The switch-case Construct

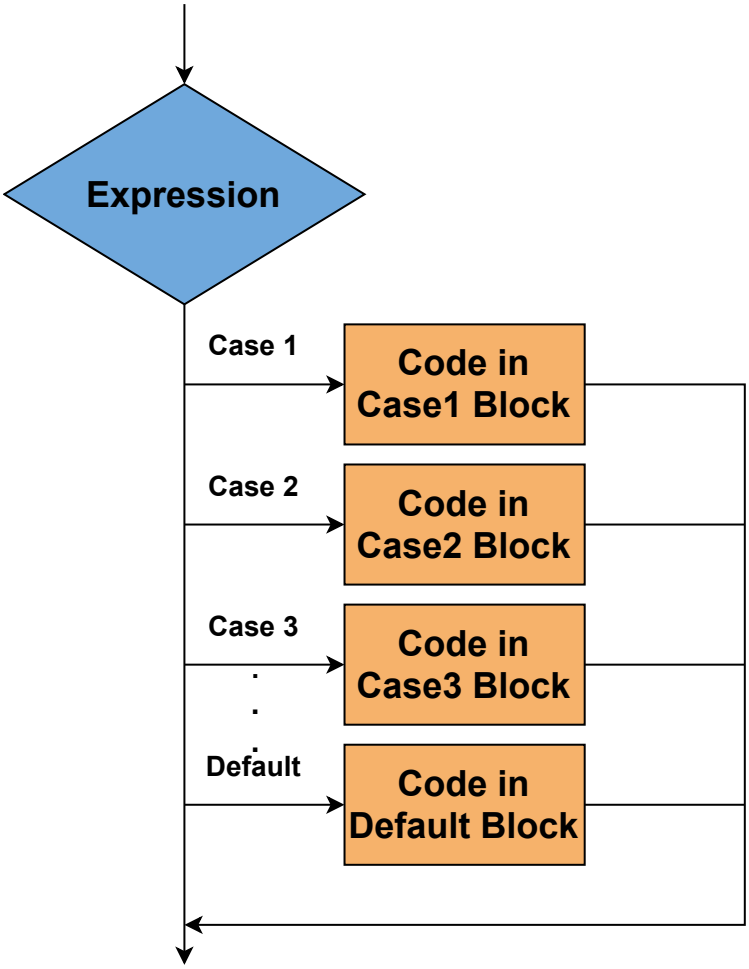
This lesson discusses the switch-case construct in detail.

WE'LL COVER THE FOLLOWING

Introduction

In the last two lessons, we studied the `if-else` construct. There is another control structure known as the `switch-case` structure.

The keyword `switch` is used instead of long `if` statements that compare a variable to different values. The `switch` statement is a multiway branch statement that provides an easy way to transfer flow of execution to different parts of code based on the value. The following figure explains the basic structure of the `switch-case` construct.



Switch Case Construct

`switch` statement with values

Compared to the C and Java languages, switch in Go is considerably more flexible. It takes the general form:

```
switch var1 {
case val1:
...
case val2:
...
default:
...
}
```

Where `var1` is a variable which can be of any type, and `val1`, `val2`, ... are possible values of `var1`. These don't need to be *constants* or *integers*, but they must have the *same* type, or expressions evaluating to that type. The opening **{** has to be on the same line as the `switch`. The ellipses `...` here means that

after the `case` statement, multiple statements can follow without being surrounded by **{ }**, but braces are allowed.

When there is *only* 1 statement: it can be placed on the same line as `case ... :`. The last statement, in any case, can also be a return with or without an expression. When the `case` statements end with a `return` statement, there also has to be a `return` statement after the **}** of the `switch`.

More than one value can be tested in a case. For this, the values are presented in a comma-separated list like:

```
case val1, val2, val3:
```

Each `case`-branch is exclusive. They are tried first to last. We should place the most probable values first, to save the time of computation. The first branch that is correct is executed, and then the `switch` statement is complete.

Compare the following two cases:

```
switch i {
case 0: //empty case body, nothing is executed when i==0
case 1:
    f() // f is not called when i==0!
}
```

And:

```
switch i {
case 0: fallthrough
case 1:
    f() // f is now also called when i==0!
}
```

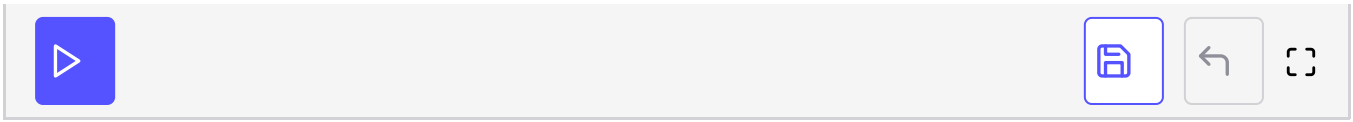
In the first case, if `i` has the value **0**, no code will be executed because `case 0` has no code body, and the `switch` terminates immediately. To obtain the same behavior in C, you need to add a *break* after `case 0`. If, on the other hand, you explicitly want to execute the code from `case 1`, when `i` has the value **0**, you need to add the keyword `fallthrough` at the end of the `case 0` branch. This is illustrated in the second case. With `fallthrough`, all the remaining case branches are executed until a branch is encountered, which does not contain a `fallthrough`.

Fallthrough can also be used in a hierarchy of cases where at each level something has to be done in addition to the code already executed in the higher cases, and a default action also has to be executed. The (optional) `default` branch is executed when no value is found to match `var1` with; it resembles the `else` clause in `if-else` statements. It can appear anywhere in the `switch` (even as the first branch), but it is best written as the last branch.

