Programming with Objects & Abstractions in

CITRINE

Gabor de Mooij

Copyright© 2023 Gabor de Mooij

Autor: Gabor de Mooij

Traductor: Jose Alejandro Usero

Ilustraciones: Robert Cabri

Ilustración de portada: Robert Cabri

1st Edición 2023, Digital PDF

2nd Edición 2024, Digital PDF

Historial de Versiones		
2023	1.0	Versión Inicial
2024-12-29	2.0	ASCIIificación, fuentes corregidas, fragmentos de código actualizados, Se eliminaron algunos capítulos que ya no son relevantes.

Indice

1. Introducción 10
   1.1 Qué es Citrine ? 11
   1.2 Hola Mundo 12
2. Reglas de escritura 16
   2.1 Acciones 17
   2.2 Variables 18
   2.3 Objetos y Mensajes 19
   2.4 El Flujo del Programa 22
   2.5 Plantillas 24
   2.6 Respondiendo 26
   2.7 Herencia y propiedades 28
   2.8 Ejercicios 34
3. Objetos 39
   3.1 Objetos simples 40
   3.2 Objeto Booleano 42
   3.3 Objeto Número 47
   3.4 Objeto Texto 51
   3.5 Conversiones 57
   3.6 Objetos Tarea 58
   3.7 Objeto Raiz 64
   3.8 Series 68
   3.9 Listas 76
   3.10 Archivos 83
   3.11 Momentos 87
   3.12 Objeto Programa 95
   3.13 Ejercicios 105
4. Avanzado 109
   4.1 Copiando 110
   4.2 Visibilidad 115
   4.3 Manejando Mensajes desconocidos 118
   4.4 Herencia, anulación y recursión 121
   4.5 Estado inicial 125
   4.6 Conversión implicita 128
   4.7 Serialización 134
   4.8 Estructura alternativa del mensaje 138
   4.9 Mensajes enviados programáticamente 140
   4.10 Modulos 143
   4.11 Detección 145
   4.12 Ejercicios 148
5. Trasladar 155
   5.1 Sistema de traducción 156
   5.2 Traducciones dinámicas 158
Apendice A: AST-exportación 163
Apendice B: Respuestas 166
   Respuestas capítulo 1 167
   Respuestas capítulo 2 168
   Respuestas capítulo 3 170
   Respuestas capítulo 4 172
Apendice C: Gestión de la memoria 177
Registro 178

1. Introducción

1.1 ¿Qué es Citrine?

La ambición del proyecto Citrine es desarrollar un lenguaje de programación simple que mejore la mantenibilidad, precisión y legibilidad del software, y que minimice la cantidad de errores de software. Para lograr esto, el proyecto tiene tres principios básicos: simplicidad, localización y uniformidad. La simplicidad significa que el lenguaje de programación consiste en una selección notablemente pequeña de reglas gramaticales y conceptos. De hecho, Citrine evita los tipos de datos, las clases y la interpolación de cadenas y solo se necesitan tres acciones esenciales para que el programa se ejecute. La localización significa que el
lenguaje de programación puede escribirse en cualquier idioma nativo. Citrine/NL, por ejemplo, se usa para el idioma holandés.

La oportunidad de escribir software en cualquier idioma nativo puede considerarse como el objetivo final en el intento de mejorar la legibilidad del código. Además, los clientes y las partes interesadas podrán verificar y revisar el trabajo del programador. Uniformidad significa que Citrine, a diferencia de los lenguajes de programación actuales, aspira a prohibir los sublenguajes como los lenguajes de marcado (HTML), las expresiones regulares, los formatos de rutas de archivos, los dialectos de bases de datos (SQL) y los protocolos de red como JSON, XML o YAML.

Citrine se basa en la premisa de que cuando todo está escrito en el mismo lenguaje localizado, no es esencial tener conocimientos de otros lenguajes de código. Estos tres principios básicos se fortalecen entre sí; por ejemplo, el minimalismo gramatical permite una mayor libertad de notación, lo que a su vez favorece la localización. Viceversa, el principio de localización impulsa la uniformidad, ya que todo el código está escrito en el mismo lenguaje. Tanto la uniformidad como la localización reducen la tensión cognitiva que se requiere al leer programas informáticos, lo que, a su vez, promueve el proceso de simplificación.

Citrine puede ser utilizado por aquellos programadores que buscan un lenguaje de programación simple que les permita escribir código en su propio idioma y que reconocen los beneficios mencionados anteriormente.
Además, Citrine se puede utilizar como lenguaje específico de dominio (DSL) para ampliar una interfaz gráfica de usuario, lo que permite al usuario avanzado automatizar tareas. Además, los tutores pueden utilizar Citrine para enseñar los principios básicos de cómo escribir código, sin que el lenguaje sea un problema.

En particular, este manual está destinado a desarrolladores avanzados y aclara las reglas de codificación para el lenguaje de programación Citrine. El manual se centra específicamente en la versión en inglés de Citrine, es decir, Citrine/EN. En los siguientes capítulos, el término Citrine/EN se reemplazará por Citrine

Traduzca sus archivos de programa utilizando las opciones -g y -t. Consulte el capítulo 5 para obtener más información. Exporte la estructura de un programa Citrine utilizando la opción -x. Esto se puede utilizar para convertir programas Citrine a otros lenguajes de programación como C y Java. Para obtener más información, consulte el Apéndice A.

Citrine es un software de código abierto y se publica bajo la licencia BSD2. Esta es una de las licencias más simplificadas y concisas, lo que significa que puede utilizar Citrine tanto para fines personales como comerciales, siempre que se especifique el acuerdo de licencia y los autores. No hay más restricciones. Para una descripción general completa de la licencia BSD2, visite:

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

y

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

La tradición dicta que el primer programa de cualquier lenguaje de programación nuevo debe ser una especie de saludo al mundo. En Citrine, un programa **Hola mundo** se ve así:

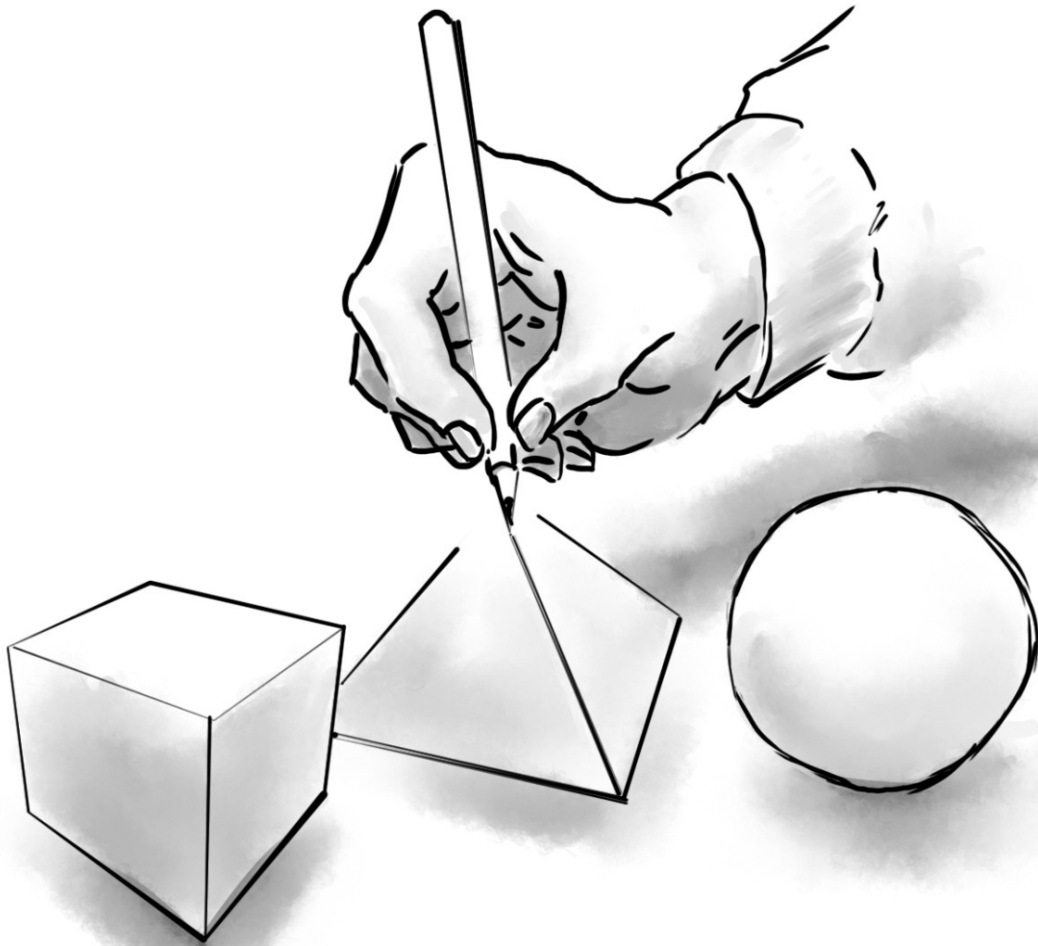
```
Salida escribir: ['Hola mundo'] , detener .
```

Cuando este texto se guarda en un archivo, p. ej. hello.ctr, se puede ejecutar de la siguiente manera: .

```
ctres hello.ctr

Hello world
```

Puede ver el resultado del programa en la ventana negra que se muestra arriba. A lo largo de este manual, se aplicará la misma visualización para mostrar el resultado. El código de programación de los programas Citrine se imprime en la fuente Citrine, igual que el ejemplo que se muestra arriba.



2. Reglas para escribir

2.2 Variables

El primer paso es asignar un objeto a una variable. Para asignar una variable, se utiliza el símbolo de declaración de variable: `>>` .El nombre de una variable puede contener todos los signos excepto: <-, := , espacios en blanco , puntos, comas, dos puntos, comillas ["] y paréntesis (). Además, una variable no puede constar de varias líneas. Tenga en cuenta que una variable no puede comenzar con un número o un signo menos.

Estos son ejemplos de variables válidas:

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

Las variables no válidas son, por ejemplo:

```
>> -123 := ['número negativo'].
>> contraseña del usuario := ['Clasificado'].
>> contraseña.del:usuario := ['Clasificado'].
>> ,x := 10.
```

Una vez declarada una variable, se puede utilizar libremente. Esto significa que solo es necesario utilizar el símbolo de declaración la primera vez que se utiliza la variable (la declaración).

Para ser más específicos, no se puede comenzar a declarar aleatoriamente un valor para la variable (`x := 2`), ya que primero se debe declarar la variable (`>> x := 2`). Sin embargo, una vez declarada se puede cambiar su valor sin utilizar el símbolo de declaración (`>> x := 2. x := 3.`).

Tenga en cuenta que es obligatorio asignar un valor a una variable en el momento de la declaración. A diferencia de otros lenguajes de programación, no se permite declarar una variable sin valor. Es esencial vincular explícitamente cada variable a un valor inicial. No obstante, siéntase libre de inicializar la variable con el objeto Nulo. En este caso, la variable puede considerarse vacía (`>> x := Nulo` .). En resumen, no está permitido declarar una variable sin un valor inicial. Por lo tanto,

```
>> x
```

no es válido y mostrará un mensaje de error. (Obtenga más información sobre el objeto Nulo en el capítulo 3).

2.1 Acciones

Citrine es un lenguaje de programación *puramente orientado a objetos* . Esto significa que Citrine percibe todo como un objeto, por lo tanto, *no hay otros tipos de datos.*
Básicamente, hay tres acciones rutinarias básicas en un programa escrito en Citrine: *asignar, enviar mensajes y responder*.
El intercambio de mensajes entre objetos es la parte clave de un programa Citrine. Las tres acciones rutinarias individuales se ilustran a continuación:

Acción	**Ejemplo**	
1. Asignar	>> x := 1.	
2. Mensajes	Unitario	x ¿impar? .
	Binario	1 + 2.
	Palabra clave	x desde: 0 longitud: 10 .
3. Respuesta	<- respuesta .	

Este capítulo le dará una idea general del lenguaje Citrine y describirá sus principios básicos. Citrine es un lenguaje pequeño y se basa en tres acciones (ver ilustración) que se traducen en aproximadamente seis reglas gramaticales. Eso es todo lo que se necesita para dominar el lenguaje.

No se preocupe si las cosas no le resultan obvias de inmediato: todos los elementos básicos se explorarán en el próximo capítulo (3) y se repetirán las reglas básicas. El capítulo actual, sin embargo, sirve principalmente como una introducción general al lenguaje. Los principios básicos se explicarán brevemente e ilustrarán con varios ejemplos relevantes. Cualquier detalle que falte se describirá en un momento posterior.

Los comentarios en Citrine tienen como prefijo el símbolo #. Por lo tanto, se ignorará la siguiente línea:

```
# esto es solo un comentario
```

2.3 Objetos y mensajes

Citrine percibe todo como un objeto; es decir, todos los números, textos y fragmentos de código. Los números, como 1, 2, 100, -999 y 1234 son objetos de número. Todos los textos entre comillas simples son objetos de texto. Todos los fragmentos de código agrupados entre llaves {…} son objetos de tarea.

Nombre	**Objeto Raiz**	**Ejemplo**
numeros	Número	1,2,3…. -100 1,5 1.000.000
textos	Texto	['Brevity is the soul of wit']
tareas	Tarea	{ 1 + 2. }.

Los objetos como números y textos, siempre encuentran su origen en un objeto raíz. Por ejemplo, todos los números derivan del objeto **Número**. Todos los textos derivan de **Texto** y todos los fragmentos de código , del objeto **Tarea**. A su vez, todos estos objetos provienen de **Objeto**, que es, de hecho, el objeto raíz de todos los objetos.
Programar en Citrine significa básicamente enviar mensajes a objetos. La notación general para enviar un mensaje a un objeto es la siguiente:

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

Para saber si el número 2, por ejemplo, es un número par, se envía el mensaje **¿par?** al objeto **2** .

```
2 ¿par?
```

La respuesta será **Verdadero** (de nuevo, un objeto). Los objetos suelen ignorar los mensajes desconocidos; por lo tanto, no se producirá ningún error. Algunos objetos (por ejemplo, números o textos) responden a un mensaje desconocido de una manera predefinida (más sobre este tema más adelante).

Un objeto puede recibir tres tipos de mensajes. En primer lugar, están mensajes unitarios, véase el ejemplo anterior, que no tienen argumentos. En segundo lugar, hay mensajes de palabras clave, que tienen uno o varios argumentos, por ejemplo:

```
>> x := Número entre: 1 y: 10 .
```

En este caso, el mensaje **entre: y:** se envía a **Número**, que es el objeto raíz de todos los números.El resultado será un número aleatorio entre 1 y 10. Por último, hay mensajes binarios, que tienen solo un carácter y un argumento:

```
2 + 3
```

Esto parece una suma matemática, pero en realidad es solo otro mensaje. El mensaje + se envía a 2, con argumento 3 que devolverá la respuesta 5. Se permite escribir mensajes binarios sin dos puntos.

Los mensajes binarios se pueden encadenar:

```
>> x := 3 + 2 - 1.
```

En este fragmento, **+ 2** se envía primero al objeto **Número 3**, lo que da como resultado el objeto **Número 5**, después de lo cual se envía **-1** a este número. Los lectores observadores tienen razón al notar la discrepancia que muestra este protocolo con respecto a las secuencias matemáticas convencionales de operadores. Citrine ignora la secuencia matemática a favor de la coherencia en su sistema de mensajes. Como resultado, la suma:

```
2 + 3 * 5 = 25
```

no 17.

Esto es intencional. Se pueden usar paréntesis para modificar el orden de la secuencia:

```
2 + (3 * 5) = 17
```

La mayoría de los objetos se devuelven a sí mismos como resultado en respuesta a un determinado mensaje. Esto es útil ya que estimula un diálogo adicional con este objeto al enviarle un mensaje de seguimiento:

```
Salida escribir:
     [' hello '] recortar mayúsculas .
```

```
HELLO
```

Aquí, se envían dos mensajes al objeto **Texto** : **recortar** , seguido de **mayúsculas**.

En el siguiente fragmento, es necesario el uso de una coma para indicar que se está entrando un nuevo mensaje. De lo contrario, Citrine se confundirá.

```
Salida escribir: ['Hola!'], detener .
```

Primero, se envía el mensaje **escribir:** al objeto Salida , seguido del mensaje **detener** . Sin la coma, Citrine pensaría que desea enviar el mensaje **detener** al objeto de **Texto Hola** , lo cual resultaría un resultado incorrecto.

La secuencia de proceso de mensajes es la siguiente: de izquierda a derecha; comience con los **mensajes entre paréntesis** seguidos de los **mensajes unitarios**. A continuación, **los mensajes binarios** y luego termine con los mensajes de **palabras clave** . Vea el siguiente ejemplo:

```
Salida escribir: 0.5 redondear + (2 - 1), detener .
```

Citrine siempre lee de izquierda a derecha: primero, se envía el mensaje `escribir:` a `Salida` y es seguido por el mensaje `detener` .Dentro del argumento en sí,(el **argumento** de un mensaje es aquello que va despues de los 2 puntos :) Citrine lee de izquierda a derecha, por lo que **0,5 redondear** y luego el
+.
Además, **redondear** tiene prioridad sobre **+**, porque es un **mensaje unitario**.
Como hay paréntesis, primero se calcula **2 – 1**, después de lo cual se suma **1** al resultado de **0,5 redondear** (1).

Todo el proceso dará como resultado **2**, que también es la respuesta a escribir. Después de la coma sigue el mensaje de parada, que mueve el cursor a una nueva línea. Por supuesto, la mejor manera de comprender completamente el orden de secuencia de los programas de Citrine es con la práctica.

Es importante señalar que en Citrine, al contrario de muchos otros lenguajes de programación, los espacios en blanco son una parte fundamental de su sintaxis. En particular, cuando se trata de mensajes binarios, los espacios en blanco pueden causar cierta confusión.

Utilice siempre un espacio en blanco después de un mensaje binario.
No puede adjuntar directamente un número al mensaje, por ejemplo: **3 + 2** es diferente de **3 +2**. El primer ejemplo (**3 + 2**), envía el mensaje **+** al número **3**, con el argumento el número **2**. El resultado, en este caso, será **5**. En el segundo ejemplo (**3 +2**), el mensaje unario **+2** se envía al número **3**. Dependiendo del contexto eso podría producir un resultado muy diferente.

2.4 El flujo del programa

En Citrine no hay necesidad de reglas gramaticales separadas para bucles y condicionales. Una declaración condicional es simplemente un simple mensaje **verdadero**: o **falso:** a un objeto **Verdadero** o a un objeto **Falso** .

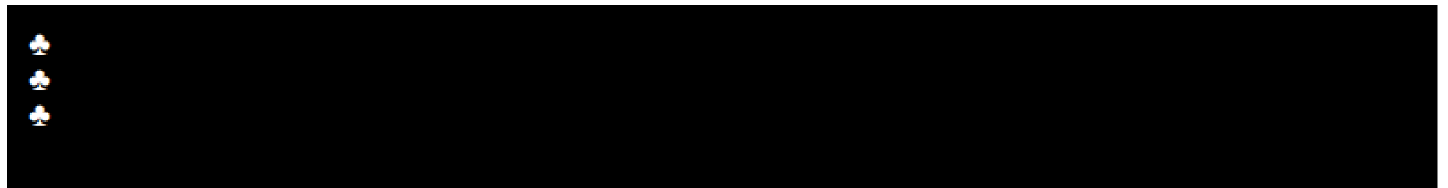
```
2 ¿par? verdadero: {
Salida escribir: ['Dos es un número par'] , detener.
}.
```

Los objetos **Verdadero** y **Falso** también se conocen como objetos de toma de decisiones. Esta denominación a veces es más precisa, porque especifica que estos objetos se utilizan para hacer que el programa maneje ciertas decisiones. Al igual que otros lenguajes de programación, un programa escrito en Citrine sigue un curso específico y conlleva varios patrones de ramificación. Las formas seleccionadas dentro de un programa dependen en gran medida de estos objetos de toma de decisiones.

En el ejemplo anterior, el mensaje **verdadero:** se envía a **Verdadero** (la respuesta a la pregunta: ¿es 2 un número par?). Como argumento, se envía un fragmento adicional de código: un objeto **Tarea.** Esta tarea escribe en la pantalla que 2 es, de hecho, un número par.

Tampoco es necesario que existan reglas de escritura independientes para un bucle. Para ejecutar un fragmento de código tres veces, simplemente envíe * con el argumento **3** a esa **tarea:**

```
{ Salida escribir: ['♣'], detener. } * 3.
```



Consideremos otro ejemplo. Cuando se necesita imprimir una tabla de conversión de kJ (kilojulio) a kcal (kilocaloría) en pasos de 100, consulte a continuación la notación correcta:

```
{ :linea
   >> kJ := linea * 100.
   Salida escribir: kJ, detener.
   Salida escribir: kJ * 0.239, detener.
} * 10.
```

Este programa generará la siguiente lista:

```
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
```

Aquí, el número de la línea actual se transmite al parámetro **:linea**. Al comienzo de una tarea, se definen los parámetros de la tarea. Los parámetros no utilizados permanecen vacíos (**Nulo**). Los parámetros se colocan siempre al comienzo de la tarea, directamente después del corchete inicial y están precedidos por dos puntos.

El mensaje **mientras:** es una combinación de un *bucle* y una *condición*. Se pueden vincular dos tareas utilizando el mensaje **mientras:** La tarea receptora continuará ejecutándose hasta que la tarea después de los dos puntos dé un resultado negativo. Así es como funciona:

```
>> x := 0.
{ x añadir: 1. } mientras: { <- x < 5. }.
Salida escribir: x, detener.
```

```
5
```

En el fragmento de código ilustrado anteriormente, se agrega **1** a **x** siempre que **x** sea menor que **5**. Cuando este ya no sea el caso, la segunda tarea responderá **Falso** y, en consecuencia, la ejecución de la primera tarea finalizará.

2.5 Plantillas

La lista de kJ/kcal del capítulo anterior podría hacerse más presentable. Preferiblemente, la lista sería como sigue:

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

Cuando convierte la salida que quieres obtener en una plantilla, su notación podría ser:

```
número1 kJ → número2 kcal
```

Esto significa que **número1** representa el valor de kJ y **número2** el valor de kcal. Así es como funciona la interpolación de cadenas en Citrine. No se necesitan reglas gramaticales diferentes para hacer esto en Citrine. Simplemente envíe la palabra que necesita ser reemplazada al texto con el texto sustituto como argumento:

```
>> texto :=
   ['número1 kJ → número2 kcal']
   número1: 100,
   número2: 23.9.
```

El resultado:

```
100 kj → 23,9 kcal
```

La regla de sustitución funciona para cada mensaje indefinido que recibe un objeto **Texto**. Cada mensaje que no sea reconocido por el objeto **Texto** será interpretado de la siguiente manera: *reemplace el texto del mensaje con el texto dentro del argumento mensaje*. Puede adaptar el programa de la siguiente manera:

```
{ :linea
    >> kJ := linea * 100.
    >> kcal := kJ * 0.239.
    Salida escribir: (
          ['número1 kJ → número2 kcal']
          número1: kJ,
          número2: kcal
    ), detener .
} * 10.
```

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

Para evitar confusiones sobre qué mensaje se puede o no utilizar como sustituto, es mejor introducir un carácter no designado, por ejemplo, un rombo (⬧) (U+2B27, ALT+Z), antes de los segmentos de texto en la plantilla que se deben reemplazar. Este tipo de carácter es a veces no recomendable en el contexto internacional.

2.6 Respondiendo

Hasta ahora, ha enviado mensajes a objetos. Sin embargo, todavía no ha respondido usted a ningún mensaje . Para responder un mensaje se utiliza la flecha de retorno (<-). El siguiente ejemplo ilustra cómo crear una tarea para calcular un porcentaje:

```
>> porcentaje := {
     :numero :porciento
     <- numero / 100 * porciento.
}.

Salida escribir:
( porcentaje aplicar: 100 y: 7 ),
detener.
```

```
7
```

Una vez que se ha definido la tarea y se la ha asignado a la variable **porcentaje**, se envía el mensaje **aplicar:y:** con los argumentos **100 y 7**.

Esto ejecutará la tarea aplicadando a 100 y 7, es decir, 7% de 100. Al utilizar la flecha de retorno, la respuesta se devuelve desde la tarea al programa principal. Aunque este código es válido, tiene una desventaja porque se debe recordar la secuencia de los argumentos. Entonces, ¿por qué no escribirlo así: *7 porciento de: 100 ?* Se vería mucho más natural. Para que esto sea posible, tenemos que adaptar el objeto padre de 7, que es el objeto **Número,** para que comprenda el mensaje **porciento de :**. Esto se puede lograr enviando **en:hacer:** al objeto Número, de la siguiente manera:

```
Número en: ['porciento-de:'] hacer: {
:número
<- número / 100 * yo.
}.
Salida escribir:
(7 porciento-de: 100),
detener.
```

Como el porcentaje , en este caso, es el número en sí, nos referimos a yo, la palabra clave *yo*. En resumen, el símbolo **yo** significa: envíame este mensaje a mí mismo. Después de haber ejecutado el código mencionado anteriormente,
podemos hacer:

```
Salida escribir:
(7 porciento-de: 100),
detener.
```

Naturalmente el resultado mostrado será:

```
7
```

2.7 Herencia y propiedades

Además de ajustar y expandir objetos existentes, también puede crear nuevos objetos usted mismo simplemente enviando el mensaje **nuevo** . Supongamos que desea crear un objeto de factura que aplique una secuencia de numeración; en ese caso, primero necesitará un objeto de factura.

