# Master Class: Go Places with SystemWorkers!

Presented by: Keisuke Miyako, Technical Account Manager – 4D Japan.

## Introduction

As of v20, there are **1334** commands in the 4D lexicon.[1] That's 11 new commands compared to v19, or 53 new commands compared to v18. If you have been with 4D for many years, you may recall a time when the number of new commands added to the language with a major release could exceed 100.

That is not to say that 4D is slowing down on innovation or that v20 is a mild upgrade. To the contrary, there are literally hundreds of powerful new features to explore in this landmark release. So why are there so few commands added to the language? Because there has been a major **paradigm shift**.

## Paradigm shift

The classic 4D language is often described as **procedural programming**. An application is basically a sequence of simple, imperative instructions. The code is intuitive and relatively easy to follow. The number of commands is directly correlated with the complete feature set as well as productivity.

In general, procedural programming is a great time saver. It allows you to quickly change or adapt your code locally without the risk of breaking existing parts of the application. However, such easiness could turn out to be a mixed blessing. If you are not careful, you may find yourself saddled with technical debt.

One way to prevent such negative consequences is to consciously practice **functional programming**. Instead of referencing shared objects implicitly, you design and use functions that require no context other than their arguments and return values. This would be quite a departure from early 4D programming style where the execution context (current table, current record, current selection, variables, sets, etc.) did not need to be explicitly defined for the current process or dialog.

In general, functional programming takes more time at the beginning since you need to first create all the necessary building blocks according to specification before you can even have a working prototype. Over time, the consistent, modular design will start to pay off. Your code would be easier to maintain.

**Object-oriented programming** is like a mash-up of procedural and functional. It follows the imperative model where a task is accomplished procedurally. The difference is that the execution context is stored in an object instance instead of being shared globally. It also follows the declarative model in that functions require no context other than their arguments and return values. The difference is that the information stored locally in the instance (**This**) is accessible from any member function of that object.

Native object-oriented coding in 4D is a major paradigm shift. Moving forward, you can expect more and more features to be added as new **classes**, not as new commands. The few new commands added in v20 are either related to the user-interface, which is procedural, or something global like error handling.[2]

---

[1] See "Count number of valid commands" at the end of this document.

[2] See "Commands added in v20" at the end of this document.

## System Workers

v20 features a new native class, **4D.SystemWorker.** Basically, it is an object-oriented implementation of the existing **LAUNCH EXTERNAL PROCESS** command. The new class can basically do everything that was possible with the old command.

Example using LAUNCH EXTERNAL PROCESS:

```
LAUNCH EXTERNAL PROCESS($command; $stdIn; $stdOut; $stdErr)
```

Same task, except using 4D.SystemWorker:

```
$stdOut:=4D.SystemWorker.new($command).wait().response
```

Of course, we wouldn't be investing in this new feature unless it had something more to offer.

The LAUNCH EXTERNAL PROCESS command is executed atomically and synchronously. This would not be a problem if the external process completes its job in an instant. Problem arises when the task needs to access the network or process many files, which may take a while. The command or shell script in charge of the task might print progress or other incremental information to the standard error stream but LAUNCH EXTERNAL PROCESS can only receive the complete response right at the end at which point the task is already finished.

In classic procedural 4D programming, the way to handle time-consuming tasks without blocking the main process is to run the task incrementally in another process and coordinate the two processes using CALL WORKER, CALL FORM, POST OUTSIDE CALL, or some other kind of inter-process communication.[3]

The 4D.SystemWorker class has a similar system to execute tasks asynchronously. The difference is that you don't need to start a new background process yourself or regularly update the original process from the process that is running the task incrementally in a loop. That part is taken care of. Instead, you need to have an "options" object (we will call it the **controller** instance in this session) to process messages posted by the asynchronous System Worker. This object must exist in a running execution context to receive such messages. In other words, you need to use System Workers in a dialog or a worker to process asynchronous calls.

Although it has "worker" in its name, the 4D.SystemWorker class does not create a new worker process[4]. In fact, the naming has more to do with the fact that the API is inspired by JavaScript **Web Workers.** The class has more in common with the **4D.HTTPRequest** class[5] than, say, CALL WORKER.

---

[3] Polling with inter-process variables is not recommended. Same with GET PROCESS VARIABLE or similar commands.

[4] You can confirm that no new processes are created in the Runtime Explorer.

[5] See HTTP classes: A New Way to Perform Asynchronous HTTP Requests in the official blog.

## Object-oriented programming in 4D

Since the introduction of classes, many developers have taken to rewrite their application using classes. However, **simply using classes and dot-notation does not necessarily make the code object-oriented**. The 4D.SystemWorker example on the previous page replaced a command with a class, but other than the syntactical change, there was no substantial difference in the code or its underlying logic.

Some developers might object (pun intended) that OOP in 4D does not generate strongly typed code and is therefore not real. That might be true if absolute speed and memory efficiency in compiled mode was our only metric. But I would like to suggest an alternative view.

4D has incorporated OOP in recent feature releases in a way that enhances productivity and rewards developers for following good coding practice. For example, declaring properties does not help the compiler generate faster code but it does make **code completion** more intelligent. Defining a **getter function** does not protect a property at the runtime level but it does make the editor display better **help tips**. Similarly, **prefixing a function or property name with an underscore** does not hide them from the debugger or prevent you from using them but it does make the editor filter its type-ahead candidates list for objects that are declared as instances of a specific class.

The same can be said of **documentation**. Ideally you will start from top-level abstract classes and work your way down the class hierarchy so that you only describe shared features once. **Comment tags** can be used to highlight member functions that are private, virtual, or special in some other way.

