To everyone who participated

in publishing this book

Table of Contents

INTRODUCTION TO BOOK

Go is a general-purpose language designed with systems programming in mind. It was initially developed at Google in the year **2007** by **Robert Griesemer**, **Rob Pike**, and **Ken Thompson**. It is strongly and statically typed, provides inbuilt support for garbage collection, and supports concurrent programming.

Programs are constructed using packages, for efficient management of dependencies. Go programming implementations use a traditional compile and link model to generate executable binaries. The Go programming language was announced in November 2009 and is used in some of the Google's production systems.

Go or Golang is a programming language created at Google, by Google developers and other programmers. This programming language is free and open-source and currently being maintained by Google. One of the founding members of Go is Ken Thompson who is best known for his work on Unix operating system development. Go compiler was initially written in C but now, it is written in Go itself, making it self-hosted.

Ruby was created by Yukihiro Matsumoto, or "Matz", in Japan in the mid 1990's. It was designed for programmer productivity with the idea that programming should be fun for programmers. It emphasizes the necessity for software to be understood by humans first and computers second.

Ruby continues to gain popularity for its use in web application development. The Ruby on Rails framework, built with the Ruby language by **David Heinemeier Hansson**, introduced many people to the joys of programming in Ruby. Ruby has a vibrant community that is supportive for beginners and enthusiastic about producing high-quality code.

GO PROGRAMMING LANGUAGE

Go - Overview

Go is a general-purpose language designed with systems programming in mind. It was initially developed at Google in the year **2007** by **Robert Griesemer**, **Rob Pike**, and **Ken Thompson**. It is strongly and statically typed, provides inbuilt support for garbage collection, and supports concurrent programming.

Programs are constructed using packages, for efficient management of dependencies. Go programming implementations use a traditional compile and link model to generate executable binaries. The Go programming language was announced in November 2009 and is used in some of the Google's production systems.

Features of Go Programming

The most important features of Go programming are listed below −

- Support for environment adopting patterns similar to dynamic languages. For example, type inference (x := 0 is valid declaration of a variable x of type int)
- Compilation time is fast.
- Inbuilt concurrency support: lightweight processes (via go routines), channels, select statement.
- Go programs are simple, concise, and safe.
- Support for Interfaces and Type embedding.
- Production of statically linked native binaries without external dependencies.

Features Excluded Intentionally

To keep the language simple and concise, the following features commonly available in other similar languages are omitted in Go −

- Support for type inheritance
- Support for method or operator overloading
- Support for circular dependencies among packages

- Support for pointer arithmetic
- Support for assertions
- Support for generic programming

Go Programs

A **Go** program can vary in length from 3 lines to millions of lines and it should be written into one or more text files with the extension "**.go**". For example, **hello.go**.

You can use "**vi**", "**vim**" or any other **text editor** to write your Go program into a file.

Go - Environment Setup

Local Environment Setup

If you are still willing to set up your environment for Go programming language, you need the following two software available on your computer −

- A text editor
- Go compiler

Text Editor

You will require a text editor to type your programs. Examples of text editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

The name and version of text editors can vary on different operating systems. For example, Notepad is used on Windows, and vim or vi is used on Windows as well as Linux or UNIX.

The files you create with the text editor are called source files. They contain program source code. The source files for Go programs are typically named with the extension ".go".

Before starting your programming, make sure you have a text editor in place and you have enough experience to write a computer program, save it in a file, compile it, and finally execute it.

The Go Compiler

The source code written in source file is the human readable source for your program. It needs to be compiled and turned into machine language so that your CPU can actually execute the program as per the instructions given. The Go programming language compiler compiles the source code into its final executable program.

Go distribution comes as a binary installable for FreeBSD (release 8 and above), Linux, Mac OS X (Snow Leopard and above), and Windows operating systems with 32-bit (386) and 64-bit (amd64) x86 processor architectures.

Download Go Archive

Download the latest version of Go installable archive file. The following version is used in this tutorial: go1.4.windows-amd64.msi.

It is copied it into C:\>go folder.

OS	Archive name
Windows	go1.4.windows-amd64.msi
Linux	go1.4.linux-amd64.tar.gz
Mac	go1.4.darwin-amd64-osx10.8.pkg
FreeBSD	go1.4.freebsd-amd64.tar.gz

Installation on UNIX/Linux/Mac OS X, and FreeBSD

Extract the download archive into the folder **/usr/local**, creating a Go tree in **/usr/local/go**. For example −

tar -C /usr/local -xzf go1.4.linux-amd64.tar.gz

Add **/usr/local/go/bin** to the PATH environment variable.

OS	Output
Linux	export PATH = $PATH:/usr/local/go/bin
Mac	export PATH = $PATH:/usr/local/go/bin
FreeBSD	export PATH = $PATH:/usr/local/go/bin

Installation on Windows

Use the MSI file and follow the prompts to install the Go tools. By default, the installer uses the Go distribution in c:\Go. The installer should set the c:\Go\bin directory in Window's PATH environment variable. Restart any open command prompts for the change to take effect.

Verifying the Installation

Create a go file named **test.go in C:\>Go_WorkSpace**.

File: **test.go**

```
package main

import "fmt"

func main() {
   fmt.Println("Hello, World!")
}
```

Now run **test.go** to see the result –

```
C:\Go_WorkSpace>go run test.go
```

Output

```
Hello, World!
```

Go - Program Structure

Before we study the basic building blocks of Go programming language, let us first discuss the bare minimum structure of Go programs so that we can take it as a reference in subsequent tutorials.

Hello World Example

A Go program basically consists of the following parts −

- Package Declaration
- Import Packages
- Functions
- Variables
- Statements and Expressions
- Comments

Let us look at a simple code that would print the words "**Hello World**" −

```
package main

import "fmt"

func main() {
   /* This is my first sample program. */
   fmt.Println("Hello, World!")
}
```

Let us take a look at the various parts of the above program −

- The first line of the program package main defines the package name in which this program should lie. It is a mandatory statement, as Go programs run in packages. The main package is the starting point to run the program. Each package has a path and name associated with it.
- The next line import "**fmt**" is a preprocessor command which tells the Go compiler to include the files lying in the package **fmt**.
- The next line **func main()** is the main function where the program execution begins.

- The next line **/*...*/** is ignored by the compiler and it is there to add comments in the program. Comments are also represented using // similar to Java or C++ comments.
- The next line **fmt.Println(...)** is another function available in Go which causes the message "Hello, World!" to be displayed on the screen. Here **fmt** package has exported **Println** method which is used to display the message on the screen.
- Notice the capital P of **Println** method. In Go language, a name is exported if it starts with capital letter. Exported means the function or variable/constant is accessible to the importer of the respective package.

Executing a Go Program

Let us discuss how to save the source code in a file, compile it, and finally execute the program. Please follow the steps given below −

- Open a text editor and add the above-mentioned code.
- Save the file as hello.go
- Open the command prompt.
- Go to the directory where you saved the file.
- Type go run hello.go and press enter to run your code.
- If there are no errors in your code, then you will see "Hello World!" printed on the screen.

```
$ go run hello.go
Hello, World!
```

Make sure the Go compiler is in your path and that you are running it in the directory containing the source file **hello.go**.

Go - Basic Syntax

Tokens in Go

A Go program consists of various tokens. A token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following Go statement consists of six tokens –

```
fmt.Println("Hello, World!")
```

The individual tokens are –

```
fmt
.
Println
(
   "Hello, World!"
)
```

Line Separator

In a Go program, the line separator key is a statement terminator. That is, individual statements don't need a special separator like “;” in C. The Go compiler internally places “;” as the statement terminator to indicate the end of one logical entity.

For example, take a look at the following statements –

```
fmt.Println("Hello, World!")
fmt.Println("I am in Go Programming World!")
```

Comments

Comments are like helping texts in your Go program and they are ignored by the compiler. They start with /* and terminates with the characters */ as shown below –

```
/* my first program in Go */
```

You cannot have comments within comments and they do not occur within a string or character literals.

Identifiers

A Go identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore _ followed by zero or more letters, underscores, and digits (0 to 9).

identifier = letter { letter | unicode_digit }.

Go does not allow punctuation characters such as @, $, and % within identifiers. Go is a case-sensitive programming language. Thus, Manpower and manpower are two different identifiers in Go. Here are some examples of acceptable identifiers –

```
mahesh     kumar    abc    move_name    a_123
myname50   _temp    j      a23b9        retVal
```

Keywords

The following list shows the reserved words in Go. These reserved words may not be used as **constant** or **variable** or any other identifier names.

break	default	func	interface	select
case	defer	Go	map	Struct
chan	else	Goto	package	Switch
const	fallthrough	if	range	Type
continue	for	import	return	Var

Whitespace in Go

Whitespace is the term used in Go to describe blanks, tabs, newline characters, and comments. A line containing only whitespace, possibly with a comment, is known as a blank line, and a Go compiler totally ignores it.

Whitespaces separate one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins. Therefore, in the following statement –

```
var age int;
```

There must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them. On the other hand, in the following statement–

```
fruit = apples + oranges;   // get the total fruit
```

No whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish for readability purpose.

Go - Data Types

Go - Data Types

In the Go programming language, data types refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The types in Go can be classified as follows –

Sr.No.	Types and Description
1	**Boolean types** They are boolean types and consists of the two predefined constants: (a) true (b) false
2	**Numeric types** They are again arithmetic types and they represents a) integer types or b) floating point values throughout the program.
3	**String types** A string type represents the set of string values. Its value is a sequence of bytes. Strings are immutable types that is once created, it is not possible to change the contents of a string. The predeclared string type is string.
4	**Derived types** They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types f) Slice types g) Interface types h) Map types i) Channel Types

Array types and structure types are collectively referred to as **aggregate types**. The type of a function specifies the set of all functions with the same parameter and result types. We will discuss the basic types in the following section, whereas other types will be covered in the upcoming chapters.

Integer Types

The predefined architecture-independent integer types are –

Sr.No.	Types and Description
1	**uint8** Unsigned 8-bit integers (0 to 255)
2	**uint16** Unsigned 16-bit integers (0 to 65535)
3	**uint32** Unsigned 32-bit integers (0 to 4294967295)
4	**uint64** Unsigned 64-bit integers (0 to 18446744073709551615)
5	**int8** Signed 8-bit integers (-128 to 127)
6	**int16** Signed 16-bit integers (-32768 to 32767)
7	**int32** Signed 32-bit integers (-2147483648 to 2147483647)
8	**int64** Signed 64-bit integers (-9223372036854775808 to 9223372036854775807)

Floating Types

The predefined architecture-independent float types are –

Sr.No.	Types and Description
1	**float32** IEEE-754 32-bit floating-point numbers
2	**float64**

	IEEE-754 64-bit floating-point numbers
3	**complex64** Complex numbers with float32 real and imaginary parts
4	**complex128** Complex numbers with float64 real and imaginary parts

The value of an n-bit integer is n bits and is represented using two's complement arithmetic operations.

Other Numeric Types

There is also a set of numeric types with implementation-specific sizes –

Sr.No.	Types and Description
1	**byte** same as uint8
2	**rune** same as int32
3	**uint** 32 or 64 bits
4	**int** same size as uint
5	**uintptr** an unsigned integer to store the uninterpreted bits of a pointer value

Go - Variables

A **variable** is nothing but a name given to a storage area that the programs can manipulate. Each variable in Go has a specific type, which determines the size and layout of the variable's memory, the range of values that can be stored within that memory, and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because Go is case-sensitive. Based on the basic types explained in the previous chapter, there will be the following basic variable types –

Sr.No	Type & Description
1	**byte** Typically a single octet(one byte). This is an byte type.
2	**int** The most natural size of integer for the machine.
3	**float32** A single-precision floating point value.

Go programming language also allows to define various other types of variables such as **Enumeration**, **Pointer**, **Array**, **Structure**, and **Union**, which we will discuss in subsequent letters. In this chapter, we will focus only basic variable types.

Variable Definition in Go

A **variable** definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows −

```
var variable_list optional_data_type;
```

Here, **optional_data_type** is a valid Go data type including byte, int, float32, complex64, **boolean** or any user-defined object, etc., and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here –

```
var  i, j, k int;
var  c, ch byte;
var  f, salary float32;
d =  42;
```

The statement **"var i, j, k;"** declares and defines the variables i, j and k; which instructs the compiler to create variables named **i**, **j**, and **k** of type int.

Variables can be initialized (assigned an initial value) in their declaration. The type of variable is automatically judged by the compiler based on the value passed to it. The initializer consists of an equal sign followed by a constant expression as follows –

```
variable_name = value;
```

For example,

```
d = 3, f = 5;    // declaration of d and f. Here d and f are int
```

For definition without an initializer: variables with static storage duration are implicitly initialized with nil (all bytes have the value 0); the initial value of all other variables is zero value of their data type.

Static Type Declaration in Go

A **static type** variable declaration provides assurance to the compiler that there is one variable available with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail of the variable. A variable declaration has its meaning at the time of compilation only, the compiler needs the actual variable declaration at the time of linking of the program.

Example

Try the following example, where the variable has been declared with a type and initialized inside the main function −

```
package main

import "fmt"

func main() {
   var x float64
   x = 20.0
   fmt.Println(x)
   fmt.Printf("x is of type %T\n", x)
}
```

When the above code is compiled and executed, it produces the following result –

```
20
x is of type float64
```

Dynamic Type Declaration / Type Inference in Go

A **dynamic type** variable declaration requires the compiler to interpret the type of the variable based on the value passed to it. The compiler does not require a variable to have type statically as a necessary requirement.

Example

Try the following example, where the variables have been declared without any type. Notice, in case of type inference, we initialized the variable **y** with **:=** operator, whereas x is initialized using **=** operator.

```
package main

import "fmt"

func main() {
   var x float64 = 20.0

   y := 42
   fmt.Println(x)
   fmt.Println(y)
   fmt.Printf("x is of type %T\n", x)
   fmt.Printf("y is of type %T\n", y)
}
```

When the above code is compiled and executed, it produces the following result –

```
20
42
x is of type float64
y is of type int
```

Mixed Variable Declaration in Go

Variables of different types can be declared in one go using type inference.

Example

```
package main

import "fmt"

func main() {
   var a, b, c = 3, 4, "foo"

   fmt.Println(a)
   fmt.Println(b)
   fmt.Println(c)
   fmt.Printf("a is of type %T\n", a)
   fmt.Printf("b is of type %T\n", b)
   fmt.Printf("c is of type %T\n", c)
}
```

When the above code is compiled and executed, it produces the following result –

```
3
4
foo
a is of type int
b is of type int
c is of type string
```

The lvalues and the rvalues in Go

There are two kinds of expressions in Go −

- **lvalue** − Expressions that refer to a memory location is called "lvalue" expression. An lvalue may appear as either the left-hand or right-hand side of an assignment.
- **rvalue** − The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an **rvalue** may appear on the right- but not left-hand side of an assignment.

Variables are **lvalues** and so may appear on the left-hand side of an assignment. Numeric literals are **rvalues** and so may not be assigned and cannot appear on the left-hand side. The following statement is valid –

```
x = 20.0
```

The following statement is not valid. It would generate compile-time error –

```
10 = 20
```

Go – Constants

Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals.

Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are also enumeration constants as well.

Constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of **U** and **L**, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals –

```
212         /* Legal */
215u        /* Legal */
0xFeeL      /* Legal */
078         /* Illegal: 8 is not an octal digit */
032UU       /* Illegal: cannot repeat a suffix */
```

Following are other examples of various type of Integer literals –

```
85         /* decimal */
0213       /* octal */
0x4b       /* hexadecimal */
30         /* int */
30u        /* unsigned int */
30l        /* long */
30ul       /* unsigned long */
```

Floating-point Literals

A **floating-point** literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by **e** or **E**.

Here are some examples of floating-point literals –

```
3.14159       /* Legal */
314159E-5L    /* Legal */
510E          /* Illegal: incomplete exponent */
210f          /* Illegal: no decimal or exponent */
.e55          /* Illegal: missing integer or fraction */
```

Escape Sequence

When certain characters are preceded by a backslash, they will have a special meaning in Go. These are known as Escape Sequence codes which are used to represent newline (**\n**), tab (**\t**), backspace, etc. Here, you have a list of some of such escape sequence codes –

Escape sequence	Meaning
\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits

\xhh . . .	Hexadecimal number of one or more digits

The following example shows how to use \t in a program –

```
package main

import "fmt"

func main() {
   fmt.Printf("Hello\tWorld!")
}
```

When the above code is compiled and executed, it produces the following result –

```
Hello World!
```

String Literals in Go

String literals or constants are enclosed in double quotes "". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separating them using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"
```

The const Keyword

You can use const prefix to declare constants with a specific type as follows –

```
const variable type = value;
```

The following example shows how to use the const keyword –

```
package main

import "fmt"

func main() {
   const LENGTH int = 10
   const WIDTH int = 5
   var area int

   area = LENGTH * WIDTH
   fmt.Printf("value of area : %d", area)
}
```

When the above code is compiled and executed, it produces the following result –

```
value of area : 50
```

Note that it is a good programming practice to define constants in CAPITALS.

Go – Operators

An **operator** is a symbol that tells the compiler to perform specific mathematical or logical manipulations. Go language is rich in built-in operators and provides the following types of operators −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Miscellaneous Operators

This tutorial explains arithmetic, relational, logical, bitwise, assignment, and other operators one by one.

Arithmetic Operators

Following table shows all the arithmetic operators supported by Go language. Assume variable A holds 10 and variable B holds 20 then –

Operator	Description	Example
+	Adds two operands	A + B gives 30
-	Subtracts second operand from the first	A - B gives -10
*	Multiplies both operands	A * B gives 200
/	Divides the numerator by the denominator.	B / A gives 2
%	Modulus operator; gives the remainder after an integer division.	B % A gives 0
++	Increment operator. It increases the integer value by one.	A++ gives 11
--	Decrement operator. It decreases the integer value by one.	A-- gives 9

Relational Operators

The following table lists all the relational operators supported by Go language. Assume variable A holds 10 and variable B holds 20, then −

Operator	Description	Example
==	It checks if the values of two operands are equal or not; if yes, the condition becomes true.	(A == B) is not true.
!=	It checks if the values of two operands are equal or not; if the values are not equal, then the condition becomes true.	(A != B) is true.
>	It checks if the value of left operand is greater than the value of right operand; if yes, the condition becomes true.	(A > B) is not true.
<	It checks if the value of left operand is less than the value of the right operand; if yes, the condition becomes true.	(A < B) is true.
>=	It checks if the value of the left operand is greater than or equal to the value of the right operand; if yes, the condition becomes true.	(A >= B) is not true.
<=	It checks if the value of left operand is less than or equal to the value of right operand; if yes, the condition becomes true.	(A <= B) is true.

Logical Operators

The following table lists all the logical operators supported by Go language. Assume variable A holds 1 and variable B holds 0, then –

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.

!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

The following table shows all the logical operators supported by Go language. Assume variable **A** holds true and variable **B** holds false, then –

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are false, then the condition becomes false.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is true, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Bitwise Operators

Bitwise operators work on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows –

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume A = 60; and B = 13. In binary format, they will be as follows −

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The **Bitwise** operators supported by C language are listed in the following table. Assume variable **A** holds **60** and variable **B** holds **13**, then −

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Assignment Operators

The following table lists all the assignment operators supported by Go language –

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Miscellaneous Operators

There are a few other important operators supported by Go Language including sizeof and ?:.

Operator	Description	Example
&	Returns the address of a variable.	&a; provides actual address of the variable.
*	Pointer to a variable.	*a; provides pointer to a variable.

Operators Precedence in Go

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

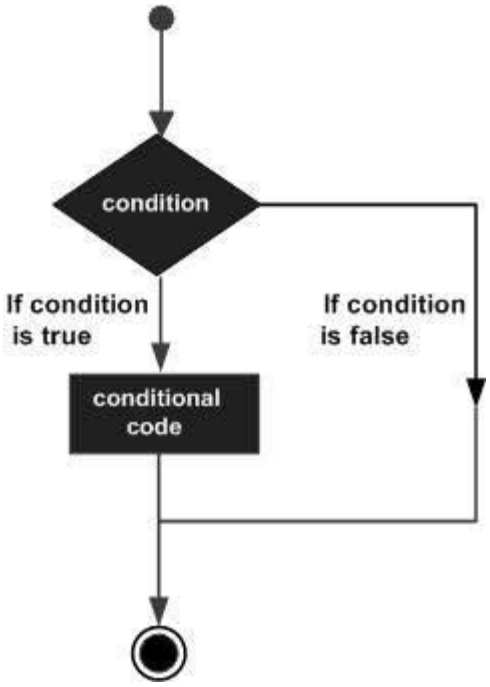
Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right

Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Go - Decision Making

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision-making structure found in most of the programming languages –



Go programming language provides the following types of decision-making statements. Click the following links to check their detail.

Sr.No	Statement & Description
1	if statement An **if statement** consists of a boolean expression followed by one or more statements.
2	if...else statement An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false.

3	nested if statements You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
4	switch statement A **switch** statement allows a variable to be tested for equality against a list of values.
5	select statement A **select** statement is similar to **switch** statement with difference that case statements refers to channel communications.

Go - if statement

An **if statement** consists of a Boolean expression followed by one or more statements.

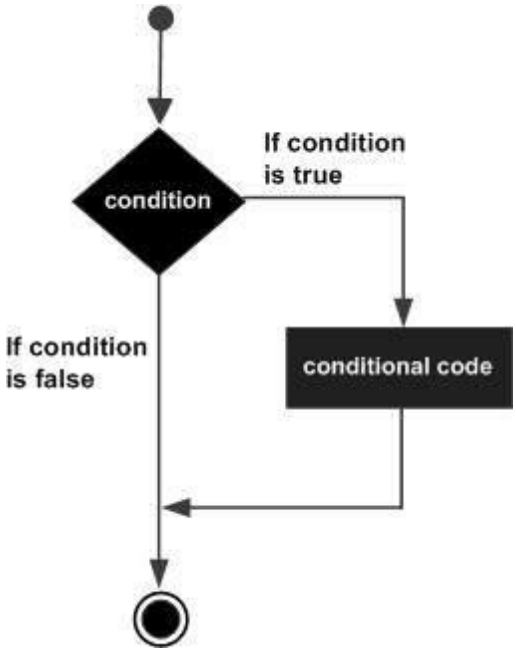
Syntax

The **syntax** of an if statement in Go programming language is –

```
if(boolean_expression) {
   /* statement(s) will execute if the boolean expression is true */
}
```

If the **Boolean** expression evaluates to true, then the block of code inside the if statement is executed. If **Boolean** expression evaluates to false, then the first set of code after the end of the if statement (after the closing curly brace) is executed.

Flow Diagram



Example

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 10

   /* check the boolean condition using if statement */
   if( a < 20 ) {
      /* if condition is true then print the following */
      fmt.Printf("a is less than 20\n" )
   }
   fmt.Printf("value of a is : %d\n", a)
}
```

When the above code is compiled and executed, it produces the following result –

```
a is less than 20;
value of a is : 10
```

Go - if...else statement

An **if statement** can be followed by an optional else statement, which executes when the Boolean expression is false.

