OOP: Chess Engine in Kotlin und Go

Konzepte der Programmiersprachen

Julia Ihrenberger

07.01.26

Source: https://go.googlesource.com/website/+/refs/heads/master/_content/doc/gopher/modelsheet.jpg
https://kotlinlang.org/docs/kotlin-brand-assets.html#kotlin-logo

Content

- Kotlin Refresher
    - Packages
    - Class Code Structure
- Chess Excursion
- Chessboard Representation: Bitboards
- Implementation
    - Code Flow
    - Code Structure
    - Bitboard Usage
- Addendum

Source: https://kotlinlang.org/docs/basic-syntax.html

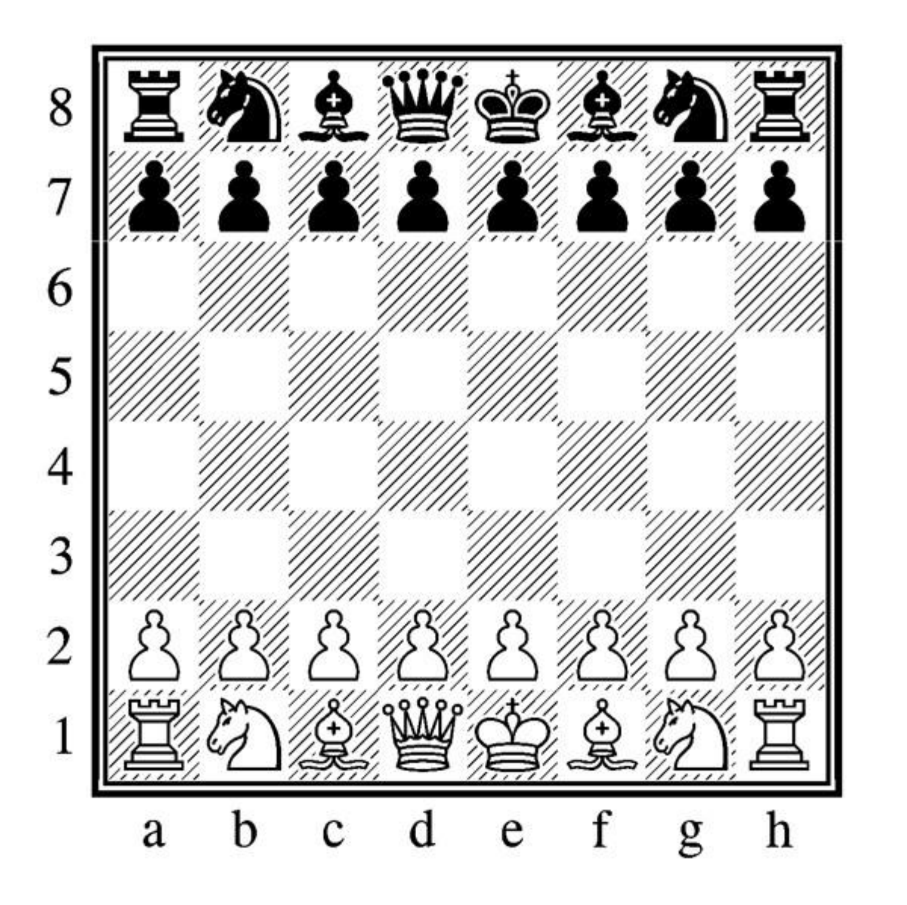
Refresh: Packages and Classes

```
package packageName

import otherPackage.*

open class InheritableClass(
    val Immutable: DataType,
    var Mutable: DataType = DefaultValue){

        //Code Block
}
```

- similar structure to Go with packages
- Import per function use or selector
- Kotlin values Immutability over all else, so classes and function need to be marked as open for inherit and override

Source: https://kotlinlang.org/docs/basic-syntax.html

Excursion: Chess Rules



- King: one step in each direction
- Queen: omni-directional
- Bishop: omni-diagonal
- Knight: jumps in L shape
- Rook: cardinal
- Pawn: >:c

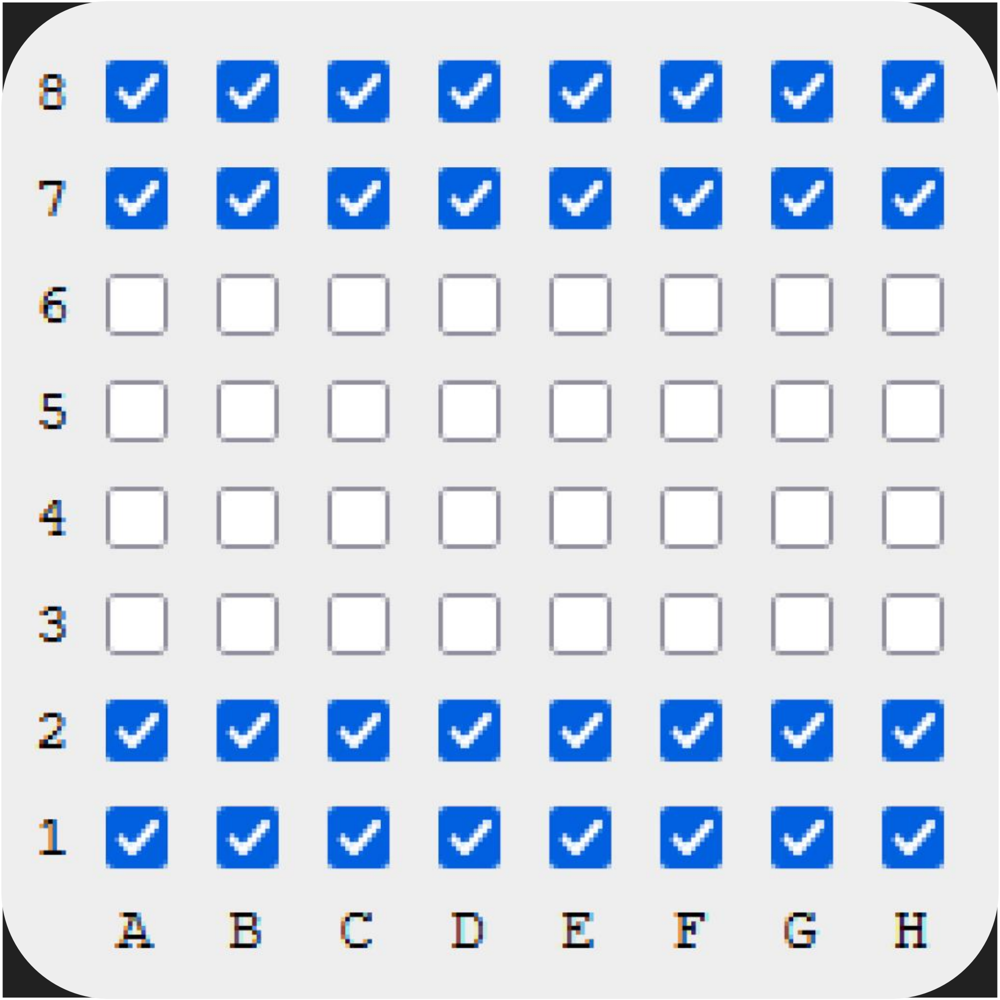
- Extras:
  - En Passant Capture: captures enemy pawn during double step
  - Castling
  - Pawn Promotion

Source: https://computerchess.com/Schach-lernen/Schachregeln-einfach-erklaert/

How to represent the Chessboard?

Bit Boards

- chessboard representation via 64 bits (one for each square)
- faster and easier computation via bit operations (optimized for chess AI implementation)



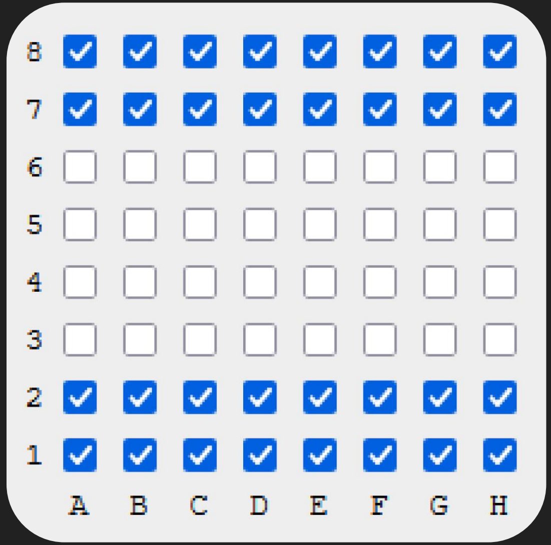
8	☑	☑	☑	☑	☑	☑	☑	☑
7	☑	☑	☑	☑	☑	☑	☑	☑
6	□	□	□	□	□	□	□	□
5	□	□	□	□	□	□	□	□
4	□	□	□	□	□	□	□	□
3	□	□	□	□	□	□	□	□
2	☑	☑	☑	☑	☑	☑	☑	☑
1	☑	☑	☑	☑	☑	☑	☑	☑
	A	B	C	D	E	F	G	H

hex: ffff00000000ffff

- each chess piece type gets its own bit board in each color
- meaning a bit board solution consists of 12 bit boards

all bitboard presentations made with https://gekomad.github.io/Cinnamon/BitboardCalculator/

Bit Boards: Indices



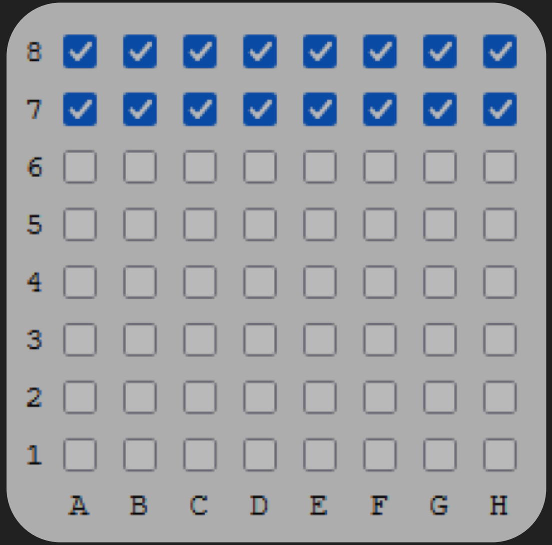
8	☑	☑	☑	☑	☑	☑	☑	☑
7	☑	☑	☑	☑	☑	☑	☑	☑
6	□	□	□	□	□	□	□	□
5	□	□	□	□	□	□	□	□
4	□	□	□	□	□	□	□	□
3	□	□	□	□	□	□	□	□
2	☑	☑	☑	☑	☑	☑	☑	☑
1	☑	☑	☑	☑	☑	☑	☑	☑
	A	B	C	D	E	F	G	H

00	01	02	03	04	05	06	07
08	09	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

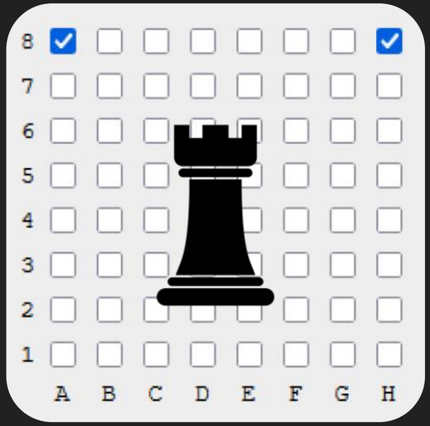
- flipped in the rendering to have view from White Player's perspective

Source: https://gekomad.github.io/Cinnamon/BitboardCalculator/

Bit Boards: Chess Pieces

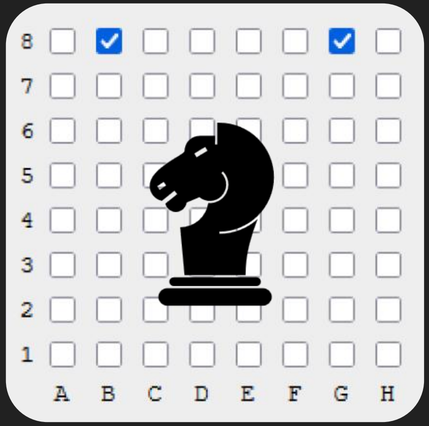


8	☑	☑	☑	☑	☑	☑	☑	☑
7	☑	☑	☑	☑	☑	☑	☑	☑
6	□	□	□	□	□	□	□	□
5	□	□	□	□	□	□	□	□
4	□	□	□	□	□	□	□	□
3	□	□	□	□	□	□	□	□
2	□	□	□	□	□	□	□	□
1	□	□	□	□	□	□	□	□
	A	B	C	D	E	F	G	H



