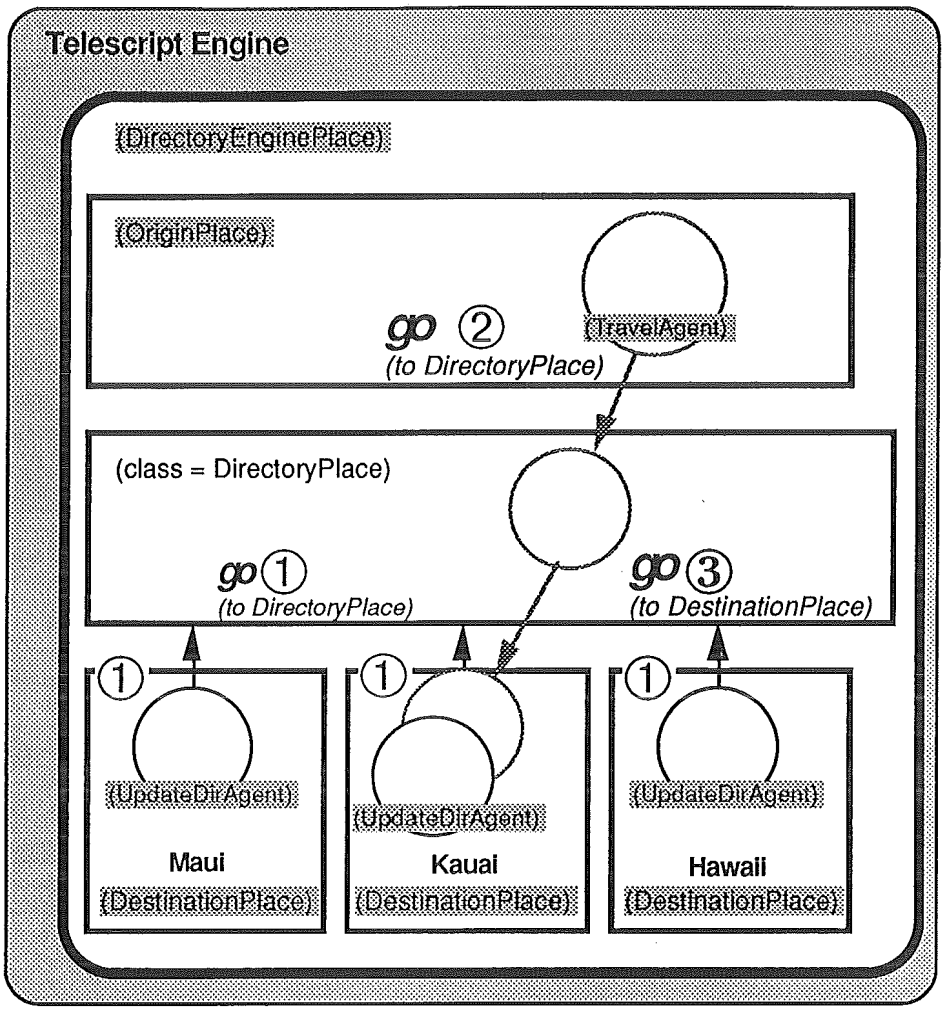

Telescript Engine
(DirectoryEnginePlace)
(OriginPlace)
go ②
(to DirectoryPlace)
(TravelAgent)
(class = DirectoryPlace)
go ①
(to DirectoryPlace)
go ③
(to DestinationPlace)
①
(UpdateDirAgent)
Maui
(DestinationPlace)
①
(UpdateDirAgent)
Kauai
(DestinationPlace)
①
(UpdateDirAgent)
Hawaii
(DestinationPlace)

7 Building a directory of places

Starting file: `07_nameDirectory.t`

This example shows how to set up a simple directory as you might use to send Mail agents from place to place within a cloud. We'll begin with a DirectoryPlace that maintains a list of Telenames, each one of which has a corresponding ASCII string "key"[4].

This script also creates multiple DestinationPlace objects. Each DestinationPlace must "register" itself by sending an agent (of class UpdateDirAgent) to the DirectoryPlace. Each registered place adds its Telename to the DirectoryPlace's list of destinations.

There's one more type of place we have to consider, and that's an OriginPlace. The OriginPlace can create TravelAgent objects and provide them with a "destination" string. Each TravelAgent object then goes to the DirectoryPlace, and uses its string to look up a Telename, creates a ticket based on that Telename, and goes to the specified DestinationPlace.

For this exercise, extend the basic script in four ways:

1) Modify the DirectoryPlace class so that a key-value pair can't be added to the directory if a pair already exists with the same key.

2) Extend the DestinationPlace class so that each DestinationPlace object is created with a string that a TravelAgent object can get as a souvenir of its trip to that place. (The TravelAgent should print out this string using `dump()`.)

3) Create a MultiTravelAgent class that accepts a list of destination names and goes to each of the destinations in that list.