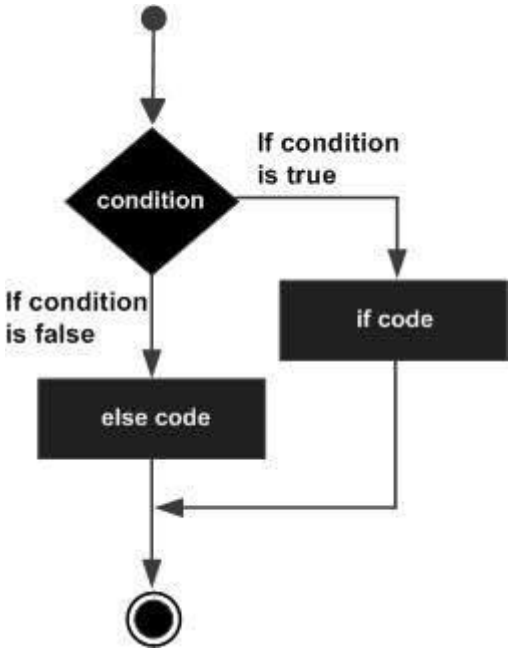
Syntax

The **syntax** of an if...else statement in Go programming language is –

```
if(boolean_expression) {
   /* statement(s) will execute if the boolean expression is true */
} else {
   /* statement(s) will execute if the boolean expression is false */
}
```

If the Boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code is executed.

Flow Diagram



Example

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 100;

   /* check the boolean condition */
   if( a < 20 ) {
      /* if condition is true then print the following */
      fmt.Printf("a is less than 20\n" );
   } else {
      /* if condition is false then print the following */
      fmt.Printf("a is not less than 20\n" );
   }
   fmt.Printf("value of a is : %d\n", a);
}
```

When the above code is compiled and executed, it produces the following result −

```
a is not less than 20;
value of a is : 100
```

The if...else if...else Statement

An **if statement** can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind −

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax

The syntax of an if...else if...else statement in Go programming language is −

```
if(boolean_expression 1) {
   /* Executes when the boolean expression 1 is true */
} else if( boolean_expression 2) {
   /* Executes when the boolean expression 2 is true */
} else if( boolean_expression 3) {
   /* Executes when the boolean expression 3 is true */
} else {
   /* executes when the none of the above condition is true */
}
```

Example

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 100

   /* check the boolean condition */
   if( a == 10 ) {
      /* if condition is true then print the following */
      fmt.Printf("Value of a is 10\n" )
   } else if( a == 20 ) {
      /* if else if condition is true */
      fmt.Printf("Value of a is 20\n" )
   } else if( a == 30 ) {
      /* if else if condition is true  */
      fmt.Printf("Value of a is 30\n" )
   } else {
      /* if none of the conditions is true */
      fmt.Printf("None of the values is matching\n" )
   }
   fmt.Printf("Exact value of a is: %d\n", a )
}
```

When the above code is compiled and executed, it produces the following result –

```
None of the values is matching
Exact value of a is: 100
```

Go - nested if statements

It is always legal in Go programming to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

Syntax

The syntax for a nested if statement is as follows −

```
if( boolean_expression 1) {
   /* Executes when the boolean expression 1 is true */
   if(boolean_expression 2) {
      /* Executes when the boolean expression 2 is true */
   }
}
```

You can **nest else if...else** in the similar way as you have nested if statement.

Example

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 100
   var b int = 200

   /* check the boolean condition */
   if( a == 100 ) {
      /* if condition is true then check the following */
      if( b == 200 )  {
         /* if condition is true then print the following */
         fmt.Printf("Value of a is 100 and b is 200\n" );
      }
   }
   fmt.Printf("Exact value of a is : %d\n", a );
   fmt.Printf("Exact value of b is : %d\n", b );
}
```

When the above code is compiled and executed, it produces the following result –

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

Go - The Switch Statement

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.

In Go programming, switch statements are of two types −

Expression Switch − In expression switch, a case contains expressions, which is compared against the value of the switch expression.

Type Switch − In type switch, a case contain type which is compared against the type of a specially annotated switch expression.

Expression Switch

The syntax for expression switch statement in Go programming language is as follows –

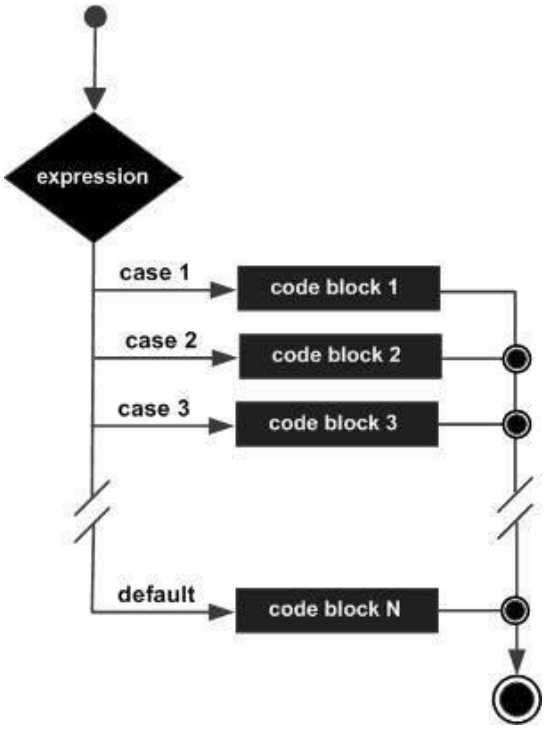
```
switch(boolean-expression or integral type){
   case boolean-expression or integral type :
      statement(s);
   case boolean-expression or integral type :
      statement(s);

   /* you can have any number of case statements */
   default : /* Optional */
      statement(s);
}
```

The following rules apply to a switch statement −

- The expression used in a switch statement must have an integral or boolean expression, or be of a class type in which the class has a single conversion function to an integral or boolean value. If the expression is not passed then the default value is true.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute. No break is needed in the case statement.

- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Flow Diagram



Example

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var grade string = "B"
   var marks int = 90

   switch marks {
      case 90: grade = "A"
      case 80: grade = "B"
      case 50,60,70 : grade = "C"
      default: grade = "D"
   }
   switch {
      case grade == "A" :
         fmt.Printf("Excellent!\n" )
      case grade == "B", grade == "C" :
         fmt.Printf("Well done\n" )
      case grade == "D" :
         fmt.Printf("You passed\n" )
      case grade == "F":
         fmt.Printf("Better try again\n" )
      default:
         fmt.Printf("Invalid grade\n" );
   }
   fmt.Printf("Your grade is  %s\n", grade );
}
```

When the above code is compiled and executed, it produces the following result –

```
Excellent!
Your grade is  A
```

Type Switch

The syntax for a type switch statement in Go programming is as follows –

```
switch x.(type){
   case type:
      statement(s);
   case type:
      statement(s);
   /* you can have any number of case statements */
   default: /* Optional */
      statement(s);
}
```

The following rules apply to a switch statement −

- The expression used in a switch statement must have an variable of interface{} type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The type for a case must be the same data type as the variable in the switch, and it must be a valid data type.
- When the variable being switched on is equal to a case, the statements following that case will execute. No break is needed in the case statement.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example

```
package main

import "fmt"

func main() {
   var x interface{}

   switch i := x.(type) {
      case nil:
         fmt.Printf("type of x :%T",i)
      case int:
         fmt.Printf("x is int")
      case float64:
         fmt.Printf("x is float64")
      case func(int) float64:
         fmt.Printf("x is func(int)")
      case bool, string:
         fmt.Printf("x is bool or string")
      default:
         fmt.Printf("don't know the type")
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
type of x :<nil>
```

Go - The Select Statement

The syntax for a select statement in Go programming language is as follows –

```
select {
   case communication clause  :
      statement(s);
   case communication clause  :
      statement(s);
   /* you can have any number of case statements */
   default : /* Optional */
      statement(s);
}
```

The following rules apply to a select statement −

- You can have any number of case statements within a select. Each case is followed by the value to be compared to and a colon.
- The type for a case must be the a communication channel operation.
- When the channel operation occured the statements following that case will execute. No break is needed in the case statement.
- A select statement can have an optional default case, which must appear at the end of the select. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example

```
package main

import "fmt"

func main() {
   var c1, c2, c3 chan int
   var i1, i2 int
   select {
      case i1 = <-c1:
         fmt.Printf("received ", i1, " from c1\n")
      case c2 <- i2:
         fmt.Printf("sent ", i2, " to c2\n")
      case i3, ok := (<-c3):  // same as: i3, ok := <-c3
         if ok {
            fmt.Printf("received ", i3, " from c3\n")
         } else {
            fmt.Printf("c3 is closed\n")
         }
      default:
         fmt.Printf("no communication\n")
   }
}
```

When the above code is compiled and executed, it produces the following result −

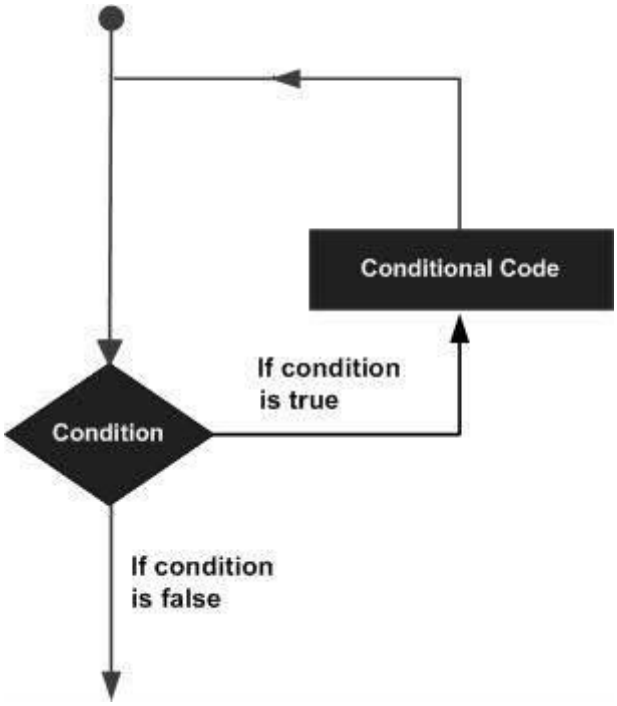
```
no communication
```

Go – Loops

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages –



Go programming language provides the following types of loop to handle looping requirements.

Sr.No	Loop Type & Description
1	for loop It executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

2	nested loops These are one or multiple loops inside any for loop.

Loop Control Statements

Loop control statements change an execution from its normal sequence. When an execution leaves its scope, all automatic objects that were created in that scope are destroyed.

Go supports the following control statements –

Sr.No	Control Statement & Description
1	break statement It terminates a **for loop** or **switch** statement and transfers execution to the statement immediately following the for loop or switch.
2	continue statement It causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	goto statement It transfers control to the labeled statement.

The Infinite Loop

A loop becomes an infinite loop if its condition never becomes false. The for loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty or by passing true to it.

```
package main

import "fmt"

func main() {
   for true  {
       fmt.Printf("This loop will run forever.\n");
   }
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(;;) construct to signify an infinite loop.

Note − You can terminate an infinite loop by pressing Ctrl + C keys.

Go - for Loop

A **for loop** is a repetition control structure. It allows you to write a loop that needs to execute a specific number of times.

Syntax

The syntax of for loop in Go programming language is –

```
for [condition |  ( init; condition; increment ) | Range] {
   statement(s);
}
```

The flow of control in a for loop is a follows −

If a condition is available, then for loop executes as long as condition is true.

If a for clause that is (init; condition; increment) is present then −

The init step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

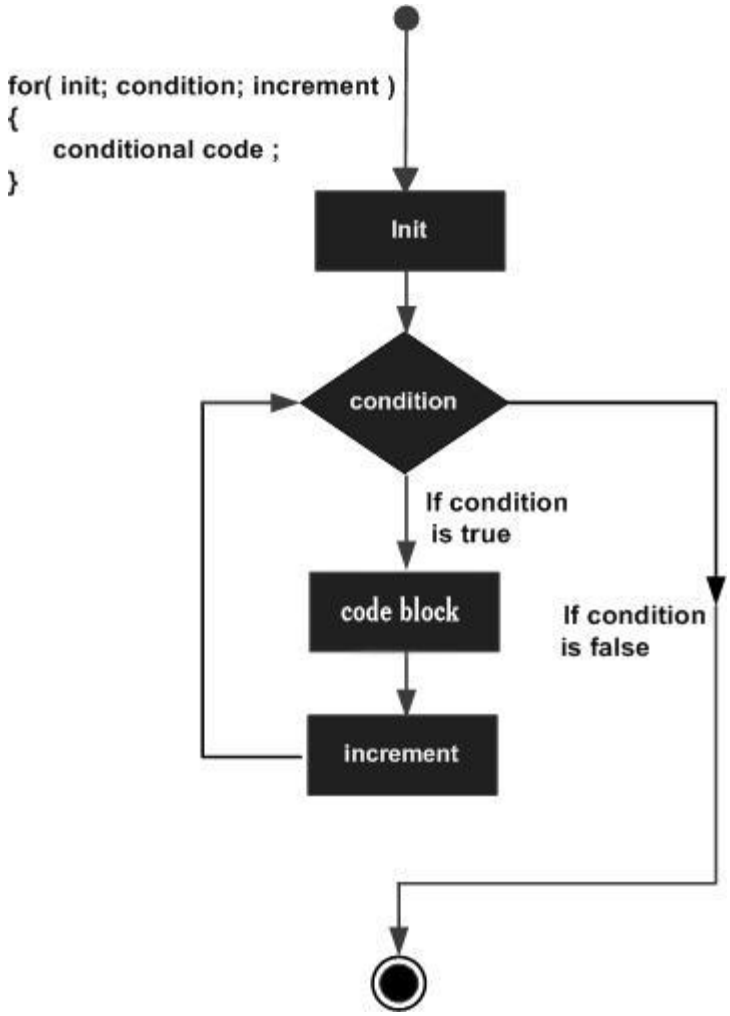
Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the for loop.

After the body of the for loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again, the condition). After the condition becomes false, the for loop terminates.

If range is available, then the for loop executes for each item in the range.

Flow Diagram


for(init; condition; increment)
{
conditional code ;
}
Init
condition
If condition
is true
code block
If condition
is false
increment

Example

```go
package main

import "fmt"

func main() {
   var b int = 15
   var a int
   numbers := [6]int{1, 2, 3, 5}

   /* for loop execution */
   for a := 0; a < 10; a++ {
      fmt.Printf("value of a: %d\n", a)
   }
   for a < b {
      a++
      fmt.Printf("value of a: %d\n", a)
   }
   for i,x:= range numbers {
      fmt.Printf("value of x = %d at %d\n", x,i)
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
value of a: 0
value of a: 1
value of a: 2
value of a: 3
value of a: 4
value of a: 5
value of a: 6
value of a: 7
value of a: 8
value of a: 9
value of a: 1
value of a: 2
value of a: 3
value of a: 4
value of a: 5
value of a: 6
value of a: 7
value of a: 8
value of a: 9
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of x = 1 at 0
value of x = 2 at 1
value of x = 3 at 2
value of x = 5 at 3
value of x = 0 at 4
value of x = 0 at 5
```

Go - Nested for Loops

Go programming language allows to use one loop inside another loop. The following section shows a few examples to illustrate the concept −

Syntax

The syntax for a nested for loop statement in Go is as follows –

```
for [condition |  ( init; condition; increment ) | Range] {
   for [condition |  ( init; condition; increment ) | Range] {
      statement(s);
   }
   statement(s);
}
```

Example

The following program uses a nested for loop to find the prime numbers from 2 to 100 –

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var i, j int

   for i = 2; i < 100; i++ {
      for j = 2; j <= (i/j); j++ {
         if(i%j==0) {
            break; // if factor found, not prime
         }
      }
      if(j > (i/j)) {
         fmt.Printf("%d is prime\n", i);
      }
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

Go - break statement

The break statement in Go programming language has the following two usages −

- When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- It can be used to terminate a case in a switch statement.

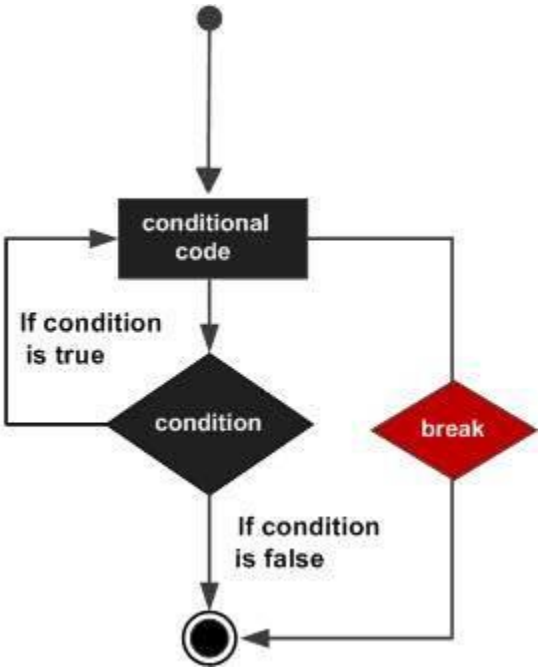
If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for a break statement in Go is as follows –

```
break;
```

Flow Diagram



Example

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 10

   /* for loop execution */
   for a < 20 {
      fmt.Printf("value of a: %d\n", a);
      a++;
      if a > 15 {
         /* terminate the loop using break statement */
         break;
      }
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

Go - The continue Statement

The **continue** statement in Go programming language works somewhat like a break statement. Instead of forcing termination, a continue statement forces the next iteration of the loop to take place, skipping any code in between.

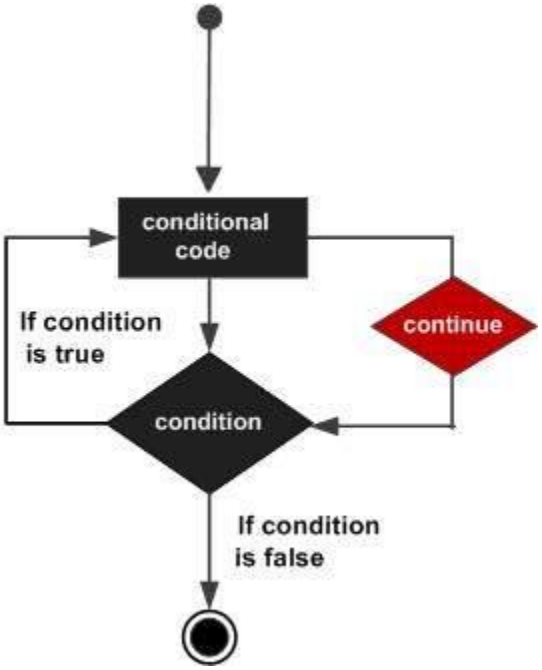
In case of the for loop, continue statement causes the conditional test and increment portions of the loop to execute.

Syntax

The syntax for a continue statement in Go is as follows –

