalEdgesForBoard(ofSize width: Int, by
height: Int) -> [Position] {
  var edgePositions = [Position]()

  for y in 0..<height {
    edgePositions.append(Position(x: -1, y: y))
    edgePositions.append(Position(x: width, y: y))
  }

  return edgePositions
}
```

```
func edgesForBoard(ofSize width: Int, by height:
Int) -> [Position] {
  return horizontalEdgesForBoard(ofSize: width,
by: height) + verticalEdgesForBoard(ofSize: width,
by: height)
}
```

1. Sum two arrays into one use plus operator.
2. Functions: (use autocompletion)
   - **argument label** and **parameter name**
   - to skip **argument label** we use an **underscore**
3. There is no classic **C-style for-loops** with a counter, instead we can **for-in** through a **range** of integers
   - statements have no parenthesis around the condition
   - `Range` a half-open interval from a lower bound up to, but not including, an upper bound.
   - `ClosedRange` a closed interval from a lower bound up to, and including, an upper bound.
   - to add an element to an existing array use `append`

Exercise:

- Let’s implement a method to generate horizontal edges.
  - What “y” value should we start from?
  - What is “x” value for the left edge?
- How to return side edges and horizontal edges?

Chapter 4. Closures and Control Flow

```
extension Position {
  func distance(from position: Position, in
direction: Direction) -> Int {
    if direction.horizontal {
      return abs(position.x - self.x)
    } else {
      return abs(position.y - self.y)
    }
  }
}

extension Board {
  func playerMoves(_ direction: Direction) {
    findObstacleClosestToPlayer(moving: direction)
  }

  func findObstacleClosestToPlayer(moving
direction: Direction) {
    obstaclesInTheWay = obstacles.filter
      { (obstacle) -> Bool in
      return obstacle.isOnSameAxis(as: player, in:
direction) && obstacle.isInFront(of: player, in:
direction)
    }

    closestObstacle = obstaclesInTheWay.min
{ (lhs, rhs) -> Bool in
      let lhsDist = player.distance(from: lhs, in:
direction)
      let rhsDist = player.distance(from: rhs, in:
direction)
      return lhsDist < rhsDist
    }
  }}
```

1. `filter`
   - function takes the closure as an argument but the parenthesis are removed due to **trailing closure** syntax
   - (params) -> return type is closure type, `obstacle` type is **inferred**
2. Distance function:
   - `if` statement has no parenthesis around the condition
   - Condition has to evaluate to a `Bool`
3. Use built-in min method of array:
   - Use auto completion and pay attention how parenthesis are removed
   -
   - body starts after the `in` keyword

Exercise:

- Show how the filter highlights the blocks on the way
- How to decide which element is closer to the player?
- How to calculate a distance?

Chapter 5. Optionals and Switches

```
extension Position {
  static func contiguous(to position: Position,
movingFrom direction: Direction) -> Position {
    switch direction {
    case .up:
      return Position(x: position.x, y: position.y
+ 1)
    case .down:
      return Position(x: position.x, y: position.y
- 1)
    case .left:
      return Position(x: position.x + 1, y:
position.y)
    case .right:
      return Position(x: position.x - 1, y:
position.y)
    }
  }}

extension Board {
    func playerMoves(_ direction: Direction) {
        findObstacleClosestToPlayer(moving:
direction)
        updatePlayersPositionAfter(moving:
direction)
    }

    func updatePlayersPositionAfter(moving
direction: Direction) {
    guard let closestObstacle =
self.closestObstacle else { return }
    player = .contiguous(to: closestObstacle,
movingFrom: direction)
    }
}
```

1. Switch:
   - switch needs to be **exhaustive**
   - there is a default in case of course
   - **break** is implicit
   - comma means the direction can match one of the two to pass
2. Optionals:
   - Swift is statically typed which means the info about possibility of value missing needs to be explicit at compile time
   - **optional** a type that can hold a value or nothing, has only two states
   - `Type?` is a sugar syntax, in fact the optional is a generic enum `Optional<Type>`
3. Guard:
   - similar purpose to `if` statement
   - `guard` is Swift's way to fight **pyramid of doom**
   - `guard`'s block is actually a **fallback** in case the **condition** is not met, it's required for this block to **leave the scope**
4. We can omit type name when accessing a **static member** of a type if it can be **inferred** from the context

Exercise:

- Update the player's position

Chapter 6. Playtime!

```swift
let name = "Type your name here!"
let message = """
 \ ( ^ O ^ ) /

🥳 Congrats, \(name)! 🎉

 \ ( ^ O ^ )/
"""

let game = Game.start(withCompletionMessage: message)


game.move(in: .right).move(in: .down).move(in: .left).move(in: .up)
```

1. Strings:
   - are **Unicode-compliant**
   - can be treated as **collection of characters**
   - `\(input)` syntax for **interpolation**
   - **multiline** string begins with triple quotation

Exercise:

- Change content of the `name` string to your name.
- If you with you can customise the message to appear on victory.
- Play calling move method on the `game` variable!