

The Dragon programming language

DRAGON BOOK - EDITION 1

Programming dragon language (first edition)

**THE DRAGON PROGRAMMING
LANGUAGE -
PROGRAMMING DRAGON LANGUAGE**

FIRST EDITION - EDITION 1



Author: Adesina Festus

Community: Dragon Developmental Team

Date: 8/2/2020

Github: <https://dragon-language-projects>

The Dragon Programming Language

Dragon is an innovative and practical general-purpose, multi-paradigm scripting language. It supports multiple programming paradigms such as imperative, object-oriented, functional, natural

programming, and declarative programming using nested structures. The language is portable (Windows, Linux, Mac OS X, Android, etc.) and can be used to create console and GUI applications. The language is designed to be simple, small, flexible, and fast. It's a dynamically- and weakly-typed language that interprets the source code through the JVM or LLVM. The language was also made to be a scripting language and server side programming. The first version of the language was released on January 4th, 2018. The language was influenced by Lua, Python, Java, Ruby, QML.



Overview	
Paradigm	Imperative & Procedural, Small, Simple & Fast
Designed by	Aavesh J ilani
First Appeared	J anuary 4th, 2018
Stable release	v1.9.7/ April 28, 2020
Typing discipline	Dynamic, Weak
Operating System (OS)	Linux, macOS, Microsoft windows, Android
License	The MIT License
Filename extension	.dgn
Language Type	Dynamic Type and Weakly Typed language
Influenced by	Lua, Python, J ava, QML, Ring, Ruby
Website	https://dragon-lang.org

HISTORY

In Oct. 2016, Aavesh Jilani (creator) started the design and the implementation of the Dragon programming language. After 15 months of development, In Jan 2018 the language Interpreter and environment were ready for use!

The first version of the language Dragon 1.0 was released in January 4th, 2018

Dragon 1.1 was released on March 6th, 2018

Dragon 1.2 was released on May 24th, 2018

Dragon 1.3 was released on August 14th, 2018

Dragon 1.4 was released on January 18th, 2019

Dragon 1.5 was released on April 25th, 2019

Dragon 1.6 was released on May 27th 2019

Dragon 1.7 was released on July 11th 2019

Dragon 1.8 was released on August 9th 2019

Dragon 1.9 was released on August 29th 2019

Dragon 1.9.1 was released on September 15th 2019

Dragon 1.9.2 was released on September 18th 2019

Dragon 1.9.3 was released on October 13th 2019

Dragon 1.9.4 was released on October 22nd 2019

Dragon-native was released on January 7th 2020

Dragon 1.9.5 was released on February 20th 2020

Dragon 1.9.6 was released on February 28th 2020

Dragon 1.9.7 was released on April 28th 2020

Dragon Versions:

Dragon programming language has two types of versions:

Dragon native: The dragon native works on LLVM (Low Level Virtual Machine) which doesn't require JVM (Java Virtual Machine)

Dragon normal: The dragon normal which works on JVM (Java Virtual Machine), it comes with an inbuilt JVM which does not require JRE (Java Runtime Environment) installation.

Dragon and other Programming Languages

Dragon is an innovative programming language that comes with better support for Natural Language Programming and Declarative Programming. The innovation comes in supporting these paradigms with new practical techniques on the top of Object-Oriented Programming and Functional Programming.



Also Dragon is also influenced by the next programming languages:

- * Lua
- * Python
- * Ruby
- * Java
- * QML
- * Ring

Features of Dragon programming language

The Dragon language comes with the next features

Tip The language is ready for production!

1. Interpreter
2. Declarative programming on the top of Object-Oriented programming
3. No explicit end for statements (No ; or ENTER is required)
4. Portable (Windows, Linux & Mac OS X, Android)
5. Comments (One line, In-line & Multi-lines)
6. Dynamic Typing
7. Weakly typed
8. Default scope for variables inside functions (Local)
9. Default scope for variables outside functions (global)
10. Garbage Collector - Automatic Memory Management (Escape Analysis and Reference Counting)
11. Structure Programming
12. Rich control structures & Operators
13. No Main Function
14. Call Function before the definition
15. Recursion
16. Multi-line literals
17. Reflection and Meta-programming
18. Clear program structure (Statements then functions)
19. Exception Handling
20. I/O commands
21. Math functions
22. String functions
23. Standard functions
24. File processing functions
25. Database support
26. Create GUI Applications for Desktop and Android
27. .dgn file extension

Table of Contents:

REQUIREMENT

- ~ Installation
- ~ Dragon on Android
- ~ IDE (Integrated Development Environment)
- ~ Quick reference

❖ Getting Started

- o Hello World 13
- o Run the Program 13
- o Multi-line Literals 14
- o Getting Inputs15 - 16
- o Writing Comments 17

❖ Variable

- o Variables 18
- o Dynamic typing19 - 20
- o Deep copy21 - 23
- o Weekly typed24 - 25

❖ Data types

- o Number 26 - 28
- o Strings 29 - 30
- o Arrays 31 - 33
- o Map - Objects (an associative array) 34 - 35
- o Boolean (true or false) 36

❖ Operators

- o Operators 37 - 38
- o Arithmetic Operators 38 - 40
- o Logical Operators 41 - 43
- o Bitwise Operators 43 - 45
- o Assignment Operators 46 - 48
- o Conditional Operators (if... else... else if)49 - 51
- o Operator Precedence 52 - 53

❖ Control Structures

- o Control Structures 54
- o Branching54
- o Looping 55
- o While loop 55
- o Do while Loop 56
- o For loop 57
- o ForEach Loop 57
- o Continue statement 58
- o Break statement 58

❖ **Functions**

o Functions	59
o Define Functions	59
o Call Functions	60
o Declare Parameters	60
o Send Parameters	61
o Variables Scope	61
o Return Value.	62

❖ **Classes**

o Classes	62 - 65
-----------------	---------

❖ **Modules**

o std	66 - 81
o types	82 - 84
o math	85 - 98
o date	99 - 101
o files	102 - 119
o http	120 - 121
o base64	122 - 123
o json	124
o yaml	125
o ounit	126 - 128
o GUI	129 - 131

❖ Installation

Dragon language now has the latest version which is v1.9.7, it requires installation processes. To install the latest version of Dragon on your windows or Linux operating system. You should consider using the Dragon website which provides the latest version of the language by using this link <https://dragon-lang.org/dragon-197> and choose which one to download, either for windows or Linux both 64bit and 32bit. Or simply head to <https://dragon-lang.org> (If you're on mobile, click the menu bar, click on downloads and choose which version to download).

THE Dragon programming language can be run on windows or Linux operating system simply by using the ZDragon IDE for Dragon.

ZDragon IDE: is an Integrated Development Environment used for running Dragon codes and displays its output. It was originally developed by one of the Dragon Developmental Team named Richard Dan. The purpose of the IDE is for running Dragon codes, creating console and GUI applications on Windows or Linux Operating systems.

Features of ZDragon IDE: The ZDragon IDE comes with lots of inbuilt functionalities or features that make the IDE great, easy and faster for running Dragon. Some of these features include:

1. Syntax Highlighting
2. Word suggestions
3. Dark Mode
4. Flexibility
5. Auto Save
6. Create files, edit files & delete files option
7. Copy & Paste
8. Undo & Redo program
9. Cut
10. Search option
11. Tools option
12. Help option
13. Close and open programs
14. File tab
15. Next & Previous tab
16. Source
17. Navigation bar (easily access your projects)
18. Shows number of codes written
19. Run program on IDE
20. Run program in terminal
21. .dgn file extension run for Dragon (This way makes it run Dragon programs)

How to install ZDragon IDE in Linux:

To install Dragon ZDragon IDE in Linux, simply follow the below steps and type the text in command line to install.

```
wget https://dragon-lang.org/installZDragon1.6 && bash installZDragon1.6
```

❖ Dragon on Android (quick guide)

Since Dragon is an innovative and practical general-purpose programming language used for creating console and GUI applications, and it supports different platforms such as Windows, MacOS X, Linux and Android and the language was made to be easy, fast, flexible and simple syntax for immediate development without boilerplate codes, this language thus has made it possible to run on Android platforms, this way you'll be able to fully create GUI applications using Dragon on Android devices.

The Android development processes or installation processes comes thus in a variety of ways for developing applications on your mobile devices, not just only mobile applications, also for server side programming, front-end development (web scripting). The Web Scripting features that comes with Dragon is initially called or known as **DragonJS, DragonJS** was introduced in Dragon for programming web functionalities which is known as Scripting language. This feature was thus introduced to make development more easier without boilerplate programs on the web for quicker functionality that JS wouldn't provide.

The language also can be used for server side programming, working with databases for storing, retrieving and sending information on the server and it works with SQL.

To use Dragon programming language on android is quite easy to install using terminal apps for android like Termux or Userland.

Termux: Termux is an Android terminal emulator and Linux environment app that works directly with no rooting or setup required. A minimal base system is installed automatically - additional packages are available using the APT package manager. Termux allows you to run Dragon programs and outputs its programs, this way thus made Dragon more accessible to any platform and compatible with Android to run and execute codes. Though GUI applications cannot be developed or built with Termux as it only allows you to code and run Dragon codes.

Installation in termux: To install Dragon in termux, download and install the app from play store or any other apk stores. Install and open up the app and type the following commands. You can make reference to the screenshot below:

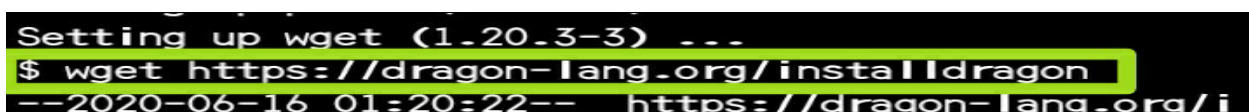
>_ pkg install wget

And press "Enter" key.

A screenshot of the Termux terminal interface. The command '\$ pkg install wget' is entered at the prompt. Below the command line, there is a row of touchable icons: ESC, a back arrow, CTRL, ALT, a horizontal line, a down arrow, and an up arrow.

>_ wget https://dragon-lang.org/installdragon

And press "Enter" key.

A screenshot of the Termux terminal showing the execution of the 'wget' command. The first line says 'Setting up wget (1.20.3-3) ...'. The second line shows the command '\$ wget https://dragon-lang.org/installdragon' with the URL highlighted in green. The third line shows the progress bar '--2020-06-16 01:20:22-- https://dragon-lang.org/i'.

```
>_ bash installdragon
```

```
$ bash installdragon
```

And press "Enter" key.

After this command, termux will start downloading Dragon. It is about 45mb storage it takes. If installation is successful, termux will display a message that depicts that the installation was successful and you're ready to use Dragon in termux Android terminal app. You need a simple text editor to write, edit, save and run dragon codes in termux after installation. Download nano text editor, download by running these commands:

```
>_ pkg install nano
```

After nano is successfully installed, type

```
>_ nano yourfilename.dgn
```

yourfilename.dgn is the name of the file you created in nano editor. Note that dragon saves file using .dgn file extension.

To display a simple hello world program in Dragon using termux. Type in nano editor:

```
show "hello world"
```

And press CTRL + O to save your life and run the program. To exit nano editor. Type CTRL + X

UserLand: UserLand is a free, open-source tool that enables you to install and manage Linux applications on your Android device like you would a native app and to also run full Linux distributions e.g. Ubuntu, Kali Linux, Debian, Etc. With Userland, you can develop/built GUI applications using dragon programming language.

Installation in Userland: To install and use dragon programming language to develop GUI applications in Userland:

1. Download Userland from play store or any other apk stores
2. Install and open Userland and then install Ubuntu distribution in Userland
3. Select VNC Type in Ubuntu
4. Install J DK in Ubuntu
5. Install Dragon
6. Now you're done installing dragon in UserLand, and can fully use and develop GUI applications fully in dragon

❖ **Quick Reference**

This book is recommended for both absolute beginners, intermediate or advanced learners..

o Hello World

This part of the tutorial is about core Dragon, the language itself.
Welcome to the Dragon programming language!

The following program prints the "Hello World" message on the screen (std-out).

```
show "Hello, World" // Hello, World
```

In dragon, semi-colon (;) is not necessary, and also brackets () are not necessary. Both **show 'hello word';** or **show ('hello word')** will result in a same output

Run the Program:

To run the program, save the code in a file (for example: hello.dgn), then from the command line or terminal, run it using the Dragon Interpreter:

```
dragon -r hello.dgn
```

Multi-line Literals:

Using Dragon we can write multi-line literals, as in this example:

```
show "  
    Hello  
    Yo, I am Learning Dragon programming language  
    It fun  
    "/* Hello  
        Yo, I am Learning Dragon programming language  
        it fun */
```

OR:

```
showln "Hello\nWorld\nFrom Dragon"
```

in Dragon \n is used for line break

In dragon, there are two methods of outputting text, usually by using **show()** or **showln()**
show() writes to the program without appending a newline on the end.

While:

The **showln()** method is used to write a program with additional property of newline character after each statement. This method is similar to the **show()** method.

```
show "hello, world! "  
show "hello, dragon! " // hello, world!hello, dragon!
```

The **show** or **show()** method write to the program without appending a newline.

While:

```
showln "hello, world"  
showln "hello, dragon"  
// hello, word  
// hello, dragon
```

showln() writes to the program appending a newline at the end.

Getting Input:

User defined inputs are used with **readln** function. Which is in **std** module. by default the input types are string. The **readln** is an input function used for collecting something or information or some data from the user or used for getting input from the user.

```
select "std"  
showln "Enter your name"  
a = readln()  
show "Welcome " + a
```

No Explicit End For Statements:

You don't need to use ';' or ENTER to separate statements. Some lines in the previous program that were separate can be combined on one line:

```
select "std"  
  
showln "Enter your name"  
a = readln() show "Hello " + a
```

Writing Comments:

In computer programming, a comment is a programmer-readable explanation or annotation in the source code of a computer program. They are added with the purpose of making the source code easier for humans to understand, and are generally ignored by compilers and interpreters. The syntax of comments in various programming languages varies considerably. Comments are sometimes processed in various ways to generate documentation external to the source code itself by documentation generators , or used for integration with source code management systems and other kinds of external programming tools . The flexibility provided by comments allows for a wide degree of variability, but formal conventions for their use are commonly part of programming style guides.

In Dragon, Comments are generally formatted as either block comments (also called prologue comments or stream comments) or line comments (also called inline comments).

1. Block Comments delimit a region of source code which may span multiple lines or a part of a single line. This region is specified with a start delimiter and an end delimiter For example, Dragon has block comments delimited by `/*` and `*/` that can span multiple lines.

```
/* This is a block comments  
   or prologue comments  
   and can span  
   multiple lines  
*/
```

2. Line comments either start with a comment delimiter and continue until the end of the line, or in some cases, start at a specific column (character line offset) in the source code, and continue until the end of the line. Note that a line comments cannot span multiple lines. Line comment can be repeated multiple time but cannot span multiple lines. in Dragon, Line comment is delimited by // that can only span a single line.

```
// Line comment  
  
// Line comments can be  
// repeated multiple times  
// but cannot span  
// multiple lines
```


Including comments in your program:

Since comments are omitted by the interpreter or IDE in your program, you can use comment to explain the function of the source code in your program in order to make it easier to be read by humans or by other developers who review your programs and wants to reuse them.

```
select "std"

showln "Enter your age"
a = readln() // print message on screen and get input from the user
show "Age: " + a // Output age!

// Program executed!!!
```

Or simply by using Block comments for spanning multiple lines in your program

```
select "std"
/* This program will
   request for user input and
   print message on screen */

showln "Enter your age"
a = readln()
show "Age: " + a
```

Summary:

- **show** and **showln** are used for outputting text in Dragon programs.
- Semi-colon or brackets are not necessary in Dragon.
- With Dragon, you can write multi-line literals.
- `\n` is used for line break in Dragon, which in return can be used for Multi-line Literals.
- **showln** writes to a program appending a newline at the end of statement.
- **show** writes to a program without appending a newline at the end of the statement.
- **readln** is used for getting input from users.
- Block comments is delimited by `/*` and `*/` and can span multiple lines.
- Line or inline comments is delimited by `//` and cannot span multiple lines.
- Comments are ignored by the interpreter or IDE and won't be executed.

o Variables

One of the most fundamental characteristics of a programming language is the set of data types it supports. These are the types of values that can be represented and manipulated in a programming language.