This session will show how to practice object-oriented programming – **abstraction, encapsulation, polymorphism, and inheritance** – within the current constraints of 4D. If you are not yet convinced that classes are useful, perhaps disappointed after spending some time playing around with them, my hope is that this session will present them in a new light and encourage you to give them another shot.

## The _CLI class

_CLI is the **base class** to implement console program proxies. The class serves two purposes:

1. Create an abstraction layer on top of 4D.SystemWorker that executes either a system command or an executable program installed in the project/component's /RESOURCES/ folder.

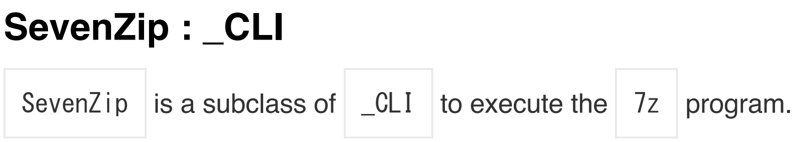
2. Present a platform neutral API that works on Windows or macOS.

A base class is a generic, abstract class. _CLI is not meant to be used directly. Rather, it should be extended to implement a specific console program proxy suitable for each CLI. The **class name is prefixed with an underscore** to exclude it from code completion and the component namespace[6].

Example (the SevenZip class, explained later):

```
Class extends _CLI

Class constructor($controller : 4D.Class)

 Super("7z"; $controller)
```

For clarity, the class inheritance is also indicated using a colon symbol in the **class documentation[7]**:



**SevenZip : _CLI**

`SevenZip` is a subclass of `_CLI` to execute the `7z` program.

The class or its subclass can be instantiated in 2 ways:

1. Pass the name of the program file that is installed in /RESOURCES/bin/{platform}. The ".exe" extension on Windows should be omitted.

2. Pass the name of the system command (searchable according to the $PATH environment variable).

---

[6] See [Access your component classes from your host project](#) in the official documentation.
[7] See [Project Databases: Method documentation is back](#) in the official documentation.

In client-server mode, on macOS, the UNIX execution bit is lost during resources synchronisation. The class restores this status if the executable file is located inside /RESOURCES/.

```
This._chmod()
```

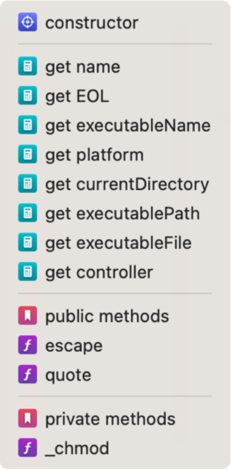
This member function is also prefixed with an underscore as it is intended to be used internally.

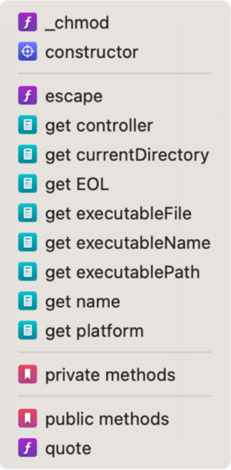
This function needs to be **synchronous** so here we have no reason to use 4D.SystemWorker:

```
//MARK:-private methods

Function _chmod()

 If (Is macOS)
  If (Application type=4D Remote mode)
   SET ENVIRONMENT VARIABLE("_4D_OPTION_CURRENT_DIRECTORY";
This.currentDirectory.platformPath)
   SET ENVIRONMENT VARIABLE("_4D_OPTION_BLOCKING_EXTERNAL_PROCESS"; "true")
   LAUNCH EXTERNAL PROCESS("chmod +x "+This.executableName)
  End if
 End if
```

The old LAUNCH EXTERNAL PROCESS is good enough for the job.

Notice the “//MARK-” comment tag[8]. This inserts a line in the method editor’s popup menu.





Within the context of a class source file, function definitions are “hoisted” to the top, meaning functions that are defined somewhere can be used anywhere in the file. Although technically you can write your functions in any order, you might to sort them in the source for searchability.

By default, tagged elements are listed in order of appearance. You can force alphabetical order by holding down the Alt (Windows) or option (Mac) key. Notice the sorting breaks any arbitrary groups created using comment tags. For this reason, I would suggest that functions, properties, and comment tags are logically ordered at the source level.

---

[8] See [Tag your comments, a feature by developers for developers]() in the official blog.

The class defines 2 public functions, quote() and escape(). These are made public as they should be useful in any subclasses extended from this base class. It is also significant that these **basic functions are implemented at the top of the class hierarchy**. If they need to be updated in the future, you don't have to rewrite every subclass that uses them.

The escape rules are slightly different between Windows and Mac:

```
Function escape($in : Text)->$out : Text

 $out:=$in
 var $i; $len : Integer
 Case of
  : (Is Windows)
   /*
    argument escape for cmd.exe; other programs may be incompatible
   */
   $shoudQuote:=False
   $metacharacters:="&|<>()%^\" "
   $len:=Length($metacharacters)
   For ($i; 1; $len)
    $metacharacter:=Substring($metacharacters; $i; 1)
    $shoudQuote:=$shoudQuote | (Position($metacharacter; $out; *)#0)
    If ($shoudQuote)
     $i:=$len
    End if
   End for
   If ($shoudQuote)
    If (Substring($out; Length($out))="\\")
     $out:="\""+$out+"\\\""
    Else
     $out:="\""+$out+"\""
    End if
   End if
  : (Is macOS)
   /*
    argument escape for bash or zsh; other programs may be incompatible
   */
   $metacharacters:="\\!\"#$%&'()=~|<>?;*`[] "
   For ($i; 1; Length($metacharacters))
    $metacharacter:=Substring($metacharacters; $i; 1)
    $out:=Replace string($out; $metacharacter; "\\"+$metacharacter; *)
   End for
 End case

Function quote($in : Text)->$out : Text

 $out:="\""+$in+"\""
```

These functions are documented for future reference:

## .escape()

**.escape()**($in : Text) : Text

Add double quotes on each sides of a string that contains metacharacters on Windows, except when the last character is a backslash in which case the backslash is escaped with a backslash before the whole string is quoted. Escape metacharacters on Mac. Not all CLI agent respect this type of escaping.

