Direct Abstraction 15 User Manual

Matthew Kaes

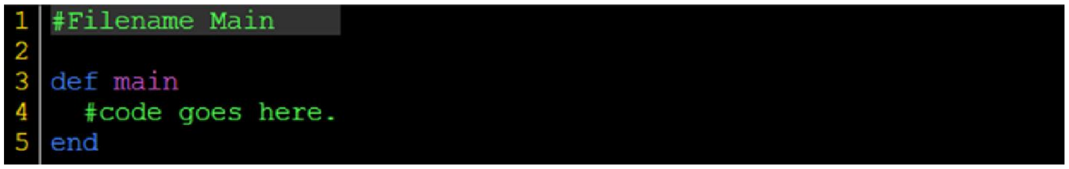
Table of Contents

Using Direct Abstraction:

Direct Abstraction is based on two parts; the interpreter which parses you're code and executes you're program, and the abstraction library which informs the interpreter how to perform the required operations. In most cases the interpreter and the abstraction library are built into one binary file, but this is not always the case as the interpreter can have added functionality with the use of an extension library in the form of a .DLL file. To run the interpreter simply have a file named "main.txt" which has you're code in plain text format in the same directory as the interpreter and the abstraction library. A new instance of the interpreter will launch and your code will be loaded into memory. For DOS support direct abstraction can run natively from the command prompt and can also run natively in Cygwin.

Running Code:

In order for your code to run it must have a valid entry point. Depending on your version of DA (direct abstraction) your code will have different entry points. For version of DA older than 11 the entry point for the code will be the function main. The smallest executable code for DA 11 and above is as follows.



```
#Filename Main

def main
  #code goes here.
end
```

Direct Abstraction IDE:

Direct Abstraction comes with a dedicated IDE (Integrated Development Environment) on the Windows operating system. From the Direct Abstraction IDE you can run all of your code with a click of a button, choose which version of DA you wish to run under, and manage your code in a simple file like system. The IDE also comes with features such as syntax highlighting and will embed the require headers for your code.

Building Code in the IDE:

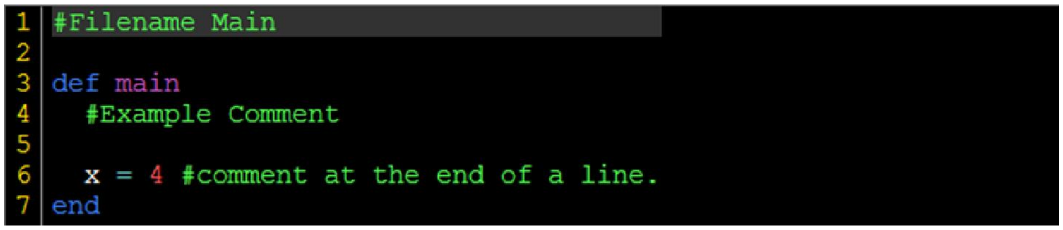
When you first start up the IDE you will be met with a stub project that includes a test Main script. From here you add to the project to run your own code or add new scripts to the project with the "Add Script" button. The text box above this button will make the name for the new script that you add to your project. This name is not permanent however and can be changed at any time by modifying the "#Filename" declaration at the start of your file. Once you are happy with your code you can run it at any time from the IDE by pressing the "Test Code" button. Testing code does not lock up the IDE and you are able to spawn multiple test session at the same time by clicking the button multiple times. You can also change what version of DA you are using by changing the drop down box next to the "Test Code" button.

Syntax:

Direct Abstraction has syntax that is similar to most higher level languages such as Ruby and Python. Direct Abstraction supports weak typing like most of these languages as well so all types are inferred at runtime.

Comments:

All comments in Direct Abstraction use the "#" character. All characters after the "#" character are considered comments and will be stripped out at run time. Directives are also a type of comment at the start of the file. It is also possible to use a comment mid line by placing a second "#" to terminate the comment; however this is not supported in all versions of DA.



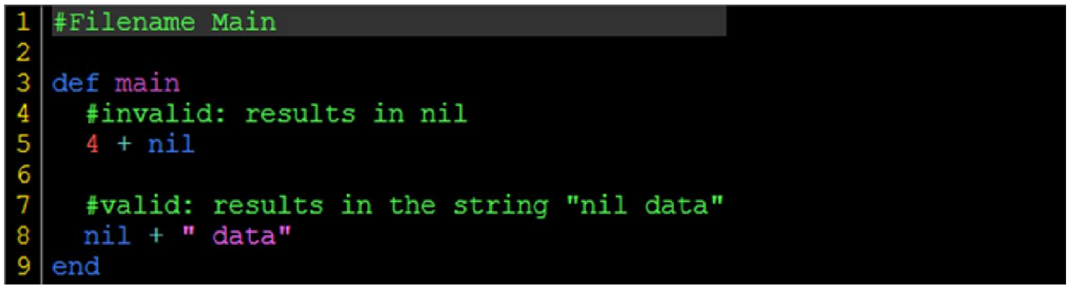
```
#Filename Main

def main
  #Example Comment

  x = 4 #comment at the end of a line.
end
```

Base types:

Direct Abstraction has several base data types. From these base types all other types are built on. Since in DA you have no control over implicit casting DA follows the "lossless" implicit cast. This means that base types will only go up to higher information types but never down. All the base times from lowest order to highest order are; Nil, Integer, Double, String, Container, Reference. From this order we can easily derive the rules for implicit casting. If an operation takes in an Integer and a Double the Integer will be converted to a Double and the operation will be performed however the Double will never be converted to an Integer. You can force a type conversion (explicit casting) by using the "Convert" function. (See "Built In Library")

Nil:

Nil is any non-initialized data. Nil converts into zero when cast into an Integer, 0.0 when cast into a Double, “nil” when cast to a string, and ‘empty’ when cast into a container. “nil” is also a registered keyword and can be used to create and return nil whenever you desire. Nil can also be a byproduct of a function or an operation that didn’t have valid arguments such as “nil + 4” or ““string” / 6”.



```
#Filename Main

def main
  #invalid: results in nil
  4 + nil

  #valid: results in the string "nil data"
  nil + " data"
end
```

Integer:

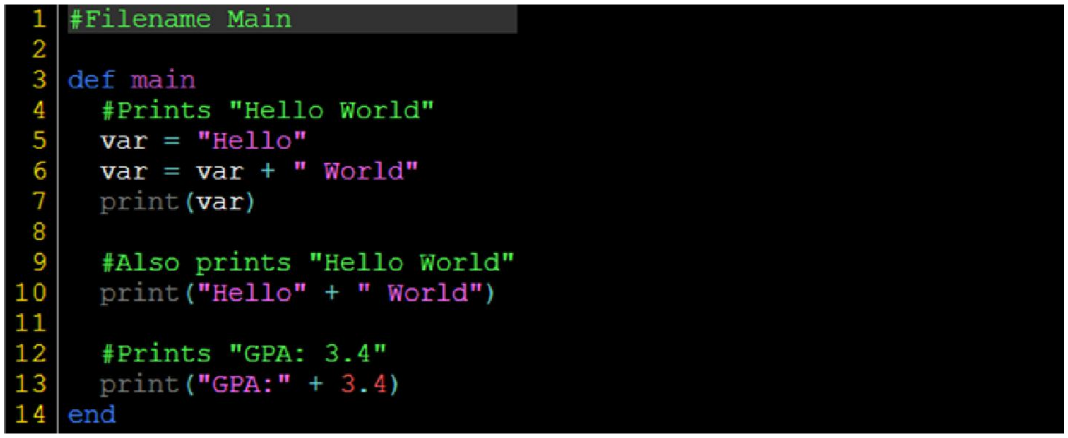
Integer is the lowest order base type to hold useful information. The integer type holds onto all whole numbers from $-(2^{63})$ to $2^{63} - 1$. Integers are similar to the “long long int” data type in C/C++.

Double:

Doubles hold onto and deal with all floating point arithmetic. Doubles take the form of 0.0 where the decimal place is required. If a double is a whole number such as 2 the form of the double must be 2.0. This resolves issues that could be mistaken for the dot operator (.) and also makes sure that it is differentiable from an integer.

Strings:

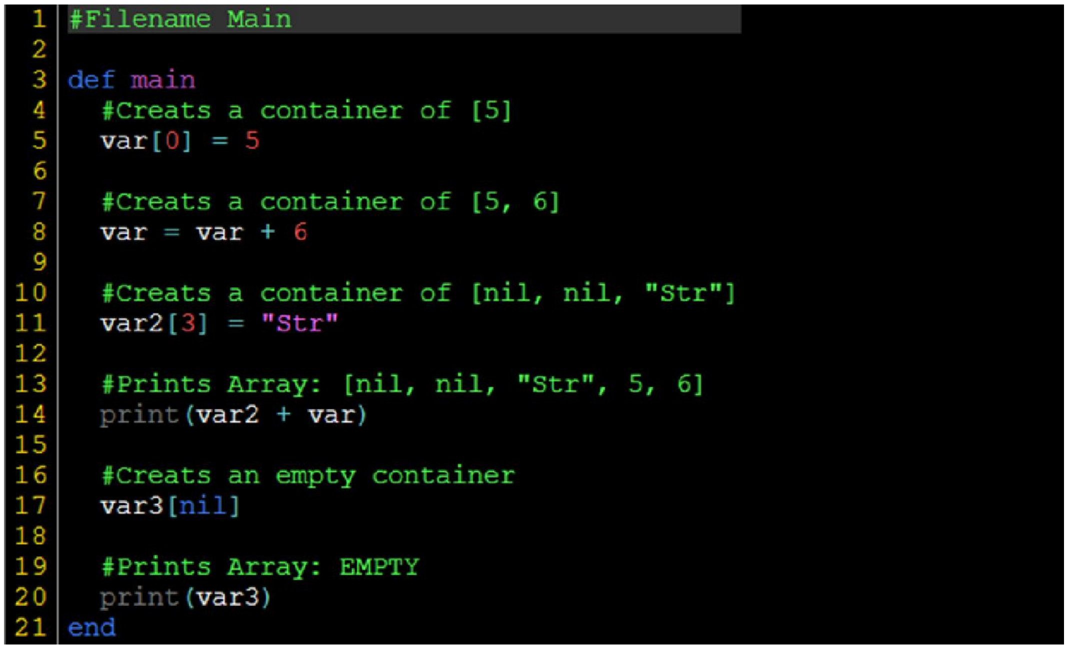
Strings are created by putting any line of code in quotes (“ “). Strings are not limited in any size and will grow in memory accordingly in order to match the size required for them. You can manipulate strings by using the addition operator (+) to append to the string. Strings also support a small list of special escape characters such as “\n” which will replace the two characters with the newline character at runtime.



```
#Filename Main

def main
  #Prints "Hello World"
  var = "Hello"
  var = var + " World"
  print(var)

  #Also prints "Hello World"
  print("Hello" + " World")

  #Prints "GPA: 3.4"
  print("GPA:" + 3.4)
end
```

Containers:

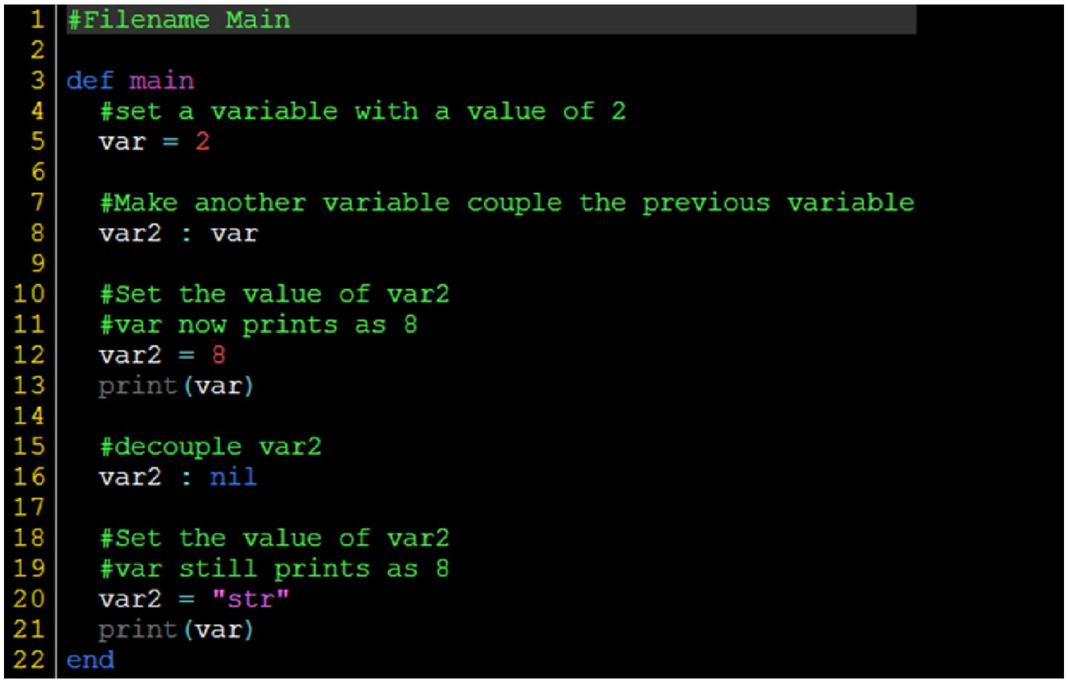
Containers hold any number of objects from Integers, nil objects, doubles, strings, and even other containers. Containers can hold either a single type of object, or any number of objects. Containers in this sense are not limited in size or types held. When constructing containers the container will always allocated and release whatever memory is required in order for the container to operate. Containers have two operators that can be used on them including the bracket operator ([,]) and the addition operator (+) that appends an element to the end of the container. When using the addition operator on two arrays the second container is instead concatenated to the first container and a new container composed of both sets is returned. Because Direct Abstraction always assures that there is memory for an array a symbol will be created into a container if the bracket operator is ever used on it and the maximum required index for the container will become its new size. Even if a new value isn't assigned to the memory location the container will still grow. If a container grows and it has no data to fill with all of the empty space will be padded with "nil" data. Since containers always assure that the index you are accessing is valid you can use the "nil" keyword as an index to create and empty set.



```
#Filename Main

def main
  #Creats a container of [5]
  var[0] = 5

  #Creats a container of [5, 6]
  var = var + 6

  #Creats a container of [nil, nil, "Str"]
  var2[3] = "Str"

  #Prints Array: [nil, nil, "Str", 5, 6]
  print(var2 + var)

  #Creats an empty container
  var3[nil]

  #Prints Array: EMPTY
  print(var3)
end
```

Reference:

A reference is created only in two instances, when two variables are coupled with the couple operator (":") and when a dynamic object is created with the "new" keyword. References act like the object and can be assigned to and used as if it was that object but is technically a different symbol. References allow for objects to be passed to functions in such a way that the function can affect an external value rather than relying solely on return values.



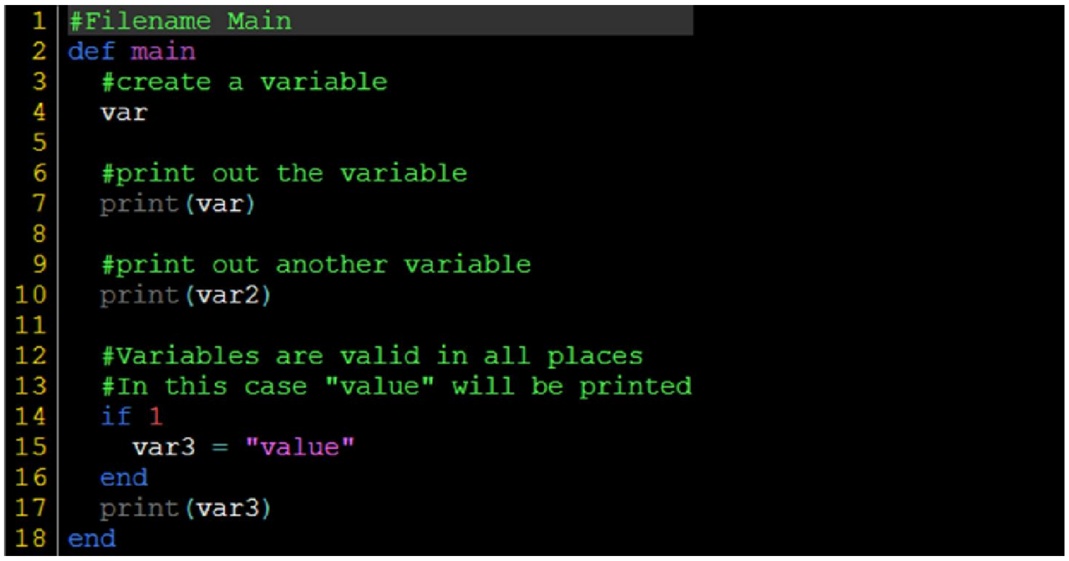
```
#Filename Main

def main
  #set a variable with a value of 2
  var = 2

  #Make another variable couple the previous variable
  var2 : var

  #Set the value of var2
  #var now prints as 8
  var2 = 8
  print(var)

  #decouple var2
  var2 : nil

  #Set the value of var2
  #var still prints as 8
  var2 = "str"
  print(var)
end
```

Symbols:

There are three different types of symbols that you can use to store all of your information in and preform you're operators on. The three types of operators are variables, globals, and attributes. These three types of symbols are all the same and have the exact same behavior and the only thing that differs between them is the scope in which the symbol exists. Variables are function scoped and exist for the entirety of the function. Globals will remain in effect for the entirety of the program. Attributes are special and are dynamic object scoped and will remain for as long as the dynamic object that they are attached to remains in working memory. This means that attributes will be tied to another symbol be it a variable, a global, or another attribute.

Variables:

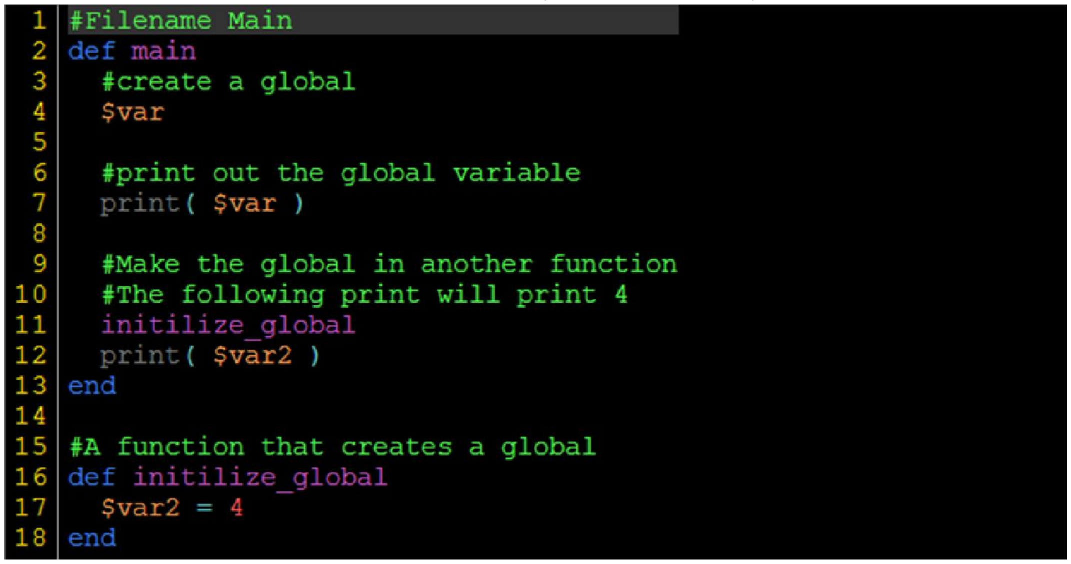
Variables take the form of any regular word that is not a keyword or a registered word (such as a function or class made by the user). A variable must start with a readable character and can contain numbers, underscores, and other characters. Variables are function scoped which means that in the function that they are declared in they are useable in all places. Since variables are function scoped, all variables used in a function are constructed when the function is called.



```
#Filename Main
def main
  #create a variable
  var

  #print out the variable
  print(var)

  #print out another variable
  print(var2)

  #Variables are valid in all places
  #In this case "value" will be printed
  if 1
    var3 = "value"
  end
  print(var3)
end
```

Globals:

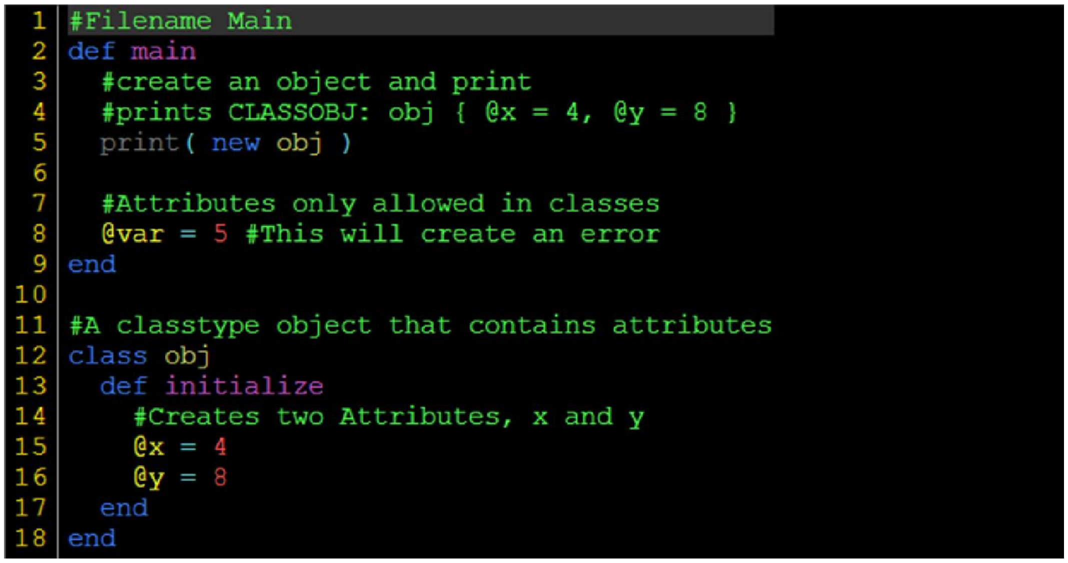
Globals have all of the same syntax requirements as variables but require a "$" at the start of the variable name. Globals can be created anywhere and at any time.



```
#Filename Main
def main
  #create a global
  $var

  #print out the global variable
  print( $var )

  #Make the global in another function
  #The following print will print 4
  initilize_global
  print( $var2 )
end

#A function that creates a global
def initilize_global
  $var2 = 4
end
```

Attributes:

Attributes have all of the same syntax requirements as variables but require a "@" at the start of the variable name. Unlike globals and variables, attributes can only be created inside of a class object. Attributes cannot be created anywhere else in the source code.



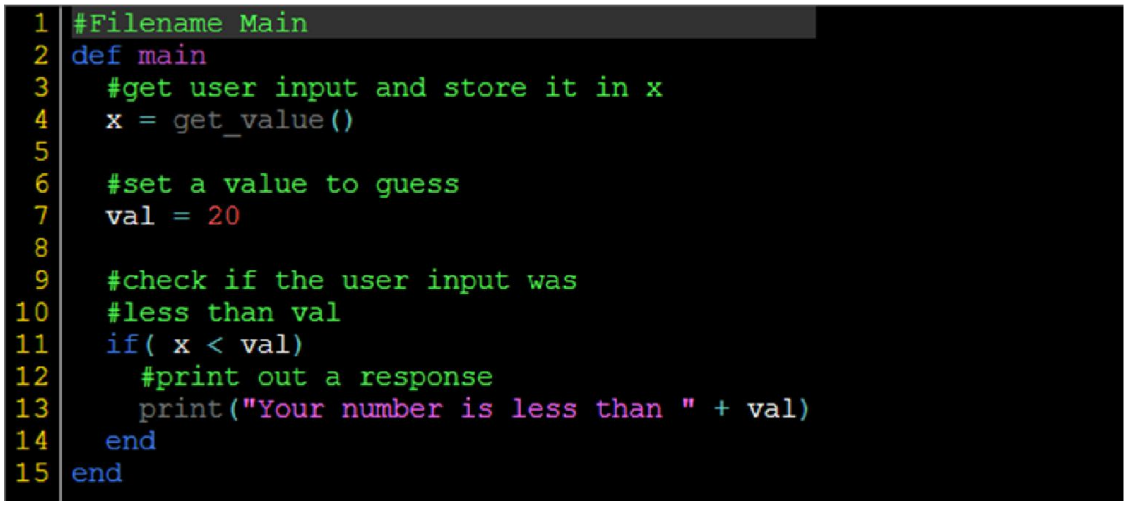
```
#Filename Main
def main
  #create an object and print
  #prints CLASSOBJ: obj { @x = 4, @y = 8 }
  print( new obj )

  #Attributes only allowed in classes
  @var = 5 #This will create an error
end

#A classtype object that contains attributes
class obj
  def initialize
    #Creates two Attributes, x and y
    @x = 4
    @y = 8
  end
end
```

Control Statements:

There are three different types of control statements that are used in Direct Abstraction. Control statements are used to control the flow of the program and control logic. Control statements fall into two simple categories, conditionals and loops. Conditionals execute a set of instructions if a certain condition is validated. If the condition isn't validated then the code will not be executed. Loops are used to rerun code multiple times. The number of times the code is run and the exit requirements for a loop are based on the type of loop used.

