

*Programming with
Objects & Abstractions in*

CITRINE



Gabor de Mooij

Programming with Objects & Abstractions in Citrine

Copyright © 2023 Gabor de Mooij

Author: Gabor de Mooij

Translation: Bernadette Peeters

Illustrations: Robert Cabri

Cover illustration: Robert Cabri

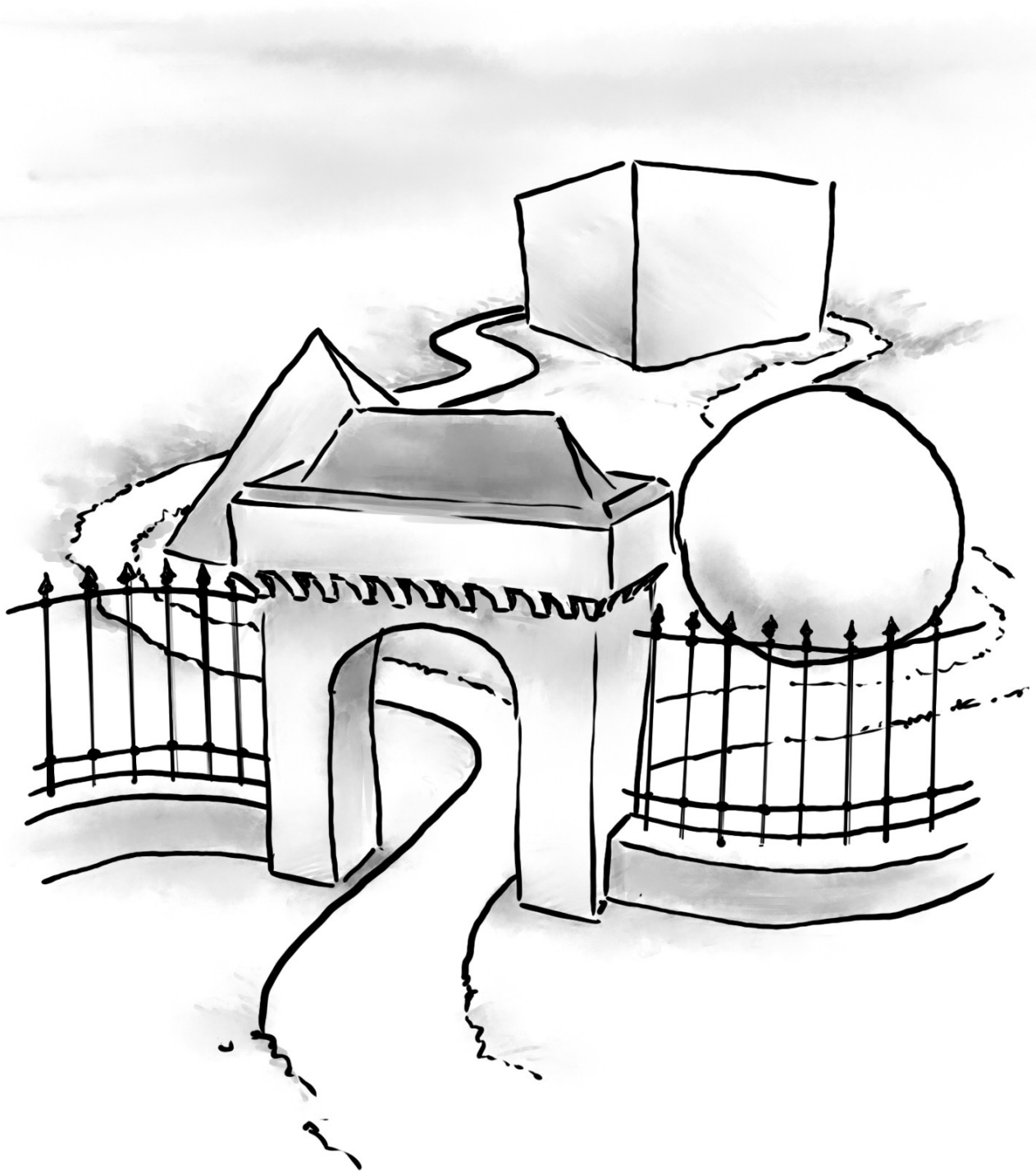
1st Edition 2023, Digital PDF

2nd Edition 2024, Digital PDF

Version History		
2023	1.0	Initial version
2024-12-29	2.0	ASCIIfication, fixed fonts, updated code fragments, fixed some mistakes. Removed some chapters that are no longer relevant.
2025-01-09	2.1	Abort → break
2025-01-13	2.2	Decision → boolean
2025-01-21	2.3	Per: → By:

Index

1. Introduction.....	10
1.1 What is Citrine?.....	11
1.2 Hello World.....	12
2. Rules for Writing.....	16
2.1 Actions.....	17
2.2 Variables.....	18
2.3 Objects and Messages.....	19
2.4 The Flow of the Program.....	22
2.5 Templates.....	24
2.6 Answering.....	26
2.7 Inheritance and Properties.....	28
2.8 Exercises.....	34
3. Objects.....	39
3.1 Simple Objects.....	40
3.2 Boolean Object.....	42
3.3 The Number Object.....	47
3.4 The Text Object.....	51
3.5 Conversions.....	57
3.6 Task Objects.....	58
3.7 The Root Object.....	64
3.8 Sequences.....	68
3.9 Lists.....	76
3.10 Files.....	83
3.11 Moments.....	87
3.12 The Program Object.....	95
3.13 Excercises.....	105
4. Advanced.....	109
4.1 Copying.....	110
4.2 Visibility.....	115
4.3 Handling Unknown Messages.....	118
4.4 Inheritance, Overriding, Recursion.....	121
4.5 Initial State.....	125
4.6 Implicit Conversion.....	128
4.7 Serialisation.....	134
4.8 Alternative Message Structure.....	138
4.9 Programmatically-sent Messages.....	140
4.10 Modules.....	143
4.11 Detection.....	145
4.12 Excercises.....	148
5. Translate.....	155
5.1 Translation System.....	156
5.2 Dynamic translations.....	158
Appendix A: AST-export.....	163
Appendix B: Answers.....	166
Answers chapter 1.....	167
Answers chapter 2.....	168
Answers chapter 3.....	170
Answers chapter 4.....	172
Appendix C: Memory Management.....	177
Register.....	178



1. Introduction

1.1 What is Citrine?

The ambition of the Citrine project is to develop a simple programming language that improves software maintainability, accuracy and readability, and that minimises the number of software errors. In order to achieve this, the project has three basic principles: simplicity, localisation and uniformity. Simplicity means that the programming language consists of a notably small selection of grammar rules and concepts. In fact, Citrine shuns data types, classes and string interpolation and it takes only three essential actions to make the program run. Localisation means that the programming language can be written in any native language. Citrine/NL, for example, is used for the Dutch language.

The opportunity to write software in any native language can be regarded as the ultimate objective in the attempt to enhance code readability. In addition, clients and stakeholders will be able to check and review the programmer's work. Uniformity means that Citrine, contrary to current programming languages, aspires to ban sublanguages such as markup languages (HTML), regular expressions, file path formats, database dialects (SQL), and network protocols such as JSON, XML or YAML.

Citrine is built on the premise that when everything is written in the same localised language, having knowledge of other code languages becomes inessential. These three basic principles strengthen one another; for example, grammar minimalism allows for more notation freedom, which in turn, serves localisation. Vice versa, the principle of localisation boosts uniformity, since all code is written in the same language. Uniformity and localisation both reduce the cognitive strain that is required when reading computer programs, which, in turn, promotes the process of simplification.

Citrine can be used by those programmers who seek a simple programming language that allows them to write code in their own first language and who recognise the previously mentioned benefits. In addition, Citrine can be used as a domain-specific language (DSL) to extend a graphical user interface, which allows the advanced user to automate matters. In addition, tutors can use Citrine to teach the basic principles of how to write code, without language being an issue.

Notably, this manual is meant to serve advanced developers and clarifies the coding rules for the programming language Citrine. The manual specifically focuses on the English version of Citrine, i.e., Citrine/EN. In the following chapters, the term Citrine/EN will therefore be replaced with Citrine.

1.2 Hello World

After it has been transferred to your computer, you can extract the file and start Citrine by using the following command:

```
ctren
```

After starting Citrine, the welcome screen is displayed:

```
Citrine/EN  
Written by Gabor de Mooij © all rights reserved 20XX  
Licensed BSD.  
X.X.X
```

This shows that Citrine has been correctly installed. The X.X.X will be replaced by the software version number.

To run a Citrine program, store the program as a file by adding -ctr suffix (optional). In addition, insert the file name of your program after the Citrine command:

```
ctren myprogram.ctr
```

When you run a Citrine command while using your program file as a parameter, Citrine will read and execute your program file. In addition, Citrine introduces 4 extra functions:

Optie	Voorbeeld	Functie
-t	<pre>ctrnl -t <dictionary> <program></pre>	Translates the program file using the entered dictionary file
-g	<pre>ctren -g <header> <header></pre>	Generates a dictionary file for basic vocabulary based on two header files (dictionary.h)
-x	<pre>ctren -x <program file></pre>	Exports an abstract image (AST) of the program (see: Appendix A)

Translate your program files by using the **-g** and **-t** options. See chapter 5 for further information. Export the structure of a Citrine program by using the **-x** option. This can be used to convert Citrine programs into other programming languages such as *C* and *Java*. For additional information, see Appendix A.

Citrine is open-source software and released under BSD2-license. This is one of the most simplified and concise licenses, which means that you may use Citrine for both personal and commercial ends, as long as the license agreement and the authors are specified. There are no further restrictions. For a complete overview of BSD2 license go to:

<https://citrine-lang.org/download.ctr>

and

<https://opensource.org/licenses/BSD-2-Clause>

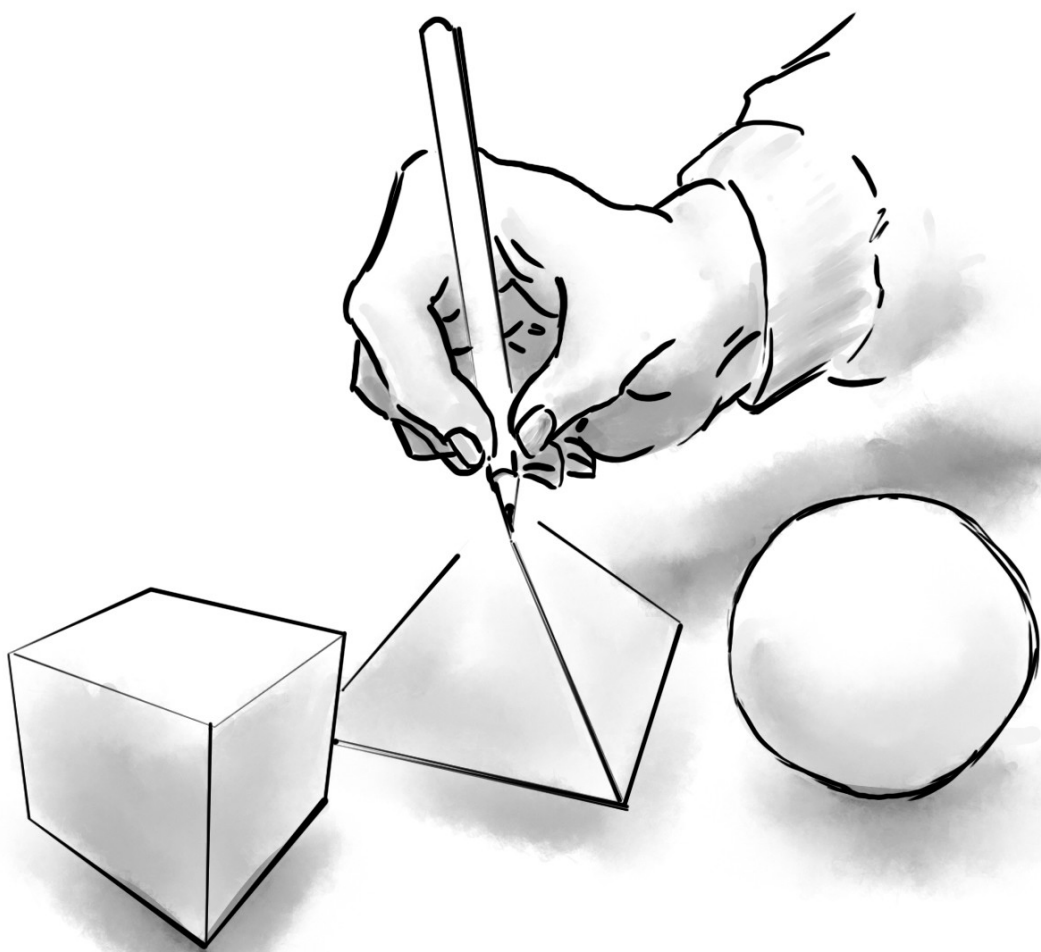
Tradition dictates that the first program in any new programming language should be a kind of salute to the world. In Citrine, such a **Hello world** program looks like this:

```
Out write: ['Hello world'], stop.
```

When this text is saved to a file, e.g. `hello.ctr`, it can be executed as follows: .

```
ctren hello.ctr  
Hello world
```

You can see the program output in the above-shown black window. Throughout this manual, the same visualisation will be applied to show the output. Programming code of Citrine programs are printed in the Citrine font, equal to the example shown above.



2. Rules for Writing

2.1 Actions

Citrine is a *pure object-oriented* programming language. This means that Citrine perceives everything to be an object, therefore there are no other *data types*. There are essentially three basic routine actions in a program that is written in Citrine: *assigning*, *sending messages* and *responding*. Exchanging messages between objects is the key part of a Citrine program. The three individual routine actions are illustrated below:

Action	Example	
1. Assign	>> x := 1.	
2. Messages	Unary	x even?.
	Binary	1 + 2.
	Keyword	x from: 0 length: 10.
3. Answer	<- answer.	

This chapter will give you a general impression of the language Citrine and describes its basic principles. Citrine is a small language and founded on three actions (see illustration) which translate into approximately six grammar rules. That is all it takes to master the language.

No worries if things are not instantly obvious to you: all building stones will be explored in the next chapter (3) and the basic rules will be repeated. Current chapter, however, serves mainly as a general introduction to the language. The basic principles will be briefly explained and illustrated by several relevant examples. Any missing details will be outlined at a later moment.

Comments in Citrine are prefixed with a # symbol. So the following line will be ignored:

```
# this is just a comment
```

2.2 Variables

The first step to take, is to assign an object to a variable. To assign a variable you use the variable declaration symbol: `>>`. The name of a variable may contain all signs except for: `<-`, `:=`, whitespaces, periods, commas, colons, quotation marks `["]`, and parentheses `()`. Furthermore, a variable cannot consist of multiple lines. Please note that a variable cannot begin with a number or a minus sign.

These are examples of valid variables:

```
>> password := ['Secret'].
>> ♥♥♥ := 3.
>> $ := ['dollar'].
>> +plus := True.
>> user password := ['Pssst!'].
```

Invalid variables are for example:

```
>> -123 := ['negative number'].
>> password of user := ['Classified'].
>> password.of:user := ['Classified'].
>> ,x := 10.
```

After a variable has been declared, it can be used freely. This means that it is only necessary to use the declaration symbol the first time you use the variable (the declaration).

To be more specific, you cannot randomly start to declare a value to your variable (`x := 2`), as you must first declare the variable (`>> x := 2`). However, once declared you are allowed to change its value without using the declaration symbol (`>> x := 2. x := 3.`).

Note that it is mandatory to assign a value to a variable at the time of declaring. Contrary to other programming languages, you are not allowed to declare a variable without value. It is essential to explicitly tie each variable to an initial value. Nonetheless, feel free to initialise the variable with the **None** object. In this case, the variable can be regarded as being empty (`>> x := None.`). In short, to declare a variable without an initial value is not allowed. So,

```
>> x
```

is invalid and will give an error message. (Find out more about the **None** object in chapter 3).

2.3 Objects and Messages

Citrine perceives everything to be an object; i.e., all numbers, texts and code fragments. Numbers, such as 1, 2, 100, -999 and 1,234 are Number objects. All texts between single quotation marks are Text objects. All code fragments grouped between curly brackets {...} are Task objects.

Name	Root object	Example
numbers	Number	1,2,3.... -100 1,5 1.000.000
texts	Text	['Brevity is the soul of wit']
tasks	Task	{ 1 + 2. }.

Objects such as numbers and texts, always find their origin in a root object. For example, all numbers derive from the object **Number**. All texts derive from **Text** and all code fragments from **Task**. In turn, all these objects stem from **Object**, which is, in fact, the root object of all objects. Programming in Citrine basically means sending messages to objects. The general notation to send a message to an object is as follows:

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

To find out if the number **2**, for example, is an even number, the message **even?** is sent to object **2**:

```
2 even?
```

The answer will be **True** (again, an object). Unknown messages are usually ignored by objects; so no error will occur. Some objects (e.g., numbers or texts) respond to an unknown message in a predefined way (more on this subject later).

An object can receive three kinds of messages. First, there are **unary messages**, see example above, which are without arguments. Second, there are **keyword messages**, which have one or multiple arguments, for example:

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

In this case the message **between:and:** is sent to **Number**, which is the root object of all numbers. The result will be a random number between 1 and 10. Finally, there are **binary messages**, which have only one character and one argument:

2 + 3

This looks like a math sum, but it actually just another message. The message + is sent to 2, with argument 3 which will return answer 5. Binary messages are allowed to be written without a colon.

Binary messages can be chained:

>> x := 3 + 2 - 1.

In this fragment + 2 is first sent to **Number** object 3, which results in Number object 5, after which -1 is sent to this number. Observant readers are correct to notice the discrepancy which this protocol shows regarding the conventional mathematical sequences of operators. Citrine ignores the mathematical sequence in favour of consistency in its message system. As a result, the sum:

2 + 3 * 5 = 25

not 17.

This is by design. Parentheses can be used to modify the sequence order:

2 + (3 * 5) = 17

Most objects return themselves as a result in response to a certain message. This is helpful as it encourages further dialogue with this object by sending it a follow-up message:

```
Out write:
  [' hello '] trim capitals.
```

```
HELLO
```

Here, two messages are sent to the **Text** object: **trim**, followed by **capitals**.

In the following fragment, the use of a comma is necessary to indicate that a new message is incoming. If not, Citrine will get confused.

```
Out write: ['Hello!'], stop.
```

First, the message **write:** is sent to the pencil symbol followed by **stop**. Without the comma, Citrine would think that you wish to send **stop** to **Text Hello!**, a futile exercise.

The process sequence of messages is as follows: from left to right; start with messages in parentheses followed by unary messages. Next, binary messages and then end with keyword messages. See the example below:

```
Out write: 0.5 round + (2 - 1), stop.
```

Citrine always reads from left to right: first, the message **write:** is sent to Out and is followed by the **stop** message. Within the argument itself, Citrine reads from left to right, so **0.5 rounded** then **+**.

Moreover, **rounded** takes precedence over **+**, because it is a **unary message**. Given the parentheses, **2 - 1** is calculated first, after which **1** is added to the result of **0.5 rounded** (1).

The whole process will give the result **2**, which is also the written answer. After the comma follows the stop message, which moves the cursor to a new line. Of course, the best way to fully comprehend the sequence order of Citrine programs is with practise.

It is worthy to note that in Citrine, contrary to many other programming languages, whitespaces are a fundamental part of its syntax. In particular when it comes to binary messages, whitespaces may cause some confusion. Always use a whitespace after a binary message. You cannot directly attach a number to the message, for example: **3 + 2** is different from **3 +2**. The first example (**3 + 2**), sends the message **+** to the number **3**, with argument the number **2**. The result, in this case, will be **5**. In the second example (**3 +2**), the unary message **+2** is sent to number **3**. Depending on the context that might yield a very different result.

2.4 The Flow of the Program

In Citrine there is no need for separate grammar rules for *loops* and *ifs*. An if-statement is just a simple **true:** or **false:** message to either a **True** object or a **False** object:

```
2 even? true: {  
    Out write: ['Two is an even number'], stop.  
}.
```

True and **False** objects are also referred to as decision-making objects or booleans. This naming is sometimes more accurate, because it specifies that these object are used to make the program handle certain decisions. Equal to other programming languages, a program written in Citrine follows a specific course and entails various patterns of branching. The selected ways within a program largely rely upon these decision-making objects.

In the previous example, the message **true:** is sent to **True**, (the answer to the question: is 2 an even number?) As argument an additional fragment of code is sent along: a **Task** object. This task writes on the screen that 2 is, in fact, an even number.

