[Print](javascript:window.print())

**Course Transcript**

Programming in Go

**Go Language Basics**

[1. Course Introduction](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t2)

[2. Introducing Go](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t6)

[3. Hello World! in Go](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t10)

[4. Go Packages and Imports](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t14)

[5. Variables and Constants in Go](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t18)

[6. Go Pointers](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t22)

[7. Types in Go](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t26)

[8. Functions in Go](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t30)

**Flow Control Statements in Go**

[1. The for Statement in Go](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t35)

[2. The if Statement in Go](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t39)

[3. The switch Statement in Go](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t43)

[4. The defer Statement in Go](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t47)

**Practice: Flow Control Statements in Go**

[1. Exercise: Using Flow Control Statements in Go](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t52)

**Additional Types in Go**

[1. Arrays in Go](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t57)

[2. Slices and the Slice Header](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t61)

[3. Slices and Functions](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t65)

[4. Manipulating Slices](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t69)

[5. Structs in Go](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t73)

[6. Maps in Go](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t77)

[7. Advanced Functions](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t81)

**Methods, Interfaces, and Concurrency in Go**

[1. Methods in Go](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t86)

[2. Go Interfaces](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t90)

[3. Concurrency in Go](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t94)

[4. Creating a Concurrent Program in Go](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t98)

[5. Go Channels](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t102)

**Practice: Working with Slices in Go**

[1. Exercise: Using Slices in a Go Program](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#t107)

Course Introduction

Learning Objective

*After completing this topic, you should be able to*

* *start the course*

**1.**

Learning the fundamentals of the Go programming language is vital to your success in utilizing language to its fullest. Hi, I'm Joe Khoury, and in this course, I'll discuss the basic language syntax, working with variables, types, structs, slices, and maps. I'll also explain flow control statements, pointers, Goroutines, and channels.

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Introducing Go

Learning Objective

*After completing this topic, you should be able to*

* *describe the background and basic features of the Go programming language*

**1.**

In this video, we'll introduce the Go programming language. The Go programming language is a relatively new, general-purpose programming language. That is, it is designed for creating software over a wide variety of applications. Other general-purpose programming languages include – for example – C, C++, C#, and Java. Go is also open source. So the source code and all the documentation is freely available to anyone. What this implies is that it is somewhat of a collaborative effort. So a community of developers can freely modify the source code and share those changes.

Now, although the language resembles the C programming language, it was designed with differences that make it easier to program with, easier to read, and with a very concise and efficient syntax. Some key features of Go include a very well-documented library – fully available online. And because of its concurrency features, it's excellent for multicore applications and for network services. Go relies heavily on functions and packages. And because of this, it is ideally suited for modular software design of which the benefits are well known. One of the keys that its designers identified early on, based on their experience with C++, was the lengthy build times associated with that language. And so, when they designed Go, they designed it to compile very quickly.

Another feature of Go is that it supports run-time reflection. And this refers to the capability of a program to inspect or investigate its own structure. This is implemented through types. Benefits of run-time reflection include tremendous flexibility, for example, the ability to work with software code not yet written. That means that during compile time, your software doesn't necessarily have to be aware of everything it will interface with. It can figure out some of this at runtime. Go was developed at Google in 2007. While it was announced in 2009, development continued and a stable version of Go was released in 2015. Of course, development continues on this relatively young, general-purpose programming language. So, in this video, we introduced the Go programming language.

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Hello World! in Go

Learning Objective

*After completing this topic, you should be able to*

* *recognize a simple Go program and how it is created and executed*

**1.**

In this video, we'll write a simple Hello World program in Go. Then we'll compile and run it at the command prompt. Before we get started, there are a couple of things. One – if you need to download and install Go, you just go to golang.org and you'll see a download link where you will be able to download the binary distribution for your platform whether it's Linux, Mac OS X, Windows, or one of many others. And one other thing – if you're using Windows as your development platform, you'll have to create an environment variable. Now you go to System Properties and click on the Advanced tab. And then you click on Environment Variables. In the bottom pane, you'll see two variables that are associated with the Go program – GOROOT, which I've highlighted here is created by Go when it's installed; GOPATH, you'll create yourself. And you give it a name – GOPATH. And, for the value, you specify your absolute folder path to your workspace.

*[The "Sublime Text 2" text editor window is open. Running along the top of the window is a menu bar with ten menus, namely File, Edit, Selection, Find, View, Goto, Tools, Project, Preferences, and Help.]*

Okay, so let's get started. We'll create a New File. And the first line that you're going to type in any executable program written in Go is package main. And that signifies that this is a special package. It's an executable program. Okay. Now, at this point, I'm going to save the file. I'm going to save it as hello.go. And yes, you need the extension. And I'll Save it. Now the reason why I saved it at this point is because the text editor I'm using recognizes the Go syntax and the programming language among many others actually. And there are many, many excellent text editors out there like Notepad++, Rackets, but I'm trying this one out. Anyway let's move on. Okay. So, at this point, if we were not going to import any packages – that is if we were not referencing any functions that are associated or that are contained within any other package – then we wouldn't have to import anything, but we are going to import a package. And we're going to import a format package. And the format package or fmt for short helps us to...well, it contains a number of functions for formatting output.

*[To create a new file, the presenter clicks the File menu. A drop-down list appears containing several options like New File, Open File, and so on. He selects the New File option. To save the file, he again clicks the File menu and chooses the Save As option from the list. To import the format package, he types the following line of code: import "fmt"]*

Okay, so now we've imported format. Now we can start to define our function. So similar to package main, any executable program that we write in Go will have this function – func main. And this is the entry point for that program. Then we specify a curly brace, press Enter, and make this note that your curly brace has to be on the same line as your function declaration. If it's not, the compiler will take a fit. Okay. So I pressed Enter. I'm in a new line. Now I can start typing the body of my function, which is pretty simple. It's a one liner. And here is where we're referencing the Println function. And what we do is we send that in argument – a text argument. And that text argument in this case is Hello World, okay. So we're ready to roll. But, before we do that, I actually want to show you one other thing. Okay, this is not the same as this. In fact, this will throw a compiler error and I'll save it by using the Ctrl+S hot key combination. And now we would go and try to compile and run our program and see what happens.

*[To print the "Hello World," he inputs the following lines of code: func main () { fmt.Println("Hello World") } The presenter points out that in the above print statement, the "P" of "Println" should be in capital, otherwise the compiler will generate an error. To show the error generated, he saves the program keeping the wrong syntax.]*

So, to compile and run it, we type go run followed by the name of the program, okay. And we need the extension. If we were just to leave it like this, well, Go wouldn't find it. So we need the .go extension. And well, we've got a compiler error. And of course, as we thought – undefined: fmt.println. Okay, so let's go back and let's fix that. One of the things about Go is that it's a strongly typed language, which means it's picky about some things. And one of the things it's picky about is, well, the fact that it's case sensitive. So "Println" is not the same as "println". Okay, so now let's save that and we'll compile and run it. First, we'll get rid of the garbage and then we'll go run hello.go – Hello World. Okay, so just to recap couple of things – any executable program that you write in Go is going to have as its first line package main, any executable program is also going to have a function called main, and you've to explicitly import any packages whose functions or methods that you're going to reference. Okay so, in this video, we wrote a simple Hello World program and then we compiled and ran it at the command prompt.

*[To compile and run the program, he opens the Command Prompt and inputs the following command: go run hello The following output is generated: go run: no go files listed He again inputs the following command: go run hello.go The compiler generates the error "undefined: fmt.println." To correct the program, the presenter switches back to the text editor window. He changes the casing of "P" in "Println" and switches back to the Command Prompt. To clear the screen, he executes the "cls" command. He again executes the go run hello.go command and "Hello World" is generated as output.]*

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Go Packages and Imports

Learning Objective

*After completing this topic, you should be able to*

* *work with packages and imports, and recognize the more commonly used packages from the Go standard library*

**1.**

In this video, we'll consider Go packages and imports. A Go program is comprised of packages. And each Go program begins execution in a package named main. In fact, the package main statement is the first line you'll see in any executable program written in Go. Go supports modular design and is heavily dependent on functions and packages. As for packages, there are a lot of packages integrated with the Go language that provide a lot of functionality over and above the language's native functionality. And we'll get into that in more details shortly. But you should understand this – packages are not automatically included with every compiled program. In fact, you've to import them in order to use them. You do this with a factored or nonfactored import statement, which comes after your main package declaration. It's also considered good programming style to import packages using factored imports.

*[Heading: Import Statement. The factored import statement is as follows: import ( "binary" "build" ) The nonfactored import statement is as follows: import "binary" import "build"]*

Some key packages in Go include strings. Strings contains a lot of functions for manipulating and managing, guess what, string values. So, for example, the replaced function can be use to replace the occurrence of a specified character or characters in a string with some other characters. The io package is also another key package and contains some functions, but one of its most important contributions is the interfaces that it defines as these are used in other packages, the two foremost interfaces being reader and writer. The bytes package contains functions for manipulating byte data and provides similar functionality on byte data that strings does for string data. For example, in the byte package, we'll replace specified bytes of data in what is called a slice with new byte values.

The os package supplies an interface that is platform independent for manipulating and managing files and folders. For example, the open function is used to open files on an operating system. And also, similarly for file and folder manipulation, we can use the path/filepath packages. The errors package contains functions for manipulating errors. Although, you should note that Go already has a built-in error type for handling errors. So the errors package can allow you to create a custom error in addition to the built-in ones. The container/list package implements what is known as a doubly linked list, which allows you to – for example – iterate through a list of values and sort provides functions for sorting data. For managing hashes and cryptography, we've the hash package. And the encoding/gob package allows you to encode values in Go so that other programs or even parts of the same program can read them. This is utterly important because what it does, for example, is allow you to create a network server that can handle client requests. For managing remote procedure calls, there is net/rpc and net/jsonrpc packages. These packages provide a way to make methods available so that they could be called over a network. So, in this video, we considered Go packages and imports.

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Variables and Constants in Go

Learning Objective

*After completing this topic, you should be able to*

* *declare and initialize variables and constants in Go*

**1.**

In this video, we'll have a look at variables and constants in Go. Variables provide us with a way to store data of a specific type and associate a name with them. In Go, variables can be declared at the package or function level. And we'll have more on that later. But for now, know this – they must start with a letter, they can contain numbers, they can also contain the underscore character. And you can use the var statement to declare a variable anywhere inside or outside functions. But, inside of a function, you can use the short declaration version, which is :=, followed by the value of the variable. You can declare one or more variables in the same declaration statement. You use the var statement followed by the variables that you're declaring. And then you specify a type. So for example, here on the left side, we're declaring a variable named "a". And it's of Boolean type. And, on the right, we're declaring three variables – "a", "b", "c". And they are all Boolean type as well. Naming variables is entirely up to you. The compiler won't care as long as you don't use one of Go's keywords that is. But there are some fundamentally sound practices for naming variables. First – you should use descriptive names for your variables. And second – you should use a consistent naming convention. Let me illustrate with this example. In the top example, we have a variable named "y", which is initialized with a string value of "Cat". Well, the variable "y" doesn't really describe what the value is in any way or how it's going to be used.