```
continue;
```

Flow Diagram



Example

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 10

   /* do loop execution */
   for a < 20 {
      if a == 15 {
         /* skip the iteration */
         a = a + 1;
         continue;
      }
      fmt.Printf("value of a: %d\n", a);
      a++;
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Go - The goto Statement

A **goto** statement in Go programming language provides an unconditional jump from the **goto** to a labeled statement in the same function.

Note − Use of **goto** statement is highly discouraged in any programming language because it becomes difficult to trace the control flow of a program, making the program difficult to understand and hard to modify. Any program that uses a goto can be rewritten using some other construct.

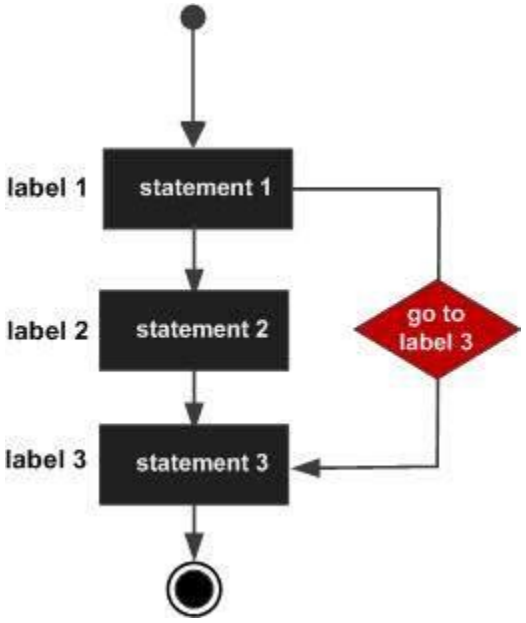
Syntax

The syntax for a **goto** statement in Go is as follow

```
goto label;
..
.
label: statement;
```

Here, **label** can be any plain text except Go keyword and it can be set anywhere in the Go program above or below to **goto** statement.

Flow Diagram



Example

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 10

   /* do loop execution */
   LOOP: for a < 20 {
      if a == 15 {
         /* skip the iteration */
         a = a + 1
         goto LOOP
      }
      fmt.Printf("value of a: %d\n", a)
      a++
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Go – Functions

A function is a group of statements that together perform a task. Every Go program has at least one function, which is main(). You can divide your code into separate functions. How you divide your code among different functions is up to you, but logically, the division should be such that each function performs a specific task.

A function **declaration** tells the compiler about a function name, return type, and parameters. A function **definition** provides the actual body of the function.

The Go standard library provides numerous built-in functions that your program can call. For example, the function **len()** takes arguments of various types and returns the length of the type. If a string is passed to it, the function returns the length of the string in bytes. If an array is passed to it, the function returns the length of the array.

Functions are also known as method, sub-routine, or procedure.

Defining a Function

The general form of a function definition in Go programming language is as follows –

```
func function_name( [parameter list] ) [return_types]
{
   body of the function
}
```

A function definition in Go programming language consists of a function header and a function body. Here are all the parts of a function −

- **Func** − It starts the declaration of a function.
- **Function Name** − It is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

- **Return Type** − A function may return a list of values. The **return_types** is the list of data types of the values the function returns. Some functions perform the desired operations without returning a value. In this case, the **return_type** is the not required.
- **Function Body** − It contains a collection of statements that define what the function does.

Example

The following source code shows a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum between the two –

```
/* function returning the max between two numbers */
func max(num1, num2 int) int {
   /* local variable declaration */
   result int

   if (num1 > num2) {
      result = num1
   } else {
      result = num2
   }
   return result
}
```

Calling a Function

While creating a Go function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with its function name. If the function returns a value, then you can store the returned value. For example –

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 100
   var b int = 200
   var ret int

   /* calling a function to get max value */
   ret = max(a, b)

   fmt.Printf( "Max value is : %d\n", ret )
}

/* function returning the max between two numbers */
func max(num1, num2 int) int {
   /* local variable declaration */
   var result int

   if (num1 > num2) {
      result = num1
   } else {
      result = num2
   }
   return result
}
```

We have kept the **max()** function along with the **main()** function and compiled the source code. While running the final executable, it would produce the following result –

```
Max value is : 200
```

Returning multiple values from Function

A Go function can return multiple values. For example −

```
package main

import "fmt"

func swap(x, y string) (string, string) {
   return y, x
}
func main() {
   a, b := swap("Mahesh", "Kumar")
   fmt.Println(a, b)
}
```

When the above code is compiled and executed, it produces the following result −

```
Kumar Mahesh
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function –

Sr.No	Call Type & Description
1	Call by value This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	Call by reference This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, Go uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. The above program, while calling the **max()** function, used the same method.

Function Usage

A function can be used in the following ways:

Sr.No	Function Usage & Description
1	Function as Value Functions can be created on the fly and can be used as values.
2	Function Closures Functions closures are anonymous functions and can be used in dynamic programming.
3	Method Methods are special functions with a receiver.

○ Go - Call by value

The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, **Go** programming language uses call by value method to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

```
/* function definition to swap the values */
func swap(int x, int y) int {
   var temp int

   temp = x /* save the value of x */
   x = y    /* put y into x */
   y = temp /* put temp into y */

   return temp;
}
```

Now, let us call the function **swap()** by passing actual values as in the following example –

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 100
   var b int = 200

   fmt.Printf("Before swap, value of a : %d\n", a )
   fmt.Printf("Before swap, value of b : %d\n", b )

   /* calling a function to swap the values */
   swap(a, b)

   fmt.Printf("After swap, value of a : %d\n", a )
   fmt.Printf("After swap, value of b : %d\n", b )
}
func swap(x, y int) int {
   var temp int

   temp = x /* save the value of x */
   x = y    /* put y into x */
   y = temp /* put temp into y */

   return temp;
}
```

Put the above code in a single C file, and then compile and execute it. It will produce the following result –

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

It shows that there is no change in the values though they had been changed inside the function.

○ Go - Call by reference

The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument pointers are passed to the functions just like any other value. Accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.

```
/* function definition to swap the values */
func swap(x *int, y *int) {
   var temp int
   temp = *x    /* save the value at address x */
   *x = *y      /* put y into x */
   *y = temp    /* put temp into y */
}
```

To learn more about pointers in Go programming, you will take it in **Go - Pointers**.

For now, let us call the function swap() by passing values by reference as in the following example –

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 100
   var b int = 200

   fmt.Printf("Before swap, value of a : %d\n", a )
   fmt.Printf("Before swap, value of b : %d\n", b )

   /* calling a function to swap the values.
   * &a indicates pointer to a ie. address of variable a and
   * &b indicates pointer to b ie. address of variable b.
   */
   swap(&a, &b)

   fmt.Printf("After swap, value of a : %d\n", a )
   fmt.Printf("After swap, value of b : %d\n", b )
}
func swap(x *int, y *int) {
   var temp int
   temp = *x    /* save the value at address x */
   *x = *y    /* put y into x */
   *y = temp    /* put temp into y */
}
```

Put the above code in a single Go file, and then compile and execute it. It produces the following result –

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

It shows that the change has reflected outside the function as well, unlike call by value where the changes do not reflect outside the function.

Go - Scope Rules

A **scope** in any programming is a region of the program where a defined variable can exist and beyond that the variable cannot be accessed. There are three places where variables can be declared in Go programming language −

- Inside a function or a block (local variables)
- Outside of all functions (global variables)
- In the definition of function parameters (formal parameters)

Let us find out what are local and global variables and what are formal parameters.

Local Variables

Variables that are declared inside a function or a block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example uses local variables. Here all the variables a, b, and c are local to the **main()** function.

```
package main

import "fmt"

func main() {
   /* local variable declaration */
   var a, b, c int

   /* actual initialization */
   a = 10
   b = 20
   c = a + b

   fmt.Printf ("value of a = %d, b = %d and c = %d\n", a, b, c)
}
```

When the above code is compiled and executed, it produces the following result −

```
value of a = 10, b = 20 and c = 30
```

Global Variables

Global variables are defined outside of a function, usually on top of the program. Global variables hold their value throughout the lifetime of the program and they can be accessed inside any of the functions defined for the program.

A **global variable** can be accessed by any function. That is, a global variable is available for use throughout the program after its declaration. The following example uses both global and local variables −

```
package main

import "fmt"

/* global variable declaration */
var g int

func main() {
   /* local variable declaration */
   var a, b int

   /* actual initialization */
   a = 10
   b = 20
   g = a + b

   fmt.Printf("value of a = %d, b = %d and g = %d\n", a, b, g)
}
```

When the above code is compiled and executed, it produces the following result −

```
value of a = 10, b = 20 and g = 30
```

A program can have the same name for local and global variables but the value of the local variable inside a function takes preference. For example −

```
package main

import "fmt"

/* global variable declaration */
var g int = 20

func main() {
   /* local variable declaration */
   var g int = 10

   fmt.Printf ("value of g = %d\n",  g)
}
```

When the above code is compiled and executed, it produces the following result −

```
value of g = 10
```

Formal Parameters

Formal parameters are treated as local variables with-in that function and they take preference over the global variables. For example –

```
package main

import "fmt"

/* global variable declaration */
var a int = 20;

func main() {
   /* local variable declaration in main function */
   var a int = 10
   var b int = 20
   var c int = 0

   fmt.Printf("value of a in main() = %d\n",  a);
   c = sum( a, b);
   fmt.Printf("value of c in main() = %d\n",  c);
}
/* function to add two integers */
func sum(a, b int) int {
   fmt.Printf("value of a in sum() = %d\n",  a);
   fmt.Printf("value of b in sum() = %d\n",  b);

   return a + b;
}
```

When the above code is compiled and executed, it produces the following result −

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

Initializing Local and Global Variables

Local and **global** variables are initialized to their default value, which is **0**; while pointers are initialized to nil.

Data Type	Initial Default Value
int	0
float32	0
pointer	nil

Go – Strings

Strings, which are widely used in Go programming, are a readonly slice of bytes. In the Go programming language, strings are slices. The Go platform provides various libraries to manipulate strings.

- unicode
- regexp
- strings

Creating Strings

The most direct way to create a string is to write −

```
var greeting = "Hello world!"
```

Whenever it encounters a string literal in your code, the compiler creates a string object with its value in this case, "Hello world!'.

A **string** literal holds a valid **UTF-8** sequences called runes. A String holds arbitrary bytes.

```
package main

import "fmt"

func main() {
  var greeting =  "Hello world!"

  fmt.Printf("normal string: ")
  fmt.Printf("%s", greeting)
  fmt.Printf("\n")
  fmt.Printf("hex bytes: ")

  for i := 0; i < len(greeting); i++ {
     fmt.Printf("%x ", greeting[i])
  }

  fmt.Printf("\n")
  const sampleText = "\xbd\xb2\x3d\xbc\x20\xe2\x8c\x98"
```

```
  /*q flag escapes unprintable characters, with + flag it escapses non-ascii
  characters as well to make output unambigous */
  fmt.Printf("quoted string: ")
  fmt.Printf("%+q", sampleText)
  fmt.Printf("\n")
}
```

This would produce the following result −

```
normal string: Hello world!
hex bytes: 48 65 6c 6c 6f 20 77 6f 72 6c 64 21
quoted string: "\xbd\xb2=\xbc \u2318"
```

Note − The string literal is immutable, so that once it is created a string literal cannot be changed.

String Length

len(str) method returns the number of bytes contained in the string literal.

```
package main

import "fmt"

func main() {
   var greeting =  "Hello world!"

   fmt.Printf("String Length is: ")
   fmt.Println(len(greeting))
}
```

This would produce the following result –

```
String Length is : 12
```

Concatenating Strings

The strings package includes a method join for concatenating multiple strings –

```
strings.Join(sample, " ")
```

Join concatenates the elements of an array to create a single string. Second parameter is seperator which is placed between element of the array.

Let us look at the following example –

```
package main

import ("fmt" "math" )"fmt" "strings")

func main() {
   greetings :=  []string{"Hello","world!"}
   fmt.Println(strings.Join(greetings, " "))
}
```

This would produce the following result −

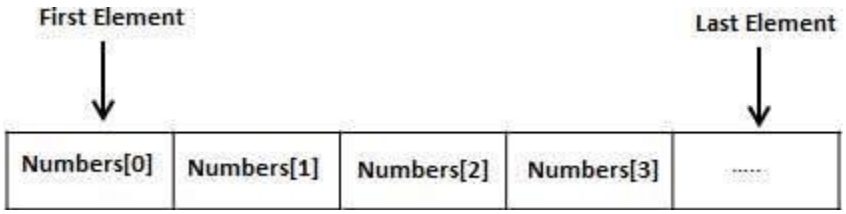
```
Hello world!
```

Go - Arrays

Go programming language provides a data structure called the array, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use **numbers[0]**, **numbers[1]**, and ..., **numbers[99]** to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in Go, a programmer specifies the type of the elements and the number of elements required by an array as follows –

```
var variable_name [SIZE] variable_type
```

This is called a single-dimensional array. The **arraySize** must be an integer constant greater than zero and type can be any valid Go data type. For example, to declare a 10-element array called balance of type float32, use this statement –

```
var balance [10] float32
```

Here, balance is a variable array that can hold up to 10 float numbers.

Initializing Arrays

You can initialize array in Go either one by one or using a single statement as follows –

```
var balance = [5]float32{1000.0, 2.0, 3.4, 7.0, 50.0}
```

The number of values between braces **{ }** can not be larger than the number of elements that we declare for the array between square brackets **[ ]**.

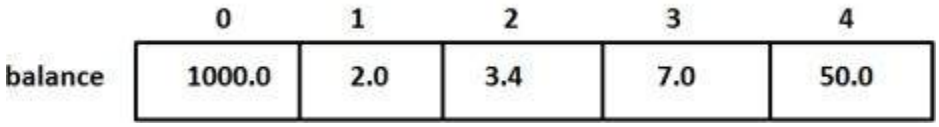
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write –

```
var balance = []float32{1000.0, 2.0, 3.4, 7.0, 50.0}
```

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array –

```
balance[4] = 50.0
```

The above statement assigns element number 5th in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called base index and last index of an array will be total size of the array minus 1. Following is the pictorial representation of the same array we discussed above –



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example –

```
float32 salary = balance[9]
```

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays –

```
package main

import "fmt"

func main() {
   var n [10]int /* n is an array of 10 integers */
   var i,j int

   /* initialize elements of array n to 0 */
   for i = 0; i < 10; i++ {
      n[i] = i + 100 /* set element at location i to i + 100 */
   }

   /* output each array element's value */
   for j = 0; j < 10; j++ {
      fmt.Printf("Element[%d] = %d\n", j, n[j] )
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

Go Arrays in Detail

There are important concepts related to array which should be clear to a Go programmer

Sr.No	Concept & Description
1	Multi-dimensional arrays Go supports multidimensional arrays. The simplest form of a multidimensional array is the two-dimensional array.

2	Passing arrays to functions You can pass to the function a pointer to an array by specifying the array's name without an index.

Go – Pointers

Pointers in Go are easy and fun to learn. Some Go programming tasks are performed more easily with pointers, and other tasks, such as call by reference, cannot be performed without using pointers. So, it becomes necessary to learn pointers to become a perfect Go programmer.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which will print the address of the variables defined –

```
package main

import "fmt"

func main() {
   var a int = 10
   fmt.Printf("Address of a variable: %x\n", &a  )
}
```

When the above code is compiled and executed, it produces the following result –

```
Address of a variable: 10328000
```

So you understood what is memory address and how to access it. Now let us see what pointers are.

What Are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is –

```
var var_name *var-type
```

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk * you used to declare a pointer is the same

asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration –

```
var ip *int        /* pointer to an integer */
var fp *float32    /* pointer to a float */
```

The actual data type of the value of all pointers, whether integer, float, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to Use Pointers?

There are a few important operations, which we frequently perform with pointers: (a) we define pointer variables, (b) assign the address of a variable to a pointer, and (c) access the value at the address stored in the pointer variable.

All these operations are carried out using the unary operator * that returns the value of the variable located at the address specified by its operand. The following example demonstrates how to perform these operations −

```
package main

import "fmt"

func main() {
   var a int = 20   /* actual variable declaration */
   var ip *int      /* pointer variable declaration */

   ip = &a  /* store address of a in pointer variable*/

   fmt.Printf("Address of a variable: %x\n", &a  )

   /* address stored in pointer variable */
   fmt.Printf("Address stored in ip variable: %x\n", ip )

   /* access the value using the pointer */
   fmt.Printf("Value of *ip variable: %d\n", *ip )
}
```

When the above code is compiled and executed, it produces the following result –

```
Address of var variable: 10328000
Address stored in ip variable: 10328000
Value of *ip variable: 20
```

Nil Pointers in Go

Go compiler assign a Nil value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned nil is called a nil pointer.

The nil pointer is a constant with a value of zero defined in several standard libraries. Consider the following program –

```
package main

import "fmt"

func main() {
   var  ptr *int

   fmt.Printf("The value of ptr is : %x\n", ptr  )
}
```

When the above code is compiled and executed, it produces the following result –

```
The value of ptr is 0
```

On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the nil (zero) value, it is assumed to point to nothing.

To check for a nil pointer, you can use an if statement as follows –

```
if(ptr != nil)     /* succeeds if p is not nil */
if(ptr == nil)    /* succeeds if p is null */
```

Go Pointers in Detail

Pointers have many but easy concepts and they are very important to Go programming. The following concepts of pointers should be clear to a Go programmer −

Sr.No	Concept & Description
1	Go - Array of pointers You can define arrays to hold a number of pointers.
2	Go - Pointer to pointer Go allows you to have pointer on a pointer and so on.
3	Passing pointers to functions in Go Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function.

Go - Array of pointers

Before we understand the concept of arrays of pointers, let us consider the following example, which makes use of an array of 3 integers –

```
package main

import "fmt"

const MAX int = 3

func main() {
   a := []int{10,100,200}
   var i int

   for i = 0; i < MAX; i++ {
      fmt.Printf("Value of a[%d] = %d\n", i, a[i] )
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
Value of a[0] = 10
Value of a[1] = 100
Value of a2] = 200
```

There may be a situation when we want to maintain an array, which can store pointers to an int or string or any other data type available. The following statement declares an array of pointers to an integer –

```
var ptr [MAX]*int;
```

This declares ptr as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value. The following example makes use of three integers, which will be stored in an array of pointers as follows –

```
package main

import "fmt"

const MAX int = 3

func main() {
   a := []int{10,100,200}
   var i int
   var ptr [MAX]*int;

   for  i = 0; i < MAX; i++ {
      ptr[i] = &a[i] /* assign the address of integer. */
   }
   for  i = 0; i < MAX; i++ {
      fmt.Printf("Value of a[%d] = %d\n", i,*ptr[i] )
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
Value of a[0] = 10
Value of a[1] = 100
Value of a[2] = 200
```

○ Go – Pointer to pointer

A pointer to a pointer is a form of chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the

address of the second pointer, which points to the location that contains the actual value as shown below.

A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following statement declares a pointer to a pointer of type int –

```
var ptr **int;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown in the following example

```
package main

import "fmt"

func main() {
   var a int
   var ptr *int
   var pptr **int

   a = 3000

   /* take the address of var */
   ptr = &a

   /* take the address of ptr using address of operator & */
   pptr = &ptr

   /* take the value using pptr */
   fmt.Printf("Value of a = %d\n", a )
   fmt.Printf("Value available at *ptr = %d\n", *ptr )
   fmt.Printf("Value available at **pptr = %d\n", **pptr)
}
```

When the above code is compiled and executed, it produces the following result –

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

Go - Passing pointers to functions

Go programming language allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.

In the following example, we pass two pointers to a function and change the value inside the function which reflects back in the calling function −

```
package main

import "fmt"

func main() {
   /* local variable definition */
   var a int = 100
   var b int = 200

   fmt.Printf("Before swap, value of a : %d\n", a )
   fmt.Printf("Before swap, value of b : %d\n", b )

   /* calling a function to swap the values.
   * &a indicates pointer to a ie. address of variable a and
   * &b indicates pointer to b ie. address of variable b.
   */
   swap(&a, &b);

   fmt.Printf("After swap, value of a : %d\n", a )
   fmt.Printf("After swap, value of b : %d\n", b )
}
func swap(x *int, y *int) {
   var temp int
   temp = *x    /* save the value at address x */
   *x = *y      /* put y into x */
   *y = temp    /* put temp into y */
}
```

When the above code is compiled and executed, it produces the following result −

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

Go – Structures

Go arrays allow you to define variables that can hold several data items of the same kind. Structure is another user-defined data type available in Go programming, which allows you to combine data items of different kinds.

Structures are used to represent a record. Suppose you want to keep track of the books in a library. You might want to track the following attributes of each book −

- Title
- Author
- Subject
- Book ID

In such a scenario, structures are highly useful.

Defining a Structure

To define a structure, you must use type and struct statements. The struct statement defines a new data type, with multiple members for your program. The type statement binds a name with the type which is struct in our case. The format of the struct statement is as follows −

```
type struct_variable_type struct {
   member definition;
   member definition;
   ...
   member definition;
}
```

Once a structure type is defined, it can be used to declare variables of that type using the following syntax.

