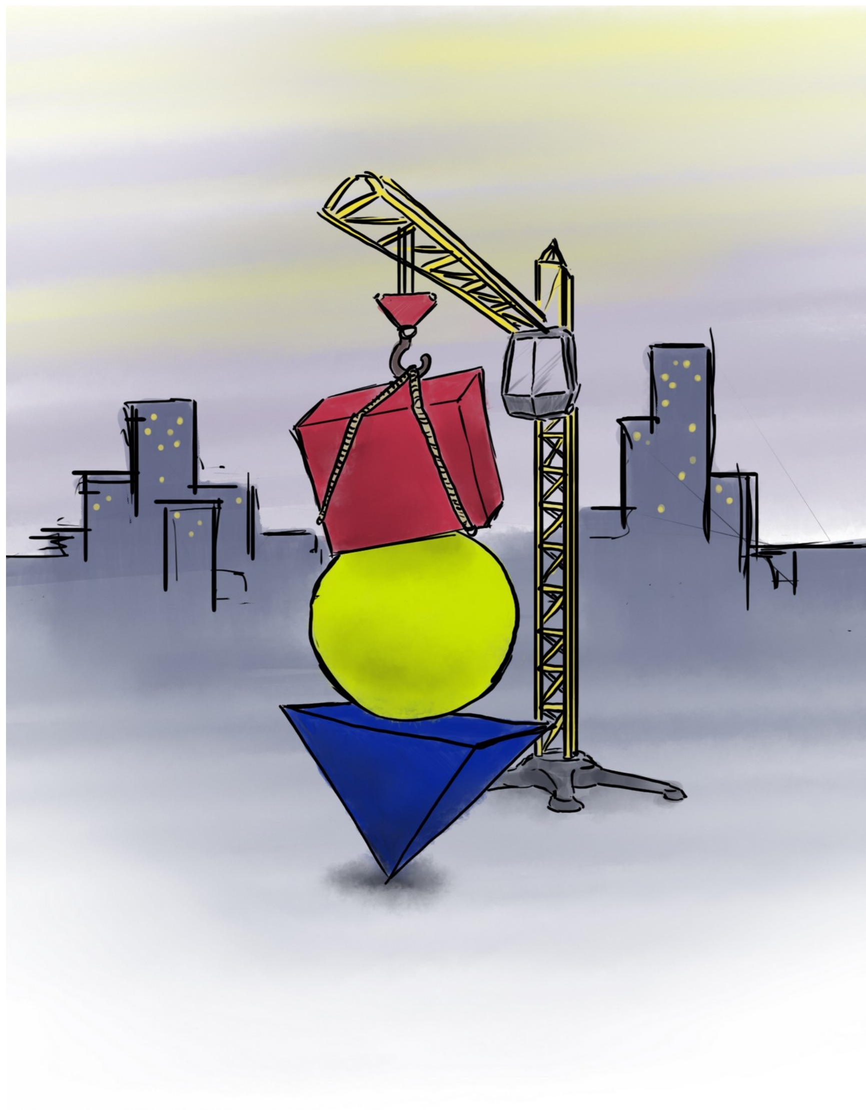
# CITRINE

## Manual / Handleiding 2025

Gabor de Mooij



Copyright © 2025 Gabor de Mooij

Illustrations by Robert Cabri

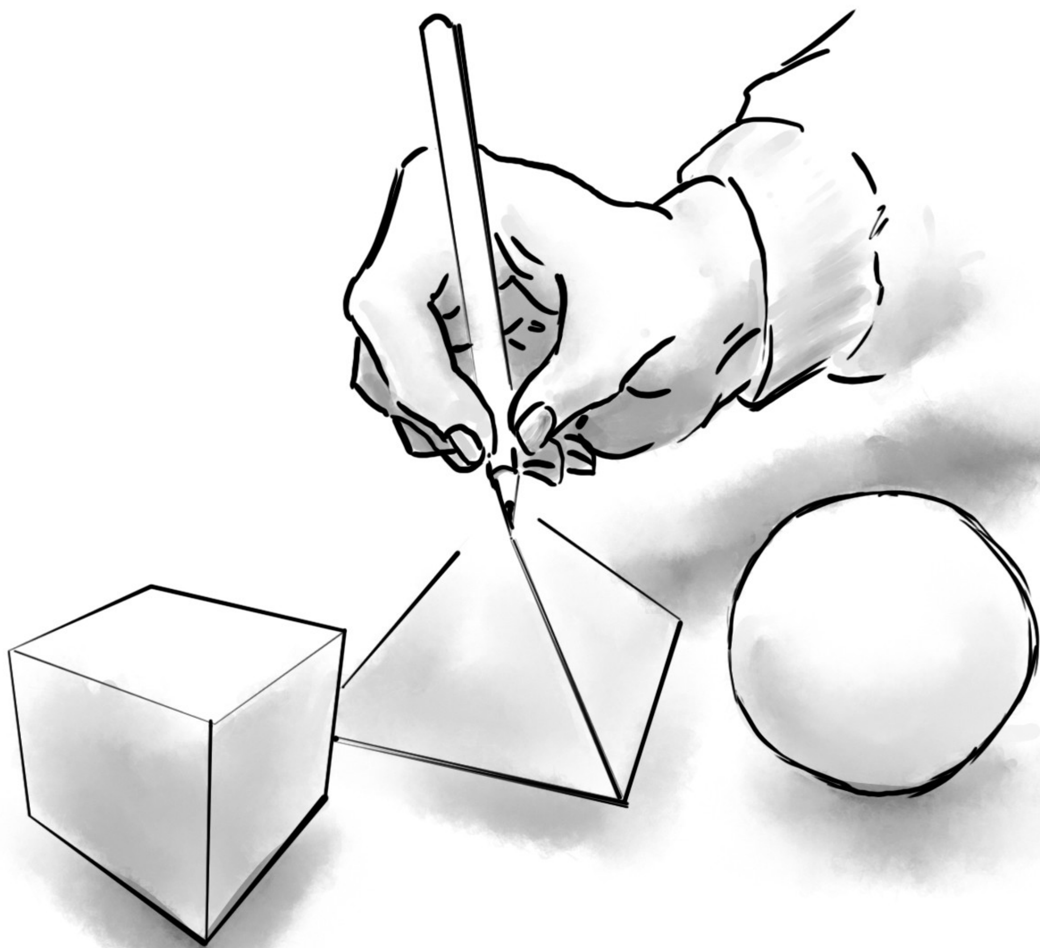
Geschreven door Gabor de Mooij

Tekeningen door Robert Cabri

Versie 2025-05

# Inhoudsopgave

1 Berichten....9
1.1 Programma’s....10
1.2 Berichten....11
1.3 Taken en Lussen....15
1.4 Vertakkingen....17
1.5 Reeksen en Lijsten....20
1.6 Teksten....24
2 Objecten....26
2.1 Vensters & Plaatjes....27
2.2 Gebeurtenissen en Methodes....30
2.3 Eigenschappen....32
2.4 Besturing....35
2.5 Tekenen....40
2.6 Muziek & Geluid....44
2.7 Netwerk....45
2.8 Bestanden en Data....46
3 Geavanceerd....48
3.1 Kopiëren....49
3.2 Conversies....52
3.3 Dynamische Scope....54
3.4 Onbekende berichten....58
3.5 Kettingmodus....60
3.6 Kwalificering....61
4 Uitbreidingen....63
4.1 Plugins....64
4.2 Opdrachtregel....65
4.3 FFI....66
5. Naslag....69
5.1 Overzicht objecten en berichten....70



# 1 Berichten

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.
```

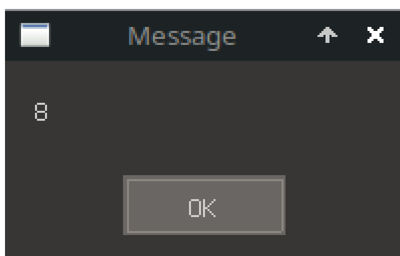
Nu hebben we het maandnummer (bijvoorbeeld 8) opgeslagen onder de naam 'm'. Dit noemen we een variabele. Geheugenruimte reserveren wordt *declareren* genoemd. Nu we het maandnummer onder 'm' hebben opgeslagen, kunnen we het later in het programma gebruiken. Bijvoorbeeld, als we het maandnummer op het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