## .quote()

**.quote**($in : Text) : Text

Simply add double quotes on each sides of a string. Some CLI agent prefer this type of escaping.

The class keeps some context information as properties. These are supposed to be immutable, so their values are stored in properties prefixed with an underscore:

```
This._name:=OB Class(This).name

 Case of
  : (Is macOS)
   This._platform:="macOS"
   This._executableName:=$executableName
   This._EOL:="\n"
  : (Is Windows)
   This._platform:="Windows"
   This._executableName:=$executableName+".exe"
   This._EOL:="\r\n"
 End case
```

These **read-only properties** are accessible via a **getter function**[9]:

```
Function get name()->$name : Text

 $name:=This._name.
```

These properties are listed in the class documentation:

---

[9] See Function get in the documentation.

## .new()

**.new**($executableName : Text) : cs._CLI

Pass the name of the CLI agent. It should be placed in the following location:

`/RESOURCES/bin/{macOS|Windows}/`

The constructor defines the following properties:

| Property | Type | Description |
|---|---|---|
| controller | cs._CLI_Controller | read-only |
| currentDirectory | 4D.Folder | read-only |
| EOL | Text | read-only |
| executableFile | 4D.File | read-only |
| executableName | Text | read-only |
| executablePath | Text | read-only |
| name | Text | read-only |
| platform | Text | read-only |

The class is intended to be subclassed. `.name` returns the value `OB Class(This).name` which is the name of the extended class, not this class.

# The _CLI_Controller class

_CLI_Controller is the **base class** to implement controller instances for _CLI subclasses. The class has several purposes:

1. Create a 4D.SystemWorker instance.

2. Keep a queue of commands to execute in sequence with the new 4D.SystemWorker instance.

3. Serve as the “options” object that is passed to the 4D.SystemWorker instance.

4. Terminate the 4D.SystemWorker instance when there are no more commands to execute.

5. Force terminate the 4D.SystemWorker even if there are more commands waiting in queue.

A controller in this context is the “options” object you define when you create a 4D.SystemWorker instance. A controller has several properties that define how 4D interacts with the external program as well as callback functions that are invoke at critical junctures.

In addition to the standard properties expected of the “options” objects, _CLI_Controller defines additional read-only properties which are listed in the class documentation:

## .new()

**.new**($CLI : cs._CLI) : cs._CLI_Controller

The constructor should not be called directly. It is invoked when a `_CLI` is instantiated.

The constructor defines the following properties:

| Property | Type | Description |
|---|---|---|
| commands | Collection | read-only |
| complete | Boolean | read-only |
| instance | cs._CLI | read-only |
| worker | 4D.SystemWorker | read-only |

The controller can receive up to 5 different call back events: onData, onDataError, onError, onResponse, onTerminate. The call signature is basically the same for each event, so the class simply uses the same private function for all events:

```
//MARK:-private methods

Function _onEvent($worker : 4D.SystemWorker; $params : Object)

 Case of
  : ($params.type="data") && ($worker.dataType="text")

  : ($params.type="data") && ($worker.dataType="blob")

  : ($params.type="error")

  : ($params.type="termination")

  : ($params.type="response")

 End case
```

The class constructor performs an introspection of itself (which should be a subclass of _CLI_Controller):

```
Class constructor($CLI : cs._CLI)

 //use default event handler if not defined in subclass definition
 For each ($event; New collection("onData"; "onDataError"; "onError";
"onResponse"; "onTerminate"))
  If (Not(OB Instance of(This[$event]; 4D.Function)))
   This[$event]:=This._onEvent
  End if
 End for each
```

This ensures that any functions defined at the subclass level takes precedence. Of course, the same can be accomplished by explicitly defining the 5 functions. This is just to illustrate how **a class can redefine itself at runtime**. The same technique is used later.

The constructor expects an **instance** (should be a subclass) of _CLI. This binding is established when you instantiate a _CLI subclass, whose constructor expects a **class** (not an instance) of _CLI_Controller.

```
Class constructor($executableName : Text; $controller : 4D.Class)

If ($controller=Null)
  This._controller:=cs._CLI_Controller.new(This)  //default controller
 Else
  This._controller:=$controller.new(This)  //custom controller
End if
```

The class binding is to assert that any subclass of _CLI should work with any subclass of _CLI_Controller.

By default, the controller creates a new 4D.SystemWorker instance for each command. This is because a typical CLI program is designed to execute a line of command at a time.

```
This._worker:=4D.SystemWorker.new(This._commands.shift(); This)
```

Note that the worker instance is stored in a read-only property named “worker”. The instance is set to Null on termination. The controller check this property every time it is asked to execute a new command. If the worker is not Null, meaning it is still running the last command, the new command is simply added to the end of the queue.