If Statement:

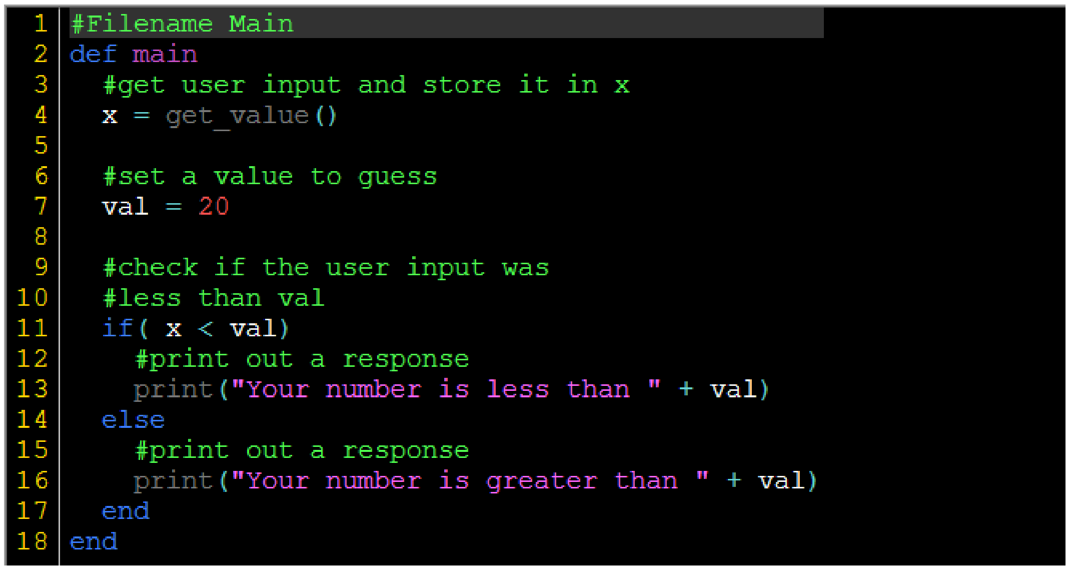
If statement is a conditional that executes if a statement is valid. A valid statement is any statement that evaluates as "true" when "convert(statement, "boolean")" is used on it. Nil statements are considered not valid and are not executed.



```
#Filename Main
def main
  #get user input and store it in x
  x = get_value()

  #set a value to guess
  val = 20

  #check if the user input was
  #less than val
  if( x < val)
    #print out a response
    print("Your number is less than " + val)
  end
end
```

Else Statement:

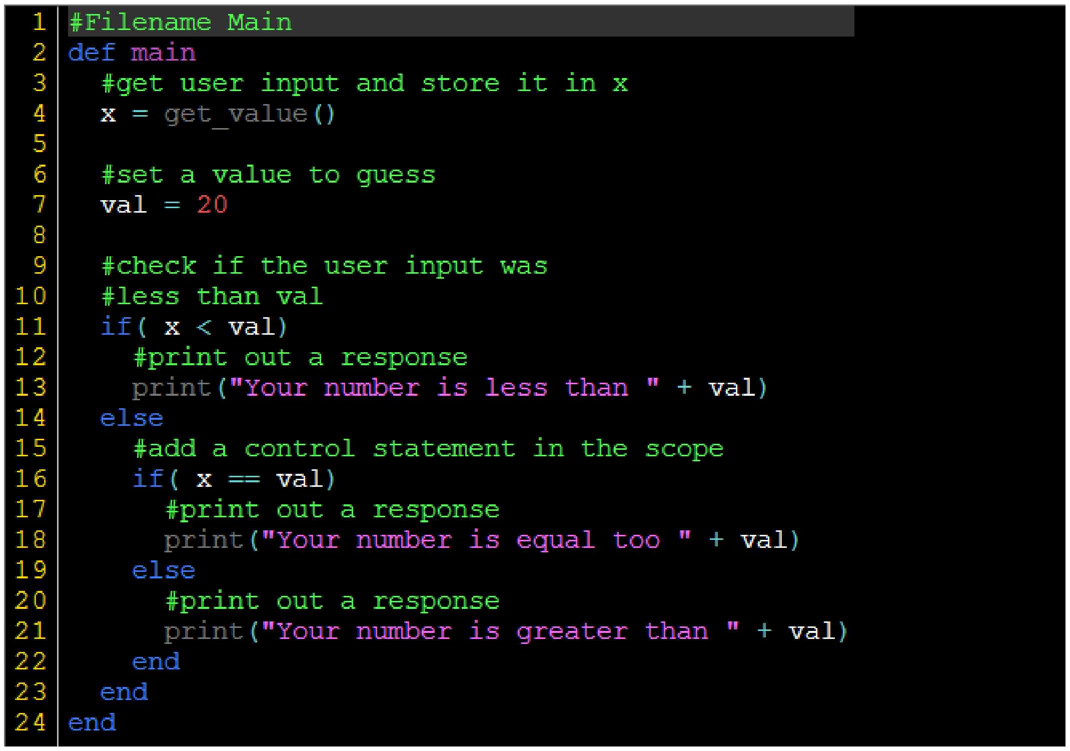
"Else" statements can be added to the end of an "if" statement. If the "if" statement is passed an invalid value then the else statement will be executed instead. "Else" statements can only be used in conjunction with "if" statements.



```
#Filename Main
def main
  #get user input and store it in x
  x = get_value()

  #set a value to guess
  val = 20

  #check if the user input was
  #less than val
  if( x < val)
    #print out a response
    print("Your number is less than " + val)
  else
    #print out a response
    print("Your number is greater than " + val)
  end
end
```

Scoping:

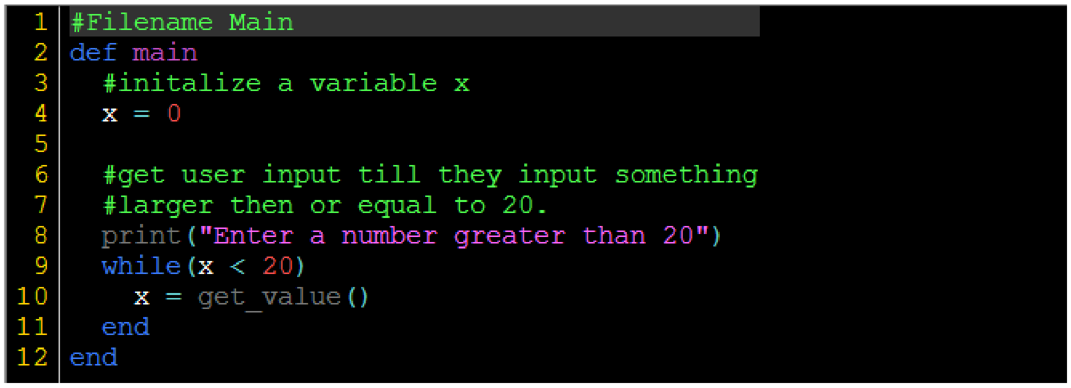
Every time a control statement is used it creates its own scope. Scope is defined as blocks of code to be executed inside of control statements. Unlike languages like C and C++ scope does not affect variable declarations.



```
#Filename Main
def main
  #get user input and store it in x
  x = get_value()

  #set a value to guess
  val = 20

  #check if the user input was
  #less than val
  if( x < val)
    #print out a response
    print("Your number is less than " + val)
  else
    #add a control statement in the scope
    if( x == val)
      #print out a response
      print("Your number is equal too " + val)
    else
      #print out a response
      print("Your number is greater than " + val)
    end
  end
end
```

While statements:

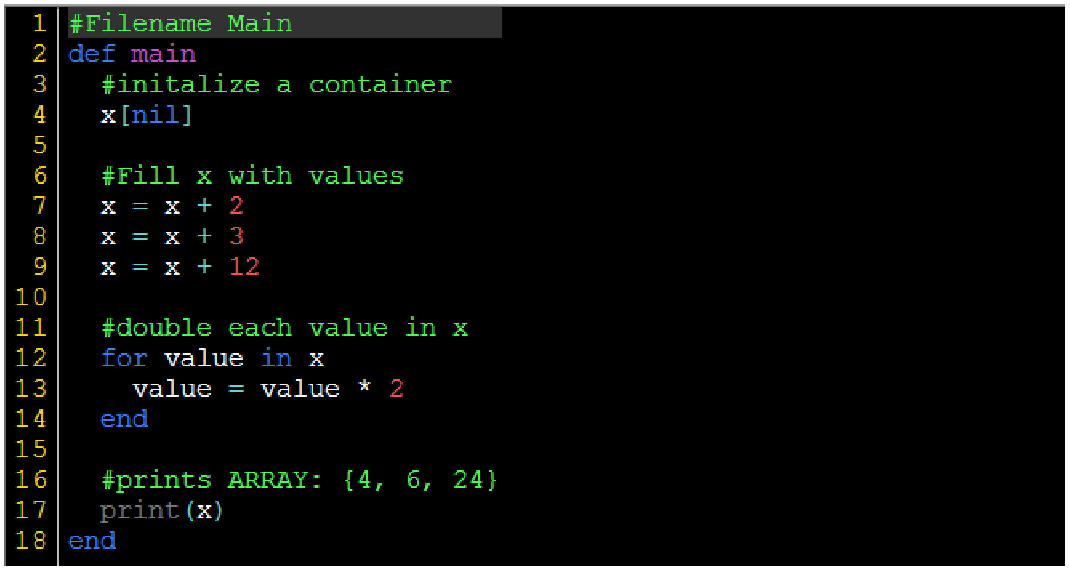
While statements are a loop that will continue to execute as long as the statement it qualifies is valid. While loops are good at controlling repeated code and setting up persistent programs.



```
#Filename Main
def main
  #initalize a variable x
  x = 0

  #get user input till they input something
  #larger then or equal to 20.
  print("Enter a number greater than 20")
  while(x < 20)
    x = get_value()
  end
end
```

For statements:

For loops are used to loop over all the values in a container. A variable is given to act as the element to be used as a reference to each element in the container. If an empty container or a symbol that is not a container then that means the input is considered not valid and the loop is not executed.



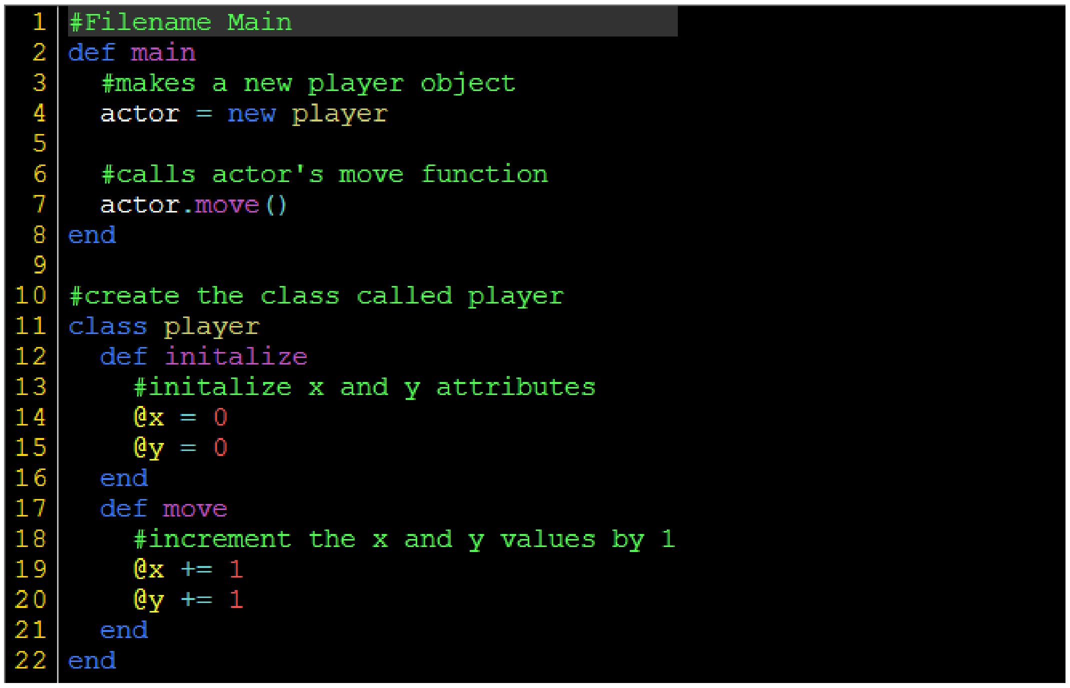
```
#Filename Main
def main
  #initalize a container
  x[nil]

  #Fill x with values
  x = x + 2
  x = x + 3
  x = x + 12

  #double each value in x
  for value in x
    value = value * 2
  end

  #prints ARRAY: {4, 6, 24}
  print(x)
end
```

Declarations:

Declarations are used to inform Direct Abstraction how to handle symbols as objects and how to call functions in memory. There are two type of declarations used in Direct Abstraction.