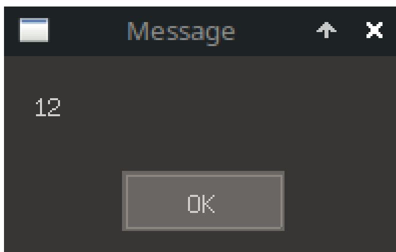
Het deel tussen de boogjes wordt eerst uitgevoerd (als je meerdere boogjes in elkaar gebruikt leest de computer van binnen naar buiten). We kunnen ook tekst op het scherm tonen (in plaats van getallen). Het is traditie om te beginnen met het bericht "Hello World”. In Citrine schrijf je dit als:

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

Je ziet dat we hier ook boogjes gebruiken om ( i !=: 13 ). Dat komt omdat het anders niet duidelijk is dat  'ja:' een nieuw bericht is. Immers je zou het ook kunnen opvatten als i !=: 13 ja: {..} - 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, ja: {
            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 + [‘!’].
```

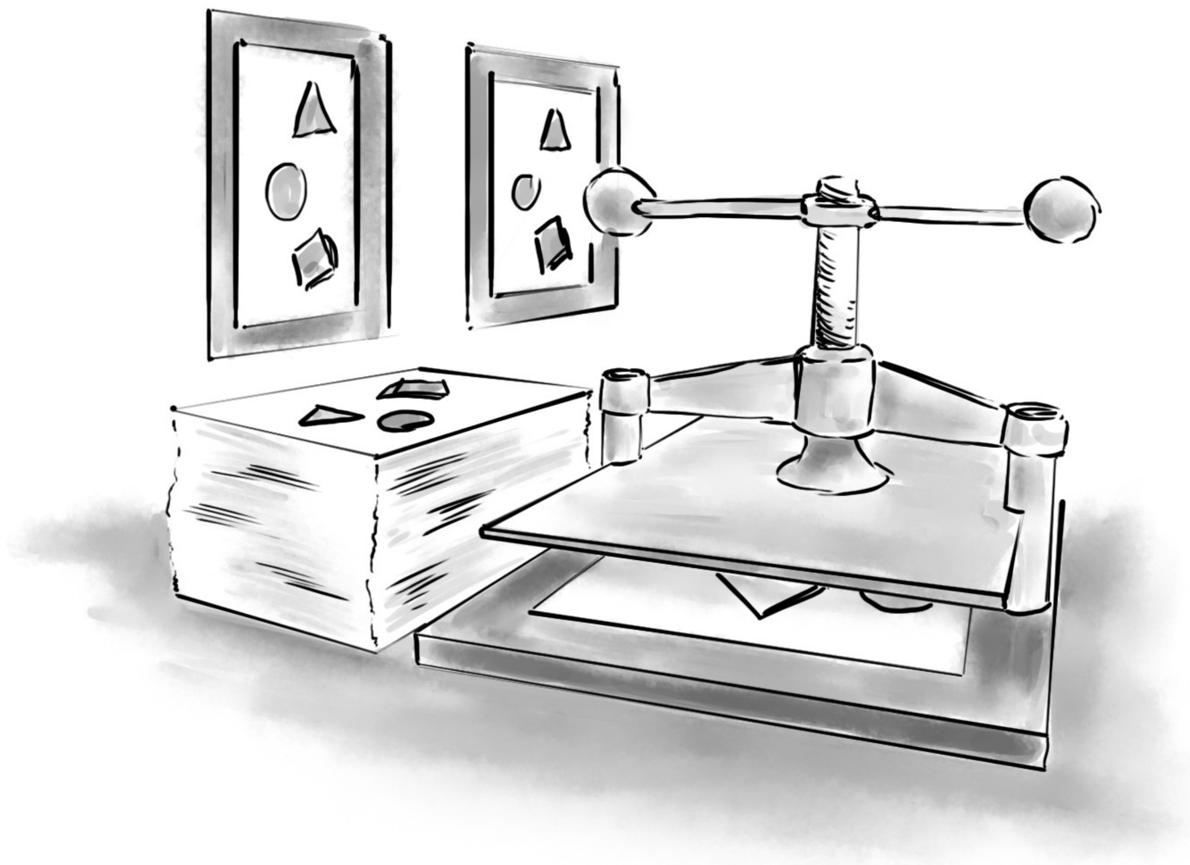
Een andere manier is om een woord uit de tekst te vervangen door de score. Je maakt dan eerst de tekst met op de plek waar de score moet komen een tijdelijke markering zoals ‘<score>’ (maar je mag ook zelf wat verzinnen). Vervolgens stuur je die plaatsvervangende tekst als bericht naar het tekstobject 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!’].
```



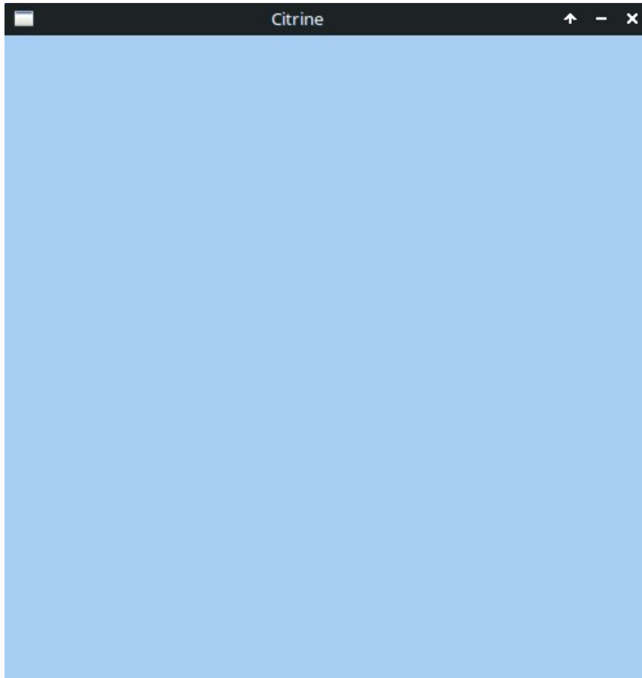
# 2 Objecten

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’].
```



