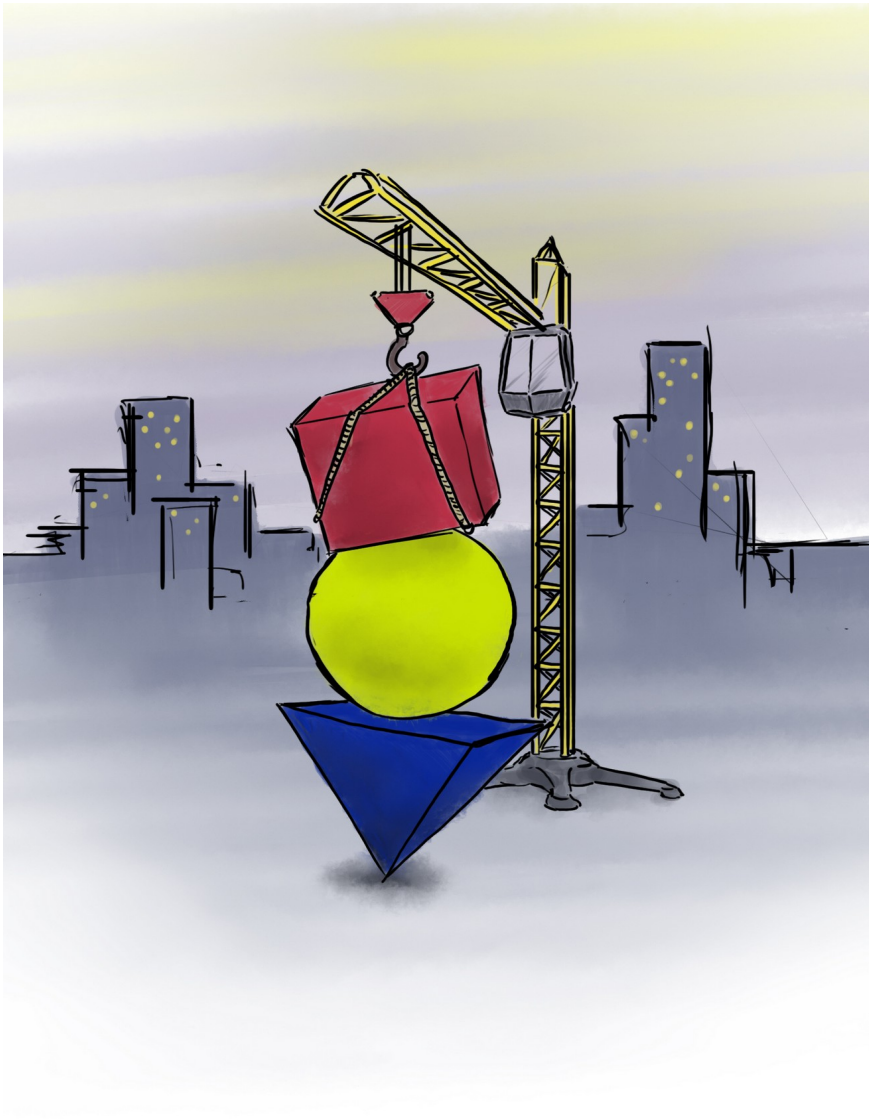
# CITRINE

## Manual / Handleiding 2025

Gabor de Mooij



Illustrations by Robert Cabri

Geschreven door Gabor de Mooij

Tekeningen door Robert Cabri

# Inhoudsopgave

# Versiegeschiedenis / Version history

| Versie/Version | Datum/Date | Verandering/Change |
|---|---|---|
| 1.0 | 2024-12-01 | Eerste versie/first version |
| 1.1 | 2024-12-28 | Div. aanpassingen/verbeteringen NL-deel |
| 1.2 | 2025-01-16 | Aanpassing in 2.3 mbt recursie |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# Preface / Voorwoord

Writing a guide for Citrine is both exciting and challenging. Citrine is a unique programming language that allows you to code in your own native language—whether that’s Italian, French, or something else! While I'm fluent in English and Dutch, Citrine currently supports around 110 languages. As you can imagine, it’s impossible for me to create guides for all of them overnight!

Another challenge is that Citrine can be used in many different ways. Some people want to integrate it into educational programs, while others just want to use it to write code in their own language. This means I have to balance the needs of both beginners and experienced programmers.

As a result, this guide is written with a broad audience in mind. If you're an advanced programmer, you might find some sections a bit basic. If you're a beginner, you may find a few concepts tricky. So, it’s a balancing act! My aim is to keep things clear and simple, focusing on the fundamentals. I’ve also included the Media Plugin from the start, so you can dive into fun stuff like graphics, music, and networking right away.

This guide is also bilingual, in English and Dutch. The first part is in English, followed by the Dutch version. I’ve combined both in the same booklet to keep things organized. So, this guide may feel a bit unconventional, but I hope it doesn’t detract from your enjoyment of working with Citrine. Citrine is a fun programming language that allows you to quickly create apps and games, which can then be exported to PC, mobile, and the web. Plus, you get to program in your own language.

While the examples in this booklet are in English and Dutch the online manual on the Citrine website offers examples of almost every functionality of the Citrine programming language in every supported human language.

While Citrine embraces all human languages, I’ve chosen not to support artificial languages (like Esperanto), fictional ones, or extinct languages (like Latin). While these languages are fascinating, including them would make the project unmanageable. There are about 6,500 human languages, and I’m aiming to support as many as possible! That’s already too many and will likely never be fully achievable, but I’ll see how far I can go.

Through my international work, I’ve come to appreciate the beauty of all human languages. Sadly, some are nearly extinct. With Citrine, I want to give people the opportunity to do more with their own languages, and I hope to contribute in some small way to preserving them. After all, what’s more enjoyable than creating something beautiful in your own language?

Learning to program is also incredibly useful—it helps you think abstractly and understand technology better. Even if you don’t become a professional developer, it’s great to have some scripting skills in today’s tech-driven world.

As you can see, Citrine’s goals are ambitious but exciting! I’ve done my best to create a concise guide, but I’m always open to feedback and suggestions. Feel free to reach out with any ideas or questions. Let’s make Citrine a fun and collaborative project!

I hope you have a fantastic time working with Citrine!

-

Een handleiding schrijven voor Citrine is geen gemakkelijke opgave. Citrine is namelijk een programmeertaal waarmee je in je eigen taal kunt programmeren, zoals bijvoorbeeld Italiaans. Zelf spreek ik echter alleen Engels en Nederlands. Hoewel Citrine momenteel ongeveer 110 talen ondersteunt, kan ik op korte termijn geen 110 handleidingen produceren (nee, zelfs niet met AI). Daarnaast is er nog een uitdaging: Citrine kan in verschillende contexten worden gebruikt. Sommige gebruikers willen Citrine als onderdeel van een lespakket gebruiken, terwijl anderen gewoon graag in hun eigen taal willen programmeren.

Dit betekent dat ik zowel beginners als gevorderde programmeurs moet aanspreken. Het resultaat is dat deze handleiding geschreven is voor een zeer breed publiek. Gevorderde programmeurs, voor wie Citrine ‘gewoon een nieuwe taal’ is, zullen het niveau van deze handleiding soms te laag vinden. Beginnende programmeurs daarentegen kunnen de leercurve juist te steil vinden. Het schrijven van deze handleiding is dus een kwestie van balanceren.

Om iedereen zo goed mogelijk tevreden te stellen, heb ik ervoor gekozen om de handleiding zo kort mogelijk te houden en me vooral te richten op de basisprincipes. Ik gebruik vanaf het begin de Media Plugin, zodat je meteen kunt werken met grafische toepassingen, muziek, geluid en netwerken. Deze handleiding is tweetalig: Engels en Nederlands. Eerst komt het Engelse deel, gevolgd door de Nederlandse versie, in hetzelfde boekje zodat alles bij elkaar blijft. Dit maakt de handleiding een beetje een buitenbeentje, maar ik hoop dat dit je plezier in het werken met Citrine niet in de weg staat.

Citrine is een leuke programmeertaal waarmee je snel en eenvoudig apps en games kunt maken en exporteren naar pc, mobiel en het web. En, je kunt programmeren in je eigen taal. De voorbeelden in dit boekje zijn in het Engels en Nederlands, en voor andere talen kun je terecht op de website. De online documentatie biedt voorbeelden van elke functionaliteit in elke ondersteunde menselijke taal. Naarmate de tijd vordert, zal ik deze handleiding uitbreiden met voorbeelden in andere talen. Het is dus een werk in uitvoering.

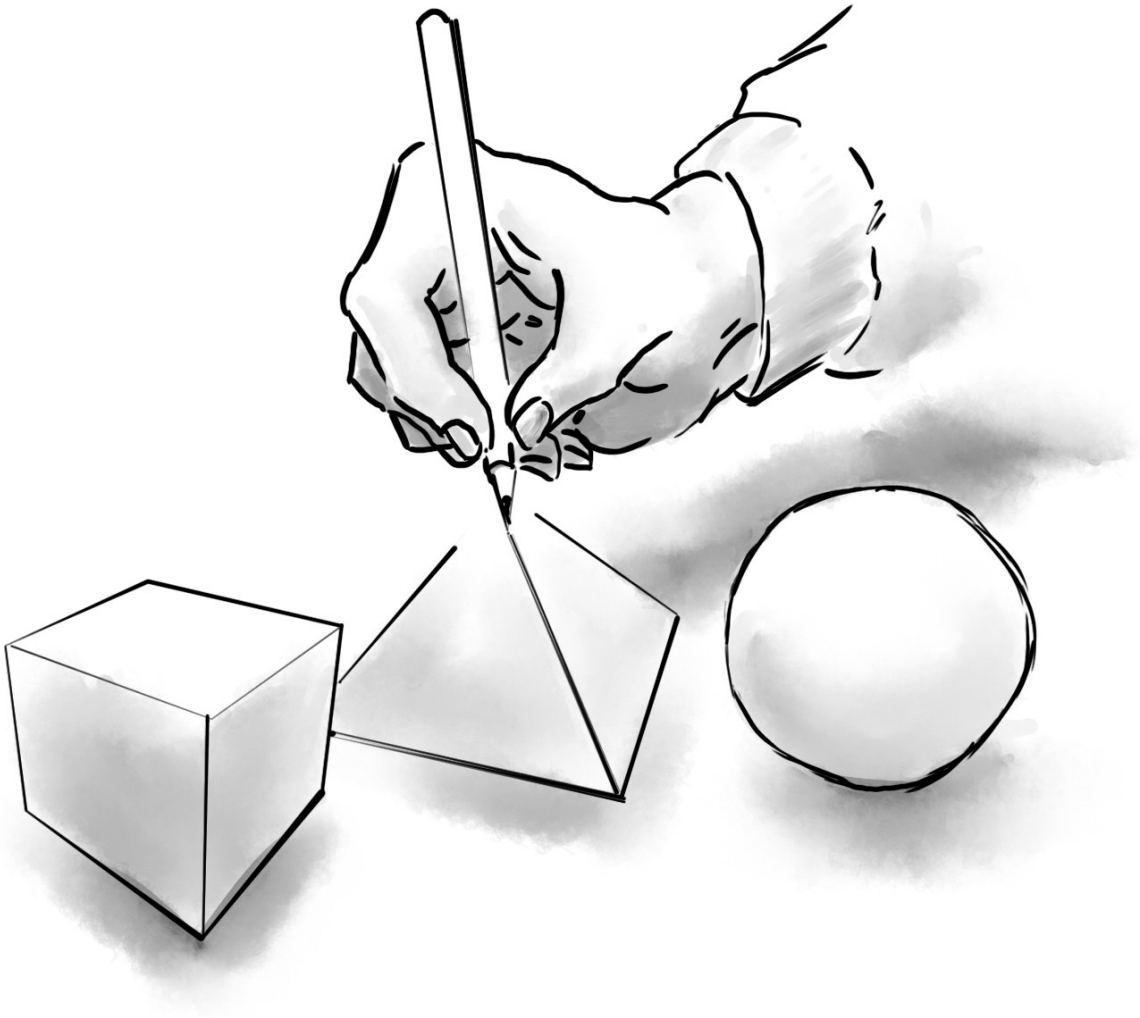
Vanaf eind 2024 wordt Citrine mijn fulltime project en heb ik (bijna) geen andere inkomsten dan uit Citrine. Daarom is het belangrijk dat, als je Citrine waardeert, je me steunt. Elke vorm van steun is welkom, en iedereen is welkom. Ik houd me niet bezig met politiek; we proberen hier samen een mooie programmeertaal te maken en te gebruiken, zodat we in onze moedertaal kunnen programmeren en mooie dingen kunnen maken.

Ik trek de lijn alleen bij kunstmatige talen (zoals Esperanto), fantasietalen en uitgestorven talen (zoals Latijn). Hoewel deze talen prachtig zijn, zou hun toevoeging het project te zeer uit de hand laten lopen. Er zijn ongeveer 6.500 menselijke talen in de wereld, en met Citrine wil ik ze allemaal ondersteunen. Dat is natuurlijk ambitieus en onmogelijk, maar ik kijk gewoon hoe ver ik kom. Elke taal is welkom. Alleen je begrijpt dat als ik daarbij ook nog eens alle ‘overige’ talen zou gaan ondersteunen het project compleet ontspoort.

Door mijn internationale werk heb ik gemerkt hoe mooi menselijke talen zijn. Helaas dreigen sommige talen te verdwijnen (zoals Jakoets). Met Citrine wil ik mensen de mogelijkheid geven om meer met hun eigen taal te doen, en zo hopelijk een bijdrage te leveren aan het behoud van hun moedertaal. En wat is er leuker dan in je eigen taal iets moois te maken?

Daarnaast is leren programmeren essentieel. Het leert je abstract denken en vergroot je begrip van technologie. Ook als je geen programmeur bent, is het handig om te kunnen ‘scripten’. In een samenleving die zo sterk afhankelijk is van technologie, is het nuttig als je begrijpt hoe dingen werken en een beetje kunt programmeren of op zijn minst weet hoe je code moet lezen.

Zoals je ziet, zijn de doelen van Citrine divers en breed. Dat maakt het lastig om een compacte handleiding te schrijven, maar ik heb geprobeerd om er het beste van te maken. Ik sta altijd open voor suggesties, verbeteringen en vragen. Je mag me altijd een e-mail sturen of met me chatten. Laten we er samen een leuk project van maken! Hopelijk steun je me ook bij de verdere ontwikkeling van Citrine. Veel plezier met Citrine!



# 1 Getting started

This guide is intended for both beginner and advanced programmers. Experienced programmers who are learning Citrine as a new language will probably be able to skip certain sections here and there.

If you're just starting to program, you're about to embark on a very adventurous journey. Along the way, you'll gain all sorts of new experiences. Strangely enough, you'll also discover a lot about yourself! You'll find that your thinking isn't as logical as you might have hoped, and you'll notice many "gaps" in your thought process. Every programmer learns daily that the world is far more complex than they expected. At the same time, dealing with that complexity becomes an art in itself. Sometimes, the solutions embedded in lines of code are so beautiful and elegant that you could almost call them works of art!

Personally, I find most computer books quite boring. I just can't get through them. After a few pages, I start experimenting and jumping from chapter to chapter and back again. Most computer books are tedious because the authors over-explain everything. They're terrified of leaving out any detail. This book works differently. As I said, I compare learning to program—or, for advanced programmers, learning a new language—to going on a journey. You don't make that journey by just reading this book. This booklet is meant to pack your metaphorical backpack in preparation for that journey. The real adventure begins after the last page (or perhaps even earlier). I won't take that journey for you. You'll have to do it yourself.

In this guide, I will only explain the basics, show you what's possible, and give you pointers on how to continue finding your way. But ultimately, you must take the journey on your own. You need to explore and experiment for yourself. Sometimes that will be difficult and frustrating. Don't make it harder than it needs to be. If things aren't going well, just set it aside for the day. The journey isn't going anywhere. You can always decide to pick it back up tomorrow. And I promise you one thing: that path is full of surprises!

## 1.1 Programs

With Citrine, you can create apps, games, or even serious business applications—all of which are computer programs. To create a computer program, you need to tell the computer what to do. This process is called programming. Computer programs are written in a programming language, and Citrine is one such language. A programming language defines the rules for writing a program. You write instructions in a text file using a simple text editor like Notepad. You can start your program by dragging it onto the Citrine icon. Another way to run your program is through the terminal. To do this, open the terminal and type:

```
ctren <program>
```

Here, replace `<program>` with the name of the program you've written.

The computer reads a Citrine program from top to bottom and from left to right—just like regular language. A Citrine program consists of sentences, and each sentence ends with a period. In each sentence, you describe what the computer should do. Sentences that start with a hashtag (#) are ignored by the computer. You can use this feature to add comments to your code.

Use ctren for the English version of Citrine. All language editions have their own interpreter, for Frysian you use ctrfy, for Italian use ctrit and so on…

Fortunately, you don't have to reinvent the wheel entirely by yourself. A lot of functionality has already been programmed for your computer. You can simply build upon it and create your own projects.

## 1.2 Messages

Citrine is a pure object-oriented language. This means, among other things, that all the built-in functionalities are provided in the form of objects. If you want to accomplish something, you need to "send a message" to an object. For example:

```
Moment month.
```

This is a small sentence in Citrine. In this example, we are sending the message 'month' to the object 'Moment.' The object here is 'Moment,' and the message is 'month.' Generally speaking, when you want to send a message to an object, you write it like this:

```
<object> <message>
```

In place of `<object>`, you can insert any object of your choice, and in place of `<message>`, you insert the message you want to send. In our case, we send the message 'month' to the object 'Moment.' The result is that we ask the 'Moment' object for the current month. The 'Moment' object contains functionality for handling dates and times. In this case, it might return the number 8, which represents August (the 8th month). Since we are not using this value immediately, it will be forgotten. To use this value later in the program, we need to store it.

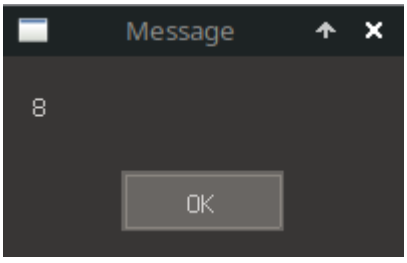
Suppose we want to store the month number as 'm.' First, we need to reserve a place in the computer's memory for 'm' using the declaration symbol (`>>`). Then, we assign the result to 'm' using the assignment symbol (`:=`). It looks like this:

```
>> m := Moment month.
```

Now we have stored the month number (e.g., 8) under the name 'm.' This is called a variable. Reserving memory space is known as declaring. Now that we have stored the month number under 'm,' we can use it later in the program. For example, if we want to display the month number on the screen, we use another object, the 'Media' object. We send the message 'show:' to the 'Media' object, and as an additional piece of information, we specify what we want to display—namely 'm,' the month number. This additional piece of information is called an argument. We end the sentence with a period:

```
>> m := Moment month.
Media show: m.
```

This works and displays the month number in a window like this:



Instead of a month number, we can also display a random number. To get a random number, you can send the message 'between: and:' to the 'Number' object. In this case, you need to send two arguments: the lower and upper limits between which the random number should fall. For instance, if you want a random number between 1 and 10, you write:

```
>> x := Number between: 1 and: 10.
Media show: x.
```

Each time you run this program, it shows a random number. However, this doesn't mean you'll see a different number every time—it's possible to see the same number multiple times. You just can't predict which number will appear.

Technically, we don't need to store the random number first to display it. We could combine the two lines of code into one. In that case, we put the expression that generates the random number inside parentheses after the 'show:' message:

```
Media show: (Number between: 1 and: 10).
```

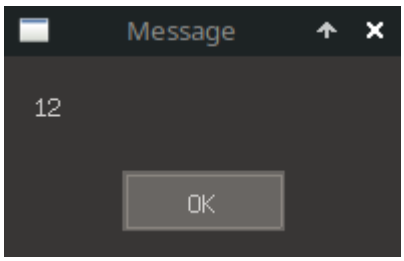
We can also display text on the screen (instead of numbers). It's a tradition to start learning a new programming language by displaying "Hello World" on the screen. In Citrine, you write this as:

```
Media show: ['Hello World!'].
```

Since we want to distinguish the text in the argument from the code itself, we need to specify that this is literal text. Otherwise, Citrine might get confused and not know what is meant as code and what should be displayed. The solution is to enclose all literal text in `['…']`.

Without realizing it, you've already created new objects. Whenever Citrine encounters a number (1, 2, 3…) or text (['Hello World']), it automatically creates an object behind the scenes. This means you can also send messages to a number or a piece of text. For example, the following program displays the number of characters in 'Hello World':

```
Media show: ['Hello World!'] length.
```



Here, we have two messages. First, we send the message 'length' to the text object [‘Hello World’], and then we send the 'show:' message to the 'Media' object with the result of the previous operation (i.e., the number of characters). How does Citrine determine the order of execution?

As mentioned earlier, Citrine reads from left to right. Clauses in parentheses are read first. Messages without arguments are executed first, followed by messages with one argument that don’t have a colon (I’ll give an example shortly). Finally, messages with colons and one or more arguments are executed.

The message 'length' returns a number. We can also send a message to this number, such as `+ 1`. This is a message that takes one argument and doesn’t have a colon. The rule is that for messages consisting of a single character and one argument, you omit the colon. In practice, these messages are often arithmetic operations such as `+` (addition), `-` (subtraction), `/` (division), and `*` (multiplication).

```
Media show: [‘Hello World!’] length + 1.
```

In the example above, we display the length plus 1. First, the 'length' message is sent to the text 'Hello World,' which returns 12. Then we add 1 to it because messages without colons are executed after messages with no arguments. Finally, the message with the colon ('show:') is executed, and we see 13 on the screen.

Please note that you must use a space before and after +.

## 1.3 Tasks and Loops

It's possible to chain multiple messages together. For instance, if we want to display a countdown on the screen (3...2...1...0), we could do it by sending several `show` messages like this:

```
Media show: 3.
Media show: 2.
Media show: 1.
Media show: 0.
```

But there's an easier way. If you want to send multiple messages to the same object, you can chain them together. If a message has an argument, as with `show:`, you need to separate them with a comma, otherwise Citrine might get confused.

```
Media show: 3, show: 2, show: 1, show: 0.
```

For readability, you can also spread this over multiple lines:

```
Media
    show: 3,
    show: 2,
    show: 1,
    show: 0.
```

However, we can simplify this even further. Instead of manually counting down, we can let a task handle it. You can turn a series of statements into a task by wrapping them in `{...}`. For example, we can turn the display sequence into a task like this:

```
{
  Media show: 3.
}.
```

Behind the scenes, Citrine creates a task object from the sentences within `{...}`. You can also send messages to this object. To start a task, you send the message `start`.

```
{ … } start.
```

Sending `* 3` will make the task repeat three times. A task that repeats in this way is called a *loop*. But how do we know which iteration of the loop we're in? Are we in the first round, the second, or the third?

When you send a message to a task, relevant information, such as the current *iteration* number, is passed along as a parameter. Parameters are placed at the beginning of the task and are used to store information from outside the task. These parameters are only available within the task. If you start

the task with `:i`, the current iteration number is stored in `i`. The iteration number is also known as the index, and it's passed as the parameter `i`.

```
{ :i
    Media show: i.
} * 3.
```

This will display 1, 2, 3. But we actually want a countdown... 3, 2, 1, 0. That means we need 4 iterations. Additionally, instead of counting up, we need to count down. During the first iteration, `i` is equal to 1, and to display 3, we need to subtract 1 from 4, which gives us `4 - 1`. In the second iteration, `i` is equal to 2, so we do `4 - 2`, and so on. In general, we can say that `4 - i` will always give us the correct countdown number:

```
{ :i
    Media show: 4 - i.
} * 4.
```

This will display 3, 2, 1, 0 as expected.