A variable in any programming language is a named piece of computer memory, containing some information inside.

In computer programming, a variable is also a storage address (identified by a memory address) paired with an associated symbolic name, which contains some known or unknown quantity of information referred to as a value. The variable name is the usual way to reference the stored value, in addition to referring to the variable itself, depending on the context. This separation of name and content allows the name to be used independently of the exact information it represents. The identifier in computer source code can be bound to a value during run time, and the value of the variable may thus change during the course of program execution.

A real-life analogy

We can easily grasp the concept of a "variable" if we imagine it as a "box" for data, with a uniquely-named sticker on it.



For instance, the variable message can be imagined as a box labeled "message" with the value "Hello!" in it:

Like in other programming languages, dragon has variables. In other words, a variable can be thought of as a named container. You can place data into these containers and then refer to the data simply by naming the container.

To create a new variable in dragon, you just need to determine the variable name & value. The variable name is where the information will be stored or kept, while the value is the information stored or kept in the variable name. The value will determine the variable type and you can change the value to switch between the types using the same variable name.

Syntax:

```
<Variable Name> = <Value>
```

```
message = "Hello" // Hello
```

Here , **message** is the variable name which is where our information or data is being stored, while **hello** is the value which is the information or data stored in the variable name.

The string is now saved into the memory area associated with the variable. We can access it using the variable name:

```
message = "Hello"  
showln message // Hello
```

Storing a value in a variable is called **variable initialization**.

tip::

The operator '=' is used here as an assignment operator and the same operator can be used in conditions, but for testing equality of expressions. (operators will be taught on chapter 4)

The value will determine the variable type and you can change the value to switch between the types using the same variable name.

note::

The variable will contain the real value (not a reference). This means that once you change the variable value, the old value will be removed from memory (even if the variable contains a list or object).



```
message = "Hello"  
message = "World"  
showln message // World
```

The **show** and **showln** are used to output a text.

White Space insensitive

Whitespace is the stuff you type that is typically invisible on the screen, including spaces, tabs, and carriage returns (end-of-line characters).

Dragon whitespace insensitive means that it almost never matters how many whitespace characters you have in a row. one whitespace character is the same as many such characters.

For example:

```
a = 2 // white space
      b =
      8 // or tabs doesn't matters
      showln a + b // 10
```

Untyped Language

Dragon is an un-typed language. This means that a Dragon variable can hold a value of any data type. Unlike many other languages, you don't have to tell Dragon during

variable declaration what type of value the variable will hold. The value type of a variable

can change during the execution of a program and Dragon takes care of it automatically. This feature is termed as **dynamic typing**.

```
x = "Hello" // x is a string
x = 10 // x is a number (integer)
x = [1, 2, 3] // x is an array
x = 1.25 // x is a floating point number (double)
```

```
data = 10; // it holds a number (integer)
data = 'Hello World!'; // string
showln data // no error caused
```

Weakly typed language

A weakly typed language has looser typing rules and may produce unpredictable results or may perform implicit type conversion at runtime. Or

A weakly-typed language on the other hand is a language in which variables are not bound to a specific data type; they still have a type, but type safety constraints are lower compared to strongly-typed languages.

Dragon is a weakly typed language. This means that the language can automatically convert between data types (like string & numbers) when that conversion makes sense.

Weakly typed languages place no restrictions on how data types can be mixed

```
data = "I love " // string
data = data + 10; // string + number, no error caused
show data // I love 10
```

```
data = "I love " // string
rank = 100 // number
show data + rank // I love 100
```

Strings can be added together with number, when string is being added with a number,

Dragon will understand and automatically concatenate the values.

```
num1 = "1"
num2 = 2
showln num1 + num2 // 12
```

Naming Variables:

There are two limitations on variable names in Dragon:

The name must contain only letters, digits, or the symbols \$ (dollar sign) and _ (underscore).

The first character must not be a digit (number).

Examples of valid names:

```
myName;
number123;
$test
```

Examples of Invalid names: Variables with invalid names will return a `ParseError`:

Unknown expression on the defined line of code /* `ParseError` on line 1: Unknown expression: */

```
123Name // cannot start with a digit
```

```
-myself // hyphens '-' aren't allowed in the name
```

```
my-Test // hyphens '-' aren't allowed in the name
```

```
/* ParseError on line 1: Unknown expression: */
```

Note: Variable names should not start with - (minus sign), digit(numbers) or any other symbol apart from letters, _ (underscore) and \$ (dollar sign).

When the name contains multiple words, camelCase is commonly used. That is: words go one after another, each word except first starting with a capital letter:

Example:

```
customerDetails // example of camelCase letter
```

What's interesting – the dollar sign '\$' and the underscore '_' can also be used in names. They are regular symbols, just like letters, without any special meaning.

These names are valid:

```
$ = 1; // declared a variable with the name "$"
```

```
_ = 5; // and now a variable with the name "_"
```

```
showln($ + _); // 6
```

Case Matters:

Variables named orange and ORANGE are two different variables. Note that Dragon is case sensitive, and cases matters in Dragon.

This will produce an error because case sensitivity matters:

```
orange = "I love orange"
show ORANGE // Error: Variable ORANGE does not exists in main
```

Reserved Words:

While naming your variable name in Dragon, you should keep in mind that variable names should not start with any of the Dragon reserved words. The reserved words are the words that are used by the language itself and which cannot be used as a variable name in your program or else will produce an error. Some of the reserved words include: show, showln, readln, echo, for

Dragon reserved words:

A list of all the Dragon reserved words are given in the following table, they cannot be used as Dragon variables, functions, methods, loops labels or any object names. You don't really need to memorize these reserved words.

show	showln	echo	readln	for
while	else	do	true	false
if	func	break	match	select
continue	std	include	else if	input
charAt	std	addition	indexOf	parseLong
lastIndexOf	join	length	newarray	parseInt
rand	range	replace	replaceAll	replaceFirst
sleep	sort	split	sprintf	time
substring	sync	func	http	thread
toChar	toHexString	toLowerCase	toUpperCase	trim
try	byte	types	double	float
int	long	number	typeof	short
string	extract	count	toArray	limit
custom	skip	stream	reduce	map
dropwhile	flatMap	filter	foreach	urlencode
download	reachChar	reallAllBytes	mkdirs	flush
fclose	fopen	delete	exists	fileSize
isHidden	newDate	newFormat	formatDate	toTimeStamp
isFile	listFiles	isDirectory	mkdir	

Naming Variables the right approach:

Talking about variables, there's one more extremely important thing.

A variable name should have a clean, obvious meaning, describing the data that it stores.

Variable naming is one of the most important and complex skills in programming. A quick glance at variable names can reveal which code was written by a beginner versus an experienced developer.

In a real project, most of the time is spent modifying and extending an existing code base rather than writing something completely separate from scratch. When we return to some code after doing something else for a while, it's much easier to find information that is well-labeled. Or, in other words, when the variables have good names.

Please spend time thinking about the right name for a variable before declaring it. Doing so will repay you handsomely.

Some good-to-follow rules are:

Use human-readable names like `customerName` or `userName`.

Stay away from abbreviations or short names like `a`, `b`, `c`, unless you really know what you're doing. Make names maximally descriptive and concise. Examples of bad names are `data` and `value`. Such names say nothing. It's only okay to use them if the context of the code makes it exceptionally obvious which data or value the variable is referencing.

Agree on terms within your team and in your own mind. If a site visitor is called a "user" then we should name related variables `currentUser` or `newUser` instead of `currentVisitor` or `newManInTown`.

Summary:

- Variables are used for storing pieces of information.
- Variables in Dragon can store any value of data type.
- Variables are accessible by area associated name.
- A variable value can be changed and the old is removed.
- Variable names should only start with letters, `_` (underscore) or `$` (dollar sign)
- Variable names cannot start with a digit. E. g **123test**
- Variable names can be written in camelCase form. E. g **newUser**.
- **apple** and **APPLE** are different thing entirely, because Dragon is case sensitive.
- Reserved Words cannot be a variable name in Dragon.

o Data Types

A data type is a classification of data which tells the compiler or interpreter how the programmer intends to use the data. Most programming languages support various types of data, including integer (number), floating point (real), character or string, objects, arrays and Boolean (true or false).

Discussion

Our interactions (inputs and outputs) with a program are treated in many languages as a stream of bytes. These bytes represent data that can be interpreted as representing values that we understand. Additionally, within a program, we process this data in various ways such as adding them up or sorting them. This data comes in different forms. Examples include:

- your name – a string of characters
- your age – usually an integer
- the amount of money in your pocket – usually a value measured in dollars and cents (something with a fractional part) - floating point. E. g \$12.2
- Some details or credentials about you - such as your country, city, your father's name or mother's name, date of birth (This can be associated with an Object) for storing those multiple or complex data/entities.
- Your Status (Either single or married) - This can be associated with a Boolean (Usually true or false).

A major part of understanding how to design and code programs is centered in understanding the types of data that we want to manipulate and how to manipulate that data.

Common data types in Dragon include:

Data Types	Represents	Examples
integer	whole numbers	-5, 0, 12, 9, 2.
floating point (double, real)	fractional numbers	-3.14159, 0.15, 1.25
string	a sequence of characters	"Hello, world!"
Array	can contain string, objects, integer, floating point (for ordered collection)	fruits = ["Apple","Orange", "Plum", 20, 1.25];
Map - Objects	an associative arrays, for storing complex entities	{"key1":1, "key2":2, "key3": 3 }
Boolean	Logical True or False	True or False

The common data types usually exist in most programming languages and act or behave similarly from language to language. Additional complex and/or composite data types may exist and vary from language to language.

Key Terms

Data types

Defines a set of values and a set of operations that can be applied on those values.

Boolean

A data type representing logical true or false.

Floating point

A data type representing numbers with fractional parts.

Integer

A data type representing whole numbers.

String

A data type representing a sequence of characters.

Map (Objects)

Map (objects) are used to store collections of data and more complex entities.

Array

Array is used for storing a list of something for ordered collection.

Since Dragon is dynamic programming language, which means that explicitly declare the types is not necessary.

```
u = ["text",12] // Array
v = {"key1":1,"key2":2} // Map
w = true // boolean
x = 10 // integer
y = 1.61803 // float
z = "abcd" // string
```

String

A string in Dragon must be surrounded by quotes.

```
str = "Hello"; // double quotes
str2 = 'Single quotes are ok too'; // single quote
```

In dragon, there are 2 types of quotes.

Double quotes: "Hello".

Single quotes: 'Hello'.

Double and single quotes are “simple” quotes. There’s practically no difference between them in dragon programming language.

Two strings can be joined using the + operator:

```
name = "A " + "string"
showln name // A string
```

You can use the + operator to interpolate variables:

```
name = 'Festus'
showln "My name is " + name // My name is Festus
```

Number

The number type represents both integer and floating point numbers.

```
n = 123; // integer
n = 12.345; // floating point number
```

Note: Number type (integer or floating point number) should not be surrounded with quotes (double or single) else the value will be treated automatically as a string.

There are many operations for numbers, e.g. multiplication *, division /, addition +, subtraction -, and so on. (We’ll cover more of this in Operators)

Boolean (logical type)

The boolean type has only two values: true and false.

This type is commonly used to store yes/no values: true means “yes, correct”, and false means “no, incorrect”.

For instance:

```
nameFieldChecked = true; // yes, name field is checked
ageFieldChecked = false; // no, age field is not check
```

The variable with value true will return 1 which means true, while the variable with value false will return 0 which means false.

Boolean values can come as a result of comparisons

```
isGreater = 4 > 3 // if 4 is greater than 3
showln isGreater // true
```

This will return 1 which means it is true. 0 is false while 1 is true. since 4 is greater than 3, the expression will return 1 which is true.

You can use the Boolean logical type also to check for a letter characters or strings.

For instance:

```
name = "dog" > "hi"
show name // false, 0
```

Boolean can be used with conditional statement if and else. If true will return (true) otherwise false

```
if (true) {
  showln "true";
}
else {
  showln "false"
} // will return true
```

if condition is false, will return false.

```
if (false) {
  showln "true";
}
else {
  showln "false"
}
```

Interpreting other types as Booleans

Here are the rules for determine the "truth" of any value not already of the Boolean type –

- ★ If the value is a number, it is false if exactly equal to zero and true otherwise.
- ★ If the value is a string, it is false if the string is empty (has zero characters) or is the string "0", and is true otherwise.
- ★ If the value is an array, it is false if it contains no other values, and it is true otherwise. For an object, containing a value means having a member variable that has been assigned a value.
- ★ Valid resources are true (although some functions that return resources when they are successful will return FALSE when unsuccessful).
- ★ Don't use double as Booleans

Map (Objects)

In Dragon, almost "everything" is a map (object).

Dragon is designed on a simple object-based paradigm. A map (object) is a collection of properties, and a property is an association between a name (or key) and a value

Dagon Primitive Values:

Primitives are values, they have no properties and methods.

Primitive Data Types is data that has a primitive value.

Dragon defines 3 types of primitive data types:

1. String
2. Number
3. Boolean (true or false)

Everything else is an object.

The fundamental difference between **primitives** and **non-primitives** is that **primitives** are immutable (they are hardcoded and therefore cannot be changed). and **non-primitives** are mutable.

Objects are used to store keyed collections of various data and more complex entities.

An object can be created with figure brackets `{...}` with an optional list of properties. A property is a “key: value” pair, where key is a string (also called a “property name”), and value can be anything.



We can imagine an object as a cabinet with signed files. Every piece of data is stored in its file by the key.

It's easy to find a file by its name or add/remove a file.

An empty object (“empty cabinet”) can be created using this syntax:

```
user = {} // "object literal" syntax
```

Usually, the figure brackets `{...}` are used. That declaration is called an object literal.

Literals and properties

We can immediately put some properties into `{...}` as “key: value” pairs:

```
user = {      // an object
  "name": "Festus", // by key "name" store value "Festus"
  "age": 17    // by key "age" store value 17
};
```

Spaces and line breaks are not important. An object definition can span multiple lines

The values are written as **name : value** pairs (name and value separated by a colon). A Dragon map (object) is a collection of named values.

Map (Object) Properties

The named values, in Dragon objects, are called properties.

Property	Value
name	Festus
age	17

A property has a key (also known as “name” or “identifier”) before the colon “:” and a value to the right of it.

In the user object, there are two properties:

The first property has the name "name" and the value "Festus".

The second one has the name "age" and the value 17.

The resulting user object can be imagined as a cabinet with two signed files labeled “name” and “age”.

Property values are accessible using the dot notation

```
user = {      // an object
  "name": "Festus", // by key "name" store value "Festus"
  "age": 17  // by key "age" store value 17
};
showln user.name // Festus
showln user.age  // 17
```


Array

An array is a collection of elements.

Arrays in Dragon are not a type on their own.

Arrays are objects.

We can initialize an empty array in this way, using this square bracket []

Declaration

```
arr = []; // we can store elements inside []
```

We can supply initial elements in the brackets:

```
fruits = ["Orange", "Apple", "Guava"];
```

An array can store elements of any type.

For instance:

```
// mix of values
arr = ['Apple', { name: 'John' }, true, function() { console.log('hello'); }];

// get the object at index 1 and then show its name
console.log(arr[1].name); // John

// get the function at index 3 and run it
arr[3](); // hello
```

Array elements are numbered, starting with zero [0].

Flowchart

```
arr = ["a", "b", "c", "d"]
showln arr // output: a, b, c, d
```

We can get an element by its number in square brackets:

```
fruits = ["Orange", "Apple", "Guava"];  
showln fruits[0] // Orange  
showln fruits[1] // Apple  
showln fruits[2] // Guava
```

We can replace an element:

```
fruits[2] = 'Mango'; // now ["Apple", "Orange", "Mango"]
```

Since Arrays are numbered starting from 0 zero. We can then get the length of an Array using the `length()` method.