*[To declare a Boolean type variable "a," the syntax is as follows: var a bool]*

Now below that, in the second example, we have a variable named "thing", which is a little bit better but still not perfect. Now, in the bottom example, we have a variable named "myPetType" and that is far more descriptive. It actually does describe exactly what we're talking about. We're talking about a specific pet type. So one other thing to notice, also, is the fact that it's written in what is called camel case – the variable name that is. And what that means is that if a variable has multiple words in it, then the first word is lowercase, the subsequent words have the first letter in the word capitalized. So here we have three words – my, Pet, Type – "my" is all lowercase, "Pet" has the first character capitalized, and the same with "Type". And, as long as you maintain a consistent naming convention like this, then your program will be easier to read and it will be easier to understand exactly what you're doing.

Initializing a variable gives it an initial value as well as a name and a type. Declaring a variable without an initial value means that's initialized with a zero value. So, in these three examples, we have a variable named "a" which is a string. And it's initialized with the value of "hello". In the second example, we're declaring and initializing two variables – "i" and "z". They are both integers. And "i" has a value of 2. And "z" has a value of 3. And, in the third example, we're declaring and initializing a variable named "h". And its value is the Boolean value of true. Now, as noted, you can also use the := shorthand for declaring and initializing variables. For example, here p := "hi" initializes the value of the "p" variable as the string "hi". And this is called int inferred type. And the inference here is that it's a string type. Now variables that don't change are called constants and are declared with the "const" keyword. For example, here we see in func main(), we're initializing a variable named "y" as a constant. And it's a string type constant. And it's initialized with the value of "Hello World". And that won't change, that's a constant.

Now, when you declare variables without an initial value, they have zero values. And for example, a numeric zero value is zero whether it's an integer or a float type or what have you. Any numeric type initializes to zero if it doesn't have an initial value associated with it. For a Boolean value, a zero value is false. So what that means is that a Boolean type – if not initialized with a true or a false value – it is initialized with false. That's the Boolean type zero value. And for strings, the zero value is considered an empty string that means there is nothing there – no space, nothing. Now variable scope means where in your code data variable can be used. For example, here in the example on the left side, you can see that we have declared a variable named "z". It's a string variable and its value is "Hello World". And it's available only to func main(). On the right side, we've declared the variable at the package level outside the func main(). So that variable "z" will be available to the func main() and to any other function in our entire package. So, in this video, we looked at variables and constants in Go.

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Go Pointers

Learning Objective

*After completing this topic, you should be able to*

* *work with pointers in Go programs, including passing by value and passing by reference*

**1.**

In this video, we'll have a look at pointers in Go. When we make a call to a function that expects an argument to be passed to it – for example, an integer variable – a copy of that integer variable is passed to the function. And so normally, we can't change the original variable's value. However, pointers refer to an area in memory that a value is stored in. So this actually makes it possible to modify the variable's original value. Let's take a look at an example. So, in the example on the left, func main() is the first function that gets executed and is so in any program. And what we're doing in func main() is initializing an integer variable "y" with a value of 10. Then we call the "fun" function with the value that's represented by "y". Now remember, it's not actually sending the function the same variable, it's actually a copy of the variable "y". So up here, when function executes, we see that it's expecting integer argument. And it's named "y" up here, but really we could have named it anything. It doesn't matter in the function declaration. It has no relation with the original value of "y" other than the fact that it's a copy of the original variable "y".

*[The presenter explains the following lines of code to print the value of a variable without using pointers: func fun(y int) { y=0 } func main() { y := 10 fun (y) fmt.Println(y) // y is 10 }]*

So here, when it's setting y = 0, it's setting y = 0 in the local function only. So, in the next statement, when it sends control back to the func main(), func main() will then print the value of "y". And it's going to print 10 because we did not change the original variable's value. Now, in the example on the right – a couple of things. First there are two operands here – the & and the \*. And let me explain those. So basically func main() executes. Again, we set an integer value to the variable "y" and the value is 10. Then we call the func fun(), but this time using the & operator. And what this does is it basically causes us to send the memory location that the value of "y" is stored in to the func fun(). So, when the func fun() executes, we see that it's expecting a value to be passed to it. And that value, which is referred to here by the \*int, is a pointer to an integer value.

*[The presenter explains the following lines of code to print the value of a variable using pointers: func fun(yPtr \*int) { \*yPtr = 0 } func main() { y := 10 fun (&y) fmt.Println(y) // y is 0 }]*

So here dereferencing basically means that we're providing access to the underlying value that the pointer points to. So here \*yPtr = 0 means store the integer 0 in the memory location pointed to by "y" pointer. So now – at this point – when the func fun() finishes executing and returns control back to the func main() and prints the value of "y", it's going to print 0 because we've changed the value in that memory location. Okay, so you can also use the "new" function to obtain a pointer. And this is how the "new" function works. You pass new – a type – as an argument. It then allocates enough memory for a value of the type passed to it and returns a pointer to that memory location. So, in this example, let's have a closer look. So let's figure out what is going on here in this example. So we have the func main(). And we're setting a variable called yPtr. And what are we setting it with? We're setting it with a memory location because that's what the "new" function returns.

*[The presenter explains the following lines of code to demonstrate the "new" function: func num(yPtr \*int) { \*yPtr = 5 } func main() { yPtr := new(int) num (yPtr) fmt.Println(\*yPtr) // y is 5 }]*

So we send it the type integer as an argument. It allocates the memory space required to store a variable of type integer and returns the memory address of that integer value. So then – in the next line – when we call the "num" function, we're sending it a memory address. So, up in the function "num" here, we see in the function declaration we have got a variable called yPtr and it's a type of pointer. And it points to an integer value. So, in the next line – again using \* to do reference – we're basically changing the value that's stored in that memory location. And so then, when we return control back to our calling function "main", it prints out the value of the dereferenced memory location. And that value is 5. Okay so, in this video, we looked at pointers in Go.

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Types in Go

Learning Objective

*After completing this topic, you should be able to*

* *recognize the basic types in the Go language and describe type inference*

**1.**

In this video, we'll have a look at types in Go. First let's have a look at integer types. Obviously, the integer type that you declared depends on how the variable is going to be used. But, as you can see here, there is no lack of the different types of integers that you can declare. So you can declare unsigned integers. Those are uint8 to 64-bit values. And you can also declare int8 to 64-bit values with the sign. These types are architecture independent. But note that if you declare a variable with just the int keyword and no size specification, it is platform dependent. So it could be 32 or 64 bit depending on the architecture of the system it's compiled on. Now, for floating point arithmetic – that's big math – there are float32 and 64-bit values. And, for crazy complex math, there are complex64 and complex128 values.

*[The unsigned 8-bit integers are declared using "uint8" and its value ranges from 0 to 255. The unsigned 16-bit integers are declared using "uint16" and its value ranges from 0 to 65535. The unsigned 32-bit integers are declared using "uint32" and its value ranges from 0 to 4294967295. The unsigned 64-bit integers are declared using "uint64" and its value ranges from 0 to 18446744073709551615. The signed 8-bit integers are declared using "int8" and its value ranges from -127 to 128. The signed 16-bit integers are declared using "int16" and its value ranges from -32768 to 32767. The signed 32-bit integers are declared using "int32" and its value ranges from -2147483648 to 2147483647. The signed 64-bit integers are declared using "int64" and its value ranges from -9223372036854775808 to 9223372036854775807.]*

Then we've the good old standard Boolean variable type, which is declared using the bool keyword and this can be either true or false. Well, string declares – you guessed it – a string variable. Incidentally, string variables may be initialized as empty strings or set as empty strings as well. And you can determine the length of the string in bytes by using the built-in len function. Type inference involves setting the variable type implicitly. You use the var statement without a type, but set the value directly. Or, inside functions, you can use the := operand. Type is inferred by what appears on the right side of the statement. So here the variable h is set as a string with a value of "hi". So, when this program runs, the output is h is type string. And just to note here, %T here is known as a format verb and it comes from the fmt package or format package. And it represents the type of value specified in Printf – here as the second argument h. So, in this video, we looked at types in Go.

*[The presenter explains the following lines of code for type inference: package main import "fmt" func main() { h := "hi" fmt.Printf("h is type %T\n", h) } The output of the above program is "h is type string."]*

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Functions in Go

Learning Objective

*After completing this topic, you should be able to*

* *work with functions, including passing arguments and returning values*

**1.**

In this video, we'll have a look at functions in Go. A function – or in some circles a subroutine or procedure – is a piece of code that may be referred to by other functions to perform some action. So they may accept parameters, but they don't have to. And they may return values, but they don't have to do that either. The components of a function declaration include the func keyword. And that tells the compiler that it's a function that we're declaring. Then we give it a name – hopefully, a descriptive name and not one of Go's built-in keywords. Then we've got a parameter definition here. We've defined two parameters – "x" and "y". They are both integers. And by the way, because they are both integers, we could have just said here "x, y" and one int keyword, but we didn't. We have it this way. Then we have return type. It's set as int or integer. And together the parameter definition and the return type comprise what is called the signature of the function. Then we have the heavy lifting of the function. And this is where it does what it does. And, when you define the body of the function, you specify it within a set of curly braces. And the opening curly brace has to be on the same line as the function declaration. Otherwise, the compiler throws a fit. And then you've a closing curly brace after the last statement in the function's body.

*[The presenter explains the following line of code for function components: func add(x int, y int) int { return x + y}]*

Let's look at an example. Okay, so here we're calling a function called subt. And we're sending it two parameters. So let's take a look at the function declaration. Here it is. We're using the func keyword. Here's the name – subt. Here is where we're defining what parameters that it's going to accept. And we're accepting two integer values – "a" and "b". And they are separated by a comma of course. And then we're returning one value. And it's an integer value and opening curly brace on the same line. And then we're returning the value of "a - b" and then the closing brace. Alright so, when this function runs in this program, it's going to send two parameters – 18 and 10. And it's going to return the value of 18 - 10, which by my math is 8.

*[The presenter explains the following lines of codes for a simple subtraction function: package main import "fmt" func subt(a, b int) int { return a - b } func main() { fmt.Println(subt(18, 10)) }]*

Now something else about functions is that you can have multiple results returned. So here, in the body of our func main(), we're calling a function called "mult". And, on the left side of the statement, when we're calling it, notice we have defined two values "x" and "y" separated by a comma. And we're assigning "x" and "y" the values returned from the "mult" function. So let's have a look at our "mult" function declaration. Again, here we have the func keyword, we have the name of the function, we have the two parameters – "a" and "b". And they are both strings to match what we're expecting. And here is the difference. Look at that. We've got two strings that we're returning. They are inside parenthesis and they are separated by a comma. And there is the opening curly brace on the same line again. And then we return those two values – "a" and "b". So then, when we return to our main function, we're printing out "x, y" which actually in Go language means with two strings – x, a space, and y. So it would print "hello", a space, and "world".