Si sabe que cada objeto deriva de otro objeto, que, a su vez, eventualmente se origina en el objeto raíz de todos los objetos llamado **Objeto**, debe elegir en cuál de esos otros objetos se basará su nuevo objeto. Su nuevo objeto hereda todas las propiedades del objeto en el que se basa, es decir, al que se envió inicialmente el mensaje **nuevo**.

En este ejemplo, se prefiere un objeto bastante neutral, uno sin demasiadas propiedades heredadas. Esto ofrece una elección fácil, porque en ese caso el nuevo objeto puede basarse en el objeto raíz en sí mismo, que es **Objeto**. La notación del sistema de facturas deseado será entonces la siguiente:

```
>> factura := Objeto nuevo.
factura en: ['empezar'] hacer: {
mi numero := 0.
}.

factura en: ['numero'] hacer: {
mi numero añadir: 1.
<- mi numero copiar bruto.
}.
```

Utilice este objeto de la siguiente manera:

```
factura empezar.
Salida escribir: factura numero, detener.
Salida escribir: factura numero, detener.
```

Salida:

```
1
2
```

El número de factura actual se almacena en el objeto, por eso se coloca una palabra clave **propia** delante de él. Esto se llama *propiedad*; más detalles a continuación.

Algunas empresas prefieren tener el año incorporado en los números de factura. En este caso, puede crear un nuevo objeto de factura basado en un objeto de factura anterior, pero que ofrece la posibilidad de su usuario de ingresar un año específico:

```
>> factura-anual := factura nuevo.
factura-anual en: ['empezar:'] hacer: { :año
mi numero := año.
}.
```

Este objeto de factura de año se puede usar como:

```
factura-anual empezar: 202000.
Salida escribir: factura-anual numero, detener.
Salida escribir: factura-anual numero, detener.
```

Salida:

```
202001
202002
```

No es necesario volver a escribir la implementación del mensaje **número**, ya que se hereda del **objeto factura** escrito anteriormente; en consecuencia, se puede reutilizar el código antiguo. Citrine carece de conceptos como clases y otros conceptos relacionados. Esto significa que los objetos solo pueden heredar de otros objetos, también conocido como *herencia prototípica.*

Ahora, volvamos a las propiedades. Las propiedades de los objetos solo se pueden abordar **desde dentro**, por lo que otros objetos no pueden percibirlas. A diferencia de otros lenguajes de programación, todas las propiedades de los objetos son visibles exclusivamente para el objeto que ha creado la propiedad y para los objetos derivados. A continuación, se muestra una breve demostración de este principio. Digamos que desea saber el número de visitantes de una tienda minorista en un momento determinado. Se coloca un sensor en la entrada y se opera mediante el programa del proveedor que se muestra a continuación:

```
>> sensor := Objeto nuevo.
sensor en: ['nuevo-día'] hacer: {
mi contador := 0.
}.
sensor en: ['abrir-puerta'] hacer: {
mi contador añadir: 1.
}.
```

Cada nuevo día, el contador se pondrá a **0** y cada vez que se abra la puerta se añadirá **1** al **contador**. Este objeto en particular podría servir a un programa informático que reciba señales de un sensor físico colocado cerca de la puerta de la tienda. Sin embargo, el objeto **sensor** proporcionado por el proveedor no tiene en cuenta a las personas que salen de la tienda, por lo que es mejor ampliar el objeto.
De forma bastante creativa, la versión mejorada del **sensor** en este ejemplo se llama **sensor2**:

```
>> sensor2 := sensor nuevo.
sensor2 en: ['abrir-salida'] hacer: {
mi contador restar: 1.
}.
```

El objeto **sensor2** se basa en **sensor**, sin embargo, reducimos el contador cada vez que se abre la puerta de salida. El objeto se puede utilizar mediante un programa que procese señales de un sensor físico de la siguiente manera:

```
sensor2
    nuevo-día
    abrir-puerta
    abrir-puerta
    abrir-puerta
    abrir-salida.
```

Después de esta secuencia de mensajes, el contador indica **2**. En este caso, ambos sensores comparten la propiedad **contador**. Como sensor2 se deriva de sensor, el valor del contador se puede modificar. Otros objetos no pueden modificar el valor, por lo tanto, el programador no puede cambiar el valor externamente. Esto evita sorpresas, sabiendo que la propiedad contador solo se puede modificar desde dentro del objeto.
Por eso, el siguiente fragmento de código no tiene sentido:

```
>> sensor3 := Objeto nuevo.
sensor3 en: ['extra'] hacer: {
mi contador añadir: 2.
}.
```

Como **sensor3** no se basa en **sensor** ni en en **sensor2,** el contador de **sensor/sensor2** no se modificará. En cualquier caso, este código generará un error, porque el objeto ni siquiera conoce la propiedad **contador**, ya que el contador no se ha inicializado.

Además, la siguiente construcción no es válida:

```
Salida escribir: sensor mi contador.
```

Además, la palabra clave **mi** nunca se utiliza fuera de los corchetes de un objeto de tarea. En el momento en que veas un fragmento de código de este tipo, sabrás que algo anda mal. La propiedad **contador** solo se puede obtener de una tarea que ya sea parte del objeto que ha declarado la propiedad, o de una tarea que sea parte de un objeto derivado. Aún así, por medio de un mensaje, la propiedad se puede poner a disposición del mundo externo:

```
sensor2 en: ['contador'] hacer: {
<- mi contador.
}.
Salida escribir: sensor2 contador, detener.
```

```
2
```

En la práctica, sin embargo, esto no es una medida inteligente. Un programador descuidado ahora podrá modificar el contador externamente y, al hacerlo, violará la *encapsulación del objeto*. Como lector del código de programación, ya no sabe si los datos del contador son confiables o no, ya que el contador puede modificarse en todo el código de programación. Si desea compartir los detalles del número contado de visitantes de la tienda, pero no quiere que un programador realice modificaciones en otra parte del programa, es recomendable devolver una copia:

```
sensor2 en: ['contador'] hacer: {
<- mi contador copiar.
}.
```

En ese caso, el programador puede leer el contador, por ejemplo para mostrarlo en la pantalla, pero no puede modificarlo. Esto evita errores innecesarios del programa. Tenga en cuenta que esta misma técnica exacta se aplicó en el ejemplo anterior con respecto a los números de factura al enviar el número de factura de vuelta. En ese ejemplo, también se utilizó una **copia** en lugar del original. Se ilustrarán más detalles sobre el mensaje de **copia** en el capítulo 4. Es importante tener en cuenta que Citrine siempre usa referencias. Esto significa que cuando declara un objeto con := o lo devuelve con <-, no se realiza una copia automáticamente. Debe indicarlo explícitamente. El siguiente ejemplo ilustra el efecto de usar el mensaje de copia:

```
sensor2
    nuevo-día
    abrir-puerta
    abrir-puerta
    abrir-puerta
    abrir-salida.
```

En el capítulo siguiente se explicará cómo se aplica el objeto Punto Nótese que un punto no puede leer las coordenadas del otro. Es necesario enviar un mensaje al otro punto para solicitarle estos datos: coordenadas x e y.
Después de todo, como ya se explicó, las propiedades solo se pueden recuperar desde dentro de la propia familia de objetos. El siguiente ejemplo ilustra cómo aplicar el objeto Punto:

```
>> muelle := Punto nuevo x-coordenadas: 5, y-coordenadas: 6.
>> ayuntamiento := Punto nuevo x-coordenadas: 7, y-coordenadas: 1.
>> restaurante := Punto nuevo x-coordenadas: 5, y-coordenadas: 6.
```

Para determinar si el que está situado en el muelle es el restaurante o el ayuntamiento:

```
Salida escribir: muelle = ayuntamiento, detener.
Salida escribir: muelle = restaurante, detener.
```

Salida

```
False
True
```

Tenga en cuenta que aquí se utiliza el mensaje = para ejecutar la comparación de puntos. De igual modo, se podría haber utilizado un mensaje diferente, como: **es:** o **igual:**, sin embargo, reutilizar el carácter = parecía apropiado en
este caso.

```
sensor2 contador añadir: 100.
Salida escribir: sensor2 contador, detener.
```

Resultado

```
2
```

Esto todavía muestra **2**, sin embargo, sin agregar la copia del mensaje, el resultado mostraría **102**.

Una técnica que se usa comúnmente es usar un objeto como una especie de plano, que es, por ejemplo, el caso de las facturas que analizamos anteriormente. Otro buen ejemplo es el **Punto**. Digamos que creas un programa de ordenador que realiza cálculos utilizando puntos en un mapa. Puedes desarrollar el objeto **Punto** que contiene las propiedades de una coordenada x y una coordenada y, que se pondrán a disposición del mundo externo a través de mensajes:

```
>> Punto := Objeto nuevo.
Punto en: ['x-coordenada:'] hacer: { :x mi x := x. }.
Punto en: ['y-coordenada:'] hacer: { :y mi y := y. }.
Punto en: ['x-coordenada'] hacer: { <- mi x copiar. }.
Punto en: ['y-coordenada'] hacer: { <- mi y copiar. }.
```

Ahora, podemos agregar un mensaje a **Punto** para comparar dos instancias, es decir, comparar puntos

```
Punto en: ['='] hacer: { :otra
<- (
mi x = otra x-coordenada
y: mi y = otra y-coordenada
).
}.
```

Este mensaje compara los dos puntos utilizando el mensaje **y:** . La primera comparación da como resultado **Verdadero** o **Falso**. Estos son objetos de decisión. Cuando envía el mensaje **y:** a un objeto Verdadero, y el objeto de argumento después de los dos puntos es igualmente **Verdadero**, entonces recibirá como respuesta nuevamente **Verdadero**.

Cuando cualquiera de los dos sea **Falso**, recibirá la respuesta **Falso**. Más información sobre el objeto de decisión

2.8 Ejercicios

1. ¿Cuál de los siguientes nombres de variable no es válido?

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

2. ¿Qué valor representa x en el siguiente fragmento de código?

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

3. Encuentra el error.

El valor de **x** en el siguiente programa debería ser **42**, pero muestra **2**.
Mejora el programa para que el valor **x** sea igual a **42**.
¿Cuál es el error?

```
>> x := 2.
{ x añadir: x elevar-a: 2. } * 2.
```

4. Coloca una coma en el siguiente fragmento de código, de modo que **x** sea igual a **10**.

```
>> x := 2 elevado a la potencia: 3 + 2.
```

5. Al final del siguiente programa **x** es igual a **-19**. ¿Qué mensaje se debe escribir en la línea de puntos?

```
>> x := 11.
{ :contador
contador ... verdadero: {
x restar: contador.
}.
} * 10.
Salida escribir: x, detener.
```

6. Al final de este próximo programa, **x** es igual a **104**. ¿Qué mensaje se debe escribir en la línea de puntos?

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

7. ¿Qué se debe escribir en la línea de puntos para ver el nombre de una bebida popular impresa en la pantalla? Puedes elegir repetidamente entre **verdadero:** o **falso:**.

```
>> x := 7.
(x > 7) ...: { Salida escribir: ['p']. }.
(x < 7) ...: { Salida escribir: ['o']. }.
(x >=: 7) ...: { Salida escribir: ['t']. }.
(x <=: 7) ...: { Salida escribir: ['e']. }.
(x !=: 7) ...: { Salida escribir: ['n']. }.
(x = 7) ...: { Salida escribir: ['t']. }.
(7 > x) ...: { Salida escribir: ['i']. }.
(7 < x) ...: { Salida escribir: ['a']. }.
(7 >=: x) ...: { Salida escribir: ['l']. }.
(7 <=: x) ...: { Salida escribir: ['l']. }.
(7 !=: x) ...: { Salida escribir: ['y']. }.
```

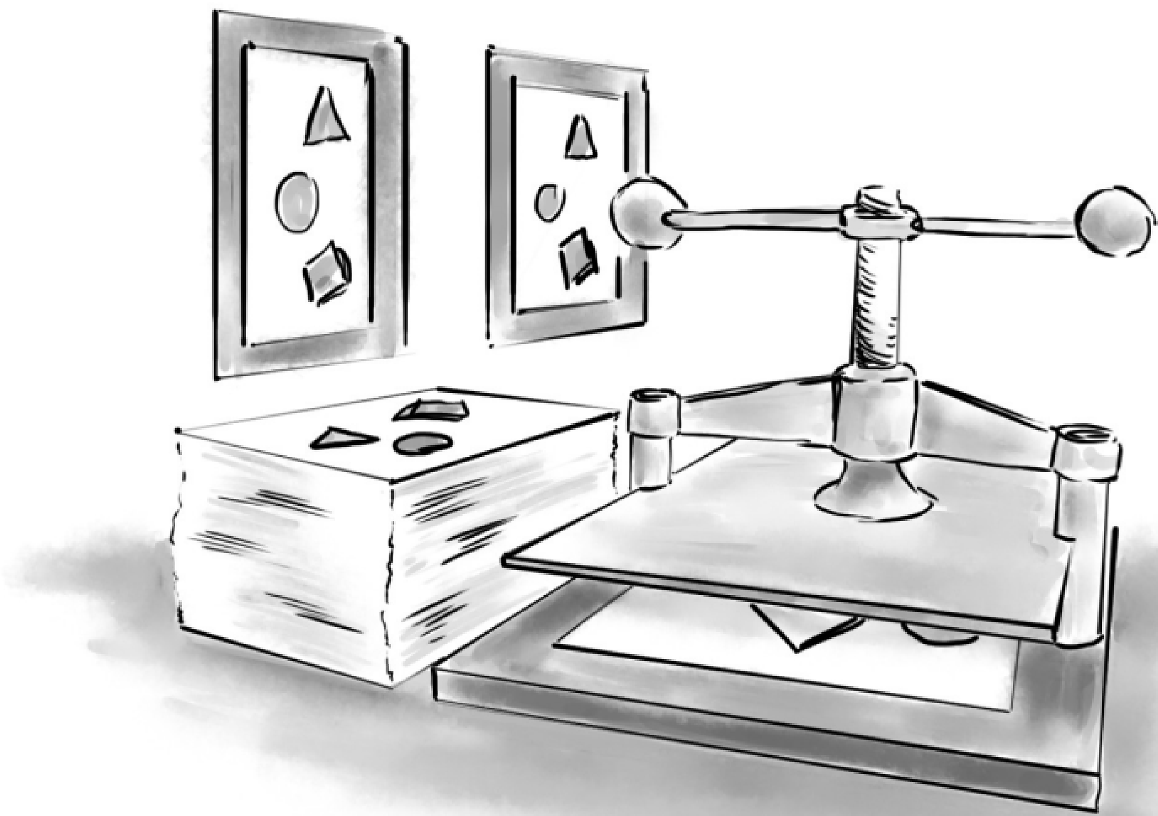
8. ¿Cuáles son las respuestas a los siguientes problemas?

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

9. El siguiente programa genera un resultado **Falso** mientras que se esperaba un **Verdadero**. Esto se debe a un error de código. Mejore el programa, de modo que el resultado proporcione la respuesta correcta en 6 días soleados y 3 días tropicales.

```
>> ¿Ola-de-calor? := {
       :días-soleados :días-tropicales
       (días-soleados >=: 5) verdadero: {
              (días-tropicales >=: 3) verdadero: {
                 <- Verdadero.
              }.
       }.
       <- False.
}.
```

```
Salida escribir: ( ¿Ola-de-calor? aplicar: 6 y: 3), detener.
```



3. Objetos

3.1 Objetos simples

El mundo de Citrine está lleno de una diversidad de objetos, que están disponibles para su programa desde el principio. En este capítulo, se revisarán todos estos objetos del sistema. Citrine reconoce 12 objetos base en total; a saber, el objeto **Nulo**, el objeto raíz llamado **Objeto**, el objeto **Decisión** (y sus dos instancias, **Verdadero** y **Falso**), **Número**, **Texto**, **Tarea**, **Serie**, **Lista**, **Archivo**, **Momento**, **Programa** y **Salida** (el objeto de salida). Conocer estos 12 objetos es conocer Citrine. Además, hay dos objetos implícitos: **Ubicación-del-archivo** y **Instrucción.** Estos dos se analizarán junto con el objeto **Archivo** y el objeto **Programa**. Su función es muy limitada y depende de los objetos mencionados anteriormente. Algunos de los objetos de la lista mencionada anteriormente no son muy complejos.

El más sencillo es el objeto de salida. Este objeto gestiona la salida del programa a otros sistemas, que en la mayoría de los casos es la pantalla, pero también podría ser un servidor, otro programa o una impresora. El objeto de salida entiende sólo dos mensajes, que se han visto en los capítulos anteriores: **escribir:** (escribir algo) y **detener** para comenzar una nueva línea.

```
Salida escribir: ['¡La brevedad es el alma del ingenio!'] , detener .
```

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

El mensaje **escribir:** siempre debe ir seguido de un objeto Texto. Si envías algo diferente, el objeto de salida intentará convertir el mensaje en texto, es decir, conversión implícita. La conversión exacta depende del objeto en sí. En general, con los números, se utiliza su representación textual, lo que también se aplica a los objetos de decisión. Otros objetos pueden decidir por sí mismos cómo presentarse como texto. Más sobre la conversión en los capítulos 3.5 y 4.6.

Otro objeto muy básico es el objeto **Nulo**. Este objeto representa el vacío, o mejor aún, la ausencia de información. En ocasiones, recibirás este objeto como respuesta a un mensaje, en caso de que el resultado sea nada. La pregunta más esencial que puedes hacerle al objeto **Nulo** es: **¿nulo?**.
La respuesta siempre será **Verdadero**

```
Salida escribir: Nulo ¿nulo? .
```

```
True
```

Cualquier objeto que no sea **Nulo** responderá con **Falso**. El fragmento de código anterior puede parecer un poco filosófico, pero el objeto Nulo ciertamente tiene aplicaciones extremadamente prácticas. Por ejemplo,

recibirá el objeto Nulo como respuesta en caso de que solicite un elemento de una serie que no existe.
También puede usar el objeto Nulo si desea declarar una variable, pero no desea especificar un valor aún. En su lugar, simplemente asigne el valor especial **Nulo**. En Citrine **no está permitid**o declarar una variable sin valor, como:

```
>> valor.
```

En su lugar, debe escribir lo siguiente:

```
>> valor := Nulo.
```

3.2 Objeto booleano

En los capítulos anteriores se han analizado en profundidad los objetos **Verdadero** o **Falso**. Hemos utilizado estos objetos, por ejemplo, para ejecutar tareas en función de ciertas condiciones. **Verdadero** y **Falso** se basan en el álgebra de Boole.

A diferencia de la mayoría de los lenguajes de programación más populares en el momento de escribir este artículo, Citrine proporciona solo un único objeto Verdadero y un único objeto Falso. Para aclarar, cada vez que escribe **Verdadero** no implica que se haya creado un nuevo objeto. En cambio, siempre utiliza una referencia. Esto significa que cuando escribe lo siguiente:

```
>> x := Verdadero.
```

La **x** se refiere al objeto **Verdadero**. El código condicional y los bucles también verifican esta referencia. En Citrine, el significado de Verdadero y Falso no es fijo. De hecho, un programa Citrine se confunde bastante con una declaración como esta:

```
Verdadero := Falso.
```

El resultado de tales acciones no está definido, pero sigue siendo una acción válida y, por lo tanto, está permitida formalmente. Además, hay un objeto Decision, que es el objeto raíz de **Verdadero y Falso**, ya que ambos son derivados del objeto raíz. Sin embargo, el objeto Decision en sí no proporciona ninguna
aplicación práctica.

Por medio de los objetos Verdadero y Falso, puede ejecutar partes de su programa bajo condiciones específicas:

```
>> precio := Número entre: 0 y: 20.
>> presupuesto := 10.
precio > presupuesto verdadero: {
    Salida escribir: ['demasiado caro'].
}, falso: {
    Salida escribir: ['¡vendido!'].
}.
```

Para facilitar la lectura, en lugar de **falso:** también puede utilizar el mensaje **otro:**. Esto es básicamente lo mismo , porque el mensaje **otro:** es un sinónimo de **falso:**. (otro , hace referencia a - otro caso -) Por lo tanto, el siguiente fragmento muestra exactamente el mismo procedimiento:

```
>> precio := Número entre: 0 y: 20.
>> presupuesto := 10.
precio > presupuesto verdadero: {
    Salida escribir: ['demasiado caro'].
}, otro: {
    Salida escribir: ['¡vendido!'].
}.
```

Depende totalmente de usted cuál de las dos variaciones prefiere utilizar para facilitar la lectura.

Los objetos de decisión también se pueden utilizar para combinar diferentes condiciones. Por ejemplo, una tarea se puede ejecutar cuando dos cosas producen un objeto Verdadero. O bien, si se cumple una de dos o incluso más condiciones, como se ilustra en el siguiente ejemplo:

```
>> azúcar := Verdadero.
>> leche := Falso.
>> x := azúcar y: leche.
```

En el fragmento anterior se crean dos nuevos objetos, que son **azúcar** y **leche**. El objeto **azúcar** es un objeto **Verdadero** mientras que **leche** es un objeto **Falso**. Ahora, cuando quieres saber si alguien prefiere tanto leche como azúcar en su café o té, envías el mensaje **y:** a un objeto y envías el otro objeto junto con él como argumento. La respuesta a **y:** será **Falso**, porque **y:** solo responde con **Verdadero** cuando el objeto en sí es afirmativo, así como el argumento. En cualquier otro caso, devolverá **Falso**. Entonces, en este caso **x** es igual a **Falso**. Además del mensaje **y:**, también puedes usar **o:** , de la siguiente manera:

```
>> azúcar := Verdadero.
>> leche := Falso.
>> x := azúcar o: leche.
```

En este caso, el azúcar, coincidentemente el primer objeto indicado, responderá **Verdadero**. El objeto Verdadero al que se refiere la variable azúcar, en realidad siempre responde a la pregunta **o:** con **Verdadero** si efectivamente el objeto en sí es afirmativo o efectivamente el objeto en el argumento es afirmativo, o si ambos son **Verdadero**. En caso de que ninguno de los objetos sea **Verdadero**, entonces la respuesta será **Falso**.

Al usar el mensaje **o:** puede esperar **Verdadero** si está rodeado por dos objetos **Falso**.

```
>> azúcar := Falso.
>> leche := Falso.
>> x := azúcar o: leche.
```

Con el mensaje **no** se invierte un objeto de decisión, es decir, un **Verdadero** se convierte en **Falso**

```
>> azúcar := Falso no.
>> leche := Falso no.
>> x := azúcar y: leche.
```

Entonces, en este caso **x** es **Verdadero**, porque tanto azúcar como leche son **Falso no**, es decir, no no, por lo tanto **Verdadero**.

Con **entonces:caso-contrario**: puedes seleccionar, por medio de un objeto Verdadero o Falso, un objeto diferente de un par de objetos. De la siguiente manera:

```
>> azúcar := Verdadero.
>> x :=
    azúcar
    entonces: 1 cuchara caso-contrario: Nulo.
```

En el fragmento anterior **x** será igual a **1 cuchara**. Cuando el mensaje de recepción es **Verdadero,** se seleccionará el primer objeto y si no, se seleccionará el segundo.

También puedes convertir un **Verdadero** o un **Falso** en un número enviando el número del mensaje. El objeto **Verdadero** responderá con **1** y **Falso** responderá con **0**. Obviamente, también puedes comparar objetos Verdadero y Falso entre sí:

```
>> x := (Verdadero !=: Falso).
>> y := (Verdadero = Falso).
```

En este caso **x** es igual a **Verdadero** e **y** es igual a **Falso**.

Los mensajes **continuar** y **salir** se pueden utilizar dentro de un bucle. Por lo tanto, cuando envía **salir** dentro de un bucle a un objeto Verdadero, el bucle finalizará desde ese punto. El programa continuará en la primera instrucción después del bucle.

El mensaje **continuar** está relacionado con **salir**, sin embargo, en lugar de finalizar todo el bucle, solo finalizará la iteración actual. La parte restante de la ronda actual se omite y se inicia la siguiente ronda del bucle.

A continuación, se muestran algunos ejemplos:

```
{ :i
   Salida escribir: i, detener.
   (i > 9) salir.
} * 20.
```

Salida:

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

Este fragmento de código en particular ilustra el efecto de la interrupción del mensaje. Muestra una pequeña lista de los números del 1 al 10; sin embargo, tan pronto como el número en i supera diez, el bucle se interrumpe en ese punto.

```
{ :i
    (i > 10 and: i < 15) continuar.
    Salida escribir: i, detener.
} * 20.
```

Salida

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

Este fragmento de código produce una pequeña lista de 1 a 10 seguida de 15 a 20. Los números entre 10 y 15 se omiten. Esto se debe a que para este intervalo la condición (**i > 10 y: i < 15**) da como resultado **Verdadero**, por lo que el mensaje **continuar** se envía a este objeto Verdadero. Esto significa que se omite la parte restante del bucle y comienza la siguiente iteración. Los mensajes continuar y salir son ignorados por el objeto **Falso**.