`length()` - returns length of string, array/map size or number of function arguments

The `length` is an inbuilt function that works with "std" modules in dragon. Modules has been explained in the modules section of this book. `length()` function which is in "std" modules. Go through the Modules section to understand this fully.

To use the `length()` function. Do this first: select "std"

For Example:

```
select "std"  
arr = [2, 4, 8, 6, 9, 0]; // arr  
showln length(arr) // return 6, length of array
```

Note that semi-colon is not necessary in Dragon.

```
select "std"  
s = "hello"  
showln length(s) //return 5, length of string
```

Nested Arrays

In Dragon: Arrays can be nested, meaning that an array can contain another array as an element. Using this characteristic of Dragon arrays, multi-dimensional arrays can be created. The following code creates a two-dimensional array.

```
arr = ["a", "b", [1, 2, 3]] // nested array
showln arr // a, b, 1, 2, 3
```

We can also access the elements of the nested array (inner array) by the numbered element since Arrays are numbered starting from 0

For instance:

```
arr = ["a", "b", [1, 2, 3]] // nested array
showln arr[2] // output: 1, 2, 3
```

You can Join non-nested Arrays together with nested Arrays using the + operator to concatenate the elements

For instance:

```
arr = ["i ", "am ", ["nested array"]]

showln arr[0] + arr[1] + arr[2] // i am nested array
```

More Details:

Let's go into a more detailed description of nested array and how it works in Dragon programming language.

The following syntax defines a two dimensional array named [activities](#)

```
activities = [
    ['Code', 9],
    ['Eat', 1],
    ['Commute', 2],
    ['Play Game', 1],
    ['Sleep', 7]
];
```

In the `activities` array, the first dimension represents the activity and the second one shows the number of hours spent per day for each.

To access an element of the multidimensional array, you first use square brackets to access an element of the outer array that returns an inner array, and then use another square bracket to access the element of the inner array.

The following example returns the second element of the first inner array in the `activities` array above:

```
show\nactivities[0][1] // 9
```

o Operators

An expression is a special kind of statement that evaluates to a value. Every expression is composed of:

Operands: Represents the data.

Operator: Defines how the operands will be processed to produce a value.

Consider the following expression- $2 + 3$. Here in the expression, 2 and 3 are operands and the symbol + (plus) is the operator. Dragon supports the following types of

operators:

- * Arithmetic Operators.
- * Logical Operators.
- * Relational operators
- * Bitwise operators
- * Assignment operators
- * Conditional operators
- * String operators
- * Typeof operators

Arithmetic Operators

Assume the values in variables **a** and **b** are 10 and 5 respectively.

Operators	Function	Example
+	Addition: Returns the sum of the operands	$a + b = 15$
-	Subtraction: Returns the difference of the value	$a - b = 5$
*	Multiplication: Returns the product of the value	$a * b = 50$
/	Division: Performs a division operation Returns the quotient	$a / b = 2$
%	Modulus: Divide and Returns the remainder of the value	$a \% b = 0$

++	Increments the value of the variable by one	a++ = 11
--	Decrements the value of the variable by one	a-- = 9

Increments means to **increase** or **add** while Decrements means to **decrease** or **remove**.

Example: Arithmetic Operator

Addition:

```
a = 10
b = 5
showln a + b // 15
```

Subtraction:

```
a = 10
b = 5
showln a - b // 5
```

Multiplication:

```
a = 10
b = 5
showln a * 5 // 50
```

Division:

```
a = 10
b = 5
showln a / b // 2
```

Modulus:

```
a = 10
b = 5
showln a % b // 0
```

Increments:

```
a = 10
showln a++ // 11
```

Decrements:

```
a = 10
showln a-- // 9
```

Relational Operators

Relational operators test or define the kind of relationship between two entities. Relational operators return a boolean value, i.e. true/false.

Assume the value of **a** is 10 and **b** is 20.

Operators	Description	Examples
>	Greater than	a > b is false
<	Less than	a < b is true
>=	Greater than or Equal to	a >= b is false
<=	Less than or Equal to	a >= b is true
==	Equality	a == b is false
!=	Not Equal	a != b is true

Example: Relational Operators

Greater than

```
a = 10
b = 20
showln a > b // false
```

Less than

```
a = 10
b = 20
showln a < b // true
```

Greater than or Equal to

```
a = 10
b = 20
showln a >= b // false
```

Less than or Equal to

```
a = 10
b = 20
showln a <= b // true
```

Equality

```
a = 10
b = 20
showln a == b // false
```

Not Equal

```
a = 10
b = 20
showln a != b // true
```

Logical Operators

Logical operators are used to combine two or more conditions. Logical operators, too, return a Boolean value. Assume the value of variable **a** is 10 and **b** is 20.

Operators	Description	Examples
&&	And: The operator returns true only if all the expressions specified return true	a > 10 && b > 10 is false
	OR: The operator returns true if at least one of the expression specified is true	a > 10 b > 10 is true
!	NOT: The operator returns the inverse of the expressions result. For E.g.: !(3>2) returns false	!(a > 10) is True

&& (AND operator)

```
a = 10
b = 20
showln a > 10 && b > 10 // false
```

|| (OR operator)

```
a = 10
b = 20
showln a > 10 || b > 10 // true
```



```
!(NOT operator)
```

```
a = 10
```

```
b = 20
```

```
showln !(a > 10) // true
```

The Short-circuit Operators

Short-circuit evaluation is the semantics of some Boolean operators in some programming languages such like Dragon in which the second argument is executed or evaluated only if the first argument does not suffice to determine the value of the expression: when the first argument of the **AND** function evaluates to false , the overall value must be false ; and when the first argument of the **OR** function evaluates to true , the overall value must be true .

The **&&** and **||** operators are used to combine expressions. The **&&** operator returns true only when both the conditions return true.

Let us consider an expression:

```
a = 10
result = a < 10 && a > 5
showln result // false
```

This happens because the **&&** operator returns first falsy expression.

In the above example, $a < 10$ and $a > 5$ are two expressions combined by an **&&** operator. Here, the first expression returns false. However, the **&&** operator requires both the expressions to return true. So, the operator skips the second expression.

Therefore, Due to this behavior of the **&&** and **||** operator, they are called as short-circuit operators.

Bitwise Operators

Bitwise operators are used for manipulating a data at the bit level, also called as bit level programming. Bit-level programming mainly consists of 0 and 1. They are used in numerical computations to make the calculation process faster.

Dragon supports the following bitwise operators. The following table lists all the Dragon's bitwise operators. Assume variable $a = 3$ and variable $b = 2$, then:

Operators	Description	Usage/Symbol	Result
Bitwise AND	Returns a one in each bit position for which the corresponding bits of both operands are ones	$a \& b$	2

Bitwise OR	Returns a one in each bit position for which the corresponding bits of either or both operands are ones	$a \mid b$	3
Bitwise XOR	Returns a one in each bit position for which the corresponding bits of either but not both operands are ones	$a \wedge b$	1
Bitwise COMPLEMENT	reverse the bits of an expression.	\sim	1
Left Shift	Shifts a in binary representation b (< 32) bits to the left, shifting in zeroes from the right	$a \ll b$	12
Right Shift	Shifts a in binary representation b (< 32) bits to the right, discarding bits shifted off	$a \gg b$	0

For instance:

```
a = 3
b = 2
showln a & b // 2
```

```
a = 3
b = 2
showln a ^ b // 1
```

```
a = 3
b = 2
showln a ~ b // 1
```

```
a = 3
b = 2
showln a | b // 3
```

```
a = 3
b = 2
showln a << b // 12
```

```
a = 3
b = 2
showln a >> b // 0
```

Assignment Operators:

Assignment Operators are used to assign a values to a property or variable.

The following table summarizes assignment operator

Sr. No	Operators and Description
1	= (Simple Assignment) Assigns values from the right side operand to the left side operand Example: c = a + b will assign the value of a + b into c
2	+= (Add and Assignment) It adds the right operand to the left operand and assigns the result to the left operand. Example: c += a is equivalent to c = c + a
3	-= (Subtract and Assignment) It subtracts the right operand from the left operand and assigns the result to the left operand. Example: c -= a is equivalent to c = c - a
4	*= (Multiply and Assignment) It multiplies the right operand with the left operand and assigns the result to the left operand. Example: c *= a is equivalent to c = c * a
5	/= (Divide and Assignment) It divides the left operand with the right operand and assigns the result to the left operand.
6	%= (Modules and Assignment) it takes modulus using two operands and assigns the result to the left operand will return the remainder of the value Example: c % a is equivalent to c = c % a

Examples:

* Simple Assignment (=)

```
// =  
one = 1  
two = 2  
one = two  
showln "one = two": +one // 1
```

* Add and Assignment (+=)

```
// +=  
one = 1  
two = 2  
showln one+=two // 3
```

* Subtract and Assignment (-=)

```
// -=  
one = 1  
two = 2  
showln one-=two // 1
```

* Multiply and Assignment (*=)

```
// *=  
num = 2  
num1 = 2  
showln num*=num1 // 4
```

* Divide and Assignment (/=)

```
// /=  
num = 2  
num1 = 2  
showln num/=num // 1
```

* Modulus and Assignment (%=)

```
// /=  
num = 2  
num1 = 2  
showln num/=num // 0
```

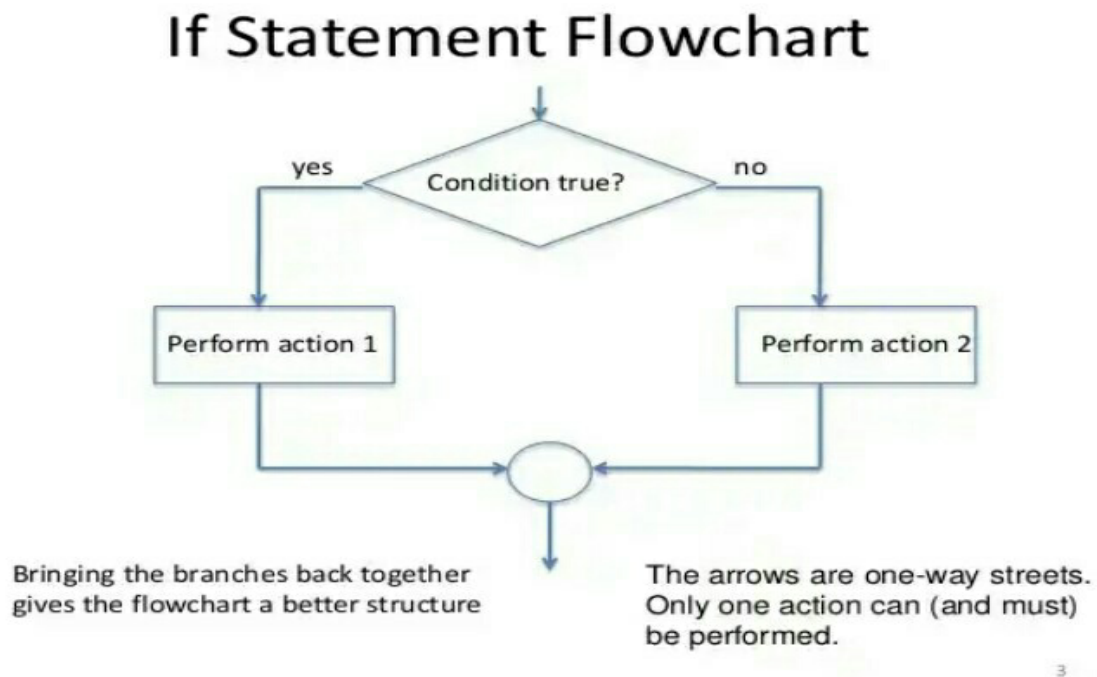
Conditional Operators (if.. Else.. Else if) and ternary operator

Sometimes, we need to perform different actions based on different conditions.

To do that, we can use the if statement and the conditional operator ?, that's also called a "question mark" operator. The question mark operator '?' and colon ':' for if else

The "if" statement

Flowchart:



The if(...) statement evaluates a condition in parentheses and, if the result is true, executes a block of code.

For example:

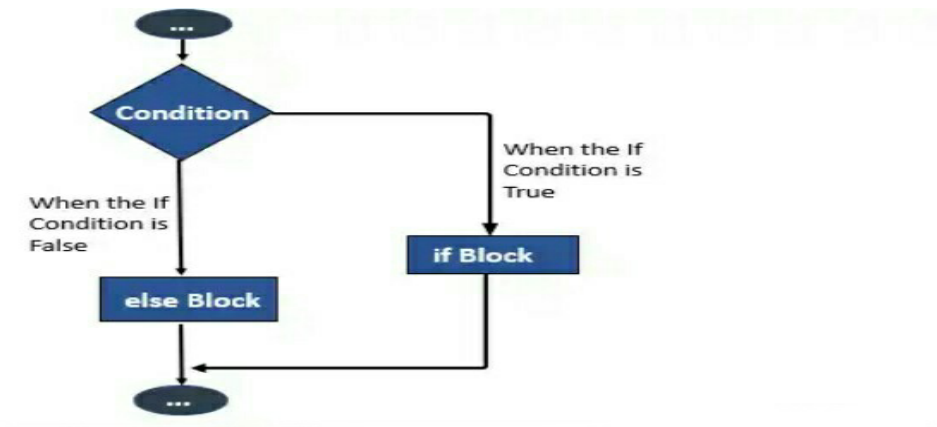
```
age = 15
if ( age == 15 ) {
  showln "true "+age
} // true 15
```

The above code is a simple equality check. We have assigned a value (int 15) to the variable name age and check if the age == 15. Since 15 is assigned to age the code returns true and print true 15.

Though curly brackets isn't necessary in Dragon programming language but we recommend wrapping your codes in curly brackets {...} doing so will improve readability and more cleaner codes.

The 'else' statement

Flowchart:



The if statement may contain an optional “else” block. It executes when the condition is false.

For example:

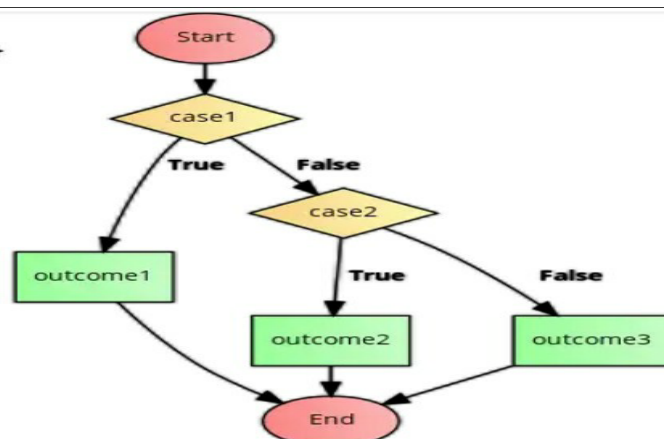
```
version = readln ("When was first version of Dragon released")
    if ( version == 2018 ) {
        showln "true "+version
    } else {
        showln "false, it is not "+version
    }
    // false
```

The above is also a simple equality check that requires an input from the user, if the user enters 2018 will print true otherwise will print false. Therefore, the else statement is used for checking condition in a program and evaluate true if the condition is true otherwise false.

Several conditions: “else if”

Flowchart:

```
Start;
if (case1) {outcome1}
else if (case2) {outcome2}
else {outcome3}
End;
```



Sometimes, we'd like to test several variants of a condition. The else if clause lets us do that.