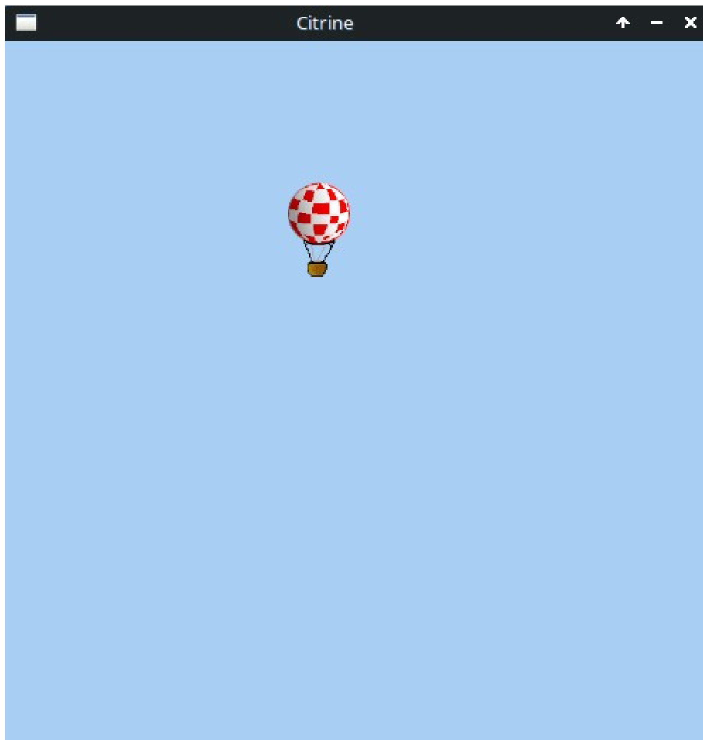

Als je dit programma uitvoert (en het afbeeldingsbestand lucht.png is aanwezig in dezelfde map), wordt er een venster geopend met als achtergrondplaat de lucht. Het venster blijft open totdat de gebruiker op het kruisje klikt. Het programma blijft ook ‘hangen’ bij het bericht ‘scherm:’ totdat de gebruiker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

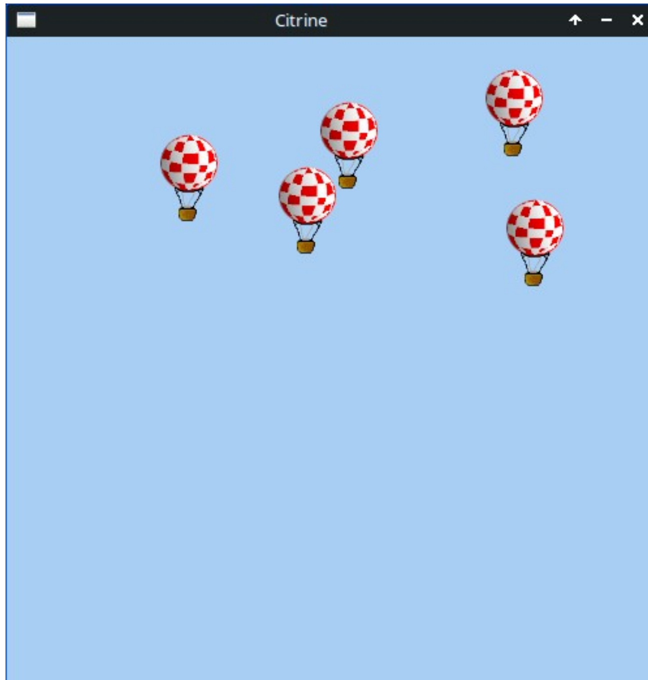
Een oplettende (of gevorderde) programmeur zal zich misschien afvragen hoe het komt dat het programma hier niet in een eindeloze lus terecht komt. Immers, ‘Ballon nieuw’ roept ‘Ballon nieuw’ aan. Goed gezien! Toch is hier geen sprake van een eindeloze lus. In de eerste plaats omdat we in ‘nieuw:’ sturen 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’:

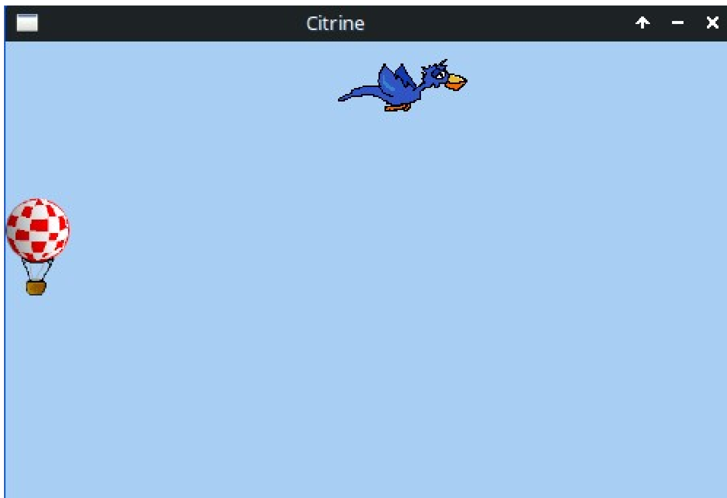
- **Kikker die een drukke weg oversteekt (bovenaf):**
  besturing: 1.
- **Springend konijn (platformspel):**
  besturing: 1, zwaartekracht: 1, springhoogte: 2.
- **Zwevend ruimteschip (2D schietspel):**
  besturing: 1, zwaartekracht: 0.5.
- **Tennisspel (alleen verticaal):**
  besturing: 2.
- **Sjoelen (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.
```

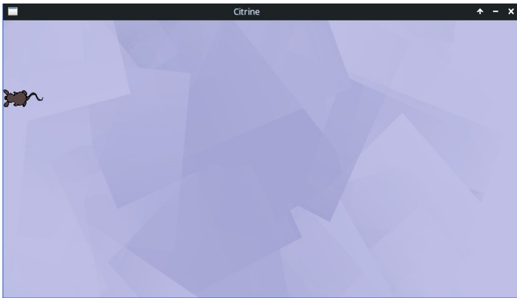
Het is ook mogelijk om een filmpje automatisch te laten afspelen, zonder dat het beweegt, dat is bijvoorbeeld handig voor een kampvuur. Je stuurt dan: `autospeel: Ja`.

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’.

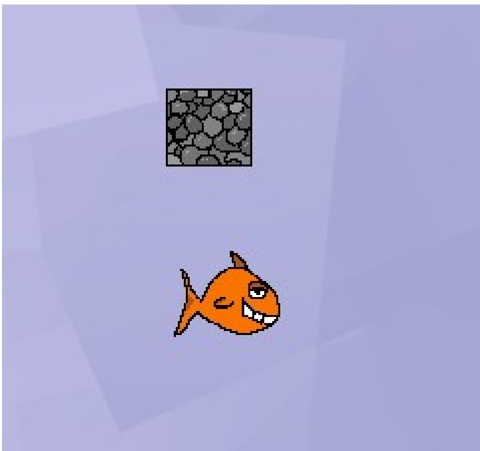
Als je een digitale schiettent wilt maken waarbij de muiscursor als vizier dient hoef je geen besturingsbericht te sturen. Bij een muisklik ontvangt een ‘actief’ plaatje namelijk het bericht ‘klik’. Het mediaobject zelf ontvangt ‘klik-x:y:’ bij elke klik. Dit kun je ook gebruiken om klikjes op knoppen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

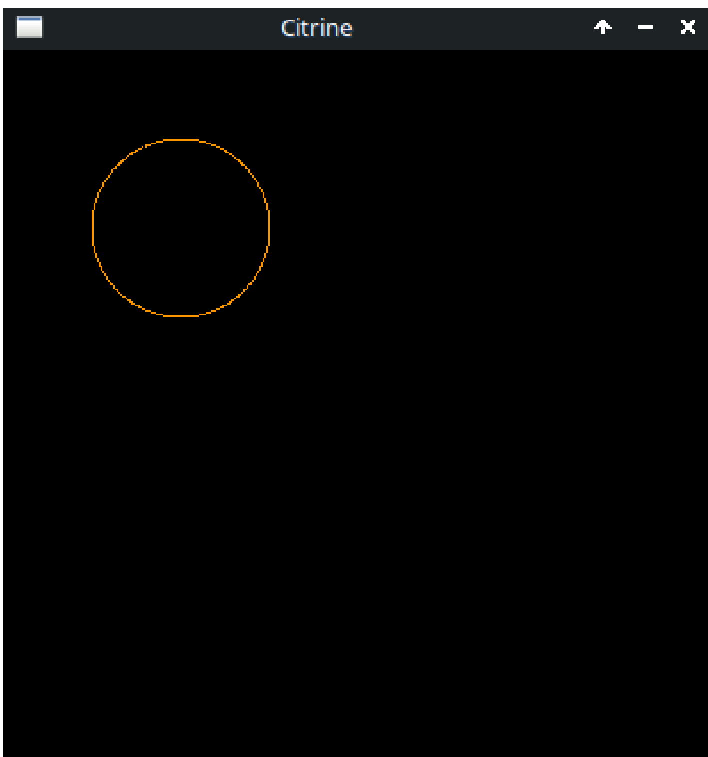
Afhankelijk van je achtergrondafbeelding voor de score en het lettertype, krijg je iets zoals dit:

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.
```