```
variable_name := structure_variable_type {value1, value2...valuen}
```

Accessing Structure Members

To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure

member that we wish to access. You would use struct keyword to define variables of structure type. The following example explains how to use a structure –

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}
func main() {
   var Book1 Books    /* Declare Book1 of type Book */
   var Book2 Books    /* Declare Book2 of type Book */

   /* book 1 specification */
   Book1.title = "Go Programming"
   Book1.author = "Mahesh Kumar"
   Book1.subject = "Go Programming Tutorial"
   Book1.book_id = 6495407

   /* book 2 specification */
   Book2.title = "Telecom Billing"
   Book2.author = "Zara Ali"
   Book2.subject = "Telecom Billing Tutorial"
   Book2.book_id = 6495700

   /* print Book1 info */
   fmt.Printf( "Book 1 title : %s\n", Book1.title)
   fmt.Printf( "Book 1 author : %s\n", Book1.author)
   fmt.Printf( "Book 1 subject : %s\n", Book1.subject)
   fmt.Printf( "Book 1 book_id : %d\n", Book1.book_id)
   /* print Book2 info */
   fmt.Printf( "Book 2 title : %s\n", Book2.title)
   fmt.Printf( "Book 2 author : %s\n", Book2.author)
   fmt.Printf( "Book 2 subject : %s\n", Book2.subject)
   fmt.Printf( "Book 2 book_id : %d\n", Book2.book_id)
}
```

When the above code is compiled and executed, it produces the following result –

```
Book 1 title      : Go Programming
Book 1 author     : Mahesh Kumar
Book 1 subject    : Go Programming Tutorial
Book 1 book_id    : 6495407
Book 2 title      : Telecom Billing
Book 2 author     : Zara Ali
Book 2 subject    : Telecom Billing Tutorial
Book 2 book_id    : 6495700
```

Structures as Function Arguments

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the same way as you did in the above example –

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}
func main() {
   var Book1 Books    /* Declare Book1 of type Book */
   var Book2 Books    /* Declare Book2 of type Book */

   /* book 1 specification */
   Book1.title = "Go Programming"
   Book1.author = "Mahesh Kumar"
   Book1.subject = "Go Programming Tutorial"
   Book1.book_id = 6495407

   /* book 2 specification */
   Book2.title = "Telecom Billing"
   Book2.author = "Zara Ali"
   Book2.subject = "Telecom Billing Tutorial"
```

```
   Book2.book_id = 6495700

   /* print Book1 info */
   printBook(Book1)

   /* print Book2 info */
   printBook(Book2)
}
func printBook( book Books ) {
   fmt.Printf( "Book title : %s\n", book.title);
   fmt.Printf( "Book author : %s\n", book.author);
   fmt.Printf( "Book subject : %s\n", book.subject);
   fmt.Printf( "Book book_id : %d\n", book.book_id);
}
```

When the above code is compiled and executed, it produces the following result −

```
Book title     : Go Programming
Book author    : Mahesh Kumar
Book subject   : Go Programming Tutorial
Book book_id   : 6495407
Book title     : Telecom Billing
Book author    : Zara Ali
Book subject   : Telecom Billing Tutorial
Book book_id   : 6495700
```

Pointers to Structures

You can define pointers to structures in the same way as you define pointer to any other variable as follows –

```
var struct_pointer *Books
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows −

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the "." operator as follows –

```
struct_pointer.title;
```

Let us re-write the above example using structure pointer −

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}
func main() {
   var Book1 Books   /* Declare Book1 of type Book */
   var Book2 Books   /* Declare Book2 of type Book */

   /* book 1 specification */
   Book1.title = "Go Programming"
   Book1.author = "Mahesh Kumar"
   Book1.subject = "Go Programming Tutorial"
   Book1.book_id = 6495407

   /* book 2 specification */
   Book2.title = "Telecom Billing"
   Book2.author = "Zara Ali"
   Book2.subject = "Telecom Billing Tutorial"
   Book2.book_id = 6495700

   /* print Book1 info */
   printBook(&Book1)

   /* print Book2 info */
   printBook(&Book2)
}
func printBook( book *Books ) {
   fmt.Printf( "Book title : %s\n", book.title);
   fmt.Printf( "Book author : %s\n", book.author);
   fmt.Printf( "Book subject : %s\n", book.subject);
   fmt.Printf( "Book book_id : %d\n", book.book_id);
}
```

When the above code is compiled and executed, it produces the following result −

```
Book title     : Go Programming
Book author    : Mahesh Kumar
Book subject   : Go Programming Tutorial
Book book_id   : 6495407
Book title     : Telecom Billing
Book author    : Zara Ali
Book subject   : Telecom Billing Tutorial
Book book_id   : 6495700
```

Go – Slices

Go Slice is an abstraction over Go Array. Go Array allows you to define variables that can hold several data items of the same kind but it does not provide any inbuilt method to increase its size dynamically or get a sub-array of its own. Slices overcome this limitation. It provides many utility functions required on Array and is widely used in Go programming.

Defining a slice

To define a slice, you can declare it as an array without specifying its size. Alternatively, you can use make function to create a slice.

```
var numbers []int /* a slice of unspecified size */
/* numbers == []int{0,0,0,0,0}*/
numbers = make([]int,5,5) /* a slice of length 5 and capacity 5*/
```

len() and cap() functions

A **slice** is an abstraction over array. It actually uses arrays as an underlying structure. The **len()** function returns the elements presents in the slice where cap() function returns the capacity of the slice (i.e., how many elements it can be accommodate). The following example explains the usage of slice –

```
package main

import "fmt"

func main() {
   var numbers = make([]int,3,5)
   printSlice(numbers)
}
func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

When the above code is compiled and executed, it produces the following result –

```
len = 3 cap = 5 slice = [0 0 0]
```

Nil slice

If a **slice** is declared with no inputs, then by default, it is initialized as nil. Its length and capacity are zero. For example –

```
package main

import "fmt"

func main() {
   var numbers []int
   printSlice(numbers)

   if(numbers == nil){
      fmt.Printf("slice is nil")
   }
}
func printSlice(x []int){
   fmt.Printf("len = %d cap = %d slice = %v\n", len(x), cap(x),x)
}
```

When the above code is compiled and executed, it produces the following result −

```
len = 0 cap = 0 slice = []
slice is nil
```

Subslicing

Slice allows lower-bound and upper bound to be specified to get the subslice of it using[lower-bound:upper-bound]. For example −

```
package main

import "fmt"

func main() {
  /* create a slice */
  numbers := []int{0,1,2,3,4,5,6,7,8}
  printSlice(numbers)

  /* print the original slice */
  fmt.Println("numbers ==", numbers)
```

```
   /* print the sub slice starting from index 1(included) to index 4(excluded)*/
   fmt.Println("numbers[1:4] ==", numbers[1:4])

   /* missing lower bound implies 0*/
   fmt.Println("numbers[:3] ==", numbers[:3])

   /* missing upper bound implies len(s)*/
   fmt.Println("numbers[4:] ==", numbers[4:])

   numbers1 := make([]int,0,5)
   printSlice(numbers1)

   /* print the sub slice starting from index 0(included) to index 2(excluded) */
   number2 := numbers[:2]
   printSlice(number2)

   /* print the sub slice starting from index 2(included) to index 5(excluded) */
   number3 := numbers[2:5]
   printSlice(number3)

}
func printSlice(x []int){
   fmt.Printf("len = %d cap = %d slice = %v\n", len(x), cap(x),x)
}
```

When the above code is compiled and executed, it produces the following result −

```
len = 9 cap = 9 slice = [0 1 2 3 4 5 6 7 8]
numbers == [0 1 2 3 4 5 6 7 8]
numbers[1:4] == [1 2 3]
numbers[:3] == [0 1 2]
numbers[4:] == [4 5 6 7 8]
len = 0 cap = 5 slice = []
len = 2 cap = 9  slice = [0 1]
len = 3 cap = 7 slice = [2 3 4]
```

append() and copy() Functions

One can increase the capacity of a slice using the **append()** function. Using **copy()**function, the contents of a source slice are copied to a destination slice. For example

```
package main

import "fmt"

func main() {
   var numbers []int
   printSlice(numbers)

   /* append allows nil slice */
   numbers = append(numbers, 0)
   printSlice(numbers)

   /* add one element to slice*/
   numbers = append(numbers, 1)
   printSlice(numbers)

   /* add more than one element at a time*/
   numbers = append(numbers, 2,3,4)
   printSlice(numbers)

   /* create a slice numbers1 with double the capacity of earlier slice*/
   numbers1 := make([]int, len(numbers), (cap(numbers))*2)

   /* copy content of numbers to numbers1 */
   copy(numbers1,numbers)
   printSlice(numbers1)
}
func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

When the above code is compiled and executed, it produces the following result −

```
len = 0 cap = 0 slice = []
len = 1 cap = 2 slice = [0]
len = 2 cap = 2 slice = [0 1]
len = 5 cap = 8 slice = [0 1 2 3 4]
len = 5 cap = 16 slice = [0 1 2 3 4]
```

Go – Range

The range keyword is used in for loop to iterate over items of an array, slice, channel or map. With array and slices, it returns the index of the item as integer. With maps, it returns the key of the next key-value pair. Range either returns one value or two. If only one value is used on the left of a range expression, it is the 1st value in the following table.

Range expression	1st Value	2nd Value(Optional)
Array or slice a [n]E	index i int	a[i] E
String s string type	index i int	rune int
map m map[K]V	key k K	value m[k] V
channel c chan E	element e E	none

Example

The following paragraph shows how to use range –

```
package main

import "fmt"

func main() {
   /* create a slice */
   numbers := []int{0,1,2,3,4,5,6,7,8}

   /* print the numbers */
   for i:= range numbers {
      fmt.Println("Slice item",i,"is",numbers[i])
   }

   /* create a map*/
   countryCapitalMap := map[string] string
{"France":"Paris","Italy":"Rome","Japan":"Tokyo"}

   /* print map using keys*/
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[country])
```

```
   }

   /* print map using key-value*/
   for country,capital := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",capital)
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
Slice item 0 is 0
Slice item 1 is 1
Slice item 2 is 2
Slice item 3 is 3
Slice item 4 is 4
Slice item 5 is 5
Slice item 6 is 6
Slice item 7 is 7
Slice item 8 is 8
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
```

Go – Maps

Go provides another important data type named map which maps unique keys to values. A key is an object that you use to retrieve a value at a later date. Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.

Defining a Map

You must use make function to create a map.

```
/* declare a variable, by default map will be nil*/
var map_variable map[key_data_type]value_data_type

/* define the map as nil map can not be assigned any value*/
map_variable = make(map[key_data_type]value_data_type)
```

Example

The following example illustrates how to create and use a map −

```
package main

import "fmt"

func main() {
   var countryCapitalMap map[string]string
   /* create a map*/
   countryCapitalMap = make(map[string]string)

   /* insert key-value pairs in the map*/
   countryCapitalMap["France"] = "Paris"
   countryCapitalMap["Italy"] = "Rome"
   countryCapitalMap["Japan"] = "Tokyo"
   countryCapitalMap["India"] = "New Delhi"

   /* print map using keys*/
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[country])
   }
```

```
   /* test if entry is present in the map or not*/
   capital, ok := countryCapitalMap["United States"]

   /* if ok is true, entry is present otherwise entry is absent*/
   if(ok){
      fmt.Println("Capital of United States is", capital)
   } else {
      fmt.Println("Capital of United States is not present")
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
Capital of India is New Delhi
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of United States is not present
```

delete() Function

delete() function is used to delete an entry from a map. It requires the map and the corresponding key which is to be deleted. For example –

```
package main

import "fmt"

func main() {
   /* create a map*/
   countryCapitalMap := map[string] string
{"France":"Paris","Italy":"Rome","Japan":"Tokyo","India":"New Delhi"}

   fmt.Println("Original map")

   /* print map */
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[country])
   }

   /* delete an entry */
   delete(countryCapitalMap,"France");
```

```
   fmt.Println("Entry for France is deleted")

   fmt.Println("Updated map")

   /* print map */
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[country])
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
Original Map
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of India is New Delhi
Entry for France is deleted
Updated Map
Capital of India is New Delhi
Capital of Italy is Rome
Capital of Japan is Tokyo
```

Go – Recursion

Recursion is the process of repeating items in a self-similar way. The same concept applies in programming languages as well. If a program allows to call a function inside the same function, then it is called a recursive function call. Take a look at the following example

```
func recursion() {
   recursion() /* function calls itself */
}
func main() {
   recursion()
}
```

The Go programming language supports recursion. That is, it allows a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go on to become an infinite loop.

Examples of Recursion in Go

Recursive functions are very useful to solve many mathematical problems such as calculating factorial of a number, generating a Fibonacci series, etc.

Example 1: Calculating Factorial Using Recursion in Go

The following example calculates the factorial of a given number using a recursive function –

```
package main

import "fmt"

func factorial(i int)int {
   if(i <= 1) {
      return 1
   }
   return i * factorial(i - 1)
}
func main() {
   var i int = 15
   fmt.Printf("Factorial of %d is %d", i, factorial(i))
}
```

When the above code is compiled and executed, it produces the following result −

```
Factorial of 15 is 2004310016
```

Example 2: Fibonacci Series Using Recursion in Go

The following example shows how to generate a Fibonacci series of a given number using a recursive function −

```
package main

import "fmt"

func fibonaci(i int) (ret int) {
   if i == 0 {
      return 0
   }
   if i == 1 {
      return 1
   }
   return fibonaci(i-1) + fibonaci(i-2)
}
func main() {
   var i int
   for i = 0; i < 10; i++ {
      fmt.Printf("%d ", fibonaci(i))
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
0 1 1 2 3 5 8 13 21 34
```

Go - Type Casting

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a long value into a simple integer then you can type cast long to int. You can convert values from one type to another using the cast operator. Its syntax is as follows –

```
type_name(expression)
```

Example

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating number operation.

```
package main

import "fmt"

func main() {
   var sum int = 17
   var count int = 5
   var mean float32

   mean = float32(sum)/float32(count)
   fmt.Printf("Value of mean : %f\n",mean)
}
```

When the above code is compiled and executed, it produces the following result −

```
Value of mean : 3.400000
```

Go – Interfaces

Go programming provides another data type called interfaces which represents a set of method signatures. The struct data type implements these interfaces to have method definitions for the method signature of the interfaces.

Syntax

```
/* define an interface */
type interface_name interface {
   method_name1 [return_type]
   method_name2 [return_type]
   method_name3 [return_type]
   ...
   method_namen [return_type]
}

/* define a struct */
type struct_name struct {
   /* variables */
}

/* implement interface methods*/
func (struct_name_variable struct_name) method_name1() [return_type] {
   /* method implementation */
}
...
func (struct_name_variable struct_name) method_namen() [return_type] {
   /* method implementation */
}
```

Example

```
package main

import ("fmt" "math")

/* define an interface */
type Shape interface {
  area() float64
}
```

```
/* define a circle */
type Circle struct {
   x,y,radius float64
}

/* define a rectangle */
type Rectangle struct {
   width, height float64
}

/* define a method for circle (implementation of Shape.area())*/
func(circle Circle) area() float64 {
   return math.Pi * circle.radius * circle.radius
}

/* define a method for rectangle (implementation of Shape.area())*/
func(rect Rectangle) area() float64 {
   return rect.width * rect.height
}

/* define a method for shape */
func getArea(shape Shape) float64 {
   return shape.area()
}

func main() {
   circle := Circle{x:0,y:0,radius:5}
   rectangle := Rectangle {width:10, height:5}

   fmt.Printf("Circle area: %f\n",getArea(circle))
   fmt.Printf("Rectangle area: %f\n",getArea(rectangle))
}
```

When the above code is compiled and executed, it produces the following result −

```
Circle area: 78.539816
Rectangle area: 50.000000
```

Go - Error Handling

Go programming provides a pretty simple error handling framework with inbuilt error interface type of the following declaration –

```
type error interface {
   Error() string
}
```

Functions normally return error as last return value. Use **errors.New** to construct a basic error message as following –

```
func Sqrt(value float64)(float64, error) {
   if(value < 0){
      return 0, errors.New("Math: negative number passed to Sqrt")
   }
   return math.Sqrt(value), nil
}
```

Use return value and error message.

```
result, err:= Sqrt(-1)

if err != nil {
   fmt.Println(err)
}
```

Example

```
package main

import "errors"
import "fmt"
import "math"

func Sqrt(value float64)(float64, error) {
  if(value < 0){
    return 0, errors.New("Math: negative number passed to Sqrt")
  }
  return math.Sqrt(value), nil
}
func main() {
```

```
  result, err:= Sqrt(-1)

  if err != nil {
     fmt.Println(err)
  } else {
     fmt.Println(result)
  }

  result, err = Sqrt(9)

  if err != nil {
     fmt.Println(err)
  } else {
     fmt.Println(result)
  }
}
```

When the above code is compiled and executed, it produces the following result −

```
Math: negative number passed to Sqrt
3
```

RUBY PROGRAMMING LANGUAGE

Ruby – Overview

Ruby is a pure object-oriented programming language. It was created in 1993 by Yukihiro Matsumoto of Japan.

You can find the name Yukihiro Matsumoto on the Ruby mailing list at www.ruby-lang.org. Matsumoto is also known as Matz in the Ruby community.

Ruby is "A Programmer's Best Friend".

Ruby has features that are similar to those of Smalltalk, Perl, and Python. Perl, Python, and Smalltalk are scripting languages. Smalltalk is a true object-oriented language. Ruby, like Smalltalk, is a perfect object-oriented language. Using Ruby syntax is much easier than using Smalltalk syntax.

Features of Ruby

- Ruby is an open-source and is freely available on the Web, but it is subject to a license.
- Ruby is a general-purpose, interpreted programming language.
- Ruby is a true object-oriented programming language.
- Ruby is a server-side scripting language similar to Python and PERL.
- Ruby can be used to write Common Gateway Interface (CGI) scripts.
- Ruby can be embedded into Hypertext Markup Language (HTML).
- Ruby has a clean and easy syntax that allows a new developer to learn very quickly and easily.
- Ruby has similar syntax to that of many programming languages such as C++ and Perl.
- Ruby is very much scalable and big programs written in Ruby are easily maintainable.
- Ruby can be used for developing Internet and intranet applications.
- Ruby can be installed in Windows and POSIX environments.
- Ruby support many GUI tools such as Tcl/Tk, GTK, and OpenGL.
- Ruby can easily be connected to DB2, MySQL, Oracle, and Sybase.

- Ruby has a rich set of built-in functions, which can be used directly into Ruby scripts.

Tools You Will Need

For performing the examples discussed in this tutorial, you will need a latest computer like Intel Core i3 or i5 with a minimum of 2GB of RAM (4GB of RAM recommended).

You also will need the following software −

- Linux or Windows 95/98/2000/NT or Windows 7 operating system.
- Apache 1.3.19-5 Web server.
- Internet Explorer 5.0 or above Web browser.
- Ruby 1.8.5

This tutorial will provide the necessary skills to create GUI, networking, and Web applications using Ruby. It also will talk about extending and embedding Ruby applications.

Local Environment Setup

If you are still willing to set up your environment for Ruby programming language, then let's proceed. This tutorial will teach you all the important topics related to environment setup. We would recommend you to go through the following topics first and then proceed further −

- Ruby Installation on Linux/Unix − If you are planning to have your development environment on Linux/Unix Machine, then go through this chapter.
- Ruby Installation on Windows − If you are planning to have your development environment on Windows Machine, then go through this chapter.
- Ruby Command Line Options − This chapter list out all the command line options, which you can use along with Ruby interpreter.
- Ruby Environment Variables − This chapter has a list of all the important environment variables to be set to make Ruby Interpreter works.

Popular Ruby Editors

To write your Ruby programs, you will need an editor −

If you are working on Windows machine, then you can use any simple text editor like Notepad or Edit plus.

VIM (Vi IMproved) is a very simple text editor. This is available on almost all Unix machines and now Windows as well. Otherwise, your can use your favorite vi editor to write Ruby programs.

RubyWin is a Ruby Integrated Development Environment (IDE) for Windows.

Ruby Development Environment (RDE) is also a very good IDE for windows users.

Interactive Ruby (IRb)

Interactive Ruby (IRb) provides a shell for experimentation. Within the IRb shell, you can immediately view expression results, line by line.

This tool comes along with Ruby installation so you have nothing to do extra to have IRb working.

Just type irb at your command prompt and an Interactive Ruby Session will start as given below –

```
$irb
irb 0.6.1(99/09/16)
irb(main):001:0> def hello
irb(main):002:1> out = "Hello World"
irb(main):003:1> puts out
irb(main):004:1> end
nil
irb(main):005:0> hello
Hello World
nil
irb(main):006:0>
```

Ruby – Syntax

Let us write a simple program in ruby. All ruby files will have **extension .rb**. So, put the following source code in a **test.rb** file.