There is no need either for separate writing rules for a *loop*. To execute a fragment of code three times, simply send with argument 3 to that **task**:

```
{ Out write: ['♣'], stop. } * 3.
```



Let's consider another example. When a conversion table needs to be printed from *kJ* (kilojoule) into *kcal* (kilocalorie) by steps of 100, see below for the correct notation:

```
{ :line  
    >> kJ := line * 100.  
    Out write: kJ, stop.  
    Out write: kJ * 0.239, stop.  
} * 10.
```

This program will generate the following list:

```
100
23.9
200
47.8
300
71.7
400
95.6
500
119.5
600
143.4
700
167.3
800
191.2
900
215.1
1,000
239
```

Here, the number of the current line is transmitted to the parameter **:line**. At the beginning of a task, the task parameters are being defined. Unused parameters remain empty (**None**). Parameters are always placed at the start of the task, directly after the initial bracket and are preceded by a colon.

The message **while:**, is a combination of a *loop* and a *condition*. Two tasks can be linked using the **while:** message. The receiving task will continue to run until the task after the colon will give a negative result. This is how it works:

```
>> x := 0.
{ x add: 1. } while: { <- x < 5. }.
Out write: x, stop.
```

```
5
```

In the above-illustrated code fragment **1** is added to **x** as long as **x** is less than **5**. When this is no longer the case, the second task will answer with **False**, consequently, the execution of the first task will end.

2.5 Templates

The *kJ/kcal* list from the previous chapter could be made more presentable. Preferably, the list would be as follows:

```
100 kJ → 23,88 kcal
200 kJ → 47,76 kcal
```

When you convert the preferred output into a template, its notation could be:

```
number1 kJ → number2 kcal
```

This means that **number1** stands for kJ-value and **number2** for kcal-value. This is how string interpolation operates in Citrine. There are no separate grammar rules needed for this in Citrine. Simply send the word that needs to be replaced to the text with the substitute text as argument:

```
>> text :=
    ['number1 kJ → number2 kcal']
    number1: 100,
    number2: 23.9.
```

The result:

```
100 kJ → 23,9 kcal
```

This substitution rule works for each undefined message that is received by a **Text** object. Each message that is not recognised by the **Text** object, will be interpreted as follows: *replace the message text with the text within the message argument*. You can adapt the program as follows:

```
{ :line
    >> kJ := line * 100.
    >> kcal := kJ * 0.239.
    Out write: (
        ['number1 kJ → number2 kcal']
        number1: kJ,
        number2: kcal
    ), stop.
} * 10.
```



```
100 kj → 23,88 kcal  
200 kj → 47,76 kcal  
...
```

To avoid confusion about which message can or cannot be used as a substitute, it is best to introduce a non-appointed character, e.g. a lozenge ◊ (U+2B27, ALT+Z), prior to text segments in the template that need to be replaced. Such a character is sometimes advisable in international context.

2.6 Answering

So far, you have sent messages to objects. However, you have not yet *answered* any messages yourself. To answer a message the return arrow (`<-`) is used. The following example illustrates how to create a task to compute a percentage:

```
>> percentage := {  
    :number :percent  
    <- number / 100 * percent.  
}.  
  
Out write:  
( percentage apply: 100 and: 7 ),  
stop.
```

7

After the task has been defined and assigned to variable **percentage**, the message **apply:and:** is sent with arguments **100** and **7**.

This will execute the task applied to **100** and **7**, i.e., 7% of 100. Using the return arrow, the answer is returned from the task back to the main program. Although this code is valid, it has a disadvantage because the sequence of the arguments has to be remembered. So, why not write it like this: *7 percent of: 100*? That would look a lot more *natural*. To make this possible, we have to adapt the parent object of **7**, which is the **Number** object, so that it will understand the message **percent of:**. This can be achieved by sending **on:do:** to the Number object, like so:

```
Number on: ['percent-of:'] do: {  
    :number  
    <- number / 100 * self.  
}.  
  
Out write:  
(7 percent-of: 100),  
stop.
```

As the percentage is, in this case, the number itself, we refer to self, the *self* keyword. In short, the symbol self signifies: *send this message to myself*. After having executed the code mentioned above, we can do:

Out write:

```
(7 percent-of: 100),  
stop.
```

Of course, the result will be:

7

2.7 Inheritance and Properties

Besides adjusting and expanding existing objects, you can also create new objects yourself simply by sending the message **new**. Suppose you would like to create an invoice object that enforces a numbering sequence, in that case, you will first need an invoice object.

Knowing that each object derives from another object, which, in turn, eventually originates from the root object of all objects called **Object**, a choice needs to be made regarding on which of those other objects your new object will be founded. Your new object inherits all properties of the object on which it is founded, so, to which the **new** message initially was sent.

In this example a rather neutral object is preferred, one without too many inherited properties. This offers an easy choice, because in that case the new object can be based on the root object itself which is **Object**. The notation of the intended invoice system will then be as follows:

```
>> invoice := Object new.  
  
invoice on: ['start'] do: {  
    own number := 0.  
}.  
  
invoice on: ['number'] do: {  
    own number add: 1.  
    <- own number copy plain.  
}.
```

Use this object as follows:

```
invoice start.  
Out write: invoice number, stop.  
Out write: invoice number, stop.
```

Output:

```
1  
2
```

The current invoice number is stored in the object, this is why there is a **own** keyword placed in front of it. This is called a *property*; more details will follow.

Some businesses prefer to have the year incorporated into the invoice numbers. In this case, you can create a new invoice object founded on a former invoice object, but which offers the possibility to its user to enter a specified year:

```
>> year-invoice := invoice new.  
  
year-invoice on: ['start:'] do: { :year  
    own number := year.  
}.  

```

This **year invoice** object can be used like:

```
year-invoice start: 202000.  
  
Out write: year-invoice number, stop.  
Out write: year-invoice number, stop.  

```

Output:

```
202001  
202002
```

There is no need to write the implementation for the message **number** again, as it is inherited from the previously written **invoice object**; consequently, old code can be reused. Citrine lacks concepts such as classes and other related concepts. This means that objects can solely inherit from other objects, also known as *prototypical inheritance*.

Now, to get back to properties. Object properties can only be approached from *within*, so these properties cannot be perceived by other objects. Contrary to other programming languages, all properties of objects are exclusively visible to the object that has created the property and to derived objects. The following is a brief demo regarding this principle. Let's say that you would like to know the number of visitors of a retail store at a fixed moment. A sensor is placed at the entrance and is operated by means of the supplier's program shown below:

```
>> sensor := Object new.  
  
sensor on: ['new-day'] do: {  
    own counter := 0.  
}.  
  
sensor on: ['door-opens'] do: {  
    own counter add: 1.  
}.  

```

Every new day, the counter will be reset to **0**, and each time the door opens **1** is added to the **counter**. This particular object could serve a computer program that receives signals from a physical sensor that is placed near the store door. However, the provided object **sensor** of the supplier does not take into account the people that leave the store, so it is better to extend the object. Quite creatively, the improved **sensor** version in this example is called **sensor2**:

```
>> sensor2 := sensor new.  
  
sensor2 on: ['exit-opens'] do: {  
    own counter subtract: 1.  
}.
```

Object **sensor2** is based on **sensor**, however we reduce the counter each time the exit door opens. The object can be used by means of a program that processes signals from a physical sensor as follows:

```
sensor2  
    new-day  
    door-opens  
    door-opens  
    door-opens  
    exit-opens.
```

After this message sequence the counter reads **2**. In this case, both sensors share the property **counter**. Because **sensor2** is derived from **sensor**, the value of the counter can be modified. Other objects are unable to modify the value, therefore, the programmer cannot change the value externally. This avoids surprises, knowing that the property **counter** can solely be modified from within the object.

This is why this next code fragment is pointless:

```
>> sensor3 := Object new.  
  
sensor3 on: ['extra'] do: {  
    own counter add: 2.  
}.
```

Since **sensor3** is not based on **sensor** and neither on **sensor2**, the counter of **sensor/sensor2** will not be altered. In any case, this code will result in error, because the object does not even know the property **counter**, because the counter has not been initialised. Also the following construction is invalid:

```
Out write: sensor own counter.
```

What is more, the **own** keyword is never used outside the brackets of a task object. The moment you spot such a code fragment, you know something is amiss. The property **counter** can only be obtained from a task that is already part of the object that has declared the property, or from a task that is part of a derived object. Still, by means of a message, the property can be made available to the external world:

```
sensor2 on: ['counter'] do: {  
    <- own counter.  
}.
```

```
Out write: sensor2 counter, stop.
```

2

In practice, however, this is not a smart move. A sloppy programmer will now be able to modify the counter externally and, by doing so, will *breach the encapsulation of the object*. As a reader of the programming code you no longer know whether the data of the counter is reliable or not, as the counter can be modified throughout the programming code. If you wish to share the details of the counted number of store visitors, but do not want a programmer to make modifications elsewhere in the program, it is advisable to return a copy:

```
sensor2 on: ['counter'] do: {  
    <- own counter copy.  
}.
```

In that case, the programmer is able to read the counter, for example to show on screen, but it cannot be modified. This avoids needless program errors. Note that this exact same technique was applied in the previous example regarding the invoice numbers when sending the invoice number back. In that example, a copy was also used instead of the original. More details on the message **copy** will be illustrated in chapter 4. It is important to keep in mind that Citrine always uses *references*. This means that when you declare an object with `:=` or return it with `<-`, a copy is not made automatically. You need to explicitly state this. The next example illustrates the effect of using the copy message:

```
sensor2  
    new-day  
    door-opens  
    door-opens  
    door-opens  
    exit-opens.
```

```
sensor2 counter add: 100.  
Out write: sensor2 counter, stop.
```

Result:

```
2
```

This still shows **2**, however without adding the message **copy** the result would show **102**.

A commonly used technique is to use an object as a kind of *blueprint*, which is, for instance, the case with the invoices discussed earlier. Another good example is the **Point**. Let's say that you create a computer program that makes calculations by using points on a map. You can develop the object **Point** containing the properties of an x-coordinate and a y-coordinate, which will be made available to the external world through messages:

```
>> Point := Object new.  
Point on: ['x-coordinate:'] do: { :x own x := x. }.  
Point on: ['y-coordinate:'] do: { :y own y := y. }.  
Point on: ['x-coordinate'] do: { <- own x copy. }.  
Point on: ['y-coordinate'] do: { <- own y copy. }.
```

Now, we can add a message to **Point** to compare two instances, i.e. points:

```
Point on: ['='] do: { :other  
    <- (  
        own x = other x-coordinate  
        and: own y = other y-coordinate  
    ).  
}.
```

This message compares the two points using the message **and:**. The first comparison results in a **True** or a **False**. These are boolean objects. When you send the message **and:** to a **True** object, and the argument object after de colon is equally **True**, then you will receive as answer again **True**. When either one is **False**, you will receive the answer **False**. More on the boolean object will follow

in the next chapter. Note that one point cannot read the coordinates of the other one. It is necessary to send a message to the other point to ask for this data: **x-coordinate** and **y-coordinate**. After all, as has been explained already, properties are solely retrievable from within the object family itself. The following example illustrates how to apply the Point object:

```
>> pier := Point new x-coordinate: 5, y-coordinate: 6.  
>> town-hall := Point new x-coordinate: 7, y-coordinate: 1.  
>> restaurant := Point new x-coordinate: 5, y-coordinate: 6.
```

To pinpoint whether it is the restaurant or the town hall that is situated on the pier:

```
Out write: pier = town-hall, stop.  
Out write: pier = restaurant, stop.
```

Output

```
False  
True
```

Note that the message = is used here to execute the point comparison. Equally, a different message could have been used, such as: **is:** or **equal:**, however reusing the = character seemed appropriate in this case.

2.8 Exercises

1. Which of the next variable names is invalid?

P, q, -x, \$, @p

2. What value represents **x** in the following code fragment?

```
>> x := 5 * 2 + 1 * 3 / 2.
```

3. Find the error.

The value of **x** in the next program should be **42**, but shows **2**.

Improve the program to make the value **x** to equal **42**.

What is the error?

```
>> x := 2.  
{ x add: x power: 2. } * 2.
```

4. Place a comma in the following code fragment, so **x** equals **10**.

```
>> x := 2 to the power: 3 + 2.
```

5. At the end of the next program **x** equals **-19**. What message should be written on the dotted line?

```
>> x := 11.  
{ :counter  
  counter ... true: {  
    x subtract: counter.  
  }.  
} * 10.  
  
Out write: x, stop.
```

6. At the end of this next program, **x** is equal to **104**. What message should be written on the dotted line?

```
>> x := 13.  
{ x add: x. } ....: { <- x < 100. }.
```

7. What should be written on the dotted line, in order to see the name of a popular drink being printed on the screen? You can repeatedly choose between **true:** or **false:**.

```
>> x := 7.  
  
(x > 7) ....: { Out write: ['p']. }.  
(x < 7) ....: { Out write: ['o']. }.  
(x ≥ 7) ....: { Out write: ['t']. }.  
(x ≤ 7) ....: { Out write: ['e']. }.  
(x ≠ 7) ....: { Out write: ['n']. }.  
(x = 7) ....: { Out write: ['t']. }.  
(7 > x) ....: { Out write: ['i']. }.  
(7 < x) ....: { Out write: ['a']. }.  
(7 ≥ x) ....: { Out write: ['l']. }.  
(7 ≤ x) ....: { Out write: ['l']. }.  
(7 ≠ x) ....: { Out write: ['y']. }.
```

8. What are the answers to the following problems?

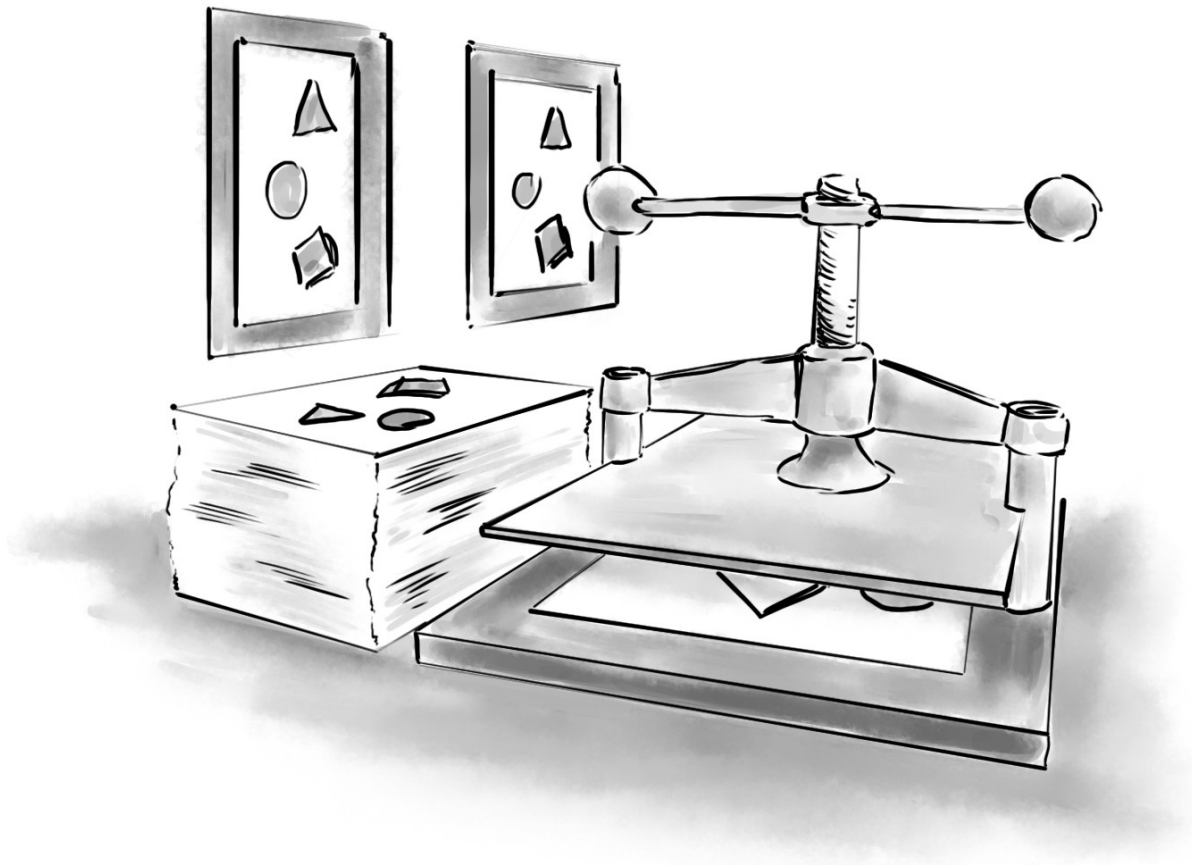
```
>> a := 5 + 3.  
>> b := 1 + 2.  
>> c := 3 + 1.  
>> d := 4 * 5 + 1 / 2.  
>> e := 100 / (2 - 2 + 4).
```

9.

This next program results in a **False** while a **True** is expected. This is due to a code error. Improve the program, so the result will indeed provide the right answer at 6 sunny days and 3 tropical days.

```
>> Heatwave? := {  
    :sunny-days :tropical-days  
    (sunny-days ≥ 5) true: {  
        (tropical-days ≥ 3) true: {  
            <- True.  
        }.  
    }.  
    <- False.  
}.
```

Out write: (Heatwave? apply: 6 and: 3), stop.



3. Objects

3.1 Simple Objects

The world of Citrine is filled with a diversity of objects, which are available to your program right from the start. In this chapter, all of these system objects will be reviewed. Citrine recognises 12 base objects in total; namely, the **None** object, the root object called **Object**, the **Boolean** object (and its two instances, **True** and **False**), **Number**, **Text**, **Task**, **Sequence**, **List**, **File**, **Moment**, **Program**, and **Out** (the output object). To know these 12 objects, is to know Citrine. Furthermore, there are two implicit objects: **Path** and **Command**. These two will be discussed together with the **File** object and the **Program** object. Their role is very limited and depends on the earlier mentioned objects. Some of the objects from the above mentioned list are not very complex. The most straightforward one is the output object that takes the form of a pencil. This object manages the output of the program to other systems, which in most cases is the screen, but it could also be a server, another program or a printer. The output object understands only two messages, which have been viewed in the past chapters: **write**: (to write something down) and **stop** to begin a new line.

```
Out write: ['Brevity is the soul of wit!'] , stop.
```

```
Brevity is the soul of wit!
```

The message **write**: should always be followed by a **Text** object. Should you send something different, the output object will try to convert the message into text, i.e., implicit conversion. The exact conversion depends on the object itself. In general, with numbers, their textual representation is used, which also applies to boolean objects. Other objects can decide themselves how to be presented as text. More about conversion in chapters 3.5 and 4.6.

Another very basic object is the **None** object. This object represents emptiness, or better still, the absence of information. On occasion, you will receive this object as an answer to a message, in case of the result being **nothing**. The most essential question you could ask the **None** object is: **None?**. The answer will always read **True**.

```
Out write: None None?.
```

```
True
```

Any object other than **None** will reply with **False**. The above code fragment may seem a bit philosophical, yet the **None** object certainly has extremely practical applications. For example, you

will receive the `None` object as an answer in case you ask for a sequence element that does not exist. You could also use the `None` object if you like to declare a variable, but do not want to specify a value yet. Instead, you just assign the special value **`None`**. In Citrine it is not allowed to declare a variable without value, such as:

```
>> value.
```

Instead, you have to write the following:

```
>> value := None.
```

3.2 Boolean Object

In the previous chapters the **True** or **False** objects have been discussed at length. We have used these objects, for example, to execute tasks depending on certain conditions. **True** and **False** are based on *Boolean Algebra*.

Contrary to most popular programming languages at the moment of writing, Citrine provides for only one single True object and one single False object. To clarify, each time you write, **True** it does not imply that a new object has been created. Instead, you always use a reference. This means that when you write the following:

```
>> x := True.
```

The **x** refers to the **True** object. Conditional code and loops also verify this reference. In Citrine, the meaning of **True** and **False** is not fixed. In fact, a Citrine program gets pretty shaken up over a statement like this:

```
True := False.
```

The result of such actions is undefined, however it remains a valid action and therefore formally allowed. Furthermore, there is a Boolean object, which is the root object of both **True** and **False**, as both are derivatives of the root object. The Boolean object itself, however, does not provide any practical application.

By means of True and False objects, you are able to run parts of your program under specified conditions:

```
>> price := Number between: 0 and: 20.  
>> budget := 10.  
  
price > budget true: {  
    Out write: ['too expensive'].  
}, false: {  
    Out write: ['sold!'].  
}.
```

For readability purposes, instead of **false:** you can also use the message **else:**. This is basically the same, because the message **else:** is a synonym of **false:**. So, the next fragment shows exactly the same procedure:

```
>> price := Number between: 0 and: 20.
>> budget := 10.

price > budget true: {
    Out write: ['too expensive'].
}, else: {
    Out write: ['sold!'].
}.
```

It is entirely up to you which of the two variations you prefer to use for readability.