Finalmente, el mensaje **texto** le permite crear una representación textual de un objeto Verdadero o Falso.
Los resultados son, por supuesto, bastante sencillos:

```
Salida escribir: Verdadero texto, detener .
Salida escribir: Falso texto, detener.
```

El resultado:

```
True
False
```

3.3 El objeto Número

Cada vez que escriba un número, por ejemplo 9, -10 o 3,12, detrás de la pantalla,Citrine convertirá estos números en un objeto **Número**. Puede enviar mensajes a este objeto Número, o puede asignar el número a una variable y enviar mensajes después:

```
10 ¿par?
>> diez := 10.
diez ¿par?
```

Ambas notaciones son válidas. El objeto Número responde a los siguientes mensajes como: >, ≥(>=:), <, ≤ (<=:), =, ≠ (!=:), +, –, *, /, **entre:y: , ¿impar?** , **¿par?**, **añadir:** , **restar:**, **multiplicar-por:** , **dividir-por:**, **módulo:**, **elevar-a:**, **¿positivo?**, **¿negativo?**, **redondear**, **redondear-al-alza** , **redondear-a-la-baja** , **raíz-cuadrada** , **absoluto** , **texto** , **bruto** , **calificación** , **booleano** y **calificar:**

La mayoría de estos mensajes se explican por sí solos y permiten ejecutar operaciones matemáticas (+) o comparaciones (>). La diferencia entre los mensajes matemáticos binarios (+) y sus variaciones de palabras clave (**añadir:**) es que los primeros devolverán un nuevo número, que es el resultado de la operación, mientras que con los últimos **se modificará** el objeto en sí. Esto se ilustra en el siguiente ejemplo:

```
>> a := 1.
>> b := a + 3.
a añadir: 2.
Salida escribir: a, detener.
Salida escribir: b, detener.
```

En el ejemplo anterior, b = 4 y a = 3. Con **añadir: 2** el valor de a se aumenta en 2, mientras que **+ 3** crea un nuevo número que es igual a a + 3. Lo mismo se aplica a otros procesos matemáticos, por ejemplo, las multiplicaciones. Al utilizar el símbolo de multiplicación, recibirá un nuevo objeto como respuesta. En caso de que utilice el mensaje **multiplicar-por:**, multiplicará el número en sí.

Con el mensaje **entre: y:**, por ejemplo en: **Número entre: X y: Y**, obtendrá un número entre X e Y. De esta manera, se puede generar cualquier número aleatorio:

```
>> a := Número entre: 1 y: 10 .
```

```
Salida escribir: a, detener.
```

```
5
```

Tenga en cuenta que el generador integrado de números aleatorios en Citrine no es adecuado para aplicaciones criptográficas.

De manera predeterminada, los números se formatean según las reglas locales:

```
Salida escribir: 5000, detener.
Salida escribir: 5,000, detener.
```

Resultados en:

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

Para mostrar números sin separadores:

```
Salida escribir: 5000 bruto, stop.
```

Esto se traducirá como:

```
5000
```

El mensaje simple devuelve un objeto de texto que admite la representación de números sin marcas separadoras. A continuación, se muestran algunos ejemplos de otros mensajes que puede enviar a un número:

```
Salida escribir: 5.25 redondear, detener.
Salida escribir: 5.25 redondear-al-alza , detener.
Salida escribir: 5.25 redondear-a-la-baja , detener.
Salida escribir: 25 raíz-cuadrada, detener.
Salida escribir: (2 elevar-a: 8), detener.
Salida escribir: (5 módulo: 2), detener.
```

```
Salida escribir: 3 ¿par?, detener.
Salida escribir: 3 ¿impar?, detener.
Salida escribir: 3 ¿positivo?, detener.
Salida escribir: 3 ¿negativo?, detener.
```

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

Puedes adjuntar un calificador a un número, por ejemplo, 6 manzanas. Cada mensaje que no sea reconocido por un número será considerado un calificador. Puedes recuperar el calificador de un número por medio del calificador de mensaje:

```
>> cantidad := 6 monedas.
Salida escribir: cantidad.
Salida escribir: cantidad calificador.
```

```
6 coins
coins
```

Un calificador es básicamente un objeto de texto que se almacena con el objeto de número. El calificador también se imprime después del número en una asignación de escritura (**escribir:**) . Los calificadores se pueden utilizar para sumar cantidades en monedas mixtas, por ejemplo. Al sumar las cantidades, puede solicitar los calificadores. El siguiente ejemplo de programa ilustra este principio utilizando una calculadora de moneda histórica (¡ya que el tipo de cambio se mantiene razonablemente estable!).

```
Número en: ['+'] hacer: { :cantidad
     (cantidad calificador = ['ducados']) verdadero: {
         cantidad multiplicar-por: 5.
      }.
      yo añadir: cantidad.
}.
Salida escribir: (7 florines + 3 ducados), detener.
```

```
22 florins
```

En el fragmento de código anterior, el mensaje + se adapta de tal manera que se tiene en cuenta la moneda. En este ejemplo, 1 ducado equivale a 5 florines. La palabra clave **yo** indica el objeto en sí: **yo** se utiliza para enviar un mensaje al objeto en sí. Además, también se puede establecer un calificador de forma explícita utilizando el calificador de mensaje:

```
>> x := 7 calificador: ['ducatos'].
```

3.4 El objeto de texto

Cada vez que se coloca un texto entre comillas ['…'], Citrine creará un nuevo objeto de texto para usted. Sin embargo, asegúrese de utilizar las comillas correctas. Las comillas al principio del texto difieren de las que se encuentran al final.

Puede enviar mensajes a este objeto de texto, al igual que puede enviar mensajes a otros objetos:

```
['Hola'] longitud.
>> saludo := ['Hola'].
salud longitud.
```

Ambas variaciones son válidas.
También se permite utilizar comillas dentro del propio texto:

```
[' ['Me gusta mi nueva computadora'] dijo el robot. ']
```

Puede utilizar comillas en un texto, siempre que también las cierre. Por lo tanto, los siguientes textos no son válidos:

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

Sin embargo, si desea utilizar comillas de esta manera, será necesario preceder el texto con el carácter de escape \:

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

Con un \n dentro de un texto puede comenzar una nueva línea. Con un \t puede agregar un carácter TAB.

El objeto Texto responde a mensajes como: =, ≠ (!=:), +, –, >, < , ≤ (<=:) , ≥ (>=:), **añadir:** , **longitud** , **desde:longitud:** , **caracter:** , **encontrar**: , **mayúsculas** , **minúsculas**, **compensar:** , **reemplazar:con:** , **contiene:** , **recortar** , **número** , **dividir:** , **caracteres** , **código** , **objeto**, **booleano** , y **compare:** .

Los mensajes =, ≠ (!=:) , >, <, ≤ (<=:), ≥ (>=:) permiten comparar dos textos: p. ej., > devuelve **Verdadero** si el objeto Texto receptor viene después del objeto incluido, según el diccionario:

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

Asimismo, puede utilizar el mensaje **comparar:** para expresar la diferencia en la secuencia por medio de un número. El siguiente código de programación, por ejemplo:

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

devuelve 2, porque la letra **c** está dos lugares después de la letra **a**. Tenga en cuenta que al momento de escribir esto, las ecuaciones están limitadas por la implementación y siguen la codificación UTF-8.

Al utilizar el signo más (+) puede concatenar varios textos:

```
>> nombre := ['Juan'].
['Hola '] + nombre.
```

Al utilizar el mensaje **añadir:** obtendrá el mismo resultado:

```
['Hola '] añadir: nombre.
```

Sin embargo, una forma más elegante de hacerlo es la siguiente:

```
>> nombre := ['Juan'].
['Hola persona'] persona: nombre.
```

En este caso, se utiliza un texto de plantilla y se reemplaza una palabra en particular de ese texto. Esto generalmente beneficia la legibilidad. De hecho, es lo mismo que:

```
>> nombre := ['Juan'].
['Hola persona'] reemplazar: ['persona'] con: nombre.
```

Sin embargo, esta variación detallada que utiliza el mensaje **reemplazar:con:** , tiene un impacto negativo en la legibilidad.

El símbolo menos hace exactamente lo opuesto al símbolo más, porque da un nuevo objeto Texto del cual se omite la última parte:

```
>> mensaje := ['Sin signos de exclamación!!!'].
mensaje := mensaje - ['!!!'].
```

Aquí, se omiten los signos de exclamación al final del texto. El símbolo menos omite el texto correspondiente al final del objeto Texto y, por lo tanto, hace exactamente lo contrario del símbolo más. Por lo tanto, cuando el mensaje incluye
['!!! Sin signos de exclamación!!!'], solo se omitirán los últimos signos de exclamación mencionados.

Al enviar **longitud** como mensaje a un texto, obtendrá la cantidad de caracteres que contiene ese texto en particular:

```
Salida escribir: (['歡迎'] longitud ), detener.
```

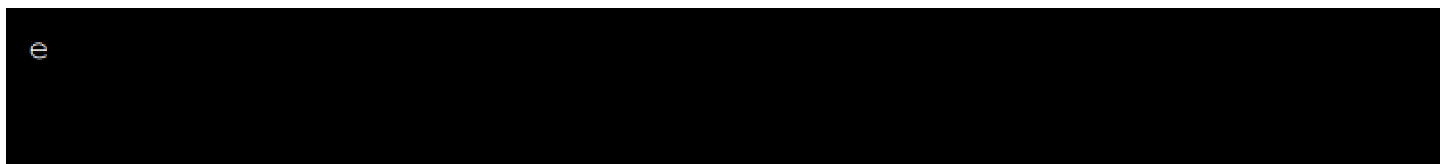
El resultado del ejemplo mencionado anteriormente muestra **2**, porque hay dos caracteres chinos (que significan "**bienvenido**") entre las comillas. Asegúrese de no confundir esta cantidad de caracteres con la cantidad de bytes. La mayoría de los lenguajes de programación devuelven la cantidad de bytes (que sería más de dos) según el conjunto de caracteres. Citrine no maneja conceptos específicos del sistema y de bajo nivel; por lo tanto, la cantidad de bytes es irrelevante. Cualquier operación de bytes requerida debe procesarse mediante un programa especializado o una biblioteca de software.

Cabe destacar que esta restricción también garantiza que Citrine siga siendo independiente de la plataforma. Citrine no se verá afectado por los cambios si, por ejemplo, en el futuro, la gente usara computadoras con una unidad básica distinta de un **byte**.

El mensaje **caracter:**, le permite recuperar el carácter en una posición determinada en el texto, por ejemplo:

```
Salida escribir: (['Hello'] caracter: 2), detener.
```

El resultado es:



Tenga en cuenta que Citrine es un **lenguaje basado en 1**, lo que significa que toda numeración **comienza con 1**. La segunda letra de **Hello** es, por lo tanto, **e** y no **l**; lo que sería en el caso de la mayoría de los lenguajes de programación populares en el momento de escribir este artículo. Utilice **contiene:** cuando desee saber si un texto incluye un fragmento determinado:

```
Salida escribir: (['Hello World'] contiene: ['He']), detener.
```

El resultado se mostrará:

```
True
```

Con **encontrar:** puedes recuperar la posición inicial del fragmento de texto:

```
Salida escribir: (['Hello World'] encontrar: ['He']), detener.
```

por lo que el resultado será:

```
1
```

Esto se debe a que el texto "**He"** comienza en el primer carácter de la sentencia "**Hello world**". Nuevamente,la numeración comienza en 1. ¿Desea recuperar la posición más a la derecha del fragmento? Utilice el
mensaje **último:**.

```
Salida escribir: (['Hello World'] último: ['Wo']), detener.
```

```
11
```

Con **desde:longitud:** puedes copiar un fragmento de una setencia.

```
Salida escribir: (
     ['Hello world'] desde: 7 longitud: 5
), detener.
```

resulta en::

```
World
```

Otro método para copiar un fragmento de texto es enviar el mensaje **saltar:**. De esta manera, puede copiar todo el texto de una posición determinada. El mismo resultado que en el ejemplo mencionado anteriormente se puede lograr con:

```
Salida escribir: (['Hello World'] saltar: 6), detener.
```

Con el mensaje **recortar** puede omitir los espacios en blanco en el lado izquierdo y derecho de un texto:

```
[' Hello '] recortar
```

dará como resultado:

```
['Hello']
```

También se eliminarán las nuevas líneas.

Además, el objeto Texto responde a los mensajes en **mayúsculas** y **minúsculas** devolviendo una copia del texto en mayúsculas y minúsculas respectivamente.

```
Salida escribir: ['Hello'] mayúsculas, detener.
Salida escribir: ['Hello'] minúsculas, detener.
```

```
HELLO
hello
```

En el capítulo 3.8 se analizará el objeto Serie. Al utilizar el mensaje **dividir:** puede crear una secuencia de texto pero se hablará más sobre este tema en el capítulo correspondiente. Ahora veamos un ejemplo.

El siguiente programa le ayuda a crear una contraseña. En primer lugar, se debe crear un tipo de letra con los caracteres adecuados. Tenga en cuenta que no todos los caracteres son igualmente adecuados para crear una contraseña.
Los caracteres que se parecen demasiado pueden causar cierta confusión al leerlos. Por eso, las letras y los números que se parecen demasiado se omiten del tipo de letra, como **l** y **1**. El generador de ejemplo crea contraseñas contando 16 caracteres. Esto se logra cuando se redacta una tarea que elige un carácter del tipo de letra y esta tarea se repite 16 veces.

```
>> typecase := ['acdefghkmnprtx35789@#&']
```

```
>> password := [''].
{
    >> caracter :=
    typecase
    caracter: (
       Número entre: 1 y : typecase longitud
    ).
    password añadir: caracter.
} * 16.
Salida escribir: password, detener.
```

```
Gen8xkhdhc99977c
```

3.5 Conversiones

En algunos casos, se devuelve un número en forma de texto. Por ejemplo, como entrada del usuario, de la siguiente manera:

```
>> x := Programa pedir.
```

El objeto **Programa** se analizará en detalle más adelante, sin embargo, por ahora será suficiente saber que el mensaje **pedir** espera la entrada del usuario hasta que se presione la tecla ENTER. La respuesta se devuelve como texto en la variable **x**. Para convertir este texto en un número, se envía el mensaje **número** y, a partir de ese momento, se pueden enviar mensajes numéricos. En general, puede crear una copia de cada objeto con un tipo diferente utilizando los siguientes mensajes:

Mensaje	Efecto
número	Convierte un objeto en un número
texto	Convierte un objeto en texto
booleano	Convierte un objeto en un objeto booleano (Verdadero o Falso)

Se aplican las siguientes reglas:

Mensaje	**Valor**				
	Nulo	Verdadero / Falso	Número	Texto	Otro
booleano	Falso	-	0 → Falso otro: Verdadero	Verdadero	Mayormente verdadero
número	0	Verdadero → 1 Falso → 0	-	['1']→ 1 ['2'] → 2 etc.. ['?'] → 0	Mayormente 1
texto	['Nulo']	Verdadero → ['Verdadero'] Falso → ['Falso']	1 → ['1'] 2 → ['2'] etc..	-	Depende...

De este gráfico se puede concluir, por ejemplo, que en caso de que el mensaje **booleano** se envíe a un objeto **Nulo**, se puede esperar la respuesta **Falso** (arriba a la izquierda del gráfico), y si se envía un **número**, se puede esperar el resultado **0**. Cuando el mensaje es de texto, la respuesta será el texto ['Nulo'] (abajo a la izquierda del gráfico).

3.6 Objetos tarea

El objeto de tarea responde a mensajes como: **empezar, *, mientras:, error:, capturar: y proceder:**.
El capítulo 2 ilustró cómo un fragmento de código determinado se puede ejecutar varias veces; es decir, enviando el mensaje * a una tarea, seguido de la cantidad de iteraciones requeridas. Para un recordatorio rápido, aquí hay dos nuevos ejemplos de ese tipo de mensajes. Primero, el bucle simple:

```
{ :grados
    Salida escribir:
    grados * 1.8 + 32, detener.
} * 30.
```

Este ejemplo devuelve una pequeña lista de temperaturas que van desde 1 a 30 grados Fahrenheit

```
33.8
35.6

...

84.2
86
```

Esta secuencia se crea cuando la tarea entre corchetes **{}** se multiplica por el número **30**.

```
>> grados := 0.
{
    Salida escribir:
    grados * 1.8 + 32, detener.
    grados añadir: 1.
} mientras: { <- grados <=: 0. }.
```

El fragmento ilustrado anteriormente devuelve el mismo resultado, sin embargo, utiliza un bucle while. En este caso, el mensaje **mientras:** se envía a la tarea con el argumento una segunda tarea. La primera tarea se ejecutará repetidamente, hasta que la segunda tarea devuelva un **Falso**. Para aclarar, mientras la respuesta a la primera tarea siga siendo **Verdadera**, la primera tarea continúa ejecutándose. Debido a que cada vez que se agrega 1 a la cantidad de grados durante la ejecución de la primera tarea, la segunda tarea devolverá Falso tan pronto como cuente 31
grados Fahrenheit.

Una forma diferente de manipular el flujo del programa informático es definir un fragmento de código con controladores en caso de error. Por ejemplo, digamos que una aplicación de presupuesto tiene que determinar el presupuesto mensual en función de la entrada de un usuario sobre los ingresos y la cantidad de meses.

```
>> presupuesto := ingresos / meses.
```

¿Puedes adivinar qué sucede cuando la cantidad de meses resulta ser 0? Lo que sucede es un mensaje de error:

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

La primera línea del mensaje muestra que se ha producido un error que no ha sido gestionado por el programa. La segunda línea de la salida indica el problema real, que es, en este caso, que no se permite la división por 0. La primera línea también revela que se puede **gestionar** un error. Este también es un caso de ejecución de código condicional; sin embargo, en esta situación se escribirá y ejecutará un fragmento de código en caso de que se produzca un error provocado por un fragmento de código diferente. Por ejemplo, vea cómo modificar este
mensaje de error:

```
{
   >> presupuesto := ingresos / meses.
} capturar: { :error
  Salida escribir: ['¡No permitido!'], detener.
}, empezar.
```

En este caso la salida será (en 0 meses)

```
Not allowed!
```

En este caso, ha gestionado su propio manejo de errores. Tenga en cuenta que falta la sentencia de -error no manejado - ; después de todo, usted manejó bien el error y, según Citrine, ese es el final.

Sin embargo, ¿cómo funciona exactamente el manejo de errores en el ejemplo anterior? En primer lugar, hay dos tareas: la tarea original y la tarea de manejo. Estas tareas están vinculadas por el mensaje **capturar:** Al recibir el mensaje capturar: una tarea con la captura, el objeto Tarea receptor cambiará, en caso de que ocurra un error, al código dentro del objeto Tarea que se ha asignado para realizar el manejador de errores. Después de que los dos bloques estén vinculados, es obvio qué hacer en caso de que ocurra un error. No olvide iniciar el primer objeto Tarea, de lo contrario, las dos tareas están vinculadas, pero

En realidad, no se está iniciando nada. Esto introduce el mensaje final en el fragmento de código: **empezar**. Esto inicia la ejecución del primer objeto Tarea, porque el mensaje **capturar:** produce el propio objeto Tarea como respuesta. Además, dado que se trata de un mensaje de palabra clave, se necesitará una coma para encadenar el nuevo mensaje unario. Formalmente, la estructura se puede observar de la siguiente manera:

```
{ Tarea 1 } capturar: { Tarea 2 }, empezar.
```

También puede provocar que se produzca un error en su programa de forma intencionada y, al hacerlo, activar el bloque manejador. Esto se hace enviando el mensaje **error:** a la tarea actual, como se muestra en la siguiente
ilustración:

```
{
    esta-tarea error: ['Whoops!'].
} capturar: { :mistake
    Salida escribir: mistake.
}, empezar.
```

Resultado:

Tenga en cuenta que el objeto de error incluido en el mensaje **error:**, se devuelve en la rutina del controlador. De esta manera, se pueden pasar diversos objetos de error autoinducidos a su tarea de controlador. Además, muestra cómo puede comunicarse con la tarea actual enviando un mensaje a *esta-tarea*.

En lugar de enviar el mensaje **empezar**, se puede utilizar el mensaje *** 1** para ejecutar la tarea una vez. Ambos mensajes terminan teniendo el mismo resultado. Un mensaje que es muy similar a **empezar (o * 1**, dependiendo de cómo lo mire) es el mensaje **proceder**.

Este mensaje se utiliza a menudo para mejorar la
legibilidad. En el siguiente ejemplo se analizarán los diversos aspectos de cómo se utiliza el objeto Tarea en la práctica. Este próximo programa de demostración tiene como objetivo convertir un número en números romanos. Este
ejemplo en particular se limita a números inferiores a 40.

```
>> numero := 17.
{
    {
        numero >=: 10 verdadero: {
```

```
      Salida escribir: ['X'].
      numero restar: 10.
   }, salir.
   numero >=: 9 verdadero: {
      Salida escribir: ['IX'].
      numero restar: 9.
   }, salir.
   numero >=: 5 verdadero: {
      Salida escribir: ['V'].
      numero restar: 5.
   }, salir.
   numero >=: 4 verdadero: {
      Salida escribir: ['IV'].
      numero restar: 4.
   }, salir.
   numero >=: 1 verdadero: {
      Salida escribir: ['I'].
      numero restar: 1.
   }, salir.
  } proceder.
} mientras: { <- numero > 0. }.
```

```
XVII
```

El programa en su forma actual devuelve el número romano correspondiente al número **17**. Al modificar el valor del número 17 en la parte superior del programa por algo diferente, por ejemplo 20, el resultado cambiará igualmente (**XX**). Ahora, observe más de cerca el programa. Los corchetes más externos pertenecen al mensaje while:. En este punto, vincula dos tareas. La primera tarea continúa ejecutándose mientras la segunda tarea siga devolviendo **Verdadero** como respuesta. Esa segunda tarea es bastante simple:

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

Esto significa que, mientras **numero** sea mayor que **0**, se ejecutará la primera tarea. La primera tarea es un poco más larga, sin embargo, es básicamente un conjunto de tareas condicionales. Eche un vistazo al primer fragmento:

```
numero >=: 10 verdadero: {
 Salida escribir: ['X'].
 numero restar: 10.
}, salir.
```

En este caso, se pregunta si el número es mayor o igual a **10**. Si la respuesta a esta pregunta es **Verdadero**, se escribe el símbolo romano del número **10**, que es **X**. Para continuar, se restan 10 al número y se termina esta ronda. En consecuencia, el mensaje **salir** salta del bucle y comprueba si el número sigue siendo mayor que **0**. Si es así, se inicia otra ronda. En el caso de que el número sea **20**, se llega de nuevo a esta primera parte. En el caso de **17**, la respuesta a la pregunta si el número **≥ 10** debe leerse claramente **Falso**, ya que 17 - 10 = 7, y 7 es menor que 10. En este caso, se desciende al siguiente bloque (**número ≥ 9**). Nuevamente, la respuesta será Falso, ya que 7 es menor que 9. Luego, se llega a (**número ≥ 5**), lo que da como resultado **V.** De esta manera, se va cortando el número romano más grande posible de su número hasta que no quede nada.

Sin embargo, hay un pequeño tecnicismo oculto en este programa. Tenga en cuenta que también se envía el mensaje **salir**, en caso de que exista la posibilidad de mostrar un número romano en la pantalla. Esto es para evitar la visualización de un número romano más pequeño demasiado pronto. En el caso de **20 (XX)**, no es deseable mostrar un **IX** después de la **X**; básicamente, el objetivo es reiniciar el proceso. De hecho, es por eso que los comparadores de números están dentro de una tarea y el mensaje **proceder** se envía a la tarea externa. Este procedimiento garantiza la posibilidad de abortar la tarea después de que se haya realizado una comparación exitosa.

Citrine proporciona a las tareas la opción de inyectar valores. Ahora, observe la tarea a continuación:

```
>> enviando := {
Boletin hacia: mi recipiente.
}.

enviando asignar: ['recipiente']
valor: ['info@citrine-lang.org'].
enviando empezar.
```

Esta ilustración presenta una tarea imaginaria que envía un boletín informativo a una dirección de correo electrónico o **destinatario**. Este destinatario se puede inyectar en la tarea, de forma externa y antes de que se inicie la tarea, enviando el mensaje **asignar:valor:** a la tarea. Al hacerlo, el valor del **destinatario** se preestablece en la tarea. Este valor también se puede modificar y la tarea se puede volver a ejecutar. Este es un método útil cuando se utilizan objetos de tarea

No se permiten tareas vacías. En teoría, una tarea vacía se vería así: **{}**, sin embargo, Citrine percibe esto como un error de lenguaje. Si desea declarar una tarea vacía, puede utilizar el objeto Nulo:

```
>> tarea := Nulo.
```

Aunque no se trata de una tarea real, sino de un objeto Nulo, se puede enviar un mensaje de inicio:

```
>> respuesta := tarea empezar.
```

De hecho, es lo mismo que:

```
>> respuesta := Nulo empezar.
```

Debido a que el objeto **Nulo** no reconoce el mensaje **empezar**, se devolverá a sí mismo como respuesta, dejando la respuesta nuevamente como **Nulo**. Por lo tanto, no es necesario tener una tarea vacía. Debido al elegante diseño del lenguaje de programación Citrine, puede simplemente utilizar el objeto Nulo para esto.