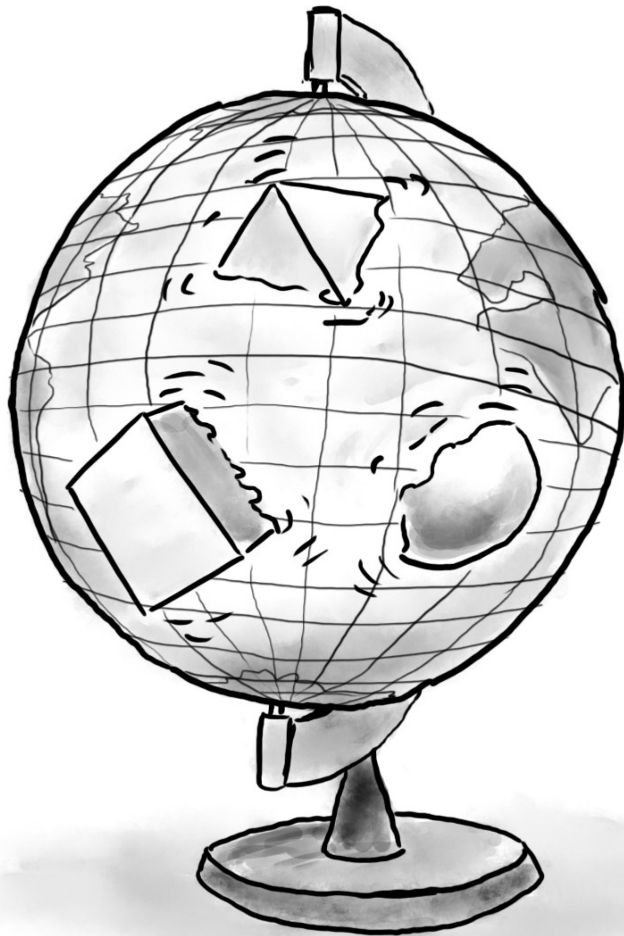
In plaats van code, afbeeldingen of muziek uit bestanden te laden, kun je ze ook toevoegen aan een datapakket en je programma ze uit dat pakket laten ophalen. Het voordeel van een datapakket is dat je middelen gebundeld zijn in één, ondoorzichtig pakket (zodat mensen de afbeeldingen niet kunnen bekijken 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.
```

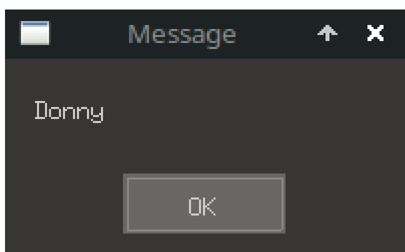
Ja, dit is mogelijk in Citrine (en voor zover ik weet, in geen enkele andere taal). Als je dit doet, is het gegarandeerd dat alles kapot gaat. Kortom, het is volledig onvoorspelbaar wat je programma zal doen nadat je deze code hebt uitgevoerd. Geen idee. De reden dat dit werkt is simpel: := maakt nooit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

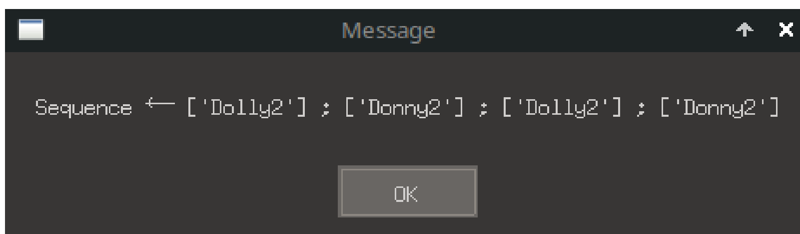
Om een echte kopie van een object te maken, moet je er expliciet om vragen:

```
>> 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.
```

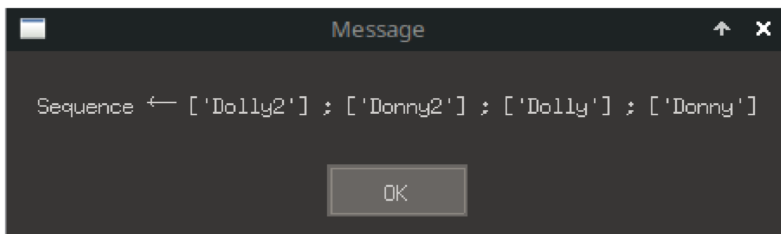
Resultaat:




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’]. }.
```