Boolean objects can also be used to combine different conditions. For instance, a task can be executed when two things both produce a **True** object. Or, if one out of two or even more conditions are met as illustrated in the following example:

```
>> sugar := True.
>> milk := False.
>> x := sugar and: milk.
```

In the above fragment two new objects are created, which are **sugar** and **milk**. The object **sugar** is a **True** object while **milk** is a **False** object. Now, when you like to find out whether someone prefers both milk and sugar in their coffee or tea, you send the message **and:** to the one object and you send the other object along with it as argument. The response to **and:** will be **False**, because **and:** only answers with **True** when the object itself is affirmative as well as the argument. In every other case, it will return **False**. So, in this case **x** equals **False**. In addition to the message **and:**, you can also use **or:**, like so:

```
>> sugar := True.
>> milk := False.
>> x := sugar or: milk.
```

In this case, **sugar**, coincidentally the first stated object, will answer **True**. The **True** object to which the variable **sugar** refers to, actually always answers to the question **or:** with **True** if indeed the object itself is an affirmative or indeed the object in the argument is an affirmative, or if both are **True**. In case neither object is **True**, then the answer will be **False**.

By using the message **nor:** you can expect **True** if it is surrounded by two **False** objects.

```
>> sugar := False.
>> milk := False.
>> x := sugar nor: milk.
```

With the message **not** you reverse a boolean object, i.e., a **True** becomes a **False**.

```
>> sugar := False not.  
>> milk := False not.  
>> x := sugar and: milk.
```

So, in this case **x** is **True**, because both **sugar** and **milk** are **False not**, i.e. not no, consequently **True**.

With **either:or:** you can select, by means of a **True** or **False** object, a different object from an object pair. Like so:

```
>> sugar := True.  
  
>> x :=  
    sugar  
    either: 1 spoon or: None.
```

In the above fragment **x** will equal **1 spoon**. When the receiving message is **True**, the first object will be selected, if not, then the second one.

You can also convert a **True** or a **False** into a number by sending the message **number**. The **True** object will answer with **1** and **False** will answer with **0**. Obviously, you can compare **True** and **False** objects also to each other:

```
>> x := (True ≠ False).  
>> y := (True = False).
```

In this case **x** equals **True** and **y** equals **False**.

The messages **continue** and **break** can be used within a loop. So, when you send **break** within a loop to a **True** object, the loop will be terminated from that point. The program will continue at the first statement after the loop.

The message **continue** is related to **break**, however, instead of the entire loop being terminated, only the current iteration will be ended. The remaining part of the current round is skipped and the next round of the loop is initiated.

Here are some examples:

```
{ :i
Out write: i, stop.
(i > 9) break.
} * 20.
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

This particular code fragment illustrates the effect of the message **break**. It shows a small list of the numbers 1 to 10, however, as soon as the number in **i** exceeds ten, the loop is aborted at that point.

```
{ :i
(i > 10 and: i < 15) continue.
Out write: i, stop.
} * 20.
```

Output:

```
1
2
3
4
5
6
7
8
9
10
15
16
17
18
19
20
```

This code fragment produces a small list of 1 to 10 followed by 15 to 20. The numbers between 10 and 15 are skipped. This is because for this interval the condition (**i > 10 and: i < 15**) results in **True**, consequently the message **continue** is sent to this True object. This means that the remainder part of the loop is skipped and the next iteration begins. The messages **continue** and **break** are ignored by the **False** object.

Finally, the message **text** enables you to create a textual representation of a True or False object. The results are, of course, pretty straightforward:

```
Out write: True text, stop.  
Out write: False text, stop.
```

The result:

```
True  
False
```

3.3 The Number Object

Each time you write a number, for example 9, -10, or 3,12, behind the screen, Citrine will convert these numbers into a **Number** object. You can send messages to this Number object, or you could assign the number to a variable and send messages afterwards:

```
10 even?
```

```
>> tenner := 10.  
tenner even?
```

Both notations are valid. The Number object responds to the following messages like:

>, ≥, <, ≤, =, ≠, +, −, *, /, between:and:, odd?, even?, add:, subtract:, multiply by:, divide by:, modulus:, to the power:, positive?, negative?, rounded, rounded down, rounded up, square root, absolute, text, bool, plain, qualifier, and qualifier:.

Most of these messages are self-explanatory and allow you to execute mathematical operations (+) or comparisons (>). The difference between binary mathematical messages (+) and their keyword variations (**add:**) is that the former will return a new number, which is the result of the operation, whereas with the latter the object itself will be modified. This is illustrated in the following example:

```
>> a := 1.  
>> b := a + 3.  
  
a add: 2.  
Out write: a, stop.  
Out write: b, stop.
```

```
3  
4
```

In the above example, $b = 4$ and $a = 3$. With **add: 2** the value of a is raised by 2, while **+ 3** creates a new number that is equal to $a + 3$. The same applies to other mathematical processes, e.g., multiplications. By using the multiplication symbol, you will receive a new object as answer. In case you use the message **multiply by:**, you will multiply the number itself.

With the message **between:and:**, for example in: **Number between: X and: Y**, you will get a number between X and Y . In this way, any random number can be generated:

```
>> a := Number between: 1 and: 10.
```

```
Out write: a, stop.
```

```
5
```

Note, that the build-in generator of random numbers in Citrine is *not* suitable for cryptographic applications.

By default numbers are formatted according to the local rules:

```
Out write: 5000, stop.  
Out write: 5,000, stop.
```

Results in:

```
5,000  
5,000
```

To display a number without separators:

```
Out write: 5000 plain, stop.
```

This will translate as:

```
5000
```

The message **plain** returns a Text object supporting the number representation without separator marks. The following illustrates some examples of other messages that you can send to a number:

```
Out write: 5.25 round, stop.  
Out write: 5.25 ceil, stop.  
Out write: 5.25 floor, stop.  
Out write: 25 square-root, stop.  
Out write: (2 power: 8), stop.  
Out write: (5 modulo: 2), stop.
```



```
Out write: 3 even?, stop.
Out write: 3 odd?, stop.
Out write: 3 positive?, stop.
Out write: 3 negative?, stop.
```

```
5
6
5
5
256
1
False
True
True
False
```

You can attach a qualifier to a number, for instance, **6 apples**. Each message that does not get recognised by a number will be considered a qualifier. You can retrieve the qualifier of a number by means of the message **qualifier**:

```
>> amount := 6 coins.

Out write: amount.
Out write: amount qualifier.
```

```
6 coins
coins
```

A qualifier is basically a Text object that is stored with the Number object. The qualifier is also printed after the number on a **write:** assignment. Qualifiers could be used to add amounts in mixed currencies for example. On adding the amounts, you can ask for the qualifiers. The following program example illustrates this principle by using a historical currency calculator (as the exchange rate remains reasonably stable!).

```
Number on: ['+'] do: { :amount
    (amount qualifier = ['ducats']) true: {
        amount multiply-by: 5.
    }.
    self add: amount.
}.
```

```
Out write: (7 florins + 3 ducats), stop.
```

```
22 florins
```

In the above code fragment, the message `+` is adapted in such a way that the currency is taken into account. In this example 1 ducat equals 5 florins. The **self keyword** indicates the object itself: `self` is used for sending a message to the object itself. In addition, a qualifier can also be set explicitly by using the message **qualifier::**:

```
>> x := 7 qualifier: ['ducats'].
```

3.4 The Text Object

Each time a text is placed between quotation marks ['...'], Citrine will create a new **Text** object for you. However, make sure to use the correct quotation marks. The quotation mark at the beginning of the text differs from the one at the end.

You can send messages to this Text object, the same as you can send messages to other objects:

```
['Hello'] length.  
  
>> salute := ['Hello'].  
  
salute length.
```

Both variations are valid.

It is also allowed to use quotation marks within the text itself:

```
[' ['I like my new computer,'] said the robot. ']
```

You may use quotation marks in a text, as long as you also close them. So, these next texts are invalid:

```
[' A [' B ']  
[' A [' B ']  
[' A [' B [' C ']
```

However, would you like to use quotation marks in this way, it will be necessary to proceed the text with the exception character \:

```
[' A \[' B ']
```

With a `\n` within a text you can start a new line. With a `\t` you can add a TAB character.

The Text object responds to messages like: `=`, `≠`, `+`, `-`, `>`, `<`, `≤`, `≥`, **add:**, **length**, **from:length:**, **character:**, **offset:**, **find:**, **lowercase**, **capitals**, **offset:**, **replace:with:**, **contains:**, **trim**, **number**, **bool**, **split:**, **characters**, **code**, **object**, and **compare:**.

The messages `=`, `≠`, `>`, `<`, `≤`, `≥` enable you to compare two texts: e.g. `>` returns **True** if the receiving Text object comes after the included object, according to the dictionary:

```
['Zouch'] > ['Arundel']
```

Likewise, you can use the message **compare:** to express the difference in sequence by means of a number. The following programming code, for example:

```
['c'].compare: ['a']
```

returns 2, because the letter **c** is two places after the letter **a**. Note that at the time of writing, equations are implementation bound, and follow the UTF-8 coding.

By using the plus sign (+) you can concatenate multiple texts:

```
>> name := ['John'].
['Hello '] + name
```

Using the message **add:** will get the same result:

```
['Hello '] add: name.
```

However, a more elegant way to do this is as follows:

```
>> name := ['John'].
['Hello person'] person: name.
```

In this case, a template text is used and a singular word of that text is replaced. This usually benefits readability. In fact, it is the same as:

```
>> name := ['John'].
['Hello person'] replace: ['person'] with: name.
```

However, this detailed variation which uses the message **replace:with:**, has a negative impact on readability.

The minus symbol does the exact opposite of the plus symbol, because it gives a new Text object of which the last part is omitted:

```
>> message := ['No exclamation marks!!!'].  
message := message - ['!!!'].
```

Here, the exclamation marks at the end of the text are omitted. The minus symbol omits the corresponding text at the end of the Text object, and therefore does the exact opposite of the plus symbol. So, when the message would include ['!!! No exclamation marks !!!'] only the last mentioned exclamation marks would be omitted.

By sending the message **length** to a text, you will get the number of characters that particular text contains:

```
Out write: (['歡迎'] length), stop.
```

The result of the above-mentioned example shows 2, because there are two Chinese characters (meaning **welcome**) between the quotation marks. Make sure you do not confuse this number of characters with the number of bytes. Most programming languages return the number of bytes (which would be more than two) depending on the character set. Citrine does not handle system-specific and low-level concepts; therefore, the number of bytes is irrelevant. Any required byte operation needs to be processed by means of a specialised program or a software library. Notably, this restriction also ensures that Citrine remains platform independent. Citrine will not be affected by changes should, for example, in the future, people will be using computers with a basic unit other than a **byte**.

The message **character:**, enables you to retrieve the character at a given position in the text, for example:

```
Out write: (['Hello'] character: 2), stop.
```

results in:



e

Note that Citrine is a 1-based language, this means that all numbering starts with 1. The second letter of **Hello** is therefore **e** and not **l**; which, will be the case for the majority of the popular programming languages at the moment of writing. Use **contains:**, when you like to know if a text includes a certain fragment:

```
Out write: (['Hello World'] contains: ['He']), stop.
```

The result will show:

```
True
```

By using **find:** you can retrieve the starting position of the text fragment:

```
Out write: (['Hello World'] find: ['He']), stop.
```

so, the result will be:

```
1
```

This is because the text **He** starts at the first character of the sentence **Hello World**. Again, numbering starts at 1. Would you like to retrieve the rightmost position of the fragment, use the message **last:**.

```
Out write: (['Hello World'] last: ['Wo']), stop.
```

```
11
```

With **from:length:** you can copy a sentence fragment:

```
Out write: (  
  ['Hello World'] from: 7 length: 5  
) , stop.
```

results in::

```
World
```

Another means to copy a text fragment, is sending the message **skip:**. In this way, you can copy all text from a given position. The same result as the formermentioned example can be achieved with:

```
Out write: (['Hello World'] offset: 6), stop.
```

With the message **trim** you can omit white spaces on the left and right side of a text:

```
[' Hello '] trim
```

will result in:

```
['Hello']
```

Also newlines will be stripped.

In addition, the Text object responds to the messages **capital**s and **lowercase** by returning a copy of the text in respectively capital letters and small letters.

```
Out write: ['Hello'] capitals, stop.  
Out write: ['Hello'] lowercase, stop.
```

```
HELLO  
hello
```

In chapter 3.8 the Sequence object will be discussed. By using the message **split:** you can create a text sequence, however more on this subject in corresponding chapter. Now let's look at an example.

The next program helps you to create a password. First of all, a typecase needs to be created with the appropriate characters. Note that not all characters are equally suitable for creating a password. Characters that look too much alike can cause a bit of a confusion when read. This is why letters and numbers that look too much alike are omitted from the typecase, such as **l** and **1**. The example generator creates passwords counting 16 characters. This is achieved, when a task is drafted that chooses a character from the typecase, and this task is then repeated 16 times.

```
>> typecase := ['acdefghkmnpqrtx35789@#&'].
```

```
>> password := [].  
  
{  
  
>> character :=  
    typecase  
    character: (  
        Number between: 1 and: typecase length  
    ).  
    password add: character.  
  
} * 16.  
  
Out write: password, stop.
```

```
Gen8xkhdhc99977c
```


3.5 Conversions

In some cases a number is returned in the shape of a text. For example as a user's input, like so:

```
>> x := Program ask.
```

The object **Program** will be discussed in detail later on, however, for now it will suffice to know that the message **ask** awaits user's input until the ENTER key is pressed. The answer is returned as text in the variable **x**. To convert this text into a number, the message **number** is sent and from then on, number messages can be send. In general, you can create a copy of each object with a different type by using the following messages:

Message	Effect
number	Converts an object into a number
text	Converts an object into text
bool	Converts an object into a boolean object (True or False)

The following rules apply:

Message	Recipient				
	<i>None</i>	<i>True/False</i>	<i>Number</i>	<i>Text</i>	<i>Other</i>
bool	False	-	0 → False else: True	True	Mostly True
number	0	True → 1 False → 0	-	['1'] → 1 ['2'] → 2 etc.. ['?'] → 0	Mostly 1
text	['None']	True → ['True'] False → ['False']	1 → ['1'] 2 → ['2'] etc..	-	Depends...

From this chart it can be concluded, for example, that in case the message **bool** is sent to a **None** object, the answer **False** can be expected (top-left on chart), and by sending **number**, the result **0** can be expected. When the message is **text**, then the answer will be the text **['None']** (lower-left on chart).

3.6 Task Objects

The **Task** object responds to messages like: **start**, *****, **while:**, **error:**, **except:** and **procedure:**. Chapter 2 illustrated how a given fragment of code can be executed multiple times; namely, by sending the message ***** to a task, followed by the required number of iterations. For a quick reminder, here are two new examples of those kinds of messages. First, the simple loop:

```
{ :degrees
  Out write:
  degrees * 1.8 + 32, stop.
} * 30.
```

This example returns a small list of temperatures ranging from 1 to 30 degrees *Fahrenheit*:

```
33.8
35.6
...
84.2
86
```

This sequence is created when the task between the brackets **{}** is multiplied by the number **30**.

```
>> degrees := 0.
{
  Out write:
  degrees * 1.8 + 32, stop.
  degrees add: 1.
} while: { <- degrees ≤ 30. }.
```

The above illustrated fragment returns the same result, however it uses a while loop. In this case, the message **while:** is sent to the task with argument a second task. The first task will be executed repeatedly, until the second task returns a **False**. To clarify, as long as the answer to the first task remains **True**, the first task continues to be executed. Because each time **1** is added to the number of degrees during the run of the first task, the second task will return **False** as soon as it counts 31 degrees Fahrenheit.

A different way to manipulate the flow of the computer program, is to define a code fragment with handlers in case of an error. For instance, say that a budget application has to determine the monthly budget based on a user's input on income and the number of months.

```
>> budget := income / months.
```

Can you guess what happens when the number of months turns out to be 0? What happens is an error message:

```
Uncatched error has occurred.  
Division by zero.
```

The first line of the message shows, that an error has occurred which had not been handled by the program. The second line in the output tells the factual problem; which is, in this case, that division by 0 is not allowed. The first line also reveals that an error can be *handled*. This is also a case of conditional code execution; however, in this situation a code fragment will be written and executed in the event of an error caused by a different code fragment. For example, see how to modify this error message:

```
{  
    >> budget := income / months.  
} except: { :error  
    Out write: ['Not allowed!'], stop.  
}, start.
```

In this case the output will be (on 0 months):

```
Not allowed!
```

In this case, you have managed your own error handling. Note, that the statement **unhandled error** is missing; after all, the error is nicely handled by you and, according to Citrine, that is the end of it. However, how does the error handling in the above example exactly works? First of all, there are two tasks: the original task and the handling task. These tasks are linked by the message **except:**. On receiving the message **except:** as well as a handling task, the receiving Task object will, in case of an error occurrence, switch to the code within the Task object that has been assigned to do the error handler. After the two blocks are linked, it is obvious what to do in case of an error occurrence. Do not forget to start the first Task object, otherwise the two tasks are indeed linked, but

nothing is actually being started. This introduces the final message in the code fragment: **start**. This starts the execution of the first Task object, because the message **except:** produces the Task object itself as answer. In addition, given that it concerns a keyword message, a comma will be needed to chain the new unary message. Formally, the structure can be noted as follows:

```
{ Task 1 } except: { Task 2 }, start.
```

You can also cause an error to occur in your program intentionally; and by doing so, activate the handler block. This is done by sending the message **error:** to the current task as shown in this next illustration:

```
{
    this-task error: ['Whoops!'].
} except: { :mistake
    Out write: mistake.
}, start.
```

Result



Whoops!

Notice that the error object included in the message **error:**, is returned in the handler routine. In this way, a variety of self-induced error objects can be passed on to your handler task. In addition, it shows how you can communicate with the current task by sending a message to **this-task**.

Instead of sending the message **start**, the message *** 1** can be used to execute the task one time. Both messages end up having the same result. A message which is very similar to **start** (or *** 1**, depending how you look at it) is the message **procedure**. This message is often used to improve readability. In the following example the various aspects of how the Task object is used in practice will be discussed. This next demo program aims to convert a number into Roman numerals. This particular example is confined to numbers below 40.

```
>> number := 17.
{
    {
        number ≥ 10 true: {
```

```

        Out write: ['X'].
        number subtract: 10.

    }, break.

number ≥ 9 true: {

    Out write: ['X'].
    number subtract: 9.

}, break.

number ≥ 5 true: {

    Out write: ['V'].
    number subtract: 5.

}, break.

number ≥ 4 true: {

    Out write: ['M'].
    number subtract: 4.

}, break.

number ≥ 1 true: {

    Out write: ['I'].
    number subtract: 1.

}, break.

} procedure.

} while: { <- number > 0. }.

```

XVII

The program in its current form returns the Roman numeral for the number **17**. By modifying the value of the number **17** at the top of the program into something different, for example **20**, the result will equally change (**XX**). Now, take a closer look at the program. The most outer brackets belong to the message **while:**. At this point you link two tasks. The first task continues to be executed as long as the second task keeps returning **True** as answer. That second task is simple enough:

```
{ <- number > 0. }
```

This means that, as long as **number** is more than **0** the first task will be executed. The first task is a bit longer, however it is basically a set of conditional tasks. Have a look at the first fragment:

```

number ≥ 10 true: {
    Out write: ['X'].
    number subtract: 10.
}, break.

```

Asked here, is whether the number is larger or equal to **10**. If the answer to this question is **True**, the Roman symbol for the number **10** is written, which is **X**. To continue, **10** is deducted from **number** leaving this round finished. Consequently, the message **break** jumps out of the loop and checks if **number** still remains larger than **0**. If yes, then another round is initiated. In the case of the number being **20**, you arrive at this first part again. In the case of **17**, the answer to the question if **number ≥ 10** should clearly read **False**, as $17 - 10 = 7$, and 7 is less than 10. In this case you descend to the next block (**number ≥ 9**). Again the answer will read **False**, as 7 is smaller than 9. You then arrive at (**number ≥ 5**), which results in **V**. In this way, you sort of nibble away the largest possible Roman numeral from your number until nothing is left.

However, there is a small technicality hidden in this program. Note that also the message **break** is sent, in case there is the possibility to display a Roman numeral on the screen. This is to prevent the display of a smaller Roman numeral too soon. In case of 20 (**XX**), it is not desirable to display a **IX** after the **X**; basically, the aim is to restart the process. In fact, this is why the number comparators are within a task and the message **procedure** is sent to the outer task. This procedure ensures the possibility to abort the task after a successful comparison has been made.

Citrine provides for tasks the option to inject values. Now, have a look at the task below:

```

>> sending := {
    Newsletter to: own recipient.
}.

sending set: ['recipient']
value: ['info@citrine-lang.org'].

sending start.

```

This illustration presents an imaginary task, which sends a newsletter to an email address or **recipient**. This recipient can be injected into the task, externally and before the task is started, by sending the message **set:value:** to the task. By doing so, the value of **recipient** is preset into the task. This value can also be modified and the task can then be run again. This is a useful method when using Task objects.

Empty tasks are not allowed. In theory an empty task would look like this: {}, however Citrine perceives this as a language error. If you like to declare an empty task, you can use the None object:

```
>> task := None.
```

Although, this is not a real task, but a None object, a start message can still be sent:

```
>> answer := task start.
```

In fact this is the same as:

```
>> answer := None start.
```

Because the object **None** does not recognise the message **start**, it will return itself as answer, leaving the answer once again **None**. So, there is no necessity to have an empty task. Due to the elegant design of the programming language Citrine, you can simply use the None object for this.

3.7 The Root Object

The object named **Object** is the root object of all objects in Citrine. This root object responds to messages like: **do**, **done**, **case:do:**, **message:arguments:**, **on:do:**, **respond:and ...**, **learn:means:**, and **recursive**. Every other object derives from this object, including the **None** object. The most frequently sent message is **on:do:**, which expands the functionalities of an object. This message is received by the root object, which consequently links the specified task to the message, and by doing so expands the derived object. This method has already been mentioned in chapter 2, however, for completeness take a look at another example.

In this next illustration, a circle object is defined with the intention of computing its area based on its radius:

```
>> circle := Object new.

circle on: ['radius:'] do: { :r
    own radius := r.
}.

circle on: ['area'] do: {
    <- own radius * 2 * 3.14.
}.
```

Two messages are added: **radius:**, which is used to set the radius of the circle, and **perimeter**, which is used to calculate its perimeter. This object can be used as follows:

```
circle radius: 4, area.
```

The result will be **25.12** (note that a very rough approximation of Pi is used, namely 3.14, therefore the result is not very accurate). This object can also be based on another object, which means that a previously defined object is expanded by sending **on:do:** messages. In this way, for instance, a **User** object can be based on a **Person** object to which a login name can be added:

```
>> User := Person new.

