Go Idiomatic Conventions Explained In Color

REVISION 1

Hawthorne-Press.com

Go Idiomatic Conventions Explained in Color

Published by

Hawthorne-Press.com
916 Adele Street
Houston, Texas 77009, USA

Table of Contents

Go Idiomatic Conventions Explained in Color

Introduction

The GO Language, as with many languages, has naming conventions for the various textual elements associated with the language. This is a short primer on creating idiomatic Go programs. More extensive versions are available, but this should give a new Go Developer a head start!

Code is rendered in blue, comments in green, and rules highlighted in yellow.

Basic Constructs

- **Identifiers –** UTF-8 Characters beginning with a letter or '_' and followed by 0 or more letters and digits. Identifiers are case sensitive.
- **Blank Identifier –** An under-bar by itself is called a *blank identifier.* It can be used in a declaration or variable assignments, like any other identifier, but is value is discarded.
- **Anonymous –** There are situations where variables, types, and functions do not require a name, as it will not be used by the following code. These become anonymous values.
- **Filenames –** All lower case letters or digits, single words preferred, multiple words separated with an under-bar, '_'. Filenames may start with digits, *but may not start with an under-bar.*
- **Source Files –** Contain Go source code lines, of any length, and have a *.go* extension.
- **Keywords –** The following 25 keywords are reserved and may not be used in Go-code.

break	**default**	**func**	**interface**	**select**
case	**defer**	**go**	**map**	**struct**
chan	**else**	**goto**	**package**	**switch**
const	**fallthrough**	**if**	**range**	**type**
continue	**for**	**import**	**return**	**Var**

- **Reserved Identifiers –** The following 39 identifiers are reserved for system use. These are names of elementary types, constants, nil and built-in functions.

Types:

bool	**byte**	**complex64**	**complex128**	**error**	**float32**	**float64**
int	**int8**	**int16**	**int32**	**int64**	**rune**	**string**
uint	**unit8**	**uint16**	**uint32**	**uint64**	**uintptr**	

Constants:

true **False** **iota**

Zero Value:

nil

Built-in Functions:

append	**cap**	**close**	**complex**	**copy**	**delete**	**imag**	**len**
make	**new**	**panic**	**print**	**println**	**real**	**recover**	

- **Delimiters –** Parentheses ( ), brackets [ ], braces { }.
- **Punctuation –** Period '.' , comma ',' , semicolon ';' , ellipse '...' .
- **Programs –** Are composed of statements which are made up of *keywords, constants, variables, operators, types, and functions.* These are interspersed with non-functional constructs such as *comments and spacing.*

PROGRAMS

- **Minimal Working Program –** The often heralded 'hello world' program is an example.

Source File hello.go:

```
package main

import "fmt"

func main() {
   fmt.Println("hello, world!")
}
```

- **Package –** Every Go file belongs to only one package. A package may contain many Go files. This means the package name and filename are not always the same. *Package names are written in lower case and should be one word*
- **Import –** Allows access to the exported objects from the imported packages.
- **Func –** Introduces a function. A function may or may not have parameters, followed by Brace pair enclosing the statements comprising the function body.
- **Main() -** This is a mandatory function for standalone programs. It is always located within package *main* and is the normal start of any user program.
- **Inline Comments –** A double slash anywhere on a line indicates the start of a comment and it continues until end-of-line.

  ```
  a := 100     // Length of pool

  // This function calculates the minimum pool value.
  ```

- **Block Comments –** Starts with a slash asterisk pair and the comment terminates with an asterisk slash pair. This construct can and usually does span multiple lines.

  ```
  /* This function calculates the minimum pool value. */

  /*

     This function calculates the minimum pool value.

  */
  ```

  All comments should be complete sentences ending with a period or other punctuation. This will improve the output readability produced by Godoc. See section on Godoc Comments below for more information GO documentation.