*[To demonstrate a function with multiple results, he explains the following lines of code: package main import "fmt" func mult(a, b string) (string, string) { return a, b } func main() { x, y := mult("hello", "world") fmt.Println(x, y) }]*

Now I know really if this function didn't do anything, it returned the strings that it was passed. But you get the picture, right? The main thing here is to understand that you can pass multiple variables. But more importantly, you can also accept multiple return values. And this is how you do it. Now what stacking functions means is that each time we call a function, we push it onto what is called the call stack. Then, when we return from a function, we pop the function that we just returned from off the call stack. Let's look a little bit closer at how this works. Okay so, inside our main function, we call F1(). So f1 gets pushed onto the call stack. So our call stack has f1 and main on it. Then, from an f1, we call F2(). So f2 gets pushed onto the stack. So now our stack has f2, f1, and main. When f2 is done and returns to f1, it's popped off the stack. So we have f1 and main. And then, when f1 is finished, it returns to the main function and it's also popped. And we're left with the main function on our call stack. So, in this video, we looked at functions in Go.

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

The for Statement in Go

Learning Objective

*After completing this topic, you should be able to*

* *recognize and work with the for statement in Go programs*

**1.**

In this video, we'll have a look at the for statement in Go. The for statement is a type of flow control statement. And, using a for loop, we can repeat a list of statements, which is also known as a block of statements. To implement a for, we use a conditional statement and evaluate that condition each time through the list of statements. And, as long as our condition evaluates to true, we continue through the loop. So let's have a look at an example. Okay, so take note of the parts of our code inside the orange frame. First of all, we have z := 1 that initializes a counter variable for our for loop. So we started with z equals 1. And then, for the for statement, we typed for z <= 5. And what that means is continue to repeat the series of statements inside the curly braces for this for statement as long as "z" is less than or equal to 5. So, inside the curly brace, you see the fmt.Println (z). And then we increment the value of "z". So the first time to the loop "z" is 1. Then, at the end of the for statement, z = z + 1. So the value of "z" is 2.

*[The presenter explains the following lines of code to demonstrate the "for" statement: package main import "fmt" func main () { z := 1 for z <= 5 { fmt.Println(z) z=z+1 } }]*

Let's go through this step by step. Okay, so we have already gone through the first iteration through the loop. And we're at this point where "z" equals 2. Okay, so now the next time through the loop with z equals 2. Is "z" less than or equal to 5? Well, yes. So we'll print it. And it's a value incremented at the end of our for statement is 3. So it's incremented to 3. Then the third time through the for statement – the value of "z" is 3. Is it less than 5? Yes. We print it. Then we increment again and the value is 4. And then the next time through the loop, the value is 4. It's less than 5, of course. We print it, then we increment, and its value is 5 now. Then the next time through – the value is 5. Is it less than or equal to 5? Well, 5 in this universe is less than or equal to 5. So it goes ahead, prints it, then increments it to 6. And is 6 less than or equal to 5? Well, no. Does it print? No. So it doesn't increment. It exits our for loop. Really, this is kind of an ugly way to implement for loop. So let's look at an alternate way. Okay, this is a much cleaner way to implement a for loop. It's easier to read. It's compact. And it does exactly the same thing.

*[To demonstrate the alternative for loop, the presenter explains the following lines of code: func main() { for z := 1; z <= 5; z++ { fmt.Println(z) }]*

Okay, so the difference here is that again it's all on one line. But here is what we're doing. This first part, z := 1, is the initialization of our counting variable. Then we separate with a semicolon. And then we have our conditional statement z <= 5, another semicolon, and then our increment. Or if we were going the other way – a decrement statement. And, in this case, we're incrementing "z" by 1. And that's what z++ means each time through the loop. So, after it's done printing the value of "z", it increments and goes back and tests the condition again. And then it goes through "z", increments, and again repeats that cycle until the condition is no longer met – exactly the same as our previous example, but a lot cleaner. So, in this video, we looked at the for statement in Go.

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

The if Statement in Go

Learning Objective

*After completing this topic, you should be able to*

* *recognize and work with the if statement in Go programs*

**1.**

In this video, we'll have a look at the if statement in Go. An if statement is a flow control structure that allows us to determine how to branch the flow of our program based on some condition. So, when our program encounters an if statement, it checks to see if it's true or false. And, if it's true, it executes some block of code within the if statement. And, if it's not, it exits the if statement. Also note that if statements can be nested within other if statements and...actually, within any other control structure as well. Now closely related to the if statement is an if...else statement. So let's consider how these two statements operate in comparison to each other. First, we'll have a look at the if statement. Okay, so we have our variable values. Our program encounters the if condition and checks it in sequence. If the if condition evaluates to true, then it executes the block of code. And, if it evaluates to false, then it's done. Finito. It moves on to the next statement after the if statement.

Now let's consider the if...else statement structure. Okay, so with the if...else statement structure, we have our variable values. Our program encounters the if...else condition and checks it in sequence. If the if condition evaluates to true, then it executes this block of code. If it doesn't evaluate to true – if it evaluates to false – then it executes this block of code. In other words, it doesn't exit the if...else structure immediately. It executes some other block of code then it exits. Let's look at an example to see how this works. Okay, so here is a crafty little if...else statement example. We have the if...else nested inside a for statement. So, when func main() executes, it encounters the for statement. And it will cycle through this statement a number of times until this condition is no longer met. Okay, so just take my word for it. The first time through the loop, the condition is met that is...so "y" starts out as 1. Is it less than or equal to 5? Yes. So it goes ahead and moves on to the next statement within the for structure. So then it encounters this line of code.

*[The presenter explains the following lines of code for the if-else statement: func main() { for y := 1; y <=5; y++ { if y % 2 == 0 { fmt.Println (y, "even number") } else { fmt.Println (y, "odd number") } } }]*

Okay, if y % 2...what does that mean? Well, actually what this means is divide "y" by 2 and determine if there is any remainder. This, by the way, is a remainder operand. So, if the remainder is zero, then it means it's exactly divisible by 2, which means it's an even number. If the remainder is not zero, then it means that it's not exactly divisible by 2. And, in fact, it means that it's an odd number. Okay, so the first time through this loop, "y" is 1. Is 1 exactly divisible by 2 without any remainder? Well no. It's not. So it's an odd number. So it will not perform the block of code in the if condition. Instead, it goes down and performs the block of code that's associated with the else condition. So it will print the value of "y" which is 1 and the words "odd number". Now the next time...well, actually, at this point after printing that, it exits the if. And it encounters the end curly braces for the for statement which means go to the top of the for statement and evaluate our condition once again after "y" has been incremented. So "y" is incremented by 1. So, in this sequence, it's 2. So then our program encounters the if statement. Is "y" exactly divisible by 2? Is 2 exactly divisible by 2? Well, of course it is. The remainder is zero. So, in this instance, it will perform the block of code associated with our if condition. And so it will print the number 2 and the words "even number". And then the for statement just repeats until our conditional statement is no longer true. Alright. So, in this video, we looked at the if statement in Go.

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

The switch Statement in Go

Learning Objective

*After completing this topic, you should be able to*

* *recognize and work with the switch statement in Go programs*

**1.**

In this video, we'll have a look at the switch statement in Go. The switch statement is a versatile flow control structure that allows us to determine how to branch the flow of the program in a number of ways. And this is referred to as a multiway execution. So really, in this manner, we can express conditions across many branches within our program. Now there are two types of switches. There is an expression switch and the type switch. With an expression switch, each case contains an expression. And this expression is then compared to the switch expression. A type switch, on the other hand, has cases that contain types. And these types are compared with the type represented by the switch expression. Other than this distinction, the type switch is similar to the expression switch. So, to clarify things, let's look at some examples. Okay, here are a couple of simple little examples. In our first example on the left here, we have our func main() executing. And it sets a variable named "x" to an integer value of 4. It then prints out this integer value and then encounters the switch statement. And this is the switch expression. It's simply "x" which represents 4. So then the switch structure will compare each case until it finds a match. So, in the first case, it essentially does the evaluation – is 3 equal to x? So does 3 equal "x"? In other words, is 3 equal to 4? Well no. So it moves on to the next case. Is 4 equal to 4? Well yes. So it's found its match. So it prints the word "four" and exits the switch structure.

Now, in the example on the right, again func main() executes. And this time it sets a variable named "length" to an integer value of 9. Then it encounters the switch statement. And notice the lack of an expression here. Well, what that does is it essentially allows each case to be evaluated on its own. So, in this case – the first case – no pun intended it says or it evaluates is length, which is 9, less than or equal to 7? Well no. Is "length" or 9 less than or equal to 8? Well not again. And, in this case, is "length" greater than 8? Is 9 greater than 8? And the answer is yes. So it would print the word "Long" and then exit the switch structure. Here are a couple of other cool examples. Okay, in the example on the left, we have our switch expression which basically evaluates to the day of the week that it is right now. And then, for our first comparison case, here we have multiple expressions. We have time.Monday, time.Tuesday, time.Wednesday, time.Thursday, and time.Friday. And, basically, what this is doing is it's comparing the case expressions to the switch expression. So it's saying, is time.Monday? Or is the day of week Monday? Or is the day of week Tuesday or a Wednesday or a Thursday or a Friday? And, if it is, then it prints that "it's a weekday". And, if it isn't, then it goes to the default case which is "it's the weekend", okay. And you can have a default case exactly once in every switch structure.

Now, in the example on the right, again we're implementing somewhat of an if...else logic. Really, I mean we have got the time here. We set a variable of time with the current time. And then we encounter our switch. And again, there is no expression. So we're going to look at the first case and compare it, so time.Hour() < 12. So is the current time less than 12? And, if it is, it prints "before noon". And let's just say it's not, okay. So, then it evaluates to the default case. So it would print "afternoon". So here is our If...else logic. So, in the first case, if time is less than 12 or if the hour that is just less than 12, then print "before noon", else print "afternoon". So that shows you how versatile the switch statement is. Okay. So, in this video, we looked at the switch statement in Go.

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

The defer Statement in Go

Learning Objective

*After completing this topic, you should be able to*

* *recognize and work with the defer statement in Go programs*

**1.**

In this video, we'll have a look at the defer statement in Go. A defer statement defers the execution of a function until the function that is contained under returns. Deferred function's arguments get evaluated right away. But the call – the actual call to the function – is not executed right away. Rather it's deferred. Let's look at an example. Okay, so here we've a func main() which gets executed. And the first line defers the first call to Println. So that's put on the deferred calls list. Then the second instance of the call to Println is executed and it prints "hello" to the I/O device – whether it's the screen or what have you. Then the end of the main function is encountered. And it's at this point where the surrounding function of the defer – that's func main() – occurs. So that means that the first call to Println at that point gets executed. So it prints "world". So the net effect here is that it prints "hello" first and then "world".