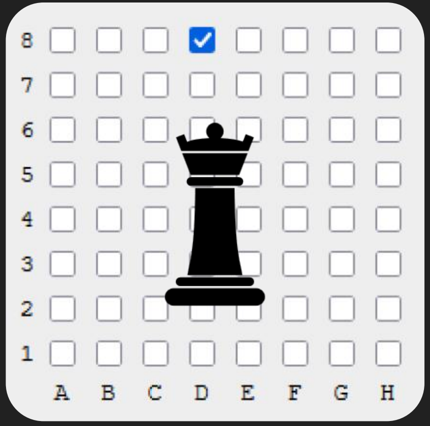
8	☑	□	□	□	□	□	□	☑
7	□	□	□	□	□	□	□	□
6	□	□	□	□	□	□	□	□
5	□	□	□	□	□	□	□	□
4	□	□	□	□	□	□	□	□
3	□	□	□	□	□	□	□	□
2	□	□	□	□	□	□	□	□
1	□	□	□	□	□	□	□	□
	A	B	C	D	E	F	G	H

0x81



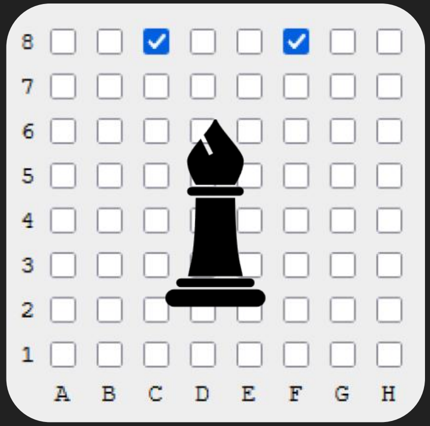
8	□	☑	□	□	□	□	☑	□
7	□	□	□	□	□	□	□	□
6	□	□	□	□	□	□	□	□
5	□	□	□	□	□	□	□	□
4	□	□	□	□	□	□	□	□
3	□	□	□	□	□	□	□	□
2	□	□	□	□	□	□	□	□
1	□	□	□	□	□	□	□	□
	A	B	C	D	E	F	G	H

0x42



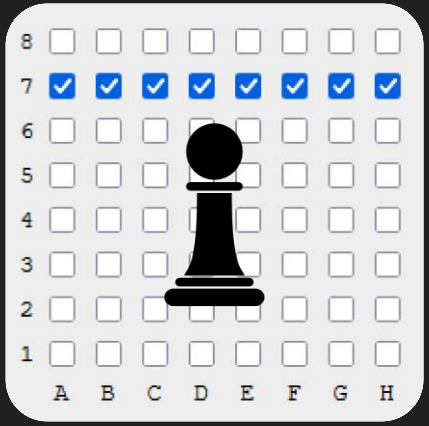
8	□	□	□	☑	□	□	□	□
7	□	□	□	□	□	□	□	□
6	□	□	□	□	□	□	□	□
5	□	□	□	□	□	□	□	□
4	□	□	□	□	□	□	□	□
3	□	□	□	□	□	□	□	□
2	□	□	□	□	□	□	□	□
1	□	□	□	□	□	□	□	□
	A	B	C	D	E	F	G	H

0x10



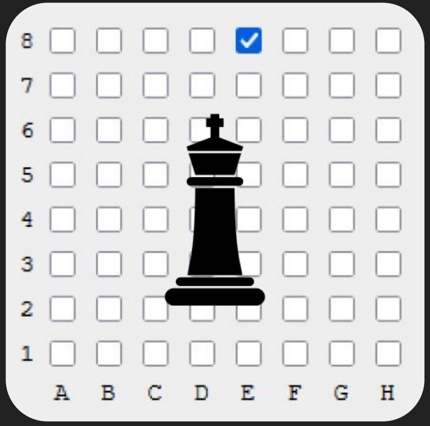
8	□	□	☑	□	□	☑	□	□
7	□	□	□	□	□	□	□	□
6	□	□	□	□	□	□	□	□
5	□	□	□	□	□	□	□	□
4	□	□	□	□	□	□	□	□
3	□	□	□	□	□	□	□	□
2	□	□	□	□	□	□	□	□
1	□	□	□	□	□	□	□	□
	A	B	C	D	E	F	G	H

0x24



8	□	□	□	□	□	□	□	□
7	☑	☑	☑	☑	☑	☑	☑	☑
6	□	□	□	□	□	□	□	□
5	□	□	□	□	□	□	□	□
4	□	□	□	□	□	□	□	□
3	□	□	□	□	□	□	□	□
2	□	□	□	□	□	□	□	□
1	□	□	□	□	□	□	□	□
	A	B	C	D	E	F	G	H

0xFF00



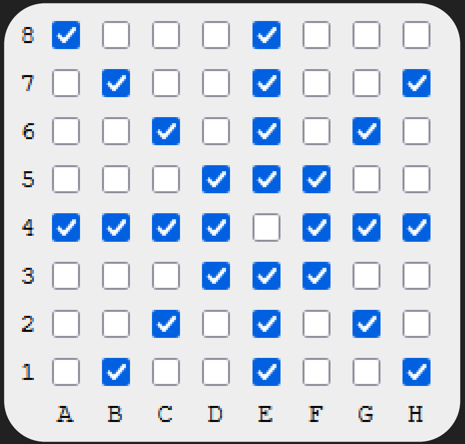
8	□	□	□	□	☑	□	□	□
7	□	□	□	□	□	□	□	□
6	□	□	□	□	□	□	□	□
5	□	□	□	□	□	□	□	□
4	□	□	□	□	□	□	□	□
3	□	□	□	□	□	□	□	□
2	□	□	□	□	□	□	□	□
1	□	□	□	□	□	□	□	□
	A	B	C	D	E	F	G	H

0x8

Source: https://staging.svgrepo.com/collection/chess-vectors/
https://gekomad.github.io/Cinnamon/BitboardCalculator/

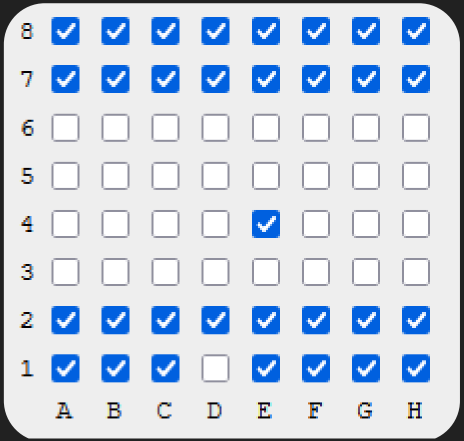
Bit Boards Calculations

Example:
WQueen on E4

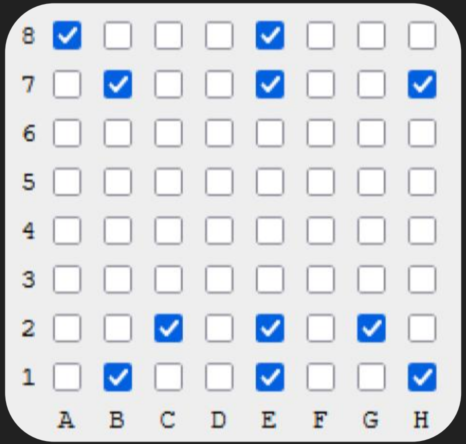


8	☑	☐	☐	☐	☑	☐	☐	☐
7	☐	☑	☐	☐	☑	☐	☐	☑
6	☐	☐	☑	☐	☑	☐	☑	☐
5	☐	☐	☐	☑	☑	☑	☐	☐
4	☑	☑	☑	☑	☐	☑	☑	☑
3	☐	☐	☐	☑	☑	☑	☐	☐
2	☐	☐	☑	☐	☑	☐	☑	☐
1	☐	☑	☐	☐	☑	☐	☐	☑
	A	B	C	D	E	F	G	H

AND
operation



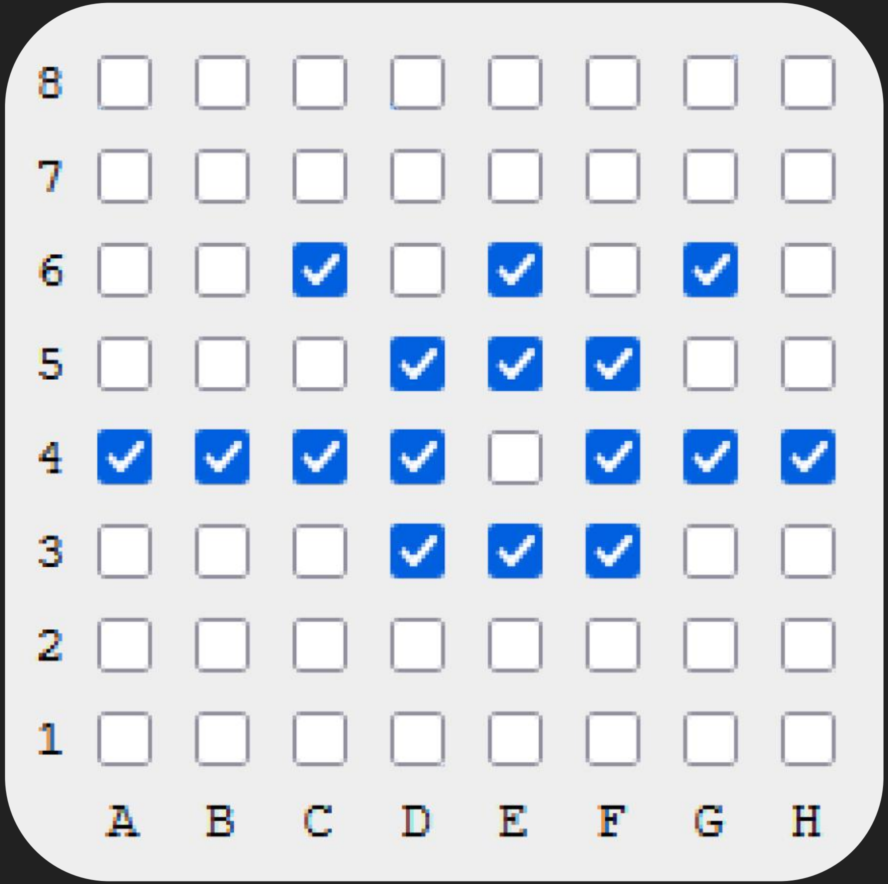
8	☑	☑	☑	☑	☑	☑	☑	☑
7	☑	☑	☑	☑	☑	☑	☑	☑
6	☐	☐	☐	☐	☐	☐	☐	☐
5	☐	☐	☐	☐	☐	☐	☐	☐
4	☐	☐	☐	☐	☑	☐	☐	☐
3	☐	☐	☐	☐	☐	☐	☐	☐
2	☑	☑	☑	☑	☑	☑	☑	☑
1	☑	☑	☑	☐	☑	☑	☑	☑
	A	B	C	D	E	F	G	H



8	☑	☐	☐	☐	☑	☐	☐	☐
7	☐	☑	☐	☐	☑	☐	☐	☑
6	☐	☐	☐	☐	☐	☐	☐	☐
5	☐	☐	☐	☐	☐	☐	☐	☐
4	☐	☐	☐	☐	☐	☐	☐	☐
3	☐	☐	☐	☐	☐	☐	☐	☐
2	☐	☐	☑	☐	☑	☐	☑	☐
1	☐	☑	☐	☐	☑	☐	☐	☑
	A	B	C	D	E	F	G	H

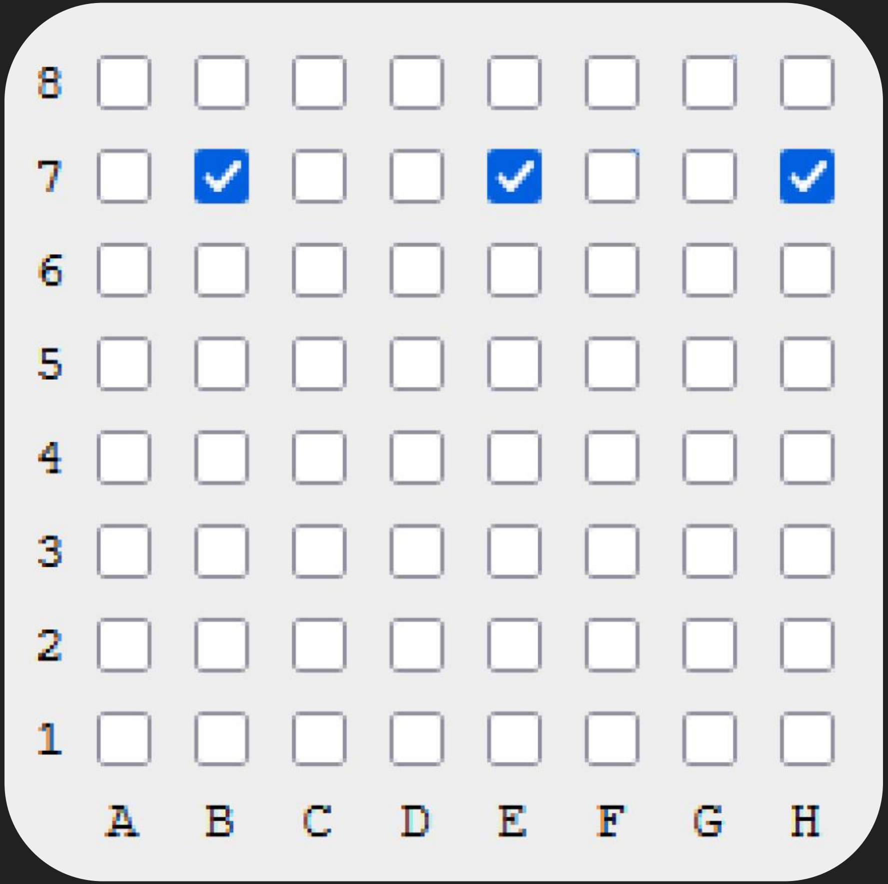
result: all possible
attacks on all other
pieces