```
#!/usr/bin/ruby -w

puts "Hello, Ruby!";
```

Here, we assumed that you have Ruby interpreter available in /usr/bin directory. Now, try to run this program as follows –

```
$ ruby test.rb
```

This will produce the following result −

```
Hello, Ruby!
```

You have seen a simple Ruby program, now let us see a few basic concepts related to Ruby Syntax.

Whitespace in Ruby Program

Whitespace characters such as spaces and tabs are generally ignored in Ruby code, except when they appear in strings. Sometimes, however, they are used to interpret ambiguous statements. Interpretations of this sort produce warnings when the -w option is enabled.

Example

```
a + b is interpreted as a+b ( Here a is a local variable)
a  +b is interpreted as a(+b) ( Here a is a method call)
```

Line Endings in Ruby Program

Ruby interprets semicolons and newline characters as the ending of a statement. However, if Ruby encounters operators, such as +, −, or backslash at the end of a line, they indicate the continuation of a statement.

Ruby Identifiers

Identifiers are names of variables, constants, and methods. Ruby identifiers are case sensitive. It means Ram and RAM are two different identifiers in Ruby.

Ruby identifier names may consist of alphanumeric characters and the underscore character (_).

Reserved Words

The following list shows the reserved words in Ruby. These reserved words may not be used as constant or variable names. They can, however, be used as method names.

BEGIN	do	Next	then
END	else	Nil	true
alias	elsif	Not	undef
and	end	Or	unless
begin	ensure	Redo	until
break	false	Rescue	when
case	for	Retry	while
class	if	Return	while
def	in	Self	__FILE__
defined?	module	Super	__LINE__

Here Document in Ruby

"Here Document" refers to build strings from multiple lines. Following a << you can specify a string or an identifier to terminate the string literal, and all lines following the current line up to the terminator are the value of the string.

If the terminator is quoted, the type of quotes determines the type of the line-oriented string literal. Notice there must be no space between << and the terminator.

Here are different examples –

```
#!/usr/bin/ruby -w

print <<EOF
   This is the first way of creating
   here document ie. multiple line string.
EOF

print <<"EOF";                # same as above
   This is the second way of creating
   here document ie. multiple line string.
EOF

print <<`EOC`                 # execute commands
        echo hi there
        echo lo there
EOC

print <<"foo", <<"bar"  # you can stack them
        I said foo.
foo
        I said bar.
bar
```

This will produce the following result −

```
   This is the first way of creating
   her document ie. multiple line string.
   This is the second way of creating
   her document ie. multiple line string.
hi there
lo there
      I said foo.
      I said bar.
```

Ruby BEGIN Statement

Syntax

```
BEGIN {
   code
}
```

Declares code to be called before the program is run.

Example

```
#!/usr/bin/ruby

puts "This is main Ruby Program"

BEGIN {
   puts "Initializing Ruby Program"
}
```

This will produce the following result –

```
Initializing Ruby Program
This is main Ruby Program
```

Ruby END Statement

Syntax

```
END {
   code
}
```

Declares code to be called at the end of the program.

Example

```
#!/usr/bin/ruby

puts "This is main Ruby Program"

END {
   puts "Terminating Ruby Program"
}
BEGIN {
   puts "Initializing Ruby Program"
}
```

This will produce the following result –

```
Initializing Ruby Program
This is main Ruby Program
Terminating Ruby Program
```

Ruby Comments

A comment hides a line, part of a line, or several lines from the Ruby interpreter. You can use the hash character (#) at the beginning of a line –

```
# I am a comment. Just ignore me.
```

Or, a comment may be on the same line after a statement or expression −

```
name = "Madisetti" # This is again comment
```

You can comment multiple lines as follows −

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

Here is another form. This block comment conceals several lines from the interpreter with =begin/=end –

```
=begin
This is a comment.
This is a comment, too.
This is a comment, too.
I said that already.
=end
```

Ruby - Classes and Objects

Ruby is a perfect **Object-Oriented** Programming Language. The features of the object-oriented programming language include −

- Data Encapsulation
- Data Abstraction
- Polymorphism
- Inheritance

An object-oriented program involves classes and objects. A class is the blueprint from which individual objects are created. In object-oriented terms, we say that your bicycle is an instance of the class of objects known as bicycles.

Take the example of any vehicle. It comprises wheels, horsepower, and fuel or gas tank capacity. These characteristics form the data members of the class Vehicle. You can differentiate one vehicle from the other with the help of these characteristics.

A vehicle can also have certain functions, such as halting, driving, and speeding. Even these functions form the data members of the class Vehicle. You can, therefore, define a class as a combination of characteristics and functions.

A class Vehicle can be defined as –

```
Class Vehicle {

   Number no_of_wheels
   Number horsepower
   Characters type_of_tank
   Number Capacity
   Function speeding {
   }

   Function driving {
   }

   Function halting {
   }
}
```

By assigning different values to these data members, you can form several instances of the class Vehicle. For example, an airplane has three wheels, horsepower of 1,000, fuel as the type of tank, and a capacity of 100 liters. In the same way, a car has four wheels, horsepower of 200, gas as the type of tank, and a capacity of 25 liters.

Defining a Class in Ruby

To implement object-oriented programming by using Ruby, you need to first learn how to create objects and classes in Ruby.

A class in Ruby always starts with the keyword class followed by the name of the class. The name should always be in initial capitals. The class Customer can be displayed as –

```
class Customer
end
```

You terminate a class by using the keyword end. All the data members in the class are between the class definition and the end keyword.

Variables in a Ruby Class

Ruby provides four types of variables −

- **Local Variables** − Local variables are the variables that are defined in a method. Local variables are not available outside the method. You will see more details about method in subsequent chapter. Local variables begin with a lowercase letter or _.
- **Instance Variables** − Instance variables are available across methods for any particular instance or object. That means that instance variables change from object to object. Instance variables are preceded by the at sign (@) followed by the variable name.
- **Class Variables** − Class variables are available across different objects. A class variable belongs to the class and is a characteristic of a class. They are preceded by the sign @@ and are followed by the variable name.
- **Global Variables** − Class variables are not available across classes. If you want to have a single variable, which is available across classes, you need to define a global variable. The global variables are always preceded by the dollar sign ($).

Example

Using the class variable @@no_of_customers, you can determine the number of objects that are being created. This enables in deriving the number of customers.

```
class Customer
   @@no_of_customers = 0
end
```

Creating Objects in Ruby using new Method

Objects are instances of the class. You will now learn how to create objects of a class in Ruby. You can create objects in Ruby by using the method new of the class.

The method new is a unique type of method, which is predefined in the Ruby library. The new method belongs to the class methods.

Here is the example to create two objects cust1 and cust2 of the class Customer –

```
cust1 = Customer. new
cust2 = Customer. new
```

Here, cust1 and cust2 are the names of two objects. You write the object name followed by the equal to sign (=) after which the class name will follow. Then, the dot operator and the keyword new will follow.

Custom Method to Create Ruby Objects

You can pass parameters to method new and those parameters can be used to initialize class variables.

When you plan to declare the new method with parameters, you need to declare the method initialize at the time of the class creation.

The initialize method is a special type of method, which will be executed when the new method of the class is called with parameters.

Here is the example to create initialize method –

```
class Customer
   @@no_of_customers = 0
   def initialize(id, name, addr)
      @cust_id = id
      @cust_name = name
      @cust_addr = addr
   end
end
```

In this example, you declare the initialize method with id, name, and addr as local variables. Here, def and end are used to define a Ruby method initialize. You will learn more about methods in subsequent chapters.

In the initialize method, you pass on the values of these local variables to the instance variables **@cust_id**, **@cust_name**, and **@cust_addr**. Here local variables hold the values that are passed along with the new method.

Now, you can create objects as follows –

```
cust1 = Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2 = Customer.new("2", "Poul", "New Empire road, Khandala")
```

Member Functions in Ruby Class

In Ruby, functions are called methods. Each method in a class starts with the keyword def followed by the method name.

The method name always preferred in lowercase letters. You end a method in Ruby by using the keyword end.

Here is the example to define a Ruby method –

```
class Sample
   def function
      statement 1
      statement 2
   end
end
```

Here, statement 1 and statement 2 are part of the body of the method function inside the class Sample. These statments could be any valid Ruby statement. For example we can put a method puts to print Hello Ruby as follows –

```
class Sample
   def hello
      puts "Hello Ruby!"
   end
end
```

Now in the following example, create one object of Sample class and call hello method and see the result –

```
#!/usr/bin/ruby

class Sample
   def hello
      puts "Hello Ruby!"
   end
end

# Now using above class to create objects
object = Sample. new
object.hello
```

This will produce the following result −

```
Hello Ruby!
```

Ruby - Variables, Constants and Literals

Variables are the memory locations, which hold any data to be used by any program.

There are five types of variables supported by Ruby. You already have gone through a small description of these variables in the previous chapter as well. These five types of variables are explained in this chapter.

Ruby Global Variables

Global variables begin with $. Uninitialized global variables have the value nil and produce warnings with the -w option.

Assignment to global variables alters the global status. It is not recommended to use global variables. They make programs cryptic.

Here is an example showing the usage of global variable.

```
#!/usr/bin/ruby

$global_variable = 10
class Class1
   def print_global
      puts "Global variable in Class1 is #$global_variable"
   end
end
class Class2
   def print_global
      puts "Global variable in Class2 is #$global_variable"
   end
end

class1obj = Class1.new
class1obj.print_global
class2obj = Class2.new
class2obj.print_global
```

Here **$global_variable** is a global variable. This will produce the following result −

NOTE − In Ruby, you CAN access value of any variable or constant by putting a hash (#) character just before that variable or constant.

```
Global variable in Class1 is 10
Global variable in Class2 is 10
```

Ruby Instance Variables

Instance variables begin with @. Uninitialized instance variables have the value nil and produce warnings with the -w option.

Here is an example showing the usage of Instance Variables.

```
#!/usr/bin/ruby

class Customer
   def initialize(id, name, addr)
      @cust_id = id
      @cust_name = name
      @cust_addr = addr
   end
   def display_details()
      puts "Customer id #@cust_id"
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
   end
end

# Create Objects
cust1 = Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2 = Customer.new("2", "Poul", "New Empire road, Khandala")

# Call Methods
cust1.display_details()
cust2.display_details()
```

Here, @cust_id, @cust_name and @cust_addr are instance variables. This will produce the following result –

```
Customer id 1
Customer name John
Customer address Wisdom Apartments, Ludhiya
Customer id 2
Customer name Poul
Customer address New Empire road, Khandala
```

Ruby Class Variables

Class variables begin with @@ and must be initialized before they can be used in method definitions.

Referencing an uninitialized class variable produces an error. Class variables are shared among descendants of the class or module in which the class variables are defined.

Overriding class variables produce warnings with the -w option.

Here is an example showing the usage of class variable −

```
#!/usr/bin/ruby

class Customer
   @@no_of_customers = 0
   def initialize(id, name, addr)
      @cust_id = id
      @cust_name = name
      @cust_addr = addr
   end
   def display_details()
      puts "Customer id #@cust_id"
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
   end
   def total_no_of_customers()
      @@no_of_customers += 1
      puts "Total number of customers: #@@no_of_customers"
   end
end

# Create Objects
cust1 = Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2 = Customer.new("2", "Poul", "New Empire road, Khandala")

# Call Methods
cust1.total_no_of_customers()
cust2.total_no_of_customers()
```

Here @@no_of_customers is a class variable. This will produce the following result –

```
Total number of customers: 1
Total number of customers: 2
```

Ruby Local Variables

Local variables begin with a lowercase letter or _. The scope of a local variable ranges from class, module, def, or do to the corresponding end or from a block's opening brace to its close brace {}.

When an uninitialized local variable is referenced, it is interpreted as a call to a method that has no arguments.

Assignment to uninitialized local variables also serves as variable declaration. The variables start to exist until the end of the current scope is reached. The lifetime of local variables is determined when Ruby parses the program.

In the above example, local variables are id, name and addr.

Ruby Constants

Constants begin with an uppercase letter. Constants defined within a class or module can be accessed from within that class or module, and those defined outside a class or module can be accessed globally.

Constants may not be defined within methods. Referencing an uninitialized constant produces an error. Making an assignment to a constant that is already initialized produces a warning.

```ruby
#!/usr/bin/ruby

class Example
   VAR1 = 100
   VAR2 = 200
   def show
      puts "Value of first Constant is #{VAR1}"
      puts "Value of second Constant is #{VAR2}"
   end
end

# Create Objects
object = Example.new()
object.show
```

Here VAR1 and VAR2 are constants. This will produce the following result –

```
Value of first Constant is 100
Value of second Constant is 200
```

Ruby Pseudo-Variables

They are special variables that have the appearance of local variables but behave like constants. You cannot assign any value to these variables.

- self − The receiver object of the current method.
- true − Value representing true.
- false − Value representing false.
- nil − Value representing undefined.
- __FILE__ − The name of the current source file.
- __LINE__ − The current line number in the source file.

Ruby Basic Literals

The rules Ruby uses for literals are simple and intuitive. This section explains all basic Ruby Literals.

Integer Numbers

Ruby supports integer numbers. An integer number can range from -230 to 230-1 or -262 to 262-1. Integers within this range are objects of class Fixnum and integers outside this range are stored in objects of class Bignum.

You write integers using an optional leading sign, an optional base indicator (0 for octal, 0x for hex, or 0b for binary), followed by a string of digits in the appropriate base. Underscore characters are ignored in the digit string.

You can also get the integer value, corresponding to an ASCII character or escape the sequence by preceding it with a question mark.

Example

```
123                  # Fixnum decimal
1_234                # Fixnum decimal with underline
-500                 # Negative Fixnum
0377                 # octal
0xff                 # hexadecimal
0b1011               # binary
?a                   # character code for 'a'
?\n                  # code for a newline (0x0a)
12345678901234567890 # Bignum
```

NOTE − Class and Objects are explained in a separate chapter of this tutorial.

Floating Numbers

Ruby supports floating numbers. They are also numbers but with decimals. Floating-point numbers are objects of class Float and can be any of the following −

Example

```
123.4                # floating point value
1.0e6                # scientific notation
4E20                 # dot not required
4e+20                # sign before exponential
```

String Literals

Ruby strings are simply sequences of 8-bit bytes and they are objects of class String. Double-quoted strings allow substitution and backslash notation but single-quoted strings don't allow substitution and allow backslash notation only for \\ and \'

Example

```
#!/usr/bin/ruby -w

puts 'escape using "\\"';
puts 'That\'s right';
```

This will produce the following result −

```
escape using "\"
That's right
```

You can substitute the value of any Ruby expression into a string using the sequence #{ expr }. Here, expr could be any ruby expression.

```
#!/usr/bin/ruby -w

puts "Multiplication Value : #{24*60*60}";
```

This will produce the following result –

```
Multiplication Value : 86400
```

Backslash Notations

Following is the list of Backslash notations supported by Ruby –

Notation	Character represented
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\a	Bell (0x07)
\e	Escape (0x1b)

\s	Space (0x20)
\nnn	Octal notation (n being 0-7)
\xnn	Hexadecimal notation (n being 0-9, a-f, or A-F)
\cx, \C-x	Control-x
\M-x	Meta-x (c | 0x80)
\M-\C-x	Meta-Control-x
\x	Character x

For more detail on Ruby Strings, you will learn in **Ruby Strings**.

Ruby Arrays

Literals of Ruby Array are created by placing a comma-separated series of object references between the square brackets. A trailing comma is ignored.

Example

```
#!/usr/bin/ruby

ary = [  "fred", 10, 3.14, "This is a string", "last element", ]
ary.each do |i|
   puts i
end
```

This will produce the following result −

```
fred
10
3.14
This is a string
last element
```

Ruby Hashes

A literal Ruby Hash is created by placing a list of key/value pairs between braces, with either a comma or the sequence => between the key and the value. A trailing comma is ignored.

Example

```
#!/usr/bin/ruby

hsh = colors = { "red" => 0xf00, "green" => 0x0f0, "blue" => 0x00f }
hsh.each do |key, value|
   print key, " is ", value, "\n"
end
```

This will produce the following result –

```
red is 3840
green is 240
blue is 15
```

For more detail on Ruby Hashes, you will lean it in **Ruby Hashes**.

Ruby Ranges

A Range represents an interval which is a set of values with a start and an end. Ranges may be constructed using the s..e and s...e literals, or with Range.new.

Ranges constructed using .. run from the start to the end inclusively. Those created using ... exclude the end value. When used as an iterator, ranges return each value in the sequence.

A **range** (1..5) means it includes 1, 2, 3, 4, 5 values and a range (1...5) means it includes 1, 2, 3, 4 values.

Example

```
#!/usr/bin/ruby

(10..15).each do |n|
   print n, ' '
end
```

This will produce the following result −

```
10 11 12 13 14 15
```

For more detail on Ruby Ranges, you will learn it in **Ruby Ranges**.

Ruby – Operators

Ruby supports a rich set of operators, as you'd expect from a modern language. Most operators are actually method calls. For example, a + b is interpreted as a.+(b), where the + method in the object referred to by variable a is called with b as its argument.

For each operator (+ - * / % ** & | ^ << >> && ||), there is a corresponding form of abbreviated assignment operator (+= -= etc.).

Ruby Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
+	Addition − Adds values on either side of the operator.	a + b will give 30
−	Subtraction − Subtracts right hand operand from left hand operand.	a - b will give -10
*	Multiplication − Multiplies values on either side of the operator.	a * b will give 200
/	Division − Divides left hand operand by right hand operand.	b / a will give 2
%	Modulus − Divides left hand operand by right hand operand and returns remainder.	b % a will give 0
**	Exponent − Performs exponential (power) calculation on operators.	a**b will give 10 to the power 20

Ruby Comparison Operators

Assume variable a hold 10 and variable b holds 20, then –

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(a == b) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
<=>	Combined comparison operator. Returns 0 if first operand equals second, 1 if first operand is greater than the second and -1 if first operand is less than the second.	(a <=> b) returns -1.

===	Used to test equality within a when clause of a *case* statement.	(1...10) === 5 returns true.
.eql?	True if the receiver and argument have both the same type and equal values.	1 == 1.0 returns true, but 1.eql?(1.0) is false.
equal?	True if the receiver and argument have the same object id.	if aObj is duplicate of bObj then aObj == bObj is true, a.equal?bObj is false but a.equal?aObj is true.

Ruby Assignment Operators

Assume variable a hold 10 and variable b holds 20, then –

Operator	Description	Example
=	Simple assignment operator, assigns values from right side operands to left side operand.	c = a + b will assign the value of a + b into c
+=	Add AND assignment operator, adds right operand to the left operand and assign the result to left operand.	c += a is equivalent to c = c + a
-=	Subtract AND assignment operator, subtracts right operand from the left operand and assign the result to left operand.	c -= a is equivalent to c = c - a
*=	Multiply AND assignment operator, multiplies right operand with the left operand and assign the result to left operand.	c *= a is equivalent to c = c * a
/=	Divide AND assignment operator, divides left operand with the right operand and assign the result to left operand.	c /= a is equivalent to c = c / a

%=	Modulus AND assignment operator, takes modulus using two operands and assign the result to left operand.	c %= a is equivalent to c = c % a
**=	Exponent AND assignment operator, performs exponential (power) calculation on operators and assign value to the left operand.	c **= a is equivalent to c = c ** a

Ruby Parallel Assignment

Ruby also supports the parallel assignment of variables. This enables multiple variables to be initialized with a single line of Ruby code. For example –

```
a = 10
b = 20
c = 30
```

This may be more quickly declared using parallel assignment −

```
a, b, c = 10, 20, 30
```

Parallel assignment is also useful for swapping the values held in two variables −

```
a, b = b, c
```

Ruby Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation.

Assume if a = 60; and b = 13; now in binary format they will be as follows −

```
a    =  0011 1100
b    =  0000 1101
-----------------
a&b  =  0000 1100
a|b  =  0011 1101
a^b  =  0011 0001
~a   =  1100 0011
```

The following Bitwise operators are supported by Ruby language.

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(a & b) will give 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(a | b) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(a ^ b) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~a) will give -61, which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	a << 2 will give 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 will give 15, which is 0000 1111

Ruby Logical Operators

The following **logical operators** are supported by Ruby language

Assume variable a hold 10 and variable b holds 20, then –

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true, then the condition becomes true.	(a and b) is true.
or	Called Logical OR Operator. If any of the two operands are non zero, then the condition becomes true.	(a or b) is true.
&&	Called Logical AND operator. If both the operands are non zero, then the condition becomes true.	(a && b) is true.

||	Called Logical OR Operator. If any of the two operands are non zero, then the condition becomes true.	(a || b) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(a && b) is false.
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	not(a && b) is false.

Ruby Ternary Operator

There is one more operator called Ternary Operator. It first evaluates an expression for a true or false value and then executes one of the two given statements depending upon the result of the evaluation. The conditional operator has this syntax –

Operator	Description	Example
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

Ruby Range Operators

Sequence ranges in Ruby are used to create a range of successive values - consisting of a start value, an end value, and a range of values in between.

In Ruby, these sequences are created using the ".." and "..." range operators. The two-dot form creates an inclusive range, while the three-dot form creates a range that excludes the specified high value.

Operator	Description	Example
..	Creates a range from start point to end point inclusive.	1..10 Creates a range from 1 to 10 inclusive.
...	Creates a range from start point to end point exclusive.	1...10 Creates a range from 1 to 9.

Ruby defined? Operators

defined? is a special operator that takes the form of a method call to determine whether or not the past expression is defined. It returns a description string of the expression, or **nil** if the expression isn't defined.

There is various usage of defined? Operator

Usage 1

```
defined? variable # True if variable is initialized
```

For Example

```
foo = 42
defined? foo    # => "local-variable"
defined? $_     # => "global-variable"
defined? bar    # => nil (undefined)
```

Usage 2