User on: ['login:'] do: { :name
    own login := name.
```



```
}.
```

In addition, system objects can be expanded, such as the **Text** object with a *ROT-13* encryption. This is an easy way for encrypting texts, based on the *Caesar Cipher* encryption technique (which is very weak for more serious purposes). The principle behind ROT-13 is quite simple. Each character is shifted 13 positions in the alphabet. In case a character passes the character **z**, counting continues from **a** onwards. So, **a** becomes **n**, **b** becomes **o** and **x** becomes **k**. For example, when the text **London** is encrypted with ROT-13 it will show **Lbaqba**.

If such a functionality needs to be added to each text, the easiest way to do so is to expand the Text object itself (as each text derives from this very object) using the message **on:do:**.

```
Text on: ['rot13'] do: {
    >> alphabet :=
    ['abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz'].
    >> code := ''.
    self characters each: { :i :character
        >> index := alphabet find: character.
        >> rot13 := index + 13.
        >> coded := alphabet character: rot13.
        code add: coded.
    }.
    <- code.
}.
```

From now on, every text can be encrypted as follows:

```
Out write: ['hello'] rot13, stop.
```

```
uryyb
```

An amusing detail of ROT-13 is that when the encryption is implemented for a second time, the original text reappears. In this way, the encrypted text becomes decrypted again:

```
Out write: ['hello'] rot13 rot13, stop.
```

```
hello
```

This shows that the message **on:do:** is one of the most fundamental principles of the Citrine programming language. Note that without this message, which is part of the root object Object, expanding the functionalities of objects would not be possible.

A second essential message is **equal:**, which returns **True** in case of two variables that are pointing towards the same object, such as:

```
>> x := 1.  
>> y := x.  
Out write: (x equal: y), stop.
```

```
True
```

Notice the subtle difference between `=` and **equal:**. The first mentioned compares the content value of two objects, whereas the second mentioned peruses the memory of the computer whether they are in fact one and the same object.

The message **case:do:** makes it possible to run a conditional task, just like the messages **true:** and **false:**. The difference is that you can attach a task per value. The following example, for instance, shows the value in the variable **x**. If the value of **x** is equal to **1**, the first code block is executed. If **x=2**, the second task will be executed, and so on. In this way, you select the correct code for the value **x**:

```
>> x := 2.  
  
x  
  
case: 1 do: {  
    Out write: ['Status: denied'].  
},  
case: 2 do: {  
    Out write: ['Status: pending'].  
},
```

```
case: 3 do: {  
    Out write: ['Status: approved'].  
}.  
  
Out stop.
```

```
Status: pending
```

3.8 Sequences

Citrine knows two types of collections: *sequences* and *lists*. **Sequences** are enumerations of objects in a *fixed order*. Lists have no order, instead, resemble a legenda involving a key (or term) and its corresponding value. Sequences are comparable to arrays (PHP, Java, C) and lists (Python). Lists are comparable to associative arrays (PHP) or dictionaries (Python) in other programming languages. In order to create a new Sequence write:

```
>> fibonacci := Sequence new.
```

This empty sequence can be filled using **append**:

```
fibonacci append: 0.  
fibonacci append: 1.  
fibonacci append: 1.  
fibonacci append: 2.  
fibonacci append: 3.  
fibonacci append: 5.  
fibonacci append: 8.  
fibonacci append: 13.
```

If we write the sequence to screen:

```
Out write: fibonacci, stop.
```

We will see:

```
Sequence ← 0 ; 1 ; 1 ; 2 ; 3 ; 5 ; 8 ; 13
```

This output reveals that you can use this shorter notation to declare a sequence as well:

```
>> fibonacci := Sequence ← 0 ; 1 ; 1 ; 2 ; 3 ; 5 ; 8 ; 13.  
Out write: fibonacci, stop.
```

This program shows the same output. This notation is of course much shorter and therefore far more comfortable.

The arrow is a binary message, which creates a new sequence and at the same time sends the subsequent object to be embedded instantly into this new sequence. The semicolons (;) are all binary messages which put a value into the sequence.

To find out the number of objects of a sequence use **count**:

```
Out write: fibonacci count.
```

```
8
```

Instead of using the semicolon, a bullet symbol can be used to add something to the sequence. In this way, the list looks pretty similar to one in a regular text:

```
>> dishes := Sequence new
    ~ ['Chicory']
    ~ ['Brussels sprouts']
    ~ ['Sauerkraut'].
```

Sequences respond to the messages like: **append:**, **minimum**, **maximum**, **each:**, **prepend:**, **combine:**, **position:**, **first**, **last**, **penultimate**, **put:at:**, **shift**, **pop**, **amount**, **from:length:**, **replace:length:with:**, **by:**, **copy**, **sort:**, **fill:with:**, and **find:**.

Another way to create a sequence, is to transform a word into a sequence of characters (**a,b,c**), or to break down (i.e., **split**) a text into a sequence of words:

```
Out write: (['abc'] characters), stop.
```

```
Sequence ← ['a'] ; ['b'] ; ['c']
```

```
Out write: (
    ['Chicory,Brussels sprouts,Sauerkraut']
    split: [' ','']
), stop.
```

```
Sequence ← ['Chicory'] ; ['Brussels sprouts'] ; ['Sauerkraut']
```

For example, if the message **characters** is sent to a text, the answer will be a sequence containing all the separate characters out of which the text is made. When the message **split:** is sent to a text with another text as argument, the attached small text fragment is used to break down the receiving

text into separate text fragments. The answer will be a sequence of these Text objects. In turn, it is also possible to convert a sequence into a text by sending the message **combine:**, see the following example:

```
>> dishes := Sequence ←
```

```
    ['Chicory'] ;  
    ['Brussels sprouts'] ;  
    ['Sauerkraut'].
```

```
Out write: (dishes combine: [' ~or~ ']), stop.
```

```
Chicory ~or~ Brussels sprouts ~or~ Sauerkraut
```

In this case, the attached text fragment is used to string together the texts in the sequence.

Each object within a sequence has its own place, which is called the *position*. The first object gets position 1, the second object gets position 2, and so on. By using the message **position:** (or the shorter version **?**) it is possible to retrieve the element at the specified position:

```
Out write: (dishes position: 3), stop.  
Out write: dishes ? 1.  
Out write: dishes ? 9.
```

Results in :

```
Sauerkraut  
Chicory  
None
```

In case there is nothing on the requested position, the answer will be a **None** object. There are three special positions in a sequence, which are: the **first**, **last** and **penultimate** position.

Objects on those positions can be retrieved using the respective messages.

```
>> abc := ['ABC'] characters.
```

```
Out write: abc first, stop.  
Out write: abc last, stop.
```

```
Out write: abc penultimate, stop.
```

Results in:

```
A
C
B
```

To replace an object in a specific position within the sequence itself, write: **put:at:.**

```
abc put: ['2nd'] at: 2.
```

```
Sequence ← ['A'] ; ['2nd'] ; ['C']
```

In a similar way, an object can be eliminated from the sequence using the minus symbol:

```
>> x := Sequence ← 1 ; 2 ; 3.
```

```
Out write: x, stop.
```

```
Out write: x - 2.
```

```
Sequence ← 1 ; 2 ; 3
Sequence ← 1 ; 3
```

With the message **prepend:** it is possible to insert an object at the start of the sequence:

```
>> x := Sequence ← ['Nut'] ; ['Monkey'].
```

```
x prepend: ['Apple pie'].
```

```
Out write: x.
```

```
Sequence ← ['Apple pie'] ; ['Nut'] ; ['Monkey'].
```

It is equally possible to generate a sequence by filling an empty sequence with the message **fill:with:.** In case it is preferred to fill in the top scores of a computer game in advance with zeroes, write:

```
>> hiscores := Sequence new fill: 10 with: 0.  
Out write: hiscores, stop.
```

```
Sequence ← 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0
```

It's also possible to copy a sequence:

```
>>copy := hiscores copy.
```

It is possible to take away objects from the sequence by cutting into its start and end:

```
>> words := Sequence ← ['Apple pie'] ; ['Nut'] ; ['Monkey'].  
>> original := words copy.  
  
>> a := words shift.  
>> b := words pop.  
  
Out write: original, stop.  
  
Out write: a, stop.  
Out write: b, stop.  
  
Out write: words, stop.
```

```
Sequence ← ['Apple pie'] ; ['Nut'] ; ['Monkey']  
Apple pie  
Monkey  
Sequence ← ['Nut']
```

In the above example, by using the message **shift** the start object is cut away of the sequence, which is consequently put into **a**. Furthermore, the last object is cut away by using the message **pop**, which is then put into **b**. As a result, the middle Text object, **Nut**, remains. The original copy is also displayed, thus demonstrating the copy function.

The message **find**: makes it possible to find information within a sequence:

```
>> paintings := Sequence new  
    ~ ['Fighting Temeraire']
```



```
~ ['Lady of Shalott']  
~ ['Hay Wain'].
```

```
Out write: ( paintings find: ['Hay Wain']), stop.  
Out write: ( paintings find: ['Lady of Shalott']), stop.  
Out write: ( paintings find: ['Sunflowers']), stop.
```

```
3  
2  
None
```

By using the message **each:**, it is possible to apply a task to each element of a sequence:

```
>> meters := Sequence ← 1200 ; 4000 ; 6210.  
  
meters each: { :number :meters  
  
    Out write: meters / 0.3048, stop.  
  
}.
```

```
3,937.0078740157  
13,123.3595800525  
20,374.0157480315
```

This above-illustrated example shows a conversion of meters into foot. The conversion task is applied to each individual element of the sequence. In the first parameter (**:number**) the task receives the position of the element that is to be handled. In the second parameter (**:meters**) the task receives the object which is placed at the earlier-mentioned position. Feel free to name the parameters yourself, however, the sequence order is a fixed given. The first parameter is always the position number, and the second one is the object.

If the lowest number is required, the message **minimum** can be sent. Indeed, if the highest number within the sequence is required send the message **maximum**. The next illustration shows a sequence of scores from a game. A new sequence is created with two elements containing the extremes of the scores:

```
>> scores := Sequence ← 100 ; 50 ; 200 ; 350.  
  
>> extremes := Sequence ← scores minimum ; scores maximum.  
  
Out write: extremes, stop.
```

This results in:

```
Sequence ← 50 ; 350
```

In a sequence it is possible to execute a replacement by using the message **replace:length:with:**. So, it can be used to replace a sequence fragment by another sequence:

```
>> x := Sequence ← 1 ; 2 ; 3 ; 4 ; 5.  
>> z := Sequence ← 9.  
  
>> y := x replace: 2 length: 1 with: z.  
Out write: y, stop.
```

```
Sequence ← 1; 9; 3; 4; 5
```

Sorting is a useful action which is frequently applied to sequences. Given that only sequences are ordered, this action is unique for this kind of collection. A sequence can be sorted by sending the message **sort:** to a Sequence object, together with an argument containing a sort task that includes two parameters:

```
>> continents := Sequence new  
  ~ ['Oceania']  
  ~ ['America']  
  ~ ['Asia']  
  ~ ['Europe']  
  ~ ['Africa']  
  ~ ['Antartica']  
  ~ ['Australia'].  
  
continents sort: { :a :b <- a > b. }.  
  
Out write: continents, stop.
```

```
Sequence ← ['Africa'] ; ['America'] ; ['Antartica'] ; ['Asia'] ;  
['Australia'] ; ['Europe'] ; ['Oceania']
```

The message **sort:** will consistently fetch two objects from a sequence, and consequently requests the task to compare the two. The sort task should answer each round with **True** or **False**. In the

illustrated fragment the continents are sorted in alphabetical order by comparing each word pair by using the message `>`. To sort in the reverse order apply the message `<`.

3.9 Lists

Both Sequence and List are collections. However, contrary to a sequence, a list has no order. Another distinction between the two is that a list consists of object pairs. One object acts as entry (or keyword) in order to search for the other object. A good example of a list is a price list, such as:

```
>> menu := List new

    put: ['£5'] at: ['apple pie'],
    put: ['£6'] at: ['carrot cake'],
    put: ['£3'] at: ['fudge'].
```

Lists can receive the following messages: **type**, **put:at:**, **entries**, **values**, **at:**, **each:**, **contains:**.

Similar to a sequence, the message **put:at:** is used to add an object to a list. The difference is that, contrary to a sequence, two objects are being linked. The first object is, similar to a sequence, the object that you will store in a list. The second object is not its position within the collection, but the *entry* which enables the previous object to be retrieved at a later time. In short, a list works a bit like a dictionary; so, by using the keyword the meaning can be found. For instance, to retrieve from the above-mentioned list the price of a fudge:

```
>> price := menu at: ['fudge'].
```

In this case, you may also use the concise notation:

```
>> price := menu ? ['fudge'].
```

The answer will show **£3**.

Should the entries that you apply to store objects be without spaces, the next notation can be also be used:

```
>> menu := List new

    pie:    ['£5'],
    cake:   ['£6'],
    fudge:  ['£3'].
```

No doubt that the above notation reads more natural, because it resembles a list commonly used in a document. To request the price of a fudge, the following simplified notation can be used:

```
>> price := menu fudge.
```

Similar to a sequence, the list can be printed on screen:

```
Out write: menu, stop.
```

```
(List new) put:['£3'] at:['fudge'], put:['£6'] at:['cake'], put:
['£5'] at:['pie']
```

In case you intend to remove the pie from the menu, the binary message “-” is sent, followed by a Text object carrying the same name as the entry object of the pair that needs to be removed:

```
menu - ['pie'].
```

```
Out write: menu, stop.
```

```
(List new) put:['£3'] at:['fudge'], put:['£6'] at:['cake']
```

There is no possibility to search within a List object, since a list does not know any order. This means that the search function can never determine on which position the result is situated. However, it is possible to find out whether a particular object which holds the desired criteria is indeed included, by using the message **contains**:

```
>> receipt := List new
      coffee: ['cappuccino'],
      pie: ['apple'].
```

```
Out write: (receipt contains: ['cappuccino']), stop.
Out write: (receipt contains: ['beer']), stop.
```

Result:

```
True
False
```

The message **has:** can be used as an alternative to the message **contains:**, and, as it is an alias, its functionality is the same. So, feel free to use the most suitable word in context.

The message **each:** is equally understood by List objects. For instance, the following list illustrates the number of worked hours on a project:

```
>> labour := List new

    plastering: 2,
    painting: 4,
    wallpapering: 6.
```

In order to draw up the invoice, these hours have to be multiplied by the wages per hour (e.g. 50 p.h.) and then added up:

```
>> invoice := 0.

labour each: { :labour :hours

    invoice
    add: hours * 50.

}.
```

```
Out write: invoice, stop.
```

After the execution of this program the invoice will read **600**. It is also possible to generate a list from a couple of sequences. This can be done with the message **by:**, as in the following example:

```
>> place := Sequence new

    ~ ['London']
    ~ ['Belfast']
    ~ ['Edinburgh']
    ~ ['Cardiff'].
```

```
>> inhabitants := Sequence new
    ~ 8,799,800
    ~ 345,418
    ~ 553,569
    ~ 359,512.

>> population := inhabitants by: place.

Out write: population, stop.
```

Next, the population of Cardiff can be retrieved as follows:

```
Out write: population Cardiff, stop.
```

```
359,512
```

The above example shows two sequences, which, imaginable, could be obtained from a databank. Now, to create a list with populations whereby the number of inhabitants of each city can be retrieved, the two separate sequences must be compiled into a list. This can be done by using the message **by:**. When a sequence receives this message, with as argument another sequence, the result will be a list of which the keywords form the second list (place names) and the sequence to which the message was sent to will show the values.

In turn, it is possible to generate two sequences from a singular list. By using the message **entries**, a sequence of entries is received from the receiving list. The message **values** provides a sequence of values. This means that the list **population** can, in turn, be divided into two separate sequences in the following way:

```
>> names := population entries.

>> numbers := population values.
```

Lists and sequences can be combined; for example, putting a sequence into a sequence or a sequence into a list or vice versa. In fact, this is rather the rule than the exception. Take for instance, a list of addresses from a customer file. Theoretically, it could look like this:

```
>> addresses := Sequence new

    ~ ['1 Citrine Square GA2 3MO Citrineshire'],
    ~ ['4 Objects Road MA5 6VR Scopeford'].
```

These kinds of addresses are difficult to process. What to do in case it is necessary to search, based on place name or postal code? In that case it would be necessary to peruse the text to find out where the street name ends and the postcode begins. This is why structured addresses are preferred such as:

```
List new
  street: ['Citrine Square'],
  property: 1,
  zipcode: ['GA2 3MO'],
  place: ['Citrineshire'].
```

This works a lot easier. If you would like to know the place name of this address, the message **place** is sent to the list. The answer will be **Citrineshire**.

```
>> addresses := Sequence new
  ~ (
    List new
      street: ['Citrine Square'],
      property: 1,
      zipcode: ['GA2 3MO'],
      place: ['Citrineshire']
  )
  ~ (
    List new
      street: ['Objects Road'],
      property: 4,
      zipcode: ['MA5 6VR'],
      place: ['Scopeford']
  ).
```

Structured data like this is better suited for processing. In this way, it is easy to read all postal codes:

```
addresses each: { :index :address
  Out write: address zipcode, stop.
}.
```

```
GA2 3MO
MA5 6VR
```

So, without structured data, the previous would have been a lot more complicated. All combinations are possible. Lists can be put into lists, sequences into sequences, which, in turn, can be filled with new sequences. Feel free to be creative and colourful. Now, have a look at a practical application of the List object. List objects, for example, are great for creating one-to-one translations. The next program translates a sentence into *Morse code*. In this example, both a list and a sequence are used.

The program uses two collections. The first one consists of characters, which form the original sentence. The second collection is a list that works a bit similar to the *Rosetta stone*, and presents the correct Morse code for each individual character.

```
>> sentence := ['help'].
```

```
>> morse := List new
```

```
  a: ['.-'],
  b: ['-...'],
  c: ['-.-'],
  d: ['-..'],
  e: ['.'],
  f: ['..-'],
  g: ['--.'],
  h: ['....'],
  i: ['..'],
  j: ['.---'],
  k: ['-.-'],
  l: ['.-.'],
  m: ['--'],
  n: ['-.'],
  o: ['---'],
  p: ['.--'],
  q: ['--.-'],
  r: ['.-.'],
  s: ['...'],
  t: ['-'],
  u: ['..-'],
  v: ['...-'],
  w: ['.--'],
  x: ['-.-'],
  y: ['-.-'],
  z: ['-..'].
```

```
sentence characters each: { :i :character
  >> code := morse ? character.
  Out write: code, stop.
}.
```

After running the Morse code program, the following output is presented:

```
....
.
.-..
.--.
```

To translate a different word or sentence into Morse code, simply assign them to the variable **sentence**. In fact, the program is pretty straightforward. First, there is a **legenda**, a list, that saves the corresponding Morse code for each character. Second, the message **characters** is sent to **sentence**, which results in a sequence containing each individual character from the sentence. Third, iteration starts by sending the message **each:** to this sequence while attaching a task to find the required Morse code and to print the resulting character on screen.

3.10 Files

The File object responds to messages like: **path**, **read**, **write:**, **exists**, **delete**, **size**, and **sequence:**. Citrine allows for essential file editing. To do so, use the system object File.

In this example, a text file containing [*forgotten vegetables*] needs to be sorted alphabetically:

Content of the file *vegetables* in the folder *documents*:

Parsnip

Jerusalem artichoke

Winter radish

Turnip

To print the contents of this file on screen, write the following:

```
>> file := File new: Path documents vegetables.
```

```
Out write: file read, stop.
```

To sort the content and then save the sorted sequence write:

```
>> vegetables :=  
    File new:  
    Path documents vegetables,  
    read  
    split: ['↵'],  
    sort: { :a :b <- a > b. }.
```

```
File new:  
    Path documents sorted,  
    write: (vegetables combine: ['↵']).
```

So, also in this example, a file is read and split up into lines by means of the message **split:**, with in the argument the **new line** sign (↵). To continue, it is sorted according to the method mentioned in chapter 3. Eventually, it is saved to disk, after having combined the lines into one singular text.

Besides **read** and **write:**, other matters can be handled as well. By using the message **append:**, text can be added to a file, such as in the following example:

```
File new:  
  Path documents sorted,  
  append: ['Zucchini'].
```

The file location can be retrieved with the message **path**:

```
>> location := file path.
```

To find out whether a file exists or not, the message **exists** can be sent after which a **True** or **False** object is returned.