## 1.4 Branching

Instead of counting down to 0, we can also count up from 1 to 20:

```
{ :i
        Media show: i.
} * 20.
```

Now, suppose we want to skip the unlucky number 13—how can we do that? In this case, we send the message `!=:` to `i` with 13 as the argument:

```
i !=: 13
```

This checks if `i` is not equal to 13. If it is equal, the result will be `True`, and if it's not, the result will be `False`. Both `True` and `False` are objects, and you can send messages to these objects. A useful message for such objects is `true:`, which takes a task as an argument. The `True` object executes the task passed as an argument, while the `False` object ignores it. The opposite is true for the `false:` message—`False` will execute the task, and `True` will ignore it. Using this trick, we can perform a task based on certain conditions. In our case, we want to show the number only if it is not equal to 13. To check if a number is equal to another, we use `=`. To check if it's unequal, we use `!=:`. Since `=` consists of a single character, it doesn't require a colon. But `!=:` has two characters, so a colon is needed. In this case, we write `i !=: 13`, which gives us either `True` or `False`.

```
{ :i

        (i !=: 13) true: {
                Media show: i.
        }.
} * 20.
```

Notice that we use parentheses around `(i !=: 13)`. This is because, without them, it might be unclear that `false:` is a separate message. Otherwise, Citrine might interpret it as `i !=: 13 false: {..}`, assuming `!=:` and `false:` are part of the same message with multiple arguments! That's not what we want. Another way to write it is by using a comma:

```
{ :i
      i !=: 13, true: {
            Media show: i.
      }.
} * 20.
```

This is also allowed, and you can choose the style that you prefer. There's another way to skip the unlucky number 13:

```
{ :i
        (i = 13) continue.
        Media show: i.
} * 20.
```

Here, we send `=` to `i` with 13 as the argument. The result will be `True`. When you send the message `continue` to `True`, the rest of the task is skipped, and the program moves to the next iteration. In this case, `i` will become 14, and the `Media show` message will be skipped when `i = 13`.

In all these cases, the program branches into two paths. One path displays the number (for all values except 13), and the other skips displaying the number (for 13).

If we not only want to skip 13 but also stop the task entirely (meaning we skip all subsequent iterations), we can send the message `break` instead of `continue`.

```
{ :i
        (i = 13) break.
        Media show: i.
} * 20.
```

You can also return a result from a task using `<-`. This allows you to continue executing a task until another task returns `False`. In this case we increment the variable i ourselves using the message add:.

```
>> i := 1.
{
        Media show: i.
        i add: 1.
} while: { <- i < 13. }.
```

By the way, note the difference between + and add:. The message + yields a new number, while add: changes the number. So i + 1 will yield a new number but won't change the value of i. On the other hand add: will change i. Same applies to messages like multiply: (versus *) and so on.

You can also combine `True` and `False` objects. For example, if you want to perform a task when someone has a score greater than 100 and is in level 2, you can write:

```
(score > 100 and: level = 2) true: { … }
```

The message `and:` returns `True` if both `True` objects are involved. Similarly, you can use `or:`:

```
(score > 100 or: level = 2) true: { … }
```

In this case, the task is executed if the player has more than 100 points **or** is in level 2. The message `nor:` returns `True` only if both sides are `False`.

## 1.5 Sequences and Lists

Instead of counting with numbers, we can also count with words like "one," "two," "three." To do this, we first place the words in a sequence. For this, we use the sequence object. To create a new sequence, you send the message `new` to `Sequence`. Then, you can add objects to the sequence using the message `add:`. In the following example, we create a new sequence called `r` and add the words for counting to it:

```
>> r := Sequence new
add: ['one'],
add: ['two'],
add: ['three'].
```

As you can see, there's no comma between `new` and `add`. That's because `new` doesn't require a colon, so there's no ambiguity that `add` is a separate message. Instead of `add`, you can also use `;`:

```
>> r := Sequence new ; ['one'] ; ['two'] ; ['three'].
```

This is not only shorter but also allows you to omit the commas because `;` doesn't require colons. To make our loop use the sequence, we need to convert the parameter `:i` into a word. We'll receive a number and need to map it to a word. The sequence object can do this for us. If we send the message `at:` followed by a number, it will return the object at that position in the sequence. So, to get `['one']`, we can send `at: 1`. Since our words are already at positions 1, 2, and 3, we can pass `i` as the position:

```
>> r := Sequence new ; ['one'] ; ['two'] ; ['three'].
{ :i
        Media show: (r at: i).
} * 3.
```

Note: instead of `at:` you may also use `?`.

However, there's a drawback to the code above. Every time you add a word to `r`, you also need to update the number of repetitions (currently 3). This means you have to make two changes—otherwise, not all words will be shown (for example, if there are 4 words, only 3 will be displayed).

There's a better way. Sequences also support the `each:` message. The task you pass will be performed on each element in the sequence. Your task receives two parameters: one for the position number and one for the object at that position.

```
>> r := Sequence new ; ['one'] ; ['two'] ; ['three'].

r each: { :i :word
        Media show: word.
}.
```

You can create a sequence from a text by sending the `split:` message to a text object, followed by the character used to split the text. We could recreate the sequence from above like this:

```
>> r := ['one,two,three'] split: [','].
```

To verify, let's count the number of elements with the message `count`.

```
Media show: r count.
```

This will give us the exact same sequence.

Let's explore some other useful functions of `Sequence`.

To retrieve a specific range of elements from a sequence, use `from:length:`:

```
>> fruitBasket := Sequence new ; ['apple'] ; ['pear'] ; ['banana'] ; ['kiwi'] ;
['lemon'].
>> fruits := fruitBasket from: 2 length: 3.
```

This will result in the collection `['pear'] ; ['banana'] ; ['kiwi']` in the variable `fruits`. You can also replace elements:

```
fruitBasket replace: 3 length: 2 with: (
      Sequence new ; ['grape'] ; ['melon'] ; ['tomato']
).
```

Now, the `fruitBasket` will contain `['apple'] ; ['pear'] ; ['grape'] ; ['melon'] ; ['tomato'] ; ['lemon']`.

This will replace 2 items starting from item 3 with the items in the provided replacement sequence. Note that, in contrast to other programming languages we always start counting from 1. So the first element in a sequence is number 1, not 0.

If you have two sequences, you can combine them to create another type of collection: a list. A list consists of object pairs and works like a dictionary. A list consists of *keys* and values. You can find a value by supplying the corresponding key. Suppose we have two sequences, one for scores and one for player names. The score of player 1 in the player sequence corresponds to the score at position 1 in the score sequence.

```
>> scores := Sequence new ; 100 ; 200 ; 300.
>> players := Sequence new ; ['Anna'] ; ['Rob'] ; ['Tessa'].
```

With the `per:` message, you can combine the sequences into a list:

```
>> high-scores := scores by: players.
```

Now, if you want to know Rob's score, you can find out as follows:

```
Media show: (high-scores at: ['Rob']).
```

Full code:

```
>> scores := Sequence new ; 100 ; 200 ; 300.
>> players := Sequence new ; ['Anna'] ; ['Rob'] ; ['Tessa'].
>> high-scores := scores by: players.
Media show: (high-scores at: ['Rob']).
```

Or you can write:

```
Media show: high-scores Rob.
```

Similarly, you can easily add a new player and their score to the high scores:

```
high-scores put: 500 at: ['Max'].
```

Or:

```
high-scores Max: 500.
```

Both in sequences and lists, it's allowed to request a non-existent element:

```
Media show: high-scores Tom.
```

This won't result in an error. The result is `None`. `None` is a special object that represents emptiness or the absence of information. You can perform an important test on any object to check if it is `None`:

```
high-scores Tom None? true: {
        Media show: ['This player is not in the game!'].
}.
```

Only the `None` object will return `True` for the question `None?`. All other objects will always return `False`.

## 1.6 Text Objects

A common problem when working with text is that you often need to combine different types of information. In a game, for example, you might have a score (let's say 100), and you want to display it in a message like: "Your score is 100!". So, how do we combine the number 100 (the score) with the text "Your score is..."? In Citrine, there are two ways to do this. First, you can use +, like so:

```
>> score := 100.

Media show: ['Your score is: '] + score + ['!'].
```

Another method is to replace a placeholder in the text with the actual score. You first create the text with a temporary marker where the score should appear, like `<score>` (but you can use any marker you prefer). Then, you send this placeholder text as a message to the text object, with the replacement text (in this case, the actual score, the number 100) as an argument. The code would look like this:

```
>> score := 100.

Media show: (['Your score is: <score>!'] <score>: score).
```

The advantage of this second method is that it’s easier on the eyes. The full text, with the temporary marker, is more readable since it’s not broken up by various symbols. This reduces the likelihood of making mistakes. You can also replace parts of the text using `replace:with:`:

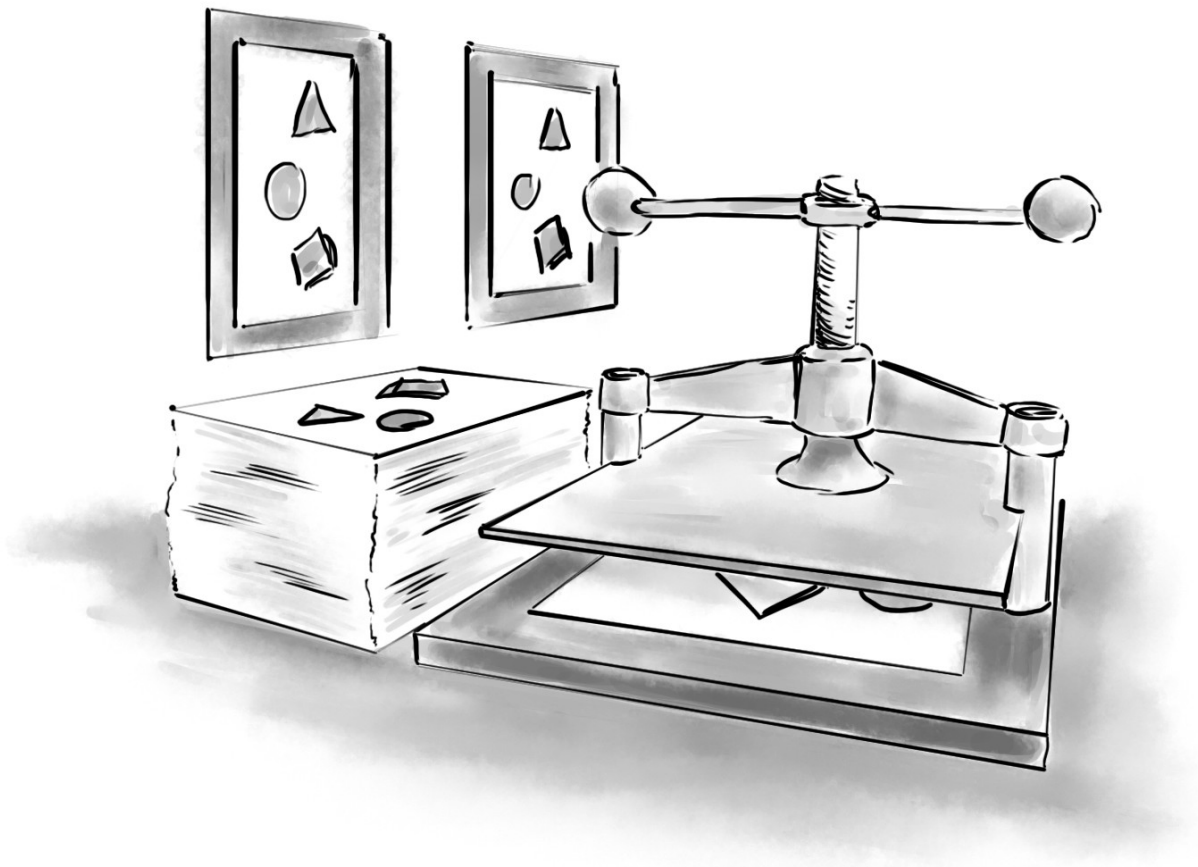
```
>> text := ['Hello <name>!'] replace: ['<name>'] with: ['Alice'].
```

Instead of using dialog windows, you can also write text to standard output (also known as stdout), which is often linked to the terminal window:

```
Out write: ['Hello stdout!'].
```

Errors are typically send to an error log using:

```
Program error: ['Test Test!'].
```



# 2 Objects

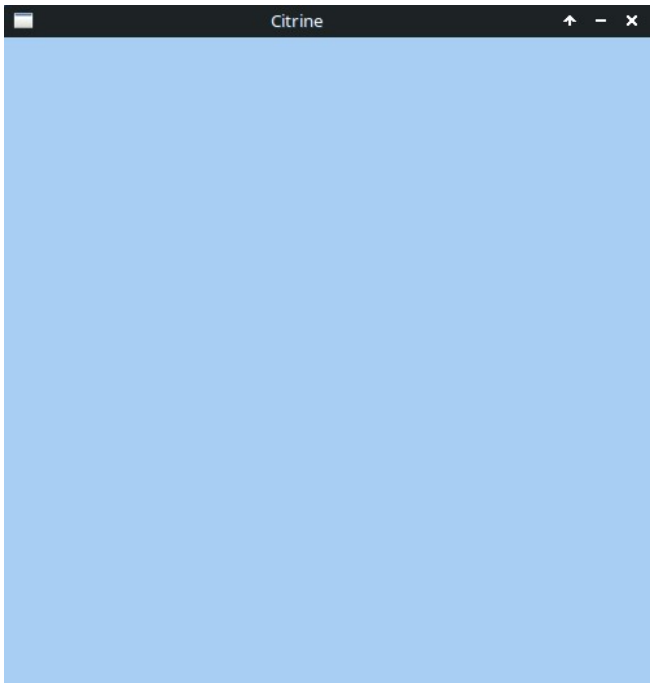
Playing with dialogs and texts is great to learn the basics of Citrine. However, you also want to see some real action, like graphics, sound, music and more. In this chapter I'm going to discuss how to use the more sophisticated aspects of objects. While doing so, I will show you how you can display images, make them move over the screen, animate them and use all sorts of other features that the Media Plugin offers.

## 2.1 Windows and Images

So far, we've only displayed 'boring' message windows on the screen. Let's change that! Take a look at the following code:

```
Media screen: ['sky.png'].
```

If you run this program (and the image file `sky.png` is in the same folder), a window will open with the sky as the background image. The window stays open until the user clicks the close button. The program also 'pauses' at the `screen:` message until the user closes the window.



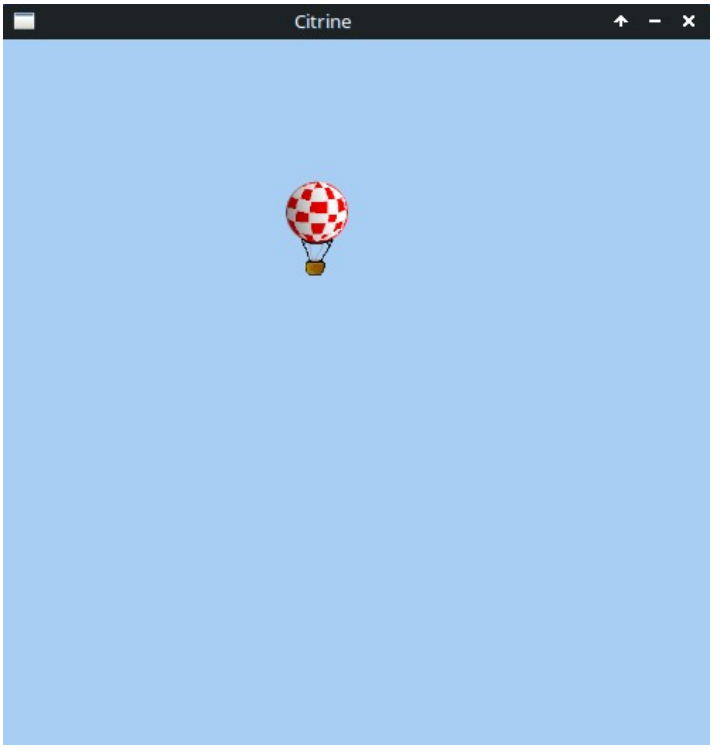

The `Media` object offers many more possibilities. For this reason, the `Media` object 'brings along' several related objects, such as `Image`, `Music`, and `Sound`. You can probably guess what these are for. But before you can use these objects, you need to load the `Media` object first. Previously, this happened automatically because we were using the `show:` message. As soon as you send a message to `Media`, it gets loaded. This works the same for any plugin. If Citrine does not know the object you reference, it will try to load it as a plugin. However, now we don't want to show anything directly; instead, we want to use the other objects from the media family. Therefore, we need an excuse to load the `Media` object. The easiest way is to say we want our own `Media` object:

```
>> m := Media new.
```

This creates your own (media) object. The `new` message creates a new object, using the receiver as the blueprint. In this case, we are creating a new object `m`, based on `Media`. This means that you can now send any message that you could send to `Media` directly to `m` as well. However, it's not the same object; it's a different object with the same messages. We call this *inheritance*. The object `m` *inherits* the messages from `Media`, unless we later add new messages to `m` that `Media` doesn't have. I'll explain more about this later. Plus, using `m` means less typing than `Media`, so that's a nice bonus.

Thanks to the message sent to `Media`, we now also have access to the `Image` object, which allows us to draw images over the background in the window. In the following example we use a picture of a balloon that I created myself.

```
>> m := Media new.
>> balloon := Image new: ['balloon.png'].
balloon x: 200 y: 100.
m screen: ['sky.png'].
```




With the `new:` message, we create our own image, providing the file that serves as the source for the picture as an argument. After that, we need to choose a position in the window to place the balloon. We pick (200, 100), meaning 200 pixels from the left side of the window (x) and 100 pixels from the top (y). We can also make the balloon move:

```
balloon to-x: 500 y: 100.
```

You can adjust the speed of the movement using the `speed:` message.

## 2.2 Events and Methods

As mentioned earlier, we can send messages to objects, which then perform tasks. You can also extend the functionality of an object using a special message called `on:do:`. This message requires two arguments: the message that the receiving object should respond to, and the task to be executed when the new message is received.

Let's say we want our balloon from the previous example to float back and forth across the screen. When we send the balloon to position x = 500, we need to know when it gets there so we can reverse its direction. Fortunately, the image object sends a notification when it reaches its destination. It sends a message to itself called `destination`. However, this message doesn't trigger any task by default, and it essentially gets ignored. We need to attach a task to this message ourselves.

It might seem strange that an object sends a message to itself when nothing happens as a result. The reason is that it gives the programmer an opportunity to 'intervene' during key events. We'll extend our balloon with the `destination` *method* as follows:

```
>> m := Media new.
>> balloon := Image new: ['balloon.png'].

balloon x: 10 y: 100, to-x: 500 y: 100.
balloon on: ['destination'] do: {
        self x? >=: 500, true: {
                balloon to-x: 10 y: self y?.
        }, else: {
                balloon to-x: 500 y: self y?.
        }.
}.

m screen: ['sky.png'].
```

Here, we say we are adding the `destination` *method*. A *method* is the combination of a message and the associated task. When using `on:do:` to add a method, make sure to write the message without its arguments. For example, a message like `between: 1 and: 2` would be written as `between:and:`. The colons indicate where the arguments will go. At the start of the task block, you should define any parameters that correspond to the method's name.

In this example, when the balloon reaches its destination, it checks whether it is on the left or right side of the screen by querying its x position (using the `x?` message) and comparing it to 500 (the right side). We send this message (`x?`) to `self`. The keyword `self` always refers to the object to which the method belongs. If the x-position is greater than or equal to 500, the balloon changes direction and moves to the left side of the window. Since the y-position (the balloon's height) remains the same, we set it to its current value (`y?`). For the opposite condition, instead of using `false:`, we use `else:`, which is an *alias*. Both messages do the same thing.

Another kind of event you can trigger is a timer. This allows you to make something happen after a certain amount of time. Here's how you set a timer:

```
media timer: 1 after: 1000.
```

The first argument is the timer number, and the second is the number of milliseconds. In this example, timer 1 will go off after 1 second. When the timer goes off, `media` receives a `timer:` message with the timer number as an argument. To link a task to the timer, you can do the following:

```
>> m := Media new.

m timer: 1 after: 1000.

m on: ['timer:'] do: { :number
        (number = 1) true: {
                Media show: ['Time's Up!'].
        }.
}.

m screen: ['sky.png'].
```

There are also other events. Once the window opens you receive a the `start` message, upon closing you get the `end` message. So you can prepare something at the start of your app or game. Also, upon each step in the app or game, you get the message `step`.

To switch off a timer, set it to -1.

## 2.3 Properties

In the following example, we'll create a real airshow with balloons! We'll arrange a formation of 5 balloons, each with its own starting position, flying back and forth across the screen. The challenge is, how do we keep track of which balloon should return to which starting position? Ideally, we'd like to store this information with the balloon itself, so we don't have to manage it separately, which would quickly become chaotic. Fortunately, this is possible because an object can have *properties*.

To achieve this, we first create a new `Balloon`.

```
>> Balloon := Image new.
```

This time we use a capital letter and omit the source file `balloon.png`, because this `Balloon` is not just any balloon. It's the *prototype* we'll use to create our 5 balloons. In this `Balloon` object, we define the behaviors and properties that all derived balloons should have. One of the first things we'll do is ensure that our prototype `Balloon` can create new balloons:

```
Balloon on: ['new'] do: {
        <- Balloon new: ['balloon.png'].
}.
```

Here, you immediately see the advantage of using a prototype. Now, you can simply call:

```
Balloon new.
```

to get a new balloon, without needing to specify the image file each time. But an even greater benefit is that each balloon you create will have its own unique properties, separate from the prototype.

A sharp-eyed (or experienced) programmer might wonder why this doesn't cause an infinite loop. After all, `Balloon new` calls `Balloon new` again. Good question! First, we send 'new:' instead of 'new'. So it's not the same message at all! But what would happen if this actually was the same message? Citrine intercepts this recursion and ensures that the task isn't repeated endlessly. Instead, the task from the `Balloon` prototype's parent object, `Image`, is executed. If you *do* want to intentionally use *recursion*, meaning you want the method to actually call itself (even if it risks creating an infinite loop), you need to prepend the message with the special keyword `recursive`.

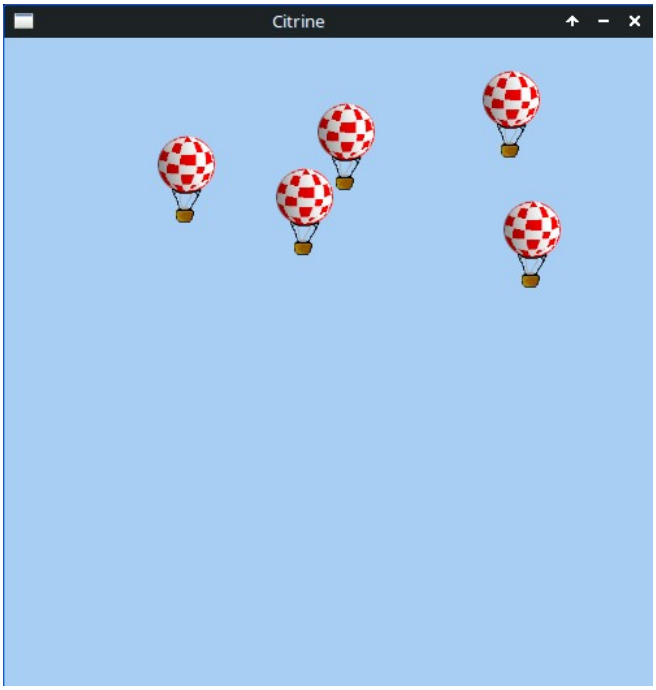
Our `new` method calls the `new` method from `Image` to create a new image, but this time it includes all the additional methods from `Balloon`. Instead of 'Balloon new' you may also use 'self new'.