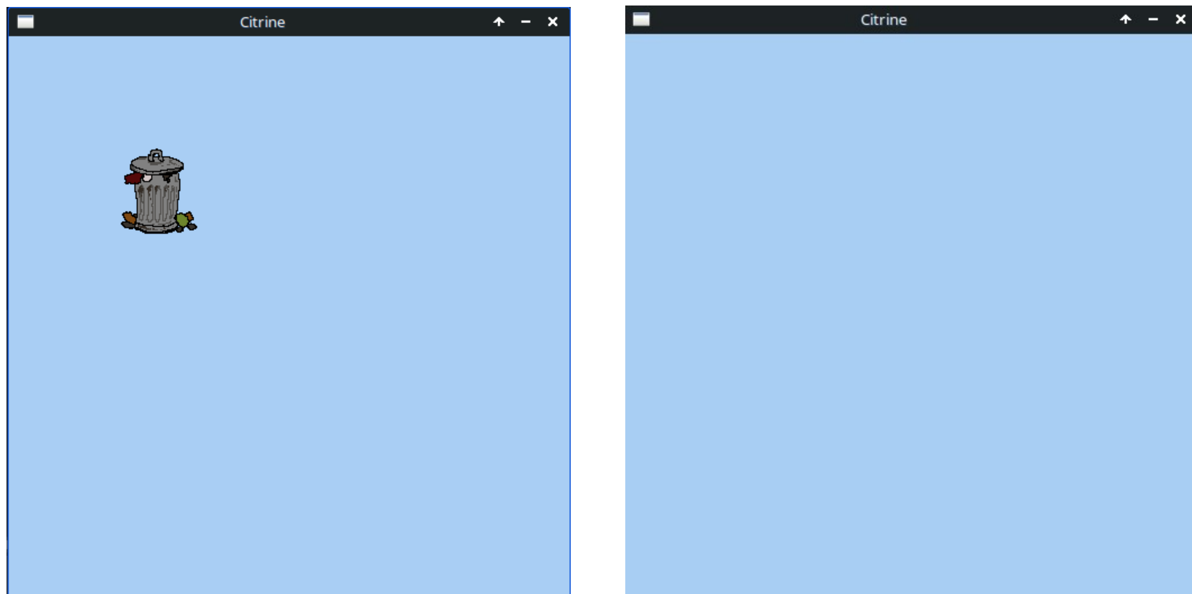
In dit geval geldt dezelfde regel. Als je wilt voorkomen dat de variabele verloren gaat, kun je deze aan het object koppelen door het een eigenschap te maken:

```
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`:

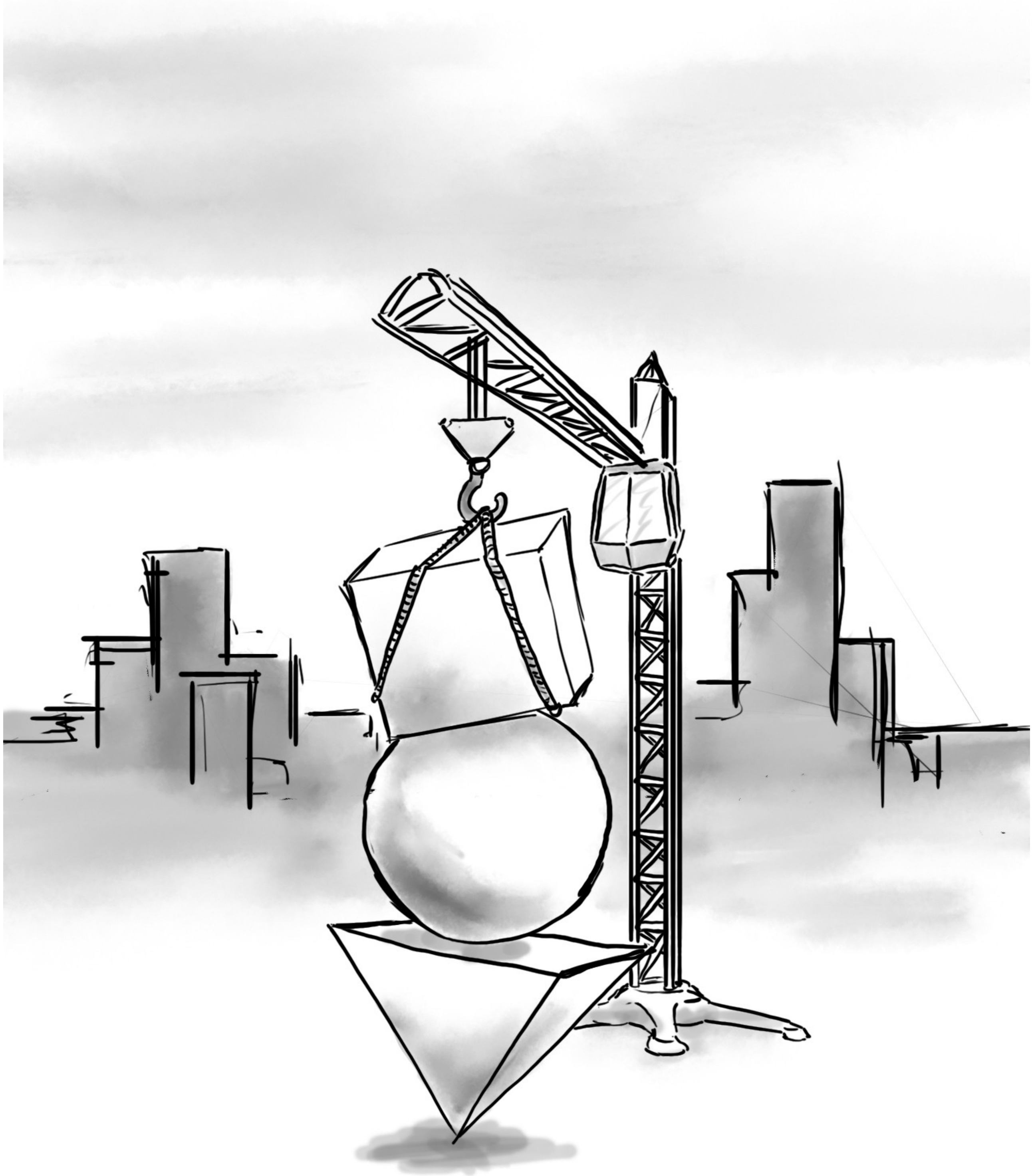
```
>> bedrag := 6 muntjes.

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

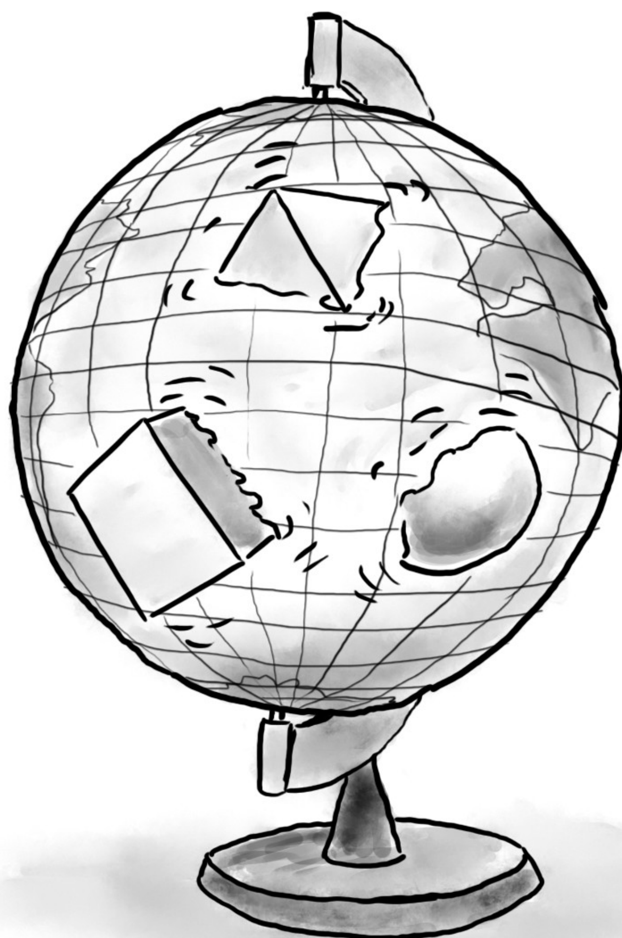
6. Het bericht waar deze functie aan gekoppeld moet worden

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.



# 5. Naslag

In dit hoofdstuk geef ik een overzicht van welke objecten er allemaal zijn in de wereld van Citrine/NL, en welke berichten je allemaal kunt sturen naar deze objecten. Dit is dus meer een naslagwerk. Je kunt hier even snel dingen opzoeken!

