Telescript Programming Guide

Version 0.5 (ALPHA)

October 19, 1995

General Magic Confidential

General Magic, Inc.
420 North Mary Avenue
Sunnyvale, CA 94086

General Magic, Inc.
420 North Mary Avenue
Sunnyvale, CA 94086 USA

Tel.: 408 774 4000
Fax: 408 774 4010
E-mail: dev-info@genmagic.com
URL: http://www.genmagic.com/

Table of Contents

1. Introduction

Welcome to the world of Telescript programming! This programming guide is designed to help you learn to program in Telescript. It explains and illustrates proper usage of features in the Telescript language that tend to trip up those new to the language.

1.1. Where Do I Start?

The Telescript Developer Environment comes with four manuals:

- The TDE User Guide
- The Telescript Language Reference
- The Telescript Programming Guide
- The TDE Developer Libraries Reference

The TDE User Guide describes the development environment. It contains a tutorial that will introduce you to the Tool Manager, the Compiler Tool, the Browser, the Cloud Manager, the Module Manager, and the Debugger. When you are ready to try out the Telescript Developer Environment, you should read this manual. You should also read this manual to learn how to break your Telescript source into individual files.

The Telescript Language Reference documents the Telescript language. It covers both the syntax of Telescript and the set of classes defined as part of the language. You need to read the syntax portion before reading this manual. This manual contains Telescript code fragments.

The TDE Developer Libraries Reference documents the libraries that are shipped as part of the Telescript Developer Environment. It contains both explanatory text and class reference material.

This document is the third manual in the set. This first chapter provides a roadmap for this document. The next three chapters each focus on one aspect of programming in Telescript.

If you like to learn by doing, skip this document for now. Read and try out the tutorial in the TDE User Guide. Come back to this document when you get stuck debugging your code.

If you like to study first, go ahead and read Chapter 2 now. Chapter 2 explains aspects of Telescript that are likely to remain mysterious even after reading the Telescript Language Reference.

Chapters 3 and 4 look at more specialized facets of Telescript programming. You shouldn't worry about these right away. Write some simple Telescript code, try out the sample applications, and then come back.

1.2. A Quick Review

If you've read the TDE User Guide and Chapter 3 of the TDE Developer Libraries Reference, you can skip this section. If you haven't, here's a brief introduction to Telescript, the kinds of files that make up a Telescript program, and the tools you will use to produce a Telescript program that you can run.

Telescript is an object-oriented programming language. Everything is an object. Even classes are objects. The Telescript compiler knows about the classes defined as part of the Telescript language. If you want to use a library class or a class you have defined, you must include the class's interface in your Telescript source file. A class interface serves the same function as a C function prototype.

This Telescript Developer Environment contains a set of makefile rules and compilation tools that assume you will use the following file naming conventions:

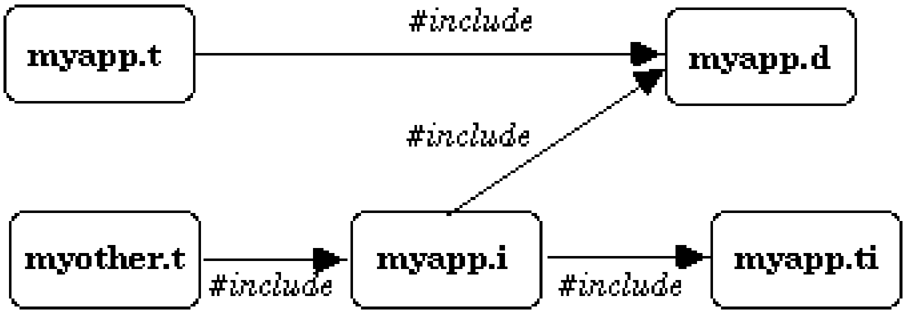
Extension	Usage
.t	A Telescript source file containing class definitions or free-standing code.
.i	A Telescript header file designed for inclusion by source files needing the classes defined by the corresponding ***.t*** file.
.d	A Telescript header file designed for direct inclusion in the Telescript source file defining some classes and for direct inclusion in the corresponding ***.i*** file

This Telescript Developer Environment creates files with the following extensions:

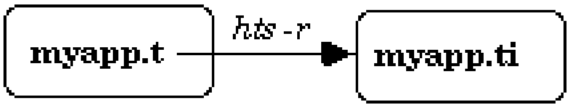
Extension	Usage
.s	A compiled Telescript file that can be interpreted by the Telescript engine. Note that if there is also a file with a ***.tsp*** suffix, you should submit the ***.tsp*** file and not the ***.s*** file. The ***.s*** file will just define a dummy class.
.tsp	A compiled Telescript file that can be interpreted by the Telescript engine. Files with the suffix ***.tsp*** contain only free-standing Telescript. They do not define classes.

.ti A Telescript header file automatically generated by the Telescript compiler. This file contains Telescript class interfaces and macros mapping to the internal names of Telescript classes.

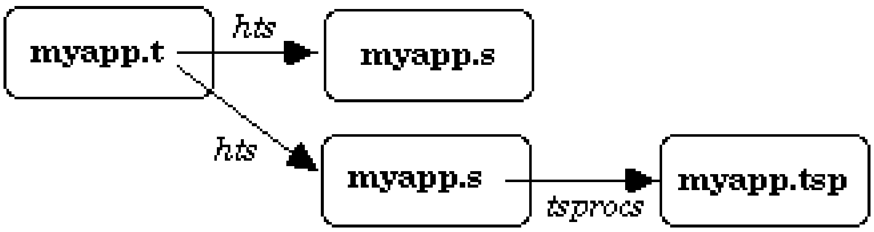
Consider a source file ***myapp.t***. The header file with extension ***i***, ***myapp.i***, continues to be designed for inclusion from other source files such as ***myother.t***. The defines and calls to nested include files that might be needed to compile ***myapp.t*** are broken out to a separate file called ***myapp.d***. The source file ***myapp.t*** includes ***myapp.d*** instead of ***myapp.i***.



The interfaces for the classes defined in ***myapp.t*** are defined in the file ***myapp.ti***. ***myapp.ti*** is created by the Telescript compiler.

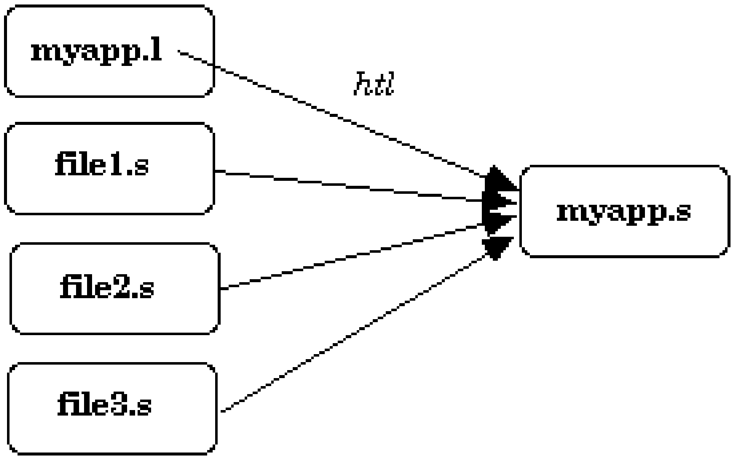


You must run the Telescript compiler ***hts*** on your Telescript source to produce a file that can be interpreted by a *Telescript engine*. Unfortunately, the present Telescript compiler (the one you received with this TDE) doesn't know how to compiler code using library or developer-defined classes in free-standing code. (Free-standing code means Telescript code that can be submitted to the Telescript engine to create instances of objects. Code that isn't free-standing just defines classes.) You will need to run the post-processing program ***tsprocs*** on the output of ***hts*** to get free-standing code.

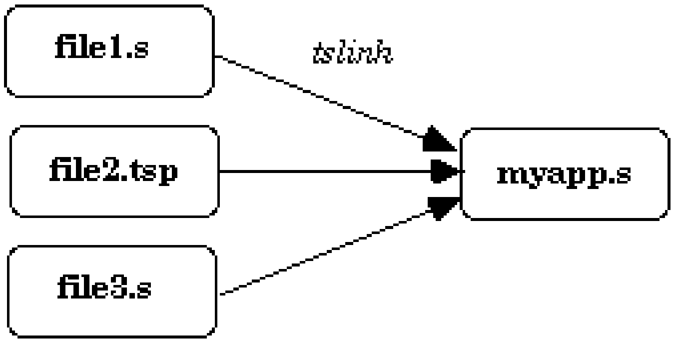


You can load either ***.s*** or ***.tsp*** files. The ***.s*** files produced by compiling free-standing Telescript, however, will do nothing useful. The ***.s*** file will define the class DummyClass. It is the ***.tsp*** file that contains the code you really want to execute.

The compilation tools also include a pair of linkers that combine multiple compiled files into a single interpretable file. The linker ***htl*** is to be used only with compiled class definitions. You will need a linker input file to guide the linking process.



The other linker, ***tslink***, does not require an input file and can handle a mix of compiled class definitions and free-standing code.

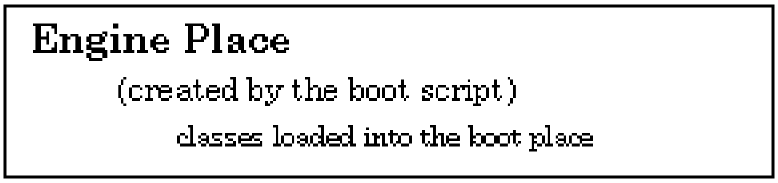


In the Telescript language, instances of the classes *Agent* and *Place* act as processes or programs. Both have a thread of control that executes a **live** method. A place represents a virtual location. Agents can come together within a place to interact with one another or with the place itself. Every agent lives within some place. Every place except the outermost one lives within another place, too.

A Telescript engine can support many agents and places. The outermost place, however, has some unique characteristics. The outermost place is the only one that doesn't reside within another place. In addition, its owner (as identified by its authority) is granted special power as the superuser for the whole Telescript engine. Because of this special role, the outermost place must be a member of the class *EnginePlace*. An engine place in many ways acts as the operating system for a Telescript engine.

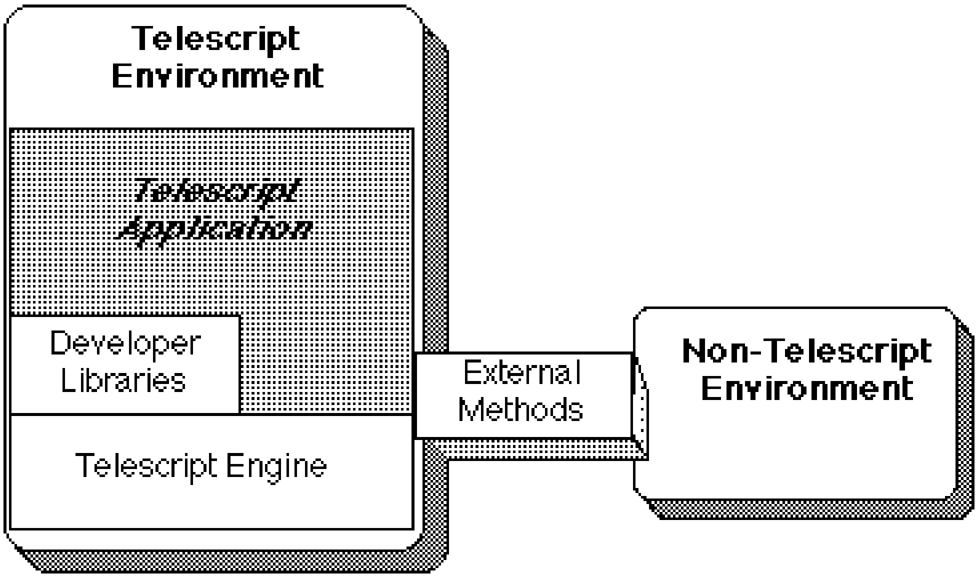
When a Telescript engine is first started, it executes the boot script within a subclass of EnginePlace. The boot script is executed by a *boot place*. The boot place's class is built into the Telescript engine. It acts almost as if it were an instance of the class EnginePlace. The TDE Developer Libraries provide a boot strap script, ***boot.s***, to be executed by the boot place.

When the engine place is created, the boot place vanishes. The real engine place appears instead. Normal engine place creation scripts pass on to the new engine place all the classes known to the boot place.



The TDE Developer Libraries contain two EnginePlace subclasses, *TDKEnginePlace* and *TDKGUIEnginePlace*. The default cloud uses the TDKGUIEnginePlace because this engine place has a GUI console.

The Telescript engine and the family of developer library classes collectively known as the External Application Framework (or EAF) allow Telescript programs to communicate with non-Telescript programs. The classes implemented as part of the *external methods* group provide operations that let Telescript code open a TCP/IP socket to some other program. Using external methods, Telescript code can interact with programs written in other programming languages. A non-Telescript program that interacts with Telescript is called an *external process.*



The External Application Framework contains a framework for defining C++ analogs to Telescript classes. The C++ analogs contain the data stored in the Telescript classes. They do not implement the functionality of Telescript.

The External Application Framework also has a translation and rpc framework that makes it possible to make remote procedure calls from Telescript into C++ code. Arguments are converted into C++ objects. The C++ result object is re-created as a Telescript object.

<u>Heads Up</u>

This is the next to last major release of the TDE that will support external methods. External methods is being phased out in favor of a different mechanism that provides greater security.

1.3. Review of Coming Attractions

This chapter has laid the foundation for programming in Telescript. It reviewed the kinds of files you will be creating, the compilation process, and the resulting output files. It also explained a few key concepts: the boot place, the engine place, and the External Application Framework.

You should read the next chapter, Chapter 2, as soon as your are ready to start programming. Chapter 2 clarifies aspects of Telescript programming that have proven confusing to those new to the language. It also describes some common programming pitfalls.

Chapters 3 and 4 contain more advanced topics. You don’t need to read them right away. Chapter 3 teaches everything you need to know about the intermediate code into which Telescript is currently compiled (i.e., Low Telescript).

Chapter 4 is likely to be of interest at either the design phase or after you have a working prototype. Chapter 4 describes techniques for making your Telescript programs more secure and safe.

2. Programming in Telescript

Telescript is a new programming language. The language and its compiler tools are in their first generation. Like any first generation computer language and software, Telescript 1.0 has its limitations. This chapter contains a more detailed discussion of areas of the language that have proven confusing to new Telescript programmers and explains how to work around the limitations of the 1.0 tools.

2.1. The Telescript Compiler

The Telescript compiler ***hts*** converts Telescript source files into output files that can be interpreted by the Telescript engine. Version 1.0 of the Telescript compiler has limitations. This section explains how to work around the common problems.

2.1.1. Modules and #include

The Telescript 1.0 compiler allows you to use the **`#include`** directive to include other files. With the command line argument ***-r***, the Telescript compiler ***hts*** will generate interface definition files for you to include. You will need to include interfaces for all the classes referred to by the code in your source files.

Version 1.0 of the compiler has one big limitation in its handling of interfaces. It forgets each interface as it has been read if the interface appears outside of **`module`** declaration. It forgets all interfaces declared in a module at the end of that **`module`**.

Suppose you have a header file *fruit.i* containing the following:

```
Fruit: interface(Object) = (
     public
          name: String;
          initialize: op(name: String);
);

Apple: interface(Fruit) = ();
```

Suppose you have a source file *makepie.t* that looks like:

```
#include "fruit.i"

MakePieModule: module = (
     PieMaker: class(Object) = (
          public
               makePie: op(content: Fruit) = (
                    /* omitted from this code sample */
               };

     );
);
```

If you try to compiler *makepie.t*, the compiler will complain first that it doesn't know the superclass of Apple. It will tell you that Apple was defined in *fruit.i*. The compiler will also tell you when it compiles the operation **makePie** that it doesn't know class Fruit. The problem is that the **#include** line occurred before the **module** line. To make this sample compile, *makepie.t* would need to be rewritten as:

```
MakePieModule: module = (

#include "fruit.i"

     PieMaker: class(Object) = (
          public
                makePie: op(content: Fruit) = (
                      /* omitted from this code sample */
                };

     );
);
```

2.1.2. Interfaces and Circular References

The Telescript 1.0 compiler needs to see the class definition or interface for every class before the class can be used. If the compiler complains that a class isn't known, you've done one of the following:

- forgotten to implement the class
- made a typo in the class name
- forgotten to include the file holding the class's interface
- put the #include line outside of your module block
- defined the class in a difference source file whose interfaces haven't been generated yet.

Check first for the first three problems. The previous section tells you how to deal with the fourth. To fix the sixth problem, try the following first:

- If the class is in another source file, check the order of files in the `AUTO_INCS` macro in your makefile. Move the name of the file defining the missing class before the name of the file that isn't compiling.