Here's the complete example of how to create the balloon formation:

```
>> m := Media new.
>> Balloon := Image new.
>> fleet := Sequence new.
```

```
Balloon on: ['new'] do: {
        >> balloon := self new: ['balloon.png'].
        <- balloon.
}.

Balloon on: ['start-x:y:'] do: { :x :y
        own x := x.
        own y := y.
        self x: own x y: own y.
}.

Balloon on: ['destination'] do: {
        self x? >=: 500, true: {
                self to-x: own x y: own y.
        }, else: {
                self to-x: 500 y: own y.
        }.
}.

{ :i
        >> pos := i * 25.
        >> b := Balloon new
                start-x: pos y: pos,
                to-x: 500 y: pos.
        fleet add: b.
} * 5.

m screen: ['sky.png'].
```

Here is what it looks like:




In this example, we create a new `Balloon` prototype, and for each balloon in our fleet, we assign a unique starting position and make them fly back and forth. The `start-x:y:` method stores the

starting position, and the `destination` method makes the balloon fly to the right edge of the screen (x >=: 500) and then return to its starting position. The properties are stored by prefixing the variables with the *own* keyword. They will be accessible only from within methods that belong to the object (or other objects that have been derived from it – using `new`).

Each balloon operates independently, but thanks to the prototype-based system, they share common behaviors while maintaining their own properties.

## 2.4 Joystick and Mouse

Instead of moving a sprite on its own, you can also link it to the joystick, gamepad, or arrow keys of the user. You do this by sending the message `controllable:` followed by a code. There are different control codes that you can send to an object. If you send code 1, the sprite will move in all directions, just like in a top-down game. If you send code 2, the player can only move vertically, like in a digital tennis game. Code 3 restricts movement to horizontal only. With code 4, the sprite behaves like a race car, where the left-right keys steer and the up key accelerates. You can combine these codes with other messages to create more complex control schemes. Here's a small "recipe book":

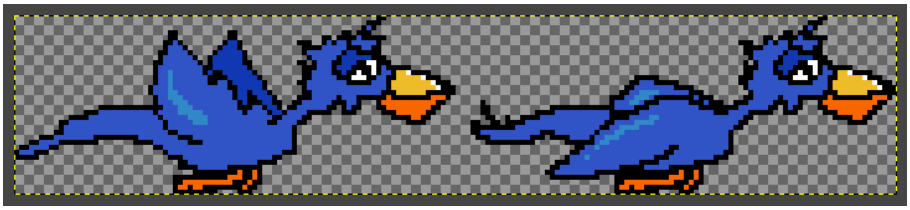
- **Frog crossing a busy road (top-down view):**
  controllable: 1.
- **Jumping rabbit (platformer):**
  controllable: 1, gravity: 1, jump-height: 2.
- **Hovering spaceship (2D shooter):**
  controllable: 1, gravity: 0.5.
- **Tennis (vertical movement only):**
  controllable: 2.
- **Shuffleboard (horizontal movement only):**
  controllable: 3.
- **Racing game (top-down):**
  controllable: 4.
- **Balloon race:**
  controllable: 1, fixate: True.

There are several messages you can use to make the controls more interesting. For example, the message `gravity:` makes your sprite move like in a platformer. With gravity set above 1, the sprite can also jump. You can adjust the `jump-height:` to your preference. The sprite is also automatically animated. If it has a nose, the nose will face left when you move left, and right when moving right. Without gravity, the nose points up when you move upward. With gravity (>= 1), the sprite "jumps" when you press up. With gravity between 0 and 1, the sprite floats, and the animation looks a bit different. If you don't want the sprite to rotate in any direction, send `fixate: True`, which is useful for things like a hot air balloon.

With the message `accelerate:`, you can further tweak the sprite's movement, for instance, making it "slide" across ice. You can slow the sprite down with `friction:`, as if it's walking through thick mud. You can even allow a sprite to walk through walls with `ghost: True`.

When you send `active: True`, the sprite will receive a signal whenever it collides with another sprite. For example, we can make a simple game where a balloon must avoid a fluttering bird.

To make the bird flutter in this mini-game, we'll create a sprite with two frames, and we can tell this sprite to act like a filmstrip that plays at a certain speed while moving:



```
bird reel: 2 speed: 10.
```

It's also possible to have an animation play automatically without the sprite moving, which is useful for something like a campfire. In that case, send the message `autoplay: True`.

The following code demonstrates how to make a little game where you control the hot air balloon and must avoid the bird:

```
>> m := Media new.
>> balloon := Image new: ['balloon.png'].
>> bird := Image new: ['bird.png'], reel: 2 speed: 10.
```

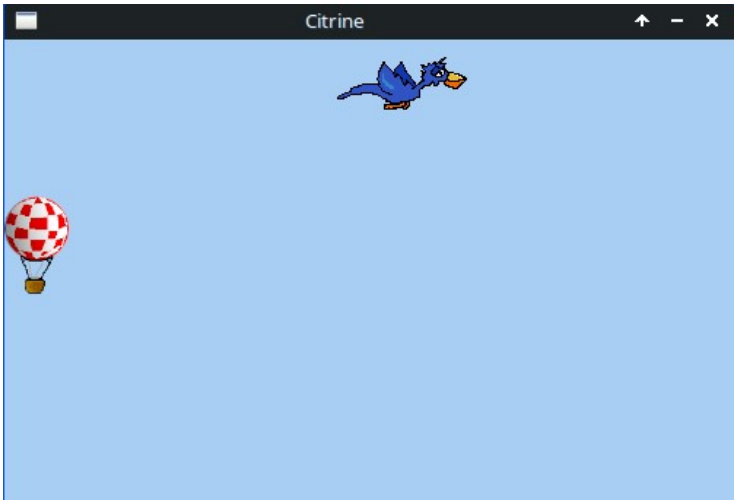
```
bird on: ['fly'] do: {
      >> right := Number between: 0 and: 500.
      >> height := Number between: 0 and: 100.
      self to-x: right y: height.
}.
```

```
bird on: ['destination'] do: { self fly. }.
```

```
balloon on: ['collision:'] do: { :other
      (other = bird) true: { Program end. }.
}.
```

```
m on: ['start'] do: {
      balloon
            fixate: True,
            active: True,
            controllable: 1,
            gravity: 0.2.
```

```
      bird
            x: 300 y: 10,
            gravity: 0.01,
            speed: 1,
            fly.
}.
m screen: ['sky.png'].
```




By sending the message `width:height:`, you set up a camera that follows the player in the game. To temporarily disable player controls (without affecting the camera), you send: `freeze: True`.

If you want to create a digital shooting gallery where the mouse cursor acts as the crosshair, you don't need to send any controllable message. When the mouse is clicked, an "active" sprite receives the message `click`. The media object itself receives `click-x:y:` for every click, which you can also use to detect button presses. This also handy for buttons in apps.

## 2.5 Drawing

To have a scoreboard on the screen where the player can see their current score. We simply create an image with text on it.

```
>> s := Image new: [‘score’].
```

And we add the text like this:

```
s write: 0.
```

Unfortunately, we get an error at this point. This happens because we haven't chosen a font, so the system doesn't know how the final result should appear. We can fix this by creating a new font object:

```
>> f := Font new
    source: ['Shortcake.ttf']
    size: 20.
```

In this case, we choose the included font ‘simple.ttf’ (an imaginary font) and set the font size to 20 points. Next, we link the font to the image:

```
s font: f.
```

If you want to make the text editable, allowing users to type directly into the image, you can set:

```
image editable: True.
```

This is useful for applications.

We can also change the color. There are a vast number of colors to choose from, but every color displayed on a screen is composed of three components. Think of them as three paint buckets: red, green, and blue. You can use between 0 and 255 units from each bucket to mix any color you want. For example, if you want bright green, use 255 units from the green bucket only. Want bright red? Take 255 units from the red bucket. Bright yellow? Use 255 units from both the red and green buckets. The Color object is your palette, allowing you to mix your perfect shade. For our scoreboard, let's use the color orange. We can create the color orange by creating a new color object:

```
>> orange := Color new red: 250 green: 150 blue: 0.
```

Next, we use ‘orange’ as ink to write on our scoreboard:

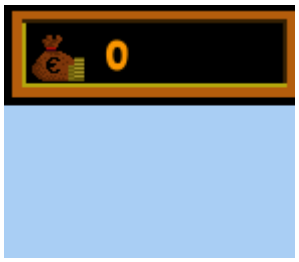
```
s ink: orange.
```

You can align the text with: `align-x: y:`.

Here is the complete code:

```
>> m := Media new.
>> f := Font new
    source: ['Shortcake.ttf']
    size: 20.
>> s := Image new: ['score.png'].
>> orange := Color new red: 250 green: 150 blue: 0.
s ink: orange, font: f, align-x: 50 y: 15, write: 0.

m screen: ['sky.png'].
```

Depending on your background image for the score and the font, you will something like this:



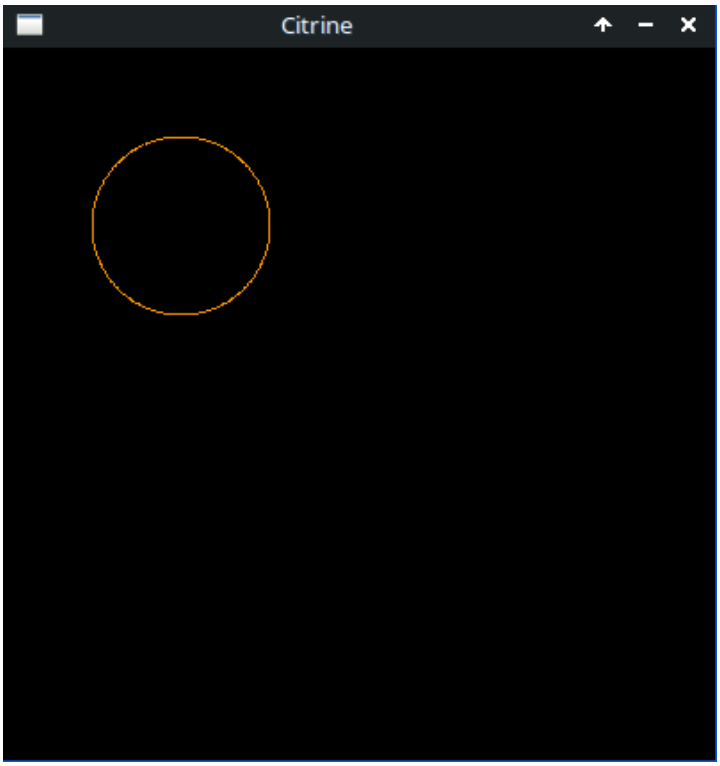

You can also draw individual points and lines on an image. To draw a point, do the following:

```
>> m := Media new.
>> orange := Color new red: 250 green: 150 blue: 0.
>> canvas := Image new: ['canvas.png'].
>> points := Sequence new.
points append: (Point new x: 10 y: 10).
canvas draw: points color: orange.
m screen: ['canvas.png'].
```

To draw something more interesting like a circle you can do:

```
>> m := Media new.
>> orange := Color new red: 250 green: 150 blue: 0.
>> canvas := Image new: ['canvas.png'].
>> points := Sequence new.
```

```
>> r := 50. # radius
>> c := 100. # center


{ :i
      points append: (
            Point new
            x: r * i cos + c
            y: r * i sin + c
      ).
} * 360.


m on: ['start'] do: {
      canvas draw: points color: orange.
}.


m screen: ['canvas.png'].
```




In the same way, you can also draw lines on an image. In this case, you need to specify the two points between which the line should be drawn:

```
>> line := Sequence new ;
     (Point new x: 0 y: 0) ;
     (Point new x: y…) ;
```

and so on...

Note that, although it’s technically allowed to start putting images on the screen and drawing things outside of the start-task of a screen, it’s often wiser to do so within the start-task due to timing issues. However, never declare variables inside a start-task for things that remain on the screen, we will discuss this in chapter 3.3 (scoping).

## 2.6 Audio

Citrine offers two Audio objects: Music and Sound. To play a piece of (background) music:

```
>> music := Music new: ['Vivaldi.mp3'].
music play.
```

The music will loop automatically. To rewind the music send the message `rewind`. To stop the music, send the message `silence`. Then there is the `Sound` object. This object is very similar but simpler and only support the play message. Sounds never loop.

## 2.7 Networking

With the Network object, which is part of Media, you can retrieve data from the internet and send data to the internet. You use the message `send:to:` for this. If you send `None`, you will only retrieve data from a webpage. Under the hood, an HTTP-GET request will be sent. If you send a text to a webpage, it will be processed as a POST request under the hood. For example, this is how you retrieve the content of Citrine's website:

```
>> m := Media new.

>> n := Network new.

>> response := n
        send: None
        to: ['https://www.citrine-lang.org'].

Out write: response, stop.
```

## 2.8 Files & Data

Instead of images or music, you can also use code from other files. To include another Citrine program as part of your program:

```
Program use: [‘/myprogram.ctr’].
```

This will cause the file ‘myprogram.ctr’ to be loaded and inserted at this point.

Citrine supports some basic functions to read and write files, here is an example:

```
>> f := File new: (Path /tmp: ['test.txt']).
f write: ['test'].
>> q := File new: (Path /tmp: ['test.txt']).
Out write: q read, stop.
```

Instead of loading code, images or music from files you can also add them to a data package and have your program retrieve them from this package as well. The advantage of a data package that your resources are bundled in one, opaque package (so people can’t see the images or listen to the music files separately). Data packages are also required for exports to other platforms, mobile and game consoles.

With the Package object, you can place files into a data package. You can then link this package to Media, and from that moment, all files will be retrieved from your data package:

```
>> data := Package new: ['datapackage'].
data add: ['rabbit.png'].
```

To use the data package:

```
Media link: ['datapackage'].
```

# 3 Advanced

In this chapter, you enter Citrine's version of a ‘funhouse’. As in life, everything eventually needs to come together and be woven into one whole. But then, out of nowhere, come the exceptions, the edge cases, and the trapdoors. Like any other programming language, Citrine has its own collection of quirks. It's crucial to be aware of these oddities, or you might end up quite confused!

There's one more thing to keep in mind: Citrine is a programming language without ‘guardrails’. This is a feature, not a bug. Some languages try to protect programmers from themselves, but Citrine intentionally does not. Citrine allows you to take risks and can let you ‘fly off the rails’ in spectacular ways. On the flip side, it gives you the freedom to do things other languages simply won’t allow. You have total freedom—no limits on your creativity! But also, no safety net.

## 3.1 Copying

First, let me give you the formula for total chaos:

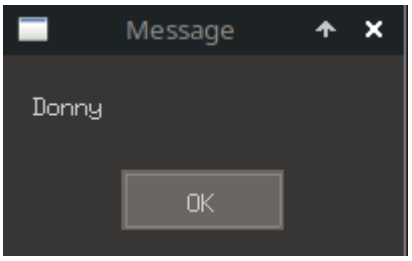
```
True := False.
```

Yep, this is possible in Citrine (and as far as I know, in no other language). If you do this, you're guaranteed to break everything. In short, it's completely unpredictable what your program will do after running this code. No clue. The reason this works is simple: := never makes a copy. In other languages, assigning to a variable sometimes creates a copy and sometimes doesn’t—it depends. That can get complicated. In Citrine, we keep it simple: everything is a reference. There's only one True object and one False object, and all other objects decide what to do based on these two.

Using :=, we store objects under a specific name. You can even store an object under multiple names. But remember, this doesn't create a copy. Here's an example:

```
>> sheep := ['Dolly'].
>> clone := sheep.

clone replace: ['l'] with: ['n'].

Media show: sheep.
```

Result:




You might expect to see **Dolly**, but both names point to the same object. So when we replaced the ‘l’ with the ‘n’ we did that to both sheep and clone, because they are the same object saved under a different name!

To make an actual copy of an object, you need to explicitly ask for it:

```
>> sheep := ['Dolly'].
>> clone := sheep copy.

clone replace: ['l'] with: ['n'].

Media show: sheep.
```

The message ‘copy’ creates a copy of the text. You can modify it without changing the original. `Copy` works for texts, numbers, True/False objects, times, lists, and sequences. For other objects, you’ll need to write your own copy method. Note that when copying a list, the elements in the list are not copied!

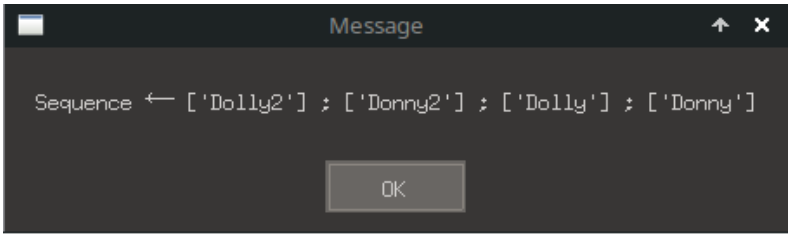
```
>> sheep := Sequence new ; ['Dolly'] ; ['Donny'].
>> clones := sheep copy.


clones each: { :number :sheep
  sheep append: ['2'].
}.

Media show: clones + sheep.
```

Output:


If you want to copy the elements inside a list as well, that's called a ‘deep copy’. You can program this yourself, like so:

```
Sequence on: ['copy'] do: {
  >> deepcopy := Sequence new.
  self each: { :number :element
    deepcopy append: element copy.
  }.
  <- deepcopy.
}.

>> sheep := Sequence new ; ['Dolly'] ; ['Donny'].
>> clones := sheep copy.

clones each: { :number :sheep
  sheep append: ['2'].
}.

Media show: clones + sheep.
```

Output:




Without the `copy`, you'd end up with both lists showing the second version. This also works for nested lists.

It's important to note that while a copy of an object looks like the original, it’s never the same. The parent object has a message `equal:` that compares object identities. Check out this example:

```
>> a := 1.
>> b := a copy.
>> c := a.

Media
      show: ( a = b ),
      show: ( a = c ),
      show: ( a equals: b ),
      show: ( a equals: c).
```

Output:

```
True
True
False
True
```

In this case, `a = b` because `a` is a copy of `b`, and the `=` message compares object values—in this case, numbers. Since `a = 1` and `b = 1`, and `1 = 1`, the answer is True. The same goes for `c`. But when we use the `equals:` message, inherited from Object, we see a difference. Even though `c` is equal to `a` because they both reference the same object, `a` is not equal to `b`, since the copy is a physically different object in memory. To check if you have a reference to the exact same object in memory, use the `equals:` message.

## 3.2 Conversions

Sometimes you get a number back as a text object, for example if you read the input from user from an editable image. To convert a text into a number, you send the `number` message. From that point, you can send number-related messages.

In general, you can convert any object to a different type using these messages:

| **Message** | **Does...** |
|---|---|
| number | Returns a new Number object based on the recipient |
| text | Returns a new Text object based on the recipient |
| bool | Returns True or False based on the recipient |

The following rules apply:

| **Message** | **Recipient** | | | | |
|---|---|---|---|---|---|
| | *None* | *Boolean* | *Number* | *Text* | *Other* |
| bool | False | - | 0 → False<br>else: True | True | True |
| number | 0 | True → 1 False → 0 | - | Tries to interpret text as number. | 1 |
| text | [‘None’] | True → [‘True’]<br>False → [‘False’] | Textual representation. | - | [‘Object’] or textual representation as defined by recipient. |

Citrine converts objects internally when needed. When printing a sequence to the screen, for example, Citrine internally sends the `text` message to the sequence. You can take advantage of this. Suppose you want to print a sequence as a comma-separated list:

```
>> sum := Sequence new ; 1 ; 2 ; 3.

sum on: ['text'] do: {
        >> summed-up := 0.
        self each: { :i :element
                summed-up add: element.
        }.
        <- summed-up text.
}.

Media show: sum.
```

This gives 6 as a result.

## 3.3 Dynamic Scope

When a task needs a variable but can't find it, it will check the variables of the task that called it, and continue checking up the chain of tasks. This search process goes on until the variable is found. If the variable is still not found, an error message will appear. This method of searching for a variable is called *Dynamic Scoping*.

To help you understand this concept, let’s do a quiz called *Bob or Alice*. For each example, try to predict whose name will appear on the screen!

```
>> name := [‘Bob’].

{
  media show: name.
} start.
```

In this example, the name ‘Bob’ will appear. The task doesn’t have its own `name` variable, so it looks for the variable in the calling task, which is the main program.

```
>> name := [‘Bob’].

{
      name := [‘Alice’]
      {
            media show: name.
      } start.

} start.
```

What name will be shown? In this case, it’s ‘Alice’. The outer task overwrites the value of `name` from ‘Bob’ to ‘Alice’. Now, let’s modify the example a bit:

```
>> name := [‘Bob’].

{
      >> name := [‘Alice’]

} start.

media show: name.
```

So, is it Bob or Alice? This time, it’s ‘Bob’! Even though `name` was changed to ‘Alice’ inside the task, that change is discarded once the task ends.

```
>> name := [‘Bob’].

{
      name := [‘Alice’]

} start.
```

`media show: name.`

Yields ‘Alice’ though, because now, name references the global variable outside the task.

Since the display command is outside the task, it doesn’t see the change. Let’s make this even clearer—can you predict the following?

```
>> show := { Media show: name }.

>> change := { name := [‘Bob’] }.

{
    >> name := [‘Alice’].

    change start.
    show start.


} start.
```

The outcome is ‘Bob’. By following the sequence of tasks, you can see that the variable `name` is ultimately set to ‘Bob’ before the `show` task is called.

Citrine also has an automatic cleanup feature (*garbage collector*). Once a task finishes, all variables declared within that task are cleared. For example:

```
{
    >> name := [‘Alice’].

} start.
```

So, in this case, the variable name will be ‘forgotten’ after the task ends.