Bit Boards Calculations



8	☐	☐	☐	☐	☐	☐	☐	☐
7	☐	☐	☐	☐	☐	☐	☐	☐
6	☐	☐	☑	☐	☑	☐	☑	☐
5	☐	☐	☐	☑	☑	☑	☐	☐
4	☑	☑	☑	☑	☐	☑	☑	☑
3	☐	☐	☐	☑	☑	☑	☐	☐
2	☐	☐	☐	☐	☐	☐	☐	☐
1	☐	☐	☐	☐	☐	☐	☐	☐
	A	B	C	D	E	F	G	H

Filtered Movement



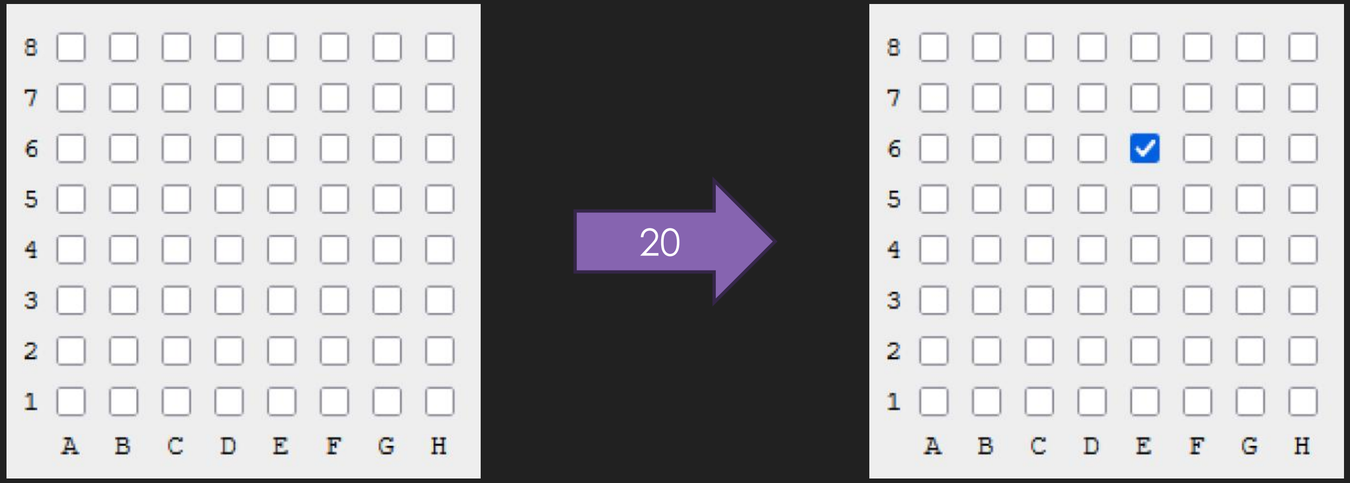
8	☐	☐	☐	☐	☐	☐	☐	☐
7	☐	☑	☐	☐	☑	☐	☐	☑
6	☐	☐	☐	☐	☐	☐	☐	☐
5	☐	☐	☐	☐	☐	☐	☐	☐
4	☐	☐	☐	☐	☐	☐	☐	☐
3	☐	☐	☐	☐	☐	☐	☐	☐
2	☐	☐	☐	☐	☐	☐	☐	☐
1	☐	☐	☐	☐	☐	☐	☐	☐
	A	B	C	D	E	F	G	H

Filtered Attack

Not True Bitboards

- not a proper bitboard implementation
- more like an amalgamation of both bitboards and normal array handling
- has the added speedboost and simplicity from bit operations
- helper functions:
  - FlipBit()
  - SwapBit()

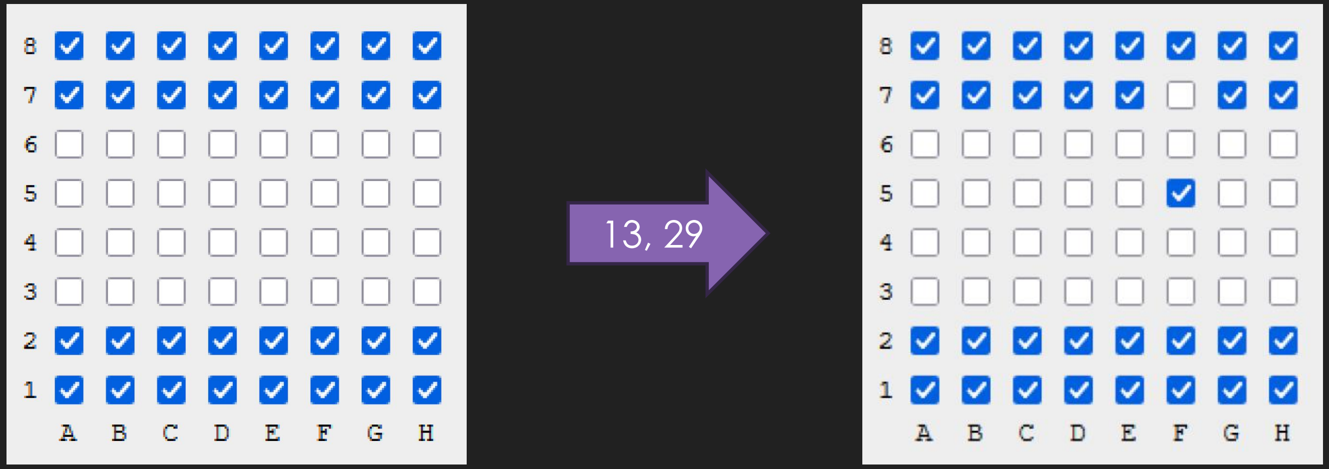
FlipBit(board, index)



8	☐	☐	☐	☐	☐	☐	☐	☐
7	☐	☐	☐	☐	☐	☐	☐	☐
6	☐	☐	☐	☐	☐	☐	☐	☐
5	☐	☐	☐	☐	☐	☐	☐	☐
4	☐	☐	☐	☐	☐	☐	☐	☐
3	☐	☐	☐	☐	☐	☐	☐	☐
2	☐	☐	☐	☐	☐	☐	☐	☐
1	☐	☐	☐	☐	☐	☐	☐	☐
	A	B	C	D	E	F	G	H

8	☐	☐	☐	☐	☐	☐	☐	☐
7	☐	☐	☐	☐	☐	☐	☐	☐
6	☐	☐	☐	☐	☑	☐	☐	☐
5	☐	☐	☐	☐	☐	☐	☐	☐
4	☐	☐	☐	☐	☐	☐	☐	☐
3	☐	☐	☐	☐	☐	☐	☐	☐
2	☐	☐	☐	☐	☐	☐	☐	☐
1	☐	☐	☐	☐	☐	☐	☐	☐
	A	B	C	D	E	F	G	H

SwapBit(board, index, index)

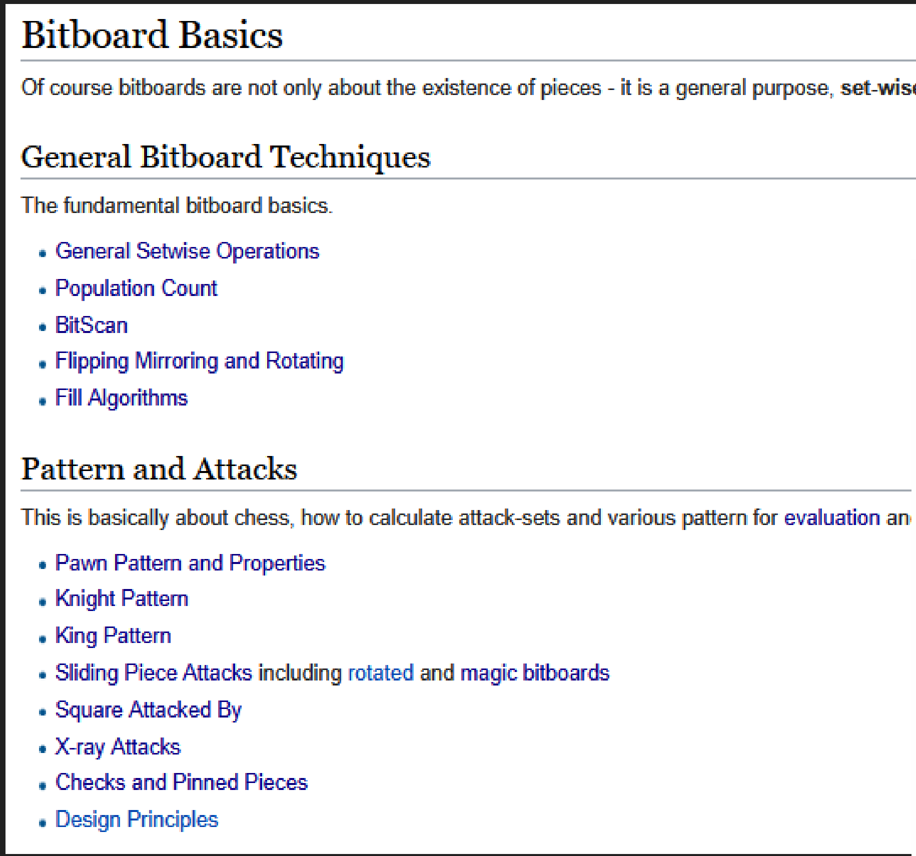


8	☑	☑	☑	☑	☑	☑	☑	☑
7	☑	☑	☑	☑	☑	☑	☑	☑
6	☐	☐	☐	☐	☐	☐	☐	☐
5	☐	☐	☐	☐	☐	☐	☐	☐
4	☐	☐	☐	☐	☐	☐	☐	☐
3	☐	☐	☐	☐	☐	☐	☐	☐
2	☑	☑	☑	☑	☑	☑	☑	☑
1	☑	☑	☑	☑	☑	☑	☑	☑
	A	B	C	D	E	F	G	H

8	☑	☑	☑	☑	☑	☑	☑	☑
7	☑	☑	☑	☑	☑	☐	☑	☑
6	☐	☐	☐	☐	☐	☐	☐	☐
5	☐	☐	☐	☐	☐	☑	☐	☐
4	☐	☐	☐	☐	☐	☐	☐	☐
3	☐	☐	☐	☐	☐	☐	☐	☐
2	☑	☑	☑	☑	☑	☑	☑	☑
1	☑	☑	☑	☑	☑	☑	☑	☑
	A	B	C	D	E	F	G	H

`A deep deep rabbit hole to fall into ...`

Source: https://www.chessprogramming.org/Bitboards



Bitboard Basics

Of course bitboards are not only about the existence of pieces - it is a general purpose, **set-wise** data-structure fitting in one 64-bit register. For example, a bitboard can represent things like attack- and defend sets, move-target sets and so on.