```
>> x := File new: Path unknown.  
Out write: x exists, stop.
```

```
No
```

Files can be deleted by sending the message **delete**:

```
file delete.
```

The **size** (in bytes) can also be retrieved:

```
file size.
```

To request the contents of a folder, the message **sequence** can be sent to the File object with as argument the folder location. For example, if the contents of the folder documents is requested do:

```
files := File sequence: Path documents.
```

As a result, a sequence is returned containing on each position a list with the entries **files** and **type** holding the file name and its type, in that order. The next program shows the contents of each user-entered file location:

```
>> files := File sequence: (Program argument: 3).

files each: { :number :details

    >> line := ['#number name (type).']
        number: number,
        name: details file,
        type: details ? ['type'].

    Out write: line, stop.
}.

```

When this program is saved as **list**, the program can be used as follows:

```
ctren list games

#1 . . (folder)
#2 . (folder)
#3 breakout (file)
#4 mahjong (file)
#5 pac-man (file)

```

File locations can be entered as text:

/tmp/myfolder

or as a Path object:

```
Path /tmp myfolder.
```

This last notation is preferred according to the principle of uniformity. File locations are eventually converted into Text objects, which indicate the file of the system path. File location objects benefit from system independence:

```
Path documents templates spreadsheet1.
```

Will show on systems with a path-separator sign / :

```
documents/templates/spreadsheet1
```

And on systems using a path-separator sign \ :

```
documents\templates\spreadsheet1
```

3.11 Moments

For a structured representation of time and date, Citrine uses the Moment object. In case a new Moment object is printed on screen, the current date and time are shown:

```
Out write: Moment new, stop.
```

```
06/11/2020 12:16:39
```

It is also easy to read out the individual time components. So, for instance, if it is only necessary to know the current year, the message **year** can be sent to the moment and, in turn, the reply will be a Number object containing solely the year:

```
>> m := Moment new.  
Out write: m year plain, stop.
```

```
2020
```

Please note that the message **plain** should be sent to the resulting Number object. This is to avoid the output of **2,020**, which is, of course, a correct notation for numbers, however, usually not used in the context of year notations. It is also possible to request matters that might be less obvious to determine at first, for instance the day of the week (using the message **weekday**):

```
>> m := Moment new.  
Out write: m weekday, stop.
```

```
3
```

Here, the number 3 represents Tuesday. A weekday always starts on 1 (Sunday), Monday stands for the second day, Tuesday for the third day, and so on. Saturday is always the last day of the week, according to Citrine.

In the same way, the number of the week can be retrieved (**week**), as well as, months, days, hours, minutes and seconds. The next example shows how a new Moment object is created and how you can retrieve all the individual time and date components: the **second**, the **minute**, the **hour**, the **day**, the **month**, the day of the year (**day-of-the-year**), the day of the week (**weekday**), and the time zone (**zone**).

```
>>now := Moment new.  
  
Out write: now, stop.  
Out write: now second, stop.  
Out write: now minute, stop.  
Out write: now hour, stop.  
Out write: now day, stop.  
Out write: now day-of-the-year, stop.  
Out write: now weekday, stop.  
Out write: now month, stop.  
Out write: now zone, stop.
```

The output:

```
06/11/2023 21:32:48  
48  
32  
21  
6  
309  
2  
11  
Europe/London
```

To change one of the time components, you send along a Number object. So, to modify the second hand to **13**, you do:

```
>> m := Moment new.  
  
m second: 13.  
  
Out write: m, stop.
```



```
31/12/2023 21:32:13
```

It is equally possible to set other time components by using **year:**, **month:**, **day:**, **hour:**, **minute:**, and **second:**:

```
>> then := Moment new
      year: 2020,
      month: 12,
      day: 31,
      hour: 23,
      minute: 59,
      second: 59.
```

Out write: then, stop.

```
31/12/2020 23:59:59
```

It is not possible to set or change **weekday** and **day of the year**.

Please note that the settings should be from large to small; in fact, setting a larger component erases the smaller components. The next example illustrates this principle:

```
>> then := Moment new
      year: 2020,
      month: 12,
      day: 31,
      hour: 23,
      minute: 59,
      second: 59,
      year: 2021.
```

Out write: then, stop.

This will show::

```
01/01/2021 00:00:00
```

In this case, the smaller time components are set back to their initial values, because at the very last moment the year is set.

Should you select a number that is too high for a time component, the clock will count on in the usual manner:

```
>> then := Moment new
      year: 2020,
      month: 2,
      day: 30.
```

```
Out write: then, stop.
```

The result will be::

```
01/03/2020 00:00:00
```

In February there are never more than 29 days, so the calendars counts on to March 1. In this case 2020 was a leap year.

If you would like to use a different notation for time display, simply combine the time components freely:

```
>> display := ['day month year'].
```

```
>> date := Moment new.
```

```
display
  year: date year plain,
  month: date month,
  day: date day.
```

```
Out write: display, stop.
```

```
6 11 2023
```

Do not forget to take into account the various time zones to avoid any surprises. The time zone (**zone:**) in the British variation of Citrine will display as:

```
>> m := Moment new.
```

```
Out write: m zone, stop.
```

```
Europe/London
```

By sending the message **time** you'll get the number of seconds that have passed since January 1, 1970. This is a reference point in time from which most computers calculate system time.

```
>> now := Moment new time.
```

```
Out write: now, stop.
```

```
1.595.961.187
```

You can also create a new **Moment** based on such a *timestamp*:

```
>> now := Moment new: 0.
```

```
Out write: now year plain, stop.
```

```
1970
```

These timestamps can be useful for sharing time data with other systems. They can equally be used to create an elegant time description, commonly desired in various programs:

```
Moment on: ['describe'] do: {
  >> now := Moment new.
  >> description := ['-'].
  >> difference := now time - self time.
  {
    (difference < 60) true: {
      description := ['right now'].
    }, break.
    (difference < 3600) true: {
      >> minutes := (difference / 60) round.
      description := ['some minutes ago'] some: minutes.
    }, break.
    description := (
      ['day-month-year']
      day: (now day),
      month: (now month),
      year: (now year plain)
    ).
  }
}
```

```

    } procedure.
  <- description.
}.

>> m := Moment new.
Out write: m describe, stop.

>> m := Moment new: (Moment new time - 1,500).
Out write: m describe, stop.

>> m := Moment new: (Moment new time - 5,000).
Out write: m describe, stop.

```

Result:

```

right now
25 minutes ago
6-11-2023

```

Note that in the above example everything is placed between brackets {}, followed by the message **procedure**. This is an optimisation in order to speed up the program. Putting the everything into a process block, enables the usage of the message **break** after each conditional, in order to prevent that excessive conditionals are also processed.

To calculate times, simply add up each separate time component. For example, in case you would like to know what time it will be within an hour:

```

>> m := Moment new.

Out write: m, stop.

m add: 1 hour.

Out write: m, stop.

```

```

06/11/2023 22:07:48
06/11/2023 23:07:48

```

See how a trick is applied here to benefit readability: **add: 1 hour**. In fact, the message **hour** is sent to 1, which is a qualifier (explained in chapter 3).

In this same way, you can add, for instance, 100 seconds. There is no need to focus on conversion into the right units, as the Moment object will do that for you (using the qualifier).

Now, take a look at a more complicated example regarding the usage of **add:** and **subtract:**:

```
>> then := Moment new
      year: 2020,
      month: 2,
      day: 29,
      hour: 23,
      minute: 59,
      second: 59,
      subtract: 1 second,
      add: 2 second,
      subtract: 9000 hour.
```

Out write: then, stop.

The result: :

```
20/02/2019 00:00:00
```

To create a copy of a moment do:

```
>> then := Moment new.
>> now := then copy.
Out write: then = now, stop.
Out write: (then equals: now), stop.
```

Result:

```
True
False
```

In the above example a copy is created of the moment **then**. Both moments describe the same point in time, which is the reason why `=` results in **True**. However, note that it remains only a copy, not the original. Because it physically is, indeed, a different object, at a different location in the memory of the computer, the message **equals:** can only result in the answer **False**.

An additional feature of the Moment object is to momentarily halt the process. You can pause the execution of the program by using the message **wait:** followed by the number of seconds. Take a look at the next example for a brief illustration of this principle:

```
>> before := Moment new.
Moment wait: 2.
```

```
>> later := Moment new.  
Out write: before, stop.  
Out write: later, stop.
```

Output:

```
06/11/2023 22:15:54  
06/11/2023 22:15:56
```

After 2 seconds, the above program fragment will show the output of two moments: the recorded moment before the pause and the recorded moment after that. As a result, the time descriptions on your screen should show a difference of 2 seconds.

3.12 The Program Object

This chapter describes the Program object, and this is the last object to review in the series of basic system objects. The basic function of the Program object is to communicate with the outside world, similar to the File object and the Pen object. These three represent the gateway to the outside world. Furthermore, this object contains a number of functions to benefit advanced memory management. You can also load other Citrine programs, utilising the Program object. To combine various Citrine programs, you can send the message **use:**. In this example there are two Citrine files, which are named **tool** and **program**:

Content of tool:

```
>> Tool := Object new.
```

Content of program:

```
Program use: Path tool.
```

```
Out write: Tool type, stop.
```

```
ctren program
```

When you run the Citrine program titled **program**, the code from the file **tool** will be loaded thanks to the message **use:**.

```
Object
```

In this way, you can include other people's work in your own program and make use of functionality provided by external libraries.

During the execution of the included code, the working directory will be changed to its parent folder.

You can execute system commands from Citrine, by sending the message **os:** to **Program**. This allows you, for example, to copy files, to relocate files, or invoke other software to execute specific tasks. On a Linux system, for example, you can apply the following step to show the files in the current folder:

```
Out write: (Program os: Command ls).
```

The above-mentioned code executes the command **ls** from the system shell, and returns the result in text form. The **Command** object is a simple tool, which can be used instead of a Text object. You can specify commands for the **Program**, either as text or as an **Command** object. The next lines are identical:

```
Out write: Command Hello World, stop.
```

```
Out write: ['Hello World'], stop.
```

And they will show the exact same result:

```
Hello World
Hello World
```

Regarding the uniformity principle, which aims to keep the Citrine code as pure as possible, the use of the Command variation is preferred. Naturally, the commands which are specified by means of the Command object, or which are sent as text to the command line are system dependent. In short, which commands you should use depends on the operating system in question. The command-line message serves merely as an intermediary.

The standard input (*stdin*, depending on your system) can be read by sending the message **input**, to the Program object. In return **Program** will provide you a Text object containing the transmitted information. In this way, you can create a small program to sort the input from the command line:

```
Out write: (
  Program input
    split: [' ',''],
    sort: { :a :b <- (a > b). },
    combine: [' ','']
).
```

```
echo "c,a,b" | ctren test.ctr
a,b,c
```

A program can also export data to other programs (through *stdout*). In general, this can simply be achieved via the Pen object, which you normally use to write to the screen. However, in case the

output of your program is connected to the default input of a different program, the data you are writing will flow into the input channel (stdin) of that other program (**Program input**).

You can write texts to the error channel (stderr) by sending the message **error:** to the **Program**, for example:

```
Program error: ['An error occurred.'], stop.
```

Such notifications often end up in logs and can be used to locate program errors at a later time.

Depending on your system settings, screen output or output to other programs is not executed constantly, but in chunks, which is referred to as buffering. This is done for efficiency purposes. Should you prefer to enforce emptying the output buffers of the program, you can do so by sending the message **flush**:

```
Program flush.
```

In case your Citrine program is started at the command line, you can read possible parameters by using the message **argument: X**, thereby replacing **X**, with the location of the parameter you wish to retrieve, as in:

```
Out
write: (Program argument: 3),
write: (Program argument: 4), stop.
```

```
ctren test.ctr hello world
helloworld
```

Please pay attention to the counting order of the arguments:

Argument #1: ctren (Citrine)

Argument #2: test.ctr (your program)

Argument #3: the first parameter following your program (hello)

Argument #4: the second parameter following your program (world)

Likewise, you can retrieve the number of arguments with **arguments**:

Out write: Program arguments, stop.

In case of the above-mentioned fragment the result will be 4.

In addition, you can retrieve the so-called environment variables, for example:

```
proverb="There is no place like home" ctren test.ctr
```

In your program the setting **proverb** can be retrieved with the message **setting::**

Out write: (Program setting: ['proverb']), stop.

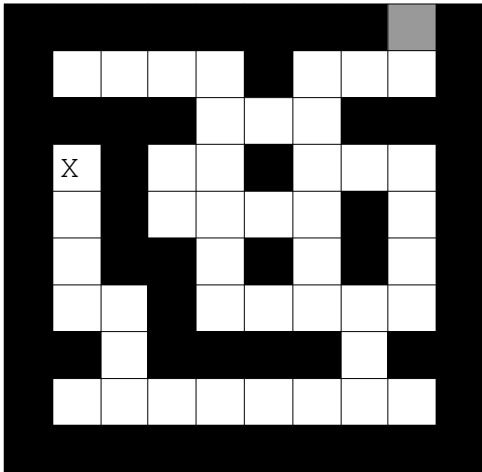
You can also set environment variables yourself using **setting:value::**

```
Program
  setting: ['program_status']
  value: ['started'].
```

By sending the message **ask** to **Program**, the program will wait for the user's reply. The following concise text game will illustrate this very principle.

In this game the player navigates through a haunted castle in order to find a treasure. A text game is ideal to put acquired knowledge into practice. It is free from graphic systems, which makes the program fairly independent and rather timeless, and it bypasses useless complexity. The workings of a text game are simple, as the program reflects the present situation to which the user responds by using a textual command.

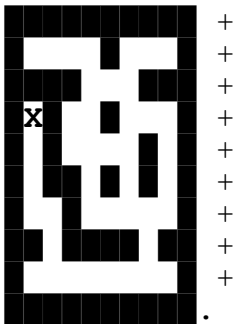
A haunted castle holding a treasure is programmed in the following example. First, a simple grid is created to represent the hallways of the castle. In this example, a grid measuring 10 x 10 is drawn up, and the illustration below shows a possible concept. The outside edges represent the enclosure, with on the top-right the entrance door and on the top-left, with X marking the spot, the place where the treasure is hidden.



In order to transform this grid into a game, it is essential to be able to save a representation of it in the memory of the computer. To accomplish this we use a space to represent a hallway, a cube to represent a wall, and the letter **X** to represent the treasure.

The code will look like this:

```
>> map :=
```



Here, the binary message + is used to spread the range of characters over multiple lines in order to make it visually attractive.

Now that the map of the castle has been coded as a Text object, you can decide from where the player is going the start. The player will, in this case, start at the top-right side, or more specifically: on the ninth square of the second line. Given that each line is composed of ten squares and all the lines together represent one long sequence, the correct location can be found with:

$$1 * 10 + 9 = 19.$$

In order to start on the second line, it is necessary to jump over the first ten squares on the first line. Next, the correct start location is, indeed, nine squares further. So, to capture the player's start location do:

```
>> location := 19.
```

The rest of the game consists of two tasks which are connected by a **while** message. In fact, the gist of the game is:

```
{ search... } while: { treasure not found }.
```

Now, take a look at the second task first, as this is the most straightforward one. Assuming the variable **location** indicates the player's current location inside the castle, the location of the treasure can be checked as follows:

```
(map character: location) ≠ ['X']
```

The search will continue as long as there is no treasure on the player's location.

Searching is basically asking in which direction the player likes to go. This is followed by checking whether the preferred direction is actually valid, in order to avoid players bumping into a castle wall. If the chosen direction is indeed possible, an adjustment of the player's location will be necessary. The whole of it will look like the following:

Program ask

```
case: ['n'] do: { x := -10. },
case: ['s'] do: { x := 10. },
case: ['e'] do: { x := 1. },
case: ['w'] do: { x := -1. }.
```

As long as the treasure has not been found, a text is printed in which the player is asked to choose a direction. The actual direction is asked for by the message **ask** to Program, which will return a Text object. To this Text object a chain of **case:do:** messages is sent, whereby a task is linked to a possible answer of the player. For example, should a player respond with *n* (north), then *x* should equal -10. In this case, *x* is equal to the intended motion. So, *x* = -1 means 1 position to the left. If *x* = -10, then this means 10 positions to the left, which is in fact 1 position up. Consequently, it is calculated if there actually is a passage according to the map. This is illustrated in the following example:

```
>> passage := map character: location + x.
```

x is added to the current location and the map is asked which character is written on this position. In case there is a wall, the player cannot move forward and the question has to be repeated without updating of the player's current location. If not, the player can be moved.

```
>> passage ≠ ['■'] true: {  
    location add: x.  
}, else: {  
    Out write: ['Ouch! You bumped into a wall!'], stop.  
}.
```

The whole computer game looks like this:

```
>> map :=
```

```
['■■■■■■■■■■'] +  
['■ ■■■■■■ ■'] +  
['■ ■■■■■■ ■'] +  
['■ ■■■■■■ ■'] +  
['■ ■■■■■■ ■'] +  
['■ ■■■■■■ ■'] +  
['■ ■■■■■■ ■'] +  
['■ ■■■■■■ ■'] +  
['■ ■■■■■■ ■'] +  
['■■■■■■■■■■'].
```

```
>> location := 19.
```

```
{  
    Out write: ['Which direction?'], stop.  
  
    >> x := 0.  
  
    Program ask  
  
    case: ['n'] do: { x := -10. },  
    case: ['s'] do: { x := 10. },  
    case: ['e'] do: { x := 1. },  
    case: ['w'] do: { x := -1. }.  
  
    >> passage := map character: location + x.  
  
    >> passage ≠ ['■'] true: {  
        location add: x.
```

```

    }, else: {
        Out write: ['Ouch! You bumped into a wall!'], stop.
    }.
} while: { <- (map character: location) ≠ ['X']. }.

```

The beginning of a game session could look like this...

```

Which direction?
n
Ouch! You bumped into a wall!
Which direction?
w

```

In the above example the message **ask** is used to ask the player a question. In addition, the player's input is printed on screen. In some situation this is not desirable, for example regarding passwords. In that case, you can send the message **ask password** to the Program object. Consequently, the Program object will ask for user's input without printing the results of pressed keys on screen, for example:

```

Out write: ['Enter your password ... '], stop.
>> password := Program ask-password.
Out write: ['Repeat:'], stop.
>> control := Program ask-password.
(password = control) true: {
    Out write: ['Passwords match!'], stop.
}, else: {
    Out write: ['Passwords do not match!'], stop.
}.

```

An example of a possible output:

```
Enter your password ...  
Repeat:  
  
Passwords do not match!
```

To end the program prematurely you can use the message **end**.

```
{ :i  
  
  Out write: i.  
  
  i = 5 true: { Program end. }.  
  
} * 10.
```

```
12345
```

Finally, there are a number of messages relating to the system memory. You can retrieve information regarding memory use with the message **memory**:

```
Out write: Program memory, stop.
```

Possible output (illustrative):

```
Sequence ← 80.568 ; 0 ; 0 ; 0 ; 0
```

Here, the first number indicates the memory in available bytes, which the program can use. The second number indicates the number of objects located in the memory. The third number indicates the objects that cannot be cleared yet; so, objects that are kind of stuck. The forth number indicates the number of objects which during the last cleanup could not be cleared to free memory. The final number indicates the number of objects that were cleared during the last cleanup (the waste). Note that this information is system-dependent as it works on bytes, which is rather rare for Citrine. At the start of a program, the amount of preferred available memory can be set. At the moment of writing, the standard is 10 MB. The memory amount which your program is allowed to use can be set with **memory::**

Program memory: MB.

The number of bytes can be specified with the qualifiers MB or KB. Note that it is also possible to set the memory using an environment variable (see, Appendix C).

Furthermore, it is possible to have Citrine clean up at any desired moment by using the message **clean up**.