*[To demonstrate the defer statement, the presenter explains the following lines of code: package main import "fmt" func main() { defer fmt.Println("world") fmt.Println("hello") }]*

Now let's consider this example of stacking defers. When a function is deferred, it gets pushed on to a list. And the way that deferred functions are executed is last in, first out. So what that means is that the last function to get deferred is the first one to get executed. Okay, so let's have a look at our example over here. We've a func main() that gets executed. Then we've a call to Println. So it prints "counting" to the I/O device. So there is counting. Then it encounters the for statement. And, within this for statement, it will repeat this statement inside the for, a total of five times. The first time through, it will defer the call to Println with the value 0. And the second time through, a value of 1; third time through, a value of 2; and so on. So those calls are deferred. So program moves on, makes the call to Println, and prints "finished". So we see that here.

*[To demonstrate the stacking of defer statements, he explains the following lines of code: package main import "fmt" func main() { fmt.Println("counting") for x := 0; x < 5; x++ { defer fmt.Println(x) } fmt.Println("finished") }]*

Then the end of funct main() is encountered and that is the surrounding program. So now the functions get called last in, first out. So the last in would have been when "x" had a value of 4 here. So we see that printed here, followed by 3, then 2, then 1, then 0. Here is the example of using a defer. When we have our code open a file, we would later need to close that file as well. So here, in this example, we issue a defer on our close statement to close the file. There are three advantages in doing it this way. First, the call to close is near the call to open. So we can more easily understand what we're doing and why. Second, if our surrounding function had multiple return statements, close will occur before them. And third, deferred functions are run even if a runtime panic occurs. And let's take a look at an example of that.

Okay, in this example, we write the code this way because it defers the call to an unnamed function. So again, the call to the unnamed function is deferred. So the program goes on. And it will execute the call to panic. Panic is a function, which is a way to create a runtime error. We use the recover function which is a built-in to halt the panic and recover. And it also returns the value that was passed to panic. So here it's placed in a variable named "str", okay. So, when the main function – which is their surrounding function – ends, it prints the value of "str" which is "PANIC". And the thing is here that panic has to run first. It has to run first in order for recover to work. So that's why we defer that unnamed function with the call to recover. So, in this video, we looked at the defer statement in Go.

*[To demonstrate the application of defer function to Recover and Panic, he explains the following lines of code: package main import "fmt" func main() { defer func() { str := recover() fmt.Println(str) }() panic("PANIC") }]*

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Exercise: Using Flow Control Statements in Go

Learning Objective

*After completing this topic, you should be able to*

* *write a Go program that uses flow control statements*

**1.**

In this exercise, we'll practice using flow control statements in Go. The objective in this exercise is to recognize and work with for, if, switch, and defer statements. So let's go over to our workspace and practice.

Okay, so here is a very simple implementation of a for statement. We initialize a counter variable, we evaluate based on some condition, and then we perform a couple of statements inside the for loop. And the second statement is necessary where incrementing "z" by 1 each time, because if we didn't, this would just continue to run forever. There would be no way to increment "z". So let's go and try this on the command line – go run forStatement.go. And there it printed out 0, 1, 2, 3, 4, 5. Now one quick thing here – we can replace these two statements and this statement with one line of code and that's this one. So let's just do that. Let's get rid of this. We don't need this increment anymore. Okay, so much cleaner, much more compact. And it's going to do exactly the same thing.

*[In the "Sublime Text 2" text editor window, four tabs are open. The tabs are forStatement.go, ifStatement.go, switch.go, and defer1.go. By default, the forStatement.go tab is open. It contains the following program: package main import "fmt" func main() { // for z:= 0; z <= 5; z++ z := 0 for z <= 5 { fmt.println(z) z = z + 1 } } He runs the program in the Command Prompt and the following output is generated: 0 1 2 3 4 5 To make the above program look cleaner, he makes the following change in the main function: func main() { for z:= 0; z <= 5; z++ fmt.println(z) } He runs the program again and the same output is generated.]*

So, in this case, what we're doing is we're initializing "z" with the value of 0. And then we separate by semicolon that, and the test expression, which is "z <= 5". And then we separate that by semicolon with the increment expression which is "z++", which is just a shorthand way of saying "z" equals "z+1". Okay, so let's go and check this out. First we'll save it. Then we'll go to our command line and we'll run this again. And voila, same thing – exactly same thing only much better, much cleaner code. Okay, so let's check out the if statement. Okay, now this if statement is nested within a for loop, okay. And the for loop is going to be evaluated for y <= 10. Starting with the value of y := 1. So it's going to go through this loop ten times.

*[The presenter switches to the ifStatement.go tab. It contains the following program: package main import "fmt" func main() { for y := 1; y <=10; y++ { if y % 2 == 0 { fmt.Println(y, "even number") } else { fmt.Println(y, "odd number") } } }]*

Now the first time through, it is a value of 1. Okay, so keep that in mind. And it encounters this if statement. And what this if statement does or the evaluation expression here – if y % 2 == 0 – what does that mean? Well, what that means is if "y" is exactly divisible by 2. The percent sign is the remainder operand. So it means divide "y" by 2 and the remainder has to be 0. So it has to be exactly divisible by 2. Is 1 exactly divisible by 2? Well, no it's not. So the first time through this loop, the if condition is not met. So it hits the else condition and it runs this line of code which prints the value of "y" separated by a space by virtue of the comma and the word "odd number". So the second time through the loop, the value of "y" is 2. So it encounters the if statement. Is 2 exactly divisible by 2? Well, yes. It certainly is. So it executes this line of code. So we should see a printout of the numbers 1 to 10 and have them accurately identified as odd or even numbers. Well, let's go and check this out.

Okay, go run ifStatement.go. And there we have it – 1 through 10 all accurately recognized as odd or even numbers. Okay, let's check out the switch statement. Now this switch statement is an attempt to implement the if...else logic in a way that we just saw in our if statement. And how this works is we have got a condition here, which...or an expression – the time.Hour(). And, if that hour of the time is less than 12, then it executes this statement and prints "before noon". If it's not, then it encounters the default case. And the default case prints "afternoon". And then it exits a switch structure. So that's it, just one time through. Alright, so let's check this out. And it is afternoon here so we should see the word "afternoon" printed there. Okay, and we do. So it works. Okay, next example – defer,

*[He runs the ifStatement.go program in the Command Prompt and the following output is generated: 1 odd number 2 even number 3 odd number 4 even number 5 odd number 6 even number 7 odd number 8 even number 9 odd number 10 even number The presenter opens the switch.go tab. It contains the following program: package main import ( "fmt" "time" ) func main() { time := time.Now() // implement if/else logic in a switch structure switch { case time.Hour() < 12: fmt.Println("before noon") default: fmt.Println("afternoon") } } He runs the program in the Command Prompt and "afternoon" is printed as an output.]*

okay. So this defer is written in a way to demonstrate what is called stacked defers. So basically, when you defer functions, they get pushed onto a list of deferred functions. And what happens is when the calling or the encompassing function that defers is called in or the defers are created in, when it closes, then the last one in is going to be the first one out. Okay, so in this particular example, we have...it's going to go through this loop five times. It's going to defer the printing of the value of "x" five times. And it's also not going to defer this line of code. It's going to print that a certain number was deferred, okay. And then it will continue through that loop until again the value "x < 5" condition is not met.

*[The presenter opens the defer1.go tab. It contains the following program: package main import "fmt" func main() { fmt.Println("\n Last In ---> First Out\n") for x := 0; x < 5; x++ { defer fmt.Println(x, "deferred") } fmt.Println("\n") }]*

So the first time through the loop, "x" is 0. Let's just go through a couple of examples. So it will defer the calling of Println with a value of 0. And then it will print the value of 0 with the word "deferred" beside it. Next time through, "x" is 1, it will defer that and so on. And the last time through, "x" has a value of 4 because remember the condition is "x" less than 5. So the last value when these two statements will be executed, "x" will have a value of 4. So the last deferred call to Println will have a value of 4. So we should see that printed first. Alright, so let's go and check it out. Okay, so first we see our message "Last In ---> First Out", then we see 0 is deferred first. 1, 2, 3, and then 4 are deferred. And then we see that 4, as we expected, is the first one that's going to be popped then 3, 2, 1, and 0. Okay, so in this exercise, we practiced using flow control statements in Go.

*[He runs the defer1.go program in the Command Prompt and the following output is generated: Last In ---> First Out 0 deferred 1 deferred 2 deferred 3 deferred 4 deferred 4 popped 3 popped 2 popped 1 popped 0 popped]*

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Arrays in Go

Learning Objective

*After completing this topic, you should be able to*

* *work with arrays in Go programs, including declaring and initializing them*

**1.**

In this video, we'll have a look at how arrays are implemented in Go. An array is a data structure that contains a sequence of elements of a single type with a fixed length – length referring to the number of elements it contains. It goes without saying, but I'll say it anyway. In case you are new to programming, an array cannot have a negative number of elements. In other words, you cannot refer to the elements of an array with a negative number. I mean they just don't work like that. But you should understand that they can be multidimensional. And I'll show you couple of examples of that shortly. One more thing, in practice, you really don't see arrays much in Go, at least not directly. Most often in Go, you will use slices which are based on arrays. So an understanding of arrays is actually necessary.

Okay, so the length of an array – that is the number of elements it has – is part of its type. And, when declaring an array, the length must be a nonnegative constant of type int. But what do I mean – a nonnegative constant? Well, in the first example here, 18 is a constant. I mean 18 is 18, that won't change. But they can also be declared programmatically meaning you can declare them with an expression like this one – 4\*N – with N being a constant of type integer. Then you can also declare an array with any number of different element types, but each element in any specific array has to be exactly the same type. So an array of integers has only integers, floats – floats, pointers – pointers, and so on. You get the picture. Okay, so here are a couple of other examples. So this is an array of length 3500 of pointers. This is a multidimensional array. So it's got two dimensions – 5 and 8 of integer values. And this is another multidimensional array with three dimensions of float64 values.

*[The syntax of an array is as follows: ArrayType = "[" ArrayLength "]" ElementType The examples of arrays are as follows: [18] byte [4\*N] struct {a, b int32} [3500] \*float64 [5][8] int [4][2][4] float64]*

Okay, now in referencing the elements in an array, so that you can get the value stored there, you need to be aware that they are indexed starting at 0. So the first element in an array is referenced as a zeroth item. So, for example, n[0] has a value of 3 stored at that location. So, in a program, if I wrote an expression like n[0]\*4, then it would really mean three times 4. If instead my expression were n[3]\*4, then my program would go and get the value stored at n[3], which is 8. So my expression would actually be eight times 4. Okay, couple of examples. First, in this example, we declare a variable "z" which is an array of string values with the length of 3. z[0] is set to "Hello", z[1] is set to "My", and z[2] is set to "World". So, when we print that out on the next line, the output is [Hello My World]. In this example, we declare a variable "y" which is an array of integer values of length 4. Now y[3], which is again the fourth element, recall we said y[3] = 10. And this brings up another point. When you declare a variable, like we have here in the previous example, we haven't initialized the values to anything. So there is zeroed values which means that in this case with the four integer values, they would be 0, 0, 0, and 0. But, in the next line y[3] = 10, we set the fourth element to 10. So our output would be [0 0 0 10]. Okay, so in this video, we had a brief look at arrays in Go.