3.7 El objeto raíz

El objeto llamado **Objeto** es el objeto raíz de todos los objetos en Citrine. Este objeto raíz responde a mensajes como: **hacer, hecho, en-caso-de:hacer: , en:hacer: , mensaje:argumentos: , responder:y: … , aprender:significa:** , y **recursivo**. Todos los demás objetos se derivan de este objeto, incluido el objeto Nulo. El mensaje que se envía con más frecuencia es **en:hacer:** , que amplía las funcionalidades de un objeto. Este mensaje es recibido por el objeto raíz, que en consecuencia vincula la tarea especificada al mensaje y, al hacerlo, expande el objeto derivado. Este método ya se mencionó en el capítulo 2, sin embargo, para completar, observe otro ejemplo.
En la siguiente ilustración, se define un objeto circular con la intención de calcular su área en función de
su radio:

```
>> circulo := Objeto nuevo.
circulo en: ['radio:'] hacer: { :r
   mi radio := r.
}.
círculo en: ['área'] hacer: {
   <- mi radio * 2 * 3,14.
}.
```

Se añaden dos mensajes: **radio:**, que se utiliza para establecer el radio del círculo, y **area**, que se utiliza para calcular su area. Este objeto se puede utilizar de la siguiente manera:

```
circulo radio: 4, área.
```

El resultado será **25,12** (tenga en cuenta que se utiliza una aproximación muy aproximada de Pi, es decir, 3,14, por lo tanto, el resultado no es muy preciso). Este objeto también puede basarse en otro objeto, lo que significa que un objeto definido previamente se expande enviando mensajes **en:hacer:**. De esta manera, por ejemplo, un objeto Usuario puede basarse en un objeto Persona al que se le puede agregar un nombre de inicio de sesión:

```
>> Usuario := Persona nuevo.
Usuario en: ['login:'] hacer: { :nombre
 mi login := nombre.
}.
```

Además, los objetos del sistema pueden expandirse, como el objeto **Texto** con un cifrado ROT-13. Esta es una manera fácil de cifrar textos, basada en la técnica de cifrado Cifrado César (que es muy débil para propósitos más serios). El principio detrás de ROT-13 es bastante simple. Cada carácter se desplaza 13 posiciones en el alfabeto. En caso de que un carácter pase el carácter **z**, el conteo continúa desde **a** en adelante. Por lo tanto, a se convierte en **n**, **b** se convierte en **o** y x se convierte en **k**. Por ejemplo, cuando el texto **Londres** se cifra con ROT-13, mostrará **Lbaqba**.

Si se necesita agregar una funcionalidad de este tipo a cada texto, la manera más fácil de hacerlo es expandir el objeto Texto en sí (ya que cada texto deriva de este mismo objeto) utilizando el mensaje **en:hacer:**

```
Texto en: ['rot13'] hacer: {
    >> alfabeto :=
      ['abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz'].
    >> código := [''].
    yo caracteres cada: { :i :carácter
          >> índice := alfabeto encontrar: carácter.
          >> rot13 := índice + 13.
          >> codificado := alfabeto carácter: rot13.
          código añadir: codificado.
     }.
     <- código.
}.
```

A partir de ahora, cada texto se puede cifrar de la siguiente manera:

```
Salida escribir: ['hola'] rot13, detener.
```

Un detalle curioso de ROT-13 es que cuando se implementa el cifrado por segunda vez, el texto original vuelve a aparecer. De esta manera, el texto cifrado vuelve a ser descifrado:

```
Salida escribir: ['hello'] rot13 rot13, detener.
```

```
hello
```

Esto demuestra que el mensaje **en:hacer:** es uno de los principios más fundamentales del lenguaje de programación Citrine. Nótese que sin este mensaje, que es parte del objeto raíz Objeto , no sería posible ampliar las funcionalidades de los objetos.

Un segundo mensaje esencial es **es-igual-a:**, que devuelve **Verdadero** en el caso de dos variables que apuntan hacia el mismo objeto, como:

```
>> x := 1.
>> y := x.
Salida escribir : (x es-igual-a: y), detener.
```

```
True
```

Observe la sutil diferencia entre = y **es-igual-a:** . El primero mencionado compara el valor del contenido de dos objetos, mientras que el segundo mencionado examina la memoria de la computadora para ver si son, de hecho, el mismo objeto.

El mensaje **en-caso-de:hacer:** permite ejecutar una tarea condicional, al igual que los mensajes **verdadero:** y **falso:** . La diferencia es que puede adjuntar una tarea por valor. El siguiente ejemplo, por ejemplo, muestra el valor en la variable **x**. Si el valor de **x** es igual a **1**, se ejecuta el primer bloque de código. Si **x=2**, se ejecutará la segunda tarea, y así sucesivamente. De esta manera, selecciona el código correcto para el valor **x:**

```
>> x := 2.
x
en-caso-de: 1 hacer: {
  Salida escribir: ['Status: denegado'].
},
en-caso-de: 2 hacer: {
  Salida escribir: ['Status: pendiente'].
},
```

```
en-caso-de: 3 hacer: {
  Salida escribir: ['Status: aprovado'].
}.
Salida detener.
```

```
Status: pending
```

3.8 Series

Citrine conoce dos tipos de colecciones: series y listas. Las **series** son enumeraciones de objetos en un orden fijo. Las listas no tienen orden, sino que se parecen a una leyenda que incluye una clave (o término) y su valor correspondiente. Las series son comparables a las matrices o arreglos (PHP, Java, C) y las listas (Python). Las listas son comparables a las matrices asociativas o arreglos asociativos (PHP) o a los diccionarios (Python) en otros lenguajes de programación. Para crear una nueva serie, escriba:

```
>> fibonacci := Serie nuevo.
```

Esta secuencia vacía se puede rellenar con **adjuntar:**

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

Si escribimos la secuencia en la pantalla:

```
Salida escribir: fibonacci, detener.
```

Veremos:

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

Este resultado revela que también se puede utilizar esta notación más corta para declarar una serie:

```
>> fibonacci := Serie ← 0 ; 1 ; 1 ; 2 ; 3 ; 5 ; 8 ; 13.
Salida escribir: fibonacci, detener.
```

Este programa muestra el mismo resultado. Por supuesto, esta notación es mucho más corta y, por lo tanto, mucho más
cómoda

La flecha es un mensaje binario que crea una nueva serie y al mismo tiempo envía el objeto subsiguiente para que se incruste instantáneamente en esta nueva serie. Los puntos y coma (;) son todos mensajes binarios que colocan un valor en la serie.

Para averiguar la cantidad de objetos de una serie, use **contar**

```
Salida escribir: fibonacci contar.
```

```
8
```

En lugar de utilizar el punto y coma, se puede utilizar un símbolo de viñeta para añadir algo a la serie. De esta manera, la lista se parece bastante a una de texto normal:

```
>> platos := Serie nuevo
~ ['Achicoria']
~ ['Coles de Bruselas']
~ ['Chucrut'].
```

Las series responden a mensajes como: **adjuntar: , mínimo , máximo , cada: , anteponer: , unirse: , posición: , primero , último, penúltimo, poner:en: , toma-el-último , toma-el-primero , contar , desde:longitud: , reemplazar:longitud:con:: , por: , copiar , ordenar: , llenar:con: , y encontrar:**

Otra forma de crear una serie es transformar una palabra en una serie de caracteres (a, b, c) o
dividir (es decir, dividir) un texto en una serie de palabras:

```
Salida escribir: (['abc'] caracteres), detener.
```

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

```
Salida escribir: (
   ['Chicory,Brussels sprouts,Sauerkraut']
   dividir: [',']
), detener.
```

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

Por ejemplo, si el mensaje **caracteres** se envía a un texto, la respuesta será una serie que contiene todos los caracteres separados de los que está hecho el texto. Cuando el mensaje **dividir:** se envía a un texto con otro texto como argumento, el pequeño fragmento de texto adjunto se utiliza para desglosar el texto recibido.

El texto en fragmentos de texto separados. La respuesta será una serie de estos objetos de texto. A su vez, también es posible convertir una serie en un texto enviando el mensaje **unirse:** , vea el siguiente ejemplo:

```
>> platos := Serie ←
['Achicoria'] ;
['Coles de Bruselas'] ;
['Chucrut'].
Salida escribir: (platos unirse: [' ~or~ ']), detener.
```

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

En este caso, el fragmento de texto adjunto se utiliza para unir los textos de la serie.

Cada objeto dentro de una serie tiene su propio lugar, que se denomina posición. El primer objeto obtiene la posición 1, el segundo la posición 2, y así sucesivamente. Al utilizar el mensaje **posición:** (o la versión más corta **?**) es posible recuperar el elemento en la posición especificada:

```
Salida escribir: (platos posición: 3), detener.
Salida escribir: platos ? 1.
Salida escribir: platos ? 9.
```

Resultados en :

```
Sauerkraut
Chicory
None
```

En caso de que no haya nada en la posición solicitada, la respuesta será un objeto Nulo. Hay tres posiciones especiales en una serie, que son sus valores: el **primero**, el **último** y el **penúltimo** elemento.

Los objetos en esas posiciones se pueden recuperar utilizando los mensajes respectivos.

```
>> abc := ['ABC'] caracteres.
Salida escribir: abc primero, detener.
Salida escribir: abc último, detener.
```

```
Salida escribir: abc penúltimo, detener.
```

Resultados en

```
A
C
B
```

Para reemplazar un objeto en una posición específica dentro de la secuencia misma, escriba: **poner:en:**

```
abc poner: ['2nd'] en: 2.
```

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

De manera similar, se puede eliminar un objeto de la serie utilizando el símbolo menos:

```
>> x := Secuencia ← 1 ; 2; 3.
Salida escribir: x, stop.
Salida escribir: x - 2.
```

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

Con el mensaje **anteponer:** es posible insertar un objeto al inicio de la serie:

```
>> x := Serie ← ['Nut'] ; ['Monkey'].
x anteponer: ['Apple pie'].
Salida escribir: x.
```

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

También es posible generar una serie rellenando una serie vacía con el mensaje **llenar:con:** En caso de que se prefiera rellenar de antemano con ceros las puntuaciones máximas de un juego de ordenador,
escribir:

```
>> puntuaciones := Serie nuevo llenar: 10 con: 0.
Salida escribir: puntuaciones, detener.
```

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

También es posible copiar una serie:

```
>>copia := puntuaciones copiar.
```

Es posible quitar objetos de la serie cortando su inicio y final:

```
>> palabras := Serie ← ['Tarta de manzana'] ; ['Nuez'] ; ['Mono'].
>> original := palabras copiar.

>> a := palabras toma-el-primero.
>> b := palabras toma-el-último.

Salida escribir: original, detener.

Salida escribir: a, detener.
Salida escribir: b, detener.

Salida escribir: palabras, detener.
```

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

En el ejemplo anterior, al utilizar el mensaje **toma-el-primero** , se elimina el objeto inicial de la serie, que se coloca en consecuencia en **a**. Además, se elimina el último objeto utilizando el mensaje **toma-el-último**, que se coloca en **b**. Como resultado, es el objeto Texto del medio, “Nut”, el cual permanece. La copia original también se muestra, demostrando así la función de copia.

El mensaje **encontrar:** permite encontrar información dentro de una serie:

```
>> pinturas := Serie nuevo
~ ['Fighting Temeraire']
~ ['Lady of Shalott']
~ ['Hay Wain'].
```

```
Salida escribir: ( pinturas encontrar: ['Hay Wain']), detener.
Salida escribir: ( pinturas encontrar: ['Lady of Shalott']), detener.
Salida escribir: ( pinturas encontrar: ['Sunflowers']), detener.
```

```
3
2
None
```

Al utilizar el mensaje **cada:**, es posible aplicar una tarea a cada elemento de una serie:

```
>> metros := Serie ← 1200 ; 4000 ; 6210.
metros cada: { :numero :metros
   Salida escribir: metros / 0.3048 , detener.
}.
```

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

El ejemplo que se muestra arriba muestra una conversión de metros a pies. La tarea de conversión se aplica a cada elemento individual de la serie. En el primer parámetro (**:numero**), la tarea recibe la posición del elemento que se va a manipular. En el segundo parámetro (**:metros**), la tarea recibe el objeto que se coloca en la posición mencionada anteriormente. Siéntase libre de nombrar los parámetros usted mismo, sin embargo, el orden dentro de la tarea es fijo. El primer parámetro es siempre el **número de posición** y el segundo es el **objeto**.

Si se requiere el número más bajo, se puede enviar el mensaje **mínimo**. De hecho, si se requiere el número más alto dentro de la serie, se envía el mensaje **máximo**.La siguiente ilustración muestra una serie de puntajes de un juego. Se crea una nueva serie con dos elementos que contienen los extremos de los puntajes:

```
>> puntajes := Secuencia ← 100 ; 50 ; 200 ; 350.
>> extremos := Secuencia ← puntajes mínimo ; puntajes máximo.
Salida escribir: extremos, detener
```

Esto da como resultado:

```
Sequence ← 50 ; 350
```

En una serie es posible ejecutar un reemplazo utilizando el mensaje **reemplazar:longitud:con**

Por lo tanto, se puede utilizar para reemplazar un fragmento de serie por otra serie:

```
>> x := Serie ← 1 ; 2 ; 3 ; 4 ; 5.
>> z := Serie ← 9.
>> y := x reemplazar: 2 longitud: 1 con: z.
Salida escribir: y, detener.
```

```
Sequence ← 1; 9; 3; 4; 5
```

La ordenación es una acción útil que se aplica con frecuencia a las series. Dado que solo se ordenan las series, esta acción es única para este tipo de colección. Una serie se puede ordenar enviando el mensaje **ordenar:** a un objeto Serie, junto con un argumento que contiene una tarea de ordenación que incluye dos parámetros:

```
>> continentes := Serie nuevo
~ ['Oceania']
~ ['America']
~ ['Asia']
~ ['Europe']
~ ['Africa']
~ ['Antartica']
~ ['Australia'].
continentes ordenar: { :a :b <- a > b. }.
Salida escribir: continentes, detener.
```

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

La ordenación de mensajes: obtendrá de forma constante dos objetos de una serie y, en consecuencia, solicitará que la tarea compare los dos. La tarea de ordenación debe responder cada ronda con Verdadero o Falso.

En el fragmento ilustrado, los continentes se ordenan en orden alfabético comparando cada par de palabras con el mensaje >. Para ordenar en orden inverso, aplique el mensaje <.

3.9 Listas

Tanto Serie como Lista son colecciones. Sin embargo, a diferencia de una serie, **una lista no tiene orden**. Otra distinción entre las dos es que una lista consta de pares de objetos. Un objeto actúa como entrada (o palabra clave) para buscar el otro objeto (valor). Un buen ejemplo de una lista es una lista de precios, como:

```
>> menu := Lista nuevo
poner: ['£5'] en: ['tarta manzana'],
poner: ['£6'] en: ['tarta zanahoria'],
poner: ['£3'] en: ['algodón de azucar'].
```

Las listas pueden recibir los siguientes mensajes: **tipo, poner:en: , entradas, valores, en : , cada : , contiene: .**

De manera similar a una serie, el mensaje **poner:en:** se usa para agregar un objeto a una lista. La diferencia es que, a diferencia de una serie, se vinculan dos objetos. El primer objeto es, de forma similar a una secuencia, el objeto que almacenarás en una lista. El segundo objeto no es su posición dentro de la colección, sino la entrada (clave) que permite recuperar el objeto anterior en un momento posterior. En resumen, una lista funciona un poco como un diccionario; por lo tanto, al usar la palabra clave se puede encontrar el significado. Por ejemplo, para recuperar de la lista mencionada anteriormente el precio de un dulce de azúcar:

```
>> precio := menu en: [’algodón de azucar'].
```

En este caso, también puedes usar la notación concisa:

```
>> precio := menu ? [’algodón de azucar'].
```

La respuesta mostrará £3.

Si las **entradas** (o claves) que aplicas para almacenar objetos **no tienen espacios**, también se puede usar la siguiente notación:

```
>> menu := Lista nuevo
pastel: ['£5'],
tarta: ['£6'],
dulce: ['£3'].
```

Sin duda, la notación anterior se lee de forma más natural, porque se parece a una lista que se usa comúnmente en un documento. Para solicitar el precio de un dulce, se puede utilizar la siguiente notación simplificada:

```
>> precio := menu dulce.
```

De forma similar a una secuencia, la lista se puede imprimir en pantalla:

```
Salida escribir: menu, detener.
```

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

En caso de que se desee eliminar - pastel - del menú, se envía el mensaje binario "-", seguido de un objeto de texto que lleva el mismo nombre que el objeto de entrada del par que se desea eliminar:

```
menu - ['pastel'].
Salida escribir: menu, detener.
```

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

No existe la posibilidad de realizar búsquedas dentro de un objeto de lista, ya que una lista no conoce ningún orden. Esto significa que la función de búsqueda nunca puede determinar en qué posición se encuentra el resultado.

Sin embargo, es posible averiguar si un objeto en particular que contiene los criterios deseados estárealmente incluido, utilizando el mensaje que **contiene:**

```
>> receta := Lista nuevo
café: ['cappuccino'],
tarta: ['manzana'].
Salida escribir: (receta contiene: ['cappuccino']), detener.
Salida escribir: (receta contiene: ['cerveza']), detener.
```

Resultado:

```
True
False
```

El mensaje **tiene:** se puede utilizar como alternativa al mensaje **contiene:** y, como es un alias, su funcionalidad es la misma. Por lo tanto, siéntete libre de utilizar la palabra más adecuada en el contexto.

El mensaje **cada:** es igualmente comprendido por los objetos Lista. Por ejemplo, la siguiente lista ilustra la cantidad de horas trabajadas en un proyecto:

```
>> trabajo := Lista nuevo
pladur: 2,
pintar: 4,
empapelar: 6.
```

Para elaborar la factura, estas horas deben multiplicarse por el salario por hora (p. ej. 50 p.h.) y luego sumarse:

```
>> factura := 0.
trabajo cada: { :trabajo :horas
    factura añadir: horas * 50.
}.
Salida escribir: factura, detener.
```

Después de la ejecución de este programa, la factura será 600. También es posible generar una lista a partir de un par de series. Esto se puede hacer con el mensaje **por:** , como en el siguiente ejemplo:

```
>> lugares := Serie nuevo
~ ['Londres']
~ ['Belfast']
~ ['Edimburgo']
~ ['Cardiff'].
```

```
>> habitantes := Serie nuevo
~ 8.799.800
~ 345.418
~ 553.569
~ 359.512.
>> población := habitantes por: lugares.
Salida escribir: población, detener.
```

A continuación, la población de Cardiff se puede recuperar de la siguiente manera:

```
Salida escribir: población Cardiff, detener.
```

```
359,512
```

El ejemplo anterior muestra dos series que, imaginablemente, podrían obtenerse de un banco de datos. Ahora, para crear una lista con poblaciones mediante la cual se pueda recuperar el número de habitantes de cada ciudad, las dos series separadas deben compilarse en una lista. Esto se puede hacer utilizando el mensaje **por:** Cuando una serie recibe este mensaje, con como argumento otra serie, el resultado será una lista cuyas palabras clave forman la segunda lista (nombres de lugares) y la serie a la que se envió el mensaje mostrará los valores.

A su vez, es posible generar dos series a partir de una lista en particular. Al utilizar el mensaje **entradas:** , se recibe una serie de entradas de la lista receptora. El mensaje **valores:** proporciona una serie de valores. Esto significa que la lista **población** puede , a su vez, dividirse en dos series separadas de la siguiente manera:

```
>> nombres := población entradas.
>> números := población valores.
```

Las listas y las series se pueden combinar; Por ejemplo, poner una serie dentro de una serie o una serie dentro de una lista o viceversa. De hecho, esto es más la regla que la excepción. Tomemos como ejemplo, una lista de direcciones de un archivo de clientes. Teóricamente, podría verse así:

```
>> direcciones := Serie nuevo
~ ['1 Citrine Square GA2 3MO Citrineshire'],
~ ['4 Objects Road MA5 6VR Scopeford'].
```

Este tipo de direcciones son difíciles de procesar. ¿Qué hacer en caso de que sea necesario buscar, basándose en el nombre del lugar o el código postal? En ese caso, sería necesario examinar el texto para averiguar dónde termina el nombre de la calle y comienza el código postal. Por eso se prefieren las direcciones estructuradas, como:

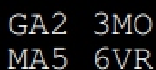
```
Lista nuevo
    calle: ['Citrine Square'],
    propiedad: 1,
    códigopostal: ['GA2 3MO'],
    localidad: ['Citrineshire'].
```

Asi es mucho más fácil. Si desea saber el nombre de la localidad de esta dirección, se envía el mensaje **localidad** a la lista. La respuesta será **Citrineshire.**

```
>> direcciones := Serie nuevo
   ~ (
      Lista nuevo
           calle: ['Citrine Square'],
           propiedad: 1,
           códigopostal: ['GA2 3MO'],
           localidad: ['Citrineshire']
     )
   ~ (
      Lista nuevo
            calle: ['Objects Road'],
            propiedad: 4
            códigopostal: ['MA5 6VR'],
            localidad: ['Scopeford']
     ).
```

Los datos estructurados como este son más adecuados para el procesamiento. De esta manera, es fácil leer todos los códigos postales:

```
direcciones cada: { :indice :dirección
     Salida escribir: dirección códigopostal, detener.
}.
```

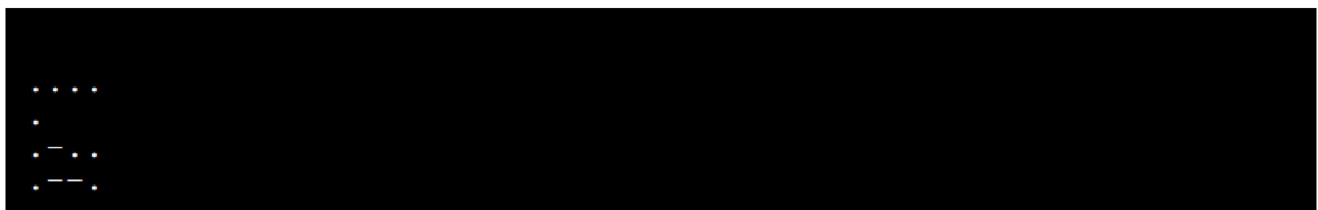


Así, sin datos estructurados, lo anterior habría sido mucho más complicado. Todas las combinaciones son posibles. Las listas se pueden poner en listas, las series en series, que, a su vez, se pueden llenar con nuevas series. Siéntete libre de ser creativo y colorido. Ahora, eche un vistazo a una aplicación práctica del objeto Lista. Los objetos Lista, por ejemplo, son excelentes para crear traducciones uno a uno. El siguiente programa traduce una sentencia a código Morse. En este ejemplo, se utilizan tanto una lista como una serie.

El programa utiliza dos colecciones. La primera consta de caracteres que forman la sentencia original. La segunda colección es una lista que funciona de forma similar a la piedra Rosetta y presenta el código Morse correcto para cada carácter individual.

```
>> sentencia := ['ayuda'].
>> morse := Lista nuevo
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: ['–..'].
sentencia caracteres cada: { :i :carácter
   >> código := morse ? carácter.
   Salida escribir: código, detener.
}.
```

Después de ejecutar el programa de código Morse, se presenta la siguiente salida:



Para traducir una palabra o frase diferente al código Morse, simplemente asígnelas a la variable **sentencia**. De hecho, el programa es bastante sencillo. Primero, hay una referencia, una lista, que guarda el código Morse correspondiente para cada carácter. Segundo, el mensaje **caracteres** se envía a **sentencia**, lo que da como resultado una **serie** que contiene cada carácter individual de la sentencia. Tercero, la iteración comienza enviando el mensaje **cada:** a esta serie mientras se adjunta una tarea para encontrar el código Morse requerido e imprimir el carácter resultante en la pantalla.

3.10 Archivos

El objeto Archivo responde a mensajes como: **ruta, leer, escribir: , existe, eliminar, tamaño y serie:.**
Citrine permite la edición esencial de archivos. Para ello, utilice el objeto de sistema Archivo.
En este ejemplo, un archivo de texto que contiene ['verduras olvidadas'] debe ordenarse alfabéticamente:
Contenido del archivo ***verduras*** en la carpeta *documentos:*

Chirivía
Alcachofa de Jerusalén
Rábano de invierno
Nabo

Para imprimir el contenido de este archivo en la pantalla, escriba lo siguiente:

```
>> fichero := Archivo nuevo: Ubicación-del-archivo documentos verduras.
Salida escribir: fichero leer , detener.
```

por ejemplo en Windows , la ubicación del archivo seria : [’C:\\documentos\\verduras.txt’]
Puede usar una extension o no para el fichero , en el ejemplo *verduras* es el nombre del archivo

Para ordenar el contenido y luego guardar la serie ordenada, escriba:

```
>> verduras :=
Archivo nuevo:
Ubicación-del-archivo documentos verduras ,
leer
dividir: ['↵'] ,
ordenar: { :a :b <- a > b. }.

Archivo nuevo:
Ubicación-del-archivo documentos nuevoficheroordenado ,
escribir: (verduras combinar: [’↵']).
```

Así, también en este ejemplo, se lee crea un archivo con (nuevo) despues se lee el contenido, y se divide en líneas mediante el mensaje **dividir:** , como argumento usamos el signo de **nueva línea (↵).** o tambien podemos usar el salto de linea tradiccional en otros lenguajes de programación que es (**\n**) Para continuar, se ordena según el método mencionado en capítulo 3. Finalmente, se guarda en el disco, después de haber combinado las líneas en un solo texto.