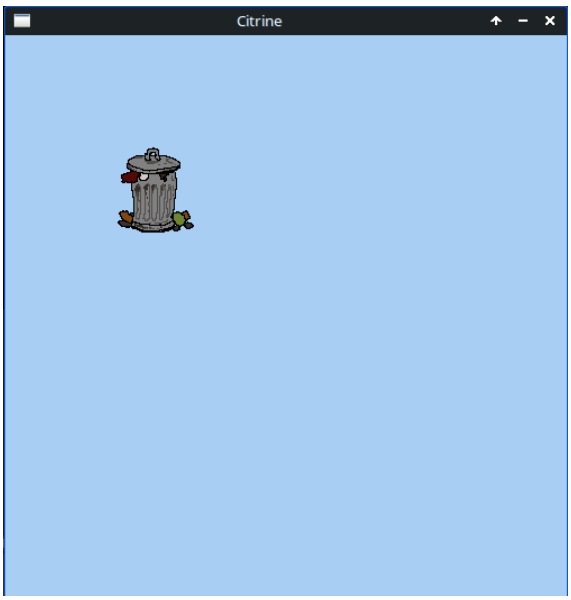
```
Person on: [‘name’] do: {
    >> name := [‘Alice’].
}.
```

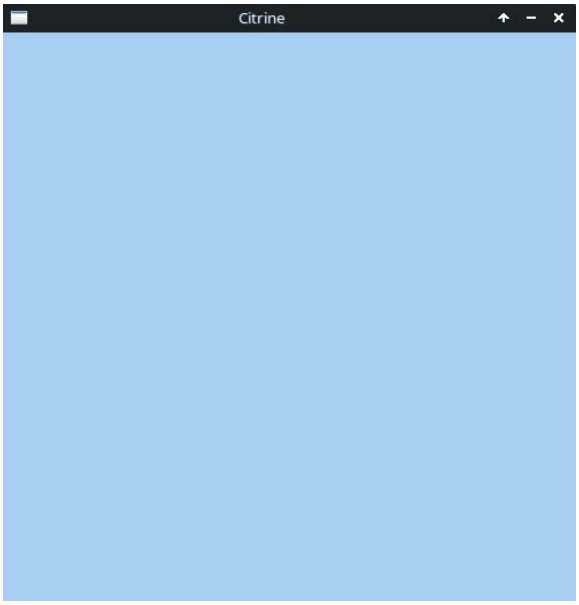
In this case, the same rule applies. If you want to prevent the variable from getting lost, you can attach it to the object by making it a property:

```
Person on: [‘name’] do: {
    own name := [‘Alice’].
}.
```

You can observe the result of the garbage collector with the following example:

```
>> m := Media new.

m on: ['start'] do: {
      >> garbage := Image new: ['garbage.png'].
      garbage x: 100 y: 100.
}.

m on: ['step'] do: {
      >> x := Sequence new.
      x fill: 100000 with: ['xxx'].
}.

m screen: ['background.png'].
```




In this example we draw a garbage can on the screen. We also add a task to be executed on every ‘step’. This is needed because by default, the garbage collector will only become active if memory gets cluttered up. So after a while the garbage collector will clean up the garbage bin on the screen because the variable `garbage` will be ‘forgotten’. Another way to activate the garbage collector is to set it to hyper active:

```
Program memory-management: 4.
```

If you add this line on top of the program, you’ll see the garbage can will be removed instantly. You can also trigger the garbage collector manually:

```
>> m := Media new.

m on: ['start'] do: {
      >> garbage := Image new: ['garbage.png'].
      garbage x: 100 y: 100.
      m timer: 1 after: 3000.
}.

m on: ['timer:'] do: {
      Program clean-up.
}.

m timer:1 after:1000, screen: ['achtergrond.png'].
```

There are 3 memory management modes: 0 means no cleaning up at all, 1 (default) only cleans when about 80% of the allocated memory (10 MB) is used up, 4 means clean-up after every cycle. There are also some experimental memory management settings. You can change the default memory allocation using the environment variable CITRINE_MEMORY_LIMIT_MB.

## 3.4 Unknown messages

Suppose we want to hide all the images in a game. We've put all the images in a collection called `junk`. We can do this as follows:

```
junk each: { :index :image
    image visible: False.
}.
```

But what if we want to send different messages to all the images in `junk`? It would be convenient if we could send a message to the entire collection and have that message forwarded to each image in it. Good news—it's possible! When you call a method that doesn't exist on an object, the message will eventually be caught by the following block:

```
junk on: [‘message:’] do: {
}.
```

If you send a message with one argument, it's caught like this:

```
junk on: [‘message:argument:’] do: {
}.
```

For two arguments:

```
junk on: [‘message:argument:argument:’] do: {
}.
```

And for three arguments:

```
junk on: [‘message:argument:argument:argument:’] do: {
}.
```

This way, we can catch all 'unknown' messages for which no method exists.

Once we've caught such a message, you can forward it like this:

```
junk on: [‘message:argument:’] do: { :message :argument
    self each: { :index :object
        object
            message: message
            arguments: (Sequence new ; argument).
    }.
}.
```

Using the message `message:arguments`, you can forward a message to another object. You can now forward a message like ‘visible’ to all images in the junk collection:

```
junk visible: False.
```

To make them reappear:

```
junk visible: True.
```

You can also send other messages with one argument:

```
junk speed: 1.
```

In all cases, the message you send is forwarded to all objects in the `junk` collection.

## 3.5 Chain Mode

Suppose you have a sequence:

```
>> x := Sequence new 1 ; 2 ; 3.
```

You want to remove the first and last element. In that case, you might want to write:

```
x
shift
pop.
```

Unfortunately, this won't work as expected. The message `shift` will return the first element of the sequence, and that becomes the receiver of the `pop` message. So, instead of sending `pop` to `x`, you're sending it to `2`.

One possible solution is to split this into two separate statements:

```
x shift.
x pop.
```

However, that can be inconvenient, especially when you want to remove more than two elements. For these types of situations, Citrine offers *chain mode*. In chain mode, the responses from objects are ignored, and the original receiving object is always returned. You start chain mode by sending the `do` message to an object and end it with the `done` message. In our example, you could apply the custom structure as follows:

```
x
do
    shift
    pop
done.
```

## 3.6 Qualifiers

In Citrine, you can attach a *qualifier* to a number. A qualifier is a word or label that describes what the number represents. For example, you can attach "apples" to the number 6, like this: `6 apples`.

Any word that follows a number is treated as a qualifier.

```
amount := 6 coins.
```

Here, `coins` is the *qualifier* for the number 6.

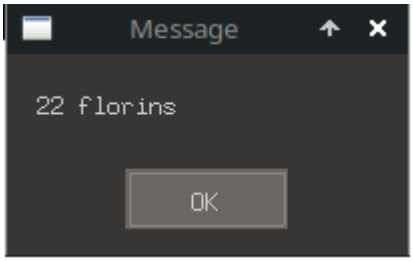
To get the qualifier from a number, use the `qualifier` message:

```
amount qualifier.
```

This will return the qualifier, which is stored as a *Text* object alongside the number. When you print the number, the qualifier is displayed after the number.

Qualifiers can be useful for working with different units, like currencies. For example, let's say you want to add amounts in different currencies, like florins and ducats. In the following example, 1 ducat is worth 5 florins. When adding these amounts, the program checks the qualifier to make sure the conversion is correct.

```
>> m := Media new.

Number on: ['+'] do: { :n

    n qualifier = ['ducats'] true: {
        n multiply-by: 5.
    }.

    self add: n.
}.


m show: 7 florins + 3 ducats.
```

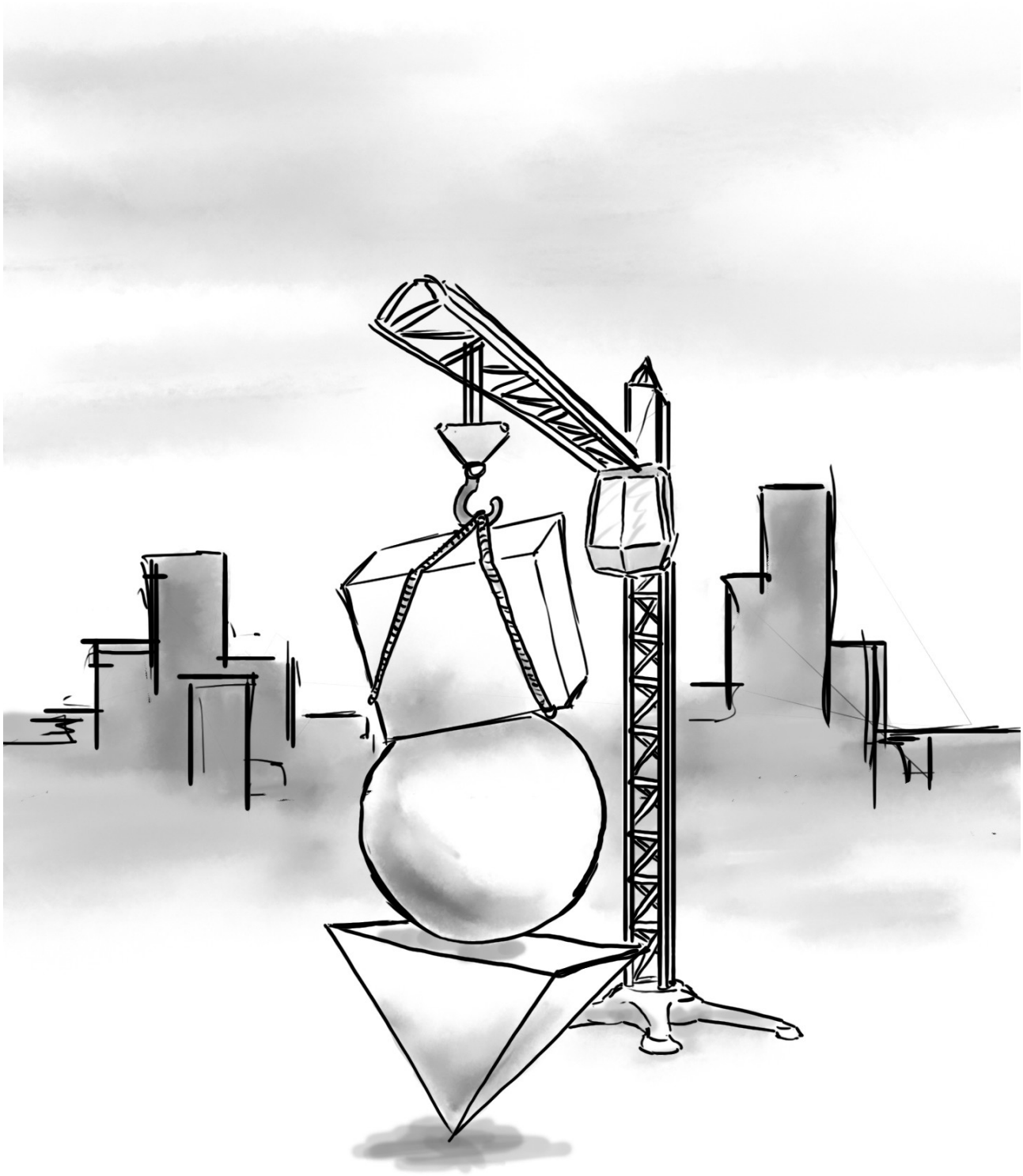


In this code, the `+` message is customized to handle different currencies. When adding `7 florins` and `3 ducats`, the program checks if the qualifier is "ducats". If it is, the program converts `3 ducats` to `15 florins` (because 1 ducat = 5 florins) before adding the amounts. This way we end up with 22 florins because 15 + 7 = 22. As you can see the word ‘florins’ is also added to the number upon displaying.

You can also explicitly set a qualifier for a number using the `qualifier:` message:

```
x := 7 qualifier: [‘ducats’].
```

Now, the number 7 is associated with the qualifier "ducats".



# 4 Extensions

There are several ways to extend Citrine's functionality. You can install plugins, call external programs through the command line, or invoke routines from external software libraries. The latter is called FFI (Foreign Function Interface) and is a powerful feature of Citrine, though it’s also quite complex. In this chapter, I’ll discuss all three methods. However, I’ll only cover the Citrine side of things, meaning this is just half the story. You’ll need to consult the relevant plugin’s documentation (for plugins), your operating system’s manual (for the command line), or the external software library’s documentation (for FFI) for the rest.

Unlike previous chapters, this isn’t a complete guide. You should think of this chapter as a springboard to other systems. Thanks to Citrine's open architecture, the possibilities are virtually endless. With FFI, in particular, you can build almost anything you want. Want to write a complete office suite for the desktop? You can do that with FFI. On the flip side, FFI isn’t easy, and you’ll need to do a lot of research, particularly by digging through the documentation of the systems you want to interact with through FFI.

Despite its complexity, FFI is a highly useful tool. It’s simply impossible to include every functionality imaginable directly in Citrine. That’s not feasible. But with FFI, you can leverage systems that have been developed over the years. From processing PDF files to using GUI systems like GTK, FFI opens the door to the entire ecosystem.

## 4.1 Plugins

The `Media` object is an example of a plugin. In addition to the media plugin, you can install other plugins for Citrine as well. You do this by copying the corresponding files (ending in `.dll` or `.so`) into the 'mods' directory. The full path of a plugin file always follows this format:

```
mods/<ObjectName>/libctr<ObjectName>.<extension>
```

So, if we wanted to install the fictional plugin 'Aquarium', we would need the following file in 'mods':

```
mods/aquarium/libctraquarium.so (Linux)
mods/aquarium/libctraquarium.dll (Windows)
```

We can then load the object in our code like this:

```
>> a := Aquarium new.
```

Just like with the media plugin, we simply send a message to the object to use it.

## 4.2 Command Line

In Citrine, you can execute system commands by sending the `os:` message to the Program object (OS stands for operating system). This allows you to copy or move files, run other software, or perform specific tasks.

For example, if you want to list the files in the current directory on a Linux system, you can do it like this:

```
Program os: Instruction ls.
```

The code above runs the `ls` command in the system shell and returns the result as text. You can also write it like this:

```
Program os: [‘ls’].
```

The `Instruction` object is a simple helper you can use instead of a `Text` object. You can specify commands for the Program as either text or an `Instruction` object.

## 4.3 FFI

FFI stands for *Foreign Function Interface*. It allows you to use functionality written by others in different programming languages, provided through DLL files, SO files, or Dylib files. These could be a variety of functions, and there's a vast range of available functionality through this method.

Let's start with an example:

```
>> media := Media new.
media link: (
    Sequence new ;
    ['/usr/lib/x86_64-linux-gnu/libc.so.6'] ;
    ['printf'] ;
    ( Sequence new ; ['pointer'] ; ['int'] ) ;
    ['void'] ;
    ['Printf'] ;
    ['template:number:']
).
>> s := Blob utf8: ['FFI has %d letters.\n'].
Printf template: s number: 3.
s free.
```

The result of this code is that you'll see the following on the command line:

```
FFI has 3 letters.
```

Now, you're probably thinking, "That's a lot of code for something so simple." I mean, couldn't we just solve this with a `Media show` command? The answer is a resounding yes! But this is meant as an illustrative example. FFI is usually used for more complex tasks, but those don't make good examples, so I chose something trivial.

The `link` method might look familiar from chapter 2. It has multiple purposes. The first is to link external resources, such as images or music from a data package. The second purpose, which is a bit more complex, allows you to link functions from external sources (like DLLs or SO files). The arguments for this are passed as a sequence with the following elements:

1. The DLL or SO file you want to use.
2. The function in that file you want to link.
3. A sequence with the names of the data types of the function's arguments.
4. The name of the function's return type.
5. The name of the object you want to link this function to (if it doesn't exist, it will be created automatically).
6. The message that this function should be linked to.

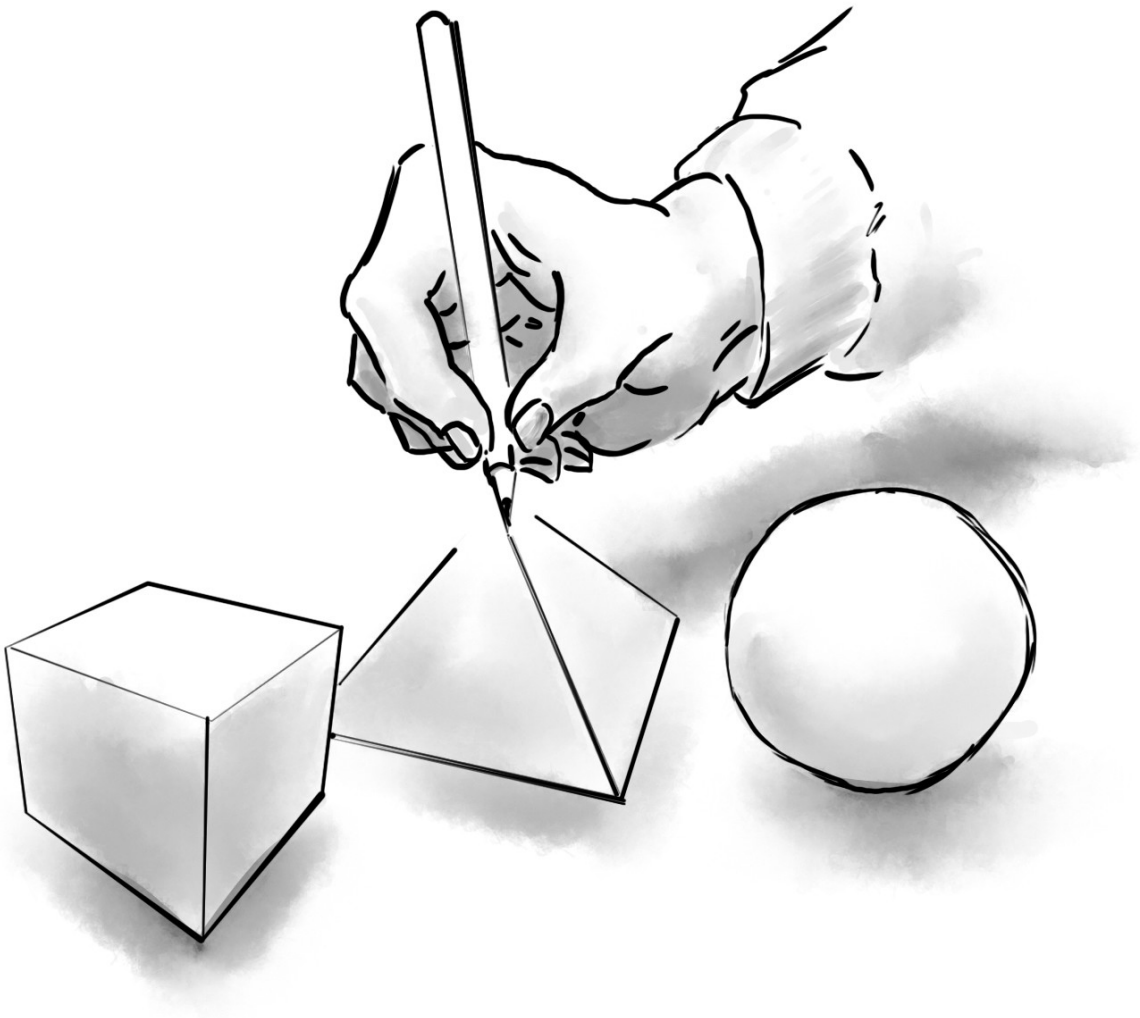
In the example above, we want to link the `printf` function from `libc.so`. You can find the data type names in the documentation of the software you are linking to. The available types are:

- `void`, `pointer`, `float`, `double`, `int`, `uint`, `char`, `uchar`, `intX`, and `uintX`, where X can be 8, 16, 32, or 64.

These types refer to the number of bytes required to store the data. For `printf`, we are linking to a new object called `Printf` and the message `template:number:`. You translate the external function into the Citrine dialect before using it. The message `template:number:` expects a buffer with the template text as its first parameter. We create this buffer using a `Blob` object, which allows you to manually allocate memory. You are responsible for freeing this memory afterward with the `free` message.

You can fill a memory blob in various ways. In our example, we fill it with text, so we use the `utf8:` message (UTF-8 is an encoding to convert text into bytes). You can also fill a blob with `fill:`, passing a sequence of byte values. To read the contents of a blob, use `from:length:`. You'll get the bytes back as a sequence. You can even create a C-struct with a Blob using the `struct:` message, passing a sequence of C types. This may be necessary when calling a C function in an external software library that expects a pointer to a struct.

# Nederlands



# 1 Berichten

Deze handleiding is voor zowel beginnende als gevorderde programmeurs. Ervaren programmeurs voor wie Citrine een nieuwe programmeertaal is, zullen waarschijnlijk her en der stukken kunnen overslaan.

Als je gaat beginnen met programmeren sta je op het punt om een zeer avontuurlijke reis te gaan maken. Onderweg zul je allerlei nieuwe ervaringen opdoen. Gek genoeg ontdek je ook veel over jezelf! Je zult er namelijk achter komen dat je helemaal niet zo logisch denkt als je had gehoopt. Ook zul je veel ‘gaten’ ontdekken in je gedachtes. Iedere programmeur leert iedere dag weer dat de wereld vele malen complexer in elkaar zit dan hij of zij had verwacht. Tegelijkertijd is het omgaan met die complexiteit ook een soort kunst op zich. Soms zijn de oplossingen die in stukjes computercode besloten liggen zo mooi en elegant, dat je het bijna een kunstwerk kunt noemen!

Persoonlijk vind ik de meeste computerboeken nogal saai. Ik kom er gewoon niet doorheen. Na een aantal bladzijden ga ik gewoon dingen uitproberen en spring ik van hoofdstuk naar hoofdstuk en weer terug. De meeste computerboeken zijn zo taai omdat de schrijvers ervan alles helemaal uitkauwen voor je. Auteurs van computerboeken zijn als de dood dat ze een detail overslaan. Dit boekje werkt anders. Zoals gezegd vergelijk ik het leren van programmeren, of het leren van een nieuwe programmeertaal (voor de gevorderde programmeurs onder ons) met het maken van een reis. Die reis maak je niet door dit boekje te lezen. Met dit boekje vul je je spreekwoordelijke rugzak ter voorbereiding op die reis. De echte reis gaat pas beginnen na de laatste bladzijde (of misschien wel eerder). Die reis ga ik niet voor je maken. Dat moet je zelf doen. Ik leg je in deze handleiding alleen de basisbeginselen uit, ik laat je zien wat er zoal mogelijk is en ik geef je aanwijzingen hoe je verder je weg kunt gaan vinden. Maar uiteindelijk moet je de reis zelf gaan maken. Je moet vooral zelf gaan ontdekken en experimenteren. Soms kan dat lastig en frusterend zijn. Maak het jezelf niet te moeilijk. Als het even niet lukt, dan stop je er gewoon mee voor die dag. De reis loopt niet weg. Je kunt elke dag besluiten je weg te vervolgen. En, ik beloof je een ding, die weg zit vol met verassingen!

## 1.1 Programma’s

Met Citrine kun je *apps*, *games* en zelfs serieuze bedrijfsapplicaties maken—dit zijn allemaal *computerprogramma's*. Om een computerprogramma te maken, moet je de computer vertellen wat hij moet doen. Dit proces heet *programmeren*. Computerprogramma's worden geschreven in een *programmeertaal*, en Citrine is zo'n taal. Een programmeertaal definieert de regels voor het schrijven van een programma. Je schrijft instructies in een tekstbestand met behulp van een eenvoudige teksteditor zoals Kladblok. Je kunt je programma starten door het naar het Citrine-pictogram te slepen. Een andere manier om je programma uit te voeren is via de terminal (opdrachtregel). Open deze en typ:

```
ctrnl <programma>
```

Hierbij vervang je <programma> door de naam van het programma dat je geschreven hebt.

De computer leest een Citrine-programma van boven naar beneden en van links naar rechts. Net als gewone taal dus. Het Citrine-programma bestaat uit zinnen. Elke zin eindigt met een punt. In een zin beschrijf je wat de computer moet doen. Zinnen die beginnen met een hekje (#) worden overgeslagen. Dat kun je gebruiken om een opmerking toe te voegen aan je programma.

## 1.2 Berichten

Gelukkig hoef je niet helemaal zelf het wiel opnieuw uit te vinden. Er is al heel veel geprogrammeerd voor je computer. Van al deze functionaliteiten kun je gebruikmaken. Je kunt er gewoon op voortborduren en zo je eigen ding maken.