For example:

```
version = readln("When was first version of Dragon released");

if (version < 2018) {
    showln( 'Too early...' );
} else if (version > 2018) {
    showln( 'Too late' )
} else {
    showln( 'Correct!' )
}
```

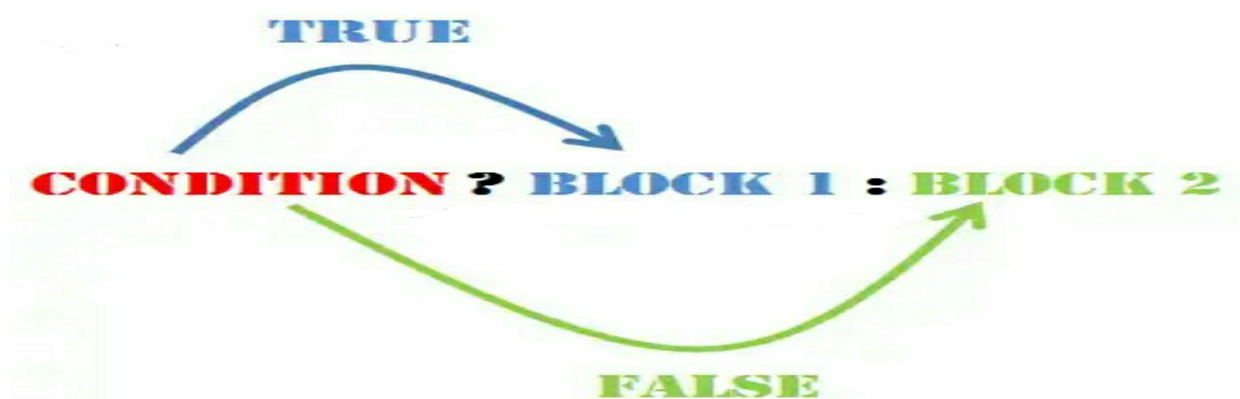
In the code above, Dragon first checks **version < 2018**. If that is falsy, it goes to the next condition **version > 2018**. If that is also falsy, it shows the last condition.

There can be more else if blocks. The final **else** is optional.

Conditional Operator '?'

The **'?'** is known as **Ternary Operator**. The ternary operator is an operator that exists in some programming languages, which takes three operands rather than the typical one or two that most operators use. It provides a way to shorten a simple if else block. ... With this type of comparison, you can shorten the code using the ternary operator.

Flowchart:



It is sometimes called "question mark" operator. The operator is represented by a question mark ?. it's called "ternary", because the operator has three operands.

String Operator

The + operator is used for concatenating strings known string operator, Which means the + operator can be used for joining or putting together two or more string together which is called string operator.

For instance:

```
myName = "I am"+" "+"J ohn Doe"
showln myName // I am J ohn Doe
```

The above code will output "I am J ohn Doe".

Dragon also supports multiplying strings to form a string with a repeating sequence

```
hello = "hello" *3
showln hello // hellohellohello
```

Note that the * operator is not a string operator. Other operators such as / (division), - (subtraction) cannot be used with strings in Dragon.

For instance:

```
hello = "hello" - 3
showln hello // Error

hello = "hello" /3
showln hello // Error
```

The above code will cause the program to return an error message displayed on the screen because it is not possible.

Miscellaneous Operator

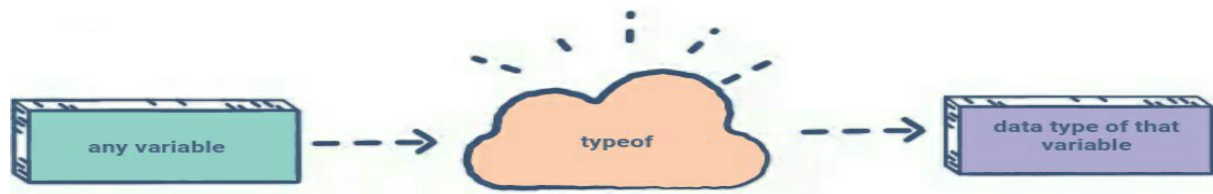
The Miscellaneous operator uses (-) which is known as **negation operator**.

```
x = 4
y = -x
showln "The value of x is " +x // 4
showln "The value of y is " +y // -4
```

The above code, we've assigned 4 to the variable x and assigned -x to the variable y since x = 4, then -x will be -4 which is assigned to y.

Typeof Operator:

The typeof operator is used to get the data type (returns a string) of its operand. The operand can be either a literal or a data structure such as a variable, an array, or an object. The operator returns the data type. The **typeof operator** which is in types module (select "types"). Modules has been treated the section of this book in the **Modules section** (types module) section of this tutorial.



Flowchart:

There are four possible values that typeof returns: map(object), array, number, and string. The following table summarizes possible values returned by the typeof operator.

Type of the Operand	Returns
Number	"number [1]"
String	"string [2]"
Array	"array [3]"
Map (object)	"map [4]"

For instance:

The following operand returns number - 1

```
age = 30
showln typeof(age) // number [1]
```

Same thing applied to strings, map (object) and array.

```
str = "string"
arr = []
obj = {}
showln typeof(str) // string [2]
showln typeof(obj) // array [3]
showln typeof(arr) // object [4]
```

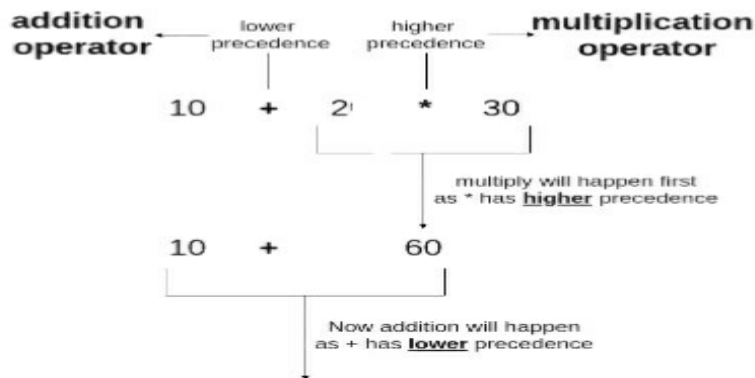
If the **typeof** of the variable is number (will return 1), if string (will return 2), if array (will return 3), if map (object) - will return 4.

Operator Precedence:

Operator precedence determines which operator is performed first in an expression with more than one operators with different precedence.

For example: Solve

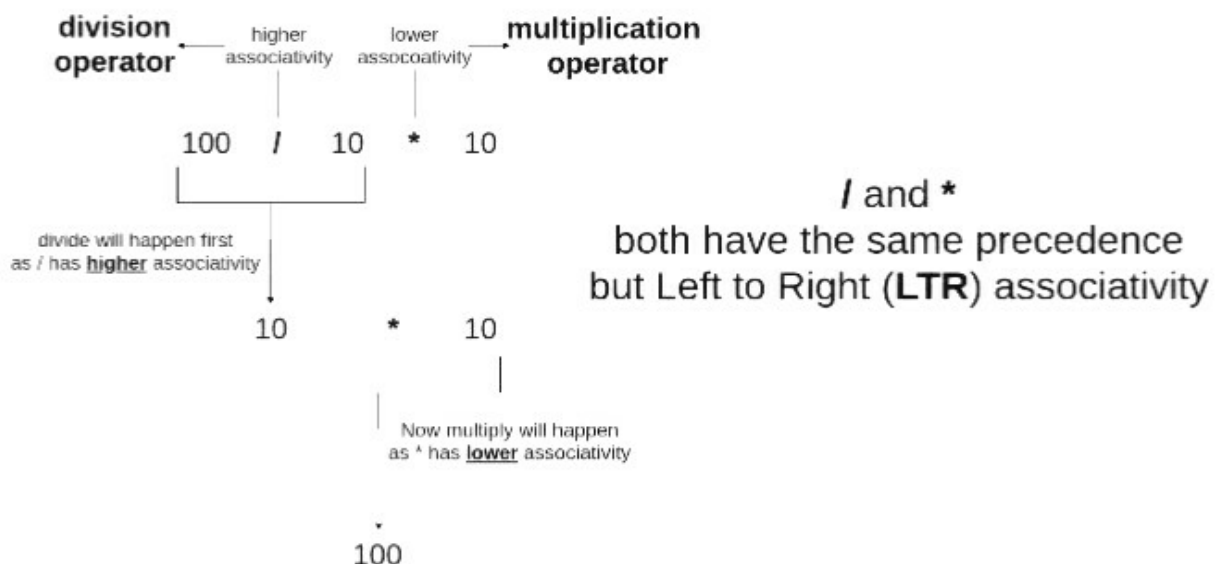
$$10 + 2 * 30$$



$10 + 20 * 30$ is calculated as $10 + (20 * 30)$
and not as $(10 + 20) * 30$

Operators Associativity is used when two operators of same precedence appear in an expression. Associativity can be either Left to Right or Right to Left.

For example: $*$ and $/$ have same precedence and their associativity is Left to Right, so the expression " $100 / 10 * 10$ " is treated as " $(100 / 10) * 10$ ".

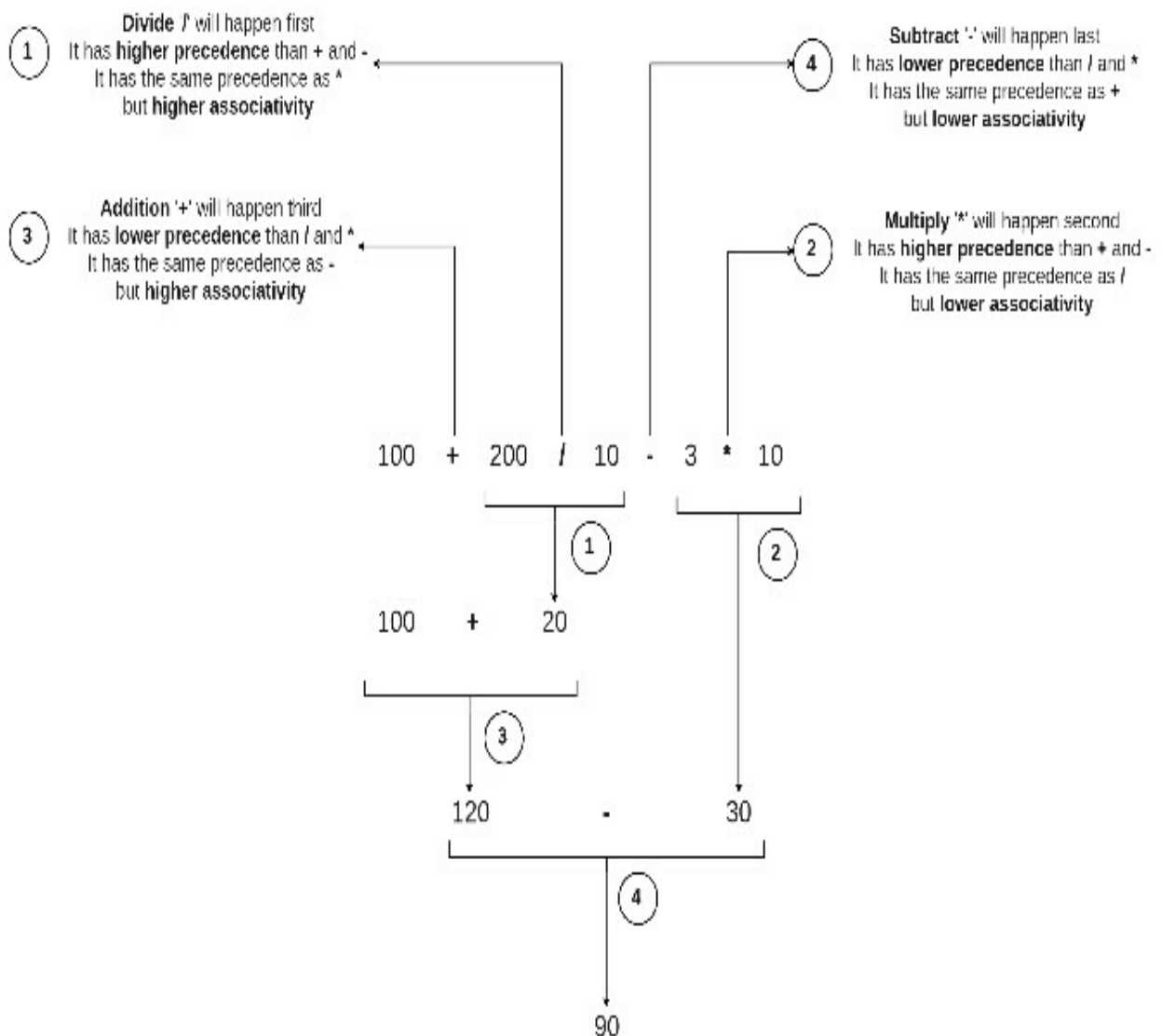


Division will happen first as $/$ has the higher Precedence, Then multiplication will happen next has a lower Associativity.

Operators Precedence and Associativity are two characteristics of operators that determine the evaluation order of sub-expressions in absence of brackets

For instance:

$$100 + 200 / 10 - 3 * 10$$



Associativity is only used when there are two or more operators of same precedence.

Precedence for Arithmetic, Relational and Logical Operators.

This table summarizes the Precedence for Arithmetic, Relational and Logical Operators.

Precedence	Operation	Associativity
1	()	innermost
2	++ -- + - ! (type)	Right to Left
3	* / %	Left to Right
4	+ -	Left to Right
5	< <= > >=	Left to Right
6	== !=	Left to Right
7	&&	Left to Right
8		Left to Right
9	= += -= *= /= %=	Right to Left

It is good to know precedence and associativity rules, but the best thing is to use brackets, especially for less commonly used operators (operators other than +, -, *, .. etc). Brackets increase the readability of the code as the reader doesn't have to see the table to find out the order.

Control Structures:

Control structures are specific commands which are used to control the logical flow of the program during its execution. These commands control which part of the program the computer will execute next.

There are two main types of control structures:

- branching control structures
- loop control structures

Branching control structures:

With branching control structures code contained inside the control structure is either executed or not, based on the calculation of a condition.

The branching control structures include:

- if-else-else-if structures

If-else-else-if structures

Syntax:

```
if(Expression)
{
    Block of statements
}
else if(Expression)
{
    Block of statements
}
else {
    Block of statements
}
```

Loop control structures:

While code in a branch control structure can be left out of execution during runtime if the condition is false, code in a loop control structure is usually executed one or several times over.

For Loop:

The for-loop is a control flow statement for specifying iteration, which allows code to be executed repeatedly or several times. It's the **repetition** of a code.

Syntax:

```
for initializing, condition, increment {  
    // body  
}
```

or:

```
for (initializing, condition, increment) {  
    // body  
}
```

For instance:

```
for i = 0, i < 6, i++ {  
    showln i  
}
```

Iteration is a process wherein a set of instructions or structures are repeated in a sequence a specified number of times or until a condition is met. When the first set of instructions is executed again, it is called an iteration.

While Loop:

The while loop will repeat a set of instructions based on a condition. As far as the loop returns a boolean value of True.

Syntax:

```
while condition {  
    // body  
}
```

How while loop works?

- The while loop evaluates the test expression inside the parenthesis ().
- If the test expression is true, statements inside the body of while loop are executed. Then, the test expression is evaluated again.
- The process goes on until the test expression is evaluated to false.
- If the test expression is false, the loop terminates (ends).

For instance:

```
i = 0
while ( i < 5) {
    showln i++
}
```

Do while Loop:

The do while loop is similar to while loop with one exception that it executes the statements inside the body of do-while before checking the condition. On the other hand in the while loop, first the condition is checked and then the statements in while loop are executed. So you can say that if a condition is false at the first place then the do while would run once, however the while loop would not run at all.

Syntax:

```
do {
    // statements inside the body of the loop
}
while (condition);
```

How do...while loop works?

- The body of do...while loop is executed once. Only then, the condition is evaluated.
- If the condition is true, the body of the loop is executed again and the condition is evaluated.
- This process goes on until the condition becomes false.
- If the condition is false, the loop ends.

For instance:

```
i = 0
do {
    show i++
} while i < 5
```

ForEach Loop:

Foreach loop (or for each loop) is a control flow statement for traversing items in a collection . Foreach is usually used in place of a standard for loop statement . Unlike other for loop constructs, however, foreach loops usually maintain no explicit counter: they essentially say "do this to everything in this set", rather than "do this x times". This avoids potential off-by-one errors and makes code simpler to read.