```
Program clean up.
```

As soon as you send the message, Citrine will try to clean up all unused objects in order to increase the available memory space for your program. It is also possible to instruct Citrine how to do the memory management for you. This is set by utilising the message **memory management:**, which is defaulted on **1**. If you do nothing, your program is set as follows:

```
Program memory management: 1.
```

See table for possible settings below:

Code	Meaning
0	Object are never cleaned up. Your program becomes faster, however the memory fills up rapidly
1	When nearly 80% of memory is used by your program, Citrine will try to free up space by cleaning up objects.
4	Citrine cleans up continuously. Your program becomes slow, however memory use is minimal.
8	Whenever possible, memory is split into standard blocks with a fixed size. The blocks are recycled, which is faster. Objects are never cleaned up.
9	9 Similar to setting 8, however the blocks are truly recycled.
12	Combination of setting 4 and 8

3.13 Exercises

1. Write a program that shows the temperatures from -10 to 40 degrees Celsius into the corresponding degrees Fahrenheit (multiply by 1.8 and then add 32).

To help you start:

```
>> celsius := -10.
{
  >> fahrenheit := ...
  Out write: fahrenheit, stop.
} while: { <- ... }.
```

2. This program expresses the saying ['Strike while the iron is hot']. For that purpose the template tile is adjusted. What was the original text on the tile?

```
>> tile := ['...'].
tile buy: ['strike'], cheap: ['hot'].
Out write: tile, stop.
```

3. Expand the Number object using the message **square** which returns the square of that number.

To help you start:

```
Number on: ['...'] do: { <- ... }.
```

4. Create a new object **Distance**, which accepts two points and can calculate the distance between these points in order to:

```
>> Point := Object new.
Point on: ['x-coordinate:'] do: { :x own x := x. }.
Point on: ['y-coordinate:'] do: { :y own y := y. }.
Point on: ['x-coordinate'] do: { <- own x. }.
Point on: ['y-coordinate'] do: { <- own y. }.
>> pier := Point new x-coordinate: 5, y-coordinate: 6.
```

```
>> town-hall := Point new x-coordinate: 7, y-coordinate: 1.
```

```
>> Distance := Object new.
```

```
...
```

```
>> x := Distance new  
    start: pier  
    end: town-hall,  
    number.
```

```
Out write: x, stop.
```

In case the output returns the number 7.

Clue: send the message **absolute** to turn a negative number into a positive one.

5. In the following code, what is the value of **x** ?

```
>> x := 2 + 3 four + 5.
```

6. Convert the following task into a message for **Number**:

```
>> Double := { :number <- number * 2. }.
```

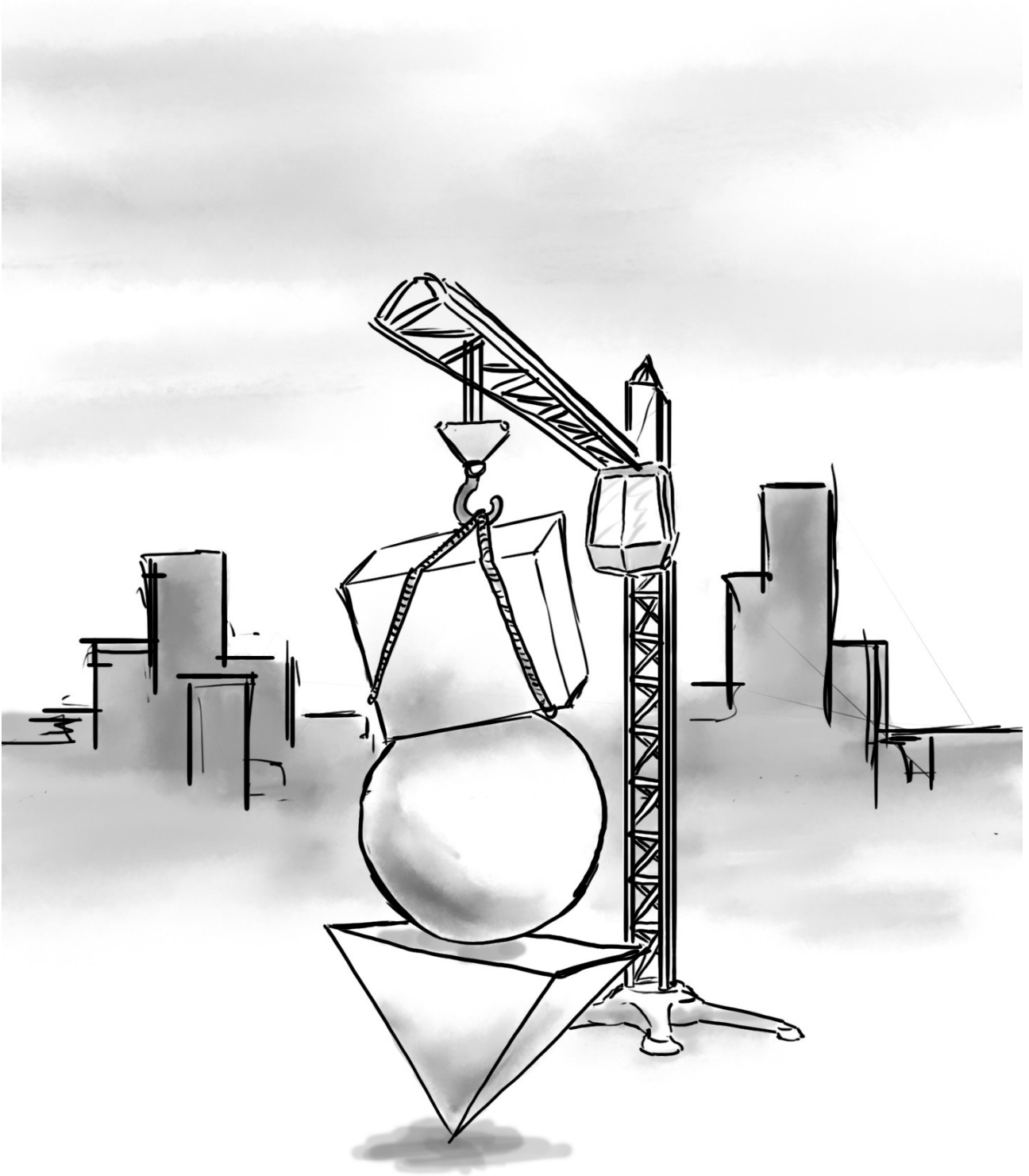
```
>> x := Double apply: 4.
```

```
Out write: x, stop.
```

7. Expand the Number object with a message **digits:**, with which a number containing a fixed quantity of digits can be displayed (2 → 02, 5 → 005 etc...).

```
Out write: (9 digits: 9), stop.
```

```
000000009
```

4. Advanced

4.1 Copying

By using `:=` it is possible to save objects under a given name (i.e. assigning to a variable). It is equally possible to save an object under multiple names. Please note that this action is different from making an actual copy.

```
>> sheep := ['Dolly'].  
>> clone := sheep.  
clone replace: ['l'] with: ['n'].  
Out write: sheep, stop.
```

```
Donny
```

Here, you might have expected that the output would be Dolly, instead of Donny. However, this is not the case as both names refer to the same object. When working with a loop, something similar occurs:

```
>> points := Sequence <- 1 ; 2 ; 3.  
points each: { :number :quantity quantity add: 1. }.  
Out write: points, stop.
```

In fact, Citrine *always* uses references, so on **:quantity** the above-illustrated loop also indicates the reference to the element in the sequence:

```
Sequence <- 2 ; 3 ; 4
```

In order to copy an object, you have to specify this action *explicitly*:

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

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

```
Out write: sheep, stop.
```

```
Dolly
```

By sending the unary message **copy** to a Text object, the object returns a copy of that same Text object. It is possible to copy Number objects, Boolean objects, Sequences, Lists and Moment objects in this same way.

You can also define your own copy implementation, this is even a necessity if you create your own objects. Let's create an alternative copy implementation for sequences. The default implementation of copy for a sequence makes a shallow copy, it creates a new list with the same elements.

```
>> a := Sequence ← 1 ; 2 ; 3.
```

```
>> b := a copy ; 4.
```

```
Out write: a, stop write: b, stop.
```

```
Sequence ← 1 ; 2 ; 3  
Sequence ← 1 ; 2 ; 3 ; 4
```

In this case, 4 is added only to the copy.

However, because the copy is shallow, the objects in both sequences are the same:

```
>> sheep := Sequence ← ['Dolly'] ; ['Shaun'].
```

```
>> clones := sheep copy.
```

```
clones each: { :number :sheep
```

```
    sheep append: ['2'].
```

```
}.
```

```
Out write: sheep, stop.
```

```
Out write: clones, stop.
```

So if we append 2 to each name in the copy, the original sequence is affected as well:

```
Sequence ← ['Dolly2'] ; ['Shaun2']  
Sequence ← ['Dolly2'] ; ['Shaun2']
```

To remedy this we need to make a *deep copy*. Such a copy action for a sequence could be composed as follows:

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

You can use it like this:

```
>> sheep := Sequence ← ['Dolly'] ; ['Shaun'].  
  
>> clones := sheep copy.  
  
clones each: { :number :sheep  
  
    sheep append: ['2'].  
  
}.
```

```
Out write: sheep, stop.
```

```
Out write: clones, stop.
```



```
Sequence ← ['Dolly'] ; ['Shaun']
Sequence ← ['Dolly2'] ; ['Shaun2']
```

Should you remove the copy action, the result would be two prints of the second sequence. At first sight, it seems that the addition merely copies the sheep in the main sequence and leaves out the subsequences. However, should the calling code be adjusted into:

```
>> sheep := Sequence ← ['Dolly'] ; (Sequence ← ['Shaun']).
```

then, despite **Shaun** being in a subsequence, it is indeed nicely copied. How does this work? Well, it has everything to do with naming conventions. Because the copy action has been connected to the message named **copy**, all objects will respond with their respective copying behaviour, it does not matter whether the receiving object happens to be Text or a (sub)Sequence.

Also, note the message *recursive*, this message is necessary to send before the copy message. More about this message in chapter 4.4

It is essential to keep in mind that although a copy of an object often has the same appearance of the original, it will, in fact, never be the same. The root object defines a message **equal:**, which can be used to compare the identity of objects. Take a look at the following example:

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

```
Out write: a = b, stop.
Out write: a = c, stop.
```

```
Out write: ( a equals: b ), stop.
Out write: ( a equals: c ), stop.
```

```
True
True
False
True
```

In this case, $a = b$, because a is a copy of b and the message `['=']` looks at the **value of the object**, which in the above, is the number. Given that $a = 1$ and $b = 1$ and $1 = 1$, the answer will be **True**. The same applies to c . It is only when the message **equals:** is used, which **Number** inherits from **Object**, that a difference can be noted. While c equals a , because here the reference is assigned by utilising the symbol `:=`, a **is not** (equals) b . This is explained by the fact that inside the computer

memory the copy is physically a different object. So, to find out if indeed the reference is to the same physical object in the computer memory, you can apply the message **equals:**.

4.2 Visibility

As you know, an object can be assigned to a variable, however, first you need to allocate memory in the memory under this name, and to do this you use a declaration symbol. This action is called *declaring a variable*. Of course, this has already been discussed in the previous chapters. However, not all sections of the memory are visible everywhere, because in large programs, which could possibly carry code of third parties, names are likely to clash. For this reason, these memory locations are separated from each other, whereby tasks form the dividing lines. Here should be taken into account that a variable that is declared within a task, is solely visible during the execution of that specific task and during the execution of all the tasks *initiated* by this task. When a task in which the variable is declared has ended, it will be sort of forgotten. Have a look at the following example:

```
>> x := 9.  
{ Out write: x. } start.
```

The number **9** is printed on screen by the above-shown program. Object name **x** has been declared outside the task and consequently visible everywhere, to all the tasks, even when **x** is sent along as argument to the Pen object. The area outside the task can be considered as a kind of umbrella task. All the variables which are declared in this area are visible everywhere in the program, indeed, all tasks are within the boundaries of this umbrella task, as it were. Variables which are declared outside all the tasks of the program are also known as *global variables*, because of their *global* visibility.

```
{ >> x := 9. } start.  
Out write: x.
```

In this case an error message will occur. Object name **x** is forgotten as soon as the task ends. Given that **x** has been declared within the task, it is not visible outside it. In this case **x** lives exclusively within the small task that has been written. Now, have a look at the next fragment, there are no error messages; however, which number will appear on screen?

```
>> x := 1.  
{ >> x := 9. } start.  
Out write: x.
```

Here, the correct answer is **1** instead of **9**. In this case, there are two locations in memory by the name **x**. Thanks to the boundaries between the tasks, they do not affect one another. During the

execution of the task, **x** equals **9**, and as soon as the task has ended, the **other x** becomes visible again and **x** will be equal to **1**.

And now for the trickiest version:

```
>> y := { x := 2. }.  
{ >> x := 1. y start. Out write: x. } start.
```

Which one will appear on screen: **1** or **2** ? Let's see. First of all, a location is allocated in memory under the name **y** for a task. In that particular task, the number **2** is saved under the name **x**. However, this task is not initiated. On the second line a new task is created, which starts right away. In this new task, **x** is declared. So, a location in memory is allocated under the name **x**. Consequently, under the name **x** the number **1** is saved. In that moment $x=1$. Next, the task named **y**, which has been written on the first line of this program, is initiated. During the execution of that task, the number **2** is being saved under the name **x**. Is this at all possible? The answer to this question is a resounding yes. Here, the crucial line is indeed **during execution**. Although task **y**, from a visual point of view, falls outside the task on line **2**, the task **y** is indeed being executed during the execution of the task on the second line. This means that, the **1** that was in **x**, is overwritten by the number **2**. In fact, at that moment $x = 2$. Then, the task **y** has ended and the execution of the program returns to the task on the second line, ready to print **x** on the screen. However, what is inside **x** now that task **y** has ended? To clarify, only **x** was declared outside **y**, namely in the task which is currently processed, the one on the second line, so **x** still contains the number **2** and as result will become visible on the screen. This mechanism, as just described is also called *Dynamic Scoping*. For comparison, in JavaScript a similar code results in 1:

```
y = function() { x = 2; };  
(function(){ var x = 1; y(); console.log(x); })()
```

1

In the above-shown examples, separate Task objects are used each time. The same can be illustrated on the basis of other objects. The next example illustrates how the flavour of the ice cream is consistently modified into *malaga*:

```
>> Ice-cream-parlour := Object new.  
Ice-cream-parlour on: ['ice-cream'] do: {  
    flavour := ['Malaga'].
```

```
}.  
  
Ice-cream-parlour on: ['visit'] do: {  
    >> flavour := ['Pistachio'].  
    Ice-cream-parlour ice-cream.  
    Out write: flavour, stop.  
}.  
  
Ice-cream-parlour visit.
```

Consequently, the output of the program is:

Malaga

As soon as the ice cream parlour receives the message **ice cream**, the flavour is being adjusted into **Malaga**. To persons whom have never worked with any other programming language save Citrine, this would not be very surprising. After all, this totally makes sense according to general procedures of Citrine. However, in most of the other programming languages, the output would be **Pistachio**, or they would even return an error message, depending on the rules of visibility.

4.3 Handling Unknown Messages

If you would like to create an object that will automatically show on screen, all words you send to it (having multiple characters) on individual lines, you could write something like this:

```
Out Humpty Dumpty sat on the wall.
```

```
Humpty
Dumpty
sat
on
the
wall
```

You can achieve this by using the so-called *generic* messages. This technique helps you to connect a task to undefined messages:

```
Out on: ['respond:'] do: { :text
    Out write: text, stop.
}.
```

The task that is connected to **respond:** runs at each unknown message. Given that the message **Humpty** is unknown, Citrine will internally run the following message:

```
Out respond: ['Humpty'].
```

There are various applications conceivable for general messages. One of its main applications is in combination with principle of uniformity; namely, the translation of Citrine software into other technical languages. Suppose that you would like to write HTML code in Citrine, in that case a HTML object could be created in order to convert messages into tags:

```
>> HTML := Object new.

HTML on: ['respond:and:'] do: {
    :tag :content

    <- ['<tag>content</tag>']
        tag: tag - [':'],
        content: content.
}.

Out write: (HTML H1: ['Title']), stop.

Out write: (HTML P: ['Body text']), stop.

Out write: (HTML B: ['bold text']), stop.
```

```
<H1>Title</H1>
<P>Body text</P>
<B>bold text</B>
```

Note that **respond:and:** is used, because two arguments follow. It is equally possible to capture generic messages containing three or four arguments; respectively, **respond:and:and:**, and **respond:and:and:and:**.

Now, for example, when you add the following message:

```
HTML on: ['respond:and:and:'] do: { :message :content :value
    >> share := message split: [':'].
    >> tag := share ? 1.
    >> attribute := share ? 2.
    <- ['<tag attribute=['value']>content</tag>']
        tag: tag,
        content: content,
        attribute: attribute,
        value: value.
}.
```

we can create a HTML hyperlink like this:

```
Out write:(
    HTML
    A: ['Link']
```

```
      HREF: ['https://citrine-lang.org']  
    ), stop.
```

Result:

```
<A HREF="https://citrine-lang.org">Linkje</A>
```

This mechanism to automatically convert messages into something else, is also used internally by Citrine. The objects **Path** and **Command** both use this mechanism behind the scenes.

4.4 Inheritance, Overriding, Recursion

Chapter 2 has illustrated that objects can be based on other objects. In this way, you can reuse previously written code to create a hierarchy of objects. For example:

```
>> Animal := Object new.  
>> Dog := Animal new.  
>> Poodle := Dog new.
```

In this case, **Dog** inherits all the behaviour of **Animal** and, in turn, the object **Animal** inherits all behaviour of **Object**, the root object of all objects.

In the following example a new kind of sequence is created: **Combination**, in which each element is ensured to be unique. The functions of the **Sequence** are reused through inheritance:

```
>> Combination := Sequence new.  
Combination on: ['append:'] do: { :element  
    >> seen := self find: element.  
    seen false: { self append: element. }.  
}.  
  
>> colours := Combination new.  
  
colours  
append: ['red'],  
append: ['green'],  
append: ['blue'],  
append: ['red'].  
  
Out write: colours, stop.
```

Observe how the second **red** is excluded from the sequence:

```
Sequence ← ['red'] ; ['green'] ; ['blue']
```

Occasionally, it may be necessary to override a behaviour of an object. For instance, when adding numbers and units of measurements have to be taken into account. In the following example the object **Size** is created. This object returns a number which, during the addition, takes into account the unit of the number to be added. The **Number** object checks whether it is about *inches* or *foot*. The programming code could resemble the following:

```
>> Size := Object new.

Size on: ['is:'] do: { :number

    number on: ['+'] do: { :number

        >> unit := number qualifier.

        >> factor := 1.

        unit
        case: ['inch'] do: { factor := 0.0833. }.

        >> answer := self + (number * factor).

        <- answer.

    }.

    <- number.

}.
```

This size number can be used as follows:

```
>> plank := Size is: 6 foot.
>> edge := Size is: 50 inch.
Out write: plank + edge, stop.
```

This will show as:

```
10.165
```

In the previous example, the plus sign is being overridden. Note that eventually the final addition still has to be done, which, in fact, takes place on the following line:

```
>> answer := self + (number x factor).
```

Now, *how does Citrine understand that this plus sign refers to the original logic of addition?* For instance, another interpretation could be that Citrine will repeatedly send the same message to the same object which, in turn, would result into an *endless loop*. Clearly, this is not the intention. As soon as you send a message to an object, which would execute the *exact* same code, Citrine will realise, in this case, that the underlying, overridden message is meant. So, your program will automatically be safeguarded against this form of endless loops. However, when it is indeed your aim to run the same task from the current task, it will be necessary to first send the message **recursive**. In this case, the result will be an infinite loop. However there are useful applications for recursive tasks, for instance, suppose that you want to calculate the factorial of a given number. In that case, simply add the message **factorial** to Number:

```
Number on: ['factorial'] do: {  
    >> answer := 1.  
    self > 0 true: {  
        >> previous := self.  
        >> next := previous - 1.  
        answer := previous * next recursive factorial.  
    }.  
    <- answer.  
}.
```

This calls for recursion. In fact, the task that is connected to the message **factorial**, needs to be executed again within that task. So, it is necessary to invoke the factorial task from within the factorial task itself. As a rule, Citrine will prevent this routine, because of the risk of ending up in an infinite loop. For this reason, it is vital to precede the message by the word **recursive**. This is done in order to make Citrine aware of the fact that it is your intention to execute the same task, and that you did not make mistake. The following loop shows the factorials of the numbers 1-10:

```
{ :x Out write: x factorial, stop. } * 10.
```

```
1
2
6
24
120
720
5.040
40.320
362.880
3.628.800
```

Without the word **recursive**, only the initial multiplication would take place. The list would end up as follows:

```
0
2
6
12
20
30
42
56
72
90
```

4.5 Initial State

Creating an object that is set in a given initial state, presents a frequent issue. Suppose an object **Rectangle** has to be created, in order to calculate perimeter and area. A possible notation would be:

```
>> Rectangle := Object new.  
Rectangle on: ['area'] do: {  
    <- own length * own width.  
}.  

```

Obviously, setting a length and width is a precondition. For that purpose, the messages **length:** and **width:** can be added:

```
Rectangle on: ['length:'] do: { :length  
    own length := length.  
}.  
Rectangle on: ['width:'] do: { :width  
    own width := width.  
}.  

```

This rectangle can then be utilised as follows:

```
>> rectangle := Rectangle new length: 2, width: 3.  
Out write: rectangle area.
```

Result:

```
6
```

However, if the initial settings of length and width are overlooked, an error message will occur:

```
>> rectangle := Rectangle new.  
Out write: rectangle area.
```

```
Key not found: length
```

In order to prevent this, it is preferable that a rectangle should always have an initial length and width, for example 0. So, in this case, it is necessary to *override* the message **new**.