General Bitboard Techniques

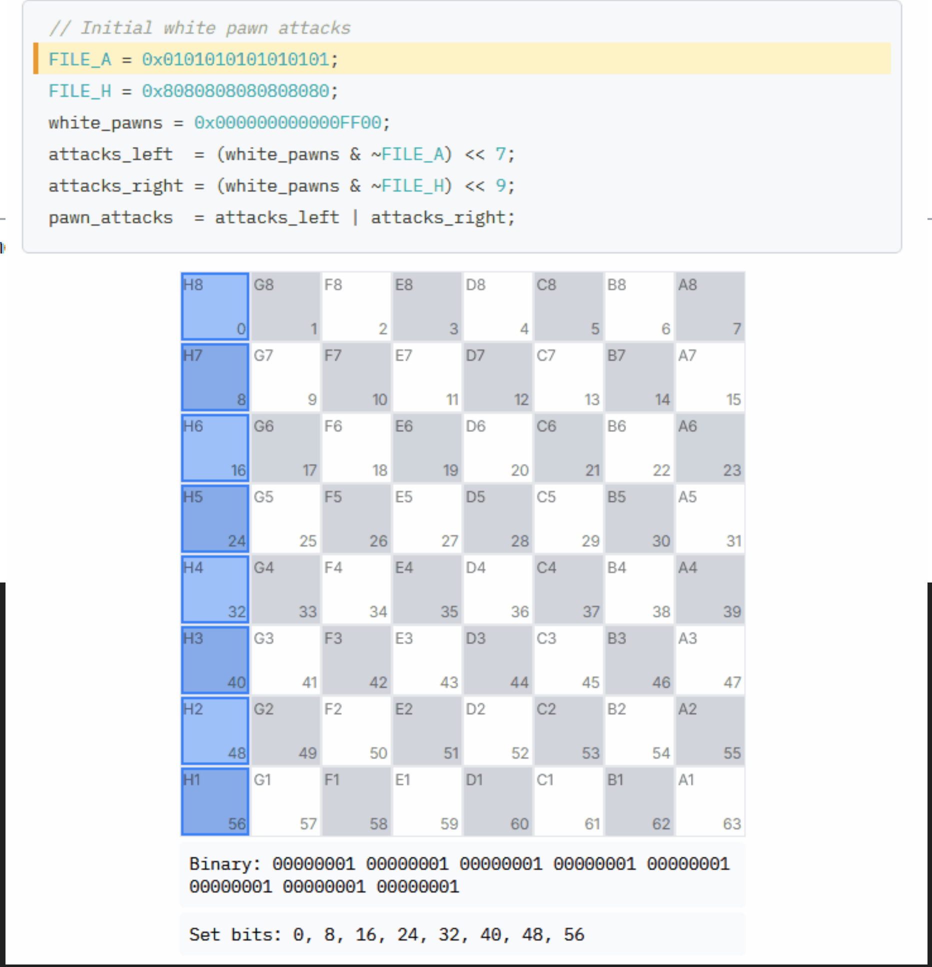
The fundamental bitboard basics.

- General Setwise Operations
- Population Count
- BitScan
- Flipping Mirroring and Rotating
- Fill Algorithms

Pattern and Attacks

This is basically about chess, how to calculate attack-sets and various pattern for evaluation an

- Pawn Pattern and Properties
- Knight Pattern
- King Pattern
- Sliding Piece Attacks including rotated and magic bitboards
- Square Attacked By
- X-ray Attacks
- Checks and Pinned Pieces
- Design Principles

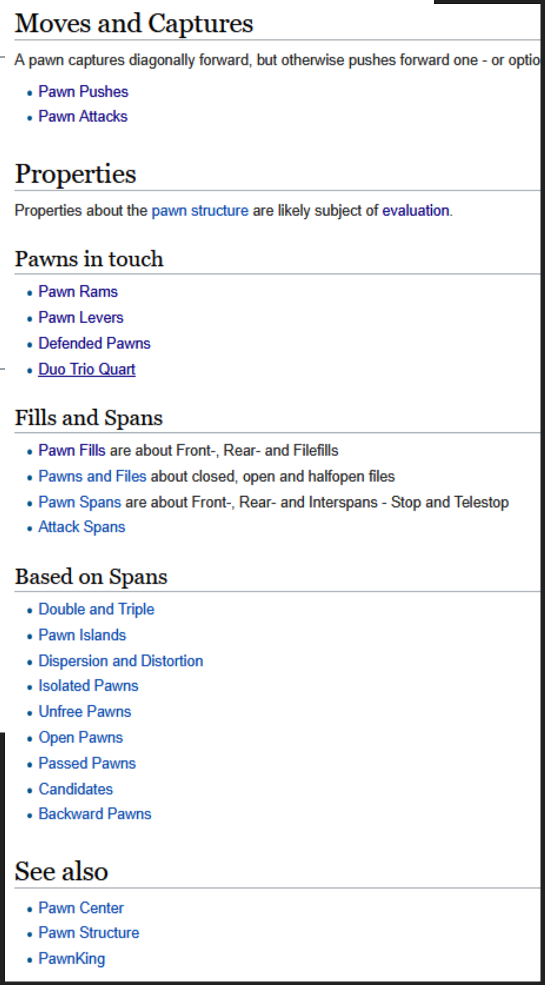


```
// Initial white pawn attacks
FILE_A = 0x0101010101010101;
FILE_H = 0x8080808080808080;
white_pawns = 0x000000000000FF00;
attacks_left  = (white_pawns & ~FILE_A) << 7;
attacks_right = (white_pawns & ~FILE_H) << 9;
pawn_attacks  = attacks_left | attacks_right;
```

H8 0	G8 1	F8 2	E8 3	D8 4	C8 5	B8 6	A8 7
H7 8	G7 9	F7 10	E7 11	D7 12	C7 13	B7 14	A7 15
H6 16	G6 17	F6 18	E6 19	D6 20	C6 21	B6 22	A6 23
H5 24	G5 25	F5 26	E5 27	D5 28	C5 29	B5 30	A5 31
H4 32	G4 33	F4 34	E4 35	D4 36	C4 37	B4 38	A4 39
H3 40	G3 41	F3 42	E3 43	D3 44	C3 45	B3 46	A3 47
H2 48	G2 49	F2 50	E2 51	D2 52	C2 53	B2 54	A2 55
H1 56	G1 57	F1 58	E1 59	D1 60	C1 61	B1 62	A1 63

```
Binary: 00000001 00000001 00000001 00000001 00000001
00000001 00000001 00000001
```

```
Set bits: 0, 8, 16, 24, 32, 40, 48, 56
```



Moves and Captures

A pawn captures diagonally forward, but otherwise pushes forward one - or optio

- Pawn Pushes
- Pawn Attacks

Properties

Properties about the pawn structure are likely subject of evaluation.

Pawns in touch

- Pawn Rams
- Pawn Levers
- Defended Pawns
- Duo Trio Quart

Fills and Spans

- Pawn Fills are about Front-, Rear- and Filefills
- Pawns and Files about closed, open and halfopen files
- Pawn Spans are about Front-, Rear- and Interspans - Stop and Telestop
- Attack Spans

Based on Spans

- Double and Triple
- Pawn Islands
- Dispersion and Distortion
- Isolated Pawns
- Unfree Pawns
- Open Pawns
- Passed Pawns
- Candidates
- Backward Pawns

See also

- Pawn Center
- Pawn Structure
- PawnKing

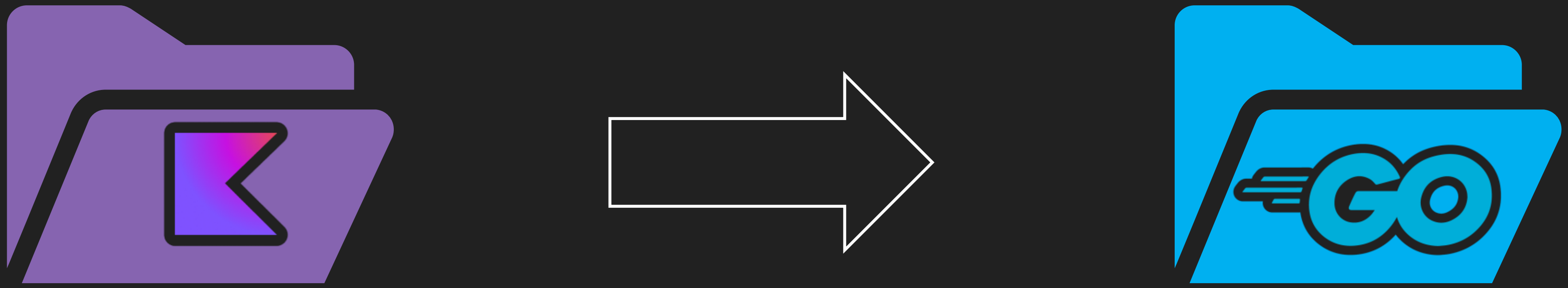
SO MANY EDGE CASES.