The best way to fix other kinds of circular dependencies is to remove them by defining a mixin class that defines the API for one of your classes. Update your sources to build the mixin class first.

Consider the following sample. The file *pieplace.t* defines a PieShop place. This place will sell a pie if it has one. Otherwise, it will create a PieBuyer agent and send it to another pie shop to buy the pie. The PieBuyer agent is defined in *piebuyer.t*. It uses the buyPie operation of class PieShop to buy the pie.

```
PieShop: class(Place) = (
     public
          shellPie: op() Pie = {
                 aPie: Pie | Nil;

                 aPie = *.getPieFromStock;
                 if (aPie = nil) {
                       PieBuyer(*.distributorTicket,
                                Permit(nil));
                       aPie = *.waitForPie();
                 };
                 return aPie;
          };
     );

PieBuyer: class(Agent) = (
     public
          live: sponsored op() = {
                 *.go(*.destination);
                 myPie = place@PieShop.sellPie();
                 *.go(*.originPlace);
          };
);
```

You could define the pie-purchase API in a new mixin class. Suppose you call this new class SellsPies. It is defined in the new file *pieapi.t*.

```
SellsPies: mixin class(Place) = (
     public
          sellPie: op() Pie = {
                 throw NotImplemented();
          };
     );
```

You can now include *pieapi.i* in your old source file. Note that PieShop is now a subclass of SellsPie. The PieBuyer agent now asserts it the current place is a member of SellsPies instead of PieShop.

```
#include "pieapi.i"

PieShop: class(Place,SellsPies) = (
     public
          sellPie: op() Pie = {
                 aPie: Pie | Nil;
```

```
                aPie = *.getPieFromStock;
                if (aPie = nil) {
                      PieBuyer(*.distributorTicket,
                               Permit(nil));
                      aPie = *.waitForPie();
                };
                return aPie;
          };
     );

PieBuyer: class(Agent) = (
     public
          live: sponsored op() = {
                *.go(*.destination);
                myPie = place@SellsPies.sellPie();
                *.go(*.originPlace);
          };
);
```

You are not required to define a separate API class to resolve circular dependencies. On your **module** line, add an **imports** clause:

```
PieModule: module imports PieShop = (
     /* define class PieBuyer */

     /* define class PieShop */
);
```

This syntactic form is not forward compatible with Telescript 1.1. The compiler for Telescript 1.1, however, will be able to handle circular references within the same file. You will simply need to strip the **imports PieShop** from the source file when upgrading to Telescript 1.1.

2.1.3. Dealing with a Confused Compiler

Version 1.0 of the Telescript compiler is confused by sufficiently complex statements. The most common circumstances for this are:

- Escalation of **initialize** when a class has mixin superclasses
- Creation of an instance of a class that has mixin superclasses and does not explicitly define the signature for **initialize**.
- Complaining that an operation is not defined when you are confident it is defined.
- Using the **[]** syntax on a dictionary or list computed as the result of an expression
- A ***.ti*** file needs to be rebuilt, but hasn't been.

The first two problems will be discussed in section on the operation initialize. Skip ahead if those are the problems causing you grief. The last problem can be cured by deleting all ***.ti*** files and trying again.

The Telescript compiler enforces type restrictions, based on the declared type of a variable, attribute, argument, or result. The actual value, however, can be a subclass of the declared type. Suppose you define a class with an operation **listify** that creates a list holding its argument(s). The arguments are declared as being objects and so the result is declared as returning a list of objects.

```
listify: op(theArgs:  Object...) List[Object,Equal] = (
     return List[Object,Equal](theArgs...);
};
```

Suppose you have code that passes two strings into this operation. You want to get back the second item in the returned list:

```
valueTwo: String;
valueList: List[Object,Equal];

valueList = MySillyClass().listify("alpha", "beta");
valueTwo = valueList[2];
```

The compiler doesn't know that the implementation of **listify** returns a list containing the arguments. The compiler will normally complain if you attempt to assign the second value of the returned list to a string variable. You can, however, use the **@** syntax to tell the compiler that you know the the expression **valueList[2]** returns a String.

```
valueTwo = valueList[2]@String;
```

Most of the time you can use the **@** syntax to tell the compiler about the real type of an object. The compiler, however, will not let you use **@** to coerce to a class that isn't a subclass of the declared type. There are also occasions in which the compiler just gets too confused by an expression to accept **@**. If that happens, you may have to resort to dropping down into Low Telescript. Read the next chapter for information on using Low Telescript.

If you have used a complex expression to compute a list or dictionary, particularly an expression that uses **@**, the compiler may tell you that use of the **[]** syntax is invalid. You can work around this by explicitly calling the operation **get** or **set**. You can also assign the result to a variable of type List or Dictionary and then use the variable. If the following doesn't compile:

```
valueTwo: String;

valueTwo =
     ((/* complex expression */)@List[String,Equal])[2];
```

try:

```
valueTwo =
     ((/* complex expression */)@List[String,Equal])
          .get(2);
```

or:

```
tmpList: List[String,Equal];

tmpList =
     (/* complex expression */)@List[String,Equal];
valueTwo = tmpList[2];
```

2.2. Loading Class Definitions

In Telescript, a class is just another object. You can look at its public attributes. Agents can make copies and take their copies to other places. Classes can even be carried to another Telescript engine and then used.

A Telescript class is created by writing a Telescript source file containing a module implementing one or more class definitions. When you compile a module defining classes, the Telescript compiler generates code that will create a Package containing your classes.

Simply creating a package of class objects doesn't do you much good. You normally need to do something with the classes. You can put an extra wrapper outside the module declaration to specify what you want to do with your package of classes.

Normally, you want to install your classes in such a way that the rest of your code can use them at need. How does the rest of your code find classes? Let's take a look at what happens when you write a code fragment that creates an instance of the class MyGreetingClass.

```
HelloObject:  class(Object) = (
     public
          sayHello:  op() = {
                helloGreeting:  MyGreetingClass;

                helloGreeting = MyGreetingClass("hello");
                helloGreeting.dump();
          };
);
```

Each class object keeps a *vocabulary* listing the symbolic names and internal names of the classes its uses. The vocabulary for the class HelloObject for this release of the TDE will contain an entry remembering that you used the identifier MyGreetingClass to refer to the class whose internal name is 4d794772656574436c6a7373.(Forthose of you fluent in the hex values for ASCII, yes, the internal name for classes in Telescript 1.0 is just the binary equivalent of their identifiers.)

The compiled code for HelloObject uses the symbolic form to identify the class. When this code is executed, the Telescript engine looks the symbolic form up in the vocabulary to obtain the internal name of the class. It uses the internal name to search for an installed package of classes containing a class by that name.

That brings us back to issue of installing a package of classes. What exactly does this mean?

Every agent and place has private attributes named **publicPackages** and **privatePackages**. Both attributes hold lists of packages. The packages can contain classes, but are also allowed to hold packages of other objects. When a proces (i.e., an agent or place) executes its own code or the code of some other object, the Telescript engine looks up a class's internal name first in the process's own **privatePackages**. If the engine doesn't find the class in the list of **privatePackages**, it will search the **publicPackages** list. If the class isn't in either attribute, the engine will search the **publicPackages** of the place in which the process reside. The engine repeats the search of **publicPackages** in the place containing the place until it either finds the class or has searched the engine place.

Installing a class, then, means putting a package containing the class into a **publicPackages** or **privatePackages** list somewhere on this search chain. The most convenient place to install a packages of classes is in the **publicPackages** of the engine place.

<u>Key Classes Implemented in the Developer Libraries</u>

```
TDKPackageControlled: mixin class(Process) = (...);
```

<u>Key Macros Defined by the Developer Libraries</u>

```
ANY_SCRIPT_PRELUDE
ANY_SCRIPT_CONCLUSION
LINK_FOR_ANY_SCRIPT
```

Installing a package of classes into the engine place may sound trivial. After all, **publicPackages** is just a list. The class List has many public operations that let you add items to the list (**include**, **add**, **append**, **appendElement**). One problem is that **publicPackages** and **privatePackages** are private attributes. Only code implemented as part of the engine place's class can use these attributes. The library class *TDKPackageControlled* provides a solution to this dilemma.

The class TDKPackageControlled defines public operations for updating the publicPackages attribute. You can install a package of classes by calling either the operation **installPackage** or the operation **install**. The operation **install** takes one argument, the package you want to install. **installPackage** takes two arguments, a package and an identifier. If you are still at the prototype stage, either use the operation **install** or pass the identifier **'Public** into **installPackage**.

The operation **installPackage** will accept two other identifiers, **'Both** and **'Traveled**. You can use these identifiers only if you are installing a package into a place that also inherits from the class *Traveled*. If you want to know more about class Traveled or want to fine-tune your application to run on a multi-engine cloud, read the section appearing later in this chapter on engine place issues.

The public operations of TDKPackageControlled call the operation **canAdministerPackages** before touching the **publicPackages** attribute. The operation **canAdministerPackages** performs a simple access control check. It permits processes of either the engine place's authority or the authority of the TDKPackageControlled process to access the **publicPackages** attribute.

The engine place classes supplied as part of the TDE developer libraries inherit from TDKPackageControlled. You can install class packages into them if your installer agent is of the engine place's authority.

Let's step back a moment and review the situation. You, the developer, can write Telescript code that gets turned into a packages of classes that will be installed if you put the right wrapping around your module declaration. What wrapper?

The TDE developer library supplies a file, ***tl.i***, that contains many useful macros. The easy way to get a wrapper that will install your class properly under all the usual circumstances is by using the macros ANY_SCRIPT_PRELUDE and ANY_SCRIPT_CONCLUSION. Put the first macro after including ***tl.i*** and before starting your module declaration. Call the macro ANY_SCRIPT_CONCLUSION after your module declaration.

```
#include "tl.i"

ANY_SCRIPT_PRELUDE

MySourceModule:  module = (
     MyGreetingClass:  class(Object) = (
          public
                greeting:  String;

                initialize: op(greeting: String) = { ^; };
     );
);

ANY_SCRIPT_CONCLUSION;
```

If you are creating a single class library file by using the Telescript linker ***htl*** to link several compiled source files, you can omit the wrapper from each of the source files. However, you must supply an equivalent wrapper in your linker input file (by convention, linker input files have extension ***.l***). The macro `LINK_FOR_ANY_SCRIPT` expands to the proper incantation, However, the program ***htl*** does not support macro expansion. You'll have to manually cut and paste the expanded form of `LINK_FOR_ANY_SCRIPT` into your .l file. The result will look like:

```
Telescript 1.0 {
#MARK

#include "your_source_1.s"
#include "your_source_2.s"
/* you can have as many #include directives as you need
*/

/* next comes the value of LINK_FOR_ANY_SCRIPT: */

[ "Autoloader" asOctetString :ClassName ]
               * isMemberByName
[ "TDKPackageControlled" asOctetString :ClassName ]
            * isMemberByName or
     { :Exception
         { 'Public ~swap #place installPackage }
       catch #nil isEqual not
         { * publicPackages include }
       if }
     { 'Public ~swap * installPackage }
either

/* the value of LINK_FOR_ANY_SCRIPT is complete */

}
```

2.3. Free-Standing Telescript Code

Suppose you've defined the class for your application's place. Now you want to write a script that will create an instance of your application place. You'd like to write a piece of code that looks like:

```
#include "gameplace.i"

GamePlace(Permit());
```

If you try this, you'll find that the Telescript compiler will tell you it doesn't know class GamePlace. It doesn't matter that the header file ***gameplace.i*** contains the interface for the class. The key difference between writing a class definition and free-standing code is due to a limitation in the version of the Telescript compiler shipped with this release of the TDE. Any code that uses classes not defined as part of the base language can only be written as part of the implementation of an operation of some class.

This release of the TDE contains a special tool, ***tsprocs***, and a set of macros that you can use to work around this compiler limitation.

Key Macros Defined as Part of the Developer Libraries

```
BEFORE_INCLUDE
BEFORE_CODE
AFTER_CODE
```

These macros wrap your free-standing code in a faked class definition. The fake class definition satisfies the compiler. You need to include the header file ***tl.i*** at the start of your source file. Call the macro `BEFORE_INCLUDE` before including any of the header files that load class interfaces. Call the macro `BEFORE_CODE` just before your free-standing code. Add `AFTER_CODE` after the code.

```
#include "tl.i"

BEFORE_INCLUDE

#include "gameplace.i"

BEFORE_CODE

GamePlace(Permit());

AFTER_CODE
```

Do not use the macros `ANY_SCRIPT_PRELUDE` or `ANY_SCRIPT_CONCLUSION` in this file. A source file containing free-standing Telescript cannot define any classes.

The fake class definition introduced by the macros `BEFORE_INCLUDE`, `BEFORE_CODE`, and `AFTER_CODE` needs to be stripped out of the compiled form of your source. The TDE contains a special post-processor program named ***tsprocs*** that will do this for you. The set of makefile rules shipped with the TDE will automatically invoke first the compiler ***hts*** then ***tsprocs*** if you use the ***.t*** suffix for your source and the ***.tsp*** suffix for the compiled Telescript file to be loaded.

You can load and execute your compiled free-standing Telescript in the same ways that you load class definitions. Just be careful to choose acceptable names for your variables.

Variable name restrictions

When writing a boot script, you are normaly limited in your choice of variable names. The only safe variable names are **var1** through **var30**, **arg1** through **arg10**, and **exc1** through **exc10**.

The Telescript compiler ***hts*** does not always use the variable names appearing in your Telescript source in its output file. Variables declared at the beginning of an operation use the name appearing in the source file. Variables declared within a nested scope (e.g., in a `for` loop or `catch` statement) are changed so that variables with the same name defined in difference scope have different internal names.

The program ***tsprocs*** knows how to undo the changes made by the High Telescript compiler. You can safely declare your variables in any scope. Just be sure to limit your variable names to one of the choices mentioned earlier.

2.4. Linking and Loading Telescript Code

You can load a compiled Telescript file without passing it through a linker. If you do want to link two or more files, you need to pick a linker. This version of the Telescript Developer Environment contains two Telescript linkers, ***htl*** and ***tslink***. The linker ***htl*** combines compiled class definition source files in such a way that a single package is generated holding all the classes. The Low Telescript code in the linker input file (extension ***.l***) determines how the class is loaded.

Unfortunately, you can't always use ***htl***. The program ***htl*** deletes any Telescript outside of the code building a package of classes. The linker ***tslink*** does not delete code, but neither does it combine class definitions into a single package.

If you have forgotten about the boilerplate that loads your classes, you should go back and read Section 2.2 now. You might also want to read Section 2.9. The way in which you load your classes determines which classes travel with your agents to other Telescript engines.

This section looks at the kinds of problems that arise when trying to load or use classes. It points out common mistakes and gives some suggestions for diagnosing the problem.

2.4.1. Diagnosing the Culprit: SuperclassesInvalid

You've written your classes. They compile. They link. Now you try to submit them to the Telescript engine and all it tells you is "SuperclassesInvalid" or, even worse, "InternalException". There is no output saying which class is being defined or which superclass is invalid. What then?

The first thing you need to do is to increase the debug level for script interpretation. If you type:

```
lscr=6
```

to the Telescript engine's front panel and then resubmit the file loading your classes, the Telescript engine chatters about each class it has successfully created. It also will normally tell you which class failed to load. If not, the problem class is probably one in your source file right after the last class that worked. (The

Telescript engine does sometimes take classes slightly out of order if they have interdependencies.)

There following are the common reasons for getting SuperclassesInvalid:

- You haven't loaded the compiled form of the class
- You loaded the compiler output for the class, but you forgot to wrap your source file's module block in the right boilerplate. If you linked your sources into one file, the boilerplate in the linker file may be wrong.
- You linked your files in the wrong order.
- One of your classes inherits from the same class twice.
- You dropped into Low Telescript and wrote code with unbalanced **[** and **]**.

The common reasons for getting the exception InternalException are:

- One of your classes has the same name as a class that is part of the TDE's developer libraries or some other module you have loaded.
- You previously loaded a different version of your class.

Here's a little snippet of code that you can run to see if a class has been loaded. Create a file containing this code, change the string "MyClassName" to be the name of the class you are concerned about, and submit it using the Telescript Engine's front panel:

```
Telescript 1.0 {
     [  "MyClassName" asOctetString :ClassName] #place
findClass dump
}
```

This diagnostic code is written in Low Telescript. It asks the engine place to try to locate the class MyClassName and then to dump the result of its search. If you see "nil" in the output, the class has not been loaded into the engine place. Otherwise, the class has been loaded. If you want to wade through a complete listing of everything loaded into the engine place, you can submit the following diagnostic script:

```
Telescript 1.0 {
     * publicPackages dump
}
```

If the Telescript engine threw InternalException and the diagnostic script says the class was loaded, try doing make clean, remaking your source, and then cold booting the engine. If that doesn't work, you must be using the name of a class used in some other module. You'll have to rename your class.