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Slices and the Slice Header

Learning Objective

*After completing this topic, you should be able to*

* *declare and initialize slices in Go programs*

**1.**

In this video, we'll take a look at slices in Go. In Go, you're going to use slices rather than arrays directly. I mean in Go, you don't see that many arrays. You more often see slices, which are based on an underlying array. Actually you can have a slice of a slice too, but for now don't worry about that. So, because they're based on arrays, they are a lot like them. Like they are indexed and they have length. But unlike arrays, their length can vary. Now, since slices are based on an underlying array, they're like references to the underlying array. So they share storage with it. This makes them very cheap in terms of resources. And you can have any number of slices based on a particular array. How is that for value? So look, when you declare a slice, the length is variable. Length is therefore not a part of its type – which is different than arrays – because arrays have length as part of their type. You can programmatically use the built-in len function to get a slice's length if need be.

*[The syntax of a slice is as follows: SliceType = "[" "]" ElementType]*

There are a couple of ways to create slices. First here is an example of declaring a slice and initializing it with some values using a slice literal. Here are the parts of this declaration. First we give it a name. In this case, slice. Then we're using the colon-equals operand to declare. Next, square brackets indicating that it's a slice and the type. The type indicates what kind of elements that the slice has in it. And, in the curly braces here, we are initializing some values. In this case, four of them: 7, 9, 10, and 12. Another way to create a slice is to use the built-in make function this way. First the keyword "make". And, since it's a function, you need parenthesis. For parameters, the first one includes the square brackets telling Go. It's a slice along with a type represented by T here. So, for example, instead of T, you would write integer, float64, string, and so on. Then you specify the length, then you specify the capacity. What? What is a capacity? Well, simply put – the capacity is the size of a slice's underlying array. Remember we're taking a slice of it.

*[The presenter explains the following line of code for declaring and initializing a slice: slice := [] int {7, 9, 10, 12} He explains the following line of code for declaring a slice using the "make" function: make([]T, length, capacity)]*

Now, if you leave out capacity and don't specify it, the capacity is assumed to be equal to the length. Okay, so let's look inside this slice and see what is actually going on. Okay, so here we're using the make function to create a slice. And it's a slice of 5 byte values. Can you see we've left out the capacity? So it's assumed to be equal to the length. Now one other thing to remember, when you start counting, the values start at index 0. So the first one to the far left would be s0 then s1, s2, s3, and s4. Okay, in the second example here, we are taking a slice of that slice. And we're doing it using what is called the [high : low] expression. And the [high : low] expression consists of two parts – a low index, which is 2 and a high index, in this case, which is 4. Now, in this case, what we're looking at in the underlying array...and we're taking a slice of it. So here is the overall array. So let's count from the left – s0, s1, and s2. So our new slice is s2. And it goes up to s4 - 1 because the high index is not inclusive. So these two here – s2 and s3. However, as for capacity, it goes from the base or the low index to the end of the slice, which is a capacity of 3. So we have a length of 2 and a capacity of 3 in this case.

Now, in the final example, again we are taking a slice of our slice. This time check this out. We have left out the low index. So what does that mean? That means it starts at a base index of 0. So it would start out here at s0. And, if we go to cap(s) – which is the capacity of s – cap is similar to len and that it gets the capacity of a slice, so the capacity of the slice is 3. It will go from 0 to the capacity s0, s1, s2 - 1, right, because capacity is 3 and 3 - 1 is s2, so that's s0, s1, s2. So that's why now we have a length of 3 and a capacity of 3. Okay. So, in this video, we had a brief look at slices in Go.

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Slices and Functions

Learning Objective

*After completing this topic, you should be able to*

* *work with slices within functions and methods of your Go programs*

**1.**

In this video, we'll consider some examples of slices in Go. All right. So, in this first example, we are using slice literal to declare and initialize a slice of integer values with the length of 4. Then we print out the slice, then we encounter this for loop. And what is this doing? Well, from left to right, we've got i equals 0. So it starts at 0 and the expression is or the conditional expression that is i less than length of slice – so for i, less than 4. So we'll go for i equals 0 up to and including 3. Okay, and then we see the print statement, which basically prints the slice index number in square brackets equals and the actual value of the slice. Okay, so let's check out our output. That's exactly what we expected. Okay. First line, slice equals and our entire slice. Second line, slice 0 equals 7, slice 1, 9, slice 2, and slice 3.

*[In the first example of slice, the presenter explains the following program: package main import "fmt" func main() slice := [] int {7, 9, 10, 12} fmt.Println("slice ==", slice) for i := 0; i < len(slice); i++ { fmt.Printf("slice[%d] == %d\n", i, slice[i]) } } The output of this program is as follows: slice == [7 9 10 12] slice[0] == 7 slice[1] == 9 slice[2] == 10 slice[3] == 12]*

Okay, here again, we see a slice literal of exactly the same as our last example. But, in this case, in the second print statement, we're only including slice 1 through slice 4-1 because the high index is not inclusive. So it will print out slice 1, slice 2, and slice 3. In the second print statement, the low index is omitted. So a value of 0 is assumed. So, in this case, it will print out slice 0, slice 1, and slice 2 because again, the high index is not inclusive and it's set at 3 here. So it will print out up to slice 3-1, which is 2. Again finally, the high index is omitted here. And so Go assumes that the high index is length of slice or in this case 4. Now what that means? Again, because the high index is not inclusive, it will print s2 or slice 2 and slice 3. So let's have a look at our output and see, awesome. All right. So exactly what we expected – slice, the whole slice in the first Println. In second Println – check it out – we've got slice 1 through 4, which means print slice 1, slice 2, and slice 3, which is 9, 10, and 12 exactly as we expected. And slice [ :3] prints out slice 0, slice 1, and slice 2. And, in the last output, slice 2 and slice 3.

*[In the second example of slice, the presenter explains the following program: package main import "fmt" func main() slice := [] int {7, 9, 10, 12} fmt.Println("slice ==", slice) fmt.Println("slice[1:4] ==", slice[1:4]) fmt.Println("slice[:3] ==", slice[:3]) fmt.Println("slice[2:] ==", slice[2:]) } The output of this program is as follows: slice == [7 9 10 12] slice[1:4] == [9 10 12] slice[:3] == [7 9 10] slice[2:] == [10 12]]*

Okay. So, in this last example, we see we are declaring a variable named "pow" to represent power. And we declare as an array of integer values. And pow 0 is 1, pow 1 is 2, 4, 8, 16, and so on. And notice that these represent the values if you were to take 2 to the 0 powers 1, 2 to the 1 power is 2, 2 to the second power is 4, third is 8, 16, and so on. Now, inside func main(), notice that we've got a for loop and our for loop has a new word range. Now what that does is it causes the for loop to loop through each value within our slice because we've defined it as the range of slice pow. So it will go through each element in pow. And, on the left of the colon equals operand, you see two values n and p. These two values represent n – the index and P – the value that exist at the index location. And then we finally see a print of our values. So the first time through the value of n will be 0. So let's work through this. So the first time through, it will say 2 to the 0 power equals p. And that is 2 to the 0 is equal to 1. In the second time through, 2 to the 1 equals 2, and 2 squares 4, 2 cubes is 8, so on. So let's check out our output, right. Look at that 2 to the 0 is 1, 2 to the first power is 2, 2 square is 4, 8, 16, 32, and so on. All right. So, in this video, we had a look at some slice examples in Go.

*[In the last example of slice, the presenter explains the following program: package main import "fmt" var pow = []int{1, 2, 4, 8, 16, 32, 64, 128} func main() { for n, p := range pow { fmt.Printf("2\*\*%d = %d\n", n, p) } } The output of this program is as follows: 2\*\*0 = 1 2\*\*1 = 2 2\*\*2 = 4 2\*\*3 = 8 2\*\*4 = 16 2\*\*5 = 32 2\*\*6 = 64 2\*\*7 = 128]*

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Manipulating Slices

Learning Objective

*After completing this topic, you should be able to*

* *manipulate Go slices using Go's built-in make, copy, and append functions*

**1.**

In this video, we'll spend some time manipulating slices in Go. Okay, so let's have a look at this first example. First we create a couple of slices using calls to the make function, then we manipulate some slices, okay. So look, the easiest way to understand this is to compare the output to our code. So let's bring our output up, and let's have a look at our definition for our slice called a. So here we're defining "a" using the make function. And we're defining it as a slice of integer values with a length of 4. Now we haven't set a capacity so that's why we see a capacity of 4 because it's assumed to be exactly the same as length. Now why do we see zeros between the square brackets in our output, because we didn't initialize any values so they are zeroed and zero integer values are 0. So we see four zeros, okay. So now b – b is created using, again, the make function. It's a slice of integer values, but we've set the length to 0 and the capacity to 4. So that's why we see a length of 0, capacity of 4, and nothing between the brackets. Now c is interesting because it's a slice of b.

*[In the first example, the presenter explains the following program: package main import "fmt" func main() { a := make([]int, 4) printSl ("a", a) b := make([]int, 0, 4) printSl ("b", b) c := b[:1] printSl ("c", c) d := c[2:4] printSl ("d", d) } func printSl (s string, x []int) { fmt.Printf("%s len=%d cap=%d %v\n", s, len(x), cap(x), x) } The output of this program is as follows: a len=4 cap=4 [0 0 0 0] b len=0 cap=4 [] c len=1 cap=4 [0] d len=2 cap=2 [0 0]]*

So one of the other things you'll notice is that the [high : low] expression. We've omitted the low value. So the low index is assumed to be 0. So what we'll see in c is a slice of b from b0 to b0, which is just b0. So we should see a length of 1 – which we do – capacity of 4, and just one 0 between brackets. And we do. Now d is interesting as well because it's a slice of c. However, we are taking a slice of c from c2 to c3 because again the high-index value and our [high : low] expression is not inclusive. So c2 and c3 don't have a value. So they're zeroed. So we should see 0 is between the brackets and we do, we should see a length of 2, and we should see a capacity of 2 as well because those values didn't exist before we created them. And that's exactly what we see. Okay, so let's check out this example. Okay, so here we're using the append function. And one of the things to recognize about the append function is it – for the backing array – is too small to fit the given values, then a bigger array will be allocated. The return slice will point to the newly allocated array. Okay, so here we see slice 1, which is created and initialized as an integer slice with three values.