```
defined? method_call # True if a method is defined
```

For Example

```
defined? puts         # => "method"
defined? puts(bar)    # => nil (bar is not defined here)
defined? unpack       # => nil (not defined here)
```

Usage 3

```
# True if a method exists that can be called with super user
defined? super
```

For Example

```
defined? super      # => "super" (if it can be called)
defined? super      # => nil (if it cannot be)
```

Usage 4

```
defined? yield   # True if a code block has been passed
```

For Example

```
defined? yield    # => "yield" (if there is a block passed)
defined? yield    # => nil (if there is no block)
```

Ruby Dot "." and Double Colon "::" Operators

You call a module method by preceding its name with the module's name and a period, and you reference a constant using the module name and two colons.

The **::** is a unary operator that allows: constants, instance methods and class methods defined within a class or module, to be accessed from anywhere outside the class or module.

Remember in Ruby, classes and methods may be considered constants too.

You need to just prefix the **:: Const_name** with an expression that returns the appropriate class or module object.

If no prefix expression is used, the main Object class is used by default.

Here are two examples –

```
MR_COUNT = 0         # constant defined on main Object class
module Foo
   MR_COUNT = 0
   ::MR_COUNT = 1    # set global count to 1
   MR_COUNT = 2      # set local count to 2
end
puts MR_COUNT        # this is the global constant
puts Foo::MR_COUNT   # this is the local "Foo" constant
```

Second Example

```
CONST = ' out there'
class Inside_one
   CONST = proc {' in there'}
   def where_is_my_CONST
      ::CONST + ' inside one'
   end
end
class Inside_two
   CONST = ' inside two'
   def where_is_my_CONST
      CONST
   end
end
puts Inside_one.new.where_is_my_CONST
puts Inside_two.new.where_is_my_CONST
puts Object::CONST + Inside_two::CONST
puts Inside_two::CONST + CONST
puts Inside_one::CONST
puts Inside_one::CONST.call + Inside_two::CONST
```

Ruby Operators Precedence

The following table lists all operators from highest precedence to lowest.

Method	Operator	Description
Yes	::	Constant resolution operator
Yes	[ ] [ ]=	Element reference, element set
Yes	**	Exponentiation (raise to the power)
Yes	! ~ + -	Not, complement, unary plus and minus (method names for the last two are +@ and -@)
Yes	* / %	Multiply, divide, and modulo
Yes	+ -	Addition and subtraction
Yes	>> <<	Right and left bitwise shift
Yes	&	Bitwise 'AND'

Yes	^ |	Bitwise exclusive `OR' and regular `OR'
Yes	<= < > >=	Comparison operators
Yes	<=> == === != =~ !~	Equality and pattern match operators (!= and !~ may not be defined as methods)
	&&	Logical 'AND'
	||	Logical 'OR'
	.. ...	Range (inclusive and exclusive)
	? :	Ternary if-then-else
	= %= { /= -= += |= &= >>= <<= *= &&= ||= **=	Assignment
	defined?	Check if specified symbol defined
	Not	Logical negation
	or and	Logical composition

NOTE − Operators with a Yes in the method column are actually methods, and as such may be overridden.

Ruby – Comments

Comments are lines of annotation within Ruby code that are ignored at runtime. A single line comment starts with # character and they extend from # to the end of the line as follows –

```
#!/usr/bin/ruby -w
# This is a single line comment.

puts "Hello, Ruby!"
```

When executed, the above program produces the following result –

```
Hello, Ruby!
```

Ruby Multiline Comments

You can comment multiple lines using **=begin** and **=end** syntax as follows –

```
#!/usr/bin/ruby -w

puts "Hello, Ruby!"

=begin
This is a multiline comment and con spwan as many lines as you
like. But =begin and =end should come in the first line only.
=end
```

When executed, the above program produces the following result –

```
Hello, Ruby!
```

Make sure trailing comments are far enough from the code and that they are easily distinguished. If more than one trailing comment exists in a block, align them. For example –

```
@counter      # keeps track times page has been hit
@siteCounter  # keeps track of times all pages have been hit
```

Ruby - if...else, case, unless

Ruby offers conditional structures that are pretty common to modern languages. Here, we will explain all the conditional statements and modifiers available in Ruby.

Ruby if...else Statement

Syntax

```
if conditional [then]
   code...
[elsif conditional [then]
   code...]...
[else
   code...]
end
```

if expressions are used for conditional execution. The values false and nil are false, and everything else are true. Notice Ruby uses elsif, not else if nor elif.

Executes code if the conditional is true. If the conditional is not true, code specified in the else clause is executed.

An if expression's conditional is separated from code by the reserved word then, a newline, or a semicolon.

Example

```
#!/usr/bin/ruby

x = 1
if x > 2
   puts "x is greater than 2"
elsif x <= 2 and x!=0
   puts "x is 1"
else
   puts "I can't guess the number"
end
```

```
x is 1
```

Ruby if modifier

Syntax

```
code if condition
```

Executes code if the conditional is **true**.

Example

```
#!/usr/bin/ruby

$debug = 1
print "debug\n" if $debug
```

This will produce the following result –

```
debug
```

Ruby unless Statement

Syntax

```
unless conditional [then]
   code
[else
   code ]
end
```

Executes code if conditional is false. If the conditional is true, code specified in the else clause is executed.

Example

```
#!/usr/bin/ruby

x = 1
unless x>=2
   puts "x is less than 2"
 else
   puts "x is greater than 2"
end
```

This will produce the following result –

```
x is less than 2
```

Ruby unless modifier

Syntax

```
code unless conditional
```

Executes code if conditional is false.

Example

```
#!/usr/bin/ruby

$var =  1
print "1 -- Value is set\n" if $var
print "2 -- Value is set\n" unless $var

$var = false
print "3 -- Value is set\n" unless $var
```

This will produce the following result −

```
1 -- Value is set
3 -- Value is set
```

Ruby case Statement

Syntax

```
case expression
[when expression [, expression ...] [then]
   code ]...
[else
   code ]
end
```

Compares the expression specified by case and that specified by when using the === operator and executes the code of the when clause that matches.

The expression specified by the when clause is evaluated as the left operand. If no when clauses match, case executes the code of the else clause.

A when statement's expression is separated from code by the reserved word then, a newline, or a semicolon. Thus −

```
case expr0
when expr1, expr2
   stmt1
when expr3, expr4
   stmt2
else
   stmt3
end
```

is basically similar to the following −

```
_tmp = expr0
if expr1 === _tmp || expr2 === _tmp
   stmt1
elsif expr3 === _tmp || expr4 === _tmp
   stmt2
else
   stmt3
end
```

Example

```
#!/usr/bin/ruby

$age =  5
case $age
when 0 .. 2
   puts "baby"
when 3 .. 6
   puts "little child"
when 7 .. 12
   puts "child"
when 13 .. 18
   puts "youth"
else
   puts "adult"
end
```

This will produce the following result −

```
little child
```

Ruby – Loops

Loops in Ruby are used to execute the same block of code a specified number of times. This chapter details all the loop statements supported by Ruby.

Ruby while Statement

Syntax

```
while conditional [do]
   code
end
```

Executes code while conditional is true. A while loop's conditional is separated from code by the reserved word do, a **newline**, **backslash **, or a **semicolon ;**.

Example

```
#!/usr/bin/ruby

$i = 0
$num = 5

while $i < $num  do
   puts("Inside the loop i = #$i" )
   $i +=1
end
```

This will produce the following result −

```
Inside the loop i = 0
Inside the loop i = 1
Inside the loop i = 2
Inside the loop i = 3
Inside the loop i = 4
```

Ruby while modifier

Syntax

```
code while condition

OR

begin
  code
end while conditional
```

Executes code while conditional is true.

If a while modifier follows a begin statement with no rescue or ensure clauses, code is executed once before conditional is evaluated.

Example

```
#!/usr/bin/ruby

$i = 0
$num = 5
begin
   puts("Inside the loop i = #$i" )
   $i +=1
end while $i < $num
```

This will produce the following result −

```
Inside the loop i = 0
Inside the loop i = 1
Inside the loop i = 2
Inside the loop i = 3
Inside the loop i = 4
```

Ruby until Statement

```
until conditional [do]
   code
end
```

Executes code while conditional is false. An until statement's conditional is separated from code by the reserved word do, a newline, or a semicolon.

Example

```
#!/usr/bin/ruby

$i = 0
$num = 5

until $i > $num  do
   puts("Inside the loop i = #$i" )
   $i +=1;
end
```

This will produce the following result −

```
Inside the loop i = 0
Inside the loop i = 1
Inside the loop i = 2
Inside the loop i = 3
Inside the loop i = 4
Inside the loop i = 5
```

Ruby until modifier

Syntax

```
code until conditional

OR

begin
   code
end until conditional
```

Executes code while conditional is false.

If an until modifier follows a begin statement with no rescue or ensure clauses, code is executed once before conditional is evaluated.

Example

```
#!/usr/bin/ruby

$i = 0
$num = 5
begin
   puts("Inside the loop i = #$i" )
   $i +=1;
end until $i > $num
```

This will produce the following result −

```
Inside the loop i = 0
Inside the loop i = 1
Inside the loop i = 2
Inside the loop i = 3
Inside the loop i = 4
Inside the loop i = 5
```

Ruby for Statement

Syntax

```
for variable [, variable ...] in expression [do]
   code
end
```

Executes *code* once for each element in expression.

Example

```
#!/usr/bin/ruby

for i in 0..5
   puts "Value of local variable is #{i}"
end
```

Here, we have defined the range **0..5**. The statement for **i** in **0..5** will allow i to take values in the range from 0 to 5 (including 5). This will produce the following result –

```
Value of local variable is 0
Value of local variable is 1
Value of local variable is 2
Value of local variable is 3
Value of local variable is 4
Value of local variable is 5
```

A for...in loop is almost exactly equivalent to the following –

```
(expression).each do |variable[, variable...]| code end
```

except that a for loop doesn't create a new scope for local variables. A for loop's expression is separated from code by the reserved word do, a newline, or a semicolon.

Example

```
#!/usr/bin/ruby

(0..5).each do |i|
   puts "Value of local variable is #{i}"
end
```

This will produce the following result −

```
Value of local variable is 0
Value of local variable is 1
Value of local variable is 2
Value of local variable is 3
Value of local variable is 4
Value of local variable is 5
```

Ruby break Statement

Syntax

```
break
```

Terminates the most internal loop. Terminates a method with an associated block if called within the block (with the method returning nil).

Example

```
#!/usr/bin/ruby

for i in 0..5
   if i > 2 then
      break
   end
   puts "Value of local variable is #{i}"
end
```

This will produce the following result −

```
Value of local variable is 0
Value of local variable is 1
Value of local variable is 2
```

Ruby next Statement

Syntax

```
next
```

Jumps to the next iteration of the most internal loop. Terminates execution of a block if called within a block (with yield or call returning nil).

Example

```
#!/usr/bin/ruby

for i in 0..5
   if i < 2 then
      next
   end
   puts "Value of local variable is #{i}"
end
```

This will produce the following result −

```
Value of local variable is 2
Value of local variable is 3
Value of local variable is 4
Value of local variable is 5
```

Ruby redo Statement

Syntax

```
redo
```

Restarts this iteration of the most internal loop, without checking loop condition. Restarts yield or call if called within a block.

Example

```
#!/usr/bin/ruby

for i in 0..5
   if i < 2 then
      puts "Value of local variable is #{i}"
      redo
   end
end
```

This will produce the following result and will go in an infinite loop −

```
Value of local variable is 0
Value of local variable is 0
.............................
```

Ruby retry Statement

Syntax

```
retry
```

If retry appears in rescue clause of begin expression, restart from the beginning of the begin body.

```
begin
   do_something # exception raised
rescue
   # handles error
   retry  # restart from beginning
end
```

If retry appears in the iterator, the block, or the body of the for expression, restarts the invocation of the iterator call. Arguments to the iterator is re-evaluated.

```
for i in 1..5
   retry if some_condition # restart from i == 1
end
```

Example

```
#!/usr/bin/ruby
for i in 0..5
   retry if i > 2
puts "Value of local variable is #{i}"
end
```

This will produce the following result and will go in an infinite loop −

```
Value of local variable is 1
Value of local variable is 2
Value of local variable is 1
Value of local variable is 2
Value of local variable is 1
Value of local variable is 2
.............................
```

Ruby – Methods

Ruby methods are very similar to functions in any other programming language. Ruby methods are used to bundle one or more repeatable statements into a single unit.

Method names should begin with a lowercase letter. If you begin a method name with an uppercase letter, Ruby might think that it is a constant and hence can parse the call incorrectly.

Methods should be defined before calling them, otherwise Ruby will raise an exception for undefined method invoking.

Syntax

```
def method_name [( [arg [= default]]...[, * arg [, &expr ]])]
   expr..
end
```

So, you can define a simple method as follows −

```
def method_name
   expr..
end
```

You can represent a method that accepts parameters like this −

```
def method_name (var1, var2)
   expr..
end
```

You can set default values for the parameters, which will be used if method is called without passing the required parameters –

```
def method_name (var1 = value1, var2 = value2)
   expr..
end
```

Whenever you call the simple method, you write only the method name as follows −

```
method_name
```

However, when you call a method with parameters, you write the method name along with the parameters, such as –

```
method_name 25, 30
```

The most important drawback to using methods with parameters is that you need to remember the number of parameters whenever you call such methods. For example, if a method accepts three parameters and you pass only two, then Ruby displays an error.

Example

```
#!/usr/bin/ruby

def test(a1 = "Ruby", a2 = "Perl")
   puts "The programming language is #{a1}"
   puts "The programming language is #{a2}"
end
test "C", "C++"
test
```

This will produce the following result –

```
The programming language is C
The programming language is C++
The programming language is Ruby
The programming language is Perl
```

Return Values from Methods

Every method in Ruby returns a value by default. This returned value will be the value of the last statement. For example –

```
def test
   i = 100
   j = 10
   k = 0
end
```

This method, when called, will return the last declared variable k.

Ruby return Statement

The return statement in ruby is used to return one or more values from a Ruby Method.

Syntax

```
return [expr[`,' expr...]]
```

If more than two expressions are given, the array containing these values will be the return value. If no expression given, nil will be the return value.

Example

```
return

OR

return 12

OR

return 1,2,3
```

Have a look at this example –

```
#!/usr/bin/ruby

def test
   i = 100
   j = 200
   k = 300
return i, j, k
end
var = test
puts var
```

This will produce the following result –

```
100
200
300
```

Variable Number of Parameters

Suppose you declare a method that takes two parameters, whenever you call this method, you need to pass two parameters along with it.

However, Ruby allows you to declare methods that work with a variable number of parameters. Let us examine a sample of this –

```
#!/usr/bin/ruby

def sample (*test)
   puts "The number of parameters is #{test.length}"
   for i in 0...test.length
      puts "The parameters are #{test[i]}"
   end
end
sample "Zara", "6", "F"
sample "Mac", "36", "M", "MCA"
```

In this code, you have declared a method sample that accepts one parameter test. However, this parameter is a variable parameter. This means that this parameter can take in any number of variables. So, the above code will produce the following result –

```
The number of parameters is 3
The parameters are Zara
The parameters are 6
The parameters are F
The number of parameters is 4
The parameters are Mac
The parameters are 36
The parameters are M
The parameters are MCA
```

Class Methods

When a method is defined outside of the class definition, the method is marked as private by default. On the other hand, the methods defined in the class definition are marked as public by default. The default visibility and the private mark of the methods can be changed by public or private of the Module.

Whenever you want to access a method of a class, you first need to instantiate the class. Then, using the object, you can access any member of the class.

Ruby gives you a way to access a method without instantiating a class. Let us see how a class method is declared and accessed –

```
class Accounts
   def reading_charge
   end
   def Accounts.return_date
   end
end
```

See how the method **return_date** is declared. It is declared with the class name followed by a period, which is followed by the name of the method. You can access this class method directly as follows –

```
Accounts.return_date
```

To access this method, you need not create objects of the class Accounts.

Ruby alias Statement

This gives alias to methods or global variables. Aliases cannot be defined within the method body. The alias of the method keeps the current definition of the method, even when methods are overridden.

Making aliases for the numbered global variables ($1, $2,...) is prohibited. Overriding the built-in global variables may cause serious problems.

Syntax

```
alias method-name method-name
alias global-variable-name global-variable-name
```

Example

```
alias foo bar
alias $MATCH $&
```

Here we have defined foo alias for bar, and $MATCH is an alias for **$&**

Ruby undef Statement

This cancels the method definition. An undef cannot appear in the method body.

By using undef and alias, the interface of the class can be modified independently from the superclass, but notice it may be broke programs by the internal method call to self.

Syntax

```
undef method-name
```

Example

To undefine a method called bar do the following –

```
undef bar
```

Ruby – Blocks

You have seen how Ruby defines methods where you can put number of statements and then you call that method. Similarly, Ruby has a concept of Block.

- A block consists of chunks of code.
- You assign a name to a block.
- The code in the block is always enclosed within braces ({}).
- A block is always invoked from a function with the same name as that of the block. This means that if you have a block with the name test, then you use the function test to invoke this block.
- You invoke a block by using the yield statement.

Syntax

```
block_name {
   statement1
   statement2
   ..........
}
```

Here, you will learn to invoke a block by using a simple **yield statement**. You will also learn to use a yield statement with parameters for invoking a block. You will check the sample code with both types of yield statements.

The yield Statement

Let's look at an example of the yield statement

```
#!/usr/bin/ruby

def test
   puts "You are in the method"
   yield
   puts "You are again back to the method"
   yield
end
test {puts "You are in the block"}
```

This will produce the following result −

```
You are in the method
You are in the block
You are again back to the method
You are in the block
```

You also can pass parameters with the yield statement. Here is an example −

```
#!/usr/bin/ruby

def test
   yield 5
   puts "You are in the method test"
   yield 100
end
test {|i| puts "You are in the block #{i}"}
```

This will produce the following result −

```
You are in the block 5
You are in the method test
You are in the block 100
```

Here, the yield statement is written followed by parameters. You can even pass more than one parameter. In the block, you place a variable between two vertical lines (||) to accept the parameters. Therefore, in the preceding code, the yield 5 statement passes the value 5 as a parameter to the test block.

Now, look at the following statement –

```
test {|i| puts "You are in the block #{i}"}
```

Here, the value 5 is received in the variable i. Now, observe the following puts statement

```
puts "You are in the block #{i}"
```

The output of this puts statement is −

```
You are in the block 5
```

If you want to pass more than one parameters, then the yield statement becomes −

```
yield a, b
```

and the block is −

```
test {|a, b| statement}
```

The parameters will be separated by commas.

Blocks and Methods

You have seen how a block and a method can be associated with each other. You normally invoke a block by using the yield statement from a method that has the same name as that of the block. Therefore, you write –

```
#!/usr/bin/ruby

def test
   yield
end
test{ puts "Hello world"}
```

This example is the simplest way to implement a block. You call the test block by using the yield statement.

But if the last argument of a method is preceded by &, then you can pass a block to this method and this block will be assigned to the last parameter. In case both * and & are present in the argument list, & should come later.

```
#!/usr/bin/ruby

def test(&block)
   block.call
end
test { puts "Hello World!"}
```

This will produce the following result –

```
Hello World!
```

BEGIN and END Blocks

Every Ruby source file can declare blocks of code to be run as the file is being loaded (the BEGIN blocks) and after the program has finished executing (the END blocks).

```
#!/usr/bin/ruby

BEGIN {
   # BEGIN block code
   puts "BEGIN code block"
}

END {
   # END block code
   puts "END code block"
}
   # MAIN block code
puts "MAIN code block"
```

A program may include multiple BEGIN and END blocks. BEGIN blocks are executed in the order they are encountered. END blocks are executed in reverse order. When executed, the above program produces the following result –

```
BEGIN code block
MAIN code block
END code block
```

Ruby - Modules and Mixins

Modules are a way of grouping together methods, classes, and constants. Modules give you two major benefits.

- Modules provide a namespace and prevent name clashes.
- Modules implement the mixin facility.

Modules define a namespace, a sandbox in which your methods and constants can play without having to worry about being stepped on by other methods and constants.

Syntax

```
module Identifier
   statement1
   statement2
   ...........
end
```

Module constants are named just like class constants, with an initial uppercase letter. The method definitions look similar, too: Module methods are defined just like class methods. As with class methods, you call a module method by preceding its name with the module's name and a period, and you reference a constant using the module name and two colons.

Example

```
#!/usr/bin/ruby

# Module defined in trig.rb file

module Trig
   PI = 3.141592654
   def Trig.sin(x)
   # ..
   end
   def Trig.cos(x)
   # ..
   end
end
```

We can define one more module with the same function name but different functionality

```
#!/usr/bin/ruby

# Module defined in moral.rb file

module Moral
   VERY_BAD = 0
   BAD = 1
   def Moral.sin(badness)
   # ...
   end
end
```

Like class methods, whenever you define a method in a module, you specify the module name followed by a dot and then the method name.

Ruby require Statement

The require statement is similar to the include statement of C and C++ and the import statement of Java. If a third program wants to use any defined module, it can simply load the module files using the Ruby require statement −

Syntax

```
require filename
```

Here, it is not required to give **.rb** extension along with a file name.

Example

```
$LOAD_PATH << '.'

require 'trig.rb'
require 'moral'

y = Trig.sin(Trig::PI/4)
wrongdoing = Moral.sin(Moral::VERY_BAD)
```

Here we are using **$LOAD_PATH << '.'** to make Ruby aware that included files must be searched in the current directory. If you do not want to use $LOAD_PATH then you can use **require_relative** to include files from a relative directory.

IMPORTANT − Here, both the files contain the same function name. So, this will result in code ambiguity while including in calling program but modules avoid this code ambiguity and we are able to call appropriate function using module name.

Ruby include Statement

You can embed a module in a class. To embed a module in a class, you use the include statement in the class −

Syntax

```
include modulename
```

If a module is defined in a separate file, then it is required to include that file using require statement before embedding module in a class.

Example

Consider the following module written in **support.rb** file.