- **Exported Objects –** Identifiers are only visible outside the package if they start with a capital letter. Identifiers starting with lower case letters are private, only accessible within their package.

  ```
  fmt.Println("hello world!")  // Notice that Println is capitalized       (Code Works)
  fmt.println("hello world!")  // Notice that println is not capitalized  (Code Fails)
  ```

- **Globals Declarations –** After the import statement, zero or more variables, types, and constants may be declared. These items will have *global scope* and can be seen by all elements of the package.

```
package main

import “fmt”

const s string = "constant"    // Global constant string

var _hello = "_HELLO"          // Global Variable of string

type Stringer interface {      // Global Stringer Interface
String() string
}

func init() {
   fmt.Println(“Initialization”)
}

func main() {
   fmt.Println(“hello, world!”)
}
```

- **Initialization Function –** The function init() will perform processing after top level objects are evaluated, but before the function main() is executed. This provides a method for special initializations. This function should proceed the main() function (*see above*), but it may be placed elsewhere if it seems appropriate.

- **Godoc Comments –** While comments do not generate code, they are used by Godoc to create documentation and as such do impose some general rules on their content.

  A comment should proceed the package statement describing the purpose of the package and any functions or data it provides. It must immediately proceed the package statement. The godoc program will provide this information when requested for this package. If a package contains multiple files, the package comment should only be present in one file.

  A comment should proceed every top level type, var, func, or const and absolutely proceed any exported object. The name of these items should be the first word in the comment. See func main example below.

```
/*
Package hello implements a simple “hello world!”
message and exits.
*/
package main

import “fmt”

// main – This function prints “hello world!” and quits.
func main() {

   fmt.Println(“hello, world!”)

}
```

Naming Conventions

Names of Go objects should be short and concise. In general If necessary to use multiple words to name an object, use MixedCaps or mixedCaps depending whether the object is exported or not.

- **Filenames –** Filenames should single words made up of lowercase letters, numbers and under-bars. Unlike identifiers filenames may start with digits, but may not start with an under-bar. If filenames are made up of multiple words, they should be separated by under-bars. For example: collection_file.
- **Method/Function Names -** A name of method or function that returns a result should be a *noun*. If it changes the data of an object, use s*etNoun*. For example: file and setFile.
- **Global Objects –** Such as constants, variables, and types should be single words that unambiguously identify its purpose. Using multiple words with mixed caps if necessary. Since these objects will generally be used throughout the package their purpose and function should be immediately recognizable.
- **Short Lived Variables –** Many variables have only function or sub-function scope and are short lived, such as indexes. These variables should named with single letter, or only a couple of letters. For example: I, j, or kb.
- **Error Messages –** Error strings should be all lower case unless starting with a proper noun or acronyms (ie URL…) and unlike comments no period at the end. Since error messages are often made up of multiple components this will make them more readable.
- **Acronyms -** Words in these categories (“URL” or “NATO”) should always be presented with constant case. Either “URL” or “url”, but never “Url”.
- **Package Names –** Exported Names *should not* contain an indication of the package to which they belong, as they will be qualified with the package name when evoked. For example, if your package is myData, a possible command would be Open not myDataOpen.
- **Receiver Names –** A method’s receiver should be one or two letters as it will be used often in most cases. If the receiver’s identity is “Client” then the generally acceptable names would be either “c” or “cl”.

Suggestions on Coding

- **Value, OK Pattern –** In writing your own functions, if the functions produces errors the programmer should return an error code unless the failure is catastrophic enough for a panic.

```
If value, ok := func(param); ok {
Process()
}

If value, err := func(param); err != nil {
    ...
    return err
}
```

- **Declaring Empty Slices –** The following is the preferred form:

```
var  t  []string
```

- **Slice Preference –** Slices are almost always preferable to fixed arrays.
- **Indent Error Flow –** Keep normal code as unindented as possible. Try to handle errors first and indent them so the stand out in the code as exceptions.

```
If err  != nil {
// error handling
return    // or continue, etc
}
// normal processing
```

- **Empty String Check –** Check for empty string, not for zero string len. This is inefficient, it calls a function instead of using a simple comparison.

```
If s == “” {
    …
}
```

As you find other coding best practices, and there are many more than these, document them for your own reference. It will help you maintain consistent coding practices.

Revision History

Date	By	Section	Changes
05/27/2018	C.E. Thornton	All	Initial Document