Citrine is een puur objectgeoriënteerde taal. Dit betekent onder andere dat al die ingebouwde functionaliteiten worden aangeboden in de vorm van objecten. Als je de computer iets wilt laten doen, moet je een *boodschap* (ook wel *bericht* genoemd) sturen naar een object. Bijvoorbeeld:

```
Moment maand.
```

Dit is een korte zin in Citrine. In dit voorbeeld sturen we de boodschap 'maand' naar het object 'Moment'. Het object hier is 'Moment' en de boodschap is 'maand'. In het algemeen, wanneer je een boodschap naar een object wilt sturen, schrijf je het als volgt:

**<object> <bericht>**

In plaats van <object> kun je elk willekeurig object invullen dat Citrine kent, en op de plek van <boodschap> zet je de boodschap die je wilt sturen. In ons geval sturen we de boodschap 'maand' naar het object 'Moment'.

Het gevolg is dat we aan het 'Moment'-object vragen wat de huidige maand is. Het 'Moment'-object bevat functionaliteit voor het werken met datums en tijden. In dit geval kan het bijvoorbeeld het nummer 8 teruggeven, dat staat voor augustus (de 8ste maand). Omdat we deze waarde niet direct gebruiken, zal het vergeten worden. Om deze waarde later in het programma te gebruiken, moeten we het bewaren.

Stel dat we het maandnummer willen opslaan als 'm'. Eerst moeten we ruimte reserveren in het geheugen van de computer voor 'm' door het *declaratiesymbool* (>>) te gebruiken. Als je eenmaal een plekje hebt gereserveerd voor ‘m’ hoef je dat daarna niet nogmaals te doen. Daarna wijzen we het resultaat toe aan 'm' met behulp van het *toewijzingssymbool* (:=). Dat ziet er zo uit:

```
>> m := Moment maand.
```

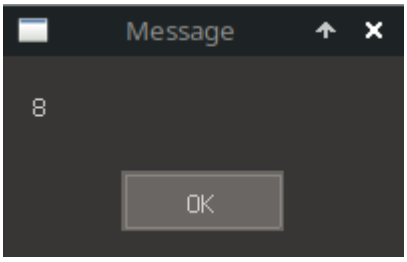
Nu hebben we het maandnummer (bijvoorbeeld 8) opgeslagen onder de naam 'm'. Dit noemen we een variabele. Geheugenruimte reserveren wordt *declareren* genoemd. Nu we het maandnummer onder 'm' hebben opgeslagen, kunnen we het later in het programma gebruiken. Bijvoorbeeld, als we het maandnummer op het scherm willen tonen, gebruiken we een ander object, het 'Media'-

object. We sturen de boodschap `toon:` naar het 'Media'-object, en als extra informatie geven we op wat we willen tonen, namelijk 'm', het maandnummer. Deze extra informatie noemen we een *argument*.

We sluiten de zin af met een punt:

```
>> m := Moment maand.
Media toon: m.
```

Dit werkt en toont het maandnummer in een venster zoals hieronder:




In plaats van een maandnummer kunnen we ook een willekeurig getal tonen. Om een willekeurig getal te krijgen, kun je de boodschap `tussen:en:` sturen naar het 'Getal'-object. In dit geval moet je twee argumenten opgeven: de onder- en bovengrens waarbinnen het willekeurige getal moet vallen. Stel dat je een willekeurig getal tussen 1 en 10 wilt, dan schrijf je:

```
>> x := Getal tussen: 1 en: 10.
Media toon: x.
```

Elke keer dat je dit programma uitvoert, toont het een willekeurig getal. Dit betekent niet dat je elke keer een ander getal ziet – het is mogelijk dat hetzelfde getal meerdere keren verschijnt. Je kunt alleen niet voorspellen welk getal er zal komen.

Technisch gezien hoeven we het willekeurige getal niet eerst op te slaan om het te tonen. We kunnen de twee regels code combineren in één regel. In dat geval plaatsen we de uitdrukking die het willekeurige getal genereert tussen boogjes na de boodschap `toon:`

```
Media toon: (Getal tussen: 1 en: 10).
```

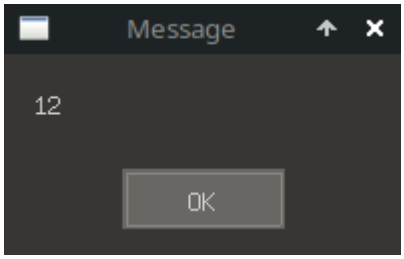
Het deel tussen de boogjes wordt eerst uitgevoerd (als je meerdere boogjes in elkaar gebruikt leest de computer van binnen naar buiten). We kunnen ook tekst op het scherm tonen (in plaats van getallen). Het is traditie om te beginnen met het bericht "Hello World”. In Citrine schrijf je dit als:

```
Media toon: [‘Hello World!’].
```

Omdat we de tekst in het argument willen onderscheiden van de code zelf, moeten we aangeven dat dit letterlijke tekst is (en dus geen opdracht voor Citrine). Anders raakt de computer in de war. Het weet dan niet wat code is en wat letterlijke tekst is. De oplossing is om alle letterlijke tekst tussen [‘…’] te plaatsen.

Zonder dat je het doorhad, heb je al nieuwe objecten gemaakt. Elke keer dat Citrine een getal (1, 2, 3…) of tekst ([‘Hello World’]) tegenkomt, maakt het automatisch een object achter de schermen. Dit betekent dat je ook boodschappen kunt sturen naar een getal of een stuk tekst. Het volgende programma toont het aantal letters in 'Hello World':

```
Media toon: [‘Hello World!’] lengte.
```




Hier hebben we twee boodschappen. Eerst sturen we de boodschap `lengte` naar het tekstobject [‘Hello World’], en daarna sturen we de boodschap `toon:` naar het 'Media'-object met het resultaat van de vorige operatie (dat wil zeggen, het aantal letters). Hoe bepaalt Citrine de volgorde van uitvoering?

Zoals eerder vermeld, leest Citrine van links naar rechts. Zinsdelen tussen boogjes worden eerst gelezen. Boodschappen zonder argumenten worden als eerste uitgevoerd, gevolgd door boodschappen met één argument die geen dubbele punt hebben (ik zal zo een voorbeeld geven). Ten slotte worden boodschappen met dubbele punten en één of meer argumenten uitgevoerd.

De boodschap 'lengte' geeft een getal terug. We kunnen ook een boodschap naar dit getal sturen, zoals + 1. Dit is een boodschap die één argument heeft en geen dubbele punt gebruikt. De regel is dat voor boodschappen die bestaan uit één teken en één argument, je de dubbele punt weglaat. In de praktijk zijn dit vaak rekenkundige bewerkingen zoals + (optellen), – (aftrekken), / (delen) en * (vermenigvuldigen).

```
Media toon: [‘Hello World!’] lengte + 1.
```

In het bovenstaande voorbeeld tonen we de lengte plus 1. Eerst wordt de boodschap 'lengte' naar de tekst 'Hello World' gestuurd, wat 12 oplevert. Vervolgens tellen we er 1 bij op. Boodschappen zonder dubbele punten worden uitgevoerd na boodschappen zonder argumenten, dus de volgorde gaat zo: eerst lengte dan optellen. Ten slotte wordt de boodschap met de dubbele punt ('toon:') uitgevoerd en zien we 13 op het scherm.

**Let op:** je moet een spatie voor en na de + gebruiken.

**Let op:** Getallen in Citrine/NL schrijf je op Nederlandse wijze. Dus anderhalf schrijf je als 1,5 en duizend als 1.000.

## 1.3 Taken en Lussen

Het is mogelijk om meerdere berichten aaneen te rijgen. Bijvoorbeeld, als we een aftelling op het scherm willen tonen (3...2...1...0), kunnen we dit doen door meerdere toon-berichten te sturen:

```
Media toon: 3.
Media toon: 2.
Media toon: 1.
Media toon: 0.
```

Maar er is een makkelijkere manier. Als je meerdere berichten naar hetzelfde object wilt sturen, kun je ze aan elkaar koppelen. Als een bericht een argument heeft, zoals bij `toon:`, moet je ze scheiden met een komma, anders raakt Citrine in de war en denkt het dat je het bericht ‘toon:toon:toon:toon:’ wilt sturen. Maar dat is niet zo. Je wilt het bericht ‘toon:’ 4 keer sturen, dus je scheid de berichten met een komma.

```
Media toon: 3, toon: 2, toon: 1, toon: 0.
```

Voor de leesbaarheid mag je dit trouwens ook over meerder regels verspreiden:

```
Media toon: 3,
      toon: 2,
      toon: 1,
      toon: 0.
```

We kunnen dit nog verder vereenvoudigen. In plaats van handmatig af te tellen, kunnen we er een taak van maken. Je kunt een reeks instructies omzetten in een taak door ze tussen {...} te plaatsen. We kunnen de aftelling bijvoorbeeld omzetten in een taak zoals dit:

```
{ Media toon: 3. }.
```

Achter de schermen maakt Citrine een taakobject van de zinnen binnen {...}. Je kunt ook berichten naar dit object sturen. Om een taak te starten, stuur je het bericht `start`.

```
{ … } start.
```

Als je  * 3 stuurt, wordt de taak drie keer herhaald. Een taak die op deze manier herhaald wordt, noemen we een *lus*. Maar hoe weten we in welke ronde (*iteratie*) van de lus we zitten? Zijn we in de eerste, tweede of derde ronde?

Wanneer je een bericht naar een taak stuurt, wordt relevante informatie, zoals het huidige *iteratienummer*, als een *parameter* meegegeven. Parameters worden aan het begin van de taak geplaatst en worden gebruikt om informatie van buiten de taak op te slaan. Deze parameters zijn alleen beschikbaar binnen de taak. Als je de taak start met * 3, wordt het huidige iteratienummer opgeslagen in *i*. Het iteratienummer wordt doorgegeven als de parameter *i*. Je mag de naam van de parameter zelf kiezen, je mag deze dus ook *ronde*, *index* of *k* noemen, of iets anders wat je zelf verzint.

```
{ :i
      Media toon: i.
} * 3.
```

Dit zal 1, 2, 3 tonen. Maar we willen eigenlijk een aftelling... 3, 2, 1, 0. Dat betekent dat we 4 iteraties nodig hebben. Bovendien moeten we in plaats van omhoog te tellen, juist omlaag tellen. Tijdens de eerste iteratie is *i* gelijk aan 1, en om 3 te tonen moeten we 1 van 4 aftrekken, wat 4 - 1 geeft. In de tweede iteratie is *i* gelijk aan 2, dus doen we 4 - 2, enzovoort. In het algemeen kunnen we zeggen dat 4 - *i* altijd het juiste aftelnummer zal geven:

```
{ :i
      Media toon: 4 - i.
} * 4.
```

Dit zal 3, 2, 1, 0 tonen zoals verwacht.

## 1.4 Vertakkingen

In plaats van aftellen naar 0, kunnen we ook tellen van 1 naar 20:

```
{ :i
      Media toon: i.
} * 20.
```

Stel dat we het ongeluksgetal 13 willen overslaan, hoe doen we dat? In dat geval sturen we het bericht '!=:' naar i met als argument 13.  De symbolen != betekenen 'is niet gelijk aan'. Dat is van oudsher zo gegroeid.

```
i !=: 13
```

Op die manier vragen we aan i of het getal gelijk is aan 13. Als dit zo is krijgen als antwoord *Ja* terug. Als het niet zo is krijgen we als antwoord *Nee* terug. Ook *Ja* en *Nee* zijn weer objecten (*booleans*). En ook naar deze objecten kunnen we berichten sturen. Een goed voorbeeld van zo'n bericht is `ja:`, dat bericht heeft als argument een taak. Het Ja-object voert de taak welke als argument wordt meegegeven uit. Het Nee-object negeert deze taak. Voor het bericht `nee:` geldt precies het tegenovergestelde, het Nee-object voert de taak die je meestuurt uit, terwijl het Ja-object deze taak juist negeert. Met die truc kunnen we dus een taak onder bepaalde voorwaarden laten uitvoeren. In ons geval willen dus alleen het getal tonen als het niet gelijk is aan 13. Om te bepalen of een getal gelijk is aan een ander getal gebruiken we '=', om te bepalen of het ongelijk is gebruiken we '!=:'. Omdat '=' uit slechts een enkel teken bestaat behoeft het geen dubbele punt. Omdat '!=' twee tekens heeft moeten we een dubbele punt gebruiken. In ons geval schrijven we dus i !=: 13. Dat levert Ja of Nee op.

```
{ :i
      (i !=: 13) nee: {
            Media toon: i.
      }.
} * 20.
```

Je ziet dat we hier ook boogjes gebruiken om ( i !=: 13 ). Dat komt omdat het anders niet duidelijk is dat  'nee:' een nieuw bericht is. Immers je zou het ook kunnen opvatten als i !=: 13 nee: {..} - dus dat !=: en nee: samen een bericht vormen met meerdere argumenten! Dat is niet de bedoeling. Een andere manier op het op te schrijven is natuurlijk door gebruik te maken van de komma:

```
{ :i
      i !=: 13, nee: {
            Media toon: i.
      }.
} * 20.
```

Mag dus ook, net wat je mooier vind. Je kunt het probleem van het ongeluksgetal 13 overslaan ook anders oplossen. Namelijk zo:

```
{ :i
      (i = 13) doorgaan.
      Media toon: i.
} * 20.
```

In dit geval sturen we ‘=’ naar i met als argument 13. Als antwoord krijgen we `Ja`. Als je het bericht `doorgaan` stuurt naar `Ja`, dan wordt de rest van de taak overgeslagen en gaat het programma verder met de volgende iteratie. Dan is i dus 14. Op deze manier wordt ‘Media toon’ dus overgeslagen als `i = 13`.

In alle gevallen is er sprake van een vertakking. De ene tak van het programma toont het nummer wel (alles behalve 13) en de andere tak van het programma toont het nummer niet (13).

Als we 13 niet alleen willen overslaan maar ook meteen helemaal met de taak stoppen, dus ook alle volgende iteraties weglaten, dan kun je het bericht ‘afbreken’ sturen in plaats van ‘doorgaan’.

Je kunt vanuit een taak ook antwoord teruggeven. Dat doe je met ‘<-’. Op deze manier kun je bijvoorbeeld de ene taak net zo lang laten uitvoeren totdat de andere taak Nee antwoordt.

```
>> i := 1.
{
      Media toon: i.
      i optellen: 1.
} zolang: { <- i < 13. }.
```

Je kunt Ja/Nee-objecten ook combineren. Stel dat je een taak wilt uitvoeren als iemand een score heeft van meer dan 100 en in level 2 zit. Je kunt dit dan zo opschrijven.

```
score > 100 en: level = 2 ja: { … }
```

Het bericht ‘en:’ geeft een Ja terug als je het stuurt naar een Ja-object en als het argument ook een Ja-object is. Op diezelfde manier kun je ook ‘of:’ gebruiken:

```
score > 100 of: level = 2 ja: { … }
```

In dit geval voeren we de taak uit als de speler meer dan 100 punten heeft verzameld of in level 2 zit. Het bericht ‘noch:’ geeft Ja terug als beide kanten Nee zijn.

## 1.5 Reeksen en Lijsten

In plaats van met cijfers te tellen, kunnen we ook met woorden tellen, zoals "één," "twee," "drie." Om dit te doen, plaatsen we de woorden eerst in een reeks. Hiervoor gebruiken we het reeksobject. Om een nieuwe reeks te maken, stuur je het bericht nieuw naar Reeks. Vervolgens kun je objecten aan de reeks toevoegen met het bericht add:. In het volgende voorbeeld creëren we een nieuwe reeks genaamd `r` en voegen we de woorden voor het tellen eraan toe:

```
>> r := Reeks nieuw
toevoegen: [‘een’],
toevoegen: [‘twee’],
toevoegen: [‘drie’].
```

Zoals je ziet staat er tussen nieuw en toevoegen geen komma. Dat hoeft in dit geval niet want ‘nieuw’ heeft geen dubbele punt. Er kan dus geen misverstand ontstaan dat ‘toevoegen’ niet een nieuw bericht is. In plaats van ‘toevoegen’ mag je ook ‘;’ gebruiken:

```
>> r := Reeks nieuw ; [‘een’] ; [‘twee’] ; [‘drie’].
```

Dat is niet alleen korter, maar doordat ; geen dubbele punten bevat mag je ook de komma’s weglaten. Om onze lus gebruik te laten maken van de reeks moeten we parameter :i omzetten naar een woord. We krijgen dus een getal binnen en we moeten er een woord van maken. Het Reeks-object kan dit voor ons doen. Als we het bericht `positie:` sturen gevolgd door een nummer, dan krijgen we het object terug dat op deze positie in de reeks staat. Dus om [‘een’] te krijgen zouden we kunnen sturen: ‘positie: 1’. Omdat onze telwoorden al op posities 1,2 en 3 staan kunnen we i gewoon doorgeven als positie:

```
>> r := Reeks nieuw ; [‘een’] ; [‘twee’] ; [‘drie’].
{ :i
      Media toon: (r positie: i).
} * 3.
```

Er kleeft wel een nadeel aan de code hierboven. Telkens als een telwoord toevoegt aan ‘r’ moet je ook het aantal herhalingen (nu 3) aanpassen. Dat betekent dus dat je steeds twee wijzigingen moet doen (anders worden bijvoorbeeld maar 3 van de 4 telwoorden getoond).

Het kan handiger. Reeks kent namelijk ook het bericht ‘elk:’. De taak die je dan meegeeft wordt telkens uitgevoerd op elk element uit de reeks. Je taak krijgt twee parameters, een voor het positienummer en een voor het object op die positie.

```
>> r := Reeks nieuw ; [‘een’] ; [‘twee’] ; [‘drie’].
r elk: { :i :telwoord
      Media toon: telwoord.
}.
```

Je kunt van een tekst een reeks maken door het bericht `opsplitsen:` te sturen naar een tekstobject, gevolgd door het teken waarmee de tekst opgeslitst moet worden. We zouden bovenstaande reeks ook kunnen maken met:

```
>> r := [‘een,twee,drie’] opsplitsen: ‘,’.
```

Ter controle tellen we het aantal elementen, dat doen we met het bericht ‘aantal’.

```
Media toon: r aantal.
```

Dit levert precies 3 op.

Laten we nog eens wat handige functies van `Reeks` bekijken.

Om een aantal elementen uit een reeks op te vragen gebruik je `van:lengte:`

```
>> fruitmand := Reeks nieuw ; [‘appel’] ; [‘peer’] ; [‘banaan’] ; [‘kiwi’] ;
[‘citroen’].
>> vruchten := fruitmand van: 2 lengte: 3.
```

Geeft bijvoorbeeld een verzameling `[‘peer’] ; [‘banaan’] ; [‘kiwi’]` in de variabele ‘vruchten’. Je kunt element ook vervangen:

```
>> fruitmand := Reeks nieuw ; [‘appel’] ; [‘peer’] ; [‘banaan’] ; [‘kiwi’] ;
[‘citroen’].
fruitmand vervang: 3 lengte: 2 door: (Reeks nieuw ; [‘druif’] ; [‘meloen’] ;
[‘tomaat’]).
```

Nu houden we `[‘appel’] ; [‘peer’] ; [‘druif’] ; [‘meloen’] ; [‘citroen’]` over in de fruitmand.

Als je twee reeksen hebt, kun je er een ander type verzameling van maken: een *lijst*, door ze te combineren. Een lijst bestaat uit object-paren, net zoals een woordenboek of een legenda. Je zoekt bijvoorbeeld naar de betekenis van een woord in een woordenboek. Het woord noemen we de ‘sleutel’. De betekenis noemen we de ‘waarde’. Stel dat we twee reeksen hebben. Een reeks met scores, en een reeks met spelernamen. Beide reeksen horen bij elkaar. De score van speler 1 uit de spelersreeks staat op positie 1 van de scorereeks.

```
>> scores := Reeks nieuw ; 100 ; 200 ; 300.
>> spelers := Reeks nieuw ; [‘Anna’] ; [‘Rob’] ; [‘Tessa’].
```

Met het bericht per: kun je de reeksen omzetten naar een lijst:

```
>> hiscores := scores per speler.
```

Als we nu de score van Rob willen weten kunnen we dat als volgt te weten komen:

```
Media toon: ( hiscores bij: [‘Rob’] ).
```

Maar dit mag ook:

```
Media toon: hiscores Rob.
```

Omgekeerd kunnen we ook eenvoudig een speler en zijn score toevoegen aan de hiscores:

```
hiscores zet: 500 bij: [‘Max’].
```

Maar dit mag ook:

```
hiscores Max: 500.
```

Zowel voor Reeksen als voor Lijsten geldt dat het is toegestaan om te vragen naar een niet-bestaand element:

```
Media toon: hiscores Tom.
```

Geeft dus geen fout. Het antwoord is *Niets.* Niets is een speciaal object dat leegte repersenteert of de afwezigheid van informatie. Er is een belangrijke test die je kunt doen op elk object om te kijken of het ‘Niets’ is:

```
hiscores Tom Niets? Ja: {
      Media toon: [‘Deze speler doet niet mee!’].
}.
```

Alleen het Niets-object zegt ‘Ja’ op de vraag Niets? Alle andere objecten zullen altijd ‘Nee’ zeggen.