```
Rectangle on: ['new'] do: {  
    >> rectangle := self new.  
  
    rectangle length: 0.  
    rectangle width: 0.  
  
    <- rectangle.  
}.  

```

Notice that on the second line the first **initial new** is invoked again. However, this will not result into an infinite loop, as Citrine will prevent this. The mechanism behind this, has been highlighted previously in chapter 4.4. When a new rectangle is created, it will now automatically measure 0 by 0, right from the start. In this way, no error will occur while computing the area:

```
>> rectangle := Rectangle new.
```

```
Out write: rectangle area.
```

Result:

```
0
```

```
>> rectangle := Rectangle new length: 2, width: 3.
```

```
Out write: rectangle area.
```

Result:

4.6 Implicit Conversion

Citrine uses *implicit conversion* to convert objects. To print a sequence on screen, Citrine will, for example, send the message **text** internally to the sequence. This can prove very useful, in case you would like to print a sequence as a comma-separated list. The message **text** can be overwritten:

```
>> sum := Sequence <- 1 ; 2 ; 3.  
  
sum on: ['text'] do: {  
    <- self combine: [','].  
}.  
  
Out write: sum, stop.
```

And this way we get:

```
1,2,3
```

With mathematical operations on numbers, the message **number** is sent internally. In this way, it is possible to create a sequence which automatically provides the sum during a mathematical operation:

```
>> sum := Sequence <- 1 ; 2 ; 3.  
  
sum on: ['number'] do: {  
    >> total := 0.  
    self each: { :number :element  
        total add: element.  
    }.  
    <- total.  
}.  
  
Out write: 1 + sum, stop.
```


Another example is combining a sequence of objects into a Text object. Suppose you have an address book, which is filled with objects of the type **address**:

```
>> Address := Object new.
Address on: ['street:'] do: { :street
    own street := street.
}.
Address on: ['number:'] do: { :number
    own number := number.
}.
Address on: ['addition:'] do: { :addition
    own addition := addition.
}.
```

This makes it possible to enter an address in a structured way. However, when the address is ready to be printed, it is preferable to read a text that follows the common notation of: property number, possible addition, and street name.

This is why it is necessary to implement a text message:

```
Address on: ['text'] do: {
    <- own number + own addition + [' '] + own street + [' '].
}.
```

The address object is used in the following way:

```
>> a := Address new
    number: 12,
    addition: ['a'],
    street: ['Church Street'].
```

```
>> b := Address new
      number: 3,
      addition: ['c'],
      street: ['Abbey Lane'].
```

Next, you complete the address book:

```
>> addresses := Sequence ← a ; b.
```

The textual representation of the sequence is adapted to:

```
addresses on: ['text'] do: { <- self combine: ['↵']. }.
```

To finish, you write the address book to the screen:

```
Out write: addresses, stop.
```

The result:

```
12a Church Street
3c Abbey Lane
```

Now, how does Citrine know that the addresses should be printed in this mode? Well, as soon as **write:** is sent to the Pen object, the sequence of addresses is converted into a text. To achieve this, Citrine internally sends the message **text** to the sequence. This is an implicit conversion, which is nothing new. However, in this example, a task for **text** has been arranged within the address book. This task combines all elements, only to be separated by line breaks. Normally, a sequence would show very different on screen, however, by explicitly specifying how the sequence is to be represented as text, Citrine can adopt this. The next step in the process is for the message **text** to be sent to each individual element. In this code it has been specified that each time an address object receives the message **text**, it converts the address into text according to the prearranged format. Eventually, this result is shown in the output. So, it takes basically two steps to get a result: the textual conversion of the address object and the textual conversion of the address book; which is, in fact, the sequence of addresses. Ultimately, the combination of these two implicit conversions ensures a proper summary of addresses.

Now suppose an address **c** is added to the sequence, for example:

```
>> c := Address new.  
>> addresses := Sequence ← a ; b ; c.
```

This will obviously return an error message, because on **c** there is no street name, property number and addition filled in. To prevent this from happening, use the method which has previously been explained in chapter 4.5:

```
Address on: ['blank'] do: {  
    own number := [].  
    own addition := [].  
    own street := [].  
}.  
Address on: ['new'] do: {  
    >> address := self new.  
    address blank.  
    <- address.  
}.
```

Once again, the task associated with the **new** message is being replaced with a task that fills out the required properties in advance.

Here is the complete code:

```
>> Address := Object new.  
  
Address on: ['street:'] do: { :street  
  
    own street := street.  
}.
```

```
Address on: ['number:'] do: { :number
```

```
    own number := number.
```

```
  }.
```

```
Address on: ['addition:'] do: { :addition
```

```
    own addition := addition.
```

```
  }.
```

```
Address on: ['text'] do: {
```

```
    <- own number + own addition + [' '] + own street + [' '].
```

```
  }.
```

```
Address on: ['blank'] do: {
```

```
    own number := [''].
```

```
    own addition := [''].
```

```
    own street := [''].
```

```
  }.
```

```
Address on: ['new'] do: {
```

```
    >> address := self new.
```

```
    address blank.
```

```
    <- address.
```

```
}.  
  

```

```
>> a := Address new  
      number: 12,  
      addition: ['a'],  
      street: ['Church Street'].
```

```
>> b := Address new  
      number: 3,  
      addition: ['c'],  
      street: ['Abbey Lane'].
```

```
>> c := Address new.
```

```
>> addresses := Sequence <- a ; b ; c.
```

```
addresses on: ['text'] do: { <- self combine: ['↵']. }.
```

```
Out write: addresses, stop.
```

Result:

```
12a Church Street  
3c Abbey Lane  
  

```

Note, the last line is empty (because of address c).

4.7 Serialisation

As seen previously, sequences and lists can be converted into texts. Another way to do this is by sending the message **code**. If you use this message, your objects can be brought to ['life'] again from text. This is called serialisation. After serialisation, the resulting text can be saved on disk or transmitted over a connection. As such, serialized objects can be shared with other systems and are therefore useful for communication between systems.

The following example shows a serialisation of a shopping basket for storage in a file, consequently it is read again and the amount of groceries is being counted:

```
>> groceries := Sequence new
  ~ ['syrup wafes']
  ~ ['cofee']
  ~ ['sprinkles'].

>> basket := File new: ['basket'].

basket write: groceries code.

>> x := File new: ['basket'], read object count.

Out write: x, stop.
```

3

There is a subtle difference between the messages **text** and **code**. The message **text** results in a textual representation of the object. Instead, the message **code** results in a representation in executable programming code. For comparison, here are two answers to the messages **text** and **code** of a Text object:

```
Out write: ['Ceci n'est pas une pipe'] text, stop.
```

```
Out write: ['Ceci n'est pas une pipe'] code, stop.
```

```
Ceci n'est pas une pipe
['Ceci n'est pas une pipe']
```

Note that in the second case the text output has been placed between single quotation marks, in order to enable Citrine to read and process this output again. As a rule, serialisation is not limited to lists and sequences alone. It is also possible to obtain code representations of other Citrine objects:

```
>> x := ['Out'] object.  
x write: 123, stop.
```

```
123
```

This trick can be easily applied when a textual representation of an object needs to be brought back into an object.

Now, this next example shows a simple sum which is applied to a number of variables:

```
Out write: (  
  { <- x + 1. } apply: ['a b c']  
) .
```

As a rule, this is not feasible in Citrine, however Citrine can be extended in a way that will allow for the sum **X + 1** to be applied to a series of variables (**a**, **b** and **c**). In that case, you can extend the Task object with the following functionality:

```
>> a := 1.  
>> b := 2.  
>> c := 3.
```

```
Task on: ['apply:'] do: {  
  :variables  
  >> answer := Sequence new.  
  variables split: [' '], each: {  
    :index :variable  
    >> x := variable object.  
    x := self start.
```

```

        answer append: x, stop.

    }.

<- answer. }.
.

```

The result:

```
Sequence ← 2 ; 3 ; 4
```

Now, if we would not have used **variable object** but just **variable**, the outcome would be completely different. After all, the action (**X + 1**) would be applied to the letters **a**, **b** and **c** instead of to the numbers **1**, **2** and **3**:

```
Sequence ← ['a1'] ; ['b1'] ; ['c1']
```

In this case, the number 1 would be added to the texts **a**, **b** and **c**.

Sending the message **object** to a text which refers to a non-existing object, a message error will occur:

```
>> x := ['magic lamp'] object.
```

The result will be:

```
Key not found: magic lamp
```

Neither can complete code fragments be executed in this way, because Citrine is protected against this way of code injection:

```
Out write: ['{ <- 1 + 1. } start'] object, stop.
```

```
Out write: ['1 + 1'] object, stop.
```

This will show no result:


```
None
None
```

Nonetheless, it is possible to exchange executable code via a Text object. In this case, it is necessary to capture the **code** at the object that you would like to convert into text. This technique is applied in the following example:

```
>> bouquet := Object new.

bouquet on: ['code'] do: {
    <- ['bouquet'] .
}.

bouquet on: ['text'] do: {
    <- ['roses'] .
}.

>> text := (Sequence <- bouquet ; ['card']) code.
Out write: text object first, stop.
```

The answer of this program is:

```
roses
```

The reason why this functions correctly, is because the message **code** is also sent automatically to all underlying elements of the sequence, consequently also to the object **bouquet**. The bouquet object answers with the reference to itself. During the conversion of the text to a Sequence object, the reference to the bouquet object is, indeed, restored. Consequently, when this object is printed on screen, the message **text** is internally sent to **bouquet** by means of implicit conversion, and will return the answer **roses**.

4.8 Alternative Message Structure

Suppose you have a sequence:

```
>> x := Sequence ← 1 ; 2 ; 3.
```

If you would like to delete the first and the last element, then your message would be:

```
x shift pop.
```

Unfortunately, the previous notation will not function correctly. The message **shift** returns the first element of the sequence, which, in turn, will become the receiver of the message **pop**. So, the message **pop** is not sent to **x**, but instead to 2. A possible solution to this problem would be to create two separate sentences, such as:

```
x shift  
x pop
```

However, this is rather impractical, in particular when you need to delete more than two elements. Therefore, Citrine offers an alternative message structure for these kinds of situations. In alternative message structures, the answers of objects are ignored and you will get the receiving object back as answer, time and again. An alternative message structure is initiated by sending the message **do** to an object and the alternative message structure can be ended by sending the message **done**. In the example illustrated, this could be applied as follows:

```
>> x := Sequence ← 1 ; 2 ; 3.
```

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

```
Out write: x, stop.
```

Note that in order to avoid unpleasant surprises always close with **done**. See what happens in the next example:

```
>> x := Sequence ← 1 ; 2.
```

```
x do shift.
```

```
Out write: x, stop.
```

The previous code will result in the error message: **Must answer with text**. The reason for this is that the message **write** wants to create a text of **x**; so, internally it sends the message **text**. As a rule, a Text object is returned as answer, however as the alternative message structure is still active, the sequence itself is being returned. This causes further processing to fail and Citrine will show an error message.

4.9 Programmatically-sent Messages

Instead of sending out a message directly to an object, it can also be sent via a variable. To do so, you use the message **message:arguments:.** For instance, in the next example the result shows **14** (in a very elaborate way):

```
>> a := (Sequence ← 7) .  
>> b := ['add:'] .  
>> x := 7 message: b arguments: a .  
Out write: x, stop.
```

Output:

```
14
```

You specify the intended message as Text object and arguments as sequence. A message containing two parameters (e.g. **between: 0 and: 10**), is specified as **between:and:**, thus, combining the message components:

```
>> a := (Sequence ← 1 ; 10) .  
>> b := ['between:and:'] .  
Out write: (Number message: b arguments: a), stop.
```

Output (possibly):

```
8
```

Thanks to programmatically-sent messages, objects can be converted into messages. This enables the end-user to communicate directly with objects. In the example shown below, a sort of mini word processor is created, simply by providing the user with a blank Text object. The user can draw up a text by using Citrine messages. These messages can be passed on to the result object.

```
>> result := [].

{
  Out write: result, stop.

  result

  message: Program ask

  arguments: (Program ask split: ['']).

}

while: { <- True. }.
```

This next illustration shows a transcript of a session:

```
$ miniwrite

add:
Hello
Hello
replace:with:
e,E
hEllo
capitals

HELLO
```

While this is the shortest word processor one can imagine, it's not very user friendly. It works quite simple though.

In the variable, **result**, the user of the program can modify the text through interactive communication with the object. In this session, the user fills the Text object with the word **Hello** by sending **add:** with argument **Hello**. Next, the lowercase **e** needs to be replaced with an uppercase letter. This can be accomplished by typing the command **replace:with:** followed with **e,E**. This leads to the following message:

```
replace: ['e'] with: ['E']
```

On the next line, the user sends the message **capitals** to the text; consequently, all letters are being replaced by capital letters. The result behind the screens will be:

```
['hEllo'] capitals
```

In fact, in this case, the users themselves are unknowingly programming in Citrine, in a kind of clumsy way. The resulting program can also be viewed as a rudimentary implementation of a REPL system (Read-Eval-Print Loop).

4.10 Modules

The world of Citrine can be expanded with new system objects, by installing modules . A new system object can be added to the world of Citrine by placing the module file (usually a file with the suffix .so or .dll) into the mods folder. Suppose you would like to convert a given sequence into the JSON protocol; for that, the JSON plugin module can be used. To install do:

```
mods/json/libctrjson.so
```

The exact file suffix differs per system. The previous example is based on a Linux system. However, the file location of an expansion file will always show similar to following:

```
mods / <object name> / libctr <object name> . extension (so/dll/dylib)
```

When you send a message to the object that is made available through the plugin module, it will be automatically loaded by Citrine:

```
>> cards := Sequence new
    ~ ['poker']
    ~ ['belote'].

>> boards := Sequence new
    ~ ['chutes and ladders']
    ~ ['ludo'].

>> games := List new
    cards: cards,
    boards: boards.

>> text := Json jsonify: games.

Out write: text.
```

```
{"boards":["chutes and ladders","ludo"], "cards":
["poker","belote"]}
```

In the previous example, the JSON module has been applied. The list of games is converted into a Text object, which represents the data according to JSON protocol: To this end, the message **jsonify:** is sent with the list as argument. The output is shown in the above example.

As soon as **Json** is being invoked, Citrine will search if there is a module in the mods folder that makes this object available. For this action, Citrine implements a text comparison. Citrine searches for the file **libctrjson**, to locate the object **Json**. In case it involves an object named **Xml**, Citrine will look whether the file **libctrxml** is present or not. This scan is only executed when a message is sent to an object that is not present in the current program. As a rule, it will not affect any performance of the computer program itself. The scan is only being executed in case a program error is likely to occur anyway. As soon as it is identified, it is then loaded and the program will further handle the message.

Should you send a message to a non-existing module, an error message will occur. This is the same error message as a message that is sent to a non-existing object. So, in both cases the error message is the same. Consequently, Citrine will initiate a search for each unknown object to see whether the object may still be loaded as an expansion. If this is not the case, the process fails. The next example tests if an expansion module is indeed present:

```
{ Teapot brew. } except: {  
    Out write: ['No tea for thee!'], stop.  
}, start.
```


4.11 Detection

The previous chapter showed how to find out whether an expansion module is present or not. However, there are various ways for system exploring during program execution. Citrine holds a couple of methods to detect which objects are present and to which messages these objects respond. Firstly, each object can be asked what **type** it is:

```
>> thing := Object.  
Out write: thing type, stop.  
Out write: 12345 type, stop.  
Out write: ['word salad'] type, stop.  
Out write: True type, stop.
```

```
Object  
Number  
Text  
Boolean
```

Contrary to many common programming languages, types are malleable. The standard type of an object is, for example, **object**, however feel free to modify this:

```
>> Person := Object new.  
Person on: ['type'] do: { <- ['human'] }.  
Out write: Person type.
```

In this case the output will be as follows:

```
human
```

Note how this doesn't influence the object **Person**, as solely the answer to the message **type** has been modified, nothing more.

Types can be useful to verify if the message is receiving the correct arguments. For example, they make it possible to check during an addition if the arguments are in fact numbers. However, there

are more pragmatic ways to retrieve the properties of objects. Because Citrine Objects do not error on invalid messages you can just treat objects you expect to be numerical as such and see what happens. If the object you get, happens to be something else, it will remain unaffected by your messages.

The Program object has been introduced in chapter 3.12. With this object and by using the message **use:**, it is possible to include programs of others in your own program. The objects that are made available in external program files, can equally be used in your own program. It is also possible to ask the Program object if a given object is already present in the program:

```
Program Tool true: { ... }.
```

The code on the place of the dotted line will, in this case, be executed if the object **Tool** indeed exists. In case of an object name which consists of one single symbol, it is best to apply the following notation to avoid confusion:

```
Program find: ['X'], true: { ... }.
```

After all, :

```
Program X
```

is not a valid message.

Besides asking about objects, it is equally possible to ask the Program object about messages. For instance, it can be asked if the object **Number** knows the message **between:and:**, in the way as is illustrated below:

```
Out write: ( Program Number: ['between:and:'] ), stop.
```

```
Out write: ( Program Number: ['replace:with:'] ), stop.
```

Result:

```
True
False
```

Mind the notation of the message: the message that has to be recognised by the relevant object needs to be written together and free from arguments. When, for example, you would like to find out if it can be asked to object Easter Bunny to jump:5 meters to:left, the following message could be sent to Program:

```
Program Easter Bunny: ['jump:to:'].
```

In short, the relevant message that must be answered by the **Easter Bunny**, is basically written as **jump:to:.**

The Program object can also provide information regarding the version of Citrine is using at a given moment. As an answer to the message **text**, the Program object shows **The Programming Language Citrine/UK**. As an reply to the message **number**, the version number will show, for example:

```
Out write: Program, stop.
```

```
Out write: Program number, stop.
```

```
The Programming Language Citrine/UK
94
```

The version number is always shown as a round number. Version 0.9.4 becomes 94.

4.12 Exercises

1. See example **Combination** in chapter 4.4. Through which by-pass is element duplication still possible? In what way can this be prevented?

2. This next code only counts numbers with the same qualifier:

```
Number on: ['+'] ...: { :other
    >> ... := self copy.
    self qualifier = other ... ...: {
        ... add: other.
    }.
    <- sum.
}.
```

Out write: 2 apples + 3 pears, stop.

Out write: 2 apples + 3 apples, stop.

The result:

```
2 apples
5 apples
```

Which code should be on the dotted lines? Fill in the blanks.

3. Implement the message **copy:** for File.

4. Create a new object named **Reflection**. This object converts each message it receives into a Text object and flips the characters:

Out write: Reflection pepper.

The output:

```
reppep
```

What does the code or definition of the object **Reflection** look like?

5. A palindrome or mirror word, is a word of which the letters or characters read the same reverse, as they are arranged symmetrically. Extend the Text object with the message **palindrome?**, so that users can determine if the entered word is, in fact, a palindrome. To this end, fill in the missing code on the dotted line below:

```
Text on: ['palindrome?'] do: { ... }.
```

```
Out write: ['pepper'] palindrome?, stop.  
Out write: ['level'] palindrome?, stop.
```

Output:

```
False  
True
```

Clue: use the previously written object **Reflection**.

6. Suppose you have an object **Pair**, which has been derived from **Sequence**, to create data pairs

Example:

```
>> pair := Pair new of: ['pepper'] and: ['salt'].
```

```
Out write: pair first, stop.
```

```
Out write: pair second, stop.
```

a. What would be the output of the previous code?

b. The object **Pair** returns the first element on receiving the message **first**. That code looks like the following:

```
Pair on: ['first'] do: {
    <- own elements first.
}.

```

Now, write the accurate code to return the second element.

c. The elements of a pair can be set by sending the message **of: and:**; for example, pair of: **pepper** and: **salt**. Write the implementation of **of: and:**, by filling in the missing code on the dotted lines:

```
Pair on: ['of:and:'] do: {
    :first :second
    own ... ..: ..., ..: ...
}.

```

d. Add the message **blank** to start with an empty pair:

```
Pair on: ['blank'] do: {
    ...
}.

```

e. Whilst creating a new pair, it is necessary to first set the elements (containing blank values), in order to avoid errors. Create the code for the message **new**.