4) Create a BroadcastAgent class that accepts a list of destination names and sends clones to each of those places.

This script introduces Dictionary objects, which are used frequently in Telescript programming, and the send command for cloning an agent.

There's even more excitement in the air after reading this script because it gives a taste for how the Magicmail name directory works.

The next diagram shows how the nameDirectory should work once you complete this exercise.

---

[4] In other words, a Dictionary object.

6 Setting up a meeting

Starting file: `06_stringDelivery.t`

This script creates StringBearer objects that carry strings from a StringStart place to a StringDump place in order to meet with a StringDumpster object there and deliver strings to it.

To complete this lab:

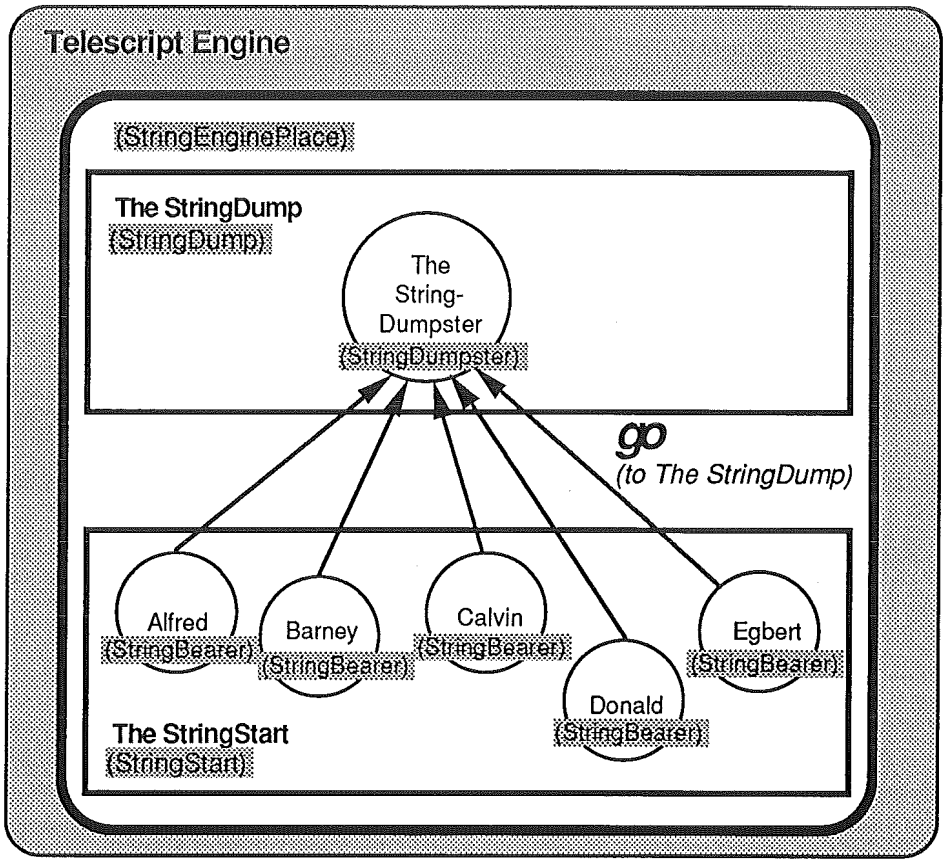
1) Modify the StringBearer objects to return to the StringStart after delivery.

2) Create a new class of TwoStringBearer objects.

3) Modify the StringDump class to allow only StringBearer objects to enter.

The script is a source of real excitement when you think about StringBearer objects as couriers, String objects as messages, and StringDumpsters as mailboxes.

The following diagram shows where the StringBearer objects go:



Hint: You can use one of `entering`'s parameters to determine the Agent's origin.[3]

5.2 Even more optional code

4) If you are feeling brave, try sending the agent to some place that doesn't exist. ("Atlantis" is an interesting choice.) What happens?

[3] One more hint: these parameters are unnamed in the source file, so you'll have to assign names to one or more of them. Remember that a named parameter can not follow an unnamed parameter, so if you give one parameter a name, you might have to name some of the others.

5 Creating a simple Agent

Starting file: `05_simpleAgent.t`

This exercise introduces you to Agents in Telescript.

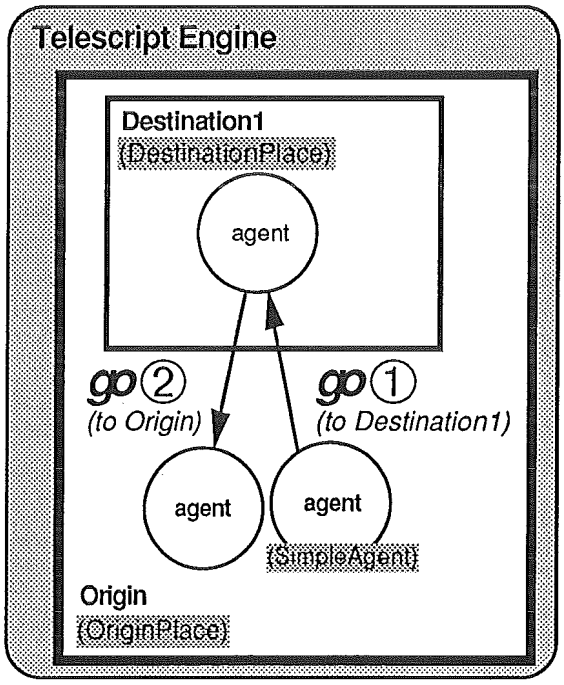
To complete this lab:

1) Add a property to your Agent which lists its destination[2]. Pass this value in as an argument to `initialize`.