If the class isn't loaded, you need to figure out why. The easiest thing to check is the link order. Be sure that the order in which you list files in #include directives in your linker input file (extension ***.l***) or on the command line for ***tslink*** uses the same order as compilation. Also double check that you didn't forget to list some file.

Suppose you've double checked the input to the linker. You've figured out which class is being built, but you're not sure what superclass is wrong. You need to trace through all of the superclasses. You can use the Browser tool to bring up a tree view of the class and its superclasses. Look for duplications. If there are no duplicates, use the diagnostic scripts just presented to find out which one wasn't loaded.

Once you know which superclass hasn't been loaded, you need to figure out why. Check the obvious first. Did you load the file that defines the superclass? If you thought you did, try again by submitting it to the front panel. Did the front panel complain about an exception? If so, fix that. Otherwise, use a diagnostic script to see if the class is now loaded. If the class is now loaded, you have either got boiler plate that doesn't work with the mechanism you used to install the class, a search path problem such that your configuration file (or your configuration file was never read), or your memory is wrong (you didn't try to load the class).

You can insert statements like:

```
"I am here" dump
```

anywhere in the linker boilerplate. You might want to try inserting the following just before linking boilerplate:

```
"Current process=" dump #process dump
"Current place =" #place dump
"Self = " dump * dump
```

If self is neither an Autoloader nor a class that inherits from TDKPackageControlled, the current place is of a class that does not inherit from TDKPackageControlled, and self is not the boot place (class EnginePlace), the boilerplate listed in ***tl.i*** will not work. Either rethink how you are loading your script or read the next chapter.

2.4.2. Where Did My Classes Go?

Your class definition code has been loaded, but when you try to create your application place or a test agent, your free-standing Telescript throws ClassUnavailable. What gives?

The usual problem is that the Telescript engine read your class definitions, but you had the wrong boiler plate. The classes weren't really installed. Follow the procedure outlined in the previous subsection for diagnosing this.

Another reason you'll see ClassUnavailable is when you try to use a Copyrighted class. (This is a good time to read about copyrighted classes in the TDE Developer Libraries Reference.) You will get the exception ClassUnavailable if the copyright enforcer object isn't installed (in a visible package) or if the call to the copyright enforcer's **isAuthorized** operation does not return true. You can use the operations of PackageControlled to see if the copyright enforcer object has been installed. If there is a copyright enforcer object, the problem is the result of **isAuthorized**. Try changing the authority of the agent or place trying to use the copyrighted class. You may be trying to do something only agents of the engine place's authority are allowed to do. It could be that you simply need to make an operation on an application place **sponsored**.

You will also see the exception ClassUnavailable if your agent travels to a remote engine and then tries to use a class that isn't in its vocabulary. You can work around this by having the agent execute code like the following before it leaves:

```
*.privatePackages.include(
     Package[Object,Object,Equal](nil, nil,
          'MissingClassOne, MissingClassOne,
          'MissingClassTwo, MissingClassTwo));
```

This code adds a package with the classes your agent needs to its **privatePackages** attribute. All classes contained in the **publicPackages** or **privatePackages** attributes will go with the agent.

If the class is still missing, then the problem is likely to be in the boilerplate. Use the front panel to turn up the transfer diagnostic level:

```
lxfr=6
```

When the agent leaves, you'll see output lines saying "Sending class: 414243". Use a table of ASCII to hex mappings or your favorite tool to convert these to class names, for example, "414243" is ASCII "ABC". Compare the classes sent with the one that your agent still couldn't find. If you don't see it, then the class was loaded into the engine place's **travelPackages**. Go back to the linker input file (or the source if not linked). In a source file, change `ANY_SCRIPT_TRAVELED_CONCLUSION` to `ANY_SCRIPT_CONCLUSION`. In a linker input file, change:

```
'Both ~swap installPackage
```

or:

```
install
```

to:

```
'Public ~swap installPackage
```

2.4.3. Why Is My Agent Carrying *That* Class?!

When an agent travels to another Telescript engine, it always carries the data in its properties. In addition, it may carry its own class, the classes in its vocabulary, the classes for all values in its properties, and any other classes pulled in by the vocubularies of classes being sent. A class will be sent unless one of the following is true:

- The class is defined as part of the Telescript language.
- The engine place inherits from Traveled and the class is stored in the engine place's travelPackages attribute.
- All objects refering to the class are frozen (and the class isn't in a vocabulary).

It is very easy for an agent to drag along either more or less than it should. An agent will try to drag along all of the objects referred to by variables in the all the methods on the call stack. If your agent is trying to drag along an unexpected class, look at variables. See if your code is clearing out variables containing unwanted data. If you add `|Nil` to the type declaration of a variable, you can clear it by assigning the value nil.

If you are sure that the remote Telescript engine will have your application-defined class, you can either change the boilerplate that loads the class or write a separate script that loads just selected classes into the engine place.

Use the macro `ANY_SCRIPT_TRAVELED_CONCLUSION` instead of `ANY_SCRIPT_CONCLUSION` if a source file is not linked using ***htl*** and you want none of its classes to be sent to another Telescript engine. If the source file is linked using ***htl*** and you want none of the classes in any of the linked files to be sent, copy the boilerplate of the macro `LINK_TRAVELED_FOR_ANY_SCRIPT`. Use this macro's value instead of that for `LINK_FOR_ANY_SCRIPT` in your linker input file (i.e., the file with extension ***.l***).

What do you do if there is a particular class that you don't want to send, but you do potentially want to send others in the same source file or in other files linked together using ***htl***? You can restructure your sources to isolate the classes you do not want to send. Alternatively, you can write a script that loads a package with selected classes. The following code fragment creates a package holding the classes MyClassOne and MyClassTwo. It is designed to be submitted via the front panel, the console (Thumper), as a script loaded by the engine place (listed in the value of configuration option **AutoLoadFiles** *),* or the debugger.

```
place@TDKEnginePlace.installPackage(
     Package[Object,Object,Equal](nil, nil,
          'MyClassOne, MyClassOne,
          'MyClassTwo, MyClassTwo), 'Traveled);
```

Freezing the right objects is awkward in Telescript 1.0, but that is the way to go if:

- A class is only pulled in from instances of the class.
- It's okay if you can't touch the instances of the class on the remote engine.

You need to construct a package with the classes you want to freeze out, put the package in the public packages of an appropriate agent or place, lock the objects of that class (or the objects containing them), and then pass the object and optionally the name of the package into a call to **freeze**. Read the documentation of the Telescript language for more information on **freeze**.

2.4.4. Why Isn't My Agent Carrying This Class?

The other problem you're likely to encounter when sending agents between Telescript engines are agents that try to take too few classes with them. If an agent is carrying too few classes, it will either fail its trip with OccupancyDenied or it will encounter the exception ClassUnavailable at its destination. The destination engine will generate diagnostic output about the class that was missing in the former case (assuming the script [**scr**] diagnostic level is high enough).

Version 1.0 of the Telescript language specifies that a class will be carried with an agent only if it is the class or or a superclass of the agent itself or one of the data objects being carried by the engine. Classes mentioned in the signatures of operations or used in operations are not automatically taken.

The Telescript engine does not know which operations will be used locally and which remotely. Rather than burden an agent with a potentially huge number of classes that aren't needed, the Telescript engine errs on the side of efficiency. You need to code your agent to specify which classes it really needs. You can do this by adding them to the agent's **privatePackages** or **publicPackages** attribute before departure.

The multi-engine sample shipped with this version of the TDE has a StockMarketProbe agent that goes to another Telescript engine to see if the remote engine is hosting a stock market service. For each stock market service found, the StockMarketProbe agent creates a StockMarketRecord object. This agent might also use the class StockName, so before it calls go, it executes the following code:

```
*.publicPackages.include(
    Package[Object,Object,Equal](nil,nil,
        'StockMarketRecord, StockMarketRecord,
        'StockName, StockName));
```

If the classes are in the **travelPackages** (of the engine place), the agent will just carry the names of the classes with it. If the classes are not in the **travelPackages**, the agent will carry the entire definitions of the classes.

2.4.5. Why Isn't My Code Being Executed?

You've loaded the classes for your application, but the code that is supposed to create your application place or a test agent does nothing. No exception. Absolutely nothing happens. But you're positive that you have submitted your code file. No, you are not going batty. This happens to the best Telescript programmers.

The common causes for this problem are:

- Using ***htl*** when you need to use ***tslink***
- Failing to run ***tsprocs***
- Loading a ***.s*** file instead of the ***.tsp*** file

The linker ***htl*** strips out free-standing Telescript. If you have dropped into Low Telescript outside of a **module** declaration or gave a ***.tsp*** file as input to ***htl***, this linker will merrily strip out your code without any warning. Check to see if you used ***htl***. If you did, try ***tslink*** instead.

If you are not linking your file or you did use ***tslink***, odds are that you either failed to run ***tsprocs*** on your compiled ***.s*** file to generate a ***.tsp*** file or you loaded the ***.s*** file instead of the ***.tsp*** file. The ***.s*** file generated by compiling free-standing Telescript written with the `BEFORE_INCLUDE`, `BEFORE_CODE`, and `AFTER_CODE` macros creates a fictitious class named DummyClass. Submitting the ***.s*** file causes the Telescript engine to interpret the definition of the class DummyClass. The class definition is parsed. The Telescript engine is happy. But the code in the class definition was not executed, so you are left confused.

2.4.6. But I Don't Want to Load That Class

Many applications are designed to consist of client and server components. The idea is to load only the client code into the client's Telescript engine. The previous section explained techniques that prevent a class from being sent from the client's engine to the server's, or vice versa. This section looks at techniques that help you control the vocabulary of your classes.

The implementation of class A uses a second class, B, if:

- B is a superclass of A
- Class A contains an "is B" clause
- Class A creates an instance of class B
- Class B is a subclass of Exception and class A contains a try/catch block catching class B.
- The code of class A uses class B as an object in an expression, for example, A.name or even just A.

The vocabulary and implementation currently do *not* include classes used only in the following ways. Only the compiler needs to know about the class. The Telescript engine does not.

- to declare the type of an attribute
- to declare the type of operation's argument or an operation's result.
- to declare the type of a property
- to declare the type of a variable
- following an @character asserting an expression's type

You can carefully code your class to take advantage of these rules. Consider the following sample. The file *bookshop.t* defines a BookShop place.

```
BookShop: class(Place) = (
     public
          orderBook: op(title: String; author: String) = {
                /* code to place the order would appear */
                /* here but isn't shown in this sample */
          };

);
```

The file *mybooks.t* defines a PersonalLibrary place and a BookBuyer agent. The PersonalLibrary place creates a BookBuyer agent that goes to the BookShop to special order a book.

```
PersonalLibrary: class(Place) = (
     public
          orderBook: op(title: String; author: String) = {
                BookBuyer(Ticket(nil,nil,BookShop.name),
                          Permit());
          };
);
```

You would like to be able to use the class PersonalLibrary on the client's Telescript engine without loading *bookshop.s*. The code that creates the ticket contains the expression:

```
BookShop.name
```

This expression uses the class BookShop. To do get rid of this dependency, you need to know about a feature of ***hts***. The Telescript compiler, when invoked with the command line argument ***-r***, generates macros that expand to the octet string name for each class. The macro name consists of the class name plus `_INTF_HTS`.

You can write the class PersonalLibrary to use the macro `BookShop_INTF_HTS` to create the ticket.

```
PersonalLibrary: class(Place) = (
     public
          orderBook: op(title: String; author: String) = {
                BookBuyer(Ticket(nil,nil,
                          ClassName(BookShop_INTF_HTS)),
                          Permit());
          };
);
```

2.5. The Operation **Initialize**

If you understand how to implement the operation **initialize**, you know almost everything you need to know to implement any operation. The complexity of **initialize** stems from three factors.

- You must escalate **initialize**
- The arguments passed up to the superclass when escalating are often not those received. The subclass can add and substract arguments.
- Version 1.0 of the Telescript compiler is almost always confused by **initialize** when your class inherits from mixin classes.

This section attempts to demystify the operation **initialize**.

2.5.1. Escalation Order

To fully initialize an object, the Telescript engine must execute the initialize operation for the object's class and each of its superclasses. The Telescript engine can only skip those classes that do not adapt initialize.

If you were programming in C++, you the programmer are responsible for explicitly calling the constructor for each superclass. Telescript programmers are required tell to the Telescript engine when to call the superclasses' initialize operations. Telescript programmers do not invoke the initialize operations of individual superclasses.

You tell the Telescript engine when to escalate to the superclasses' initialize operations by writing an escalation statement:

```
^;
```

or

```
^(arg1, arg2);
```

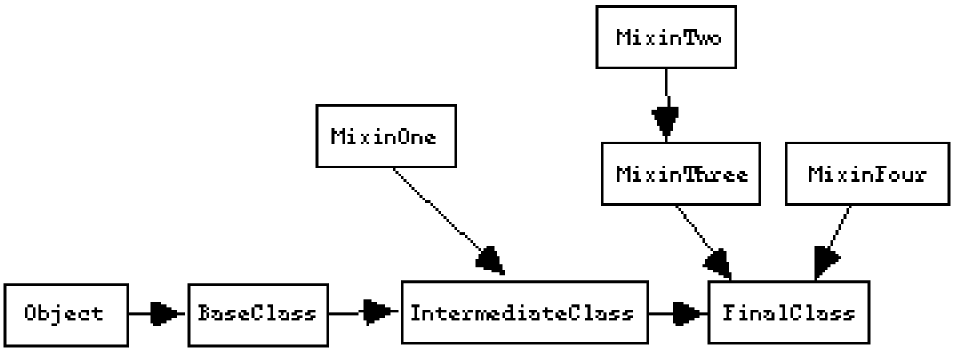
Either form causes the Telescript engine to escalate to the next superclass in escalation order.

Escalation order is basically the reverse of the order in which classes are defined. Consider the following classes:

```
BaseClass: class(Object) = ();
MixinOne: mixin class(Object) = ();
IntermediateClass: class(BaseClass,MixinOne);

MixinTwo: mixin class (Object) = ();
MixinThree: mixin class(Object,MixinTwo);
MixinFour: mixin class(Object);
FinalClass:
class(IntermediateClass,MixinThree,MixinFour);
```

The class hierarchy for these seven classes graphically looks like:



Let's look at the escalation order for FinalClass. Let's start our list with the class FinalClass.

```
1. FinalClass
```

Now append to the list the superclasses for FinalClass in the reverse of the order appearing in the definition of FinalClass:

```
I.   FinalClass
I.A. MixinFour
I.B. MixinThree
I.C. IntermediateClass
```

Now add subbullets for each immediate superclass. Again, list the superclasses in the reverse of the order of their escalation. Never list the anchor class of a mixin. The anchor class is not a superclass.

```
I.     FinalClass
I.A.    MixinFour
I.B.    MixinThree
I.B.1. MixinTwo
I.C.    IntermediateClass
I.C.1. MixinOne
I.C.2. BaseClass
```

Repeat this procedure until you've added all the superclasses of every class. In this sample, you only need to repeat once more. The superclass of BaseClass is Object. Object has no superclasses.

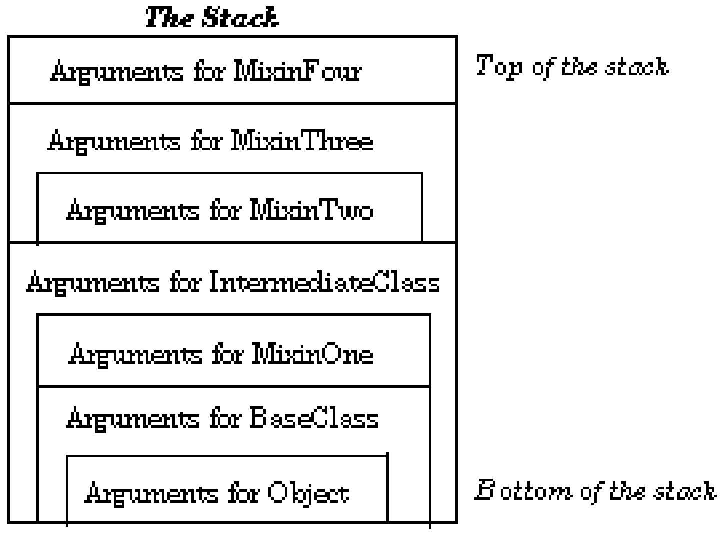
```
I.        FinalClass
I.A.      MixinFour
I.B.      MixinThree
I.B.1.    MixinTwo
I.C.      IntermediateClass
I.C.1.    MixinOne
I.C.2.    BaseClass
I.C.2.a. Object
```

What you are left with is the escalation order. In this sample, escalation from FinalClass will visit MixinFour, MixinThree, MixinTwo, IntermediateClass, MixinOne, BseClass, and Object (in the order listed).