Además de leer y **escribir:**, también se pueden manejar otros asuntos. Mediante el mensaje **adjuntar:** , se puede agregar texto a un archivo, como en el siguiente ejemplo

```
Archivo nuevo:
Ubicación-del-archivo documentos nuevoficheroordenado ,
adjuntar : ['Zucchini'].
```

La ubicación del archivo se puede recuperar con el mensaje **ruta .**

```
>> location := fichero ruta.
```

Para saber si un archivo existe o no, se puede enviar el mensaje **existe** después de lo cual se devuelve un objeto **Verdadero** o **Falso**.

```
>> x := Archivo nuevo: Ubicación-del-archivo desconocido.
Salida escribir: x existe, detener.
```

```
No
```

Los archivos se pueden eliminar enviando el mensaje **eliminar .**

```
fichero borrar.
```

También se puede obtener el tamaño (en bytes):

```
fichero tamaño.
```

Para solicitar el contenido de una carpeta, se puede enviar el mensaje **serie** al objeto Archivo como argumento la ubicación de la carpeta. Por ejemplo, si se solicita el contenido de la carpeta documentos, se debe hacer:

```
ficheros := Archivo serie: Ubicación-del-Archivo documentos.
```

Como resultado, se devuelve una serie que contiene en cada posición una lista con las entradas ficheros y tipo que contienen el nombre del archivo y su tipo, en ese orden. El siguiente programa muestra el contenido de cada ubicación de archivo ingresada por el usuario:

```
>> ficheros := Archivo serie: ( Programa argumento: 3).

ficheros cada: { :número :detalles
     >> línea := ['#número nombre (tipo).']
            número: número,
            nombre: detalles archivo,
            tipo: detalles ? ['tipo'].
     Salida escribir: línea, detener.
}.
```

Cuando este programa se guarda como **lista**, se puede utilizar de la siguiente manera:

```
ctren list games

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

Las ubicaciones de los archivos se pueden introducir como texto:

/tmp/micarpeta

o como un objeto **Ubicación-del-archivo** :

```
Ubicación-del-archivo /tmp micarpeta .
```

Esta última notación es la preferida según el principio de uniformidad. Las ubicaciones de los archivos se convierten finalmente en objetos de texto, que indican el archivo de la ruta del sistema. Los objetos de **Ubicación-de-archivos** se benefician de la independencia del sistema:

```
Ubicación-del-archivo documentos templates spreadsheet1.
```

Se mostrará en sistemas con un signo separador de ruta / :

```
documents/templates/spreadsheet1
```

Y en sistemas que utilizan el signo separador de ruta \ :

```
documents\templates\spreadsheet1
```

3.11 Momentos

Para una representación estructurada de fecha y hora, Citrine utiliza el objeto Momento. En caso de que se imprima un nuevo objeto Momento en pantalla, se muestran la fecha y hora actuales:

```
Salida escribir: Momento nuevo, detener.
```

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

También es fácil leer los componentes de tiempo individuales. Así, por ejemplo, si solo es necesario conocer el año actual, se puede enviar el mensaje **año** al momento y, a su vez, la respuesta será un
objeto Número que contiene únicamente el año:

```
>> m := Momento nuevo.
Salida escribir: m año bruto, stop.
```

```
2020
```

Tenga en cuenta que el mensaje **bruto** debe enviarse al objeto Número resultante. Esto es para evitar la salida de **2,020** , que es, por supuesto, una notación correcta para números, sin embargo, no se usa normalmente en el contexto de notaciones de años. También es posible solicitar cuestiones que podrían ser menos obvias de determinar al principio, por ejemplo, el día de la semana (usando el mensaje **día-semanal**):

```
>> m := Momento nuevo.
Salida escribir: m día-semanal, detener.
```

```
3
```

Aquí, el número 3 representa el martes. Un día laborable siempre comienza el **1** (**domingo**), el lunes representa el segundo día, el martes el tercer día, y así sucesivamente. El **sábado** es siempre el **último** día de la semana, según Citrine.

De la misma manera, se puede recuperar el número de la semana **(semana**), así como los meses, días, horas, minutos y segundos. El siguiente ejemplo muestra cómo se crea un nuevo objeto Momento y cómo se pueden recuperar todos los componentes individuales de fecha y hora: el **segundo**, el **minuto**, la **hora,** el **día**,
el **mes**, el día del año **(dia-del-año**), el día de la semana (**día-semana**l) y la zona horaria (**zona**).

```
>> ahora := Momento nuevo.
Salida escribir: ahora, detener.
Salida escribir: ahora segundo, detener.
Salida escribir: ahora minuto, detener.
Salida escribir: ahora hora, detener.
Salida escribir: ahora día, detener.
Salida escribir: ahora dia-del-año, detener.
Salida escribir: ahora día-semanal, detener.
Salida escribir: ahora mes, detener.
Salida escribir: ahora zona, detener.
```

la salida:

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

Para cambiar uno de los componentes de tiempo, se envía un objeto Número. Por lo tanto, para modificar el segundero a 13, se hace:

```
>> m := Momento nuevo.
m segundo: 13.
Salida escribir: m, detener.
```

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

También es posible configurar otros componentes de tiempo usando **año: , mes: , día: , hora: , minuto: , y segundo:**

```
>> cuando := Momento nuevo
año: 2020,
mes: 12,
día: 31,
hora: 23,
minuto: 59,
segundo: 59.
Salida escribir: cuando, detener.
```

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

No es posible configurar ni cambiar el día de la semana ni el día del año.
Tenga en cuenta que los ajustes deben ser de mayor a menor; de hecho, configurar un componente mayor borra los componentes menores. El siguiente ejemplo ilustra este principio:

```
>> cuando := Momento nuevo
año: 2020,
mes: 12,
día: 31,
hora: 23,
minuto: 59,
segundo: 59,
año: 2021.
Salida escribir: cuando, detener.
```

Esto mostrará::

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

En este caso, los componentes de tiempo más pequeños vuelven a sus valores iniciales, porque en el último momento se establece el año.

Si selecciona un número demasiado alto para un componente de tiempo, el reloj seguirá contando de la manera habitual:

```
>> entonces := Momento nuevo
año: 2020,
mes: 2,
día: 30.
Salida escribir: entonces, detener.
```

El resultado será::
01/03/2020 00:00

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

En febrero nunca hay más de 29 días, por lo que el calendario cuenta desde el 1 de marzo. En este caso,2020 fue un año bisiesto.

Si desea utilizar una notación diferente para la visualización del tiempo, simplemente combine los componentes del tiempo libremente:

```
>> pantalla := ['día mes año'].
>> fecha := Momento nuevo.
pantalla
año: fecha año bruto,
mes: fecha mes,
día: fecha día.
Salida escribir: display, detener.
```

```
6 11 2023
```

No olvides tener en cuenta las distintas zonas horarias para evitar sorpresas. La zona horaria (**zona:**) en la variante británica de Citrine se mostrará como:

```
>> m := Momento nuevo.
Salida escribir: m zona, detener.
```

```
Europe/London
```

Al enviar el mensaje **tiempo** obtendrás la cantidad de segundos que han pasado desde el 1 de enero de 1970. Este es un punto de referencia en el tiempo a partir del cual la mayoría de las computadoras calculan la hora del sistema.

```
>> ahora := Momento nuevo tiempo.
Salida escribir: ahora, detener.
```

```
1.595.961.187
```

También puedes crear un nuevo Momento basado en una marca de tiempo como esta:

```
>> ahora := Momento nuevo: 0.
Salida escribir: ahora año bruto, detener.
```

```
1970
```

Estas marcas de tiempo pueden ser útiles para compartir datos de tiempo con otros sistemas. También se pueden utilizar para crear una descripción elegante del tiempo, algo que se desea comúnmente en varios programas:

```
Momento en: ['describe'] hacer: {
     >> ahora := Momento nuevo.
     >> descripción := ['-'].
     >> diferencia := ahora tiempo - yo tiempo.
     {
        (diferencia < 60) verdadero: {
              descripción := ['ahora mismo'].
      }, salir.
      (diferencia < 3600) verdadero: {
             >> minutos := (diferencia / 60) redondo.
             descripción := ['hace algunos minutos'] algunos: minutos.
      }, salir.
      descripción := (
           ['día-mes-año']
           día: (ahora día),
           mes: (ahora mes),
           año: (ahora año bruto)
      ).
```

```
} proceder.
<- descripción.
}.
>> m := Momento nuevo.
Salida escribir: m describe, detener.
>> m := Momento nuevo: (Momento nuevo tiempo - 1,500).
Salida escribir: m describe, detener.
>> m := Momento nuevo: (Momento nuevo tiempo - 5,000).
Salida escribir: m describe, detener.
```

Resultado:

```
right now
25 minutes ago
6-11-2023
```

Tenga en cuenta que en el ejemplo anterior, todo se coloca entre corchetes {}, seguido del mensaje **proceder**. Esta es una optimización para acelerar el programa. Al colocar todo en un bloque de proceso, se habilita el uso del mensaje **salir** después de cada condicional, para evitar que también se procesen condicionales excesivos.

Para calcular los tiempos, simplemente sume cada componente de tiempo por separado. Por ejemplo, en caso de que desee saber qué hora será dentro de una hora:

```
>> m := Momento nuevo.
Salida escribir: m, detener.
m añadir: 1 hora.
Salida escribir: m, detener.
```

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

Vea cómo se aplica aquí un truco para mejorar la legibilidad: **añadir: 1 hora**. De hecho, el mensaje **hora** se envía a 1, que es un calificador (explicado en el capítulo 3).

De la misma manera, puede agregar, por ejemplo, 100 segundos. No es necesario concentrarse en la conversión a las unidades correctas, ya que el objeto Momento lo hará por usted (usando el calificador).

Ahora, observemos un ejemplo más complicado sobre el uso de add: y subtract::

```
>> entonces := Momento nuevo
año: 2020,
mes: 2,
día: 29,
hora: 23,
minuto: 59,
segundo: 59,
restar: 1 segundo,
añadir: 2 segundo,
restar: 9000 hora.
Salida escribir: entonces, detener.
```

El resultado: :

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

Para crear una copia de un momento, haga lo siguiente:

```
>> entonces := Momento nuevo.
>> ahora := entonces copiar.
Salida escribir: entonces = ahora, detener.
Salida escribir: (entonces es-igual-a: ahora), detener.
```

Resultado:

```
True
False
```

En el ejemplo anterior, se crea una copia del momento **entonces.** Ambos momentos describen el mismo punto en el tiempo, por lo que = da como resultado **Verdadero**. Sin embargo, tenga en cuenta que sigue siendo solo una copia, no el original. Debido a que físicamente es, de hecho, un objeto diferente, en una ubicación diferente en la memoria de la computadora, el mensaje **es-igual-a:** solo puede dar como resultado la respuesta **Falso**.

Una característica adicional del objeto Momento es detener momentáneamente el proceso. Puede pausar la ejecución del programa utilizando el mensaje **esperar:** seguido del número de segundos. Eche un vistazo al siguiente ejemplo para ver una breve ilustración de este principio:

```
>> antes := Momento nuevo.
Momento esperar: 2.
```

```
>> después := Momento nuevo.
Salida escribir: antes, detener.
Salida escribir: despúes, detener.
```

Salida:

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

Después de 2 segundos, el fragmento de programa anterior mostrará el resultado de dos momentos: el momento grabado antes de la pausa y el momento grabado después de esa pausa. Como resultado, las descripciones de tiempo en su pantalla deberían mostrar una diferencia de 2 segundos.

3.12 El objeto Programa

En este capítulo se describe el objeto Programa, y este es el último objeto que se revisará en la serie de objetos básicos del sistema. La función básica del objeto Programa es comunicarse con el mundo exterior, de manera similar al objeto Archivo y al objeto Salida. Estos tres representan la puerta de entrada al mundo exterior.

Además, este objeto contiene una serie de funciones que benefician la administración avanzada de la memoria. También puede cargar otros programas de Citrine utilizando el objeto Programa. Para combinar varios programas de Citrine, puede enviar el mensaje **utilizar:** . En este ejemplo, hay dos archivos de Citrine, que se denominan herramientas y programa:

Contenido de herramientas:

```
>> Herramienta := Objeto nuevo.
```

Contenido del programa:

```
Programa utilizar: Ubicación-del-Archivo heramientas.
Salida escribir: Heramienta tipo, detener.
```

```
ctren program
```

Al ejecutar el programa Citrine titulado **programa**, se cargará el código del archivo **heramientas** gracias al mensaje **utilizar:**

```
Object
```

De esta manera, puede incluir el trabajo de otras personas en su propio programa y hacer uso de la funcionalidad proporcionada por bibliotecas externas.
Durante la ejecución del código incluido, el directorio de trabajo se cambiará a su carpeta principal.

Puede ejecutar comandos del sistema desde Citrine, enviando el mensaje **sistema:** a **Programa**. Esto le permite, por ejemplo, copiar archivos, reubicarlos o invocar otro s oftware para ejecutar tareas específicas. En un sistema Linux, por ejemplo, puede aplicar el siguiente paso para mostrar los archivos en la carpeta actual:

```
Salida escribir: (Programa sistema: Instrucción ls).
```

El código mencionado anteriormente ejecuta el comando **ls** desde el shell del sistema y devuelve el resultado en formato de texto. El objeto **Instrucción** es una herramienta sencilla que se puede utilizar en lugar de un objeto Texto. Puede especificar comandos para **Programa**, ya sea como texto o como un objeto **Instrucción**. Las siguientes líneas son idénticas:

```
Salida escribir: Instrucción Hola Mundo, detener.
Salida escribir: ['Hola Mundo], detener.
```

Y mostrarán exactamente el mismo resultado:

```
Hello World
Hello World
```

En cuanto al principio de uniformidad, que pretende mantener el código Citrine lo más puro posible, se prefiere el uso de la variante Instrucción. Naturalmente, los comandos que se especifican mediante el objeto **Instrucción** o que se envían como texto a la línea de comandos dependen del sistema. En resumen, los comandos que debe utilizar dependen del sistema operativo en cuestión. El mensaje de la línea de comandos sirve simplemente como intermediario.

La entrada estándar (*stdin*, según su sistema) se puede leer enviando la entrada del mensaje al objeto **Programa**. A cambio, **Programa** le proporcionará un objeto **Texto** que contiene la información transmitida. De esta manera, puede crear un pequeño programa para ordenar la entrada desde la línea de comandos:

```
Salida escribir: (
    Programa entrada
       dividir: [','],
       ordenar: { :a :b <- (a > b). },
       unirse: [',']
).
```

```
echo “c,a,b” | ctren test.ctr

a,b,c
```

Un programa también puede exportar datos a otros programas (a través de **stdout**). En general, esto se puede lograr simplemente mediante el objeto Salida, que normalmente se usa para escribir en la pantalla. Sin embargo, en caso de que el

La salida de su programa está conectada a la entrada predeterminada de un programa diferente, los datos que está escribiendo fluirán al canal de entrada (*stdin*) de ese otro programa (**Programa de entrada**).

Puede escribir textos en el canal de error (stderr) enviando el mensaje **error:** al **Programa**, por ejemplo:

```
Programa error: ['Ocurrió un error.'], detener.
```

Estas notificaciones suelen terminar en registros y se pueden usar para localizar errores del programa en un momento posterior.

Dependiendo de la configuración de su sistema, la salida de pantalla o la salida a otros programas no se ejecuta constantemente, sino en fragmentos, lo que se conoce como almacenamiento en búfer. Esto se hace por razones de eficiencia. Si prefiere forzar el vaciado de los búferes de salida del programa, puede hacerlo enviando el mensaje **limpiar**

```
Programa limpiar.
```

En caso de que su programa Citrine se inicie en la línea de comandos, puede leer los parámetros posibles utilizando el mensaje **argumento: X ,** reemplazando así X por la ubicación del parámetro que desea recuperar, como en:

```
Salida
escribir: (Programa argumento: 3),
escribir: (Programa argumento: 4), detener.
```

```
ctren test.ctr hello world

helloworld
```

Preste atención al orden de conteo de los argumentos:
Argumento n.° 1: ctren (Citrine)
Argumento n.° 2: test.ctr (su programa)
Argumento n.° 3: el primer parámetro que sigue a su programa (hola)
Argumento n.° 4: el segundo parámetro que sigue a su programa (mundo)
De la misma manera, puede recuperar la cantidad de argumentos con **argumentos**

```
Salida escribir: Programa argumentos, detener.
```

En el caso del fragmento mencionado anteriormente, el resultado será **4.**
Además, puede recuperar las llamadas variables de entorno, por ejemplo:

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

En su programa, la configuración **proverb** se puede recuperar con el mensaje **establecer:**

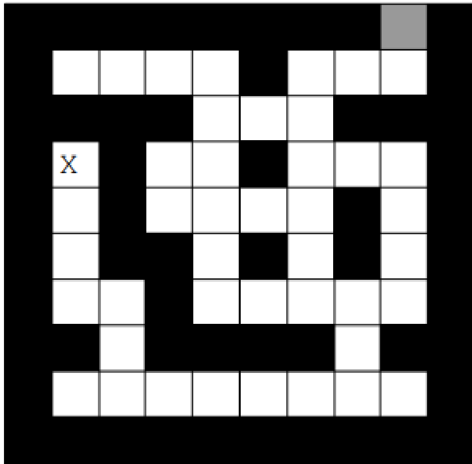
```
Salida escribir: (Programa establecer: ['proverb']), detener.
```

También puede configurar las variables de entorno usted mismo utilizando **establecer:valor:**

```
Programa
establecer: ['programa_estado']
valor: ['empezado'].
```

Al enviar el mensaje **pedir** al Programa, el programa esperará la respuesta del usuario. El siguiente conciso juego de texto ilustrará este mismo principio.
En este juego, el jugador navega por un castillo encantado para encontrar un tesoro. Un juego de texto es ideal para poner en práctica los conocimientos adquiridos. No tiene sistemas gráficos, lo que hace que el programa sea bastante independiente y bastante atemporal, y evita la complejidad innecesaria. El funcionamiento de un juego de texto es simple, ya que el programa refleja la situación actual a la que el usuario responde utilizando un comando de texto.

En el siguiente ejemplo se programa un castillo encantado que contiene un tesoro. Primero, se crea una cuadrícula simple para representar los pasillos del castillo. En este ejemplo, se dibuja una cuadrícula de 10 x 10 y la ilustración siguiente muestra un posible concepto. Los bordes exteriores representan el recinto, con la puerta de entrada en la parte superior derecha y con una **X** que marca el lugar donde se esconde el tesoro.



Para poder transformar esta cuadrícula en un juego, es imprescindible poder guardar una representación de la misma en la memoria del ordenador. Para ello, utilizamos un espacio para representar un pasillo, un cubo para representar una pared y la letra **X** para representar el tesoro.

El código quedaría así:

```
>> map := ██████████ +
          █    █   █ +
          ████   ███ +
          █X█  █   █ +
          █ █    █ █ +
          █ ██ █ █ █ +
          █  █     █ +
          ██ ████ ██ +
          █        █ +
          ██████████.
```

Aquí, el mensaje binario + se utiliza para distribuir el rango de caracteres en varias líneas con el fin de hacerlo visualmente atractivo.

Ahora que el mapa del castillo ha sido codificado como un objeto de texto, puedes decidir desde dónde comienza el jugador. El jugador, en este caso, comenzará en el lado superior derecho, o más específicamente:
en el noveno cuadrado de la segunda línea. Dado que cada línea está compuesta por diez cuadrados y todas las líneas juntas representan una secuencia larga, la ubicación correcta se puede encontrar con:

1 * 10 + 9 = 19

Para empezar en la segunda línea, es necesario saltar sobre los primeros diez cuadrados de la primera línea.A continuación, la ubicación de inicio correcta está, de hecho, nueve cuadrados más adelante. Por lo tanto, para capturar la ubicación de inicio del jugador, haga lo siguiente:

```
>> ubicación := 19.
```

El resto del juego consta de dos tareas que están conectadas por un mensaje **mientras:** De hecho, la esencia del juego es:

```
{ buscar… } mientras: { tesoro no encontrado }.
```

Ahora, echemos un vistazo a la segunda tarea primero, ya que es la más sencilla. Suponiendo que la variable **ubicación** indica la ubicación actual del jugador dentro del castillo, la ubicación del tesoro se puede verificar de la siguiente manera:

```
(mapa carácter: ubicación) !=: ['X']
```

La búsqueda continuará mientras no haya ningún tesoro en la ubicación del jugador. La búsqueda consiste básicamente en preguntar en qué dirección le gusta ir al jugador. A continuación, se comprueba si la dirección preferida es realmente válida, para evitar que los jugadores se choquen contra una pared del castillo. Si la dirección elegida es realmente posible, será necesario ajustar la ubicación del jugador. El resultado final será el siguiente:

```
Programa pedir
en-caso-de: ['n'] hacer: { x := -10. },
en-caso-de: ['s'] hacer: { x := 10. },
en-caso-de: ['e'] hacer: { x := 1. },
en-caso-de: ['w'] hacer: { x := -1. }.
```

Mientras no se haya encontrado el tesoro, se imprime un texto en el que se le pide al jugador que elija una dirección. La dirección real se solicita mediante el mensaje **pedir** a **Programa**, que devolverá un objeto Texto a este objeto Texto se le envía una cadena de mensajes **en-caso-de:hacer**: , mediante los cuales se vincula una tarea a una posible respuesta del jugador. Por ejemplo, si un jugador responde con n (norte), entonces x debería ser igual a -10. En este caso, x es igual al movimiento previsto. Por lo tanto, x = -1 significa 1 posición a la izquierda. Si x = -10, entonces esto significa 10 posiciones a la izquierda, que en realidad es 1 posición hacia arriba. En consecuencia, se calcula si realmente hay un paso según el mapa. Esto se ilustra en el siguiente ejemplo:

```
>> pasaje := mapa carácter: ubicación + x.
```

x se añade a la ubicación actual y se pregunta al mapa qué carácter está escrito en esta posición. En caso de que haya una pared, el jugador no puede avanzar y la pregunta tiene que repetirse sin actualizar la ubicación actual del jugador. Si no, el jugador puede moverse.

```
>> pasaje !=: ['█'] verdadero: {
   ubicación añadir: x.
}, else: {
   Salida escribir: ['¡Ay! ¡Te has topado con una pared!'],detener.
}.
```

Todo el juego de ordenador se ve así:

```
>> mapa :=  ['██████████'] +
            ['█    █   █'] +
            ['████   ███'] +
            ['█X█  █   █'] +
            ['█ █    █ █'] +
            ['█ ██ █ █ █'] +
            ['█  █     █'] +
            ['██ ████ ██'] +
            ['█        █'] +
            ['██████████'].
>> ubicación := 19.
{
   Salida escribir: ['que dirección ?'], detener.
   >> x := 0.
   Programa pedir
   en-caso-de: ['n'] hacer: { x := -10. },
   en-caso-de: ['s'] hacer: { x := 10. },
   en-caso-de: ['e'] hacer: { x := 1. },
   en-caso-de: ['w'] hacer: { x := -1. }.
   >> pasaje := mapa carácter: ubicación + x.
   >> pasaje  !=:  ['█'] verdadero: {
        ubicación añadir: x.
    }, else: {
        Salida escribir: ['Oh! has chocado con una pared!'],detener.
    }.
} mientras: { <- (mapa carácter: ubicación)   ['X']. }.
```

El comienzo de una sesión de juego podría verse así…

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

En el ejemplo anterior, el mensaje **pedir** se utiliza para hacer una pregunta al jugador. Además, la entrada del jugador se imprime en la pantalla. En algunas situaciones, esto no es deseable, por ejemplo, en relación con las contraseñas.
En ese caso, puede enviar el mensaje **pedir-contraseña** al objeto Programa. En consecuencia, el objeto Programa solicitará la entrada del usuario sin imprimir los resultados de las teclas presionadas en la pantalla, por ejemplo:

```
Salida escribir: ['Ingrese su contraseña … '], detener.
>> contraseña := Programa pedir-contraseña.
Salida escribir: ['Repite:'], detener.
>> control := Programa pedir-contraseña.
(contraseña = control) verdadero: {
Salida escribir: ['¡Las contraseñas coinciden!'], detener.
}, otro: {
Salida escribir: ['¡Las contraseñas no coinciden!'], detener.
}.
```

Un ejemplo de una posible salida:

```
Enter your password …
Repeat:

Passwords do not match!
```

Para finalizar el programa antes de tiempo, puede utilizar el mensaje **finalizar.**

```
{ :i
Salida escribir: i.
i = 5 verdadero: { Programa finalizar. }.
} * 10.
1234
```

```
12345
```

Por último, hay una serie de mensajes relacionados con la memoria del sistema. Puede recuperar información sobre el uso de la memoria con el mensaje **memoria:**

```
Salida escribir: Programa memoria, detener.
```

Posible salida (ilustrativa):

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

Aquí, el primer número indica la memoria en bytes disponibles que el programa puede utilizar. El segundo número indica la cantidad de objetos ubicados en la memoria. El tercer número indica los objetos que aún no se pueden borrar; es decir, los objetos que están atascados. El cuarto número indica la cantidad de objetos que durante la última limpieza no se pudieron borrar para liberar memoria. El último número indica la cantidad de objetos que se borraron durante la última limpieza (los desechos). Tenga en cuenta que esta información depende del sistema, ya que funciona en bytes, lo que es bastante raro para Citrine.

Al iniciar un programa, se puede establecer la cantidad de memoria disponible preferida. En el momento actual de escribir este manual, el estándar es de 10 MB. La cantidad de memoria que su programa puede utilizar se
puede establecer con **memoria:**

```
Programa memoria: 40 MB.
```

La cantidad de bytes se puede especificar con los calificadores MB o KB. Tenga en cuenta que también es posible establecer la memoria utilizando una variable de entorno (consulte el Apéndice C).

Además, es posible hacer que Citrine limpie en cualquier momento deseado utilizando el mensaje **limpiar-memoria**

```
Programa limpiar-memoria.
```

Tan pronto como envíe el mensaje, Citrine intentará limpiar todos los objetos no utilizados para aumentar el espacio de memoria disponible para su programa. También es posible indicarle a Citrine cómo realizar la gestión de memoria por usted. Esto se establece utilizando el mensaje **memoria gestionar**: , que está predeterminado en 1. Si no hace nada, su programa se configura de la siguiente manera:

```
Programa memoria gestionar: 1.
```

Consulte la tabla para ver las posibles configuraciones a continuación:

código	**significado**
0	Los objetos nunca se limpian. El programa se vuelve más rápido, pero la memoria se llena rápidamente
1	Cuando su programa utiliza casi el 80% de la memoria, Citrine intentará liberar espacio limpiando objetos.
4	Citrine se limpia continuamente. El programa se vuelve lento, pero el uso de memoria es mínimo.
8	Siempre que sea posible, la memoria se divide en bloques estándar con un tamaño fijo. Los bloques se reciclan, lo que es más rápido. Los objetos nunca se limpian.
9	Similar al escenario 8, sin embargo los bloques son realmente reciclados.
12	Combinación de los ajustes 4 y 8

3.13 Ejercicios

1. Escribe un programa que muestre las temperaturas de -10 a 40 grados Celsius en los grados Fahrenheit correspondientes (multiplica por 1,8 y luego suma 32).
Para ayudarte a empezar:

```
>> celsius := -10.
{
>> fahrenheit := ...
Salida escribir: fahrenheit, detener.
} mientras: { <- ... }.
```

2. Este programa expresa el dicho ['Golpea mientras el hierro está caliente']. Para ello se ajusta la plantilla tile. ¿Cuál era el texto original en el tile?

