Factory Method

This lesson tells what a factory is and how to make and use a factory. In the later part, we revisit some important functions.

WE'LL COVER THE FOLLOWING

- A factory of structs
- `new()` and `make()` revisited for maps and struct

A factory of structs

Go doesn't support constructors as in OO-languages, but constructor-like **factory** functions are easy to implement. Often, a factory is defined for the type for convenience. By convention, its name starts with `new` or `New`. Suppose we define a `File` *struct* type:

```
type File struct {
  fd int // file descriptor number
  name string // filename
}
```

Then, the *factory*, which returns a pointer to the struct type, would be:

```
func NewFile(fd int, name string) *File {
  if fd < 0 {
    return nil
  }
  return &File{fd, name}
}
```

Often a Go constructor can be written succinctly using initializers within the factory function. An example of calling it:

```
f := NewFile(10, "./test.txt")
```

If `File` is defined as a struct type, the expressions `new(File)` and `&File{}` are equivalent. Compare this with the clumsy initializations in most OO languages:

```
File f = new File( ...)
```

In general, we say that the factory instantiates an object of the defined type, just like in class-based OO languages. How to force using the factory method? By applying the *Visibility rule*, we can force the use of the factory method and forbid using `new`, effectively making our type *private* as it is called in OO-languages.

```
package matrix
type matrix struct {
  ...
}
function NewMatrix(params) *matrix {
  m := new(matrix)
  // m is initialized

  return m
}
```

Because of the `m` of `matrix`, we need to use the factory method in another package:

```
package main
import "matrix"
...
wrong := new(matrix.matrix) // will NOT compile (the struct matrix can't b
e used directly)
right := matrix.NewMatrix(...) // the ONLY way to instantiate a matrix
```

`new()` and `make()` revisited for maps and struct

The difference between these two built-in functions was clearly defined in Chapter 5 with an example of slices. By now, we have seen two of the three types for which `make()` can be used: *slices* and *maps*. The 3rd make type is *channels*, which we will discuss in Chapter 12.

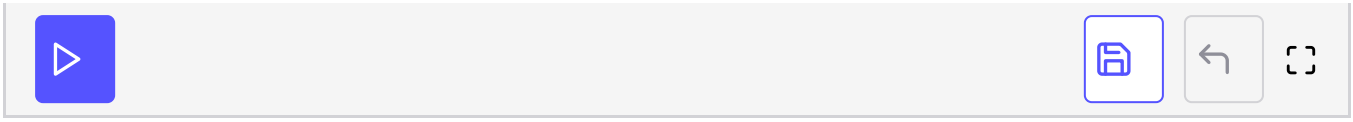
To illustrate the difference in behavior for maps and possible errors, experiment with the following program:

```
package main

type Foo map[string]string

type Bar struct {
  thingOne string
  thingTwo int
}

func main() {
  // OK:
  y := new(Bar)
  (*y).thingOne = "hello"
  (*y).thingTwo = 1
  // not OK:
  z := make(Bar) // compile error: cannot make type Bar
  z.thingOne = "hello"
  z.thingTwo = 1
  // OK:
  x := make(Foo)
  x["x"] = "goodbye"
  x["y"] = "world"
  // not OK:
  u := new(Foo)
  (*u)["x"] = "goodbye" // !! panic !!: runtime error: assignment to entry in nil map
  (*u)["y"] = "world"
}
```



The code doesn't compile successfully. To find the reason, let's study code line by line. At **line 3**, we declared a type called `Foo` for all the maps having keys and their values with type *string*.

Let's first study the use of `new` and `make` in case of structs. At **line 5**, we make a struct `Bar` with two fields `thingOne` a *string* variable and a `thingTwo` an *integer* variable. Now, look at `main`. At **line 12**, we are making a `Bar` type variable via `new()` function called `y`. As `y` is a *pointer* variable, to set the values of its fields, we have to dereference `y`. See **line 13**, where we are dereferencing `y` and using the selector to give the value of **hello** to `thingOne` as: `(*y).thingOne = "hello"`. In the next line, we are dereferencing `y` and using the selector to give value of **1** to `thingTwo` as: `(*y).thingTwo = 1`.

Now, let's do the same thing but with the `make()` function. Look at **line 16**, we are making a `Bar` type variable via `make()` function called `z`. As `z` is a *value* type variable, to set the values of its fields, we have to use the selector directly without dereferencing `z`. See **line 17**, where we are using the selector to give the value of **hello** to `thingOne` as: `z.thingOne = "hello"`. In the next line, we are giving the value of **1** to `thingTwo` as: `z.thingTwo = 1`. This piece of code will give the compiler error because you can never make a struct type variable with `make`. You can only use the `new` function for this purpose.

This is about the structs. Now, let's see the implementation on maps. At **line 20**, we make a `Foo` type map called `x` using `make` function. At **line 21** and **line 22**, we are making a key `x` with value **goodbye** and a key `y` with value **world**, respectively. At **line 24**, we make a `Foo` type map called `u` using the `new` function. As `u` is a *pointer* type variable, to set values or keys `u` will be dereferenced when used. At **line 25** and **line 26**, we are making a key `x` with value **goodbye** after dereferencing `u` and a key `y` with value **world** after dereferencing `u`, respectively. This piece of code will give a runtime error because making keys and assigning values in the case of a *pointer* type map is like an assignment to the entry in *nil* map. Using new to create a map and trying to fill it with data gives a runtime error because the new(Foo) results in a pointer to a `nil` not yet allocated, map. So be very cautious with this!

That's it about the factory of methods. In the next lesson, you'll see a new addition to the declaration of structs, i.e., *tags*.