```
module Week
   FIRST_DAY = "Sunday"
   def Week.weeks_in_month
      puts "You have four weeks in a month"
   end
   def Week.weeks_in_year
      puts "You have 52 weeks in a year"
   end
end
```

Now, you can include this module in a class as follows −

```
#!/usr/bin/ruby
$LOAD_PATH << '.'
require "support"

class Decade
include Week
   no_of_yrs = 10
   def no_of_months
      puts Week::FIRST_DAY
      number = 10*12
      puts number
   end
end
d1 = Decade.new
puts Week::FIRST_DAY
Week.weeks_in_month
Week.weeks_in_year
d1.no_of_months
```

This will produce the following result −

```
Sunday
You have four weeks in a month
You have 52 weeks in a year
Sunday
120
```

Mixins in Ruby

Before going through this section, we assume you have the knowledge of Object Oriented Concepts.

When a class can inherit features from more than one parent class, the class is supposed to show multiple inheritance.

Ruby does not support multiple inheritance directly but Ruby Modules have another wonderful use. At a stroke, they pretty much eliminate the need for multiple inheritance, providing a facility called a mixin.

Mixins give you a wonderfully controlled way of adding functionality to classes. However, their true power comes out when the code in the mixin starts to interact with code in the class that uses it.

Let us examine the following sample code to gain an understand of mixin −

```
module A
   def a1
   end
   def a2
   end
end
module B
   def b1
   end
   def b2
   end
end

class Sample
include A
include B
   def s1
   end
end

samp = Sample.new
samp.a1
samp.a2
samp.b1
samp.b2
samp.s1
```

Module A consists of the methods a1 and a2. Module B consists of the methods b1 and b2. The class Sample includes both modules A and B. The class Sample can access all four methods, namely, a1, a2, b1, and b2. Therefore, you can see that the class Sample inherits from both the modules. Thus, you can say the class Sample shows multiple inheritance or a **mixin**.

Ruby – Strings

A **String object** in Ruby holds and manipulates an arbitrary sequence of one or more bytes, typically representing characters that represent human language.

The simplest string literals are enclosed in single quotes (the apostrophe character). The text within the quote marks is the value of the string −

```
'This is a simple Ruby string literal'
```

If you need to place an apostrophe within a single-quoted string literal, precede it with a backslash, so that the Ruby interpreter does not think that it terminates the string –

```
'Won\'t you read O\'Reilly\'s book?'
```

The backslash also works to escape another backslash, so that the second backslash is not itself interpreted as an escape character.

Following are the string-related features of Ruby.

Expression Substitution

Expression substitution is a means of embedding the value of any Ruby expression into a string using #{ and } –

```
#!/usr/bin/ruby

x, y, z = 12, 36, 72
puts "The value of x is #{ x }."
puts "The sum of x and y is #{ x + y }."
puts "The average was #{ (x + y + z)/3 }."
```

This will produce the following result −

```
The value of x is 12.
The sum of x and y is 48.
The average was 40.
```

General Delimited Strings

With general delimited strings, you can create strings inside a pair of matching though arbitrary delimiter characters, e.g., !, (, {, <, etc., preceded by a percent character (%). Q, q, and x have special meanings. General delimited strings can be –

```
%{Ruby is fun.}  equivalent to "Ruby is fun."
%Q{ Ruby is fun. } equivalent to " Ruby is fun. "
%q[Ruby is fun.]  equivalent to a single-quoted string
%x!ls! equivalent to back tick command output `ls`
```

Character Encoding

The default character set for Ruby is ASCII, whose characters may be represented by single bytes. If you use UTF-8, or another modern character set, characters may be represented in one to four bytes.

You can change your character set using $KCODE at the beginning of your program, like this –

```
$KCODE = 'u'
```

String Built-in Methods

We need to have an instance of String object to call a String method. Following is the way to create an instance of String object –

```
new [String.new(str = "")]
```

This will return a new string object containing a copy of str. Now, using str object, we can all use any available instance methods. For example –

```
#!/usr/bin/ruby

myStr = String.new("THIS IS TEST")
foo = myStr.downcase

puts "#{foo}"
```

This will produce the following result −

```
this is test
```

Ruby – Arrays

Ruby arrays are ordered, integer-indexed collections of any object. Each element in an array is associated with and referred to by an index.

Array indexing starts at 0, as in C or Java. A negative index is assumed relative to the end of the array --- that is, an index of -1 indicates the last element of the array, -2 is the next to last element in the array, and so on.

Ruby arrays can hold objects such as String, Integer, Fixnum, Hash, Symbol, even other Array objects. Ruby arrays are not as rigid as arrays in other languages. Ruby arrays grow automatically while adding elements to them.

Creating Arrays

There are many ways to create or initialize an array. One way is with the new class method

```
names = Array.new
```

You can set the size of an array at the time of creating array –

```
names = Array.new(20)
```

The **array names** now have a size or length of 20 elements. You can return the size of an array with either the size or length methods −

```
#!/usr/bin/ruby

names = Array.new(20)
puts names.size  # This returns 20
puts names.length # This also returns 20
```

This will produce the following result −

```
20
20
```

You can assign a value to each element in the array as follows −

```
#!/usr/bin/ruby

names = Array.new(4, "mac")
puts "#{names}"
```

This will produce the following result −

```
["mac", "mac", "mac", "mac"]
```

You can also use a block with new, populating each element with what the block evaluates to –

```
#!/usr/bin/ruby

nums = Array.new(10) { |e| e = e * 2 }
puts "#{nums}"
```

This will produce the following result −

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

There is another method of Array, []. It works like this −

```
nums = Array.[](1, 2, 3, 4,5)
```

The Kernel module available in core Ruby has an Array method, which only accepts a single argument. Here, the method takes a range as an argument to create an array of digits

```
#!/usr/bin/ruby

digits = Array(0..9)
puts "#{digits}"
```

This will produce the following result −

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Array Built-in Methods

We need to have an instance of Array object to call an Array method. As we have seen, following is the way to create an instance of Array object –

```
Array.[](...) [or] Array[...] [or] [...]
```

This will return a new array populated with the given objects. Now, using the created object, we can call any available instance methods. For example –

```
#!/usr/bin/ruby

digits = Array(0..9)
num = digits.at(6)
puts "#{num}"
```

This will produce the following result −

```
6
```

Array pack Directives

Example

Try the following example to pack various data.

```
a = [ "a", "b", "c" ]
n = [ 65, 66, 67 ]
puts a.pack("A3A3A3")   #=> "a  b  c  "
puts a.pack("a3a3a3")   #=> "a\000\000b\000\000c\000\000"
puts n.pack("ccc")      #=> "ABC"
```

This will produce the following result −

```
a  b  c
abc
ABC
```

Ruby – Hashes

A Hash is a collection of key-value pairs like this: "employee" = > "salary". It is similar to an Array, except that indexing is done via arbitrary keys of any object type, not an integer index.

The order in which you traverse a hash by either key or value may seem arbitrary and will generally not be in the insertion order. If you attempt to access a hash with a key that does not exist, the method will return nil.

Creating Hashes

As with arrays, there is a variety of ways to create hashes. You can create an empty hash with the new class method −

```
months = Hash.new
```

You can also use new to create a hash with a default value, which is otherwise just nil −

```
months = Hash.new( "month" )

or

months = Hash.new "month"
```

When you access any key in a hash that has a default value, if the key or value doesn't exist, accessing the hash will return the default value –

```
#!/usr/bin/ruby

months = Hash.new( "month" )

puts "#{months[0]}"
puts "#{months[72]}"
```

This will produce the following result −

```
month
month
```

```
#!/usr/bin/ruby

H = Hash["a" => 100, "b" => 200]

puts "#{H['a']}"
puts "#{H['b']}"
```

This will produce the following result −

```
100
200
```

You can use any Ruby object as a key or value, even an array, so the following example is a valid one –

```
[1,"jan"] => "January"
```

Hash Built-in Methods

We need to have an instance of Hash object to call a Hash method. As we have seen, following is the way to create an instance of Hash object –

```
Hash[[key =>|, value]* ] or

Hash.new [or] Hash.new(obj) [or]
Hash.new { |hash, key| block }
```

This will return a new hash populated with the given objects. Now using the created object, we can call any available instance methods. For example −

```
#!/usr/bin/ruby

$, = ", "
months = Hash.new( "month" )
months = {"1" => "January", "2" => "February"}

keys = months.keys
puts "#{keys}"
```

This will produce the following result −

```
["1", "2"]
```

Following are the public hash methods (assuming hash is an array object) −

Sr.No.	Methods & Description
1	**hash == other_hash** Tests whether two hashes are equal, based on whether they have the same number of key-value pairs, and whether the key-value pairs match the corresponding pair in each hash.
2	**hash.[key]** Using a key, references a value from hash. If the key is not found, returns a default value.
3	**hash.[key] = value** Associates the value given by *value* with the key given by *key*.
4	**hash.clear** Removes all key-value pairs from hash.
5	**hash.default(key = nil)** Returns the default value for *hash*, nil if not set by default=. ([] returns a default value if the key does not exist in *hash*.)
6	**hash.default = obj** Sets a default value for *hash*.
7	**hash.default_proc** Returns a block if *hash* was created by a block.
8	**hash.delete(key) [or]** **array.delete(key) { |key| block }** Deletes a key-value pair from *hash* by *key*. If block is used, returns the result of a block if pair is not found. Compare *delete_if*.
9	**hash.delete_if { |key,value| block }** Deletes a key-value pair from *hash* for every pair the block evaluates to *true*.

10	**hash.each { |key,value| block }** Iterates over *hash*, calling the block once for each key, passing the key-value as a two-element array.
11	**hash.each_key { |key| block }** Iterates over *hash*, calling the block once for each key, passing *key* as a parameter.
12	**hash.each_key { |key_value_array| block }** Iterates over *hash*, calling the block once for each *key*, passing the *key* and *value* as parameters.
13	**hash.each_key { |value| block }** Iterates over *hash*, calling the block once for each *key*, passing *value* as a parameter.
14	**hash.empty?** Tests whether hash is empty (contains no key-value pairs), returning *true* or *false*.
15	**hash.fetch(key [, default]) [or]** **hash.fetch(key) { | key | block }** Returns a value from *hash* for the given *key*. If the *key* can't be found, and there are no other arguments, it raises an *IndexError* exception; if *default* is given, it is returned; if the optional block is specified, its result is returned.
16	**hash.has_key?(key) [or] hash.include?(key) [or]** **hash.key?(key) [or] hash.member?(key)** Tests whether a given *key* is present in hash, returning *true* or *false*.
17	**hash.has_value?(value)** Tests whether hash contains the given *value*.
18	**hash.index(value)**

	Returns the *key* for the given *value* in hash, *nil* if no matching value is found.
19	**hash.indexes(keys)** Returns a new array consisting of values for the given key(s). Will insert the default value for keys that are not found. This method is deprecated. Use select.
20	**hash.indices(keys)** Returns a new array consisting of values for the given key(s). Will insert the default value for keys that are not found. This method is deprecated. Use select.
21	**hash.inspect** Returns a pretty print string version of hash.
22	**hash.invert** Creates a new *hash*, inverting *keys* and *values* from *hash*; that is, in the new hash, the keys from *hash* become values and values become keys.
23	**hash.keys** Creates a new array with keys from *hash*.
24	**hash.length** Returns the size or length of *hash* as an integer.
25	**hash.merge(other_hash) [or]** **hash.merge(other_hash) { |key, oldval, newval| block }** Returns a new hash containing the contents of *hash* and *other_hash*, overwriting pairs in hash with duplicate keys with those from *other_hash*.
26	**hash.merge!(other_hash) [or]** **hash.merge!(other_hash) { |key, oldval, newval| block }** Same as merge, but changes are done in place.

27	**hash.rehash** Rebuilds *hash* based on the current values for each *key*. If values have changed since they were inserted, this method reindexes *hash*.
28	**hash.reject { |key, value| block }** Creates a new *hash* for every pair the *block* evaluates to *true*
29	**hash.reject! { |key, value| block }** Same as *reject*, but changes are made in place.
30	**hash.replace(other_hash)** Replaces the contents of *hash* with the contents of *other_hash*.
31	**hash.select { |key, value| block }** Returns a new array consisting of key-value pairs from *hash* for which the *block* returns *true*.
32	**hash.shift** Removes a key-value pair from *hash*, returning it as a two-element array.
33	**hash.size** Returns the *size* or length of *hash* as an integer.
34	**hash.sort** Converts *hash* to a two-dimensional array containing arrays of key-value pairs, then sorts it as an array.
35	**hash.store(key, value)** Stores a key-value pair in *hash*.
36	**hash.to_a** Creates a two-dimensional array from hash. Each key/value pair is converted to an array, and all these arrays are stored in a containing array.

37	**hash.to_hash** Returns *hash* (self).
38	**hash.to_s** Converts *hash* to an array, then converts that array to a string.
39	**hash.update(other_hash) [or]** **hash.update(other_hash) {|key, oldval, newval| block}** Returns a new hash containing the contents of *hash* and *other_hash*, overwriting pairs in *hash* with duplicate keys with those from *other_hash*.
40	**hash.value?(value)** Tests whether *hash* contains the given *value*.
41	**hash.values** Returns a new array containing all the values of *hash*.
42	**hash.values_at(obj, ...)** Returns a new array containing the values from *hash* that are associated with the given key or keys.

Ruby - Date & Time

The Time class represents dates and times in Ruby. It is a thin layer over the system date and time functionality provided by the operating system. This class may be unable on your system to represent dates before 1970 or after 2038.

This chapter makes you familiar with all the most wanted concepts of date and time.

Getting Current Date and Time

Following is the simple example to get current date and time −

```
#!/usr/bin/ruby -w

time1 = Time.new
puts "Current Time : " + time1.inspect

# Time.now is a synonym:
time2 = Time.now
puts "Current Time : " + time2.inspect
```

This will produce the following result −

```
Current Time : Mon Jun 02 12:02:39 -0700 2008
Current Time : Mon Jun 02 12:02:39 -0700 2008
```

Getting Components of a Date & Time

We can use Time object to get various components of date and time. Following is the example showing the same –

```ruby
#!/usr/bin/ruby -w

time = Time.new

# Components of a Time
puts "Current Time : " + time.inspect
puts time.year    # => Year of the date
puts time.month   # => Month of the date (1 to 12)
puts time.day     # => Day of the date (1 to 31 )
puts time.wday    # => 0: Day of week: 0 is Sunday
puts time.yday    # => 365: Day of year
puts time.hour    # => 23: 24-hour clock
puts time.min     # => 59
puts time.sec     # => 59
puts time.usec    # => 999999: microseconds
puts time.zone    # => "UTC": timezone name
```

This will produce the following result −

```
Current Time : Mon Jun 02 12:03:08 -0700 2008
2008
6
2
1
154
12
3
8
247476
UTC
```

Time.utc, Time.gm and Time.local Functions

These two functions can be used to format date in a standard format as follows –

```ruby
# July 8, 2008
Time.local(2008, 7, 8)
# July 8, 2008, 09:10am, local time
Time.local(2008, 7, 8, 9, 10)
# July 8, 2008, 09:10 UTC
Time.utc(2008, 7, 8, 9, 10)
# July 8, 2008, 09:10:11 GMT (same as UTC)
Time.gm(2008, 7, 8, 9, 10, 11)
```

Following is the example to get all the components in an array in the following format −

```
[sec,min,hour,day,month,year,wday,yday,isdst,zone]
```

Try the following −

```
#!/usr/bin/ruby -w

time = Time.new
values = time.to_a
p values
```

This will generate the following result −

```
[26, 10, 12, 2, 6, 2008, 1, 154, false, "MST"]
```

This array could be passed to **Time.utc** or **Time.local** functions to get different format of dates as follows –

```
#!/usr/bin/ruby -w

time = Time.new
values = time.to_a
puts Time.utc(*values)
```

This will generate the following result −

```
Mon Jun 02 12:15:36 UTC 2008
```

Following is the way to get time represented internally as seconds since the (platform-dependent) epoch −

```
# Returns number of seconds since epoch
time = Time.now.to_i

# Convert number of seconds into Time object.
Time.at(time)

# Returns second since epoch which includes microseconds
time = Time.now.to_f
```

Timezones and Daylight Savings Time

You can use a Time object to get all the information related to Timezones and daylight savings as follows –

```
time = Time.new

# Here is the interpretation
time.zone       # => "UTC": return the timezone
time.utc_offset # => 0: UTC is 0 seconds offset from UTC
time.zone       # => "PST" (or whatever your timezone is)
time.isdst      # => false: If UTC does not have DST.
time.utc?       # => true: if t is in UTC time zone
time.localtime  # Convert to local timezone.
time.gmtime     # Convert back to UTC.
time.getlocal   # Return a new Time object in local zone
time.getutc     # Return a new Time object in UTC
```

Formatting Times and Dates

There are various ways to format date and time. Here is one example showing a few –

```
#!/usr/bin/ruby -w

time = Time.new
puts time.to_s
puts time.ctime
puts time.localtime
puts time.strftime("%Y-%m-%d %H:%M:%S")
```

This will produce the following result −

```
Mon Jun 02 12:35:19 -0700 2008
Mon Jun  2 12:35:19 2008
Mon Jun 02 12:35:19 -0700 2008
2008-06-02 12:35:19
```

Time Formatting Directives

These directives in the following table are used with the method **Time.strftime**.

Sr.No.	Directive & Description
1	**%a** The abbreviated weekday name (Sun).
2	**%A**

	The full weekday name (Sunday).
3	**%b** The abbreviated month name (Jan).
4	**%B** The full month name (January).
5	**%c** The preferred local date and time representation.
6	**%d** Day of the month (01 to 31).
7	**%H** Hour of the day, 24-hour clock (00 to 23).
8	**%I** Hour of the day, 12-hour clock (01 to 12).
9	**%j** Day of the year (001 to 366).
10	**%m** Month of the year (01 to 12).
11	**%M** Minute of the hour (00 to 59).
12	**%p** Meridian indicator (AM or PM).
13	**%S**

	Second of the minute (00 to 60).
14	**%U** Week number of the current year, starting with the first Sunday as the first day of the first week (00 to 53).
15	**%W** Week number of the current year, starting with the first Monday as the first day of the first week (00 to 53).
16	**%w** Day of the week (Sunday is 0, 0 to 6).
17	**%x** Preferred representation for the date alone, no time.
18	**%X** Preferred representation for the time alone, no date.
19	**%y** Year without a century (00 to 99).
20	**%Y** Year with century.
21	**%Z** Time zone name.
22	**%%** Literal % character.

Time Arithmetic

You can perform simple arithmetic with time as follows –

```
now = Time.now           # Current time
puts now

past = now - 10          # 10 seconds ago. Time - number => Time
puts past

future = now + 10  # 10 seconds from now Time + number => Time
puts future

diff = future - past     # => 10  Time - Time => number of seconds
puts diff
```

This will produce the following result −

```
Thu Aug 01 20:57:05 -0700 2013
Thu Aug 01 20:56:55 -0700 2013
Thu Aug 01 20:57:15 -0700 2013
20.0
```

Ruby – Ranges

Ranges occur everywhere: January to December, 0 to 9, lines 50 through 67, and so on. Ruby supports ranges and allows us to use ranges in a variety of ways −

- Ranges as Sequences
- Ranges as Conditions
- Ranges as Intervals

Ranges as Sequences

The first and perhaps the most natural use of ranges is to express a sequence. Sequences have a start point, an end point, and a way to produce successive values in the sequence. Ruby creates these sequences using the ".." and "..." range operators. The two-dot form creates an inclusive range, while the three-dot form creates a range that excludes the specified high value.