## 5.1 Overzicht objecten en berichten

| Object | Bericht | Wat doet het? |
|---|---|---|
| Uit | schrijf: | Schrijft tekst naar het uitvoervenster |
| | stop | Maakt een nieuwe regel in het uitvoervenster |
| Niets | Niets? | Antwoord met Ja, alle andere objecten antwoorden met Nee. |
| | getal | Geeft 0 |
| | tekst | ‘Niets’ |
| | bool | Geeft Nee als antwoord |
| Object | type | Geeft de naam van het objecttype |
| | kopieer | Maakt een kopie |
| | geval:doen: | Hiermee kun je een taak laten uitvoeren als het object gelijk is aan een bepaalde waarde.<br><br>Hier is een voorbeeld:<br><br>`>> x := ['**'].`<br>`x`<br>`geval: ['*'] doen: { Uit schrijf: 1. },`<br>`geval: ['**'] doen: { Uit schrijf: 2. },`<br>`geval: ['***'] doen: { Uit schrijf: 3. }.`<br>`Uit stop.` |
| | bij:doen: | Met dit bericht voeg je functionaliteit toe aan het object. |
| | bericht:<br>bericht:argument:<br>bericht:argument+... | Hiermee stuur je ook een bericht naar een object, alleen kun je nu een bericht sturen dat in een variabele zit. |
| Boolean | ja:/nee: | Hiermee laat je Ja of Nee een taak uitvoeren. |
| | of:/en:/noch: | Hiermee kun je taken uitvoeren als X of Y waar is etc. |
| | afbreken/doorgaan | Afbreken van een lus of doorgaan naar de volgende iteratie van de lus. |
| | niet | Dit maakt van Ja een Nee en andersom |
| Getal | + - / * > < >=: <=: = !=: | Rekenkundige bewerkingen en vergelijkingen |
| | optellen:, aftrekken:, gedeeld:, keer: | Hetzelfde maar dan zonder een kopie te maken |
| | tussen: en: | Hiermee krijg je een willekeurig getal. |
| | sin cos tan atan log | Wiskundige operaties |

| | & \| ^ | Binaire operaties |
|---|---|---|
| | ruw | Geeft een tekstuele representatie van het getal zonder scheidingstekens (punten). |
| | positief? / negatief? | Geven Ja/Nee terug als het getal positief of negatief is |
| | macht: | Machtsverheffen |
| | vierkantswortel | Berekent de vierkantswortel van het getal |
| | kwalificering: | Hiermee hang je een tekst aan het getal, zo weet je bijvoorbeeld dat het getal opgegeven is in een bepaalde eenheid zoals liters of meters |
| | hoog/laag/afgerond | Afronden naar boven of naar beneden of afhankelijk van het getal |
| Tekst | > < >=: <=: | Vergelijk de ene tekst met de andere |
| | opsplitsen: | Hiermee maak je van een tekst een reeks, de tekst wordt opgesplitst met het gegeven splitsteken |
| | van:lengte: | Knipt een fragment uit de tekst |
| | overslaan: | Knipt de eerste X tekens af |
| | lengte | Het aantal letters in je tekst |
| | bytes | Het aantal bytes dat je tekst lang is |
| | vervang:door: | Vervangt een stuk uit je tekst door een andere tekst |
| | zoek: | Geeft de positie van fragment uit tekst (of Niets) |
| | bevat: | Geeft Ja terug als fragment in tekst voorkomt |
| | laatste: | Zelfde als zoek, maar zoekt vanaf de rechterkant |
| | letter: | Geeft een kopie terug van de letter op de gegeven positie |
| | kern | Knipt omliggende spaties af |
| | letters | Geeft een reeks met alle letters in je tekst |
| | - | Knipt tekst af vanaf de rechterkant |
| | hoofdletters | Geeft kopie in hoofdletters |
| | kleine-letters | Geeft kopie in kleine letters |
| Taak | start | Start de taak |
| | * | Voert de taak een aantal keer uit (* 10) |
| | zolang: | Voert de taak net zolang uit todat de rechtertaak Nee antwoordt |
| | afhandelen: | Koppelt een foutafhandelingstaak aan je taak |
| Reeks | nieuw | Maakt een nieuwe, lege reeks |
| | ; | Voegt een object toe aan je reeks |
| | toevoegen: | Voegt een object toe aan je reeks |
| | aantal | Telt het aantal elementen in je reeks |

| | samenvoegen: | Voegt de elementen van je reeks samen tot een Tekst, met een lijmteken |
|---|---|---|
| | elk: | Voert een taak uit voor elk element in de reeks |
| | per: | Maakt van linker en rechter reeks een lijst |
| | zoek: | Zoekt het object in de reeks en geeft de positie |
| | positie: | Geeft het object op de positie |
| | zet:bij: | Zet een object op positie |
| | invoegen: | Net als toevoegen maar dan aan het begin van de reeks |
| | sorteren: | Sorteert de reeks met een taak |
| | vervang:lengte:door: | Vervangt een deel van de reeks door een andere reeks |
| | vul:met: | Vult X elementen in de reeks met een object |
| | kopieer | Kopieert de reeks |
| | van:lengte: | Kopieert een deel van de reeks |
| | + | Breidt de reeks uit met een andere reeks |
| | eerste laatste voorlaatste | Geeft het eerste, laatste of voorlaatste element uit de reeks terug |
| | lknip rknip | Knipt het linkerelement of rechterelement af |
| Lijst | nieuw | Maakt een nieuwe lege lijst |
| | zet:bij: | Zet object op plek in de lijst |
| | bij: | Geeft het object dat bij de zoekterm hoort |
| | elk: | Voert een taak uit voor elk object in de lijst |
| | ingangen | Geeft alle zoektermen/sleutels uit de lijst als reeks |
| | waarden | Geeft alle waarden in de lijst als reeks |
| | aantal | Telt het aantal waarden in de lijst |
| | heeft: | Geeft Ja als deze waarde in de lijst zit |
| | - | Haalt object met sleutel uit de lijst |
| Bestand | nieuw: | Maakt een nieuw bestand dat naar een pad verwijst |
| | lezen | Geeft de inhoud van een bestand |
| | schrijf: | Schrijft tekst naar bestand |
| | verwijderen | Verwijdert het bestand |
| | toevoegen: | Voegt tekst toe aan bestand |
| | bestaat | Geeft Ja als het bestand al bestaat |
| | grootte | Geeft de grootte van het bestand in bytes |
| Moment | nieuw | Maakt een nieuw Momentobject |
| | wacht: | Wacht X seconden |