## 1.6 Teksten

Een bekend probleem bij teksten is dat je vaak verschillende informatie in een tekst wilt combineren. In een game bijvoorbeeld kunt je een score hebben (laten we zeggen 100), en die score wil je tonen in een bericht zoals: ‘Je score is 100!’. Hoe combineren we nu het getal 100 (de score) en de tekst ‘Je score is…’ ? In Citrine zijn hier twee manieren voor. De eerste manier is om verschillende objecten met + aan elkaar te rijgen:

```
>> score := 100.
Media toon: [‘Je score is: ’] + score + [‘!’].
```

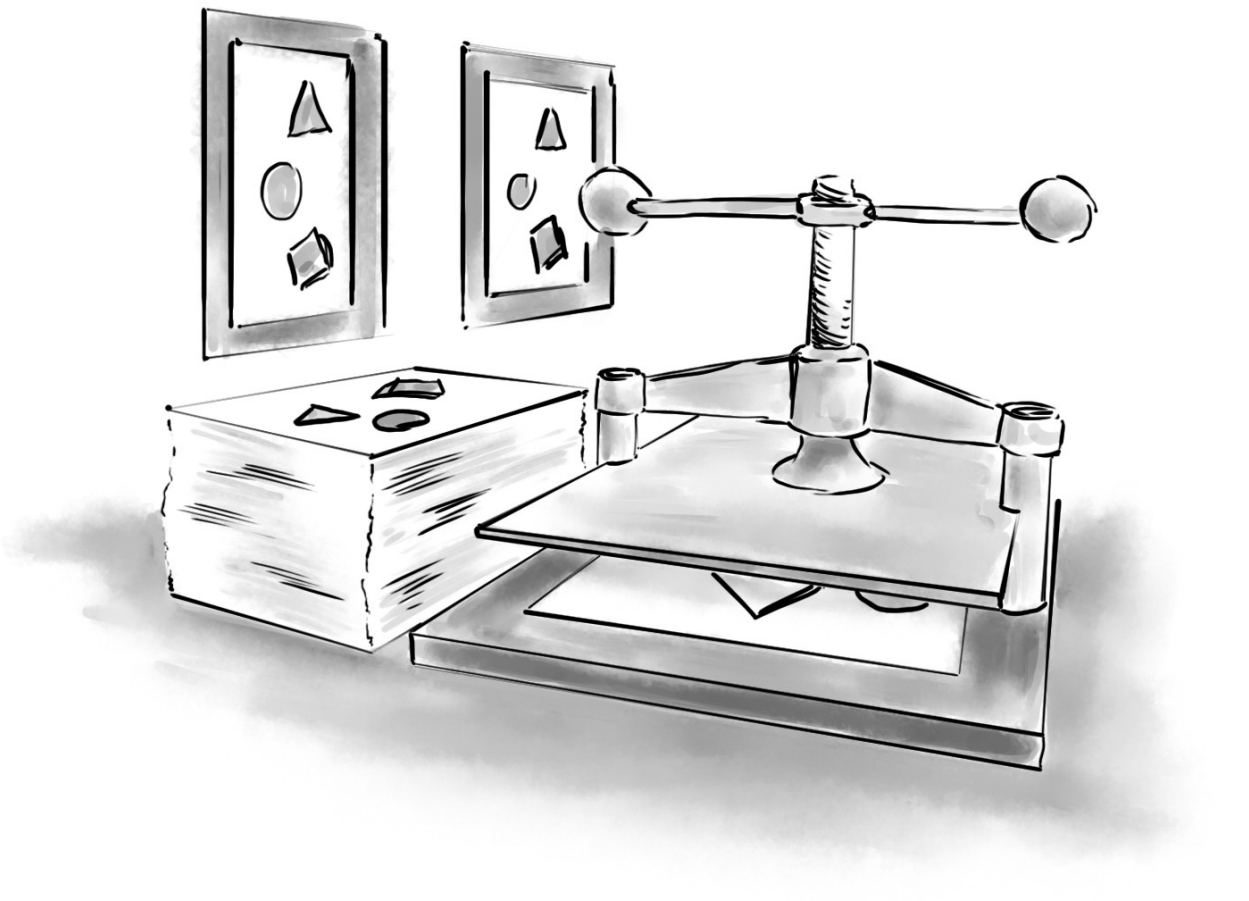
Een andere manier is om een woord uit de tekst te vervangen door de score. Je maakt dan eerst de tekst met op de plek waar de score moet komen een tijdelijke markering zoals ‘<score>’ (maar je mag ook zelf wat verzinnen). Vervolgens stuur je die plaatsvervangende tekst als bericht naar het tekstobject met als argument de vervangende tekst (de echte score, het getal 100). In dit geval komt onze code er zo uit zien:

```
>> score := 100.
Media toon: ([‘Je score is: <score>!’] <score>: score).
```

Het voordeel van deze laatste methode is dat het wat rustiger voor je ogen is. De volledige tekst, met tijdelijke markering is gemakkelijk leesbaar. De tekst wordt niet uit elkaar gedrukt door verschillende tekens. Daardoor is de kans op fouten ook wat minder. Je mag trouwens ook stukjes tekst vervangen met ‘vervang:door:’:

In plaats van gebruik te maken van dialoogvensters kun je tekst ook naar *standaarduitvoer* (ook wel bekend als stdout) schrijven, dit is vaak gekoppeld aan het *terminalvenster*:

```
Uit schrijf: [‘Hallo stdout!’].
```



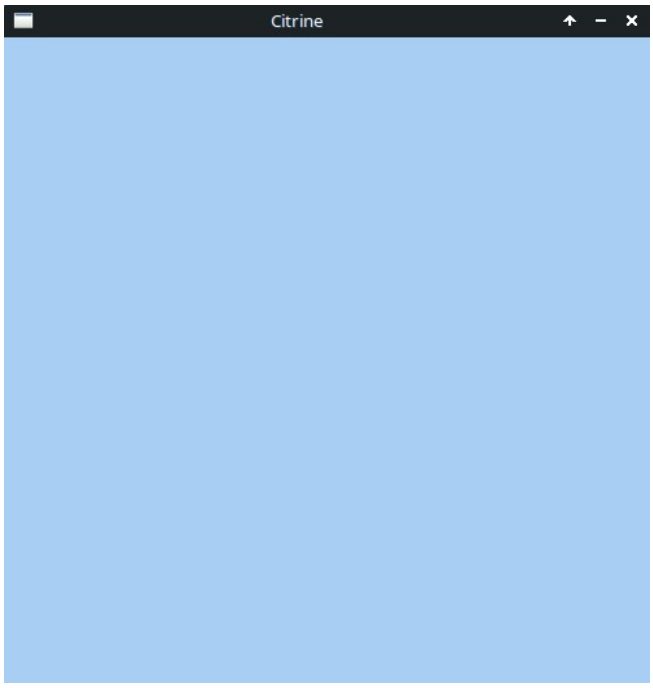
# 2 Objecten

Het spelen met dialoogvensters is een geweldige manier om de basisprincipes van Citrine te leren. Maar je wilt ook wat echte actie zien, zoals graphics, geluid, muziek en meer. In dit hoofdstuk ga ik bespreken hoe je de meer geavanceerde aspecten van objecten kunt gebruiken. Daarbij laat ik je zien hoe je afbeeldingen kunt weergeven, ze over het scherm kunt laten bewegen, animaties kunt maken en allerlei andere functies kunt gebruiken die de Media Plugin biedt.

## 2.1 Vensters & Plaatjes

Tot nu toe hebben we alleen ‘saaie’ berichtvensters op het scherm getoond. Tijd om daar verandering in te brengen. Kijk bijvoorbeeld eens naar de volgende code:

```
Media scherm: [‘lucht.png’].
```



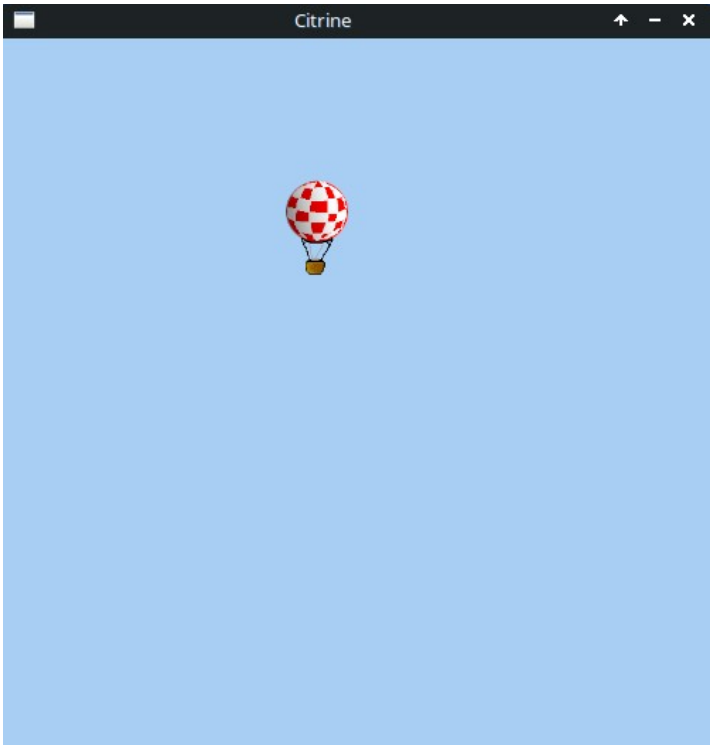

Als je dit programma uitvoert (en het afbeeldingsbestand lucht.png is aanwezig in dezelfde map), wordt er een venster geopend met als achtergrondplaat de lucht. Het venster blijft open totdat de gebruiker op het kruisje klikt. Het programma blijft ook ‘hangen’ bij het bericht ‘scherm:’ totdat de gebruiker het venster sluit.

Het Media-object biedt veel meer mogelijkheden dan alleen berichtvensters. Om die reden ‘sleept’ het Media-object ook allerlei bevriende objecten je programma binnen, zoals *Plaatje*, *Muziek*, en *Geluid*. Je raadt al waar die voor dienen. Maar voordat je die objecten kunt gebruiken moet je dus eerst het Media-object inladen. Hiervoor gebeurde dat automatisch omdat we het bericht ‘toon:’ steeds gebruikten. Zodra je een bericht stuurt naar Media wordt het namelijk ingeladen. Maar nu willen we niet direct iets gaan tonen, maar juist van de andere objecten uit de mediafamilie gebruikmaken. Om die reden moeten we een smoes verzinnen om Media in te laden. De makkelijkste manier is om te zeggen dat we een eigen Media object willen hebben:

```
>> m := Media nieuw.
```

Op deze manier maak je je eigen (media) object. Het bericht ‘nieuw’ maakt een nieuw object. Als blauwdruk wordt de ontvanger gebruikt. In ons geval maken we dus een nieuw object ‘m’ dat gebaseerd is op Media. Dat betekent dat je alle berichten die ja naar Media kunt sturen vanaf nu ook naar ‘m’ kunt sturen. Toch is het niet hetzelfde object. Het is wel degelijk een ander object, alleen met dezelfde berichten. We noemen dit overerving. Het object ‘m’ erft de berichten van Media. Tenzij we later nieuwe berichten toevoegen aan ‘m’ die Media niet heeft. Daar ga ik straks verder op in. Bovendien is ‘m’ minder typwerk dan ‘Media’, dus dat is mooi meegenomen. Dankzij het bericht naar Media, hebben we nu ook de beschikking over onder andere het Plaatje-object. Hiermee kunnen we plaatje over de achtergrond heen tekenen in het venster.

```
>> m := Media nieuw.
>> ballon := Plaatje nieuw: [‘ballon.png’].
ballon x: 200 y: 100.
m scherm: [‘lucht.png’].
```




Met het bericht ‘nieuw:’ maken we ons eigen plaatje. We geven als argument het bestand mee dat als bron dient voor de afbeelding. Daarna is het zaak om een plek in het venster uit te zoeken om de ballon neer te zetten. We kiezen 200,100. Dat is 200 punten van de linkerkant van het venster (x) en 100 punten vanaf de bovenkant (y). We kunnen de ballon ook laten bewegen:

```
ballon naar-x: 500 y: 100.
```

Met het bericht ‘snelheid:’ kun je de snelheid aanpassen.

## 2.2 Gebeurtenissen en Methodes

Zoals gezegd kunnen we berichten sturen naar objecten. Objecten voeren dan taken uit. Je kunt het takenpakket van een object uitbreiden. Daarvoor gebruiken we een speciaal bericht, namelijk ‘bij:doen:’. Met dit bericht kun je elk object uitbreiden. Dit bericht vereist twee argumenten. Het bericht waarop het ontvangende object moet gaan reageren en de taak die bij ontvangst van het nieuwe bericht gestart moet worden.

Stel dat we onze ballon, uit het vorige voorbeeld, heen en weer willen laten zweven over het scherm. Als we onze ballon naar positie x 500 sturen moeten we weten wanneer hij daar komt. Want op dat moment kunnen we de ballon omkeren. Gelukkig geeft het plaatje een seintje als de bestemming is bereikt. Het plaatje stuurt dan een bericht naar zichzelf: ‘bestemming’. Dat is natuurlijk leuk en aardig, maar dat bericht belandt in de prullenmand, want er is geen taak aan gekoppeld. Dat moeten we dus zelf doen. Het klinkt een beetje zot dat een object een bericht stuurt naar zichzelf, terwijl het bekend is dat dit bericht niks doet. De reden dat dit toch gebeurt is omdat het object op die manier de programmeur de mogelijkheid geeft om ‘in te grijpen’ bij relevante gebeurtenissen. Wij breiden onze ballon uit met de *methode* ‘bestemming’:

```
>> m := Media nieuw.
>> ballon := Plaatje nieuw: ['ballon.png'].
ballon x: 10 y: 100, naar-x: 500 y: 100.
ballon bij: ['bestemming'] doen: {
      zelf x? >=: 500, ja: {
            ballon naar-x: 10 y: zelf y?.
      }, anders: {
            ballon naar-x: 500 y: zelf y?.
      }.
}.
m scherm: ['lucht.png'].
```

We zeggen ook wel dat we de *methode* ‘bestemming’ toevoegen. De methode is dus de combinatie van het bericht en de bijbehorende taak. Let erop dat als we met ‘bij:doen:’ een methode willen toevoegen, we altijd het bericht opschrijven zonder de argumenten. Dus een bericht zoals ‘tussen: 1 en: 2’ noteren we als ‘tussen:en:’. De dubbele punten geven aan waar de argumenten komen te staan. Aan het begin van de taak moeten de parameters staan die overeenkomen met de argumenten in de *methodenaam*.

In ons voorbeeld zal de ballon bij aankomst op zijn bestemming gaan kijken of hij links of rechts van het scherm zit. Dit doen we door de x-positie op te vragen (met bericht x?) en dit te vergelijken met 500 (rechts). We sturen dit bericht (x?) naar zelf. Het sleutelwoord *zelf* verwijst altijd naar het object waarvan de methode deeluitmaakt. Als x groter of gelijk aan 500 is, maken we rechtsomkeert naar de linkerkant van het venster. Omdat de y-positie, de hoogte van de ballon gelijk blijft stellen

we die gelijk aan de huidige (y?). In plaats van ‘nee:’ gebruiken we voor het tegenovergestelde geval het bericht ‘anders:’. Dit is een *alias*. Beide berichten doen hetzelfde.

Een ander soort gebeurtenis die je kunt laten plaatsvinden is het afgaan van een wekker. Op die manier kun je op bepaalde tijden iets laten gebeuren. Zo stel je een wekker in:

```
media wekker: 1 over: 1000.
```

Het eerste argument is het nummer van de wekker. Het tweede argument is het aantal milliseconden . In dit voorbeeld gaat wekker 1 af over 1 seconde. Als de wekker afgaat krijgt media het bericht ‘wekker:’ toegestuurd met als argument het wekkernummer. Zo koppel je een taak aan de wekker:

```
media bij: [‘wekker:’] doen: { :nummer
      (nummer = 1) ja: {
            … schrijf hier wat er gedaan moet worden…
      }
}.
```

Om een wekker uit te zetten geef je als tijd -1 mee.

## 2.3 Eigenschappen

In het volgende voorbeeld gaan we van de ballonnen een ware luchtshow maken. We gaan een formatie maken van 5 balonnen die elk hun eigen startpositie hebben en heen-en-weer vliegen.

Het probleem is nu alleen hoe houden we bij welke ballon naar welke startpositie terug moet vliegen? Het liefst zouden we dat ergens bij de ballon zelf opslaan, zodat we het niet apart ergens moeten gaan bijhouden, anders wordt het een zooitje. Gelukkig kan dat. Een object kan namelijk eigenschappen bevatten.

Om dit te bereiken maken we eerst een nieuwe Ballon.

```
>> Ballon := Plaatje nieuw.
```

Deze keer gebruiken we een hoofdletter en laten we het bronbestand balloon.png weg, omdat deze Ballon niet zomaar een ballon is. Het is het prototype dat we zullen gebruiken om onze 5 ballonnen te creëren! In dit Ballon-object definiëren we de gedragingen en eigenschappen die alle afgeleide ballonnen moeten hebben. Een van de eerste dingen die we zullen doen, is ervoor zorgen dat ons prototype Ballon nieuwe ballonnen kan creëren:

```
Ballon bij: [‘nieuw’] doen: {
      <- Ballon nieuw: [‘ballon.png’].
}.
```

Hier zie je meteen al het voordeel van het gebruik van een prototype. Je kunt nu namelijk gewoon zeggen:

```
Ballon nieuw.
```

Om een nieuwe ballon te krijgen, je hoeft niet telkens het bronbestand van de afbeelding mee te geven. Maar een belangrijker voordeel is dat je nu steeds een ‘eigen’ ballon krijgt die aparte eigenschappen kan hebben, los van de prototype-ballon.

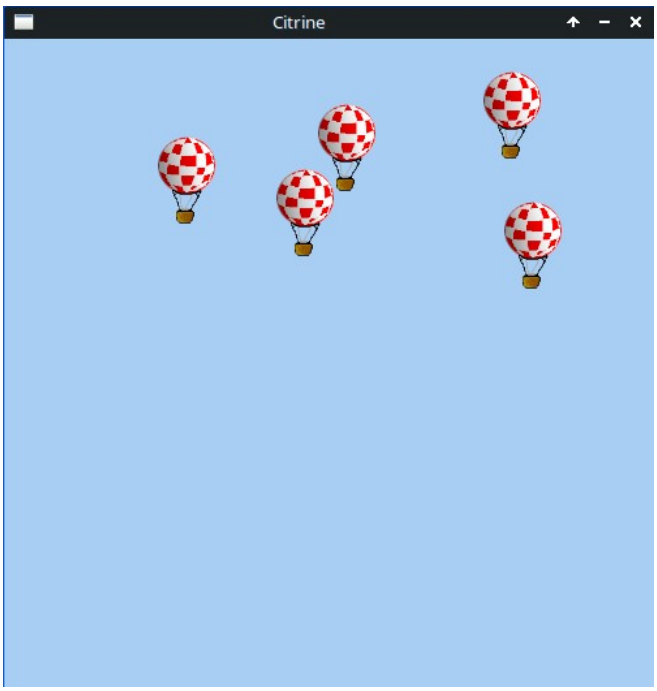
Een oplettende (of gevorderde) programmeur zal zich misschien afvragen hoe het komt dat het programma hier niet in een eindeloze lus terecht komt. Immers, ‘Ballon nieuw’ roept ‘Ballon nieuw’ aan. Goed gezien! Toch is hier geen sprake van een eindeloze lus. In de eerste plaats omdat we in ‘nieuw:’ sturen in plaats van ‘nieuw’, dus net even anders. Maar stel dat dat niet zo was? Citrine onderschept deze ‘recursie’ en zorgt dat hier niet dezelfde taak wordt uitgevoerd. In plaats daarvan

wordt de taak uitgevoerd van het prototype van Ballon, dat is Plaatje. Als je wel gebruik wilt maken van ‘recursie’, dus je wilt dat de methode die je probeert aan te roepen ook echt wordt uitgevoerd, ondanks het feit dat dat dezelfde methode is als waar je nu op dit moment in zit (en er dus de kans op een oneindige lus gaat ontstaan), dan moet je het bericht vooraf laten gaan door het speciale bericht ‘recursief’. In plaats van ‘Ballon nieuw’ mag je ook zeggen ‘zelf nieuw’, zelf verwijst immers al naar Ballon.

Onze nieuw-methode roept de nieuw-methode van Plaatje aan om een nieuw Plaatje te maken. Maar dit keer met alle extra methodes van Ballon erbij.

```
>> m := Media nieuw.
>> Ballon := Plaatje nieuw.
>> vloot := Reeks nieuw.


Ballon bij: ['nieuw'] doen: {
      >> ballon := zelf nieuw: ['ballon.png'].
      <- ballon.
}.


Ballon bij: ['start-x:y:'] doen: { :x :y
      eigen x := x.
      eigen y := y.
      zelf x: eigen x y: eigen y.
}.


Ballon bij: ['bestemming'] doen: {
      zelf x? >=: 500, ja: {
            zelf naar-x: eigen x y: eigen y.
      }, anders: {
            zelf naar-x: 500 y: eigen y.
      }.
}.


{ :i
      >> pos := i * 25.
      >> b := Ballon nieuw
            start-x: pos y: pos,
            naar-x: 500  y: pos.
      vloot toevoegen: b.
} * 5.
m scherm: ['lucht.png'].
```

En zo ziet dat er dan uit:




In dit voorbeeld creëren we een nieuw Ballon-prototype, en voor elke ballon in onze vloot wijzen we een unieke startpositie toe en laten we ze heen en weer vliegen. De start-x:y:-methode slaat de startpositie op, en de bestemming-methode laat de ballon naar de rechterkant van het scherm vliegen (x >=: 500) en vervolgens terugkeren naar zijn startpositie. De eigenschappen worden opgeslagen door de variabelen voor te laten gaan met het trefwoord *eigen*. Ze zijn alleen toegankelijk vanuit methoden die bij het object horen (of andere objecten die ervan zijn afgeleid - met behulp van een nieuw-bericht).

Elke ballon werkt onafhankelijk, maar dankzij het prototype-gebaseerde systeem delen ze gemeenschappelijke gedragingen terwijl ze hun eigen eigenschappen behouden.

## 2.4 Besturing

In plaats van een plaatje op eigen houtje te laten bewegen kunnen we het ook koppelen aan de joystick, gamepad en pijltjestoetsen van de gebruiker. Dat doe je door het bericht ‘besturing:’ te sturen, gevolgd door een code. Er zijn verschillende codes voor besturing die je kunt sturen naar een object. Als je code 1 stuurt dan zal het plaatje reageren door elke richting op te gaan, net zoals een spel dat je van ‘bovenaf’ speelt. Als je code 2 stuurt kan de speler alleen verticaal bewegen, voor een digitaal potje tennis bijvoorbeeld. Code 3 is alleen horizontaal. Bij code 4 reageert het plaatje alsof het een racewagen is, met links-rechts draai je aan het stuur en met ‘omhoog’ geef je gas. Je kunt de codes combineren met andere berichten om overige spelbesturingsvormen te maken, hier is een klein ‘receptenboekje’:

- **Kikker die een drukke weg oversteekt (bovenaf):**
  besturing: 1.
- **Springend konijn (platformspel):**
  besturing: 1, zwaartekracht: 1, springhoogte: 2.
- **Zwevend ruimteschip (2D schietspel):**
  besturing: 1, zwaartekracht: 0.5.
- **Tennisspel (alleen verticaal):**
  besturing: 2.
- **Sjoelen (alleen horizontaal):**
  besturing: 3.
- **Racespel (bovenaf, links-recht = draaien):**
  besturing: 4.
- **Ballonrace (zelfde als 1 maar dan zonder animatie):**
  besturing: 1, fixeer: Ja.

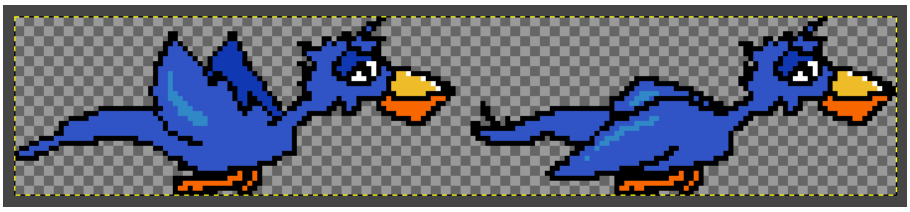
Er zijn diverse berichten waarmee je de besturing interessanter kunt maken. Met het bericht zwaartekracht: zorg je bijvoorbeeld dat je plaatje beweegt alsof het in een platformspel zit. Met een zwaartekracht van meer dan 1 kan het plaatje ook springen. De ‘springhoogte:’ kun je zelf instellen. Ook wordt het plaatje automatisch ‘ge-animeerd’. Dus als het een neus heeft dan staat die neus naar links als je naar links loopt en naar rechts als je naar rechts loopt. Zonder zwaartekracht staat de neus naar boven als je omhoog stuurt. Met zwaartekracht (>= 1) ‘springt’ het plaatje op dat moment. Met een zwaartekracht tussen 0 en 1, zweeft het plaatje en is de ‘animatie’ net iets anders. Als je helemaal niet wilt dat jouw plaatje in een bepaalde richting gedraaid wordt moet je ‘fixeer: Ja’ sturen, dat is bijvoorbeeld handig in het geval van een heteluchtballon. Met ‘versnelling:’ kun je de

besturing van het plaatje verder aanpassen, bijvoorbeeld het plaatje laten ‘glijden’ over ijs. Met ‘weestand:’ kun je het plaatje afremmen, alsof het door dikke modder moet lopen. Een bestuurbaar plaatje kan niet door andere plaatjes heen die het bericht ‘muur: Ja’ hebben ontvangen. Dat soort plaatjes zullen veranderen in muurtjes. Je kunt een plaatje wel door muren heen laten lopen met ‘spook: Ja’.