```
>> tile := ['...'].
tile comprar: ['strike'], barato: ['hot'].
Salida escribir: tile, detener.
```

3. Expande el objeto Número usando el mensaje square que devuelve el cuadrado de ese número. Para ayudarte a empezar:

```
Número en: ['...'] haz: { <- ... }.
```

4. Crea un nuevo objeto Distancia, que acepta dos puntos y puede calcular la distancia entre estos puntos para:

```
>> Punto := Objeto nuevo.
Punto en: ['coordenada-x:'] hacer: { :x mi x := x. }.
Punto en: ['coordenada-y:'] hacer: { :y mi y := y. }.
Punto en: ['coordenada-x'] hacer: { <- mi x. }.
Punto en: ['coordenada-y'] hacer: { <- mi y. }.
>> muelle := Punto nuevo coordenada-x: 5, coordenada-y: 6
```

```
>> ayuntamiento := Punto nueva coordenada x: 7, coordenada y: 1.
>> Distancia := Objeto nuevo.
...
>> x := Distancia nuevo
inicio: muelle
fin: ayuntamiento,
número.
Salida escribir: x, detener.
```

En caso de que la salida devuelva el número 7.

Pista: envía el mensaje **absoluto** para convertir un número negativo en uno positivo.

5. En el siguiente código, ¿cuál es el valor de x?

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

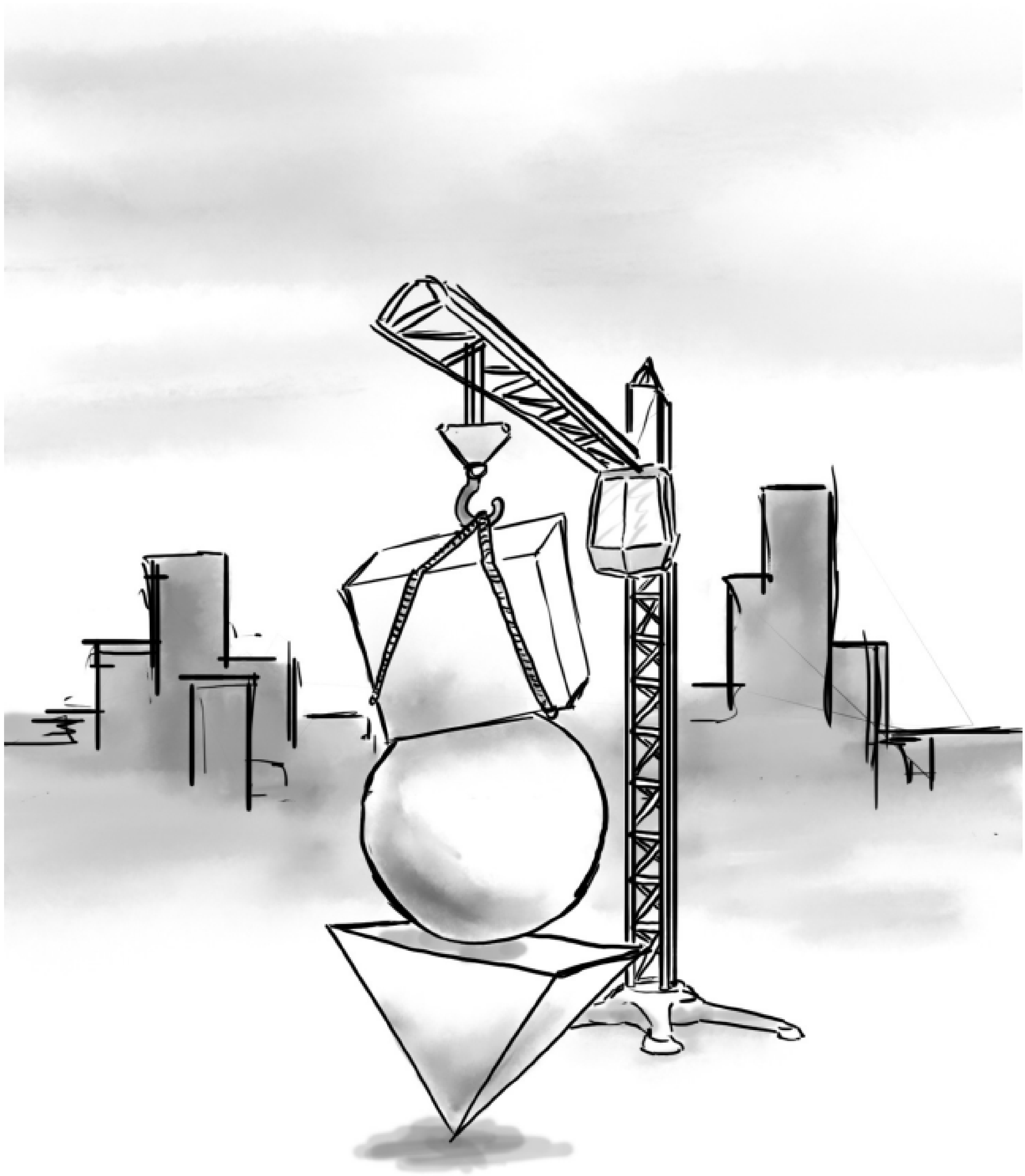
6. Convierte la siguiente tarea en un mensaje para Número:

```
>> Double := { :número <- número * 2. }.
>> x := Double aplicar: 4.
Salida escribir: x, detener.
```

7. Expanda el objeto **Número** con un mensaje dígitos:, con el que se puede mostrar un número que contenga una cantidad fija de dígitos (2 → 02, 5 → 005, etc.).

```
Salida escribir: (9 dígitos: 9), detener.
```

```
000000009
```



4. Avanzado

4.1 Copiar

Al usar := es posible guardar objetos con un nombre determinado (es decir, asignarlos a una variable). También es posible guardar un objeto con varios nombres. Tenga en cuenta que esta acción es diferente a la de realizar una copia real.

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

clone reemplazar: ['l'] con: ['n'].
Salida escribir: sheep, detener.
```

```
Donny
```

Aquí, podría haber esperado que la salida fuera Dolly, en lugar de Donny. Sin embargo, este no es el caso, ya que ambos nombres hacen referencia al mismo objeto. Cuando se trabaja con un bucle, ocurre algo similar:

```
>> puntos := Serie ← 1 ; 2 ; 3.
puntos cada: { :número :cantidad cantidad añadir: 1. }.
Salida escribir: puntos, detener.
```

De hecho, Citrine *siempre* usa referencias, por lo que en **:cantidad** el bucle ilustrado anteriormente también indica la
referencia al elemento en la secuencia:

```
Sequence ← 2 ; 3 ; 4
```

Para copiar un objeto, debes especificar esta acción explícitamente:

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

```
clone reemplazar: ['l'] con: ['n'].
Salida escribir: sheep, detener.
```

```
Dolly
```

Al enviar el mensaje unario **copiar** a un objeto Texto, el objeto devuelve una copia de ese mismo objeto Texto. Es posible copiar objetos Número, Booleano, Serie, Lista y Momento de la misma manera.

También puede definir su propia implementación de copia, esto es incluso una necesidad si crea sus propios objetos. Creemos una implementación de copia alternativa para series. La implementación predeterminada de copiar para una serie hace una copia superficial, crea una nueva lista con los mismos elementos.

```
>> a := Serie ← 1 ; 2 ; 3.
>> b := a copiar ; 4.
Salida escribir: a, detener escribir: b, detener.
```

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

En este caso, se agrega 4 solo a la copia.
Sin embargo, debido a que la copia es superficial, los objetos en ambas series son los mismos:

```
>> sheep := Serie ← ['Dolly'] ; ['Shaun'].
>> clones := sheep copiar.
clomes cada: { :número :sheep
sheep adjuntar: ['2'].
}.
Salida escribir: sheep, detener.
Salida escribir: clones, detener.
```

Entonces, si agregamos **2** a cada nombre en la copia, la serie original también se ve afectada.

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

Para solucionar esto, necesitamos hacer una copia profunda. Esta acción de copia para una secuencia podría estar compuesta de la siguiente manera:

```
Serie en: ['copiar'] hacer: {
  >> copiar := Serie nueva.
  yo cada: { :number :sección
   copiar adjuntar: sección recursivo copiar.
   }. <- copiar.
}.
```

Puedes usarla de esta manera:

```
>> oveja := Serie ← ['Dolly'] ; ['Shaun'].
>> clones := oveja copiar.
clones cada: { :número :sheep
   sheep adjuntar: ['2'].
}.
Salida escribir: oveja, detener.
Salida escribir: clones, detener.
```

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

Si elimina la acción de copia, el resultado serían dos impresiones de la segunda serie. A primera vista, parece que la adición simplemente copia la oveja en la serie principal y deja de lado las subseries. Sin embargo, si el código de llamada se ajusta a:

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

Entonces, a pesar de que **Shaun** está en una subserie, de hecho se copia correctamente. ¿Cómo funciona esto? Bueno, tiene todo que ver con las convenciones de nombres. Debido a que la acción de copia se ha conectado al mensaje llamado **copiar,** todos los objetos responderán con su respectivo comportamiento de copia, no importa si el objeto receptor es Texto o una (sub)Serie.

Además, tenga en cuenta que el mensaje es recursivo, este mensaje es necesario para enviar antes del mensaje de copia. Más
sobre este mensaje en el capítulo 4.4

Es esencial tener en cuenta que, aunque una copia de un objeto a menudo tiene la misma apariencia que el original, de hecho, nunca será el mismo. El objeto raíz define un mensaje **es-igual-a:** , que puede usarse para comparar la identidad de los objetos. Observa el siguiente ejemplo:

```
>> a := 1.
>> b := a copiar.
>> c := a.
Salida escribir: a = b, detener.
Salida escribir: a = c,  detener.
Salida escribir: ( a es-igual-a: b ),  detener.
Salida escribir: ( a es-igual-a: c ),  detener.
```

```
True
True
False
True
```

En este caso, a = b, porque a es una copia de b y el mensaje ['='] mira el **valor del objeto**,que en el caso anterior, es el número. Dado que a = 1 y b = 1 y 1 = 1, la respuesta será **Verdadero**. Lo mismo se aplica a c. Solo cuando se utiliza el mensaje **es-igual-a:** , que **Número** hereda de **Objeto**, se puede notar una diferencia. Mientras que c es igual a a, porque aquí la referencia se asigna utilizando el símbolo :=, a (**no es igual a**) b. Esto se explica por el hecho de que dentro de la computadora

La copia de la memoria es físicamente un objeto diferente. Por lo tanto, para averiguar si efectivamente la referencia es al mismo objeto físico en la memoria de la computadora, puede aplicar el mensaje **es-igual-a:**

4.2 Visibilidad

Como ya sabes, un objeto puede asignarse a una variable, pero primero debes asignar memoria en la memoria bajo este nombre, y para ello utilizas un símbolo de declaración. Esta acción se llama *declarar una variable*. Por supuesto, esto ya se ha tratado en los capítulos anteriores. Sin embargo, no todas las secciones de la memoria son visibles en todas partes, porque en programas grandes, que podrían llevar código de terceros, es probable que los nombres entren en conflicto. Por este motivo, estas ubicaciones de memoria están separadas entre sí, por lo que las tareas forman las líneas divisorias. Aquí se debe tener en cuenta que una variable que se declara dentro de una tarea, solo es visible durante la ejecución de esa tarea específica y durante la ejecución de todas las tareas iniciadas por esta tarea. Cuando una tarea en la que se declara la variable ha finalizado, se
olvidará de alguna manera. Observa el siguiente ejemplo:

```
>> x := 9.
{ Salida escribir: x. } empezar.
```

El número **9** se imprime en la pantalla con el programa que se muestra arriba. El nombre de objeto **x** se ha declarado fuera de la tarea y, en consecuencia, es visible en todas partes, para todas las tareas, incluso cuando **x** se envía como argumento al objeto Salida. El área fuera de la tarea se puede considerar como una especie de tarea paraguas.

Todas las variables que se declaran en esta área son visibles en todas partes en el programa, de hecho, todas las tareas están dentro de los límites de esta tarea paraguas, por así decirlo. Las variables que se declaran fuera de todas las tareas del programa también se conocen como **variables globales**, debido a su visibilidad global.

```
{ >> x := 9. } empezar.
Salida escribir: x.
```

En este caso, aparecerá un mensaje de *error*. El nombre de objeto **x** se olvida tan pronto como finaliza la tarea. Dado que **x** se ha declarado dentro de la tarea, no es visible fuera de ella. En este caso, **x** vive exclusivamente dentro de la pequeña tarea que se ha escrito. Ahora, observe el siguiente fragmento, no hay mensajes de error; Sin embargo, ¿qué número aparecerá en la pantalla?

```
>> x := 1.
{ >> x := 9. } empezar.
Salida escribir: x.
```

Aquí, la respuesta correcta es 1 en lugar de 9. En este caso, hay dos ubicaciones en la memoria con el nombre **x**. Gracias a los límites entre las tareas, no se afectan entre sí. Durante la

ejecución de la tarea, **x** es igual a **9**, y tan pronto como la tarea ha terminado, la **otra x** se vuelve visible de nuevo y **x** será igual a **1**.
Y ahora la versión más complicada:

```
>> y := { x := 2. }.
{ >> x := 1. y empezar. Salida escribir: x. } empezar.
```

¿Cuál aparecerá en pantalla: **1** o **2** ? Veamos. En primer lugar, se asigna un lugar en la memoria con el nombre y para una tarea. En esa tarea en particular, el número **2** se guarda con el nombre **x**.
Sin embargo, esta tarea no se inicia. En la segunda línea se crea una nueva tarea, que comienza de inmediato. En esta nueva tarea, se declara **x**. Por lo tanto, se asigna un lugar en la memoria con el nombre **x**. En consecuencia, con el nombre **x** se guarda el número **1**. En ese momento **x=1**. A continuación, se inicia la tarea llamada **y,** que se ha escrito en la primera línea de este programa. Durante la ejecución de esa tarea, el número **2** se guarda bajo el nombre **x**. ¿Es esto posible? La respuesta a esta pregunta es un **rotundo sí.** Aquí, la línea crucial está efectivamente **durante la ejecución**. Aunque la tarea y, desde un punto de vista visual, queda fuera de la tarea de la línea 2, la tarea y se está ejecutando de hecho durante la ejecución de la tarea en la segunda línea. Esto significa que, el **1** que estaba en **x**, se sobrescribe con el número **2**. De hecho, en ese momento **x = 2**. Entonces, la tarea y ha terminado y la ejecución del programa vuelve a la tarea en la segunda línea, lista para imprimir **x** en la pantalla.

Sin embargo, ¿qué hay dentro de **x** ahora que la tarea y ha terminado? Para aclarar, solo **x** se declaró fuera de **y**, es decir, en la tarea que se está procesando actualmente, la de la segunda línea, por lo que x todavía contiene el número **2** y, como resultado, se hará visible en la pantalla. Este mecanismo, como se acaba de describir, también se llama *Alcance dinámico*. A modo de comparación, en JavaScript un código similar da como resultado **1**:

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

```
1
```

En los ejemplos que se muestran arriba, se utilizan objetos Tareas separados cada vez. Esto mismo se puede ilustrar en base a otros objetos. El siguiente ejemplo ilustra cómo el sabor del helado se modifica consistentemente en malaga:

```
>> heladería := Objeto nuevo.
heladería en: ['helado'] hacer: {
sabor := ['Malaga'].
```

```
}.
Heladería en: ['visitar'] hacer: {
>> sabor := ['Pistachio'].
Heladería helado.
Salir escribir: sabor, detener.
}.
Heladería visitar.
```

En consecuencia, la salida del programa es:

```
Malaga
```

Tan pronto como la heladería recibe el mensaje helado, el sabor se ajusta a **Málaga.** Para las personas que nunca han trabajado con otro lenguaje de programación que no sea Citrine, esto no sería muy sorprendente. Después de todo, esto tiene todo el sentido según los procedimientos generales de Citrine. Sin embargo, en la mayoría de los otros lenguajes de programación, la salida sería **Pistachio,** o incluso devolverían un mensaje de error, dependiendo de las reglas de visibilidad.

4.3 Manejo de mensajes desconocidos

Si desea crear un objeto que muestre automáticamente en la pantalla todas las palabras que le envíe (que tengan varios caracteres) en líneas individuales, podría escribir algo como esto:

```
Salida Humpty Dumpty se sentó en la pared.
```

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

Puede lograr esto utilizando los llamados mensajes genéricos. Esta técnica le ayuda a conectar una tarea a mensajes indefinidos:

```
Salida en: ['responder:'] hacer: { :texto
    Salida escribir: texto, detener.
}.
```

La tarea que está conectada a **responder:** se ejecuta en cada mensaje desconocido. Dado que el mensaje **Humpty** es desconocido, Citrine ejecutará internamente el siguiente mensaje:

```
Salida responder: ['Humpty'].
```

Existen varias aplicaciones concebibles para los mensajes generales. Una de sus principales aplicaciones es en combinación con el principio de uniformidad; es decir, la traducción del software de Citrine a otros lenguajes técnicos. Supongamos que desea escribir código HTML en Citrine, en ese caso se podría crear un objeto HTML para convertir los mensajes en etiquetas:

```
>> HTML := Objeto nuevo.
HTML en: ['responder:y:'] hacer: {
:etiqueta :contenido
<- ['<etiqueta>contenido</etiqueta>']
etiqueta: etiqueta - [':'],
contenido: contenido.
}.
Salida escribir: (HTML H1: ['Title']), detener.
Salida escribir: (HTML P: ['Body text']), detener.
Salida escribir: (HTML B: ['bold text']), detener.
```

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

Tenga en cuenta que se utiliza **responder:y:** porque siguen dos argumentos. Es igualmente posible capturar mensajes genéricos que contengan tres o cuatro argumentos; respectivamente, **responder:y:y:** y **responder:y:y:y: ...**

Ahora, por ejemplo, cuando agrega el siguiente mensaje:

```
HTML en: ['responder:y:y:'] hacer: { :mensaje :contenido :valor
>> compartir := mensaje dividir: [':'].
>> etiqueta := compartir ? 1.
>> atributo := compartir ? 2.
<- ['<etiqueta atributo=['valor']>contenido</etiqueta>']
etiqueta: etiqueta,
contenido: contenido,
atributo: atributo,
valor: valor.
}.
```

Podemos crear hipervinculo HTML asi:

```
Salida escribir:(
  HTML
  A: ['Link']
  HREF: ['https://citrine-lang.org']
), detener.
```

Resultado

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

Este mecanismo para convertir automáticamente los mensajes en otra cosa también lo utiliza internamente Citrine. Los objetos **Ubicación-del-Archivo** e **Instrucción** utilizan este mecanismo en segundo plano.

4.4 Herencia, anulación, recursión

En el capítulo 2 se ha ilustrado que los objetos pueden basarse en otros objetos. De esta manera, se puede reutilizar el código escrito previamente para crear una jerarquía de objetos. Por ejemplo:

```
>> Animal := Objeto nuevo.
>> Perro := Objeto nuevo.
>> Caniche := Objeto nuevo.
```

En este caso, **Perro** hereda todo el comportamiento de **Animal** y, a su vez, el objeto Animal hereda todo el comportamiento de **Objeto**, el objeto raíz de todos los objetos. En el siguiente ejemplo se crea un nuevo tipo de secuencia: Combinación, en la que se garantiza que cada elemento sea único. Las funciones de la secuencia se reutilizan mediante herencia:

```
>> Combinación := Serie nuevo.
Combinación en: ['adjuntar:'] hacer: { :elemento
>> ver := yo encontrar: elemento.
   ver falso: { yo adjuntar: elemento. }.
}.
>> colores := Combinación nuevo.
colores
adjuntar: ['rojo'],
adjuntar: ['verde'],
adjuntar: ['azul'],
adjuntar: ['rojo'].
Salida escribir: colores, detener.
```

Observe cómo el segundo rojo queda excluido de la serie:

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

Ocasionalmente, puede ser necesario anular el comportamiento de un objeto. Por ejemplo, al sumar números y unidades de medida, se deben tener en cuenta. En el siguiente ejemplo, se crea el objeto **Tamaño**. Este objeto devuelve un número que, durante la suma, tiene en cuenta la unidad del número que se va a sumar. El objeto **Número** comprueba si se trata de pulgadas o pies.
El código de programación podría parecerse al siguiente:

```
>> Tamaño := Objeto nuevo.
Tamaño en: ['es:'] hacer: { :número
    número en: ['+'] hacer: { :número
        >> unidad := numéro calificador.
        >> factor := 1.
        unidad
        en-caso-de: ['pulgadas'] hacer: { factor := 0.0833. }.
        >> respuesta := yo + (número * factor).
        <- respuesta.
    }.
    <- número.
}.
```

Este número de tamaño se puede utilizar de la siguiente manera:

```
>> tablero := Tamaño es: 6 pies.
>> borde := Tamaño es: 50 pulgadas.
Salida escribir: tablero + borde, detener.
```

Esto se mostrará como:

```
10.165
```

En el ejemplo anterior, se está anulando el signo más. Tenga en cuenta que, finalmente, todavía se debe realizar la suma final, que, de hecho, tiene lugar en la siguiente línea:

```
>> respuesta := yo + (número x factor).
```

Ahora bien, *¿cómo entiende Citrine que este signo más se refiere a la lógica original de la suma?* Por ejemplo, otra interpretación podría ser que Citrine enviará repetidamente el mismo mensaje al mismo objeto, lo que, a su vez, daría como resultado un *bucle infinito.* Claramente, esta no es la intención. Tan pronto como envíe un mensaje a un objeto, que ejecutaría exactamente el mismo código, Citrine se dará cuenta, en este caso, de que se trata del mensaje subyacente anulado.
Por lo tanto, su programa estará automáticamente protegido contra esta forma de bucles infinitos. Sin embargo, cuando su objetivo sea, de hecho, ejecutar la misma tarea desde la tarea actual, será necesario enviar primero el mensaje de forma **recursiva**. En este caso, el resultado será un bucle infinito. Sin embargo, existen aplicaciones útiles para las tareas recursivas. Por ejemplo, supongamos que desea calcular el factorial de un número determinado. En ese caso, simplemente agregue el mensaje **factorial** a Número:

```
Número en: ['factorial'] hacer: {
  >> respuesta := 1.
  yo > 0 verdadero: {
       >> anterior := yo.
       >> siguiente := anterior - 1.
       respuesta := anterior * siguiente recursivo factorial.
  }.
  <- respuesta.
}.
```

Esto requiere recursión. De hecho, la tarea que está conectada al mensaje **factorial** debe ser ejecutada nuevamente dentro de esa tarea. Por lo tanto, es necesario invocar la tarea factorial desde dentro de la tarea factorial misma. Como regla, Citrine evitará esta rutina, debido al riesgo de terminar en un bucle infinito. Por esta razón, es vital preceder el mensaje con la palabra **recursivo**. Esto se hace para que Citrine sepa que es tu intención ejecutar la misma tarea y que no cometiste ningún error.
El siguiente bucle muestra los factoriales de los números del 1 al 10:

```
{ :x Salida escribir : x factorial , detener. } * 10.
```