Source: https://healeycodes.com/visualizing-chess-bitboards

Implementation

*Code was shortened and adjusted for readability

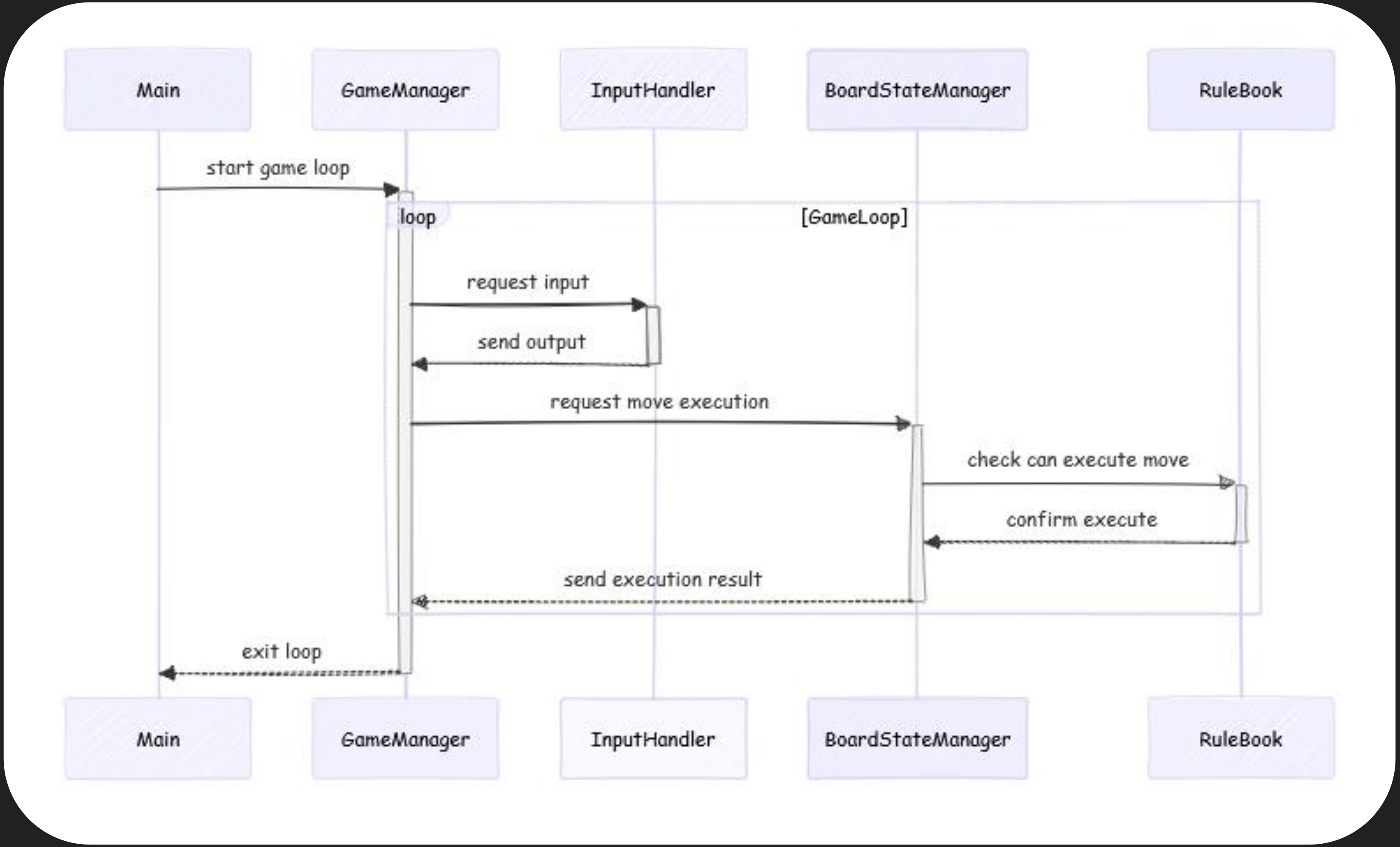
Comparison Method



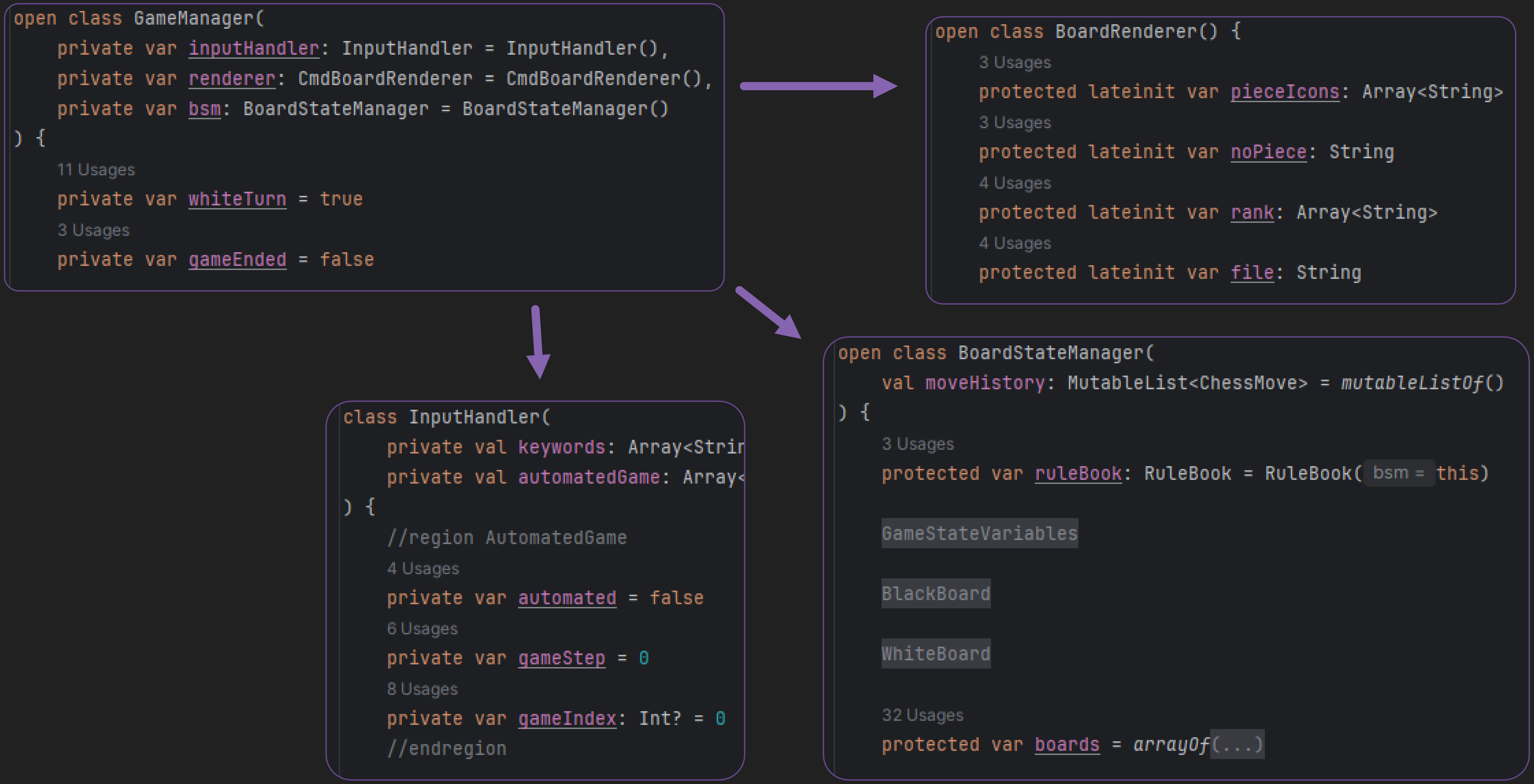
- used copy paste to see how easy it was to do a one-to-one implementation of the same code architecture

- problems: unused Go potential, circular dependencies

Chess Engine Flow



Kotlin



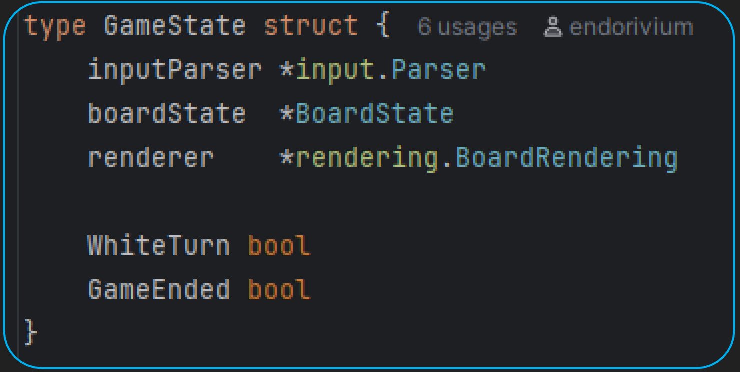
```
open class GameManager(
    private var inputHandler: InputHandler = InputHandler(),
    private var renderer: CmdBoardRenderer = CmdBoardRenderer(),
    private var bsm: BoardStateManager = BoardStateManager()
) {
    11 Usages
    private var whiteTurn = true
    3 Usages
    private var gameEnded = false
```

```
open class BoardRenderer() {
    3 Usages
    protected lateinit var pieceIcons: Array<String>
    3 Usages
    protected lateinit var noPiece: String
    4 Usages
    protected lateinit var rank: Array<String>
    4 Usages
    protected lateinit var file: String
```

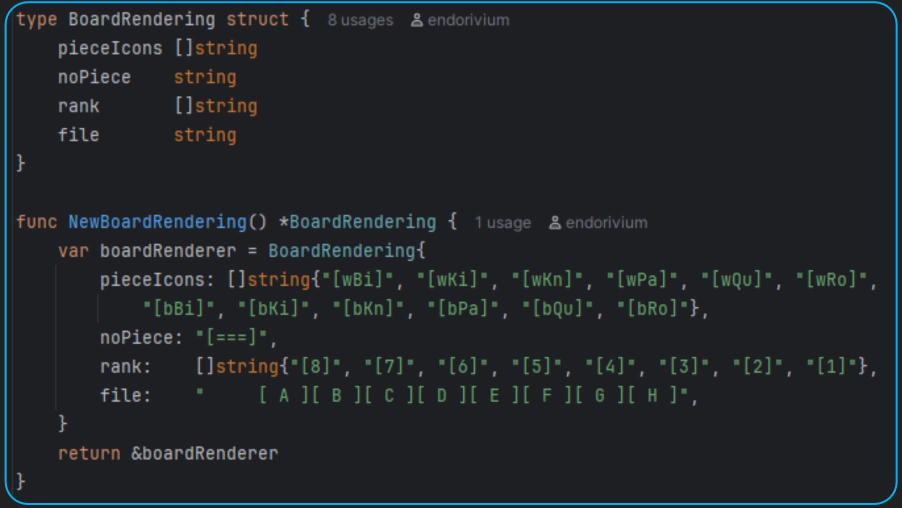
```
class InputHandler(
    private val keywords: Array<Strin
    private val automatedGame: Array<
) {
    //region AutomatedGame
    4 Usages
    private var automated = false
    6 Usages
    private var gameStep = 0
    8 Usages
    private var gameIndex: Int? = 0
    //endregion
```

```
open class BoardStateManager(
    val moveHistory: MutableList<ChessMove> = mutableListOf()
) {
    3 Usages
    protected var ruleBook: RuleBook = RuleBook( bsm = this)

    GameStateVariables

    BlackBoard

    WhiteBoard

    32 Usages
    protected var boards = arrayOf(...)
```

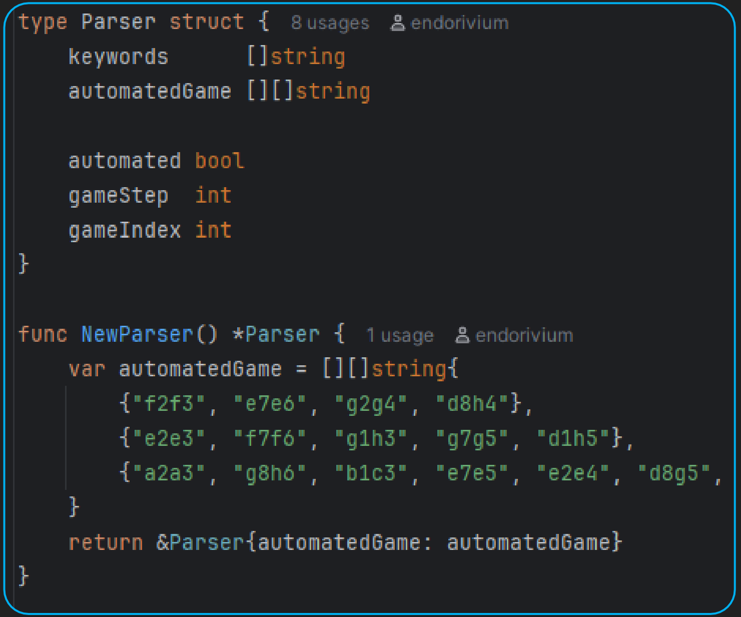
Go



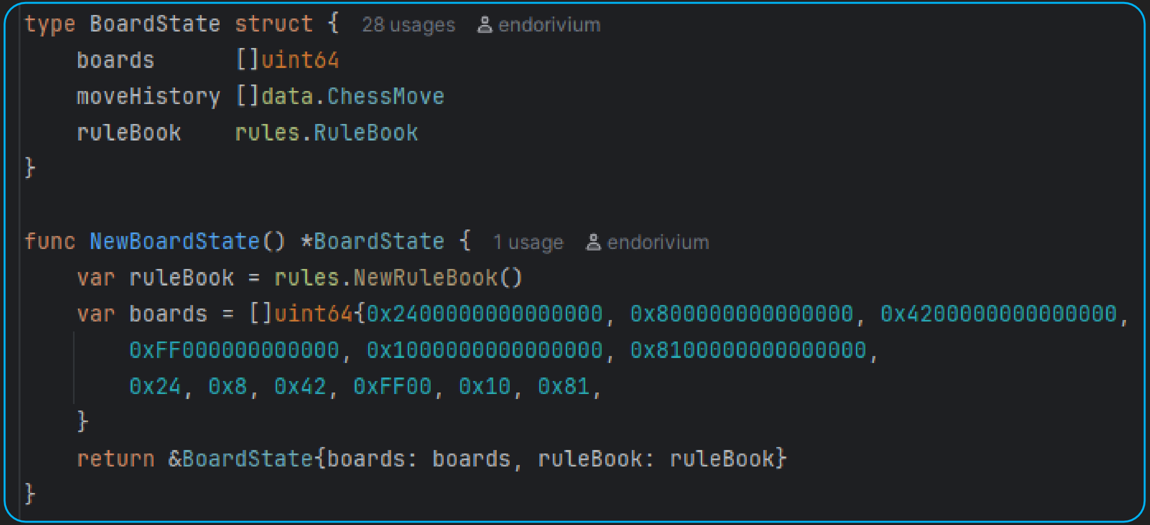
```go
type GameState struct {  6 usages  endorivium
    inputParser *input.Parser
    boardState  *BoardState
    renderer    *rendering.BoardRendering

    WhiteTurn bool
    GameEnded bool
}
```



```go
type BoardRendering struct {  8 usages  endorivium
    pieceIcons []string
    noPiece    string
    rank       []string
    file       string
}

func NewBoardRendering() *BoardRendering {  1 usage  endorivium
    var boardRenderer = BoardRendering{
        pieceIcons: []string{"[wBi]", "[wKi]", "[wKn]", "[wPa]", "[wQu]", "[wRo]",
            "[bBi]", "[bKi]", "[bKn]", "[bPa]", "[bQu]", "[bRo]"},
        noPiece: "[===]",
        rank:    []string{"[8]", "[7]", "[6]", "[5]", "[4]", "[3]", "[2]", "[1]"},
        file:    "     [ A ][ B ][ C ][ D ][ E ][ F ][ G ][ H ]",
    }
    return &boardRenderer
}
```