In addition to the 5 standard call back events, the controller implements a private event handler named _onExecute. The event is invoked every time a command from the queue returns a complete response. Internally it is hooked to the standard onResponse event handler. The function intercepts the call, shifts the queue by one, and passes control to whichever function (either the default _onEvent function or the function that is defined at the subclass level) that was originally designated to receive the event.

```
Function _onExecute($worker : 4D.SystemWorker; $params : Object)

 If (This._commands.length=0)
  This._abort()
 Else
  This._execute()
 End if

 If (OB Instance of(This._onResponse; 4D.Function))
  This._onResponse.call(This; $worker; $params)
 End if
```

The original onResponse event handler is temporarily stored in a private property, prefixed with an underscore. The class redefines itself at runtime, once before a command is about to be executed and again when it has completed.

```
Function execute($command : Variant)

 var $commands : Collection

 Case of
  : (Value type($command)=Is text)
   $commands:=New collection($command)
  : (Value type($command)=Is collection)
   $commands:=$command
 End case

 If ($commands#Null) && ($commands.length#0)
  This._commands.combine($commands)
  If (This._worker=Null)
   This._onResponse:=This.onResponse
   This.onResponse:=This._onExecute
   This._onTerminate:=This.onTerminate
   This.onTerminate:=This._onComplete
   This._execute()
  End if
 End if

Function _terminate()

 This.onResponse:=This._onResponse
 This.onTerminate:=This._onTerminate
 This._worker:=Null
```

Notice the execute() method expects a **variant argument**. The argument can be an object or a collection of objects. Internally a single object is converted to a collection with just 1 element. It's not a big deal, but using a variant allows for a flexible syntax.

The _CLI and _CLI_Controller classes provide the boilerplate on which to base our 4D.SystemWorker implementation. Now let's look at the SevenZip example.

## The SevenZip class

SevenZip is a **subclass** of _CLI to execute the [7-Zip] program. it can be used in a dialog or a worker with no user interface. Pass a subclass of _CLI_Controller implement callbacks or to customise the behaviour.

Example use in a worker with the default controller:

```
$zipper:=cs.SevenZip.new()
$zipper.add($z; New collection($a; $b; $c))
$zipper.extract($d; $z)
```

Example use in a worker with custom controller:

```
$zipper:=cs.SevenZip.new(cs._Worker_Controller)
$zipper.add($z; $a).add($z; $b).add($z; $c).extract($d; $z)
```

Example use in a dialog with customer controller:

```
$form:=cs.SevenZip.new(cs._SevenZip_Form_Controller)
$window:=Open form window("TEST")
DIALOG("TEST"; $form; *)
```

out of the many CLI arguments available in 7z, only the basic "a (add)" and "x (extract)" commands are implemented:

### .add()

**.add**($destination : 4D.File; $sources : Variant)->$this : cs.SevenZip

Adds a collection of `4D.File` or `4D.Folder` objects to an archive. Alternatively archive a single object.

Chained calls are queued as sequential tasks.

### .extract()

**.extract**($destination : 4D.Folder; $sources : Variant)->$this : cs.SevenZip

Extracts all files from an archive.

Chained calls are queued as sequential tasks.

Note how the functions accept a **4D.File** of **4D.Folder** object instead of paths. It is better to be specific with regards to the class of object to pass as an argument, while being platform agnostic when possible.

A file or folder object may reference a file system such as /RESOURCES/. You can't pass such paths directly to an external program. They need to be cast as objects based on native platform paths:

```
Case of
   : (OB Instance of($source; 4D.File))
    $source:=File($source.platformPath; fk platform path)
   : (OB Instance of($source; 4D.Folder))
    $source:=Folder($source.platformPath; fk platform path)
End case
```

White spaces, quotes, slashes, and other characters may break the command line if used in raw format. The safe way to pass such values depends on the platform as well as the console program being called.

The 7-zip program does not seem to work well with standard escape sequences, so the class uses double quotes to safely pass path names:

```
$command:=This.escape(This.executablePath)
  $command:=$command+" a -snl -bso1 -bsp2 -y
"+This.quote($destination.path)+" "+This.quote($source.path)
/*
   -snl : store symbolic links as links (tar)
   -bso1: redirect(b) stream(s) output  (p) stdout(1)
   -bsp2: redirect(b) stream(s) progress(p) stderr(2)
 */
```

Notice how the functions return **This** as their return value. This is to allow chained calls:

```
Function add($destination : 4D.File; $source : Variant)->$this : cs.SevenZip

 $this:=This
```

The functions also accept a variant as their 2nd argument, which allows passing a single object or a collection of objects. Whether a collection is passed in a single call or over multiple chained calls, they are all stacked as a queue of command lines and executed in sequence.

By default, the class create an instance of _CLI_Controller as its controller. The base class focuses on managing the command queue and does nothing during the call back events. For that, you need to extend the class and implement custom event handlers.

## The _Worker_Controller class

_Worker_Controller is a **template** subclass of _CLI_Controller. It contains 5 placeholders, or **virtual functions**, where you can implement your custom event handers.

```
//MARK:-polymorphism

Function onDataError($worker : 4D.SystemWorker; $params : Object)

Function onData($worker : 4D.SystemWorker; $params : Object)

Function onResponse($worker : 4D.SystemWorker; $params : Object)

Function onTerminate($worker : 4D.SystemWorker; $params : Object)

Function onError($worker : 4D.SystemWorker; $params : Object)
```

The function are already implemented in the parent _CLI_Controller class. Subclasses possess these function by means of **inheritance**, but they can also implement their own version, effectively over-riding the default inherited behaviour. This kind of inheritance is called **polymorphism**.

A **worker** is typically a process that runs driverless in the background without any human interaction. The controller does not need to do much, other than perhaps log events or send out notifications.

The controller for a **dialog** is more complex. At the very least, you will need way to start or stop the controller. You will also likely need a progress indicator. or that, you need a _CLI_Controller derivative class that takes care of the user interface aspect.

## The _Form_Controller class

_Form_Controller is a subclass of _CLI_Controller to execute a _CLI instance in a form. In addition to basic controller features inherited from its superclass, an instance of this class does the following:

1. Capture standard output and standard error streams.

2. Bind to start and stop buttons on the form.

You can pass this class or a derivative of this class to the constructor of a _CLI subclass. An instance is internally created and attached to the _CLI as its “controller” property.