Als je ‘actief: Ja’ stuurt, ontvangt het plaatje een seintje als het botst met een ander plaatje. Zo kunnen we bijvoorbeeld een klein spel maken waarbij de ballon een rondfladderende vogel moet ontwijken:

```
>> m := Media nieuw.
>> ballon := Afbeelding nieuw: ['ballon.png'].
>> vogel := Afbeelding nieuw: ['vogel.png'], filmrol: 2 snelheid: 10.

vogel bij: ['vlieg'] doen: {
      >> rechts := Getal tussen: 0 en: 500.
      >> hoogte := getal tussen: 0 en: 100.
      zelf naar-x: rechts y: hoogte.
}.


vogel bij: ['bestemming'] doen: { zelf vlieg. }.

ballon bij: ['bots:'] doen: { :ander
      (ander = vogel) ja: { Programma einde. }.

}.


m bij: ['start'] doen: {
      ballon
            fixeer: Ja,
            actief: Ja,
            besturing: 1,
            zwaartekracht: 0,2.
      vogel x: 300 y: 10,
            zwaartekracht: 0,01,
            snelheid: 1,
            vlieg.
}.

m screen: ['lucht.png'].
```


Om de vogel in dit minispel te laten fladderen maken we een plaatje dat uit twee beelden bestaat, we kunnen nu tegen dit plaatje zeggen dat het een filmrolletje is en dat die film met een bepaalde snelheid afgespeeld moet worden tijdens het bewegen:



```
vogel filmrol: 2 snelheid 10.
```

Het is ook mogelijk om een filmpje automatisch te laten afspelen, zonder dat het beweegt, dat is bijvoorbeeld handig voor een kampvuur. Je stuurt dan: `autospeel: Ja`.

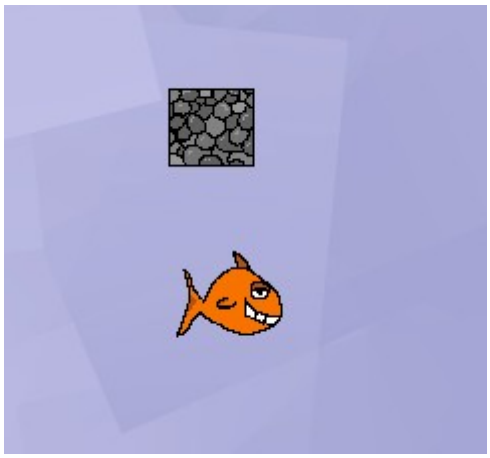
Door het bericht `breedte:hoogte:` te sturen, stel je een camera in, deze volgt de speler in het spel. Om de besturing van de speler tijdelijk uit te schakelen (zonder dat dit effect heeft op de camera) stuur je: ‘bevriezen: Ja’.

Als je een digitale schiettent wilt maken waarbij de muiscursor als vizier dient hoef je geen besturingsbericht te sturen. Bij een muisklik ontvangt een ‘actief’ plaatje namelijk het bericht ‘klik’. Het mediaobject zelf ontvangt ‘klik-x:y:’ bij elke klik. Dit kun je ook gebruiken om klikjes op knoppen af te vangen.

De leukste manier om Citrine wat beter te leren kennen is om te spelen met de demo’s. Bij Citrine krijg je een aantal voorbeeldprogramma’s, die noemen we demo’s. De bestanden heten demo1.ctr, demo2.ctr, demo3.ctr, en ga zo maar door…

In demo 1 zie je bijvoorbeeld hoe je een muis bestuurbaar kan maken en over het scherm kan laten bewegen:


In demo 2 voegen we een beetje zwaartekracht toe om een zweef- of drijfeffect te krijgen, we kunnen dan een vis nabootsen:



In demo 4 maken we een klein platformspel:

In demo 6 maken we een tennisspel:

Je kunt alle demo’s openen met een tekstprogramma en de inhoud bekijken en aanpassen!

## 2.5 Tekenen

Om een scorebord op het scherm te hebben waar de speler hun huidige score kan zien, maken we eenvoudig een afbeelding met tekst erop.

```
>> s := Plaatje nieuw: ['score'].
```

En zo voegen we tekst toe:

```
s schrijf: 0.
```

Helaas krijgen we op dit punt een foutmelding. Dit gebeurt omdat we nog geen lettertype hebben gekozen, waardoor het systeem niet weet hoe het eindresultaat eruit moet zien. We kunnen dit oplossen door een nieuw lettertype-object aan te maken:

```
>> f := Lettertype nieuw
bron: ['Shortcake.ttf']
grootte: 20.
```

In dit geval kiezen we het lettertype 'Shortcake.ttf' en stellen we de lettergrootte in op 20 punten. Vervolgens koppelen we het lettertype aan de afbeelding:

```
s lettertype: f.
```

Als je de tekst bewerkbaar wilt maken, zodat gebruikers direct in de afbeelding kunnen typen, kun je instellen:

```
s bewerkbaar: Ja.
```

Dit is handig voor applicaties.
We kunnen ook de kleur veranderen. Er zijn talloze kleuren om uit te kiezen, maar elke kleur die op een scherm wordt weergegeven, bestaat uit drie componenten. Zie het als drie verfemmers: rood, groen en blauw. Je kunt tussen 0 en 255 eenheden uit elke emmer gebruiken om elke gewenste kleur te mengen. Als je bijvoorbeeld felgroen wilt, gebruik je 255 eenheden uit de groene emmer. Wil je felrood? Neem 255 eenheden uit de rode emmer. Felgeel? Gebruik 255 eenheden uit zowel de rode als de groene emmer. Het Kleur-object is je palet waarmee je jouw perfecte tint kunt mengen. Voor

ons scorebord gebruiken we de kleur oranje. We kunnen oranje creëren door een nieuw kleur-object aan te maken:

```
>> oranje := Kleur nieuw rood: 250 groen: 150 blauw: 0.
```

Vervolgens gebruiken we 'oranje' als inkt om op ons scorebord te schrijven:

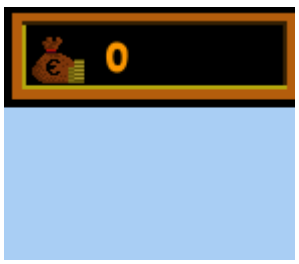
```
s inkt: oranje.
```

Je kunt de tekst uitlijnen met: uitlijnen-x: y:.

Hier is de complete code:

```
>> m := Media nieuw.
>> f :=
      Lettertype nieuw
      bron: ['Shortcake.ttf']
      grootte: 20.
>> s := Afbeelding nieuw: ['score.png'].
>> oranje :=
      Kleur nieuw
      rood: 250 groen: 150 blauw: 0.


s
 inkt: oranje,
 lettertype: f,
 uitlijnen-x: 50 y: 15,
 schrijf: 0.


m scherm: ['achtergrond.png'].
```

Afhankelijk van je achtergrondafbeelding voor de score en het lettertype, krijg je iets zoals dit:

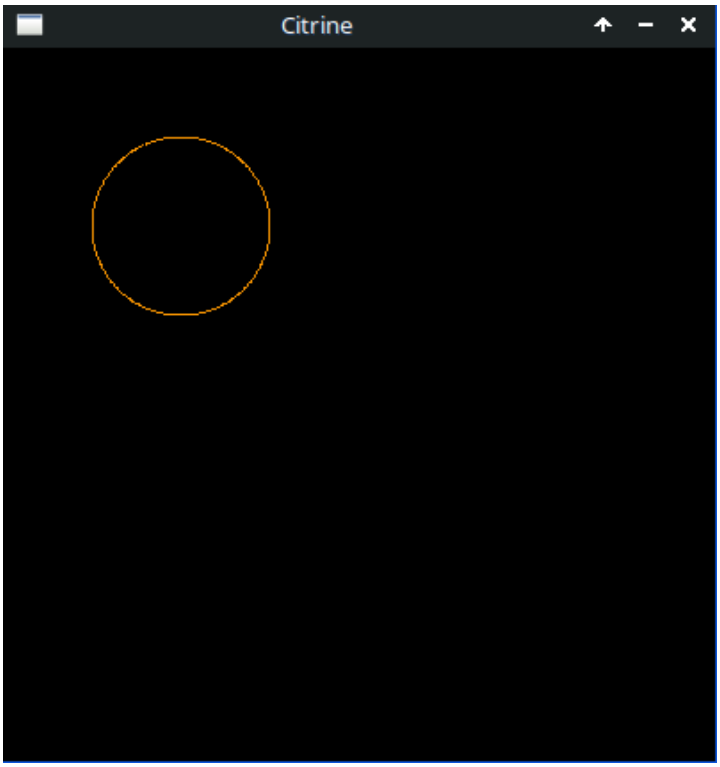



Je kunt ook individuele punten en lijnen op een afbeelding tekenen. Om een punt te tekenen, doe je het volgende:

```
>> m := Media nieuw.
>> oranje := Kleur nieuw rood: 250 groen: 150 blauw: 0.
>> doek := Afbeelding nieuw: ['doek.png'].
>> punten := Reeks nieuw.
punten voeg toe: (Punt nieuw x: 10 y: 10).
doek teken: punten kleur: oranje.
m scherm: ['doek.png'].
```

Om iets interessanters te tekenen, zoals een cirkel, kun je dit doen:

```
>> m := Media nieuw.
>> oranje := Kleur nieuw rood: 250 groen: 150 blauw: 0.
>> doek := Afbeelding nieuw: ['doek.png'].
>> punten := Reeks nieuw.
>> r := 50. # straal
>> c := 100. # middelpunt

{

      punten voeg toe: ( Punt nieuw
            x: r * i cos + c
            y: r * i sin + c
            ).

} * 360.

m bij: ['start'] doe: {
      doek teken: punten kleur: oranje.
}.

m scherm: ['doek.png'].
```



Op dezelfde manier kun je ook lijnen op een afbeelding tekenen. In dit geval moet je de twee punten specificeren waartussen de lijn getekend moet worden:

```
>> lijn := Reeks nieuw ;
(Punt nieuw x: 0 y: 0) ;
(Punt nieuw x: y…) ;
```
enzovoort...

Let op, hoewel het technisch gezien toegestaan is om afbeeldingen op het scherm te zetten en dingen buiten de start-taak van een scherm te tekenen, is het vaak verstandiger om dit binnen de start-taak te doen vanwege timingkwesties. Echter, declareer nooit variabelen binnen een start-taak voor elementen die op het scherm blijven staan, dit bespreken we in hoofdstuk 3.3 (scoping).

## 2.6 Muziek & Geluid

Citrine biedt twee Audio-objecten aan: Muziek en Geluid. Om een muziekstuk (achtergrondmuziek) af te spelen:

```
>> muziek := Muziek nieuw: ['lalala.mp3'].
muziek afspelen.
```

De muziek zal automatisch in een lus worden herhaald. Om de muziek terug te spoelen, stuur je het bericht `terugspoelen`. Om de muziek te stoppen, stuur je het bericht `stil`. Dan is er nog het Geluid-object. Dit object is vergelijkbaar, maar eenvoudiger en ondersteunt alleen het bericht `afspelen`. Geluiden worden nooit herhaald.

## 2.7 Netwerk

Met het Netwerk-object, dat deel uitmaakt van Media, kunt je gegevens ophalen van het internet en versturen naar het internet. Je gebruikt hiervoor het bericht *stuur:naar:*. Als je Niets stuurt, haal je alleen gegevens op van een internetpagina. Als je een tekst stuurt naar een internetpagina zal deze door de internetpagina worden verwerkt en krijg je het antwoord terug.

Om bijvoorbeeld de inhoud van de website van Citrine op te halen doen we:

```
>> m := Media nieuw.

>> n := Netwerk nieuw.

>> antwoord := n
      stuur: Niets
      naar: ['https://www.citrine-lang.org'].

m toon: antwoord.
```

## 2.8 Bestanden en Data

In plaats van afbeeldingen of muziek kun je ook code uit andere bestanden gebruiken. Om een ander Citrine-programma als onderdeel van je programma op te nemen:

```
Programma gebruik: ['myprogram.ctr'].
```

Dit zorgt ervoor dat het bestand 'myprogram.ctr' wordt geladen en op dat punt wordt ingevoegd. Citrine ondersteunt enkele basisfuncties voor het lezen en schrijven van bestanden. Hier is een voorbeeld:

```
>> f := Bestand nieuw: (Pad /tmp: ['test.txt']).
f schrijf: ['test'].
>> q := Bestand nieuw: (Pad /tmp: ['test.txt']).
Uit schrijf: q lezen, stop.
```

In plaats van code, afbeeldingen of muziek uit bestanden te laden, kun je ze ook toevoegen aan een datapakket en je programma ze uit dat pakket laten ophalen. Het voordeel van een datapakket is dat je middelen gebundeld zijn in één, ondoorzichtig pakket (zodat mensen de afbeeldingen niet kunnen bekijken of de muziekbestanden afzonderlijk kunnen afspelen). Datapakketten zijn ook vereist voor export naar andere platforms, zoals mobiele apparaten en spelcomputers.

Met het Pakket-object kun je bestanden in een datapakket plaatsen. Je kunt dit pakket vervolgens koppelen aan Media, en vanaf dat moment worden alle bestanden uit je datapakket opgehaald:

```
>> data := Pakketje nieuw: ['datapakket'].
data toevoegen: ['konijn.png'].
```

Om het datapakket te gebruiken:

```
Media koppel: ['datapakket'].
```

# 3 Geavanceerd

In dit hoofdstuk betreedt je soort van de ‘funhouse’ van Citrine. Zoals altijd moet uiteindelijk alles bij elkaar komen en aan elkaar gebreid worden in het leven en dan komen opeens de uitzonderingen, de bijzondere gevallen en de valluiken tevoorschijn. Net als elke andere programmeertaal heeft Citrine zo zijn rare kanten. Het is belangrijk dat je op de hoogte bent van deze gekkigheid anders kun je behoorlijk in de war raken! Daarbij komt nog een klein dingetje kijken. Citrine is een programmeertaal zonder ‘vangrails’.  Dat is een feature, geen bug. Sommige programmeertalen proberen de programmeur tegen zichzelf te beschermen. Citrine doet dat bewust niet. Dat betekent dat als je domme dingen doet, je domme prijzen wint! Met Citrine kun je behoorlijk ‘uit de bocht’ vliegen. Daar tegenover staat dat Citrine je vrij laat om dingen te doen die andere talen niet toestaan. Je hebt dus complete vrijheid! Geen belemmeringen voor je creatieve geest! Maar dus ook geen vangnet. Aan het einde van dit hoofdstuk bespreken we ook wat geavanceerde functies.

## 3.1 Kopiëren

Laten we beginnen met de formule voor totale chaos:

```
Ja := Nee.
```

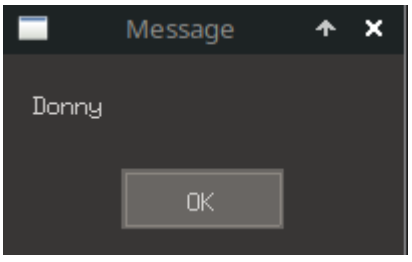
Ja, dit is mogelijk in Citrine (en voor zover ik weet, in geen enkele andere taal). Als je dit doet, is het gegarandeerd dat alles kapot gaat. Kortom, het is volledig onvoorspelbaar wat je programma zal doen nadat je deze code hebt uitgevoerd. Geen idee. De reden dat dit werkt is simpel: := maakt nooit een *kopie*.

In andere programmeertalen maakt het toewijzen aan een variabele soms een kopie en soms niet—het hangt ervan af. Dat kan ingewikkeld worden. In Citrine houden we het simpel: alles is een referentie. Er is maar één Ja-object en één Nee-object, en alle andere objecten bepalen hun gedrag op basis van deze twee.

Met := slaan we objecten op onder een specifieke naam. Je kunt zelfs een object onder meerdere namen opslaan. Maar onthoud, dit maakt geen kopie. Hier is een voorbeeld:

```
>> schaap := ['Dolly'].
>> kloon := schaap.

kloon vervang: ['l'] door: ['n'].

Media toon: schaap.
```

Resultaat:




Je zou verwachten Dolly te zien, maar beide namen wijzen naar hetzelfde object. Dus toen we de 'l' vervingen door de 'n', deden we dat zowel voor schaap als voor kloon, omdat ze hetzelfde object zijn, opgeslagen onder een andere naam!

Om een echte kopie van een object te maken, moet je er expliciet om vragen:

```
>> schaap := ['Dolly'].
>> kloon := schaap kopieer.

kloon vervang: ['l'] door: ['n'].

Media toon: schaap.
```

Het bericht `kopieer` maakt een kopie van de tekst. Je kunt deze daarna wijzigen zonder het origineel te veranderen. `Kopieer` werkt voor teksten, getallen, Ja/Nee-objecten, tijden, lijsten en reeksen. Voor andere objecten moet je je eigen kopie-methode schrijven. Let op dat bij het kopiëren van een lijst de elementen in de lijst **niet** worden gekopieerd!

```
>> schapen := Reeks nieuw ; ['Dolly'] ; ['Donny'].
>> klonen := schapen kopieer.

klonen elk: { :nummer :schaap schaap toevoegen: ['2']. }.
Media toon: klonen + schapen.
```

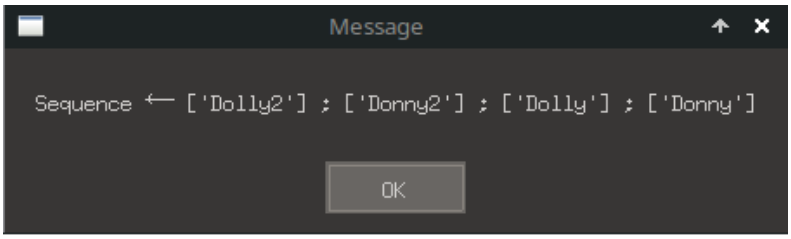
Resultaat:


Als je de elementen in een lijst ook wilt kopiëren, noemen we dat een *diepe kopie*. Je kunt dit zelf programmeren:

```
Reeks bij: ['kopieer'] doen: {
      >> dieptekopie := Reeks nieuw.
      zelf elk: { :nummer :element
            dieptekopie toevoegen: element kopieer.
      }.

      <- dieptekopie.

}.
```

```
>> schapen := Reeks nieuw ; ['Dolly'] ; ['Donny'].
>> klonen := schapen kopieer.
Media toon: klonen.
klonen elk: { :nummer :schaap schaap toevoegen: ['2']. }.
Media toon: klonen + schapen.
```

Resultaat:




Zonder de kopie zou je beide lijsten de tweede versie zien tonen. Dit werkt ook voor geneste lijsten.

Het is belangrijk op te merken dat, hoewel een kopie van een object op het origineel lijkt, het nooit hetzelfde is. Er is een object genaamd ‘Object’ dat de moeder van alle objecten is. Dit object heeft een methode ‘gelijk:’ dat objectidentiteiten vergelijkt. Elk object in de wereld van Citrine erft deze methode. Bekijk dit voorbeeld:

```
>> a := 1.
>> b := a kopieer.
>> c := a.

Media
      toon: ( a = b ),
      toon: ( a = c ),
      toon: ( a gelijk: b ),
      toon: ( a gelijk: c ).
```

Uitvoer:

Ja
Ja
Nee
Ja

In dit geval is a = b omdat a een kopie van b is, en het = bericht vergelijkt objectwaarden—in dit geval getallen. Aangezien a = 1 en b = 1, en 1 = 1, is het antwoord Ja. Hetzelfde geldt voor c. Maar wanneer we het bericht 'gelijk:' gebruiken, geërfd van Object, zien we een verschil. Hoewel c gelijk is aan a omdat ze beide naar hetzelfde object verwijzen, is a niet gelijk aan b, omdat de kopie fysiek een ander object is in het geheugen. Om te controleren of je een verwijzing hebt naar exact hetzelfde object in het geheugen, gebruik je het bericht `gelijk:`.

## 3.2 Conversies

Soms krijg je een getal terug als een tekstobject, bijvoorbeeld wanneer je invoer van de gebruiker leest vanuit een bewerkbare afbeelding. Om een tekst naar een getal te converteren, stuur je het bericht *getal*. Vanaf dat moment kun je berichten sturen die met getallen te maken hebben. Op dezelfde manier kun je het bericht bool sturen naar een object. Je krijgt dan Ja of Nee terug. Bool betekent eigenlijk ‘beslis of het Ja of Nee is’, dat woord is afgeleid van Boole, de naam van de logicus George Boole die leefde van 1815 – 1864. In plaats van Ja en Nee spreken we dan ook wel over *Booleans*. Als je Ja en Nee gebruikt in je computerprogramma zeggen we ook wel dat je gebruikmaakt van *Booleaanse algebra*.

In het algemeen kun je elk object omzetten naar een ander type met deze berichten:

| **Bericht** | **Gevolg** |
|---|---|
| getal | Maakt van een object een getal |
| tekst | Maakt van een object tekst |
| bool | Maakt van een object een beslissingsobject (Ja of Nee) |

Wat gebeurt er als je één van deze berichten stuurt naar een object? Hiervoor gelden de volgende regels:

| **Bericht** | **Ontvangend object** | | | | |
|---|---|---|---|---|---|
| | *Niets* | *Ja/Nee* | *Getal* | *Tekst* | *Andere Objecten* |
| bool | Nee | - | 0 → Nee<br>anders: Ja | Ja | *meestal* Ja |
| getal | 0 | Ja → 1  Nee → 0 | - | Probeert tekst als getal te interpreteren dus “1” → 1, anders 0. | *meestal* 1 |
| tekst | [‘niets’] | Ja → [‘Ja’]<br>Nee → [‘Nee’] | Tekstuele representatie | - | Tekstuele representatie afhankelijk van het object. |

Citrine zet objecten intern om wanneer dat nodig is. Wanneer je bijvoorbeeld een reeks op het scherm afdrukt, stuurt Citrine intern het *tekst* bericht naar de reeks. Hier kun je slim gebruik van maken. Stel dat je een reeks wilt afdrukken als een door komma's gescheiden lijst:

```
>> som := Reeks nieuw ; 1 ; 2 ; 3.

som bij: ['tekst'] doen: {
```

```
      >> opgeteld := 0.

      zelf elk: { :i :element
            opgeteld optellen: element.
      }.
      <- opgeteld tekst.

}.

Media toon: som.
```

Dit geeft 6 als resultaat.

## 3.3 Dynamische Scope

Wanneer een taak een variabele nodig heeft maar deze niet kan vinden, zal het de variabelen van de aanroepende taak controleren en verder zoeken langs de keten van taken. Dit zoekproces gaat door totdat de variabele gevonden is. Als de variabele nog steeds niet gevonden is, verschijnt er een foutmelding. Deze methode van het zoeken naar een variabele wordt *Dynamische Scoping* genoemd.