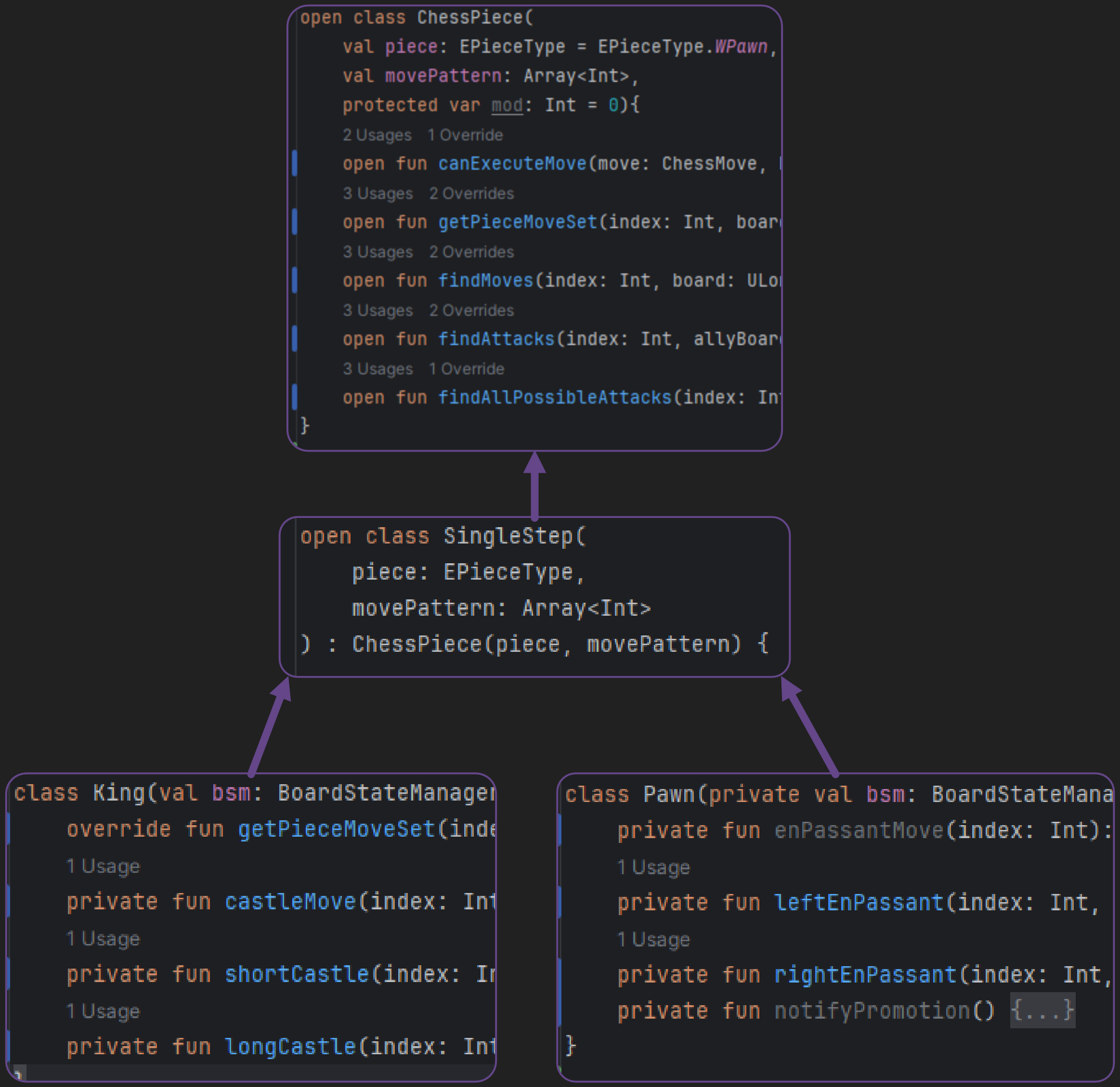
```go
type Parser struct {  8 usages  endorivium
    keywords      []string
    automatedGame [][]string

    automated bool
    gameStep  int
    gameIndex int
}

func NewParser() *Parser {  1 usage  endorivium
    var automatedGame = [][]string{
        {"f2f3", "e7e6", "g2g4", "d8h4"},
        {"e2e3", "f7f6", "g1h3", "g7g5", "d1h5"},
        {"a2a3", "g8h6", "b1c3", "e7e5", "e2e4", "d8g5",
    }
    return &Parser{automatedGame: automatedGame}
}
```



```go
type BoardState struct {  28 usages  endorivium
    boards      []uint64
    moveHistory []data.ChessMove
    ruleBook    rules.RuleBook
}

func NewBoardState() *BoardState {  1 usage  endorivium
    var ruleBook = rules.NewRuleBook()
    var boards = []uint64{0x2400000000000000, 0x8000000000000000, 0x4200000000000000,
        0xFF000000000000, 0x1000000000000000, 0x8100000000000000,
        0x24, 0x8, 0x42, 0xFF00, 0x10, 0x81,
    }
    return &BoardState{boards: boards, ruleBook: ruleBook}
}
```

Kotlin: Code Structure

- ChessPiece as default/ base implementation
  - assumes the default piece is white and can move indefinitely according to its movePattern
- SingleStep only moves one step in any given direction (according to movePattern)
- King implements Castling
- Pawn implements En Passant and Promotion



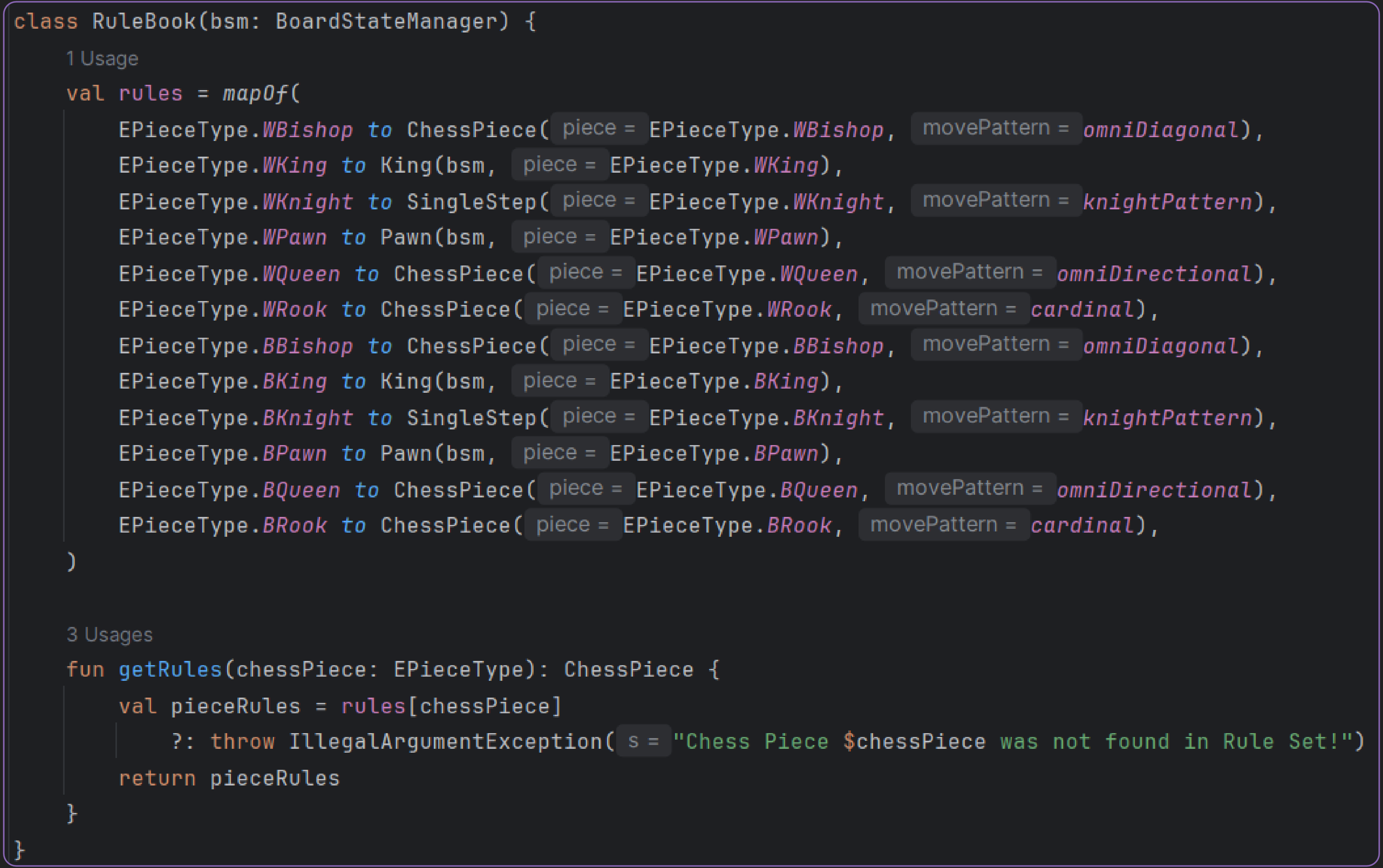
```kotlin
open class ChessPiece(
    val piece: EPieceType = EPieceType.WPawn,
    val movePattern: Array<Int>,
    protected var mod: Int = 0){
    2 Usages   1 Override
    open fun canExecuteMove(move: ChessMove,
    3 Usages   2 Overrides
    open fun getPieceMoveSet(index: Int, boar
    3 Usages   2 Overrides
    open fun findMoves(index: Int, board: ULo
    3 Usages   2 Overrides
    open fun findAttacks(index: Int, allyBoar
    3 Usages   1 Override
    open fun findAllPossibleAttacks(index: In
}
```

```kotlin
open class SingleStep(
    piece: EPieceType,
    movePattern: Array<Int>
) : ChessPiece(piece, movePattern) {
```

```kotlin
class King(val bsm: BoardStateManager
    override fun getPieceMoveSet(inde
    1 Usage
    private fun castleMove(index: Int
    1 Usage
    private fun shortCastle(index: In
    1 Usage
    private fun longCastle(index: Int
```

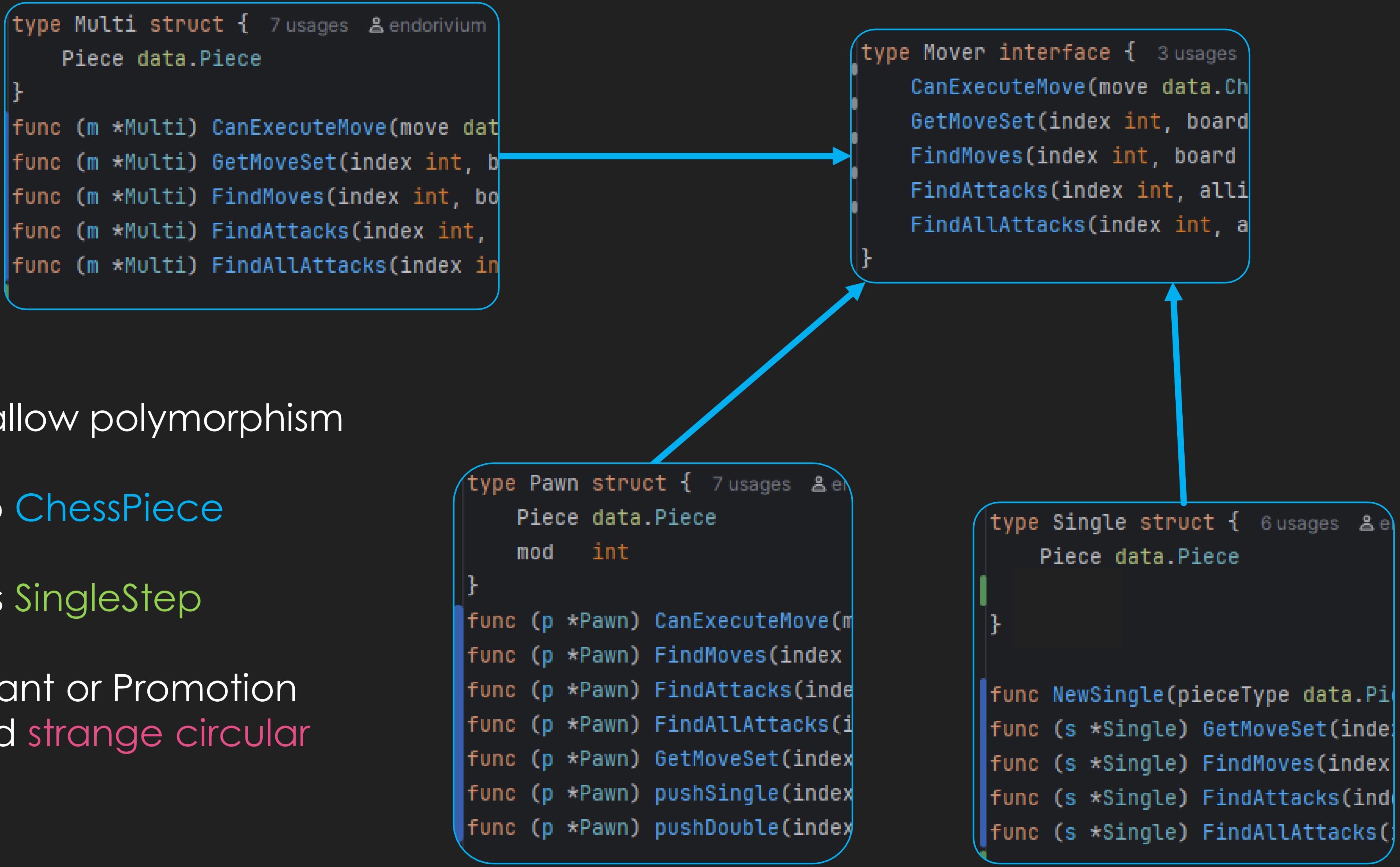
```kotlin
class Pawn(private val bsm: BoardStateMana
    private fun enPassantMove(index: Int):
    1 Usage
    private fun leftEnPassant(index: Int,
    1 Usage
    private fun rightEnPassant(index: Int,
    private fun notifyPromotion() {...}
}
```