Class Declarations:

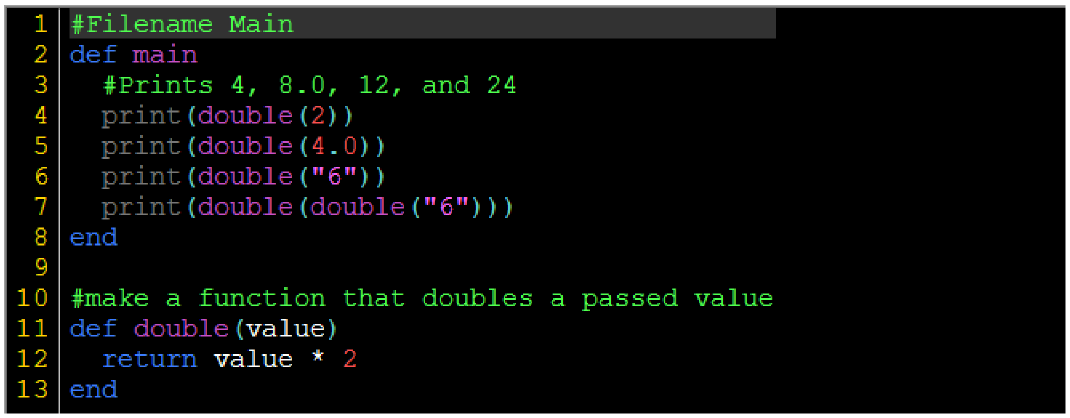
The first kind of declaration is class declarations. Class declarations tell Direct Abstraction what symbols should be considered objects instead of variables. Objects can have their own functions embedded in them and are able to have persistent values inside of them known as attributes.



```
#Filename Main
def main
  #makes a new player object
  actor = new player

  #calls actor's move function
  actor.move()
end

#create the class called player
class player
  def initalize
    #initalize x and y attributes
    @x = 0
    @y = 0
  end
  def move
    #increment the x and y values by 1
    @x += 1
    @y += 1
  end
end
```

Function Declarations:

Function declarations tell Direct abstraction that the following symbol is a function and maps to a segment of code that should be called. At the end of a function declaration should be the arguments that are to be passed to the function.



```
#Filename Main
def main
  #Prints 4, 8.0, 12, and 24
  print(double(2))
  print(double(4.0))
  print(double("6"))
  print(double(double("6")))
end

#make a function that doubles a passed value
def double(value)
  return value * 2
end
```

Operators:

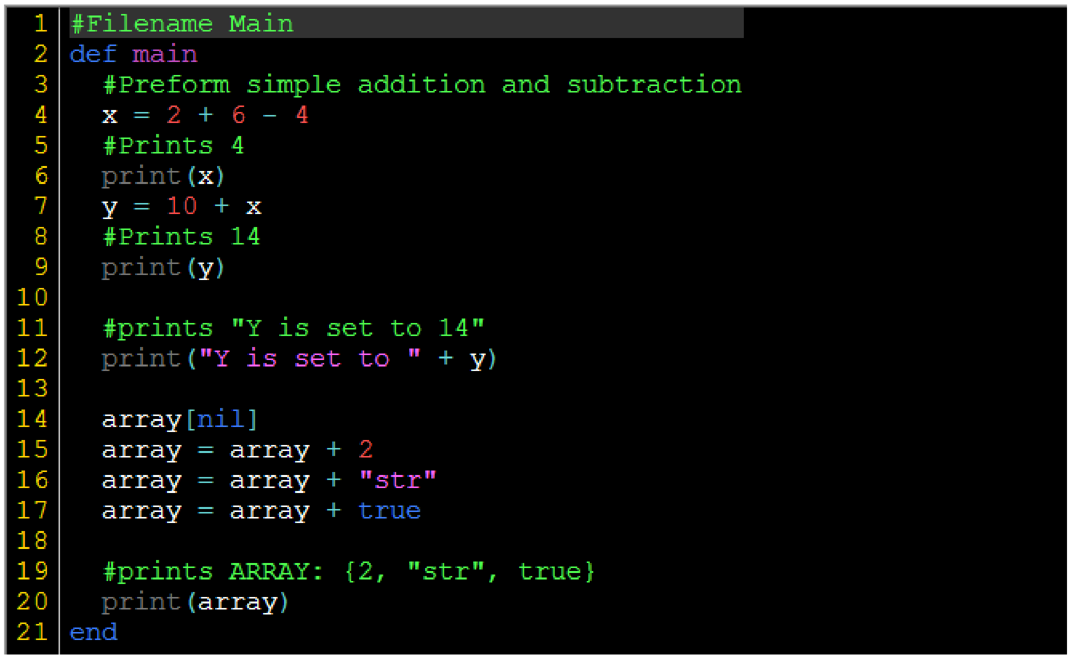
Direct Abstraction has support for a number of operators to preform math and other features. The order in which operators get executed is based on the priority of the operation. Operations with higher priorities are executed first followed by lower priority operations. If two operations with the same priority are on the same line then they will be executed from left to right.