2.5.2. The Stack

When you write code to create an instance of a Telescript class, you need to supply values for all of its initalize arguments. Your class needs to pass the right arguments to each of its superclasses when it escalates. How on earth do you do that? Telescript doesn't let you say which values go to which superclass.

What you need to do is set up the stack so that the arguments for all of your superclasses are on the stack, in the right order. The superclasses are visited in escalation order, so the arguments are popped in escalation order. For the class FinalClass mentioned earlier, the stack needs to look like:



FinalClass must first pick off its own arguments from the stack. An argument is automatically picked off the stack by using the `variable: Type` syntax in the signature for an operation. Arguments declared just as `Type` are not picked off the stack. The contents of the stack, from top to bottom, originally matches the order of the arguments to **`initialize`**.

Suppose FinalClass and each of its superclasses take one argument to initialize their own (native) attributes. The **`initialize`** operation could be written as:

```
initialize: op(finalClassArg:  FinalClassArgType;
     /* other arguments aren't shown */) = {
     ^;
};
```

FinalClass uses the `variable: Type` syntax, so the top of the stack is popped and assigned to **`finalClassArg`**. The rest of the arguments are still on the stack. Escalation order requires that the the remaining arguments be one each for MixinFour, MixinThree, etc. in that order. This means that the natural declaration of **`initialize`** for FinalClass looks like:

```
initialize: op(finalClassArg: FinalClassArgType;
                MixinThreeArgType; MixinTwoArgType;
                IntermediateArgType; MixinOneArgType;
                BaseClassArgType) = {
     ^;
};
```

The class Object expects no arguments for **`initialize`**, so none are listed.

Unless you are an experienced Telescript programmer, you're best off declaring arguments for **`initialize`** in this natural order.

Things get messy when your class needs to look at or hardwire a value for an argument destined for its superclass. If possible, wait until after escalation to the superclasses. That isn't always possible, though. It may be that you need to construct an argument for a superclass, or supply a default value.

If your class has no mixin superclasses, the easiest thing to do is to use the **^()** syntax to escalate. You will need to pull your superclass's arguments from the stack. You later put the arguments for your superclass's **initialize** in parenthesis. As an example, consider a subclass of Object that takes one argument to **initialize**:

```
ClassOne: class(Object) = (
     public
          initialize: op(argOne: String) = {
                ^();
          };
);
```

The class Object takes no initialization arguments, so nothing appears between the parentheses.

Suppose you want to write a subclass, ClassTwo, that takes one argument of its own. ClassTwo wants to make the argument for ClassOne optional. It does this by saying it accepts a string or a nil as its second argument. If nil is given, ClassTwo generates its own value (the string hello).

```
ClassTwo: class(ClassOne) = (
     public
          initialize: op(argTwo: Integer;
                         argOne: String | Nil) = {
                if argOne = nil {
                      argOne = "hello";
                };
                ^(argOne);
          };
);
```

Another reason for modifying the argument of a subclass is that you want to escalate with a particular, hardwired value. ClassThree always wants to escalate with the string "I am a ClassThree". Its signature for **initialize** does not list the argument for ClassOne.

```
ClassThree: class(ClassOne) = (
     public
          initialize: op(argTwo: Integer) = {
                ^("I am a ClassThree");
          };
);
```

If you have mixin superclasses, version 1.0 of the Telescript compiler will rarely let you use the **^()** syntax. The compiler doesn't know how to combine the arguments for the mixin superclasses with each other or with those for the non-mixin superclass. You must use the plain **^** syntax.

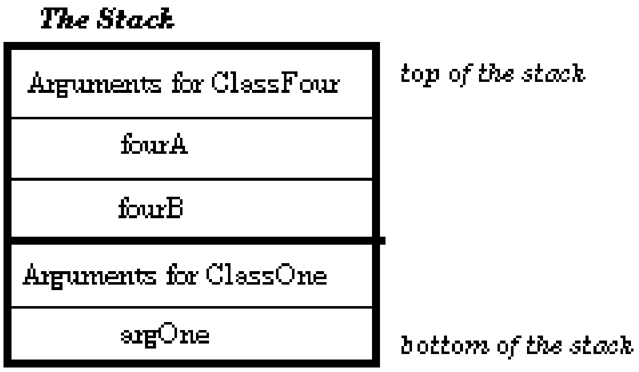
You need to push any values that aren't already on the stack by dropping into Low Telescript. The next chapter will explain everything you're likely to need to know about coding in Low Telescript. The important thing to know regarding initialize is that you need to push the missing arguments onto the stack. Suppose you have a mixin class ClassFour that takes two arguments for its **initialize** operation.

```
ClassFour: mixin class(Object) = (
     public
          initialize: op(fourA: Integer;
                         fourB: Boolean) = {
               ^;
          };
);
```

Suppose you have a class DerivedClass that subclasses from ClassOne and ClassFour. DerivedClass needs one **initialize** argument for itself. It wants to supply hardwired values for the initialization arguments of classes ClassOne and ClassFour.

```
DerivedClass: class(ClassOne,ClassFour) = (
     public
          initialize: op(derivedArg: Boolean) = {
               << "For ClassOne" >>;
                    /* argument for ClassOne          */
               << #false >>;
                    /* argument "fourB" for ClassFour */
               << 12 >>;
                    /* argument "fourA" for ClassFour */
               ^;
          };
);
```

The implementation of DerivedClass's **initialize** operation needs to drop into Low Telescript to push onto the stack one argument for ClassOne and two for ClassFour.



The arguments for ClassFour need to be on the top of the stack, so they are pushed last. Of the two arguments to ClassFour, the first listed argument, **fourA**, needs to be at the very top of the stack. It is pushed onto the stack after the argument for **fourB**.

2.5.3. Argument Names and Properties

The Telescript compiler has a couple of features that enable your to write more efficient initialization code. You've already read about one of them. The previous section explained how declaring an operation's argument using just Type instead of variable: Type leaves the value on the stack. Leaving the value on the stack if all your class isn't going to use the value is more efficient.

The other feature supposed by the Telescript compiler is direct assignment into properties. Suppose your class has a property (or an ordinary attribute) named **myColor**. If you refer to **myColor** in the signature for **initialize** (or any other operation), the compiler does not create a local variable named **myColor**. It stores the passed argument into the property instead. Consider the following class:

```
Painting: class(Object) = (
     property
          myColor:  Identifier;
     public instance
          originalWork: Boolean;

          initialize: op(myColor: Identifier;
                      originalWork: Boolean) = {
               ^;
          };
);
```

This class has two properties. One of these, **myColor**, is explicitly declared as a property. The other, **originalWork**, is implicitly declared by defining a non-abstract attribute that is not implemented via a getter and setter.

The signature for initialize assigns the identifier argument to the *property* **myColor** (not a local variable named **myColor**). It assigns the boolean argument to the *property* **originalWork**. This code is more efficient than similar code using local variables:

```
     initialize: op(theColor: Identifier;
                 isOriginal: Boolean) = {
          ^;
          myColor = theColor;
          originalWork = isOriginal;
     };
```

You pay a risk when you use the more efficient coding style. If you make a typo in a property's name, the compiler will create a local variable. The local variable will store the initialization argument. The property will go uninitialized.

2.5.4. Escalation and Variable Arguments (Ellipsis Syntax)

Telescript allows you to declare that an operation takes a variable number of arguments. To do this, use `variable: Type...` or `Type...` as the last declaration in the signature of the operation. If the type of the variable arguments is Object, you can use ... instead of `Type...` in your declaration.

```
myOp:  op(numberOfPaintingsWanted: Integer;
          favoriteColors:  Identifier...) = {};
```

The Telescript engine determines the number of actual values passed by looking for a special marker object. If you use the Type... syntax instead of the variable: Type... form, the variable number of values and the special marker are left on the stack. Suppose you have a subclass of List that takes an additional initialization argument defining the maximum length for the list.

```
BoundedList: class(List[Object,Equal]) = {
     public
          maxListLength:  Integer;

          initialize: op(maxListLength: Integer; ...) = {
               ^;
          };
);
```

The class BoundedList will function properly. Its implementation of **`initialize`** leaves the variable arguments for List on the stack. The marker object is still there, too. Suppose that BoundedList wanted to look at the arguments to see if their number satisifies the maximum list length condition. The version of the Telescript compiler that ships with the TDE knows to restore the marker in **`initialize`** after it assigns the list of values to **`listElements`**. You can use either the **`^()`** or **`^`** syntax to escalate. In this case, because it wants to pass the list elements up to class List, the code sample that follows uses the **`^()`** syntax.

```
BoundedList: class(List[Object,Equal]) = {
     public
          maxListLength:  Integer;

          initialize: op(maxListLength: Integer;
                         listElements: ...) = {
               if (listElements.length > maxListLength) {
                    throw ArgumentInvalid();
               };
               ^(listElements...);
          };
);
```

Be sure to use the ellipsis syntax to escalate with the individual items in `listElements` **instead of the list** `listElements`. It is very easy to forget and say:

```
^(listElements);
```

If you did make this mistake, BoundedList would contain 1 element. That one element would be a list containing its variable arguments.

2.5.5. Limitations of the Version 1.0 Compiler

Version 1.0 of the Telescript compiler suffers from one significant limitation in its support for initialization. The compiler simply does not know how to combine the initialization signatures for mixin superclasses with each other or with the initialization signature for the non-mixin superclass.

An earlier subsection pointed out one consequence of this problem.

- You can rarely use the **^()** syntax to escalate in **initialize** if your class has mixin superclasses.

Here's another:

- If your class has mixin superclasses, you will almost always have to implement an **initialize** operation for your class.

You won't notice this second problem when compiling your class. You will, however, get a compiler error when you attempt to compile code that creates an instance of the class. The compiler assumes that the initialization signature for your class matches that of the mixin superclass mentioned last. You can easily correct this assumption by explicitly defining an **initialize** operation spelling out the correct signature.

2.6. Ownership and Garbage Collection

Learning how to implement **initialize** is one large step towards becoming a strong Telescript programmer. The other large step is learning how to control what the Telescript engine discards during garbage collection. This section examines the issue of garbage collection in more depth.

2.6.1. How Ownership Works

There are two kinds of objects: normal objects and shared objects. Normally an object is owned by one particular agent or place. Shared objects, however, are in a sense owned by everyone and no one. Because shared objects are uncommon, they will not be discussed further in this section. Read the section on advanced topics if you want more information about shared objects.

An agent or place owns itself. The owner of any other object is determined by the value of owner at the time the object is created or copied.

Note: **owner** is different from **myObject.owner**. **owner**

identifies which process will own any objects that are created, this is, it specifies the would-be owner. **myObject.owner** identifies the process that owns **myObject**.

In the absence of modifying conditions, the would-be owner is the current process. Trace back the call stack and eventually you will reach either the **live** operation of some agent or place or you will reach one of the other operations called only by the Telescript engine (i.e., **entering**, **meeting**, or **transferredIn**). If you find a **live** operation, the current process is its agent or place. If you reach one of the other special operations, the current process is the agent or place upon which the operation was called.

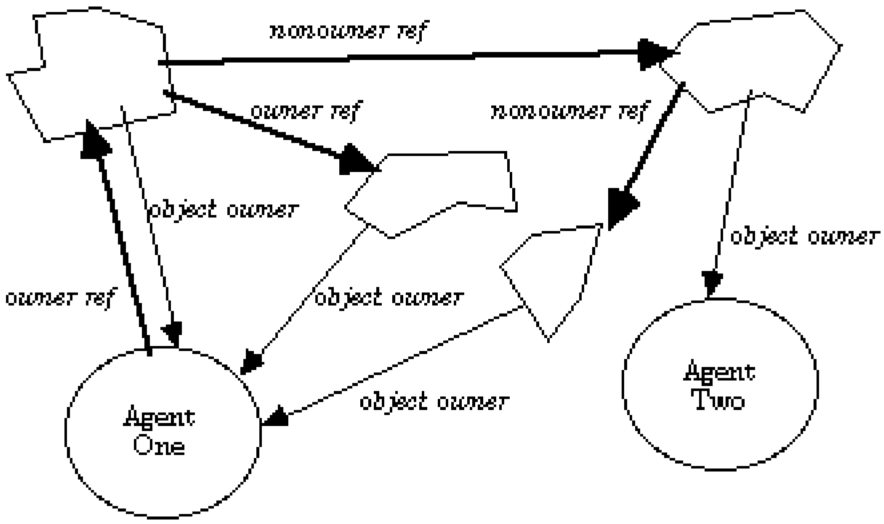
An object can refer to objects that its owner doesn't own. Consider two processes, Agent-A and Place-B. Place-B has a public attribute **saleItemName**. Agent-A is now sitting in Place-B. The code for Agent-A can create a list containing the value of **saleItemName**.

```
/* excerpt from the live operation of Agent-A: */

bargainsToReport = List[Object,Equal](
                       place@Place-B.saleItemName);
```

The value of Place-B's attribute **saleItemName** is likely to be owned by Place-B. Assume that's true in this example. The code creating **bargainsToReport** is part of Agent-A's live operation, so the list (container) assigned to **bargainsToReport** is owned by Agent-A. The first and only element of the list is still owned by Place-B. The value of **bargainsToReport** exemplifies the concept of an object with mixed ownership.

The Telescript language treats references to objects owned by a process differently from references to objects owned by other processes. A reference to an object by the process owning the object is called an *owner reference*. A reference from an object owned by some other process is called a *nonowner reference*. These terms are also used when the reference is held indirectly via an intermediate object.

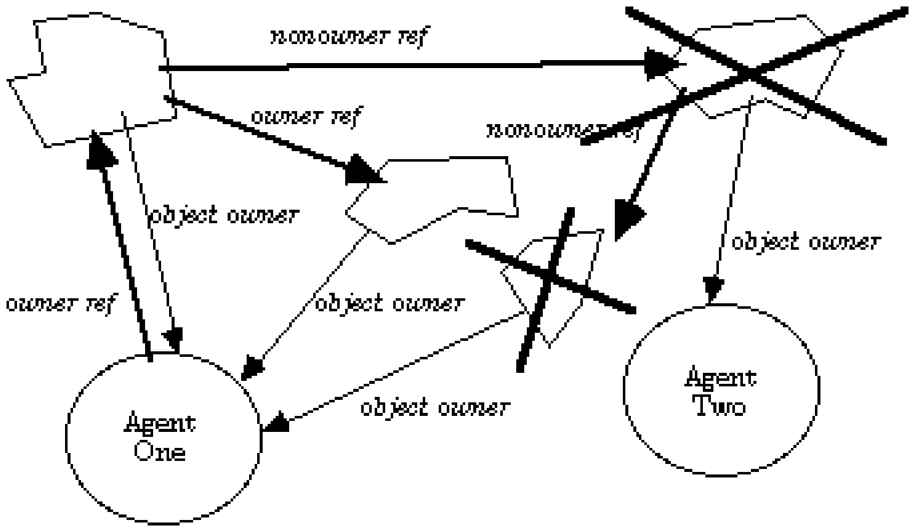


The figure above illustrates two agents and four other objects. Agent One owns two of the objects, as indicated by the fine lines labeled "object owner". Agent Two owns one object. Agent One has a reference to the object in the upper left. Since Agent One owns this object, this reference is an owner reference. The object in the upper left has references to two other objects. The object is the middle is also owned by Agent One, so the reference is an owner reference. The objects in the upper right and lower-middle, however, are owned by Agent Two. The reference from the object in the upper left to the one in the upper right is a non-owner reference. So is the one from the object in the upper right to the one in the lower-middle.

2.6.2. Objects Carried When Doing **go** or **send**

An agent executing **go** or **send** takes with it only those objects for which it has owner references. These references can be held in properties of the agent, including properties implicitly declared when defining an attribute. They can also be stored in the local variables or on the stack of any operations on the call stack. If an object is owned by the agent doing a **go** or **send**, but the reference to it is held indirectly via an object owned by some other process, the object is ***not*** carried along.