Kotlin: Code Structure



```
class RuleBook(bsm: BoardStateManager) {
    1 Usage
    val rules = mapOf(
        EPieceType.WBishop to ChessPiece( piece = EPieceType.WBishop,  movePattern = omniDiagonal),
        EPieceType.WKing to King(bsm,  piece = EPieceType.WKing),
        EPieceType.WKnight to SingleStep( piece = EPieceType.WKnight,  movePattern = knightPattern),
        EPieceType.WPawn to Pawn(bsm,  piece = EPieceType.WPawn),
        EPieceType.WQueen to ChessPiece( piece = EPieceType.WQueen,  movePattern = omniDirectional),
        EPieceType.WRook to ChessPiece( piece = EPieceType.WRook,  movePattern = cardinal),
        EPieceType.BBishop to ChessPiece( piece = EPieceType.BBishop,  movePattern = omniDiagonal),
        EPieceType.BKing to King(bsm,  piece = EPieceType.BKing),
        EPieceType.BKnight to SingleStep( piece = EPieceType.BKnight,  movePattern = knightPattern),
        EPieceType.BPawn to Pawn(bsm,  piece = EPieceType.BPawn),
        EPieceType.BQueen to ChessPiece( piece = EPieceType.BQueen,  movePattern = omniDirectional),
        EPieceType.BRook to ChessPiece( piece = EPieceType.BRook,  movePattern = cardinal),
    )

    3 Usages
    fun getRules(chessPiece: EPieceType): ChessPiece {
        val pieceRules = rules[chessPiece]
            ?: throw IllegalArgumentException( s = "Chess Piece $chessPiece was not found in Rule Set!")
        return pieceRules
    }
}
```

Go: Code Structure



```
type Multi struct {  7 usages  endorivium
    Piece data.Piece
}
func (m *Multi) CanExecuteMove(move dat
func (m *Multi) GetMoveSet(index int, b
func (m *Multi) FindMoves(index int, bo
func (m *Multi) FindAttacks(index int,
func (m *Multi) FindAllAttacks(index in
```

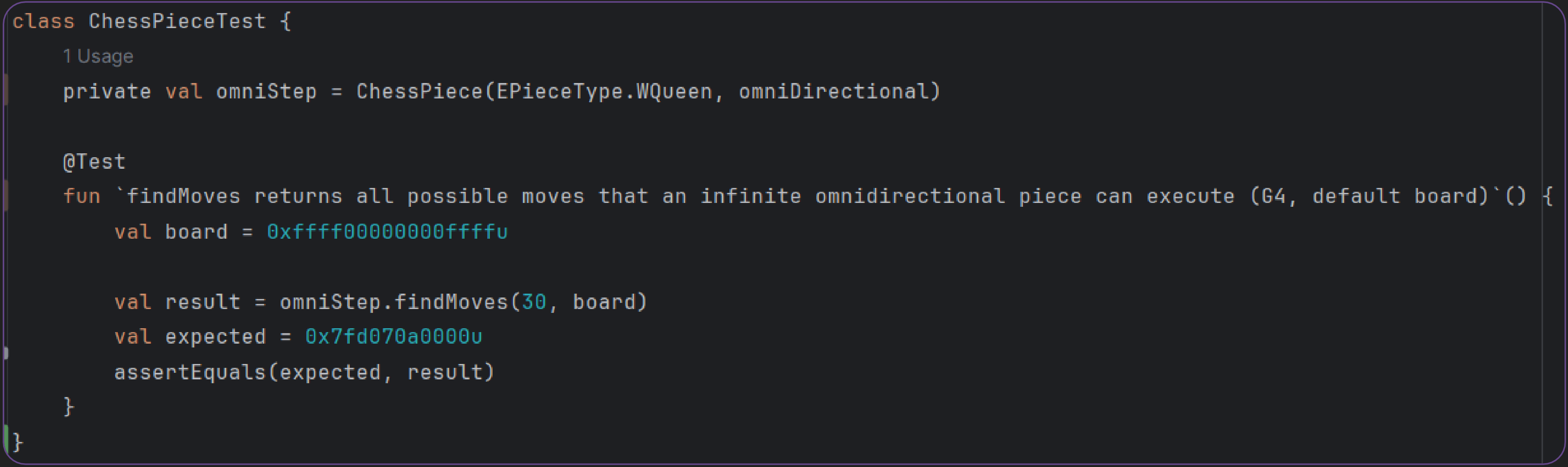
```
type Mover interface {  3 usages
    CanExecuteMove(move data.Ch
    GetMoveSet(index int, board
    FindMoves(index int, board
    FindAttacks(index int, alli
    FindAllAttacks(index int, a
}
```

```
type Pawn struct {  7 usages  e
    Piece data.Piece
    mod   int
}
func (p *Pawn) CanExecuteMove(m
func (p *Pawn) FindMoves(index
func (p *Pawn) FindAttacks(inde
func (p *Pawn) FindAllAttacks(i
func (p *Pawn) GetMoveSet(index
func (p *Pawn) pushSingle(index
func (p *Pawn) pushDouble(index
```

```
type Single struct {  6 usages  e
    Piece data.Piece

}

func NewSingle(pieceType data.Pi
func (s *Single) GetMoveSet(inde
func (s *Single) FindMoves(index
func (s *Single) FindAttacks(ind
func (s *Single) FindAllAttacks(
```

- Mover interface to allow polymorphism
- Multi is equivalent to ChessPiece
- Single is the same as SingleStep
- No Castling, En Passant or Promotion because of time and strange circular dependencies

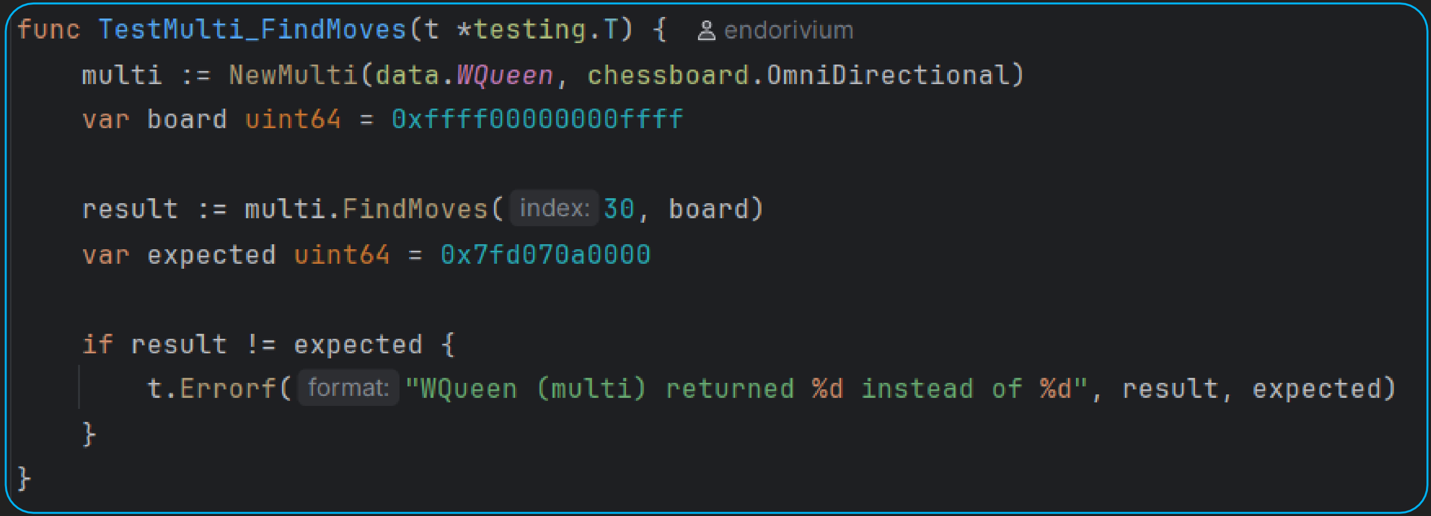
Testing



```kotlin
class ChessPieceTest {
    1 Usage
    private val omniStep = ChessPiece(EPieceType.WQueen, omniDirectional)

    @Test
    fun `findMoves returns all possible moves that an infinite omnidirectional piece can execute (G4, default board)`() {
        val board = 0xffff00000000ffffu

        val result = omniStep.findMoves(30, board)
        val expected = 0x7fd070a0000u
        assertEquals(expected, result)
    }
}
```

- Testing was easy in both languages

- Except for testing with dependencies in Kotlin (mock libraries did not work)



```go
func TestMulti_FindMoves(t *testing.T) {  endorivium
    multi := NewMulti(data.WQueen, chessboard.OmniDirectional)
    var board uint64 = 0xffff00000000ffff

    result := multi.FindMoves( index: 30, board)
    var expected uint64 = 0x7fd070a0000

    if result != expected {
        t.Errorf( format: "WQueen (multi) returned %d instead of %d", result, expected)
    }
}
```

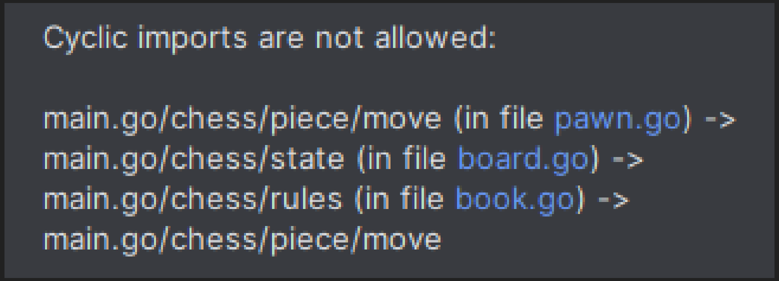
Demo Break

Some Thoughts

- initially confusing but ultimately not that difficult to do a one-to-one implementation

- however, could not get Pawn and King implementations to work
  - needed: reference to board state manager
  - but: introduced circular dependency

- could not fix it in time

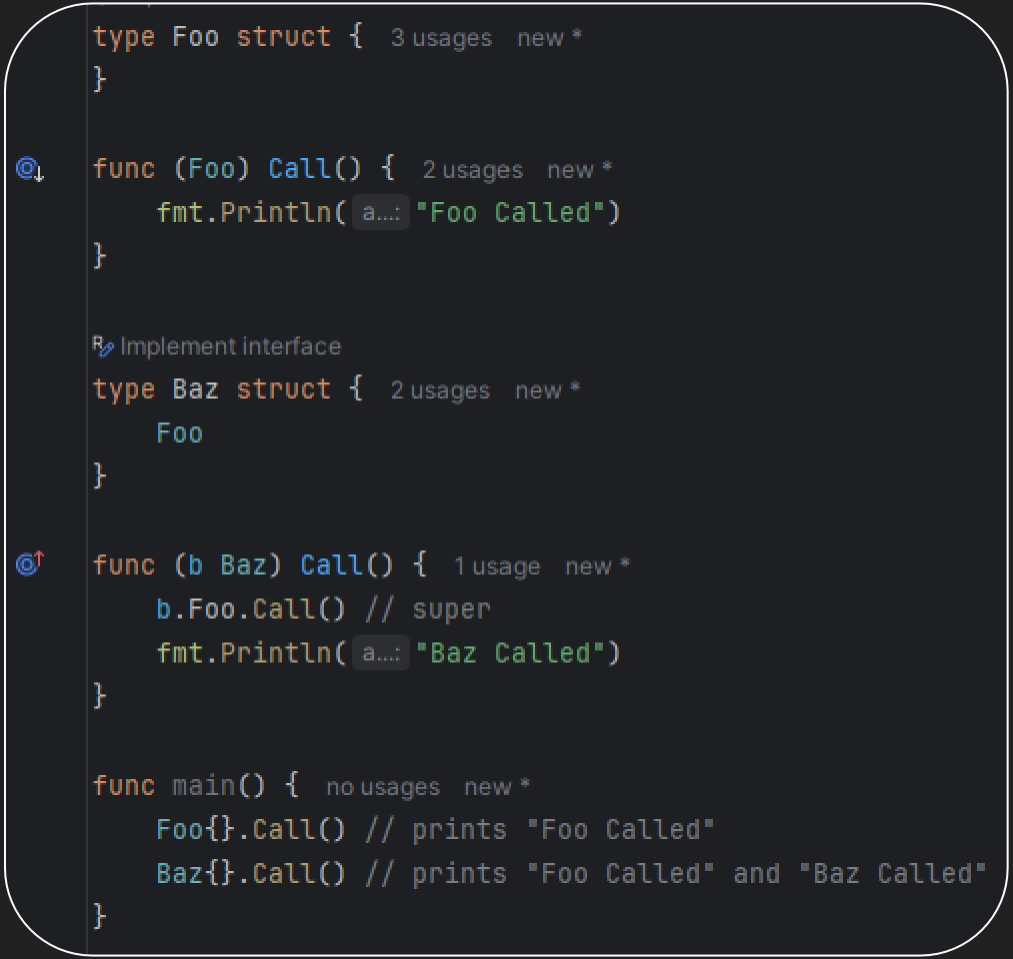


```
Cyclic imports are not allowed:

main.go/chess/piece/move (in file pawn.go) ->
main.go/chess/state (in file board.go) ->
main.go/chess/rules (in file book.go) ->
main.go/chess/piece/move
```