Example:

```
$form:=cs.SevenZip.new(cs._Form_Controller)

 $window:=Open form window("TEST")
 DIALOG("TEST"; $form; *)
```

In the example above, **Form** will be an instance of SevenZip and Form.controller will be an instance of _Form_Controller.

Console applications emit data through 2 streams: the standard output and standard error. The **output stream** is typically used to display informative text on the Command Prompt or Terminal screen. The **error stream** is supposed to communicate warnings or errors, but many applications use this channel to display supplementary information that are not necessarily errors. For example, the 7-zip program uses its error stream to project an animated progress bar to the command line interface.

An instance of 4D.SystemWorker can be configured to handle streams as **Text** or **4D.Blob objects**. Streams are sent in chunks of arbitrary length, at seemingly random intervals. The controller receives these messages in its **onData** and **onDataError** call back events.

It is important to understand that what you receive in these events are short clips of an ongoing data stream. It could be cut off right in the middle of some formatted string. To parse the information correctly you need to store partial streams in a buffer and concatenate the streams that follow until you get a complete message.

If the stream is Text you can simply concatenate the strings using the "+=" operator[10]. If the stream is 4D.Blob, you might have to use a classic Blob or create a new copy to grow the buffer because 4D.Blob objects are immutable[11].

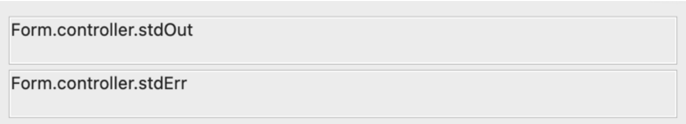
The _Form_Controller class has a pair of private properties, prefixed by an underscore, for each stream. One is to store the latest stream, and another is to use as a buffer to store aggregate data:

```
Class extends _CLI_Controller

property _stdErr; _stdOut : Text
property _stdErrBuffer; _stdOutBuffer : Text
```

Notice these private properties are exposed as read-only properties:

```
Function get stdOut()->$stdOut : Text

 $stdOut:=This._stdOut

Function get stdErr()->$stdErr : Text
 $stdErr:=This._stdErr
```

As explained earlier, when the _CLI subclass instance is used as the Form object, its controller becomes available as Form.controller. This means you can directly use this expression as the data source of a form object, which will be updates automatically without any code:



Form.controller.stdOut

Form.controller.stdErr

When you execute a 4D.SystemWorker in the context of a form, you would most likely bind it to button objects that start or terminate the external process.

## .bind()

**.bind**($options : Object)

Binds the supplied form object names to a specific functionality.

The following object names are supported:

---

[10] See [4D language: The improvements you requested are here]() on the official blog.
[11] See [Create a BLOB attribute in an object]() on the official blog.

| Property | Type | Description |
|---|---|---|
| startButton | Text | |
| stopButton | Text | |

The enabled/disabled status of each button object is toggled automatically.

The class implements several polymorphism function to over-ride the default behaviour of its superclass, _CLI_Controller. **This** and **Form** are references to the proxy and controller objects respectively:

```
Function onResponse($worker : 4D.SystemWorker; $params : Object)

 //Form is an instance of _CLI
 //This is an instance of _Form_Controller

 If (Form#Null)

  This._onResponseCount+=1

  If (This.complete)
   //end of the queue
  End if

 End if
```

Notice the Form object is compared to **Null**[12]. This is because the call back function could be invoked after the form has been closed. Dialog closure does not necessarily mean that the worker process has expired. The controller object is tied to the execution context of the worker process, not the form, which means it may survive the Form object.

---

[12] See [Comparison with an Undefined Value]() on the official blog.

## The _SevenZip_Form_Controller class

_SevenZip_Form_Controller is a subclass of _Form_Controller to execute the 7-zip program in a form.

As mentioned earlier, the 7-zip program displays an animated ASCII progress bar using the error stream. This class over-rides the **onDataError** function of its superclass to extract percentage information from the formatted error stream:

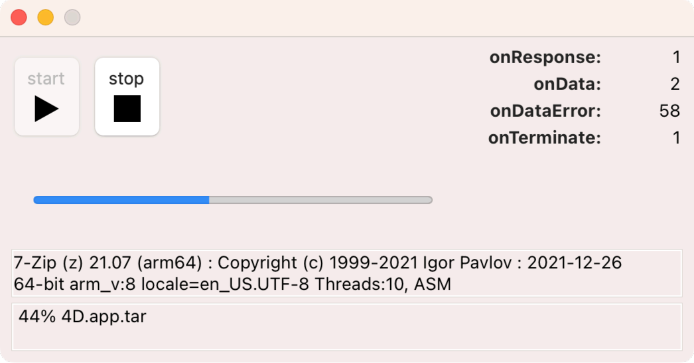
```
Function onDataError($worker : 4D.SystemWorker; $params : Object)

 $data:=This._stdErrBuffer

  ARRAY LONGINT($pos; 0)
  ARRAY LONGINT($len; 0)

  /*
   on Windows this stream is separated by CR, not CRLF.
   CR does not count as end of line in regex
   so don't use "s" metacharacter here
  */

  Case of
   : (Match regex \
   ("\\s*(\\d+)%[\\u0020]+(\\d+)[\\u0020]+(\\S+)[\\u0020]+(.+)"; \
   $data; 1; $pos; $len))
    $progress:=Num(Substring($data; $pos{1}; $len{1}))
    $size:=Num(Substring($data; $pos{2}; $len{2}))
    $flag:=Substring($data; $pos{3}; $len{3})
    $info:=Substring($data; $pos{4}; $len{4})
    This._stdErr:=String($progress; "^^0")+"%"+" "+$info
    This._stdErrBuffer:=""
    This._progress:=$progress

   : (Match regex \
   ("\\s*(\\d+)%[\\u0020]+(\\S+)[\\u0020]+(.+)"; \
   $data; 1; $pos; $len))
    $progress:=Num(Substring($data; $pos{1}; $len{1}))
    $flag:=Substring($data; $pos{2}; $len{2})
    $info:=Substring($data; $pos{3}; $len{3})
    This._stdErr:=String($progress; "^^0")+"%"+" "+$info
    This._stdErrBuffer:=""
    This._progress:=$progress
  End case
```

Notice the buffer is cleared when a viable piece of string has been detected. There is no need to keep the entire error stream (or output stream) from start to finish because that information will be available in the onResoponse event.