| | | |
|---|---|---|
| | zone: | Stelt de tijdzone in |
| | zone | Haalt de tijdzone op |
| | jaar maand dag uur minuut weekdag jaardag | Geeft de tijdseenheid van het moment terug, i.e. welke dag etc. Met dubbele punt zet je de tijdseenheid. |
| | optellen: aftrekken: | Tel een andere tijd op of trek een andere tijd af |
| | tijd | Geeft de tijd van het moment als seconden sinds 1 januari 1970 om 00.00 (Unix time) |
| Programma | gebruik: | Laad een extern programma in als onderdeel van het huidige programma |
| | einde | Stopt de uitvoering van het huidige programma |
| | opdrachregel: | Laat een opdracht door het besturingssysteem uitvoeren |
| | aantal-argumenten | Geef het aantal argumenten terug waarmee het programma is aangeroepen |
| | argument: | Geeft een specifiek argument terug |
| | instelling: | Geeft de waarde van een omgevingsvariabele terug |
| | instelling:is: | Stelt een omgevingsvariabele in met een bepaalde waarde |
| | invoer | Geeft de standaardinvoer van het programma terug als tekst |
| | vraag | Stelt een vraag op de opdrachtregel en wacht op antwoord |
| | vraag-wachtwoord | Zelfde alleen dan zonder toetsaanslagen te tonen |
| | fout: | Schrijft een tekst naar het standaard foutkanaal van het programma |
| Media | nieuw | Maakt een nieuw Media-object |
| | toon: | Toont tekst in dialoogvenster op het scherm |
| | scherm: | Toont afbeelding in venster als achtergrond |
| | website: | Gaat naar website |
| | koppel: | Koppelt een datapak pakket of FFI-commando |
| | selectie | Geeft geselecteerde tekst terug |
| | klembord | Geeft tekst in het klembord terug |
| | klembord: | Zet tekst in het klembord |
| | breedte:hoogte: | Stelt omvang van het venster in |
| | wekker:over: | Stelt een wekker (nummer) in over X aantal seconden |
| Plaatje | nieuw: | Laad een plaatje van pad |
| | bron: | Je mag de bron van het plaatje ook later, na nieuw opgeven of veranderen |

| | | |
|---|---|---|
| | x:y: | Zet het plaatje op een positie op het venster |
| | x? y? | Haalt x of y van plaatje op |
| | besturing: | Maakt het plaatje bestuurbaar via pijltjestoetsen, gamepad of joystick |
| | actief: Ja | Maakt het plaatje actief, dan krijgt je plaatje een bericht als het bijvoorbeeld tegen iets anders aan botst (bots:) |
| | zwaartekracht: weestand: snelheid: zichtbaar: versnel: springhoogte: | Instellingen van je plaatje voor een spel |
| | muur: | Ja = plaatje is een muur |
| | spook: | Ja = plaatje kan door muren heen |
| | naar-x:y: | Beweeg het plaatje automatisch naar positie |
| | filmrol: snelheid: | Maakt van je plaatje een klein filmpje met X deelplaatjes die met snelheid Y bewegen |
| | autospeel | Filmrol automatisch afspelen |
| | tekst: inkt: stift: uitlijnen-x:y: regelhoogte: beschrijfbaar: lettertype: toevoegen: | Tekstinstellingen voor je plaatje |
| | tekenen: | Tekent de objecten in de reeks op je plaatje (Punten en Lijnen) |
| Punt | nieuw... | Maakt een nieuw punt: Punt nieuw x: 10 y: 10 |
| Lijn | nieuw... | Maakt een nieuwe lijn: Lijn nieuw vanuit: punt1 naar: punt2 |
| Geluid/Muziek | nieuw... | Maakt een nieuw geluidje of muziekje:<br>>> m:= Muziek nieuw: ['lalala.mp3'] |
| | afspelen | Speelt geluid of muziekje af |
| | stil | (alleen voor muziek) |
| | terugspoelen | (alleen voor muziek, begint opnieuw) |
| Netwerk | nieuw | Maakt een nieuw netwerkobject aan waarmee je gegevens kunt uitwisselen met het internet |
| | stuur:naar: | Stuurt tekst naar internetadres en geeft antwoord van website. Als je niks stuurt haal je alleen informatie op. |
| JSON | van-json: | Maakt lijst van json-tekst |
| | json-maken: | Maakt json code van lijst |
| Blob | nieuw: | Maakt een nieuw blok geheugen aan met aangegeven grootte in bytes (FFI) |
| | utf8: | Maakt geheugenblok van tekst (FFI) |

| | | |
|---|---|---|
| | struct: | Maakt een geheugenstructuur voor gebruik in C/FFI met indeling als in reeks |
| | van:lengte: | Kopieert fragment uit geheugenblok in een nieuw geheugenblok (FFI) |
| | vul: | Vult geheugenblok met bytes uit reeks (FFI) |
| | vrij | Geeft geheugenblok vrij (FFI) |
| | deref | Derefenced een pointer naar geheugenblok (FFI) |
| | freestruct | Geeft struct vrij (FFI) |

**Gebeurtenissen**

Als er in de app/game iets gebeurt krijg je een signaal, zodat je je programma daarop kunt laten reageren.

| Ontvanger | Gebeurtenis | Wanneer? |
|---|---|---|
| Media | start | Nadat het venster klaar is voor gebruik (nadat je Media scherm: hebt aangeroepen) |
| | wekker: | Er gaat een wekker af, als argument krijg je het nummer van de wekker mee |
| | klik-x:y: | Er is ergens op het scherm geklikt, je krijg de x en y door |
| | toetsin: | Er is een toets op het toetsenbord ingedrukt (en deze is nog steeds ingedrukt), de code van de toets wordt meegestuurd |
| | toets: | Er is een toets op het toetsenbord ingedrukt en weer logeslaten, de code wordt meegestuurd |
| | gamepadin: | Er is een knop op de gamepad in gedrukt (of joystick), de knop is nog steeds ingedrukt, de code van de knop wordt meegestuurd |
| | gamepad: | Er is een knop op de gamepad in gedrukt (of joystick), de knop is inmiddels losgelaten, de code van de knop wordt meegestuurd |
| | stap | Er is 1 kloktik in het spel verstreken |

| | | |
|---|---|---|
| | einde | Nadat het venster is afgesloten. |
| Plaatje | bestemming | Het plaatje heeft de bestemming x/y bereikt |
| | bots: | Het plaatje is gebotst tegen een ander plaatje, dat andere plaatje wordt meegestuurd als argument |
| | klik | Er is op het plaatje geklikt |