8.25 발표

Java와 비교하면서 공통점은 서술하지 않고 차이점 + 중요하다고 생각한 내용을 위주로 서술함.

1.4 Go도구

(1)컴파일 언어

Go는 컴파일형 프로그래밍 언어이다.

자바는 메서드를 호출할 때, 중복된 코드를 체크해둔다. 이후, JIT 컴파일러로 바이트코드를 인터프리터를 이용하여 번역된 코드를 캐싱해두고 다음에 똑같은 코드가 있다면 기계어로 또 번역하지 않고 캐싱해둔 값을 이용한다

→ 인터프리터 방식보다는 속도를 단축시킨다.

컴파일형은 소스코드가 모두 기계어로 번역이 되고 실행파일도 따로 만들어진다.

따라서 소스코드가 모두 기계어로 번역이 된 후에 에러가 발견 된다.

인터프리터 방식은 한줄씩 기계어로 번역하고 실행하기 때문에 에러의 위치를 더 빨리 발견할 수 있지만, 실행속도는 훨씬 느리다.

3. 타입

(2)Go는 정적 타입 프로그래밍 언어

→ 지금까지 보던거와는 다르게 타입이 변수명 뒤에 온다!

```
//4장 : 변수
package main

import "fmt"

func main() {
    var hello string = "Hello"
    fmt.Println(x)
}
```

```
x :="Hello World"
```

타입을 지정하지 않음

→ Go 컴파일러가 변수에 할당하는 리터럴 값을 토대로 타입을 추론한다.

따라서 런타임시에 변수의 형이 결정된다.

이 방식은 동적 타입 언어 아닌가?

Go는 정적 타입 언어를 기반으로 만들어졌지만, 동적 타입 방식의 선언도 지원하기 때문에 위 방식이 가능

그래서 타입을 지정하였을 시에는 컴파일러에 의해서 에러를 찾을 수 있고,

타입을 지정하지 않았을 때에는 자동으로 Runtime시에 자료형이 결정된다.

아주 만능이다…

4. 변수

변수는 저장공간이며 타입과 그와 연관된 이름을 가지고 있다.

```
package main

import "fmt"

func main() {
    var x string = "Hello World"
    fmt.Println(x)
}
```

var 키워드를 이용해 x(변수명) string(타입)을 지정한 다음 해당 변수에 값을 할당함.

```
x :="Hello World"
```

타입을 지정하지 않음

→ Go 컴파일러가 변수에 할당하는 리터럴 값을 토대로 타입을 추론한다.

```
var x string = "hello"
var y string = "hello"
fmt.Println(x == y)
```

4.1 변수의 이름을 짓는 법

camelCase로 변수명을 구성한다.

- 문자로 시작
- 문자,숫자,_를 포함할 수 있다.
- 기호는 포함가능한지?

4.2 유효범위

- 유효범위란?

  해당 변수를 사용하도록 허용되는 공간의 범위
- Go에서의 유효범위는?

  Go는 {블록}을 이용해서 유효범위를 결정함

  가장 가까운 중괄호 내에 존재

```
var hello string = "Hello"

func main() {
    fmt.Println(x)
}

func f() {
    fmt.Println(x)
}
```

변수가 main함수 밖에 위치함 → 따라서 해당 변수는 전역변수로 다른 함수에서도 해당변수를 사용할 수 있다.

```
func main() {
    var hello string = "Hello"
    fmt.Println(x)
}

func f() {
    fmt.Println(x)
}
```

f함수에서는 hello변수에 접근하지 못한다.

→ hello 변수의 유효범위는 main함수이기 때문

```
.\main.go:11: undefined: hello
```

마찬가지로 컴파일러가 hello라는 변수가 없다고 알려준다.

4.4 여러 개의 변수 정의

```
var(
  a=5
  b=10
  c=15
)
```

(3)GC(Garbage Collection)

자바에도 있는 GC가 마찬가지로 있다!
참조하지 않는 값의 메모리 해제를 대신 해줌

5.4 switch

해당 부분을 보면서 가장 의아했던 것은 break문이 없다는 것이다.

자바같은 경우는 break문이 없기 때문에, 조건이 일치하는 값을 찾고, 그 이후의 값도 모두 출력이 될텐데

Go 같은 경우는 break문을 따로 써주지 않아도 된다.

```
package main

import "fmt"

func main() {
  n := 2

  switch n {
  case 3:
    fmt.Println("n = ", n)
  case 2:
    fmt.Println("n = ", n)
  case 1:
    fmt.Println("n = ", n)
  }
}
```

따라서 해당 코드는 2만 출력하게 된다.

근데 만약! 자바에서 break문을 쓰지 않은 것 처럼 동작하려면 `fallthrough`를 사용해주면 된다.

```
package main

import "fmt"

func main() {
  n := 2

  switch n {
  case 3:
    fmt.Println("n = ", n)
    fallthrough
  case 2:
    fmt.Println("n = ", n)
    fallthrough
  case 1:
    fmt.Println("n = ", n)
    fallthrough
  }
}
```

case 2: 와 조건이 일치하기 때문에 n=2의 값을 출력하고

이후에 작성된 값도 출력한다.

`fallthrough` → 그 다음 case 문에 작성한 조건문을 무시하기 때문이다.

또한 JDK14부터 지원하는 조건값 여러 개 연결도 가능하다!

```
package main

import "fmt"

func main() {
  n := 2
//  n := 3

  switch n {
  case 4:
    fmt.Println("n = ", n)
  case 3,2:
    fmt.Println("n = ", "True")
  case 1:
    fmt.Println("n = ", n)
  }
}
```

n의 값이 3이든 2이든 True가 출력된다.

6.1 배열

Go에서는 for문에서 배열의 각 원소에 접근하기 위한 방법으로 `range`키워드를 제공한다

```
package main

import "fmt"

func main() {
  tesetArr := [5]int{0,1,2,3,4}
  for i := 0; i < len(arr); i++ {
    fmt.Println(i, ":", arr[i])
  }
}
```

`range 배열명` 의 연산 결과를 두 변수에 대입하면, 인덱스와 해당 인덱스의 원소의 값을 반환한다.

```
package main

import "fmt"

func main() {
  testArr := [5]int{0,1,2,3,4}

  for i, value := range arr {
    fmt.Println(i, ":", value)
  }
}
```

```
//결과값
//0 : 0
//1 : 1
//2 : 2
//3 : 3
//4 : 4
```

두 코드의 결과값은 같다.

`range 배열명` 의 연산 결과를 하나의 변수에 대입하면, 인덱스 값만 반환한다.

```
package main

import "fmt"

func main() {
  testArr := [5]int{0,1,2,3,4}

  for i := range arr {
    fmt.Println(i)
  }
}
```

해당 코드의 결과값은 다음과 같다.

```
//결과값
//0
//1
//2
//3
//4
```

인덱스를 생략하고, 해당 인덱스의 원소값만 얻고자 하면 _(언더바)를 사용하면 된다.

`_,값변수:=range 배열명`

```
package main

import "fmt"

func main() {
  testarr := [5]int{0,1,2,3,4}

  for _, value := range arr {
    fmt.Println(value)
  }
}
```

이 경우 _(언더바)는 컴파일러에게 인덱스용 변수가 필요하지 않다고 알려주는 것이다.

```
testArr := [5]int{
    0,
    1,
    2,
    3,
 //   4,
}
```

Go에서는 이렇게 하는 것이 필수이다.