The progress is converted to an integer percentage value and stored in a private property prefixed with an underscore. The value is exposes with a getter function as a read-only property:

```
Function get progress()->$progress : Integer
$progress:=This._progress
```

The expression Form.controller.progress as a data source for a thermometer object. Its value will be updated automatically without any code.



onResponse: 1
onData: 2
onDataError: 58
onTerminate: 1

```
7-Zip (z) 21.07 (arm64) : Copyright (c) 1999-2021 Igor Pavlov : 2021-12-26
64-bit arm_v:8 locale=en_US.UTF-8 Threads:10, ASM
```

```
44% 4D.app.tar
```

In summary:

- Define what is common to all use cases as an abstract proxy class: **_CLI**
- Implement the specific command line interface in a proxy subclass: **SevenZip**
- Define a controller class separate from the proxy class: **_CLI_Controller**
- Implement a controller for the worker context with no user interface: **_Worker_Controller**
- Implement a controller for the worker/process context with a form: **_Form_Controller**
- Implement a controller class for a specific proxy subclass: **_SevenZip_Form_Controller**

Object-oriented programming – **abstraction, encapsulation, polymorphism, and inheritance** –helps you to design classes that are easier to repurpose. For example, to support a tool other than the 7-zip program you just need to replace the 2 “SevenZip” classes with another pair of subclasses with command line interface implementation specific to that tool. The same calling conversion will work for any tool or command, with or without the user interface. Improvements applied to base classes will benefit all extended classes. By making good use of markdown documentation, comment tags, and property declaration, your code will be modular, self-explanatory, and easier to maintain.

## The _Marked class

The preview pane of the Explorer is a web area that uses an open-source JavaScript library to render markdown content as HTML. The _Marked class is a utility that invokes the same library in an off-screen web area[13] to generate static HTML documentation for classes, methods, or forms:

```
var $marked : cs._Marked

$marked:=cs._Marked.new()
$path:=METHOD Get path(Path class; "_CLI")
$htmlFile:=$marked.generate($path)
OPEN URL($htmlFile.platformPath)
```

The WA Run offscreen area command accepts a “parameters” object which may implement an **onEvent** callback function to be called asynchronously from JavaScript. The object is effectively a controller. The _Marked class instance is fully compatible with the “parameters” object which mean it can submit itself:

```
Function parse($mdFile : 4D.File)->$htmlFile : 4D.File

 This._md:=$mdFile.getText()

 $htmlFile:=WA Run offscreen area(This)
```

Not to be too paranoid, but the onEvent function is supposed to be private to the class instance and should not be exposed. We can’t change the predefined call back function name, but we can obfuscate it by associating a hidden member function at runtime:

```
This["onEvent"]:=This._onEvent
```

Properties defined in the constructor using **dot notation** and **This** are suggested as public properties. However, when a property is defined using **bracket notation** in the constructor, it doesn’t show up in auto complete.

Stating with v20, you can **specify the data types** and optionally **help tips** of class properties[14]:

```
property area : Text  //option for WA Run offscreen area

property url; result : 4D.File  //result from WA Run offscreen area
```

---

[13] See Use an offscreen web area in the official blog.
[14] See Declaring Class Properties in the official blog.

## Getting Started with Go

Go in an ideal language for developing cross-platform, utility executables that can be deployed with 4D.

On Mac the recommended installation method is to use a Homebrew distribution. The package manger makes it easier to keep track of updates.

If you chose to use the official installer, you would have to remove your current copy manually each time there is an update:

```
sudo rm –rf /usr/local/g
```

```
sudo rm –rf /etc/paths.d/go
```

The official installer is somewhat better on Windows in that you can uninstall Go from the control panel.

Still, it may make sense to install with Chocolatey for its ease of managing updates.

Here is a very simple example of a CLI application written in Go:

```
package main

func main() {
    print("Hello world!")
}
```

On Mac you might want to build twice and create a universal binary for Intel and Apple Silicon:

```
GOOS=darwin GOARCH=amd64 go build -o hello-world_amd64 hello-world.go
GOOS=darwin GOARCH=arm64 go build -o hello-world_arm64 hello-world.go
lipo -create -output hello-world hello-world_amd64 hello-world_arm64
```

On Windows you probably need to build once for the current architecture:

```
go build hello-world.go
```

Go can be executed from the command line. There are also **Integrated Development Environment** options on the market. For our basic usage the Go extension for Visual Studio Code is more than sufficient.

## Where to go from here

The introduction to the plugin SDK includes the following passage:

Why the need for a plug-in? Although 4th Dimension provides hundreds of commands used to manipulate records and implement user interface, some special use or feature (sometimes platform dependent) may be needed...**specific statistics tools, file access over the network, a special user interface, or a private picture structure**. It is obvious that covering all areas of both the Macintosh and Windows operating systems by way of 4th Dimension commands would certainly lead to a product with thousands of commands, and at the same time, most users would have no need for such a large set of capabilities. Also, creating such an all-encompassing tool would make the 4th Dimension environment incredibly complex and would take most users months of study before useful results could be expected.