Let's write a simple program to see how the `switch` statement works.

```
package main
import "fmt"

func main() {
    var num1 int = 100
    // Adding switch on num1
    switch num1 {
        case 98, 99:        // first case: num1 = 98 or 99
            fmt.Println("It's equal to 98")
        case 100:           // second case: num1 = 100
            fmt.Println("It's equal to 100")
        default:            // optional/ default case
            fmt.Println("It's not equal to 98 or 100")
    }
}
```



Switch Case Construct with Values

In the above code:

- We declare a variable `num1` and give a value **100** to it. Now we use a `switch` statement against `num1` value, which means different cases will be written on `num1`. We made a total of **3** cases, including the `default` case.
- The first `case` is for *two* values: **98** and **99**. If this `case` is *true*, then **line 9** will be executed. If not, then control will be transferred directly to **line 10** for the second `case`.
- The second case is for the value of **100**. If this `case` is *true*, then **line 11** will be executed. If not, then control will be transferred directly to **line 12** for the third `case`.
- The third case is the `default` case. If none of the above cases are

executed, then this case will be true in any case, and **line 13** will be executed.

For the above program, the second case: `case 100` is *true*, and **It's equal to 100** will be printed on the screen.

`switch` statement with conditions

In this form of the `switch` statement, no variable is required (this is, in fact, a switch true), and the cases can test different conditions. The first true condition is executed. It looks very much like `if-else` chaining and offers a more readable syntax if there are many branches. The syntax is as follows:

```
switch {
case condition1:
...
case condition2:
...
default:
...
}
```

For example:

```
switch {
case i < 0:
    f1() // function call
case i == 0:
    f2() // function call
case i > 0:
    f3() // function call
}
```

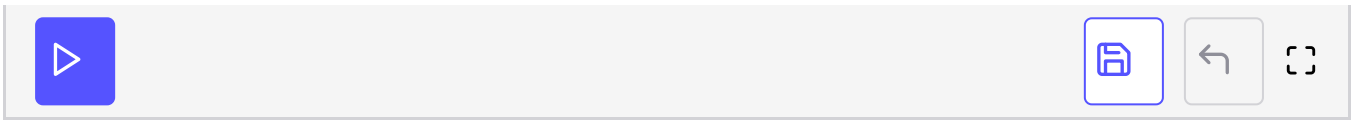
Any type that supports the equality comparison operator, such as ints, strings or pointers, can be used in these conditions. Look at the following program to see how `switch` works with conditions instead of values.

```
package main
import "fmt"

func main() {
    var num1 int = 100
    switch {
        case num1 < 0:
            fmt.Println("Number is negative")
```

```
        case num1 > 0 && num1 < 10:
            fmt.Println("Number is between 0 and 10")


        default:
            fmt.Println("Number is 10 or greater")
    }
}
```



Switch Case Construct with Conditions

In the above code, we declare a variable `num1` and give it a value **100**. We can use a `switch` statement *without* any value, which means different cases will be based on conditions. We made a total of **three** cases, including the `default` case.

The first case is for numbers less than 0: `case num1 < 0:`. If this case is *true*, then **line 8** will be executed. If not, then control will be transferred directly to **line 10** for the second case.

The second case is for the condition: `case num1 > 0 && num1 < 10:` If this case is *true*, then **line 11** will be executed. If not, then control will be transferred directly to **line 13** for the third case.

The third case is the `default` case. If none of the above cases are executed, then this case will be true in any case, and **line 14** will be executed. For the above program, the third(`default`) case is *true*, and **Number is 10 or greater** will be printed on the screen.

Initialization within the `switch` statement

A `switch` can also contain an *initialization statement*, like the `if` construct:

```
switch initialization; {
case val1:
...
case val2:
...
default:
...
}
```

After writing the `switch` keyword, we can do initialization and add a `;` at the end:

```
switch a, b := x[i], y[j]; {
case a < b: t = -1
case a == b: t = 0
case a > b: t = 1
}
```

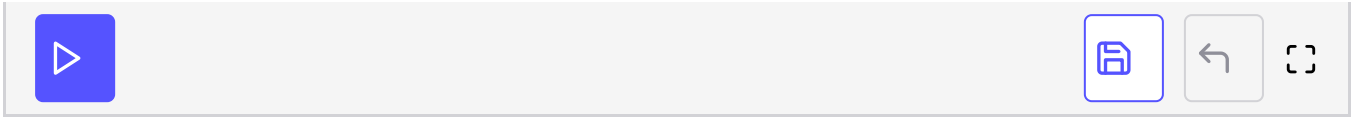
Here, `a` and `b` are retrieved in the *parallel initialization*, and the cases are *conditions*. According to the above code, `a` is equal to `x[i]` and `b` is equal to `y[j]`. Let's write a program that covers the concept of initialization within a `switch` statement.

```
package main
import "fmt"

func main() {
    // initialization within switch block
    switch num1 := 100;{
        case num1 < 0:
            fmt.Println("Number is negative")

        case num1 > 0 && num1 < 10:
            fmt.Println("Number is between 0 and 10")

        default:
            fmt.Println("Number is 10 or greater")
    }
}
```



Switch Case Construct with Conditions and Initialization

It is the same program we wrote previously, but with a little modification. Can you notice the difference? Previously, we declared `num1` separately. But now, we declare the variable in the `switch` block. Notice **line 6** where we declare and initialize `num1` in the same line as `switch`: `switch num1 := 100;`. The rest of the program is the same and will produce the same output.

That's it about how control is transferred using a switch-case construct. In the next lesson, you'll have to write a function to solve a problem.