Syntax:

```
for value : array {
    // body
}

for key, value : map {
    // body
}

for (value : array) {
    // body
}

for (key, value : map) {
    // body
}
```

For instance:

```
arr = [1, 2, 3, 4]
for v : arr {
    showln v
}

map = {"key1": 1, "key2": 2}
for key, value : map
showln key + " = " value
}
```

Break statement:

The break statement is a loop control statement which is used to terminate the loop. As soon as the break statement is encountered from within a loop, the loop iterations stops there and control returns from the loop immediately to the first statement after the loop.

For instance:

```
for ( i = 1, i <= 10, i++ ) {  
    if ( i == 5 ) {  
        //breaking the loop  
        break  
    }  
    showln i  
} // output: 1234
```

Continue Statement:

The continue statement is used inside loops. When a continue statement is encountered inside a loop, control jumps to the beginning of the loop for next iteration, skipping the execution of statements inside the body of loop for the current iteration. It can be used with for loop and while loop.

For instance:

```
//for loop  
for ( i = 1, i <= 10, i++ ){  
    if ( i == 5 ) {  
        //using continue statement  
        continue  
        //it will skip the rest statement  
    }  
    showln i  
} // output: from 1 to 10
```

Function

In programming, a named section of a program that performs a specific task. In this sense, a function is a type of procedure or routine that perform a specific task. As the name suggests, it performs a 'function'. It can be either a user defined function or a library function. It is usually used by the other parts of the program to accomplish the overall goal of the program.

To define a Function, Dragon uses the `func` keyword which stands for function. Follow by the function name

Syntax:

```
func <function name> {  
    // block of statement  
}  
// call function
```

For instance:

We will create a function named `hello()`, then Our new function can be called by its name

```
func hello() {  
    showln "Hello from function"  
}  
// call function hello()  
hello() //output: Hello from function
```

Calling functions:

We can call the function before the function definition.

Example:

```
// call function before function definition  
hello()  
func hello() {  
    showln "Hello from function"  
}
```

Declare parameters

To declare the function parameters, write them in parentheses.

Example:

```
func sum(x, y) // parameters x, y  
{  
    showln x + y  
}
```

Send Parameters

To send parameters to function, type the parameters inside () after the function name

Syntax:

```
name(parameters)
```

For instance:

```
func sum(x, y) {  
    showln x + y  
}  
// send parameter into function x is 5, y is 10  
sum(5, 10) // output: 15
```

or

```
// call function before function definition and send parameters  
sum(5, 10)  
func sum(x, y) {  
    showln x + y  
} // output: 15
```

Local variables

Variables declared inside of any function are called local variables. A variable declared inside a function is only visible inside that function.

For instance:

```
func hello() {  
    text = "Hello, I am Dragon"  
    showln text  
}  
// call function  
hello() // Hello, I am Dragon  
showln text // Error, The variable is local to the function
```

Global variables

Variables declared outside of any function are called global. Global variables are visible from any function (unless shadowed by locals).

For instance:

```
name = "Festus"
func hello() {
  text = "Welcome " + name
  showln text
}
hello() // Welcome Festus
```

It's a good practice to minimize the use of global variables. Modern code has few or no globals. Most variables reside in their functions. Sometimes though, they can be useful to store project-level data.

Returning a value

A function can return a value back into the calling code as the result. The simplest example would be a function that sums two values:

For instance:

```
func sum(a, b) {
  return a + b;
}
result = sum(1, 2);
showln result // 3
```

The directive `return` can be in any place of the function. When the execution reaches it, the function stops, and the value is returned to the calling code (assigned to `result` above).

Nested functions

You can define function in other function.

For instance:

```
func fibonacci(count) {
  func fib(n) {
    if n < 2 return n
    return fib(n-2) + fib(n-1)
  }
  return fib(count)
}
show fibonacci(10) // 55
```

Naming Functions:

- Functions are actions. So their name is usually a verb. It should be brief, as accurate as possible and describe what the function does, so that someone reading the code gets an indication of what the function does.
- A function should do exactly what is suggested by its name, no more.
- Functions should be short and do exactly one thing. If that thing is big, maybe it's worth it to split the function into a few smaller functions. Sometimes following this rule may not be that easy, but it's definitely a good thing.

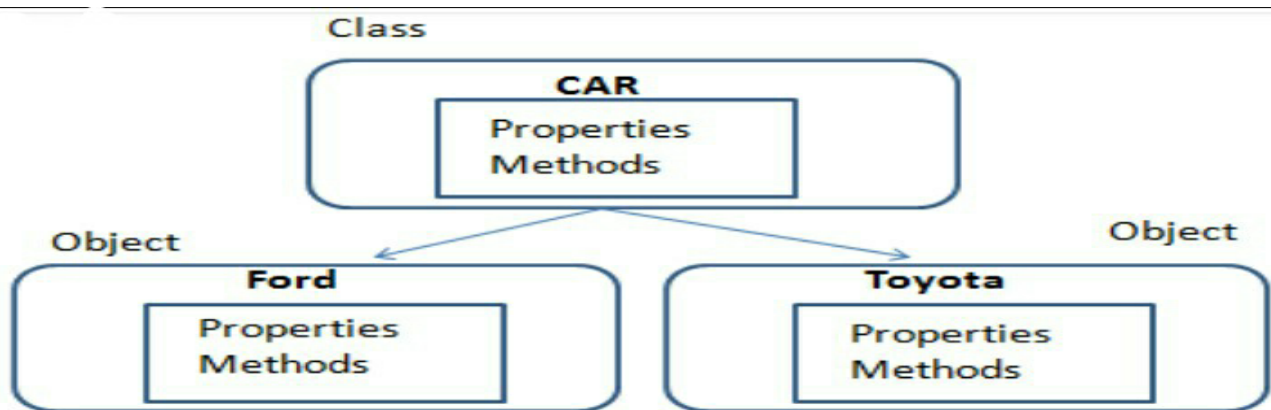
A separate function is not only easier to test and debug – its very existence is a great comment!

Classes

In object-oriented programming , a class is an extensible program-code-template for creating objects , providing initial values for state (member variables) and implementations of behavior (member functions or methods). In many languages, the class name is used as the name for the class (the template itself), the name for the default constructor of the class (a subroutine that creates objects), and as the type of objects generated by instantiating the class; these distinct concepts are easily conflated. When an object is created by a constructor of the class, the resulting object is called an instance of the class, and the member variables specific to the object are called instance variables, to contrast with the class variables shared across the class.

Below is a flowchart of how class looks like for a better and clearer understanding.

Flowchart



For example , a class could be a car, which could have a color field, four tire fields, and a drive method. Another related class could be a truck, which would have similar fields, but not be exactly the same as a car. Both a car and a truck could be a kind of a third class which could be called a vehicle class. In this way, the programmer could create the parts of the program that are the same for both the car and the truck in programming the vehicle class, yet the programmer would be free to program how a car is different from a truck without duplicating all of the programming.

To define classes use the `class` keyword, followed by the name.

For instance:

```
class classname{
    func function(arg1, arg2) {
        show arg1
    }
}

obj = new classname() // class object creation

obj.function(arg1,arg2) // function calling with class
```

Constructor

Constructor name should be same as class name and will be created with func keyword:

```
class cons{  
  func cons(val){  
    showln val  
  }  
}  
  
obj = new cons(val) // class object creation with constructor arguments
```

PATTERN MATCHING

The match operator allows to match values by pattern.

```
x = 2  
show match x {  
  case 1: "One"  
  case 2: "Two"  
  case "str": "String"  
  case _: "Unknown"  
}  
  
x = "str"  
match x {  
  case "": {  
    showln "Empty string"  
  }  
  case "str": {  
    showln "String!"  
  }  
}
```

In this case value and type are checking. If none of case branches doesn't match, the body of case _ branch will executes.

In addition to the constant values, you can set variable name to case.

For instance:

```
func test(x) = match x {  
  case a: "case a: " + a  
  case b: "case b: " + b  
  case c: "case c: " + c  
}  
a = 10  
b = 20  
showln test(15) // case c: 15  
showln test(20) // case b: 20  
showln test("test") // case c: test
```

In this case is two scenarios:

Variable is already defined. Matching to its value.

Variable is not defined. Assign matching value to it and executes body of the case branch.

In the example above, the interpreter sees the first two branches as:

case 10:

case 20:

For the last branch c variable is not defined, so assign c = x and execute body of the case c branch.

MODULES

DEFINING MODULES IN YOUR PROGRAM

Before we deeply going into how to use modules in your program in dragon programming Language, I will like to quickly take you through a glance on what modules really means, and it's important in a program especially in dragon programming language. So let's get started

What is Module:

Modules are used for dividing up the program into smaller, more easily understood, reusable parts. Hence, Dragon can be called a Modular programming language. Modular Programming allows development to be divided by splitting down a program into smaller programs in order to execute a variety of tasks. This enables developers to work simultaneously and minimizes the time taken for development.

Importants of Modules in a program:

1. Reusable Code

The modular codes provide programmers with easy code reusability. In case certain tasks are classified into specific functions or classes, the programmers can reuse that specific code whenever they need to perform that task again.

2. Efficient Program Development

Programs can be developed more quickly with the modular approach since small subprograms are easier to understand, design, and test than large programs.

3. Easier to Debug

However, in case the program is created with modularity in mind, each task has its own individual section of code. As a result, if a problem arises in a particular function, the programmers realize where to look and how to manage a smaller portion of code.

4. Quality

Modularization in programming also improves the quality of a specific piece of code. When the overall program is broken down into smaller parts

To sum it up, Modules in a program allows for simplicity, flexibility, easy debugging, and it pose easy to understand codes. This way making the program more simple to understand.

With this, you should understand Modules, and It's importance in a program. In this section of this tutorial, We will explain the built-in modules in Dragon programming language, and how to use them in a program efficiently. And also including programs. Let get started.....

In Dragon, Module is a library that is used as per the requirement. It is a bunch of predefined functions. Those are required to use the functionality of Dragon. Every Module contains different functions. Some Module names in dragon are std, math, types http, etc
For using the modules in program the `select` keyword is used along with the module name in double quotes.

MODULE 1: THE STD MODULE

The std module contains common functions. The std word stands for “standard”. Which ALSO means the collection of standard functions. The basic requirement of the program needs these functions.

Constants

ARGS : string = command-line arguments

Example:-

```
select "std"
showln ARGS[0] // This will
print the first command line
argument value.
```

INCLUDING A PROGRAM

A program in Dragon also can be used as an user defined module. The `include` keyword is used along with the program name in double quotes for including the program data.

Example:-

```
include "program2.dgn" // include a another program into a program

addition(20,30) // a user defined function inside program2.dgn file
```

Functions:-

1) arrayCombine(keys, values) - combining arrays into one.

It is used for combining two arrays into one. It creates a map of the values of both arrays. Two parameters are required in this function. In which the first one is key and second values, both will be arrays.

Syntax:-

arrayCombine(key,value)

Example:-

```
select "std"
a = ["val1", "val2"]
b = [20, 30]
c = arrayCombine(a,b) // This will create a map
showln c // output: {val1=20, val2=30}
```

2) arrayKeyExists(key, map) - checks existing key in map. returns 1 if exists exist, 0 if not exist.

The arrayKeyExists is used for finding a specific key in the map. It will give output in the form of 1 or 0. Which means exists or not exists. There are two parameters required in this function. The first one is key and the second is a map.

Syntax:-

arrayKeyExists(key, map)

Example:-

```
select "std"
ab = {"val1":10, "val2":20}
showln arrayKeyExists("val2", ab) //return 1, val2 exist in map ab
showln arrayKeyExists("val3", ab) //return 0, val3 does not exist in map ab
```

3) arrayKeys(map) - returns array of map keys

The arrayKeys method is used to make an array of keys in the map. There is one parameter requires in this function. Which will be a map.

Syntax:-

arrayKeys(map)

Example:-

```
select "std"
ar = {"a":3,"b":4, "c":8} //map

showln arrayKeys(ar) // map given as parameter

//output:- [a, b, c]
```

4) arrayValues(map) - to return the value in the map

This method makes an array of values in the map. While the arrayKeys was only for Keys, while arrayValues is to return the values in the map. One parameter is required in the method. That is a map.

Syntax-

arrayValues(map)

Example:-

```
select "std"
ar = {"a":3, "b":4, "c":8} //map

showln arrayValues(ar) // map given as parameter

//output:- [3, 4, 8]
```

5) charAt(input, index) - returns char code in position index of string input

It returns the character of the specific location of the given input. There are two parameters requires in this method. The first one is input and the second is the index number.

Syntax-

charAt(input, index)

Example:-

```
select "std"
text = "Dragon"

showln charAt(text, 3) //return 103, Character code of "g"
```

6) echo(arg...) - prints values to console, separate them by space and puts newline at the end.
Takes variable number of arguments

The echo() function is different from show and showln

Syntax-

echo (arg)

Example:-

```
select "std"
echo (1, "abc") // prints "1 abc" to console
echo (1, 2, 3, 4, 5, "a", "b") // prints "1 2 3 4 5 a b"
```

7) indexOf(input, what, index = 0) - finds first occurrence of what in string input, starting at position index

It finds the first occurrence of character or word from the starting index of input (index starts from 0 and so on). It requires three parameters. Those are input, what and index.

Minimum 2 parameters require. array and delimiter. Maximum 3 parameters accepted. with prefix and suffix. Third one is starting position of search

Syntax-

indexOf(input, what, index = 0)

Example:-

```
select "std"
s = "HELLO"
indexOf(s,"L") //return 2, the position of first "L"
indexOf(s,"L",3) //return 3, the position of "L" from search postion 3
```

Note: Index starts counts from 0 not 1. so **H** is 0, **E** is 1, **L** (first L) is 2, **L** (second L) is 3, **O** is 4.

This way:

H - 0 (not 1)

E - 1 (not 2)

L - 2 (not 3)

L - 3 (not 4)

O - 4 (not 5)

8) `join(array, delimiter = "", prefix = "", suffix = "")` - join array to string with delimiter, prefix and suffix

This function joins the array to string with delimiter, suffix and prefix. The suffix and prefix are not necessary parameters.

Syntax-

`join(array, delimiter, prefix, suffix)`

Example:-

```
select "std"
text = ["h","e","l","l","o"]
showln join(text,"") //return hello
showln join(s,"","abc","xyz") //return abchelloxyz
```

9) `lastIndexOf(input, what, index = 0)` - finds last occurrence of what in string input, starting at position index

Minimum 2 parameters required input and character. Maximum 3 parameters accepted. third one is back position of search.

Syntax-

`lastIndexOf(input, what, index = 0)`

Example:-

```
select "std"
s = "hello"
showln lastIndexOf(s,"l") //return 3, the position of last "l"
showln lastIndexOf(s,"l",2) //return 2, the position of "l" from back search
                        position 2
```

`indexOf()` and `lastIndexOf()` performs almost the same operation the only different is that the `indexOf()` will search for the index from the starting position from left to right, while `lastIndexOf()` will search for the index from the back search position starting from right to left.

10) `length(x)` - returns length of string, array/ map size or number of function arguments

The `length()` function counts the number of items in an object. The objects can be a string, integer, array or map.

Syntax-

`length(x)`

Example:-

```
select "std"
text = "hello"
showln length(text) //return 5, length of string
```

```
select "std"
arr = ["a","b"]
showln length(arr) //return 2, size of array
```

11) `newarray(size...)` - creates array with size. `newarray(x)` - creates 1D array, `newarray(x,y)` - creates 2D array

Apart from creating and initializing array using the square bracket `[]` dragon provides a built-in function to create array with size.

Syntax:-

`newarray(size...)`

`newarray(x)` - 1 Dimensional array

`newarray(x, y)` - 2 Dimensional array

Example:-

```
select "std"
a = newarray(4) // [0, 0, 0, 0]
b = newarray(2, 3) // [[0, 0, 0], [0, 0, 0]]
```

12) `parseInt(str, radix)` - parses string into integer in the radix

The `parseInt()` function parses a string argument and returns an integer of the specified radix (the base in mathematical numeral system). Minimum one parameters require. string, Maximum two parameters accepted. second is radix.