The same could be said of System Workers. When it comes to domain specific data processing, 4D the product or its developer community might not have the perfect toolset ready-made for that task. Before you expend time in developing a proprietary solution, it might be worth your while to search for examples written in Go with a good track record. A generic, precompiled binary CLI utility would be your best match. Alternatively, it might be possible to furnish a proprietary CLI tool thanks to the ever-growing catalog of libraries written in Go.

## Case study: CSV to JSON

4D is not particularly versatile when it comes to processing proprietary data formats. In fact, 4D doesn't really support CSV. System variables such as FldDelimit and RecDelimit, or the IMPORT DATA/EXPORT DATA commands purport to support comma-separated values but not to the degree expected in a business application, in particular the 2005 RFC 4180 standard.

Fortunately, there are standard libraries for CSV and JSON in Go. You can find good tutorials in no time.

**Tutorial**: How to create a CLI tool in Golang by Andrew Davis Escalona, Published in Level Up Coding

**Repository**: https://github.com/Andrew4d3/go-csv2json

The repository does not have binaries for Windows or Mac, but no worries, you know how to compile.

1. Download the repository

2. Locate the go-csv2json-master folder

3. Open the folder in Visual Studio Code

4. Open in Integrated Console

Build on Mac:

```
GOOS=darwin GOARCH=amd64 go build -o csv2json_amd64 csv2json.go
GOOS=darwin GOARCH=arm64 go build -o csv2json_arm64 csv2json.go
lipo -create -output csv2json csv2json_amd64 csv2json_arm64
```

Build on Windows:

```
go build csv2json.go
```

Test the executable in console:

```
csv2json -h
Usage: csv2json [options] <csvFile>
Options:
  -pretty
        Generate pretty JSON
  -separator string
        Column separator (default "comma")
```

Thanks to object-oriented programming, we can repurpose a large part of our previous project to incorporate this new console program.

Classes that require no modifications:

- _CLI
- _CLI_Controller
- _Form_Controller
- _Worker_Controller

New class for worker implementation:

```
Class extends _CLI

Class constructor($controller : 4D.Class)

 Super("csv2json"; $controller)
```

Implement the “execute” method according to the documented command line syntax:

```
Function execute($option : Variant)->$this : cs.Csv2Json
 
 $this:=This
 
 var $commands; $options : Collection
 $commands:=New collection
 
 Case of
  : (Value type($option)=Is object)
   $options:=New collection($option)
  : (Value type($option)=Is collection)
   $options:=$option
 End case
 
 var $csvFile : 4D.File
 
 For each ($option; $options)
  If (OB Instance of($option.src; 4D.File))
   /*
    resolve filesystem path
   */
   $csvFile:=File($option.src.platformPath; fk platform path)
   $command:=This.escape(This.executablePath)
   /*
    -pretty: generate pretty JSON
    -separator: comma or semicolon
   */
   If (Bool($option.pretty))
    $command:=$command+" -pretty"
   End if
   Case of
    : (Value type($option.separator)#Is text)
    : ($option.separator="semicolon")
     $command:=$command+" -separator semicolon"
   End case
   $command:=$command+" "+This.quote($csvFile.path)
   $commands.push($command)
  End if
 End for each
 
 This.controller.execute($commands)
```

## Sidebar: ChatGPT

The Go programming language has the tendency to result in formulaic code. Add to that the large population of programmers who actively work with the language, it makes an ideal candidate for AI assisted coding.

For example, if you ask ChatGPT:

> write me code in go that converts csv to json

you may get a response like:

> Here's an example Go code that reads a CSV file and converts it to JSON format:

```
package main

import (
        "encoding/csv"
        "encoding/json"
        "fmt"
        "io/ioutil"
        "os"
)

func main() {
        // Open the CSV file
        file, err := os.Open("input.csv")
        if err != nil {
                panic(err)
        }
        defer file.Close()

        // Parse the CSV file
        reader := csv.NewReader(file)
        records, err := reader.ReadAll()
        if err != nil {
                panic(err)
        }

        // Convert CSV records to a slice of maps
        var data []map[string]string
        headers := records[0]
        for _, row := range records[1:] {
                record := make(map[string]string)
                for i, value := range row {
                        record[headers[i]] = value
                }
                data = append(data, record)
        }
        // Convert data to JSON
        jsonData, err := json.MarshalIndent(data, "", "    ")
        if err != nil {
                panic(err)
        }

        // Write JSON data to file
        err = ioutil.WriteFile("output.json", jsonData, 0644)
        if err != nil {
                panic(err)
        }

        fmt.Println("CSV converted to JSON successfully.")
}
```

> In this code, we first open the CSV file using the os.Open function and parse it using the csv.NewReader function. We then loop over the CSV records and convert them to a slice of maps. Finally, we use the json.MarshalIndent function to convert the data to JSON format and write it to a file using the ioutil.WriteFile function.

Then you may be interested to know how this would work with streams. If you ask ChatGPT:

> write me code in go that converts from csv stdin to json stdout

You may get something like:

> Here's an example Go code that reads a CSV input from standard input and writes the JSON output to standard output:

```
package main

import (
        "encoding/csv"
        "encoding/json"
        "fmt"
        "io"
        "os"
)

func main() {
        // Parse the CSV input from standard input
        reader := csv.NewReader(os.Stdin)
        var data []map[string]string
        headers, err := reader.Read()
        if err != nil {
                panic(err)
        }
        for {
```

```
		row, err := reader.Read()
		if err == io.EOF {
			break
		}
		if err != nil {
			panic(err)
		}
		record := make(map[string]string)
		for i, value := range row {
			record[headers[i]] = value
		}
		data = append(data, record)
	}

	// Convert data to JSON
	jsonData, err := json.MarshalIndent(data, "", "    ")
	if err != nil {
		panic(err)
	}

	// Write the JSON output to standard output
	fmt.Println(string(jsonData))
}
```

> In this code, we use os.Stdin to read CSV input from standard input. We parse the input using the csv.NewReader function and loop over the CSV records to convert them to a slice of maps. Finally, we use the json.MarshalIndent function to convert the data to JSON format and write it to standard output using the fmt.Println function.