*[In the second example, he explains the append function in the following program: package main import "fmt" func main() { slice1 := []int{8,9,10} slice2 := append(slice1, 11, 12) fmt.Println(slice1, slice2) }]*

Now slice 2 is initialized using the append function. And we're using the append function on slice 1 and we're adding two values to it. So what we should see in slice 2 is 8, 9, 10, 11, and 12. And that's exactly what we see. On the left, you see the output for slice 1, which is 8, 9, and 10. And, on the right, you see the new slice, which is 8, 9, 10, 11, and 12, okay. So now we'll use the copy function. And the thing about the copy function is if the slice you're copying is too small, then copy will only copy those items that will fit starting from index 0. So here we see copy (slice 2, slice 1). And what this does is slice2 is the target and slice1 is the slice that we're copying from. But again, slice1 has a length of 3 and slice2 has been defined with only a length of 2. So what we should see in our output for slice2 is just 8 and 9. So let's have a look. Okay, so slice1 is 8, 9, and 10. And slice2 is 8 and 9, okay. So, in this example, we initialize a variable x and its type is slice. It's a slice of integers. But we haven't defined a length, we haven't initialized it in anyway. So what we should see in our print out is first of all see x, which is the slice – which will be a blank slice – then the length of x should be 0, then the capacity of x should also be 0. Let's check it out and it is. And according to our if statement, what we're doing is comparing x to nil. And, if it's a nil slice, then it will print nil. And this is exactly what it does. Okay. So, in this video, we spent some time manipulating slices in Go.

*[The output of the append program is as follows: [8, 9, 10] [8, 9, 10, 11,12] In the third example, he explains the copy function in the following program: package main import "fmt" func main() { slice1 := []int{8, 9, 10} slice2 := make([]int, 2) copy(slice2, slice1) fmt.Println(slice1, slice 2) } The output of this program is as follows: [8, 9, 10] [8, 9] In the fourth example, he explains the nil slices in the following program: package main import "fmt" func main() { var x []int fmt.Println(x, len(x), cap(x)) if x == nil { fmt.Println("nil!") } } The output of this program is as follows: [] 0 0 nil!]*

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Structs in Go

Learning Objective

*After completing this topic, you should be able to*

* *work with structs in your Go programs*

**1.**

In this video, we'll spend some time introducing the struct data type in Go. So you know go has a number of standard data types. So what happens when the standard data types just don't do it for you? Well, you can define a custom data type and that is what a struct is. You can think of a struct as a data structure made up of some fields of data. The classical way of teaching this is to use geometric shapes. And so here we'll be talking about rectangles, so width and height. Okay, so you see here that we've defined the type called Rect and it's a struct and it's got Width and Height and those are both integer values, okay. So down in func main() when we print our Rect, we are setting the values of Width and Height directly to 7 and 8. So that's what we'll see in curly braces in our output. All right. Now, to access struct fields, you can use the period operator and the name of the field that you're referencing, okay. So here again, we've defined Rect as a struct type with Width and Height. And, down in func main(), we are creating a new Rect. Its name is r. And it's got values of 1 and 2 for Width and Height. Now, in the next line, we use the period operator and the field name directly to set the value to 18. And, in the println, we're just printing out the Width. And again, we are dressing it or accessing it using the period operator and the name of the field. So let's check out our output and it's 18.

*[In the first example for struct, the presenter explains the following program: package main import "fmt" type Rect struct { Width int Height int } func main() { fmt.Println (Rect{7, 8}) } The output of this program is as follows: {7, 8} In the example to demonstrate accessing struct fields, he explains the following program: package main import "fmt" type Rect struct { Width int Height int } func main() { r := Rect{1, 2} r.Width = 18 fmt.Println (r.Width) } The output of this program is 18.]*

Okay, so now we're talking about structs and using pointers with structs. So again, in this example, we have Rect struct type of Width and Height. And, down in func main() – have a look – we define r as Rect with the values of 4 and 6 for Width and Height. Then we define a variable p as a pointer to r. Then in the next line, we use p, the period operator, and the name of the field to set it to 8 directly. So, when we print out our Rect called r, we should see the values of 8 and 6 in curly braces and we do. All right. So struct literals are the idiomatic way to initialize struct instances. Okay, so here we see four variables defined. r1 is defined as a Rect with the values of 7 and 8 for Width and Height. r2, we see that it's a Rect and we've set it's Width directly with the name of the field, colon, value syntax. And that's another way we can access the values of a filed directly,

*[To demonstrate how to use pointers in struct, he explains the following program: package main import "fmt" type Rect struct { Width int Height int } func main() { r := Rect{4, 6} p := &r p.Width = 8 fmt.Println (r) } The output of this program is as follows: {8 6} To demonstrate the struct literals, he explains the following program: package main import "fmt" type Rect struct { Width, Height int } var ( r1 = Rect {7, 8} // type Rect r2 = Rect {Width:4} // Height is implicitly 0 r3 = Rect {} // Width and Height both 0 p = &Rect {7, 8} // type \* Rect ) func main() { fmt.Println(r1, r2, r3, p) }]*

okay. Now the thing here is that we've omitted Height. So Height will explicitly be set to 0. r3 is set to a Rect with no value set, so both Width and Height will both be implicitly set to 0. And p is defined as a pointer to Rect with the values of 7 and 8. Now, done in func main(), we're going to print out all of these values. So r1 should be just a Rect structure in curly braces with 7 and 8. And r2 should return a Rect with the values of 4 and 0 because again the Height was implicitly set to 0. And r3, both Width and Height will return 0. And p should return a pointer to a Rect with the values of 7 and 8. Let's have a look. All right, so we see the values for r1 – 7 and 8, r2 is 4 and 0, and 0 and 0. And then look at this, we've returned a pointer for p. Okay. So, in this video, we spent some time introducing structs in Go.

*[The output of the program demonstrating struct literals is as follows: {7, 8} {4, 0} {0, 0} &{7, 8}]*

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Maps in Go

Learning Objective

*After completing this topic, you should be able to*

* *declare, initialize, and manipulate maps in Go*

**1.**

In this video, we'll spend some time introducing maps in Go. Maps are one of Go's advanced composite types. They are similar to arrays and slices being a collection of things or a list in other words. But they're unordered. That is, you don't reference the elements in a map by referring to an index number like the way you do in arrays and slices. No, maps are essentially a collection of key-value pairs. And, if you're an experienced developer, you may have already encountered them before as hashes or dictionaries like for example a hash in Ruby. Okay, so you have an element type and this is the value. Then you have an index or key type. And keep in mind, when you're creating maps like slices, an uninitialized map is nil.

Okay, so here is how you define a map. MapType is the name of your map. The map keyword is used to tell the compiler that this is a map type. Inside the square braces, you specify the KeyType and then an ElementType. Now just one thing the KeyType must be a comparable type. In other words, a type that you can do an equals equals or a not equals comparison with, which includes numeric, Boolean, strings, and most other types made up of those types. Here is an example. Okay, so here we set a variable m equal to a map type. And we set the key type to a string and the value is a struct. That's right. We can set the value to a struct because it's comparable,

*[The syntax for defining a map is as follows: MapType = "map" "[" KeyType "]" ElementType In the first example for map literals, the presenter explains the following program: package main import "fmt" type Rect struct { Height, Width int } var m = map[string]Rect{ "Rect1" : Rect{1, 2,}, "Rect2" : Rect{4, 6}, } func main() { fmt.Println(m) }]*

okay. So, in our map, we've got two values – Rect1 and Rect2. And Rect1 is a Rect struct with values of 1 and 2 for width and height and 4 and 6 for width and height for the second one. So, down in func main(), when we print this out – check this out – we get Rect2 and then Rect1. Well, what happened there? Well, here is the thing. Because it's unordered, it could return Rect1 first or Rect2 first. You just don't know. And Go won't necessarily return at the same way every time. Okay, let's look at an example of how we can manipulate maps. So, in the first part of this code, we define our map "w" using the make function, okay. And then we set the Answer to 10 and then we print that value out.

*[The output of the program is as follows: map[Rect2: {4 6} Rect1: {1 2}] In the example for manipulating maps, the presenter explains the following program: package main import "fmt" func main() { w := make (map[string] int) w ["Answer"] = 10 fmt.Println("The value:", w["Answer"]) w ["Answer"] = 20 fmt.Println("The value:", w["Answer"]) delete (w, "Answer") fmt.Println("The value:", w["Answer"]) v, ok := w ["Answer"] fmt.Println("The value:", v, "Present?", ok) }]*

Next, we alter the value stored as Answer and then we print that out. Then we delete the value stored as Answer and we print that out. Next, we can test whether a key is present like this. You assign two values on the left side of the expression. And the first will return the value while the second will return true if the key is present and false if it's not. So let's check out our output. Okay, so the first part it prints out the value as 10. Then we alter it, it prints out 20. Then we delete it, so it's zeroed. And it sets it at 0. And again, in the next part, it's still zeroed. But then we get to test whether the key is present or not. And, of course, it's not. We deleted it. So it returns a false. Okay. So, in this video, we spent some time introducing maps in Go.

*[The output of the program is as follows: The value: 10 The value: 20 The value: 0 The value: 0 Present? false]*

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Advanced Functions

Learning Objective

*After completing this topic, you should be able to*

* *recognize and work with some of the more advanced features associated with functions in Go*

**1.**

In this video, we'll look at some advanced functions in Go. All right. So first up, let's look at variadic functions. Look here, when we define this function, we put an ellipsis as the last parameter just before the type. This tells the compiler that there could be zero or more parameters, but we aren't sure how many. Now, down in func main(), we call add with five parameters and it dutifully takes those five parameters and with the code adds them all up. Now note the use of range here. Range allows the for loop to loop through the entire range of arguments. Okay, so the final output being 8 plus 13 plus 4 plus 6 plus 3, which is 34. Okay, so now let's take a look at what is called closure. Okay, closure describes a situation where we create a local function like this one – decrement. And it's created inside another function main(). Now, when it's created like this, it can access other local variables in main() because of the scope of the variable. So, when main() calls decrement, "a" is available to it. So here in our part of the code where we are calling decrement, decrement can access "a". So our output will be 3 this time. Then the next time we call it, it started out as 3, then it's decremented to 2. So the output should be 3 and 2 and it absolutely is. Now this type of function along with the nonlocal variable that it can reference is called closure. And again, closure is one form of functional programming.