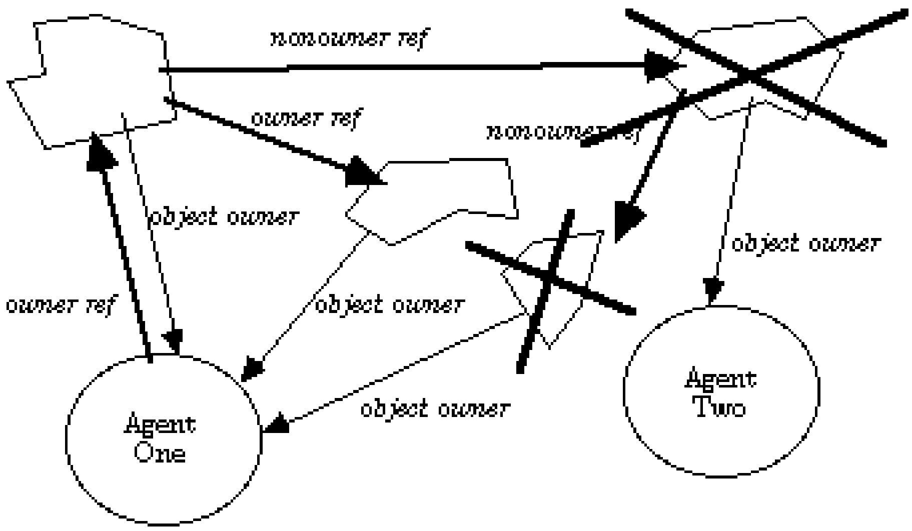
The previous subsection had a picture showing two processes and three objects. If Agent One were to call **go**, some of the objects shown in that figure would not be taken along. The picture is repeated below. Objects not taken along are marked out with a large X



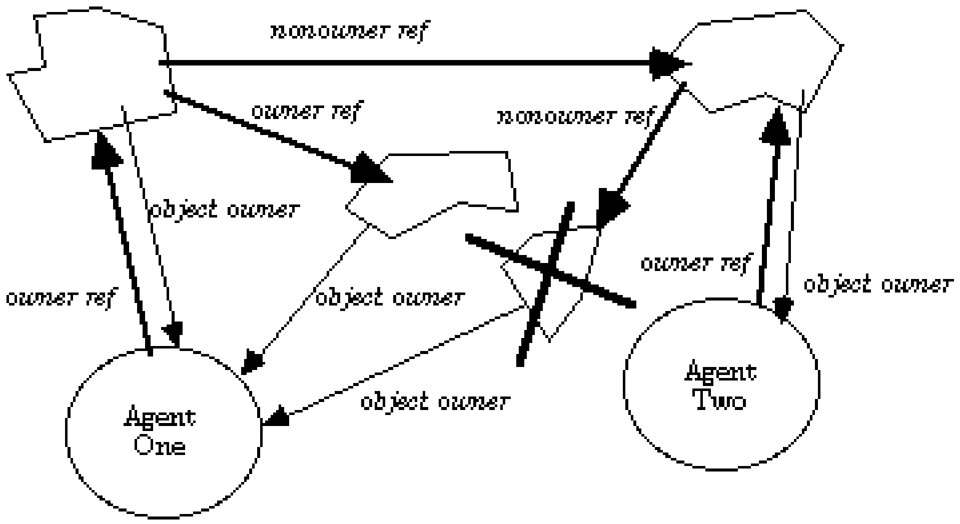
The Telescript engine voids the values of variables, stack entries, and properties that refer to objects that are not carried along. If code tries to call on operation or retrieve an attribute via a voided reference, the Telescript engine will throw the exception ReferenceVoid.

2.6.3. Garbage Collection

Garbage collection deletes all objects for which there is no owner reference. The non-owner references are voided. The figure showing four objects and two agents looks the same after garbage collection as it would should Agent One execute go.



Suppose, however, that Agent Two has a refererence to the object at the upper right. This owner reference would protect the object at the upper right from garbage collection. The reference would not protect the object in the lower/middle. The figure below shows the result of garbage collection.



A process holds references to objects that are:

- values of the process's properties (implicitly or explicitly declared)
- used as the responder for an operation on the process's call stack
- used as the responder by code executed executed in an operation implemented by the process (no matter which process called the operation)
- in a local variable in an operation on the process's call stack
- in a local variable in an operation implemented by the process
- pushed onto the stack frame for some operation on the process's call stack
- pushed onto the stack frame for some operation implemented by the process
- referred to directly or indirectly from an object the process has references to according to any of the above rules. Apply each rule to the object instead of the process.

In other words, references are held in properties, values pushed on the stack, and to objects on which operations are being called. An object is treated as if it has owner references to the stack frames it is executing and the stack frames for operations it implements.

2.6.4. Passage Modes

The Telescript language allows you to modify the way in which an object is passed into or out of an operation. Telescript also allows you to change the way in which an attribute stores or returns its value. In either case, you specify a *passage mode* as well as type. The table below lists the Telescript passage modes.

Passage Mode	Effect
`copied`	This passage mode causes copies to be made. The implementation of an operation receives a copy of any arguments passed by copy. The copy is owned by the owner of the object implementing the operation. If the result of an operation is returned by copy, the object calling the operation receives a copy of the result. The copy is owned by the calling object's owner. Attributes declared as **`copied`** return a copy of their actual value when queried. The calling object's owner owns the copy. If a value is assigned to an attribute declared as **`copied`**, the attribute is updated to hold of copy of the supplied new value. The copy is owned by the owner of the object defining the attribute.
`owned`	This passage mode changes the ownership of objects. The owner of any argument passed by owner is changed to the owner of the object implementing the operation. If the result of an operation is returned by owner, the owner of the object calling the operation is given ownership of the result. An attribute declared as **`owned`** returns a reference to its actual value when queried. If a value is assigned to an attribute declared as **`owned`**, the owner of the new value is changed to be the owner of the object defining the attribute.

protected This passage mode causes a reference to be annotated to say that the object referred to cannot be modified. In addition, any values retrieved from attributes are protected.

An argument passed as **protected** into an operation cannot be modified by the code of the operation. The object calling the operation can still modify the object after the operation completes.

If an operation returns a **protected** result, the caller will be unable to modify the object returned.

An attribute declared as **protected** returns a protected reference when queried.

readonly This passage mode can only be used when declaring an attribute. An attribute declared as **readonly** cannot be updated.

Note: The *attribute* cannot be changed. The implementation of the class defining the attribute is still allowed to change the value of the *property* storing the value behind the scenes.

You can use protected in conjuction with **copied**, **owned**, or **readonly**. You can use **readonly** only in the declaration of an attribute. You cannot use both **owned** and **copied** together, that is, when declaring a single attribute, argument, or result.

It is more efficient to use passage by owner (i.e., **owned**) than passage by copy (i.e., **copied**). However passage by owner is more difficult to use without introducing bugs. The primary problems with passing a value by owner are:

- The object making the call winds up with a non-owner reference. The non-owner reference may be voided by **go**, **send**, or garbage collection.
- You cannot pass a value by owner if you hold a protected reference to it.
- Passage by owner "isolates" an object. Isolation code can void other references.
- An object cannot call an operation that takes an argument by owner unless the calling object's owner also owns the argument.

2.6.5. The Own Block

Telescript provides two mechanisms for changing the would-be owner. You can change the current owner by writing either an **own** block or a **sponsored** operation. This subsection explains **own** blocks. The next section looks at **sponsored** operations.

An **own** block consists of the keyword **own** preceding a block. The sample operation below contains an **own** block.

```
ownBlockDemo: op() = {
     "The current owner is: ".dump();
     owner.dump();
     own {
          "The own block's current owner is:".dump();
          owner.dump();
     };
};
```

Outside of the **own** block, the current owner will be whatever it was before **ownBlockDemo** was called. Inside the **own** block, the current owner will be the owner of the object upon which **ownBlockDemo** was called. Any objects created or copied within the **own** block are owned by **ownBlockDemo**'s object's owner. If the **own** block calls an operation that returns a result by copy, the result would likewise be owned by **ownBlockDemo**'s object's owner.

The **own** block is handy because the only effect it has is to change the current owner. It does not change the accounting of system resources enforced by permits. It does not give any special rights for updating permits.

The **own** block does pose one security concern. Any agents created within the **own** block have the same authority as the owner of the object implementing the own block.

2.6.6. Sponsored Operations

The other way to change the would-be owner is to write a **sponsored** operation. A **sponsored** operation is declared by adding the keyword **sponsored** just before the keyword **op**. The effect is to change both the would-be owner and the accounting of resources enforced by permits.

```
sponsoredOpDemo: sponsored op() = {
     "The current sponsor is: ".dump();
     sponsor.dump();
     "The new effective permit is: ".dump();
     process.permit.dump();
};
```

Note : Only the operations of subclasses of Agent and Place can be **sponsored**.

All of the code within a **sponsored** operation acts if it were contained in an **own** block. Any objects created or copied within the **sponsored** operation are owned by the owner of the object implementing the **sponsored** operation.

Use **sponsored** operations with extreme caution. A **sponsored** operation also changes the *current sponsor*.

The current sponsor is the process whose permit controls what system resources can be used. Normally, for example, cpu usage is charged as teleclicks to the permit of the current process. (The current process is the process you'll find if you trace the call stack back to a **live** , **meeting** , entering, or **transferredIn** operation).

A **sponsored** operation changes the current sponsor to be the process implementing the operation. The effective permit is almost the same as that of the new current sponsor. The one difference is that the age limit of the current process still applies.

The access restrictions implemented by some Telescript engine and some application-defined classes will base their decision on the identity of the current process. If you define an operation as **sponsored**, you are potentially opening up a security hole.

The Telescript engine allows a nativePermit to be altered only if the current sponsor has the same authority as that of the target process. The regionalPermit and regionalData can be changed only if the current sponsor has the same authority as that of the engine place.

Copyrighted classes may also look at the identity of the current sponsor as well. Whether they do or not depends on the implementation of their CopyrightEnforcer objects.

2.6.7. Agents, Places, and Collections

One of the most common mistakes made by people learning Telescript is that they write a place (or agent) class with a public collection attribute.

```
MyPlace: class(Place) = (
     public
          carsForSale:  Collection[CarSaleRecord,Equal];

     /* the rest of this class isn't shown */
);
```

Then they write agents that walk up to the place, add something to the collection, and leave.

```
MyAgent: class(Agent) = (
     myOp: op() = {
          *.go(ticketToPlace);
          place@MyPlace.carsForSale.include(*.myCar);
          *.go(returnTicket);
     };
);
```

The agent's attribute **myCar** holds an object owned by the agent. The agent can add a reference to its **myCar** value to the place's dictionary. That is perfectly legal. But when the agent leaves, it takes its **myCar** value with it. The dictionary is left with a void reference.

As was explained in Section 2.10, you can use the TDE library classes DictOwned and ColOwned to avoid this mistake.

If the code for MyPlace had instead declared **carsForSale** as:

```
carsForSale: ColOwned[CarSaleRecord,Equal];
```

the code for ColOwned would have transferred ownership of the value for **myCar** from the agent to the place. The place would hold onto the value when the agent left. The agent, however, would find itself with a void reference.

2.6.8. Debugging Hints

If you see the exceptions ReferenceVoid or ObjectUnowned, your code isn't getting ownership right. ReferenceVoid indicates that garbage collection, go or send, or passage by owner caused the voiding of a reference to a value. ObjectUnowned indicates that an object didn't have the right to pass an object by owner. You'll also see ReferenceProtected exceptions if your code tries to pass an object by owner when your code holds a protected reference to the object.

Important Note: Garbage collection does ***not*** happen immediately when the last owner reference is lost. It happens at the convenience of the Telescript engine. This means that the exception ReferenceVoid is often not thrown until well after the code that caused the last owner reference to go away.

It is relatively easy to diagnose what is going on when you see an ObjectUnowned or ReferenceProtected exception. You can use the debugger to set a breakpoint on these two exceptions. When the exception is thrown, use the debugger to inspect the properties of the object(s) in question. Be sure to check their owners and whether or not the references are protected.

You can also sprinkle your code with calls to the operation **describeObj**. The operation **describeObj** is one of those that aren't part of the Telescript language but are implemented by the Telescript engine in this version of the TDE. The operation **describeObj** will dump a complete description of the object to the engine's standard output. The dump lists the owner of the object and all objects it refers to. Protected references are identified as such.

Note: The operation **describeObj** sometimes keeps the volume of output down by dumping only the first few elements in a collection.

The Telescript compiler does not know about the operation **describeObj**, so you'll have to drop into Low Telescript to invoke it. Read Chapter 7 for more information on escaping into Low Telescript.

Diagnosing bugs that manifest themselves as ReferenceVoid exceptions are harder to diagnose. Setting a break point on the exception helps a little. It will tell you what went void. Unfortunately, the breakpoint will not tell you where your code lost the last owner reference.

The Telescript engine with this version of the TDE implements a pair of extra operations on class Process (and so Agent and Place) that will help you find the problem. The operation **gc** allows you to explicitly request garbage collection on a particular process. **gcAll** allows you to invoke garbage collection on everything.

The trick is to sprinkle your code with calls to **gc** or **gcAll** in the interesting places and then dump out the suspect object. The interesting points are generally just before and after operations on objects owned by other processes and after operations that take arguments passed by owner.

Neither **gc** nor **gcAll** is known to the compiler, so you'll need to drop into Low Telescript to invoke them. Read Chapter 7 for more information.

2.7. Agents And Places

Agents and places are the heart of Telescript. Subclasses of Agent and Place are likely to be the heart of your application. This section provides from guidelines that will help you get started.

2.7.1. Should I Use an Agent or a Place?

One of the biggest architectural decisions you make when writing a Telescript application is deciding whether you ought to use an agent as opposed to a place. Here are some rules of thumb:

- If your code needs to interact with services in more than one place, use an agent.

Places cannot travel. You must use an agent to interact with services in more than one place.

- If your application is going to be hosted on someone else's

Telescript engine, you ought to have a place of your authority. You will need to implement the place class if your host doesn't supply one.

There are two reasons for this. Your own place lets you write code that has more control over the agents using your service. You can implement your operations to require agents to enter before using your service. You can code access restrictions into the operation entering to keep out unwanted agents or restrict the permits of those allowed in.

The other reason for creating a place is that a place shields its containing place from the place's residents and vice versa. You can publish data visible within the place without affecting processes outside of your place. Your application agents can signal the place's residents without taking the performance hit of signalling every process in the Telescript engine.

- If you want to hardwire tickets into your mobile agents, you should have a place of your own.

The most reliable way to generate a ticket is to supply a Telename containing the authority of the destination place, a Teleaddress supplying the address of the place, and the class of Place. To hardwire a ticket, you need to know this information in advance. You know the information if you write the code that creates a place for your application. You may not know if your agents are installed by someone else somewhere in their Telescript engine.

You may still choose to implement the real service in an agent sitting inside your place.

- If you want a visiting agent to readily interact with multiple services, coding is easier if each service is an agent.

It is easy for a visiting agent to interact with other agents in the same place. The meeting protocol defined as part of the Telescript engine allows the agent to meet with any number of other co-resident agents (simultaneously). If you really want a visiting agent to "meet" places, you can make it happen. You can implement your own (non-standard) meeting protocol.

- If you want to upgrade your application, using an agent is easier.

It is somewhere between very difficult and impossible to patch the class of an agent or place while running. If you need to upgrade your application to use a new version of an Agent or Place subclass, you'll probably need to kill your application process and create a new one using the new class. Killing an agent ends all meetings. Killing a place normally kills all visiting agents too. If you want to upgrade a place, you are likely to need to write code to warn visitors to leave and to re-create resident places and resident agents. The need for this extra code means that it is easier to upgrade an agent than a place.

2.7.2. Recommended Superclasses: Mobile Agents

The definition of a mobile agent class depends on exactly what it is to do. The only essential superclass is Agent itself. With that said, this section will point out classes that are often useful as superclasses. You can read more about class EventProcess in the Telescript Language Reference. The other three classes are documented in the TDE Developer Libraries Reference.

EventProcess

The class EventProcess enables your agent to listen for events. Mobile agents inheriting from EventProcess can poll (or block) to listen for events, for example, notifying them that a meeting was ended or that the place wants them to leave. Since mobile agents typically execute hardwired tasks, the flexibility of the library class TSEventEnabled is typically unnecessary.

TDKGetPublished

The class TDKGetPublished enables your agent to look up published, shared data even if your agent's immediate location fails to inherit from TDKPublished.

PlatformSpecified

The class PlatformSpecified puts an identification of your end-user's client platform in well-known attributes. Smart applications may be able to use the information to provide better service.

CardBearing

The class CardBearing puts contact information for your end-user in a well-known attribute. Some applications may need the information to provide service.

2.7.3. Recommended Superclasses: Places and Stationary Agents

The definition of an application place or stationary agent class also depends on exactly what it is to do. The only essential superclass is Place or Agent itself. This section points out (non-mandatory) classes that are likely to make useful superclasses. You can read more about classes MeetingAgent and MeetingPlace in the Telescript Language Reference. The other classes are documented in the TDE Developer Libraries Reference.