Does the code work? I don't know. Given the commonality of the problem, you could have searched the web the traditional way. If the request is well defined, the answer should not be too far off the mark. Also, you can keep refining your requests to get a better answer. The AI takes context into account.

AI can also be used to perform mundane work such as:

- Generate variable or function names matching task description
- Generate sample dataset
- Ask what a code fragment is doing; insert comments
- Ask why a specific error message would be thrown
- Ask about common pitfalls related to a programming challenge
- Ask for refactoring with an explanation of why one is better than the other

The results get better the more you give specific instructions, such as naming conventions and arbitrary rules. You are not talking to an actual person; you can be, in fact, you should be, as persistent and blunt as you need to be until you get a satisfactory answer.

Code snippets generated by Pretrained Transformers look suspiciously similar to the online tutorials such as the one referenced earlier, which is interesting. Even so, chatbots based on Large Language Models are known to hallucinate, so be careful!

## Count number of valid commands

```
//MARK: parse gram.4dsyntax

var $path : Text
$path:=Application file
var $isMacOS : Boolean
$isMacOS:=Is macOS
var $fileItemClass; $fileItem : Object
$fileItemClass:=$isMacOS ? 4D.Folder : 4D.File
$fileItem:=$fileItemClass.new($path; fk platform path)
var $contentsFolder : 4D.Folder
$contentsFolder:=$isMacOS ? $fileItem.folder("Contents") : $fileItem.parent
var $gramFile : 4D.File
$gramFile:=$contentsFolder.file("Resources/gram.4dsyntax")
var $gramText : Text
$gramText:=$gramFile.getText("utf-8-no-bom"; Document with CR)
var lines : Collection
$lines:=Split string($gramText; "\r")

//MARK: filter valid command names

var $commands : Collection
$commands:=New collection
$l:=New collection
For ($i; 1; $lines.length)
 $command:=Command name($i)
 Case of
  : (Match regex("@.+"; $command))
  : (Match regex("Web .+"; $command))
  : ($command="")
  : ($command="_@")
  Else
   $commands.push($command)
 End case
End for

$commands:=$commands.orderBy(ck ascending)
```

**Note**: The 3 undocumented commands **Web Event, Web Form, Web Progress** are not counted.

## Commands added in v20

1. Create deployment license
2. SET HELP MENU
3. Last errors
4. OBJECT SET SUBFORM CONTAINER VALUE
5. OBJECT Get subform container value
6. Copy parameters
7. Get license usage
8. WP DELETE TEXT BOX
9. WP New text box
10. WP Get data context
11. WP SET DATA CONTEXT

## Commands modified in v20

1. FORM LOAD
2. Get application info
3. GET PRINT OPTION
4. OBJECT Get format
5. OBJECT Get horizontal alignment
6. OBJECT SET FORMAT
7. OBJECT SET HORIZONTAL ALIGNMENT
8. Print form
9. SET CURRENT PRINTER
10. SET PRINT OPTION
11. Folder
12. LISTBOX Get auto row height
13. LISTBOX SET AUTO ROW HEIGHT
14. Method called on error
15. ON ERR CALL
16. System folder
17. HTTP GET OPTION
18. HTTP SET OPTION
19. OBJECT Get corner radius
20. OBJECT SET CORNER RADIUS
21. CALL FORM
22. CALL WORKER
23. EXECUTE METHOD IN SUBFORM
24. LISTBOX Get property
25. LISTBOX SET PROPERTY
26. Compile project
27. Get database parameter
28. OBJECT Get subform container value
29. OBJECT SET SUBFORM CONTAINER VALUE
30. SET DATABASE PARAMETER
31. WA Create URL history menu (remarks)
32. WA GET PREFERENCE (remarks)
33. WA GET URL HISTORY (remarks)
34. WA SET PREFERENCE (remarks)
35. Get locked records info
36. Get process activity
37. LOCKED BY
38. OBJECT SET ENTERABLE
39. XML SET OPTIONS
40. SOAP DECLARATION (remarks)
41. WEB SERVICE SET PARAMETER (remarks)

## Credits

Technical Note 22-09 The Power of System Workers by **Bryan Yen**. Highlights the advantages of 4D.SystemWorker compared to LAUNCH EXTERNAL PROCESS, especially its asynchronous calls. Nice demo (system_profiler/ver/wmic, find/dir) for showcasing responsiveness.

Technical Note 22-15 Intro to HTTPRequest: Asynchronous API Requests with Callbacks by **Nhat Do**. Explains in simple terms the asynchronous class API design which is actually very similar to 4D.SystemWorker. Nice demo that talks to Art Institute of Chicago IIIF API.

Technical Note 22-14 Getting Acquainted with Classes to Improve Development in 4D by **Tai Bui**. Comprehensive guide to the 4D flavour of OOP. Focuses much on studio features that increase productivity, especially declaration, code-completion and help tips.

Blog post System Worker Vs. LAUNCH EXTERNAL PROCESS by **Thomas Maul**. 3-part series that explains how the author created a platform agnostic, multi-faceted CLI wrapper class.

## References

SystemWorker on 4D Documentation. Obviously.

HTTPRequest, WebSocket on 4D Documentation. The asynchronous callback API has a lot in common with that of 4D.SystemWorker.

Web Workers on Mozilla Developer Network and WHATWG. The use of a background thread and messaging to prevent the main execution thread from being blocked or slowed down is what inspired the design of 4D.SystemWorker.

Marked. Used by the Explorer to render markdown comments as HTML.

Go Report Card. Evaluates a Go-based project hosted on GitHub.