2) Modify the live method of the SimpleAgent class so that agents go to the specified destination then return to the OriginPlace.

There are lots of new things here: tickets, interesting **live** methods, **go**, and Low Telescript mingling with High. There is also a modified version of the StrVal class that does some special formatting on Telename objects.

The following diagram shows where the SimpleAgent will go.



5.1 Optional code

3) Change the `entering` method to check whether the entering agent is being created or is traveling in from another place. The Telescript Language Reference Manual describes how to detect this difference.

---

[2] We suggest that you use a String containing the destination's name

3 A simple calculator

Starting file: `03_integerCalc.t`

This exercise contains a simple calculator which can perform basic arithmetic operations.

To complete this lab:

> Extend the simple calculator to include additional integer-based functions such as negation, integer division, truncation, and modulus.

Test cases are provided.

New ideas in this script: defining the access to operations using **public** and **private**, and exception handling

4 Formatted output

Starting files: `04_stringConvert.t` and `03_integerCalc.t`

Before you do anything else, run the Lab script and the solution, then study the Lab script. Note especially how the StrValDemo class uses the StrVal class to create formatted output.

To complete this lab:

1) Copy the StrVal class into your solution for the `03_integerCalc.t` exercise.

2) Use StrVal to format the output for the integer calculator.

The desired results are shown at the end of `04_stringConvert.t`.

1 Implementing "Factorial"

Starting file: `01_factorial.t`

This script shows one implementation of the factorial algorithm that uses a Telescript for loop.

To complete this lab:

> Write the program again using the other Telescript looping constructs: **repeat**, **loop**, and **while**.

1.1 Optional code

For extra credit, write factorial using a recursive algorithm. This requires that you define a factorial operation, and you'll have to create a new class to hold your operation. (You'll also need a **do** statement containing Low Telescript to create an instance of the class and invoke your factorial operation.)

Note: When you create a new class, the result will look something like this:

```
*@Place.publicPackages.include(
      moduleName: module = (
            // Your classes here

            factFinder: class(Object) = (
                  initialize: op() = {^};
                  fact: op(n: Integer) Integer = {
                        // You have to write this
                  };
            );
));

do {
  // Low Telescript code which creates a FactFinder
  // object and calls fact(3)
  << 3 [:factFinder] fact >>;
};
```

2 Manipulating a List

Starting file: `02_reverse.t`

For this exercise, use the attributes and operations of the Telescript **List** class to reverse the order of elements in a given list.. This script shows how List objects are declared and initialized.

To complete this lab:

> Complete the skeletal procedure given in `02_reverse.t`.

0 Using these exercises

This document contains a series of programming exercises which supplement General Magic's *Telescript Language Programming* course.

0.1 Locating an exercise

Each programming exercise is supplied in two files: a "Lab" file which contains the starting point for an exercise, and a finished "Solution" file. The initial files are in the Labs directory on your computer, while the final solutions are in the Solutions directory.

You should try running both the Lab and Solution files for an exercise before making any changes to the exercise itself.

0.2 Compiling and running an exercise

To compile an exercise, use the hts (High Telescript) command:

```
hts 01_sample.t
```

This command converts a High Telescript file into a Low Telescript file (with a .s extension), so 01_sample.t is translated into 01_sample.s.

To compile and run an exercise, run the High Telescript compiler with the -x (execute) flag:

```
hts -x 01_sample.t
```

This will compile the exercise, then run the Telescript engine (eng) with the resulting script. [1]

0.3 Using the Telescript "Front Panel"

If you run the Telescript engine by itself (i.e. without setting up a "cloud"), the engine communicates with your terminal through a simple "front panel." The Front Panel accepts commands for the engine and Telescript debugger, but not for the running script.

---

[1] If you want to run the engine yourself, the command is:

```
eng -o - 01_sample.s
```

where eng is the Telescript engine, -o - is a command to send all output to the terminal, and 01_sample.s should be replaced with the name of your compiled (Low Telescript) file.

Table of Contents

Telescript™ Language Exercises

Telescript Language Programming Course

Version 1.1

8 August 1994

Richard Clark
Susan Rayl

General Magic Confidential

General Magic, Inc.
2465 Latham Street, Suite 100
Mountain View, CA 94040