MeetingAgent

The class MeetingAgent can be mixed into agents. Agent classes subclassing from MeetingAgent inherit the code needed to respond to meeting requests from other (e.g. client) agents.

Note: The agent initiating a meeting need not inherit from MeetingAgent.

<u>MeetingPlace</u>

The class MeetingPlace can be mixed into places. Agents cannot use the standard meeting protocol built into the Telescript language unless they are sitting in a place that inherits from MeetingPlace.

If you want to encourage third parties to write agents using your application place, it's a good iead to subclass from MeetingPlace.

<u>TDKPublished</u>

The class TDKPublished provides both access to the API for retrieving published, shared objects and the ability to publish shared objects. Subclassing from TDKPublished lets you publish shared objects visible throughout your application place but not generally in throughout the whole engine place. You can also override the definition of data published in the engine place.

If you want to encourage third parties to write agents using your application place, it's a good iead to subclass from TDKPublished. Mobile application agents that don't inherit from TDKGetPublished may need to use the get-published-object API.

<u>TSEventEnabled</u>

The class TSEventEnabled can be mixed into either an agent or a place. This class implements a standard framework for dispatching events. If your agent or place inherits from TDKPublished, it should also inherit from TSEventEnabled. The class TDKPublished lets its cache get stale unless its events are properly serviced.

The class TSEventEnabled is nice because it lets you extend your application place at a later date by simply installing new event handlers.

If you are writing an application place and choose to not subclass from TSEventEnabled, please subclass from EventProcess. The proper functioning of the backstop object requires that it be invoked in a place supporting Telescript events in the TDKGUIEnginePlace.

<u>TDKPackageControlled</u>

The class TDKPackageControlled makes it possible for your application to install new class or data packages in your place or stationary agent.

2.8. Forward Compatibility with Telescript 1.1

The next major release of General Magic's Telescript products will be based on version 1.1 of the Telescript Language. This section highlights some of the differences between Telescript 1.0 and Telescript 1.1. Programming techniques that produce more nearly forward compatible code are explained.

In Telescript 1.0, classes are named with human readable names (e.g. TDKPublished, ConfigOptionGetter). While this is convenient for the Telescript developer, there is no mechanism to ensure that a class name selected by one developer does not conflict with one selected by another developer. Telescript 1.1 introduces the idea of computed class names to avoid this problem, and to provide greater flexibility in versioning classes. Every class has three unique names -

- **`interfaceName`** - this name is computed by hashing all the elements of the class that make up its interface (such as the public and private attributes, the identifier of the class, the names of the public and private operations).
- **`seriesName`** - this name is computed by hashing the interface of the class and other parts of the implementation that do not vary when a backwardly-compatible change is made. Classes that have the same seriesName are backwardly compatible with one another.
- **`implementationName`** - this name is computed by hashing the entire interface and implementation of the class. Any change to the class, no matter how minor, will change the implementation name.

Every class also has a human-readable identifier, that is not necessarily unique.

2.8.1. Symbolic Defines for Class Names

There are times when code must obtain the name of a class when the class is not available. Examples of this situation are:

- Writing a C++ program that translates between Telescript and C++ objects (using the External Application Framework).
- Writing a Telescript program that needs to generate a ticket to a class of place on a remote engine when the local engine does not have the destination class.

To support such situations, both the compiler contained in this TDE and the R1.1 compilers generate symbolic defines from class definitions. The symbol is derived from the human-readable name for the class appearing in the Telescript source file. The suffix ***_INTF*** is added for macros that evaluate to Low Telescript or C++ values. The suffix ***_INTF_HTS*** is added for macros that evaluate to High Telescript.

The output of the compiler for this release of the TDE takes advantage of the fact that the internal class names are easily derived from their print names.

```
#define StockQuoteAgent_INTF     \
     "StockQuoteAgent" asOctetString
#define StockPlaceOnDevice_INTF  \
     "StockPlaceOnDevice" asOctetString
#define StockQuoteAgent_INTF_HTS \
     "StockQuoteAgent".asOctetString()
#define StockPlaceOnDevice_INTF_HTS    \
     "StockPlaceOnDevice".asOctetString()
```

The R1.1 versions will contain octet string contants something like:

```
#define StockQuoteAgent_INTF     #O"01020304"
#define StockQuoteAgent_INTF_HTS $"01020304"
```

If you define both `NOINTERFACE` and `CINTERFACE` when compiling a C program, you can include generated Telescript header files. You can use the macros when writing translator classes for the External Application Framework classes. The makefile rules supplied with this version of the TDE automatically define both `NOINTERFACE` and `CINTERFACE`.

2.8.2. Do's and Don't's

Wherever Telescript 1.0 classes use identifiers or strings to generate class names, their code needs to be changed to use whichever class name is most suitable. Interface names provide greatest flexibility, and should be used when specifying destinations. Series names should be used when wireline compatibility is required. Implementation names should be used when you absolutely positively want a particular implementation of a class. (However, you'll have to change your reference if you want to upgrade that class, so use of the implementation name is discouraged.). The identifier can be used when you need a human readable form of the name (but be aware that it is not necessarily unique).

The most significant aspect of the type system that might impact Telescript developers is that class names are no longer human-readable. Applications that rely on users being able to enter class names (either through a user interface or through a configuration file) need to be reworked to either get the computed class name from an include file or search the running engine for a class that has the same identifier. This latter approach should be avoided unless absolutely necessary as it can impact the overall performance of the system. It also runs into problems if there is more than one class with the same identifier.

In general, the following changes were made to the General

Magic scripts to convert to Telescript 1.1. You will probably have to make a similar set of changes to your scripts. The changes below all use the interface name, as that is the most flexible. As mentioned above, there may be times where the series name is more appropriate.

Don't use foo.class.name.classDigest

When converting code to Telescript 1.1, you will need to replace all occurrences of `foo.class.name.classDigest` with `foo.class.identifier` if you want the textual representation. The macro `CLASSTITLE` will do the right thing in both this TDE and Telescript 1.1.

Replace occurrances of `foo.class.name.classDigest` with `foo.class.interfaceName.specifier` if you want the octet string. The macros `CLASS_TO_OCTET_STRING` and `OBJ_TO_OCTET_STRING` will do the right thing in both this TDE and Telescript 1.1.

Don't use foo.class.name

When converting code to Telescript 1.1, you will need to replace all occurrences of `foo.class.name` with `foo.class.interfaceName`. The macros `CLASS_TO_IFNAME`, `CLASS_TO_SERIESNAME`, and `CLASS_TO_IMPLNAME` can be used to do the right thing in both this TDE and Telescript 1.1 (and beyond).

Don't Use the Operation isMember

The operation `isMember` will go away. Replace all occurrences of '`if foo.isMember(Bar)`' with '`if foo is Bar`'. This coding style works in both this TDE and Telescript 1.1. If you have a variable or expression that is giving you a Class Name object, you can use the macro `TYPE_OF_TYPE` to define the type for the variable and plug the expression into the macro `IS_OF_TYPE`. You can use the macros `CLASS_TO_TYPE` and `OBJ_TO_TYPE` to obtain typing information from a class or object, respectively.

Don't Use !

Do not use ! when declaring attributes, variables, arguments, or results to indicate that you want an instance of a class instead of a subclass. This feature is not supported in Telescript 1.1. The ! will still be syntactically valid, but the effect will be entirely different. In Telescript 1.1, `Foo!` no longer means 'instance' - it means 'series'.

Never use ! when declaring the types of variables, properties, attributes, operation arguments, or operation results. If you want to enforce the is-instance check, do so in code (see the next section for details).

Don’t Use the Operation isInstance

The operation `isInstance` will go away, too. If you really need to say ’instance’ in Telescript 1.1, you can say ’`if foo is Foo<instance interface>`’. The macros `IS_INSTANCE` and `IS_INSTANCE_OF_EXPR` can be used in both this TDE and Telescript 1.1 code to check to see if an object is an instance of a class.

Don’t Just Mention a Class Used as an Argument

In Telescript 1.0, when you say `*.myOp(AppClass)`, the compiler generates code to pass a reference to the class App Class. The compiler for Telescript 1.1 will instead pass a reference to a type matching class App Class. If you need to pass a class as an argument, instead of a type, Telescript 1.1 code must say `~AppClass`. (This commonly occurs when publishing particular classes, such as guides.)

If you want to pass a class as an argument into an operation, use the macro `USE_CLASS`, as in `*.myOp(USE_CLASS(AppClass))`. This macro will do the right thing in both this TDE and Telescript 1.1.

Don’t Use the Keyword imports

The keyword `imports` will go away. Its presence in Telescript 1.1 will give you a syntax error.

- Do not use "imports" clauses when declaring modules.
- Do not use "imports" clauses when declaring classes.

Find and include the necessary include file instead.

Don’t Use isSubclass or isSubclassByName

If you want to say "if A is subclass of B" where A and B are classes, Telescript 1.1 code must create an extended type out of B, and say `if B.isSatisfied(A)`. You can say ’`if a is B`’ where a is an instance of A, but you can’t say ’`if A is B`’ where A is a class. The macros `IS_SUBCLASS` and `IS_SUBCLASS_BY_TYPE` can be used in both this TDE and Telescript 1.1 to achieve the effect of calling operations `isSubclass` and `isSubclassByName`, respectively.

Don’t Use isMemberByName or isInstanceByName

The operations `isMemberByName` and `isInstanceByName` will be dropped from the Telescript language. Use the construct below:

```
OBJ is CLASS
```

instead of:

```
OBJ.isMemberByName(STRING_TO_CLASSNAME("CLASS"))
```

whenever possible. If the class `CLASS` might not be loaded, use a macro for its (interface) name as an argument to the macro `OCTET_STRING_TO_TYPE`, as in:

```
OBJ is OCTET_STRING_TO_TYPE(Foo_INTF_HTS)
```

This TDE release does not supply any macros that make it easy to achieve the effect of using `isInstanceByName` in both this TDE and Telescript 1.1.

2.8.3. Free-Standing Telescript

The Telescript compiler in this TDE is unable to process code involving user defined classes except within the context of a class definition. With this release, if you want to write a script that will create an instance of a test agent, you must enclose what you really want to do inside a dummy module defining a dummy class that in turn defines a dummy operation. The implementation of the dummy operation is the code that you really want. Then you run the output of the compiler through the post-processing tool tsprocs to delete the dummy class and operation.

The Telescript 1.1 compiler will support writing this kind of High Telescript code without all the baggage. Low Telescript, on the other hand, is very difficult to write in Telescript 1.1 because you must use the computed names for classes instead of the symbolic ones.

You can use the macros `BEFORE_INCLUDE`, `BEFORE_CODE` or `BEFORE_CODE_WITH_RETURN`, and `AFTER_CODE` to add the right kind of wrapper for such scripts in either this TDE or Telescript 1.1.

2.8.4. Be Aware that Signatures Will Change

The signatures to some of the attributes and operations of classes in the Telescript language and in General Magic's libraries will change in the Telescript 1.1 release. A partial list of signatures that will change is given below. This list is not conclusive. You will have to consult the documentation shipped with Telescript 1.1 to obtain a complete list.

- Ticket - `destinationClass` will take a type instead of a class name.
- Place - `entering` will be given a class instead of a class name
- MeetingAgent - `meeting` will be given a class instead of a class name

The cleanest coding style is to go ahead and use the signatures as they are defined in this TDE, then use the Telescript 1.1 compiler's error messages to tell you where you need to change your code. If you really want to minimize your changes in Telescript 1.1, you can use the macros listed in the TDE Developer Libraries

Reference to hide some (but not all) of the type changes.

2.9. Advanced Topics

This section looks into aspects of Telescript that may not interest you at first. If you haven't written your first simple program yet, you might do better to go try your hand at coding before reading this section.

2.9.1. Defining Your Own Meeting Protocol

The Telescript language defines two classes, MeetingAgent and MeetingPlace, that implement one way of letting agents meet one another. These classes implement only one possible mechanism. You can implement your own meeting protocol. You can subclass from MeetingAgent and MeetingPlace if you like. If that isn't convenient, go ahead and roll your own meeting protocol from scratch.

If you are writing your own meeting protocol from scratch, you should go read about the following features of the Telescript language. They are likely to prove to be the building blocks you need.

`entering`	an operation on class Place
`ExitEvent`	a class of event signalled by the engine place
`DeathEvent`	a class of event signalled by the engine place
`contacts`	an attribute on all agents and places

2.9.2. Creating Parameterized Classes

Telescript class definitions can be parameterized. When an object is created or a type declared, specific classes are plugged into the parameters.

You may already be familiar with a handful of parameterized classes. The classes Collection, List, Dictionary, and Set are parameterized. For example, if you want a declare a list that contains only integers, you would write the Telescript code:

```
List[Integer,Equal]
```

The specific class values for the parameters are contained between the parentheses. List takes two parameters. The first identifies the type of element contained in the list. The list declared above is a list of integers. The second class specifies what kind of comparison is used when the operations **`find`** and **`exclude`** are called on a list. The list declare above tests to see if values are equal.

You can write your own parameterized class by adding an extra signature at the beginning of the class definition. The signature identifies the parameters accepted by the class. Suppose you want to write a coordinate class. You'd like to let other programmers create choose their own coordinate system. Some might want to measure the coordinates with integers. Others might want to use octet strings. The code fragment below declares a CoordinateClass that takes the type for its x and y coordinates as parameters.

```
CoordinateClass: class[xCoordType: Class;
                       yCoordType: Class](Object) = (
    public
        xCoordinate:  xCoordType;
        yCoordinate: yCoordType;

        initialize: op(xCoordinate: xCoordType;
                       yCoordinate: yCoordType) = {
              ^;
        };
);
```

You can use the parameters for the class in declaring the type of arguments, results, attributes, and so forth. The CoordinateClass above uses the parameter **xCoordType** when declaring the type of the attribute **xCoordinate**. It uses the parameter **yCoordType** to declare **yCoordinate**.

Important Note : The Telescript compiler included in this release understands type-checking based on parameterized classes. The Telescript engine, however, does not fully implement parameterized classes. Collection, List, Dictionary, and Set always compare elements using equality and can contain any objects. If you write a parameterized class, you will have no access at runtime to the values plugged into the parameters. No runtime checks will be done on typing of arguments, results, and attributes declared using the parameters.

You can constrain the value of a class parameter to values that are subclasses of a given class. Suppose you want to limit the coordinate types to be classes inheriting from Ordered. You want to use the operations of ordered to define the ordering of values along the X and Y axes. You can impose this constraint by changing the signature of CoordinateClass to:

```
CoordinateClass: class[xCoordType: Class <: Ordered;
                       yCoordType: Class <: Ordered]
                     (Object) = (
    /* the rest of the class doesn't change and    */
    /* so isn't shown in this code fragment        */
);
```

The constraint is specified by adding following the parameter type declaration (i.e., Class) with the token **<:** and then the name of another class.

2.9.3. Designing Telescript Classes for EAF

The External Application Framework (EAF) makes it possible to exchange the data portion of Telescript objects with an external process. The difficulty in writing the translation classes depends in large measure in how you've defined your Telescript class. If you follow the guidelines below, you'll find it fairly straightforward to write translators.

- Declare initialization arguments in the natural, escalation order.

Arguments used by a class should appear before those used by its superclasses. The arguments for the superclasses should appear in escalation order. The arguments for a particular superclass should agree in number and order with the **`initialize`** signature of the superclass.

- Define an initialization argument for every private attribute, every property without an attribute, and every read-only attribute. (At least, do this for every value that you want to set to non-default values.)

The translation framework defined in this release of EAF cannot call operations other than **`initialize`** to create the returned value in Telescript. If an attribute or property cannot be set by calling a public attribute or the operation **`initialize`**, a value cannot be sent from the C++ program.

- Define only initialization arguments that correspond to attributes of your class.

The translation framework defined in this release of EAF does not make it easy for the external process to supply a hardwired value. In principle, you can write a translator that hardwires the arguments to **`initialize`**, but doing so is not easy. It is easy, however, to plug the value of attributes into the arguments to **`initialize`**.

- If you want to be sure your external process gets the value of an attribute, make it a public attribute.

2.9.4. Effective, Native, Local, and Regional Permits

Each agent and place runs under an effective permit that combines the restrictions of several other permits. The effective permit takes the most restrictive setting from the process's nativePermit, localPermit, and regionalPermit. The effective permit is available as the value of the attribute **`permit`** .

The Telescript engine imposes access restrictions on who may change a permit. Read the Telescript language manual if you want to know the details. This section will only present the highlights.