Addendum (Annoyances/ Improvements)

Method Overriding in Go

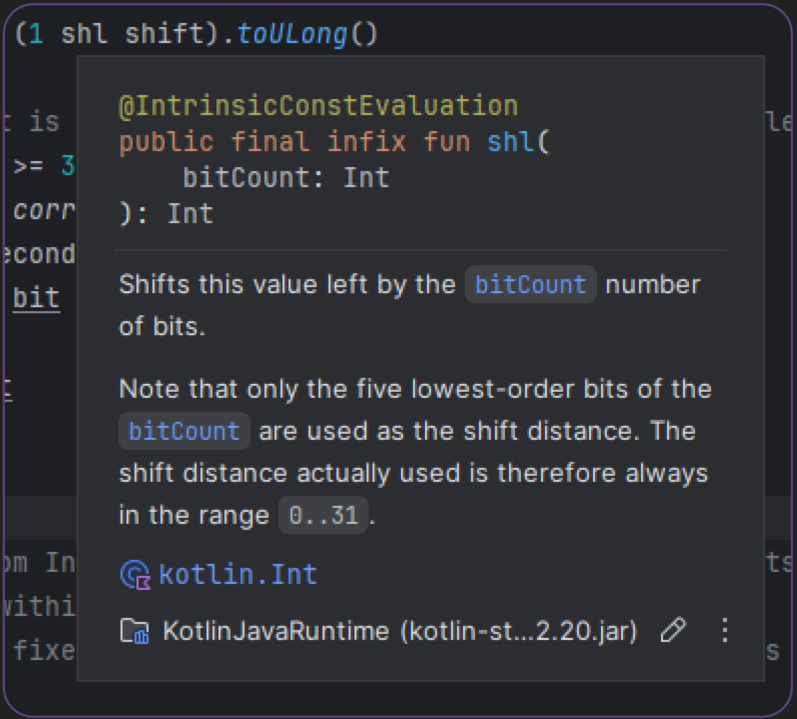
- function overriding (or at least a mimicry of it) is actually possible in Go
- override is even marked in Goland
- but: will raise error if there is ambiguity due to composition (struct has both Baz and Foo and then calls Call())



```
type Foo struct {  3 usages  new *
}

func (Foo) Call() {  2 usages  new *
    fmt.Println( a...: "Foo Called")
}

Implement interface
type Baz struct {  2 usages  new *
    Foo
}

func (b Baz) Call() {  1 usage  new *
    b.Foo.Call() // super
    fmt.Println( a...: "Baz Called")
}

func main() {  no usages  new *
    Foo{}.Call() // prints "Foo Called"
    Baz{}.Call() // prints "Foo Called" and "Baz Called"
}
```

Source: https://medium.com/random-go-tips/method-overriding-680cfd51ce40

Kotlin Bit Operations



```
(1 shl shift).toULong()
```

```
@IntrinsicConstEvaluation
public final infix fun shl(
    bitCount: Int
): Int
```

Shifts this value left by the `bitCount` number of bits.

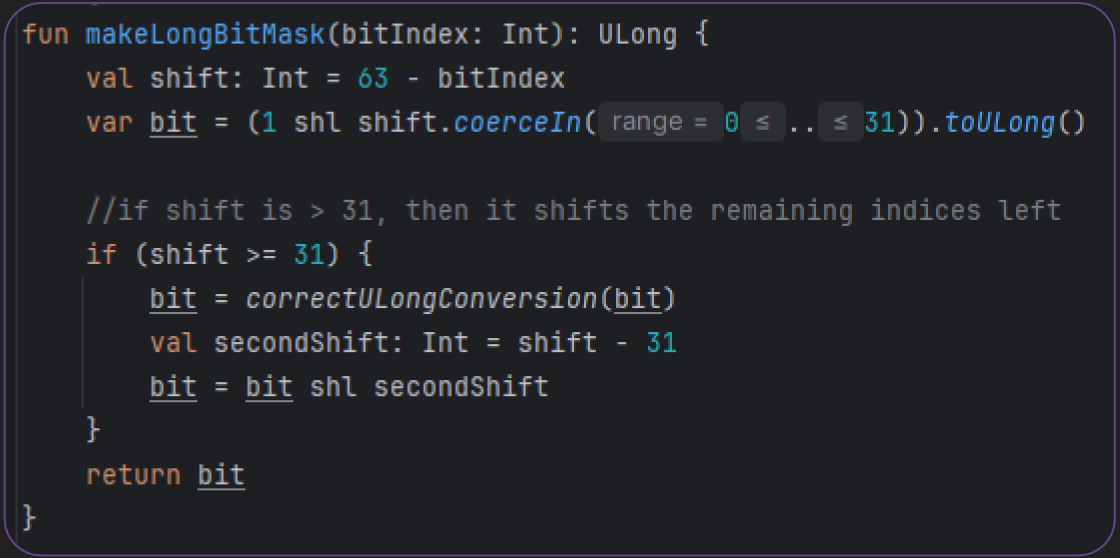
Note that only the five lowest-order bits of the `bitCount` are used as the shift distance. The shift distance actually used is therefore always in the range `0..31`.

kotlin.Int

KotlinJavaRuntime (kotlin-st...2.20.jar)

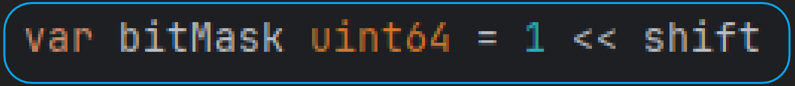
- inferred data type ruined bit operation in Kotlin

- did not catch it in time



```
fun makeLongBitMask(bitIndex: Int): ULong {
    val shift: Int = 63 - bitIndex
    var bit = (1 shl shift.coerceIn( range = 0 ≤ .. ≤ 31)).toULong()

    //if shift is > 31, then it shifts the remaining indices left
    if (shift >= 31) {
        bit = correctULongConversion(bit)
        val secondShift: Int = shift - 31
        bit = bit shl secondShift
    }
    return bit
}
```

- thought I had to compensate for language deficit and wrote own function (time waste)



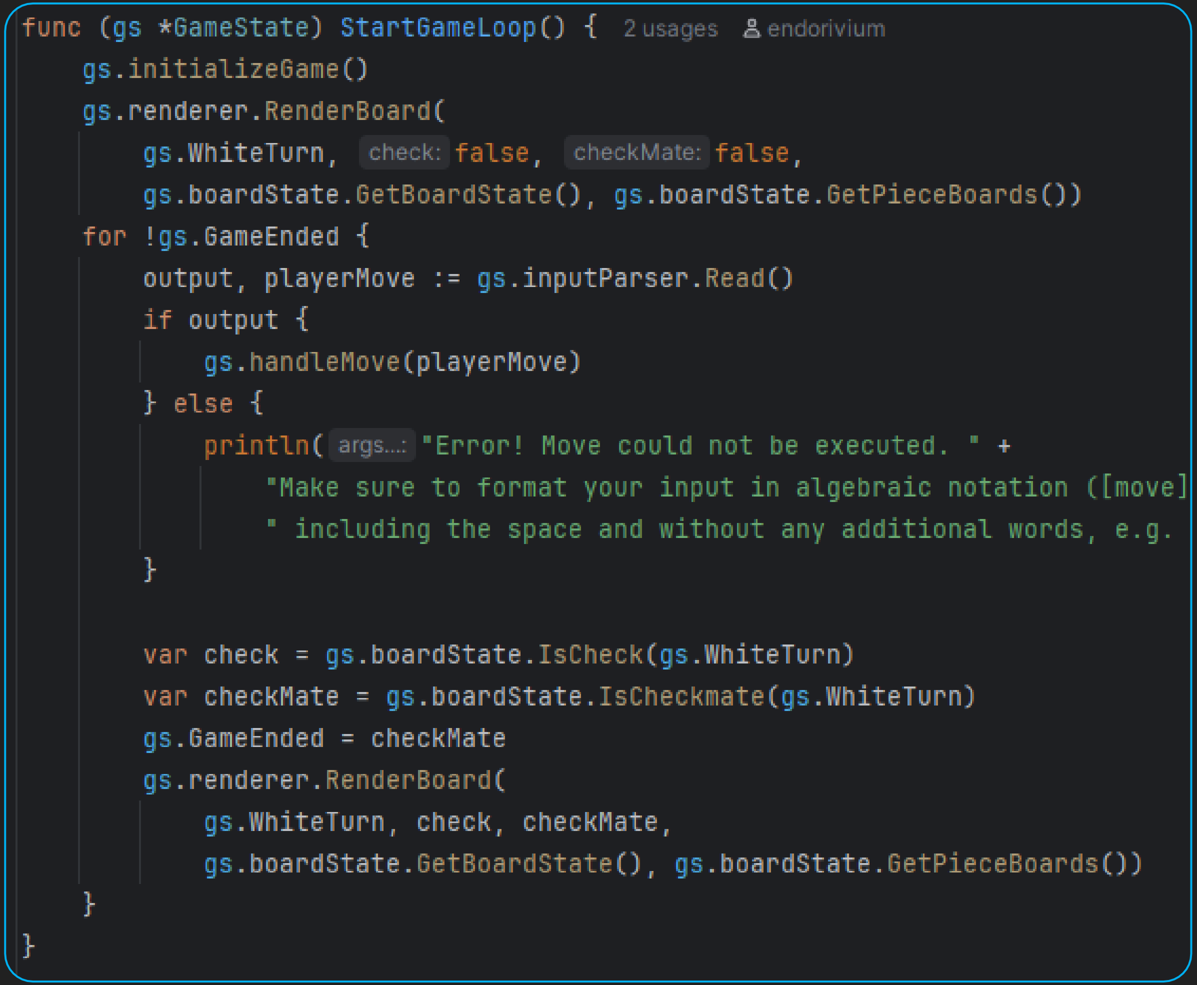
```
var bitMask uint64 = 1 << shift
```

- did not happen in Go

Go Packages

- package usage and receiver reference in Go is annoying (take a shot every time gs is references here)

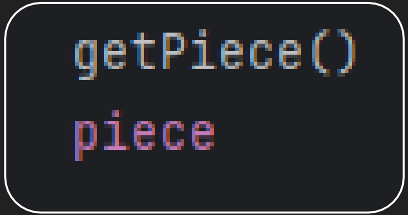
- need to reference package usage via name clashes with local variable and parameter naming, can even lead to strange errors in the code



```
func (gs *GameState) StartGameLoop() {  2 usages  endorivium
    gs.initializeGame()
    gs.renderer.RenderBoard(
        gs.WhiteTurn,  check: false,  checkMate: false,
        gs.boardState.GetBoardState(), gs.boardState.GetPieceBoards())
    for !gs.GameEnded {
        output, playerMove := gs.inputParser.Read()
        if output {
            gs.handleMove(playerMove)
        } else {
            println( args...: "Error! Move could not be executed. " +
                "Make sure to format your input in algebraic notation ([move]
                " including the space and without any additional words, e.g.
        }

        var check = gs.boardState.IsCheck(gs.WhiteTurn)
        var checkMate = gs.boardState.IsCheckmate(gs.WhiteTurn)
        gs.GameEnded = checkMate
        gs.renderer.RenderBoard(
            gs.WhiteTurn, check, checkMate,
            gs.boardState.GetBoardState(), gs.boardState.GetPieceBoards())
    }
}
```

Kotlin Function Naming

- Autogenerated getters and setters are named Get<VariableName>() and Set<VariableName>(), accidentally creating functions with the same name leads to errors
  - also: renaming those with F2 also renames all VariableName instances



```
getPiece()
piece
```

Conclusion

- Go encourages composition and interface usage for modular and easily testable code
  - but: annoying to use due to package references
    - strange architectures to circumvent limitations and achieve OOP concepts (and enums)
    - test files are forced to be in same folder as class
- Kotlin has good structures that help with code clarity (e.g. range comparison, for loop) and testing is fairly easy
  - but: primary constructor strangely separates class members if lateinit is needed and has
    - immutability makes inheritance and testing a chore

Sources

- BitBoard Calculator by gekomad
- Bitboards by ChessProgramming Wiki
- Kotlin Unit Testing guide by Kacper Wojciechowski
- Making a Chess Engine in Zig by John Murray
- Visualizing Chess Bitboards by Andrew Healey
- What is Bitmasking? by GeeksforGeeks