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

Sin la palabra recursiva, solo se produciría la multiplicación inicial. La lista quedaría así:

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

4.5 Estado inicial

La creación de un objeto que se establece en un estado inicial determinado presenta un problema frecuente. Supongamos que se debe crear un objeto Rectángulo para calcular el perímetro y el área. Una posible notación sería:

```
>> Rectángulo := Objeto nuevo.
Rectángulo en: ['área'] hacer: {
   <- mi longitud * mi ancho .
}.
```

Obviamente, establecer una longitud y un ancho es una condición previa. Para ese propósito, se pueden agregar los mensajes longitud: y
ancho:

```
Rectángulo en: ['longitud:'] hacer: { :longitud
    mi longitud := longitud.
}.
Rectángulo en: ['ancho:'] hacer: { :ancho
    mi ancho := ancho.
}.
```

Este rectángulo se puede utilizar de la siguiente manera:

```
>> rectángulo := Rectángulo nuevo longitud: 2, ancho: 3.
Salida escribir: rectángulo area.
```

Resultado

```
6
```

Sin embargo, si se pasan por alto los ajustes iniciales de longitud y ancho, aparecerá un mensaje de error.

```
>> rectángulo := Rectángulo nuevo.
Salida escribir: rectángulo area .
```

```
Key not found: length
```

Para evitar esto, es preferible que un rectángulo siempre tenga una longitud y un ancho iniciales, por ejemplo 0. Por lo tanto, en este caso, es necesario anular el mensaje nuevo.

```
Rectángulo en: ['nuevo'] hacer: {
   >> rectángulo := yo nuevo.
   rectángulo longitud: 0.
   rectángulo ancho: 0.
   <- rectángulo.
}.
```

Observe que en la segunda línea se vuelve a invocar el **primer nuevo inicial**. Sin embargo, esto no resultará en un bucle infinito, ya que Citrine lo evitará. El mecanismo detrás de esto se ha destacado anteriormente en el capítulo 4.4. Cuando se crea un nuevo rectángulo, ahora medirá automáticamente 0 por 0, desde el principio. De esta manera, no se producirá ningún error al calcular el área:

```
>> rectángulo := Rectángulo nuevo.
Salida escribir: rectángulo area.
```

Resultado:

```
0
```

```
>> rectángulo := Rectángulo nuevo longitud: 2, ancho: 3.
Salida escribir: rectángulo area.
```

Result:

```
6
```

4.6 Conversión implícita

Citrine utiliza la conversión implícita para convertir objetos. Para imprimir una secuencia en la pantalla, Citrine, por ejemplo, enviará el texto del mensaje i nternamente a la secuencia. Esto puede resultar muy útil, en caso de que quiera imprimir una secuencia como una lista separada por comas. El texto del mensaje se puede sobrescribir:

```
>> suma := Serie ← 1 ; 2 ; 3.
suma en: ['texto'] hacer: {
  <- yo unirse: [','].
}.
Salida escribir: suma, detener.
```

Y de esta manera obtenemos

```
1,2,3
```

En las operaciones matemáticas con números, el número del mensaje se envía internamente. De esta manera, es posible crear una secuencia que proporcione automáticamente la suma durante una operación matemática:

```
>> suma := Serie ← 1 ; 2 ; 3.
suma en: ['número'] hacer: {
   >> total := 0.
   yo cada: { :número :elemento
                total añadir: elemento.
            }.
    <- total.
}.
Salida escribir: 1 + suma, detener.
```

```
7
```

Otro ejemplo es combinar una serie de objetos en un objeto de texto. Supongamos que tiene una libreta de direcciones, que está llena de objetos del tipo dirección:

```
>> Dirección := Objeto nuevo.
Dirección en: ['calle:'] hacer: { :calle
    mi calle  := calle.
}.
Dirección en: ['número:'] hacer: { :número
   mi número := número.
}.
Dirección en: ['adición:'] hacer: { :adición
    mi adición := adición.
}.
```

Esto permite introducir una dirección de forma estructurada. Sin embargo, cuando la dirección está lista para imprimirse, es preferible leer un texto que siga la notación común de: número de propiedad, posible adición y nombre de la calle. Por eso es necesario implementar un mensaje de texto:

```
Dirección en: ['texto'] hacer: {
  <- mi número + mi adición + [' '] + mi calle + [' '].
}.
```

El objeto de dirección se utiliza de la siguiente manera:

```
>> a := Dirección nuevo
número: 12,
adición: ['a'],
calle: ['Calle de la Iglesia'] .
```

```
>> b := Dirección nueva
número: 3,
suma: ['c'],
calle: ['Abbey Lane'].
```

A continuación, completa la libreta de direcciones:

```
>> direcciones := Serie ← a ; b.
```

La representación textual de la secuencia se adapta a:

```
direcciones en: ['texto'] hacer: { <- yo unirse: ['↵']. }.
```

Para finalizar, escribe la libreta de direcciones en la pantalla:

```
Salida escribir: direcciones, detener.
```

El resultado es:

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

Ahora bien, ¿cómo sabe Citrine que las direcciones deben imprimirse en este modo? Bueno, tan pronto como se envía **escribir:** al objeto Salida, la serie de direcciones se convierte en un texto.
Para lograr esto,Citrine envía internamente el **texto** del mensaje a la serie. Esta es una conversión implícita, lo cual no es nada nuevo. Sin embargo, en este ejemplo, se ha organizado una tarea para texto dentro de la libreta de direcciones.

Esta tarea combina todos los elementos, solo para estar separados por saltos de línea. Normalmente, una serie se mostraría muy diferente en la pantalla, sin embargo, al especificar explícitamente cómo se representará la serie como texto, Citrine puede adoptar esto. El siguiente paso en el proceso es que el **texto** del mensaje se envíe a cada elemento individual. En este código se ha especificado que cada vez que un objeto de dirección recibe el **texto** del mensaje, convierte la dirección en texto de acuerdo con el formato preestablecido.
Finalmente, este resultado se muestra en la salida. Por lo tanto, se necesitan básicamente dos pasos para obtener un resultado: la conversión textual del objeto de dirección y la conversión textual de la libreta de direcciones, que es, de hecho, la serie de direcciones. En última instancia, la combinación de estas dos conversiones implícitas garantiza un resumen adecuado de las direcciones.

Ahora supongamos que se añade una dirección c a la serie, por ejemplo:

```
>> c := Dirección nuevo.
>> direcciones := Serie ← a ; b ; c.
```

Obviamente, esto devolverá un mensaje de error, porque en c no hay nombre de calle, número de propiedad ni adición completada. Para evitar que esto suceda, utilice el método que se ha explicado anteriormente en el capítulo 4.5:

```
Dirección en: ['blanco'] hacer: {
   mi número := [''].
   mi adicción := [''].
   mi calle := [''].
}.
Dirección en: ['nuevo'] hacer: {
  >> dirección := yo nuevo.
  dirección blanco.
  <- dirección.
}.
```

Una vez más, la tarea asociada con el nuevo mensaje se reemplaza con una tarea que completa las propiedades requeridas de antemano.
Aquí está el código completo:

```
>> Dirección := Objeto nuevo.
Dirección en: ['calle:'] hacer: { :calle
    mi calle := calle.
}.
```

```
Dirección en: ['número:'] hacer: { :número
  mi número := número.
}.
Dirección en: ['adicción:'] hacer: { :adicción
  mi adicción := adicción.
}.
Dirección en: ['texto'] hacer: {
  <- mi número + mi addición + [' '] + mi calle + [' '].
}.
Dirección en: ['blanco'] hacer: {
  mi número := [''].
  mi adicción := [''].
  mi calle := [''].
}.

Dirección en: ['nuevo'] hacer: {
  >> dirección := yo nuevo.
  dirección blanco.
  <- dirección.
}.
```

```
>> a := Dirección nuevo
         número: 12,
         adicción: ['a'],
         calle: ['Church Street'].
>> b := Dirección nuevo
         numéro: 3,
         adicción: ['c'],
         calle: ['Abbey Lane'].
>> c := Dirección nuevo.
>> direcciones := Serie ← a ; b ; c.
   direcciones en: ['texto'] hacer : { <- yo unirse: ['↵']. }.
   Salida escribir: direcciones, detener.
```

Resultado:

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

Nota: la última línea está vacía (debido a la dirección c)

4.7 Serialización

Como se ha visto anteriormente, las series y listas se pueden convertir en textos. Otra forma de hacerlo es enviando el **código** del mensaje. Si utiliza este mensaje, sus objetos pueden volver a cobrar vida a partir del texto. Esto se denomina serialización. Después de la serialización, el texto resultante se puede guardar en el disco o transmitir a través de una conexión. De este modo, los objetos serializados se pueden compartir con otros sistemas y, por tanto, son útiles para la comunicación entre sistemas.

El siguiente ejemplo muestra una serialización de una cesta de la compra para almacenarla en un archivo, por lo que se vuelve a leer y se cuenta la cantidad de productos:

```
>> comestibles := Serie nuevo
~ ['barquillos con jarabe']
~ ['café']
~ ['granas'].
>> cesta := Archivo nuevo: ['cesta'].
cesta escribir: comestibles código.
>> x := Archivo nuevo: ['cesta'], leer objeto contar.
Salida escribir: x, detener.
```

```
3
```

Existe una sutil diferencia entre el **texto** y el **código** de los mensajes. El **texto** del mensaje da como resultado una representación textual del objeto. En cambio, el código del mensaje da como resultado una representación en código de programación ejecutable. A modo de comparación, aquí hay dos respuestas al **texto** y **código** de los mensajes de un objeto Texto:

```
Salida escribir: ['Ceci n'est pas une pipe'] texto, detener.
Salida escribir: ['Ceci n'est pas une pipe'] código, detener.
```

```
Ceci n']est pas une pipe
['Ceci n']est pas une pipe']
```

Tenga en cuenta que en el segundo caso, la salida de texto se ha colocado entre comillas simples, para permitir que Citrine lea y procese esta salida nuevamente. Como regla, la serialización no se limita solo a listas y series. También es posible obtener representaciones de código de otros objetos de Citrine:

```
>> x := ['Salida'] objeto.
x escribir: 123, detener.
```

```
123
```

Este truco se puede aplicar fácilmente cuando se necesita devolver una representación textual de un objeto a un objeto.

Ahora, el siguiente ejemplo muestra una suma simple que se aplica a una serie de variables:

```
Salida escribir: (
{ <- x + 1. } aplicar: ['a b c']
).
```

Como regla, esto no es factible en Citrine, sin embargo, Citrine se puede extender de una manera que permitirá que la suma **X + 1** se aplique a una serie de variables **(a, b y c)**. En ese caso, puede extender el objeto Tarea con la siguiente funcionalidad:

```
>> a := 1.
>> b := 2.
>> c := 3.
Tarea en: ['aplicar:'] hacer: {
   :variables
   >> respuesta := Serie nuevo.
   variables dividir: [' '], cada: {
       :index :variable
       >> x := variable objeto.
       x := yo empezar.
       respuesta adjuntar: x, detener.
    }.
    <- respuesta.
}.
```

El resultado:

```
Sequence ← 2 ; 3 ; 4
```

Ahora bien, si no hubiéramos utilizado la **variable objeto** sino simplemente **variable**, el resultado sería completamente diferente. Después de todo, la acción (**X + 1**) se aplicaría a las letras **a, b** y c en lugar de a los números **1**, **2** y **3**:

```
Sequence ← ['a1'] ; ['b1'] ; ['c1']
```

En este caso, se añadiría el número **1** a los textos **a**, **b** y **c**.
Al enviar el **objeto** de mensaje a un texto que hace referencia a un objeto inexistente, se producirá un error de mensaje:

```
>> x := ['lámpara mágica'] objeto.
```

El resultado será:

```
Key not found: magic lamp
```

Tampoco se pueden ejecutar fragmentos de código completos de esta manera, porque Citrine está protegido contra esta forma de inyección de código:

```
Salida escribir: ['{ <- 1 + 1. } empezar'] objeto, detener.
Salida escribir: ['1 + 1'] objeto, detener.
```

Esto no mostrará ningún resultado:

```
None
None
```

Sin embargo, es posible intercambiar código ejecutable a través de un objeto de texto. En este caso, es necesario capturar el **código** en el objeto que desea convertir en texto. Esta técnica se aplica en el siguiente ejemplo:

```
>> ramo := Objeto nuevo.
ramo en: ['código'] hacer: {
   <- ['ramo'].
}.
ramo en: ['texto'] hacer: {
   <- ['rosas'].
}.
>> texto := (Serie ← ramo ; ['tarjeta']) código.
Salida escribir: texto objeto primero, detener.
```

La respuesta de este programa es:

```
roses
```

El motivo por el que esto funciona correctamente es que el **código** del mensaje también se envía automáticamente a todos los elementos subyacentes de la secuencia y, en consecuencia, también al objeto **ramo**. El objeto ramo responde con la referencia a sí mismo. Durante la conversión del texto a un objeto Serie, la referencia al objeto ramo se restablece, de hecho. En consecuencia, cuando este objeto se imprime en pantalla, el texto del mensaje se envía internamente a ramo mediante una conversión implícita y devolverá la respuesta **rosas**.

4.8 Estructura alternativa de mensaje

Supongamos que tiene una serie:

```
>> x := Serie ← 1 ; 2 ; 3.
```

Si desea eliminar el primer y el último elemento, su mensaje sería:

```
x toma-el-primero toma-el-último.
```

Desafortunadamente, la notación anterior no funcionará correctamente. El mensaje **toma-el-primero** devuelve el primer elemento de la serie, que, a su vez, se convertirá en el receptor del mensaje **toma-el-último**. Por lo tanto, el mensaje **toma-el-último** no se envía a x, sino a 2. Una posible solución a este problema sería crear dos setencias separadas, como:

```
x toma-el-primero
x toma-el-último
```

Sin embargo, esto es poco práctico, en particular cuando necesita eliminar más de dos elementos.
Por lo tanto, Citrine ofrece una estructura de mensaje alternativa para este tipo de situaciones. En las estructuras de mensajes alternativas, las respuestas de los objetos se ignoran y obtendrá el objeto receptor como respuesta, una y otra vez. Una estructura de mensaje alternativa se inicia enviando el mensaje **hacer** a un objeto y la estructura de mensaje alternativa puede finalizar enviando el mensaje **hecho**. En el ejemplo ilustrado, esto podría aplicarse de la siguiente manera:

```
>> x := Serie ← 1 ; 2 ; 3.
x hacer toma-el-primero toma-el-último hecho.
Salida escribir: x, detener.
```

Tenga en cuenta que para evitar sorpresas desagradables, siempre cierre con **echo**. Vea lo que sucede en el próximo ejemplo:

```
>> x := Serie ← 1 ; 2.
x hacer toma-el-primero.
Salida escribir: x, detener.
```

El código anterior generará el mensaje de error: **Debe responder con texto**. La razón de esto es que el mensaje de **escribir:** desea crear un texto de **x**; por lo tanto, envía internamente el **texto** del mensaje. Como regla, se devuelve un objeto de texto como respuesta, sin embargo, como la estructura de mensaje alternativa aún está activa, se devuelve la serie en sí. Esto hace que el procesamiento posterior falle y Citrine mostrará un mensaje de error.

4.9 Mensajes enviados mediante programación

En lugar de enviar un mensaje directamente a un objeto, también se puede enviar a través de una variable. Para ello, se utiliza el mensaje **mensaje:argumentos:** Por ejemplo, en el siguiente ejemplo, el resultado muestra 14
(de una manera muy elaborada):

```
>> a := (Serie ← 7).
>> b := ['añadir:'].
>> x := 7 mensaje: b argumentos: a.
Salida escribir: x, detener.
```

Salida:

```
14
```

Especifica el mensaje deseado como objeto de texto y los argumentos como serie. Un mensaje que contiene dos parámetros (**por ejemplo, entre: 0 y: 10**), se especifica como **entre: y:**, por lo tanto, se combinan los componentes del mensaje:

```
>> a := (Serie ← 1 ; 10).
>> b := ['entre:y:'].
Salida escribir: ( Número mensaje: b argumentos: a ), detener.
```

Salida (posiblemente):

```
8
```

Gracias a los mensajes enviados mediante programación, los objetos se pueden convertir en mensajes. Esto permite que el usuario final se comunique directamente con los objetos. En el ejemplo que se muestra a continuación, se crea una especie de miniprocesador de textos, simplemente proporcionando al usuario un objeto de texto en blanco. El usuario puede crear un texto utilizando mensajes de Citrine. Estos mensajes se pueden pasar al objeto de resultado.

```
>> resultado := [''].
{
  Salida escribir: resultado, detener.
  resultado
  mensaje: Programa pedir
  argumentos: (Programa pedir dividir: [',']).
}
mientras: { <- Verdadero. }.
```

La siguiente ilustración muestra una transcripción de una sesión

```
$ miniwrite

add:
Hello
Hello
replace:with:
e,E
hEllo
capitals

HELLO
```

Aunque este es el procesador de textos más corto que uno pueda imaginar, no es muy fácil de usar. Sin embargo, funciona de manera bastante simple.

En la variable, **resultado**, el usuario del programa puede modificar el texto a través de la comunicación interactiva con el objeto. En esta sesión, el usuario llena el objeto Texto con la palabra **Hello** enviando **añadir:** con el argumento **Hello**. A continuación, la **e** minúscula debe reemplazarse por una letra mayúscula. Esto se puede lograr escribiendo el comando **reemplazar:con:** seguido de **e,E**. Esto
lleva al siguiente mensaje:

```
reemplazar: ['e'] con: ['E']
```

En la siguiente línea, el usuario envía el mensaje **mayúsculas** al texto; en consecuencia, todas las letras se reemplazan por mayúsculas. El resultado detrás de las pantallas será:

```
['hEllo'] mayúsculas
```

De hecho, en este caso, los propios usuarios están programando en Citrine sin saberlo, de una manera un tanto torpe. El programa resultante también puede verse como una implementación rudimentaria de un sistema REPL (Read-Eval-Print Loop).

4.10 Módulos

El mundo de Citrine se puede ampliar con nuevos objetos de sistema, instalando módulos. Se puede agregar un nuevo objeto de sistema al mundo de Citrine colocando el archivo del módulo (normalmente un archivo con el sufijo .so o .dll) en la carpeta mods. Supongamos que desea convertir una secuencia dada al protocolo JSON; para ello, se puede utilizar el módulo de complemento JSON. Para instalarlo, haga lo siguiente:

```
mods/json/libctrjson.so
```

El sufijo exacto del archivo varía según el sistema. El ejemplo anterior se basa en un sistema Linux.
Sin embargo, la ubicación del archivo de un archivo de expansión siempre se verá similar a lo siguiente:

mods / <object name> / libctr <object name> . extension (so/dll/dylib)

Cuando envías un mensaje al objeto que está disponible a través del módulo de complemento, Citrine lo cargará automáticamente:

```
>> cartas := Serie nuevo
~ ['poker']
~ ['belote'].

>> tableros := Serie nuevo
~ ['chutes and ladders']
~ ['ludo'].

>> juegos := Lista nuevo
cartas: cartas,
tableros: tableros.

>> texto := Json jsonify: juegos.
Salida escribir: texto.
```

```
{"boards":["chutes and ladders","ludo"], "cards":
["poker","belote"]}
```

En el ejemplo anterior se ha aplicado el módulo JSON. La lista de juegos se convierte en un objeto Texto, que representa los datos según el protocolo JSON: Para ello se envía el mensaje **jsonify:** con la lista como argumento. La salida se muestra en el ejemplo anterior

Tan pronto como se invoca **Json**, Citrine buscará si hay un módulo en la carpeta mods que haga que este objeto esté disponible. Para esta acción, Citrine implementa una comparación de texto. Citrine busca el archivo l**ibctrjson**, para ubicar el objeto **Json**.

En caso de que se trate de un objeto llamado **Xml**, Citrine verificará si el archivo libctrxml está presente o no. Este análisis solo se ejecuta cuando se envía un mensaje a un objeto que no está presente en el programa actual. Como regla, no afectará el rendimiento del programa informático en sí. El análisis solo se ejecuta en caso de que sea probable que ocurra un error de programa de todos modos. Tan pronto como se identifica, se carga y el programa seguirá manejando el mensaje.

Si envía un mensaje a un módulo inexistente, aparecerá un mensaje de error. Este es el mismo mensaje de error que un mensaje que se envía a un objeto inexistente. Por lo tanto, en ambos casos el mensaje de error es el mismo. En consecuencia, Citrine iniciará una búsqueda de cada objeto desconocido para ver si el objeto aún puede cargarse como una expansión. Si este no es el caso, el proceso falla. El siguiente ejemplo prueba si un módulo de expansión está realmente presente:

```
{ Tetera infusionar . } capturar: {
  Salida escribir: ['¡No hay té para ti!'], detener.
}, empezar.
```

4.11 Detección

El capítulo anterior mostró cómo averiguar si un módulo de expansión está presente o no.
Sin embargo, existen varias formas de explorar el sistema durante la ejecución del programa. Citrine tiene un par de métodos para detectar qué objetos están presentes y a qué mensajes responden estos objetos.
En primer lugar, se puede preguntar a cada objeto qué tipo es:

```
>> cosa := Objeto.
Salida escribir: cosa tipo , detener.
Salida escribir: 12345 tipo , detener.
Salida escribir: ['word salad'] tipo , detener.
Salida escribir: Verdadero tipo , detener.
```

```
Object
Number
Text
Boolean
```

A diferencia de muchos lenguajes de programación comunes, los tipos son maleables. El tipo estándar de un objeto es, por ejemplo, **objeto**, pero puedes modificarlo:

```
>> Persona := Objeto nuevo.
Persona en: ['tipo'] hacer: { <- ['humano']. }.
Salida escribir: Persona tipo.
```

En este caso, la salida será la siguiente:

```
human
```

Observe cómo esto no influye en el objeto Persona, ya que solo se ha modificado la respuesta al tipo de mensaje, nada más.
Los tipos pueden ser útiles para verificar si el mensaje está recibiendo los argumentos correctos. Por ejemplo, permiten verificar durante una adición si los argumentos son, de hecho, números. Sin embargo,

son formas más pragmáticas de recuperar las propiedades de los objetos. Debido a que los objetos Citrine no generan errores en mensajes no válidos, puede tratar los objetos que espera que sean numéricos como tales y ver qué sucede. Si el objeto que obtiene resulta ser otro, no se verá afectado por sus
mensajes.

El objeto **Programa** se presentó en el capítulo 3.12. Con este objeto y utilizando el mensaje **utilizar:**, es posible incluir programas de otros en su propio programa. Los objetos que están disponibles en archivos de programa externos, también se pueden utilizar en su propio programa. También es posible preguntar al objeto **Programa** si un objeto dado ya está presente en el programa:

```
Programa Herramienta verdadero: { … }.
```

El código en el lugar de la línea de puntos se ejecutará, en este caso, si el objeto Herramienta realmente existe. En el caso de un nombre de objeto que consta de un solo símbolo, es mejor aplicar la siguiente notación para evitar confusiones:

```
Programa encontrar: ['X'], verdadero: { … }.
```

Después de todo, :

```
Programa X
```

no es un mensaje válido.

Además de preguntar por objetos, es igualmente posible preguntar al objeto **Programa** por mensajes. Por ejemplo, se puede preguntar si el objeto **Número** conoce el mensaje **entre:y:**, de la manera que se ilustra a continuación:

```
Salida escribir: ( Programa Número: ['entre:y:'] ), detener.
Salida escribir: ( Programa Número: ['reemplazar:con:'] ), detener.
```

Resultado

```
True
False
```

Tenga en cuenta la notación del mensaje: el mensaje que debe ser reconocido por el objeto relevante debe escribirse junto y sin argumentos. Cuando, por ejemplo, desea averiguar si se le puede pedir al objeto Conejo de Pascua que salte: 5 metros a: izquierda, se podría enviar el siguiente mensaje a Programa:

```
Programa Conejo de Pascua: ['saltar:a:'].
```

En resumen, el mensaje relevante que debe ser respondido por el **Conejo de Pascua**, se escribe básicamente como **saltar:a:.**

El objeto Programa también puede proporcionar información sobre la versión de Citrine que se está utilizando en un momento determinado. Como respuesta al texto del mensaje, el objeto Programa muestra El lenguaje de programación Citrine/UK. Como respuesta al número del mensaje, se mostrará el número de versión, por ejemplo:

```
Salida escribir: Programa, detener.
Salida escribir: Programa número, detener.
```

```
The Programming Language Citrine/UK

94
```

El número de versión siempre se muestra como un número redondo. La versión 0.9.4 se convierte en 94

4.12 Ejercicios

1. Ver ejemplo Combinación en el capítulo 4.4. ¿Mediante qué bypass es posible la duplicación de elementos? ¿De qué manera se puede evitar esto?

2. El siguiente código solo cuenta números con el mismo calificador:

```
Número en: ['+'] ...: { :otro
>> ... := yo copiar.
yo calificador = otro ... ...: {
... añadir: otro.
}.
<- sum.
}.
Salida escribir: 2 manzanas + 3 peras, detener.
Salida escribir: 2 manzanas + 3 manzanas, detener.
```

El resultado:

```
2 apples
5 apples
```

¿Qué código debería estar en las líneas de puntos? Complete los espacios en blanco.