The **nativePermit** is set from an argument to **initialize**. This is the permit that you are required to supply when creating an agent or place.

The **localPermit** is an optional argument to **initialize**. The local permit on an agent changes every time the agent enters another place. Places can control the local permits of their residents. To set the **localPermit** for an agent or place entering your place, implement the operation **entering**. Your implementation needs to update the permit passed as an argument before returning.

If you want your place to change the **localPermit** of an agent or place already residing in it, you can assign a new value to the target's **localPermit** attribute.

- Don't try to edit **localPermit**.

The attribute **localPermit** returns its value by copy. If you try to edit the attribute, you're actually editing a copy. The target process retains its current permit. You should assign an entirely new permit value.

- Make sure that the place is the current sponsor.

The attribute **localPermit** can only be updated by processes of the same authority as the local place. The code for the place itself can update the value only if done on the **live** operation's thread of control (with no **sponsored** operations enroute) or if done through a **sponsored** operation of the place.

2.9.5. Using ? and &

Normally you will refer to an attribute of an object by using a dot-delimited syntax. For example, you will refer to the **authority** subattribute of the current process's **name** attribute as:

```
process.name.authority
```

Suppose you want to get the subattribute **intensity** of the attribute **color** of the painting in the variable **thisPainting**. Suppose further that the variable **thisPainting** might be nil instead of refering to a painting. The attribute **color** might also be nil. You can write this code as:

```
paintingIntensity: Integer | Nil;

paintingIntensity = nil;
if (thisPainting != nil) && (thisPainting.color != nil) {
     paintingIntensity = thisPainting.color.intensity;
};
```

There is a simpler way to achieve the same effect. You can use **?.** instead of just plain period if you want the value nil if the responder is nil and an attribute if the responder is non-nil. To get the intensity of the painting, you can simply write:

```
thisPainting?.color?.intensity
```

The syntax above is not only simpler. It also generates faster code.

Another technique that can improve the complexity and execution speed of your code is the use of &. Suppose you have a list in the variable ageSample. You want to replace the third element with the value 5, the fifth element with the value 9, and the twelfth element with the value 2. You can do this by writing the three statements below.

```
ageSample.set(3, 5);
ageSample.set(5, 9);
ageSample.set(12, 2);
```

If you have a sequence of statements that all call operations or retrieve attribute values from the same object, you can rewrite your code as a single statement. Replace the repeated expressions selecting the object with the token &. For example, the code above can be rewritten as:

```
ageSample.set(3,5)
&.set(5,9)
&.set(12,2);
```

Notice that there are no semi-colons between the operation calls.

2.10. Other Common Pitfalls

The previous sections of this chapter have discussed many aspects of Telescript programming. This section winds up the chapter by pointing out the other pitfalls that commonly bite new Telescript programmers.

2.10.1. Pitfalls for C++ Programmers

Telescript has private attributes and operations. C++ has private data members and member functions. Don't be misled. The term *private* does ***not*** mean exactly the same thing in both languages.

In Telescript, *private* means that only the object itself can use the attribute or value. The attribute or operation can be used by code in any operation of the object's class or any of its class's superclasses. A private attribute or operation can even be invoked by code read in from a file and executed!

In C++, *private* means that the data member or member function can only be used by objects of the given class or its subclasses.

The key difference is that Telescript retricts usage to the ***same object***. In C++, a different object of the same class can use a private member function or manipulate a private data member.

2.10.2. Escalation of Variable Arguments

The section on variable arguments and initialization explained that if an argument takes a variable number of arguments, the Telescript engine uses a special marker to keep track of how many values are actually being passed. The Telescript compiler usually does the right thing to maintain the special marker. If you do something fancy that causes the marker to be lost, the Telescript engine will throw the exception MarkMissing when you escalate.

The exception MarkMissing means that you need to do something to restore the mark. If you get this exception from some operation other than initialize, you need to use the ^() syntax to escalate.

You shouldn't see the exception MarkMissing from initialize unless you have dropped into Low Telescript to execute an operation that takes a variable number of arguments. The solution in this case is to add an extra marker to the stack. Read Chapter 7 for more information.

2.10.3. Readonly Versus Protected

The earlier section on passage modes explained the passage mode **readonly**. The keyword *readonly* is very misleading. It does ***not*** mean that the value of a **readonly** attribute cannot be modified. The passage mode **readonly** only prohibits code that assigns a new value to the attribute.

Consider the readonly attribute **colorPalette**. It is declared as a list of strings.

```
colorPalette: readonly List[String,Equal];
```

The following code works:

```
obj.colorPalette.clear();
obj.colorPalette.include("Red", "Green");
```

It is perfectly legal to call operations that update the current value of a **readonly** attribute. To get an attribute that cannot be updated (except via its property), you must declare it as **readonly protected**.

```
colorPalette: readonly protected List[String,Equal];
```

The passage mode **readonly** prevents anyone from assigning a new object as the value. The passage mode **protected** prevents anyone from altering the object that is the current value.

2.10.4. The Woes of Protected References and Locked Objects

The passage mode **protected** and the operation **protect** create **protected** references to objects. Generate protected references when they're really needed. Don't declare the passage mode for arguments to an operation as **protected** just to indicate that your code won't modify the argument's value.

There are a number of things you can't do with a protected reference. Some of prohibitions make sense. For example, you can't pass a value by owner if you have a protected reference to it.

At other times, protected references just make it impossible to use classes implemented in particular ways. You can't retrieve the value of an attribute that is implemented by a getter if the getter uses a resource, updates an internal cache, or does anything else to modify the value of one of its properties. The object can't implement the getter by making a call to an external process.

You can get an unprotected reference by copying an object subject to the following limitations:

- the object is not of a class inheriting from Uncopied
- the object is not of a class inheriting from Protected
- the object isn't locked

Locked objects haven't been mentioned before. You can create them by creating instances of class Package or by setting the attribute isLocked to true. (Class Package locks its initialization arguments.)

Locked objects behave in many ways as if all references to them were protected. The two exceptions to this general rule are:

- locked objects can be passed by owner

- you need to use **unlockedCopy** instead of **copy** to get a modifiable copy of the object

Objects inheriting from class Protected act as if they are locked. The operation **unlockedCopy**, however, will not produce a copy that can be updated.

2.10.5. When **go** And **send** Cannot Be Called

An agent is allowed to call the operations **go** and **send** only if the call stack contains only operations called on the agent itself. The Telescript engine will throw the exception ProcessNotControlled if the call stack contains a call to an operation of some other object.

2.10.6. Creating Telenames and Teleaddresses

You need to be very careful when creating instances of the classes Telename and Teleaddress. These classes take their arguments by owner. If you try to pass in the name of some other process, the Telescript engine will throw an exception because you're not allowed to steal its name. It is wise to develop the safe habit of copying the initialization arguments for telenames and teleaddresses (at least, if they're not constants).

```
Telename(process.name.authority.copy(), nil);
Teleaddress(place.name.authority.copy(), $"414243");
```

2.10.7. The Operation **discard** Doesn't Delete Objects

The class Object implements an operation named **discard**. The term *discard* suggests that the operation will get rid of the object. In actuality, **discard** does not do anything useful unless you have escaped into Low Telescript. The Telescript statement:

```
Permit().discard;
```

creates a permit, pushes the permit onto the stack in preparation for calling **discard**, then pops the permit from the stack.

The operation **discard** in Telescript (not Low Telescript) is the moral equivalent of a no-op instruction. It is an operation, but it does absolutely nothing. If you have escaped into Low Telescript, you can call the operation **discard** to effectively pop the object at the top of the stack.

2.10.8. Restrict Blocks and age

The restrict block lets you impose additional temporary restrictions on the effective permit. These restrictions are effective only in the restrict block.

Suppose you want to limit the elapsed time spent executing a block of code to 20 seconds. It is very tempting to supply a Permit with an age limit of 20 seconds as the restriction:

```
restrict Permit(20) {
     /* code that should complete in 20 seconds     */
     /* goes here                                   */
};
```

The code sample above will not behave in the way you expect. Suppose the code is executed by a process that has been alive for 80 minutes. The process's age is already 4800 seconds. Because the current age of 4800 seconds exceeds the imposed limit of 20 seconds, this agent will not execute any of the code.

If you are using a restrict block to limit elapsed time, compute the age limit as an increment added to the current process's present age.

```
restrict Permit(process.age + 20) {
     /* code that should complete in 20 seconds     */
     /* goes here                                   */
};
```

2.10.9. Resource Usage and InternalException

If you encounter the exception InternalException when trying to use a resource, your problem is almost certainly that the current process already has exclusive access to the resource.

2.10.10. Initialization of Processes

One of the surprising things about the operation initialize for an agent or place is the the current process and current place depend on who is doing the creating. The current process is ***not*** the agent or place being created. If you are creating a place, the current place is not the place being created. It is the place that will be occupied by the place being craeted.

2.10.11. The Result of entering Goes Void

The arguments to entering will go void after the call to entering returns. If you want to return some or all of the arguments in the result of entering, be sure to copy the values. Don't use the original.

3. Using Low Telescript

The compilation tools in release of the Telescript Developer Environment convert Telescript into a particular intermediate code named Low Telescript. The compilation tools will let you escape into Low Telescript, much as many C compilers let you escape into assembly language.

Under normal circumstances, you should write all your code in Telescript. There is no guarantee that future versions of the Telescript engine will be able to interpret Low Telescript v1.0. In fact, it's a pretty sure bet that the next major release of the TDE will be unable to interpret Low Telescript v1.0.

There are occasions when using Low Telescript is your only alternative, or your best alternative, in this release of the TDE. You must use Low Telescript when:

- writing a configuration file for your application
- writing a linker input file for the Telescript linker ***htl***
- using the special debugging operations **gc**, **gcAll**, and **describeObj**
- escalating from **initialize** when you have mixin superclasses and need to pass a different set of arguments than your class received into initialize
- using the front panel debugger command **dEval** to correct the values of attributes or operation arguments (or supply bad values to test error handling)

This chapter explains enough of the syntax of Low Telescript 1.0 to enable you to do these things. The first section explains how to drop into Low Telescript from Telescript. The next six sections explain how to write Low Telescript. The final section, Section 7.8, tells you how to get Low Telescript that does something complex.

3.1. Escaping Into Low Telescript

Telescript defines a syntax for escaping into Low Telescript. Low Telescript code fragments are introduced by two opening angle brackets, that is, by <<. Two closing angle brackets, that is >>, mark the end of a Low Telescript escape. The following illustrates a Telescript statement, an escape into Low Telescript as a statement, and a third statement in Telescript.

```
"This is a Telescript dump statement".dump();
<< "This is a Low Telescript dump statement" dump >>;
"This is another Telescript dump statement".dump();
```

You can write drop into Low Telescript to write a single statement, as shown by the previous sample. You can also drop into Low Telescript to compute an expression used within a Telescript statement.

```
numOfItems = 1 + << 2 >>;

placeTelename = << #place >>@Place.name;
```

Low Telescript escapes are also legal outside of Telescript modules, before the first class in a module, between classes, and after the last class in a module.

```
<< "I am before a module" dump >>;

MyModule: module = (

<< "I am before the first class" dump >>;

ClassOne: class(Object) = ();

<< "I am between two classes" dump >>;

ClassTwo: class(Object) = ();

<< "I am after the last class" dump >>;

);

<< "I appear after the module" dump >>;
```

Low Telescript appearing in configuration files or typed as input to the front panel is ***not*** enclosed in **<< >>** characters.

3.2. Comments and White Space

You can need to put at least one white space character to separate Low Telescript tokens. You can use the space character, a tab, or a new line as white space.

You can embed comments in Low Telescript. The comment syntax is the same as that in Telescript.

3.3. Integers, Strings, and Other Simple Values

Low Telescript has its own syntax for supplying simple values. The table below gives samples for each of the kinds of simple values.

Class	Sample	More Information
Boolean	#true	The value true. Not case sensitive.
	#false	The value false. Not case sensitive.

Nil	#nil	The value nil. Not case sensitive.
Integer	5 -1	An optional minus sign followed by 1 or more decimal digits. Put no space between a minus sign and the first digit.
Character	#C'x' #c'\\' #C'\''	Use a pound sign, the letter C in upper or low case, a single quote, the character, and a second single quote. If you want the single quote or backslash character, use a backslash then the real character.
String	"hello" "a\\b\\c" "a\"quote"	The characters in the string are enclosed by double quotes. Use backslash to precede a backslash or double quote character you want in your string.
Octet	#O'0a'	The syntax is like that of Telescript except that you use a pound sign and letter O instead of a dollar sign. The letter O can be in either upper or lower case.
Octet String	#O"0a0123"	The syntax is like that of Telescript. Just use a pound sign and the letter O instead of a dollar sign. The letter O can be in upper or lower case.
Bit	#b'0'	The bit zero. Uppercase B is also legal instead of lowercase b.
	#b'1'	The bit one. Uppercase B is also legal instead of lowercase b.
Bit String	#b"011100"	A pound sign, the letter b in upper or lower case, a double quote, zero or more instances of 0 or 1, and another double quote.
Identifier	'Hello	A single quote mark then the text of the identifier.

Real Number	1.23 -1.3e-4	You can use either a decimal number or a form that combines a decimal number, the letter e, and then an integer exponent.

3.4. Referring to the Current Process and Place

Low Telescript has its own equivalents to the Telescript expressions that let you refer to the current process, current place, and so forth. The correspondences are listed in the table below. All of the Low Telescript tokens are case insensitive.

Telescript	Low Telescript	Meaning
place	**#place**	The current place.
process	**#process**	The current process.
here	**#here**	The place in which the current process resides.
sponsor	**#sponsor**	The current sponsor.
client	**#client**	The current client.
self or *	**#self** or *	The responder.

3.5. Referring to Variables, Properties and Attributes

Variables, properties, operation arguments, and attributes are declared in Telescript using a `name: Type` syntax.

```
originalColor:  Identifier;
favoriteColors: Set[Identifier,Equal];
```

Variables, properties and operation arguments are cited by simply mentioning their names.

```
originalColor = 'Red;
likesRed = (favoriteColors.find('Red) != nil);
```

Attributes are referenced by giving an expression that evaluates to the desired object and then appending a period and then the attribute name.

```
*.originalColor = 'Red;
likesRed = (johnSmith.favoriteColors.find('Red) != nil);
```

To retrieve the value of a property in Low Telescript, prefix its Telescript name with a percent sign.

```
%originalColor
```

If you want to assign a value to a property, give the value first. Then refer to the property by prepending equals and percent sign to the Telescript name for the property.

```
'Red =%originalColor
```

Don't put white space between the equals sign and the percent sign or between the percent sign and the property's name.

Operation arguments and variables declared in the outermost block for an operation are handled in a similar fashion. The dollar sign character is used instead of a percent sign, however. For example, the code fragment below retrieves the value of the variable **originalColor**.

```
$originalColor
```

Assignments are likewise handled by prefixing an equals sign and a dollar sign to the variable's name.

```
'Red =$originalColor
```

One of the limitations of Low Telescript v1.0 is lack of support for variables whose scope is a block other than the outermost block. The Telescript compiler in this release of the TDE works around this limitation by mangling the names of variables in nested blocks. This makes the Low Telescript names for variables in nested blocks difficult to anticipate. They depend on the number and nested of code blocks appearing earlier in the operation.

If you need to write code that drops into Low Telescript and that code needs to refer to a variable, declare the variable in the outermost block. You will need to code try/catch statements in the following way, however, because there is no way to declare the exception's variable in the outermost scope.

```
opWithTryCatch: op() = {
     errorVariable: Exception;

     try {
     } catch thrownError: Exception {
          errorVariable = thrownError;

          /* you can escape into Low Telescript here */
     };
};
```

If you want to type Low Telescript to the **dEval** command of the Telescript engine's front panel, you can determine the actual variable name by using the **dList** command. The names of nested variables look something like:

```
$_nestedVariable_5_2_3_1
```

To read an attribute, you first need to identify an object. Then you append a space and the Telescript name for the attribute. Suppose you want to get the value of the attribute name of the current process. As the previous section explained, you use **`#process`** to refer to the current process. Retrieving the process's name then is written as:

```
#process name
```

If you want to get at an attribute of an attribute, just iterate this rule. The expression above computes the name of the current process. The **`authority`** attribute of the name would be:

```
#process name authority
```

The Low Telescript code for assigning to an attribute looks rather strange. You first supply the new value. Then you specify the attribute. The trick is to prefix the final subattribute with an equals sign. To assign the octet string 010203 to the **`authority`** subattribute of the current process's name, you would write:

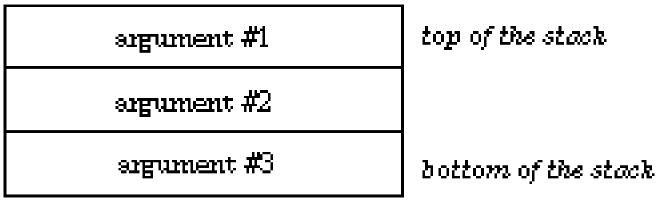
```
#O"010203" #process name =authority
```

Note : Telescript doesn't let you update the name of a process, so executing the code above will cause an exception to be thrown. It is, however, legal Low Telescript syntax-wise.