List of all Direct Abstraction operation priorities

Symbol	Priority	Symbol	Priority
Object	11	Addition / Subtraction	5
Range Operator	10	Modulo	4
Array Bracket	9	Dot Operator	4
Function	9	Compare	3
Couple Operator	8	Greater/Less Than	3
Power	7	Assignment	2
Mul / Division	6	Keyword	1

Addition - Subtraction:

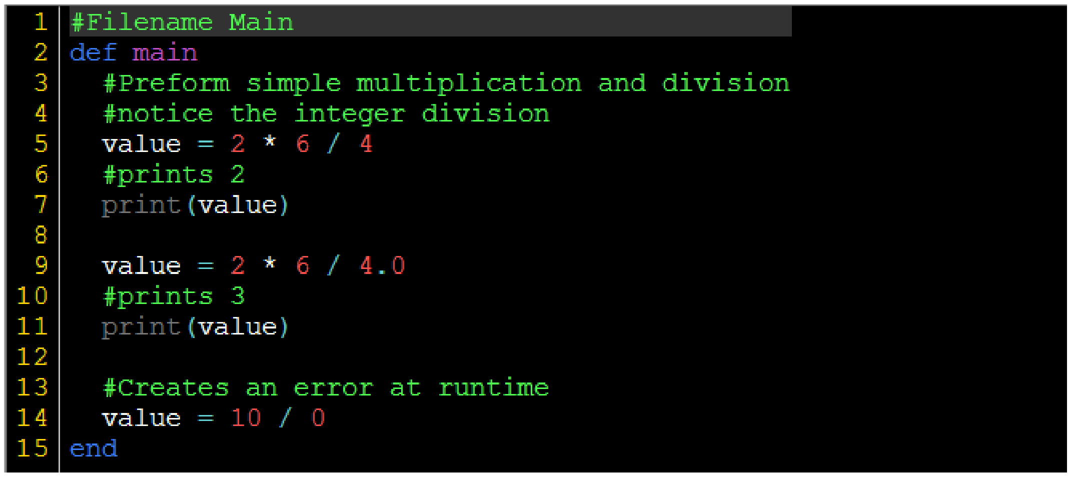
Addition and subtraction are two of the most basic arithmetic functions. Both have a priority of 5. It should be noted that addition is overloaded to have special functionality other than just arithmetic. When used on strings addition concatenates two strings together making a longer string. When used on containers addition adds the element to the container.



```
#Filename Main
def main
  #Preform simple addition and subtraction
  x = 2 + 6 - 4
  #Prints 4
  print(x)
  y = 10 + x
  #Prints 14
  print(y)

  #prints "Y is set to 14"
  print("Y is set to " + y)

  array[nil]
  array = array + 2
  array = array + "str"
  array = array + true

  #prints ARRAY: {2, "str", true}
  print(array)
end
```

Multiplication – Division:

Multiplication and division preform the same functionality as they do in other languages. Both have a priority value of 6. With division if both values are numbers then integer division is preformed and the returned value is the floor of the division.



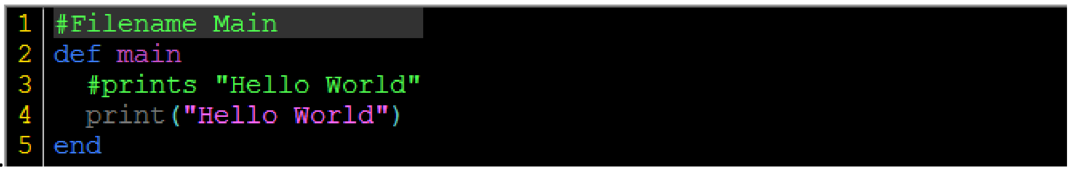
```
#Filename Main
def main
  #Preform simple multiplication and division
  #notice the integer division
  value = 2 * 6 / 4
  #prints 2
  print(value)

  value = 2 * 6 / 4.0
  #prints 3
  print(value)

  #Creates an error at runtime
  value = 10 / 0
end
```

Built in functions:

Direct Abstraction has a number of built in functionality. Since though Direct Abstraction it is impossible to gain access to low level functionality it has been supplied for the end user.

print:

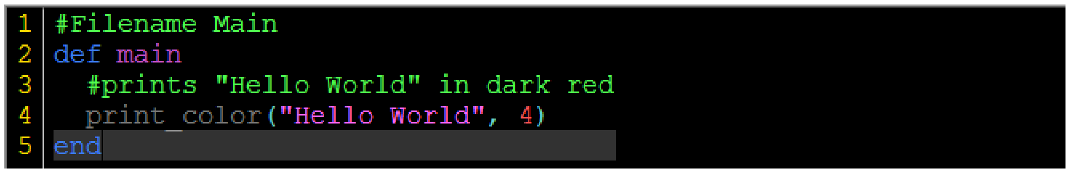
Print takes a single argument and prints it out onto the screen in white.



```
#Filename Main
def main
  #prints "Hello World"
  print("Hello World")
end
```

print_color:

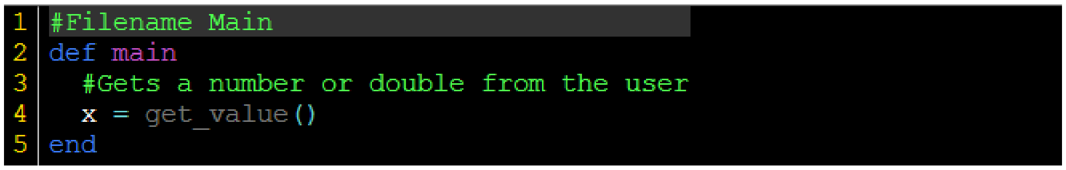
Print_color takes two arguments. The first argument is the symbol to be printed. The second dictates the color.



```
#Filename Main
def main
  #prints "Hello World" in dark red
  print_color("Hello World", 4)
end
```

get_value:

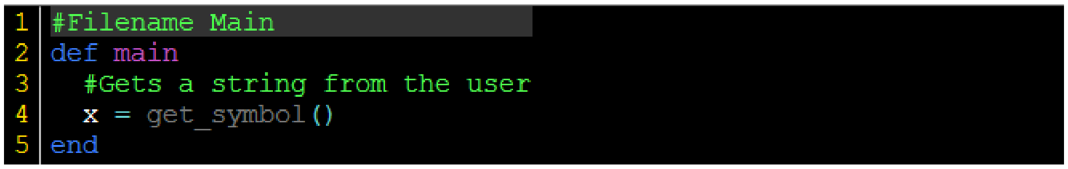
Get_value takes no arguments. Get_value gets a number or a double from stdin.



```
#Filename Main
def main
  #Gets a number or double from the user
  x = get_value()
end
```

get_symbol:

Get_symbol takes no arguments. Get_symbol a string from stdin.



```
#Filename Main
def main
  #Gets a string from the user
  x = get_symbol()
end
```