Syntax:-

`parseInt(str, radix)`

Example:-

```
select "std"
showln parseInt("10") //return 10, As Integer
showln parseInt("101",2) //return 5, As Integer
```

13) `parseLong(str, radix)` - parses string into long in the radix

Parses the specified string as a signed decimal long value.

Syntax:-

`parseLong(str, radix)`

Example:-

```
select "std"
showln parseLong("10000") // return 10000, As Long Integer
showln parseLong("10101",2) // return 21, As Long Integer
```

14) `rand(from = 0, to = ..)` - returns pseudo-random number.

`rand()` function is used in dragon to generate random numbers. If we generate a sequence of random number with `rand()` function, it will create the same sequence again and again every time program runs.

Syntax:-

`rand()` - returns float number from 0 to 1

`rand(max)` - returns random number from 0 to max

`rand(from, to)` - return random number from from to to

Tip:

The `rand()` function generates a random integer.

- If you want a random integer between 10 and 100 (inclusive), use `rand (10,100)`.

Example:-

```
select "std"
showln rand(100) // 11

//From 10 to 100
select "std"
showln rand(10, 100) // 14
```

15) range(from = 0, to, step = 1) - creates lazy array by number range

The range() function is used to generate a sequence of numbers over time. At its simplest, it accepts an integer and returns a range object (a iterable object) . It is used when a user needs to perform an action for a specific number of times.

range() only works with the integers. All arguments must be integers. You can not use float number or any other type. Aegument can also contain negative values.

Syntax-

range(from = 0, to, step = 1) - creates lazy array by number range.

range(to) - creates range from 0 to to (exclusive) with step 1

range(from, to) - creates range from from to to (exclusive) with step 1

range(from, to, step) - creates range from from to to (exclusive) with step step

Example

```
select "std"
showln range(10) // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
//or
showln range(0, 10)
```

```
//negative values
r = range(-5, 0) // [-5, -4, -3, -2, -1]
show r[0] // -5
show r[2] // -3
// iterable
for x : range(20, 9, -5) {
  showln x
} // 20 15 10
```

16) readln(x) - reads a line from console

The readln() function reads a line from console. It is used for getting user input value. J ust like prompt() in J avaScript, The readln() function gets user input value and can be displayed using show or showln.

Syntax-

readln(x)

Example:-

```

        select "std"
a = readln() //program ask for user input

        showln a //prints the value of a

```

17) replace(str, target, replacement) - replaces all occurrences of string target with string replacement

This function replaces all occurrences of string target with string replacement.

Syntax:-

replace(str, target, replacement)

Example:-

```

        select "std"
        s = "hello"
        showln replace(s,"l","p")
/* return heppo, replace all occurrences of l replaced with p */

```

18) replaceAll(str, regex, replacement) - replaces all occurrences of regular expression regex with string replacement

This function, replace all occurrence of regular expression with string replacement. replaceAll() function in dragon is similiar with str_replace() function in PHP.

Syntax:-

replaceAll(str, regex, replacement)

Explanation: the **str** stands for the string (the object declaration), **regex** stands for value to replace, and **replacement** stands for the value replaced with **regex**.

Example:-

```

        select "std"
        s = "hello"
        showln replaceAll(s,"hello","welcome")
/* return welcome, replace all occurrences of hello replaced with welcome */

```

19) `replaceFirst(str, regex, replacement)` - replaces first occurrence of regular expression `regex` with string `replacement`.

This function replaces first occurrence of regular expression `regex` with string `replacement`.

The `replaceFirst()` function is different from `replace()` and `replaceAll()`. What distinguishes `replaceFirst()` from the 2 functions is that, `replaceFirst()` will only replace the first occurrence of a character. Assuming a word like "HELLO", it has two **L** characters, instead of replacing the both **LL** with another character, the `replaceFirst()` function will only replace the first occurrence of **L** character and not the two. Mean While `replace()` function will replace both **L** characters in the word.

Syntax-

`replaceFirst(str, regex, replacement)`

Example:-

```
select "std"
s = "hello"
replaceFirst(s,"l","p")
/* return heplo, replace first occurrence of l
replaced with p */
```

20) `sleep(time)` - causes current thread to sleep for time milliseconds

The `sleep()` function shall cause the calling thread to be suspended from execution until either the number of realtime seconds specified by the argument `seconds` has elapsed or a signal is delivered to the calling thread and its action is to invoke a signal-catching function or to terminate the process. The argument may be a floating point number to indicate a more precise sleep time.

Syntax-

`sleep(time)`

Example:-

```
select "std"
sleep(5000) //wait for 5000 milliseconds
showln "Hello, World"
/* print Hello, World
after waiting for 5000 milliseconds */
```

21) `sort(array, comparator = ..)` - sorts array by natural order or by comparator function

The `sort()` method sorts the elements of an array in place and returns the sorted array. The default sort order is ascending, built upon converting the elements into strings, then comparing their sequences of UTF-16 code units values.

Syntax-

`sort(array)`

Example:-

Alphabetical Sort:

You can sort alphabetical characters.

```
select "std"
a = ["u", "o", "i", "e", "a"]
showln sort(a) // [a, e, i, o, u]
```

Numeric Sort:

You can sort numerical characters likewise.

```
select "std"
a = [1,6,3,8]
sort(a) // return [1,3,8,6]
```

22) `split(str, regex, limit = 0)` - splits string `str` with regular expression `regex` into array. `limit` parameter affects the length of resulting array

The `split()` method is used to split a string into an array of substrings, and returns the new array.

Syntax-

`split(str, regex, limit)`

Example:-

```
select "std"
a = "how are you"
showln split(a, " ") //output: [how are you]
```

Or

```
select "std"
showln split("a5b5c5d5e", "5") // ["a", "b", "c", "d", "e"]
showln split("a5b5c5d5e", "5", 3) // ["a", "b", "c5d5e"]
```

23) sprintf(format, args...) - formats string by arguments

The sprintf() function writes a formatted string to a variable.

Syntax-

sprintf(format, args...)

Example:-

```
select "std"
showln sprintf("Value of %s is %d","a",20)//return "Value of a is 20"
```

24) substring(str, startIndex, endIndex = ..) - returns string from startIndex to endIndex or to end of string if endIndex is not set

The substring() method extracts the characters from a string, between two specified indices, and returns the new sub string.

This method extracts the characters in a string between "start" and "end", not including "end" itself.

Syntax

substring(str, startindex, end index)

Example:-

```
select "std"
showln substring("abcde", 1) // bcde
showln substring("abcde", 2, 4) // cd
```

25) sync(callback) - calls an asynchronous function synchronously

Syntax-

sync(callback)

Example:-

```

select "std"
result = sync(func(ret) {
  http(url, func(t) = ret(t))
})

```

26) thread(func, args...) - creates new thread with parameters if passed

Syntax:-

thread(func, args)

Example:-

```

select "std"
thread(func() {
  show "New Thread"
})

thread(::newthread, 10)
thread("newthread", 20)

func newthread(x) {
  show "New Thread. x = " + x
}

```

27) time() - returns current time in milliseconds from 01.01.1970

time() function is a built-in function in Dragon which returns the current time measured in the number of seconds since the Unix Epoch.

Syntax:-

time()

Example:-

```

select "std"
time() //return 1593268596562

```

28) toChar(code) - converts char code to string

The toChar() function is used to convert a specified value to a Unicode integer.

Syntax-

toChar(code)

Example:-

```
select "std"
showIn toChar(48) // "0"
```

29) toHexString(number) - converts number into hex string

toHexString() function returns a string representation of the integer argument as an unsigned integer in base 16.

Syntax-

toHexString(number)

Example:-

```
select "std"
toHexString(12) //return "c"
```

30) toLowerCase(str) - converts all symbols to lower case

The toLowerCase() method returns the value of the string converted to lower case.

Syntax-

toLowerCase(str)

Example:-

```
select "std"
a = "HELLO"
showIn toLowerCase(a) // return: "hello"
```


31) toUpperCase(str) - converts all symbols to upper case

The toUpperCase() method returns the value of the string converted to uppercase.

Syntax-

toUpperCase(str)

Example:-

```
select "std"
a = "hello"
showln toUpperCase(a) // return: "HELLO"
```

32) trim(str) - removes any leading and trailing whitespaces in string

trim() function removes whitespace and other predefined characters from both sides of a string.

Syntax-

trim(str)

Example:-

```
select "std"
a = " Dragon "
trim(a) //return "Dragon"
```

33) try(unsafeFunction, catchFunction = func(type, message) = -1) - suppress any error in unsafeFunction and returns the result of the catchFunction if any error occurs

Syntax-

try()

Example:-

```
select "std"
try ( func() = "success" ) // success
try ( func() = try + 2 ) // -1
try ( func() = try(), func(type, message) = sprintf("Error handled:\n\ntype: %s\n\nmessage: %s", type,
message))
```

MODULE 2: THE TYPES MODULE

The types module in dragon Contains functions for type checking and conversion. The types module is define by `select "types"`

Constants

OBJECT : number = 0

NUMBER : number = 1

STRING : number = 2

ARRAY : number = 3

MAP : number = 4

FUNCTION : number = 5

Functions:-

1) `byte(value)` - converts value to byte returns the output value.

Syntax-

`byte(value)`

Example:-

```
select "types" //selecting types module
showln byte(10) //output:- 10
```

2) `double(value)` - converts value to double. returns the output value.

Syntax-

`double(value)`

Example:-

```
select "types"
showln double(1912412) //output:- 1912412.0
```

3) float(value) - converts value to float. returns the output value.

Syntax-

float(value)

Example:-

```
select "types"
showln float(10) //output:- 10.0
```

4) int(value) - converts value to int. returns the output value

Syntax-

int(value)

Example:-

```
select "types"
showln int("100") // return 100
```

5) long(value) - converts value to long. return the output value

Syntax-

long(value)

Example:-

```
select "types"
showln long("100000") //return 100000
```

6) number(value) - converts value to number if possible. return the output value.

Syntax-

number (value)

Example:-

```
select "types"
showln number("2.3") //return 2.3
```

7) short(value) - This function convert value to short. return the output value

Syntax-

short(value)

Example:-

```
select "types"
short("102") //return 102
```

8) string(value) - This function converts value to string. return the output value.

Syntax-

string(value)

Example:-

```
select "types"
string(1) //return "1"
```

9) typeof(value) - returns the data type of the value.

Syntax-

typeof(value)

Example:-

```
select "types"
showln typeof(1) // 1 (NUMBER)
showln typeof("text") // 2 (STRING)
showln typeof([]) // 3 (ARRAY)
showln typeof({}) // 4 (MAP)
```

MODULE 3: THE MATH MODULE

The math modules in Dragon contains functions for mathematical operations. These functions are very useful while performing mathematical operations in Dragon programming language. To define the math module, use the select keyword follow by the module name in double quote: `select "math"`

Constants

E : number = 2.718281828459045

PI : number = 3.141592653589793

Functions:-

1) `abs()` - returns the absolute number of the given value.

The `abs()` method returns the absolute value of the given number. If the number is a complex number, `abs()` returns its magnitude.

Syntax-

`abs(value)`

Example:-

```
select "math"
showln abs(1.2900) //return 1.29
```

2) `acos()` - returns the arc cosine value.

The `acos()` function returns the arc cosine of a number. Tip: `acos(- 1)` returns the value of Pi.

Syntax-

`acos(value)`

Example:-

```
select "math"
showln acos(1/2) //return
1.5707963267948966
//non radian value
```

3) atan() - This function is used to find the arc tangent of given value.

atan() takes a single argument as a double and returns the value in radians. The returned value of atan() is of type double

Syntax-

atan(value)

Example:-

```
select "math"
showln atan(1) //return 0.7853981633974483
```

4) asin() - This function is used to find the arcsine of given value.

asin() function returns the arc sine (inverse sine) of a number in radians. The asin() function takes a single argument ($-1 \leq x \leq 1$), and returns the arc sine in radians.

Syntax-

asin(value)

Example:-

```
select "math"
showln asin(2/22) //return 1.5707963267948966
```

5) cbrt() - returns the cube root of given value.

The cbrt() function returns the cube root of a number.

Syntax-

cbrt(value)

Example:-

```
select "math"
showln cbrt(64) //return 4.0
```

6) ceil() - returns the smallest integer that is greater than or equal to the value.

The ceil() function rounds a number UP to the nearest integer, if necessary.

Syntax-

ceil(value)

Example:-

```
select "math"
showln ceil(2.5) //return 3.0
```

7) atan2(y, x) - returns angle whose tangent is the ratio of two numbers

Syntax-

atan2(y, x)

Example:-

```
select "math"
showln atan2(1,2)
//return
0.4636476090008061
```

8) copySign(magnitude, sign)

The copysign() function returns a value with the magnitude of its first argument and the sign of its second argument.

Syntax-

copySign(magnitude, sign)

Example:-

```
select "math"
showln copySign(2, -2)
//return -2.0
```

9) cos(x) - trigonometric cosine

The cos() function computes the cosine of an argument.

Syntax-

cos(x)

Example:-

```
select "math"
showln cos(4) // return: - 0.6536436208636119
```

10) cosh(x) - hyperbolic cosine

cosh() function takes a single argument (angle in radians) and returns the hyperbolic cosine of that angle as type double.

Syntax-

cosh(x)

Example:-

```
select "math"
showln cosh(4) //return: 27.308232836016487
```

11) exp(x) - ex

"exp" stands for exponential. The function computes the exponential (Euler's number) raised to the given value.

Syntax-

exp(x)

Example:-

```
select "math"
showln exp(2)
//return
7.38905609893065
```


12) expm1(x) - ex-1

The expm1() function returns $e^x - 1$, where x is the arguments, and e the base of the natural logarithm.

Syntax-

expm1(x) - ex-1

Example:-

```
select "math"
showln expm1(0) //return: 0.0
showln expm1(1) //return: 1.718281828459045
```

13) floor(x) - returns floor of x

The floor() function returns the largest integer less than or equal to a given number.

Syntax-

floor(x)

Example:-

```
select "math"
showln floor(5.95) //return: 5
```

14) getExponent(x)

The getExponent() function returns the unbiased exponent used in the representation of a double or float

Syntax-

getExponent(x)

Example:-

```
select "math"
showln getExponent(4.5) //return: 2
```

15) hypot(x, y) - hypotenuse

The hypot() function returns the square of the sum of squares of its arguments,

Syntax-

hypot(x, y)

Example:-

```
select "math"
showln hypot(3, 4) // return: 5
showln hypot(5, 12) //return: 13
```

16) IEEERemainder(x, y) - remainder

The IEEERemainder() is a function which is used to return the remainder resulting from the division of a specified number by another specified number.

Syntax-

IEEERemainder(x, y)

Example:-

```
select "math"
showln remainder(31.34,2.2)
//return
0.5399999999999974
```

17) log(x) - logarithm of a number

The log() function returns the natural logarithm of a number, or the logarithm of number to base.

Syntax-

log(x)

Example:-

```
select "math"
showln log(0) // return: -infinity
showln log(2) //return: 0.6931471805599453
```

18) log1p(x) - natural logarithm

The log1p() function returns the natural logarithm (base e) of 1 + a number.

Syntax-

log1p(x)

Example:-

```
select "math"
showln log1p(0) //return: 0
showln log1p(1) // return: 0.693147180559453
```

19) log10(x)

The log10() function returns the base 10 logarithm of a number.

Syntax-

log10(x)

Example:-

```
select "math"
showln log10(1) //return: 0
showln log10(100000) //return: 5
```

20) max(x, y) - returns an item with highest value

The max() function returns an item with the highest value, or the item with the highest value in an iterable. If the values are strings, an alphabetically comparison is done.

Syntax-

max(x, y)

Example:-

```
select "math"
x = max(5, 10)
showln (x) //return: 10
```

if the values are strings, an alphabetical comparison is done.

```
select "math"
x = max("Mike", "J ohn", "Doe")
showln (x) //return: Mike
```

21) min(x, y)

The min() function returns the item with the lowest value, or the item with the lowest value in an iterable. If the values are strings, an alphabetical comparison is done.