3. Implemente el mensaje **copiar:** para Archivo.

4. Cree un nuevo objeto llamado **Reflexión**. Este objeto convierte cada mensaje que recibe en un objeto Texto y cambia los caracteres:

Salida escribir: Reflexión pimienta.
El resultado:

```
reppep
```

¿Cómo se ve el código o la definición del objeto Reflection?

5. Un palíndromo o palabra espejo es una palabra cuyas letras o caracteres se leen igual al revés, ya que están dispuestos simétricamente. Amplíe el objeto Text con el mensaje **palíndromo**?, de modo que los usuarios puedan determinar si la palabra ingresada es, de hecho, un palíndromo. Para ello, complete el código faltante en la línea de puntos a continuación:

```
Texto en: ['palíndromo?'] hacer: { ... }.
Salida escribir: ['pepper'] palíndromo?, detener.
Salida escribir: ['level'] palíndromo?, detener.
```

Output:

```
False
True
```

Pista: usa el objeto escrito anteriormente **Reflection.**

6. Supón que tienes un objeto Pair, que se ha derivado de Serie, para crear pares de datos.

Ejemplo:

```
>> pair := Pair nuevo de: ['pepper'] y: ['salt'].
Salida escribir: pair first, detener.
Salida escribir: pair second, detener.
```

a. ¿Cuál sería la salida del código anterior?
b. El objeto **Pair** devuelve el primer elemento al recibir el mensaje **primero**. Ese código se parece al siguiente:

```
Pair en: ['primero'] hacer: {
< -mi elementos primero.
}.
```

Ahora, escribe el código preciso para devolver el segundo elemento.
c. Los elementos de un par se pueden configurar enviando el mensaje **de: y :** ; por ejemplo, pair **de**: pepper **y:** salt. Escribe la implementación de **de: y:** , completando el código faltante en las líneas de puntos:

```
Pair en: ['de:y:'] hacer: {
:primero :segundo
mi ... ...: ..., ...: .…
}.
```

d. Agrega el mensaje en blanco para comenzar con un par vacío:

```
Pair en: ['blanco'] hacer: {
...
}.
```

e. Al crear un nuevo par, es necesario primero configurar los elementos (que contienen valores en blanco), para
evitar errores. Crea el código para el mensaje nuevo.

```
Pair en: ['nuevo'] hacer: {
...
}.
```

f. Cree un mensaje que permita una representación textual de un par de datos:

```
Salida escribir: pair, detener.
```

Esto muestra el siguiente resultado:

```
pepper and salt
```

7. Hay un error en el siguiente programa. La salida es 2, sin embargo, la salida correcta debería ser **3**. Encuentra el error.

```
>> puntos := Serie ← 1 ; 2 ; 3 ; 1 ; 2.
puntos toma-el-primero toma-el-primero toma-el-último toma-el-último.
>> medio := puntos primero.
Salida escribir: medio, detener.
```

8. ¿Cuántos ases hay al final de este programa?

```
>> aces := 2.
>> Cartas := Objeto nuevo.
Cartas en: ['truco'] do: { aces añadir: 1. }.
{ >> aces := 4. Cartas truco. } empezar.
Cartas truco.
```

9. Supón que se está instalando el complemento **libctraquarium.so**. ¿A qué objeto se puede enviar un mensaje para cargar este módulo?

10. ¿Cuál es la salida de este programa en la pantalla?

```
>> a := 0.
>> b := a = ['cero'].
>> c := 2.
>> d := c = ['dos'].
>> e := ['cero'].
>> f := e = 0.
Salida escribir: b, detener.
Salida escribir: d, detener.
Salida escribir: f, detener.
```

5. Traducir

5.1 Sistema de traducción

Es posible traducir programas de un idioma a otro. Citrine conoce dos métodos para traducir el código de un programa: el sistema de traducción y las técnicas de traducción dinámica (más sobre esto más adelante). Utilizando el sistema de traducción con sus sencillas reglas de traducción, el código de un programa se puede convertir de un idioma humano a otro. Las técnicas de traducción dinámica son herramientas que se pueden implementar para afinar los resultados de estas traducciones. Además, el código de Citrine se beneficiará de la utilización de los métodos mencionados anteriormente, ya que dan lugar a una apariencia más natural del código resultante. En este capítulo se trata el sistema de traducción general.

Puede utilizar este sistema para traducir un archivo de programa de un idioma a otro. Para ello, es necesario disponer de un archivo de diccionario. Más adelante se explicará cómo crear archivos de diccionario, pero por ahora se puede suponer que ya existe un archivo de diccionario. Supongamos que desea traducir un archivo escrito en inglés al italiano y que ya está disponible el archivo de diccionario necesario **enit.dict** (que significa de EN a IT), el archivo de programa se puede traducir de la siguiente manera:

```
ctren -t enit.dict program1.ctr > program2.ctr
```

El programa traducido se guardará como **program2.ctr.** Tenga en cuenta que la notación exacta para este comando puede variar según el sistema operativo que se utilice. El ejemplo mencionado anteriormente está pensado para una ejecución en sistemas similares a Linux. Después de implementar el ejemplo anterior, el resultado mostrará la traducción al italiano del programa. El sistema de traducción avisará en caso de que no reconozca ciertas palabras y, en consecuencia, estas palabras no se traducirán.

Un archivo de diccionario consta de dos columnas; a la izquierda, la palabra que se traducirá y, a la derecha, su traducción. Supongamos que desea traducir el siguiente programa al inglés:

```
Out scrivi: ['Ciao mondo!'], stop.
```

Aquí, es necesario tener un archivo de diccionario que contenga traducciones para las palabras **scrivi, stop** y **Ciao mondo!.** Un diccionario puede contener 4 grupos de palabras: palabras token (t), textos (s), símbolos decimales (d) y símbolos separadores de números (x). A cada grupo individual se le asigna su propio carácter, que se ha colocado entre corchetes para este propósito. Este carácter representa el tipo de palabra que se necesita traducir, situado en la columna de la izquierda. El siguiente archivo podría traducir el programa:

t "scrivi:" "write:"
t "stop" "stop"

s "¡Ciao mondo!" "¡Hola mundo!"

Cuando este archivo se guarda como **iten.dict**, el programa se puede traducir de la siguiente manera:

```
ctren -t iten.dict program.ctr
```

El resultado sería:

```
Salida escribir: ['Hola Mundo!'], detener.
```

Al traducir mensajes de palabras clave, como: Número entre: X y: Y, los segmentos del mensaje se combinan, lo que da como resultado la siguiente línea de traducción:

t "tra:e:" "between:and:"

Al utilizar un archivo de diccionario, pueden producirse los siguientes errores:

Error de traducción, mensaje demasiado largo.	Su traducción excede los 255 bytes, esto no está permitido.
Se encontró una palabra ambigua	El archivo del diccionario tiene la misma palabra dos veces.
Traducción ambigua	El diccionario tiene la misma traducción dos veces.
El tipo de mensaje no coincide Traducción	Por ejemplo, de un mensaje unario a un mensaje de palabras clave o un mensaje binario y viceversa.
Error general	Se ha producido un error diferente (raro).

Para facilitarle el trabajo, Citrine también ofrece un generador de diccionarios para crear un diccionario para el vocabulario principal, que luego puede completarse libremente. Se puede generar un diccionario en caso de que existan dos versiones de código fuente diferentes de Citrine. Por ejemplo, para crear un archivo de diccionario de inglés a italiano para un programa de Citrine, se puede utilizar la siguiente notación:

```
ctren -g citrineen/i18n/en/dictionary.h
citrieneit/i18n/it/dictionary.h > enit.dict
```

Tenga en cuenta que aquí usamos la opción -g (significa: generar)

5.2 Traducciones dinámicas

El uso de renombramiento de objetos, el uso de alias y la construcción de sintaxis son ejemplos de métodos de traducción dinámica. Se denominan así porque traducen el código del programa ejecutando acciones durante el tiempo de ejecución del programa. Estos métodos también se pueden aprovechar para mejorar la legibilidad del código del programa, dándole una apariencia más natural.

La herramienta más obvia y simple a su disposición es el renombramiento de objetos.

```
>> Vacio := Serie nuevo.
```

En ciertos momentos, será necesario enviar una serie vacía con una tarea o como argumento con un mensaje a un objeto. En lugar de tener que escribir repetidamente **Serie nuevo**, es posible simplemente utilizar la palabra **Vacio** después de ejecutar el código mencionado anteriormente. Por ejemplo, como se ilustra a continuación:

```
mensaje de computadora: ['boot'] argumentos: Vacio.
```

Otra herramienta que se puede utilizar es el mensaje **aprender:significa:**. Esto enseña a un objeto a percibir un mensaje como un mensaje diferente. Por ejemplo:

```
Salida aprender: ['garabateamos:'] significa: ['escribe:'].
Salida garabateamos: ['La brevedad es el alma del ingenio'], detener
```

Resultado:

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

Tenga en cuenta que el sinónimo solo se puede utilizar en el objeto al que se envía el mensaje de aprendizaje.

Appendices

Apéndice A: Exportación de AST

Con la función de exportación de AST, es posible exportar un programa de Citrine como una estructura de tipo árbol. La salida se puede traducir a diferentes lenguajes de programación, por ejemplo, C. También es posible utilizar la salida para mostrar una representación visual del programa en cuestión.
Para exportar un programa de Citrine, utilice la opción -x, como:

```
ctrXX -x program.ctr
```

Supongamos que el archivo de programa “program.ctr” tiene el siguiente contenido:

```
Salida escribir: ['Hello'].
Salida escribir: ['World'].
```

La salida de la función de exportación se verá así:

```
52;0;0;;[57;0;3;Out;;55;0;8;schrijf:;[56;0;5;Hallo;;];];52;0;0;;
[57;0;3;Out;;55;0;8;schrijf:;[56;0;6;Wereld;;];];79;0;0;;;
```

Lo que se muestra arriba es una secuencia de elementos, separados por un punto y coma. Cada elemento contiene 5 campos, separados nuevamente por un punto y coma. El primer campo indica qué tipo de elemento de programa involucra. Los elementos también pueden contener otros elementos, que se enumeran en el último campo (el quinto campo) y están encerrados entre corchetes.

Se pueden distinguir los siguientes tipos de elementos:

Código del elemento	Significado
51	Declaración >>own:
52	Un mensaje (general), detalles en subelementos
53	Un mensaje unario (sin argumentos)
54	Un mensaje binario (con un argumento)
55	Keyword message (with one or multiple arguments)
56	Fragmento de texto entre comillas, es decir, un objeto de texto
57	Referencia/Nombre (de una variable)
58	Número
59	Una tarea entre llaves {…}
60	Flecha de respuesta <-
76	Un bloque de parámetros como parte de una definición de tarea
77	Instrucciones que forman parte de la tarea (59
78	1 parámetro que forma parte de los parámetros (76
79	Fin del programa
80	Expresión entre paréntesis (…)

Si el primer campo es igual a 57 (referencia), el segundo campo indica una referencia a una variable ya declarada (0), a las propiedades de un objeto (1) o a una declaración de una variable completamente nueva (2). Si el primer campo es igual a 53 a 58 o 78 (por ejemplo, en el caso de un elemento de texto), el contenido del elemento se imprimirá en la ubicación del cuarto elemento, el búfer. El tercer elemento indica la longitud de este búfer en bytes. En el campo final, los elementos encerrados se imprimen entre corchetes. Como se proporciona la longitud del búfer, no es necesario determinar ninguna secuencia de escape.

El resultado, al inspeccionar el código del ejemplo anterior:
– 52 un mensaje
A continuación, sigue una especificación adicional:
– 57 una referencia a un objeto, 3 bytes (ya que es un símbolo UTF-8) Salida.
– 55 palabra clave message, 6 bytes “escribir:”
A continuación sigue una serie adicional de subelementos para los argumentos, en este caso solo uno:
– 56 un texto, 5 bytes “Hello”
Y así sucesivamente…
Observe que el programa termina con el código 79 “finalizar programa”. Este código marca final de un programa.

Apéndice B: Respuestas

Respuestas capítulo 1

1. Hola británicos

2. Hola británicos

3.

3a. Cuatro partes, que son: introducción, explicación, ejercicios y respuestas.
3b. Capítulo
3c. Tema

4. Pregunta

5.
5a. Un archivo de audio
5b. Colección de música
5c. 3
5d. BWV595

6. 20.
Los operadores en Citrine se aplican de izquierda a derecha.
Obtenga más información en el capítulo 2.

7.
>> declarar
yo enviar mensaje a yo
mi propiedad
← iniciar secuencia
<- devolver objeto

Respuestas capítulo 2

1. -x

2. 16.5

3.
```
>> x := 2.
{ x suma: (x potencia: 2). } * 2.
```

4.
```
>> x := 2 a la potencia: 3, + 2.
```

5. ¿par?

6. mientras:

7.
té
```
>> x := 7.
(x > 7) verdadero: { Salida escribir: ['p']. }.
(x < 7) verdadero: { Salida escribir: ['o']. }.
(x ≥ 7) verdadero: { Salida escribir: ['t']. }.
(x ≤ 7) verdadero: { Salida escribir: ['e']. }.
(x ≠ 7) verdadero: { Salida escribir: ['n']. }.
(x = 7) falso: { Salida escribir: ['t']. }.
(7 > x) verdadero: { Salida escribir: ['i']. }.
(7 < x) falso: { Salida escribir: ['a']. }.
(7 ≥ x) falso: { Salida escribir: ['l']. }.
(7 ≤ x) falso: { Salida escribir: ['l']. }.
(7 ≠ x) verdadero: { Salida escribir: ['y']. }.
```

8.
a= 8
b= 1 +2
c= 4
d= 10.5
e devuelve un error, división por 0

9.

```
>> ¿Ola de calor? := {
  :días-soleados :días-tropicales
   >> respuesta := Falso.
   (días-soleados >=:  5) verdadero: {
       (días-tropicales >=: 3) verdadero: {
           respuesta := Verdadero.
       }.
    }.
    <- respuesta.
}.
Escritura de salida: (¿Ola de calor? aplicar: 6 y: 3), detener.
```

La razón por la que este programa no funcionó fue porque la respuesta no era parte de la tarea Ola de calor, sino que era parte de una subtarea. -

Una tarea solo puede finalizar en un único punto. Otros lenguajes de programación ofrecen la posibilidad de finalizar una tarea equivalente en varios puntos, lo que causa confusión.

Respuestas capitulo 3

1.

```
>> celcius := -10.
{
    >> fahrenheit := celcius * 1.8 + 32.
    Salida escribir: fahrenheit, detener.
}
mientras: { <- (celcius añadir: 1)   40. }.
```

2.

```
>> frase := ['buy while the iron is cheap'].
frase buy: ['strike'], cheap: ['hot'].
Salida escribir: tile, detener.
```

3.

```
Número: ['square'] hacer: { <- yo to the power: 2. }.
Salida escribir: 3 square, detener.
```

4.

```
>> Punto := Objeto nuevo.
Punto en: ['x-coordenada:'] do: { :x mi x := x. }.
Punto en: ['y-coordenada:'] do: { :y mi y := y. }.
Punto en: ['x-coordenada'] do: { <- mi x. }.
Punto en: ['y-coordenada'] do: { <- mi y. }.
>> iglesia := Punto nuevo x-coordenada: 5, y-coordenada: 6.
>> torre := Punto nuevo x-coordenada: 7, y-coordenada: 1.
>> Distancia := Objeto nuevo.
Distancia en: ['empezar:finalizar:'] hacer: { :empezar :finalizar
mi empezar := empezar.
mi finalizar := finalizar.
}.
Distancia en: ['número'] hacer: {
<-
(mi empezar x-coordenada - mi finalizar x-coordenada) absoluto
+ (mi empezar y-coordenada - mi finalizar y-coordenada) absoluto.
}.
```

```
>> x := Distancia nuevo
 iniciar: iglesia
 finalizar: torre,
 número.
Salida escribir: x, detener.
```

5. 10

6.

```
Número en: ['doble'] hacer: { <- yo * 2. }.
Salida escribir: 4 doble, detener.
```

7.

```
Número en: ['dígitos:'] hacer: { :n
    >> dígitos := yo texto.
    >> caracteres := Serie nuevo.
    >> diff := (n - dígitos longitud).
    caracteres llenar: diff con: ['0'].
    >> texto := caracteres unirse: [''].
    texto añadir: dígitos.
    <- texto.
}.
Salida escribir: (9 dígitos: 9), detener .
```

Respuestas capítulo 4

1.

Por ejemplo:

```
>> Combinación := Serie nuevo.
Combinación en: ['adjuntar:'] hacer: { :elemento
>> visto := yo encontrar: elemento.
visto falso: { yo adjuntar: elemento. }.
}.
>> colores := Combinación nueva.
colores
adjuntar: ['rojo'],
adjuntar: ['verde'],
adjuntar: ['azul'],
anteponer: ['rojo'].
Salida escribir: colores, detener.
```

-Esto se puede evitar modificando también los otros mensajes que modifican la colección.

2.

```
Número en: ['+'] hacer: { :otro
   >> suma := yo copiar.
   yo calificador = otro calificador verdadero: {
      suma añadir: otro.
   }.
   <- suma.
}.
Salida escribir: 2 manzanas + 3 peras, detener.
```

```
Salida escribir : 2 manzanas + 3 manzanas, detener.
```

```
2 apples
5 apples
```

3. Caso de prueba completo:

```
Archivo en: ['copiar:'] hacer: { :fuente
yo escribir: fuente leer . }.
>> file1 := Archivo nuevo: Ruta /tmp file1.
>> file2 := Archivo nuevo: Ruta /tmp file2.
{
file1 .
file2 eliminar.
} capturar: { :err }, empezar.
file1 escribir: ['abc'].
file2 copiar: file1.
Salida escribir: file2 leer.
```

4.

```
>> Reflexión := Objeto nuevo.
Reflexión en: ['responder:'] hacer: { :texto
     >> alreves := Serie nuevo.
     texto caracteres cada: { :número :caracter
        alreves anteponer: caracter.
      }.
      <- alreves unirse: [''].
}.
Salida escribir: Reflexión pimienta.
```

5

```
Texto en: ['palíndromo?'] hacer: {
<- (reflexión mensaje: yo argumentos : Serie nuevo) = yo.
}.
```

6.
a.
pimienta
sal
b.

```
>> Par := Objeto nuevo.
Par en: ['primero'] hacer: {
  <- mi elementos primero.
}.
Par en: ['segundo'] hacer: {
  <- mi elementos último.
}.
```

c.

```
Par en: ['de:y:'] hacer: {
   :primero :segundo
    mi elementos
    adjuntar: primero,
    adjuntar: segundo.
}.
```

d.

```
Par en: ['blanco'] hacer: {
  mi elementos := Serie nuevo.
}.
```

e.

```
Par en: ['nuevo'] hacer: {
```

```
>> par := yo nuevo.
<- par blanco.
}.
```

f.

```
Par en: ['texto'] hacer: {
<- ['primero y segundo']
   primero: mi elementos primero,
   segundo: mi elementos último.
}.
```

7.

```
>> puntos := Serie ← 1 ; 2 ; 3 ; 1 ; 2.
puntos hacer
toma-el-primero
toma-el-primero
toma-el-último
toma-el-último hecho.
>> medio := puntos primero.
Salida escribir: medio, detener.
```

8.
3 ases

9.

Acuario.

10.

Verdadero Falso Falso

6. código completo

```
>> Par := Objeto nuevo.
Par en: ['primero'] hacer: {
  <- mi elementos  primero.
}.

Par en: ['segundo'] hacer: {
```

```
<- elementos propios al final.
}.

Par en: ['de:y:'] hacer: {
  :primero :segundo
   mi elementos
   adjuntar: primero,
   adjuntar: segundo.
}.
Par en: ['blanco'] hacer: {
  mi elementos  := Serie nuevo.
}.
Par en: ['nuevo'] hacer: {
>> par := yo nuevo.
<- par blanco.
}.

Par en: ['texto'] hacer: {
<- ['primero y segundo']
  primero: mi elementos primero,
  segundo: mi elementos último.
}.
>> par := Par nuevo de: ['pimienta'] y: ['sal'].
Salida escribir: par primero, detener.
Salida escribir: par segundo, detener.
Salida escribir: par, detener.
```

Apéndice C: Administración de memoria

Este capítulo puede ser de interés únicamente para aquellos lectores que deseen configurar manualmente su propia versión de Citrine. Los lectores que estén interesados únicamente en el lenguaje Citrine pueden ignorar este capítulo. Se supone que los lectores tienen algunos conocimientos básicos sobre administración de sistemas.

Citrine admite varios parámetros para la administración del sistema. En ausencia de parámetros, Citrine comenzará de manera predeterminada con un límite de memoria de 10 MB. Se pueden utilizar las siguientes variables de entorno para modificar la configuración de memoria en Citrine:

administración:

CITRINE_MEMORY_LIMIT_MB
CITRINE_MEMORY_MODE
CITRINE_MEMORY_POOL_SHARE

Para especificar el límite de memoria en MB, utilice CITRINE_MEMORY_LIMIT_MB. Utilice CITRINE_MEMORY_MODE para establecer el modo de administración de memoria (1 de manera predeterminada)

Valor	Significado
0	La memoria no se limpia. Esta configuración garantiza que la memoria no se limpie durante la ejecución del programa. Esto beneficiará la velocidad, sin embargo la memoria se llenará rápidamente
1	Se está limpiando la memoria a medida que se muestra el valor límite determinado. Esta configuración predeterminada será suficiente para la mayoría de los programas.
4	La memoria se va limpiando en cada paso del programa. Esto provocará un uso reducido de la memoria, pero es muy intensivo. La velocidad de ejecución del programa se verá afectada profundamente.
8	Modo experimental que involucra grupos de memoria

También es posible combinar valores (12=8+4). Durante la ejecución del programa, se pueden modificar los parámetros de memoria, como se indica en el capítulo 3.12. Un bloque de memoria compartida (8) solo se puede crear una vez durante la ejecución de un programa. No es posible cambiar constantemente esta configuración durante el tiempo de ejecución del programa.

Registro

añadir: 93
y: 32
respuesta..........................26
adjuntar: 68, 83
aplicar:y: 26
argumentos: 97
argumentos.......................97
pedir..................................98
pedir contraseña................102
entre:y: 47
mensajes binarios..............20
Algebra booleana.............. 42
por: 78
mayúsculas.......................55
en-caso-de:hacer: 66
caracter:............................53
caracteres.........................69
limpiar............................... 104
código............................... 134
unirse: 70
Comando..........................96
comparar: 52
contiene: 53, 77
continuar...........................44
copiar................................110
contar: 69
día....................................88
día-del-año........................ 88
día:...................................89
eliminar.............................84
hacer.................................138
hecho................................138
cada: 73, 78
entonces:caso-contrario: .44
otro: 42
finalizar............................. 103
entradas............................79
es-igual-a: 66
error:.................................60, 97
capturar: 59
existe................................84
Falso.................................42
falso: 22, 42
Archivo 83
llenar:con: 71
encontrar: 54, 72
primero............................. 70
limpiar.............................. 97
desde:longitud: 54
tiene: 78
Hola Mundo...................................... 13
hora.. 88
hora: 89
ifs... 22
palabras claves................................ 19
último.. 70
último: 54
aprender:significa: 158
longitud.. 53
Listas... 76
bucles.. 22
minúsculas....................................... 55
pastilla... 25
operaciones matemáticas................. 47
maximo... 73
memoria.. 103
orden: 10 104
mensaje:argumentos: 140
mínimo... 73
minuto.. 88
minuto: 89
modulos.. 143
Momento.. 87
mes.. 88
mes: 89
nuevo... 28
Nulo... 40
ni: 43
Número.. 19, 47
objecto... 136
Objecto.. 19, 28, 64
en:hacer: 26, 66
o: 43
sistema: 95
objeto salida.................................... 40
parametros....................................... 23
ruta... 84
Ruta.. 85
penúltimo.. 70
bruto... 48, 87
toma-el-último.................................. 72
posición: 70
anteponer: 71
procedure... 60
Programa.. 95
Programa entrada............................ 97
propiedad... 28
poner:en: 71, 76
calificador.. 49
calificador: 49

comillas.................................. 51
leer.. 83
recursivo............................... 124
reemplazar:longitud:con: ... 74
reemplazar:con: 52
responder: 118
responder:y: 119
responder:y:y: 119
responder:y:y:y: 119
segundo................................ 88
segundo: 89
yo.. 26
Series.................................... 68
asignar:valor: 62
configurar:............................. 98
establecer:valor: 98
toma-el-primero 72
saltar: 55
ordenar: 74
dividir: 69
empezar............................... 60
detener................................. 40
interpolación de cadena........ 24
restar: 93
Tarea... 19, 58
plantilla.. 24
Texto.. 19, 51
esta-tarea...................................... 60
tiempo.. 91
recortar.. 55
Verdadero...................................... 42
verdadero: 22
tipo... 145
mensaje unitario............................ 19
utilizar: 95
valores... 79
variable.. 18
esperar: 93
semana.. 88
día-semanal................................... 87
mientras:.. 23, 58
escribir:.. 40, 83
año.. 87
año: 89
zona: 90