3.6. Calling Operations

Telescript operations are called by pushing the arguments to the operation on the stack, then invoking the operation. The arguments are pushed so that the first one mentioned in the operation's signature is at the top of the stack.

The Stack Just Before Calling An Operation



To invoke an operation in Low Telescript, you must write expressions that evaluate to the arguments. The Telescript engine will push the value of the expression onto the stack when it evaluates the expression. This means that you need to start with the expression giving the final argument first.

Suppose you are going to call the operation **`add`** on a dictionary. You want to add the value 52 under the key "age". The signature for the operation **`add`** is:

```
add: op(key: Key; value:Value);
```

This means that you supply first the value then the key in Low

Telescript:

```
52 "age"
```

The next thing you need to do is identify the object upon which you want to call the operation. Suppose the dictionary is stored in the attribute **profile** on the current process. You would write the following expression to obtain the dictionary:

```
#process profile
```

Finally, you need to mention the name of the operation you want to call. The operation's name is the same in Low Telescript as it was in Telescript. To call Dictionary's **add** operation, you would write:

```
add
```

Now put it all together:

```
52 "age" #process profile add
```

You've just written the equivalent of the Telescript expression:

```
process.profile.add("age", 52)
```

There are two fine points that you need to know.

If an operation's signature says `XXX|Nil` for one or more trailing arguments, you can omit these trailing arguments when calling the operation in Telescript. The operation **substring** on class String takes two arguments, both of type **Integer|Nil**. In Telescript, you call:

```
"hello".substring()
```

You need to supply nil values for all of these optional arguments in Low Telescript. To write the equivalent of the expression above in Low Telescript, you would say:

```
#nil #nil "hello" substring
```

The other fine point you need to know concerns arguments that take a variable number of arguments. These operations are **initialize** and those operations whose signature uses an ellipsis (...) when declaring the type of the final argument. You need to put a marker on the stack to tell the Telescript engine where the arguments end. You need to create a marker before you give the expressions for any of the arguments. To create a marker, use the syntax:

```
#mark
```

Suppose you want to append three new characters to a string. The class String inherits an **appendElements** operation from class List. The operation **appendElements** takes a variable number of arguments.

```
#mark #c'c' #c'b' #c'a' "hello" appendElements
```

The Low Telescript expression above calls the operation **appendElements** on the string "hello". It gives the three characters a, b, and c as the arguments to **appendElements**.

There are four operations that are likely to be particularly useful to you in debugging. All of these operations are implemented by the version of the Telescript engine shipped with this TDE, but are not part of the Telescript language. There will be Telescript engines that do not support any of these four operations.

- **dump**
- **describeObj**
- **gc**
- **gcAll**

The first two of these dump a description of an object to the Telescript engine's output. The operation **describeObj** gives a very long and detailed description of the object. **dump** gives a short one. The operations **dump** and **describeObj** can be called on any kind of object.

The other two operations invoke garbage collection. Both can be called on any process. The operation **gc** invokes garbage collection on just the objects associated with the process. **gcAll** invokes garbage collection on everything.

The following code sample shows how to invoke these four operations on the current process.

```
#process dump
#process describeObj
#process gc
#process gcAll
```

3.7. Looking at the Current Stack Frame

You can look at the current stack frame from Low Telescript. The general procedure for referring to the current stack is to use the character tilde (**~**). Instead of giving an expression that evaluates to some object, white space, and an attribute or operation name, use a tilde. Don't put white space between the tilde and the name of the operation or attribute.

Suppose you are having trouble debugging an **initialize** operation. You'd like to dump out a here-I-am message and then a dump of the stack in your code. You can escape into Low Telescript and call the **dump** operation on the stack.

```
"Here I am".dump();
<< ~dump >>;
```

The stack frame is a member of class Stack. You can do other things, such as determine the number of elements on the stack and then dump the stack only if the number is unexpected.

```
if << ~length >> != 5 {
     "### THE STACK DOES NOT HAVE 5 ELEMENTS ###".dump();
     << ~dump >>;
};
```

If you are using the front-panel debugger's **`dEval`** command, you may want to probe at elements on the stack. Suppose the stack contains the following, listed from top to bottom:

- a string
- a list of 3 integers
- a permit

You don't want to consume anything from the stack because you want to continue execution when you're done. You can use the **`get`** operation to refer to a value on the stack. To dump the **`canGo`** attribute of the permit mentioned above, you could say:

```
3 ~get canGo dump
```

The permit is the third item from the top, so the element to get is 3. The Low Telescript above asks for the attribute **`canGo`**, then consumes the value of **`canGo`** by calling the operation **`dump`**.

There is a handy shorthand for getting the top element of the stack. Instead of writing:

```
1 ~get
```

you can write:

```
ref
```

There is one other thing you are likely to want to do. Suppose you want to dump the fourth letter of the string at the top of the stack. You could write:

```
4 2 ~get get dump
```

The operation **`get`** on a String requires an integer argument. You've got to supply the integer value (in this case, 4) before you give the reference to the string. The string was at the top of the stack. But because you just put the integer 4 on the stack, it becomes the second argument. To get the string, you need to ask for the second item the on stack.

The Stack operation **`swap`** exchanges the top two items on the stack. This means you can use **`ref`** to get a reference to the string, then push the argument to String's **`get`** operation. Use the **`swap`** operation to bring the string to the top of the stack, then call **`get`**.

```
ref 4 ~swap get dump
```

3.8. Doing Complex Things

If you want to do something more complex in Low Telescript than the previous sections taught you to do, write a little Telescript program that does what you want as free-standing Telescript. Compile your file then run it through ***tsprocs*** to obtain a ***.tsp*** file. A ***.tsp*** file is text. If you read the file, you will find the Low Telescript you want between the two boilerplate lines.

```
Telescript 1.0 {
     /* the Low Telescript you want will appear here */
}
```

The Telescript linker ***htl*** gets badly confused if your Low Telescript code contains unbalanced **[** and **]** characters. If you generate Low Telescript using tsprocs and plan to insert it into a Telescript file as a Low Telescript escape, check to make sure that **[** and **]** characters are balanced in number. If you really want to be safe, do the following substitutions:

- Substitute **#mark** for **[**
- Substitute **new** for **]**

The character **[** is a shorthand for the special marker used to indicate where the arguments begin for **initialize** and operations taking a variable number of arguments. The character **]** is a shorthand asking for an instance of a class to be created.

4. Writing a Secure Application

Telescript is a relatively safe language with regard to programming hazards. Everything is object-oriented and structured, with runtime type checking and exception handling. However, Telescript provides not only a multi-tasking, but a *multi-user* environment. As a consequence, many of the security issues that arise in multi-user operating systems can also be present in a Telescript region.

In many ways, if the engine place is considered the "operating system," the engine itself is actually the kernel, and the scripts in the engine place the "trusted process" components that perform, for example, login authentication. Allowing an unknown agent carte blanche to enter a region could be as risky as allowing a guest login to import and run any program it wants.

Telescript provides a number of process safety features for dealing with such problems. The underlying principle is that Telescript uses object encapsulation not just as a programming safety feature, but also as a protection mechanism. The engine enforces isolation of private from public (hence sharable) features. Objects can use this to implement their own security policies, for example, access controls on features.

The basic approach to process security is defensive. Telescript security mechanisms accommodate various levels of paranoia when interacting with potentially untrusted code. These can range, for instance, from simple access control checks all the way to 'xenophobic' restrictions on what code can and cannot execute.

This chapter illustrates some defensive programming techniques using language mechanisms introduced in previous chapters. It is by no means extensive in this regard. In this chapter, we tend to think paranoid.

4.1. Basic Access Controls

Operations that need to can implement identity-based access controls that base authorization decisions on the caller's identity. Telescript supplies the caller's identity in global variables. Typically, checks can be made on the basis of the class and/or authority, of

- the current process (that owns the thread of control)
- the current sponsor (which is not necessarily the current process)
- the owner of the client (which may not necessarily be owned by the current process or sponsor)

In general, sensitive operations are written as private, and public operations used as "wrappers" that do access checks. Identity-based access controls, for example, could use a list of authorized authorities and classes, as:

```
MyAccessViolation: class(Exception) = ();
MySafeClass: class(Object) = (
    public
       mySafeOp:op ( theArg: Object )  = {
              theSponsor:OctetString = sponsor.name.authority;
              theClass:Class = client.class;
              if       ((friends.examine(theSponsor) == nil) ||
                       (okClasses.examine(theClass) == nil) ) {
                       throw MyAccessViolation() ;
              };
              myRealOp ( theArg ); // could check the arg, too.
    };
    private
       friends: List[OctetString,Equal] = ... ;
       okClasses: List[Class,Equal] = ... ;
       myRealOp:op (argument: Object) = { ... };
);
```

An alternative technique could be to associate one or more "rights identifiers" with each authority or class, instead of simply putting them in a list. The access control check in each operation could then test for specific rights, and access control mixin classes would contain functionality that maintains and does lookups on the appropriate data structures.

Note that, in many cases, the sponsor and the process are the same. This leads us to why in our example only the sponsor is checked and not both the sponsor and process. In the case of a sponsored operation, the sponsor is actually the process that owns the responder, and it’s the sponsor’s permit that controls what can be done in this operation, so it’s important to check who the sponsor is. Some access policies might also care about what process the sponsored operation is ultimately acting on behalf of.

Remember that Telescript provides these same global variables to the object whose operations your object calls. Thus, those objects can get unprotected references to your object and to your process. Furthermore, the call will execute on your thread of control, even if it’s sponsored and you are not charged for it, so that your process is the process. Even if you don’t necessarily expect your public features to be called back in this way, if you really don’t want it to happen you should make the appropriate access control checks.

4.2. Restrict and Temporary Permits

Access controls are written from the perspective of the called operation. The caller, however, could be executing unknown and hence potentially untrusted code (for example, with a Trojan horse in it) under its authority and permit. Even innocent looking attributes can result in code being executed in an object's getter or setter.

The untrusted code, once it gains control as you, could attempt to masquerade under your authority to call other objects, perhaps owned by other processes such as the place or agents with which it has met. In a truly paranoid situation, one could design the application interface to make sure that certain operations are sponsored, passing all arguments and results by copy or protected reference. Fortunately, such extreme measures are usually unnecessary.

Another thing untrusted code could do is mount denial of service attacks against you. For example, it could attempt to exhaust your permit, or throw some exception (of course, you catch all exceptions, so this approach won't get it very far). Permits are useful to limit this kind of damage.

Here's an example:

```
paranoid := Permit();      // create a new Permit instance

paranoid.canCharge = false; // No capabilities
paranoid.canCreate = false; // Can't make agents
paranoid.canDeny = false;
paranoid.canGo = false;    // For good measure.
paranoid.canGrant = false;
paranoid.canRestart = false;
paranoid.canSend = false;

paranoid.age = *.age + 2;  // give it two seconds
paranoid.extent = *.size + 1000;
paranoid.charges = *.charges + 1000;

try {
     restrict paranoid {           // restrict statement
        yourObject.yourSuspectCall(); // the suspect call
     };
catch failed: PermitViolated {
    ...
     };
catch ...   // other catch clauses
```

If the call fails in a restrict statement because of a permit violation, the engine raises Permit Violated rather then the fatal Permit Exhausted. Restrict might also be used in the case when you wanted a warning that your permit was about to expire.

Note again that if the called operation was sponsored, the temporary permit would not apply. You might not be able to stop the operation from dispatching an agent carrying off some sensitive information about you. However, the agent would not be charged to you.

4.3. Entering and Local Permits

Before an agent can enter a place, the engine gives that place the opportunity to deny occupancy, by invoking its `entering` operation. The engine supplies the effective "requested" permit and identity of the agent, including its Telename and class, but not a direct reference to that agent. Note that `entering` does execute on a new thread of control as the place.

The effective requested permit is the intersection of the agent's native and regional permits, along with any permit that might have been requested in the Ticket when the agent did the go. The place gets this sanitized value, which, if it makes no changes, will become the effective local permit for that agent while it occupies that place. The place can limit the local permit, being careful in the process not to kill the agent.

Here is an example:

```
MyParanoidPlace : class ( Place ) = (
    public
       ...                      // initialize, live, etc.
    entering: sponsored op( who: protected Telename;
              theClass: protected ClassName!;
              permit: unprotected Permit;
                     // intersection of native, regional, requested
              ticket: protected Ticket | Nil) Object
    {
       *.myRestrictEntry(who.copy(), theClass); // local access check
              // throws OccupancyDenied for us
              // we'll let this agent in, but with restrictions...
    try {   // could fail in subtle ways
       permit.canCreate = false;  // can't make new agents in here.
       permit.canSend = false;    // we will let it go out eventually
       permit.canRestart = false;
       permit.canCharge = false;  // nada...
       permit.canDeny = false;
       permit.canGrant = false;
       permit.age = 120;          // live free for two minutes or die!
       if ((permit.priority == nil)||(permit.priority > *.priority)) {
              permit.priority = *.priority - 5 ;
                            // make sure lower priority
       };
       permit.authenticity = *.myAuthPolicy(who, theClass, permit);
                     // set how much we trust this guy
                     // don't care about limiting size or charges
       return(nil);
    }
    catch Exception {
```

```
        throw OccupancyDenied;  // that's it, no fooling around.
    };
};
private
    myRestrictEntry: op(name: Telename; theClass: ClassName)
        throws OccupancyDenied = { ... };
        ...            // other stuff here...
);
```

The meeting protocol also gives meeting agents and the meeting place a chance to authorize correspondents.

Unfortunately, there's not much the agent can do at present to prevent being killed by the place.

4.4. Engine Places, Regional Permits, and Purgatory

The engine place runs as superuser, and is a travel place. It gets first chance to reject an incoming agent, and it gets the last look at that agent on the way out.

The engine place gets an indication of how well the agent was authenticated in the arguments to `transferredIn`. Since it is the region, it can use the `poke` operation to set the agent's regional permit. The regional permit is like the other permits, except that the agent can only read it, and only a process of the region's authority can change it.

To use regional permits in this way, a region needs some policy (analogous to a password file) that assigns particular rights to incoming agents according to their class and authority, authentication level, and any other criteria that might be relevant. Telescript provides the mechanisms to implement such a policy, including control over where an incoming (or outgoing) agent can go. The travel place can change the agent's destination to re-route it to whatever place it wants.

For example, it might send the agent to *Purgatory*. Purgatory is a "safe" place hosted by an engine as a courtesy to agents whose trips fail. Typically, an agent gets a limited permit in Purgatory, enough for it to catch its trip exception and do something about it. While in Purgatory, an agent is totally unaware of any other occupants, and the place has no features for the agent to call. All it can do is leave or die. Agents leaving Purgatory can be constrained to leave the region, by checking in `transferOut` that the ticket supplies an appropriate way and means.

Another aspect of the regional policy is xenophobia, that is, whether or not the agent is permitted to bring in its own class implementations, or whether that agent is constrained to use classes that the engine already has. This capability to restrict "foreign" procedures is implementation-specific to the current engine and not part of the language, hence might not be in future engine releases.

It works like this. As the engine decodes the agent, it verifies that any class that the agent brings with it having the same name as a class already in the engine is the same class. Classes that don't already exist in the engine get their imported procedures flagged foreign. Only a process that has the "canExecuteForeign" attribute true can execute one of these foreign procedures. And only a region process can access the canExecuteForeign attribute at all; a process that does not have the region's authority gets a FeatureUnavailable exception if it tries. The engine throws a PermitViolated exception if a process that does not have canExecuteForeign attempts to execute a foreign procedure. This is only fatal if not caught.

4.5. System Safety and Security

The classes provided by the engine for accessing local system resources should be appropriately access controlled in the engine place. These classes, among which are File, External Handle, and Control Manager, should actually only be available to the region, and accessed through sponsored operations of "wrapper" classes that enforce finer grained access controls. Since these classes are all copyrighted, access policies for them can be defined in each's copyright enforcer.

The basic engine place provides a default copyright enforcer really doesn't do much of anything. By default, any process can instantiate objects of these classes. This is obviously unsafe, so in a multi-user region, the appropriate copyright enforcers and wrapper classes need to be defined.

Index