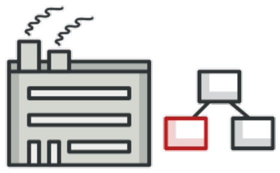

Factory Method in Go

Factory method is a creational design pattern which solves the problem of creating product objects without specifying their concrete classes.

The Factory Method defines a method, which should be used for creating objects instead of using a direct constructor call (`new` operator). Subclasses can override this method to change the class of objects that will be created.

If you can't figure out the difference between various factory patterns and concepts, then read our **Factory Comparison**.

Learn more about Factory Method →

Navigation

output

Conceptual Example

It’s impossible to implement the classic Factory Method pattern in Go due to lack of OOP features such as classes and inheritance. However, we can still implement the basic version of the pattern, the Simple Factory.

In this example, we’re going to build various types of weapons using a factory struct.

First, we create the `iGun` interface, which defines all methods a gun should have. There is a `gun` struct type that implements the iGun interface. Two concrete guns—`ak47` and `musket`—both embed gun struct and indirectly implement all `iGun` methods.

The `gunFactory` struct serves as a factory, which creates guns of the desired type based on an incoming argument. The *main.go* acts as a client. Instead of directly interacting with `ak47` or `musket`, it relies on `gunFactory` to create instances of various guns, only using string parameters to control the production.

iGun.go: Product interface

```go
package main

type IGun interface {
    setName(name string)
    setPower(power int)
    getName() string
    getPower() int
}
```

gun.go: Concrete product

```go
package main

type Gun struct {
    name  string
    power int
}

func (g *Gun) setName(name string) {
    g.name = name
}

func (g *Gun) getName() string {
    return g.name
}

func (g *Gun) setPower(power int) {
    g.power = power
}

func (g *Gun) getPower() int {
    return g.power
}
```

ak47.go: Concrete product

```
package main

type Ak47 struct {
    Gun
}

func newAk47() IGun {
    return &Ak47{
        Gun: Gun{
            name:  "AK47 gun",
            power: 4,
        },
    }
}
```

musket.go: Concrete product

```
package main
```

```go
type musket struct {
    Gun
}

func newMusket() IGun {
    return &musket{
        Gun: Gun{
            name:  "Musket gun",
            power: 1,
        },
    }
}
```

gunFactory.go: Factory

```go
package main

import "fmt"

func getGun(gunType string) (IGun, error) {
    if gunType == "ak47" {
        return newAk47(), nil
    }
    if gunType == "musket" {
        return newMusket(), nil
    }
    return nil, fmt.Errorf("Wrong gun type passed")
}
```

main.go: Client code

```go
package main

import "fmt"

func main() {
    ak47, _ := getGun("ak47")
    musket, _ := getGun("musket")

    printDetails(ak47)
    printDetails(musket)
}

func printDetails(g IGun) {
    fmt.Printf("Gun: %s", g.getName())
    fmt.Println()
    fmt.Printf("Power: %d", g.getPower())
    fmt.Println()
}
```

output.txt: Execution result

```
Gun: AK47 gun
Power: 4
Gun: Musket gun
Power: 1
```

*Based on: **Golang By Example***

RETURN

← Builder in Go

READ NEXT

Prototype in Go →

Factory Method in Other Languages

Java

php

python

Ruby

Rust

TS