*[In the example for variadic functions, the presenter explains the following lines of code: func add(args …int) int { total := 0 for \_, n := range args { total += n } return total } func main() { fmt.Println(add(8, 13, 4, 6, 3)) } The output of this program is 34. To demonstrate closure, he explains the following lines of code: func main() { a := 4 decrement := func() int { a -- return a } fmt.Println(decrement()) fmt.Println(decrement()) } The output of this program is as follows: 3 2]*

Functions can call themselves. How about that? This is called recursion and it's another type of functional programming. Here is an awesome example that demonstrates how recursion works. Let's step through this. Down in func main(), we call the factorial function, we pass it the value 9. So the first time through it does a comparison – is x equals 0? Well, no. Of course, not, it's 9. So it hits the line, return x times...oh, another call to factorial with x minus 1. So keep this in your head. Return x, which is 9 at this point. So return 9 times an another call to the factorial. So it steps through it again. And this time, it's going to return 8. So now we've got 9 times 8 on our return stack and then 9 times 8 times 7. And so it continues through the loop until x does equal 0. At this point, the last time through it returns 1 without calling factorial again, of course. It then starts exiting in the calls with returns. So it returns 1 times 1, 1 times 2 all the way back up till it's 9. So we have 9 times 8 times 7 times 6 times 5 times 4 times 3 times 2 times 1, which is what factorial is. So our output, which is 9 factorial is 362880,

*[To demonstrate the factorial function, he explains the following program: package main import "fmt" func factorial (x int) int { if x == 0 { return 1 } return x \* factorial (x-1) } func main () { fmt.Println(factorial(9)) } The output of this program is 362880.]*

okay. So here is another type of closure. This time we have an anonymous function that gets return to a function call and what happens here is like this. Okay, the function fibonacci() returns another function, which is defined as an anonymous function here. The effect is that the returned anonymous function forms a closure over the x and y values, okay. So we call fibonacci() assigning the result to f. And remember the result being the anonymous function. This function captures its own x and y values, which are updated each time we call fibonacci() in this for structure. So the effect is that we'll get 0, 1, 1, 2, 3, 5 all the way up the fibonacci sequence until we reach 55, awesome. Okay. In this video, we looked at some advanced functions in Go.

*[The program to explain the Fibonacci function is as follows: package main import "fmt" func fibonacci() func() int { x := 0 y := 1 fmt.Println(x) return func() int { x, y = y, x+y return x } } func main () { f := fibonacci () for i := 0; i < 10; i++ fmt.Println(f()) }} The output of this program is as follows: 0 1 1 2 3 5 8 13 21 34 55]*

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Methods in Go

Learning Objective

*After completing this topic, you should be able to*

* *create and use methods in your Go programs*

**1.**

In this video, we'll look at methods in Go. So you have both functions and methods in Go. A method in Go is a function declared with a receiver, which is sort of like a parameter. So it has a name and a type. It can be a value or a pointer to a named or struct type. Because we define it with the receiver, we can call it using the period operator. Okay, so here is a simple little piece of code that uses a pointer as a receiver. So first, the Root type – a struct – is defined with an A and B value. Then the function Hyp() is defined with r – a pointer to the Root struct as a receiver. So, down in func main(), we define a struct of type Root with values 5 and 6 and then we call Hyp() with r and the period operator, nice. So the returned value is the square root of 5 squared plus 6 squared, which is 7.81.

*[The program that uses pointer as a receiver is as follows: package main import ( "fmt" "math" ) type Root struct { A, B float64 } func (r \*Root) Hype() float64 { return math.Sqrt(r.A\*r.A + r.B\*r.B) } func main() { r := Root{5, 6} fmt.Println("C=", r.Hyp()) } The output of this program is C = 7.81024967.]*

Okay, so now let's look at a method that's using a value receiver. So here we've used the type declaration to create a type int Root. Then we defined the function Abs(), like so. And this means that we can now call it with the period operator. So here we define a new variable r of type Root. And it's initialized with the value of negative square root of 2 or -1.4142, then we call Abs() with our Root type, float64 variable r and the period operator and it returns the absolute value of what was passed to it. And so that is 1.4142, approximately.

*[To demonstrate a method that is using a value receiver, he explains the following program: package main import ( "fmt" "math" ) type Root float64 func (r Root) Abs() float64 { if r < 0 { return float64(-r) } return float64(r) } func main () { r := Root(-math.Sqrt2) fmt.Println(r.Abs()) } The output of this program is 1.4142135.]*

Okay, so you're considering using a value or a pointer receiver. So which one did you use? Well, if there is any doubt, use a pointer. However, having said that there are situations where a value receiver makes more sense. Here are some things to think about. First it makes sense to use a value receiver if the receiver you're considering is a map, func, or channel. It also makes sense to use a value receiver for a small structs that don't change arrays or basic value types. Value receivers, in this case, are an efficient choice. Now, on the other hand, use of pointer receiver if the receiver is going to be altered. It needs to be a pointer or if it's a struct – it has a sinking field – it also needs to be a pointer. And size really does matter. So, if it's a large struct or array, the pointer receiver is the way to go. Okay. So, in this video, we spent some time introducing methods in Go.

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Go Interfaces

Learning Objective

*After completing this topic, you should be able to*

* *recognize and work with Go's implementation of interfaces*

**1.**

In this video, we'll look at interfaces in Go. The interface type provides us with a lot of flexibility and abstraction. Basically, interfaces allow us to specify the behavior of values and specific types. In essence, an interface type defines a set of methods. So think of it this way, the methods that comprise an interface are the interface. And, to put it in other way, they are defined similarly to structs with the type keyword. However, with an interface, we don't define fields instead, we define a set of methods. Okay. So, in this example, we define an interface named Notifier. It has a single method called Notify. Now, when the interface has a single method, it's good practice and quite typical to give interfaces a name containing an -er suffix. And while this is not a hard-and-fast rule, this is typically how they're named. And it just makes good sense really. I mean it provides a clear and meaningful naming convention for your interfaces. Although you can specify any number of methods, the number of interfaces in Go's standard library with more than two methods are few and far between. So just make your life easier and go with that for now.

*[To declare the interface, the presenter explains the following lines of code: type Notifier interface { Notify() error }]*

So let's have a look at an example. Before we type into this, I just want to let you know that because of the limited real estate here, we've left out the package main declaration and the imports lines. Now let's see what is going on here. We have a Calcer interface defined and it does one thing, it Calcs. Then we have a struct named Square with two float64s and a myFloat named type. Now, in func main(), look what happens. We have two different types. We have a myFloat type, and we have a Square struct type to work on. First let's look at the output, okay. So we've got two different types to operate on. So how can we get this output? Well, because the interface Calcer has been defined to work on two different types – a myFloat type and a Square type – and the Calc employee in each case depends on the value type, so it really does allow us to specify the behavior of values and specific types with our interface. Okay. So, in this video, we spent some time introducing interfaces in Go.

*[To demonstrate the interface type with an example, he explains the following lines of code: type Calcer interface {Calc() float64} type Square struct {x, y float 64} type myFloat float64 func main() { var c Calcer f := myFloat (-math.Sqrt2) s := Square{8, 6} c = &s fmt.Println(c.Calc()) c = f fmt.Println(c.Calc()) } func (f myFloat) Calc() float64 { if f < 0 { return float64 (-f)} return float64(f)} func (s \*Square) Calc() float64 { return math.Sqrt(s.X\*s.X + s.Y\*s.Y)} The output of this program is as follows: 10 1.4142135]*

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Concurrency in Go

Learning Objective

*After completing this topic, you should be able to*

* *describe how concurrency is implemented in Go and the benefits of writing concurrent programs*

**1.**

In this video, we'll look at concurrency in Go. Okay, so you should recognize that most programs of any typical size and complexity are actually composed of a bunch of tasks. For example, let's say, you're on an e-learning site watching some instructional videos, well, so are a bunch of others. So each request that arrives at the server is a separate task. The idea behind concurrency is to be able to run a bunch of smaller tasks by simultaneously synchronizing access to shared memory. But, you know, another way to describe it is by saying that all those tasks can run independently. In other words, they don't have to wait around for one to finish before they can carry on. So this segues nice into my next slide.

At Heroku's was a conference in 2012. Rob Pike, one of the developers of Go, gave a talk about concurrency versus parallelism. And it helps to clarify and solidify the fact that they are not the same thing. And it turns out this is a commonly held misconception. Now you can Google that video if you want to watch it. And it's worth it. So, to be clear here, in programming, concurrency is a series of independently running processes. Parallelism, on the other hand, is a series of simultaneously running processes. Okay, one other thing to note, Go's implementation of concurrency is rooted back in the 1970s, really, back to communicating sequential processes or CSP. And there is a fantastic paper written by Sir Tony Hoare, it's worth a read if you want to see where this whole idea is rooted. Anyway I digress. We could go on about concurrency and its root for days. But instead, let's talk about it in practical terms. Concurrency is implemented in Go by using goroutines. So let's look at this example. This is an explicit goroutine. And it's an explicit goroutine because we call this function using the go keyword. And speaking of this function, have a look at what it does. Really, it doesn't do anything special. It just accepts a string parameter and runs through a loop printing a message. However, it simulates something that we should understand about concurrency.

*[To give an example of concurrency, the presenter explains the following program: package main import "fmt" func f(msg string) { for i := 0; i < 10: i++ { fmt.Println(msg, ":", i) } } func main() { go f("value of i") var input string fmt.Scanln(&input) }]*

Normally, what would happen here is func main() would call it and then have to wait till it came back after we finished printing all of its messages. Well, using a goroutine. But it doesn't have to wait until that function returns. It carries on with the rest of the program. In this example, it executes the next two lines of code. Now the call to Scanln is made here because if it was not, the program would exit before the loop finished printing. Scanln waits for input essentially pausing the execution of func main(). And, for doing this, in this example, to wait for func f to return. And this is necessary. And you should note this because the function main() itself is an implicit goroutine. And, in this case, if we didn't pause it using the Scanln, it wouldn't have waited for f to return and it would have exited before f had a chance to print all of its messages out. Okay. So, in this video, we spent some time introducing concurrency in Go.

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Creating a Concurrent Program in Go

Learning Objective

*After completing this topic, you should be able to*

* *recognize how to create a concurrent program in Go*

**1.**

In this video, we'll create a concurrency program in Go. Okay, so here is our code. And, as you can see right now, it's not a concurrency program. We have no goroutines defined. And I wanted it this way because I want to show you what happens. So currently what is going to happen is this – One func main() will execute. It makes the call to a function called msg(). You can see it in the comment. It simulates a time-consuming process of five seconds. So this simulates like a call across a network or some other time-consuming I/O process. All right, and the thing is that the way that it's set up right now, it makes the call or func main() – that is makes the call to message. And then it has to wait around for a function msg() to finish what it's doing and return and then it can carry on. So let's check it out. Let's see what happens. Okay, one second, two seconds, three, four yawn, five yawn. And finally, from the func msg(), we get "I'm finished!" and then from func main(), we get "I'm finished!" because it's gone ahead and finished.

*[A "Sublime Text 2" text editor window is open alongside a Command Prompt window on the screen. The concurrency.go program is open in a tab on the text editor. It displays the following program: package main import ( "fmt" "time" ) func main() { msg() // Let's print to the console when we're finished fmt.Println("\nMessage from func main, I'm finished!") time.Sleep(time.Millisecond \* 2500) } func msg() { // simulate a time-consuming process (5 seconds) for i := 1; i <=5; i++{ time.Sleep(time.Millisecond \* 1000) if i > 3 { fmt.Println(i, "seconds… yawn") } else { fmt.Println(i, "seconds") }} // Let's print to the console when we're finished fmt.Println("\nMessage from func msg: I'm finished!") } He runs the program in the Command Prompt window and the following output is generated: 1 seconds 2 seconds 3 seconds 4 seconds… yawn 5 seconds… yawn Message from func msg: I'm finished! Message from func main, I'm finished!]*

Okay, so what we want to do ideally is make message a goroutine. So we do this very difficult task by typing go in front of msg(). And that's it. That will make this a goroutine. And, now this time, we'll see that instead of func main() waiting around until message comes back with all of its messages that it will carry on executing and it will perform these two lines. So let's save it with Ctrl+S. And we'll go over and run it again and see what happens. Okay, look at that. We got the message almost immediately from func main() that it's finished. But here is the thing, it didn't wait long enough for func msg() to finish. So let's add a little bit more of a delay in here.