```
Pair on: ['new'] do: {
    ...
}.

```

f. Create a message that allows for a textual representation of a data pair:

```
Out write: pair, stop.

```

This shows the following output:

```
pepper and salt

```

7. There is an error in the following program. The output is 2, however the correct output should be 3. Find the mistake.

```
>> points := Sequence <- 1 ; 2 ; 3 ; 1 ; 2.  
points shift shift pop pop.  
>> middle := points first.  
Out write: middle, stop.
```

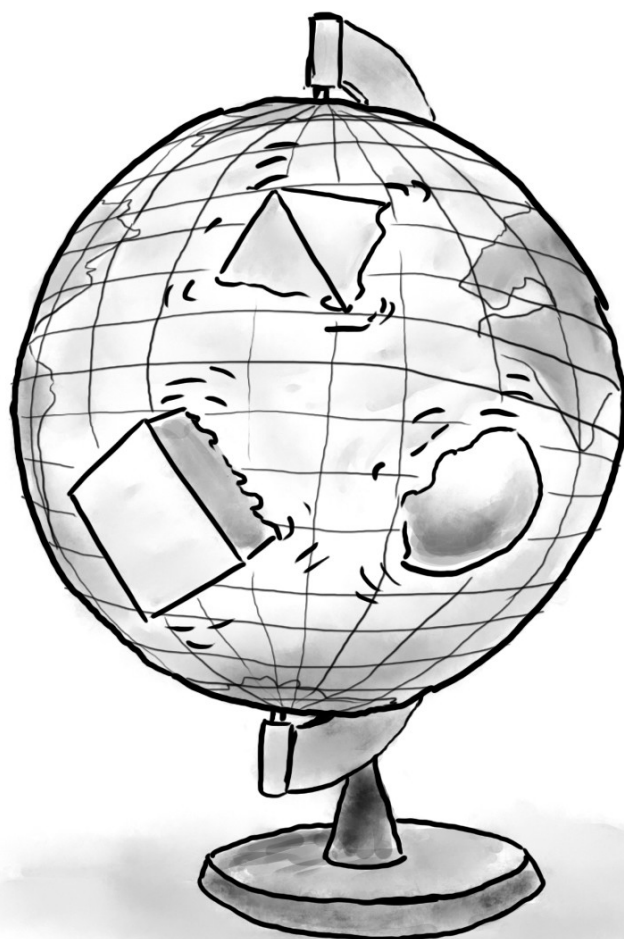
8. How many aces are there at the end of this program?

```
>> aces := 2.  
>> Cards := Object new.  
Cards on: ['cheat'] do: { aces add: 1. }.  
{ >> aces := 4. Cards cheat. } start.  
Cards cheat.
```

9. Suppose that the plugin **libctraquarium.so** is being installed. To which object can a message be sent in order to load this module?

10. What is the output of this program on screen?

```
>> a := 0.  
>> b := a = ['zero'].  
>> c := 2.  
>> d := c = ['two'].  
>> e := ['zero'].  
>> f := e = 0.  
Out write: b, stop.  
Out write: d, stop.  
Out write: f, stop.
```

5. Translate

5.1 Translation System

It is possible to translate programs from one language into another. Citrine knows two methods for translating program code: the translation system and the dynamic translation techniques (more on this later). Using the translation system with its simple translation rules, program code can be converted from one human language into another. The dynamic translation techniques are tools which can be implemented to fine-tune the results of these translations. Moreover, Citrine code will benefit from utilising the previously mentioned methods as they lead to a more natural appearance of the resulting code. Covered in this chapter is the general translation system. You can use this system to translate a program file from one language into another. To this end, it is necessary to have a dictionary file. At a later moment it will be explained how to create dictionary files, but for now it can be assumed that a dictionary file is already present. Suppose you would like to translate a file written in English into Italian, and the necessary dictionary file **enit.dict** (which means from EN into IT) is already available, the program file can be translated as follows:

```
ctren -t enit.dict program1.ctr > program2.ctr
```

The translated program will be saved as **program2.ctr**. Note that the exact notation for this command may vary depending on which operating system is used. The previously mentioned example is meant for an execution on Linux-like systems. After implementing the previous example, the result will show the Italian translation of the program. The translation system will notify in case it does not recognise certain words, and consequently these words will not be translated.

A dictionary file consists of two columns; on the left, the word to be translated, and on the right its translation. Suppose you would like to translate the following program into English:

```
Out scrivi: ['Ciao mondo!'], stop.
```

Here, it is necessary to have a dictionary file containing translations for the words **scrivi**, **stop** and **Ciao mondo!**. A dictionary can contain 4 groups of words: token words (**t**), texts (**s**), decimal symbols (**d**) and number separator symbols (**x**). Each individual group is assigned its own character, which has been put between brackets for this purpose. This character represents the type of word that needs to be translated, situated in the left column. The following file could translate the program:

```
t "scrivi:" "write:"  
t "stop" "stop"
```

```
s "Ciao mondo!" "Hello World!"
```

When this file is saved as **iten.dict**, the program can be translated as follows:

```
ctren -t iten.dict program.ctr
```

The resulting output would be:

```
Out write: ['Hello World!'], stop.
```

Whilst translating keyword messages, such as: **Number between: X and Y**, the message segments are combined which results in the following translation line:

```
t "tra:e:" "between:and:"
```

The following errors can occur when using a dictionary file:

Translation error, message too long.	Your translation exceeds 255 bytes, this is not allowed.
Ambiguous word found	The dictionary file has the same word twice.
Ambiguous translation	The dictionary has the same translation twice.
Message type does not match Translation	For example, of a unary message into a keyword message or binary message and vice versa.
General error	A different error has occurred (rare).

To make work easier for you, Citrine also offers a dictionary generator to create a dictionary for the core vocabulary, which then can be completed freely. A dictionary can be generated in case two different source code versions of Citrine are present. So, for example, to create a English to Italian dictionary file for a Citrine program, the following notation can be used:

```
ctren -g citrineen/i18n/en/dictionary.h  
citrieneit/i18n/it/dictionary.h > enit.dict
```

Note that we use the option **-g** here (means: generate).

5.2 Dynamic translations

Using renaming objects, using aliases and constructing syntaxes are all examples of dynamic translation methods. These are called so, as they translate program code by executing actions during the running time of the program. These methods can also be leveraged to enhance readability of the program code, by giving it a more natural appearance.

The most obvious and simplest tool at your disposal is renaming objects.

```
>> Empty := Sequence new.
```

At certain times, it will be necessary to send along an empty sequence with a task, or as argument with a message to an object. Rather than having to write repeatedly **Sequence new**, it is possible to simply use the word **Empty** after running the abovementioned code. For instance as illustrated next:

```
computer message: ['boot'] arguments: Empty.
```

Another tool that can be used is the message **learn:means:.** This teaches an object to perceive a message as a different message. For example:

```
Out learn: ['scribble:'] means: ['write:'].
```

```
Out scribble: ['Brevity is the soul of wit'], stop.
```

Result:

```
Brevity is the soul of wit
```

Do note that the synonym can only be used on the object to which the learn message is sent.

Appendices

Appendix A: AST-export

Using the AST export function, it is possible to export a Citrine program as a tree-like structure. The output can be transpiled into a different programming languages, for example C. It is also possible to use the output to show a visual representation of the program in question.

To export a Citrine program, use the **-x** option, such as:

```
ctrXX -x program.ctr
```

Suppose the program file “program.ctr” has the following content:

```
Out write: ['Hello'].  
Out write: ['World'].
```

The output of the export function will look like this:

```
52;0;0;;[57;0;3;Out;;55;0;8;schrjf;;[56;0;5;Hallo;;];];52;0;0;;  
[57;0;3;Out;;55;0;8;schrjf;;[56;0;6;Wereld;;];];79;0;0;;;
```

What is shown in the above is a sequence of elements, separated by a semicolon. Each element contains 5 fields, again separated by a semicolon. The first field indicates what sort of program element it involves. Elements may also contain other elements, which are listed in the last field (the fifth field) and are enclosed by square brackets.

The following kinds of elements can be distinguished:

Element code	Meaning
51	Declaration <code>>>own :=</code>
52	A message (general), details in subelements
53	An unary message (without arguments)
54	A binary message (with one argument)
55	Keyword message (with one or multiple arguments)
56	Text fragment between quotation marks, i.e. a Text Object
57	Reference/Name (of a variable)
58	Number
59	A task between curly brackets <code>{ ... }</code>
60	Response arrow <code><-</code>
76	A parameter block as part of a task definition
77	Instructions that are part of the task (59)
78	1 parameter that is part of the parameters (76)
79	End of program
80	Expression enclosed in parentheses <code>(...)</code>

If the first field is equal to 57 (reference), the second field indicates a reference to an already declared variable (0), to the properties of an object (1), or to a declaration of an entirely new variable (2). If the first field is equal to 53 through 58 or 78 (for instance, in case of a text element), the content of the element will be printed on the location of the fourth element, the buffer. The third element indicates the length of this buffer in bytes. In the final field, the enclosed elements are printed between square brackets. As the length of the buffer is given, there is no need to determine any escaping sequences.

The result, when inspecting the code from the above example:

- 52 a message

Next, a further specification follows:

- 57 a reference to an object, 3 bytes (as it is a UTF-8 symbol) Out.

- 55 keyword message, 6 bytes “write:”

Then follows a further series of subelements for the arguments, in this case just the one:

- 56 a text, 5 bytes “Hello”

And so on ...

Note that the program ends with code 79 “end program”. This code marks the end of a program.

Appendix B: Answers

Answers chapter 1

1. Hello Britons

2. Hello Britons

3.

3a. Four parts, which are: introduction, explanation, exercises and answers.

3b. Chapter

3c. Subject

4. Q

5.

5a. An audio file

5b. Music collection

5c. 3

5d. BWV595

6. 20.

The operators in Citrine are applied from left to right.

Find out more in chapter 2.

7.

```
>> declare
```

```
self send message to self
```

```
own property
```

```
← start sequence
```

```
<- return object
```

Answers chapter 2

1. -x

2. 16.5

3.

```
>> x := 2.  
{ x add: (x power: 2). } * 2.
```

4.

```
>> x := 2 to the power: 3, + 2.
```

5. even?

6. while:

7.

tea

```
>> x := 7.  
(x > 7) true: { Out write: ['p']. }.  
(x < 7) true: { Out write: ['o']. }.  
(x ≥ 7) true: { Out write: ['t']. }.  
(x ≤ 7) true: { Out write: ['e']. }.  
(x ≠ 7) true: { Out write: ['n']. }.  
(x = 7) false: { Out write: ['t']. }.  
(7 > x) true: { Out write: ['i']. }.  
(7 < x) false: { Out write: ['a']. }.  
(7 ≥ x) false: { Out write: ['l']. }.  
(7 ≤ x) false: { Out write: ['l']. }.  
(7 ≠ x) true: { Out write: ['y']. }.
```

8.

a= 8

b= 1 +2

c= 4

d= 10.5

e returns an error, division by 0

9.

```
>> Heatwave? := {  
    :sunny-days :tropical-days  
    >> answer := False.  
    (sunny-days ≥ 5) true: {  
        (tropical-days ≥ 3) true: {  
            answer := True.  
        }.  
    }.  
    <- answer.  
}.  
Out write: (Heatwave? apply: 6 and: 3), stop.
```

The reason why this program did not work was because the answer was not a part of the task **Heatwave**, but was part of a subtask instead. - A task can be ended on one single point only. Other programming languages offer the possibility to end a task equivalent on various points, which causes confusion.

Answers chapter 3

1.

```
>> celcius := -10.
{
  >> fahrenheit := celcius * 1.8 + 32.
  Out write: fahrenheit, stop.
}
while: { <- (celcius add: 1) ≤ 40. }.
```

2.

```
>> tile := ['buy while the iron is cheap'].
tile buy: ['strike'], cheap: ['hot'].
Out write: tile, stop.
```

3.

```
Number on: ['square'] do: { <- self to the power: 2. }.
Out write: 3 square, stop.
```

4.

```
>> Point := Object new.
Point on: ['x-coordinate:'] do: { :x own x := x. }.
Point on: ['y-coordinate:'] do: { :y own y := y. }.
Point on: ['x-coordinate'] do: { <- own x. }.
Point on: ['y-coordinate'] do: { <- own y. }.
>> pier := Point new x-coordinate: 5, y-coordinate: 6.
>> town-hall := Point new x-coordinate: 7, y-coordinate: 1.

>> Distance := Object new.
Distance on: ['start:end:'] do: { :start :end
  own start := start.
  own end := end.
}.
Distance on: ['number'] do: {
  <-
  (own start x-coordinate - own end x-coordinate) absolute
  + (own start y-coordinate - own end y-coordinate) absolute.
}.
```

```
>> x := Distance new
      start: pier
      end: town-hall,
      number.
```

```
Out write: x, stop.
```

5.10

6.

```
Number on: ['double'] do: { <- self * 2. }.
```

```
Out write: 4 double, stop.
```

7.

```
Number on: ['digits:'] do: { :n
```

```
    >> digits := self text.
```

```
    >> characters := Sequence new.
```

```
    >> diff := (n - digits length).
```

```
    characters fill: diff with: ['0'].
```

```
    >> text := characters combine: [''].
```

```
    text add: digits.
```

```
    <- text.
```

```
  }.
```

```
Out write: (9 digits: 9), stop.
```

Answers chapter 4

1.

For example:

```
>> Combination := Sequence new.  
Combination on: ['append:'] do: { :element  
    >> seen := self find: element.  
    seen false: { self append: element. }.  
    }.  
    }.
```

```
>> colours := Combination new.
```

```
colours  
append: ['red'],  
append: ['green'],  
append: ['blue'],  
prepend: ['red'].
```

```
Out write: colours, stop.
```

-This can be avoided by also modifying the other messages which modify the collection.

2.

```
Number on: ['+'] do: { :other  
    >> sum := self copy.  
    self qualifier = other qualifier true: {  
        sum add: other.  
    }.  
    <- sum.  
    }.
```

```
Out write: 2 apples + 3 pears, stop.
```

Out write: 2 apples + 3 apples, stop.

```
2 apples
5 apples
```

3. Full test case:

```
File on: ['copy:'] do: { :source self write: source read. }.
```

```
>> file1 := File new: Path /tmp file1.
```

```
>> file2 := File new: Path /tmp file2.
```

```
{
```

```
file1 delete.
```

```
file2 delete.
```

```
} except: { :err }, start.
```

```
file1 write: ['abc'].
```

```
file2 copy: file1.
```

```
Out write: file2 read.
```

4.

```
>> Reflection := Object new.
```

```
Reflection on: ['respond:'] do: { :text
```

```
    >> reversed := Sequence new.
```

```
    text characters each: { :number :character
```

```
        reversed prepend: character.
```

```
    }.
```

```
    <- reversed combine: [''].
```

```
    }.
```

```
Out write: Reflection pepper.
```

5.

```
Text on: ['palindrome?'] do: {  
    <- (Reflection message: self arguments: Sequence new) = self.  
}.
```

6.

a.

```
pepper  
salt
```

b.

```
>> Pair := Object new.  
  
Pair on: ['first'] do: {  
    <- own elements first.  
}.  
  
Pair on: ['second'] do: {  
    <- own elements last.  
}.
```

c.

```
Pair on: ['of:and:'] do: {  
    :first :second  
  
    own elements  
        append: first,  
        append: second.  
}.
```

d.

```
Pair on: ['blank'] do: {  
    own elements := Sequence new.  
}.
```

e.

```
Pair on: ['new'] do: {
```

```
>> pair := self new.  
<- pair blank.  
}.
```

f.

```
Pair on: ['text'] do: {  
    <- ['first and second']  
        first: own elements first,  
        second: own elements last.  
}.
```

7.

```
>> points := Sequence <- 1 ; 2 ; 3 ; 1 ; 2.  
points do shift shift pop pop done.  
>> middle := points first.  
Out write: middle, stop.
```

8.

3 aces

9.

Aquarium.

10.

True False False

6. full code

```
>> Pair := Object new.  
  
Pair on: ['first'] do: {  
    <- own elements first.  
}.  
  
Pair on: ['second'] do: {
```

```

        <- own elements last.
    }.

Pair on: ['of:and:'] do: {
    :first :second

    own elements
        append: first,
        append: second.
}.

Pair on: ['blank'] do: {
    own elements := Sequence new.
}.

Pair on: ['new'] do: {
    >> pair := self new.
    <- pair blank.
}.

Pair on: ['text'] do: {
    <- ['first and second']
        first: own elements first,
        second: own elements last.
}.

>> pair := Pair new of: ['pepper'] and: ['salt'].
Out write: pair first, stop.
Out write: pair second, stop.
Out write: pair, stop.

```


Appendix C: Memory Management

This chapter may be of interest solely to those readers who would like to manually configure their own version of Citrine. Readers who are merely interested in the language Citrine, can simply ignore this chapter. It is assumed that readers have some basic know-how of system management. Citrine supports various parameters for system management. In absence of parameters, Citrine will start by default with a memory limit of 10MB. The following environment variables can be utilised to modify memory settings in Citrine:

management:

CITRINE_MEMORY_LIMIT_MB

CITRINE_MEMORY_MODE

CITRINE_MEMORY_POOL_SHARE

To specify the memory limit in MBs use CITRINE_MEMORY_LIMIT_MB. Use CITRINE_MEMORY_MODE to set the mode of memory management (1 by default).

Value	Meaning
0	Memory is not cleaned up. This setting ensures that the memory is not cleaned up during program run. This will benefit speed, however memory will fill up fast.
1	Memory is being cleaned up, as the determined limit value comes into view. This default setting will be sufficient for most programs.
4	Memory is being cleaned up at each step in the program. This will induce a low use of memory, but it is very intensive. Program running speed will be effected profoundly.
8	Experimental mode involving memory pools.

It is certainly possible to also combine values ($12=8+4$). During the running of the program, memory parameters can be modified, as pointed out in chapter 3.12. A sharedmemory block (8) can only be created one time during a program run. It is not possible to constantly switch this setting during the running time of the program.

Register

Register

add:.....	93	Hello world.....	13
and:.....	32	hour.....	88
answer.....	26	hour:.....	89
append:.....	68, 83	ifs.....	22
apply:and:.....	26	keyword messages.....	19
argument:.....	97	last.....	70
arguments.....	97	last:.....	54
ask.....	98	learn:means:.....	158
ask password.....	102	length.....	53
between:and:.....	47	Lists.....	76
binary messages.....	20	loops.....	22
Boolean Algebra.....	42	lowercase.....	55
by:.....	78	lozenge.....	25
capitals.....	55	mathematical operations.....	47
case:do:.....	66	maximum.....	73
character:.....	53	memory.....	103
characters.....	69	memory management:.....	104
clean up.....	104	message:arguments:.....	140
code.....	134	minimum.....	73
combine:.....	70	minute.....	88
Command.....	96	minute:.....	89
compare:.....	52	modules.....	143
contains:.....	53, 77	Moment.....	87
continue.....	44	month.....	88
copy.....	110	month:.....	89
count:.....	69	new.....	28
day.....	88	None.....	40
day-of-the-year.....	88	nor:.....	43
day:.....	89	Number.....	19, 47
delete.....	84	object.....	136
do.....	138	Object.....	19, 28, 64
done.....	138	on:do:.....	26, 66
each:.....	73, 78	or:.....	43
either:or:.....	44	os:.....	95
else:.....	42	output object.....	40
end.....	103	parameters.....	23
entries.....	79	path.....	84
equal:.....	66	Path.....	85
error:.....	60, 97	penultimate.....	70
except:.....	59	plain.....	48, 87
exists.....	84	pop.....	72
False.....	42	position:.....	70
false:.....	22, 42	prepend:.....	71
File.....	83	procedure.....	60
fill:with:.....	71	Program.....	95
find:.....	54, 72	Program input.....	97
first.....	70	property.....	28
flush.....	97	put:at:.....	71, 76
from:length:.....	54	qualifier.....	49
has:.....	78	qualifier:.....	49

quotation marks.....	51	subtract:.....	93
read.....	83	Task.....	19, 58
recursive.....	124	template.....	24
replace:length:with:.....	74	Text.....	19, 51
replace:with:.....	52	this-task.....	60
respond:.....	118	time.....	91
respond:and:.....	119	trim.....	55
respond:and:and:.....	119	True.....	42
respond:and:and:and:.....	119	true:.....	22
second.....	88	type.....	145
second:.....	89	unary messages.....	19
self.....	26	use:.....	95
Sequences.....	68	values.....	79
set:value:.....	62	variable.....	18
setting:.....	98	wait:.....	93
setting:value:.....	98	week.....	88
shift.....	72	weekday.....	87
skip:.....	55	while:.....	23, 58
sort:.....	74	write:.....	40 , 83
split:.....	69	year.....	87
start.....	60	year:.....	89
stop.....	40	zone:.....	90
string interpolation.....	24		

