Garbage Collection and SetFinalizer

This lesson briefly discusses how garbage collector and finalizer work in Go.

Collecting garbage

The Go developer doesn't have to code the release of memory for variables and structures, which are not used anymore in the program. A separate process in the Go runtime, the **garbage collector**, takes care of that. It starts now and then searches for variables which are not listed anymore, and frees that memory. Functionality regarding this process can be accessed via the `runtime` package.

Garbage collection can be called explicitly by invoking the function `runtime.GC()`, but this is only useful in rare cases, e.g., when memory resources are scarce. In that case, a great chunk of memory could immediately be freed at that point of the execution, and the program can take a momentary decrease in performance (because of the garbage collection process).

If you want to know the current memory status, use:

```
fmt.Printf("%d\n", runtime.MemStats.Alloc/1024)
```

This will give you the amount of allocated memory by the program in Kb. For further measurements, visit this page.

Suppose special action needs to be taken before an object obj is removed from

Suppose special action needs to be taken before an object obj is removed from memory, like writing to a log-file. This can be achieved by calling the function:

```
runtime.SetFinalizer(obj, func(obj *typeObj))
```

The `func(obj *typeObj)` is a function that takes a pointer-parameter of the type of `obj`, which performs the additional action. `func` could also be an anonymous function.

SetFinalizer does not execute when the program comes to a normal end or when an error occurs, but before the object was chosen by the garbage collection process to be removed.

That's it about how the `runtime` package provides support to collect garbage and finds the lost references of an object. There is a quiz in the next lesson for you to solve.