Syntax-

max(x, y)

Example:-

```
select "math"
x = min(5, 10)
showln (x) //return: 5
```

If the values are string, an alphabetical comparison is done.

```
select "math"
x = min("Mike", "J ohn", "Doe")
showln (x) //return: Doe
```

22) nextAfter(x, y) - returns the floating-point number adjacent to the first argument in the direction of the second argument.

The nextafter() function in Dragon takes two arguments and returns the next representable value after x in the direction of y. If both arguments are equal then the second argument is returned.

Syntax-

nextAfter(x, y)

Example:-

```
select "math"
a = 98.2
b = 56.4
showln nextAfter(a, b)
//return 98.19999999999999
```

23) nextUp(x)

The nextUp() is a built-in math function in Dragon which returns the floating-point value adjacent to the parameter provided in the direction of positive infinity.

Syntax:-

nextUp(x)

Example:-

```
select "math"
a = 69.3
showln nextUp(a) // return: 69.30000000000001
```

24) pow(x, y) - to calculate number raised to the power of another number

The pow() function computes the power of a number. The pow() function takes two arguments (base value and power value) and, returns the power raised to the base number.

The pow() function returns the value of x to the power of y (x^y). If a third parameter is present, it returns x to the power of y, modulus z.

Syntax:-

pow(x,y)

Example:-

Return the value of 4 to the power of 3 (same as $4 * 4 * 4$):

```
select "math"
x = pow(4, 3)
showln x // 64
```

If a third parameter is present, it returns x to the power of y, modulus z.

Example:-

Return the value of 4 to the power of 3, modulus 5 (same as $(4 * 4 * 4) \% 5$):

```
select "math"
x = pow(4, 3, 5)
showln x // 12.8
```

25) rint(x)

The rint() functions round a floating-point number to the next integer value in the current rounding direction.

Syntax:-

rint(x)

Example:-

```
select "math"
x = 3.3
y = 4.3
a = rint(x)
b = rint(y)
showln (a) // 3.0
showln (b) // 4.0
```

26) round(x)

The round() function returns a floating point number that is a rounded version of the specified number, with the specified number of decimals.

The default number of decimals is 0, meaning that the function will return the nearest integer.

Syntax:-

round(x)

Example:-

```
select "math"
x = round(5.76543)
showln (x) // 6
```

27) signum(x)

This function returns the Sign function of a value passed to it as argument.

Syntax-

signum(value)

Example:-

```
select "math"
showln signum(20) //return 1.0
```

28) sin(x) - The sin() function returns the sine of a number.

Note: This method returns a value between -1 and 1, which represents the sine of the parameter x.

Syntax-

sin(x)

Example:-

```
select "math"
a = sin(0)
b = sin(3)
showln (a) // 0.0
showln (b) // 0.1411200080598672
```

29) sinh(x)

The sinh() function computes the hyperbolic sine of an argument. The sinh() function takes a single argument and returns the value of type double.

Syntax-

sinh(x)

Example:-

```
select "math"
showln sinh(3.5)
//return16.542627287634996
```

30) sqrt(x)

This function returns the square root of a value of type double passed to it as argument.

Syntax-

sqrt(x)

Example:-

```
select "math"
showln sqrt(30)
//return 5.477225575051661
```

31) tan(x)

The tan() function returns a numeric value that represents the tangent of the angle.

Syntax-

tan(x)

Example:-

```
select "math"
showln tan(toRadians(45))
//return 0.9999999999999999
```

32) tanh(x)

This function returns the hyperbolic tangent of a double value passed to it as argument.

Syntax-

tanh(value)

Example:-

```
select "math"
showln tanh(3.5)
//return 0.9981778976111987
```


33) toDegrees(x)

toDegrees() is a built-in math function which is used to convert an angle measured in radians to an approximately equivalent angle measured in degrees.

Syntax-

toDegrees(x)

Example:-

```
select "math"
showIn toDegrees(PI)
//return 180.0
```

34) toRadians(x)

The toRadians() is a built-in math function which is used to convert an angle measured in degrees to an approximately equivalent angle measured in radians.

Syntax-

toRadians(x)

Example:-

```
select "math"
showIn toRadians(180.0)
//return 3.141592653589793
```

35) ulp(x)

ulp() method is used to return the size of a ulp of the given argument, where, a ulp of the given value is the positive distance between floating-point value and the value next larger in magnitude.

Syntax-

ulp(x)

Example:-

```
select "math"
showIn ulp(34.543)
//return 7.105427357601002E-15
```

36) eval()

eval() is a function property of the global object. The argument of the eval() function is a string. If the string represents an expression, eval() evaluates the expression. If the argument represents one or more JavaScript statements, eval() evaluates the statements.

Syntax-

eval(x)

Example:-

```
select "math"
showIn eval("1 + 2 * 3")
//return 7
```

MODULE 4: THE DATE MODULE

The Date Modules contains functions for working with date and time.

To define the date module, use the select keyword follow by the module name in double quote:

```
select "date"
```

Constants

STYLE_FULL : number = 0

STYLE_LONG : number = 1

STYLE_MEDIUM : number = 2

STYLE_SHORT : number = 3

Types

DateValue

DateFormatValue

Functions:-

1) newDate()

newDate(...) - newDate() - returns
current date.

newDate(timestamp) - returns date
by given timestamp.

newDate(dateString) - parses and
returns date by given string.

newDate(pattern, dateString) -

parses and returns date by given
string in pattern format.

newDate(year, month, day) -
returns date by year, month and
day.

newDate(year, month, day, hour, minute) - returns date by year,
month, day, hour and minute.

newDate(year, month, day, hour, minute, second) - returns date by
year, month, day, hour, minute and second.

Example:-

```
select "date"
a = newDate()
showln a //return, 2020-06-25 09:26:30.141
```

```
select "date"
a = newDate(2019,11,07)
showln a //return, 2019-11-07 00:00:00.000
```

```
select "math"
a = newDate(2019,11,29,11,30)
showln (a)
//return, 2019-11-29 11:30:00.000
```

2) newFormat()

newFormat(...) - newFormat() - returns default date format. newFormat(pattern) - returns date format by given pattern.

newFormat(type) - returns format: 0 - default, 1 - date, 2 - time, 3 - date and time.

newFormat(pattern, locale) - returns date format by given pattern and locale.

newFormat(type, style) - returns format: 0 - default, 1 - date, 2 - time, 3 - date and time. style: 0 - full, 1 - long, 2 - medium, 3 - short. Returns DateFormatValue.

Example:-

```
select "date"
a = newFormat("dd/mm/yyyy")
d = newDate()
showln formatDate(d,a)
//returns, 26/07/2020
```

3) formatDate()

This function formats date by given format and returns string.

Syntax-

formatDate(date, format)

Example:-

```
select "date"
a = newFormat("dd/mm/yyyy")
d = newDate(2019,11,07)
showln formatDate(d,a)
//returns, 07/1/2019
```

4) parseDate()

This function parses date from string by given pattern. Returns DateValue.

Syntax-

parseDate(dateString, format)

Example:-

```
select "date"
showln parseDate("2019/10/09",
newFormat('yyyy/MM/dd'))
//returns, 2019/10/09
```

5) toTimestamp()

This function returns timestamp of given date in milliseconds.

Syntax-

toTimestamp(date)

Example:-

```
select "date"
d = newDate()
showln toTimestamp(d)
//return, 1593857554769
```

MODULE 5: THE FILES MODULE

The files module contains functions for working with files.

To define the files module, use the select keyword follow by the module name in double quote:

```
select "files"
```

Constants

FILES_COMPARATOR : FUNCTION =

func(f1, f2) = compare(f1, f2)

function which compares two file descriptors

Functions

1) fopen()

The fopen() function is used to open a file to perform operations such as reading, writing etc in a dragon program.

File Modes:-

- "" - opens file or directory for getting info;
- "r" - opens file for read in text mode;
- "rb" - opens file for read in binary mode;
- "w" - opens file for write in text mode;
- "w+" - opens file for append in text mode;
- "wb" - opens file for write in binary mode;
- "wb+" - opens file for append in binary mode

Syntax:-

fopen(path, mode)

Example:-

```
select "files"
f1 = fopen("text.txt")
/* opens file text.txt for read in text mode */

f2 = fopen("E:/1.dat", "rbwb")
/* opens file 1.dat on drive E for binary read and write */
```

2) fclose()

The fclose() function closes a file in dragon program.

Syntax-

fclose(file)

Example:-

```
select "files"
f1 = fopen("text.txt")
fclose(f1)
//file closed
```

3) fileSize()

The fileSize() function is an inbuilt function in dragon which is used to return the size of a specified file.

Syntax-

fileSize(file)

Example:-

```
select "files"
f1 = fopen("text.txt")
showln fileSize(f1)
//return file size
```

4) exists()

The exists() function tests the existence of the file or directory defined by the abstract pathname. Returns 1 if exists, 0 if file doesn't exist.

Syntax-

exists(file)

Example:-

```
select "files"
f1 = fopen("text.txt")
showln exists(f1)
//return 1 (exists)
```

5) canExecute()

The canExecute() function determines whether the program can execute the specified file pathname. If the file path exists the application is allowed to execute the file, This will Returns 1 if true, 0 otherwise.

Syntax-

canExecute(file)

Example:-

```
select "files"
f1 = fopen("text.txt")
showln canExecute(f1)
//return 0 (No permission) if file
does not exist.
```

6) canRead()

The canRead() function

This function checks reads permission of file. Returns 1 if true, 0 otherwise.

Syntax-

canRead(file)

Example:-

```
select "files"
f1 = fopen("text.txt")
showln canRead(f1)
//return 1 (permission true)
```

7) canWrite()

This function determines whether the program can write the file denoted by the abstract path name. The function returns true if the abstract file path exists and the application is allowed to write the file.

Syntax-

canWrite(file)

Example:-

```
select "files"  
f1 = fopen("text.txt")  
showln canWrite(f1)  
//return 1 (permission true)
```

8) copy()

This function copies file from source to destination. Returns 1 if success, -1 otherwise.

Syntax-

copy(source,destination)

Example:-

```
select "files"  
showln copy("text.txt","new.txt")  
//return 1 (copy success)
```

9) delete()

This function removes file or directory. Returns 1 if delete was successful, 0 otherwise.

Syntax-

delete(file/ directory)

Example:-

```
select "files"  
showln delete("new.txt")  
//return 1 (delete success)
```

10) getParent()

This function returns parent path of the given file.

Syntax-

getParent(file)

Example:-

```
select "files"
f = fopen("text.txt")
showln getParent(f)
//return My files(Folder name)
```

11) isDirectory()

This function checks if given value is a directory. Returns 1 if true, 0 otherwise.

Syntax-

isDirectory(directory)

Example:-

```
select "files"
showln isDirectory("text.txt")
//return 0 (not directory)
```

12) isFile()

This function checks if given value is a file. Returns 1 if true, 0 otherwise.

Syntax-

isFile(value)

Example:-

```
select "files"
showln isFile("text.txt")
//return 1 (true it's a file)
```

13) isHidden()

This function checks if file or directory is hidden. Returns 1 if hidden, 0 otherwise.

Syntax-

isHidden(file)

Example:-

```
select "files"
showln isHidden("text.txt")
//return 0 (Not hidden)
```

14) lastModified()

This function returns last modification time. Returns time in milliseconds.

Syntax-

lastModified(file)

Example:-

```
select "files"
f = fopen("text.txt")
showln lastModified(f)
//return 1593930215000
```

15) listFiles()

This function returns array with filenames in given directory.

Syntax-

listFiles(file)

Example:-

```
select "files"
f = fopen("/home/ubuntu/my","")

/* opens directory examples for getting information */
showln listFiles(f)

/* gets array with filenames in directory */
```

16) mkdir()

This function creates a directory specified by a pathname Returns 1 if operation was successful, 0 otherwise.

Syntax-

mkdir(value)

Example:-

```
select "files"
showln mkdir("newfolder")
//return 1 (Directory created!)
```

17) mkdirs

This function creates the multiple directories. Returns 1 if operation was successful, 0 otherwise.

Syntax-

mkdirs(value)

Example:-

```
select "files"
showln mkdirs("newfolder/test/exp")
/* return 1 (Multiple Directory created!) */
```

18) rename()

The rename() function is used to rename a file or directory(move a file or directory) . This method renames a source files/directory to specify destination file/directory. Return 1 if success, 0 otherwise.

Syntax-

rename(value)

Example:-

```
select "files"
showln rename("text.txt","ntext.txt")
/* return 1 (Moved or renamed)*/
```

19) setReadOnly()

The setReadOnly() function marks the specified file or directory such that only read operations are allowed on the file or directory. Returns 1 if success, 0 otherwise.

Syntax-

setReadOnly(file)

Example:-

```
select "files"
f = fopen("text.txt")
showln setReadOnly(f)
// return 1 (Success)
```

20) setExecutable()

The setExecutable() function sets TV owners or everybody's permission to execute the abstract pathname. This function sets execute permission. Returns 1 if success, 0 otherwise.

Syntax-

setExecutable(file,true or false)

Example:-

```
select "files"
f = fopen("text.txt")
showln setExecutable(f, true)
// return 1 (Success)
```

21) setReadable()

The setReadable() function sets owner's or everybody's permission to read the abstract pathname. This function sets read permission. Returns 1 if success, 0 otherwise.

Syntax-

setReadable(file,true or false)

Example:-

```
select "files"
f = fopen("text.txt")
showln setReadable(f,true)
// return 1 (Success)
```

22) setWritable()

The function sets the owner or everybody's permission to write the abstract pathname. This function sets write permission. Returns 1 if success, 0 Otherwise.

Syntax-

setWritable(file,true or false)

Example:-

```
select "files"
f = fopen("text.txt")
showln setWritable(f,true)
// return 1 (Success)
```

23) setLastModified()

This function returns the time denoted by the abstract pathname was last modified. The function returns long value measured in milliseconds representing the time the file was last modified.

This function sets last modified time. Returns 1 if success, 0 otherwise.

Syntax-

setLastModified(file,time)

Example:-

```
select "files"
f = fopen("text.txt")
showln setLastModified(f, 100000000)
//time in milliseconds.
// return 1 (Success)
```

24) flush()

This function flushes write buffer into file.

Syntax-

flush(file)

Example:-

```
select "files"
f = fopen("text.txt","rw")
writeText(f,"Hello World")
// Writing text in file.
flush(f)
```

25) readAllBytes()

The readAllBytes() is an inbuilt function that is used to open a specified or created binary file and read the content of the file into a bytes array and then closes the file.

Syntax-

readAllBytes(file)

Example:-

```
select "files"
f = fopen("file.bin","rb")
array = readAllBytes(f)
//return bytes array
```

26) readBoolean()

This function reads boolean (1 byte). Returns 0 if byte was 0, 1 otherwise.

Syntax-

readBoolean(file)

Example:-

```
select "files"
/* reads 10 bytes, starting from 21 byte */
f = fopen("file.bin","rb")
val = readBoolean(f)
```

27) readByte()

The function is used to read and return one input byte.

Syntax-

readByte(file)

Example:-

```
select "files"
f = fopen("file.bin","rb")
val = readByte(f)
```

28) readBytes()

This function reads length bytes of file to array starting from offset. Returns number of read bytes.

Syntax-

readBytes(file,array,offset,length)

Example:-

```
select "files"
select "std"
f1 = fopen("file.bin", "rb")
array = newarray(2048)
readedCount = readBytes(f1, array)
// reads 2048 bytes
readedCount = readBytes(f1, array, 10)
/* reads 2048 bytes starting from 11 byte */
readedCount = readBytes(f1, array, 20, 10)
/* reads 10 bytes, starting from 21 byte */
```

29) readChar()

This function reads one char (2 bytes). Returns number char's code.

Syntax:-

readChar(file)

Example:-

```
select "files"
f = fopen("text.txt", "r")
val = readChar(f)
//return 65 (Char code of A)
```

30) readDouble()

readDouble() method reads 8 bytes and returns one double value.