```
(1..5)        #==> 1, 2, 3, 4, 5
(1...5)       #==> 1, 2, 3, 4
('a'..'d')    #==> 'a', 'b', 'c', 'd'
```

The sequence **1..100** is held as a Range **object** containing references to two **Fixnum** objects. If you need to, you can convert a range to a list using the to_a method. Try the following example –

```
#!/usr/bin/ruby

$, =", "   # Array value separator
range1 = (1..10).to_a
range2 = ('bar'..'bat').to_a

puts "#{range1}"
puts "#{range2}"
```

This will produce the following result −

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
["bar", "bas", "bat"]
```

Ranges implement methods that let you iterate over them and test their contents in a variety of ways –

```
#!/usr/bin/ruby

# Assume a range
digits = 0..9

puts digits.include?(5)
ret = digits.min
puts "Min value is #{ret}"

ret = digits.max
puts "Max value is #{ret}"

ret = digits.reject {|i| i < 5 }
puts "Rejected values are #{ret}"

digits.each do |digit|
   puts "In Loop #{digit}"
end
```

This will produce the following result −

```
true
Min value is 0
Max value is 9
Rejected values are 5, 6, 7, 8, 9
In Loop 0
In Loop 1
In Loop 2
In Loop 3
In Loop 4
In Loop 5
In Loop 6
In Loop 7
In Loop 8
In Loop 9
```

Ranges as Conditions

Ranges may also be used as conditional expressions. For example, the following code fragment prints sets of lines from the standard input, where the first line in each set contains the word start and the last line the word ends –

```
while gets
   print if /start/../end/
end
```

Ranges can be used in case statements −

```
#!/usr/bin/ruby

score = 70

result = case score
   when 0..40 then "Fail"
   when 41..60 then "Pass"
   when 61..70 then "Pass with Merit"
   when 71..100 then "Pass with Distinction"
   else "Invalid Score"
end

puts result
```

This will produce the following result −

```
Pass with Merit
```

Ranges as Intervals

A final use of the versatile range is as an interval test: seeing if some value falls within the interval represented by the range. This is done using ===, the case equality operator.

```
#!/usr/bin/ruby

if ((1..10) === 5)
   puts "5 lies in (1..10)"
end

if (('a'..'j') === 'c')
   puts "c lies in ('a'..'j')"
end

if (('a'..'j') === 'z')
   puts "z lies in ('a'..'j')"
end
```

This will produce the following result −

```
5 lies in (1..10)
c lies in ('a'..'j')
```

Ruby – Iterators

Iterators are nothing but methods supported by collections. Objects that store a group of data members are called collections. In Ruby, arrays and hashes can be termed collections. Iterators return all the elements of a collection, one after the other. We will be discussing two iterators here, each and collect. Let's look at these in detail.

Ruby each Iterator

The each iterator returns all the elements of an array or a hash.

Syntax

```
collection.each do |variable|
   code
end
```

Executes code for each element in collection. Here, collection could be an array or a ruby hash.

Example

```
#!/usr/bin/ruby

ary = [1,2,3,4,5]
ary.each do |i|
   puts i
end
```

This will produce the following result −

```
1
2
3
4
5
```

You always associate the each iterator with a block. It returns each value of the array, one by one, to the block. The value is stored in the variable i and then displayed on the screen.

Ruby collect Iterator

The collect iterator returns all the elements of a collection.

Syntax

```
collection = collection.collect
```

The collect method need not always be associated with a block. The collect method returns the entire collection, regardless of whether it is an array or a hash.

Example

```
#!/usr/bin/ruby

a = [1,2,3,4,5]
b = Array.new
b = a.collect
puts b
```

This will produce the following result −

```
1
2
3
4
5
```

NOTE − The collect method is not the right way to do copying between arrays. There is another method called a clone, which should be used to copy one array into another array. You normally use the collect method when you want to do something with each of the values to get the new array. For example, this code produces an array b containing 10 times each value in a.

```
#!/usr/bin/ruby

a = [1,2,3,4,5]
b = a.collect{|x| 10*x}
puts b
```

This will produce the following result −

```
10
20
30
40
50
```

Ruby - File I/O

Ruby provides a whole set of **I/O**-related methods implemented in the Kernel module. All the I/O methods are derived from the class IO.

The class IO provides all the basic methods, such as **read**, **write**, **gets**, **puts**, **readline**, **getc**, and **printf**.

This chapter will cover all the basic I/O functions available in Ruby. For more functions, please refer to Ruby Class IO.

The puts Statement

In the previous chapters, you have assigned values to variables and then printed the output using puts statement.

The puts statement instructs the program to display the value stored in the variable. This will add a new line at the end of each line it writes.

Example

```
#!/usr/bin/ruby

val1 = "This is variable one"
val2 = "This is variable two"
puts val1
puts val2
```

This will produce the following result −

```
This is variable one
This is variable two
```

The gets Statement

The gets statement can be used to take any input from the user from standard screen called STDIN.

Example

The following code shows you how to use the gets statement. This code will prompt the user to enter a value, which will be stored in a variable val and finally will be printed on STDOUT.

```
#!/usr/bin/ruby

puts "Enter a value :"
val = gets
puts val
```

This will produce the following result −

```
This is variable one
This is variable two
```

The gets Statement

The gets statement can be used to take any input from the user from standard screen called STDIN.

Example

The following code shows you how to use the gets statement. This code will prompt the user to enter a value, which will be stored in a variable val and finally will be printed on STDOUT.

```
#!/usr/bin/ruby

puts "Enter a value :"
val = gets
puts val
```

This will produce the following result −

```
Enter a value :
This is entered value
This is entered value
```

The putc Statement

Unlike the puts statement, which outputs the entire string onto the screen, the putc statement can be used to output one character at a time.

Example

The output of the following code is just the character H −

```
#!/usr/bin/ruby

str = "Hello Ruby!"
putc str
```

This will produce the following result −

```
H
```

The print Statement

The print statement is similar to the puts statement. The only difference is that the puts statement goes to the next line after printing the contents, whereas with the print statement the cursor is positioned on the same line.

Example

```
#!/usr/bin/ruby

print "Hello World"
print "Good Morning"
```

This will produce the following result –

```
Hello WorldGood Morning
```

Opening and Closing Files

Until now, you have been reading and writing to the standard input and output. Now, we will see how to play with actual data files.

The File.new Method

You can create a File object using **File.new** method for reading, writing, or both, according to the mode string. Finally, you can use **File.close** method to close that file.

Syntax

```
aFile = File.new("filename", "mode")
   # ... process the file
aFile.close
```

The File.open Method

You can use **File.open** method to create a new file object and assign that file object to a file. However, there is one difference in between File.open and File.new methods. The difference is that the **File.open** method can be associated with a block, whereas you cannot do the same using the **File.new** method.

```
File.open("filename", "mode") do |aFile|
   # ... process the file
end
```

Reading and Writing Files

The same methods that we've been using for 'simple' I/O are available for all file objects. So, gets reads a line from standard input, and aFile.gets reads a line from the file object aFile.

However, I/O objects provides additional set of access methods to make our lives easier.

The sysread Method

You can use the method sysread to read the contents of a file. You can open the file in any of the modes when using the method sysread. For example −

Following is the input text file −

```
This is a simple text file for testing purpose.
```

Now let's try to read this file –

```
#!/usr/bin/ruby

aFile = File.new("input.txt", "r")
if aFile
   content = aFile.sysread(20)
   puts content
else
   puts "Unable to open file!"
end
```

This statement will output the first 20 characters of the file. The file pointer will now be placed at the 21st character in the file.

The syswrite Method

You can use the method **syswrite** to write the contents into a file. You need to open the file in write mode when using the method **syswrite**. For example –

```
#!/usr/bin/ruby

aFile = File.new("input.txt", "r+")
if aFile
   aFile.syswrite("ABCDEF")
else
   puts "Unable to open file!"
end
```

This statement will write "ABCDEF" into the file.

The each_byte Method

This method belongs to the class File. The method **each_byte** is always associated with a block. Consider the following code sample –

```
#!/usr/bin/ruby

aFile = File.new("input.txt", "r+")
if aFile
   aFile.syswrite("ABCDEF")
   aFile.each_byte {|ch| putc ch; putc ?. }
else
   puts "Unable to open file!"
end
```

Characters are passed one by one to the variable ch and then displayed on the screen as follows −

```
s. .a. .s.i.m.p.l.e. .t.e.x.t. .f.i.l.e. .f.o.r. .t.e.s.t.i.n.g. .p.u.r.p.o.s.e
...
```

The IO.readlines Method

The class File is a subclass of the class IO. The class IO also has some methods, which can be used to manipulate files.

One of the IO class methods is **IO.readlines**. This method returns the contents of the file line by line. The following code displays the use of the method **IO.readlines** –

```
#!/usr/bin/ruby

arr = IO.readlines("input.txt")
puts arr[0]
puts arr[1]
```

In this code, the variable **arr** is an array. Each line of the file input.txt will be an element in the array arr. Therefore, **arr[0]** will contain the first line, whereas arr[1] will contain the second line of the file.

The IO.foreach Method

This method also returns output line by line. The difference between the method foreach and the method **readlines** is that the method foreach is associated with a block. However, unlike the method **readlines**, the method foreach does not return an array. For example –

```
#!/usr/bin/ruby

IO.foreach("input.txt"){|block| puts block}
```

This code will pass the contents of the file test line by line to the variable block, and then the output will be displayed on the screen.

Renaming and Deleting Files

You can rename and delete files programmatically with Ruby with the rename and delete methods.

Following is the example to rename an existing file test1.txt –

```
#!/usr/bin/ruby

# Rename a file from test1.txt to test2.txt
File.rename( "test1.txt", "test2.txt" )
```

Following is the example to delete an existing file test2.txt −

```
#!/usr/bin/ruby

# Delete file test2.txt
File.delete("test2.txt")
```

File Modes and Ownership

Use the **chmod** method with a mask to change the mode or permissions/access list of a file

Following is the example to change mode of an existing file test.txt to a mask value −

```
#!/usr/bin/ruby

file = File.new( "test.txt", "w" )
file.chmod( 0755 )
```

File Inquiries

The following command tests whether a file exists before opening it –

```
#!/usr/bin/ruby

File.open("file.rb") if File::exists?( "file.rb" )
```

The following command inquire whether the file is really a file −

```
#!/usr/bin/ruby

# This returns either true or false
File.file?( "text.txt" )
```

The following command finds out if the given file name is a directory −

```
#!/usr/bin/ruby

# a directory
File::directory?( "/usr/local/bin" ) # => true

# a file
File::directory?( "file.rb" ) # => false
```

The following command finds whether the file is readable, writable or executable −

```
#!/usr/bin/ruby

File.readable?( "test.txt" )   # => true
File.writable?( "test.txt" )   # => true
File.executable?( "test.txt" ) # => false
```

The following command finds whether the file has zero size or not −

```
#!/usr/bin/ruby

File.zero?( "test.txt" )      # => true
```

The following command returns size of the file −

```
#!/usr/bin/ruby

File.size?( "text.txt" )     # => 1002
```

The following command can be used to find out a type of file −

```
#!/usr/bin/ruby

File::ftype( "test.txt" )     # => file
```

The **ftype** method identifies the type of the file by returning one of the following − file, directory, **characterSpecial**, **blockSpecial**, **fifo**, link, socket, or unknown.

The following command can be used to find when a file was created, modified, or last accessed –

```
#!/usr/bin/ruby

File::ctime( "test.txt" ) # => Fri May 09 10:06:37 -0700 2008
File::mtime( "text.txt" ) # => Fri May 09 10:44:44 -0700 2008
File::atime( "text.txt" ) # => Fri May 09 10:45:01 -0700 2008
```

Directories in Ruby

All files are contained within various directories, and Ruby has no problem handling these too. Whereas the File class handles files, directories are handled with the Dir class.

Navigating Through Directories

To change directory within a Ruby program, use **Dir.chdir** as follows. This example changes the current directory to **/usr/bin**.

```
Dir.chdir("/usr/bin")
```

You can find out what the current directory is with Dir.pwd –

```
puts Dir.pwd # This will return something like /usr/bin
```

You can get a list of the files and directories within a specific directory using **Dir.entries**

```
puts Dir.entries("/usr/bin").join(' ')
```

Dir.entries returns an array with all the entries within the specified directory. **Dir.foreach** provides the same feature –

```
Dir.foreach("/usr/bin") do |entry|
   puts entry
end
```

An even more concise way of getting directory listings is by using Dir's class array method

```
Dir["/usr/bin/*"]
```

Creating a Directory

The **Dir.mkdir** can be used to create directories –

```
Dir.mkdir("mynewdir")
```

You can also set permissions on a new directory (not one that already exists) with mkdir

NOTE – The mask 755 sets permissions owner, group, world [anyone] to **rwxr-xr-x** where r = read, w = write, and x = execute.

```
Dir.mkdir( "mynewdir", 755 )
```

Deleting a Directory

The **Dir.delete** can be used to delete a directory. The **Dir.unlink** and **Dir.rmdir** performs exactly the same function and are provided for convenience.

```
Dir.delete("testdir")
```

Creating Files & Temporary Directories

Temporary files are those that might be created briefly during a program's execution but aren't a permanent store of information.

Dir.tmpdir provides the path to the temporary directory on the current system, although the method is not available by default. To make **Dir.tmpdir** available it's necessary to use require 'tmpdir'.

You can use **Dir.tmpdir** with **File.join** to create a platform-independent temporary file

```
require 'tmpdir'
   tempfilename = File.join(Dir.tmpdir, "tingtong")
   tempfile = File.new(tempfilename, "w")
   tempfile.puts "This is a temporary file"
   tempfile.close
   File.delete(tempfilename)
```

This code creates a temporary file, writes data to it, and deletes it. Ruby's standard library also includes a library called **Tempfile** that can create temporary files for you –

```
require 'tempfile'
   f = Tempfile.new('tingtong')
   f.puts "Hello"
   puts f.path
   f.close
```

Built-in Functions

Here are the ruby built-in functions to process files and directories −

- File Class and Methods.
- Dir Class and Methods.

Ruby – Exceptions

The execution and the exception always go together. If you are opening a file, which does not exist, then if you did not handle this situation properly, then your program is considered to be of bad quality.

The program stops if an exception occurs. So, exceptions are used to handle various type of errors, which may occur during a program execution and take appropriate action instead of halting program completely.

Ruby provide a nice mechanism to handle exceptions. We enclose the code that could raise an exception in a begin/end block and use rescue clauses to tell Ruby the types of exceptions we want to handle.

Syntax

```
begin
# -
rescue OneTypeOfException
# -
rescue AnotherTypeOfException
# -
else
# Other exceptions
ensure
# Always will be executed
end
```

Everything from begin to rescue is protected. If an exception occurs during the execution of this block of code, control is passed to the block between rescue and end.

For each rescue clause in the begin block, Ruby compares the raised Exception against each of the parameters in turn. The match will succeed if the exception named in the rescue clause is the same as the type of the currently thrown exception, or is a superclass of that exception.

In an event that an exception does not match any of the error types specified, we are allowed to use an else clause after all the rescue clauses.

Example

```
#!/usr/bin/ruby

begin
   file = open("/unexistant_file")
   if file
      puts "File opened successfully"
   end
rescue
      file = STDIN
end
print file, "==", STDIN, "\n"
```

This will produce the following result. You can see that **STDIN** is substituted to file because **open** failed.

```
#<IO:0xb7d16f84>==#<IO:0xb7d16f84>
```

Using retry Statement

You can capture an exception using rescue block and then use retry statement to execute begin block from the beginning.

Syntax

```
begin
   # Exceptions raised by this code will
   # be caught by the following rescue clause
rescue
   # This block will capture all types of exceptions
   retry  # This will move control to the beginning of begin
end
```

Example

```ruby
#!/usr/bin/ruby

begin
   file = open("/unexistant_file")
   if file
      puts "File opened successfully"
   end
rescue
   fname = "existant_file"
   retry
end
```

The following is the flow of the process −

- An exception occurred at open.
- Went to rescue. **fname** was re-assigned.
- By retry went to the beginning of the begin.
- This time file opens successfully.
- Continued the essential process.

NOTE − Notice that if the file of re-substituted name does not exist this example code retries infinitely. Be careful if you use retry for an exception process.

Using raise Statement

You can use raise statement to raise an exception. The following method raises an exception whenever it's called. It's second message will be printed.

Syntax

```
raise

OR

raise "Error Message"

OR

raise ExceptionType, "Error Message"

OR

raise ExceptionType, "Error Message" condition
```

The first form simply re-raises the current exception (or a RuntimeError if there is no current exception). This is used in exception handlers that need to intercept an exception before passing it on.

The second form creates a new **RuntimeError** exception, setting its message to the given string. This exception is then raised up the call stack.

The third form uses the first argument to create an exception and then sets the associated message to the second argument.

The fourth form is similar to the third form but you can add any conditional statement like unless to raise an exception.

Example

```
#!/usr/bin/ruby

begin
   puts 'I am before the raise.'
   raise 'An error has occurred.'
   puts 'I am after the raise.'
rescue
   puts 'I am rescued.'
end
puts 'I am after the begin block.'
```

This will produce the following result −

```
I am before the raise.
I am rescued.
I am after the begin block.
```

One more example showing the usage of raise −

```
#!/usr/bin/ruby

begin
   raise 'A test exception.'
rescue Exception => e
   puts e.message
   puts e.backtrace.inspect
end
```

This will produce the following result −

```
A test exception.
["main.rb:4"]
```

Using ensure Statement

Sometimes, you need to guarantee that some processing is done at the end of a block of code, regardless of whether an exception was raised. For example, you may have a file open on entry to the block and you need to make sure it gets closed as the block exits.

The ensure clause does just this. ensure goes after the last rescue clause and contains a chunk of code that will always be executed as the block terminates. It doesn't matter if the block exits normally, if it raises and rescues an exception, or if it is terminated by an uncaught exception, the ensure block will get run.

Syntax

```
begin
   #.. process
   #..raise exception
rescue
   #.. handle error
ensure
   #.. finally ensure execution
   #.. This will always execute.
end
```

Example

```
begin
   raise 'A test exception.'
rescue Exception => e
   puts e.message
   puts e.backtrace.inspect
ensure
   puts "Ensuring execution"
end
```

This will produce the following result −

```
A test exception.
["main.rb:4"]
Ensuring execution
```

Using else Statement

If the else clause is present, it goes after the rescue clauses and before any ensure.

The body of an else clause is executed only if no exceptions are raised by the main body of code.

Syntax

```
begin
   #.. process
   #..raise exception
rescue
   # .. handle error
else
   #.. executes if there is no exception
ensure
   #.. finally ensure execution
   #.. This will always execute.
end
```

Example

```
begin
   # raise 'A test exception.'
   puts "I'm not raising exception"
rescue Exception => e
   puts e.message
   puts e.backtrace.inspect
else
   puts "Congratulations-- no errors!"
ensure
   puts "Ensuring execution"
end
```

This will produce the following result −

```
I'm not raising exception
Congratulations-- no errors!
Ensuring execution
```

Raised error message can be captured using $! variable.

Catch and Throw

While the exception mechanism of raise and rescue is great for abandoning the execution when things go wrong, it's sometimes nice to be able to jump out of some deeply nested construct during normal processing. This is where catch and throw come in handy.

The catch defines a block that is labeled with the given name (which may be a Symbol or a String). The block is executed normally until a throw is encountered.

Syntax

```
throw :lablename
#.. this will not be executed
catch :lablename do
#.. matching catch will be executed after a throw is encountered.
end

OR

throw :lablename condition
#.. this will not be executed
catch :lablename do
#.. matching catch will be executed after a throw is encountered.
end
```

Example

The following example uses a throw to terminate interaction with the user if '!' is typed in response to any prompt.

```
def promptAndGet(prompt)
   print prompt
   res = readline.chomp
   throw :quitRequested if res == "!"
   return res
end

catch :quitRequested do
   name = promptAndGet("Name: ")
   age = promptAndGet("Age: ")
   sex = promptAndGet("Sex: ")
   # ..
   # process information
end
promptAndGet("Name:")
```

You should try the above program on your machine because it needs manual interaction. This will produce the following result –

```
Name: Ruby on Rails
Age: 3
Sex: !
Name:Just Ruby
```

Class Exception

Ruby's standard classes and modules raise exceptions. All the exception classes form a hierarchy, with the class Exception at the top. The next level contains seven different types

- Interrupt
- NoMemoryError
- SignalException
- ScriptError
- StandardError
- SystemExit

There is one other exception at this level, Fatal, but the Ruby interpreter only uses this internally.

Both **ScriptError** and **StandardError** have a number of subclasses, but we do not need to go into the details here. The important thing is that if we create our own exception classes, they need to be subclasses of either class Exception or one of its descendants.

Let's look at an example –

```
class FileSaveError < StandardError
   attr_reader :reason
   def initialize(reason)
      @reason = reason
   end
end
```

Now, look at the following example, which will use this exception −

```
File.open(path, "w") do |file|
begin
   # Write out the data ...
rescue
   # Something went wrong!
   raise FileSaveError.new($!)
end
end
```

The important line here is raise **FileSaveError.new($!)**. We call raise to signal that an exception has occurred, passing it a new instance of **FileSaveError**, with the reason being that specific exception caused the writing of the data to fail.

REFERENCE

1- Schmager, Frank, Nicholas Cameron, and James Noble. "GoHotDraw: Evaluating the Go programming language with design patterns." Evaluation and Usability of Programming Languages and Tools. 2010. 1-6.

2- Pike, Rob. "The go programming language." Talk given at Google's Tech Talks (2009).

3- Balbaert, Ivo. The way to Go: A thorough introduction to the Go programming language. IUniverse, 2012.

4- Flanagan, David, and Yukihiro Matsumoto. The Ruby Programming Language: Everything You Need to Know. " O'Reilly Media, Inc.", 2008.

5- Matsumoto, Yukio, and K. Ishituka. "Ruby programming language." (2002).

6- https://www.tutorialspoint.com/ruby/index.htm

7- https://www.tutorialspoint.com/go/index.htm