Om dit concept beter te begrijpen, doen we een quiz genaamd Bob of Alice. Probeer voor elk voorbeeld te voorspellen welke naam op het scherm zal verschijnen!

```
>> naam := ['Bob'].
{ Media toon: naam. } start.
```

In dit voorbeeld zal de naam ‘Bob’ verschijnen. De taak heeft geen eigen naam-variabele, dus zoekt hij naar de variabele in de aanroepende taak, wat in dit geval het hoofdprogramma is. Als een variabele gedeclareerd is in het hoofdprogramma, dus niet een taak, dan spreken we ook wel van een *globale variabele*, omdat deze overal in het programma zichtbaar is! Oke, door naar de volgende vraag:

```
>> naam := ['Bob'].
{
      naam := ['Alice'].
      { Media toon: naam. } start.

} start.
```

Welke naam zal worden getoond? In dit geval is het ‘Alice’. De buitenste taak overschrijft de waarde van naam van ‘Bob’ naar ‘Alice’. Laten we het voorbeeld nu een beetje aanpassen:

```
>> naam := ['Bob'].
{
      >> naam := ['Alice'].
} start.

Media toon: naam.
```

Dus, is het Bob of Alice? Deze keer is het ‘Bob’! Hoewel de naam in de taak is veranderd in ‘Alice’, wordt die wijziging genegeerd zodra de taak eindigt. Aangezien het toon-commando buiten de taak staat, ziet het deze verandering niet.

Zo:

```
>> naam := ['Bob'].
{
      naam := ['Alice'].
} start.

Media toon: naam.
```

wordt het wel Alice, want nu refereert naam in de taak aan de globale variabele.

Laten we het nog duidelijker maken—kun je het volgende voorspellen?

```
>> toon := { Media toon: naam. }.
>> verander := { naam := ['Bob']. }.
{
      >> naam := ['Alice'].
      verander start.
      toon start.

} start.
```

Het resultaat is ‘Bob’. Door de volgorde van de taken te volgen, zie je dat de variabele naam uiteindelijk wordt ingesteld op ‘Bob’ voordat de toon-taak wordt aangeroepen.

Citrine heeft ook een automatische opruimfunctie (*garbage collector*). Zodra een taak eindigt, worden alle binnen die taak gedeclareerde variabelen gewist. Bijvoorbeeld:

```
{  >> naam := [‘Alice’].  } start.
```

In dit geval wordt de variabele naam ‘vergeten’ na het einde van de taak.

```
Persoon bij: ['naam'] doen: { >> naam := [‘Alice’]. }.
```

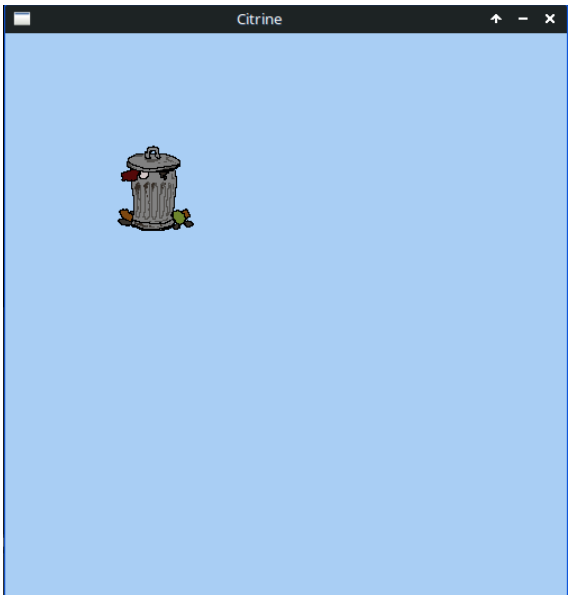
In dit geval geldt dezelfde regel. Als je wilt voorkomen dat de variabele verloren gaat, kun je deze aan het object koppelen door het een eigenschap te maken:

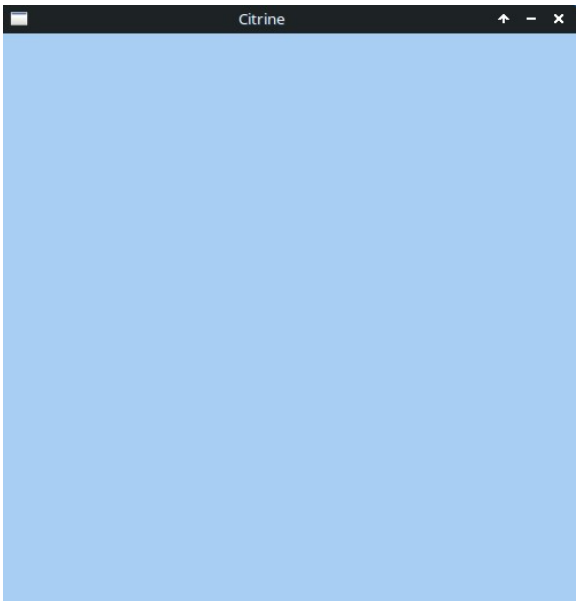
```
Persoon bij: ['naam'] doen: { eigen naam := [‘Alice’]. }.
```

Je kunt het resultaat van de garbage collector observeren met het volgende voorbeeld:

```
>> m := Media nieuw.

m bij: ['start'] doen: {
      >> afval := Plaatje nieuw: ['afval.png'].
      afval x: 100 y: 100.
}.

m bij: ['stap'] doen: {
      >> x := Reeks nieuw.
      x vul: 100000 met: ['Blah'].
}.

m scherm: ['achtergrond.png'].
```

In dit voorbeeld tekenen we een vuilnisbak op het scherm. We voegen ook een taak toe die bij elke ‘stap’ wordt uitgevoerd. Dit is nodig omdat de garbage collector standaard alleen actief wordt als het geheugen vol begint te raken. Na een tijdje zal de garbage collector de vuilnisbak op het scherm opruimen omdat de variabele *afval* ‘vergeten’ wordt.




Een andere manier om de garbage collector te activeren is door deze hyperactief te maken:

```
Programma geheugenbeheer: 4.
```

Als je deze regel bovenaan het programma toevoegt, zie je dat de vuilnisbak onmiddellijk wordt verwijderd. Je kunt de garbage collector ook handmatig activeren:

```
>> m := Media nieuw.

m bij: ['start'] doen: {
      >> afval := Plaatje nieuw: ['speler.png'].
      afval x: 100 y: 100.
      m wekker: 1 na: 3000.

}.

m bij: ['wekker:'] doen: {
      Programma vegen.
}.

m wekker:1 over: 1000, scherm: ['achtergrond.png'].
```

Er zijn 3 geheugenbeheermodi: 0 betekent helemaal geen opruiming, 1 (standaard) ruimt alleen op als ongeveer 80% van het toegewezen geheugen (64 MB) is gebruikt, en 4 betekent opruiming na elke cyclus. Er zijn ook enkele experimentele instellingen voor geheugenbeheer. Je kunt de standaard geheugentoewijzing wijzigen met de omgevingsvariabele *CITRINE_MEMORY_LIMIT_MB*.

## 3.4 Onbekende berichten

Wat gebeurt er als een object geen taak heeft gekoppeld aan een bepaald bericht? Als je een methode probeert aan te roepen die niet bestaat? Dan negeert het object je. Je bericht verdwijnt als het ware in de prullenmand. Toch kun je deze ‘prullenmandberichten’ ook ‘opvangen’ en er ‘iets mee doen’.

Stel dat we in een spel alle plaatjes willen verbergen. We hebben alle plaatjes in een reeks gestopt die ‘rommel’ heet. We kunnen dat zo doen:

```
rommel elk: { :nummer :plaatje
      plaatje zichtbaar: Nee.
}.
```

Maar wat als we nu allerlei berichten naar de rommel willen sturen? Het zou fijn zijn als we een bericht naar de reeks ‘rommel’ zouden kunnen sturen en dat dit bericht dan doorgestuurd wordt naar elk plaatje in de reeks. Dat kan! Als je een niet bestaande methode aanroept op een object komt het bericht uiteindelijk altijd binnen via:

```
rommel bij: [‘bericht:’] doen: {
}.
```

En als je een argument meestuurt komt het binnen via:

```
rommel bij: [‘bericht:argument:’] doen: {
}.
```

Bij twee argumenten:

```
rommel bij: [‘bericht:argument:argument:’] doen: {
}.
```

En bij drie:

```
rommel bij: [‘bericht:argument:argument:argument:’] doen: {
```

```
}.
```

Op die manier kunnen we dus alle ‘onbekende’ berichten, waar geen methode voor is, ‘opvangen’. Als we eenmaal zo’n bericht hebben opgevangen kun je het doorsturen met:

```
rommel bij: [‘bericht:argument:’] doen: { :bericht :argument
      zelf elk: { :nummer :object
            object
                  bericht: bericht
                  argumenten: Reeks nieuw ; argument.
      }
}.
```

Met het bericht `bericht:argumenten` stuur je dus een bericht door naar een ander object. Zo kun je nu bijvoorbeeld het bericht `zichtbaar` doorsturen naar alle plaatjes in de rommel-verzameling:

```
rommel zichtbaar: Nee.
```

En om de rommel weer zichtbaar te maken:

```
rommel zichtbaar: Ja.
```

Je kunt ook andere berichten sturen met een enkel argument:

```
rommel snelheid: 1.
```

In alle gevallen wordt je bericht doorgestuurd naar alle plaatjes in de verzameling ‘rommel’.

## 3.5 Kettingmodus

Stel, je hebt een reeks:

```
>> x := Reeks nieuw ; 1 ; 2 ; 3.
```

Je wilt het eerste en laatste element afknippen. Dat kan met de berichten `lknip` (links afknippen) en `rknip` (rechts afknippen). In dat geval zou je zeggen:

```
x lknip rknip.
```

Helaas! Dat gaat mis! Een knip-bericht geeft namelijk ook het element terug dat wordt afgeknipt. Dus `lknip` geeft het linker element van de reeks (het eerste element dus) terug, en daar zit je dan tegenaan te praten met `rknip`…. Niet de bedoeling! Een mogelijke oplossing voor dit probleem is om er twee zinnen van te maken (we gaan er even vanuit dat je dus geen belangstelling hebt voor de afgeknipte elementen, die kunnen gewoon de prullenbak in):

```
x lknip.
x rknip.
```

Maar dat is natuurlijk onhandig, vooral als je meer dan twee elementen wilt weghalen. Voor dit soort gevallen kun je de kettingmodus inschakelen in Citrine. Bij kettingmodus worden alle antwoorden van objecten altijd genegeerd en krijg je steeds weer het object zelf terug. Op die manier kun je blijven ‘doorkletsen’ met het object, ook al geeft het object iets anders dan zichzelf terug als antwoord. Op die manier kun je een ketting van berichten sturen. Vandaar ‘kettingmodus’. Je start de kettingmodus door het bericht `doen` te sturen en je stopt het door het bericht `klaar` te sturen. We gaan het nu toepassen op ons voorbeeld:

```
x doen lknip rknip klaar.
```

Bovenstaande voorbeeld werkt zoals je verwacht. Het knipt de meest linker en meest rechter elementen af van verzameling x. Dus alleen het getal 2 blijft over.

## 3.6 Kwalificering

Aan een getal kun je een *kwalificering* hangen, bijvoorbeeld: 6 appels. Elk bericht dat een getal niet herkent, wordt beschouwd als een kwalificering. Je kunt de kwalificering van een getal ophalen met het bericht `kwalificering`:

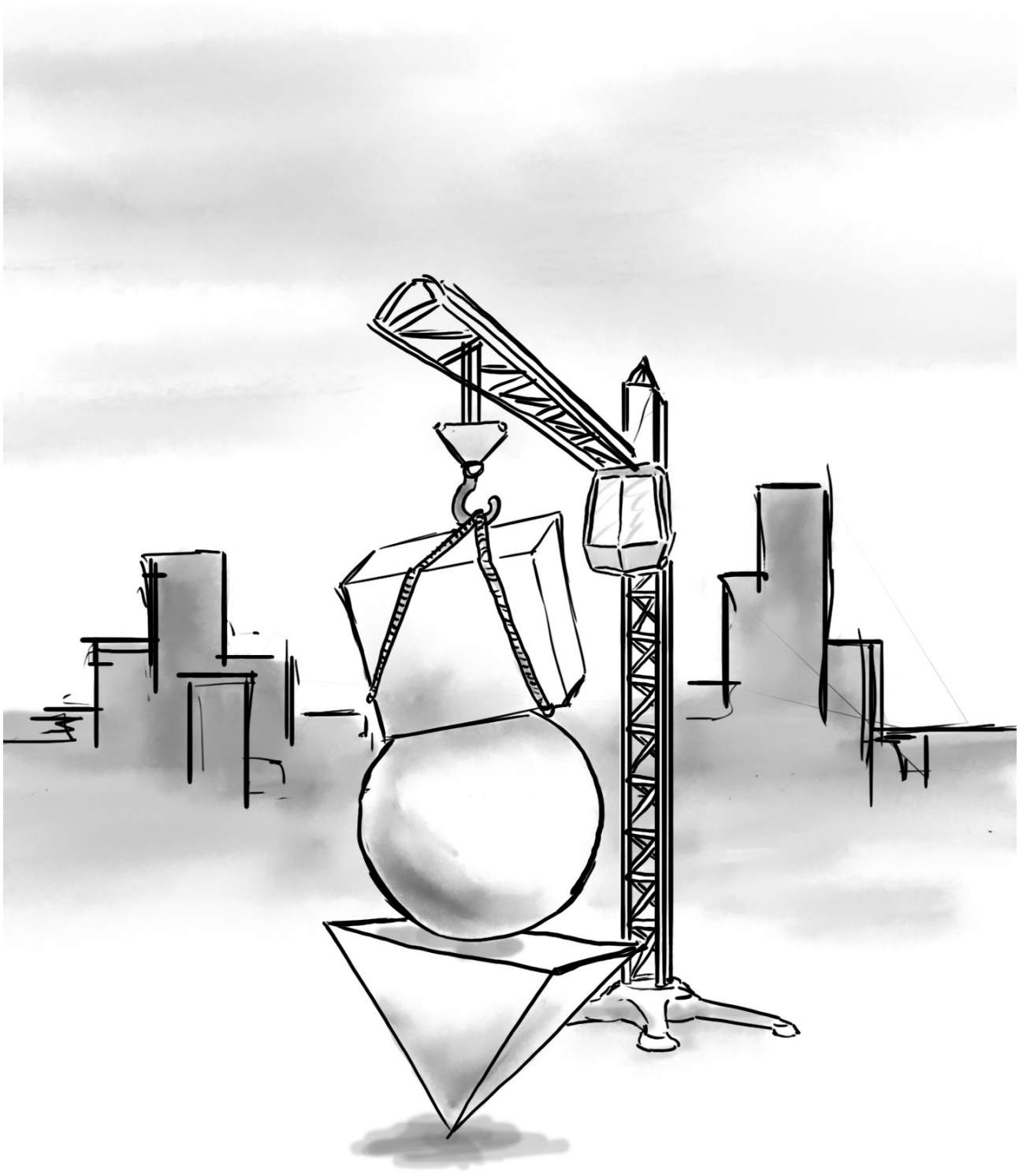
```
>> bedrag := 6 muntjes.

Media toon: bedrag.
Media toon: bedrag kwalificering.
```

Dit programma toont eerst: ‘6 muntjes’ en dan: ‘muntjes’. Een kwalificering is dus eigenlijk een tekst-object dat aan een getal zit vastgeplakt. Je kunt kwalificeringen gebruiken om bijvoorbeeld bedragen in verschillende valuta bij elkaar op te tellen. In het volgende voorbeeldprogramma illustreren we dit principe aan de hand van een historische valutacalculator, het voordeel hiervan is dat de wisselkoers enigszins stabiel blijft ;-) .

```
Getal bij: ['+'] doen: { :aantal
      (aantal kwalificering = ['dukaten']) ja: {
            aantal keer: 5.
      }.
      zelf optellen: aantal.
}.

Media toon: 7 florijnen + 3 dukaten, stop.
```

Dit programma toont: ‘22 florijnen’.

In het bovenstaande codefragment passen we het +-bericht aan zodat rekening wordt gehouden met de munteenheid, in ons voorbeeld is 1 dukaat evenveel waard als 5 florijnen.



# 4 Uitbreidingen

***Opgelet! Als je net begint met programmeren kun je dit hoofdstuk beter overslaan. We gaan hier namelijk in op de zaken die meer gericht zijn op gevorderde of professionele programmeurs.***

Er zijn verschillende manieren om de functionaliteiten van Citrine uit te breiden. Je kunt plug-ins installeren, externe programma's via de opdrachtregel aanroepen of routines aanroepen uit externe softwarebibliotheken, dit laatste heet FFI en is een hele krachtige functie van Citrine, maar ook complex. In dit hoofdstuk bespreek ik alledrie de methoden. Voor alledrie de methoden geldt dat ik alleen de Citrine-kant van het verhaal bespreek. Dat is dus een half verhaal. De rest van de documentatie moet je opzoeken bij de betreffende plug-in (in het geval van een plug-in), in de handleiding van je besturingsysteem (in het geval van de opdrachtregel) of in de handleiding van de externe softwarebibliotheek (in het geval van FFI). In tegenstelling tot voorgaande hoofdstukken is dit dus geen afegrond geheel, je moet dit hoofdstuk dus meer zien als een soort springplank naar andere systemen. Dankzij de 'open' architectuur van Citrine zijn de mogelijkheden dus eigenlijk zo goed als oneindig. Vooral met FFI kun je eigenlijk alles maken wat je maar wilt. Wil je een complete office suite schrijven voor desktop? Dat kan met FFI. Daar staat tegenover dat FFI niet makkelijk is en je veel zult moeten onderzoeken. Je zult vooral moeten grasduinen in de documentatie van de systemen waarmee je wilt communiceren via FFI. Toch is FFI wel degelijk een nuttig gereedschap. Het is immers onmogelijk om alle functionaliteiten die je kunt bedenken direct in te bakken in Citrine. Dat gaat hem niet worden. Maar met FFI kun je gebruik maken van alle systemen die door de jaren heen geschreven zijn. Van het verwerken van PDF-bestanden tot GUI-systemen zoals GTK. FFI opent de deur naar het complete ecosysteem.

## 4.1 Plugins

Het Media-object is een voorbeeld van een uitbreidingspakket. Naast de mediaplugin kun je ook andere plugins voor Citrine installeren. Dit die je door de bijbehorende bestanden (eindigent op .dll of .so) te kopieren naar de map ‘mods’.

Het volledige pad van een pluginbestand ziet er altijd zo uit:

```
mods/<Objectnaam>/libctr<Objectnaam>.<uitgang>
```

Dus als we fictieve plugin ‘Aquarium’ zouden willen installeren zouden we het volgende bestand moeten hebben in ‘mods’:

mods/aquarium/libctraquarium.so (Linux)

mods/aquarium/libctraquarium.dll (Windows)

We kunnen het object dan zo inladen in onze code:

```
>> a := Aquarium nieuw.
```

Dus gewoon, net als bij de mediaplugin, door een bericht naar het object te sturen.

## 4.2 Opdrachtregel

Vanuit Citrine kun je systeemopdrachten laten uitvoeren door het bericht `opdrachtregel:` naar `Programma` te sturen. Je kunt hiermee bijvoorbeeld bestanden kopiëren, verplaatsen of andere programmatuur aanroepen en bepaalde taken laten uitvoeren.

Stel dat we bijvoorbeeld op een Linux-systeem de bestanden wilt tonen in de huidige map, dan kunt u dat zo doen:

```
Programma opdrachtregel: Instructie ls.
```

Bovenstaande code voert het commando ‘ls’ uit in de ‘shell’ van het systeem en geeft het resultaat als tekst terug. Je mag het ook zo opschrijven:

```
Programma opdrachtregel: [‘ls’].
```

Het Instructie-object is een eenvoudig hulpstuk dat u kunt gebruiken in plaats van een Tekst-object. Je mag opdrachten voor het ‘Programma’ specificeren als tekst of als een Instructie-object.

## 4.3 FFI

FFI staat voor Foreign Function Interface. Met FFI kun je functionaliteiten gebruiken die geschreven zijn door anderen in andere programmeertalen en beschikbaar zijn gesteld in de vorm van DLL-bestanden, SO-bestanden of Dylib-bestanden. Dit kunnen allerlei functionaliteiten zijn. Er is een zeer grote hoeveelheid aan functionaliteiten beschikbaar via deze weg.

Laten we meteen beginnen met een voorbeeld:

```
>> media := Media nieuw.
media koppel: (
      Reeks nieuw ;
      ['/usr/lib/x86_64-linux-gnu/libc.so.6'] ;
      ['printf'] ;
      ( Reeks nieuw ; ['pointer'] ; ['int'] ) ;
      ['void'] ;
      ['Printf'] ;
      ['sjabloon:getal:']
).
>> s := Blob utf8: ['FFI heeft %d letters.\n'].
Printf sjabloon: s getal: 3.
s vrij.
```

Het resultaat van deze bende code is dat je op de command-line het volgende ziet:

```
FFI heeft 3 letters.
```

Het eerste wat je nu waarschijnlijk zult denken is, wat is dit veel code voor zoiets simpels. Ik bedoel, dat hadden we toch gewoon met een ‘Media toon’ kunnen oplossen. Het antwoord daarop is volmondig ja! Inderdaad. Maar het gaat hier om een illustratief voorbeeld. FFI gebruik je meestal om ingewikkeldere zaken te doen, maar die lenen zich dan weer slecht als voorbeeld. Dus heb ik iets triviaals gekozen. De methode ‘koppel’ ken je waarschijnlijk nog van hoofdstuk 2. Koppel heeft meerdere functies. De eerste functie van koppel is om externe bronnen zoals plaatjes en muziekjes uit een datapakket te koppelen. De tweede functie van koppel is iets ingewikkelder, daarmee kun je functies uit externe bronnen (lees DLLs/SOs) koppelen. Als argument geef je in dit geval een reeks mee met de volgende elementen:

1. Het DLL of SO bestand waar je gebruik van wilt maken

2. De functie uit het hiervoor gekozen bestand welke je wilt koppelen

3. Een reeks met de namen van de datatypes van de argumenten van de functie

4. De naam van het return-type van de functie

5. De naam van het object waar je deze functie aan wilt koppelen, als dit object nog niet bestaat wordt het automatisch aangemaakt

6. Het bericht waar deze functie aan gekoppeld moet worden

In het bovenstaande voorbeeld willen we de functie ‘printf’ uit ‘libc.so’ koppelen. De namen van de datatypen kun je vinden in de documentatie van de software waarmee je wilt koppelen. Je kunt kiezen uit de volgende typen:

void, pointer, float, double, int, uint, char, uchar, intX en uintX waarbij X = 8,16,32 of 64.

Deze typen verwijzen naar de hoeveelheid bytes die nodig zijn om de gegevens op te slaan.

In het geval van printf willen we koppelen aan een nieuw object Printf en het bericht ‘sjabloon:getal:’. Je vertaalt dus de functie van buitenaf naar het Citrine-dialect voordat je het gebruikt. Het bericht ‘sjabloon:getal:’ verwacht als eerste parameter een buffer met de sjabloontekst. Om deze buffer aan te maken gebruiken we het Blob-object. Met Blob kun je zelf geheugen reserveren. Je bent dan ook verantwoordelijk om dit geheugen, na gebruik weer vrij te geven. Dat doe je met het bericht ‘vrij’. Je kunt een geheugenblob op verschillende manieren vullen. In ons voorbeeld vullen we het met tekst, dus gebruiken we het bericht utf8: (utf-8 is codering om tekst om te zetten naar bytes). Je kunt een blob ook vullen met ‘vul:’, dan geef je een reeks met bytewaarden mee. Een blob uitlezen kan met `van:lengte:` . Je krijgt de bytes in de blob dan als reeks terug. Je kunt zelfs een ‘c-struct’ maken met een Blob, hiervoor gebruik je het bericht `struct:`, als argument geef je een reeks met c-typen mee. Dit kan nodig zijn als je een c-functie wilt aanspreken in een externe softwarebibliotheek die een pointer naar een struct verwacht.