Syntax:-

readDouble(file)

Example:-

```
select "files"
f = fopen("file.bin", "rb")
val = readDouble(f)
```


31) readFloat()

method reads four bytes of the input stream and returns a float value.

Syntax-

readFloat(file)

Example:-

```
select "files"
f = fopen("file.bin","rb")
val = readFloat(f)
```

32) readInt()

This function reads 4 bytes integer number.

Syntax-

readInt(file)

Example:-

```
select "files"
f = fopen("file.bin","rb")
val = readInt(f)
```

33) readLine()

This function reads line from file opened in text mode.

Syntax-

readLine(file)

Example:-

```
select "files"
f = fopen("text.txt","r")
val = readLine(f)
```

34) readLong()

readLong() method reads eight input bytes and returns a long value.

Syntax-

readLong(file)

Example:-

```
select "files"
f = fopen("text.txt","r")
val = readLong(f)
```

35) readShort()

readShort() method reads two input bytes and returns a short value.

Syntax-

readShort(file)

Example:-

```
select "files"
f = fopen("text.txt","r")
val = readShort(f)
```

36) readText()

This function reads all file's content as string.

Syntax-

readText(file)

Example:-

```
select "files"
f = fopen("text.txt","r")
showln readText(f)
//return Hello World (file data)
```

37) readUTF()

readUTF() method reads in a string that has been encoded using a modified UTF-8 format. The string of character is decoded from the UTF and returned as String.

This function reads string in binary mode.

Syntax-

readUTF(file)

Example:-

```
select "files"
f = fopen("text.txt","r")
val = readUTF(f)
```

38) writeBoolean()

This function writes boolean (0 or 1) to file.

Syntax-

writeBoolean(file, true or false)

Example:-

```
select "files"
f = fopen("text.txt","w")
writeBoolean(f,true)
flush(f)
fclose(f)
```

39) writeByte()

This function writes one byte to file.

Syntax-

writeByte(file,value)

Example:-

```
select "files"
f = fopen("text.txt","w")
writeByte(f,0)
flush(f)
fclose(f)
```

40) writeBytes()

This function writes length bytes to file from byte array starting from offset.

Syntax-

writeBytes(f, array, offset, length)

Example:-

```
select "files"
f = fopen("text.txt","w")
array = ["Dragon","Programming", "Language"]
writeBytes(f1, array)
writeBytes(f1, array, 10)
writeBytes(f1, array, 20, 10)
flush(f)
fclose(f)
```

41) writeChar()

This function writes one char (2 bytes) to file. Value can be number -writes number, or string - writes code of first symbol.

Syntax-

writeChar(file,value)

Example:-

```
select "files"
f = fopen("text.txt","w")
writeChar(f,"A")
flush(f)
fclose(f)
```

42) writeDouble()

This function writes 8 bytes double number to file.

Syntax-

writeDouble(file,value)

Example:-

```
select "files"  
f = fopen("text.txt","wb")  
writeDouble(f,34632523.3464)  
flush(f)  
fclose(f)
```

43) writeFloat()

This function writes 4 bytes float number to file.

Syntax:-

writeFloat(file,value)

Example:-

```
select "files"  
f = fopen("text.txt","wb")  
writeFloat(f,242.34)  
flush(f)  
fclose(f)
```

44) writeInt()

This function writes 4 bytes integer number to file.

Syntax:-

writeInt(file,value)

Example:-

```
select "files"  
f = fopen("text.txt","wb")  
writeInt(f,12525)  
flush(f)  
fclose(f)
```

45) writeLine()

This function writes string to file in text mode adds line break at the end of the string.

Syntax-

writeLine(file,value)

Example:-

```
select "files"
f = fopen("text.txt","w")
writeLine(f,"Hello World")
flush(f)
fclose(f)
```

46) writeLong()

This function writes 8 bytes long number to file.

Syntax-

writeLine(file,value)

Example:-

```
select "files"
f = fopen("text.txt","wb")
writeLong(f,2362623636334)
flush(f)
fclose(f)
```

47) writeShort()

This function writes 2 bytes short number to file.

Syntax-

writeShort(file,value)

Example:-

```
select "files"
f = fopen("text.txt","wb")
writeShort(f,23)
flush(f)
fclose(f)
```

48) writeText()

This function writes string to file in text mode. Unlike writeLine does not add line break.

Syntax-

writeText(file,value)

Example:-

```
select "files"  
f = fopen("text.txt","w")  
writeText(f,"This is an Example!")  
flush(f)  
fclose(f)
```

49) writeUTF()

This function writes string to file in binary mode.

Syntax-

writeUTF(file,value)

Example:-

```
select "files"  
f = fopen("text.txt","wb")  
writeUTF(f,"Test")  
flush(f)  
fclose(f)
```

MODULE 6: THE HTTP MODULE

The http module in Dragon is a module that allows Dragon to transfer data over the Hyper Text Transfer Protocol (HTTP). The HTTP module. It is used for making a web server to handle requests and provide responses.

To define the http module, use the select keyword follow by the module name in double quote:

```
select "http"
```

Functions:

1) http

This function performs request with given method, object params and connection options to url, response will be send to function callback.

Syntax-

`http(url)` - performs GET-request to url.

`http(url, method)` - performs request with method (GET, POST, PUT, DELETE, PATCH, OPTIONS) to url.

`http(url, callback)` - performs GET-request to url, response will be send to function callback.

`http(url, method, params)` - performs request with given method and object params to url.

`http(url, method, callback)` - performs request with given method to url, response will be send to function callback.

`http(url, method, params, callback)` - performs request with given method and object params to url, response will be send to function callback.

`http(url, method, params, options, callback)` - performs request with given method, object params and connection options to url, response will be send to function callback.

Connection options is a object (map):

- `header` - sets http-header (string or array).
- `encoded` - is params object already urlencoded.
- `content_type` - sets Content-Type.
- `extended_result` - marks that response should be extended and should contains:
 - o `text` - server response text
 - o `message` - server response message
 - o `code` - server response code
 - o `headers` - response http-header
 - o `content_length` - Content-Length
 - o `content_type` - Content-Type

Example:-

```
select "http"
http("http://jsonplaceholder.typicode.com/users", "POST", {'name': "Dragon", "versionCode": 10},
func(v) {
showln "Added: " + v
})
```

2) download(url) - downloads content by url as bytes array

Example:-

```
select "http"
select "files"
bytes = download("http://url")
f = fopen("file", "wb")
writeBytes(f, bytes)
flush(f)
fclose(f)
```

3) urlencode(str) -

This function converts string to URL-format

Syntax-

urlencode(string)

Example:-

```
select "http"
select "std"
url =urlencode("www.google.com")
http(url,::echo)
//chaining echo function to print response.
```

MODULE 7: THE BASE64 MODULE

Base64 module provides functions to encode binary data to Base64 encoded format and decode such encodings back to binary data. It implements Base64 encoding and decoding as specified in RFC 3548.

To define the http module, use the select keyword follow by the module name in double quote:

```
select "base64"
```

Constants:

BASE64_URL_SAFE : number = 8

Url safe encoding output

Functions

1) base64decode()

Decodes base64-encoded byte array or string into byte array.

Syntax-

base64encode(data)

Example:-

```
select "base64"
showln base64encode("Hello")
/* return [83, 71, 86, 115, 98, 71, 56, 61] */
```

2) base64decode(data)

Encodes byte array or string into base64-encoded byte array

Syntax-

base64decode(data)

Example:-

```
select "base64"
select "std"
arr = base64decode([83, 71, 86, 115, 98, 71, 56, 61])
for val : arr {
show toChar(val)
}
//return Hello
```

3) `base64encodeToString(data)`

This function encodes byte array or string into base64-encoded string.

Syntax-

`base64encodeToString(data)`

Example:-

```
select "base64"
showln base64encodeToString("Dragon")
//return RHJ hZ29u
```

MODULE 8: THE J SON MODULE

This module contains functions for working with J SON data. You can use it to process J SON that is embedded in other file formats. J SON (J avaScript Object Notation).

J SON is probably most widely used for communicating between the web server and client in an AJ AX application.

To define the json module, use the select keyword follow by the module name in double quote:

```
select "json"
```

Functions

1) jsondecode()

This function converts data to json string

Syntax-

```
jsondecode(data)
```

Example:-

```
select "json"
show jsondecode('{ "key1\ ":1,\ "key2\ ":[1,2,3],\ "key3\ ":\'text\' }')
/* return {key2=[1, 2, 3], key3=text, key1=1}
*/
```

2) jsonencode()

This function converts string to data

Syntax-

```
jsonencode(jsonString)
```

Example:-

```
select "json"
data = {
  "key1": 1,
  "key2": [1, 2, 3],
  "key3": "text"
}
show jsonencode(data)
// { "key1":1,"key2":[1,2,3],"key3":"text" }
```

MODULE 9: THE YAML MODULE

The yaml module Contains functions for working with the yaml format.

To define the YAML module, use the select keyword follow by the module name in double quote:

```
select "yaml"
```

Functions

1) yamlencode()

This function converts yaml string to data.

Syntax-

yamlencode(yamlString)

Example:-

```
select "yaml"
showln yamlencode({"Name":"Aavesh","Position":"Creator"})

/* return {Name:
  Aavesh,
  Position:
  Creator} */
```

2) yamldecode()

This function converts data to yaml string.

Syntax-

yamldecode(data)

Example:-

```
select "yaml"
showln yamldecode("Name: Aavesh \nPosition: Creator ")

/* return {Name=Aavesh,
  Position=Creator} */
```

MODULE 10: THE OUNIT MODULE

The ouunit module Contains functions for testing. Invokes all functions with prefix test and checks expected and actual values, counts execution time.

To define the ouunit module, use the select keyword follow by the module name in double quote:

```
select "ouunit"
```

Functions

1) runTests()

This function executes tests and returns information about its results.

Syntax-

```
runTests()
```

Example:-

```
select "ouunit"
showln runTests()
```

2) assertEquals()

This function checks that two values are equal.

Syntax-

```
assertEquals(expected, actual)
```

Example:-

```
select "ouunit"
func testAdditionOnNumbers() {
    assertEquals(6, 0 + 1 + 2 + 3)
}
showln runTests()

/* return, testAdditionOnNumbers [passed]
Elapsed: 0.0000 sec Tests run: 1, Failures: 0, Time elapsed: 0.0000 sec
*/
```

3) assertFalse()

This function checks that value is false (equals 0).

Syntax-

```
assertFalse(actual)
```

Example:-

```
select "ounit"
func testFail() {
  assertFalse(true)
}
showln runTests()

/* return, testFail [FAILED]
Error: Expected false, but found true.
Elapsed: 0.0001 sec Tests run: 1, Failures: 1, Time elapsed: 0.0001
sec
*/
```

4) assertEquals()

This function checks that two values are not equal.

Syntax-

assertEquals(expected, actual)

Example:-

```
select "ounit"
func testNotEqual() {
  assertEquals(5,2+3)
}
showln runTests()

/* return, testNotEqual [FAILED]
Error: Values are equals: 5
Elapsed: 0.0001 sec
Tests run: 1, Failures: 1, Time elapsed: 0.0001 sec
*/
```

5) assertSameType()

This function checks that types of two values are equal.

Syntax-

assertSameType(expected, actual)

Example:-

```
select "ounit"
func testType() {
assertSameType(5,12)
}
showln runTests()

/* return, testType [passed]
Elapsed: 0.0000 sec
Tests run: 1, Failures: 0, Time elapsed: 0.0000
sec
*/
```

6) assertTrue()

This function checks that value is true (not equals 0).

Syntax:-

assertTrue(actual)

Example:-

```
select "ounit"
func testTr() {
assertTrue(true)
}
showln runTests()

/* return, testTr [passed]
Elapsed: 0.0000 sec
Tests run: 1, Failures: 0, Time elapsed: 0.0000
sec
*/
```


MODLUE 11: THE GUI MODULE

Contains functions for working with GUI (Graphic User Interface)

To define the GUI module, use the select keyword follow by the module name in double quote:

```
select "GUI "
```

Constants

BorderLayout : map =

```
{
  AFTER_LINE_ENDS=After,
  LINE_END=After,
  LINE_START=Before,
  BEFORE_LINE_BEGINS=Before,
  CENTER=Center,
  EAST=East,
  BEFORE_FIRST_LINE=First,
  PAGE_START=First,
  AFTER_LAST_LINE=Last,
  PAGE_END=Last,
  NORTH=North,
  SOUTH=South,
  WEST=West
}
BoxLayout : map = {X_AXIS=0, Y_AXIS=1, LINE_AXIS=2, PAGE_AXIS=3}
```

DISPOSE_ON_CLOSE : number = 2

DO_NOTHING_ON_CLOSE : number = 0

EXIT_ON_CLOSE : number = 3

HIDE_ON_CLOSE : number = 1

Align : map =

```
{
  BOTTOM=3,
  CENTER=0,
  EAST=3,
  HORIZONTAL=0,
  LEADING=10,
  LEFT=2,
  NEXT=12,
  NORTH=1,
  NORTH_EAST=2,
  NORTH_WEST=8,
  PREVIOUS=13,
```

```
RIGHT=4,  
SOUTH=5,  
SOUTH_EAST=4,  
SOUTH_WEST=6,  
TOP=1,  
TRAILING=11,  
VERTICAL=1,  
WEST=7  
}
```

Functions

1) newWindow()

This function creates a new window

Syntax-

newWindow(title)

Example:-

```
select "GUI" //selecting GUI module  
win = newWindow("My Window") // Object created of newWindow. Window title is My Window  
// this object will be used for defining the properties.
```

Properties of newWindow:

a) setSize(width, height)

Example:-

```
win.setSize(400,400)  
//size of window. 2 parameters require and both integer. width and height are in pixels.
```

b) setLocation(x axis,y axis)

Example:-

```
win.setLocation(100,100)  
//location of window on screen. 2 parameters require and both integer. x axis and y axis location in  
pixels
```

`message(text = "")` - creates new Message window

`newOpenFile()` - creates new Open File window

`newSaveFile()` - creates new Save File window

`borderLayout(hgap = 0, vgap = 0)` - creates BorderLayout

`boxLayout(panel, axis = BoxLayout.PAGE_AXIS)` - creates BoxLayout

`cardLayout(hgap = 0, vgap = 0)` - creates CardLayout

`flowLayout(align = FlowLayout.CENTER, hgap = 5, vgap = 5)` - creates FlowLayout

`gridLayout(rows = 1, cols = 0, hgap = 0, vgap = 0)` - creates GridLayout

`noLayout()` - creates null layout

`newButton(text = "")` - creates new button

`newText(text = "", align = SwingConstants.LEADING)` - creates new label

`newPanel(layoutManager = ...)` - creates new panel with optional layout manager

`newTextBox(text = "")` - creates new text box

`newPassBox(text = "")` - creates new password box

`newTextarea()` - creates new Text Area

`newMenuBar()` - creates new Menu Bar

`newMenu()` - creates new Menu

`newMenuItem(text = "")` - creates new Menu Item

`newSelectBox()` - creates new Select Box

`newScroll()` - creates new Scroll Box

CONCLUSION

This book will be frequently updated on several releases of Dragon Programming Language. Dragon Book Edition 1 - First Edition contains everything you need to get started learning dragon programming language with modules section included. The next edition which is Second Edition will contain more informations and details of the language from basics to advanced level programming in Dragon language.

Note that a GUI book will also be released which will explain in details on how to get started developing GUI applications in Dragon programming language. The book will be titled "Dragon Book - Developing GUI applications". which will give more tutorials working with GUI in dragon language.

If you find any issue with this book or you have any suggestions on developing this book kindly visit this site <https://github.com/dragon-language-projects> and make a report.

ABOUT AUTHOR

Adesina Festus known also as Moralist Festus is software developer and Co community developer @dragon-language-projects. Passionate at building tools to solve real world problems, technical writing and aiding other devs and love contributing to open source projects. and building community.

Facebook: MoralistFestus

Twitter: moralistfestus

Instagram: programmer.festus

Github: github.com/MoralistFestus