*[To change this program into a go routine program, the presenter adds the word "go" in front of the msg function that is declared in the func main function. He again runs the program in the Command Prompt and the following output is displayed: Message from func main, I'm finished! 1 seconds 2 seconds]*

So, right now, I've introduced the delay of two and a half seconds here. And that's just not enough for the func msg() to finish what it's doing and come back and print all of its messages. So let's do that. We'll give it a seven and a half seconds this time because we know it's going to take at least five seconds because of this loop. Let's save that. And now we'll run it again. Okay, so now we get func main() coming back almost immediately, then three, four, five seconds, message from func msg(), it's finished. All right, cool. So that's all there is to creating a concurrency program. You make your function calls, goroutines, and then you're good to go. So, in this video, we created a concurrency program in Go.

*[To increase the time delay in func main function so that it prints the entire output, he makes the following changes in the line of code: time.Sleep(time.Millisecond \* 7500) He runs the program again in the Command Prompt and the following output is displayed: Message from func main, I'm finished! 1 seconds 2 seconds 3 seconds 4 seconds… yawn 5 seconds… yawn Message from func msg: I'm finished!]*

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Go Channels

Learning Objective

*After completing this topic, you should be able to*

* *describe the purpose of channels in Go programs and how to use unbuffered and buffered channels*

**1.**

In this video, we'll introduce channels in Go. Goroutines allow us to make our programs run concurrently. Channels are used by goroutines for communication between them and to intelligently synchronize execution. For example, one goroutine can tell another via a channel that a particular task is complete. Nice! There are two types of channels, unbuffered and buffered. Let's have a look at an example of each. Okay, channels are unbuffered by default. An unbuffered channel – also called a synchronous channel – will accept a "send' only if there is "receive" waiting for the corresponding send value. What this means is by default sends and receives will be blocked until both sender and receiver are ready. This allows us to wait until the end of our program to get the values of x and y without the necessity for any other type of synchronization. So let's check this out. Down here in func main(), we must create the channel first. In this case, it's an integer channel.

*[The presenter explains the following program as an example of unbuffered channel: func sum(a []int, ch chan int) { sum := 0 for \_, v := range a { sum += v } ch <- sum } func main() { a := []int{7, 0, -3, 5, 0, 4} ch := make(chan int) go sum(a[:len(a)/2], ch) go sum(a[len(a)/2:], ch) x, y := <-ch, <-ch fmt.Println(x, y, x+y) }]*

Then we can call our goroutines. Notice, when we call them, we pass the channel. And, up here, in the function sum, it's expecting an integer channel as a parameter. So, up in the function sum, the sum is sent on the channel. Now check out what happens. The execution of the rechannel operation gets blocked until the goroutines write data to the channel. Okay. So buffered channels can accept a defined number of values even if there isn't a corresponding receiver for those values. So here we create the channel and give it a buffer length of 5. We place the values 1 and 3 in the buffer. And then we can go and receive them. However, taking out, you have to be careful not to overfill the buffer. If that occurs, you'll get a scary deadlock error that states that all goroutines are asleep. All right. So, in this video, we had a brief look at channels in Go.

*[The presenter explains the following program as an example of buffered channel: package main import "fmt" func main() { c := make (chan int, 5) c <- 1 c <- 3 fmt.Println(<-c) fmt.Println(<-c) }]*

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

Exercise: Using Slices in a Go Program

Learning Objective

*After completing this topic, you should be able to*

* *write a Go program that creates and manipulates a slice*

**1.**

In this exercise, we're going to practice our knowledge of slices by using them in a Go program. So what you should do is write a Go program that creates and manipulates a slice. So you'll create the slice, manipulate some of the elements in it, use append to resize it and then copy the slice and then take a slice of a slice. Now pause the video and perform the exercise. And, when you return, I'll show you what I did.

Okay, so here we're defining our slice with a slice literal. It's an integer slice with four values. And here we are printing out some of the different values of the slice using an expression – a [low : high] expression. In this case, 1 to 4. In this case, we omit the low index, we omit the high index, and then we print out the length and the capacity. In the next part, we iterate through this for loop and basically iterate over the range of the slice and just change its values – subtract 5 from its initial values. Then we make a call to the report function. And what the report function does is it reports or prints the name of the slice, its length, and capacity. So, every time we call report, that's what it's going to output to the console. Okay, then we append two values to the slice to see what happens. Then we append eight values, then we append another eight values. And we're doing this to see what pattern emerges when we do our appends. Okay, then we make a copy of the slice. And we're making a copy of our large slice into a smaller copy. Then we'll do a reslice. And we'll see how that works. All right, let's go ahead and run it and see what happens.

*[A "Sublime Text 2" text editor window is open alongside a Command Prompt window on the screen. The slice.go program is open in a tab on the text editor. It displays the following lines program: package main import "fmt" func main() { slice := []int{10, 15, 20,25} fmt.Println("\nHere is our slice:") fmt.Println("slice ==", slice) fmt.Println("slice[1:4] ==", slice[1:4]) fmt.Println("slice[:3] ==", slice[:3]) fmt.Println("slice[2:] ==", slice[2:]) fmt.Println("len slice ==", len(slice)) fmt.Println("cap slice ==", cap(slice)) // now modify each element in slice for i, v := range slice { slice i = v - 5 } fmt.Println("\nThe new values in our slice:") report("slice", slice) // append 2 values to slice fmt.Println("\nNow we'll append 2 values to slice (what happens?):") slice = append(slice, 10, 20) report("slice", slice) // add 8 more values to our slice fmt.Println("\nNow we'll append 8 more values to slice (now what happens?):") slice = resize(slice) report("slice", slice) // add 8 more values to our slice fmt.Println("\nNow we'll append 8 more values to slice (guess what happens?):") slice = resize(slice) report("slice", slice) // make a copy only the copy has a cap of only 8 fmt.Println("\nMake a copy, only the copy has a cap of 8:") sliceCopy := make([]int, 8) copy(sliceCopy, slice) report("sliceCopy", sliceCopy) // Let's do a reslice! fmt.Println("\nLet's slice our slice for a reslice:") reslice := slice[1:5] report("reslice", reslice) func resize(slice []int) []int { for i := 0; i < 8; i++ { slice = append(slice, 1) } return slice }]*

Okay, so let's just follow this in our code. So, at the top, we've initialized our slice with the values 10, 15, 20, and 25. Then we print slices or the slice values from slice 1 through slice 4 minus 1, because the high index is not inclusive. So that's slice 1 to 3. So slice 1, so this is slice 0, slice 1 – 15, slice 2, and slice 3. Now, in the next example, we've omitted the low index. So it's assumed to be 0. So we'll print from slice 0 to slice 2, because again this is not inclusive, right. So slice 0 is 10, slice 1 is 15, and slice 2 is 20. And, in this example, we've omitted the high index. And what this does is it assumes that the high index is the length of our slice. And the length of our slice is 4. So it'll print form slice 2 to slice 4 minus 1, because again the high is not inclusive. So that's just slice 3. So slice 2 is 20, 0, 1, 2, and slice 3 is 25, exactly. Okay, and then we print the length and the capacity of our slice. In both cases, it's 4.

*[The output of the above lines of program is as follows: Here is our slice: slice == [10 15 20 25] slice[1:4] == [15 20 25] slice[:3] == [10 15 20] slice[2:] == [20 25] len slice == 4 cap slice == 4 The new values in our slice: slice == [5 10 15 20] len slice 4 cap slice == 4 Now we'll append 2 values to slice (what happens?): slice == [5 10 15 20 10 20] len slice 6 cap slice == 8 Now we'll append 8 more values to slice (now what happens?): slice == [5 10 15 20 10 20 1 1 1 1 1 1 1 1] len slice 14 cap slice == 16 Now we'll append 8 more values to slice (guess what happens?): slice == [5 10 15 20 10 20 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1] len slice 22 cap slice == 32 Make a copy, only the copy has a cap of 8: sliceCopy == [5 10 15 20 10 20 1 1] len slice 8 cap slice == 8 Let's slice our slice for a reslice: reslice == [10 15 20 10] len reslice 4 cap reslice == 31]*

Okay, so now again, in this part, we're modifying or manipulating the values of the slice. So, in this part, the new values of our slice – all it's doing in there is subtracting 5 from each element – so 10 minus 5 is 5, 15 minus 5 is 10, and so on. The length and the capacity don't change at all. In the next example, we append two values to the slice and see what happens. Okay, so the length of the slice – the new length – is 6 because we added the values 10 and 20. And we can see that here. Slice, it was append 10 and 20. Okay, now what happened to the capacity though? Well, the capacity doubled. And let's see if a pattern emerges here, okay. So, in the next part, we append eight more values by making a call to resize, which just iterates through this loop and adds one element to it with the value of 1 each time through the loop for a total of eight, right. So it appends eight values. So the first time it appended eight values, that would make it a length of 8 plus 6, which is 14. And the original capacity of 8 was not large enough to handle it. So again, look at that, it doubled to 16. Likewise, down here when we added eight more, we went from a length of 14 to 22. Again, 16 is not large enough, so it doubled it. So there is our pattern. So you see the capacity doubles each time using the append function when the underlying array doesn't have enough capacity.

Okay, so now when we made a copy, basically what we did was we only copied the value of our slice to slice copy. But the slice copy only had a length of 8. So what is going to happen here is that the copy copies only the first eight elements in the array. So you see the first eight elements here – 5, 10, 15, 20 up to the last two ones. Again, we see the same thing here – 5, 10, 15, 20 up to those two ones. So we see the length of the copy is 8. And we see its capacity is 8. Now, when we took a slice of our slice, we told it to slice from index 1 to index 5 minus 1, which is 4. So index 1 is 10, 2 is 15, 3 is 20, and 4 is 10. So we did that and it gave us a length of 4. So look at our capacity though. The capacity of our reslice is 31 and that's because we started at slice 1. So that would be the new slice base. So that would be slice 0 of the reslice. So what that means is that we have a capacity of 31 or 32 minus 1, which is 31. Okay. So, in this exercise, we practiced our knowledge of slices by using them in a Go program.

[Back to top](https://cdnlibrary.skillport.com/courseware/Content/cca/sd_gopf_a01_it_enus/output/html/course_transcript.html#top)

© 2020 Skillsoft Ireland Limited